UNIVERSITY OF COPENHAGEN

This thesis has been submitted to the PhD School of The Faculty of Science,

University of Copenhagen

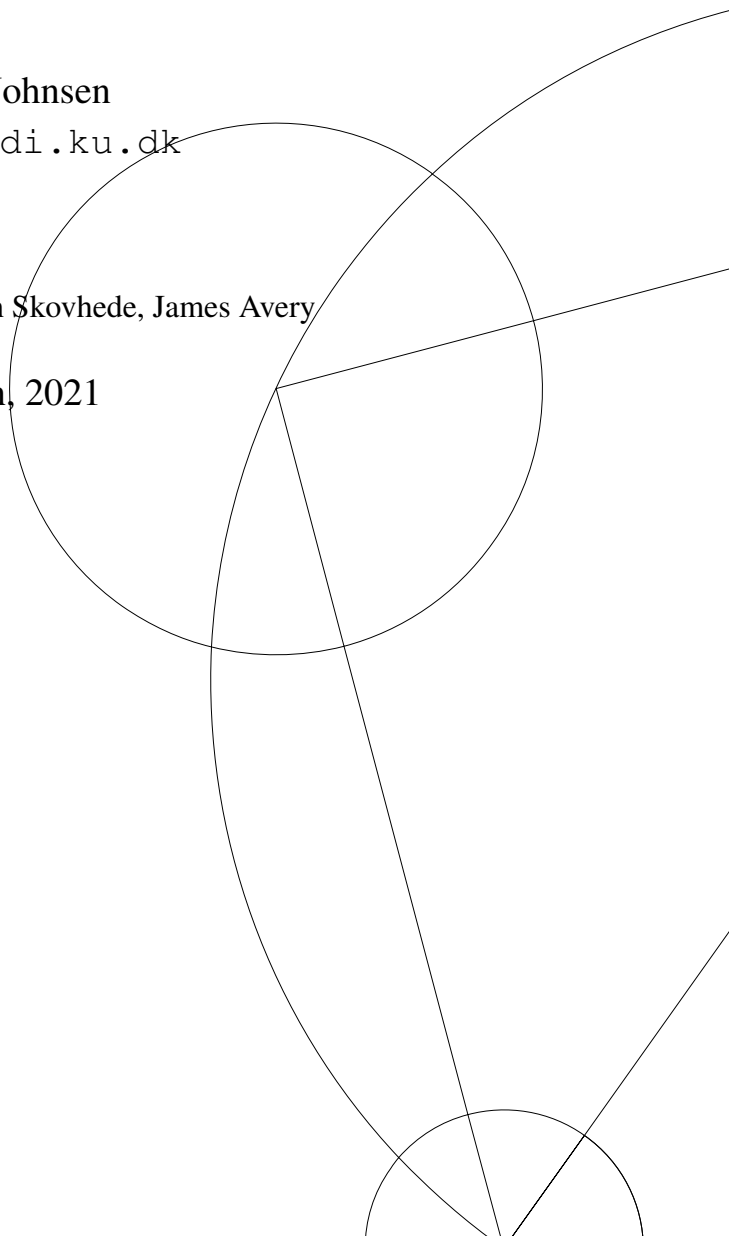
Automated AI Learning for X-Ray Based Classification

Carl-Johannes Johnsen

carl-johannes@di.ku.dk

Supervisors: Brian Vinter, Kenneth Skovhede, James Avery

December 13th, 2021



ACKNOWLEDGEMENTS

First and foremost, I want to thank my three supervisors Brian Vinter, Kenneth Skovhede, and James Avery, for your supervision while having thousands of tasks concurrent to my thesis and for your continued support even from new job positions where you were no longer forced to supervise.

I want to thank the eScience group for your open and helpful nature, always being willing to help and discuss any topic, and providing insightful viewpoints regardless of whether the subject was within your field of research. Even in a semi-disbanded state, you remain helpful.

I want to thank the AXIS partners for always being fast and helpful in providing information, data, and help for both me and the student projects whenever needed.

I want to thank the SPCL group at ETH Zurich for taking me in to be part of your group for such a long time. The quality of your work is impressive, and you have pushed me towards raising my own bar.

I want to thank my army of students for being patient with having me, a junior researcher, as your supervisor. Together we have completed some cool projects that I hope you yourself can be satisfied with - I am at least.

I want to thank my girlfriend, Louise Vedel Jensen, for putting up with my almost constant state of distraction and late-night working, in turn ruining your sleep pattern. Without you, I would forget to eat and sleep, both of which are quite essential.

Finally, I want to thank my friends and family, who have helped me keep my sanity throughout the pandemic and this thesis, always being up for grabbing beers and being patient throughout extended periods of ”radio silence” at some points.

ABSTRACT

Food inspection is a widespread measure for ensuring food quality, contents, and safety. The food inspection industry has deployed cameras and computer vision for a long time since they allow for automatic inspection and evaluation of food quality at a large scale. However, they fail to capture internal defects, which can arise due to natural and unnatural effects. X-ray imaging allows us to inspect the internals of a subject, thus countering the problem. The Adaptive X-ray InSpection (AXIS) project collaborates with multiple partners targeting utilizing X-ray imaging for food inspection. This thesis presents the University of Copenhagen's part of the AXIS project: software for automated AI learning for X-ray-based classification. The main research areas are computer vision, machine learning, and reconfigurable hardware. Computer vision can counter the side-effects of X-ray imaging, resulting in clean images ready for machine learning. In recent years, machine learning has seen a rise in image processing due to its strong predictive power in recognizing patterns in seemingly complex imagery. To keep up with the high-throughput demands of food inspection, we employ reconfigurable hardware, or Field-Programmable Gate Arrays (FPGAs), as these favor pipelines are great at online processing and have been shown to be suitable for machine learning inference. This thesis presents an FPGA-based solution for the preprocessing steps of the X-ray images, a CPU/GPU-based implementation for the machine learning model, along with an outline for an FPGA-based machine learning inference solution. The implementations have had automation in mind, minimizing the required amount of user interaction. This thesis has spawned many student projects that have been supervised throughout the thesis while contributing to the field of reconfigurable hardware through open-source projects seeking to lift the level of abstraction in hardware programming.

RESUMÉ

Fødevare inspektion er en udbredt foranstaltning til at forsikre fødevare kvalitet, indhold, og sikkerhed. Fødevare inspektions industrien har i lang tid indsat kameraer og datamatsyn, da disse tillader automatisk inspektion og evaluering af fødevare kvalitet i stor skala. Dog kan de ikke fange interne defekter, som kan forekomme på grund af naturlige og unaturlige effekter. Røntgenbilleder tillader os at inspicere et objekts indre og dermed modarbejde disse problemer. Adaptive X-ray InSpection (AXIS) projektet samarbejder med flere partnere om at udnytte røntgenbilleder i fødevare inspektion. Denne afhandling præsenterer Københavns Universitets del af AXIS-projektet: et program til automatisk kunstig intelligens læring til røntgen baseret klassificering. Hovedforskningsområderne har været datamatsyn, maskinlæring, og rekonfigurerbar hardware. Datamatsyn kan modarbejde røntgen sideeffekterne i røntgenbilleder, hvilket resulterer i rene billeder der er klar til maskinlæring. Maskinlæring har set en fremdrift i billede behandling i de senere år, på grund af deres stærke forudsigelses kræft i at genkende mønstre i umiddelbart komplekse billeder. For at kunne følge med de høje datagennemstrømningskrav i fødevare inspektion har vi anvendt rekonfigurerbar hardware, eller Felt-Programmerbar Port-Tabel (FPPT), da disse foretrækker dybe rørledninger (pipelines), er gode til online processering, og har vist sig at være passende til maskinlærings inferens. Denne afhandling præsenterer en FPPT-baseret løsning til præprocessering af røntgenbilleder, en processor/grafikkort-baseret implementering af maskinlærings modellen, samt en opridsning af en FPPT-baseret maskinlærings inferens løsning. Implantationen har haft automatisering i baghovedet for at minimere mængden af bruger interaktion. Denne afhandling har udsprunget mange studenter projekter som er blevet vejledt igennem den her afhandling, samtidig med at den har bidraget til feltet rekonfigurerbar hardware igennem open-source projekter der bestræber sig på at løfte abstraktionsniveauerne i hardware programmering.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ALU Arithmetic Logic Unit. 52, 56
AMBA Advanced Microcontroller Bus Architecture. 64, 94
AMF Adaptive Median Filter. 19, 94, 95
ANN Artificial Neural Network. 3, 26, 32–34, 37, 40, 46, 49, 100, 105
API Application Programming Interface. 62
ASIC Application-Specific Integrated Circuit. 4, 53–55, 84
AST Abstract Syntax Tree. 59–62
AUC Area Under Curve. x, 38, 40, 42, 92, 93, 102
AXI Advanced eXtensible Interface. x, 64, 65, 67, 70, 73, 94, 95, 97, 98
AXIS Adaptive Xray InSpection. 4–6, 24, 26, 49, 86, 87, 102

BLAS Basic Linear Algebra Subprograms. 63, 67
BRAM Block Random-Access Memory (RAM). 56, 62, 70, 79, 94, 96, 99

CAE Convolutional Auto Encoder. ix, 37
CCD Charge Coupled Device. 12
CGRA Coarse-Grained Reconfigurable Arrays. 76
CLAHE Contrast Limited Adaptive Histogram Equalization. 29, 30
CMOS Complementary Metal Oxide Semiconductor. 12
CNN Convolutional Neural Network. ix, xi, 3, 33–36, 49, 90, 105
CPU Central Processing Unit. ix, x, 3, 4, 51–56, 62, 64, 67, 81, 83, 84, 87, 100
CSP Communicating Sequential Processes. 58, 59, 76, 77
CSV Comma-Separated Values. 59
CVAE Convolutional Variational Auto Encoder. ix–xi, 37–39, 49, 91–93, 97, 100, 102, 104, 105

DaCe Data-Centric parallel programming. 5, 63, 64, 66, 67, 71, 73, 76, 84, 104, 105
DDoS Distributed Denial of Service. 80
DDR Double Data Rate. 54, 96
DSL Domain Specific Language. 74, 76
DSP Digital Signal Processor. 56, 70, 74
DT Decision Tree. 40

ETH Eidgenössische Technische Hochschule. 63

FDR Failures-Divergences Refinement. 59
FF Flip-Flop. 56, 70
FFC Flat-Field Correction. x, 19, 89, 94, 95, 102
FFT Fast Fourier Transform. 100
FIFO First In First Out. 79
FPGA Field-Programmable Gate Array. x, 4, 6, 51, 54–56, 58, 60, 62–66, 69–71, 74, 76, 79–81, 84, 94, 95, 97, 100, 102, 104, 105
FPN Fixed-Pattern Noise. 19
FPR False Positive Rate. 42
FSIM Feature-based Similarity Index Measure. x, 92–94, 100–102

GBDT Gradient Boosted Decision Tree. 40
GPU Graphics Processing Unit. x, 3, 4, 51–55, 84, 100

HBM High Bandwidth Memory. 70
HDL Hardware Description Language. xiii, 56
HE Histogram Equalization. 29, 30
HLS High-Level Synthesis. 55, 56, 58, 63–66, 69–71, 73, 75, 76, 84
HPC High-Performance Computing. 84

IL Intermediate Language. 61, 62
IP Intellectual Property. 54, 55, 62, 63, 69, 72, 73
IR Intermediate Representation. 55, 63, 71
IRWSR Iterative Re-Weighted Super-Resolution. 13, 14

LFSR Linear-Feedback Shift Register. x, 100
LUT LookUp Table. 56, 70, 74

MAE Mean Absolute Error. 46, 48
MFSR Multi-Frame Super-Resolution. 13
ML Machine Learning. x, 3–6, 20, 26–29, 32, 35, 40, 45, 46, 48, 49, 86, 89, 96, 97, 102, 104, 105, 169
MSE Mean Squared Error. 37, 104

ONNX Open Neural Network eXchange. 63, 104
OpenCL Open Computing Language. 64

PCA Principle Component Analysis. 37
PCIe Peripheral Component Interconnect Express. 54
PE Processing Element. 75

RAM Random-Access Memory. xii, xiii, 54, 56, 94, 96
RMSE Root Mean Square Error. 46
ROC Receiver Operator Characteristics. ix, x, 42, 92, 93
ROI Region Of Interest. 19, 20, 28, 46, 89, 95
RTL Register-Transfer Level. x, 55, 56, 58, 63–67, 69, 71, 73, 75, 76, 84

SDFG Stateful DataFlow multiGraph. x, 63, 66, 67
SDRAM Synchronous Dynamic RAM. 54
SIMD Single Instruction, Multiple Data. 4, 73
SME Synchronous Message Exchange. x, xi, 5, 58–62, 75–84, 105
SPCL Scalable Parallel Computing Laboratory. 63, 66
SSIM Structural Similarity Index Measure. x, 38, 91–93, 102

TCL Tool Command Language. 63
TCP/IP Transmission Control Protocol / Internet Protocol. x, 79, 80
TPR True Positive Rate. 42
TPU Tensor Processing Unit. 4, 180

VHDL Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language (HDL). 56, 59, 62, 63, 75
VHSIC Very High-Speed Integrated Circuit. xiii, 56

XML eXtensible Markup Language. 64
XOR eXclusive OR. 100

Part I

SETTING THE STAGE

1

INTRODUCTION

The food industry produces vast amounts of food every day, which goes through rigorous quality control. This quality control is primarily to ensure that the product is safe for human consumption and meets the high standards imposed by the average consumer. For example, everyone has had the experience of discarding certain vegetables due to them being misshapen or discolored. We define defects such as these as being mild defects. A product with a mild defect might not render the entire product unusable but rather render it unattractive. On the other end of the scale, we have hazardous defects, which constitute unsafe defects.

An example of this could be pieces of metal, which come from a broken blade slicing meat into cold cuts and could be fatal if consumed. While some cases occur from natural events, they can also occur deliberately, such as the needles in strawberries case [1]. This case was sabotage, where a person targeted a strawberry farmer by putting needles in strawberries. As a result, the farmer had to recall a whole season's harvest, thus losing income.

While quality control through manual inspection is trivial, it quickly becomes infeasible once food production is scaled up. E.g., a potato farmer can produce several tons of potatoes an hour during harvesting. Therefore, it is crucial that inspection happens as fast as possible, minimizing the time spent, to not waste resources or potential income. These defects can occur at every step of the production pipeline: growing, harvesting, packaging, and shipping. As such, inspection should not be limited to one step of the pipeline and could be done at multiple stages until right before selling the product. The defects in food occur both externally and internally. External defects can be detected using visual camera inspection, which the industry currently employs. However, this does not capture internal defects, such as the metal needles inside the subjects. We can "look inside" the scanned subject by using X-ray imaging. For example, it is a widespread technique used for internal inspection without "opening" the subject as long as the defects consist of different density materials compared to the scanned subject. A density difference will cast prominent shadows on the resulting image, allowing a trained expert to diagnose potential causes and treatments. While X-ray is harmful to living tissue through prolonged exposure, it does not affect living tissue for short bursts of exposure. Furthermore, foods have shown no signs of degradation or carrying any hazardous amounts of radiation following being scanned by X-ray [2].

This thesis investigates the steps required for automated food inspection using X-ray imaging. Once we establish a baseline for achieving this functionally, we will focus on optimizing the baseline to keep up with the high throughput requirements found in the industry.

1.1 AUTOMATED INSPECTION

A popular way to automate visual inspection is through cameras and computer vision. A camera captures a photo by exposing its photocells to light that reflects off a surface. The number of photons hitting the photocell determines the intensity of a pixel. This intensity is then translated into a digital signal, representing the number of photons that hit the photocell during the exposure period. Multiple photocells are employed to produce color images, where each photocell captures photons within a particular wavelength. X-ray images are captured much like regular images, but instead of capturing light reflected off a surface, X-ray photons penetrate matter and hit a substrate called a scintillator. When hit by photons in the X-ray spectrum, this substrate gets excited, emitting photons in the visible spectrum. This emitted light is what is the camera captures during X-ray imaging. Generally, X-ray imaging uses two different types of camera: an area scanner or a line scanner. An area scanner is a 2-dimensional grid of photocells, which make up an image. A line scanner is a single row of photocells, which only captures a single image row. Line scanners are more prevalent in X-ray imaging than area scanners because they are cheaper and often have a higher capture rate. This higher capture rate is a side effect of having fewer photocells to drain during capture than an area scanner.

The field of computer vision covers the act of programmatically having the computer understand the context in imagery or video. In recent times, Machine Learning (ML) has had a rise in popularity for image recognition problems due to the ease of expressing the problem, large amounts of data available, the exponential rise in computational resources through time, and finally, the predictive power of ML. In ML, one ”trains” a *machine* model to recognize a pattern through *learning*, achieved by providing a large dataset that contains the pattern. Roughly speaking, ML incorporates two main approaches: supervised and unsupervised learning. In unsupervised learning, the data ”speaks for itself,” and the model attempts to derive a pattern purely from the data. In supervised learning, the data features corresponding labels, and the model attempts to match this labeling during training by predicting the label based on the data. In both cases, an oversimplification is that the model is a many-dimensional function with many tunable parameters that are tuned during training to provide the best *fit* to the data.

The image recognition ML models have become more resilient at detecting patterns at the same level as humans, at least for specific problems, such as the classic example with handwritten digits [3]. The winning strategy for this example has been the Convolutional Neural Network (CNN); an extended version of the Artificial Neural Network (ANN) that has preceding convolutional layers for extracting the features of an image. The model’s complexity defines a model’s expressive power, enabled by the computing resources available in modern machines. However, the demand keeps rising for even more complex models, introducing higher demands for more computing resources.

1.2 THE STATE OF MODERN COMPUTING RESOURCES

Central Processing Units (CPUs) are excellent due to their versatility and focus on generality. CPUs have become faster and more numerous, with eight processing cores commonly available in a chip. However, most chip area constitutes speculative execution and control flow rather than computational power. While this is great for most tasks handled by a computer, it is not so great for machines that continuously only handle one particular problem, such as ML inference in an inspection pipeline.

Graphics Processing Units (GPUs) takes a step towards trading generality for computational power. GPUs consist of thousands of cores set up in a grid-like fashion. These chips favor parallel problems

as each core is weak compared to a standard CPU core. This weakness comes to light in problems that feature lots of branching and dynamic executions. However, the GPUs are overall computationally stronger due to the sheer amount of execution cores and the focus on memory access patterns. These cores specialize in vector processing: doing the same operation on multiple data elements. This type of execution is known as Single Instruction, Multiple Data (SIMD). By expressing matrix computations as a series of SIMD operations, GPUs become very efficient at doing matrix computations.

The final step in the generality trade-off ladder is custom circuitry where the user is in complete control – for better or for worse. Nevertheless, the potential gains here are significant since only the parts that the user deems necessary remain. These are known as Application-Specific Integrated Circuits (ASICs) and are the superset of all integrated circuits. However, in the context of this thesis, we will limit the term ASIC to integrated circuits, which focus on solving single tasks. Examples of ASICs focusing on ML are Tensor cores [4] found in newer NVIDIA GPUs, Tensor Processing Unit (TPU) [5] developed by Google, and Neural Engines found in the new Apple M1 chips [6].

The middle ground between generality and specificity is the Field-Programmable Gate Array (FPGA), also known as reconfigurable hardware. Where the ASIC is unmodifiable after production, an FPGA focuses on reconfigurability. Conceptually, an FPGA consist of a grid of logic gates connected through a user-defined interconnect. Thus, the user defines their hardware model by specifying how these gates are connected, resulting in a semantically equivalent circuit compared to fully purposed hardware.

However, reconfigurability comes at a price because the connections within the FPGA are not as fast as purpose-built hardware connections. As such, FPGAs cannot reach such aggressive clock rates as the ASICs, giving them orders of magnitude worse performance than ASICs. Compared to CPUs and GPUs, FPGAs excel in either deeply pipelined or very custom problems. For example, GPUs are purpose-built for vector operations, making this problem hard to beat with FPGAs. However, FPGAs are best if we compare them using more exotic problems, such as low precision math or control-oriented problems. Furthermore, they are generally more stable than CPUs and GPUs, as the reduced complexity also reduces the probability of an error occurring. Finally, cutting away generality also reduces power consumption, as the power flows through a reduced number of transistors and connections.

1.3 ADAPTIVE X-RAY INSPECTION

The Adaptive Xray InSpection (AXIS) project aims to solve automated food inspection using X-ray imaging. It is a collaborative project between four partners; the University of Copenhagen, Newtec, Qtechnology, and Magnatek.

NEWTEC has a long history of building industrial machines for quality assurance, weighing, packaging, and sorting food products. Their primary technique is visual inspection through cameras to look for defects for quality assurance. They will be building the machine, which will assemble the three other deliverables into a single product. They are also looking into finding the right scintillator, which will have suitable properties for food inspection.

QTECHNOLOGY is a daughter company of Newtec, which specializes in camera technology. Their main product is a camera and computer consisting of a CPU, GPU, and FPGA, all built into a single package. It is the driving force of Newtec's visual inspection part of their machines. They will be delivering the camera for the AXIS project.

MAGNATEK is a company that produces high-frequency X-ray sources, which a wide array of machines employ, including food inspection and sorting. They will be delivering the X-ray source for the AXIS machine.

The final piece of the AXIS puzzle is the software part delivered by the University of Copenhagen, an automated solution for building ML models for food inspection, which live up to the industry's high throughput requirements. This thesis investigates the different ways of achieving this, both functionality- and performance-wise. The final implementation is based on the findings from supervising multiple student projects that have sprung as an effect of this thesis and the research done in the various fields.

1.3.1 *Contribution*

This thesis has contributed to open-source projects in the field of programming models for reconfigurable hardware: Synchronous Message Exchange (SME) and Data-Centric parallel programming (DaCe). SME has been improved quality-wise while seeing new abstractions enabling software programmers to delve into the world of hardware development by leveraging modern development tools and language features through compositional process isolation abstractions. DaCe allows the developer to target multiple platforms from a high level of abstraction. This thesis has improved the expressibility of DaCe by providing tighter integration of low-level implementations in this higher-level representation, allowing the hardware-specific optimizations, such as our automatically applied multi-pumping optimization.

A big problem in supervised ML is the need for a complete representation of the sample space. This thesis proposes a semi-supervised approach to the classification problem, where we train a model purely on one class of samples, gaining a model that, without ever seeing samples from other classes, can distinguish between ”good” and everything else. This proposed model can further be implemented as a pipelined streaming application, keeping up with the high-throughput requirements of food inspection.

The open-source contributions have been made on Github, by the account `carljohnsen` [7].

2

OUTLINE

This thesis focuses on the University of Copenhagen's deliverable to the AXIS project, a program for automated food inspection using X-ray imaging. From a top-level view, the thesis consists of five parts, with Part i being this introduction.

One of the critical features of the AXIS project is the use of X-ray imaging. Part ii will describe the principles behind X-ray and imaging in order to describe how to capture X-ray images. These concepts should help the reader build intuition regarding the challenges of X-ray imaging. The research contribution of this part is in general image processing, explicitly targeting the challenges of X-ray imaging. This thesis is a computer science thesis, not a physics thesis, so the physics described will be conceptual.

Part iii presents some core ML concepts, such as how we build and evaluate a model. We will cover the research we have investigated during this thesis laying the foundation for our later choice of model. The research focus has been on the application of the ML field, especially preprocessing, different model architectures, and model evaluation techniques.

Part iv starts by describing general machine architecture to lay a conceptual foundation for the reader and to motivate the use of FPGAs. As high-throughput is a requirement for the final product, most of the work put into this thesis has been in accelerators, specifically reconfigurable computing. The research has focused on programming models for FPGAs, seeking to lift the abstraction of hardware design into the field of software development to allow for a broader population of hardware developers.

Finally, Part v will be combining the knowledge from the previous parts to outline the final product; a low-power, high throughput, and stable classifier.

This thesis has spawned many sub-projects whose relevance and results we cover throughout the thesis whenever relevant, along with Appendix B, covering them in full.

Part II

X-RAY AND IMAGING

3

X-RAY

This chapter will cover the physics of X-rays to build a foundational understanding of how we capture the images. We can counter some of these effects, but some are just a natural outcome of physics. As this Ph.D. thesis is in computer science, not physics, the theory will be on a conceptual level.

Light consists of photons oscillating at some wavelength, and they belong to a specific class of light depending on this wavelength. X-ray is one of these classes of light, which reside within a particular wavelength. Specifically, X-ray lies within 0.01 to 10 nanometers (nm), whereas the more well-known visible light is within 380 to 700 nm. As such, the X-ray has a much shorter wavelength compared to visible light. Given the constant speed of light, c, then the wavelength, λ, is directly tied to the frequency, f, of the photons:

$$\lambda = \frac{c}{f} \tag{1}$$

This relation translates into: the shorter the wavelength, the higher the frequency. Furthermore, higher frequencies translate into higher energies. High energy is what gives the photons their penetrative properties. Figure 1 shows the different light classes, wavelengths, and corresponding frequencies.

When X-ray photons interact with matter, the photons will be absorbed, slowed, or diverted, depending on the photons' energy and the matter's density. Attenuation curves describe how a particular material will absorb a beam of photons at a particular energy. This curve shows the probability of an interaction between the beam and material. Thus, the larger the attenuation coefficient, the more likely the material will absorb the beam.

One possible application for attenuation curves is choosing the energy that maximizes contrast on the final image. For example, imagine having to locate a needle in a haystack. We consider the chemical composition of a needle as specified in Table 2 and of hay as specified in Table 3. From the

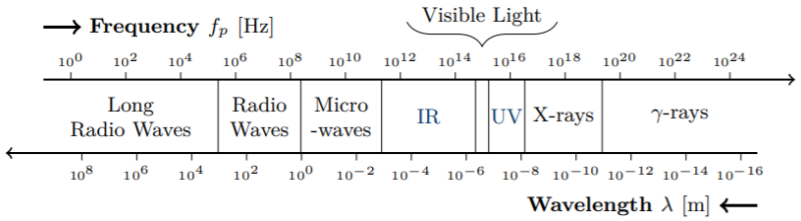


Figure 1: The electromagnetic spectrum given different frequencies and wavelengths that are related by Equation (1). Image from [8].

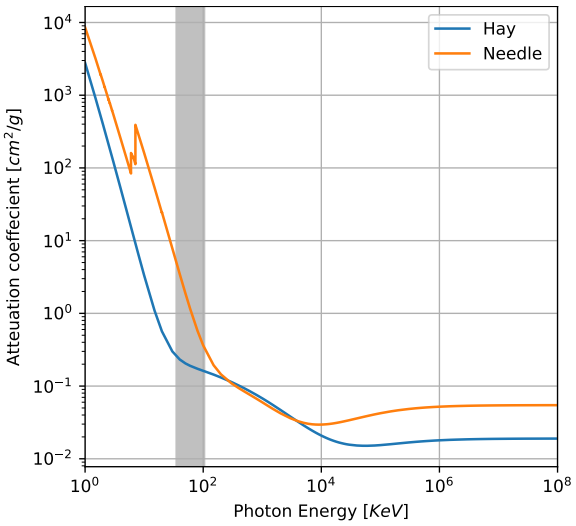


Figure 2: Log-log plot of the linear attenuation coefficient between hay and a needle. The gray area shows the spectrum achievable by the X-ray source described in Table 1.

Table 1: X-ray generator, numbers taken from the datasheet [9]. Note that the source is rated to 500W and 100kV, but during its warm up phase, it was possible to push it a bit outside its recommended working limits.

Name	**Value**
Manufacturer	Spellman
Model	XRBHR100 monoblock
Voltage range	35-105kV
Current range	0.35-7.5 mA (35-70 kV)
	0.35-5.0 mA (71-105 kV)
Max power	525 W

Table 2: Chemical composition of a needle made of hardened carbon steel. Values from [10].

Material	Amount		
Carbon (C)	1	-	1.200 %
Chromium (Cr)	16	-	18.000 %
Iron (Fe)	78	-	83.100 %
Manganese (Mn)	0	-	1.000 %
Molybdenum (Mo)	0	-	0.800 %
Phosphorus (P)	0	-	0.040 %
Silicon (Si)	0	-	1.000 %
Sulfur (S)	0	-	0.015 %

Table 3: Chemical composition of hay, assuming it has the same composition as grass [11], but with less water [12].

Material	Amount
Water (H2O)	14.000 %
Lignin (C9H10O2)	28.667 %
Lignin (C10H12O3)	28.667 %
Lignin (C11H14O4)	28.667 %

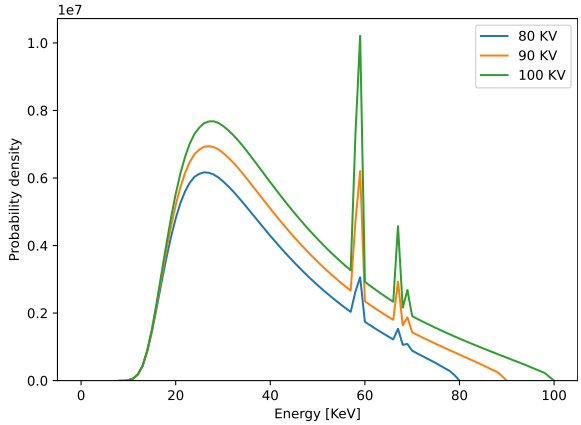


Figure 3: X-ray filtered bremsstrahlung spectrum simulation using Tungsten anode. These are the actual values produced by an X-ray source at different voltages. Similarly, increasing the ampere would increase the curves' magnitude but retain the same relative distribution. Plot is from [14].

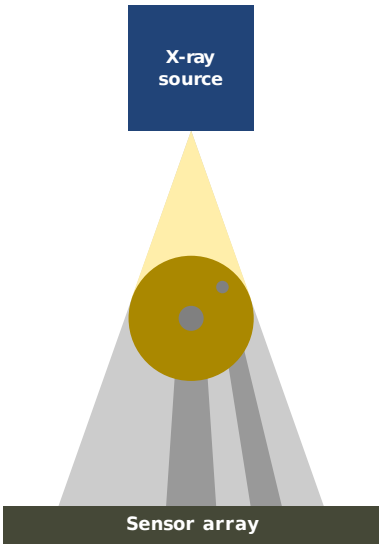


Figure 4: Visualization of the cone effect that arises from having a single point source. The brown circle depicts a potato with two internal objects with a higher density than the potato, resulting in darker shadows.

XCOM NIST database [13], we get the attenuation curves in Figure 2 by entering these compositions. Suppose we choose an energy right where the two curves overlap, then we cannot tell the difference from the penetrated beam since both materials will absorb the same amount of energy. So, to tell the two materials apart, we want to choose an energy level, where the amount of energy absorbed by each of the materials is distinguishable. For example, such a choice could be below 10^2 or above 10^4 for the needle and hay in Figure 2. While we can now choose energies that produce clear contrasts between two materials, it might not be as easy once we go to three materials. To solve this, we can gather even more information about the scanned materials by capturing the beam produced at two different energies.

An X-ray source produces X-ray photons controlled by two parameters: voltage, the energy of the photons, and ampere, the number of photons. However, these parameters describe the values going into the X-ray source, not the actual energies and amounts produced. Figure 3 shows the simulated energies produced at different voltages. We see that the energies span a wide range rather than a specific one but that the maximum energy reached matches the input voltage. We can handle the low-energy photons by using clever filtering techniques. E.g., suppose we are only interested in photons at energies higher than 30 KeV. In that case, we can insert a filter, which will absorb all the low-energy photons – with high probability. We cannot filter the high energies simultaneously, but we control the maximum energy through the voltage.

When looking at the photons, we must consider that they are all produced from a single source, effectively emitting a light cone. When we have our scanned subject right below this single source, the shadows cast will not be precisely below the subject, resulting in the shadows becoming "stretched" based on the angle. Figure 4 illustrates this effect, where we can see that even though the upper-right object is only half the width of the center object, the relational width, once they hit the detector, is only 1.59, where the actual relation is 2.00. This cone effect could hide some objects; lighter objects could "hide in the shadow" of denser objects. The cone effect also introduces a halo effect on the

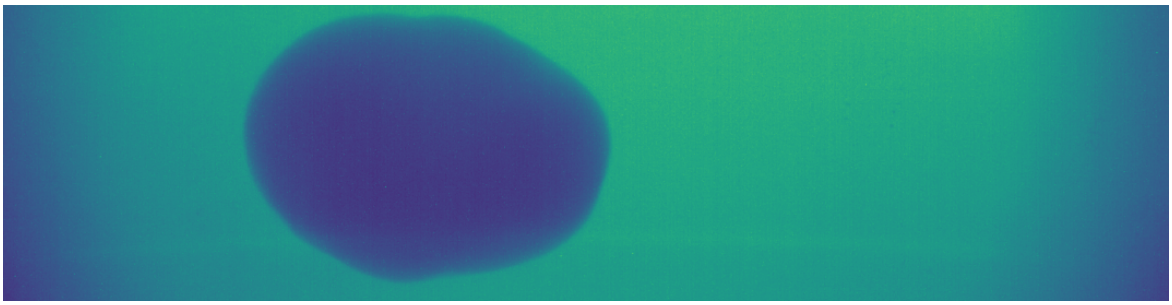


Figure 5: Halo effect on an X-ray image with a single X-ray source.

images. Figure 5 shows the effect on a resulting X-ray image, where we see a ”glow” irradiating from the center of the image. To produce clean images, we would have to correct this as well.

A final note about X-rays is that the high energies can be destructive, both for living tissue and sensitive machinery. It is generally not a problem for small bursts of exposure for living tissue since the probability of the photons interacting directly with the tissue is relatively low. Inversely, longer exposing times raises the probability of interaction. There is the probability of introducing or draining electrons, currently carrying a state, occurring both in wires or transistors for electronics. As such, any running electronics under X-ray exposure has a probability of producing random errors. In the resulting images, these random errors usually become *salt-and-pepper* noise, which are single pixels attaining either minimum or maximum possible value.

Furthermore, the introduction of electrons can translate into an overload of electrons in the system, resulting in permanent damage to the internal components. The longer exposure time, the higher the probability, which means electronics deteriorate over prolonged exposure. Therefore, we must continuously do calibration and corrections, as the X-ray side-effects would otherwise have too strong an impact.

While all of the effects covered in this chapter do not directly affect the program produced in this thesis, it is essential to keep these details in mind, as they directly limit the program’s capabilities. For instance, we can only make predictions as good as the images we receive, so choosing the proper voltage and ampere to maximize contrast for the given materials is essential.

4

IMAGING

Like X-ray, we have to dive into imaging technology, as there are some physical effects that we have to consider, as these directly impact the images. Cameras are devices that capture the light reflected off of a surface as a two-dimensional grid of values. Depending on the surface it hits, the light will be oscillating at different wavelengths. Humans interpret these wavelengths as color, with blue at the low end of the spectrum and red at the high end.

We convert the analog signal into a digital signal through discretization to obtain digital images. The most common sensors are Charge Coupled Device (CCD) or Complementary Metal Oxide Semiconductor (CMOS), which work by having a grid of sensors with each sensor constituting a pixel in the resulting image. As a result, the density of the sensor grid directly controls the image resolution. When a photon hits one of these sensors, the photon is converted into electrons, essentially counting as a photon hit. When capturing an image, the camera exposes the sensors to the scene for a set amount of time, during which the sensor accumulates the number of photons hitting it. The longer the exposure time, the brighter the image since more photons will be counted. After the exposure, the camera drains the accumulated values into a matrix of values, which it stores on its internal storage as the final image, resetting the accumulators along the way. Each color has a sensor array for color images, which commonly constitutes red, green, and blue. Each sensor array filters the photons, only allowing photons of a specific wavelength to pass through, essentially only counting photons of a specific wavelength - or color.

Images consist of multi-dimensional matrices of values. The last dimension of the matrix varies depending on what they are representing, usually relating to the number of channels. For example, the last dimension of black and white images is 1, whereas the last dimension of color images is 3 to 4. Some cameras are focused on one task, while others carry the functionality to do both. The fourth channel is commonly used as an alpha channel, describing the pixel's opacity, which is often used in computer graphics and thus not captured by a camera.

The exposure time relates to the sampling rate, as a reduced exposure time translates to a higher sampling rate. The sampling rate is essential when dealing with moving objects. Consider an object whose reflected photons would hit a single sensor if the object were stationary. Let us assume that the object is moving with a velocity corresponding to moving one pixel along the sensor grid in one direction every millisecond. If the exposure time is five milliseconds, the object will contribute photon counts to five sensors, giving the effect of a stretched, less intensive object in the resulting image. This effect is known as *motion blur* and shows as a stretching and blurring of the moving objects in the direction the object is moving.

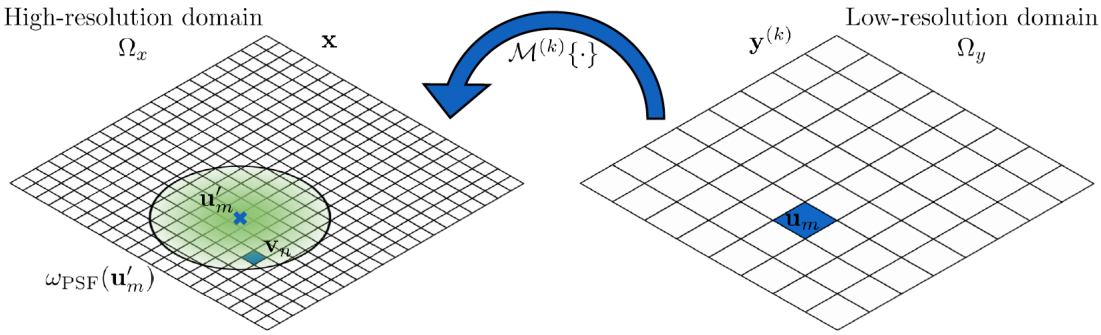


Figure 6: Visualization of how the low-resolution domain relates to the high-resolution domain. Image from [16].

4.1 SUPER RESOLUTION

The master's thesis by Simon Nyrup titled "Applied Super-Resolution for X-Ray Imaging - Virtual Potatoes And How To X-Ray Them" [15] was supervised as part of this Ph.D. thesis and is covered in Appendix B.1.7. The master's thesis had two parts: super-resolution and building an X-ray simulator (covered in Section 5.1).

Half of the thesis looked at whether we could counter motion blur through a series of images of the same object, in turn improving image resolution. It leveraged the fact that an image discretizes the natural world, which means it is a low-resolution observation of a high-resolution domain. This restoration is known as *Super-resolution*. Super-resolution is the reverse operation, where we try to restore an image to the high-resolution domain through the probability that the photon count of a sensor originated from a distribution in the high-resolution domain. Figure 6 visualizes this probability relation between the two domains.

The master's thesis proposes the Iterative Re-Weighted Super-Resolution (IRWSR) algorithm, which is a Multi-Frame Super-Resolution (MFSR) algorithm. To perform super-resolution, we first have to match up the image series. After matching, each image x contributes its values to the resulting high-resolution image $\hat{x}$ through its weights α, β, and λ. The algorithm alters between estimating the image and the weights and runs for multiple iterations.

After a set number of iterations, we obtain the reconstructed high-resolution image $\hat{x}$. Figure 7 shows the original high-resolution image, the same image artificially degraded, and the reconstructed high-resolution image based on the artificially degraded image. We see that the stripes on the scarf and some facial features are more defined than the degraded image, showing that the image quality has improved.

4.2 SEAM CARVING

While we want high-resolution images that carry a high degree of information, the higher resolutions increase the computational complexity required to process them. Downscaling the images allows us to retain some of the information but still discards potentially valuable information. The most general approach is interpolation, where neighboring pixels contribute equally to the downscaled

Figure 7: Images showing the process of super-resolution. **Left** shows the original image. **Middle** shows the artificially degraded image. **Right** shows the artificially degraded image upscaled through the IRWSR algorithm. Images from [15]

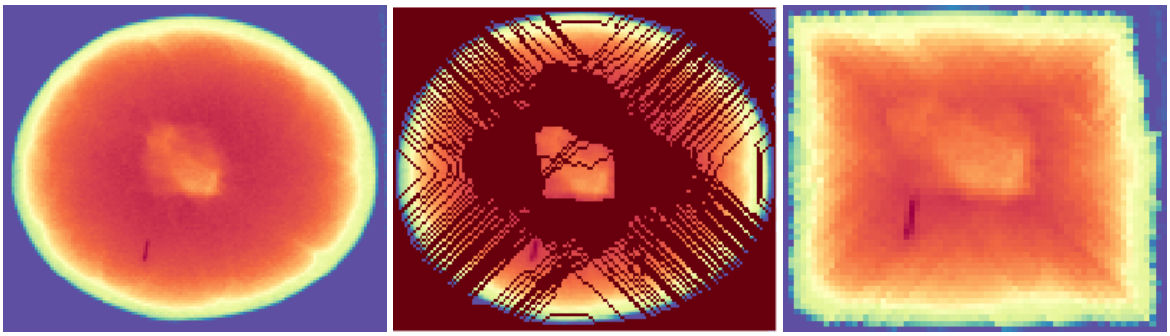


Figure 8: Seam carving applied to an orange with a needle inside. From the left we see the original image, the seams that are removed, and the image with the seams removed. The image has been reduced from 115x135 to 57x67. Images from [14].

image. However, this approach is not viable for small features, as they can get squashed, given that their contribution will have less of an impact due to their size. Optimally, we would remove the uninteresting parts of the image while keeping the prominent parts intact.

The master's thesis by Aleksandar Topic titled "Automating Classification in Food Inspection" was supervised as part of this Ph.D. thesis. Amongst other things, this thesis investigated the content-aware image resizing algorithm known as *seam carving*. The algorithm works iteratively in either the horizontal or vertical directions, where it computes an energy map of the image, describing the change in information. The idea is to find a path through the image in either direction that carries minimal information. The intuition is that we will have made the least intrusive data manipulation concerning contrasting features by removing this path. Suppose we have an image with a large homogeneous region. By removing a path through this region, a homogeneous region will still exist, but all of the other details of the image remain. Figure 8 depicts an image before and after removing seams. Notice how the algorithm preserves the center and the needle while reducing the background and homogeneous regions.

We can express the algorithm as a single equation:

$$M(i,j) = E(i,j) + \min[M(i-1,j-1), M(i-1,j), M(i-1,j+1)] \tag{2}$$

M is the minimum energy matrix, and E is the energy matrix. The row i of M has its values obtained from the accumulated sum of a path through the previous rows. The energy map, E, can be computed in many different ways, but the straightforward approach uses Sobel filters. In this case, the energy

map consists of the gradients of the input image in the x and y directions. With this algorithm, we can downscale images without removing prominent features.

5

X-RAY IMAGING

As the name implies, X-ray imaging applies traditional imaging technologies to light in the X-ray spectrum. However, where regular sensors can capture photons in the visible spectrum, photons in the X-ray spectrum cannot be captured, as a large percentage of these photons will penetrate the sensor. One solution is to ”convert” the photons back into the visible spectrum. The most common method used for converting x-rays to photons is a scintillator.

A scintillator is a material that, when hit by photons in the X-ray spectrum, gets excited, in turn emitting photons in the visible spectrum. Different scintillators have different properties, such as the number of photons emitted, the number of X-ray photons penetrating the scintillator, and the required thickness. An essential property of scintillators is how fast they become calm again. This calming factor is especially prominent when scanning moving objects, as light pollution will show a blurring effect, similar to motion blur in traditional imaging but more prominent in X-ray imaging.

X-ray imaging uses either an array of sensors (a *line scanner*) or a grid of sensors (an *area scanner*). There is a scintillator layer between the source and the detector regardless of which technique we utilize. Line scanners have the advantage of running at higher capture rates, given that there are fewer sensors to drain in between each capture. Their disadvantage comes from moving the subject to be scanned to construct a two-dimensional image as they only capture one row. Area scanners have the advantage of not reconstructing the image, as the grid of sensors handles this. Their disadvantage is their lower capture rates, which translates into increased exposure time, resulting in additional motion blur.

Both types of cameras have the problem of deterioration by prolonged exposure to X-rays. The complexity of electronics determines the rate of failure, or rather the probability of a failure happening - the denser the electronics, the higher probability of an X-ray photon interacting with the matter. As line scanners are less complex than area scanners, they are less likely to fail, increasing their popularity in X-ray imaging. One way of circumventing the deterioration problem is by using mirrors to divert, or *bend*, the visible light while being penetrated by most X-rays. As such, we can bend the visible light into a shielded region where the detector resides by using mirrors, such as Figure 9 depicts. While some X-rays will still directly hits the detectors, there will not be as many, resulting in a lower probability of failure.

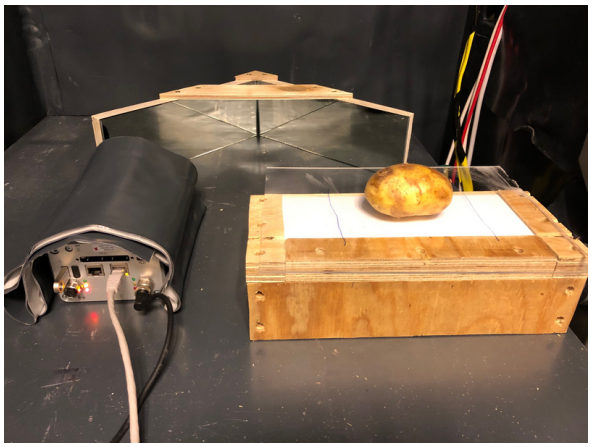


Figure 9: Closeup of how the camera is shieled by using mirrors. Image from [15].

5.1 X-RAY IMAGING SIMULATOR

The master's thesis by Simon Nyrup titled "Applied Super-Resolution for X-Ray Imaging – Virtual Potatoes and How to X-Ray Them" [15] was supervised as part of this Ph.D. thesis and is covered in Appendix B.1.7. The master's thesis had two parts: super-resolution (covered in Section 4.1) and building an X-ray simulator.

The motivation behind building the simulator arose during the 2020 covid pandemic. We needed X-ray images, but we could not go to Newtec to use their setup, so we looked into simulating these images. The simulator uses an approximation of X-ray imaging properties in a discrete space. It functions by having a single point in the three-dimensional space, acting as the X-ray source. At the bottom of the space, a grid of points spread across a plane acts as the detector. Intuitively, we place the object to be scanned somewhere between the X-ray source and the detector - simulating a real-world setup. The objects are in voxel form, which allows us to vary the internal resolution of the objects by tuning the voxel density.

When rendering an image, the simulator casts a ray for each point in the detector to the X-ray source. We then check each of these paths for whether they pass through an object. If so, the number of voxel hits and the type of the voxels are stored. Then, according to the material, its corresponding mass attenuation curve, the voltage of the source, ampere of the source, and the exposure time, we compute the final pixel value. Figure 10 visualizes this exact setup. Figure 11 shows the images produced by the simulator.

Given that this simulator was not the focus of the master's thesis, there are some shortcomings. Firstly, having the models in a voxel representation has flaws regarding precision and performance. Originally it was chosen due to its ease of implementation. By having the object in a mesh-based representation, we could increase the precision and performance at the cost of increased implementation complexity. Secondly, the simulation model does not truly capture the odd natural behavior of the photons, such as scattering or beam hardening. By applying the mesh-based representation, we could also move more quickly to using some widespread ray-tracing methods found in modern hardware, further increasing the accuracy and performance of the resulting images. Regardless of these flaws, the resulting simulator is a powerful tool for producing X-ray-like images without the risks or capital losses of building an X-ray imaging machine.

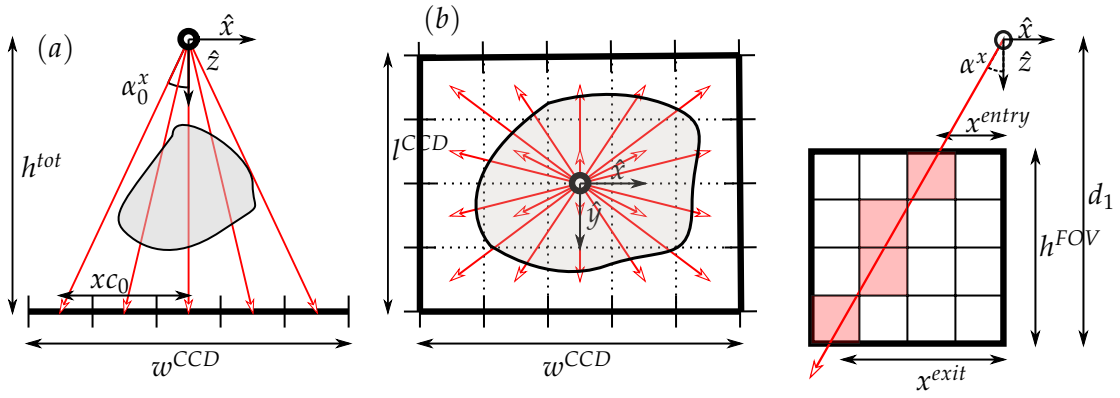


Figure 10: Visualization of the X-ray simulator setup. From the left we have a side view, top view, and line approximation through the voxels. Images from [15].

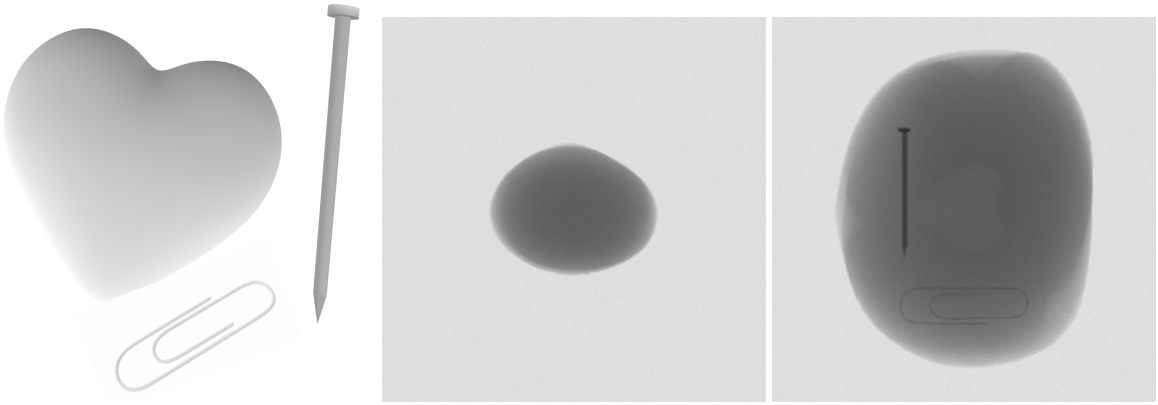


Figure 11: Output from the X-ray simulator. From the left we have anomalies, a potato, and the same potato with the anomalies ”inserted” inside. Images from [15].

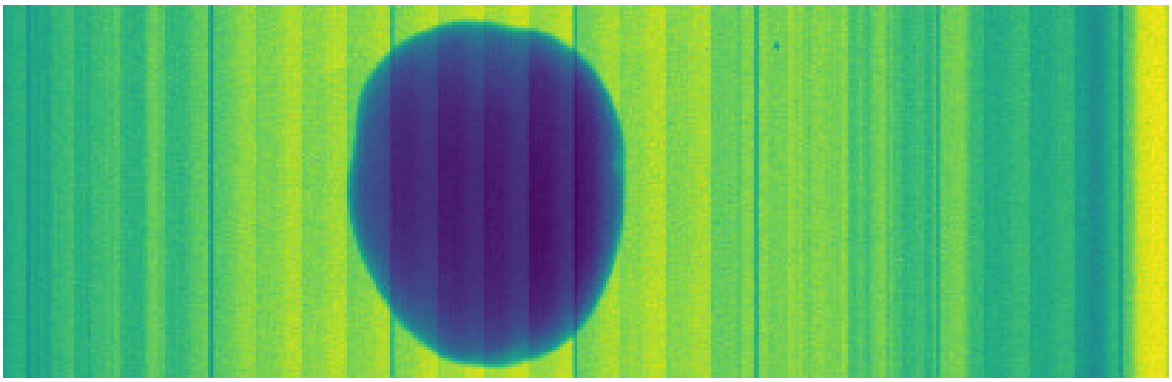


Figure 12: Raw X-ray image of a potato showing stripes generated by the setup. Image from [17].

5.2 CORRECTING THE IMAGES

We have all these physical effects from the X-ray source, scanned object, scintillator, and camera, so we are getting far from perfect pictures. While we could use them as-is, many of these effects can be reversed or corrected, given that we know their effects when they occur and to what extent they do occur.

The master's thesis by Troels Ynddal, titled "High Throughput Image Processing in X-ray Imaging," [17] ran in parallel at the beginning of this Ph.D. thesis and is covered in Appendix B.1.1. Part of it focused on improving and optimizing data quality in X-ray imaging. While the X-ray setup did have an option to calibrate the images, the process is proprietary and closed-source, leaving no room for improvement. This calibration became a problem since this calibration was too general at times, resulting in strange artifacts. We can apply extra information about the subjects and energies that the detector did not allow by implementing an open-source solution.

The first step was to correct the raw signal from the detector. As shown in Figure 12, the image contains stripes, known as Fixed-Pattern Noise (FPN). To correct these, we use a Flat-Field Correction (FFC) algorithm. To do this, we need to capture two images for calibration. One, with no X-ray exposure, D, and one with full X-ray exposure, F. Then, for each image, P, we can compensate:

$$N = \frac{P - D}{F - D} \tag{3}$$

As shown in Figure 13, we now have a cleaner image. This process can also counter the halo effect described in Chapter 3. However, as we can see, some noise remains in the image. The master's thesis suggests that using Adaptive Median Filter (AMF) essentially smooths the image, countering the high salt and pepper noise rate commonly found in X-ray imaging. Compared to a regular median filter, AMF allows itself to grow the target region, from which it gains the adaptive part. This adaptiveness is better at removing salt and pepper noise than a regular median filter, as different windows can catch strong outliers.

Finally, the thesis also describes a Region Of Interest (ROI) algorithm for extracting the objects from the rest of the scene and removing the background. The algorithm traverses the image iteratively, marking pixels as belonging to an object based on its previous neighbors. This ROI algorithm benefits line scanners since they create a continuous image, capturing several objects, and because it can work

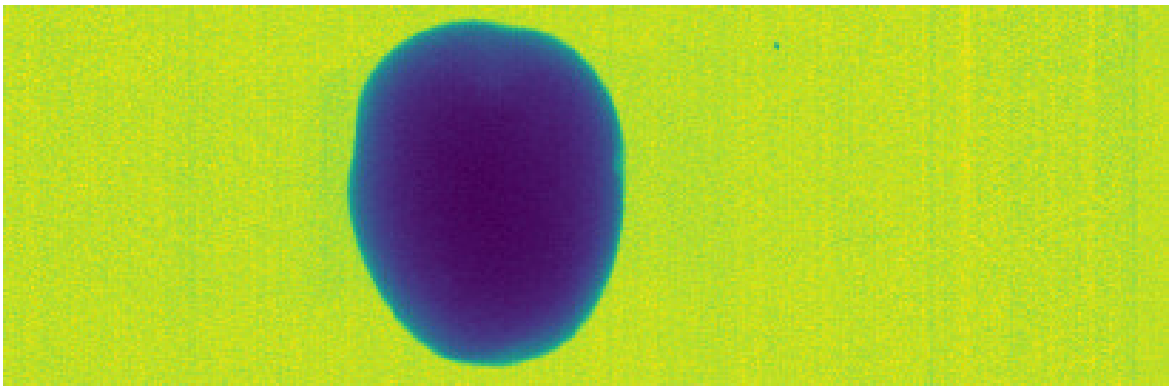


Figure 13: Corrected X-ray image of a potato from Figure 12. We see that the lines from before are not as intrusive. Image from [17].

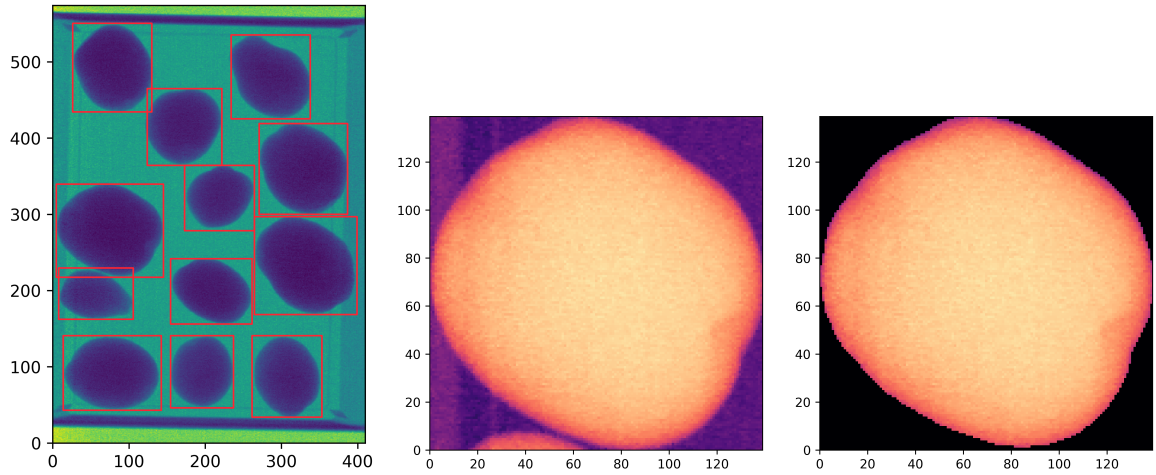


Figure 14: The three major steps of the ROI algorithm. From the left we have an image with multiple objects that are found and highlighted with red boxes, one of the potatoes extracted, and the same potato with the background removed. Images from [17].

on one line at a time. Figure 14 shows the steps of the algorithm applied to an image containing several potatoes.

5.3 SUPER-RESOLUTION ON X-RAY IMAGES

As covered in Section 4.1, we have a method for improving the image quality of objects based on a series of images from different angles. We get precisely multiple images of the same object as they move past the sensor for area scanner-based X-ray imaging setups. The idea was then to use super-resolution on this series to obtain a higher-quality image. While we improved the images slightly, as shown in Figure 15, small features did not improve significantly. Furthermore, the computational cost was too high, with the master's thesis author suggesting that using an Machine Learning (ML) approach might produce the same results at a lower computational cost.

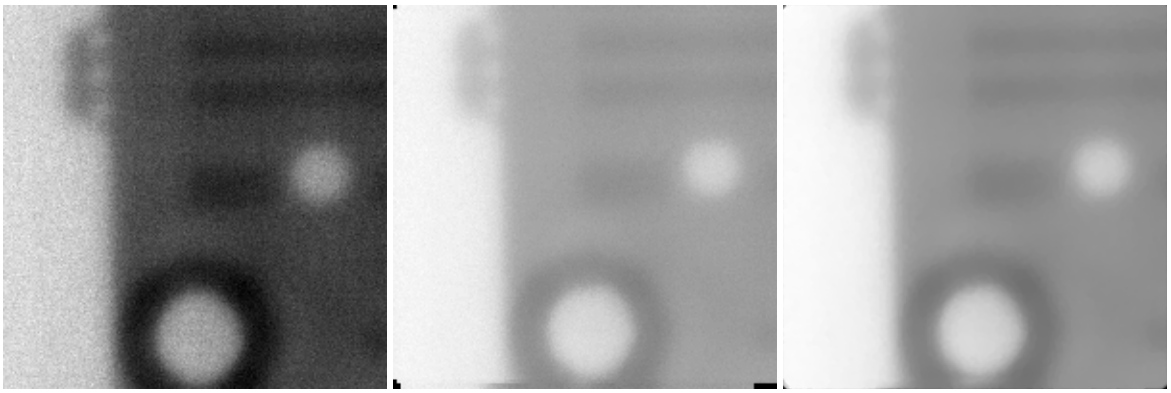


Figure 15: Super-resolution applied to X-ray images of a circuit board. From the left we have one of the original low-resolution image, the median of the combined low-resolution images, and the reconstructed super-resolution image. Images from [15].

5.4 EXTREME SEAM CARVING

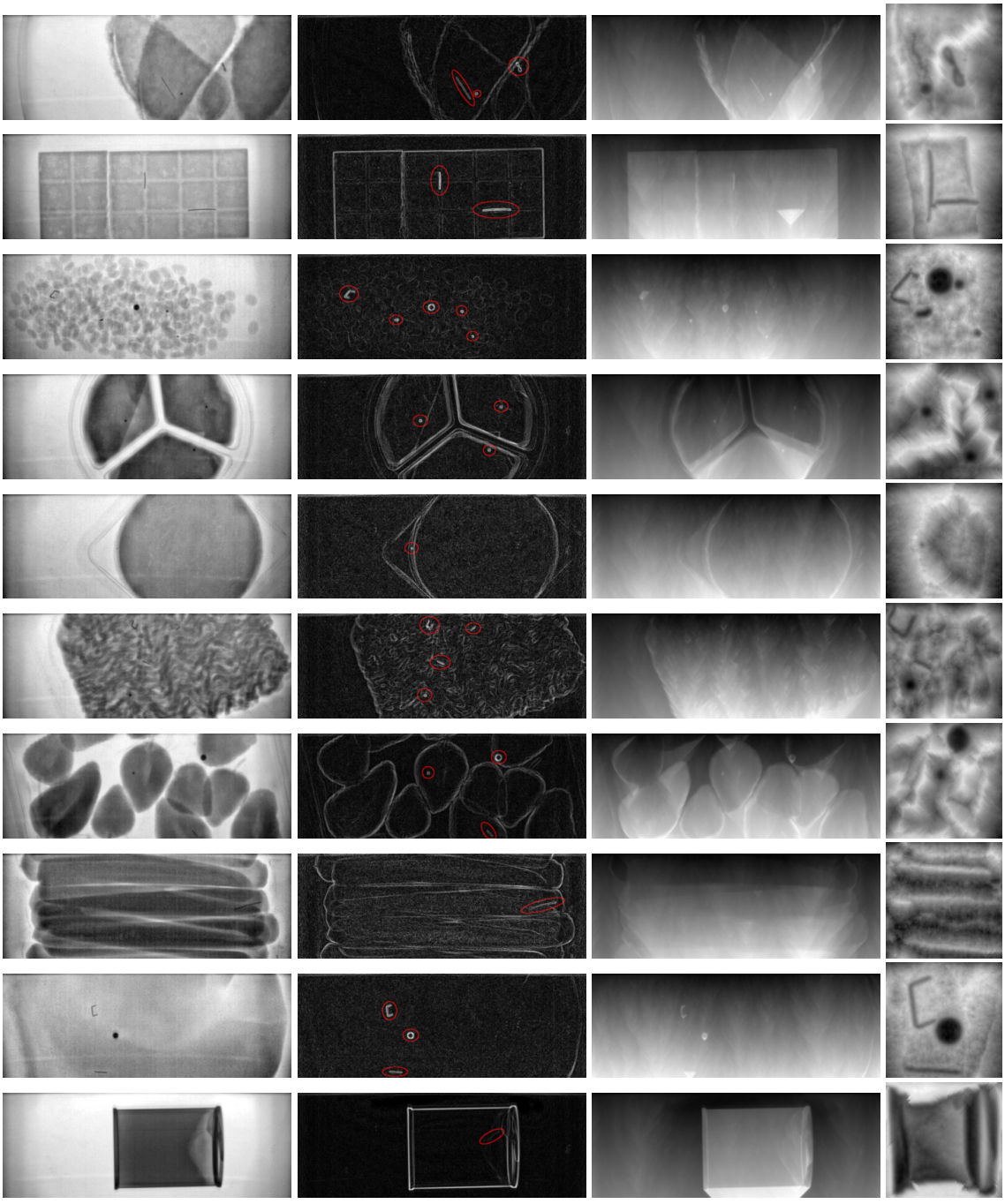
As covered in Section 4.2, we presented seam carving as an image resizing method, which retains prominent features. The results seemed promising, which is why we set up an experiment with multiple X-ray images of different kinds of food. We wanted to see how far we could push the algorithm and whether small foreign objects would remain. This work is covered in our paper, which is under submission.

We gathered the different foods based on whether we thought it would be hard to see anything in the resulting image, which primarily meant whether or not the structure of the foods carried high entropy. Table 4 lists the different foods, the voltage used, and the expected difficulty of seeing the foreign objects. Figure 16 contains the raw X-ray images, their energy map, the positions of the foreign objects, and post seam carving. We see that even though we reduce the size of the images by 98%, we can retain the foreign objects. While we cannot recognize the foods in the seam carved images, we see that the foreign objects are well preserved, both in size and shape. Interestingly, some of the images we thought would be high-difficulty to be some of the best cases. For example, breadcrumbs are almost entirely homogeneous in the X-ray image, having almost perfectly preserved foreign objects in the resulting image. Inversely, the seemingly low-difficulty case of sausages proved poor results, where the metal is hard to find in the seam carved image. This result seems to be due to the air separating the sausages after introducing the foreign object.

The work done in the paper shows that seam carving is a viable method for reducing image resolution while keeping notable features, such as foreign objects.

5.5 AUTOMATED TUNING OF THE SETUP

The master's thesis by Aleksandar Topic titled "Automating Classification in Food Inspection" was supervised as part of this Ph.D. thesis and is covered in Appendix B.1.4. Amongst other things, this thesis investigated automatically deriving the optimal parameters for maximizing contrast in an X-ray setup. The method is to take multiple images for mapping the parameter space. Then we compute a histogram for each image, which is then automatically thresholded based on Otsu's method to isolate the objects in the histogram. Finally, we choose the parameters that maximize variance in the



(A) Input (B) Energy map (C) Accumulated energy map (D) Output

Figure 16: X-ray scans of different foods with varying levels of complexity. We added foreign objects to the foods highlighted in the (B) column with red circles. The input (A) is seam carved to a final size of 72x72 pixels in (D), which translates to 98% of the pixels removed.

Product	Voltage used	Difficulty
Chocolate bars	60 kV	Low
Breadcrumbs (rasp)	60 kV	High
Chicken nuggets	75 kV	High
Humus	60 kV	Low
Sausages	60 kV	Low
Strawberries	60 kV	High
Oranges	60 kV	High
Minced meat	65 kV	High
Cold cuts (meat)	60 kV	Low
Coffee beans	35 kV	High
Can of tomato pure	90 kV	High

Table 4: Overview of the different foods scanned for the seam carving paper. Table is from [18].

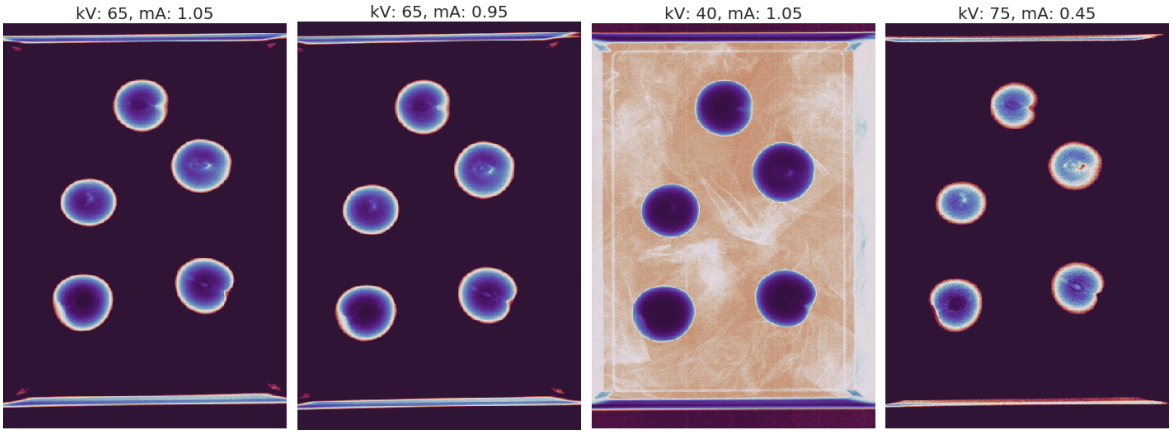


Figure 17: X-ray images of plums at a varying voltage and ampere. For the fourth image, we see that the majority of the X-rays penetrate the plums, washing out the edges. For the third image, the voltage is too low to penetrate the plums fully. The first and second images are almost inseparable, but the first image has the largest standard deviation. Image from [14].

histogram, thus maximizing contrast inside the objects. Figure 17 shows images of plums at varying parameters. We see that we can see the entire plum and its internal structure with the right parameters.

6

SUBCONCLUSION

In this part, we started by covering the fundamental X-ray physics in Chapter 3. While this primarily relates to the work of the other partners of Adaptive Xray InSpection (AXIS), we need to be resilient to the side-effects that X-rays introduce. Then we covered the process of image capturing and how it is a discretized capture of the natural world in Chapter 4. This discretization means that there will be imprecisions and flaws that we have to consider when processing the images. Combining the theory of X-ray physics and image capturing, we present the theory of X-ray imaging in Chapter 5. The key takeaway is that stochastic effects drive X-ray imaging, which we must consider to counter the images' artifacts.

In Section 4.2 we presented the work of a master's student supervised during this Ph.D. thesis: an approach for content-aware image resizing; *seam carving*, a method for removing all of the uninteresting parts of an image. We expanded this work after the student project by applying the method at extreme levels to X-ray images of food, as shown in Section 5.4 and our paper [18]. We removed 98% of the pixels while retaining the foreign objects inserted into the foods. This performance suggests that this method is of strong use for later classification.

In Section 4.1 we presented the work of a master's student supervised during this Ph.D. thesis: an approach for extracting high-dimensional information from a series of low-dimensional samples; super-resolution for countering the discretization of cameras. While the work did show an improvement when applied to X-ray images in Section 5.3, the improvement was overshadowed by the computational complexity of the algorithms, proving it to be infeasible to apply in an inspection setup.

In Section 5.1 we presented the work of a master's student supervised during this Ph.D. thesis: an X-ray simulator, which can produce X-ray images under perfect conditions. This simulator is handy since X-ray setups are costly and have high safety concerns. Furthermore, we could use this simulator to generate data for our later classification models.

In Section 5.2 we presented the work of a master's student that ran in parallel with this Ph.D. thesis: image processing techniques of X-ray images, countering the stochastic effects seen on X-ray images. These techniques are crucial for cleaning the data for later work.

Finally, in Section 5.5 we presented the work of a master's student supervised during this Ph.D. thesis: an automated approach to tuning the parameters of the X-ray setup. While we do not necessarily have direct control over these parameters for the final machine, it still paints a picture of the considerations of an X-ray imaging setup.

Part III

MACHINE LEARNING

7

LEARNING FROM DATA

Machine Learning (ML) is a subfield of applied statistics, where one tries to build an algorithm, which can learn a pattern from a given data set. The concept is that rather than having a program that knows what to look for ahead of time, a ML program will adapt itself to fit the problem at hand. A simple example is housing prices, where many factors determine the price of a house, such as location, age, and size. How much each factor contributes to the total price is not known prematurely but can be deduced from a series of entries. An ML program would try to fit the weight of each factor such that the resulting total value would match that of the actual entries. In general, while the algorithms are adaptable, they still focus on solving a single task. For the entirety of this part, we will focus on the application of ML on images, as this is beneficial for the Adaptive Xray InSpection (AXIS) project. From the highest point of view, ML consists of three main subjects: preprocessing / data cleaning, model building/tuning, and finally, model inference/evaluation. As part of this Ph.D. thesis, I have investigated these subjects through the supervision of student projects and my work. This part should stand as a guideline for the crucial parts of building a classifier for X-ray images of food.

As ML tries to learn from data, we want to make these features as distinctive as possible. Data can be considered either signal, the subject we are looking for, or noise, irrelevant or intrusive data. Before trying to learn, we want to maximize the signal-to-noise ratio. Chapter 8 covers the most widely used preprocessing steps based on what we have experienced through student projects.

While the area of ML is vast, this thesis focuses on variants of the Artificial Neural Networks (ANNs), as their power has been growing alongside the growth of computing power, especially for images. ANNs are networks of Neurons, which, based on their input, make a decision, mimicking the neurons in the human brain's function. Chapter 9 covers the variants of ANNs investigated as part of this thesis and how we tune them.

Once a model has been implemented and trained, it is ready to make predictions, known as *inference*. We can evaluate the model's performance based on these predictions by measuring generalization, confidence, and accuracy metrics. These metrics help us build an intuition about the model's strength, why it fails, and why it succeeds. Chapter 10 covers these metrics, which we will use in the later parts of the AXIS project.

Finally, to show the entire application of ML from start to finish, we present the project we did for a danish brewery in Chapter 11.

8

PREPROCESSING

As data usually consists of natural world observations, the samples will contain imperfections; a side effect of the discretization introduced by measuring equipment. As an ML model will learn the patterns of a dataset, we must do what we can to remove these imperfections, as they can represent false patterns. This process is known as *preprocessing*, the act of preparing data for ML. Preprocessing is an integral part of ML and can boost the performance or convergence of the resulting models. It comes in several forms: data manipulation, feature extraction, data augmentation, or data cleaning, to name a few.

8.1 SMOOTHING

A common first step in image processing is to smooth the image. Smoothing will push the values in the extremes, which is the usual form of noise, towards the mean value of a local neighborhood. The smoothing rate can also be a tool for controlling the size of the resulting features. E.g., if one wants to only look at objects of a specific size, using a large smoothing kernel will remove all of the small objects as they will have a small contribution, leaving only the large objects. A mix of scales can, in some cases, be beneficial, especially if one does not know the size of what to look at beforehand or if said subject can differ in size.

We apply the smoothing operation through *filtering*, where we consider a window (or local neighborhood) of the total image. This window then slides across the image, considering different localities, providing the values for the resulting image. The size of the window does not have to be fixed but can vary during the application, defined as *adaptive* filtering, as mentioned in Section 5.2.

Many different smoothing techniques exist, differing in how we handle the window. We consider four smoothing methods:

MEDIAN FILTER - the result is the median of the window. It is very good at removing extreme outliers, such as salt and pepper noise.

MEAN FILTER - the result is the window's mean. As every pixel in the window contributes equally, the smoothing will introduce some square-like artifacts.

GAUSSIAN FILTER - the result is a weighted mean with the kernel constituting the weights. Compared to the mean filter, the transitions will seem smoother, as the "near" pixels will have a higher contribution.

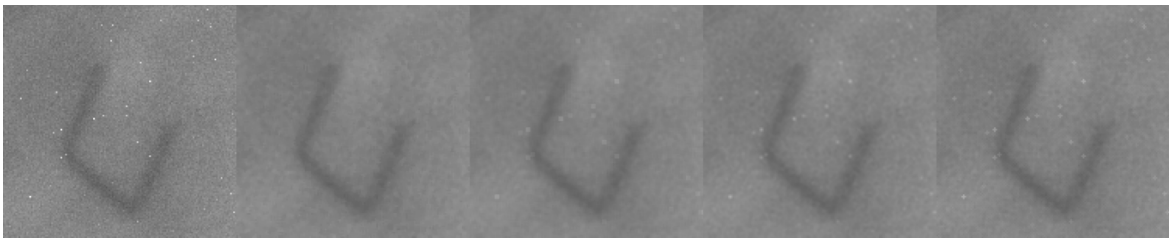


Figure 18: Effects of blurring an X-ray image of meat with a paperclip. From the left we have: the raw image, median filter, mean filter, gaussian filter, and bilateral filter. All of the kernel sizes are set to 3 by 3.

BILATERAL FILTER - which removes noise using weighted average and preserves edges using the variation in the image. Locally, it has much of the same effect as the gaussian filter, but globally it will preserve edges better.

Figure 18 shows the application of the different filters to an X-ray image of meat. Notice that the salt and pepper noise is not present in the median filter. The noise remains in some form for the other filters, but more aggressive use could remove it at the cost of additional information removal. Whether or not we want to keep the noise, we should use different smoothing techniques.

8.2 IMAGE SEGMENTATION

Image segmentation is where the object(s) is extracted from the image, resulting in one or more images with only the objects. Another term for the same operation is Region Of Interest (ROI), which we covered in Section 5.2. The extraction reduces the complexity required by a ML model, as it no longer considers the entire image but rather only the extracted parts. E.g., if we presented the entire image, the model might learn the location of the objects, which does not - at least not in food inspection - contain any vital information.

The method presented in Section 5.2 based its segmentation on the image after it had been thresholded to either 1 or 0. The master's thesis by Aleksandar Topic [14] proposed using Otsu's binarization as a non-parameterized tool for thresholding an image. It works by computing the histogram of the image, in which we consider two classes: foreground and background. The idea is to minimize the weighted in-class variance of two parts of this histogram. Figure 19 shows a histogram of an X-ray image of meat, and where Otsu's binarization will put the optimal split. Below the histogram, we see the original image, the binary mask after thresholding, and the meat isolated from the background. By using this mask, we can use the ROI method presented in Section 5.2 to extract the masks into individual images.

8.3 CONTRAST IMPROVEMENT

While ML algorithms are good at noticing small differences, emphasizing interesting parts in the image is still beneficial to the algorithm. When we improve contrast, the ML model converges faster as the magnitude of the gradients becomes greater, compared to using raw images. Overall, we consider two contrast improvement techniques: gamma correction and histogram equalization.

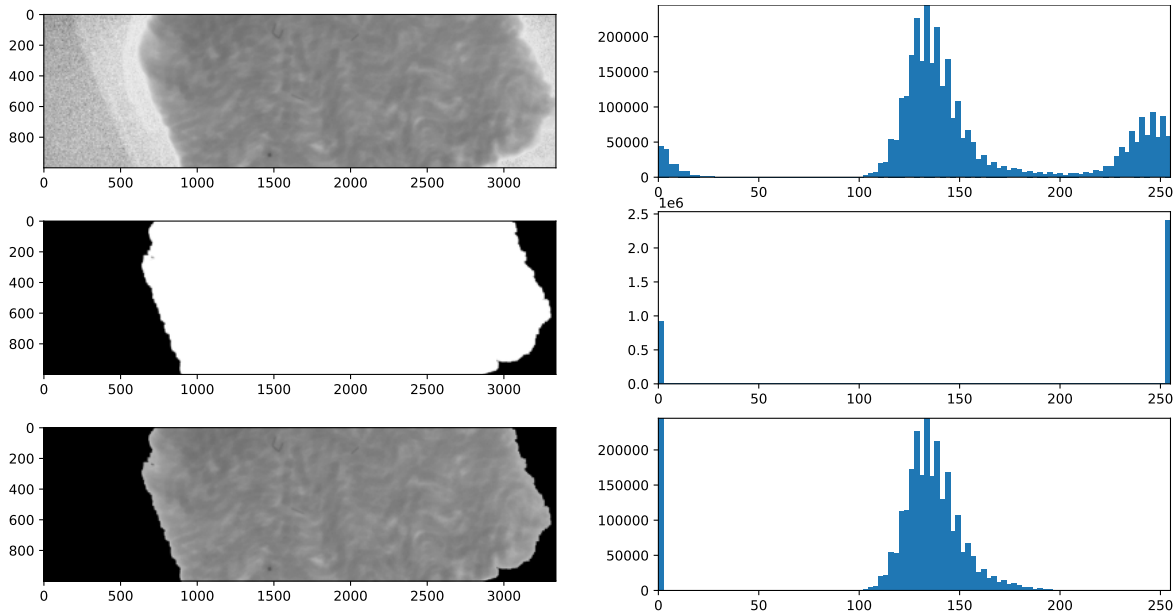


Figure 19: Histogram and segmentation of an X-ray image of meat. From the top we have the raw image, the mask obtained by otsu's thresholding, and the mask applied to the raw image.

Gamma correction is a method for correcting exposures from a camera. The idea is that while sensors might capture the world linearly, the human eye senses changes logarithmically. Gamma correction thus corrects the image's lighting, either making the image darker or brighter. While it might not improve the performance or convergence of a ML model, it is a cheap modification, which works for human perception. Figure 20 summarizes the application of gamma correction to an X-ray image. We have seen gamma correction work in multiple student projects [14], [19], [20].

The input image might not span the entirety of the input range. The black and white images presented throughout this thesis have been using unsigned 8-bit for describing a pixel, resulting in the value range 0-255. Spreading the input across the entire span will help the algorithm see differences since their now more distinctive. Figure 21 shows the application of Histogram Equalization (HE) and Contrast Limited Adaptive Histogram Equalization (CLAHE) to an X-ray image of a paperclip in meat. Notice how the histograms span the entire range and that the paperclip is more defined.

8.4 IMAGE RESIZING

While larger images contain more information than smaller images, they might contain too much redundant information. The redundant is not a problem correctness-wise, but rather performance-wise, as larger images require more computing resources, which we waste on redundant information. Decreasing the image size while keeping the essential details would reduce the required computing resources. There exist many generic resizing methods, which treat the pixel contribution differently. However, more exotic algorithms exist, which retain non-homogenous regions, thus removing the uninteresting parts of an image. Such an algorithm has been covered in Section 4.2, Section 5.4 and our paper [18]. Figure 22 shows different resizing approaches applied to an X-ray image. Notice how some retain the small objects and edges while others "squash" them.

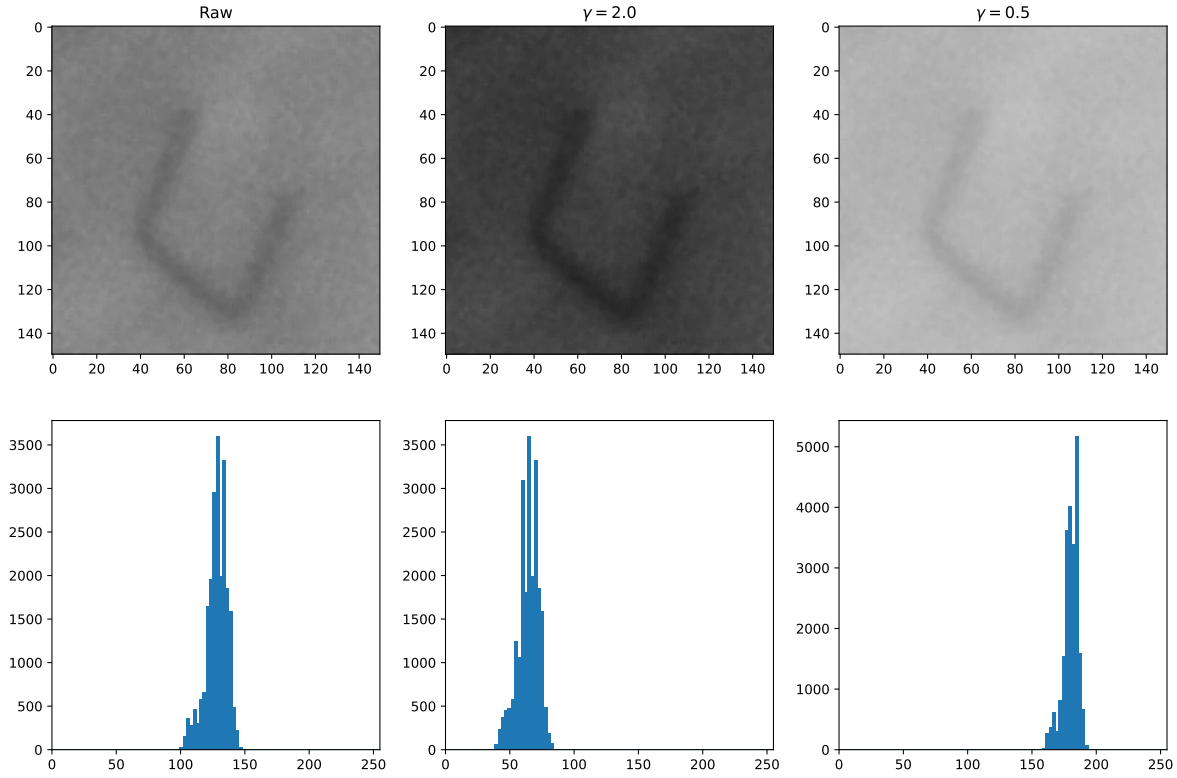


Figure 20: Gamma correction applied to an X-ray image of meat. Notice how $\gamma > 1$ brightens the image, while $\gamma < 1$ darkens the image.

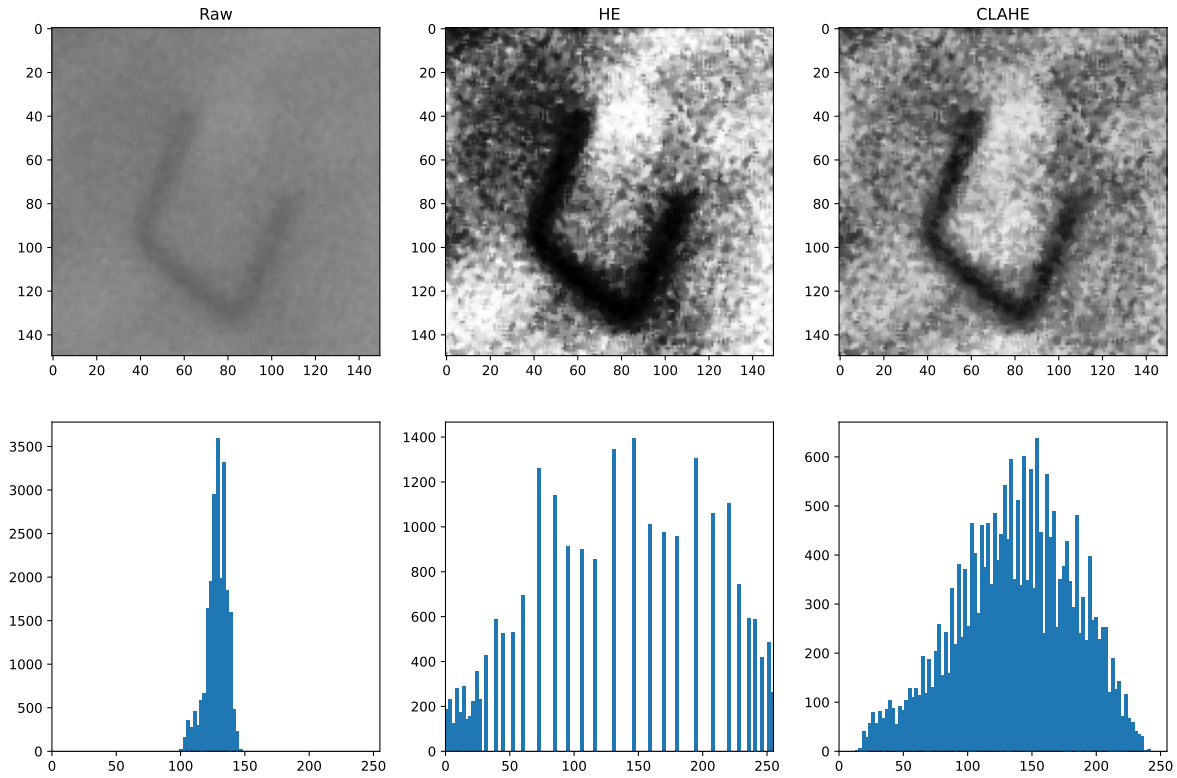


Figure 21: Different histogram equalizations on an X-ray image of meat. HE uses the default parameters. The CLAHE had a clip limit of 10 and tilegrid of 4 by 4.

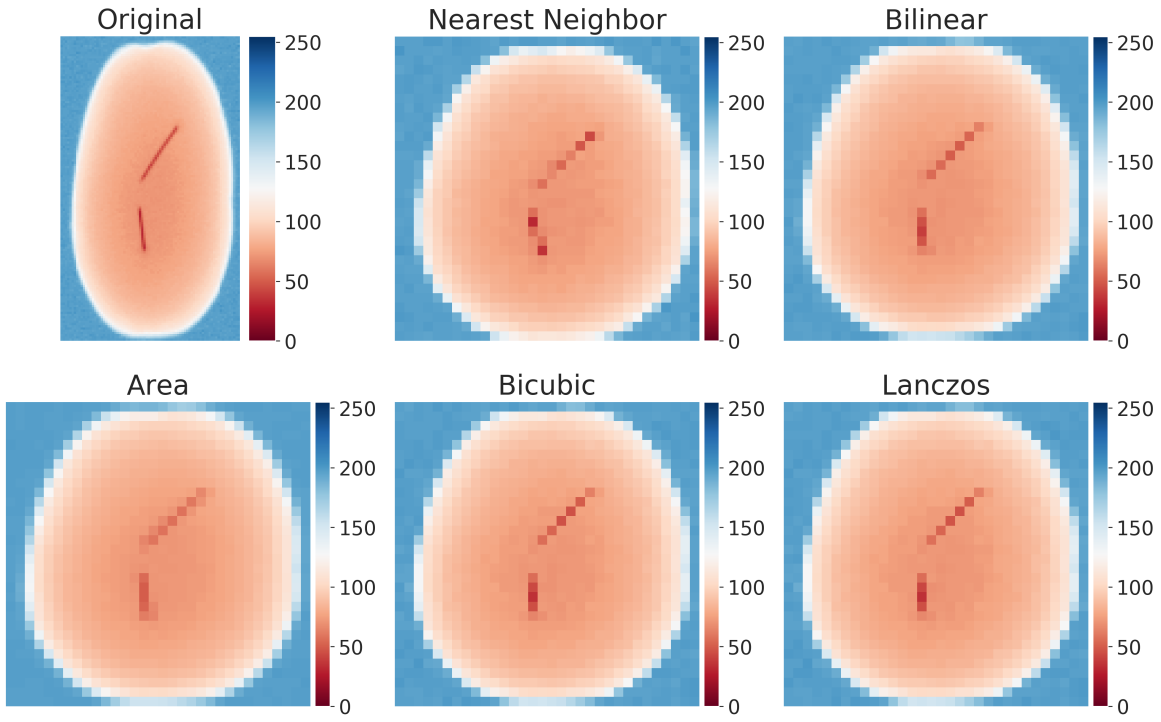


Figure 22: Application of different resizing approaches to an X-ray image. Image from [19].

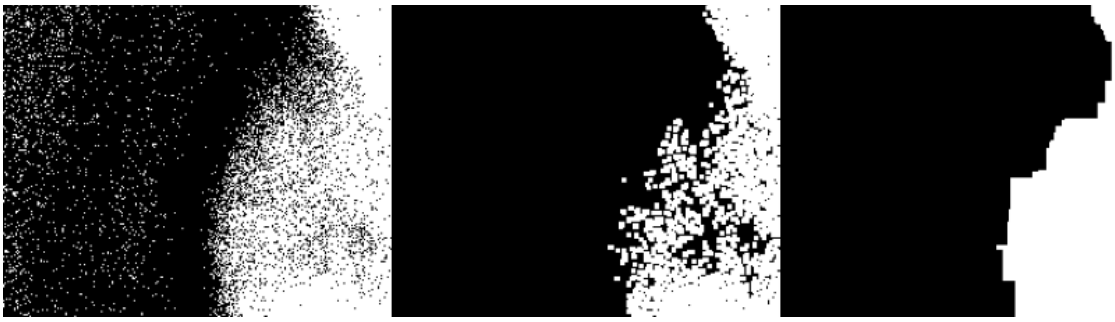


Figure 23: Morphology operations used to clean the mask after thresholding. From the left we have the raw mask, the mask ofter opening, and the opened mask after closing.

8.5 MORPHOLOGY

Morphology is the operation of manipulating shapes in an image. We apply the operation to binary images, where the pixel values are 0 or 1, such as binary masks. Like filters, they are applied to a window sliding across the image, known as a kernel, with different shapes depending on the desired output. We consider two morphology operations: *erosion* and *dilation*.

Erosion shrinks the shapes in the image. The resulting pixel will become 1 if *all* of the pixels in the kernel is 1, and 0 otherwise. Dilation grows the shapes in the image, the inverse operation of erosion; the resulting pixel will become 1, if *any* of the pixels in the kernel is 1, and 0 otherwise. We can use these two operations to build two further operations: *opening* (erosion followed by dilation) for removing noise outside the shape, and *closing* (dilation followed by erosion) for removing noise inside the shape. Figure 23 shows how we used morphology to ”clean” the mask after otsu’s threshold.

9

NEURAL NETWORK

ML is the act of learning a pattern through *training*, where we present the model with a dataset containing the pattern we want it to learn. Training is a minimization problem, where the target is to minimize what is known as the *loss function*. A loss function quantifies the problem at hand - how far from the target was the guess made by the model. We can compute a gradient for each output with this loss function, expressing how much each output contributed to the final output and how wrong it was. The optimizer uses this gradient for correcting the different factors in the model. This process is known as *back propagation*. In order to train, data is passed through the model multiple times, each time correcting the factors to lower the loss.

An ANN is one such ML model built from a collection of *perceptrons*. These perceptrons mimic the brain's neurons, producing an output based on its inputs. Equation (4) shows the equation for a perceptron, along with the visualization in Figure 24. Each input is multiplied by some weight, w_n, determining how much that particular input contributes to the final output. During training, these weights are modified based on the gradient of the loss function. Post multiplication, the product is summed and entered into what is known as an *activation function*. This function allows the perceptron to either be limited within a specific numeric range, have a smaller footprint on the output, or help it to be more easily differentiable, to name a few. Choosing which activation function to use is a matter of both experience and guessing.

$$y = f\left(b + \sum_{i=0}^{n} w_i x_i\right) \tag{4}$$

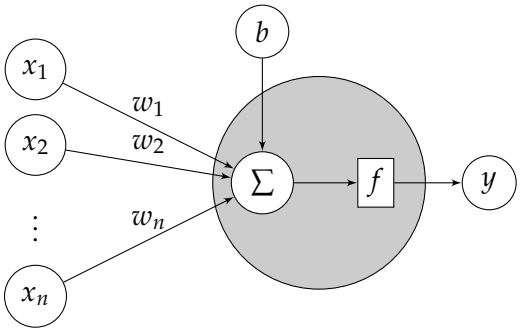


Figure 24: A single perceptron. The x values are the input, y is output, b is the bias term, and f is the activation function.

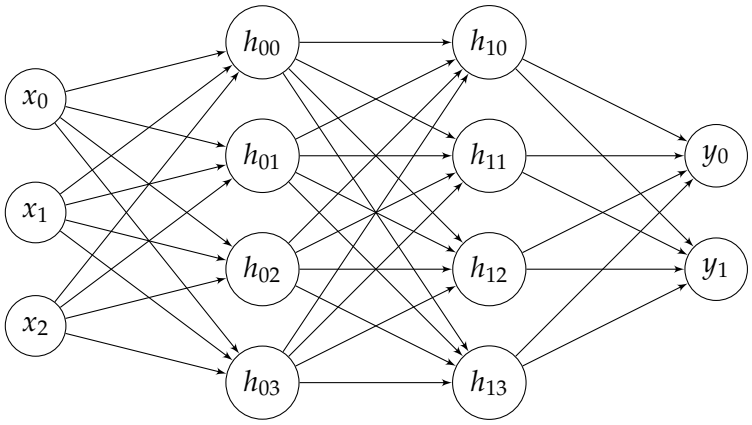


Figure 25: A fully connected deep neural network. x are the input nodes, h are the hidden nodes, and y are the output nodes.

To build an ANN model, we construct *layers* of parallel perceptrons, defined as a *network*. How each layer connects differs, but the most widespread strategy in modern applications is the *fully connected* network, where each perceptron connects to every perceptron in the following layer. The first and last layers are known as the *input* layer and *output* layer, respectively, with the intermediate layers known as the *hidden* layers. A *deep* neural network is a network consisting of multiple hidden layers. Figure 25 shows an example deep neural network with three inputs, two hidden layers that are four wide, and two outputs.

Each perceptron in the network contributes to the network's total *capacity* - how big of a pattern the network can capture. The more perceptrons we add, the more complex the pattern we can capture. Tuning the capacity of a network is non-trivial, as too much capacity can lead to *overfitting*, and too little capacity can lead to the network not capturing complex patterns. In the context of deep neural networks, we tune the capacity by the *depth* - the number of layers - and *width* - the number of perceptrons per layer.

9.1 CONVOLUTIONAL NEURAL NETWORK

Building on the foundation of ANNs, Convolutional Neural Networks (CNNs) are a variant of the ANN, but with a set of preceding convolutional layers. Convolution is the act of applying a filter to an image, much similar to regular filters as shown in Chapter 8. Figure 26 depicts an overview of the convolution operation and Equation (5) the operation as an equation. However, compared to the regular filters where the filters were pre-determined, the filters in the convolutional layer are not known beforehand but are also part of the training process. These trainable filters enable the CNNs to extract features from an image without having to introduce prior knowledge. We pass the extracted features to a regular ANN, which makes the final prediction.

$$\hat{I}(i,j) = \sum_{y=0}^{k} \sum_{x=0}^{k} K(x,y) * I\left(i - \left(x - \left(\left\lfloor \frac{k}{2} \right\rfloor\right)\right), j - \left(y - \left(\left\lfloor \frac{k}{2} \right\rfloor\right)\right)\right) \tag{5}$$

Alongside the convolution layers, the CNN usually also feature pooling layers. These layers down-sample the extracted features into a more compact representation to reduce dimensionality. While removing information reduces the information given to the ANN, pooling increases runtime perfor-

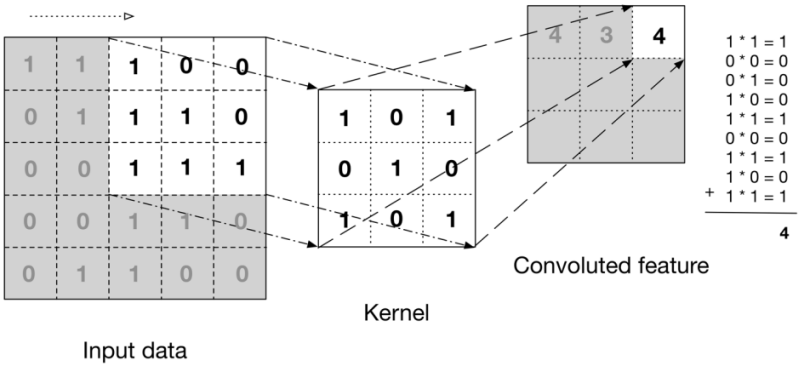


Figure 26: Overview of how the convolution operation is applied. Image from [21].

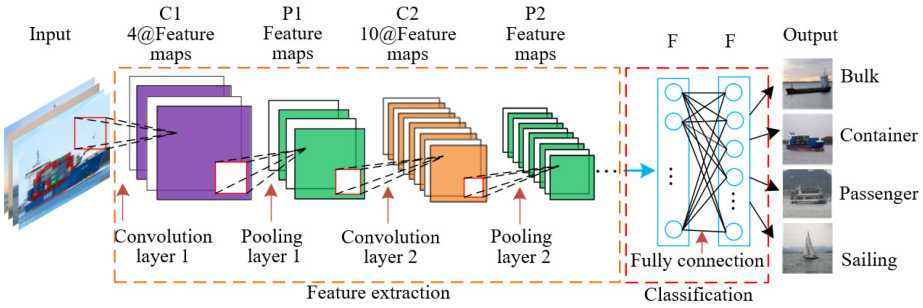


Figure 27: Full overview of a CNN. Image from [22].

mance, as the network has to process a reduced amount of data. It can also remove spatial information about the features, which the network might learn otherwise. So depending on what the target is, we should use different pooling techniques. Figure 27 shows the overview of how all of the layers of the CNN connects.

The extra expressive power we gain from the CNN does not come for free; they are also a lot more computationally complex. Whereas an ANN might have hundreds of thousands of trainable parameters, CNNs are in the millions.

9.1.1 *Online Inspection of X-ray Images*

The master's thesis made by Thorbjørn Louring Koch [23] titled "Online Inspection of X-ray Images - Detecting hollow hearts and needles in potatoes" laid the foundation for the AXIS project and was handed in prior to this Ph.D. thesis. He studied using X-ray imaging for food inspection and helped make the prototype machine, *ButeoX*. He showed that he could build an algorithm using standard thresholding techniques, which worked very well at the cost of manual tuning. He was able to beat the thresholding technique with a CNN, as Table 5 shows. He showed that using a neural network with dropout in the convolutional layers and the fully connected layers allowed the network to generalize well without having a sizeable data set. Table 6 shows the different model architectures and their performance. We see that larger networks with additional regularization provide improved results.

Method	Perfect	Needles	Hollow	Total error
Thresholding	2/50	0/49	0/36	1.48%
Radial KMeans	1/50	11/49	3/36	11.10%
potaNet	2.03%	0.07%	0.13%	0.74% (1.20%)
potaDict	1/10	1/10	0/7	3.70%

Table 5: Best results from the potaNet thesis [23].

Image size	Layer depths	Layer widths	Weights	Regularization	Error	Correct
64 × 64	[32,64,32,64,32,16]	[15,3,3,3,3]	113744	Dropout	5.32%	61%
64 × 64	[128,64,64,64,32,128]	[15,3,3,3,3]	457216	Dropout	2.50%	85%
64 × 64	[128,64,64,64,32,128]	[15,5,5,5,5]	905536	Dropout	2.18%	91%
64 × 64	[128,64,64,64,32,128]	[15,3,3,3,3]	457216	Dropout, L^2 norm	2.67%	93%
128 × 128	[64,64,64,64,64,64,64,64]	[15,3,3,3,3,3,3]	534784	Dropout, L^2 norm	2.25%	93%
128 × 128	[128,64,64,64,64,64,32,128]	[15,3,3,3,3,3,3]	567808	Dropout, L^2 norm	1.40%	94%
128 × 128	[128,64,64,64,64,64,32,128]	[15,5,5,5,5,5,5]	1059328	Dropout, L^2 norm	1.20%	95%

Table 6: Specification and results of potaNet. Table from [23].

9.1.2 *Automatic classification*

The master's thesis by Aleksandar Topic titled "Automating classification in food inspection" [14] was supervised as part of this Ph.D. thesis and is covered in Appendix B.1.4. This thesis also provided a CNN implementation for detecting potatoes that carry the hollow heart disease. As this disease is rare, especially in Denmark, the potatoes had the hollow heart artificially added as data acquisition would be too costly otherwise.

Looking at Table 7 we see similar performance to the model proposed in Section 9.1.1 with a shallower network. The performance is due to data augmentation, which increases the dataset based on the already available dataset.

9.1.3 *Foreign object detection*

The master's thesis by Jesper Pedersen titled "Foreign object detection in x-ray images using machine learning" [24] was supervised as part of this Ph.D. thesis and is covered in Appendix B.1.8. The thesis tackled the problem of detecting foreign objects in X-ray images of meat. FOSS, a Danish company that produces machines for food inspection, using X-ray imaging amongst other techniques, collaborated in this project. The thesis explored using ML in the entire pipeline of a meat inspection machine from acquisition to the decision.

Input	Batch size	Filter sizes	Layer depths	Param	Aug	Acc
64x64	64	[15,3,3,3,3,3,0,0]	[72,32,64,64,32,64,128,64]	207570	800	87%
64x64	64	[15,5,3,3,3,3,0,0]	[72,32,64,64,32,64,128,64]	211666	1000	93%
64x64	100	[15,5,3,3,0,0]	[128,64,32,32,64,128,64]	689922	1200	94%

Table 7: Overview of the best performing CNNs from Topic's thesis.

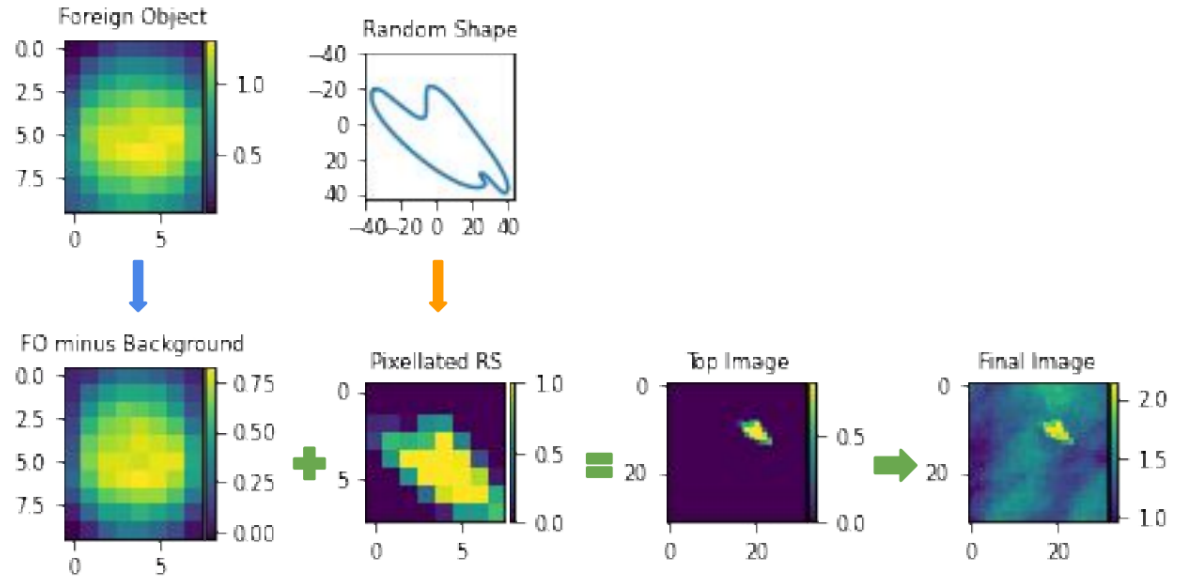


Figure 28: Overview of the process of adding artificial foreign objects. Image from [24].

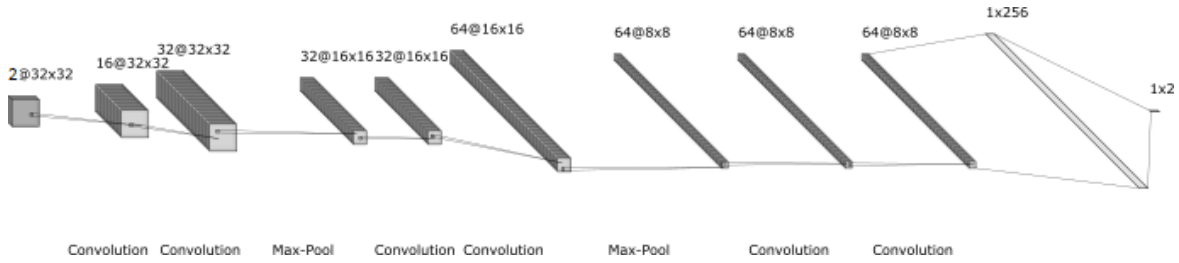


Figure 29: Architecture of the resulting CNN for detecting foreign objects in meat. Image from [24].

Rather than feeding the network the entire image, the thesis suggested segmenting the image into smaller windows, which we then feed to the network. These windows were also displaced with an overlap of 50%, ensuring that small objects would not be ”cut in half.”

As the dataset was small, the thesis investigated generating artificial foreign objects randomly. The generation process was first to generate the shape using random bezier curves. The process then discretizes the random shape into a pixelated form, matching the size and intensity found from actual foreign objects. Finally, the objects were added randomly to windows not already containing a foreign object with a 20% probability. Figure 28 shows the steps of generating and adding the artificial foreign objects.

Figure 29 shows the final CNN architecture reaching an accuracy score of 98.74%. The thesis argued that this accuracy is hard to compare without a baseline model but suggested it is insufficient for real-world usage. This performance is on a per-window basis, while the real target is the entire image. Instead, the thesis should have used the windows as an ensemble, given that each quarter of a window is covered four times. If a quarter receives three or more votes, it should consider that the window contains a foreign object, further considering the entire image as contaminated.

The thesis further investigated the model’s shortcomings and came to interesting conclusions. The first finding was that the resulting network was not resilient to anything not overly distinctive in the X-ray images, such as plastic. Secondly, the network’s generalization ability depends heavily on the data seen during training, resulting in it not handling unseen cases very well.

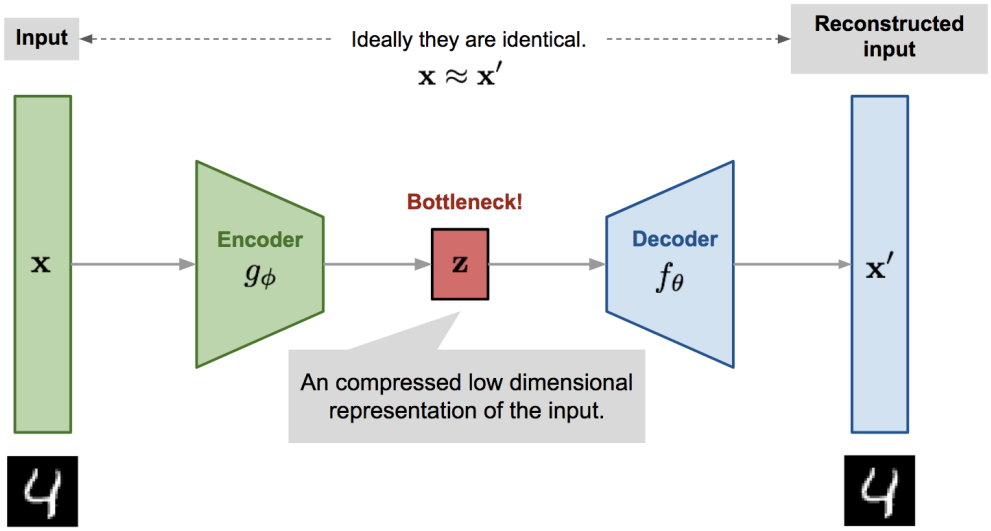


Figure 30: Coarse architecture of the CAE model. Image from [25].

9.2 AUTO ENCODERS

Another variant of the ANN is the Convolutional Auto Encoder (CAE): a neural network trained in an unsupervised way. The top-level architecture has two parts: *encoder*, g, and *decoder*, f. The encoder reduces its *input*, x, to a lower dimensionality, known as the *latent space*, z, and consists of convolution layers. The decoder does the exact opposite, as it takes a representation from the latent space and expands it back into the original dimensionality as *output*, x'. The decoder consists of *de*-convolution layers, computing the inverse of the convolution layers.

We can draw parallels between the CAE and other fields. For example, we can see the encoder and decoder parts as compression and decompression, with the latent space being the compressed representation. Another example is that the decoder can be seen as a Principle Component Analysis (PCA) as it extracts and encodes the features with the most variance into the latent space.

Figure 30 shows the general architecture of a CAE. The idea is that the network can encode an input into a representation in the latent space and decode it back into its original form. During training, the network optimizes towards minimizing *reconstruction error*; a metric for quantifying the difference between the output image and the input image, $x \approx x'$. The most basic metric being the Mean Squared Error (MSE) as Equation (6) shows.

$$\mathcal{L}_{CAE} = MSE(x, x') = MSE(x, f(g(x))) = \frac{1}{N}\sum_{i=0}^{N}\left(x_i - f\left(g\left(x_i\right)\right)\right)^2 \tag{6}$$

The CAE is not designed for exact image reconstruction but rather for image approximations [26]. It is not proficient as a generative model as it cannot interpolate between training points. The CVAE counters this fact by exchanging the encoder, trained towards the training set, with a probability distribution that generates the input data. Figure 31 shows the general architecture of a CVAE. With the introduction of this probability distribution, the CVAE is a stochastic generalization of the CAE that models the latent space probabilistically, allowing it to generate samples not seen during training but carrying a close resemblance. The probabilities also have to be introduced in the loss function as Equation (7) shows, where we introduce Gaussian densities alongside the MSE.

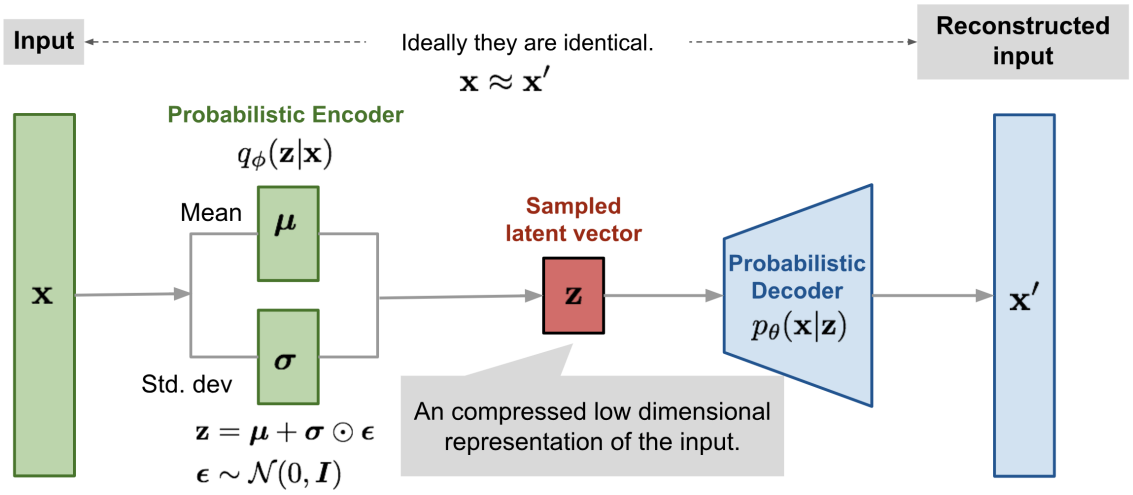


Figure 31: Coarse architecture of the Convolutional Variational Auto Encoder (CVAE) model. Image from [25].

$$\mathcal{L}_{CVAE} = \frac{1}{N}\sum_{i=0}^{N}\left(x - x'\right)^2 - \frac{1}{2}\sum_{j=0}^{J}\left(1 + \log_e\left(\sigma_{ij}\right)^2 - \mu_{ij}^2 - \exp\left(\log_e\left(\sigma_{ij}\right)^2\right)\right) \tag{7}$$

9.2.1 *Automatic Detection of Foreign Objects*

The master's thesis by Alina Hjorth Sode titled "Automatic Detection of Foreign Objects in X-Ray Images" [19] was supervised as part of this Ph.D. thesis and is covered in Appendix B.1.10. Like Section 9.1.3, the thesis by Alina focused on foreign object detection in X-ray images. However, rather than following a traditional supervised approach of having samples of every known class, the thesis investigated the semi-supervised approach of training purely with "good" samples. The intuition is that the samples generally follow the same structure in food production; there are many "good" samples and few outliers. Furthermore, having an automated outlier detector would allow us to detect defects we have never seen before, countering the generalization problems found in Section 9.1.3.

The thesis used autoencoders for generating a perfect image from an input sample. The idea was that the network could produce an image of what the sample should look like without any defects. This generated sample could then be compared to the input sample to determine whether the input sample diverges too much from the perfect case, which we define as an outlier. The thesis proposed multiple image similarity measures, each carrying different properties, but with Structural Similarity Index Measure (SSIM) proving the overall best results. The resulting network, whose architecture Table 8 depicts, could indeed produce good images from input samples as shown in Figure 32.

For the images captured using the area scanner, the proposed model along with SSIM combination provided Area Under Curve (AUC) scores of 0.997 and 0.951 for the chocolate and potato images, respectively. However, for the images of potatoes using a line scanner, the thesis failed to accurately quantify how well the images matched up, which we could do through manual inspection. Regardless, the line scanner potato images provided an AUC score of 0.923. Overall, the thesis indicates that it is possible to utilize autoencoders as a semi-supervised approach to the problem of outlier detection.

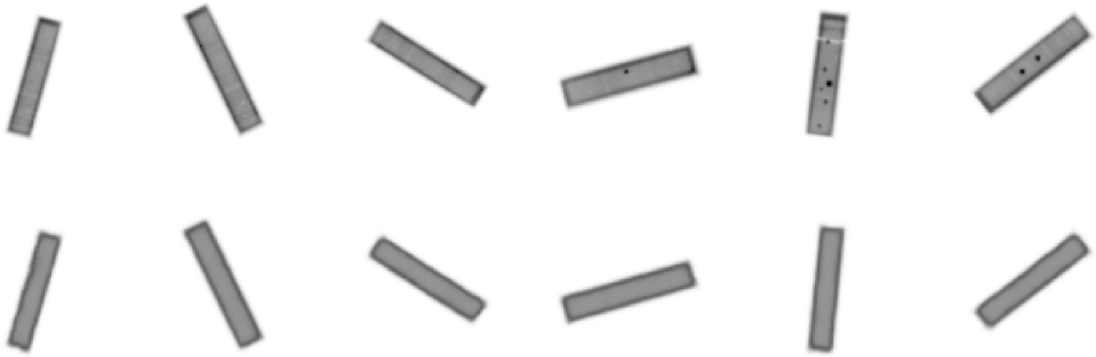


Figure 32: Chocolate after being reconstructed by a CVAE. The top row shows the raw images. The bottom row shows the reconstructed images. Notice how the foreign objects and other flaws are removed in the reconstructions. Image from [19].

Layer	Params	Kernel	Padding	Stride	Output shape
Input					(128,128,1)
Conv2D LeakyReLU	filters = 32 $\alpha = 0.1$	$[2 \times 2]$	'same'	2	(64,64,32)
Conv2D LeakyReLU	filters = 32 $\alpha = 0.1$	$[2 \times 2]$	'same'	2	(32,32,32)
Conv2D LeakyReLU	filters = 32 $\alpha = 0.1$	$[2 \times 2]$	'same'	2	(16,16,32)
Conv2D LeakyReLU	filters = 32 $\alpha = 0.1$	$[2 \times 2]$	'same'	2	(8,8,32)
Conv2D LeakyReLU	filters = 32 $\alpha = 0.1$	$[2 \times 2]$	'same'	2	(4,4,32)
Flatten					(512)
Dense LeakyReLU	size = 200 $\alpha = 0.1$				(200)
$2 \times$ Dense Linear	size = J^*				$2 \times (J^*)$

Layer	Params	Kernel	Padding	Stride	Output shape
Input					(J^*)
Dense Tanh	size = 512				(512)
Reshape					(4,4,32)
TransConv2D ReLU	filters = 32	$[2 \times 2]$	'same'	2	(8,8,32)
TransConv2D ReLU	filters = 32	$[2 \times 2]$	'same'	2	(16,16,32)
TransConv2D ReLU	filters = 32	$[2 \times 2]$	'same'	2	(32,32,32)
TransConv2D ReLU	filters = 32	$[2 \times 2]$	'same'	2	(64,64,32)
TransConv2D ReLU	filters = 32	$[2 \times 2]$	'same'	2	(128,128,32)
TransConv2D Sigmoid	filters = 1	$[2 \times 2]$	'same'	2	(128,128,1)

Table 8: Proposed CVAE architecture. Table from [19].

9.3 DECISION TREES

In addition to the ANN, the Decision Tree (DT) ML model has seen a rise ever since introducing the Gradient Boosted Decision Tree (GBDT) in projects such as LightGBM [27] and XGBoost [28]. A DT is a tree-structure where each node constitutes a *decision* to be made on a feature based on a *weight*. We make multiple decisions as we traverse the tree, with the leaf nodes constituting the final decision. Gradient boosting is the process of modifying the weights of the decision nodes through differentiation and backpropagation, just like we train the ANN. As tree traversal is linear, an ensemble of trees, or *forest*, is utilized to capture non-linearity. Compared to the ANN, GBDTs are fast and lightweight, with their model not requiring much memory or storage space.

9.3.1 *Probability of Distress*

The bachelor's thesis by Ulv Gejr Gudmann Foerlev titled "Probability of Distress" [29] was supervised as part of this Ph.D. thesis and is covered in Appendix B.2.1. This thesis reproduced the work of a paper [30], which investigates models for predicting the financial distress of Danish companies based on their financial reports. They suggest that a GBDT model could outperform similar classical statistical methods.

The thesis did not have the same data available, resulting in much time spent on data acquisition from public records and cleaning the data. In the end, the thesis proposed a GBDT model using XGBoost, which confirmed the findings of the original paper; that a GBDT model leads to a more accurate probability compared to the other models. The thesis was able to hit an AUC score of 0.801 compared to 0.822 as presented by the original paper.

10

EVALUATION

With a trained model, we have to assess the network's performance. This chapter will overview the different metrics and their ability to describe the model. For all metrics, we evaluate how the model f applied to the dataset x relates to the label y, with both sets of size N. For binary classification, where the model predicts either 0 or 1, we have the four following sets:

$$\text{True positive}(tp) = |\{x \mid f(x) = 1 \wedge f(x) = y\}| \tag{8}$$
$$\text{True negative}(tn) = |\{x \mid f(x) = 0 \wedge f(x) = y\}| \tag{9}$$
$$\text{False positive}(fp) = |\{x \mid f(x) = 1 \wedge f(x) \neq y\}| \tag{10}$$
$$\text{False negative}(fn) = |\{x \mid f(x) = 0 \wedge f(x) \neq y\}| \tag{11}$$

10.1 ACCURACY

For classification problems, the most popular metric is *accuracy*, which Equation (12) shows.

$$Accuracy = \frac{tp + tn}{N} \tag{12}$$

This metric returns a number between 0-1, constituting how many predictions were correct, with scorings closer to 1 being better than others. Inversely, we can derive the error rate from the accuracy with $1 - accuracy$. It can provide an incorrect view of the model's performance, as it does not depict how far off the model was. This incorrectness is especially prominent for unbalanced datasets, where one class dominates the distribution. For example, if one class makes up 80% of the dataset, a model guessing that class every time would get an accuracy of 0.80, even though it has no predictive power.

10.2 PRECISION AND RECALL

Precision and *Recall* are two metrics for dealing with imbalanced datasets. Precision describes how accurate the results returned are, while Recall describes whether the model returns a majority of the positive results. Equation (13) and Equation (14) shows how to compute Precision and Recall.

$$Precision = \frac{tp}{tp + fp} \tag{13}$$

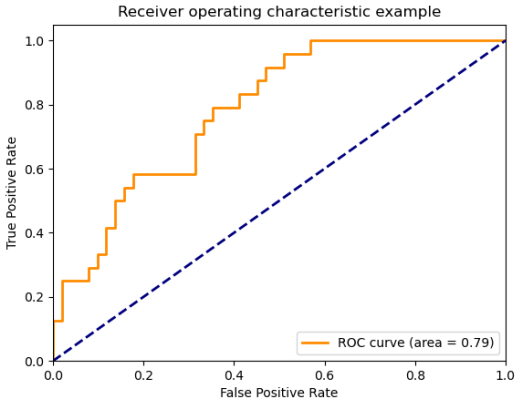


Figure 33: Example ROC curve. Image from [31].

$$Recall = \frac{tp}{tp + fn} \tag{14}$$

We can combine the two metrics into one metric: the F_1 score. It is the harmonic mean of Precision and Recall as Equation (15) shows.

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \tag{15}$$

All of the metrics in this section are between 0 and 1, with good classifiers being closer to 1.

10.3 RECEIVER OPERATOR CHARACTERISTICS AND AREA UNDER CURVE

The Receiver Operator Characteristics (ROC) curve is a plot that describes the model's performance when we vary the threshold. Note that a model rarely produces binary output, as stated at the beginning of this chapter, but instead produces a probability of class 1. The most widespread method is to set the threshold at 0.50, essentially rounding the probability to whatever class is closest. Rounding might not always be desirable, as some applications have more or less tolerance towards the amount of misclassified samples. We can control the True Positive Rate (TPR) and False Positive Rate (FPR) by varying the threshold, which is what the ROC curve depicts. Figure 33 shows a ROC curve. We want the curve close to the upper left corner, which describes a good classifier. The blue diagonal line shows a random classifier with no predictive power.

We can quantify this curve into a single number using the AUC score. For a perfect classifier, the AUC score will become 1.

10.4 CONFUSION MATRIX

A confusion matrix can give insight into the distribution of true or false positives and negatives for binary and multi-class classifications. Table 9 shows a confusion matrix for a binary classifier. For

tp	fn
fp	tn

Table 9: Overview of a confusion matrix.

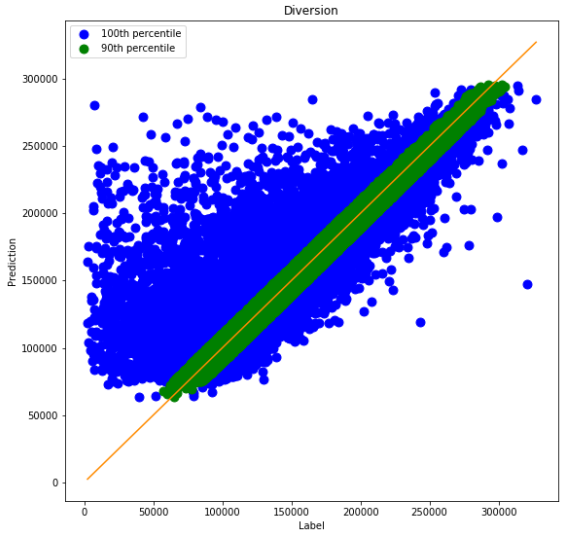


Figure 34: Diversion plot of a regressor. The plot is from previous work. The scattered dots are the predictions, with the green depicting the 90th percentile.

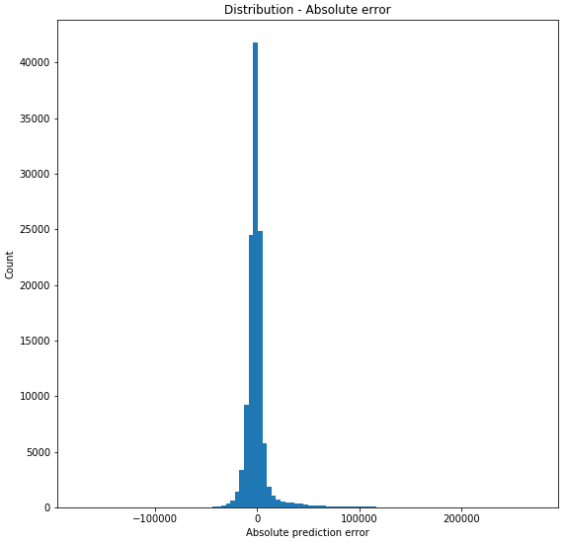


Figure 35: Distribution plot of a regressor. The plot is from previous work.

a perfect classifier, we should see a high concentration along the diagonal of the matrix. We would have a row and column for each possible class for multi-class classifiers, giving us insight into which classes correlate through typical misclassification. For example, cell (i, j) would show the amount of actual class i entries predicted as class j. Intuitively, cell $i = j$ depicts the same class prediction.

10.5 CONFIDENCE

As a final classification metric, we have the *confidence* plot. This plot shows whether the model's confidence correctly corresponds to a correct result. For a good classifier, we want to see correct predictions distributed close to either 0 or 1 and incorrect predictions distributed around 0.5.

Figure 36 shows a confidence plot from the project described in Section 9.3.1. It shows a good case; it has high accuracy whenever the model is confident (predicts close to 0 or 1). Whenever the model is not confident (predicts close to 0.5), it has lower accuracy.

10.6 DIVERSION AND DISTRIBUTION

For regression problems, there exist other qualitative measures. While not covered thoroughly through this thesis, we include them for completeness. Similar to the confusion matrix, the *diversion* plot shows how far off a prediction was compared to the corresponding label. On the x-axis, we have the

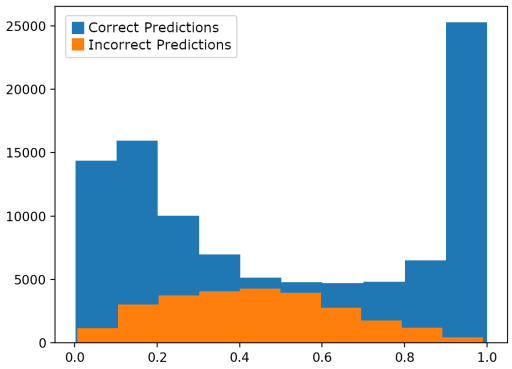


Figure 36: Confidence plot of a classifier. Image from [29].

labels, with the predictions on the y-axis. Figure 34 shows a diversion plot. We should see a high concentration along the diagonal for a perfect regressor.

Similarly, we have the overall *distribution* plot, which shows how far the predictions diverge. For a good classifier, we want a narrow tall distribution around 0, indicating low divergence. Figure 35 shows a distribution plot.

11

DETECTING FLOATERS IN BEER

As an example, this chapter will present a ML project we did for a danish brewery, showing the entire process of utilizing ML from beginning to end.

The brewery observed that proteins in beer could fold together into large structures, known as *floaters*. These floaters could, in some cases, become large enough that they were detectable by the naked eye. While the floaters do not present a health hazard, they can seem unappetizing, resulting in the possibility of a lost sale.

The primary goal of this project was to quantify the occurrence of the floaters so that the brewery could later match this quantification to additional meta-data collected in parallel with the sampling to determine when and why the floaters occur. Their first implementation was to have an employee manually inspect the beer at different times, classifying them according to an internally developed scoring. The inspection method was that the expert held a beer against light and scored the visuals based on the floater count and size. Our work was to automate this method using computer vision and ML, which a prior master's thesis already worked on [23]. Figure 37 shows the machine built to capture samples as videos and score them automatically. The task was to score floaters' sizes and count based on the video.

We can apply the preprocessing methods presented in Chapter 8 to prepare the data. The brewery captured the following data:

- 31 different beers
- Each beer was sampled at 4 different days (0,3,7,10)
- For each day, 8 videos of each beer was recorded
- Each video consists of 125 frames
- Each frame is 4012×2048 in grayscale, or $\sim$8 MB.
- In total: 992 videos or $\sim$968 GB

Figure 38 shows one of the captured images, where we first observe that the floaters are very hard to see in the image. We start by applying smoothing as Section 8.1 covers, specifically *gaussion smoothing* since the floaters are small enough to be considered noise, and we want them to contribute to the final image. While an algorithm could learn the pattern from these images, there is a risk that it might learn the pattern of the bottle, which is not the goal. The previous work [23] suggested removing the background, which we will do. Given that we have a video and that we do not care about the static elements of the video, we can *remove the background* by removing the mean image

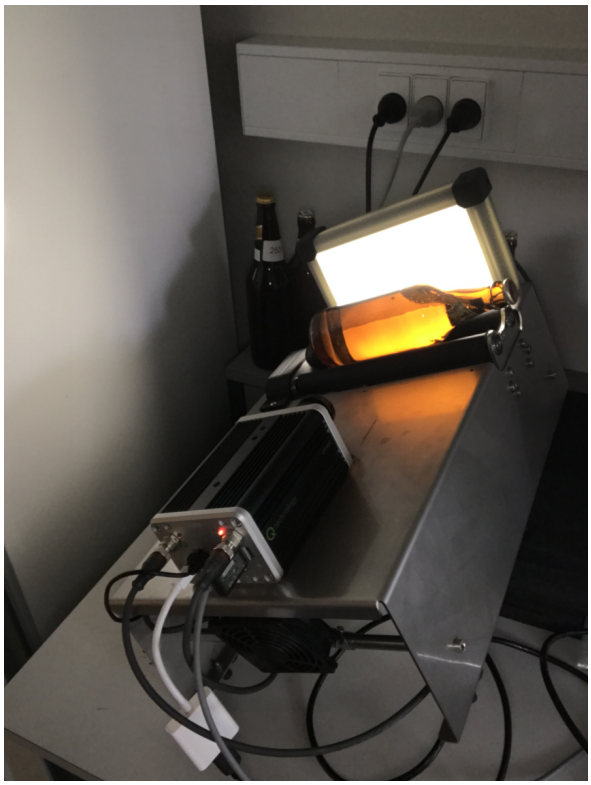


Figure 37: Image of the machine setup for capturing the beer videos. Image from [23].

of the video - the static background. We want to enhance the particles, which we do through *otsu's thresholding*, described in Section 8.2, followed by *dilation*, described in Section 8.5. After these steps, we get the image we see in Figure 39; a clean image containing only the floaters.

From the cleaned frames, we want to extract the floaters. We can apply traditional image processing at this step, rather than ML, much like the ROI algorithm from Section 8.2. However, instead of extracting the floaters into images, we compute their size based on the number of conjoined pixels in the thresholded image. This extraction gives us a list of floater sizes. ML algorithms favor data with fixed input size, so we compute a histogram from the variable list of floater sizes, giving us a fixed-size representation of each image. Figure 40 shows two histograms based on the same sample, at varying days. As we can see, the day 10 sample has a lot more floaters, *the magnitude of the orange histogram is higher than the blue*, and the floaters are bigger than the day 0 sample, *the orange histogram is wider than the blue*. By computing the histograms from the videos, we have reduced the data size from ∼968 GB to ∼50 MB, significantly reducing the training time.

The internal scoring we are matching is six values between 0-5. However, after talking to the brewery, we found that these scores do not have to be met *exactly*, and do not constitute the entire range. If a sample ever appeared, which was a worse case than they had ever seen, they would like the scores to go *beyond* their current range. As such, this is a *regression* problem, not a classification problem, requiring us to predict a continuous scoring. Setting up an ANN to do regression instead of classification is straightforward; we need to change the loss function. The most popular loss function for regression problems is Root Mean Square Error (RMSE) as shown in Equation (16), or Mean Absolute Error (MAE) as shown in Equation (17). We use RMSE as our loss function, but evaluate MAE for our own convincing.

Figure 38: Raw image from a beer video sample.

Figure 39: Image from a beer video sample with enhanced floaters.

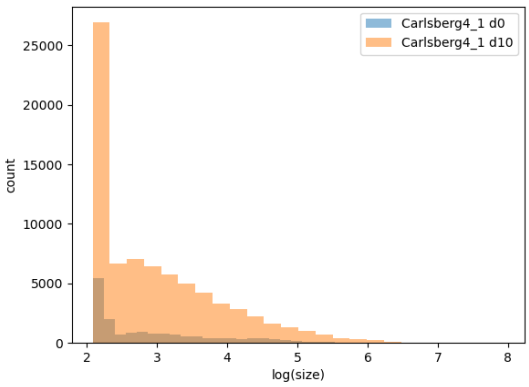


Figure 40: Histograms of floater sizes, computed from an image of beer. Note the log scale on the x-axis.

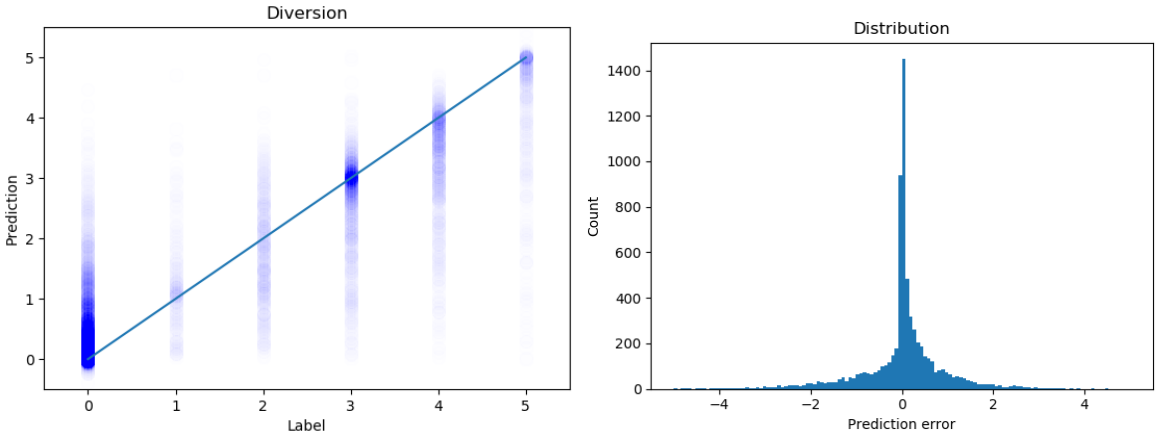


Figure 41: Diversion and Distribution plots of the trained brewery model.

$$RMSE(x, x') = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x - x')^2} \tag{16}$$

$$MAE(x, x') = \frac{1}{N}\sum_{i=1}^{N}|x - x'| \tag{17}$$

Figure 41 shows the performance of the resulting model. We see a narrow tall distribution around 0 on the distribution plot and a concentration around the diagonal blue line in the diversion plot. The MAE is at 0.5, which indicates that we are on average off by 0.5.

There are multiple flaws in the project, which is why we are not getting better results, even though this seems like a straightforward ML problem. First of all, labeling cannot be trusted completely, as only a single person labeling and only labeling once. It is not that the person did a poor job, but instead that humans are incapable of producing consistent labeling - especially over time - as multiple projects have shown [32]. Secondly, the different samples had different bottles of differing colors. The coloring is not a problem for our model, as the captures are in grayscale, and we remove the bottle, but it is a problem for humans. Different colored light is perceived differently for humans, whereas brown bottles can seem murky, translating into a worse scoring for the said bottle. Finally, while we did have many images, they originated from the same sample, in reality giving us 31 different samples. ML favors large datasets in order to capture a pattern adequately, which we did not have at the time.

We presented these results and related problems to the brewery, who agreed and did find the model usable. In the future, they would use the preprocessing steps to produce a new scoring of the beers, as they agreed that the processed video proved a cleaner representation. Furthermore, they found the histogram representation especially interesting, as it already provided them with their desired target; a human-readable quantification of the beers.

12

SUBCONCLUSION

In this part, we have covered the different parts of applied ML. We started by covering different preprocessing techniques in Chapter 8. Then we covered building and tuning ML models in Chapter 9 with a focus on the ANN variants. Then we covered different evaluation methods for ML models in Chapter 10. Finally, in Chapter 11 we showed a project we did for a Danish brewery.

We see that the methods positively impact the images for all preprocessing methods, at least for human interpretation. Section 8.1 shows methods for removing noise from an image, with the median filter being the strongest candidate for X-ray images. Section 8.2 shows how we extract the objects from the image, giving us a localized view of the subject. Section 8.3 shows how we can improve the in-object contrast, resulting in a more defined structure of the object's internals. Section 8.5 shows how we can manipulate binary masks to fit the objects better. We will be considering all of these methods in our later application.

The most promising ANN models for classification of foods in X-ray imaging have been the CNN as Section 9.1 describes. We have seen three applications in this setting in Section 9.1.1, Section 9.1.3 and Section 9.1.2. All of the applications show promising results, each with its shortcomings. The most prominent shortcoming is the lack of enough data and proper class representation. Section 9.2.1 proposes a solution to these shortcomings by utilizing a CVAE as a semi-supervised approach to outlier detection. With this method, we do not need a large dataset, and with an even stronger outcome: we do not need to have a dataset that captures every possible class. We can instead train purely on good data, to which there are plenty in the world of food inspection. The only shortcoming is that the model seemed insufficient on images from a line scanner, which the new AXIS machine utilizes. This problem is not the master's thesis working on the problem's fault, as this new dataset came late in the process. We expect to reach similar results compared to the area scanner images with further tuning.

Finally, the metrics presented in Chapter 10 will provide us with the tools necessary to evaluate our ML model. Specifically, the metrics presented in Section 10.3 and Section 10.5 are very descriptive of the model's performance and which we will use extensively.

Part IV

FIELD-PROGRAMMABLE GATE ARRAY

13

MACHINE ARCHITECTURE

This chapter will introduce the reconfigurable hardware chip, a Field-Programmable Gate Array (FPGA), as this chip is the mean to our end for a high-throughput pipeline. We start by giving a quick refresh of machine architecture in Section 13.1, focusing on Central Processing Units (CPUs) and Graphics Processing Units (GPUs) as their drawbacks motivate FPGAs, which are covered in Section 13.5. Then we will cover how to program FPGAs in Sections 13.5.1 and 13.5.2, followed by two approaches for FPGA development in Sections 14.1 and 14.2, in which this thesis spent a majority of the time improving. Finally, we will cover some projects spawned during this thesis in Section 14.4, attacking different research questions targeting FPGAs.

13.1 MACHINE ARCHITECTURE

This section will provide a quick introduction to the inner workings of a computer. The most widespread machine abstraction is the *von Neumann architecture* [33]. This architecture describes a machine, which is made up of several components:

- **A processing unit** that contains circuitry for arithmetic operations and registers for these.
- **A control unit** that contains logic and registers for manipulating the program counter.
- **Memory** containing both data and instructions.
- **External storage** holding data and instructions on mass storage.
- **Input/output** for communicating with other components/machines.

Most computing devices follow this architecture by having a CPU containing the processing unit, the control unit, and the first memory level. This containment packaging allows for shipping chips that do not require the same degree of interoperability as they carry the different units internally. Modern machines feature specialized addition boards known as *accelerators*, highly optimized for solving specific tasks. These devices communicate with the CPU through input/output channels, offloading tasks to these devices, trading generality for specificity in order to gain performance. The most common accelerator found in any modern machine is the GPU.

13.2 CENTRAL PROCESSING UNIT (CPU)

A CPU focuses on being able to do every possible task. It does so in a sequential manner by executing one instruction at a time. While there exist many different approaches to CPU design, we can describe the the more general CPU as five primary stages:

$$\textit{Instruction Fetch} > \textit{Instruction Decode} > \textit{Execute} > \textit{Memory} > \textit{Write Back}$$

Textbooks usually describe this coarse-grained architecture, as it is a minimal example great for understanding the concepts. However, actual implementations further divide these stages into sub-stages in the real world. Each stage is executed in a parallel pipelined fashion, feeding its results to the next stage. For example, while *Instruction Decode* is decoding the instruction, *Instruction Fetch* can fetch the next instruction. This perfect pipelining can only occur as long as there is no inter instruction dependency, in which case we have to stall the pipeline to ensure correct execution. Stalling can be circumvented by introducing hazard detection and forwarding at the cost of more logic.

One hard dependency to handle is that of *branching*. A branch instruction is usually not executed until after the *Execute* stage, which translates to the *Instruction Fetch* stage not knowing which instruction to fetch until three cycles after fetching the branch instruction. To reduce stalling, where the CPU idles until some action occurs, and flushing the pipeline, where we lose intermediate work, modern CPUs attempt to guess what the next instruction will be. Because of the importance of guessing correctly, the amount of dedicated logic becomes very large, as Figure 42 shows. Compared to the Arithmetic Logic Unit (ALU) and Floating-Point sections, the chip dedicates a large portion of floor space to branch prediction, Scheduler, Load/Store, and Decode. More chip roughly translates into higher power consumption or lower achievable clock rate.

The description above fits a single execution core, which we define as a core. Modern CPUs have multiple cores, defined as *multi-core CPUs*. Essentially, they are copies of a single core, each executing independently. They communicate through the various levels of memory. This independent execution can become a problem if multiple cores are working on the same address space, as one or more cores have to stall to ensure correct execution.

While the CPU has had a significant focus on executing sequential tasks well, a large amount of work has also optimized them towards running multiple programs concurrently. Concurrent execution benefits resource-limited machines, such as a single-core CPU or high-latency memory, as it interleaves the instructions depending on the scheduling strategy, rather than only running one program at a time or having the CPU idle while it is waiting for some resource. This execution model dramatically increases the versatility of the CPU as a general-purpose machine.

13.3 GRAPHICS PROCESSING UNIT (GPU)

In computer graphics, most of the operations are independent of each other, such as a translation operation applied to triangles in the three-dimensional space, where we can independently apply the application operation to the vectors. GPUs speed up this process by focusing on parallel execution. They consist of small execution cores in a grid, focusing on vector operations and raw computing

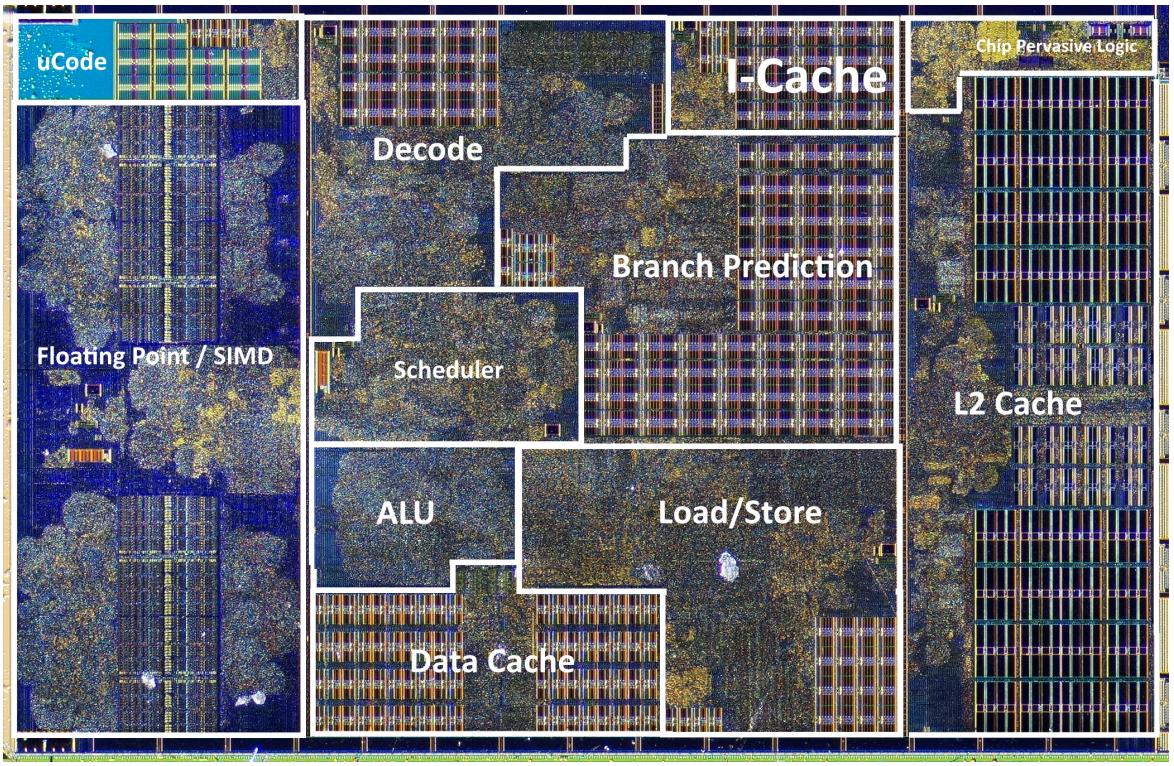


Figure 42: Die shot of a CPU, with labels of the different coarse-grained blocks. Image from [34].

power, as graphic operations rarely feature branches. In order to match the increase in computing power, GPUs also feature wide memory buses, allowing for more data to be fed into the device faster.

As graphics are assumed to be independent operations, multiple cores share the same memory patch, which is very small and fast. These cores share the same instruction memory, further reducing each core's required amount of memory. The GPU dedicates a lot more area to computing resources compared to the CPU as Figure 43 shows. This increased area gives the GPUs increased computing performance and reduced power consumption.

As the GPU carries a lot of raw computing power, tendencies have pushed the retuning of the GPU towards general computing rather than pure graphical operations. They are still a grid of vector processors at heart but can also execute more general programs. This generality does not mean that a GPU will replace a CPU, as the CPU is still the best at sequential execution, concurrent execution, and multiple unrelated independent sequential executions running in parallel. This gap is especially apparent when a GPU program contains branching, which the GPU is exceptionally bad at handling. For an application to take full advantage of the GPU, the problem must be parallel and distributable amongst thousands of cores, all executing the same program. As a final note on GPUs, memory accesses must be sequential (or coalesced) as the memory bus is not well suited for random access.

13.4 APPLICATION SPECIFIC INTEGRATED CIRCUIT (ASIC)

An Application-Specific Integrated Circuit (ASIC) is an integrated circuit, which solves one task exceptionally well. ASICs are the superset of all integrated circuits, as CPUs and GPU are also considered ASICs, targeting general-purpose programming and graphics programming, respectively. The printing of an ASIC design is final, resulting in any errors in the design will remain for the entire

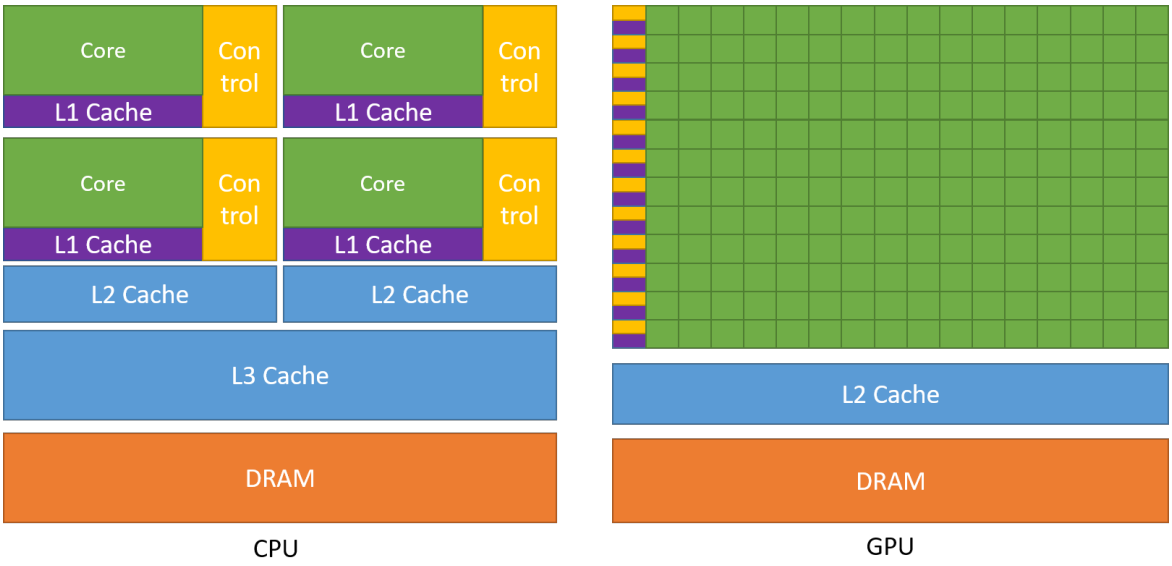


Figure 43: Coarse-grain block diagram of a CPU and a GPU. Image from [35].

chip's lifetime. Most production chips handle this by having a small region describing certain parts of the chip's behavior, known as firmware or microcode, which can be reflashed post-production.

ASICs have a very high initial investment cost but a low delta cost for additional units, resulting in bulk orders rendering the ASIC cost-effective. This effect makes off-the-shelf hardware very cheap through large production quantities and custom hardware very expensive through small production quantities. As such, most widespread chips are tuned for generality, as this allows them to target a wider audience, making the large-scale production profitable even at a lower per-unit cost.

An ASIC focused on solving a specific task will always be more efficient than a general-purpose chip, as in the worst case, the ASIC could be an implementation of the general-purpose chip. The need for specificity has made its appearance in the general-purpose chips as more and more chips gain small dedicated pieces of hard logic attached. For example, most modern CPUs carry hard logic for computing AES [36] and h264 [37], and in the most recent GPU, we find dedicated logic for ray-tracing [38].

However, ASICs are costly to produce, both design and production. Designing an ASIC takes a long time, given that they are very complex. This complexity arises from the fact that everything has to be described by the developer. Some sub-components, such as Double Data Rate (DDR) Synchronous Dynamic Random-Access Memory (RAM) (SDRAM) or Peripheral Component Interconnect Express (PCIe) controllers, can be provided in the form of Intellectual Property (IP) cores. The problem with these is that they usually carry legal agreements to use them, not providing the same open-source community of the software world. Finally, functionality verification is very computationally expensive. Production takes months, as the design must be verified electronically, e.g., no short-circuiting.

13.5 FIELD-PROGRAMMABLE GATE ARRAY (FPGA)

In between the general-purpose nature of the CPU and GPU, and the specificity of an ASIC, we find the FPGA. Conceptually, these chips consist of a grid of logic gates that are connected through an interconnect, as Figure 44 shows. The specifics of this interconnect, and the internals of the logic

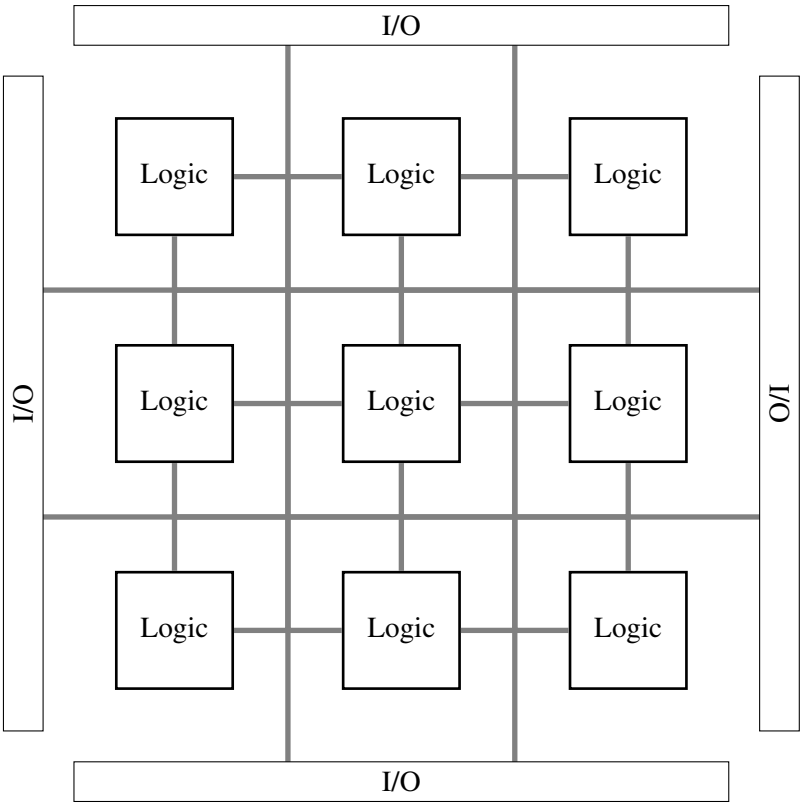


Figure 44: Block diagram of the conceptual FPGA. The gray lines are programmable interconnects.

blocks are what the programmer specifies. Thus, an FPGA can become every integrated circuit within the bounds of how many gates are needed. An FPGA is *reprogrammable in the field* rather than being fixed post-production like the ASIC. This field-programmability defines FPGAs as *reconfigurable hardware*. Because they have traded some of the specificity for general-purpose, they are cheaper in smaller volumes.

Compared to CPUs and GPUs, like ASICs, an FPGA implementation can be tuned towards an application-specific domain, making them highly efficient in that domain. By being so close to ASICs, many of the optimizations found in traditional hardware design also apply to FPGAs. For example, high-throughput FPGA designs favor deep pipelines, allowing for instruction-level parallelism-like exploitations, making them more efficient than GPUs for some computing problems. FPGAs can handle branching better than CPUs, given that the pipeline can be tuned towards misses, thus minimizing the need to flush the pipeline.

Data types in FPGAs are not as strictly defined as on CPUs and GPUs, allowing them to compute in almost any bit width. However, they favor narrow bit widths as this fits nicely onto the components of the chip. The chips are also often fitted with high-throughput interfaces, E.g., networking, making them great for online (bump-in-the-wire) computing.

Like ASICs, FPGAs are programmed through hardware descriptions and IP cores. While there exist many different levels of abstraction for programming, the lowest level is known as Register-Transfer Level (RTL), which Section 13.5.1 describes. The most popular high-level approach is known as High-Level Synthesis (HLS), which Section 13.5.2 describes. Regardless of the entry point, every design must follow the same implementation steps; first, we synthesize the design into an Intermediate Representation (IR) consisting of the components available on the chip. Then we map this IR onto

actual physical components on the chip, which are finally linked together during routing. The result is a bitstream, which configures the logic blocks and the interconnects of the FPGA.

The grid of gates view of an FPGA while beneficial for building intuition and as a high-level abstraction, is somewhat disconnected from reality. At the lowest level, FPGAs implements their basic functionality through LookUp Tables (LUTs) and Flip-Flops (FFs). LUTs are programmable logic gates featuring multiple inputs and outputs mimicking complex logic gates through programming them with a predefined truth table. FFs are registers for keeping a single bit of memory. As the FF is expensive area-wise, the Block RAM (BRAM) allows for increased memory at the cost of latency. They feature two ports, each allowing one read or write transaction per clock cycle. The final internal component is the Digital Signal Processor (DSP); a lightweight ALU for basic arithmetic operations. Newer boards even feature floating-point DSPs for native floating-point support.

13.5.1 *Register-Transfer Level (RTL)*

RTL programming is the lowest abstraction of hardware description. Here, the developer specifies the registers and the signals in between them. Some abstractions do exist, such as `if` statements for multiplexors and the + operator for specifying an adder, but these are very generic.

RTL is programmed using a Hardware Description Language (HDL). The most popular are Verilog or Very High-Speed Integrated Circuit (VHSIC) HDL (VHDL). Both of them have their pros and cons, which is why they both still exist and are supported by multiple vendors. Generally, they are very similar and can both describe the same circuit, albeit in different ways.

HDLs are verbose, cumbersome, and outdated - at least the versions supported by the major vendors. Although both languages have evolved, Xilinx Vivado and Intel Quartus only support Verilog-2001 and partially support VHDL-2008 standards to this day. While third-party tools exist for synthesizing into an acceptable format, utilizing these is suboptimal, introducing cross-project dependencies. The verbosity of the HDLs translates into hundreds, if not thousands, lines of parallel code, which raises the probability of error significantly, especially given that parallel programming is complicated - at least for software developers. As a result, there is a need for another approach to FPGA development.

13.5.2 *High-Level Synthesis (HLS)*

In order to introduce software developers to the world of hardware development and increase productivity, the two major FPGA vendors Xilinx and Intel, have both introduced HLS, which is implemented in C or C++ and translates to RTL through the vendors' respective compilers. C and C++ are both languages, which are very close to the assembly code of a CPU. Their programming model is sequential, fitting the sequential execution model. However, FPGAs are inherently parallel, which is a mismatch for the HLS model. The compilers handle this through two approaches: building a state machine that will mimic the sequential processor or the developer instructing it through decorators. The state machine approach, while correct, might introduce a great deal of overhead by not actively computing the entire time, thus having unused sections. For some problems, this will be sufficient as we can produce complex hardware from very little code.

The decorator approach allows the developer to instruct the compiler on the type of optimization and where to apply it. The problem with this approach is that some optimizations cannot be expressed in this abstraction, leading to potential losses. Furthermore, it also puts a lot of the responsibility on the developer, requiring them to have extensive domain knowledge of the underlying hardware platform.

14

HARDWARE PROGRAMMING MODELS

As described in Sections 13.5.1 and 13.5.2, programming FPGAs can be hard - especially for software developers. To solve these problems, many have made programming models targeting hardware design, which are even higher levels of abstraction. As the field of programming models for FPGAs is vast, we will only go through the two models that have been extended as part of this Ph.D. dissertation, along with honorable mentions at the end.

14.1 SYNCHRONOUS MESSAGE EXCHANGE (SME)

Synchronous Message Exchange (SME) is a programming model targeting FPGA development at an abstraction layer just above RTL level programming. The idea for SME arose when a project tried to use Communicating Sequential Processes (CSP) for hardware development. While it did show that isolation of share-nothing processes was beneficial, intermediate processes were needed to mimic the actual behavior of hardware. For example, there needed to be a global clock signal synchronizing all processes and special broadcasting processes between every process. As a result, SME was created by implementing the functional elements of CSP and introducing new concepts fitting for hardware development.

Like CSP, an SME program has the same structure; sequential processes share nothing except for their means of communication. Where CSP processes run immediately and only once, an SME process does not run until triggered and is triggered multiple times during a simulation. While the execution model inside the processes is sequential, parallel internal constructs will run in parallel, just like RTL. As in, an SME process does not (necessarily) create a state machine like HLS.

In CSP, the communication channels are rendezvous; communication does not happen until both sides are ready. In SME, the communication channels, called buses, use broadcasting. A value may or may not be read from the buses by one or more processes; the writer does not necessarily care. The propagation of values on the buses occurs based on the global clock.

This global clock drives an SME simulation. It discretizes a clock signal, is hidden, and implicitly connected to every process. During each clock cycle, the simulation triggers every process exactly once. Figure 45 shows an overview of a simulation cycle can. The order of when the processes are triggered depends on their triggering strategy. They can either depend on a previous process or the global clock. We build a dependency graph from this triggering mechanism, the execution model. Every set of processes that are disconnected can run in parallel. Thus, the parallelism in SME is explicit rather than implicit.

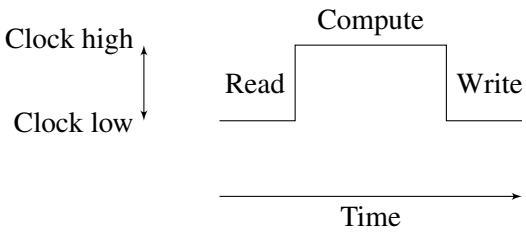


Figure 45: The simulation cycle of an SME simulation. The communication actions happen on the rising and falling edges.

We construct an Abstract Syntax Tree (AST) for each process, describing internal variables, buses, and internal computations. Thew we translate these ASTs into VHDL processes, semantically equivalent to the high-level processes. We construct the top-level VHDL files from the dependency graph files that connect the processes.

During simulation, we capture all of the values on the buses at each clock cycle and emit them into a Comma-Separated Values (CSV) file. We use this file for verification of the resulting VHDL implementation. It will take the CSV file, drive the top-level inputs with the values for each clock cycle, cycle the clock, and finally verify that all internal and external buses carry the same value as seen during the high-level simulation. As a result, we can easily argue that the resulting hardware model is clock cycle-accurate compared to the high-level model.

By being derived from CSP, SME has the further advantage of being mathematically verifiable. We can do this verification by translating SME into CSP, which by mimicking the original CSP for hardware design approaches. Following the original approach entailed converting buses to a set of channels and processes and introducing the hidden clock as a process itself. We then pass this CSP model to CSP verification tools, such as Failures-Divergences Refinement (FDR). All of this is described more in detail in the master's thesis by Alberte Thegler [39] dealing with this exact problem. It was submitted prior to this Ph.D. thesis.

14.1.1 *Improvements*

This Ph.D. thesis has sprung multiple improvements to the SME model. This subsection covers these in chronological order.

14.1.1.1 *Dynamic processes*

In the first versions of SME, we leveraged the type system for building the networks automatically. We instantiated every class which inherited from the `IProcess` interface would as a new process, and likewise for the interfaces inheriting from `IBus` would be bus instances. Connecting them would also happen automatically based on the fields of the process classes. While this automated a lot of the boilerplate code usually needed in CSP-like programs, it was also very limiting as there could only be one instance of every type of bus and process. Multiple type definitions could be generated through templates to generate multiple buses or processes to counter this limitation. However, the type definitions would unnecessarily explode for replicated designs, hindering performance as the

typing system was overloaded. Furthermore, it also broke compositionality since one would need multiple levels of templates.

We exchanged this automatic instantiation and connection for explicit instantiation and connection, as this would allow for the complete dynamic flexibility of C#, as long as we do it before running the simulation. Once we start the simulation, we collect all the process and bus instances and build the dependency graph. Thus, the resulting dependency graph is still static, as any low-level FPGA implementation requires. This functionality allows us to construct seemingly dynamic hardware models without breaking any of the requirements imposed by hardware.

14.1.1.2 *Asynchronous programming*

Regular processes execute their entire body within a single clock cycle. If the developer wants to do a computation over multiple clock cycles, e.g., through a state machine, they have to implement this and keep the state themselves manually. Manually managing states is a problem as a handwritten state machine can quickly become unmanageable, having obscure states and transitions. Thus, having an option of exploiting a higher level of abstraction for generating these would be desirable.

Asynchronous programming constructs are great for expressing concurrent events in modern programming languages. One example of using these constructs is waiting for a network package to arrive. Using asynchronous programming, the developer does not have to lock the main execution thread or write complex logic for handling polling while doing other tasks. Instead, the developer can write something like

```
auto package = await socket.read();
```

The compiler, or runtime environment, will set up a state machine to keep track of which tasks are executing when. For hardware development, these constructs can be very useful, especially because waiting for an event to happen before acting on it is a quite common action. So rather than having

```
while (true)
    if (signal)
        something();
        break;
```

the statement can become

```
await signal;
something();
```

Furthermore, we can now easily express a multi-cycle state machine, with the expression `await ClockAsync();`, as this expression tells SME that we stop the execution until the clock ticks.

To translate this behavior into hardware, we do transformations on the AST. We partition the `tick` function into fragments and the transitions between these fragments. The limitations of the state machines are that while we can have fall-through statements, we can only have them through inlining the fragments, as the vendor tools do not correctly recognize the resulting hardware as a proper state

machine otherwise. So, if the fragmented AST contains too deep levels of fall-through recursion, it will not work, as we cannot translate into static hardware.

Having implemented the asynchronous programming model, SME can now utilize dynamic constructs, such as `while` loops, which were previously unavailable in hardware development. We can further improve this model by using it to describe pipelines. For n stages, we would construct n copies of the internal variables. Then we would have all of the stages run parallel and access the previous state. The developer should be wary when using this, as the logic could rapidly explode. A downside is that it would be up to the vendor tools to perform optimizations for removing unused variables. While this would solve the problem in the resulting hardware model, we would still have to handle it in the high-level simulation. To fully mimic this behavior, SME would have to spawn n threads, all running concurrently.

14.1.1.3 *.NET Core*

In the previous versions of SME, we compiled networks using different compilers depending on the target platform; the .NET framework on Windows and Mono on OSX and Linux. Since then, Microsoft has acquired Mono and released a unified cross-platform compiler known as .NET Core. The same compiler on multiple platforms eliminates the previous compiler-independent quirks and significant performance differences. Whereas new language features first appeared on Windows, the unification ensures that all platforms are fully featured. While these features might not always be translatable into hardware, we can freely use them when building the SME network and describing simulation processes.

14.1.1.4 *Parallel execution during simulation*

With the introduction of .NET Core 3.0 `async` tasks now run in parallel. Given that SME uses tasks, we can exploit this parallelism by launching all the tasks that can run in parallel simultaneously. The compiler and runtime environment handles all the hazards introduced by parallelization, sadly at an aggressive level.

False dependencies are found on the buses, meaning that two independent processes sharing a bus will not run in parallel since the bus is locked. The bus should not be locked, as there are no data hazards due to the SME simulation cycle (Figure 45); data is not propagated between the processes during compute, which is when the data is written to the bus. Furthermore, SME does not allow multiple processes writing to the same bus field.

As an effect to the locking procedure, the processes execute sequentially, which does not break simulation correctness but decreases the potential runtime performance of the simulation. This locking mechanism should be solvable by splitting the bus into a reading end and a writing end so that one process would only lock one of the ends rather than the whole bus.

14.1.1.5 *Compiler over decompiler*

The first versions of SME constructed the AST by decompiling the Intermediate Language (IL), the binary produced by the .NET compiler. While this allowed SME to utilize optimizations made by

the compiler, these optimizations target the .NET runtime running on a CPU. These optimizations resulted in unwanted constructs, such as variable reuse and variable naming, which further obscured the AST and thus also the resulting VHDL.

New language features meant new IL code, which the decompiler did not recognize. The new IL meant additional complexity when maintaining SME, as one had to maintain both the model, the decompiler, and the transpiler. While keeping the decompiler up to date circumvented the majority of the errors, it tended to introduce significant Application Programming Interface (API) changes, resulting in rewriting much of the code.

In recent years, Microsoft has begun releasing its .NET compiler as an open-source project known as Roslyn. Though it is only partially released, it allows developers to access compiled program syntax trees and semantic models before applying any optimizations. By exchanging the decompiler with a compiler, every node of the SME AST can now point to the originating code locations rather than locations in the IL. This mapping improves naming, as we can now use the tokens from the originating code and error messages, as we can now point to the specific location in the originating code that caused the error. We can now leverage other previously heavy static code analysis tasks by accessing the semantic model. For example, we can now see the direction of variables to and from a function, as the semantic model holds information about which variables are read and written.

Compared to the decompiler, using a compiler is more time consuming when translating the network to VHDL. This increase in running time is due to having to compile the project multiple times, first when running the simulation, then when getting the initial syntax tree, and finally, every time the syntax tree is modified. The final point arises from having to recompile the project following every change. Another problem with using the compiler is that runtime types and compiled types cannot be directly compared, given that they exist in different scopes and compilations. So while they might resemble the same type, as we compile them from the same source, they are not the same runtime type. The solution to this problem is comparing the fully qualified identifier string. While this is not optimal, it should be sufficient for the SME setting.

14.1.1.6 *Custom processes*

While SME is very close to the metal, being implemented in a high-level language can have limitations. For example, bit-level manipulations are expressed through bit masking and shifting. To correctly utilize some internal components of an FPGA, such as BRAM, we have to write the code in a specific pattern or directly instantiate them through vendor tool-specific scripts. The first solution SME had to this problem was to have a library of processes that behave similarly to the low-level component during SME simulation but translate to some very specific VHDL code. This library can be small enough to cover the essential components for simple cases.

A great strength when programming is to use libraries. A library is an IP core in the FPGA world, a black-box hardware component that solves a particular task. The IP core that we use as an example is the floating-point IP offered by Xilinx. By using this, we have a vendor-optimized IEEE-754 compliant floating-point unit. Expressing the same operations is easy in C# because it natively supports the datatype. However, VHDL does not natively support the datatype. Instead of emitting VHDL code for the floating-point operations, the resulting VHDL should instantiate a floating-point IP core.

The last addition to SME is the ability to supply a custom code generator to a process. The developer still has to supply the C# code used during simulation and verification in the VHDL testbench. The

custom code generator will then produce the supplied VHDL code instead of using the regular code generator. Furthermore, it includes a field of extra commands to output in a resulting Tool Command Language (TCL) script, configuring and including the IP cores in the project. This extension is thus not limited to only Xilinx IP cores but to any IP core, which the Xilinx Vivado project includes. Furthermore, this should also be reasonably easy in the Intel toolchain due to the VHDL not changing too much, but rather the TCL commands needed by Quartus.

14.2 DATA-CENTRIC PARALLEL PROGRAMMING (DACE)

Data-Centric parallel programming (DaCe) [40] is a programming model described through its IR, the Stateful DataFlow multiGraph (SDFG). This IR focuses on explicit data management and movement, which remains the most significant performance factor in computing. It does so in pure dataflow regions by separating data containers and computations. This separation enables parallel programming and platform-specific graph transformations, further increasing the performance.

There are multiple frontends for SDFGs, such as Python, C, Open Neural Network eXchange (ONNX), or Basic Linear Algebra Subprograms (BLAS). These frontends allow the domain scientist to describe their problem at a higher level of abstraction, which can translate into an SDFG. Performance engineers can then pick up this SDFG and optimize it for multiple target platforms through the graph transformations.

The innermost level of the SDFG is made up of directed acyclic multigraphs consisting of data nodes (*accessors*), computation nodes (*tasklets*) and edges represent data movement (*memlets*). Within one multigraph, data dependencies dictate execution flow. *Map* scopes introduce concurrency and define regions of the graph that are executed multiple times and may run in parallel based on the map schedule. The multigraphs are contained within a *state* to support cyclic data dependencies and control-flow. Only one state is executing at one point, allowing for explicit control flow through state transitions when needed, such as the case with certain for-loops. Within one state, multiple multigraphs, or *subgraphs*, that are not directly connected will run in parallel.

During the Ph.D. dissertation, I visited Eidgenössische Technische Hochschule (ETH) Zurich for ”the change of environment,” where I worked with the Scalable Parallel Computing Laboratory (SPCL) group to introduce new functionality to DaCe targeting FPGAs. This work has primarily revolved around the HLS ecosystem but with a focus on DaCe. As such, the work is not bound to DaCe but enabled by DaCe; this work is applicable to any framework targeting HLS, amongst which even HLS itself resides. The work has targeted the Xilinx FPGAs and toolchain but should conceptually be applicable to any FPGA workflow.

14.2.1 *RTL backend*

In the beginning, DaCe FPGA code generator targets HLS. As stated in Section 13.5.2, there is a mismatch between the programming model and target platform. HLS is great for many things but lacks the fidelity found in the RTL languages, requiring certain use-cases and optimizations. For example, HLS does not allow IP cores except for those available in the HLS libraries. Another example is the multi-pumping optimization, where the developer exploits the different bit widths of the FPGA components through multiple clock regions. Section 14.2.2 covers multi-pumping in detail. While

rewriting the code generation backend into emitting pure RTL could solve the lack of fidelity of HLS, it is not desirable, as HLS is an excellent tool for productivity. HLS is especially good at expressing memory movement and access, which would be very tedious to describe in RTL.

The work uses multiple levels of abstractions to express high-performance FPGA designs with minimal implementation complexity. The idea arose from a technique that was quite common in early CPU development; inline assembly. With inline assembly, the developer could write the majority of the code in C, which is a lot less cumbersome than assembly, and only focus on optimizing the bottlenecks of the resulting assembly program. This feature gave the developer more freedom, as they could leverage C's productivity and the expressibility of assembly. We will do the same, but for FPGAs - give the developer the power of "inline RTL" in an HLS environment, allowing them to get the best of both worlds. This functionality is integrated into DaCe, expanding its capabilities when targeting FPGAs.

14.2.1.1 *Implementation angle*

Using Xilinx's toolchain, there are two ways of integrating RTL into HLS; through black-boxing or an RTL kernel.

Black-boxing is the closest to inline assembly. The developer provides the RTL code, a C-style function signature, and a metadata eXtensible Markup Language (XML) file. With this information, the HLS compiler can use the RTL block like a regular function, along with the provided metadata for the outside data plumbing. One problem with using this approach is that the metadata file provides latency and iteration interval information. While we could post this as a required deliverable by the developer, this is undesired. Another approach would be to do a static analysis of the RTL code, which is infeasible, especially for the general code case. As a final note, RTL black-boxing imposes the same constraints as found in HLS [41], thus removing the expressibility we aimed to achieve by leveraging RTL.

The RTL kernel approach is where the RTL code becomes its own Open Computing Language (OpenCL) kernel. External communication is the driver behind synchronizing with the host or other kernels. For this to work, the kernel must adhere to a known interface: a controller for the host program to communicate with it, and then either ARM Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) [42] or AMBA Streaming AXI [43] buses for non-scalar input/output. The memory region exposed by the controller provides the scalar arguments. After packaging the kernel, the compiler can include the kernel during linking, and then the kernel can be launched like any other kernel. From the host program's point of view, the final kernels are indistinguishable.

14.2.1.2 *Kernel requirements*

When defining a controller for an RTL kernel, there are three approaches: free-running, controlled, and chain controlled.

Free running is where the kernel has no control interface, and triggering happens through external communication from other kernels. When the bitstream is written to the FPGA, the kernel launches and can only reset when the whole FPGA board resets, which can become a problem for kernels

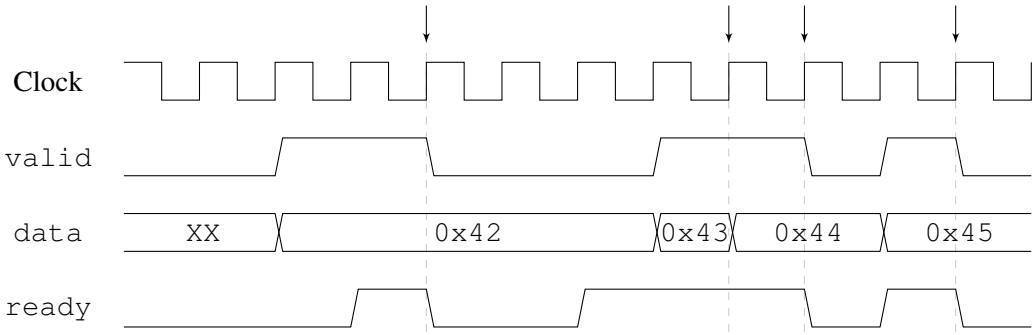


Figure 46: Waveform of four streaming AXI transactions; `0x42`, `0x43`, `0x44` and `0x45`. The first one is a single transaction, with the latter two being a burst of two consecutive, and the final being a single transaction. The arrows indicate when the transaction occur. It is not important whether `valid` is asserted before `ready`.

keeping an internal state. Scalar arguments are unsupported, as the controller interface usually supplies them.

Controlled kernels are the default type of kernel. The only requirement is that the kernel follows the controller interface as specified. It can be free-running as well, by ignoring the `start` commands and by keeping the `done` flag held high. This type of kernel can be launched and reset based on commands through the controller, thus making this the optimal kernel type for keeping a long-running state across runs within a single flashing of the FPGA.

Finally, chain-controlled kernels can also communicate controller-like commands to each other. For example, if we have a pipeline of kernels, rather than having the controller trigger the next kernel in a row, which would be undesirable given that the host is off-chip, the kernels can trigger each other.

We start with controlled kernels that mimic free-running kernels and use the streaming AXI protocol for this initial implementation. We choose this approach because we want to support scalar arguments, and the streaming AXI protocol is lightweight and simple to implement in RTL. This type of bus consists of three required signals: `valid`, `data` and `ready`, with `ready` going in the opposite direction of the first two. A transaction occurs when both `valid` and `ready` have been asserted for one clock cycle. An example set of transactions can be seen in Figure 46.

14.2.1.3 *Implementation modes*

Traditionally in FPGA development, we have three levels of implementation; software simulation, hardware emulation, and actual hardware. For each step ”downwards,” compilation times increase in orders of magnitude, resulting in seconds, minutes, and hours respectively. Software simulation is the fastest step but is only functionally equivalent with the resulting hardware. Hardware emulation is behaviorally equivalent to the resulting hardware but is very slow to emulate. The final step, hardware, has the fastest runtime but takes hours to compile. The developer should go through each step, as the first couple of steps are better for capturing early compilation errors and functional bugs.

The HLS compiler cannot simulate RTL code in software but can run the C++ kernels for software simulation. So, in order for us to simulate the RTL code, we need to make it linkable with regular C++ code. The project Verilator [44] does precisely that; it translates Verilog and System Verilog into C++ code, defined as the verilated code. The only added overhead is to specify a driver function that runs

and drives the signals in the verilated code and interfaces with any external C++ code. This software controller is generic enough to generate it and the rest of the code generation. Implementing the code generator for Verilator from DaCe was developed by a former employee in the SPCL group - Andreas Kuster. With Verilator, we can now simulate the RTL code in software alongside the HLS code and the host functions.

The next step is hardware emulation, which emulates hardware, but is orders of magnitude faster at compiling the designs while being orders of magnitude slower at running the design. Where Verilator only required us to specify two files, the Verilog and driver, we need the following files for hardware emulation: a controller for host communication, a top-level file combining the RTL code with the controller, and a packaging script for the vendor tools. Due to these files sharing the same overall pattern throughout different implementations, the kernel signature holds the information for generating all the files. We have written a library called rtllib [45], which provides Python files for generating the files, along with a CMake script for compiling these kernels. Finally, now that we have several kernels that need to communicate, we also need to emit a configuration file. This configuration file is used during linking and defines extra information about the kernel; Whether to replicate the kernels, which memory banks to connect to, which part of the FPGA to place them in, and, finally, which kernels connect to which kernels. The final implementation flow then becomes:

- Emit the source code file.
- Emit the controller source code file.
- Emit the packing script.
- Emit the configuration file.
- Invoke the packing script producing an `.xo` kernel file.
- Use the configuration file and `.xo` file during linking.

After these steps, we get a device binary, which a host program can invoke. For hardware emulation, this device binary is launched in an emulated FPGA, removing the need for and the performance of an actual FPGA board. We need to set a different flag for the compiler for the final step. With all of this in place, we can now simulate software and hardware emulation implementation while compiling the entire solution into an actual hardware implementation.

14.2.1.4 *General application architecture*

All SDFGs must follow the same general structure where the RTL subgraph is isolated and communicates through streams. In order to access global memory on the FPGA, we need to have reading and writing actions in separate subgraphs, which transfer memory between global memory and streams. This structure is simple to follow, and transformations can help achieve this structure. The transformation from a regular memory accessing SDFG to a streaming SDFG can be seen in Figure 47. There can be a mix of multiple RTL and HLS subgraphs.

Inside the RTL of the RTL subgraph, the developer is free to do whatever. They can even instantiate multiple clock signals through clock multipliers and dividers. However, RTL kernels also carry the functionality to receive multiple clock signals from the FPGA shell, which is essentially a common interface for communicating with the host. By utilizing this functionality in our RTL kernels, the clock

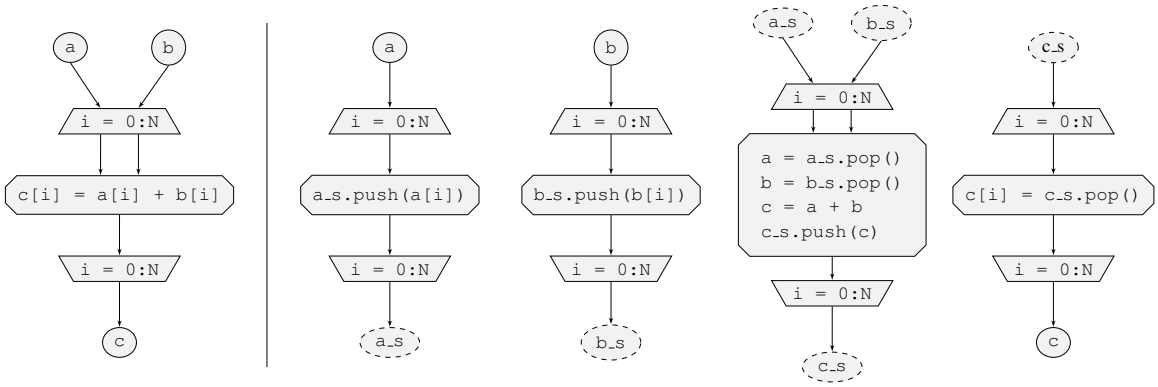


Figure 47: SDFGs depicting the transformation from (**left**) a regular memory access subgraph to (**right**) three subgraphs using streams.

frequencies are specified during linking, allowing for easy clock scaling. As additional functionality, we can unroll the maps wrapping the RTL tasklets, which the code generation then handles through replicating the RTL kernel. Rather than instantiating a single instance of the RTL code, multiple instances are instantiated in the top-level Verilog file and are connected based on the memlets in the SDFG.

14.2.1.5 *Examples*

To showcase the RTL backend, we have a couple of experiments that showcase the use. All of the examples follow the structure described in Section 14.2.1.4.

VECTOR ADDITION

Vector addition is the ”hello, world” example when implementing anything for accelerators. It computes the results of $c = a + b$, where a and b are vectors of length N. Implementing this in DaCe is straightforward; we construct a reader for a and b, a writer for c, and then we construct the mapped tasklet `vadd`, which handles the streaming AXI interface logic, along with the computation. The Verilog code can be seen in Listing 14.1.The most interesting conclusion from this example is that the backend compiles, runs, and verifies against a CPU implementation.

HISTOGRAM

This example computes a histogram given a series of values. Interestingly, a data hazard exists, which is not trivial to handle in regular HLS. DaCe supports data-hazard constructs, so this merely shows that finer-grained control enables the opportunity to handle problems directly. We can handle the hazard because we can force the addition to occur within a single clock cycle and specify that the Block RAM should be in write-first mode, ensuring correct forwarding. Figure 48 shows the block diagram.

DOUBLE PUMPED AXPY

The final example shows off all of the functionality enabled by the RTL backend through a vectorized, replicated, multi-pumped implementation of the BLAS routine `AXPY`. This routine computes $a \cdot x + y$, where a is a scalar, x and y are vectors of size N. Section 14.2.2 describes the multi-pumping optimization more in-depth, but in short, it exploits the fact that the internal components of an FPGA

```
reg valid_a = 0;
reg [31:0] a = 0;
reg valid_a = 0;
reg [31:0] a = 0;

@always(posedge ap_clk) begin
    if (s_axis_a_tvalid && s_axis_a_tready) begin
        a = s_axis_a_tdata;
        valid_a = 1;
        s_axis_a_tready = 0;
    end
    if (s_axis_b_tvalid && s_axis_b_tready) begin
        b = s_axis_b_tdata;
        valid_b = 1;
        s_axis_b_tready = 0;
    end
    if (~m_axis_c_tvalid) begin
        if (valid_a && valid_b) begin
            m_axis_c_tvalid = 1;
            m_axis_c_tdata = a + b;
            valid_a = 0;
            valid_b = 0;
        end
    else
        if (m_axis_c_tvalid && m_axis_c_tready) begin
            m_axis_c_tvalid = 0;
            s_axis_a_tready = s_axis_b_tready = 1;
        end
    end
end
```

Listing 14.1: Verilog core of the vector add example.

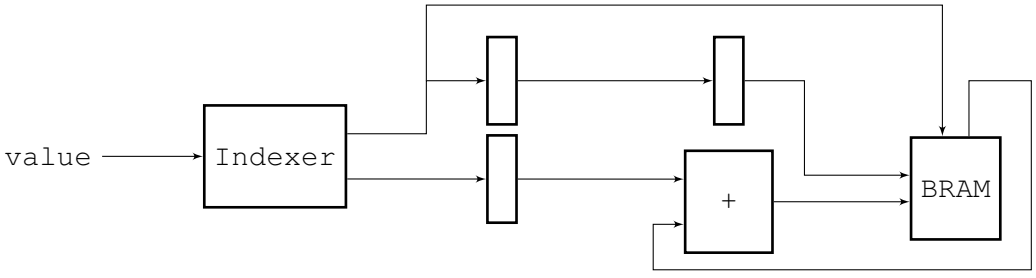


Figure 48: Block diagram showing the inner histogram implementation.

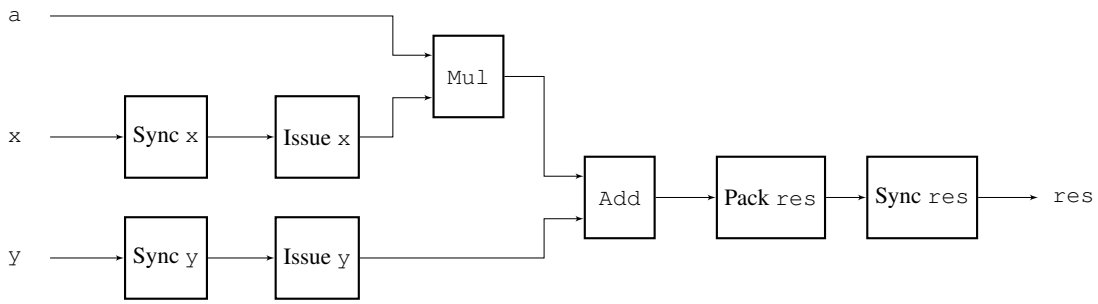


Figure 49: Block diagram of the internal RTL code for AXPY. All of the blocks are Xilinx IP cores.

can run multiple times faster than the external connections of the FPGA. Figure 49 shows the block diagram of the internal RTL code.

14.2.2 *Automatic multi-pumping*

As mentioned in Section 14.2.1, having the RTL backend now enables us to do the optimization technique known as multi-pumping. Computational components are usually densely connected by *short* paths, while the data paths connecting them are *long* paths across the chip. From the data movement perspective, we can see multi-pumping as a form of ”temporal vectorization”: widening the *long* data paths leading to and from the computation, but leaving the densely connected logic performing the computation itself unchanged, conceptually being ”vectorized” across multiple clock cycles. The densely connected short paths now only have to meet timing at a higher frequency locally, while the long data paths do not need additional buffering of the signal that high frequencies otherwise usually require.

Relative to traditional vectorization, temporal vectorization relaxes the application’s requirements. It must still be possible to parallelize the memory source/destination, but it does not impose any requirements on the computation - the computation does not even need to be analyzable. Dependencies between iterations are allowed without any additional handling.

Introducing multi-pumping to a design is an invasive procedure that requires significant effort, requiring us to perform clock domain crossing and data width conversion at either end of the higher clocked domain. In particular, in HLS development flows, multi-pumping is either not supported altogether or severely limited in scope, resulting in this optimization rarely being exploited for FPGA development in practice.

14.2.2.1 *Multi-pumping*

Programming FPGAs with HLS revolves around designing deep hardware pipelines, exploiting the spatial parallelism offered by the device. Optimizing compilers and performance engineers leverage classical high-performance computing and FPGA-oriented transformations to achieve this goal [46]. Resource utilization is a metric that we must consider when optimizing code for FPGA, as space consumption can be one of the critical factors limiting the performance of FPGA large-scale designs.

Traditionally, resource-sharing techniques reduce area consumption at the expense of degraded circuit performance. Multi-pumping aims at overcoming the limitations of other solutions by exploiting the capability of the hardware fabric of running different components at different clock rates. FPGA designs created with modern HLS tools typically run at 200-350 MHz, while the internal components can reach higher frequencies. As a concrete example, consider the Xilinx Alveo U280 [47]. The Xilinx Vitis documentation suggests that to maximize data throughput, a full AXI bus with a width of 512 bits should be used [48]. Assuming we can read an entire transaction of 512 bits every clock cycle, in order to reach the theoretical limit of 460 GB/s for the High Bandwidth Memory (HBM)2 memory, which has 32 banks, we would need a clock rate of

$$\begin{aligned} 460\ \text{GB/s} / 32\ \text{banks} &= 14.375\ \text{GB/s / bank} \\ 14.375\ \text{GB/s} \cdot 1024\ \text{MB/GB} &= 14720\ \text{MB/s} \\ 14720\ \text{MB/s} \cdot 8\ \text{b/B} &= 117760\ \text{Mb/s} \\ \frac{117760\ \text{Mb/s}}{512\ \text{b/Hz}} &= 230\ \text{MHz} \end{aligned}$$

The internal components, such as the DSPs and BRAM, can reach frequencies of up to 775 MHz and 737 MHz, respectively [49], more than three times the HBM2 and at least two times that of the average HLS design. While reaching such frequencies is infeasible (due to routing and timing closure requirements), we can further exploit the internal components in high-level FPGA designs.

14.2.2.2 *Exploiting multiple clock domains*

FPGA designs usually have a single clock region, where the entire design shares the same clock signal. To apply multi-pumping, we need to have at least two clock regions, one for the slowly clocked components (such as the reader/writer to external memory) and one highly clocked region for the internal components.

Consider the case of a v-way vectorized computation, reading a vector x of width v, every tick of its clock clk_0. We replicate the internal components (c) v times to process the entire vector. Let us assume that we can clock c at a frequency F, which is M times larger than the frequency of clk_0. By applying the multi-pumping optimization, we derive a new implementation, with an additional clock signal clk_1, clocked M times higher than clk_0, used to drive components c. We no longer need to use v units of c to keep up with the data rate since each of them can consume M times the original number of data elements per tick of clk_0. The new implementation has the same throughput as the original one, but at a reduced resource cost of v/M units of c, compared to the previous cost of v units. Figure 50 shows a waveform describing this behavior.

The multi-pumping optimization does not come without a cost: synchronizing and multiplexing the data at the clock domain crossings introduces overhead. We must convert data entering the multi-pumped region from one wide vector of size v to M narrow vectors of size v/M — and the inverse for leaving the multi-pumped region. Whether this yields a net benefit depends on the size of the multi-pumped region, the number and the size of data paths entering and leaving the region, and on the overall constraining resource (i.e., introducing the optimization might cut DSP and BRAM usage at the cost of LUTs and FFs).

Multi-pumping can affect either *inner* (e.g., leading to compute blocks), or *outer* (e.g., leading to off-chip memory) data paths. The solution described in Section 14.2.2.2 falls in the former category. The width of the external data path (used to read vector x) is not modified, while we divide the

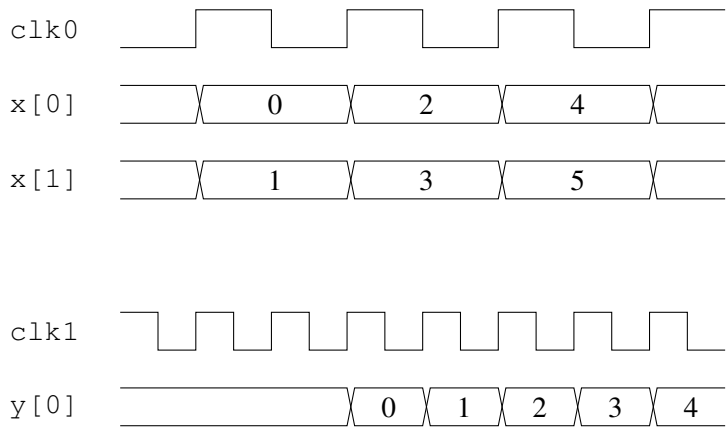


Figure 50: Waveform depicting the multi-pumping optimization with $M = 2$, $v = 2$. Image from [50].

width of the inner data path (leading to components c). The resulting design maintains the same throughput at reduced resource consumption. We can use the freed resources to enable additional scaling whenever possible. In the *outer* approach, we do not modify the internal design region, while we do multiply the external widths. The outer approach translates into increased throughput at the same resource consumption. These two approaches can be applied to vectorized computations, providing complementary results. The outer one is more versatile since we can conveniently apply it to problems that we cannot easily vectorize, as we are preserving the dependencies between iterations. FPGAs favor deep pipelines, and multi-pumping optimization should further push the benefits of deep pipelines: it enables feeding the circuit in a vectorized fashion without requiring the problem to be vectorized.

14.2.2.3 *Automatic application of the multi-pumping optimization*

Using the vendor tools, the only way to utilize the multi-pumping optimization is through hand-tuned RTL code. However, RTL is complex, verbose, and cumbersome, motivating lifting the optimization into higher levels of abstraction. HLS has been shown to improve productivity but does not support multiple clocks [41] required to implement multi-pumping. Instead, we use DaCe, where we exploit the explicit data movement given by the DaCe IR to detect program subgraphs where the multi-pumping optimization can be applied and write a transformation that does so automatically. Following the transformation, we use DaCe to generate RTL modules, enabling multiple clock domains.

Any automatic transformation on the DaCe IR has three aspects to consider: *identifying* a candidate subgraph, *assessing* transformation feasibility, and *applying* the necessary changes. Our automatic multi-pumping transformation applies to programs regardless of their computational contents, rather than by tracing and mutating their data movement properties. We summarize the steps of the multi-pumping process in Figure 51 and detail them below.

For *identification*, we attempt to greedily take the entire application in its DaCe IR form and find the largest subgraph that can be *streamed*, that is, when can convert data dependencies between two components to queue-based access. We do so by leveraging DaCe's existing infrastructure for tracing module input and output index expressions. By performing an intersection check on each pair of connected modules, we can determine if pipelining the memory between two modules can be performed.

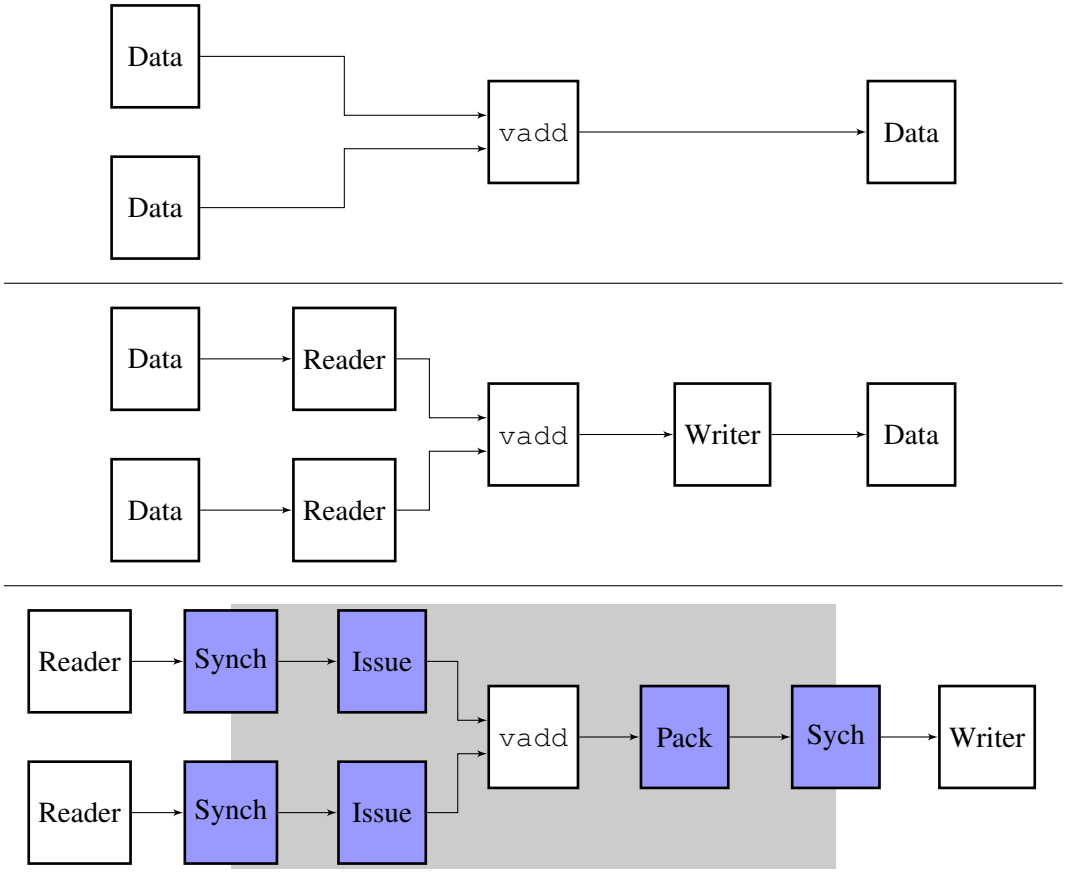


Figure 51: Block diagram for a vector addition computation core. The top diagram shows the original implementation. The middle diagram shows the conversion to a streaming implementation. The bottom diagram is after applying the multi-pumping optimization. Dark blue rectangles are IP cores, and the shaded area is the higher-clocked region. Image from [50].

Once we stream the computational modules, we can automatically inject *reader* and *writer* modules externally to the computational part. The only change required is working on these streams rather than accessing memory directly. For example, the vector addition example computing $c = a + b$ consists of a single loop, reading from a and b, computing the result, and writing the result to c. To change it to a streaming implementation, we construct two new functions that read from a and b and push the values onto streams, and similarly for c (see Figure 51, middle for example). Now that the communication on the streams drives control flow, all three functions can run in parallel, allowing us to modify the rates of the interfaces to increase potential throughput.

To *assess* the feasibility of multi-pumping a streaming implementation, we build upon techniques used by compiler auto-vectorizers. As we equate the process to temporal vectorization, the same conditions that apply to Single Instruction, Multiple Data (SIMD)-capable code apply in our tests (i.e., running the same operations on multiple data, predication instead of conditional control flow). Moreover, temporal vectorization is slightly more relaxed than the traditional SIMD paradigm — as the instructions run in sequence (albeit faster), internal sequential dependencies across data are allowed.

We adapt the existing vectorization infrastructure available in DaCe to perform the relaxed check and *apply* the data-centric graph modification. The only requirement we impose for the multi-pumped region is that it follows a predefined set of interfaces that is instantiable from within RTL. In all our experiments, our computations are generated through HLS, using streaming AXI as the communication protocol. With that, we can modify the clock rate of our identified subgraph, keeping the rest of the design intact.

The remainder of the transformation performs global clock management. We inject "plumbing" modules that synchronize the data paths between the clock regions and distribute the data over the temporal dimension. We use the built-in Xilinx streaming AXI infrastructure IP cores [51], which implement three types of modules:

DATA SYNCHRONIZERS These IP cores synchronize a data stream between two clock regions. We use them in both directions, both going in and out of the multi-pumped region. It is the first IP core in the chain since the remaining IP cores must run at the multiplied clock rate.

DATA ISSUERS These IP cores divide a single transaction from a wide data stream into multiple transactions on a narrow data stream. We use these IP cores when moving data into the multi-pumped region.

DATA PACKERS These IP cores are the inverse of the issuers; they take multiple transactions from a narrow data stream and pack them onto a single transaction to a wide data stream. We use these IP cores when moving data out of the multi-pumped region.

The three modules are customized based on the provided clock signal and the widths of the data streams. Continuing the vector addition example (Figure 51, bottom), we construct two clock regions: one region for the readers and the writer and one for the computation core. For each stream handled by the readers, we insert an instance of a synchronizer and a data issuer. For the stream handled by the writer, we insert an instance of a data packer and a synchronizer.

14.2.2.4 *Evaluation*

We evaluate our approach on a Xilinx Alveo U280 accelerator. Kernels are built with Vitis `2020.2`, targeting the `xilinx_u280_xdma_201920_3` shell. To remove external sources of congestion that are not our direct control, unless otherwise specified, we consider the following configuration:

SINGLE SLR The U280 is a multi-chiplet FPGA, where multiple Super Logic Regions (SLRs) are combined to form the overall chip. While this allows for larger chips, die-crossing interconnects further complicate floor planning, significantly lowering the maximum achievable frequency. Therefore, for our evaluation, we restrict to using a single SLR.

DIRECT ACCESS TO HBM BANKS The U280 is equipped with 32 HBM banks, all connected to `SLR0` [47]. We use each bank exclusively to store a single container to remove potential congestion when multiple entities access the same memory bank.

We showcase the optimization applied to four applications: vector addition, matrix multiplication, three-dimensional Jacobi stencil, and three-dimensional diffusion stencil. We have reported each application's resource utilization, design frequencies, and performance. While we can apply our approach to other pumping factors, the maximum achievable frequency by Vivado limits us (which is 650 MHz for the used version) for this version. We summarize the findings in Figure 52. Vector addition has been omitted in this plot but was able to reduce consumption of the critical DSP resource at the cost of a rise in every other resource. It hit a higher internal clock than the regular implementation, but this did not translate into a speedup since the regular version already ran fast. We can see that we obtain a speedup for the other applications while reducing the resource consumption of all resources except the LUT memory.

14.3 HONERABLE MENTIONS

While this chapter has touched upon two unconventional programming models for FPGA, there exist many different, each with its own strengths. This section will briefly cover them to give the reader a quick overview of the FPGA development landscape. Note that this list is *not* comprehensive and should not be read as such.

CλASH [52] is a Haskell-based programming model seeking to leverage the implicit parallelism and functions without side effects found in functional programming languages. It is common to solve problems with recursive function definitions in functional languages. Recursion is limited in Cλash, as their targeting FPGAs do not allow for dynamic constructs, thus unrolling the recursion to fit the statically allocated hardware. The same limitation applies to lists, which must also have a fixed size at compile time.

HETEROCL [53] is a programming infrastructure comprised of a Python-based Domain Specific Language (DSL) and a compilation flow. HeteroCL provides an abstraction that decouples algorithm specification from three hardware customizations: compute, data types, and memory architectures. This abstraction allows the developer to explore different tradeoffs systematically.

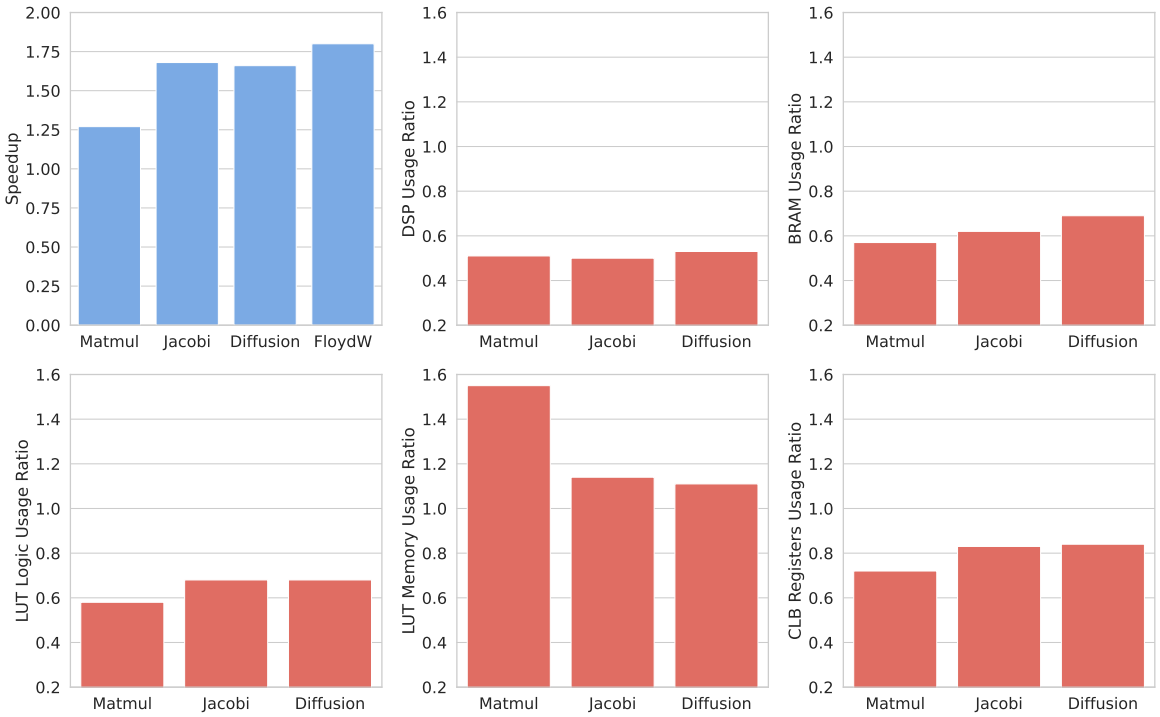


Figure 52: Performance and resource-saving overview. Blue: the speedup of the best performing double-pumped version over single-pumped, for each considered application. Red: ratio of resources used by double pumped version over the single one, considering the same application configuration (32 Processing Elements (PEs) for Matmul, 16 times iterations for stencils). Image from [50].

DYNAMATIC [54] is an open-source HLS compiler that produces synchronous dynamically-scheduled circuits from C/C++ code. When pipelining a loop, a typical HLS tool creates a static schedule — a conservative execution plan for the various operations. Dynamatic achieves a dynamic schedule, where the schedule is adapted at runtime to detailed data and control outcomes. Dynamatic generates synthesizable RTL that delivers improvements compared to commercial HLS tools in specific situations, such as applications with irregular memory accesses or control-dominated code. The compilation flow is LLVM-based and is customizable and extensible to target different hardware platforms.

CHISEL [55] is a programming model for constructing hardware, written in Scala. It has been developed by the same team that made RISC-V at Berkeley. The team behind Chisel provides regular updates and reports that it has good use in education. The Chisel approach is very similar to the SME approach in that items are created as isolated units with explicit communication.

PYRTL [56] provides a collection of classes for Pythonic RTL design, simulation, tracing, and testing suitable for teaching and research. PyRTL builds the hardware structure as explicitly defined and is thus not an HLS.

MYHDL [57] is an open-source package for using Python as a hardware description and verification language. Hardware is described through Python generators, which can be seen as resumable functions. MyHDL generators are similar to always blocks in Verilog and processes in VHDL.

SYSTEMC [58] is a C++-based modeling platform supporting design abstractions at the register-transfer, behavioral and system levels. The advantages of SystemC include the establishment of a common design environment consisting of C++ libraries, models, and tools, laying a foundation for hardware-software co-design.

SPATIAL [59] a DSL and compiler for high-level descriptions of application accelerators. Spatial feature hardware-centric abstractions for both programmer productivity and design performance. It achieves performance through these abstractions by having pipeline scheduling, automatic memory banking, and automated design tuning driven by machine learning. Spatial targets FPGAs and Coarse-Grained Reconfigurable Arrayss (CGRAs) from common source code.

FLEET [60] a framework that offers a massively parallel streaming model for FPGAs. Fleet requires that the user provides an RTL core that serially processes every input token in a stream, which it then integrates and replicates through Chisel.

14.4 CASE STUDIES

This dissertation has sprung many FPGA-related projects supervised throughout the dissertation. While we have not directly measured it, these projects gauge the feasibility and usability of the frameworks used, given that none of the primary project authors have had any prior experience with FPGAs. These projects include both SME and generic HLS, but for DaCe, only the examples showcased in Section 14.2 have been explored — at least as a direct outcome of this dissertation.

14.4.1 *RISC-V*

Daniel Ramyar carried out this project in his master thesis [61], where he implemented a RISC-V [62] processor in SME. It built upon a previous master thesis [63], which implemented a MIPS processor in SME. This project was especially exciting from the usability point of view, as Daniel was a physics student and thus had not taken any machine architecture or electrical engineering courses. He was able to implement the complete RV64I instruction set. He concluded that SME was an excellent tool for expressing the implementation details that he needed. He was able to synthesize the processor to a Pynq board, which is described in Appendix C, and reached 124 MHz. Figure 53 shows the block diagram of the SME design of the resulting RISC-V processor.

14.4.2 *Transputer*

As an ongoing project, we have been looking at reviving the Transputer through SME. The Transputer [64] is a microprocessor architecture introduced in the 1980s. It is a stack machine, which focuses on concurrent execution and communication amongst multiple Transputers. It is tightly linked to Occam [65], which is a programming language utilizing the CSP programming model [66].

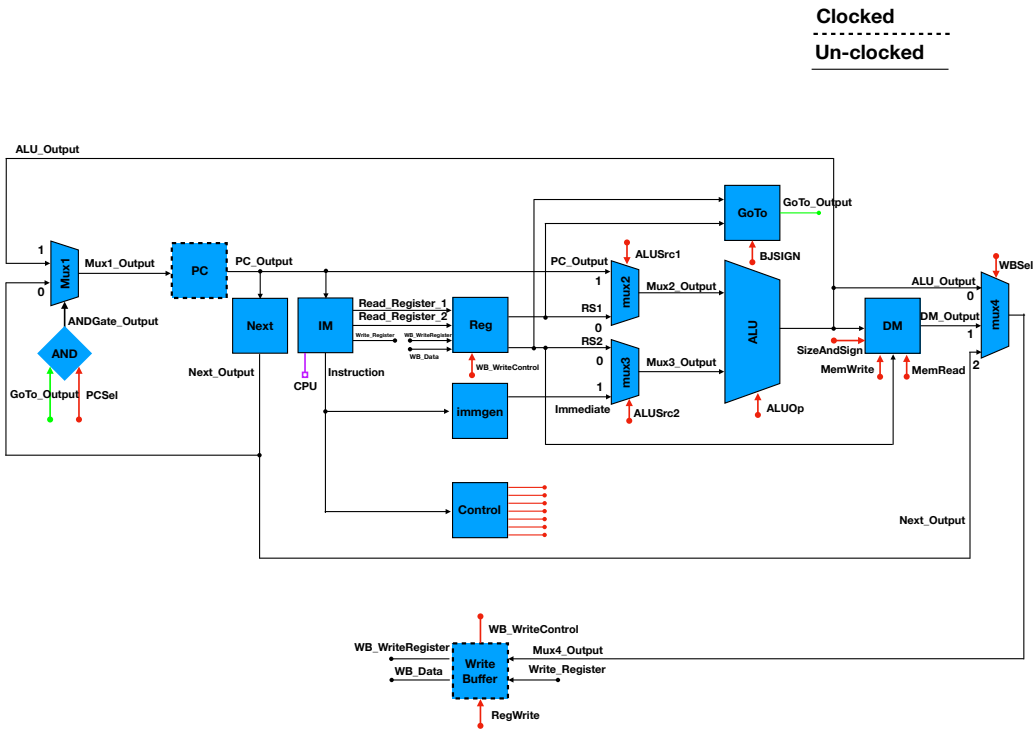


Figure 53: Block diagram of the RISC-V processor. Image from [61]

Implementation	LUTs	Clockrate
SME Transputer	8686	26.32 MHz
OpenTransputer	14744	41.00 MHz
T42	4000	100.00 MHz

Table 10: LUT utilization and frequency for the SME Transputer implemented on a Pynq, compared to similar projects. Values from [64]

The paper [64] showed an implemented, placed, and routed SME Transputer written in a short timeframe. It can run Transputer bytecode at suboptimal performance, which is shown in Table 10. However, optimizing performance should be low-hanging fruit, as we have not focused on performance in any manner but rather on correctness. The block diagram is shown in Figure 54.

14.4.3 *Occam to Go*

A side project, which sprung from the SME Transputer was the bachelor project by Matilde Broløs [67], which was later published as a paper [68]. The target was to ”revive” old Occam [65] programs by translating them to the modern CSP-based language Go [69].

The resulting translator showed that it was possible to translate Occam to Go at least for a subset. Table 11 shows the benchmarking numbers of the three programs *count*, *extended* and *commstime*. The table shows that the programs run slower but at decreased memory consumption.

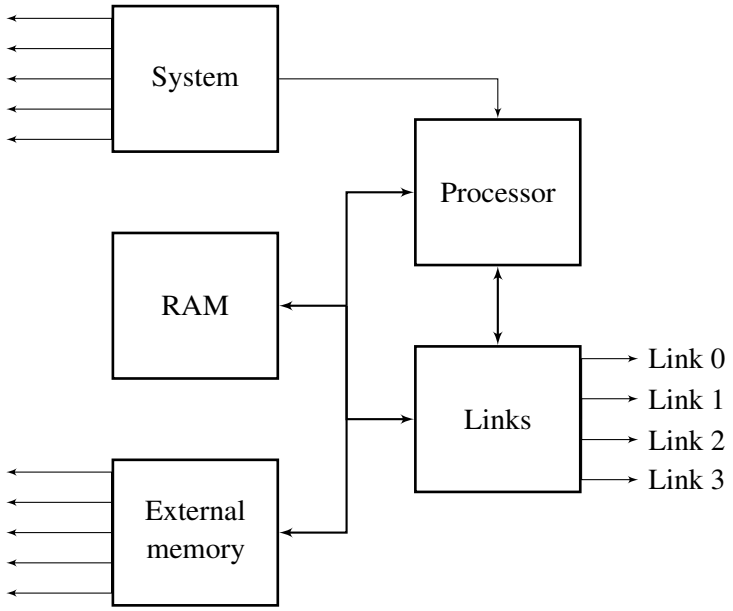


Figure 54: Block diagram of the SME Transputer. Image from [64].

Program	Translation (s)	Exec Occ (s)	Exec Go (s)	Mem Occ (MB)	Mem Go (MB)
count	0.030	0.003297	0.001636	2.944	1.820
extended	0.030	0.003494	0.001771	3.024	1.935
commstime	0.032	0.003528	0.002347	3.152	1.910

Table 11: Execution time of translating, running time and the memory consumption of running the three benchmark programs. Table from [68].

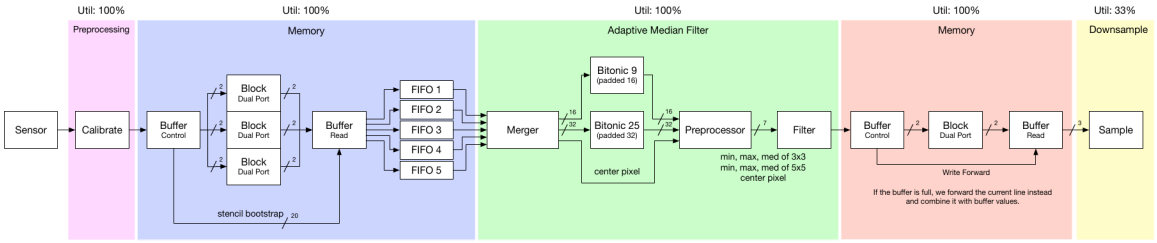


Figure 55: Final pipeline design for Troels' FPGA implementation. Image from [17]

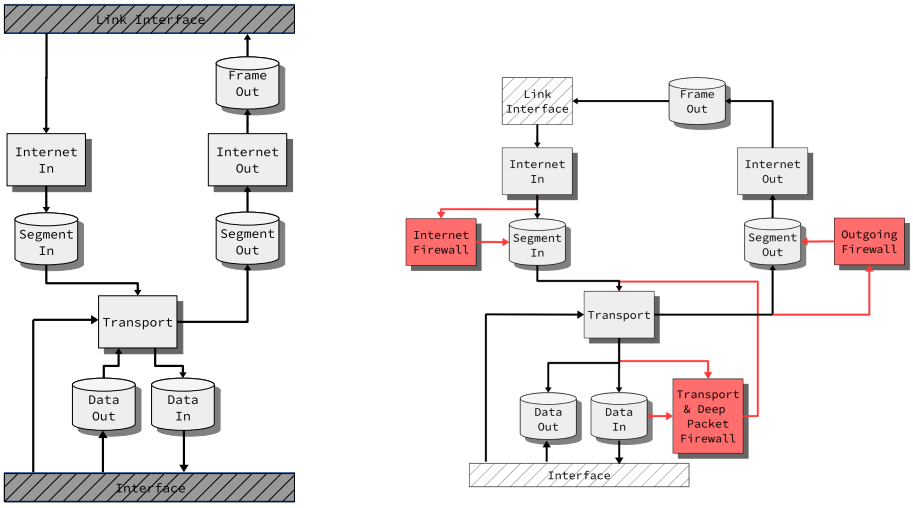


Figure 56: Final design for the TCP/IP implementation, alongside an overview of how the firewall project "connects" to the TCP/IP engine. Images from [70], [71]

14.4.4 *Image filtering*

The final part of the master's thesis by Troels Ynddal [17], focused on providing a high-throughput implementation of his image processing pipeline on FPGA through SME as part of his thesis. The hardware exploits he used were to do a bitonic sorting algorithm, which can be done efficiently on FPGA, used the dual-port nature of the BRAM, and finally a First In First Out (FIFO)-like buffer structure. For the Pynq board, the design clocked at 177 MHz at 10% utilization. This clock rate translates to a theoretical throughput of 14.2 Gb/s, short of the 40 Gb/s. However, if we moved to a larger board, he showed that the design could reach 187.5 Gb/s, far beyond the target. Figure 55 shows the overall block design.

14.4.5 *TCP/IP*

The thesis made by Jan Meznik and Mark Jan Jacobi [70] designed and implemented a Transmission Control Protocol / Internet Protocol (TCP/IP) networking protocol stack in hardware using SME. They successfully implemented the four layers of the Internet Protocol Suite model. Figure 56 shows the finalized design. While the design never hit actual hardware, they suggested that at 10 MHz, which is a reasonable target frequency, they should handle 80 Mb/s. While this is not exceptional, they provide suggestions for further improvements to their design, which should scale the performance multiple times.

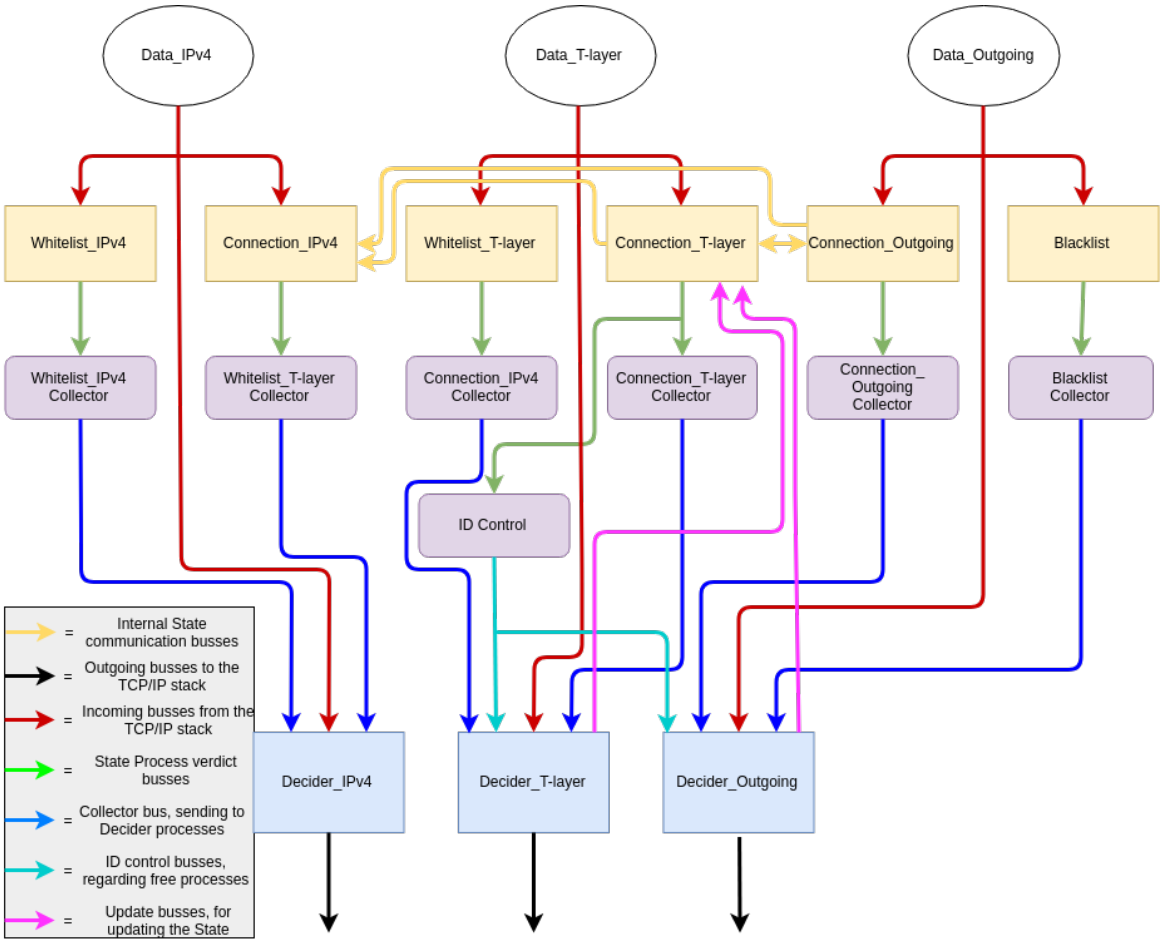


Figure 57: Block design of the firewall. Image from [71].

14.4.6 *Firewall*

In parallel with the TCP/IP thesis in Section 14.4.5, another thesis by Patrick Dyhrberg Sørensen and Emil Sander Bak titled ”High-Performance FPGA Firewall using SME” was made. They designed a reprogrammable stateful firewall that focused on speed and parallelization. While it did not target full integration, they still designed their implementation with the TCP/IP projects implementation in mind. Figure 56 shows their suggestion of how the two projects would integrate.

They were able to have a allow/blocklist and prevent certain types of Distributed Denial of Service (DDoS) attacks. Although the design never reached actual hardware, at least not before submission, they suggested that the design was stable and should lead to high throughput. Their biggest shortcoming, except for not reaching hardware, was that they did not get to do a deep packet inspection. They expected that at 25 MHz, they should hit 1 Gb/s, which further suggests 10 Gb/s at 250 MHz. Figure 57 shows their final design.

14.4.7 *Bohrium*

The thesis by Tor Skovsgaard [72] implemented a transpiler from vectorized code to FPGAs by using Bohrium [73] as the frontend and SME as backend. The resulting FPGA versions varied widely in

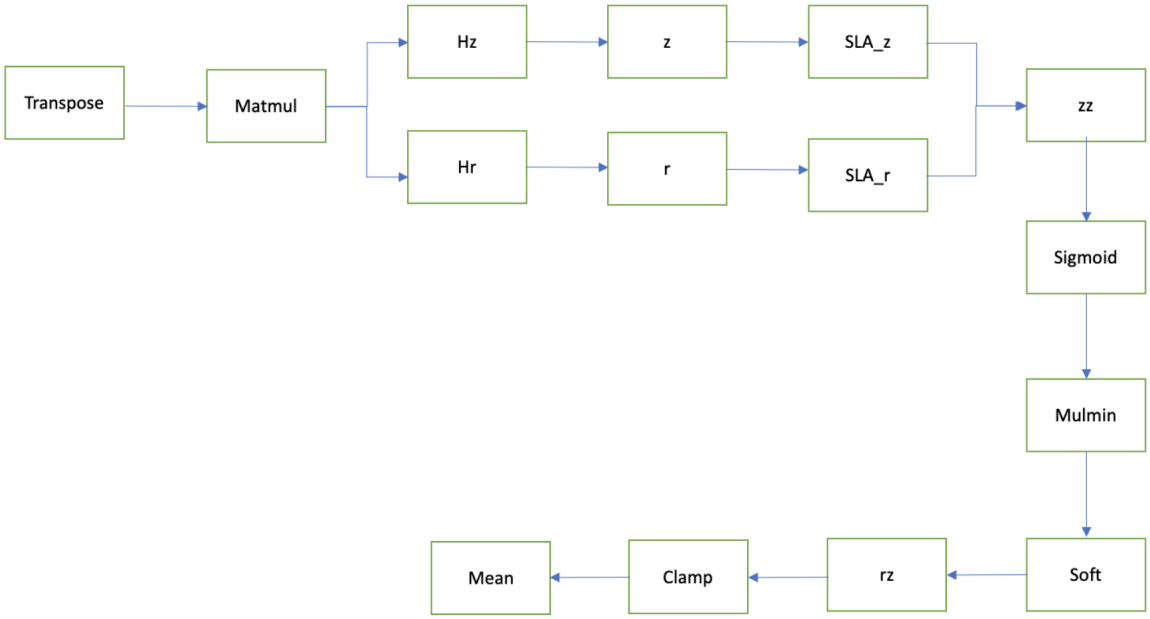


Figure 58: The block diagram of the overall feed-forward neural network. Image from [74].

performance but did outperform the CPU by one order of magnitude and power efficiency by two orders of magnitude.

14.4.8 *Machine Learning*

This thesis by Amira Moussa [74] translated a PyTorch [75] feed-forward neural network to SME in order to target FPGAs. The implementation exploited the PyTorch models' abstractions and implemented abstract computational blocks for solving each abstract operation. The FPGA implementation ran 21 times faster compared to the Python implementation, indicating FPGAs as a strong platform for ML inference. Figure 58 shows the block diagram for the final design. To further emphasize the scope of the project, one of the blocks, `Matmul`, is shown in Figure 59. As a final note, this project also shows SME usability since the student did not have a strong programming experience, and especially no experience with FPGAs.

14.4.9 *Lennard-Jones*

A project that ran in parallel with this dissertation was the project by Alberte Thegler. The work has been published in the paper "Accelerating Molecular Dynamics with the Lennard-Jones potential for FPGAs" [76] and is included in Appendix A.1.3. The project targeted FPGAs through SME and provided a multi-dimensional molecular dynamics simulation loop. Even though we have not fully optimized the SME network, the design hit a frequency of 13.603 MHz at a maximum of 28.87% resource utilization, targeting the XCKU5P board. While the clock rate sounds low, it translates into higher performance than the baseline Numpy implementation. Figure 60 shows the overall block diagram.

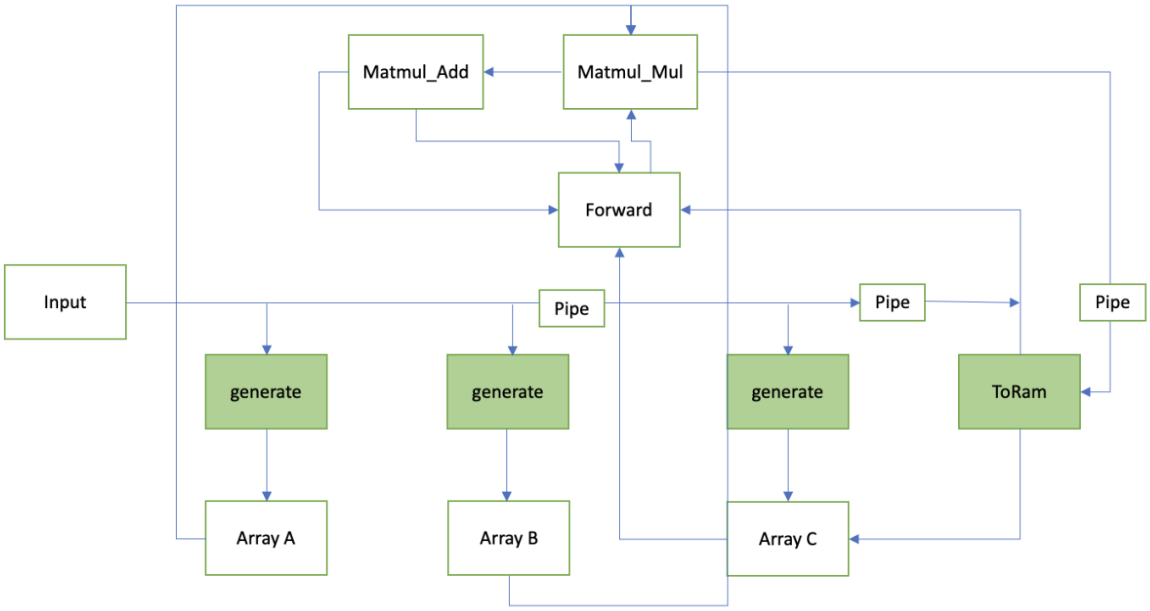


Figure 59: The block diagram of the internal structure of the `Matmul` block seen in Figure 58. Image from [74].

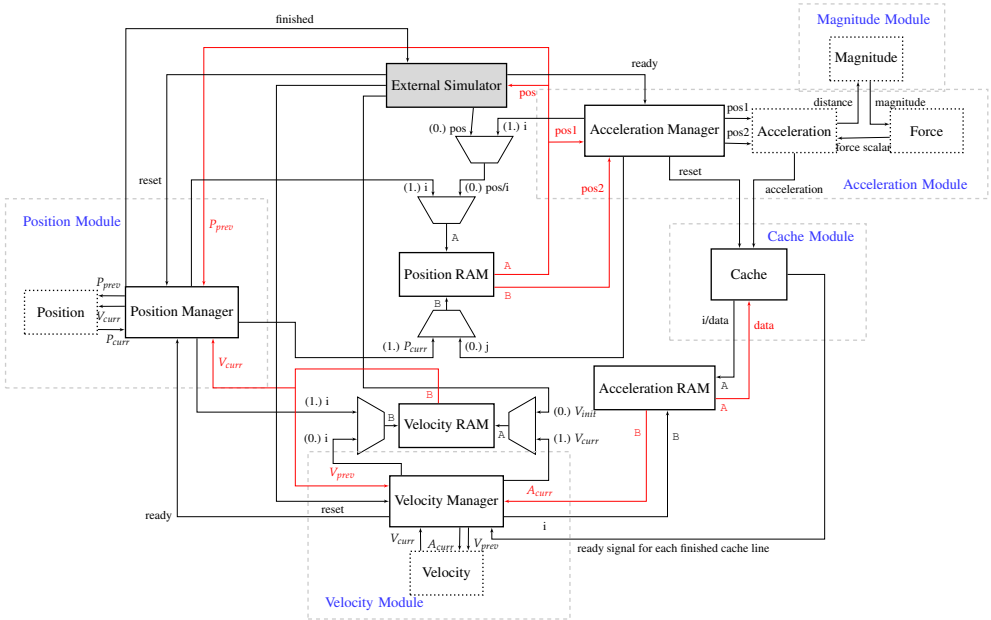


Figure 60: Block diagram of the SME network computing molecular dynamics simulation. Image from [76].

14.4.10 *Cryptography*

The bachelor thesis by Jacob Herbst and Jonas Flach Jensen implemented a cryptographic library in SME [77]. They implemented a library of four cryptographic functions: MD5, SHA256, AES, and ChaCha20. By pipelining their design, they beat some state-of-the-art CPU implementations on the Pynq while being competitive against their base implementation on the CPU.

15

SUBCONCLUSION

In this part, we have covered our research in reconfigurable computing. We established a baseline for machine architecture in Chapter 13, motivating the use of FPGAs over CPUs, GPUs, and ASICs. In Chapter 14 we presented two programming models, SME and DaCe, both of which being improved by our research. Finally, we presented the reconfigurable computing projects that have been spawned or run in parallel to this Ph.D. thesis.

Section 13.2 showed that CPUs are great at versatility, Section 13.3 showed that GPUs are great at handling parallel problems, Section 13.4 showed that ASICs are excellent at handling a specific task, and finally, Section 13.5 showed that FPGAs are between general-purpose and application-specific. We also covered the traditional way of FPGA programming: RTL and HLS, in Sections 13.5.1 and 13.5.2.

Then we described two alternative methods for FPGA programming: SME and DaCe, in Sections 14.1 and 14.2.

The SME abstraction lies just above the RTL abstraction, offering modern software development tools for hardware development. We have shown that many students could make meaningful hardware implementations in SME, suggesting SMEs place as an educational tool. We have presented all of the improvements to SME done as part of this Ph.D. thesis; dynamic constructs in Section 14.1.1.1, utilizing asynchronicity in Section 14.1.1.2, modern framework in Section 14.1.1.3, parallel execution Section 14.1.1.4, compilation over decompilation in Section 14.1.1.5, and custom processes in Section 14.1.1.6. All of these improvements further enable SME as a tool for hardware development.

The DaCe model is a versatile tool enabling domain experts to implement high-performance implementations without requiring them to be platform experts. We have made two significant improvements to the DaCe framework: allowing the developer to specify RTL tasklets for highly optimized inline regions in Section 14.2.1, and an automated approach to applying the multi-pumping optimization to any subdomain already targeting FPGAs in Section 14.2.2.2. Both of these improvements should further strengthen DaCe as a viable programming model in the field of High-Performance Computing (HPC) targeting FPGAs.

Part V

AUTOMATIC INSPECTION OF X-RAY IMAGES OF FOOD

16

ADAPTIVE X-RAY INSPECTION (AXIS)

The Adaptive Xray InSpection (AXIS) aims to improve the quality of the current food inspection by utilizing X-ray imaging to quantify the foods' internal structures and possible defects. The primary goal is to have a fully automated machine, which can keep up with the industry's massive amounts of food production. While models can be hand-built to detect one type of defect on one type of dataset, it is not a viable solution, given that the subjects differ over time, and the machine deteriorates, requiring semi-live adaptation capabilities, which does not correspond to a hand-built solution.

Instead, we want to leverage the growing field of Machine Learning (ML) to construct a resilient model, which requires little user intervention. We need the entire pipeline from data acquisition to decision-making to build this solution.

17

BUILDING THE FIRST DRAFT

This chapter will cover the first implementation of the program that the AXIS project should use. While the initial prototype machine featuring full automation will not demand high-throughput, we will still base our choices on this metric. We will reflect on the experiences gained in the previous parts of this dissertation to choose which methods we will be using. Following these choices, we will construct the base implementation in Python running on Central Processing Unit (CPU), as this will both give us a baseline to optimize against and will, in any case, indicate the feasibility of a high-throughput implementation.

17.1 THE DATA

Throughout the project, multiple variants of the X-ray setup have been proposed. The initial prototype was the ButeoX, which consisted of a conveyor belt, X-ray source, and line scanner, similar to the setup shown to the left in Figure 61. By stitching together these lines, we would obtain a ”regular” image, such as the one shown in Figure 12, which did require additional image correction to achieve a clean image like the one in Figure 13.

The later prototype consisted of a static scan area, X-ray source, and area scanner, packed together into a refrigerator-like structure, similar to the setup shown in the middle of Figure 61. In contrast to the line scanner, these images already came in the two-dimensional format but still required some corrections in order to remove the X-ray side-effects, such as seen in Figure 5.

The current prototype is a roller, X-ray source, and line scanner setup, similar to the setup shown to the right in Figure 61. The images captured by this setup require the same actions as the ones taken for the ButeoX. The images in this dataset are what Alina’s thesis [19] refer to as ”the new potato dataset,” with Figure 62 showing four samples from the dataset. Given that it looks like the final setup will follow this structure, we will focus our work on this dataset, although the techniques should be applicable for the two other proposed setups.

The most current dataset consists of 216 images labeled by the person capturing the data. In the dataset, 78 are ”good,” 85 are ”bad,” 5 are ”unknown,” and the remaining 48 are ”probably do not have a hole.” While this seems like a small dataset, we will show that this is sufficient for capturing the general structure.

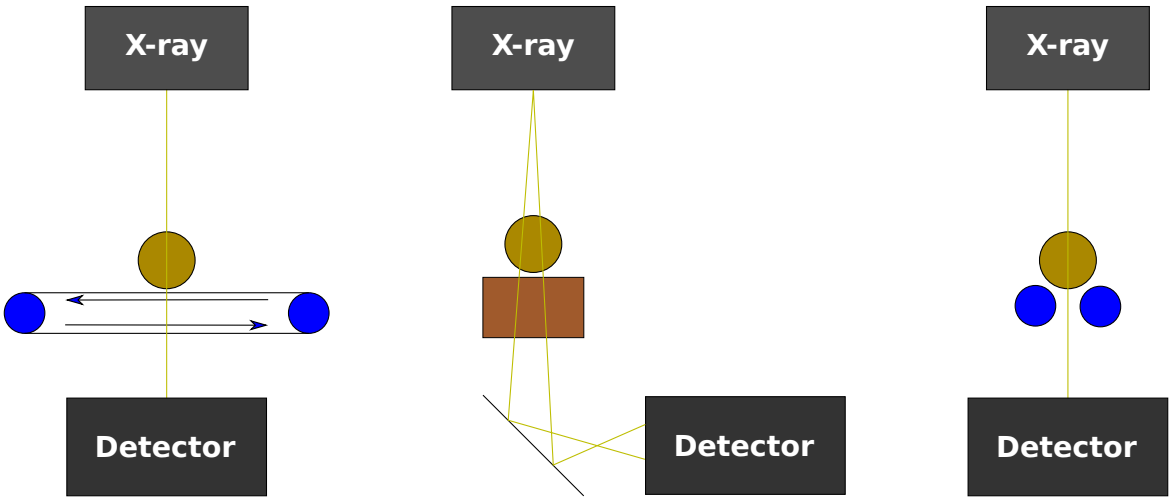


Figure 61: High-quality schematics of the different X-ray capturing setups. From the left, the ButeoX, the refrigerator-like, and the current line scanner setup.

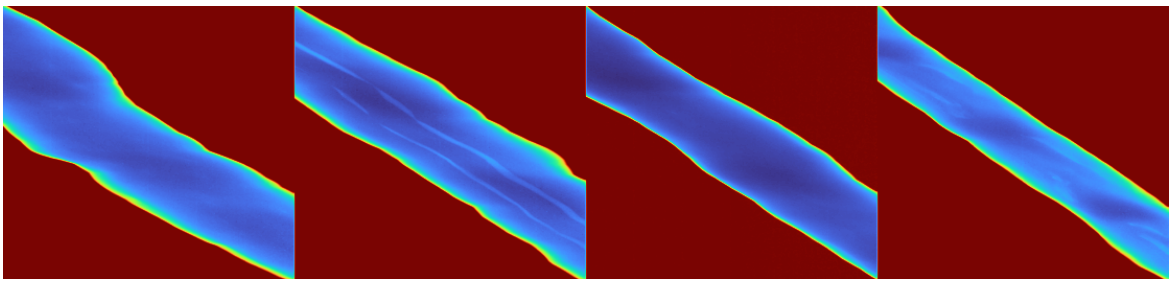


Figure 62: Four raw X-ray images captured by the new line scanner setup in Figure 61. From the left: a good sample, a sample with an internal hole, an ”unknown” sample, and a ”probably good” sample. Images provided by Newtec.

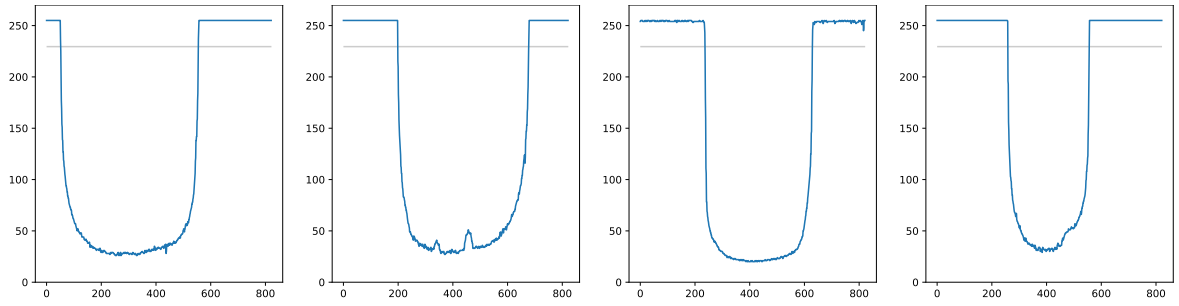


Figure 63: Lines extracted from the images in Figure 62. The gray horizontal line indicates 90% of the background value.

17.2 PREPROCESSING

As covered in Chapter 8, we have investigated different preprocessing techniques. First and foremost, as described in Section 5.2, we need to counter the effects that X-ray imaging introduces. We choose Flat-Field Correction (FFC), noise reduction through median filtering, and finally, a Region Of Interest (ROI) extraction method.

We introduce a new algorithm for the ROI since the images are not as ”natural” as the first dataset but seem more elongated and diagonal. Instead, we introduce the concept of *peeling*. This new method is motivated by two observations; Newtec already has a setup for inspecting the external faces of the potato using visual inspection, and the ML model we use has a hard time near the edges.

By programmatically *peeling* the potato, we primarily remove the edge of the potato and the background, which we do not care about, but we also reduce the images to be of a fixed size. We process the image line by line both because we do not need to consider the entire image, and it fits a setup that captures the images line by line. We can express the algorithm in two ways; the naive approach of removing anything above some threshold, such as 90% of the maximum value (the value that the background carries in X-ray imaging), or the advanced approach of finding the valley of the line, which should be the potato. Figure 63 shows a line along with a line depicting where the first approach would cut. We can even exploit parallelism since we need to find both the left and the right changepoint, and they should not depend on each other. Once we have found these two indices, we keep anything between the changepoints, discarding the rest.

Looking at Figure 62 we see that as the potato enters (top of the images) and exits (bottom of the images) the view, we capture a varying amount of pixels as being a potato. Because of this variation, the lines will have differing widths as we peel the potatoes. We propose two methods for dealing with this; either we interpolate the small lines upwards or remove all of the lines too far from the mean width and then interpolate to the minimum remaining width. We choose the first approach to ensure that we do not lose data, and the additional information we introduce should not be too intrusive, as it would follow a smooth pattern. After this method, we obtain square images like the one shown in Figure 64.

ML algorithms tend to favor data of a fixed size, even though some algorithms, like decision trees, can be resilient to missing data entries. We need to scale the images to be of the same size. Like with peeling, we can either scale up or down. As covered in Section 4.2 and shown in Section 5.4, we have a powerful method for resizing the images while retaining as much information as possible: the seam carving algorithm. Using this method, we achieve a better runtime without losing too much

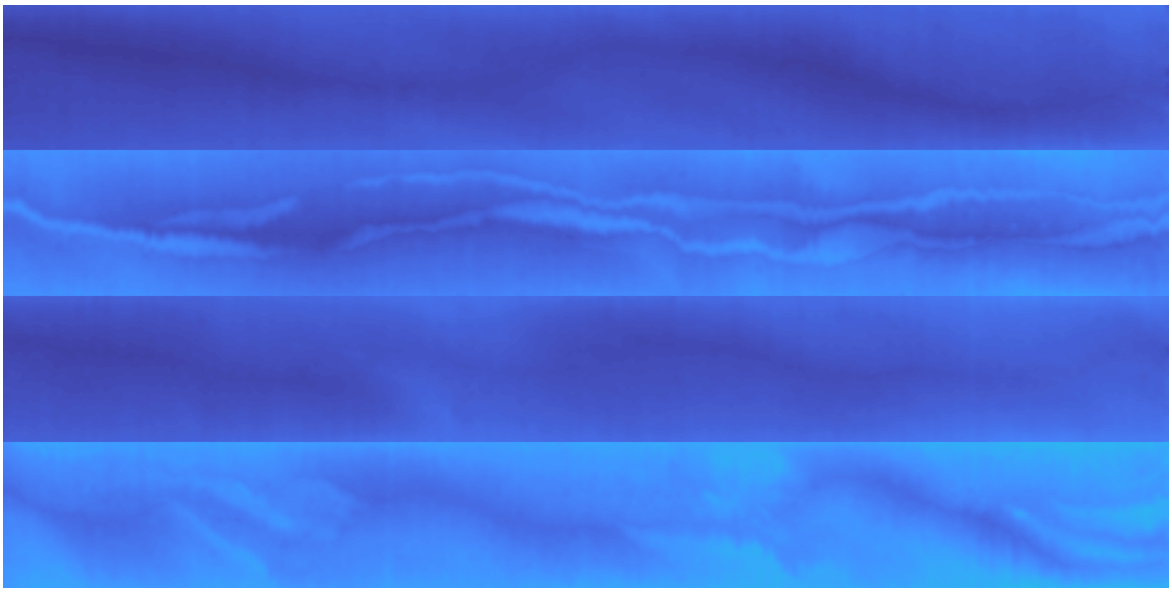


Figure 64: The images from Figure 62 after peeling and interpolation. The order is the same, but the images have been rotated.

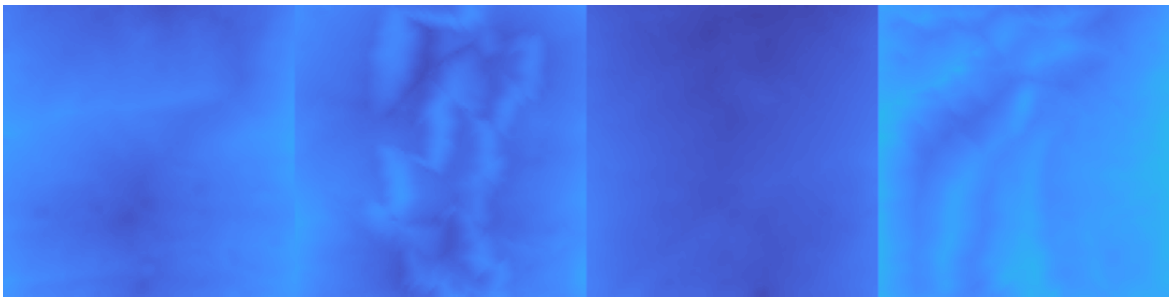


Figure 65: The images from Figure 64 after applying seam carving at aggressive levels.

information, and we get images of a fixed size. To tune which size to target is a hyperparameter and will be covered in Section 17.3. Figure 65 shows the image from Figure 64 after applying seam carving.

The final preprocessing step is to convert the values between 0-1, as neural network-based algorithms favor input in this range. Given that the images are in grayscale, we know that the values span the range of 0-255, so we can normalize relatively quickly by dividing by 255.

17.3 MACHINE LEARNING MODEL

While the preliminary results in Sections 9.1.1 to 9.1.3 using traditional supervised learning and Convolutional Neural Networks (CNNs) were quite good, we want an automated solution. Traditionally, we achieve full automation through unsupervised learning, where the ”data speaks for itself.” We adapt the same idea but draw from the knowledge we have and have in abundance - the good samples.

In foods, hazardous defects are rare and far in between. As such, if we used the traditional supervised learning approach, we would not be resilient to never-seen-before samples as the probability of the training dataset capturing all cases is relatively low. We do not necessarily need to distinguish between

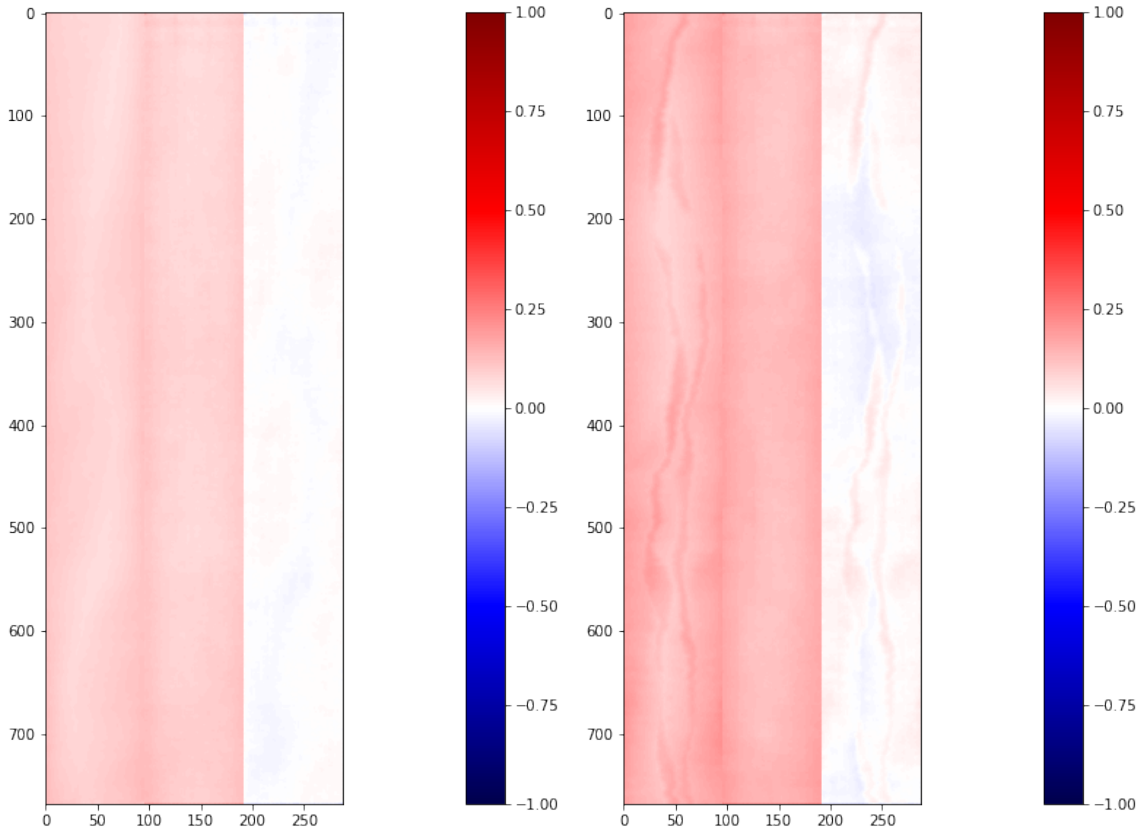


Figure 66: Input, reconstruction, and difference for **Left:** a good sample and **Right:** a bad sample.

the different defects but can limit the problem to knowing that a defect has occurred. This filtering alone can vastly reduce the dataset, allowing for heavier processing later in the pipeline for these samples carrying defects.

We propose to build upon the work carried out by the master project covered in Section 9.2.1. In summary, this project explored a semi-supervised approach, in which we trained an *oracle* model, which given an image, could produce an image of the *perfect* sample of the same shape. We carry out this semi-supervision by training a Convolutional Variational Auto Encoder (CVAE) using purely ”good” samples, giving us precisely this *oracle*.

17.4 DETECTING OUTLIERS

The final idea is: if we get an image from the oracle of what the sample should look like if it were a good sample, then we could compare it to the input, as too big of a difference would translate into it *not* being a good sample. Figure 66 shows the general intuition, where we see the error from reconstructing a good and a bad sample. While the master's thesis provided good comparison results for some samples, it did not achieve our desired performance for the ”new potato dataset.” Furthermore, we could see the difference between the input and oracle images, suggesting that the problem lied in the images' final comparison.

The thesis showed that Structural Similarity Index Measure (SSIM) provided the best results, looking at structure rather than a pixel-based strategy. Following that prior knowledge and intuition, we try

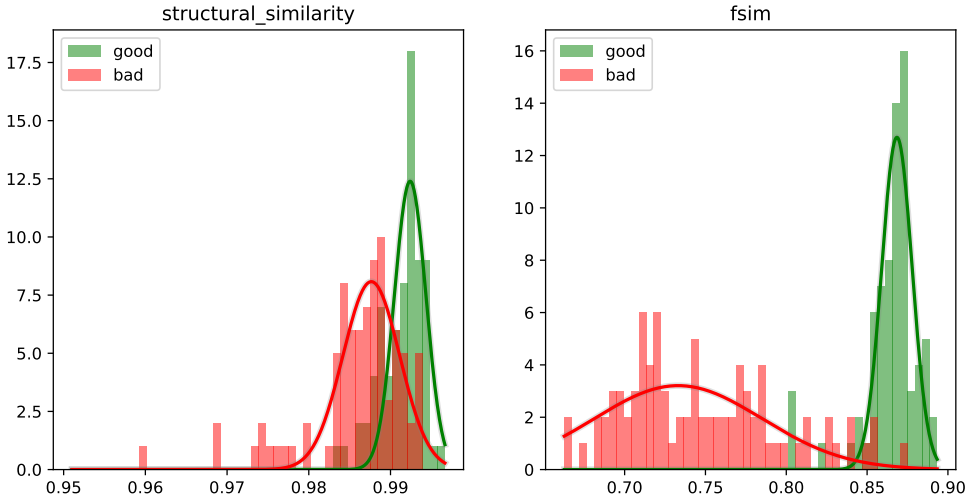


Figure 67: Histograms depicting how the samples are distributed when compared using SSIM (**Left**) and FSIM (**Right**).

the Feature-based Similarity Index Measure (FSIM) since it builds upon the ideas from SSIM but considers lower-level features. After training the CVAE and comparing the input and prediction using SSIM and FSIM, we obtain the distributions seen in Figure 67. Here we see that the two groups have less overlap and are now easier to separate, indicating that FSIM is a better metric, at least for these two classes. We further show this strong ability to separate the classes when looking at the Receiver Operator Characteristics (ROC) curve and the corresponding Area Under Curve (AUC) scores in Figure 68, which gives us a direct comparable metric for how well we can separate the two classes.

However, how can we be sure that the model has not just overfitted towards the seen training dataset? We can do this by using a test, or validation, set, which is ”kept” from the model during training. We are training purely on the ”good” samples, so we cannot extract images from the ”bad” dataset. After training, the test set is run through the model and compared using FSIM. If everything goes well, we should see the test-set land in the same distribution as the ”good” samples.

Additionally, as mentioned in Section 17.1, we have two additional classes: ”unknown” and ”probably good,” which we can also use. We expect them to land between ”good” and ”bad” for the ”unknown” and towards ”good” for the ”probably good.” Looking at the distributions in Figure 69, we see that the classes land as expected, further indicating the predictive power of our model.

We adopt the robustness metric usually used in hyperparameter optimization: k-fold cross-validation as a final convincing measure. This method is a tool for detecting whether a model has overfitted or has actual generalized predictive power. It works by splitting the training set into k subsets, or *folds*. Then k independent models are trained, where $k - 1$ of the folds are used for training, keeping one fold as the validation set. For each model, we choose a different fold as the validation set. By looking at the distributions and AUC scores in Figure 70, we see that the model is very robust for all folds.

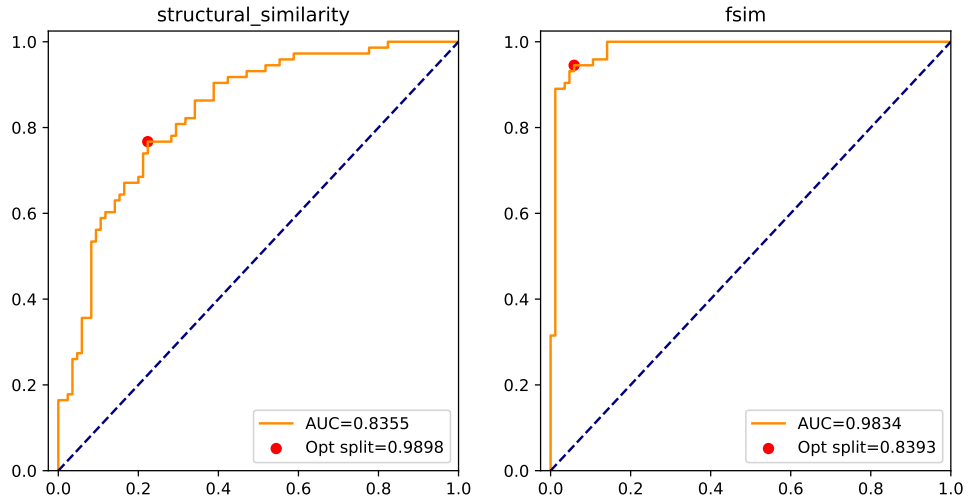


Figure 68: ROC curves and corresponding AUC score depicting how well we can seperate the two classes using SSIM (**Left**) and FSIM (**Right**).

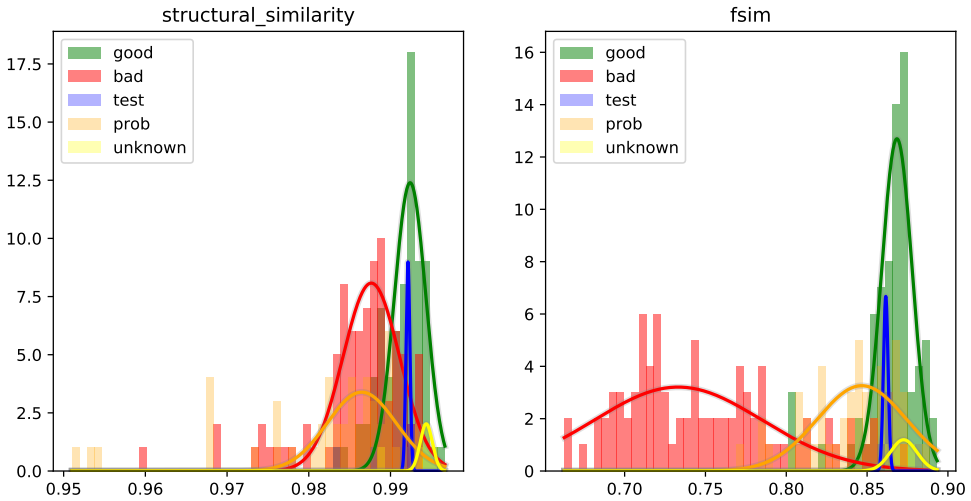


Figure 69: Histogram showing the distribution of the four classes using FSIM.

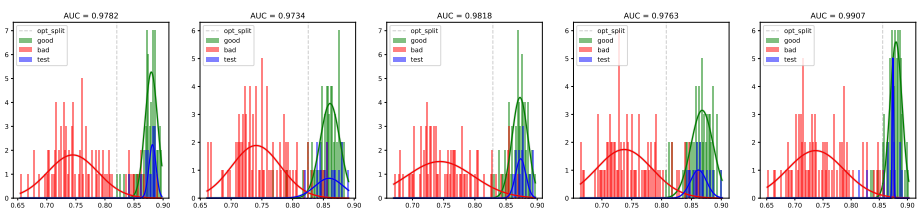


Figure 70: Distributions of a 5-fold cross validation using the CVAE and FSIM. Above each distribution, there is the corresponding AUC score.

18

OPTIMIZING FOR FPGA

As presented, our solution has seven steps: FFC, median filtering, peeling, seam carving, normalization, inference, and FSIM. Each step will become a hardware kernel, focusing on its particular step. As Field-Programmable Gate Arrays (FPGAs) favor deep pipelines, each step will be pipelined, keeping up with the entire pipeline of steps. Based on our findings in Part iv, we will adhere to the Advanced Microcontroller Bus Architecture (AMBA) streaming Advanced eXtensible Interface (AXI) protocol, as this allows us to integrate seamlessly with the outside world.

18.1 FLAT-FIELD CORRECTION (FFC)

This step corrects the flaws of X-ray imaging by correcting the images using calibration images with the X-ray source turned on and off. Section 5.2 describes the process more in-depth, but is summarized in Equation (3). Looking at the equation, we observe that the only changing variable is P, with D and F remaining static over the acquisition of multiple images. As such, we can precompute the bottom part of the fraction into a new variable C, giving us the equation:

$$N = \frac{P - D}{C} \tag{18}$$

The data we need to hold during a run consists of the dark image, D, and the calibration image, C. While this seems excessive to keep two rather large images in the local storage of the FPGA, we make another observation: a line scanner captures one line of the image at a time, so we only need to keep a single line of each calibration image for correcting, as the errors are consistent for each line captured. This effect is prominently shown as stripes in the image in Figure 12. Note that this only works for line scanner setups, as the area scanner is prone to the halo effect shown in Figure 5, requiring us to have the entire calibration image. As a single line consists of 768 pixels, we need to hold $768 * 2 * 8 = 12288$ bits, which can easily fit within a single 36k Block Random-Access Memory (RAM) (BRAM), and we can read both calibration lines in parallel since the BRAM has two independent ports. Figure 71 shows the block diagram of FFC on FPGA.

18.2 MEDIAN FILTER

This step removes strong outliers, such as salt and pepper noise, and other outliers, both of which X-ray imaging commonly contains. In Section 14.4.4 we saw an FPGA implementation for an Adaptive

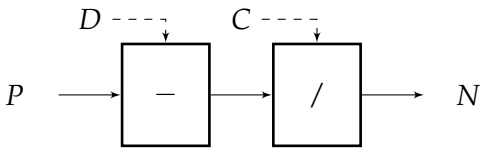


Figure 71: Block diagram of FFC. The solid arrows indicate streaming AXI, with the dashed being constant values.

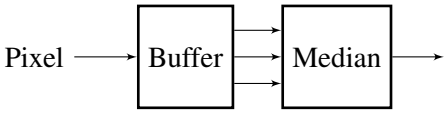


Figure 72: Block diagram showing the median filter.

Median Filter (AMF) using filters of size 3 and 5. While this implementation showed promising results for application on images, we do not need to work with the entire image but can, like in Section 18.1, work on a single line. To find the median, we have to sort the pixels, to which we leverage the bitonic sort proposed by Troels [17], as this method functions well in a pipelined FPGA setting. Given that we are working with a single line, we do not need as big networks as the original proposal but instead need a smaller network sorting 3 and 5 values.

For the first implementation, we disregard the adaptive part of the AMF and instead focus on a median filter of size, k of 3. The overall method becomes simple: sort the incoming values, forwarding the median. We are only interested in the parts where the subject lies in the center of the line, and since we are peeling afterward, we do not need to consider padding but can reduce the line width by $k - 1$ as we apply the filter. Figure 72 shows the block diagram for the median filtering.

18.3 PEELING

This step removes the background and part of the edge, and we can view it as a ROI algorithm. Again we exploit the fact that the images are captured line by line, as we only need to consider a single line. We buffer the line as it arrives and only forward values once we have found the changepoint of the curve. We find the changepoint by looking at the past five values, and if the value range exceeds some threshold, we start forwarding. Given that we have applied the median filter, there should not be any strong outliers except for the changepoint introduced by the potato.

The streaming AXI protocol specifies signals for conveying the completion of a burst of transactions through the `tlast` signal. If this signal is asserted alongside a transaction, indicating that it was the last in the burst. We utilize this signal to indicate that the following steps have gathered a complete line post peeling. Figure 73 shows the block diagram for peeling.

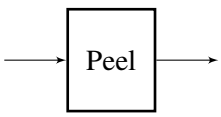


Figure 73: Block diagram of the peeling process.

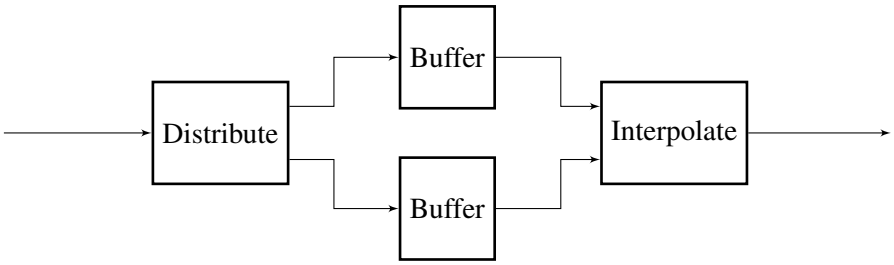


Figure 74: Block diagram for resizing one line of an image. The strategy is equivalent for both directions, assuming the data is presented in correct ordering.

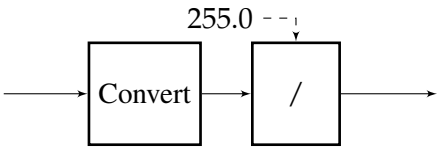


Figure 75: Block diagram for normalization.

18.4 IMAGE RESIZING

We now have a collection of varying count and sized lines, so we need to resize them to a square size. As mentioned in Section 17.2, we can interpolate the lines to have a fixed size. We apply this to the individual lines during acquisition as we already know the network's input size based on its architecture.

While seam carving shows promising results, it is computationally heavy, which is why we use standard image interpolation as our resizing technique for the initial implementation. Image interpolation requires that we know the dimensions of the input and output images, so we have to gather the entire image. When we have gathered the entire image, we have the size of the dimension to compute the scaling factor. Figure 74 shows the block diagram for resizing one dimension.

However, we need to gather the entire image for the other dimension. As the sizes of the images are too big to hold in BRAM, we have to utilize the off-chip Double Data Rate (DDR) RAM. Off-chip memory is not a problem, as long as we can keep the bus saturated to keep a high transmission rate. As we have mentioned in Section 14.2.2, we should align our transactions to 512 bits to fully utilize the bus, which is why the total size of the image should be a multiple of 512. We apply the same `tlast` signal as in Section 18.3 to indicate when we have fully gathered an image.

18.5 NORMALIZATION

While we store the image as 8-bit integers to save space, neural networks favor values in the range 0-1, known as normalization. Normalization is a trivial operation in our case, given that the range of 8-bit integers is 0-255. We can divide the values by 255 to bring the range down. The process becomes: convert the value to floating-point followed by a floating-point division of 255. We perform this step right before passing the images to the ML model. Figure 75 shows the block diagram for normalization.

18.6 INFERENCE

In order for us to perform inference on the FPGA, we need to translate the ML model to FPGA. Following the work in Section 14.4.8, we divide the network into individual types of operation. The CVAE architecture proposed in Section 9.2.1 consists of 6 operations:

- Conv2D LeakyReLU
- Dense LeakyReLU
- Dense Linear
- sample z
- Dense Tanh
- TransConv2D ReLU
- TransConv2D Sigmoid

Each step can be performed in a pipelined fashion, each feeding into the next. During inference, there is no training, and as such, no backpropagation or backward dependencies. All of the steps benefit from the `tlast` signal, as it indicates the finalization of an image. We do not need to keep an external controller managing the entire network with that signal.

18.6.1 *Activation functions*

All of the activation functions have to be implemented with a low iteration interval, support streaming AXI interfaces, and not hold a state. These requirements are easy to meet, as the functions are simple:

$$LeakyReLU(x) = \max(0,x) + -\alpha * min(0,x) Linear(x) \qquad = x \tag{19}$$

$$Tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{20}$$

$$ReLU(x) = \max(0,x) \tag{21}$$

$$Sigmoid(x) = \frac{1}{1 + \exp(-x)} \tag{22}$$

Figure 76 shows the block diagrams for the activation functions. Each block itself is simple, containing only protocol logic for streaming AXI and their single operation.

18.6.2 *2-dimensional convolution*

As previously covered, convolution is applying a filter to an image. Where Section 14.4.8 shows each stage holding their intermediate values, convolution shares the same problem with resizing: we do not have enough internal memory to hold the entire image. In a ML setting, the convolution layer applies a set amount of filters, whose output is then accumulated and sent through an activation function. So we do not need the entire image; we only need to buffer enough data to apply the convolution

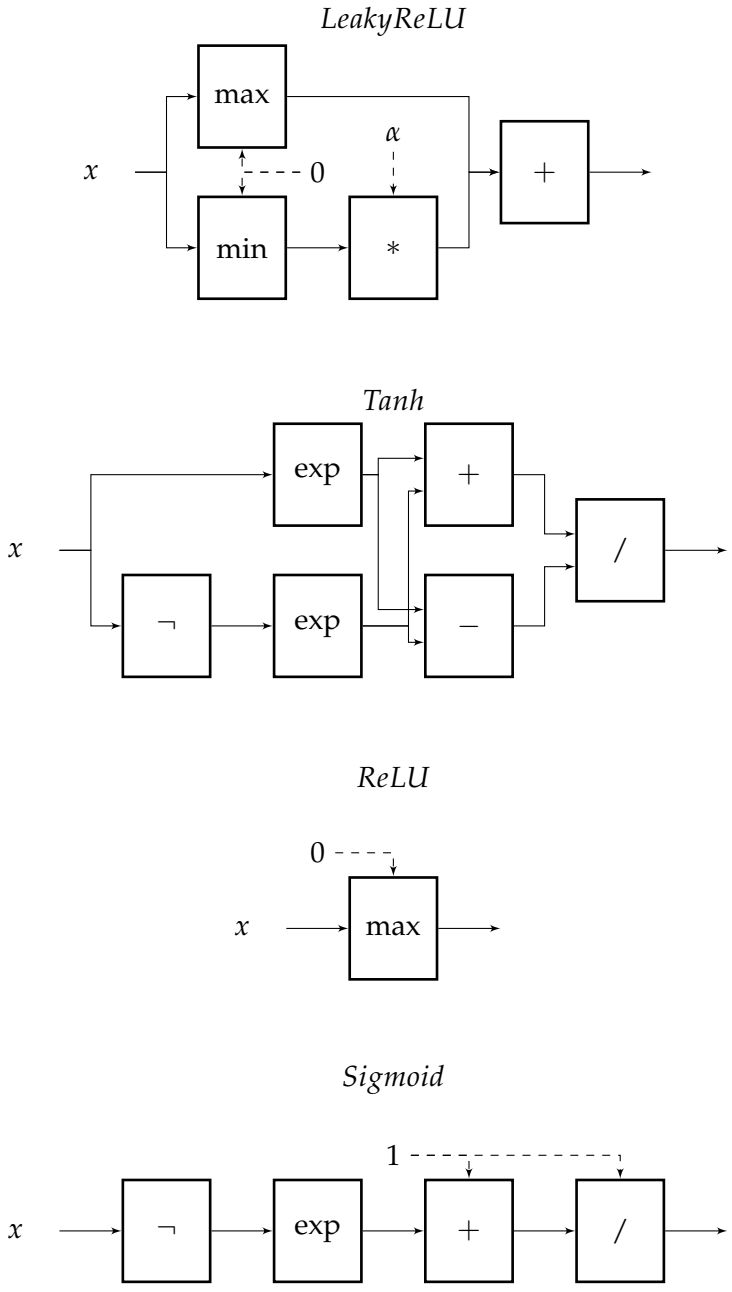


Figure 76: Block diagrams for the implementation of the activation functions. The solid arrows are streaming AXI buses. The dashed lines are constants.

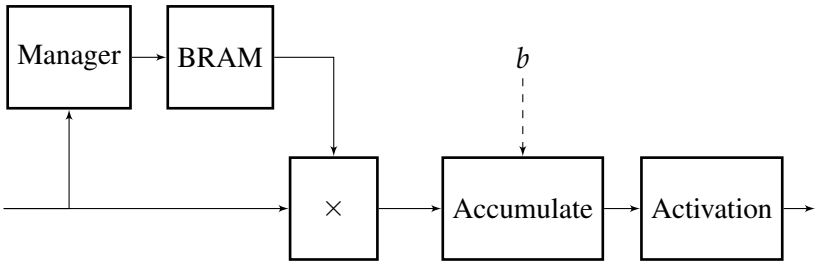


Figure 77: Block diagram for a two-dimensional convolution.

operation, which for a filter of size k constitutes as $k \times k$, with a single value as output. We thus need to buffer $k \times k$ from the previous state between each convolution layer. Furthermore, we do not overlap with perfect striding s (where $s = k$), so we can stream the images through the network only once, without keeping intermediate steps or rebroadcasting values. Without perfect striding, we need to buffer an additional $k - s$ lines to keep streaming the memory from memory only once.

To only use $k \times k$ buffer space, we need to read k columns from k rows in memory, assuming perfect striding. While this would be optimal for maintaining a high iteration interval, this strategy does not go well with external memory, as they require sequential access to saturate the memory bus properly. As stated in Section 14.2.2, Xilinx recommends transactions of 512 bits [48], corresponding to 16 32-bit floating-point values. For a filter of size k, we need to buffer k rows to apply the filter, so to keep up with the external memory, we need to buffer $t \times k$ of the image. Regardless of striding, we can hide this latency by deploying a circular buffer, similar to the double buffer we utilized in our molecular dynamics paper [76], in which we fill one end of the buffer while consuming from the other. As long as the buffer is large enough, we do not need to stall. Computing each filter can be done in parallel, based on the same data. After computing a pixel, we accumulate them into a single value, passing them to the corresponding activation function. Figure 77 shows the block diagram for the convolution operation.

The other state that we need is the stored weights for the filters in the convolution layer, the dense layer, and the transposed convolution layer. For the largest layers with 16 filters, each set of filters is of size 3×3 with 32-bit floating-point values, we need to hold $16 \times 3 \times 3 \times 32 = 4608$ bits. As such, we can feed two sets of filter weights into a single 36k BRAM. We could even fit four sets into one, but the number of ports is limited. However, we should be able to leverage double-pumping, as covered in Section 14.2.2, effectively gaining four ports.

18.6.3 *Dense layer*

The dense layer constitutes a fully connected layer The dense computation is straightforward; we read one value from the input, multiply it by the corresponding weights, accumulate, and pass it to the activation function. For the dense layer of size d, we need to keep d weights per input i. For our suggested architecture, $d = 200$ and $i = 1024$, giving $200 \times 1024 \times 32 = 6553600$ bits of storage, requiring 183 BRAM. While this would enable high throughput, as we gain 366 parallel ports, it is far too excessive, suggesting that we should look into exotic data types, such as block floating point, where multiple values share the same mantissa, as previous work shows a reduction of the memory footprint by significant amounts, while retaining high accuracy [78].

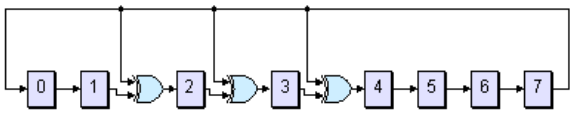


Figure 78: Overview of an LFSR implementation. Image from [79].

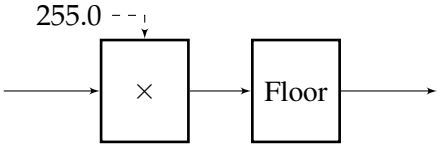


Figure 79: Block diagram for denormalization.

18.6.4 *Sample z*

As a non-standard part of an Artificial Neural Network (ANN), the CVAE feature a sampling layer that draws values from a random distribution. Unlike in the CPU and Graphics Processing Unit (GPU), the FPGA does not feature a library for generating random numbers, leading us to implement a pseudo-random generator. One way is the Linear-Feedback Shift Register (LFSR), where we implement a shift register, whose next input bit is a function of the previous number, usually generated through eXclusive OR (XOR) gates. Figure 78 shows the outline of how we efficiently would implement this.

18.6.5 *Transposed 2-dimensional convolution*

Compared to the regular convolution, the transposed convolution is much simpler, as we do not need to buffer any intermediate values; we multiply the inputs with the weights in the filters, gaining a new larger image. As such, the only buffering we need is into one transaction to external memory, as we still cannot fit the entire image into the internal FPGA memory.

18.6.6 *Denormalization*

The final step of inference is to convert the image back into the range of 0-255, as the FSIM algorithm works better with values in this range. As the resulting image should be normalized between 0 and 255, we must multiply the final pixel value with 255 to ”denormalize” back into the original domain. Figure 79 shows the block diagram for denormalization.

18.7 FEATURE-BASED SIMILARITY INDEX MEASURE

For the FSIM algorithm, we need three parts: phase congruency PC, an edge detection method expressing the image in the frequency domain through Fast Fourier Transforms (FFTs), similarity S, a metric quantifying the similarity between the features of two images, and gradient magnitude G, the image derivatives suggested as Scharr through the FSIM paper [80]. The algorithm then becomes:

- Compute the phase congruencies of the two images as PC_0 and PC_2.

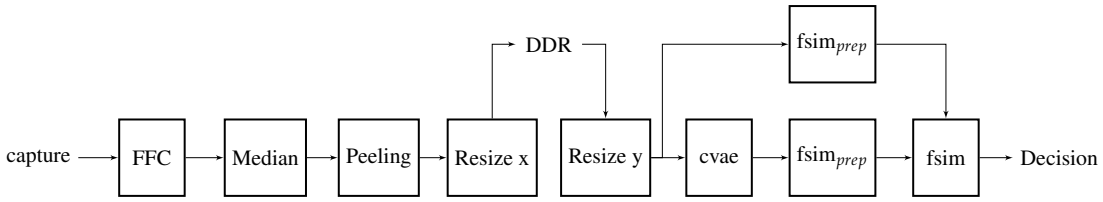


Figure 80: Overview of the components of the final solution connect.

- Compute the similarity between the two phase congruencies as S_{PC}.
- Compute the gradient magnitude of the two images as G_0 and G_1.
- Compute the similarity between the two gradient magnitudes as S_G.
- Compute the similarity between the two similarities S_{PC} and S_G as S_l.
- Compute the maximum value of S_0 and S_1 as PC_m.
- FSIM is then returned as:

$$\frac{\sum(S_l * PC_m)}{\sum PC_m}$$

While this seems like many computations, many parts can run in parallel, and we can do it using the streaming strategy, requiring little to no intermediate computations.

18.8 FINAL DESIGN

By combining all of the individual parts from this chapter, we can build the final solution, as Figure 80 shows. Each step is a pipeline, which means that as we fill the pipeline, we can perform many of the steps in parallel, retaining throughput. While we have not implemented this solution, the architecture suggests low iteration intervals, which should translate into a model that is at least as fast as data acquisition.

19

SUBCONCLUSION

In this part, we have presented our solution to the software part of the AXIS project. Chapter 17 presented our baseline implementation based on our findings in Parts ii and iii. We then transformed the baseline an FPGA-based solution, based on our findings in Part iv.

We have presented the preprocessing steps the models use in Section 17.2: FFC, median filtering, peeling, seam carving, and normalization. With the prepared data, we trained a CVAE in Section 17.3, with the architecture proposed in Section 9.2.1. Finally, we exchange the originally proposed image similarity measure, SSIM, with a new measure, FSIM, showing excellent performance concerning distinguishing between ”good” and ”bad” samples. We are consistently training models that reach AUC scores of 0.97-0.99, indicating that this is a robust model. This model is very desirable since it can be trained purely on ”good” samples, relaxing the otherwise required ”pile of data” imposed by ML models.

We have outlined an FPGA-based solution for the AXIS project, covering considerations along the way. The preprocessing in Sections 18.1 to 18.3 have been constructed with the line scanner in mind, suggesting that they can be performed on an isolated line, keeping up with the throughput of the data acquisition. Section 18.4 also keeps up with this strategy for the width of the image but has to gather the entire image before being able to scale the height of the image. Section 18.6 presents the needed functionality, in order to translate the ML model to an FPGA. However, we have argued that while the CVAE is a heavy ML model, it seems feasible to implement this model on an FPGA. Finally, we have outlined the steps to implement FSIM in Section 18.7, which follows the streaming nature of the otherwise proposed model, suggesting it should keep up with the throughput.

Part VI

WRAP-UP

20

FUTURE WORK

The most obvious first further step is to implement the model proposed in Chapter 18 to actual Field-Programmable Gate Array (FPGA) hardware, as we would be able to argue about the performance. Following this optimization, we could apply the multi-pumping optimization presented in Section 14.2.2, as this would counter the resource consumption issues we predicted in Chapter 18

Our final model proposed using regular interpolation image resizing methods, which are simple to implement, resulting in a stable-throughput implementation. In this thesis, we have investigated *seam carving* as a resizing technique for retaining prominent features while reducing the image dimensions. It would have been interesting to investigate the effects of presenting seam carved images to the final Machine Learning (ML) model.

In Part v, we presented the problems of translating the Convolutional Variational Auto Encoder (CVAE) model to FPGA due to excessive resource consumption. A popular technique in recent years for ML inference is the use of quantization, where we reduce the precision of the internal datatypes of the network, reducing both the required amount of computing and memory resources.

In its current form, translating the ML model to FPGA is a very manual process, guided by the findings of the master's thesis by Amira Moussa [74]. Given that we have to construct multiple architectures for different food products and target hardware platforms, it would be beneficial to automate this process, reducing the required amount of user interaction. One approach could be through the Open Neural Network eXchange (ONNX) [81] interface, as the translation parts should not be too different from the parts in the ONNX framework. There are multiple entry points for doing this, such as Data-Centric parallel programming (DaCe) or Xilinx Vitis AI [82].

The current implementation of the CVAE as proposed by the master's thesis by Alina Sode [19] in Section 9.2.1 uses a loss function, to which 50% are the Mean Squared Error (MSE) of the reconstructed image. It would be interesting to try other loss functions or combinations, as the resulting model might be better at producing samples or converge faster during training.

21

CONCLUSION

Throughout this thesis, we have covered the process of X-ray-based food inspection using a semi-supervised ML model from data acquisition to the final decision.

We investigated the physics behind X-ray imaging to be able to counter the effects using computer vision, obtaining clean images ready for ML training. While we cannot directly control the physics, knowing what occurs helps build an intuition for why the corrections are required and why they work.

Building on the foundation of previous projects, we explored multiple Artificial Neural Network (ANN) architectures. While the regular Convolutional Neural Networks (CNNs) shows promising results, it is a fully supervised approach, requiring our dataset to represent the entire sample space properly to gain good results. Defects are rare and far apart in the food industry, leaving us with little to no probability of adequately representing the entire sample space.

In this thesis, we propose utilizing a CNN variant, the CVAE; a semi-supervised approach, training purely on ”good” samples, of which we can gather a sizeable amount of samples. By being semi-supervised, we gain a potent tool for classifying whether a sample is considered ”good” without having to know the entire sample space prematurely.

As the food industry imposes high-throughput requirements, we have investigated the feasibility of exploiting custom-built hardware solutions utilizing reconfigurable hardware. As part of this process, we have contributed to the field of programming models targeting hardware design through improvements to the Synchronous Message Exchange (SME) and DaCe programming models, seeking to leverage multiple levels of abstraction. Among the contributions, we propose an automated approach to the hardware optimization multi-pumping, exploiting underutilized sub-components to reduce overall area consumption.

While we do not have a fully implemented solution, we have suggested the architecture of a translation of the preprocessing steps, ML model, and image similarity measure to FPGAs to gain a stable-throughput model that should be able to keep up with the high-throughput demands of the food industry.

Part VII

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. Prendergast and D. Weber, *Strawberry needle scare: Growers to look to metal detectors to contain contamination crisis*, http://web.archive.org/web/20200321232242/https://www.abc.net.au/news/2018-09-17/wa-drawn-into-strawberry-contamination-scare/10254746, Australian Broadcasting Corporation, [Accessed online: 13th December 2021], 2018.

[2] J. Diehl, *Safety of irradiated foods*. New York: Marcel Dekker, 1995, ISBN: 9780824793449.

[3] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[4] NVIDIA, *Nvidia tensor cores*, https://www.nvidia.com/en-us/data-center/tensor-cores/, [Accessed online; 13th December 2021].

[5] Google, *Cloud tpu*, https://cloud.google.com/tpu, [Accessed online; 13th December 2021].

[6] Apple, *Apple unleashes m1*, https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/, [Accessed online: 13th December 2021].

[7] C.-J. Johnsen, *Carl johnsen's github page*, https://github.com/carljohnsen, [Accessed online: 13th December 2021].

[8] *Medical Imaging Systems An Introductory Guide* (Image Processing, Computer Vision, Pattern Recognition, and Graphics ; 11111), eng, 1st ed. 2018. Cham: Springer International Publishing, 2018, ISBN: 3-319-96520-4.

[9] Spellman, *Xrbhr monoblock integrated x-ray source*, https://www.spellmanhv.com/en/high-voltage-power-supplies/XRBHR, [Accessed online: 13th December 2021], 2018.

[10] Matmatch, *Aisi 440c hardened*, https://matmatch.com/materials/mitf943-aisi-440c-hardened, [Accessed online: 13th December 2021].

[11] ChemistryIsLife, *The chemistry of grass*, https://www.chemistryislife.com/the-chemistry-of-grass, [Accessed online: 13th December 2021].

[12] Wikipedia, *Hay - chemical composition of hay*, https://en.wikipedia.org/wiki/Hay#Chemical_composition_of_hay, [Accessed online: 13th December 2021].

[13] S. Seltzer, *Xcom-photon cross sections database, nist standard reference database 8*, en, [Accessed online; 13th December 2021], 1987. DOI: 10.18434/T48G6X. [Online]. Available: http://www.nist.gov/pml/data/xcom/index.cfm.

[14] A. Topic, *Automating classification in food inspection*, eng, 2019.

[15] S. Nyrup, *Applied super resolution for x-ray imaging - virtual potatoes and how to x-ray them*, eng, 2020.

[16] T. Köhler, *Multi-frame super-resolution reconstruction with applications to medical imaging*, 2018. arXiv: 1812.09375 [cs.CV].

[17] T. Ynddal, *High throughput image processing in x-ray imaging*, eng, 2019.

[18] A. Topic, C.-J. Johnsen, and B. Vinter, "Extreme downsampling using content-aware seam carving for very low resolution foreign object preservation," submitted.

[19] A. H. Sode, *Automatic detection of foreign objects in x-ray images*, eng, 2021.

[20] S. Langkilde, *Training a neural network to distinguish between potatoes with or without a hollow heart*, eng, 2021.

[21] J. Patterson, *Deep learning : a practitioner's approach*. Sebastopol, CA: O'Reilly, 2017, ISBN: 1491914254.

[22] Y. Ren, J. Yang, Q. Zhang, and Z. Guo, "Multi-feature fusion with convolutional neural network for ship classification in optical images," *Applied Sciences*, vol. 9, no. 20, 2019, ISSN: 2076-3417. [Online]. Available: https://www.mdpi.com/2076-3417/9/20/4209.

[23] T. Louring Koch, *Online inspection of x-ray images: Detecting hollow hearts and needles in potatoes*, eng, 2017.

[24] J. R. Pedersen, *Foreign object detection in x-ray images using machine learning*, eng, 2020.

[25] L. Weng, "From autoencoder to beta-vae," *lilianweng.github.io/lil-log*, 2018, [Accessed online: 13th December 2021]. [Online]. Available: http://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html.

[26] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[27] G. Ke, Q. Meng, T. Finley, *et al.*, "Lightgbm: A highly efficient gradient boosting decision tree," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[28] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, San Francisco, California, USA: ACM, 2016, pp. 785–794, ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785.

[29] U. G. G. Foerlev, *Probability of distress*, eng, 2019.

[30] B. Christoffersen, R. Matin, and P. Mølgaard, "Can machine learning models capture correlations in corporate distresses?" *Available at SSRN 3273985*, 2019.

[31] scikit-learn, *Receiver operating characteristic (roc)*, https://scikit-learn.org/stable/modules/model_evaluation.html#roc-metrics, [Accessed online: 13th December 2021], 2021.

[32] C. H. Lin, D. S. Weld, *et al.*, "To re (label), or not to re (label)," in *Second AAAI conference on human computation and crowdsourcing*, 2014.

[33] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017, ISBN: 0128122757.

[34] A. Webb, *High detail 'zen 2' cpu core layout (with zen 1 for reference)*. https://www.eridonia-archives.com/post/high-detail-zen-2-cpucore-layout-with-zen-1-for-reference, [Accessed online: 13th December 2021], 2020.

[35] NVIDIA, *Programming guide :: Cuda toolkit documentation*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, [Accessed online: 13th December 2021], 2021.

[36] Intel, *Intel®data protection technology with aes-ni and secure key*, https://www.intel.com/content/www/us/en/architecture-and-technology/advanced-encryption-standard-aes/data-protection-aes-general-technology.html, [Accessed online: 13th December 2021], 2021.

[37] Intel, *Intel®quick sync video*, https://www.intel.com/content/www/us/en/architecture-and-technology/quick-sync-video/quick-sync-video-general.html, [Accessed online: 13th December 2021], 2021.

[38] NVIDIA, *Rtx technology*, https://developer.nvidia.com/rtx/raytracing, [Accessed online: 13th December 2021], 2021.

[39] A. Thegler, *Towards automatic program specification from sme models*, eng, 2018.

[40] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, 2019.

[41] Xilinx, *Vivado design suite user guide - high-level synthesis*, https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf, [Accessed online; 13th December 2021], 2021.

[42] ARM, *Amba axi and ace protocol specification axi3, axi4, and axi4-lite ace and ace-lite*, https://developer.arm.com/documentation/ihi0022/e/, [Accessed online; 13th December 2021], 2013.

[43] ARM, *Amba 4 axi4-stream protocol specification*, https://developer.arm.com/documentation/ihi0051/a, [Accessed online; 13th December 2021], 2010.

[44] Veripool, *Verilator*, https://www.veripool.org/verilator/, [Accessed online; 13th December 2021], 2021.

[45] C.-J. Johnsen, *Rtllib - utility library for integrating rtl kernels into higher level workflows*, https://github.com/carljohnsen/rtllib, [Accessed online: 13th December 2021], 2020.

[46] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 32, no. 5, pp. 1014–1029, 2020.

[47] Xilinx, *Alveo u280 data center accelerator card data sheet*, https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds963-u280.pdf, [Accessed online; 13th December 2021], 2021.

[48] Xilinx, *Vitis unified software development platform 2021.1 documentation - using full axi data width*, https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/optimizingperformance.html#jpk1504034303169, [Accessed online; 13th December 2021], 2021.

[49] Xilinx, *Virtex ultrascale+ fpga data sheet: Dc and ac switching characteristics*, https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf, [Accessed online; 13th December 2021], 2021.

[50] C.-J. Johnsen, T. De Matteis, J. de Fine Licht, T. Ben-Nun, and T. Hoefler, "Automatic multi-pumping of computational subdomains," Submitted to DAC'22, 2021.

[51] Xilinx, *Axi4-stream infrastructure ip suite v3.0*, https://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf, [Accessed online; 13th December 2021], 2018.

[52] C. Baaij, "Digital circuit in cλash: Functional specifications and type-directed synthesis," Undefined, eemcs-eprint-23939, Ph.D. dissertation, University of Twente, Netherlands, Jan. 2015, ISBN: 978-90-365-3803-9. DOI: 10.3990/1.9789036538039.

[53] Y.-H. Lai, Y. Chi, Y. Hu, *et al.*, "Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[54] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18, Monterey, CALIFORNIA, USA: Association for Computing Machinery, 2018, pp. 127–136, ISBN: 9781450356145. DOI: 10.1145/3174243.3174264. [Online]. Available: https://doi.org/10.1145/3174243.3174264.

[55] J. Bachrach, H. Vo, B. Richards, *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12, San Francisco, California: Association for Computing Machinery, 2012, pp. 1216–1225, ISBN: 9781450311991. DOI: 10.1145/2228360.2228584. [Online]. Available: https://doi.org/10.1145/2228360.2228584.

[56] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, IEEE, 2017, pp. 1–7.

[57] J. Decaluwe, "Myhdl: A python-based hardware description language," *Linux journal*, vol. 2004, no. 127, p. 5, 2004.

[58] P. R. Panda, "Systemc: A modeling platform supporting multiple design abstractions," in *Proceedings of the 14th International Symposium on Systems Synthesis*, ser. ISSS '01, Montréal, P.Q., Canada: Association for Computing Machinery, 2001, pp. 75–80, ISBN: 1581134185. DOI: 10.1145/500001.500018. [Online]. Available: https://doi.org/10.1145/500001.500018.

[59] D. Koeplinger, M. Feldman, R. Prabhakar, *et al.*, "Spatial: A language and compiler for application accelerators," *SIGPLAN Not.*, vol. 53, no. 4, pp. 296–311, Jun. 2018, ISSN: 0362-1340. DOI: 10.1145/3296979.3192379. [Online]. Available: https://doi.org/10.1145/3296979.3192379.

[60] J. Thomas, P. Hanrahan, and M. Zaharia, "Fleet: A framework for massively parallel streaming on fpgas," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 639–651, ISBN: 9781450371025. DOI: 10.1145/3373376.3378495. [Online]. Available: https://doi.org/10.1145/3373376.3378495.

[61] D. Ramyar, *Risc-v processor in sme*, eng, 2020.

[62] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.

[63] C.-J. Johnsen, *Implementing a mips processor using sme*, eng, 2017.

[64] C. Whitby-Strevens, "The transputer," in *ACM SIGARCH Computer Architecture News*, IEEE Computer Society Press, vol. 13, 1985, pp. 292–300.

[65] D. Pountain and D. May, *A tutorial introduction to OCCAM programming*. McGraw-Hill, Inc., 1987.

[66] C. Hoare, *Communicating Sequential Processes*. London: Prentice-Hall, 1985, ISBN: 0-131-53271-5.

[67] M. Broløs, *Occam transpiler to go*, eng, 2021.

[68] M. Broløs, C.-J. Johnsen, and K. Skovhede, "Occam to go translator," in *2021 IEEE Concurrent Processes Architectures and Embedded Systems Virtual Conference (COPA)*, 2021, pp. 1–8. DOI: 10.1109/COPA51043.2021.9541431.

[69] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*, 1st. Addison-Wesley Professional, 2015, ISBN: 0134190440.

[70] J. Meznik and M. J. Jacobi, *Tcp/ip in hardware using sme*, eng, 2019.

[71] P. D. Sørensen and E. S. Bak, *High performance fpga firewall using sme*, eng, 2019.

[72] T. Skovsgaard, *A bohrium to sme translator*, eng, 2020.

[73] M. Kristensen, S. Lund, T. Blum, K. Skovhede, and B. Vinter, "Bohrium: Unmodified numpy code on cpu, gpu and cluster," Nov. 2013.

[74] A. Moussa, *Acceleration of machine learning through an fpga*, eng, 2021.

[75] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[76] A. Thegler, C.-J. Johnsen, K. Skovhede, and B. Vinter, "Accelerating molecular dynamics with the lennard-jones potential for fpgas," in *2021 IEEE Concurrent Processes Architectures and Embedded Systems Virtual Conference (COPA)*, 2021, pp. 1–8. DOI: 10.1109/COPA51043.2021.9541435.

[77] J. Flach-Jensen and J. C. Herbst, *Cryptographic library for fpgas*, eng, 2021.

[78] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-performance fpga-based cnn accelerator with block-floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1874–1885, 2019. DOI: 10.1109/TVLSI.2019.2913958.

[79] EETimes, *Tutorial: Linear feedback shift registers (lfsrs) – part 1*, https://www.eetimes.com/tutorial-linear-feedback-shift-registers-lfsrs-part-1/, [Accessed online: 13th December 2021], 2006.

[80] L. Zhang, L. Zhang, X. Mou, and D. Zhang, "Fsim: A feature similarity index for image quality assessment," *IEEE transactions on Image Processing*, vol. 20, no. 8, pp. 2378–2386, 2011.

[81] ONNX, *Open neural network exchange (onnx)*, https://github.com/onnx/onnx, [Accessed online; 13th December 2021].

[82] Xilinx, *Vitis ai*, https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html, [Accessed online; 13th December 2021].

[83] T. Köhler, X. Huang, F. Schebesch, A. Aichert, A. Maier, and J. Hornegger, "Robust multiframe super-resolution employing iteratively re-weighted minimization," *IEEE Transactions on Computational Imaging*, vol. 2, no. 1, pp. 42–58, 2016. DOI: 10.1109/TCI.2016.2516909.

[84] TUL, *Pynq-z2 product page*, https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html, [Accessed online 13th December 2021].

Part VIII

APPENDICES

A

PAPERS

A.1 PUBLISHED PAPERS

A.1.1 *Implementing a Transputer for FPGA in less than 800 lines of code*

Communicating Process Architectures 2018
K. Chalmers, J.B. Pedersen et al. (Eds.)
IOS Press, 2018

Implementing a Transputer for FPGA in less than 800 lines of code

Carl-Johannes Johnsen [a,1], Kenneth Skovhede [a], Brian Vinter [a], Lindsay Quarrie [b], Larry Dickson [b]

[a] *Niels Bohr Institute, University of Copenhagen*
[b] *Space Sciences Corporation*

Abstract. By utilizing Synchronous Message Exchange (SME) [1] for hardware design, we see that going from a hardware schematic to an implementation becomes a much shorter process. This in turn shifts the focus to the architectural details of the implementation. This is shown by constructing an implementation of the Transputer [2] in SME. This implementation has been made in less than 800 lines of code within the timeframe of ∼4 months, where the majority of the time spent has been on the Transputer architecture. The resulting naive implementation is suboptimal compared to similar projects [3,4]. However, since no optimizations have been made, reaching a more reasonable resource consumption and clockrate should be attainable within a few months.

Keywords. Transputer, SME, FPGA, Hardware, Processor architecture, Occam, CSP

Introduction

The Transputer [2] is a microprocessor architecture introduced in the 1980s. It is a stack machine, which focuses on concurrent execution and communication amongst multiple Transputers. It comes in different variations: 16-bit [5], 32-bit [6] and 32-bit with a floating point unit [7]. However, while this architecture engages many of the challenges faced by modern architectures, it was ahead of its time. Back then, 8-bit processors ruled the market due to low pricing and increasing performance.

The Transputer is closely linked to Occam [8]. Occam is a programming language, which uses the Communicating Sequential Processes (CSP) programming model [9]. In CSP, a computation is executed by multiple concurrent processes that are communicating through channels. Communication occurs in a rendezvous fashion, where both processes need to synchronize with each other before transferring data.

Constructing custom hardware, such as Application Specific Integrated Circuits (ASICs), is a long complex and tedious task. When hardware has been designed, it must be implemented using a Hardware Description Language (HDL), such as VHDL and Verilog. Then it must be physically produced and verified, both of which are expensive and time consuming. By using Field Programmable Gate Arrays (FPGAs), the production time of hardware is decreased from months to hours. FPGAs are prototyping boards, which implement many basic components, which when wired together produce a circuit that is semantically equivalent to actual hardware. However, FPGAs are also programmed using HDLs, which are tedious to work with.

[1]Corresponding Author: *Carl-Johannes Johnsen, Niels Bohr Institute, Blegdamsvej 17, DK-2100 Copenhagen OE.*. Tel.: +45 35331501; E-mail: cjjohnsen@nbi.ku.dk.

This has changed with Synchronous Message Exchange (SME) [1]. SME is a programming model, which closely resembles the CSP programming model as it consists of processes communicating through busses. The key difference is that SME is globally synchronous, has broadcasting channels, a hidden clock, and the structure is closer to the semantics of hardware. Because of this structure, SME can be transpiled into VHDL, which can be further synthesized onto an FPGA. As such, SME gives software developers an entry point for hardware design, which is closer to the world of software, rather than the jungle of low level hardware design.

Contribution

This paper describes the Transputer microprocessor implemented in SME. First, an introduction to the Transputer architecture will be given. Then a quick description on how the components of the Transputer are implemented in SME. Finally, this paper describes how the Transputer is benchmarked and verified.

1. Related work

The Transputer already exists in modern forms. Primarily as an emulator in the Occam compiler, KRoC [10], as the Transterpreter [11]. The Transterpreter can emulate the Transputer byte code generated by the KRoC compiler. However, the Transterpreter is just an interpreter for the Transputer byte code and as such only runs on the platform for which the compiler has been compiled (E.g. x86).

Another project is the T42 [4], which is a reimplementation of the Transputer on an FPGA. While this project solves the same problem as proposed by this paper, it is a full reimplementation written purely in VHDL. As such, going from design to implementation and the verification of the implementation is a lengthy and complex process.

Finally another FPGA implementation of the Transputer is the OpenTransputer [3]. This project aims to accept the same instruction set as the original Transputer, but with a new micro architecture. This project uses Verilog for its hardware implementation, which is why it has the same challenges as the T42.

2. Transputer

This section describes some of the concepts of the original Transputer. All of the information has been gathered from the documents related to the Transputer [5,6,7,12,13].

2.1. Overview

The original Transputer came in a variaty of models. First there was the T200 series [5], which were 16-bit processors. The next line was the T400 series [6], which were 32-bit processors. Finally there were the T800 series [7], which were identical to the T400 processors, except they carried a floating point unit. A newer version of the Transputer was developed: the T9000 [14]. The T9000 never entered production, as it was unable to reach its target performance. This paper aims to reimplement a Transputer from the T400 series, i.e. a 32-bit Transputer without floating point support.

The simple block diagram for the architecture of a T400 Transputer can be seen in Figure 1. It consists of 5 core components:

- **Processor** - The core of the Transputer. It is this component, which decodes and executes the instructions of a program. It can control the links, access the memory (both

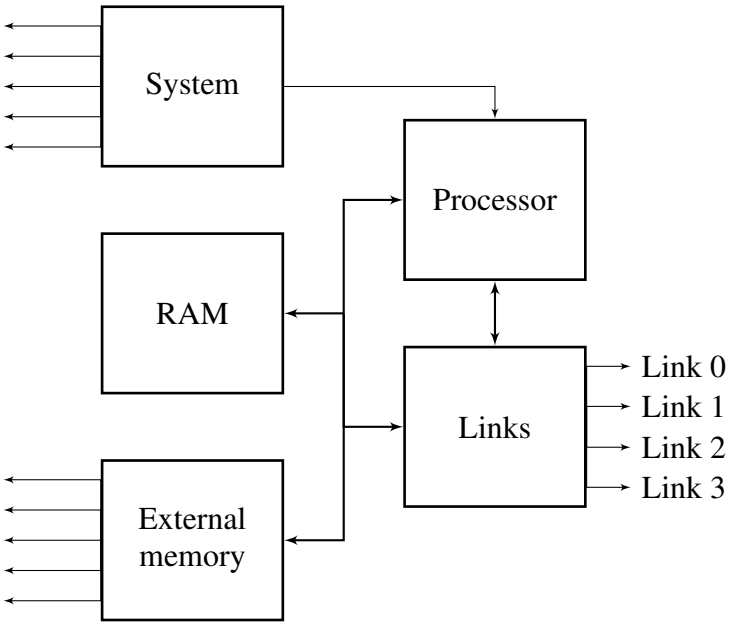


Figure 1. The simple block diagram of a Transputer.

internal and external) and it also reacts to the external inputs through the System services.

- **RAM** - The fast memory of the Transputer. Its size varies between various Transputer models.
- **External memory** - The slow memory of the Transputer. In a 32-bit Transputer, ∼4 GB of memory can be accessed.
- **Links**- The external links of the Transputer. Each link is in fact a bidirectional serial link. They are either connected to other Transputers or to devices such as disks.
- **System services** - This component contains the timers and external signals of the Transputer. Which signals this component is connected to varies from the Transputer models.

The Transputer is a processor closely related to Occam: its main features are concurrency and communication. Each Transputer is able to run multiple processes concurrently and communicate amongst them, without the need for an operating system. Furthermore, it has 4 serial links, which can be used to communicate to other Transputers. As such, the Transputer was built with concurrency and parallalism in mind.

2.2. Instruction set

Even though the Transputer is a 32-bit processor, its instructions are 8-bit. The most significant 4 bits are the opcode and the least significant 4 bits are the operand. The opcode can only be used to describe 16 instructions. These are called the direct functions. The motivation for only having 16 direct functions is that (at least at the time of the Transputer) ∼70% of the instructions used in programs, were these instructions [15] (Section 3.2.5).

To support more instructions, the Transputer is also able to utilize the operand register as opcode. These are called operations. Whenever an instruction is decoded, the operand of the instruction is inserted into the 4 least significant bits of the operand register. Two instructions (`pfix`, `nfix`) are used to compose operands larger than 4 bits. `pfix` shifts the operand register left 4 bits. `nfix` negates the operand register and then shifts the operand register left 4 bits. A third instruction (`opr`) is used to interpret the operand register as opcode.

3 of the direct functions are used to utilize the operand register as opcode:

- `pfix` which shifts the operand register left by 4.

- `nfix` which negates the operand register and left shifts it by 4.
- `operate` which interprets the operand register as opcode.

By using these instructions, every 32-bit number can be constructed, which in turn increases the amount of instructions in the instruction set. The amount of clock cycles needed for each instruction is specified in the INMOS documents [12,13].

2.3. Registers

The Transputer has 3 general purpose registers: `a`, `b` and `c`. These are used in a stack like behaviour. E.g. when the load constant (`ldc`) instruction is executed, `b` is pushed into `c`, `a` is pushed into `b` and the operand register is pushed into `a`. The Transputer models carrying a floating point unit have three similar registers `fa`, `fb`, `fc`, and an additional register defining the rounding mode.

The Transputer has 3 registers related to the current running process:

- **operand** - This register is the operand register described in Section 2.2. It can be either an immediate operand for an instruction or an opcode in case of the `opr` instruction.
- **instruction pointer** - The register which contains the address for the next instruction. Each time an instruction is executed, this register is incremented.
- **workspace** - This register contains the workspace (`wptr`) pointer of the current process. This is equivalent to the stack pointer on other architectures such as x86 or MIPS.

Finally, the Transputer has 8 registers related to the scheduler of the Transputer:

- `fptrreg0` and `fptrreg1` - These registers point to the front of the scheduling queues. There are two queues: high priority (0) and low priority (1).
- `bptrreg0` and `bptrreg1` - These registers point to the back of the scheduling queues. Again, there are two priority queues.
- `tptrloc0`and `tptrloc1` - These registers point to the head of the timer queues. Again, there are two priority queues.
- `clockreg0` and `clockreg1` - These are the clock registers. They contain the value of the current timer for each priority queue.

2.4. Memory model

The Transputer has 3 types of memory:

- `RAM` - This is the fastest main memory of the Transputer. As such, this should be the memory space where most processes reside. On a T425 it is 4 KB in size. It goes from the minimum 32-bit integer (`0x80000000`) and its size up (E.g. `0x80001000` on the T425). User RAM starts at `MemStart` (`0x80000070` for the T425).
- `ROM` - This is where the boot program of the Transputer, if any, resides. The two bytes in the highest integer address (`0x7FFFFFFF` and `0x7FFFFFFE`) contains a backward jump into the ROM. Since there are only two bytes, the jump can be a maximum of -256.
- `External Memory` - The rest of the memory space, from the end of RAM to the start of ROM, is External Memory. This is the slowest, but also largest, memory of the Transputer.

2.5. Process memory

The memory related to a process contains certain reserved locations related to the context of the process. These are stored below the workspace pointer. These locations contain:

- The instruction pointer of the process. This is used when scheduling and descheduling the process.
- The `NextProcess` pointer. This is a pointer to the next process in the scheduling queue. If there are no further processes on the scheduling queue then the value is `NotProcess`.
- The buffer. This is used for either the source or destination address in a communication. E.g. if a process needs to output, it will store the adress of the data to be communicated befor it deschedules itself.
- Flag used during timer ALTs to indicate a valid time to wait for.
- The Time. This field contains the desired wakeup time for the process. If the timer reaches this value the process will be scheduled.

2.6. System services

The Transputer features many System services. We will only name a few, which are important for the internal Transputer:

- **Power** - Power is supplied to the Transputer with the `VCC` and `GND` pins.
- **ClockIn** - The clock for the Transputer is used for driving the internal timers. As such, the clock must match what the Transputer is expecting.
- **BootFromROM** - This flag is used to indicate whether the Transputer should boot from the internal ROM or wait for a program over the external links. This is also described in Section 2.7
- **Analyse** - This pin is used to halt the Transputer at the next descheduling point. Once halted, the Transputer retains its internal state for further analysis (I.e. memory is not reset). From here, a small boot program can be loaded.
- **Peek and poke**- This pin is used to read or write the memory contents of the Transputer. This can be done after a reset, when the BootFromROM pin is low, when the Transputer is waiting for a bootstrap from link.
- **ErrorIn and ErrorOut** - These pins are used to communicate externally whether or not there is an error somewhere in the Transputer network. These pins can be used to stop a network of Transputers. ErrorIn is an input signal from the rest of the network. ErrorOut is the OR value of ErrorIn and the local Transputer Error flag.

2.7. Booting/Debug

After a reset, the Transputer starts in a state where it checks the `BootFromROM` pin. If it is high, it reads the two bytes in the maximum integer address, and performs the backward jump into ROM. If it is low, it is in a network boot/debug mode. This mode supports peeking and poking into the memory of the Transputer. Furthermore, it can load a program read from one of the external links into its memory, and then run it. As such, a single Transputer can be used to boot a network of Transputers.

3. SME Transputer

This section describes how the Transputer is implemented using the SME programming model [1]. At the time of writing, it is implemented using the SME state machines [16]. This is done to both reduce the execution path of the processor, when heavy instructions are executed, and to handle instructions, which consist of multiple accesses to memory.

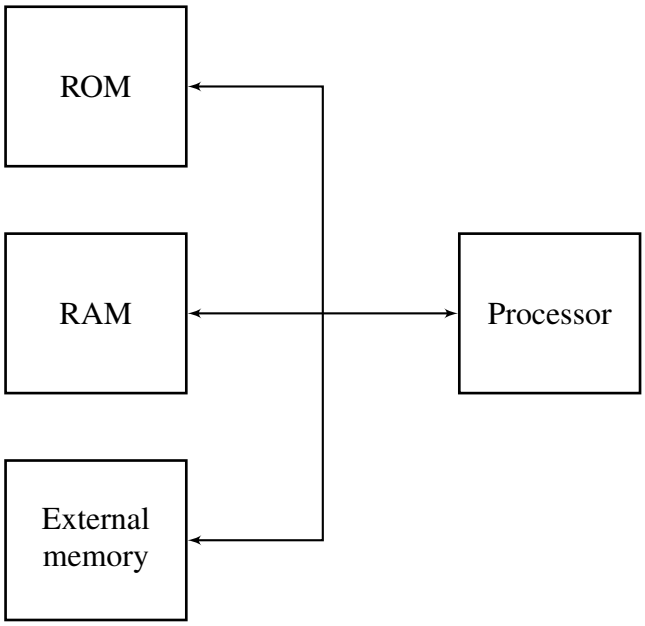


Figure 2. Overview of the SME processes and buses making up the SME Transputer.

3.1. Core processer

The core of the processor is implemented using a single SME process. It issues a read request to the memory, in order to fetch an instruction. Once fetched, the instruction is split into its two parts: opcode (bits 7 to 4) and operand (bits 3 to 0). The processor process then performs a `switch` on the opcode, which indicates which instruction to execute. Each part of the Processor is divided into its own state. Executing an instruction thus takes a minumum of three clock cycles: fetch, decode and execute.

3.2. Memory model

The memory of the Transputer is handled in 3 different processes: Internal ROM, Internal RAM and External RAM. Both of the internal processes are implemented on the FPGA using Block RAM, in order to increase performance and reduce resource consumption. All three processes share the same bus. Usually on FPGAs this is not desired. However, since the memory space of the three processes are mutually exclusive, only one of them will be reacting to a memory request at a given time. There is still a need for an multiplexor in between the memory processes output and the processor. This is due to the FPGA synthesis tools not allowing multiple drivers (multiple processes writing to the same bus).

At this time, the SME Transputer does not interface with the external memory on an FPGA. It currently only exists in SME simulation. In later implementations, it should interface with e.g. the DDR memory of a board. Furthermore, it should communicate with the processor whether or not it is waiting for a memory transaction to be completed, in which case the processor should be stalled.

3.3. Scheduler

The scheduler is implemented exactly as specified by the Transputer technical documents. There are two queues, a high and a low priority, both of which are linked lists managed by a front and a back pointer. Whenever a process is scheduled it is popped from the front of the list. Whenever it is descheduled (E.g. at the end of its timeslice) it is put at the back of the queue. It is not in any queue while executing. If it is descheduled due to waiting for

communication the process is queued on the channel (I.e. the process puts its `Wdesc` into memory at the channel address).

When a descheduling point is reached, the Transputer will deschedule the process and schedule the next ready process. If a high priority process becomes ready and no other high priority process is running then it will start to run immediately. Any low priority process will be preempted and its status saved in the `IntSaveLoc` memory locations, to be resumed from there if the high priority process is descheduled. A high priority process will run until it reaches a descheduling point. It is therefore up to the compiler to prevent starvation.

3.4. Communication

The Transputer supports internal communication (I.e. communication among processes running on the local Transputer), by using the internal memory, and external communication, by using serial links. These links communicate with the memory using a DMA. Each link has its own DMA.

3.4.1. Internal

When two processes initiate communication, the first process to execute a communication instruction starts by inspecting the memore stored at the channel address to see if its partner process has stored its `Wdesc` in it. The channel word must be initialised to `NotProcess` before it can be used. When it sees the value `NotProcess` it knows it was the first to arrive. It then stores the buffer address and its instruction pointer in its workspace, its `Wdesc` in the memory at the channel address and stops. It is not in any queue and can only be rescheduled by the second process to arrive. Once the second process arrives, it also checks the channel word, this time seeing the other process' `Wdesc`. It then retrieves the buffer address from the first process' `Wdesc` and copies data. When the data has been copied it reschedules the first process and continues.

Note that there has been no negotiation on neither communication direction nor the size of the communication. E.g. two processes both trying to output can succesfully communicate. This is expected to be handled by the usage checker of the compiler.

3.4.2. External (Links)

This part of the Transputer is not fully implemented in the SME Transputer at the time of writing. However, the overall structure for it is in place. Each link should consist of three processes: an DMA, input link and output link. Each of the four links should go through a single process, to ensure that multiple communications will not interact with each other. This is due to the memory of an FPGA only having two ports. The first port is used by the processor, and the second should then be used by the links. This will also ensure that the external communication will run in parallel with the processors execution. The processor will also be able to control the links, in order to ensure that it does not schedule processes that are waiting for communication, until the communication is succesful.

3.5. Code example

To illustrate how SME eases the development process, we show how a process is enqueued. The enqueue routine is originally described in a patent by INMOS [17] on page 11 in the description of `PROC RUN` lines 7-13. We follow the same procedure (Note: all of the steps need to be performed as one atomic operation):

- If the associated queue is empty, then the front pointer should be set to the newly enqueued process. Otherwise, it should update the `NextProcess` pointer of the previous last process.

```
1  // Find the appropriate queues
2  uint fqueue = highpri ? fptrreg0 : fptrreg1;
3  uint bqueue = highpri ? bptrreg0 : bptrreg1;
4  if (fqueue == (uint)Constants.notprocess) // Empty queue
5  {
6      // Update the front pointer
7      if (highpri)
8          fptrreg0 = wptr;
9      else
10         fptrreg1 = wptr;
11 }
12 else
13 {
14     // Update the NextProcess pointer of the previous last process
15     // to point to the newly enqueued process
16     memin.ena = true;
17     memin.addr = bqueue + Constants.w_nextp; // (nextp = -8)
18     memin.wrena = true;
19     memin.wrdata = wptr;
20     memin.wrbyte = Constants.word_mask;
21     await ClockAsync();
22 }
23
24 // Update the nextp pointer of the new process to notprocess
25 memin.ena = true;
26 memin.addr = wptr + Constants.w_nextp; // (nextp = -8)
27 memin.wrena = true;
28 memin.wrdata = (uint)Constants.notprocess;
29 memin.wrbyte = Constants.word_mask;
30 await ClockAsync();
31
32 // Update the back pointer
33 if (highpri)
34     bptrreg0 = wptr;
35 else
36     bptrreg1 = wptr;
```

Listing 1 SME code for enqueueing a process.

- The `NextProcess` pointer of the newly enqueued process should point to nothing.
- The back pointer should be updated to point to the newly enqueued process.

The corresponding SME code can be seen in Listing 1 and the generated VHDL code in Listing 2. Because we are using SME state machines, we see that the structure of the implementation is close to the description, but still a bit different due to the state machine flow. The generated VHDL code is both larger in size and more complex due to the state machine logic. However, the overall flow of the SME program can still be found in the VHDL code, which makes debugging and modifying the VHDL code easier.

4. Verification/Benchmarks

The SME Transputer has been verified by multiple programs. Each of these programs are verified by having a tester process, which knows the expected output. Everytime an `out`, `outbyte` or `outword` is executed, it asserts that the output matches the expected output.

```
1  if FSM_RunState = State2 then
2  ...
3      if operand = TO_UNSIGNED(13, 32) then
4          if highpri = '1' then
5              queue5 := fptrreg0;
6          else
7              queue5 := fptrreg1;
8          end if;
9          if highpri = '1' then
10             queue6 := bptrreg0;
11         else
12             queue6 := bptrreg1;
13         end if;
14         FSM_RunState := State50;
15 ...
16 if FSM_RunState = State50 then
17     if queue5 = TO_UNSIGNED(16#80000000#, 32) then
18         if highpri = '1' then
19             fptrreg0 := workspace;
20         else
21             fptrreg1 := workspace;
22         end if;
23         FSM_RunState := State51;
24     else
25         memin_ena <= '1';
26         memin_addr <= UNSIGNED((SIGNED(queue6) + TO_SIGNED(-8, 32)));
27         memin_wrena <= '1';
28         memin_wrdata <= workspace;
29         memin_wrbyte <= resize(TO_UNSIGNED(15, 8), T_UINT4'length);
30         FSM_NextState <= State51;
31     end if;
32 end if;
33 if FSM_RunState = State51 then
34     memin_ena <= '1';
35     memin_addr <= UNSIGNED((SIGNED(workspace) + TO_SIGNED(-8, 32)));
36     memin_wrena <= '1';
37     memin_wrdata <= TO_UNSIGNED(16#80000000#, 32);
38     memin_wrbyte <= resize(TO_UNSIGNED(15, 8), T_UINT4'length);
39     FSM_NextState <= State52;
40 end if;
41 if FSM_RunState = State52 then
42     if highpri = '1' then
43         bptrreg0 := workspace;
44     else
45         bptrreg1 := workspace;
46     end if;
47     FSM_RunState := State53;
48 end if;
```

Listing 2 Generated VHDL code for enqueueing a process.

For the hand written assembly, the expected output is computed by hand. For the compiled programs, the expected output is taken from the results of the x86 program from KRoC.

- **direct** - This program is a hand written Transputer assembly, which executes all of the direct functions of the Transputer.
- **operations**- This program is a hand written Transputer assembly, which executes some

Table 1. The count of source lines of code for the SME Transputer and its generated VHDL.

Language	SLOC
SME	732
VHDL	1778

of the operations of a Transputer.

- **simple** - This is an hello world program in Occam, compiled with KRoC.
- **fib** - An Occam program compiled with KRoC, which computes the first 10 fibonacci numbers. It can be seen in Appendix A in Listing 3.
- **callnreturn** - An Occam program compiled with KRoC, which calls multiple functions multiple times. This is to test that the `call` and `ret` instructions behave as expected. It can be seen in Appendix A in Listing 4.
- **conc** - This program consists of multiple processes, which run concurrently. This is to test the scheduler of the Transputer. It can be seen in Appendix A in Listing 5.

The SME Transputer has succesfully been simulated in SME and we have generated VHDL code. If we look at the resulting source lines of code (SLOC) in Table 1, we see that the SME uses less than 50% lines of code compared to the generated VHDL. This alone should indicate a reduced complexity during implementation.

As the external communication is not fully functional at the time of writing, actual benchmarks of a fully placed and routed Transputer running on an FPGA cannot be made. However, the numbers reported on resource consumption and clockrate from the synthesis tools can be seen in Table 2. Only the amount of LUTs is being used for comparison, as the numbers from the T42 [4] are from a different board, than the SME Transputer and the OpenTransputer [3]. The board used by both the OpenTransputer and the SME Transputer is the ZedBoard [18], which features an Xilinx Zynq XC7Z020-CLG484 [19].

The results shows that the SME Transputer is the slowest of all the Transputer implementations. This is due to the lack of performance optimizations. The current focus of the project have been to correctly implement a Transputer running real Transputer binaries. Looking at the resource consumption, we see that we use a lot more LUTs than the T42.

We obtain better resource consumption than the OpenTransputer, but with a lower clockrate. The OpenTransputer was able to implement more of the Transputer than the SME Transputer currently implements, which is why its resource consumption is expected to be higher. This goes for the T42 as well, but it has been in development for longer. This should give the T42 additional hardware improvements regarding resource consumption and clockrate over the OpenTransputer.

Two SME implementations of a MIPS processor have been added, seen in Table 3, in order to show the effects of leveraging hardware design when working with SME. The ”Simple MIPS” implementation [20] follows the same approach as the current version of the SME Transputer, by using SME state machines [16]. This approach is simpler, from a developers view, but less efficient. The ”Pipelined MIPS” implementation [21] follows the design often taught in machine architecture classes [22].

First, we see that the resource consumption is lower in the pipelined implementation, compared to the simple. As such, we can assume that the same improvement will affect the SME Transputer. Second, although the clockrate is higher in the simple implementation, the overall performance is much better in the pipelined implementation. This is due to the simple implementation only executing a single pipeline stage at once. The pipelined implementation executes all 5 pipeline stages in parallel, thus increasing maximum throughput. Again, we should see the same effect in later iterations of the SME Transputer.

Finally, one of the real Occam benchmarks, commstime, has run succesfully on the SME Transputer. It is a slight simplified version of the original, as timer functionality has not yet

Table 2. Resource consumption and achieved clockrate of the SME Transputer compared to the same numbers from similar projects.

Implementation	LUTs	Clockrate
SME Transputer	8686	26.32 MHz
OpenTransputer	14744	41.00 MHz
T42	4000	100.00 MHz

Table 3. Resource consumption, achieved clockrate, maximum instructions per clock and maximum instructions per second of the two SME implementations of a MIPS processor.

Implementation	LUTs	Clockrate	Max IPC	Max IPs
Simple MIPS	8061	232.45 MHz	0.2	46.49
Pipelined MIPS	3724	71.43 MHz	1.0	71.43

Table 4. The performance of the simplified commstime on the SME Transputer and the corresponding x86 program. The x86 program ran on an Intel i7-7700HQ (3.8 GHz). Both programs have been compiled using KRoC 1.5.0-pre4. The Raspberry Pi (700 MHz) benchmark is gathered externally [23].

Platform	Clock ticks per loop	Clock ticks per context switch
SME Transputer	239	29
Intel i7-7700HQ	438	54
Raspberry Pi	7480	935

been implemented. The code for the simplified commstime can be seen in Listing 6 Appendix A. The performance of the simplified commstime can be seen in Table 4. Given the lack of timer functionality, we have compared it to other microarchitectures based on the amount of clock ticks needed. The benchmark has been run with an n value of 1000000. The amount of clock ticks for the non-Transputer processors has been estimated by multiplying the execution time with the processors clockrate. E.g. for the i7 with an clockrate of 3.8 GHz, the clock ticks 3800 times per microsecond. Given the execution time of 115399 microseconds, we get that the clocked ticked $3800 \times 115399 = 438516200$ times.

If we look at the numbers, we see that the SME Transputer, even in its current state, uses little more than 50% of the same clock ticks, as the Intel i7. This is very good, as the SME Transputer in its current state does not feature any hardware optimizations. We expect this number to further improve as instruction level parallelism will be exploited, when we introduce pipelining. We have also included numbers found from an Raspberry Pi implementation running commstime [23]. Surprisingly, once running on ARM, the amount of clock ticks needed grows. This is probably due to the Raspberry Pi still being a new platform, where KRoC have not received the same amount of work as the x86 version.

5. Conclusion

We have succesfully implemented, placed and routed the SME Transputer in a short timeframe. It is targeted for running Transputer bytecode, which has been shown. Although the results are suboptimal, better performance should be low hanging fruit once the full Transputer instruction set has been implemented in the SME Transputer, and when it has been entirely verified.

6. Future work

The next step is to implement external communication. Once this is in place, we should be able to start running more of the standard Occam benchmarks, which should make this project

more interesting and verifiable.

Then the rest of the instructions should be added, especially the ones regarding the `ALT` Occam construct.

Then the external memory interface should be added. As mentioned before, it should also include communication with the processor regarding the status of the memory request.

Then the system services should be correctly implented. With these, we will be able to set up the timer functionality in Occam.

Along the system services comes the need for proper booting. In its current state, the SME Transputer generates VHDL where the ROM is loaded with a program to run. It will always boot from ROM. It should, just like the original Transputer: be able to boot from network and be debugged from network.

Once all of these components are implemented, then the SME Transputer is fully featured. At this point it should be cross verified to produce the same result as both Transputer emulators and other Transputer implementations (including an original Transputer if possible). To fully support this, we should also have it be fully compatible with the original INMOS toolchain.

With the SME Transputer being fully verified and with the verification environment being fully automatized, we will look into optimizing the hardware design. Here we will look at pipelining the processor and, if possible, reuse some of the logic, thus reducing the complexity of the control system and the ALU.

Finally, to be 100% true to the original Transputer, the SME Transputer will have a flag at compile time indicating whether or not it should be true to the amount of clock cycles needed for each instruction. This would further improve the compatibility, as there might exist some programs, which assume that a certain instruction will take X amount of clock cycles at a given clockrate.

Acknowledgements

This project has been supported by Space Sciences Corporation.

References

[1] Kenneth Skovhede and Carl-Johannes Johnsen. SME source code. `https://github.com/kenkendk/sme`. [Online; accessed April 2018].

[2] Colin Whitby-Strevens. The transputer. In *ACM SIGARCH Computer Architecture News*, volume 13, pages 292–300. IEEE Computer Society Press, 1985.

[3] Andres Amaya-Garcia, David Keller, and David May. ”opentransputer: Reinventing a parallel machine from the past”. In Kevin Chalmers, Jan Bækgaard Pedersen, Frederick R. M. Barnes, Jan F. Broenink, Ruth Ivimey-Cook, Adam T. Sampson, and Peter H. Welch, editors, *”Communicating Process Architectures 2015”*, pages 5 – 26. ”Open Channel Publishing Ltd., Bicester, UK”, ”August” 2015.

[4] Martin Zabel, Michael Bruestle, and Uwe Mielke. ”t42 – transputer design in fpga (year three design status report)”. In Jan Bækgaard Pedersen, Kevin Chalmers, Jan F. Broenink, Brian Vinter, Kevin Vella, and Peter H. Welch, editors, *”Communicating Process Architectures 2017”*. ”Open Channel Publishing Ltd., Bicester, UK”, ”August” 2017. Fringe Presentation.

[5] SGS-Thomson microelectronics. *IMS T225*, July 1995.

[6] SGS-Thomson microelectronics. *IMS T425*, February 1996.

[7] SGS-Thomson microelectronics. *IMS T805*, February 1996.

[8] Dick Pountain and David May. *A tutorial introduction to OCCAM programming*. McGraw-Hill, Inc., 1987.

[9] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-131-53271-5.

[10] The kent retargetable occam compiler. `https://github.com/concurrency/kroc`. [Online; accessed April 2018].

[11] The transterpreter. http://www.transterpreter.org/. [Online; accessed April 2018].
[12] Bristol INMOS Limited. *Transputer instruction set: a compiler writer's guide*. Prentice Hall, 1988.
[13] John Roberts. *Transputer assembly language programming*. John Wiley & Sons, Inc., 1992.
[14] INMOS. *T9000 Hardware Reference Manual*, 1993.
[15] INMOS Limited. *Transputer Reference Manual*. Prentice Hall, 1988.
[16] Kenneth Skovhede and Carl-Johannes Johnsen. Building fpga state machines from sequential code. 2018.
[17] Michael D May and Roger M Shepherd. System for executing, scheduling, and selectively linking time dependent processes based upon scheduling time thereof, January 29 1991. US Patent 4,989,133.
[18] Zedboard zynq-7000 arm/fpga soc development board. https://store.digilentinc.com/zedboard-zynq-7000-arm-fpga-soc-development-board/. [Online; accessed May 2018].
[19] Zynq-7000 ap soc family product tables and product selection guide. https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf. [Online; accessed May 2018].
[20] Simple mips sme implementation. https://github.com/kenkendk/sme/tree/state_examples/src/Examples/SimpleMIPS. [Online; accessed July 2018].
[21] Carl-Johannes Johnsen. "implementing a mips processor using sme". In Jan Bækgaard Pedersen, Kevin Chalmers, Jan F. Broenink, Brian Vinter, Kevin Vella, and Peter H. Welch, editors, *"Communicating Process Architectures 2017"*, pages 197 – 224. "IOS Press, Amsterdam, The Netherlands", "August" 2017.
[22] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.
[23] Ccc hackathon: occam-pi on the raspberry pi. http://concurrency.cc/2012/06/12/hackathon-occam-pi-raspberry-pi. [Online; accessed May 2018].

A. Occam programs

```
1  PROC fib(CHAN OF BYTE screen)
2    BYTE a, b, i, tmp :
3    SEQ
4      a := 1
5      screen ! a
6      b := 1
7      screen ! b
8      i := 0
9      WHILE i < 10
10       SEQ
11         tmp := a + b
12         a := b
13         b := tmp
14         screen ! tmp
15         i := i + 1
16 :
```

Listing 3: fib.occ

```
INT FUNCTION inc(VAL INT num)
  INT computed:
  VALOF
    SEQ
      computed := num + 1
    RESULT computed
:

[3]BYTE FUNCTION pretty(VAL BYTE val)
  INT i:
  BYTE tmp, factor:
  [3]BYTE res:
  VALOF
    SEQ
      tmp := val
      factor := 100
      i := 0
      WHILE i < 3
        SEQ
          res[i] := (tmp / factor) + 48
          tmp := tmp REM factor
          factor := factor / 10
          i := i + 1
    RESULT res
:

PROC main(CHAN OF BYTE screen)
  INT i, j, result:
  BYTE val:
  [3]BYTE pret:
  SEQ
    i := 0
    WHILE i < 10
      SEQ
        result := inc(i)
        val := BYTE result
        pret := pretty(val)
        j := 0
        WHILE j < 3
          SEQ
            screen ! pret[j]
            j := j + 1
        screen ! 32
        i := i + 1
:
```

Listing 4: callnreturn.occ

```
[3]BYTE FUNCTION pretty(VAL BYTE val)
  INT i:
  BYTE tmp, factor:
  [3]BYTE res:
  VALOF
    SEQ
      tmp := val
      factor := 100
      i := 0
      WHILE i < 3
        SEQ
          res[i] := (tmp / factor) + 48
          tmp := tmp REM factor
          factor := factor / 10
          i := i + 1
    RESULT res
:

PROC inc(CHAN OF INT in, CHAN OF INT out)
  INT i, tmp:
  SEQ
    i := 0
    WHILE i < 10
      SEQ
        in ? tmp
        tmp := tmp + 1
        out ! tmp
        i := i + 1
:

PROC gen(CHAN OF INT out)
  INT i:
  SEQ
    i := 0
    WHILE i < 10
      SEQ
        out ! i
        i := i + 1
:

PROC printer(CHAN OF INT num, CHAN OF BYTE output)
  INT i, j, tmp:
  BYTE val:
  [3]BYTE pret:
  SEQ
    i := 0
    WHILE i < 10
      SEQ
        num ? tmp
        val := BYTE tmp
        pret := pretty(val)
        j := 0
        WHILE j < 3
          SEQ
            output ! pret[j]
            j := j + 1
        output ! 32
        i := i + 1
:
```

```
60
61 PROC main(CHAN OF BYTE keyboard, screen, error)
62   CHAN OF INT intermediate, result:
63   PAR
64     gen(intermediate)
65     inc(intermediate, result)
66     printer(result, screen)
67 :
```

Listing 5: conc.occ

```
1  PROC Prefix (VAL INT n, CHAN OF INT in, out)
2    SEQ
3      out ! n
4      INT value:
5      SEQ i = 0 FOR 1000000
6        SEQ
7          in ? value
8          out ! value
9  :
10
11 PROC Delta (CHAN OF INT in, CHAN OF INT out.0, out.1)
12   INT value:
13   SEQ
14     SEQ i = 0 FOR 1000000
15       SEQ
16         in ? value
17         PAR
18           out.0 ! value
19           out.1 ! value
20     in ? value
21 :
22
23 PROC Succ (CHAN OF INT in, CHAN OF INT out)
24   INT value:
25   SEQ i = 0 FOR 1000000
26     SEQ
27       in ? value
28       out ! value + 1
29 :
30
31 PROC Consume (CHAN OF INT in, CHAN OF BYTE out)
32   INT value:
33   SEQ
34     SEQ i = 0 FOR 1000000
35       in ? value
36     out ! 'D'
37     out ! 'o'
38     out ! 'n'
39     out ! 'e'
40 :
41
42 PROC ComsTime (CHAN OF BYTE keyboard, screen, error)
43   CHAN OF INT a, b, c, d:
44   SEQ
45     PAR
46       Prefix (0, b, a)
47       Delta (a, c, d)
48       Succ (c, b)
49       Consume (d, screen)
50 :
```

Listing 6: commstime.occ

TODO NOTE: interrupt 7 schedule 1 to 9 deschedule 2 to 10 run 1 to 8 start 1 to 9

Table 5. Instructions implemented in the SME Transputer. The clock ticks are without considering possible delay from either external memory or from external communication.

Instruction	**SME**	**Original** [13]
adc	1	1
add	1	1
ajw	1	1
alt	2	2
altend	2	4
altwt	(2/(5-13)[1])[2]	5/17[2]
and	1	1
bcnt	1	2
bsub	1	1
call	5	7
cctn1	1	3
cj	1	2/4[3]
csub0	1	2
cword	1	5
diff	1	1
disc	1/5[4]	8
diss	1/3[4]	4
div	1	39
enbc	1/3[2]	7/5[2]
enbc3	1/3[2]	N/A
enbs	1/2[2]	3
enbs3	1	N/A
endp	4/(4-12)[15]	13
eqc	1	2
gajw	1	2
gcall	1	3
getpas	1	N/A
gt	1	2
in	(4-12)[1]+2w	2w+19
j	1	3
ladd	1	2
lb	2	5
ldc	1	1
ldiff	1	2
ldl	2	2
ldlp	1	1
ldnl	2	1
ldnlp	1	1
ldpi	1	2
ldpri	1	1
lend	5/3[7]	10/5[7]
lmul	1	33
lshl	1	n+3/n-28[8]
lshr	1	n+3/n-28[8]
mint	1	1
move	2w	2w+8
mul	1	38
nfix	1	1
norm	1	n+5/n-26/3[9]
not	1	1
opr	varies	varies
or	1	1
out	(4-12)[1]+2w	2w+19
outbyte	(5-13)[1]	23
outword	(5-13)[1]	23
pfix	1	1
prod	1	b+4
rem	1	37
resetch	3	3
ret	2	5
rev	1	1
runp	2-10[1]	10
sb	2	5
seterr	1	1
sethalterr	1	1
shl	1	n+2
shr	1	n+2
startp	3-11[1]	12
sthb	1	1
sthf	1	1
stl	2	1
stlb	1	1
stlf	1	1
stnl	2	2
sttimer	1	1
stoperr	1	2
stop	3-11[1]	11
sub	1	1
sum	1	1
testerr	1	2/3[10]
testpranal	1	2
wcnt	1	5
wsub	1	2
xdble	1	2
xor	1	1
xword	1	4

B. Implemented instructions

This appendix states all of the implemented instructions and all of the non implemented instructions. The instructions implemented in the SME Transputer, their required amount of clock ticks and the original minimum amount of clock ticks, can be seen in Table 5. At the moment of writing, 86 instructions are implemented in the SME Transputer, while lacking the remaining 73. As such, ∼54% (∼78% if we disregard floating point instructions) of the instruction set is implemented.

Note: not all of the instructions are described in this paper (E.g. the ALT instructions). This is due to the project having been under development (and still is).Additional instructions have been added since the first revision of the paper was submitted.

The following instructions have not been implemented in the SME Transputer:

`bitcnt`	`bitrevnbits`	`bitrevword`	`cflerr`	`clrhalterr`
`crcbyte`	`crcword`	`csngl`	`dist`	`dup`
`enbt`	`fmul`	`fpadd`	`fpb32tor64`	`fpchkerr`
`fpdiv`	`fpdup`	`fpentry`	`fpeq`	`fpgt`
`fpi32tor32`	`fpi32tor64`	`fpint`	`fpldnladddb`	`fpldnladdsn`
`fpldnldb`	`fpldnldbi`	`fpldnlmuldb`	`fpldnlmulsn`	`fpldnlsn`
`fpldnlsni`	`fpldzerodb`	`fpldzerosn`	`fpmul`	`fpnan`
`fpnotfinite`	`fpordered`	`fpremfirst`	`fpremstep`	`fprev`
`fprtoi32`	`fpstnli32`	`fpstnlsn`	`fpsub`	`fptesterr`
`fpuabs`	`fpuchki32`	`fpuchki64`	`fpuclrerr`	`fpudivby2`
`fpuexpdec32`	`fpuexpinc32`	`fpumulby2`	`fpunoround`	`fpur32tor64`
`ldinf`	`ldtimer`	`lsub`	`lsum`	`move2dall`
`move2dinit`	`move2dnonzero`	`move2dzero`	`postnormsn`	`roundsn`
`saveh`	`savel`	`talt`	`taltwt`	`testhalterr`
`tin`	`unpacksn`	`wsubdb`		

[1] min-max. The range depends on the state of the scheduler.
[2] ready / not ready
[3] if not taken / if taken
[4] if `b != 1` / if `b == 1`
[5] if the last process to end / if not the last process to end
[6] internal / external
[7] loop / exit
[8] $n < 32$ / $n >= 32$
[9] $norm < 32$ / $norm >= 32$/$norm = 64$
[10] no error / error

A.1.2 *Teaching Concurrent and Distributed Programming With Concepts Over Mathematical Proofs*

Teaching Concurrent and Distributed Programming With Concepts Over Mathematical Proofs

1st David Marchant
Niels Bohr Institute
University of Copenhagen
Copenhagen, Denmark
david.marchant@nbi.ku.dk

2nd Carl-Johannes Johnsen
Neils Bohr Institute
University of Copenhagen
Copenhagen, Denmark
carl.johnsen@nbi.ku.dk

3rd Brian Vinter
Neils Bohr Institute
University of Copenhagen
Copenhagen, Denmark
brian.vinter@nbi.ku.dk

4th Kenneth Skovhede
Neils Bohr Institute
University of Copenhagen
Copenhagen, Denmark
kenneth.skovhede@nbi.ku.dk

***Abstract*—This paper describes how a concept-based approach to teaching was used to update how concurrent and distributed systems were taught at the University of Copenhagen. This approach focuses on discussion to drive student engagement whilst fostering a deeper understanding of the presented topics compared to more traditional displays of crude facts. The course is split into three sections: local concurrency, networked concurrency, and concurrency in hardware. This allows for an easier student journey through the course, as they are introduced to all core concepts in the first section, then have them reinforced in greater detail in the subsequent sections. Finally, the experience gained in updating this course is presented so others attempting to do similar may learn from it.**

***Index Terms*—Concurrent, Parallel, CSP, SME, ZeroMQ, Teaching, Concepts**

I. Introduction

Scientific data processing is a considerable computing task that necessitates the use of High Performance Computing. Despite the presence of various libraries[1] to manage all aspects of parallel programming, knowledge of how a distributed system works is still essential to make full use of parallel hardware. This can present a problem as parallelisation is not a trivial topic, and scientists running experiments may not have an extensive background in computer science or programming.

At The University of Copenhagen, distributed computing courses are taught by the eScience group, part of the Niels Bohr Institute. These courses are mostly taught to physics students, who need this knowledge so that they can set up experiments to make use of parallel computing. A new *Concurrent and Distributed Systems* course has been introduced to replace an older course. The previous course had a theoretical focus, with rigid facts rather than engaging concepts. This would turn students off and give only a surface level of understanding.

This paper proposes replacing a fact-based approach to teaching with a concept-based approach. This will be broken down into 3 linked stages, local concurrent programming, distributed concurrent programming, and concurrent programming at a hardware level. In this paper these areas are demonstrated using PyCSP, PyZMQ and SME, but any similar library could be used in their place as it is the shared concepts beneath them that are important. These underlying concepts, such as deadlock, race conditions, and distributed states are the core of concurrent programming and so are the true learning goals.

Ultimately this was partially successful, with students responding well, though at great time investment on the part of the lecturers. Despite this, the techniques demonstrated their validity and could be applied to other similar courses going forward.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765604.

[1]As illustrated by, but not limited to: CUDA [1], OpenMP [2], CSP [3], ZeroMQ [4], and MPI [5]

II. Objectives

This paper is an account of, and reflection on, teaching carried out within the *Concurrent and Distributed Systems* course. This is done with 3 goals in mind. In no particular order these are:

1) To record how a new approach to teaching concurrent systems design was put into practice.
2) To evaluate and reflect on how this new methodology worked.
3) To recommend for similar future courses what could be carried forward and what should be changed.

All of these stated objectives will be considered in the context of teaching parallel programming to non-computer scientists. Within this paper 'computer scientists' are those students whose primary area of study falls within computing. Any other students are 'non computer scientists' and are assumed to have no more than a passing familiarity with programming.

III. Background

The Niels Bohr Institute is a research institute specialising in astronomy, biophysics, condensed matter physics, geophysics, quantum physics, and particle physics. In addition to this, the eScience department conducts its own research into scientific methods using computers. As well as maintaining physical hardware and software to support the other departments, it is also responsible for teaching High Performance Computing. A new Masters level course in concurrent and distributed systems was designed to be run in the academic year 2018-2019 with the authors as the teaching team. This course was

titled *Concurrent and Distributed Systems* and was intended as a refreshed version of older courses in similar areas that were now deemed insufficient.

The *Concurrent and Distributed Systems* course used as a basis for this paper was taught at the University of Copenhagen, from the 19th of November 2018, until the 27th of January 2019. It was taught with 2 lecture slots a week, each 1.45 hours in duration. There was a weekly practical session also 1.45 hours in duration. 3 assignments were given out over the course, each lasting roughly 2 weeks and a final examination was given in the form of a take home exam with students having 11 days to complete it.

IV. Facts VS Concepts

The base assumption for the new course was that focusing on the *concepts* of distributed computing was preferable to focusing on *facts*. By *facts*, *mathematical proofs*, *information*, or *technical details* we refer to pieces of knowledge that are (probably[2]) true. It is often fundamental to the subject area, can be easily rote memorised [7], and can easily be expressed in a book, lecture or other one way communication. The syntax of a built in function, the clock speed of different machines, particular communication protocols, would all be examples of *data*, *information* or *technical details*. For the rest of this paper these shall all be referred to as *facts*.

Ideas or *concepts* refer to pieces of knowledge that are not necessarily verifiable. These are the grand approaches that can emerge from multiple facts and used to explain or guide systems. For example, consider system architectures, design approaches, or algorithm structures. All of these demand engagement from someone to understand and cannot be meaningfully rote memorised. These depend on many facts to be understood, and our understanding of them is constantly morphing and being updated. They can also be used to extrapolate new areas of knowledge and understanding [7]. For the rest of this paper these shall be referred to as *concepts*.

Concepts such as object-orientation are already widely taught and understood within computing, demonstrating their utility within computing education. This is especially important as the difference between sequential and parallel programming is not one of technical details, but one of thought. How you approach a parallel problem is fundamentally different to a sequential one, with a completely different structure and thought process behind it. Put another way, the problems of parallel are concepts, rather than facts, and so they require teaching focused on those concepts rather than on facts [8]. Therefore, even though *Concurrent and Distributed Systems* is aimed at potentially novice computer scientists, the teaching should be concept-based.

We could say that facts are more basic than concepts in a learning context, as facts are specific and cannot broadly be applied. However, facts are still required to act as a base for understanding concepts, and so both concepts and facts should be taught together. This means that when this paper refers to teaching concepts rather than facts, it is meant that concepts should be emphasised over facts, but not that facts should be ignored entirely as they form a necessary part of students learning.

In response to all this it was decided to start from scratch with a new series of lectures. The university format meant that we had to keep to 2 lectures a week with a practical. As concepts are much more difficult to explain than facts, as they tend to require a back and forth discussion in order to teach [8], the lecture format may have presented a problem. However, the class was expected to be small enough that the necessary discussions could take place. Note that despite all that has been said, facts are still extremely important. Concepts without facts become meaningless as they cannot be applied to the external world. It was hoped then, that the resulting slides could communicate the necessary concepts of concurrent and parallel systems, with enough supporting facts so as to be understandable.

V. Course Goal

Concurrent and Distributed Systems was aimed primarily at non computer scientists with limited programming experience. This is justified by the increasing requirement for parallel processing by scientists in all areas of physics studied at the Niels Bohr Institute [9]. In addition, students may be involved in designing scientific instruments or experiments which contain concurrent systems. Although there exist libraries that purport to take care of all parallelisation for the user, such as CUDA [1], MPI [10] or OpenMP [2], these still need a good base of knowledge from the user before they can be fully utilised. It would be possible to run a course that only related the facts of how these systems work, by highlighting specific commands and their expected outcome. However, this would leave students with a very narrow pool of knowledge, specific only to the exact software and problems described in the course. By adopting a concept-based approach, where instead the base concepts of distributed programming are explored and understood, non computer scientists can claim a theoretical understanding of parallel programming. This should suffice for them to effectively use any of these preexisting systems, and potentially even start designing their own custom implementations.

As most undergraduate physicists are not expected to be running big enough experiments to justify the use of high performance systems, *Concurrent and Distributed Systems* was set at a Masters level with classes expected to have between 6 and 12 students enrolled. The sought after learning objective would be some measurable understanding of asynchronous concurrent and distributed systems. That is, a system comprised of multiple processes, where the order of processing is not and cannot be determined at the start of processing. By the end of the course students should be able to design and implement concurrent and parallel systems in both hardware and software. These systems should be robust

[2]At the very least it is expected to be true, even if it is up for debate. Within the field of Epistemology it could be said that these statements are justified beliefs, that are true as far as can currently be determined. Consider this in the context of the works of Edmund Gettier and others. [6]

to common design problems such as deadlock, livelock and race conditions. Students should also be able to demonstrate their systems correctness using diagrams and descriptions.

VI. Selecting Course Content

To introduce and reinforce universal concepts of parallel computing, the decision was taken to break the subject down into three smaller sections. These could then slowly introduce concepts, and illustrate their universality within distributed programming. As these concepts would occur repeatedly through the three sections in increasing depth, it was hoped that they would be further reinforced. Starting with local parallelisation would be logical, as it meant that problems such as networking could be ignored. This allows for the introduction of the base concepts such as determinism, race conditions, deadlock, livelock, compartmentalisation, as well as identifying what sort of tasks are suitable for parallel or not.

All scientific programming within the Niels Bohr Institute is taught in Python[3], where possible. This meant that the underlying language was already set, and that some familiarity with the language could be assumed. To illustrate parallel processing concepts in the first course section, PyCSP[4] [13] was selected as all members of the teaching team were already familiar with it. It is worth noting that PyCSP has been taught in related courses previously, and has been found to be a very good introduction to concurrent and distributed concepts, even for novices [14]. Sticking with what the teaching team were familiar with was seen as important as it meant all members had already built up a body of knowledge designing systems using PyCSP. It was hoped that this would mean that the teaching team could adequately answer questions without having to rely on slides or textbooks to do the heavy lifting. This would be essential if we were to avoid dry lectures of reading technical information to students, but were instead to encourage conversation and interaction.

From a localised system the next logical step was a distributed one. These would still be using the same concepts from before, but now with added challenges such as the impossibility of global memory. This could be done using PyZMQ[5] [15] as again, it is Python based, simple to learn, and the teaching team were already familiar with it. Finally it was felt that students should have some introduction to physical devices that could be used to run a distributed system, such as in an *Internet of Things* device. This was as the problems that would be introduced in the first two sections are just as much problems in hardware as in software. To illustrate this, FPGAs[6] were used, with SME[7] [16] as the code base.

It was decided that it might help students to keep motivated and interested in the material if they could relate it to their own interests or research [7]. As at this point it was known that the FPGA boards were to be used, and that the students would build a system on the board, it followed that this system could be something scientific. A simple sound locator system was decided upon. This could be used in all course sections, so that all the assessments are tied together by a common thread. In the first section the students design a PyCSP system to process multiple microphones listening for sounds to determine the direction of a sounds source. In the second section they design a networked system, where they each link together their individual systems. In the third section they then program an individual microphone.

The hope is that these three sections will cover all essential areas of knowledge for the students, and assumes relatively little background knowledge. By starting on local systems and working up to more and more decoupled examples it is also intended that students are slowly introduced to the topic without them being hit with incomprehensible topics all at once. The students journey through the course should be simplified greatly as in the first section performance and efficiency are secondary concerns to robustness and ease of understanding. As the students continue through the course they are introduced more and more to requirements of performance and working within the already introduced concepts to get more processing done in less time.

VII. Software and Hardware

Software and hardware infrastructure was needed to run the course effectively. Students would need Python, PyCSP, ZeroMQ, and SME. These libraries and their dependencies could be time consuming to set up per individual, setting them up would not be particularly informative to the students. To get around this, JupyterLab Notebooks were used as a learning environment. JupyterLab Notebooks are documents accessible online, capable of displaying and running live code. They are centrally stored and so can have all dependencies pre-loaded onto them, meaning all students can easily start from the same point, with a complete system. For hardware, the PyNQ [17] board was selected as they contained a FPGA chip, had the necessary hardware to run Python scripts, and were reasonably priced.

VIII. Previous Teaching Material

For most of the course there already existed relevant lecture slides from previous similar courses. Naturally, these needed slight editing to fit with the new course, but in the case of section one, on PyCSP, a complete rework was required.

[3]This is due to the utility of Python for scientific analysis [11] and research, as well as its wide adoption throughout the scientific community. [12]

[4]PyCSP is a Python specific implementation of Communicating Sequential Processes (CSP) [3], a formal definition of how a system could be split up into several independent sub-sections and how those sub-sections would communicate. Other implementations exist in other languages, in varying states of completeness. The underlying principles between each are shared however.

[5]PyZMQ is a python specific implementation of ZeroMQ (also known as ØMQ) [4]. It is a library for easy asynchronous communication over a network.

[6]Field Programmable Gate Arrays (FPGA) are essentially programmable circuit boards, allowing for the implementation of many different hardware circuits using only one device, as opposed to expensive, custom made chips.

[7]Synchronous Message Exchange (SME) is a CSP derived language for programming FPGA boards. It compiles into VHDL and is designed to be more user friendly and quicker to program.

Plenty of teaching material was available [14], in the form of slides, books and workbooks. Each of these were considered in turn but rejected for a variety of reasons. These materials were often for a different length of course, meaning serious cutting or padding would be needed. As the different resources available were also from disparate sources they were all designed inconsistently, so even more editing would be needed to bring together any slides into a common visual language.

Aside from these small, practical considerations, it was also felt that the available material relied far too much on reading complex information off of slides as a method of teaching. Slides would be either extensive blocks of code, complex diagrams, or paragraphs of text. Often times, mathematical proofs of correctness were included and run through, proving the validity of a certain approach. This may be correct, but that level of detailed understanding is unnecessary for most students, especially non computer scientists. The material did not support interaction beyond asking simple memory recall questions rather than discussion and could be said to be entirely fact-based, as discuss in section IV.

It was felt that a better approach would be a more discussion based one [18], with a focus on the ideas and concepts behind CSP rather than on the technical details [19]. This should be especially possible given that the course was set at a Masters level and so should have students who can engage in a topic more in terms of ideas rather just simple facts. The expected small enrolment also meant that facilitating informal discussions rather than a strict lecture should be possible. Finally, a conceptual understanding would be preferable for non computer scientists as numerous libraries and systems exist to automate the generation of parallel code. It is the theoretical understanding behind these libraries that the non-computer scientists need.

IX. Goals for the New Material

To design the new material, objectives had to be set against which it could be designed, and success judged. These goals were devised with the aim of using an inductive approach to teaching [20]. These are presented below, with higher priority goals being at the top of the table. The material should:

	Goal
G1	Facilitate the teaching of concurrent and parallel concepts.
G2	Support the presented concepts with facts.
G3	Encourage student engagement in the class through exercises and discussion.
G4	Provoke questions and discussion from the students.
G5	Enable the teacher to explain in their own words, rather than relying on technical definitions.
G6	Be clear and easy to understand.
G7	Be reusable in subsequent courses, even by others not on the current teaching team.

Most of these goals should be self explanatory and so will not explained at length. G3 and G4 may require clarification though. Ultimately they both are the same idea, to focus more on conversation about a topic, rather than a direct lecture on it. It has been split into two goals to show that this is a two part process. In G4 we need to make sure that the teacher is accepting of this style in their teaching, whilst in G3 we should encourage the students to be interacting with the lesson. After all, if only one person is trying to start a conversation, it will not happen easily. Note the use of the words 'teacher' and 'lesson' rather than 'lecturer' and 'lecture'. This is done to suggest that the person standing at the front is not merely talking *at* the students, but *with* them [19], and does not denote any further difference.

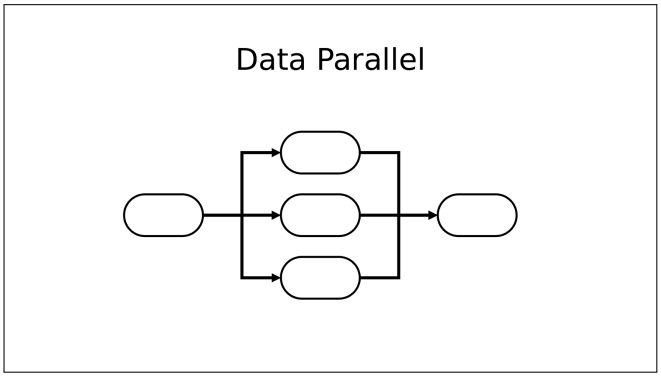


Fig. 1. Lesson 1, slide 23. This demonstrates the simple, clean design for the slides with the bare minimum of information. This diagram is intended as a conversation aid, rather than an explanation on its own.

X. Designing New Material

With these goals in mind, eight sub-topics were selected for section one of *Concurrent and Distributed Systems*[8]. These could then roughly align to the 4 lectures given in this section, with each lecture split by a small break, forming 8 half-lecture slots of roughly 45 minutes. In that time new concepts should be introduced, the facts to support them presented, and the resulting discussion engaged in. This is a lot to do, so presentations were kept to around 25 slides. Sentences on slides were kept short and well spaced. Diagrams were plain and presented without accompanying text on the slide. For examples, see figures 1 and 2.

Both of these slides are typical of the newly made slides for this course, as both of them provide one or two key facts, and very little else. This was done deliberately with the aim of fostering conversation. By having so little information on the slides the lesson could not turn into a session of just reading information from slides, as there simply is not enough information to fill the time by doing so. This approach would also be coupled with regular questions from the teacher so as to foster more dialogue than in a more traditional lecture.

[8]These were *parallel design problems*, *an introduction to PyCSP*, *deadlock and livelock*, *parallel system design principles*, *determinism and race conditions*, *compartmentalisation*, *additional CSP concepts*, and *network communication*. For a complete course description and to see the contents of each section, all course materials are available at [21]

Senders and Receivers

- Senders and Receivers allow us to avoid deadlock.
- As long as no Senders and Receivers interact in a loop, deadlock cannot occur.
- Livelock *might* still be a problem but its actually quite hard to get that to occur without trying to (famous last words...).

Fig. 2. Lesson 3, slide 15. Where text must be used it is kept to a minimum. As before, these sentences are intended as aides to what is currently being discussed rather than a lesson on their own. Even the written text is written conversationally.

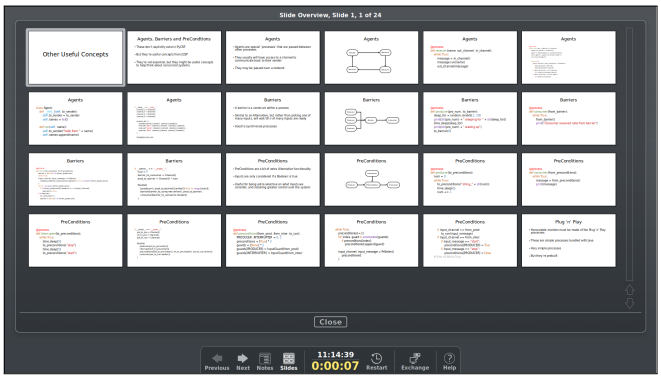


Fig. 3. Lesson 7, tiled view. This is demonstrated in LibreOffice Impress, but many other slideshow programs have similar features

The design of the slideshow itself also changed, with lessons being divided into sub-categories and where possible, not depending on a specific ordering to make sense. When displaying the slides in a lecture a tiled slide selector could be used to jump from slide to slide in a non-deterministic manner, and so follow the current direction of discussion. This is illustrated in figure 3. The simple design of the slides also helped here as it meant the correct slide is still readable on a laptop screen when in tiled view and so can be selected without difficulty.

To achieve the goal of more concepts and more discussion, regular exercises were introduced [19]. This is perhaps best exemplified by lecture 4, on designing a concurrent and parallel system that is only 5 slides long. The first slide is a title slide, while the second sets the exercise of designing a system. The third illustrates some discussion points that might occur as the students solve the problem and acts as an initial guide to students if they don't know how to start. The fourth introduces more exercise as it expands the initial problem. The final slide acts as a reinforcement to the core concept of this exercise, explicitly stating some supporting facts to ideas hopefully encountered. These slides would last no more than a few minutes, and exemplify every lectures role more as support

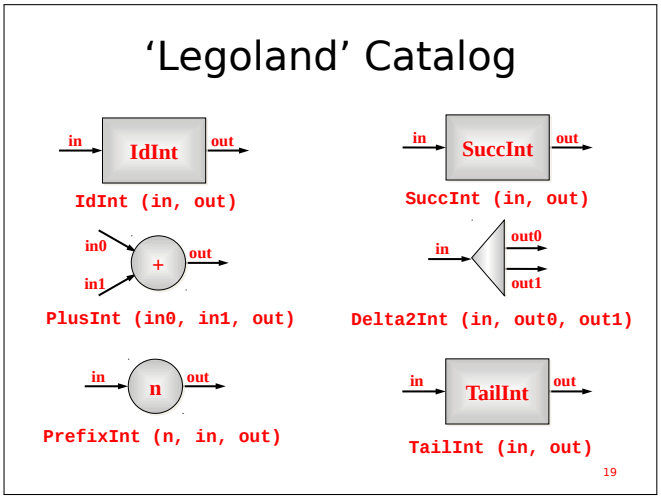


Fig. 4. An example slide from the previously used material for similar courses. This example has been picked as it is superficially similar to the newly designed slides, yet would foster a different style of teaching.

to a discussion rather than as the lesson itself.

The overall design of the slides compares favourably with the previous materials as the new ones are clearer, more condensed displays of relevant information. They act as effective notes for students as the minimalist design helps ensure that the information that is left makes effective, if brief notes as to the key supporting facts for the lessons concepts. As a comparison a sample slide from the older material is shown in figure 4. This slide appears superficially similar the a new ones, but it only displays a variety of available inbuilt CSP cookie-cutter processes[9]. These are rarely used in actual practice and so would be an example of teaching facts for facts sake as they do not lead to any wider conceptual understanding. As a result of this, the slide fosters little discussion beyond a description of the displayed information

One final brief note on the design of the new slides is that they each use the same visual language from the very beginning to the very end. Diagrams were expressed in the same way consistently, meaning that students only needed to decipher one way of reading diagrams. As there is no formal UML definition for network diagrams, previous materials visual languages could change dramatically between diagrams. Focusing on concepts over ideas may be hard enough for some students to follow, so these additional complications should be minimised by using the same style throughout. The designed slides, along with all other course material is available at a public Git repository [21].

XI. Lectures and Practicals

When teaching the lectures, to foster an environment of discussion, affairs were kept fairly informal. For instance, a

[9]These are provided processes that each perform some very basic functionality such as adding together two input numbers. They can be combined together to form more advanced functionality. In practice this is not done as the overhead from having so many processes makes for a bloated and slow system.

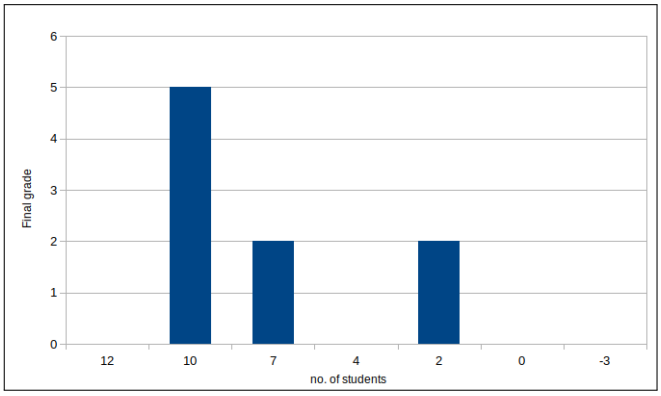


Fig. 5. Final results. Note that only nine students are shown as one is currently in the process of re-sitting the final assignment and so has not yet received a final grade. Initially they received the grade '0'.

conversational tone would be adopted throughout the lesson, with regular comments and observations. Questions from the students should be allowed as soon as they occurred to the students, rather than waiting for the end or some gap in explanations. It is also important that the teacher is open to admitting what they don't know as part of encouraging a conversation [19].

Practicals were relatively unplanned compared to lectures as again they were expected to be more conversational by their very nature. As students should already expect this from a practical rather than a lecture, less effort was needed to foster this specific atmosphere. There was no specific plan set for individual practicals and they were intended mostly as support sessions for help with assignments and troubleshooting any technical issues that emerged.

XII. Results

The course ran as expected with no major issues. Ten students enrolled on the course, though unexpectedly, eight were computer science students. This meant that a higher standard of background computing knowledge could be assumed. There was a reasonable spread of grades after the final exam, as shown in figure 5. These results are certainly higher than the expected bell curve, but with a small sample size and most students having a higher than expected familiarity with the topic, is not surprising or concerning.

The PyNQ boards caused minor software trouble throughout the course. These technical difficulties were never sufficient to derail the course and have now been ironed out so should not provide difficulty in future courses. However, the true problem with them was that they served as a distraction in students reports on their system. When asked to comment on the shortcomings of their designs, most would default to listing simple technical problems, rather than engaging critically with the conceptual problems. Most were able to engage with concepts once verbally prompted. The inability to do so in their report is potentially down to a failure to set expectations at the beginning of the assignment, rather than a fundamental problem of approach. Greater care should be taken in future to communicate what is expected from assignments, with explicit instructions in the assignment handouts.

Students responded positively in their end of course surveys [21]. Most seemed happy with the quality of teaching, but thought that the workload was too light. This may be due to the students being more familiar with the course contents than was anticipated. Students also felt that better use could have been made of the practical sessions. They were intended as informal help-sessions, and so were only lightly attended. This was expected, but a more defined structure with some set exercises may have helped give non-computer scientists more hands on time to get familiar with programming. It also would have filled out the workload. Students in related courses have also reported that having mandatory, short, defined workshop exercises every week helped them learn the topics. Workshops would also particularly suit the presented style of teaching as concepts are developed through practice and playing around with the presented facts. The workshop materials and any work produced in them would also inherently produce good revision and reference material for students if they needed to revise.

Preparing the teaching materials took roughly two weeks of work time spent on just 4 hours of lessons. The sheer length of time taken may be a function of the teachers relative inexperience, but it nevertheless illustrates the extensive planning and preparation required for such an involved teaching style. Naturally, this will reduce with practice but should be kept in mind by any new teaching teams attempting to replicate this approach.

The teaching team also feels that too much material was covered, particularly in the second section. Several types of communication protocols were discussed at length but never used or recommended. More time spent on FPGAs would have better served the students. This would help with the final point of criticism for the course. Throughout the course, but particularly in relation to hardware, the teaching team overestimated the students familiarity with programming concepts. During hardware setup they could be asked to define their board's IP address but no specific instructions on how to do this were provided at first as it was simply assumed that anyone could do this. Again, this can easily be addressed in future iterations of the course through increased support in practical sessions and more explicit instructions/guides in assignment hand outs.

XIII. Reflections by the Teaching Team

It is reasonable to conclude that the course went well. An expected number of students passed, the core topics have been taught, and students responded positively in their end of course surveys. To further evaluate the success of the course, the goals presented in section IX should be considered. These evaluations will rely heavily on the personal reflection by the primary author, who taught this section of the course. Further from student questionnaires and colleagues has also been added where possible and appropriate. Therefore, this section should not be considered scientific and unbiased. However, it was felt that the personal experience of an honest attempt at

trying to implement this style of teaching may be illuminating to others considering doing the same, especially as scientific guidance does not, and cannot, meaningfully exist for this [20].

A. G1: Facilitate the teaching of concurrent and parallel concepts

As the most important objective, G1 was kept in mind through the whole process, and was the reason that most previous teaching materials were replaced. Most sections worked extremely well, such as lesson 5 on Determinism and Race Conditions. Almost all of the slides in this lesson are prompts to student interaction, with students guessing at the outcome of some simple programs. These examples illustrate what determinism is so that hopefully by the time a quick slide explaining it through facts is shown, they already have formed the core concept, that can then be reinforced by the necessary facts.

However, several slides through the course turned out more full of explicit information than was first expected, and so became the primary teaching tool within their lessons. This is suspected to have led to less conceptual understanding from the students as the concepts communicated in those lessons did not appear in the final assessment submissions. Greater discipline is needed in limiting facts on slides, and a rule of thumb such as 'only 3 facts per slide' would help keep slides short and force discussion to the fore.

The failure of some slides illustrates the success of others. As mentioned in sections IV and X, concepts require facts to support them and so all facts on the slide should support a concept, rather than being a fact in and of itself. As most slides fostered conversation and with them conceptual understanding (as evidenced by the concepts being well understood in assignments) it is demonstrated that the presented teaching techniques are suitable for this goal.

B. G2: Support the Presented Concepts with Facts

Where the presented concepts fell down however was in the application to parallel programming as a whole. Once the course moved to topics other than CSP it seemed that several of these core concepts were forgotten, or it was not realised by the students that these were broadly applicable concepts rather than just relating to CSP. This is perhaps similar to the shape of the earth problem encountered by Vosniadou and Brewer [22]. The key similarity here is that the children and students appeared to have a complete understanding of the topic when first asked, but actually did not upon closer inspection. Vosniadou and Brewer suggest that the children did not have enough supporting facts for them to form an accurate conception of what the earth looks like when it is said to be round. It could be that a similar problem has occurred here, with insufficient examples provided illustrating the broader applications of the core concepts. Surprisingly, this issue was not limited to the physicists and even appeared with the computer scientists, who were expected to have an existing broad understanding of computing and so be able to apply concepts more accurately. Care should be taken in future to use a wider range of examples to demonstrate that these concepts are bigger than they might otherwise appear.

C. G3: Encourage student engagement in the class through exercises and discussion

Student engagement was fostered throughout the course through the use of exercises, questions, and prompts, rather than as a defined stage within the teaching cycle [7][10]. At certain points this was hard to keep up, and the teaching fell back on explaining things at the students. lesson 7 in particular suffers from this, as it is just an explanation of some common methodologies that exist in other CSP implementations. This was definitely the least successful lesson with very little seemingly being learned from it, judging on the taught material not being present in any of the students submissions. These topics are not notably more difficult than other lectures, nor were they explained worse than other topics. The lack of interaction was the only notable difference, leading to the conclusion that this is why it stuck less in the student's memory than other lessons.

D. G4: Provoke questions and discussion from the students

Students engaged willingly and consistently in discussion of the presented topics, with engagement from the whole cohort. A good way to foster conversation between the students was to set exercises and put them in groups of 2 or 3. This meant that to complete the exercise students were forced to exchange ideas, especially as the tasks where conceptual in nature, such as to design a system.

Better use could have been made of the practical sessions. As they were mainly conceived as trouble shooting sessions, attendance was low and those that did attend mostly did not interact with each other beyond to socialise. Something more structured could have given more of the students a reason to talk about the subject together, and allow for more time to reinforce how to apply the learnt concepts as discussed in section XIII-B.

It is worth noting that the exercise for the second section required the students to come to a mutual agreement on a communication protocol, so that each of their systems could communicate with each others. This section failed as no common agreement was made by the students despite repeated prompting by the teaching team. It may be that this failure was due to insufficient background being presented, and so students did not feel they had an understanding of where to begin. It may also have been that no student wanted to be the one to suggest a protocol that everyone else would have to follow. This vagueness of problem means it is hard to meaningfully reflect on the issue, and so perhaps this style of assignment is simply best avoided in future.

[10]It is worth noting that the models that present very separate, defined stages do not necessarily intend for them to be implemented as such, and often will explicitly state as such.

E. G5: Enable the teacher to explain in their own words, rather than relying on technical definitions

Similar to section XIII-D, this goal was mostly achieved. The slides were bare-bones so that they could not simply be read out, and most topics were explained ahead of displaying the relevant slide. This meant that a personal explanation, usually delivered in plain English acted as the primary introduction to a topic, with the defined points of a slide only introduced at the end to reinforce what was already said. The text that was on the slides also acted as memory prompts so that once an effectively ad-libbed explanation was complete, the slides could be checked to confirm that all essential points had been hit.

F. G6: Be clear and easy to understand

This informal approach meant that explanations or slides could be rather opaque. However, students seemed to follow along at the expected rate, and understood what was being said. This may be down to most student being computer scientists however, and so would potentially more familiar with this subject matter. I would expect that this problem could mostly be addressed by further practice at explaining the subject, and is affected mostly by practice at teaching.

G. G7: Be reusable in subsequent courses, even potentially by others not on the current teaching team

Re-usability was mostly forgotten through material creation, and in hindsight would not have been included as a goal. Much of the ambition of this style of teaching was in improvisation and discussion, with pre-made slides potentially discouraging that. The slides were only ever intended as a visual support to what was expected to be said, which may differ considerably from others lecturers. These slides may be a useful guide or starting point for another lecturers slides, but are not expected to be entirely usable by another lecturer without work.

This leads into the major downside of this approach. That being the length of time taken to produce these lectures. Whilst this process should speed up with experience, it will still need to be repeated for each course, making this a very time intensive form of teaching. The ideas presented within this paper about a concept focus are not new[11], yet it is perhaps this time commitment that limits their wider adoption.

XIV. Future Recommendations

It is recommended that *Concurrent and Distributed Systems* continues in future, and that the ideas put forward in its teaching style are iterated upon. The conceptual basis of the course worked well, and PyCSP, PyZMQ, and SME acted as good illustrators of these concepts. The use of established libraries meant that relatively little time could be spent on simple facts and allowed for discussions both broad and deep about the theoretical underpinnings of distributed systems. One of the primary goals was to make a course for physicists who needed to understand parallel programming, and yet only 20% of the eventual enrolment were physicists. The course description for students should better reflect who it is intended for. Additionally, it may be worth advertising the course directly to physics students, perhaps by making sure supervisors within the various departments at the Niels Bohr Institute are aware of its existence, and are mentioning it to those who may benefit from it.

Masters students can still need considerable prompting to engage critically with material rather than just rote learning. Being very clear about this from the beginning is essential. Even greater emphasis should also be taken on student engagement in lectures. This can be done with exercises and should lead to better learning outcomes. In addition it will foster more conversations by their very nature. In particular, small group exercises during class are extremely good at this, especially when the work is conceptual in nature.

In contrast to the success of the small group exercises, the only assignment that required agreement from the whole cohort did not demonstrate any. Each individual solved the problem in their own way. Although some guesses were made as to what caused this failure, no definitive problem could be found. This might demonstrate that long form group work may not be as effective as the short class exercises. It could also demonstrate that a group of 10 is too big to solve a small problem, and so should have been broken up.

Greater use should be made of the practical sessions by including short programming exercises. For example, after a lesson on deadlock, a short exercise to purposefully implement a deadlocking system would be good. These small exercises could be done in the practical sessions further away from assignment hand-ins to keep the students engaged through the quieter parts of the course. It would also allow them to build up their practical experience with facts they themselves have discovered, and to reinforce the concepts they have just been exposed to.

The PyNQ boards proved to be something of a distraction. They had constant minor technical difficulties, even with the JupyterLab Notebooks. With more experience these issues could be ironed out. It may also be that a more abstract series of assignments that did not have to run on a particular board would be better, as any technical problem proved too tempting students to focus on when reflecting on their own solution, rather than reflections on the concepts within their system. Eliminating the physical element may help address this.

All of this leads to following key recommendations for others seeking to introduce an inductive, concept-based approach to teaching parallel systems. In no particular order:

- The student journey of local parallelisation, distributed parallelisation and finally parallel in hardware works well. This is exemplified by PyCSP, to PyZMQ and then SME, and is even an effective introduction for non-computer scientists.
- Student engagement can be fostered through bare-bones teaching materials which force both the lecturer and the students to actively participate throughout the lecture.

[11] Consider that several references on this paper are over a decade old at this point

- Care must be taken when designing materials that they support active discussion. A non-linear slide show that can be adapted to the flow of conversation is a good example of this.
- Group work and practical assignments are essential for allowing concepts to develop and cement in students mind and should be utilised as much as possible.
- Student may be unused to this style of course and assessment, so the expectations on them should be repeated during course descriptions, introduction lectures and assignment handouts.

None of the techniques suggested in this paper (group activities, discussion, bare-bones slides) are unique to parallel computing and could in theory be applied to any subject matter. However, it does take considerable setup time. Despite this, it is hoped that with practice this preparation time will dramatically reduce, making it a more feasible teaching method.

XV. Conclusions

This paper has presented an account of how the *Concurrent and Distributed Systems* course at the Niels Bohr Institute eScience department has modernised the teaching of distributed systems to physicists. This was done because previous courses were insufficient. They were dry courses, full of facts, and with very little student interaction. By focusing instead on concepts supported by facts, discussion and student engagement it was hoped that learning outcomes could be better achieved, with non-computer scientists achieving a deeper level of understanding in the area of parallel systems. Students were introduced to local concurrent programming, followed by networked concurrent programming, and finally concurrent programming on hardware. This allowed for the core concepts to be introduced early, and then applied to different situations, demonstrating their universality.

This course was judged to be a success and so should be refined and repeated in future years. Lessons learned from it could also be applied to other similar courses. In particular, the bare-bones slides and the regular group activities were helpful in fostering an atmosphere of conversation, and elevated the education beyond a discussion of mere facts. However, there was a failure in setting a correct expectation amongst students at the beginning, and the time taken to prepare these lessons was extensive. Regardless, it is still recommended that the experience gained in teaching this course is carried forward, both in this course as it continues next and in other similar courses.

References

[1] "CUDA," https://developer.nvidia.com/cuda-zone, 2019.

[2] "OpenMP," https://www.openmp.org, 2019.

[3] C. A. R. Hoare, *Communicating Sequential Processes*. http://usingcsp.com/cspbook.pdf: Prentice Hall International, 2015.

[4] "ZeroMq," http://zeromq.org, 2019.

[5] "MPI for Python," https://mpi4py.readthedocs.io/en/stable/, 2019.

[6] E. L. Gettier, "Is Justified True Belief Knowledge?" *Analysis*, vol. 23, no. 6, pp. 121–123, 1963.

[7] N. Entwistle, *Teaching for Understanding at University*. Basingstoke: Palgrave Macmillan, 2009.

[8] J. Biggs and C. Tang, *Teaching for Quality Learning at University*. Maidenhead: Open University Press, 2007.

[9] R. Munk, "Teaching Parallel and Distributed Techniques at UCPH through JupyterLab," 7 2019.

[10] B. Barney, "Message Passing Interface (MPI)," https://computing.llnl.gov/tutorials/mpi/, 2019.

[11] J. M. Perkel, "Programming: Pick up Python," https://www.nature.com/news/programming-pick-up-python-1.16833, 2015.

[12] D. Robinson, "Why is Python Growing So Quickly?" https://stackoverflow.blog/2017/09/14/python-growing-quickly/, 2017.

[13] "PyCSP," https://pypi.org/project/pycsp/, 2016.

[14] B. Vinter and M. O. Larsen, "Teaching Concurrency: 10 years of Programming Projects at UCPH," in *Communicating Process Architecture 2017*, K. Chalmers and J. B. Pedersen, Eds. IOS Press, 2017, pp. 135–156.

[15] "PyZMQ," https://github.com/zeromq/pyzmq, 2019.

[16] B. Vinter and K. Skovhede, "Synchronous Message Exchange for Hardware Designs," in *Communicating Process Architectures 2014*, P. H. W. et al, Ed. Open Channel Publishing, 8 2014, pp. 169–179.

[17] "PYNQ: Python Productivity for ZYNQ," http://www.pynq.io, 2019.

[18] M. Prince, "Does Active Learning Work? A Review of the Research," *The Research Journal for Engineering Education*, vol. 93, no. 3, pp. 223–231, 2004.

[19] P. H. Scott, E. F. Mortimer, and O. G. Aguiar, "The Tension Between Authoritative and Dialogic Discourse: A Fundamental Characteristic of Meaning Making Interactions in High School Science Lessons," *The Research Journal for Engineering Education*, vol. 95, no. 2, pp. 123–138, 2006.

[20] M. J. Prince and R. M. Felder, "Inductive Teaching and Learning Methods: Definitions, Comparisons, and Research Bases," *The Research Journal for Engineering Education*, vol. 95, no. 2, pp. 123–138, 2006.

[21] "Concurrent and Distributed Systems course material," https://github.com/PatchOfScotland/ConcurrentAndDistributedCourseMaterial, 2019.

[22] S. Vosniadou and W. F. Brewer, "Mental Models of the Earth: A Study of Conceptual Change in Childhood," *Cognitive Psychology*, vol. 24, pp. 535–585, 1992.

A.1.3 *Lennard-Jones simulation on FPGA using SME*

Accelerating Molecular Dynamics with the Lennard-Jones potential for FPGAs

Alberte Thegler
Niels Bohr Institute
University of Copenhagen
Copenhagen, Denmark
alberte.thegler@nbi.ku.dk

Carl-Johannes Johnsen
Niels Bohr Institute
University of Copenhagen
Copenhagen, Denmark
cjjohnsen@nbi.ku.dk

Kenneth Skovhede
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
skovhede@di.ku.dk

Brian Vinter
Faculty of Technical Sciences
Aarhus University
Aarhus, Denmark
vinter@au.dk

***Abstract*—The requirements for more advanced, longer, and more precise molecular dynamics simulations are greater than ever. Even though we are better at optimizing and have more computational power than previously, there is also a continuing need to make simulations even faster, more reliable, and cheaper to run. In this paper, we are presenting a method for running a molecular dynamics simulation on an FPGA device by using Synchronous Message Exchange. The molecular dynamics simulation presented in this paper is a basic simulation using the Lennard-Jones potential. It is a work in progress, but the results are promising compared to a Python implementation using matrix calculations. We present a proof of concept of an initial solution and its performance provides results that make us believe that a full molecular dynamics implementation would be feasible and competitive.**

***Index Terms*—FPGA, Synchronous Message Exchange, Molecular Dynamics Simulation, Lennard-Jones**

1. Introduction

Simulating the physical properties of atoms and molecules is a problem that has been investigated for decades, and molecular dynamics (MD) simulations are a vital part of this research. It is most commonly used to determine the atoms' trajectory by solving Newton's equations of motion for interacting particles. Here, the forces of each particle can affect the trajectory of all other particles. The computational requirements to run such simulations are high, due to the large number of particles involved. Over the years, researchers have worked on minimizing the amount of time and power consumption used in a large molecular dynamics simulation. Some simulations, like protein folding, can take months, and even then only a small time frame has been simulated. Often, MD simulations are performed on a Graphics Processing Unit (GPU) because the GPU provides the advantage of great performance when working with matrices, and matrix calculations are some of the bottlenecks of the standard MD simulation. However, it is becoming clearer that other types of hardware also provide advantages for MD simulations. Application Specific Integrated Circuit (ASIC) is an obvious choice when needing precise, stable, and fast calculations, however, they are also very hard to design, and the cost of changing the design can be enormous. The Field-Programmable Gate Array (FPGA) is a good substitute for an expensive ASIC, as it is cheaper and reconfigurable. Because of the stability, it provides and since it is very energy efficient, it could offer an edge over GPUs. Unfortunately, developing solutions for FPGAs can be very tedious and require a lot of knowledge and experience.

1.1. Contribution

We examined if it is possible to design an MD simulation for FPGAs, and if so, how it would perform. By using Synchronous Message Exchange (SME) [1] to develop an MD simulation for an FPGA, we can simplify the implementation in comparison to other more hardware-oriented approaches. SME provides a way to develop systems for FPGAs with C#, a high-level programming language, while still providing the performance of a traditional FPGA implementation. Our solution shows how spatial architectures can have advantages both in structure and testability, as well as performance.

The system presented in this paper is a work in progress, and only the initial basic MD simulation structures have been implemented. Some functionalities, like cutoffs or periodic boundary conditions, are yet to be implemented.

The code for the project can be seen on github [2].

2. Molecular dynamics

MD is based on the idea of modeling structures that appear in nature, but cannot be viewed in detail by our own eyes. By modeling these structures on a computer, we can better understand them. This could for instance be water simulation and the understanding of why a water droplet creates rings when it lands in a pool. With MD it is possible to model each water molecule and observe how each molecule affects the position of another. MD simulations provide the possibility of viewing the interactions between molecules very closely. Depending on the amount of time and computational power we want to spend on the simulation, we can get a varying degree of precision with our simulation. With MD simulations, it is possible to create simulations that represent experiments that cannot be created in a laboratory,

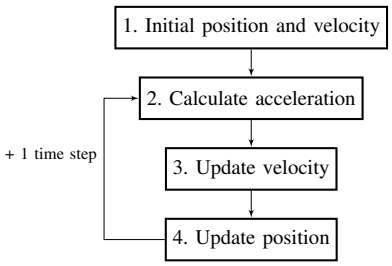


Figure 1. The specific steps of an MD simulation. Each step requires the result of the previous step. After step 4 it loops around to step 2. Each loop is a time step.

such as simulating extreme situations like very high pressure or temperature. The goal of MD is to have simulation models that can be directly compared to experimental setups made on specific materials, and therefore the effectiveness and correctness of the model are crucial.

The MD simulation is a step-by-step, numerical setup, where the equations of motion are followed. Our simulation consists of 4 steps, which are shown in Figure 1. The number of particles, time steps, and the duration of the simulation are all elements that contribute to the cost of the simulation. It is important to find a balance between these to get the best and most accurate results with the computational resources and time available. It is not unreasonable for simulations to take several CPU-days or even months, especially when simulating for instance DNA or proteins. During the MD simulation, the most computationally intensive task is usually the calculation of the potential energy as a function of the coordinates of each particle. This calculation is part of the acceleration calculation, which is step 2 in Figure 1.

2.1. Lennard-Jones

The Lennard-Jones (LJ) potential is a common potential to use in MD simulations and is used for calculating the potential energy of each particle. The LJ potential is a simple pair-potential, but it still describes essential features of the interactions between atoms and molecules, which is why it is so often used. Due to its simplicity, it is often used to simulate the movement of gasses and simple fluids. It is especially great for describing the properties of noble gasses and methane. The potential has the form

$$V_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

where r is the distance between the two particles, and both σ and ϵ are parameters specified by the known properties of the specific molecules. The LJ potential approximates the intermolecular potential energy between each pair of particles, and thereby it is possible to simulate how a group of particles move for a specified time step. The force F, between two interacting particles, is obtained by differentiating the LJ potential with respect to r, $F = dV/dr$. The distance between the particles will determine if the force is attractive or repulsive.

3. Related work

Today, most MD simulations are performed on supercomputers which provide extreme performance. Despite of this, there is still a need for even better performance and to be able to simulate even more time steps. There is a selection of widely used MD packages, which include many of the standard performance optimisations one can do in an MD simulation. Packages such as AMBER [3], CHARMM [4], GROMACS [5], LAMMPS [6], and NAMD [7] have been developed with optimisations in mind. But often MD simulation executions still end up having months-long runtimes, even on supercomputers [8].

There are two main categories within the research for optimisation of MD simulations. The first category is software, which consists of developing new and advanced algorithms for running simulations in a faster and more efficient way. We will not go deeper into this, since it is out of scope of this paper, but it is important to mention that this research area exists. The second category, and the one that we will focus on, is hardware. Here there has been a lot of development over the years to build optimal hardware for MD simulations. From CPU systems to GPU setups, to custom-built ASICs. The possibilities are many, and all have advantages and disadvantages.

Implementing MD simulations on an FPGA is not a new idea, several other projects have worked on this particular challenge [9]–[14]. However, many previous projects have been implementations of separate components and not full simulation systems. Some projects have been using a host CPU or GPU to share the work [15], other projects have been using systems like OpenCL [16], which does not provide proper performance on the FPGA. Some projects have just been theoretical and some are only concentrating on a specific part of the system or simulation [12].

The basis of the MD problem determines what kind of hardware would excel with it. For instance, Anton [17], [18] is a family of ASIC-based MD engines. These solutions have very high performance and are often used for simulations with long timescales and very small molecules. ASICs are the optimal choice for fast, reliable systems, but the development of an ASIC system is cumbersome and if the design is flawed, it is useless. Because the chip cannot be changed after creation it quickly becomes wasteful, and expensive to change the design once the chip is created.

In recent years, it has been shown that FPGA clusters for MD simulations could be created to have performances that approach the performance of an ASIC cluster [19], [20]. However, even though it is possible to develop FPGA systems that can compete, the actual development of the FPGA is still difficult and time-consuming, so the question remains: are the advantages of the FPGA solution worth the disadvantages of the implementation?

In 2019, Yang et al. presented their full-scale FPGA-based simulation engine for MD simulation [9]. This was one of the first solutions where the entire system was fully retained within the FPGA and their results were competitive to a GPU system. They created their system using Verilog

Hardware Description Language (HDL) which, along with Very high speed integrated circuit HDL (VHDL), are the typical programming languages used for FPGA development. However, Verilog and VHDL are languages that are very tedious to program in, requiring a lot of expert knowledge and time [1].

4. Synchronous Message Exchange

SME [1] is a runtime environment for developing and testing hardware designs for FPGAs in C#. With SME it is possible to create hardware structures that can be translated to VHDL, along with a testbench for this VHDL. This testbench verifies that when simulating the VHDL, it runs cycle accurate with the SME simulation. At the same time, it is possible to utilize the full C# library for creating the surrounding implementation for testing and simulating the hardware structures.

SME is derived from CSP [21] and is also structured similarly to CSP, with sequential processes that share nothing except their broadcast buses. In SME, only one process can write to a bus, but one or more processes can read from that bus at the same time. All data is propagated on the buses according to a hidden clock that drives the network.

By using SME, it was possible to develop and test the entire MD simulation while simulating a clocked network as well as the clocked data transfer from process to process. Though it can be difficult to program clocked processes and retain an overview of the data transfer time, it is much less cumbersome than developing the entire MD simulation in VHDL directly [1].

In SME there is a distinction between processes that will be translated to VHDL and those that will not. The `SimpleProcess` type is a process that will translate to VHDL and thus is limited in the available C# libraries. It is driven by the global hidden clock with the `OnTrigger` function that is triggered once in each clock cycle. Around the `SimpleProcess`, there can exist several processes which drive the C# simulation. These kinds of processes are called `SimulationProcesses` and are not translated into VHDL but are simply created for the C# simulation. Therefore we are free to use the advanced C# libraries within these processes.

5. Architecture

In this section, we will introduce the MD simulation system and describe some of the essential parts. We have designed a system in five modules that, when combined, provides a basic MD simulation. This entire system can be seen in Figure 2. Several steps have been taken to minimize the number of clock cycles, which in turn will reduce runtime. These steps will be explained throughout the next sections.

5.1. Simulation Modules

Each module is designed as an individual simulation setup so that it can be tested separately from the entirety of the system. Each individual module setup consists of one or more `SimpleProcesses` and at least one `SimulationProcess` which generates input data for the network. When the modules are connected to the entire MD system, one `SimulationProcess` generates data and verifies the result. This can be seen in Figure 2 as the `External Simulator`.

Besides generating input data, the `SimulationProcess` uses high-level C# libraries to calculate the results of the simulation as verification for the SME results. The individual design of the `Acceleration` module, created to test the module separately from the rest of the system, can be seen in Figure 3. Here the data generation and verification happen in the `Testing Simulator`. For instance, when calculating the acceleration between two particles, the `Testing Simulator` will create input data for the `Position RAM`, and then, while the acceleration is calculated in the `Acceleration` and `Force`, the `Testing Simulator` will calculate the acceleration using standard C#. This way, when the results come back from the `Acceleration`, it is simple to check whether or not these results are as expected. Often during development, we ran into errors where the calculations were correct, but the timing was wrong, which meant that the signals arrived unaligned and therefore was out of sync. By having a standard C# calculation based on the current input data, it was easy to recognize that the error did not lie in the calculations, but in the timing of the communication of data.

5.1.1. The Acceleration module. This module, as the name states, calculates the acceleration, but the calculation of the force, i.e. the potential energy, also lies within the `Acceleration` module, as can be seen in Figures 2 and 3. This module refers to step 2 in Figure 1. The acceleration is always calculated based on the positions of two different particles, and so the `Acceleration Manager` requests data from the `Position RAM` where all current positions of all particles are stored. The distance is calculated from this in `Acceleration`, and is sent to the `Magnitude` module. The result from the `Magnitude` module is sent to `Force` to calculate the potential energy, using the LJ potential calculation. The representation of this calculation can be seen in Figure 4. When this data comes back to the `Acceleration`, the last calculations are performed to get the actual acceleration and, if the module is tested on its own, the data goes back to the `Testing Simulator`, as can be seen in Figure 3. When the entire system is connected, the acceleration results are sent off to the `Cache` module, which can be seen in Figure 2.

The `Force` calculates the potential energy as a scalar, using the LJ potential, and returns this to the `Acceleration`. The structure of the `Force` consists of several `SimpleProcesses`, each consisting of a simple calculation. All `SimpleProcesses` are connected by buses. To get the most optimised result, the calculations of each `SimpleProcess` have been restricted to the simplest

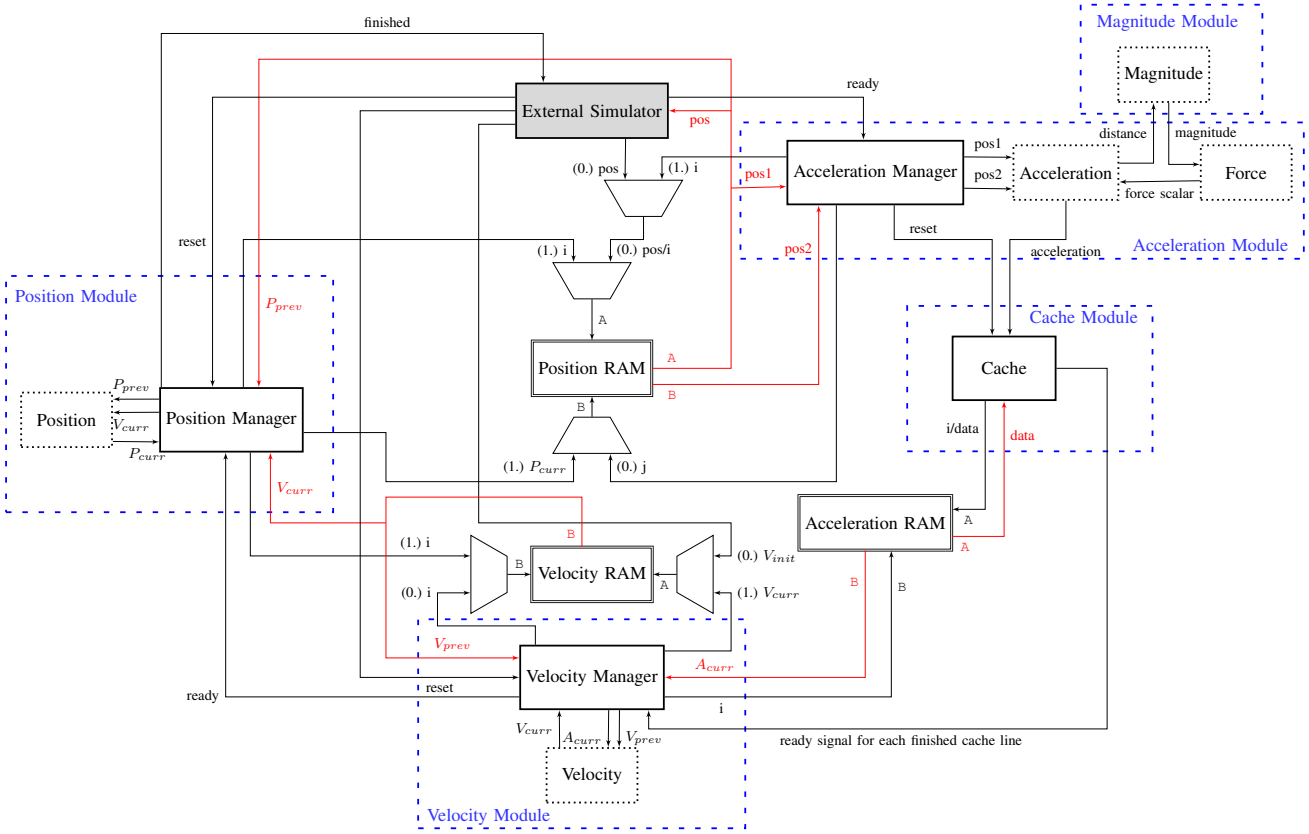


Figure 2. The entire MD simulation network. Each box represents a process. The grayed out box represents `SimulationProcess`, the unfilled boxes represent `SimpleProcesses`, and double edge boxes represents RAM. The trapeze-shaped processes are multiplexor processes that choose between one bus or another, the numbering in the figure shows the priority order. The dotted squares represents a collection of `SimpleProcesses`, for instance `Force`. The internal structure of `Force` can be seen in Figure 4. The large blue dotted squares represent the different modules. The red lines represent the data being communicated from the RAM.

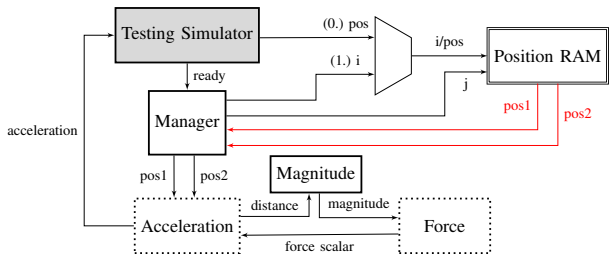


Figure 3. The complete design of the `Acceleration` module. The grayed out box represents `SimulationProcess`, the unfilled boxes represent `SimpleProcesses`, and double edge boxes represents RAM. The trapeze-shaped processes are multiplexor processes that choose between one bus or another, the numbering in the figure shows the priority order. The dotted squares represents a collection of `SimpleProcesses`, for instance `Force`. The internal structure of `Force` can be seen in Figure 4. The large blue dotted squares represent the different modules. The red lines represent the data being communicated from the RAM.

structures possible. The representation of this structure and the calculations can be seen in Figure 4.

One of the challenges we ran into when developing the calculation of the force in SME was that unlike with CPUs/GPUs, FPGAs are more limited in the available calculation methods i.e. the `SimpleProcess`. We can not just import a math library to do advanced functions. Certain calculations have a drastic better performance than others, such as division, and choosing wrong will reduce the performance of the system [22]. With this initial version of the MD simulation, we are targeting Xilinx boards and we are therefore restricted to the Intellectual Property (IP) blocks that they provide for floating point numbers [23]. To be able to do the LJ potential calculations, we had to do some restructuring to fit it into the possible IP blocks provided by Xilinx. The LJ potential consists of a power function, for instance $\left(\frac{\sigma^{12}}{r^{14}}\right)$ which unfortunately, Xilinx does not provide. However, since the power functions are all even numbers, and as such will always be positive, it was possible to use the rule: $b^x = e^{x*ln(b)}$. Therefore the entire LJ potential that can be rewritten as:

$$V_{LJ}(r) = 48\epsilon \left[\left(\frac{e^{12 \cdot ln(\sigma)}}{e^{14 \cdot ln(r)}} \right) - 24\epsilon \left(\frac{e^{6 \cdot ln(\sigma)}}{e^{8 \cdot ln(r)}} \right) \right]$$

This formula is represented in the processes seen in Figure 4. All these connected processes create the calculations of

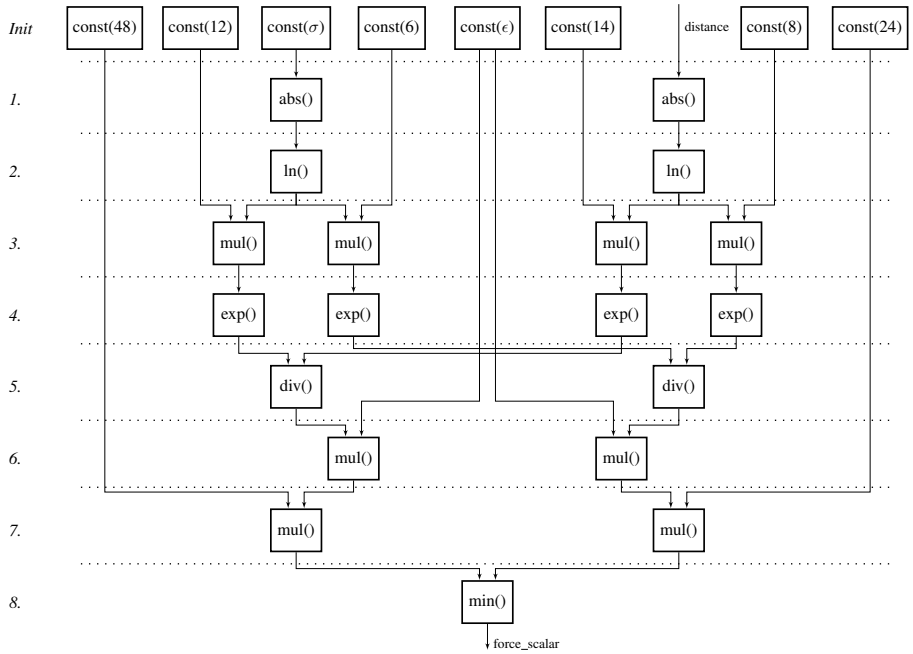


Figure 4. The `Force` calculation representing the entire LJ potential pipeline. Each square is a process that executes the function noted, given the input from the processes above. The horizontal lines represent each layer where the data flows between. The numbering in the left are for referencing purposes only. For each clock cycle, the data flows one layer down.

the LJ potential. It has been designed as a pipelined structure to ensure a fast flow through the system. This ensures that we can keep feeding data through. Therefore it will not be necessary to wait for the entire pipeline before we can begin to calculate the next LJ potential for a new pair of positions.

In Figure 4, each layer consists of a calculation and for each clock cycle, the data flows one layer down. This means that the data in layer 1, will be in the 5th layer, four clock cycles later. All the while, more data is being fed into the system and therefore we will continue to get a new result for every clock cycle, instead of having to wait for n clock cycles before beginning a new calculation.

All modules in the MD simulation have been designed to be pipelined. This was an important design consideration because each module consists of many individual calculations. If the system had to wait for each calculation to finish, it would increase the runtime considerably. The pipelined structure results in that in each clock cycle, all sections of the modules, are in use. The pipelined system also means that we cannot calculate something in one clock cycle and expect it to still be there a couple of clock cycles down the line. This means that if a result from a calculation needs to be used later on, we either have to save it into RAM or we need to send it through the pipeline until it is needed. With data that is always needed at a specific point in time, it makes sense to send it through the pipeline. An example can be seen in Figure 5, where x is calculated in `Process 1` and needed by `Process 3`. Because of the pipelined structure of the system, x will not continue to be accessible when `Process 2` has finished calculating y. Therefore, to

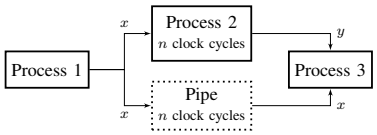


Figure 5. An example of a pipe structure that is inserted to retain data for a certain amount of clock cycles. In the figure, the `Pipe` is a collection of n pipe processes, each one taking one clock cycle.

make sure the data arrives at `Process 3` at the same time, the `Pipe` is inserted, which takes as many clock cycles as the `Process 2` takes.

This means that we create a process that accepts some data as input and the next clock cycle sends the data out on its output bus. We thereby create a structure of n piped processes that are linked together and are piping the data n clock cycles down the line. This structure can only be created if we know how many clock cycles will pass before the data is needed, and only if that amount of clock cycles are always the same.

5.1.2. The Cache module. The `Cache` was created in order to consolidate the acceleration data between particles. In other high-level implementations of MD simulations using the LJ potential, matrix calculations are used to calculate the forces between each pair of particles for all particles. As this matrix quickly grows, it's not feasible to contain the entire matrix within the fast memory available on an FPGA. Therefore it was necessary to create a structure that would have the same functionality but would work better

TABLE 1. EXAMPLE OF ACCELERATION CALCULATION BETWEEN PARTICLES. THE CALCULATIONS ABOVE THE DIAGONAL ARE THE SAME AS BELOW, JUST WITH THE THE SIGN REVERSED.

D	-AD	-BD	-CD	-
C	-AC	-BC	-	CD
B	-AB	-	BC	BD
A	-	AB	AC	AD
	A	**B**	**C**	**D**

on the FPGA. It is preferable to have data in registers or Block RAM (BRAM) on the FPGA. If we need to access DRAM, then data has to travel over the memory bus and this would require a much longer wait time before we access the data. The registers and BRAM are much closer on the board and are therefore better to use, if at all possible. With an MD simulation it is not feasible to have all positions, accelerations, and velocities of the particles in registers and therefore we are using BRAM for this information. Depending on the number of particles for the simulation and the FPGA, other FPGA design solutions might be better suited.

For all n particles, the force between any particle and all other $n-1$ particles has to be calculated, to be able to compute the acceleration. It is important to note that the force between particle A and particle B, and particle B and particle A is the same with the sign reversed. This means that when doing the calculations for $n \times n$ particles, it is only necessary to do half. An example can be seen in Table 1 where it is clear that the half above the diagonal is the same as the half below, but with the sign reversed.

The `Cache` works on continuously identifying the data needed in the next clock cycle. For this simulation, we keep all data in BRAM and so it requests the necessary data from BRAM in time for it to arrive when needed. The `Cache` receives one acceleration calculation from the `Acceleration` module and identifies which two particles the acceleration was calculated from. The `Cache` calculates which two data points are calculated next and makes sure to request this information from the BRAM in due time. At the same time, it is also sending the newly updated acceleration data back to the BRAM if it is not needed within a reasonable amount of clock cycles. This is done to avoid keeping too much data in registers.

When the `Cache` finishes an entire acceleration consolidation for one particle, it will send a signal to the `Velocity` module to start calculating on this data.

5.1.3. The Velocity and Position module. These two modules are very similar in their structure, but the results are, of course, different. Both modules are notified when the module prior is finished with the necessary calculations, and then it starts requesting data from the specific BRAM. In Figure 2 the red lines show the data being sent from BRAM to the processes that have requested data. When the data has been received, either the velocity or the position is updated with the new data received from the previous module. Then the result for the specific particle is sent back to the BRAM

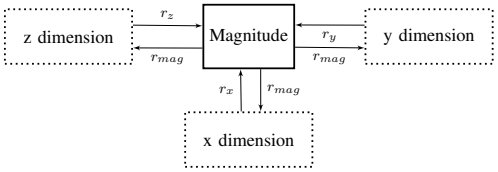


Figure 6. The magnitude is calculated between the different dimensions. The dimensions are running in parallel and communicating with the `Magnitude` module only when the magnitude is needed in the acceleration calculations.

to be stored for the next iteration of the simulation.

5.1.4. The Magnitude module. The first version of the MD simulation was created for only one-dimensional space. This, of course, is not an adequate solution, but the solution with n dimensions is not that different from the initial one-dimensional solution. The entire MD simulation circuit that has been described above, and which can be seen in Figure 2, can almost stand alone when calculating the next positions for each particle. The only step in the simulation structure, where the calculations from one dimension affect the other dimension, is the calculation of the magnitude. The magnitude is calculated as: $mag = \sqrt{\sum_{i=1}^{n} r_i^2}$. This means that we can not just layer n identical MD simulation structures and then simulate separate from each other with no communication between them. But a lot of the calculations can be done in parallel, and only when calculating the magnitude, the simulations have to sync up and communicate. This is done by defining n MD simulations, one for each necessary dimension, and then generate a `Magnitude` process that connects to all of these with a bus to each. Since each of the MD simulations are instances of the same modules, and with just as many data points as the rest, the calculations should happen within the same time frame. This means they will all send the required data to the `Magnitude` module at the same time, and when the magnitude calculation is complete, the data is sent back to each of the n MD simulations. They will then continue calculating the rest of the steps in the simulation. This structure can be seen in Figure 6. By using this we gain a parallel calculation of each dimension which means faster results.

6. Performance comparison

The system is created with double precision floating point numbers, simply because it is necessary for the MD simulation. But this results in higher resource consumption compared to a single precision solution. With the use of double precision, and due to floating point not being natively supported, we weren't able to have a working solution on a Xilinx PYNQ Z2, which was the FPGAs we had available, as we quickly ran out of resources on the FPGA. In order to produce some numbers, we instead target the bigger Xilinx XCKU5P-2FFVB676E found on the Xilinx KCU116, as this is a readily available FPGA that can be targeted using Xilinx Vivado without a license.

TABLE 2. TIMING AND UTILIZATIONS POST PLACE AND ROUTE AS REPORTED BY XILINX VIVADO TARGETING THE XILINX XCKU5P-2FFVB676E FPGA. THE PERCENTAGES ARE RELATED TO THE AVAILABLE RESOURCES ON THE FPGA.

Clock rate	Logic	Registers	Block RAM	DSPs
13.603 MHz	28.87 %	8.35 %	13.02 %	19.90 %

TABLE 3. SPECIFICATIONS OF THE MACHINE RUNNING THE BENCHMARKS, ALONG WITH THE VERSION NUMBERS OF THE PROGRAMS USED.

CPU	Intel(R) Core(TM) i7-8665U CPU 1.90GHz
RAM	48 GB DDR4 @ 2400 MHz
OS	Ubuntu 18.04.5 LTS
Xilinx Vivado	2020.2
Python	3.6.9
NumPy	1.19.1
FPGA Board	Xilinx KCU116
FPGA Chip	Xilinx XCKU5P-2FFVB676E

While it will not be a complete comparison, it can still help us build an intuition on performance estimations and scalability. The timing and utilization numbers gathered from Xilinx Vivado post place and route can be seen in Table 2. We were able to achieve a clock rate of 13.603 MHz at 28% utilization of the boards' resources. This is a feasible speed and is considerably lower than what we expect to be able to hit, given more time for optimizations.

If we use these numbers, we can compute an estimate of the execution time. We do this by counting the number of clock cycles used by SME during the simulation and multiplying them with the time needed for a single clock cycle. At 13.603 MHz one clock cycle takes 73.513 ns to complete. We have compared the estimates to actual running times of a Python implementation computing the same problem. The performance estimates, and the running times of the Python code, can be seen in Table 4. We see that for all cases, the SME implementation performs better than the Python implementation.

We have tried to make the comparison as fair as possible, but it should still be taken with a grain of salt. First of all, the Python implementation is single core, so not fully utilizing the processor. Secondly, the SME simulation does not measure moving data in and out of the FPGA. The Python implementation does measure moving data between the CPU and RAM, as this is required for doing computations on the CPU.

Furthermore, given the small sizes used in the benchmark, we won't see any significant change in running time, as the target board has a memory bandwidth of 2666 Mbps. However, memory in the SME implementation will only be read and written once, and these transactions are performed sequentially. If we assume perfect memory bandwidth, even with the big example, transferring 700 elements to and from memory will take 33.608 us. If we benchmark the performance of `numpy.copy()` of 1 GB of data on the machine described in Table 3, we see that it reaches around 8.42 GB/s. Comparing that to the theoretical bandwidth

TABLE 4. RUNNING TIMES OF AN MD SIMULATION WITH THE LJ POTENTIAL IN PYTHON, USING NUMPY, AND THE ESTIMATED RUNNING TIME IN SME. n DENOTES THE NUMBER OF PARTICLES SIMULATED. m DENOTES THE NUMBER OF INTEGRATIONS. t DENOTES THE RUNNING TIME IN SECONDS. Δ DENOTES THE SCALE FACTOR BETWEEN PYTHON AND SME. THE PYTHON CODE HAS BEEN RUN ON THE MACHINE SPECIFIED IN TABLE 3.

n	m	t_{Python}	t_{SME}	Δ
20	2	0.005	0.004	1.250
200	2	0.578	0.300	1.927
500	2	3.583	1.853	1.934
700	2	7.080	3.623	1.954
20	20	0.055	0.039	1.410
20	200	0.552	0.393	1.405
20	500	1.371	0.981	1.398
20	700	1.950	1.374	1.419

of 19.2 GB/s, we see that we can reach $\sim$44 % of the theoretical maximum. Using the same scale on the target board, we reach 1173 Mbps, resulting in a transfer of 700 elements to and from memory in 76.385 us, which is still not significant enough to have an impact on the results.

7. Future work

Since the project is a work in progress, we have several items on the list for future work.

- The most crucial task for future work is to get the network on the FPGA and test this with actual data. By doing this we can make better conclusions on the performance of the system.
- In order to finish the MD implementation, some structures are necessary to implement. This includes cut-off distance, which is common to use with the LJ potential. By implementing this, we should be able to make the simulation run for fewer clock cycles with the same amount of particles. Another simulation implementation addition would be periodic boundary condition. This is used to run a larger simulation than the defined space. When one particle accelerates outside the space, it loops back on the opposite side.
- Currently it is only possible to simulate one type of particle at a time. Simulating more than one particle would change the calculation of potential energy and therefore it is not straightforward to implement. But it would make the simulation more adaptable.
- The Particle-in-cell (PIC) method is often used to simulate plasma and can be used to economize on compute time. The concept lies in dividing the simulation space into a mesh where each cell in the mesh can be calculated in parallel. We believe that using the inherent parallelism of the FPGA would pair greatly with the PIC method.

8. Conclusion

In this paper, we have presented a simple molecular dynamics simulation on an FPGA. The project is a proof of concept and has been implemented in Synchronous Message Exchange, which is a runtime environment in C# for implementing and testing hardware designs. The simulation is simple, and some elements, that are expected in a molecular dynamics simulation, have not yet been implemented. By using different optimisations we have been able to create a structure that can calculate multiple dimensions simulations in parallel. The project has also been created with modularity in mind so that each section of the molecular dynamics structure has been created as individual modules that can be run and tested separately. This increases transparency and changes an otherwise very complex solution to a more intelligible model. We have shown that the performance estimates for the SME implementation, even without optimising low-level FPGA constructs, show that there could be something to gain, by exploring these spatial architectures.

Acknowledgment

This work was supported in parts by PRACE 6IP.

References

[1] C.-J. Johnsen, A. Thegler, K. Skovhede, and B. Vinter, "Sme: A high productivity fpga tool for software programmers," 2021.

[2] A. Thegler, "SME-MD source code," https://github.com/sme-projects/SME-MD, [Online; accessed February 2021].

[3] D. A. Case, T. E. Cheatham III, T. Darden, H. Gohlke, R. Luo, K. M. Merz Jr., A. Onufriev, C. Simmerling, B. Wang, and R. J. Woods, "The amber biomolecular simulation programs," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1668–1688, 2005. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.20290

[4] B. R. Brooks, C. L. Brooks, A. D. Mackerell, L. Nilsson, R. J. Petrella, B. Roux, Y. Won, G. Archontis, C. Bartels, S. Boresch, A. Caflisch, L. Caves, Q. Cui, A. R. Dinner, M. Feig, S. Fischer, J. Gao, M. Hodoscek, W. Im, K. Kuczera, T. Lazaridis, J. Ma, V. Ovchinnikov, E. Paci, R. W. Pastor, C. B. Post, J. Z. Pu, M. Schaefer, B. Tidor, R. M. Venable, H. L. Woodcock, X. Wu, W. Yang, D. M. York, and M. Karplus, "Charmm: The biomolecular simulation program," *Journal of computational chemistry*, vol. 30, no. 10, pp. 1545–1614, 2009.

[5] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation," *Journal of chemical theory and computation*, vol. 4, no. 3, pp. 435–447, 2008.

[6] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19, 1995.

[7] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with namd," *Journal of computational chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.20289

[8] D. Shaw, R. Dror, J. Salmon, J. Grossman, K. Mackenzie, J. Bank, C. Young, M. Deneroff, B. Batson, K. Bowers, E. Chow, M. Eastwood, D. Ierardi, J. Klepeis, J. Kuskin, R. Larson, K. Lindorff-Larsen, P. Maragakis, M. Moraes, S. Piana, Y. Shan, and B. Towles, "Millisecond-scale molecular dynamics simulations on anton," in *Proceedings of the Conference on high performance computing networking, storage and analysis*, ser. SC '09. ACM, 2009, pp. 1–11.

[9] C. Yang, T. Geng, T. Wang, R. Patel, Q. Xiong, A. Sanaullah, C. Wu, J. Sheng, C. Lin, V. Sachdeva, W. Sherman, and M. Herbordt, "Fully integrated fpga molecular dynamics simulations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. ACM, 2019, pp. 1–31.

[10] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow, "Reconfigurable molecular dynamics simulator," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Los Alamitos CA: IEEE, 2004, pp. 197–206.

[11] S. Alam, P. Agarwal, M. Smith, J. Vetter, and D. Caliga, "Using fpga devices to accelerate biomolecular simulations," *Computer (Long Beach, Calif.)*, vol. 40, no. 3, pp. 66–73, 2007.

[12] M. Chiu and M. C. Herbordt, "Molecular dynamics simulations on high-performance reconfigurable computing systems," *ACM transactions on reconfigurable technology and systems*, vol. 3, no. 4, pp. 1–37, 2010.

[13] T. Hamada and N. Nakasato, "Massively parallel processors generator for reconfigurable system," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*. IEEE, 2005, pp. 329–330.

[14] V. Kindratenko and D. Pointer, "A case study in porting a production scientific supercomputing application to a reconfigurable computer," in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2006, pp. 13–22.

[15] C. Rodrigues, D. Hardy, J. Stone, K. Schulten, and W.-M. Hwu, "Gpu acceleration of cutoff pair potentials for molecular modeling applications," in *Proceedings of the 5th conference on computing frontiers*, ser. CF '08. ACM, 2008, pp. 273–282.

[16] H. M. Waidyasooriya, M. Hariyama, and K. Kasahara, "Architecture of an fpga accelerator for molecular dynamics simulation using opencl," in *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, 2016, pp. 1–5.

[17] D. Shaw, M. Deneroff, R. Dror, J. Kuskin, R. Larson, J. Salmon, C. Young, B. Batson, K. Bowers, J. Chao, M. Eastwood, J. Gagliardo, J. Grossman, C. Ho, D. Ierardi, I. Kolossváry, J. Klepeis, T. Layman, C. McLeavey, M. Moraes, R. Mueller, E. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. Wang, "Anton, a special-purpose machine for molecular dynamics simulation," in *Proceedings of the 34th annual international symposium on computer architecture*, ser. ISCA '07. ACM, 2007, pp. 1–12.

[18] B. Towles, J. Grossman, B. Greskamp, and D. Shaw, "Unifying on-chip and inter-node switching within the anton 2 network," in *Proceeding of the 41st annual international symposium on computer architecuture*, ser. ISCA '14. IEEE Press, 2014, pp. 1–12.

[19] J. Sheng, B. Humphries, H. Zhang, and M. C. Herbordt, "Design of 3d ffts with fpga clusters," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[20] J. Sheng, C. Yang, and M. Herbordt, "Towards low-latency communication on fpga clusters with 3 d fft case study," 2015.

[21] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[22] "Performance and resource utilization for floating-point v7.1," https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html, [Online; accessed February 2021].

[23] "Floating-point operator v7.1 - logicore ip product guide," https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf, [Online; accessed February 2021].

A.1.4 *Occam to Go translator*

Occam to Go translator

1st Matilde Broløs
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
jtw868@alumni.ku.dk

2nd Carl-Johannes Johnsen
Niels Bohr Institute
University of Copenhagen
Copenhagen, Denmark
cjjohnsen@nbi.ku.dk

3rd Kenneth Skovhede
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
skovhede@di.ku.dk

***Abstract*—Occam is a programming language built on CSP, which for many years has been used for writing safety-critical systems used in space technology and at CERN among others. However, the language has not been developed or maintained for the last 25 years, which makes it difficult to maintain the programs which currently has a code base in Occam. As changing the entire code base for such systems will prove both expensive and time consuming, it is desirable to find an easy and secure way to translate Occam programs into another programming language.**

This paper lays the foundation of a transpiler from Occam to the newer programming language Go using Haskell. Go is a modern programming language which also implements many of the CSP principles found in Occam, making it a suitable target.

The transpiler is implemented for a subset of Occam including only basic functionality, and is successful in translating simple programs from Occam to Go, showing that it is indeed possible to automatically translate Occam programs into Go.

***Index Terms*—Occam, CSP, Go, transpiler**

I. Introduction

When writing software for safety-critical systems it is crucial that the program can be verified using formal proofs of correctness. In 1978 Tony Hoare expressed a formal language *Communicating Sequential Processes* (*CSP*) that provides a mathematical foundation for ensuring provable correctness of programs [1], [2], as a new basis for concurrent programming languages.

The programming language Occam is built upon the CSP model, meaning that it is an actual implementation of CSP. This makes it ideal for working with safety-critical systems.

The language is used in big projects concerning safety-critical systems, such as satellites, the SpaceWire project, and certain software used at CERN. These projects have been formally verified to ensure the correctness of the code.

The language was developed through the 80s and early 90s, but development ended with Occam 2.1 which was published in 1994 [3]. The language has not been developed upon since then, and as a result of this, Occam programs cannot be run on more recent machines. This makes maintaining Occam programs difficult, both because certain technology is required to run the Occam programs, and because the language is no longer updated to accommodate the wishes of new developers.

Go is a much newer programming language that saw the light of day in 2007. It is syntactically similar to `C`, but draws inspiration from a number of other programming languages too. Go implements CSP style concurrency, which makes certain parts of the language very similar to Occam.

As Occam is used in a number of important systems, we want to find an easy way to run old Occam programs on modern platforms, to prolong the life of Occam programs.

A. Contribution

This paper describes building a transpiler that can translate an Occam program into a functionally similar Go program. Go is chosen as target language because it implements CSP inspired concurrency constructs, while also being a modern programming language with a high level of readability. The transpiler is written in Haskell, and is able to translate simple Occam programs into Go. Haskell was chosen because it is a functional language, which makes it advantageous for building compilers. The code is available on Github [4].

The two languages Occam and Go is now briefly presented, and their syntax is described, followed by a discussion of the differences in the structure of the two languages.

B. Related work

It has previously been attempted to come up with solutions for running Occam programs on more modern machines.

1) KRoC: KRoC [5] is an Occam compiler built for Occam 2.1, which enables users to compile and run Occam programs on newer platforms. However, KRoC had its final release in 2006, and therefore only runs on older platforms. For this project KRoC has been used for testing the functionality and run time of Occam example programs, and for this a virtual machine running an older 32-bit Ubuntu machine was used.

2) Occam-π: Occam-π [6] was an extension of the original Occam 2.1 language also developed by the same group of people who were behind KRoC. This extension of the language is quite interesting, but due to the limited scope of this project, Occam-π is not a part of the project.

3) Hard Stuff Language: Hard Stuff Language [7] is a project concerned with researching the possibility of translating Occam into more modern programming languages that still accommodate the CSP principles. The project takes a starting point in Go as the target language, but addresses the possibility of translating into any CSP-friendly language. Where our project describes a compiler built from scratch for moving Occam code bases into a new language, the Hard Stuff Language project is more concerned with the possibilities of extending Occam and writing new systems in that language.

```
1  PROC count(CHAN OF BYTE in, out, err)
2    BYTE c:
3    SEQ
4      c := 48
5      SEQ i = 0 FOR 10
6        SEQ
7          out ! c
8          c := c + 1
9      out ! 10
10 :
```

Listing 1. Simple Occam program. The code defines a procedure named `count`, taking a channel of bytes as argument. The body of the procedure loops from 0 to 9, and in each iteration it sends a value on the channel `out` and increments the counter. The procedure is ended by the '`:`' symbol.

```
1 channel ? variable
2 channel ! expression
```

Listing 2. Syntax for communication in Occam. The input command on line 1 reads from `channel` into `variable`. The output command on line 2 sends a message `expression` on `channel`.

II. The Occam language

Occam is an imperative procedural programming language based on CSP. It was developed in the 1980s by INMOS, advised by Tony Hoare, as the primary language of their Transputer microprocessors [3]. It was also used in projects concerning space technology, and in CERN, among others.

A. Syntax

Occam is an indentation sensitive language, like Python or Haskell. One of the fundamental structures of Occam programs is *procedures*. A simple procedure counting from 0 to 9 can be seen in Listing 1. The body of a procedure consists of processes, which are very close to the processes that Hoare presented in CSP. Some of the most important processes in Occam are the ones that allows communication via channels, running processes concurrently, and alternations.

1) Communication: Communication in Occam is simple: input is denoted by '`?`', and output is denoted by '`!`'. The syntax can be seen in Listing 2. Communication in Occam is unbuffered, meaning only one message can be sent and received on a channel at a time. Channels are also rendezvous, which means that communication only occurs when both the sender and the receiver are ready. If either are left waiting forever, it results in a deadlock.

2) Concurrent processes: Occam also introduces parallel processes, denoted by the keyword `PAR`. The syntax can be seen in Listing 3. As defined in CSP, the parallel command in Occam runs zero or more processes in parallel, terminating only when all of these processes are terminated. A variable which is written by one process cannot be used in any of the other processes. Furthermore a channel cannot be used for input in more than one of the processes, nor can it be used for output in more than one of the processes.

3) Alternation: Equivalent to the CSP *alternative* command is the Occam alternation, which is denoted by the keyword `ALT`. The syntax can be seen in Listing 4. An alternation in Occam executes exactly one of the given alternatives, based on the guards for each alternative. If multiple alternatives are ready at the same time, the choice between them is arbitrary.

```
1 PAR
2     P1()
3     P2()
4     P3()
```

Listing 3. The syntax for running parallel processes in Occam. In this example, three processes `P1`, `P2` and `P3` are run concurrently.

```
1 ALT
2     input
3         process
4     boolean & input
5         process
6     boolean & SKIP
7         process
```

Listing 4. Syntax of alternation in Occam. The different types of alternatives are shown in line 2, where an input is ready, line 4, where an input is guarded by a boolean expression and line 6, where the boolean expression is followed by the void process `SKIP`.

III. The Go language

Go is a programming language designed at Google in 2007, and released in 2012, which makes it a very modern language. It is highly influenced by C, but also introduces some handy built-in concurrency principles that resembles the concurrency of CSP [8].

An advantage of Go is its popularity with users. It was received extremely well in programming societies, and won several prices in the coding community, and was praised for its readability and understandability [8]. Go is already used in some notable applications and companies, such as *Docker*, a tool for building containers on Linux, *CockroachDB*, a distributed database, *Dropbox*, a cloud storage solution, *Netflix*, a streaming service, *SoundCloud*, a music sharing site, and *Uber*, an app based alternative to taxis [8].

A. Syntax

A simple Go program, performing the same job as the example Occam program in Listing 1, can be seen in Listing 5. Like Occam, Go features constructs for communication, running concurrent processes, and alternations.

```
1 func count(in, out, err chan byte) {
2   var c byte
3   c = 48
4   for i := 0; i < 10; i++ {
5     out <- c
6     c = c + 1
7   }
8   out <- 10
9 }
```

Listing 5. Simple Go program. The code defines a function named `count` taking a channel of bytes as argument. The body iterates from 0 to 9, sending the value on the `out` channel. Bodies of functions and loops etc in Go are started and ended with curly brackets.

```
1 variable := <-channel
2 channel <- message
```

Listing 6. Syntax for communication in Go. The input command on line 1 reads from `channel` into `variable`. The output command on line 2 sends a `message` on `channel`.

```
1 go P1()
2 go P2()
3 go P3()
```

Listing 7. The syntax for running concurrent processes in Go. Unlike Occam, only a single goroutine is started using this command, and execution continues after starting the process. In this example, the functions `P1`, `P2` and `P3` are run concurrently. Additional steps are needed to fully mimic Occam, which is described in Section IV-E7.

1) Communication: Like in Occam, when messages are sent and received in Go they are unbuffered, so that further execution is blocked until the IO function has been performed. One can write to a channel using '<−', and read from a channel using ':= <−'. The syntax can be seen in Listing 6.

2) Concurrent processes: To run several processes in parallel in Go one uses *goroutines*. Goroutines spawn a process which executes a given function, and doesn't terminate until the process is done. The syntax for goroutines can be seen in Listing 7. Several goroutines can be started at the same time, using multiple `go` statements in sequence. Gouroutines are different from the *parallel* process in Occam because the main process does not wait for the goroutines to terminate before continuing execution.

3) Alternation: The Go construct `select` is very similar to Occam's *alternations*, in that a `select`-statement waits for communication on multiple channels, and then performs an action depending on which channel first executes a channel operation. If multiple IO actions are ready at the same time, it is chosen randomly which action to perform. The syntax can be seen in Listing 8. While Occam alternations only accept input operations, Go accept both input and output. Go `select`-statements also differ from Occam alternations because they do not accept boolean expressions as part of guards.

IV. The transpiler

The process of translation can be split into two different parts: first, the Occam program must be parsed by the transpiler. In this step the program is read and the structure is

```
1 select {
2     case variable := <- channel
3         statements
4     case channel <- variable
5         statements
6     case default
7         statements
8 }
```

Listing 8. Syntax of alternation in Go. The different types of alternatives are shown in line 2, where an input is ready, line 4, where an output is ready and line 6, which is chosen, if none of the other cases were matched.

Fig. 1. Overview of the structure of the compiler.

transformed into an abstract syntax tree. The next part then consists of interpreting this syntax tree, converting the rules into Go syntax. An overview of these steps can be seen in Figure 1.

A. Parsable Occam subset

This project implements a transpiler working an a smaller subset of Occam. The subset was chosen to best accomodate two priciples: 1) it should be possible to build meaningful, executable Occam programs from the subset, and 2) programs must be able to use concurrency principles, as concurrency and communication on channels is the basis of CSP, and thus Occam.

In full Occam, programs are a collection of definitions, which can be a number of things such as *procedures*, *functions*, *type constructors* and *protocols*. Procedures are essential for writing Occam programs and therefore these were chosen as the foundation of the subset. The body of a procedure consist of a process.

Processes introduce many useful programming structures, such as conditionals, loops and selections, which are all control structures present in many programming languages. Those processes special for Occam are parallels and alternations which were presented in Section II. Both these processes are very useful when working with parallelism and concurrency. Furthermore, we have the process `SKIP` which acts as if no action is performed, and the process `STOP`, which ends a program and is somehow equivalent to exiting in other languages. The last two special processes are `input` and `output`, which helps us communicate between channels.

B. Parser

The parser is built of many smaller parsers for parsing each of the building blocks in the grammar, i.e. for each rule in the grammar a parser has been defined, and then those parsers are combined into one main parser.

C. Abstract Syntax Tree

When the parser reads the input program, it transforms it into an Abstract Syntax Tree (AST). This AST is designed to work as a bridge between the two languages Occam and Go, and therefore the syntax is a mix of the two. A program is defined as a list of functions, where each function has a name, some arguments, a specification of the type of the result, and a body statement.

Statements in the abstract syntax are equivalent to processes in Occam. Statements are mostly built up of expressions, which can be constants, variables or channels, operations, function calls, negations, arrays or array slices, or conversions. The abstract syntax for statements can be seen in Listing 9.

```
data Stmt =
     SDef [Exp] [Exp]
   | SDecl [Exp] Spec Stmt
   | SSeq [Stmt]
   | SIf [Case]
   | SSwitch Exp [Case]
   | SGo [Stmt]
   | SSelect [Stmt]
   | SWhile Exp Stmt
   | SFor Exp Exp Exp Stmt
   | SCase Case
   | SCall Exp
   | SSend Exp Exp
   | SReceive Exp Exp
   | SContinue
   | SExit
```

Listing 9. Haskell code showing part of the abstract syntax defining a statement.

```
SGo [
     SCall Call P1 [],
     SCall Call P2 [],
     SCall Call P3 []
]
```

Listing 10. AST entry generated for the Occam code in Listing 3.

As an example of a process from Occam expressed in the intermediate language, consider the AST for the parallel process defined in Listing 3 which initialises three processes `P1`, `P2`, `P3`. The resulting AST is shown in Listing 10. This shows how a parallel is denoted by the keyword `SGo` and a list of statements to be run in parallel.

D. Code generator

The code generator takes as input a `Program` defined by an AST, and from this generates a program written in Go. It is assumed that the input `Program` is correct, meaning that it is semantically valid and follow all the rules of Occam programs. This transpiler is intended for translating already verified Occam programs, not as a tool for developing new Occam programs. Therefore it was deemed unnecessary to verify the input program, at least for the initial version.

The intention behind the code generator is to break the input program into smaller components, translate each of these components, and then put them together to form the program code. In practice this is done by implementing a generator for each component in the abstract syntax, so that code generation of for example a function is done by generating the function *name*, *arguments* and *specifications*, and then generating the body *statement*, combining these into one string of Go code.

The purpose of dividing both the parser and the generator into several smaller parts is to accommodate extension of the implementation. When expanding with a new part of Occam syntax, it is simply added as a new entry in each of the parser, AST and generator, which makes expansion easy and intuitive.

The case of translating the AST in Listing 3 into Go code is presented in the next sub section.

E. Challenges

As Occam and Go are both based on CSP principles, parts of the two languages are very similar. This is evident when looking at the way both Occam and Go implement communication via channels, constructs for running processes concurrently and alternations.

However, though they are very similar, some challenges still occur in the translation. These challenges entails that Occam programs cannot be translated directly, and the translation becomes non-trivial.

1) `SKIP` and `STOP` constructs: The `SKIP` process of Occam simply continues execution without any effect, and can therefore be translated to an empty statement in Go. The functionality of `STOP` in Occam is to never terminate, which in practice means to stop the entire program as no further execution can happen beyond that point. This is implemented as `os.Exit` in the generator.

2) Channel declarations: When declaring channels in Go these have to be explicitly instantiated, whereas in Occam they are automatically instantiated when declared. This problem is solved by simply instantiating channels when declaring them in Go, as follows. So an Occam program defining

```
CHAN OF type var:
```

is translated into the Go code

```
var := make(chan type)
```

3) Variable assignments: Most assignments can be directly translated from Occam to Go. In Go it is important that a variable has been declared before assignment, but as we assume that the input program is a valid Occam program, then we know that all variables have been declared correctly, and that the types of variables and values correspond.

However, a problem does occur when translating an assignment of an array to a variable such as `v = [1,2,3,4]`, as one cannot assign an entire array to a variable in Go, except when instantiating an array. To solve this we could either assign one element in the array at a time, which could potentially take up an inexpedient amount of lines in the generated Go program, or we could instantiate the array directly in the assignment. Thus the Occam assignment

```
nums := [2,5,4,1,3]
```

is equivalent to the Go assignment

```
nums = [5]int{2,5,4,1,3}
```

To instantiate an array in Go one must know the type of the array. Therefore the types of all variables are kept track of in the generator.

4) Replicated processes: Replicated processes in Occam repeat a given process a specified number of times, which is exactly what a for-loop in Go does. A replicated process in Occam, such as

```
SEQ i = b FOR c
    stmt
```

is translated into the following for-loop in Go

```
for i := b; i < b + c; i ++ {
    stmt
}
```

5) `WHILE` loops: While-loops does not exist in Go, but a for-loop consisting only of a condition and a statement behaves the same way as a while-loop. This means that a while-loop written as `SWhile Exp Stmt` in the intermediate language is equal to a for-loop which executes the statement `Stmt` as long as the expression `Exp` is true. E.g. the Occam code

```
WHILE n < ITERS
    stmt
```

is translated into the following for-loop in Go

```
for n < ITERS {
    stmt
}
```

6) `IF` statements: Conditional processes in Occam can be translated almost directly into Go code, with the exception that Occam conditionals behaves as a `STOP` process if none of the cases are true. Therefore when generating the Go code an else-statement is added in the end which always calls `os.Exit`, as this ensures that the program stops if none of the above cases are true. E.g. the following Occam code

```
IF
    i < 5
        stmt
```

is translated into the following if-statement in Go:

```
if i < 5 {
    stmt
} else {
    os.exit()
}
```

7) `PAR` constructs: The Go statement that comes closest to the Occam parallel process is the goroutine, but there are some significant differences. First of all, goroutines only start one process each, so if multiple processes are to run in parallel then multiple goroutines must be used. This is easy enough to implement, but the real difference is that the main process keeps executing after starting the goroutines. In Occam the parallel process waits for each of the started processes to terminate before continuing execution. To ensure that the Occam functionality is preserved in the Go code we can use *WaitGroups*, which is another feature in Go. `WaitGroups` helps keeping track of parallel processes and ensuring that execution is paused until all processes have terminated. This is done by initialising a `WaitGroup`, and for every parallel process started, the process is added to the `WaitGroup`. Then after all the desired processes have been started, the `WaitGroup` is told to wait, which means that it will not allow further execution until all processes have unregistered. Each of the parallel processes then have to communicate to the `WaitGroup` when they are terminating, so that the `WaitGroup` knows when to continue execution. If we take the example AST in Listing 10, the transpiler will produce the following Go code

```
var wg_0 sync.WaitGroup
wg_0.Add(1)
go func() { defer wg_0.Done(); P1 }()
wg_0.Add(1)
go func() { defer wg_0.Done(); P2 }()
wg_0.Add(1)
go func() { defer wg_0.Done(); P3 }()
wg_0.Wait()
```

If multiple `WaitGroups` are used in the same scope, it is important that they use different names, as to not interfere with each other. Therefore `WaitGroup` names are defined as a string `wg_x` where x is the number of `WaitGroups` currently instantiated. This way each `WaitGroup` will have a fresh name.

8) `ALT` constructs: Alternations in Occam are, as mentioned previously, very similar to the Go statement *select*, but does however differ on one important point: while Occam alternations accept either an input function as guard, or both a boolean and an input function, a Go select only takes input and output statements as guards. To implement the Occam functionality we need to find a way to choose to listen to a channel only if a boolean expression is true. It is important to notice that listening to a channel that is `nil` will never be successful, thus a case where the channel in the guard is `nil` is never entered. Therefore, if we have a case where the condition `cond` needs to be true and we wait on channel `chan` with type `type` using variable `var`, the case in Occam would be

```
ALT
    cond & chan
        stmt
```

and the case in Go would be

```
select {
    case var := <-func() type {
            if cond {return chan}
            else {return nil}}()
        stmt
}
```

This code says that if `cond` is true then wait on `chan`, else wait on `nil`. The anonymous function needs to specify the output type, i.e. the type of the channel, and for this we again use tracking of types of variables.

It should be noted that the current implementation does not accept cases where the guard consists of a boolean expression and the process `SKIP`, as all guards of a select in Go must contain an input or output statement, and no work-around has been found yet.

In Go, if multiple cases are ready at the same time, the `select`-statement chooses between the ready statements at random. In Occam it is also the case that only one of the ready processes will be performed, but it is not defined which one. By studying the implementation details for Occam it can be seen that alternations actually work somewhat like prioritised alternations, where the top process has a higher priority than the next, and so forth.

Since Go select statements are chosen at random, the program generated by the current implementation of the transpiler will not behave precisely as the corresponding Occam program, even though it follows the language specifications for Occam. This is important to notice for future work or potential use, as this dissimilarity could result in a Go program performing a different job than the original program.

9) Names: In Go identifiers may contain letters, digits and underscores [9], which is slightly different from names in Occam, which can contain letters, digits and dots. For this

project it has been decided that when translating names all dots will be replaced with underscores, to ensure that all identifiers are valid. E.g. the Occam line

```
INT tmp.var:
```

is translated into the Go line

```
int tmp_var
```

10) Imports in Go: For some of the translations, there is a need to import packages for the Go program to work. To ensure that the correct packages are imported, the generator keeps track of all the needed imports using an import list. Whenever an import is needed it is added to the list, and when generation is done all these imports are appended at the beginning of the Go program.

11) Top-level entry point process: In KRoC the top-level process is the last procedure defined in a program [5], and the terminal itself act like a process, which can communicate with the top-level process over channels. The top level process of Occam must take as input one, two or three channels of bytes, which are an input channel, an output channel, and an error channel defined in that order. These channels are used to communicate with the terminal.

In Go the top process is always called `main`, and can be located anywhere in the file. It is not possible to write to the terminal via channels. To implement the Occam behaviour for communicating with the terminal, a default main function has been defined, which simulates the terminal process in Occam. As an example, the `main` process for the program in Listing 1 would be

```
func() main {
    in  := make(chan byte, 10)
    out := make(chan byte, 10)
    err := make(chan byte, 10)

    var wg_main sync.WaitGroup
    wg_main.Add(4)

    go top_process(in, out, err)
    go read_from_terminal(in)
    go write_to_terminal(out)
    go write_to_terminal(err)

    wg_main.Wait()
}
```

The main function instantiates three channels of bytes called `in, out` and `err`, and starts the original top-level process using a goroutine. It then ensures that whenever input is written in the terminal, it is read and passed on to the top-level process via the channel `in`. Whenever output is written from the top-level process via the channel `out`, the message is printed to the terminal. If any messages are received on the `err` channel, it is also written to the terminal. This implementation solution assumes that no Occam process is ever called main, and that the top-level process of the Occam program takes as input all three channels `in, out, err`. It is not necessary that all three channels are used, but the process must accept them as arguments.

V. Results

Now that the intention and structure of the transpiler has been presented, we show how well the solution performs the task.

A. Example translation

Consider the translation of the small Occam program highlighted in Listing 1. First step in the translation is to transform the program into an AST. The AST generates the program as a list of functions denoted by the keyword `FFun`. This program consists of a single occam procedure with the name `"count"` which takes three input arguments of type `SChan BYTE`. The list of specifications for the output is empty, because the function does not return anything. Now this AST can be fed to the generator, which then generates the program shown in Listing 11. From the generated code we see how the Occam top-level procedure is translated into lines $8-16$. The rest of the generated program is wrapping to accommodate the Go syntax, e.g. the program starts with a number of necessary imports, and lines $18-57$ shows the generated main function in Go.

B. Correctness

To test the functionality of the solution two kinds of correctness tests were performed.

A unit test suite was written to systematically test both the parser and the generator separately. The intention behind the unit tests is to test each branch of the parser and generator, ensuring that each component behaves as intended. This includes negative testing, e.g. testing that incorrectly indented code blocks results in parse errors, and edge case testing, e.g. nested replicated sequences or giving an empty program as input.

To test the full functionality of the parser and generator combined, a number of small example programs were written in Occam, and translated using the transpiler. Some test programs were designed to test the basic functionality of the solution, and other were designed to test more difficult examples, such as having multiple processes running concurrently, and alternating between input on channels.

C. Benchmarks

In order to test performance of the implementation, we have run a few benchmarks. We have focused on four different aspects: Execution time of the translation, execution time of the programs, memory consumption of the programs and finally the binary footprint. The specifications of the machine and programs used in our benchmark can be seen in Table I.

We are benchmarking three programs: *count*, which is the program showcased in Listing 1, *extended*, which contains some of the more advanced Occam constructs that the transpiler supports, and finally *commstime*, which is a classical CSP benchmark for measuring communication time. *commstime* is run with $n = 400$. The numbers presented are a mean of 100 runs, with a 10 run warm-up. The Occam programs are run through KRoC running in a virtual machine. Haskell and Go are run on the host machine.

```
1  package main
2
3  import "fmt"
4  import "sync"
5  import "os"
6  import "bufio"
7
8  func count(in, out, err chan byte) {
9    defer close(out)
10   defer close(err)
11
12   for i := 0; i < 10; i++ {
13     out <- byte(i + 48)
14   }
15   out <- 10
16 }
17
18 func main() {
19   in, out, err := make(chan byte, 10),
20                   make(chan byte, 10),
21                   make(chan byte, 10)
22
23   var wg_main sync.WaitGroup
24
25   wg_main.Add(1)
26   go func() {
27     count(in, out, err)
28
29     wg_main.Done()
30   }()
31
32   go func() {
33     input := bufio.NewReader(os.Stdin)
34     for {
35       char, _, err := input.ReadRune()
36       if err == nil {in <- byte(char)}
37     }
38   }()
39
40   wg_main.Add(1)
41   go func() {
42     for i := range out {
43       fmt.Print(string(i))
44     }
45     wg_main.Done()
46   }()
47
48   wg_main.Add(1)
49   go func() {
50     for i := range err {
51       fmt.Print(string(i))
52     }
53     wg_main.Done()
54   }()
55
56   wg_main.Wait()
57 }
```

Listing 11. The program from Listing 1 translated into Go.

TABLE I
SPECIFICATIONS OF THE MACHINE AND PROGRAMS USED FOR BENCHMARKING.

Element	Version
CPU	Intel® Core™ i7-8565U 4-core @ 1.80 GHz
RAM	16 GB LPDDR3 @ 2133 MHz
Disk	512 GB NVMe SSD
OS	Ubuntu 20.04.1 LTS
Go (Go)	1.13.8
KRoC (Occam)	4676039 (git commit hash)
Cabal (Haskell)	3.2.0.0
VirtualBox	6.1.10
VM OS	Ubuntu 12.04 32-bit

TABLE II
EXECUTION TIME OF TRANSLATING THE THREE PROGRAMS. t DENOTES THE RUNNING TIME, IN SECONDS. μ DENOTES THE MEAN.

Program	$\mu(t)$
count	0.030
extended	0.030
commstime	0.032

TABLE III
EXECUTION TIME OF RUNNING THE THREE PROGRAMS, THROUGH OCCAM (USING KROC) AND GO. t DENOTES THE RUNNING TIME, IN SECONDS. μ DENOTES THE MEAN. Δ DENOTES THE SCALE FACTOR BETWEEN OCCAM AND GO.

Program	$\mu(t)$ Occam	$\mu(t)$ Go	Δ
count	0.003297	0.001636	2.02
extended	0.003494	0.001771	1.97
commstime	0.003528	0.002347	1.50

TABLE IV
MAXIMUM MEMORY CONSUMPTION DURING RUNNING THE THREE PROGRAMS, THROUGH OCCAM (USING KROC) AND GO. c DENOTES THE MEMORY CONSUMPTION, IN MEGABYTES. μ DENOTES THE MEAN. Δ DENOTES THE SCALE FACTOR BETWEEN OCCAM AND GO.

Program	$\mu(c)$ Occam	$\mu(c)$ Go	Δ
count	2.944	1.820	0.62
extended	3.024	1.935	0.64
commstime	3.152	1.910	0.61

1) Execution time of the translation: As we can see in Table II, translating the different programs does not diverge much. This is mainly due to the simplicity of the programs in question, but still indicates that the translation process is running smoothly.

2) Execution time of the resulting Go program: From Table III we see that the generated Go programs runs faster than the original Occam programs, meaning that the run time is not changed dramatically under translation. Therefore run time is not an issue when translating systems. It should be noted that the Occam programs ran within a virtual machine, whereas the Go program ran directly on the host machine.

3) Memory consumption: From Table IV we see that running Go executables use approximately 60% of the memory consumed when running the original Occam program.

TABLE V
STORAGE FOOTPRINT OF THE RESULTING BINARIES, THROUGH OCCAM (USING KROC), STATICALLY LINKED GO ($\mathbf{Go}_S$) AND DYNAMICALLY LINKED GO ($\mathbf{Go}_D$). s DENOTES THE STORAGE CONSUMPTION, IN MEGABYTES. Δ_S AND Δ_D DENOTES THE SCALE FACTOR BETWEEN OCCAM AND GO_S AND GO_D RESPECTIVELY.

Program	s **Occam**	s $\mathbf{Go}_S$	s $\mathbf{Go}_D$	Δ_S	Δ_D
count	0.016242	2.035064	0.552224	125.30	34.00
extended	0.016294	2.056320	0.557764	126.20	34.23
commstime	0.016344	2.043744	0.553336	125.05	33.86

4) Binary footprint: Table V shows the size of the binaries generated from KRoC versus the binaries generated from Go. Binaries generated by KRoC are dynamically linked, and are compared to both statically and dynamically linked Go binaries. We see that the generated Go_S binaries take up about 125 times the amount of space as the original Occam programs. This is a very big difference, and may affect the situations in which these translations are useful. If the hardware at our disposal is small, for example because of physical limitations, the generated Go binaries might take up too much space on the hardware. However, it should be noted that this is due to Go using statically linked libraries, making the binaries portable. Compiling with dynamic libraries reduces the Go programs to approximately 25% of the size, but then the perks of statically linked libraries are lost. If a smaller binary size is necessary, there are different levels of compression that can be done. Just by stripping the executable the size is reduced significantly, while still maintaining the perks of statically linked libraries.

VI. FUTURE WORK

The translator presented in this article successfully translates simple programs, but there are still significant deficiencies in the solution that leaves room for further improvement.

A. Support larger subset of Occam

An obvious idea for future development would be to extend the accepted subset of Occam. Ideally the transpiler would be able to translate any program written in the full Occam syntax, but to determine what to include first it would be interesting to do a thorough examination of existing Occam programs, to determine which language constructs are the most useful. As of now, I find that the following constructs could be interesting to include:

1) Abbreviations: Abbreviations specify names for variables, values, channels and timers, and can be used to easily reference existing variables etc.

2) Functions: Functions define value processes, which performs an enclosed process and produces one or more results of certain data types. Functions does not communicate with other processes or assign to free variables, and are thus free of side effects.

3) Timers: Timers allow us to represent time, and to delay processes until a certain amount of time has passed, or a specific time is reached. This is essential in most real time control systems. Syntactically timers are similar to channels, which should make expansion easier. Timers in Go are also syntactically similar to channels, e.g. we can block execution until a value is received on a timer's channel.

B. Verification

To decrease the amount of assumptions necessary when running a program through the code generator, it would be a good idea to implement some sort of preprocessing module for checking the semantics of the input program after it has been parsed. If it could be asserted that the program complied with the semantic rules of Occam, it would make more sense to let the generator assume that input was a valid program. This part is not strictly necessary in the sense that if the transpiler is only meant for translating programs that have already been verified, those programs are certain to comply with all the semantic rules. However, in practice it is a good assurance to have that the transpiler also checks the semantics of the program, to make for a greater use of the transpiler.

VII. CONCLUSION

We have presented a transpiler from Occam to Go that succeeds in translating small programs using a subset of the Occam syntax. We have shown that it is possible to generate Go programs performing the same job as the original Occam programs, and which maintains the basic concurrency principles of CSP. From benchmarking the solution it is evident that translation time is not significant, and that the run time of Occam programs and their equivalent Go programs is within the same scale.

This project also sheds some light on the possibility of using Go as an alternative to Occam. We have investigated how Go is based on the CSP principles and conclude that it could work as a good foundation for building concurrent systems that previously would have been built in Occam.

REFERENCES

[1] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[2] Wikipedia contributors, "Communicating sequential processes — Wikipedia, the free encyclopedia," 2020, [Online; accessed 28-September-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Communicating_sequential_processes

[3] ——, "Occam (programming language) — Wikipedia, the free encyclopedia," 2020, [Online; accessed 28-September-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Occam_(programming_language)

[4] M. Broløs, "Occam to go transpiler," https://github.com/MatildeBroloes/OccamToGoTranspiler.git, [Github repository for project].

[5] F. Barnes and P. Welch, "Getting started with KRoC," https://www.cs.kent.ac.uk/projects/ofa/kroc/getting-started.html, [Online; accessed 11-January-2021].

[6] P. H. Welch and F. R. Barnes, "Communicating mobile processes," in *Communicating Sequential Processes. The First 25 Years*. Springer, 2005, pp. 175–210.

[7] B. Vinter, L. J. Dickson, and J. Dibley, "Workshop on translating csp-based languages to common programming languages," in *39th WoTUG Conference on Communicating Process Architectures, CPA 2017 and 40th WoTUG Conference on Communicating Process Architectures, CPA 2018*. IMIA and IOS Press, 2019, pp. 135–156.

[8] Wikipedia contributors, "Go (programming language) — Wikipedia, the free encyclopedia," 2020, [Online; accessed 1-October-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Go_(programming_language)

[9] Golang.org contributors, "The go programming language specification," [Online; accessed 10-January-2021]. [Online]. Available: https://golang.org/ref/spec#StatementList

B

PROJECTS

This section contains details on all of the projects supervised as part of the Ph.D. They appear in chronological order within each section. Each entry will present the the title, my description of the project and its findings, and then abstract (or introduction + conclusion if there is no abstract). Appendix B.1 covers master thesis'. Appendix B.2 covers bachelor thesis'. Finally, Appendix B.3 covers projects in practice.

B.1 MASTER THESIS'

This section contains all of the master thesis' that were supervised as part of the Ph.D. These thesis' can be divided into 4 different main topics. Due to the nature of these projects, there exists some overlap, which is why one project might appear under several topics. They can be divided as follows:

- **X-ray Imaging**
  - Appendix B.1.1 - High Throughput Image Processing in X-Ray Imaging.
  - Appendix B.1.4 - Automating Classification in Food Inspection.
  - Appendix B.1.7 - Applied Super-Resolution for X-Ray Imaging - Virtual Potatoes And How To X-Ray Them.
  - Appendix B.1.8 - Automating X-Ray Inspection of Meat.
  - Appendix B.1.10 - Automatic Detection of Foreign Objects in X-Ray Images.
- **Machine Learning**
  - Appendix B.1.4 - Automating Classification in Food Inspection
  - Appendix B.1.8 - Automating X-Ray Inspection of Meat.
  - Appendix B.1.9 - Acceleration of Machine Learning through an FPGA.
  - Appendix B.1.10 - Automatic Detection of Foreign Objects in X-Ray Images.
- **Reconfigurable Hardware**
  - Appendix B.1.1 - High Throughput Image Processing in X-Ray Imaging.

  - Appendix B.1.2 - High Performance FPGA Firewall using SME.
  - Appendix B.1.3 - TCP/IP in Hardware using SME.
  - Appendix B.1.5 - A Bohrium to SME Translator.
  - Appendix B.1.6 - RISC-V Processor in SME.
  - Appendix B.1.9 - Acceleration of Machine Learning through an FPGA.
  - Appendix B.1.11 - Implementing Parts of Veros on an FPGA.
- **High-Performance Computing**
  - Appendix B.1.1 - High Throughput Image Processing in X-Ray Imaging.
  - Appendix B.1.5 - A Bohrium to SME Translator.
  - Appendix B.1.11 - Implementing Parts of Veros on an FPGA.
  - Appendix B.1.12 - Framework for Uploading Research data (FUR).
  - Appendix B.1.13 - Evaluation of Google TUPs for High Performance Physics Calculations.

B.1.1 *High Throughput Image Processing in X-Ray Imaging*

Troels Ynddal submitted his 30 ECTS master's thesis in computer science titled "High Throughput Image Processing in X-ray Imaging" in May 2019. Brian Vinter supervised it.

B.1.1.1 *Goals*

This thesis investigated improving and optimizing data quality in X-ray imaging. It both provided a set of methods for these improvements chosen through empirical evidence, both for CPU and FPGA.

B.1.1.2 *Abstract*

In this thesis, I will investigate the possibilities of improving and optimizing data quality in X-ray imaging. The different methods will be weighted by their effectiveness and tolerance for error. The unprocessed signal from the sensors will be calibrated and remapped to a linear range to compensate for anomalies. The signal from the sensors is prone to noise, which has a great impact on the quality of the output image. The noise will be reduced by applying an adaptive median filter and using multisampling to remove the background noise. The photo sensor is capable of producing up to 40 Gb/s, which has to be processed in real time to avoid data loss. Implementing this on a CPU or GPGPU would be nearly impossible with that data throughput. The goal is to implement these methods onto an FPGA, where it is possible to control the memory transfer very precisely, which may make it possible to process the 40 Gb/s without interruption. Implementing such an adaptive median filter on an FPGA is not a trivial task. Implementing the filter will require parallel memory controllers to correctly order the input signal and specialized sorting networks to find minimum, maximum and median values used by the filter. To gain the needed throughput, it requires integration with the low-level hardware and optimizing the physical routing on the chip. The final FPGA-design is able to process 6.25 Gb/s using a single pipeline on a $2,995 FPGA with great possibilities of deploying parallel pipelines to split the load and reach the goal of 40 Gb/s.

B.1.2 *High Performance FPGA Firewall using SME*

Patrick Dyhrberg Sørensen and Emil Sander Bak submitted their 30 ECTS master's thesis in computer science titled "High Performance FPGA Firewall using SME" in September 2019. Carl-Johannes Johnsen, Kenneth Skovhede, and Brian Vinter supervised it.

B.1.2.1 *Goals*

This thesis investigated implementing a firewall on an FPGA using SME. While they did not reach actual hardware before submission, they were able to before their defense, which hints towards their implementation holding up.

B.1.2.2 *Abstract*

This thesis presents a stateful firewall designed and created to run on an FPGA (Field-Programmable Gate Array). As the need for configurable high speed firewalls is ever increasing, a reprogrammable and hardware-based firewall is a potentially viable solution. This firewall has been created largely in a subset of the high-level programming language C# with the SME library, which has the ability to compile the written control logic into a HDL (Hardware Description Language). The design of the firewall has been made with SME's features and need of control logic in mind, with a large focus on speed and parallelization. The firewall itself is a stateful firewall, with the ability to monitor current connections, block both specific incoming and outgoing traffic, as well as prevent certain types of DDOS attacks. Tests performed on the firewall in a simulated system suggests that the correctness and stability of the firewall is valid. Due to the dependence on another master thesis project, as well as currently missing upcoming features in SME, the firewall has not been tested live on an FPGA, hence the tests are in a simulated system. One of the main goals achieved with this thesis, is to show that it is very much possible to create an extensible, reconfigurable hardware firewall capable of theoretically processing large amounts of traffic, and have it be made in a higher-level programming language like C#.

B.1.3 *TCP/IP in Hardware using SME*

Jan Meznik and Mark Jan Jacobi submitted their 30 ECTS master's thesis in computer science titled "TCP/IP in hardware using SME" in September 2019. Carl-Johannes Johnsen, Kenneth Skovhede, and Brian Vinter supervised it.

B.1.3.1 *Goals*

This thesis investigated implementing a TCP/IP controller on an FPGA using SME. Most of their time was spent on implementing the complex state machine entailed by the TCP protocol, whose dynamic nature makes it very fitting for a CPU compared to an FPGA.

B.1.3.2 *Abstract*

In this thesis, we design and implement a networking protocol stack in hardware using the Synchronous Message Exchange model – a new framework intended to help model hardware descriptions. The final pipelined design boasts with a decentralized memory model, with model division easily extensible and modifiable, closely resembling that of the architecture of the Internet Protocol Suite.

Initial tests performed on the simulated system with real captured network traffic suggests stability, promising protocol compliance, and an acceptable, though theoretical, performance. Numerous suggestions are discussed to improve the performance, such as widening the buswidths or replicating the stack itself to multiply the raw throughput.

Due to some trivial bugs and other minor missing features in the SME framework, the system could not be brought to the target FPGA hardware. However, we are optimistic that considerable performance is achievable with the current design, as well as great flexibility, extensibility and modularity of the system.

B.1.4 *Automating Classification in Food Inspection*

Aleksandar Topic submitted his 60 ECTS master's thesis in physics titled "Adaptive X-ray Inspection System (AXIS) - Automating classification in food inspection" in December 2019. Carl-Johannes Johnsen and Brian Vinter supervised it.

B.1.4.1 *Goals*

This thesis focused on building a toolset for adapting X-ray capturing setups for new use cases. It covered many topics in computer vision and Machine Learning (ML) with a focus on their application to X-ray images. The primary contributions of the thesis was a library for X-ray inspection systems, investigation of a content-aware resizing algorithm, and a convolutional neural network for the AXIS project.

B.1.4.2 *Abstract*

Current approaches in using X-ray scanners for food inspection are very case specific. It becomes cumbersome to adapt existing systems to new use cases, and with that costs increase. This work is a study on generalizing this process to become fully adaptive and automatic. First the basic principles of radiography are introduced. This helps us understand the production of X-ray images and their underlying characterization. One of the important pillars in this work is using machine learning to turn data into knowledge. The use of deep learning and more specifically convolutional neural networks have proven to be an immensely successful approach to study features in images. Before beginning the analysis of the X-ray images, we need to do some data preparation. The field of computer vision supplies many techniques for pre-processing, noise reduction and contrast enhancement. All of which are essential in order to highlight the finer details in the X-ray images. In this work we present image enhancement methods which prove to perform very stabily for two very different objects, such as potatos and oranges.

Collectively these tools lay the foundation for building the Adaptive X-ray Inspection System library (AXIS-lib). This library has been implemented with flexibility and modularity in mind. Starting at the scan process itself, an intuitive graphical user interface has been implemented for efficient data acquisition. All data is automatically uploaded to the Electronic Research Data Archive (ERDA) at University of Copenhagen. The library provides an efficient pipeline to go from raw X-ray scan to object classification in simple steps. Along the way it provides several tools and methods to probe a task within certain areas of classification in food inspection. When new types of problems arrise, the idea is that the library can be easily expanded. Thus building upon the existing framework, instead of starting from scratch each time.

By using the automated image enhancement and making regulated networks, the architectures developed in this work produce promising results on very small datasets. Identifying hollow heart potatos is done with an average top accuracy of 94% using only 18 individual potatos with 70 seperate scans and a 50/50 class representation. Finding needles in oranges is achieved with an accuracy of 81%, using only 12 individual oranges with 48 unique scans and also a 50/50 class representation.

We learn that while X-ray images of various foods differ a lot in their intensity spectrum, the images still share many common characteristics. This means that we can succesfully make some general

assumptions on the distribution of noise. This allows for some generalized image enhancement methods which help the networks towards consistent classifications. Lastly we conclude that the field of food inspection is only in its early development. Contemporary methods from AI and machine learning can tremendously increase efficiency and reduce waste in the industry. Hardware-based solutions such as using FPGAs for adaptive network classification looks very promising for future research in this field.

B.1.5 *A Bohrium to SME Translator*

Tor Skovsgaard submitted his 30 ECTS master's thesis in computer science titled "A Bohrium to SME translator" in May 2020. Carl-Johannes Johnsen, Kenneth Skovhede and Brian Vinter supervised it.

B.1.5.1 *Goals*

This thesis investigated using Bohrium as a front-end for FPGA development by translating into SME. It showed that through the intermediate representation from Bohrium, we could translate it into essentially a domain-specific processor for that exact input program.

B.1.5.2 *Abstract*

Utilizing direct and concurrent processing, the efficiency of data processing may be greatly improved, while simultaneously removing the need for intermediate data storage. The use of Field Programmable Gate-Arrays (FPGAs) is one possible approach to solving this challenge and to reducing power consumption. Programming in Hardware Description Languages (HDLs) are instrumental to writing code for FPGAs, but programming in these is notoriously tedious. This thesis presents an easy and efficient way of creating Hardware Descriptions by developing a transpiler from vectorized code to Hardware Descriptions, utilizing Bohrium as the front-end and SMEIL as the back-end. The code generated by the transpiler was tested and different versions were compared. Moreover FPGA was compared against a CPU and a General Purpose Graphics Processing Unit (GPGPU). While the FPGA versions differed widely in performance, the throughput of the CPU was outperformed by an order of magnitude and the power consumption by two orders of magnitude. Even solutions with less throughput outperformed the CPU on power per operation by an order of magnitude. These results indicate that the transpiler developed in this thesis can achieve similar performance as other frameworks, but with higher productivity and lower complexity.

B.1.6 *RISC-V Processor in SME*

Daniel Ramyar submitted his 60 ECTS master's thesis in physics titled "Implementation of RISC-V in SME" in March 2020. Carl-Johannes Johnsen and Kenneth Skovhede supervised it.

B.1.6.1 *Goals*

This thesis targeted implementing a RISC-V processor for FPGA by using SME. It built upon a previous master thesis [63], which built a MIPS processor in SME. Despite Daniel being a physics student with no prior hardware design experience, he successfully implemented a RISC-V processor in SME.

B.1.6.2 *Abstract*

Since the invention of the computer, it has become an essential tool in modern physics. Everything from the generation of advanced weather models to the fundamental act of measuring observables is now done with the aid of computers.

To address the ever growing demand for increased computational power and the inevitable end of Moore's law, alternate processor architectures, such as the RISC-V, has to be considered, as current standards are beginning to reach their limits.

In this project we have successfully designed, implemented and tested a RISC-V processor in Synchronous Message Exchange, which is a tool used for rapid development of circuits for Field Programmable Gate Arrays (FPGAs). Furthermore using vendor synthesis tools it has been shown to run at a theoretical speed of 124 MHz.

B.1.7 *Applied Super-Resolution for X-Ray Imaging - Virtual Potatoes And How To X-Ray Them*

Simon Nyrup submitted his 60 ECTS master's thesis in physics titled "Applied Super-Resolution for X-Ray Imaging - Virtual Potatoes And How To X-Ray Them" [15] in September 2020. Carl-Johannes Johnsen, Kenneth Skovhede and Brian Vinter supervised it.

B.1.7.1 *Goals*

This thesis attacked the problem of motion blur by utilizing super-resolution. As a side effect, the thesis also proposed an X-ray simulator, which can be used for generating artificial X-ray images. While super-resolution was deemed too computationally heavy to be used in the AXIS project, it still provided exciting results. The X-ray simulator provides a great tool for capturing X-ray images without an X-ray setup.

B.1.7.2 *Abstract*

In the field of food inspection, the implementation of X-ray scanners allows for a non-destructive analysis of the interior of objects. The scanner is a part of a larger pipeline that performs automatic classifcation by sending objects through the system on a conveyor-belt while doing real-time sorting. Such systems require a low latency with high precision, that allows for a high throughput of objects. The precision of such systems is dependent on the contrast and resolution of the X-ray images, but the various physical processes in the image formation degrade the final representation of the object and remove important features, that are essential to the classifcation task.

This project provides an investigation of applying multi-frame super resolution (MFSR) as a preprocessing tool for such a classifcation pipeline. MFSR is the concept of using latent information in a sequence of low resolution (LR) frames to produce a single high resolution (HR) image. The iterative re-weighted super resolution (IRWSR) algorithm, developed by [83], was implemented and applied to sequences of images acquired from an X-ray system. The resulting output shows no clear signs of feature enhancement. Furthermore, the runtime seems to be far from what is required for a high throughput system e.g. a $\times 2$ magnifcation of an image with size 200×200 is processed in the order of 1000s.

In the examination of the diferent limitations of X-ray setups, the simulation tool Xsim was created, capable of generating X-ray images of user-defned 3D objects. The versatile Xsim allows the user to simulate images for diferent X-ray source spectra, object compositions and geometrical distances. The generated images are perceptually comparable to real X-ray images, but lack efects such as scattering or beam hardening. The IRWSR algorithm was applied to a sequence of test images generated with Xsim. The resulting images contain no perceivable new information compared to the original LR frames.

This project gives an insight into the difculties of applying SR in real settings. Through the analysis and discussion of the various theories needed, a direction for further use is given. A further investigation should especially address the point-spread-function (PSF) of the X-ray imaging system in combination with a conveyor-belt, and the output should be validated with respect to the larger classifcation scheme.

B.1.8 *Automating X-Ray Inspection of Meat*

Jesper Rask Pedersen submitted his 60 ECTS master's thesis in physics titled "Foreign Object Detection in X-ray Images Ising Machine Learning" in December 2020. Carl-Johannes Johnsen, Kenneth Skovhede, and Brian Vinter supervised it.

B.1.8.1 *Goals*

This thesis looked at foreign object detection in meats by using machine learning on X-ray images. The implementation was an interesting approach, which did prove to have predictive power. We think that the model could be further improved by employing an ensemble-like approach, using the different windows as the means of voting.

B.1.8.2 *Abstract*

The development of better detectors, x-ray sources, as well as faster and cheaper computer processing resources, has made x-ray imaging have numerous applications: From airport security, through medical industry to food processing. One way to get more out of x-ray imaging is to use dual-energy x-rays. Dual-energy increases the possibilities with x-ray to measure the contents of the scanned objects. For many applications large amounts of data are collected since x-ray is introduced to do quality control. This is also the case for the food processing industry, where applications of x-ray imaging can be used in-line to scan the full production. The large amounts of data collected requires automated processing to be effective. This thesis explores Machine Learning in the data processing pipeline of a real world machine: The Meat Master II, which generates dual-energy x-ray images.

Concretely, the challenge is to detect foreign objects in the images. To detect the foreign objects a Convolutional Neural Network was trained. Furthermore, the use of synthetic data was explored. We find that it is possible to train a Convolutional Neural Network to 98.74% accuracy on detecting foreign objects, using a sliding window algorithm to preprocess the data from the Meat Master II. We observed a significant drop in accuracy when the model is evaluated on similar but yet unseen data. This is a significant issue since it is hard to guarantee that the training data represents the full test distribution. It is possible to alleviate this drop in performance, as measured by the Area Under the ROC-curve, by using our synthetic data in the form of artificial foreign objects. The accuracy is currently not good enough for real world use, but with a larger dataset to train on, it should be possible to successfully introduce machine learning to automate the detection of foreign objects in meat using machine learning.

B.1.9 *Acceleration of Machine Learning through an FPGA*

Amira Moussa submitted her 45 ECTS master's thesis in physics titled "Acceleration of machine learning through an FPGA" in May 2021. Carl-Johannes Johnsen and Kenneth Skovhede supervised it.

B.1.9.1 *Goals*

This thesis looked at accelerating a neural network on an FPGA through SME. The steps proposed in this thesis are general enough that we should be able to perform them automatically, gaining an ML model to SME translator. It should be further noted that Amira did not have any heavy programming experience, hinting towards the success of SME as a programming model for describing hardware.

B.1.9.2 *Abstract*

This thesis presents the design of an application specific hardware for machine learning which can be used in various physics applications, such as high energy physics and quantum optics. Even though we are better at optimizing and have more computational power was previously possible, there is also a continuous need to make simulations even faster, more reliable, and cheaper to run. We are specifically investigating a FeedForward Neural Network that is used to interpret market data feeds and hence enable minimal round-trip times for executing electronic stock trades. This is because there are similar trades such as hard time restriction, which is also the case for computational physics.

In this thesis the network is optimized to achieve the lowest possible latency. For this purpose, we use Synchronous Message Exchange Synchronous Message Exchange (SME) which is suitable for describing hardware and enables the flexibility to support a wide range of applied trading protocols. This demonstrates how to construct the components of a FeedForward machine learning script down to a processor as SME processes, and how to connect them by using SME busses. The complete system has been implemented in C# and evaluated on an Field Programmable Gate Array (FPGA). The results are promising compared to the Python implementation of the model. We present a proof of concept of an initial solution and its performance provides results that make us believe that a full Neural Network implementation would be feasible and competitive. The final result is a successful implementation of a FeedForward Neural Network model on a FPGA, which runs 21 times faster then the same algorithm on a CPU.

B.1.10 *Automatic Detection of Foreign Objects in X-Ray Images*

Alina Hjorth Sode submitted her 60 ECTS master's thesis in physics titled "Automatic Detection of Foreign Objects in X-Ray Images" in May 2021. Carl-Johannes Johnsen and Kenneth Skovhede supervised this.

B.1.10.1 *Goals*

This thesis focused on implementing automatic outlier detection by training purely on "good" samples. The resulting network performed very well on most datasets, although we suspect that the performance can be further improved.

B.1.10.2 *Abstract*

In this thesis an adaptive and automated pipeline for static novelty detection of foreign objects and other defectives in food products using X-ray imaging is demonstrated. First, the fundamental principles underlying the interactions of radiation with matter are presented, whereas the primary focus is placed on the contrast mechanism in X-ray imaging. To gain the necessary knowledge required to understand feature extractions in images, deep learning with images and more specially convolutional neural networks are presented. The unsupervised convolutional autoencoder (CAE) and convolutional variational autoencoder (CVAE) networks are trained using only normal samples, and their abilities at reconstructing inputs are utilized to successfully distinguish between normal and anomalous samples. But, before data are fed to the neural networks it however needs to be prepared, and the field of computer vision provides many techniques to do so, such as contrast enchantment, noise reduction and image augmentations.

Collectively these frameworks lay the foundation for the proposed pipeline as summarized by three main steps: (1) training an unsupervised deep model able to reconstruct normal samples so precisely as possible; (2) computing statistical distributions based upon a chosen anomaly score and; (3) threshold selection. Various anomaly scores are examined and compared, whereas we learn that the particular choice of anomaly score has a large impact on the evaluation scores, and the anomaly score that works the best varies from dataset to dataset. Moreover, the anomaly ratio have also been showed to have an impact on the scores, as the precision decrease with decreasing ratio.

Identifying anomalous chocolate bar from normal ones are achieved with a top accuracy of 99% using a 50% anomaly ratio by utilizing the zero-mean normalized cross-correlation (ZNCC), operating with only 10 individual chocolate bars and 22 separate scans.

Using only 50 individual potatoes with 540 distinct scans, whereas 196 scans are of inserted needles and 156 scans of artificial created hollow hearts, we find that the models repeatedly have an easier time detecting needles compared to hollow hearts. Utilizing the structural similarity index measure (SSIM) and a 50% anomaly ratio yields the top accuracy of 89%.

Additionally, the novelty pipeline was tested using a different potato dataset consisting of only 45 unique perfect potatoes and 66 potatoes with natural hollow heart disease, also given a 50% anomaly ratio. This scan data are inherently different from the former datasets, and the generative model was

found to have a more difficult time separating anomalous from normal samples, but by using the SSIM a top accuracy of 87% are achieved.

B.1.11 *Implementing Parts of Veros on an FPGA*

Joachim Kønigslieb submitted his 60 ECTS master's thesis in physics in May 2021. Carl-Johannes Johnsen and Kenneth Skovhede supervised it.

B.1.11.1 *Goals*

This thesis looked at implementing parts of VEROS on an FPGA by using HLS. Although not performing exceptionally well, Joachim did get running examples compared to a CPU or GPU implementation. This is primarily due to the examples already being optimized for matrix arithmetic, to which both the CPU and GPU are heavily optimized. He also investigated fixed-point precision over floating-point, which, as Xilinx suggests, should provide better runtime performance. However, we did not see a considerable improvement, suggesting another problem in the implementation.

B.1.11.2 *Abstract*

Climate simulation are very expensive computationally, which ironically have negative consequences on carbon emissions and leads to global warming. Existing large-scale climate simulations are generally run on archaic software stacks like C and Fotran and run on large cluster computers. These implementations are great from a pure performance perspective, but other productivity metrics are also important like ease of use and development speed. Leveraging the high productivity environment of Python to run high performance computing tasks trough the use of very efficient accelerators, like graphical processing units, have proven to be a viable solution. Here we target a hardware solution not widely used for HPC: Field Programmable Gate Arrays, as these have very favorable power/performance ratios. Trationally FPGAs have been too unwieldy to program large scale programs like climate simulations, but recent developments in high level synthesis have provided an avenue for the use of these types of chips. By exploiting the flexibility of FPGAs we aim to build flexible system that can run deep pipelines of the compute tasks at hand. A FPGA specific optimization, the use of alternative data encoding formats like fixed-point or posits is explored.

B.1.12 *Framework for Uploading Research data (FUR)*

Niels Andreas Tyndeskov Voetmann submitted his 30 ECTS master's thesis titled "Framework for Uploading Research data (FUR)" in March 2021. Carl-Johannes Johnsen, David Marchant, and Kenneth Skovhede supervised it.

B.1.12.1 *Goals*

This thesis looked at implementing an automated framework for uploading research data onto a cloud resource, such as ERDA. The project succeeded in implementing a package, upload, and cleanup program of data placed in certain folders.

B.1.12.2 *Abstract*

Capturing and storing large amounts of data on systems connected to digital measurement tools can be taxing on local storage capabilities. Therefore, captured data is usually transferred to a remote storage solution that offers centralized access to datasets. The transferred data can be packaged in advanced file formats in order to prepare it for being used efficiently in data analysis. The processes of packaging and uploading data are normally performed separately when data capturing has completed. This leaves room for improvement, as data entries can be packaged and uploaded continuously as they become available, instead of waiting for the entire data capturing process to complete.

This thesis presents the FUR framework to automate uploading, packaging and cleanup of data stored on systems connected to digital measurement tools. The main feature of FUR is continuously uploading data in batches to a HDF5 file stored on a remote storage solution and automatically remove local data once it has been uploaded. FUR uses remote file objects that are opened with the SFTP protocol to manage transferring data.

The results of testing the FUR implementation indicate that it is able to match the uploading speed of standard SFTP uploading methods when uploading typed data to a remote HDF5 file. The results also showed that using the implementation of FUR for uploading binary data to a remote HDF5 file introduces additional overhead, which leaves room for improvement. Finally, the results also indicate that FUR works cross platform and is scalable when used with faster bandwidth connections.

B.1.13 *Evaluation of Google TUPs for High Performance Physics Calculations*

Albert Alonso de la Fuente submitted his 60 ECTS master's thesis in physics titled "Evaluation of Google Tensor Processing Units (TPUs) for High Performance Physics Calculations" in May 2021. Carl-Johannes Johnsen and Kenneth Skovhede supervised it.

B.1.13.1 *Goals*

This thesis focused on utilizing a TPU for physics computations. In the past, the TPUs could only be used through the Tensorflow framework. However, Google has developed the JAX library in recent times, which allows for automatic differentiation of Python and Numpy code. The key feature of JAX is that it uses the XLA compiler, the backend compiler for Tensorflow graphs, in turn allowing Python and Numpy code to target TPUs.

B.1.13.2 *Abstract*

This thesis evaluates the use of novel AI chips as a solution for high performance large scale scientific simulations. By looking for fast and energy efficient accelerators, we assess the repercussions of porting physics calculations to Matrix Engines based accelerators, purposely designed to run efficiently the linear algebra found on Deep Learning workflows.

In our study, we focus on the use of Google's in-house Tensor Processing Units (TPU) as well as Google's machine learning research programming library JAX, due to it's versatility of backends and its similarity to the most used numerical python libraries. We present an alternative method to compute finite difference derivatives that leverages the matrix operation capabilities of TPUs outperforming by 2 times the performance of conventional vector approaches. Simulations whose main computing part mostly consists of matrix multiplications are found to be up to 3 times faster when used on a single core as opposed to their performance on a entire Graphical Processing Units.

We reproduce an implementation of the Ising Model developed on TensorFlow using our approach on JAX which indicates that the XLA compiler may performs better when used on TensorFlow graphs, despite JAX providing a more readable and familiar code.

Finally, we consider the viability of porting a general circulation model to make efficient use of TPUs while we outline the reasons to consider Matrix Engines as a viable scalable solution for certain type of physical simulations.

B.2 BACHELOR THESIS'

B.2.1 *Probability of Distress*

Ulv Gejr Gudmann Foerlev submitted his 15 ECTS bachelor's thesis in computer science titled "Probability of Distress" in June 2019. Carl-Johannes Johnsen supervised it.

B.2.1.1 *Goals*

This thesis looked at replicating the model described in a paper for predicting the probability of distress in danish companies. He successfully replicated their results, and through further investigation, we could see that by introducing a third class, the "maybe" class, the classifier would become very strong.

B.2.1.2 *Abstract*

Distress are when a company can no longer generate revenue and repay their financial obligations. When dealing with business-to-business trade, this comes with the financial risk that the business cannot pay back because they have become distressed. This means that accurate risk models are central to businesses, banks, and regulators that wishes to deal with businesses financially. We reproduced the results of the probability of distress model proposed by 2018 Christoffersen, and measured it's accuracy on the majority of financial reports from 2011 to 2016. We did this by implementing the gradient boosted tree machine learning algorithm described in their report and measured if the accuracy after tuning was comparable to theirs. Our findings suggested that their conclusion was accurate, and that a gradient boosted tree machine learning model leads to a more accurate probability of default model compared to the other models discussed in the report.

B.2.2 *Check-Pointing Long-running Applications in Python*

Jacob Olesen and Leeann Quynh Do submitted their 15 ECTS bachelor's thesis in computer science titled "Check-pointing Long-running Applications in Python" in June 2020. Carl-Johannes Johnsen and Kenneth Skovhede supervised it.

B.2.2.1 *Goals*

The goal of this project was to construct a framework for checkpointing long-running Python programs and to resume computations by restoring the state from these checkpoints.

B.2.2.2 *Abstract*

For long running `python` programs, it would be beneficial to have an easily implemented checkpointing functionality. Checkpointing is a technique for saving the state of a program in order to later resume from this exact state, to avoid data loss in case of system failure.

Partial solutions to this problem already exist, most notably the module `dill`, which allows for saving the entire interpreter session. We analyze the performance of this module as well as the feasibility of implementing the solution in arbitrary `python` programs. This module extends `python`'s own serialization module, pickle, and adds an acceptable amount of overhead, but it has severe limitations in terms of ease of use and low-level type support.

In order to overcome these shortcomings, a `C` level checkpointing solution which interacts directly with the `cPython` interpreter needs to be implemented, the strategy of which is analyzed and discussed.

B.2.3 *Training a Neural Network to Distinguish Between Potatoes with or without a Hollow Heart*

Søren Langkilde submitted his 15 ECTS bachelor's thesis in nanoscience titled "Training a Neural Network to Distinguish Between Potatoes with or without a Hollow Heart" in January 2021. Carl-Johannes Johnsen and Kenneth Skovhede supervised it.

B.2.3.1 *Goals*

This thesis looked at distinguishing between potatoes with and without the hollow-heart disease by utilizing neural networks.

B.2.3.2 *Abstract*

A hollow heart is a phenomenon that occurs in potatoes, that is essentially a hole in the center of them. This is a commercial problem, since this makes it harder for companies to ensure a homogeneous product and can in some cases result in losses of larger batches. This project revolves around making a solution to this problem by creating a convolutional neural network to classify x-ray images of artificial hollow heart (cut open, hollowed out and put back together). The classifier is compared to a crude classifier, that only evaluate how much of the image is covered and the neural network is superior, but still need some work.

B.2.4 *Emulation of the Nintendo Game Boy Color*

Christian Marslev and Jonas Grønborg submitted their 15 ECTS project in practice in computer science titled "Emulation of the Nintendo Game Boy Color" in January 2021. Carl-Johannes Johnsen, David Marchant, and Kenneth Skovhede supervised it.

B.2.4.1 *Goals*

The thesis looked at extending PyBoy, an open-source Python implementation of a GameBoy emulator, to emulate GameBoy Color ROMs. The project succeeded for most ROMs but failed for some doing exotic exploitation of hardware behavior not outlined in the documentation they could find.

B.2.4.2 *Introduction*

This project concerns our work on hardware emulation of the Nintendo Game Boy Color, known as the CGB-001 (CGB), released in 1998, the successor to the original Nintendo Game Boy, the DMG-01 (DMG). The Game Boy is a handheld game console, which was widely popular, with the DMG and CGB having combined for over 100 million unit sales until the CGB was discontinued in 2003.

The DMG and CGB had a very similar architecture with many games being supported on both systems. The most noticeably difference was that the CGB supported RGB color palettes with 15-bit colors as opposed to the monochrome palette with 2-bit colors on the DMG. Furthermore, the CGB had access to more RAM and doubled the cpu speed.

Instead of writing the emulator from scratch, we will be extending the already existing Game Boy emulator PyBoy to also support the emulation of CGB games. As the name might suggest, PyBoy is written in the Python programming language, but also uses the C-extension for Python, Cython, for increased performance. As so, this extension of the emulator will also be written in Python, possibly utilizing Cython for performance needs.

A big part of the project will be to examine the architecture of both the DMG and CGB and to understand which parts of the emulator will need to be extended or changed and in what way. It is known, that a CGB ROM do some checks to assert that it is on CGB hardware, so the main changes will be to make PyBoy accept a CGB ROM and to modify the video renderer. In this report we will describe each component changed in the PyBoy implementation in relation to the differences between the DMG and CGB hardware. Likewise, we will only describe the DMG components that are necessary to understand the CGB extensions. Finally, we will discuss and review the implementation based on the output of the ROMs chosen for testing and a rendering test ROM found online.

B.2.4.3 *Conclusion*

Several ROMs are playable on the emulator as it stands at the end of the project, despite some of the obvious missing functionality. Most of the work has been on understanding both the DMG and CGB systems such that we could extend the functionality of PyBoy. As such, most, if not all, problems with the original PyBoy will still be present in this new version, maybe with even more added by the additional complexity of emulating both systems.

The first feature added was the ability to start up a CGB ROM and pass the initial security check to convince the ROM that it did in fact run on a CGB system. This was done by modifying the value stored in the A-register immediately after boot-ROM completion in a second separate boot-ROM specific for CGB ROMs.

To implement the memory banks for both WRAM and VRAM, we refactored out the memory manager part of the original motherboard class. This way, we could create a subclass to extend the functionality needed by the CGB, but still have the same interface to it used from the motherboard class. The CGB subclass implemented the two new registers for controlling the banking in the two areas of memory.

Due to the way the original PyBoy handled DMA transfers, this could be reused when emulating the CGB. The general purpose transfer and the H-blank transfers had to be implemented from scratch. This was done based on the description of the functionality, but because of a few ambiguities on how the registers actually should behave, the actual implementation might not have the correct values in the registers at all times.

Speed switching is one of the distinctive feature of the CGB vs the DMG. However, this feature was not initially prioritized, as we did not expect it to have a big influence on the actual emulation. Furthermore, all of our initial test ROMs got to work without it. However, it was later discovered that some ROMs rely heavily on some features related to speed switching, as they did not run at all. Simply implementing the register that is used to control the speed switching behaviour convinced got these ROMs, such as Shantae, to run. The ROM renders mostly correct, however, there are still some issues with missing sprites.

The emulator was run with the cgb-acid2 rendering test, but did not pass. This further solidified that pieces of the emulator does not behave correctly or is completely missing. That said, the test shows that one of the problems has something to do with sprites not rendering at all, this was also present in Shantae. These issues can be caused by a variety of factors and are not necessarily related to the implementation of the renderer, as several test ROMs render correctly. A lot of time has been spent debugging and developing debugging tools should be a priority moving forward.

To sum up, a lot of progress has been made to make PyBoy a successful CGB renderer. Completely emulating a proprietary system with scarce documentation is an almost impossible task. To our knowledge, PyBoy can now succesfully emulate several ROMs, but a lot of work still remains. For future development, points have been made about where to continue the development.

B.2.5 *Occam to Go Translator*

Matilde Broløs submitted her 15 ECTS bachelor's thesis in computer science titled "Occam transpiler to Go" in January 2021. Carl-Johannes Johnsen and Kenneth Skovhede supervised it. Furthermore, the work has been published at the IEEE COPA conference Appendix A.1.4.

B.2.5.1 *Goals*

This thesis looked at implementing a translator from Occam to Go. This project was to "revive" the Occam programs without reimplementing everything from scratch. The project succeeded in providing a translator from the core subset of the Occam language.

B.2.5.2 *Abstract*

When building safety-critical systems it is important to be bale to prove that the code works as intended, to avoid any safety hazards. Proving correctness of code can be difficult, especially when the systems involve parallelism and concurrency. In 1978 Tony Hoare defined a formal language CSP that were to be the basis for future programming languages for building concurrent systems, and that formal language was build on an algebra so that it was easier to formally prove the correctness of the programs written.

Occam is a programming language built on CSP, and has been used for many safety-critical systems. However, the language has not been developed or maintained for the last 25 years, and this makes it difficult to maintain the programs which currently has a code base in Occam. As changing the entire code base for such systems will prove both expensive and time consuming,it is desirable to find an easy and secure way to translate Occam programs into another programming language.

To solve this problem this project attempts to build a transpiler from Occam to the newer programming language Go. The transpiler is implemented for a subset of Occam including only basic functionality, and is successful in translating simple programs from Occam to Go. The implementation is not fully functional, but it gives a good impression of how such a transpiler could be developed.

From this project it is evident that a transpiler between the two programming languages can be written, even though this solution is not fully functional. Implementing the full transpiler would require to find a way to assert that the generated code does in fact maintain the security of the original code, as it is very important that the generated code is free of e.g. deadlocks. The benefit from such a transpiler would be that Occam programs could be translated quickly and easily, but also that it would give security as one could rely on the generated program to be as correct as the original Occam program

B.2.6 *Cryptographic Library for FPGAs*

Jacob Herbst and Jonas Flach Jensen submitted their 15 ECTS bachelor's thesis in computer science titled "Cryptographic library for FPGAs" in June 2021. Carl-Johannes Johnsen and Kenneth Skovhede supervised it.

B.2.6.1 *Goals*

This thesis looked at building a cryptographic library for FPGAs by using SME. They succeeded for all of the algorithms they tried, but only some of them proved to beat a CPU or GPU. It should be noted that most of these algorithms have received far more development and optimizations for CPU and GPU, and even dedicated ASICs for some of them. However, when compared to a CPU of similar cost to their target FPGA, they did win.

B.2.6.2 *Abstract*

Computer security and cryptography is ubiquitous and is a critical aspect of computer science. Often cryptography is handled in the CPU as with the general majority of computing. Still, in some cases, CPUs might be a suboptimal solution, for instance, when low power consumption is critical. In such a case, a Field Programmable Gate Array (FPGA) is a good alternative. This report will present a library of four cryptographic functions designed for FPGA devices: MD5, SHA256, AES, and ChaCha20. We will present the underlying algorithm of the four functions and how we have implemented these using a high-level programming model Synchronous Message Exchange (SME) in C# instead of the usual approach of using a Hardware Description Language (HDL). We will further present how we, by pipelining, have achieved performance comparable to a CPU of a similar price range at much lower power consumption. In the process, we have tried investigating different parts of FPGA programming and how this can be applied in SME to see if it would further improve performance. We were, however, not able to achieve this and hence included some reflection on how approachable FPGA programming is using a high-level model such as SME. Code for the project can be found at: https://github.com/Spatenheinz/Bachelor

B.3 PROJECTS IN PRACTICE

B.3.1 *A Generic Buffer Management for High Performance FPGA Systems*

Rune Taj Clemens Petersen submitted his 7.5 ECTS project in practice in computer science titled ”A generic buffer management for high performance FPGA systems” in June 2019. Carl-Johannes Johnsen and Brian Vinter supervised it.

B.3.1.1 *Goals*

The project looked at implementing a memory interface for a vector processor in FPGAs using SME.

B.3.1.2 *Introduction*

In this project we will analyze and implement a memory interface for a vector processor. The language and framework is C# and Synchronous Message Exchange (SME). The C# and SME framework compiles to VHDL that is a hardware description language (HDL) for programming field programmable arrays (FPGAs). This project is a part of a larger project (BPU) of building a vector machine synthesized on a FPGA.

The memory bandwidth is one of the tightest bottlenecks in a computer system and has traditionally been partially achieved or accelerated with cache systems that assumes spatial locality. The cache could be made more efficient and fast if it knew what blocks of memory the CPU needed, a bit like it was done for the Cell processor. The Bohrium project is a compiler with Python/Numpy like syntax that actually knows which memory blocks it is going to need. The idea is to convey those ranges of memory that is needed, and ranges of memory that needs to be written, to a memory controller that carry that task in the background.

B.3.1.3 *Conclusion*

The goal of the project was to build a generic memory interface. Although the implementation works, it is not a plug and play solution to the target platforms, Intel and Xilinx. The reason is that the interfaces from the Logical Arrays to the DRAM or other peripherals needs to be adjusted to the particular implementation. The most universal interface without getting a too specific implementation was to go with the Memory Mapped version (AXI-MM).

The AXI has a lot of revisions through the years and have furthermore several versions with different functionality. Finding the right revision and version was cumbersome. Retrospectively, I should have focused on the chosen AXI version for the target platforms. The lack of this focus had the consequence that I had to prioritize, if I should implement a more mature AXI or finish the overall design with the BRAM implementation. The choice fell out to be expanding the tests to include address setup and latency. This revealed a few errors in the memory controller. I believe that the maturity of the system has been greatly improved by the ease of writing in a high level language that the SME project made possible.

B.3.2 *An Object Store for FPGA*

Tor Skovsgaard submitted his 7.5 ECTS project in practice in computer science titled ”An objectstore for FPGA” in June 2019. Carl-Johannes Johnsen and Brian Vinter supervised it.

B.3.2.1 *Goals*

This project looked at implementing an object store on FPGAs by using SME.

B.3.2.2 *Introduction*

To increase computing power and decrease the energy consumption programming Field Programmable Gate-Arrays (FPGA) is becoming increasingly interesting. To fit the program on FPGAs it needs to be compilable in the synthesisable subset of VHDL or verilog also known as Hardware Definition Languages (HDLs). As these are very old and lack most of the productivity enhancing features we opt instead to use of a new framework, Synchronous Message Exchange (SME). Having the advantage of being a subset of CSP the programming is more productive, especially for anyone familiar with the spliting of programs into processes. We introduce an object-storage that receives the objects, splits them up into seperate values and then sends them through a number of filter-processes and writes the proper values.

B.3.2.3 *Conclusion*

Implementing this object-store system proved to be a harder task than first anticipated. Three different design-strategies having been tested and two of them being synthesisable to hardware. We feel like the third implementation is very fitting for usage in hardware but the hardware-mindset still feels disconnected from the regular sofware-oriented programming for CPUs and even that for GPUs and clusters.

The high productivity and familarity of C# has certainly helped in the developing-process. This even though we had no experience in C# specifically before this project. When changing the system in major ways it is nice to be able to test that it has some kind of functionality within a shorter time-frame. Turning the VHDL generation off should be warned against when still familiarising oneself with the SME-framework while wanting it to be transpilable into VHDL.

While it still takes time to get into the mindset of hardware vs software SME helps a lot. Even with the transpilation that keeps most names somewhat intact, running through the VHDL code is still somewhat of a task. This inspite of the code looking a lot more like handwritten code than most generated code we have seen.

B.3.3 *PyBoy Rewind*

Jacob Olesen submitted his 7.5 ECTS project in practice in computer science titled ”PyBoy Rewind - Extending a Gameboy emulator to rewind gameplay” in October 2019. Kenneth Skovhede supervised it.

B.3.3.1 *Goals*

This project aimed to implement the rewinding functionality to PyBoy, an open-source Python implementation of a GameBoy emulator.

B.3.3.2 *Abstract*

This project covers the implementation of a rewind functionality in PyBoy, a Gameboy emulator written in Python. As this is a non-standard feature of the GameBoy, the majority of the work has been analytic in order to find the most efficient way to implement the feature. It not only covers a handful of implementations, but will also discuss possible extensions and alternative implementations.

The result is an implementation which is fast enough to run the Gameboy at it’s original speed of $\approx$ 60 frames per second, which is spacially efficient, but can be improved upon.

B.3.4 *Emulation of the Game Boys Link Cable*

Jonas Flach-Jensen submitted his 7.5 ECTS project in practice in computer science titled ”Emulation of the Game Boys Link Cable” in November 2019. Kenneth Skovhede supervised it.

B.3.4.1 *Goals*

This project aimed to introduce the link communication functionality to PyBoy, an open-source Python implementation of a GameBoy emulator.

B.3.4.2 *Abstract*

This project is covering the implementation of a Link Cable emulation to an existing Game Boy emulator PyBoy. The Link Cable functionalities is made from the known hardware specifications and has proven to be able to transfer data between two emulation processes. However the emulation is flawed and leads to wrong data being sent, which leads to a wrong output of the emulation. The implemented Serial class doesn’t handle all errors. It is however simple, efficient and should be easily upgraded.

B.3.5 *Cloud Storage Solution for Industrial Camera*

Niels Andreas Tyndeskov Voetmann submitted his 7.5 ECTS project in practice in computer science titled ”Cloud Storage Solution for Industrial Camera” in November 2020. Carl-Johannes Johnsen and Kenneth Skovhede supervised it.

B.3.5.1 *Goals*

This project looked at implementing a generic framework for uploading data captured from a smart camera and uploading it to a cloud resource, such as ERDA. The project succeeded in uploading data straight from the camera to ERDA.

B.3.5.2 *Introduction*

Scientific analysis of images is used in many fields and industries, but can rely on having access to large amounts of image data or the necessary computational power, or both. The general solution to storing large amounts of data intended for data analysis is to store it on specialized cloud servers. The idea is to have the resources and storage of a cloud solution made accessible to users through a remote connection, such that accessibility and computational power for data analysis is not limited by the resources of personal computers. However, no general solutions or frameworks that combine data processing and transferring of image data from source to cloud are available. This project will implement a cloud storage solution for an embedded Linux system connected to an industrial camera. The goal of the implementation is to automate transport of data from the camera to the cloud storage. This will allow the camera to be used in an industrial setting, while its images can be analysed by a separate and more powerful system. The implementation will focus on optimizing processing and the transfer of image data, with regards to quality, speed and precision. The performance of the implementation will be compared for different data sizes and pixel resolutions, and analysed based on the hardware limitations of the camera and its system.

B.3.5.3 *Conclusion*

This project has successfully integrated a functioning tool chain to support cloud storage on an industrial camera. The project has provided a thorough analysis of the data flow and possible bottlenecks for the chosen model. It has further analysed and discussed design choices that lead to the chosen model. The project has delivered an implementation that is tailored to the hardware and specifications of the industrial camera and embedded Linux system, in order to optimize speed and precision of the end result. It has tested the implementation in order to showcase performance, expected quality and precision for different resolutions and capture rates. This project has not reinvented the wheel, but provides a solution based upon solid frameworks and software libraries to get the job done.

C

HARDWARE

This chapter describes the different hardware platforms used during this thesis.

TUL PYNQ

This is the TUL PYNQ-Z2 board [84], featuring a Xilinx Zynq chip.

CPU	650 MHz dual-core Cortex A9
RAM	512 MB DDR3 with 16-bit bus @ 1050 Mbps
FPGA	Xilinx Zynq XC7Z020
LUT	53.2K
FF	106.4K
BRAM	140 (4.9 Mb)
DSP	220

LAPTOP

This is a Lenovo t470p

CPU	3.80 GHz quad-core Intel i7-7700HQ
RAM	32 GB DDR4 @ 2400 MHz
GPU	NVIDIA 940MX
Disk	1 TB NVMe Samsung 970 EVO Plus

THREADRIPPER

This is a workstation used as a primary build machine.

CPU	3.40 GHz 16-core AMD 1950X
RAM	128 GB DDR4 @ 3200 MHz
GPU	NVIDIA RTX 2080 Ti
Disk	RAID0 - 2 TB NVMe Samsung 970 EVO Plus

FPGA0

This is a build machine that I gained access to through the SPCL group at ETH Zurich.

CPU	3.40 GHz 16-core AMD 5950X
RAM	128 GB DDR4
FPGA	Xilinx Alveo U280
LUT	1304K
FF	2607K
BRAM	2016 (72 Mb)
UltraRAM	960
DSP	9024
HBM	8 GB @ 460 GB/s
DDR	32 GB @ 2400 MT/s

QTEC CAMERA

This is the camera from QTec, employed in the most recent AXIS prototype.

CPU	1.65 GHz dual-core AMD G-T56N
RAM	4 GB DDR3-1333
GPU	Radeon HD 6320
FPGA	Xilinx Spartan 6 XC6SLX100T
LUT	101K
FF	126.8K
BRAM	268 18Kb (4.8 Mb)
DSP	180