

University of Twente

Signal Recovery using ClaSH

D.J.M. Wentink
s1612034

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)

Computer Architecture for Embedded Systems (CAES)

Contents

1 Abstract	1
2 Introduction	2
2.1 Problem description	2
3 Physics background	4
3.1 Ion Mobility Spectrometry	4
3.2 Maximum Likelihood for signal recovery	6
4 Computer architecture background	14
4.1 FPGA	14
4.2 CLaSH	14
5 Design space exploration	18
5.1 Optimization algorithm	18
5.2 Platform	18
5.3 Development language	20
5.4 Architecture	22
5.5 Memory	23
6 Implementation	25
6.1 Software implementation of algorithm	25
6.2 Processor design using CLaSH	29
7 Conclusion	39
7.1 Future work	39
Bibliography	40
Appendix A: Haskell implementation	43
Appendix B: Naive CLaSH implementation	45
Appendix C: Processor architecture	47
Appendix D: Processor program	53

1 Abstract

D.J.M. Wentink, Computer Architecture for Embedded Systems, University of Twente
Abstract of Masters Thesis: Signal recovery using clash
Friday 15th December, 2017

Ion-mobility Spectrometry (IMS) devices are used for identifying small amounts of substances in an air sample. A problem in increasing the level of detection is the effects of noise and finite resolution. Signal recovery algorithms are used to recover the input signal of a system with a known response, and can be used to improve the level of detection of IMS devices. A signal recovery algorithm based on the Maximum Likelihood (ML) principle approaches the limit on information that can be recovered from noisy data, but is computationally very complex. The computer systems that are currently used to solve ML problems are not suited for portable devices.

A field-programmable gate array (FPGA) is a reconfigurable chip which is potentially more energy efficient than traditional computer systems. C λ aSH is a high-level programming language for creating synthesizable FPGA architectures which is well-suited for mathematical algorithms. In order to downsize IMS systems, an energy efficient FPGA implementation of a ML-based signal recovery is made using C λ aSH.

In the background study the ML signal recovery problem is analysed mathematically, and it is shown to be an optimization problem. The conjugate gradient algorithm can be used to solve this problem in a fixed number of iterations. The computational complexity comes from a large amount of matrix-vector multiplications.

A processor architecture has been designed to solve the ML signal recovery problem in a fast and energy efficient manner. The architecture uses parallel computations, multiply accumulators and a custom memory architecture to be fast at matrix-vector multiplications.

To determine energy efficiency, the FPGA implementation is compared to an implementation using a modern graphics processing unit (GPU). The FPGA is at least a factor 8 more energy efficient, at the cost of processing time. In the worst case, the FPGA takes 1.7 seconds of computation time whereas the GPU needs 0.7.

2 Introduction

This document describes the design and implementation of a maximum likelihood signal recovery algorithm using a FPGA and the programming language Clash.

Signal recovery is the problem of finding the unknown input signal of a system with a known response, given the measured outputs of that system. Signal recovery is often used with sensor platforms where the signal is heavily affected by noise or requires a very high accuracy.

A lot of different signal recovery algorithms exist with a varying degree of accuracy and computational complexity. The highest accuracy that can be obtained is defined by Shannon's theorem on channel capacity in the presence of noise (Kosarev, 2002). The algorithm described in this thesis is chosen because it approaches the limit of information that can be recovered from noisy data.

An interesting area of applications is the field of Ion Mobility Spectrometry (IMS). IMS is a technique used for rapid identification of small amounts of substances. The signals from an IMS sensor are pulsed, very noisy and require a high accuracy, making it an ideal field for developing signal recovery algorithms. Although the algorithm is designed to be used with an IMS sensor, it can be used for other sensors as well.

Using an FPGA to minimize and reduce energy consumption of an IMS system has been done before (Loo et al., 2008), but not for the signal recovery part. Optimizing and accelerating signal recovery algorithms based on maximum likelihood is an active area of research in the field of image restoration (Lanteh Ri et al., 2001) (Holmes and Liu, 1991), but the algorithms used in image restoration can't be applied directly for signal recovery.

The goal of this study is to implement a computationally complex but effective algorithm which results in a high resolving power in a way that uses less energy. In the future this maybe combined with other functionality on the same FPGA, such as sampling the signal or controlling the gate.

2.1 Problem description

In a lot of embedded systems, measurement systems are used as part of a feedback loop or in order to provide information. If the requirements of the measurements are strict, these measurement systems can be quite complex. Some of these embedded systems have a limited power and/or size budget, for example portable devices, and therefore the complexity of the measurement systems is a problem.

Systems measuring physical quantities are subject to measuring-noise and finite resolution, deteriorating the accuracy of the measured results. The goal of signal recovery is to recover the unknown input signal of a system with a known response as accurately as possible, taking into account the measuring-noise and finite resolution in the measured results.

The response of a system is the mathematical function which defines the relation between the input and output signals. In the ideal world, signal recovery applies the inverse of the system response to the measured output to obtain the exact input. The problem with inverting the system response is that it requires differentiation, which increases the effect of measuring-noise and finite resolution. In the 80's, most signal recovery problems were solved using the Fourier transform (Saxena and Singh, 2005). Using the regular and inverse Fourier transformations the need for derivation is removed, but the inverse Fourier transform limits the accuracy of measured results (Hayes, 1981) (Espy and Lim, 1983).

There are algorithms which don't require differentiation or Fourier transforms to solve signal recovery problems (Soussen et al., 2010). Generally speaking the choice of algorithm is a trade-

off between the desired quality of the results and the processing power/time required for the calculations.

There is an upper bound on the quality of the recovered signal, which is defined as the highest resolution obtainable by the complete measurement system. This optimum for signal recovery is defined by Shannon's theorem on channel capacity in the presence of noise (Kosarev, 2002).

Some algorithms can achieve a resolution close to Shannon's limit. These near optimal signal recovery algorithms bypass the need for differentiation, but instead solve an optimization problem with many independent variables. There are several methods which can be used to solve optimization problems but in all cases the computational complexity is very large due to the amount of computations required to solve the optimization problem (Michalewicz, 1995).

Solving a computationally complex problem in a small amount of time requires a lot of processing power. Processing power generally comes at the expense of energy consumption and heat generation, which are unwanted characteristics for portable devices. Solving complex signal recovery problems in a near-optimal way is ill suited for hardware compatible with portable devices. In order to be able to use portable devices for complex signal recovery problems, the energy efficiency needs to be improved.

3 Physics background

3.1 Ion Mobility Spectrometry

Ion Mobility Spectrometry (IMS) is a fast analytical technique used to separate and detect ionized molecules based on their mobility in an electric field (Eiceman and Karpas, 1995). The first study on the motion of ions in an electric field has been done in the early 20th century by Dr. Paul Langevin (1905). The first known instrumentation has been developed as *Plasma Chromatography* in 1970 (Karasek, 1970).

Currently, the technique is mostly used for the detection of explosives, drugs and dangerous chemicals (Zolotov, 2006). The technology is commercially exploited by various manufacturers (among others, Morfo-Safran (FR), Smith Detection (UK), Implant Sciences (US)).

Although there are multiple different IMS techniques, such as Drift-Time IMS, Travelling Wave IMS, High-Field Asymmetric Waveform IMS, Trapped IMS, and Open Loop IMS, the key principle is the same (Cumeras et al., 2014). The specific form of IMS explained is Drift-Time IMS. IMS measures the *Electrical Mobility of Ions* in a gas when exposed to an *Uniform electrical field*, these terms will be explained below.

3.1.1 Ions

To understand ions, the concept of *molecules* must first be explained. A molecule is the smallest particle of a substance which can exist on its own. A molecule consists of two or more *atoms* and the arrangement of these atoms defines the properties of the substance. Examples of molecules are Water (H_2O), Methane (CH_4) and TNT ($C_7H_5N_3O_6$). The properties of a molecule range from weight and melting point to the ability to react with other molecules. Molecules can have properties which are highly unwanted in certain scenarios, such as being very flammable or poisonous.

Ions are molecules with a *charge*. The charge is caused by an imbalance between the total amount of *protons* and *electrons* in the ion. Ions have several interesting properties because of this charge, the most significant being that an ion reacts to an electrical field. A ion inside an electric field will start moving. The mobility of an ion is the proportionality factor between its drift velocity and the strength of the electrical field the ion resides in.

The process of creating ions from molecules is called *ionization*. There are many methods of ionization, but not all methods can be used for all kinds of molecules (Louis and Hill, 1990). Depending on the substances which should be detected, one or multiple ionization techniques are required. The algorithm described further on can be used independent on the selection of ionization source.

Most ionization sources work better when the *electrical affinity* of the molecules is higher. One reason for the popularity of IMS in the defence and security industries is that the electron affinity of explosives is higher than most common substances, so they can be detected relatively easy.

3.1.2 Uniform electric field

In order to measure the mobility, the drift time of the ion needs to be measured while the other variables are known. The easiest way is to generate an electric field with a known field strength and *uniform field lines* in a controlled environment, called a *drift tube* (Eiceman and Karpas, 1995).

Uniformity of the field means that the magnitude and direction of its field lines are all equal. Inside an uniform electric field, all ions experience the same force applied regardless of posi-

tion, which means their acceleration and speed rely solely on their mobility. The more uniform the electric field is inside the drift tube, the better the measurements become.

At the end of the drift tube is a collector, where the ions lose their charge. The loss of charge results in an electrical signal which can be measured. This signal plotted against time results in a spectrum of ions received per time unit.

3.1.3 Electrical mobility

Electrical mobility is the ratio between velocity of a charged particle and intensity of the electric field at a given temperature. Each type of molecule has a unique mobility constant which can be used to detect the molecule. The mobility constant depends on the shape, size and weight of the molecule. When the intensity of the field is constant and the time can be measured, it is possible to identify molecules based on their mobility constant.

3.1.4 Drift Time Ion Mobility Mass Spectrometry

As has been explained in the previous sections, it is possible to measure the electrical mobility of an ionized molecule using an electric field inside a controlled environment. *Drift Time IMS (DTIMS)* systems use this effect to generate a spectrum of the molecules inside a gas by "racing" the molecules (Cumeras et al., 2014). In figure 3.1 a schematic overview of DTIMS is shown.

At the start an air sample is taken, for the best results this air sample should be of a consistent temperature, humidity and gas mixture. The air sample then is sent to the ionization chamber. Using an ionization method some of the molecules present in the air sample are ionized. In DTIMS the ionized molecules are prevented from entering the drift chamber by a Gate, until a fixed starting time. In the racing analogy the Gate would be the starting line.

At time 0, the gate opens and the ions accelerate through the drift tube towards the collector, corresponding with the finish line in the analogy. The ions reaching the collector generate an electric signal, which can be plotted against time. Inside a controlled environment molecules will arrive grouped by their mobility, and the amplitude of the signal represents the amount of molecules with the same mobility arriving at a certain time.

The resulting plot resembles the mobility of molecules present in the air sample.

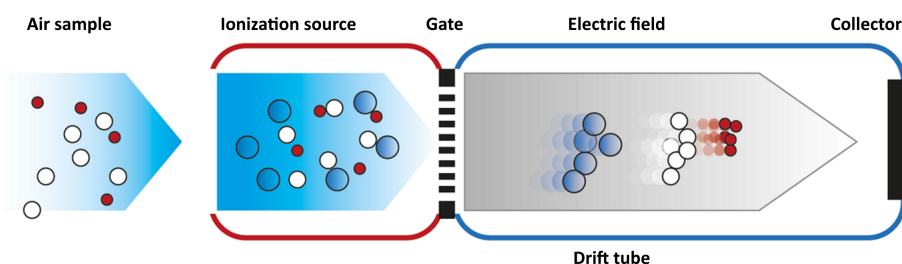


Figure 3.1: Schematic representation of IMS.

The spectrum generated by IMS devices can be used to determine whether a certain wanted or unwanted molecule is present in a carrier gas. The main advantage of IMS is that the detection level - the lowest detectable quantity of a substance - is very low. A disadvantage of IMS is that the voltage across the collector is very low, which means that the effect of noise can easily corrupt the signal. It also means the resolution of the measurement system needs to be very high.

3.2 Maximum Likelihood for signal recovery

The goal of signal recovery is to recover with maximum accuracy the unknown input signal of a system, taking into account the measuring-noise and finite resolution in the measured results. In simpler terms the output of a system is measured and then mathematics are used to estimate the corresponding input of the system.

The output of a system consists of desired information, the response to its input, and unwanted information generated by the system such as noise. The Maximum Likelihood principle is used to distinguish wanted and unwanted information, so that the latter can be reduced until only the wanted information is left. The reduction is done using Conjugate Gradient.

3.2.1 Maximum likelihood

The first important part of the algorithm is Maximum likelihood. There are less complex methods to estimate the input signal of a system, but the maximum likelihood method is chosen because it obtains an information density close to the optimum. Maximum likelihood is the concept of finding the set of input values which have the highest likelihood of having produced the measured output.

The easiest way to think of likelihood is as if it is a probability, as the two are mathematically related. In statistics, the distinction between probability and likelihood is that, while probability describes possible future outcomes from a known input, likelihood starts from a known output and describes possible inputs.

In figure 3.2 the probability distribution for a single input sample is visualised for the system $y = x$ with *dispersion* 0.2. If the value 0.4 is inserted, the most probable outcome is 0.4, but the value can be anything between 0.3 and 0.5. Likelihood is reasoning the opposite way, if the value 0.4 is measured for this system, the most likely input value is 0.4.

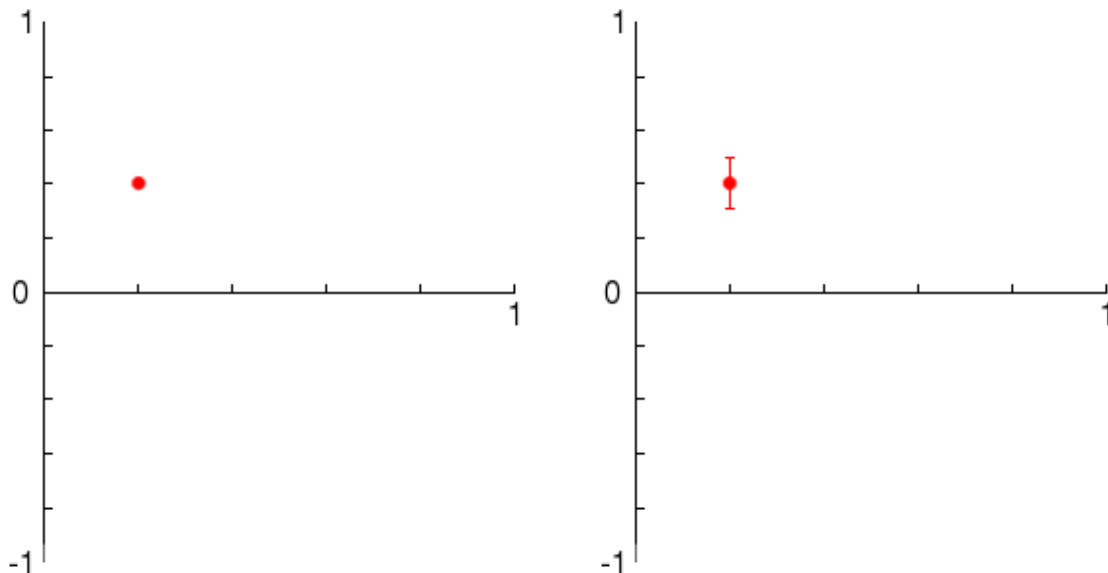


Figure 3.2: Probability of obtaining output sample (right) from known input (left)

The mathematical side is that likelihood is a distribution of possible input values, centred around the most likely input value according to the system response function. Small errors

are more likely than large errors. The mathematical definition is shown in equation 3.1. For a single sample likelihood is not a very useful concept.

$$\Phi(k) = e^{-\frac{(S(k)-S(k)^{ideal})^2}{\delta^2}} \quad (3.1)$$

Likelihood $\Phi(k)$ to obtain measured sample $S(k)$ is defined as the difference between sample $S(k)$ and the *ideal signal value* $S(k)^{ideal}$, compared to the *dispersion* δ . The ideal signal value, or *signal corrupted by response*, is a free variable corresponding with the output of a system without corruption by noise, which will be further explained in chapter 3.2.2. Dispersion is the statistical standard deviation of the noise distribution function, the square of it is called *variance*.

Likelihood is more useful with more samples, as shown in figure 3.3. Connecting the blue dots gives a wrong signal, while the red signal might have been reconstructed if the errors had been filtered out. One of the methods to remove the error is to search the most likely signal.

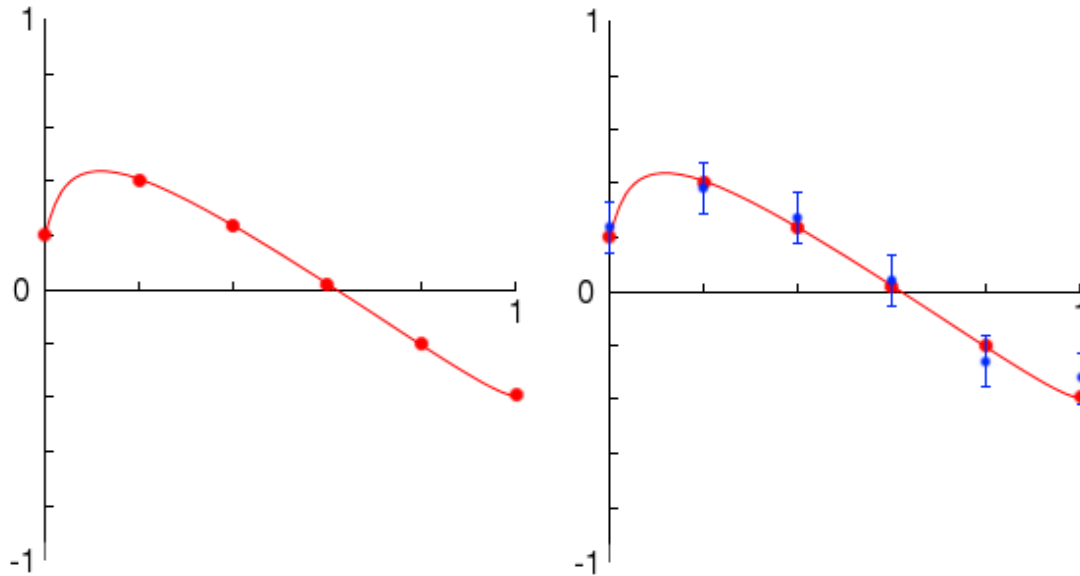


Figure 3.3: Probability of obtaining input signal (Red) from output samples (Blue)

The most likely signal is the signal which has the least cumulative random error. It should be noted that maximum likelihood is not about correcting for behaviour of the system, such as delay or dampening, but about reducing errors. Behaviour of the system should be captured in the system response function in order for maximum likelihood to work well.

The equation for signal likelihood, or total likelihood, is the product of individual likelihoods:

$$\Phi = \prod_k \Phi(k) = \prod_k e^{-\frac{(S(k)-S(k)^{ideal})^2}{\delta^2}} \quad (3.2)$$

In order to find the point where the likelihood is maximum, a transformation needs to be applied to 3.2, which is shown in equation 3.3.

$$\Phi = \prod_k e^{-\frac{(S(k)-S(k)^{ideal})^2}{\delta^2}} = e^{-\sum_k \frac{(S(k)-S(k)^{ideal})^2}{\delta^2}} \quad (3.3)$$

In equation 3.4 the ideal signal values and measured output values are written as vectors. Since the measured values are known, the parameter is the ideal output signal S_k^{ideal} . In order to

solve this equation the dispersion should be known, but later it will be shown that this is not necessary for Maximum Likelihood.

$$\Phi(S_k^{ideal}) = e^{-\sum_k \frac{(S_k - S_k^{ideal})^2}{\delta^2}} \quad (3.4)$$

The intuitive proof can be obtained by modifying this equation and looking at the limits. If S_k , the measured signal, is split into the ideal signal S_k^{ideal} and error ϵ_k , equation 3.4 becomes equation 3.5. If the error at all measured points approaches zero, the likelihood that the input signal matches the matching measured output signal approaches 100%. If the errors decrease, the likelihood increases.

$$\Phi(S_k^{ideal}) = \lim_{\epsilon \rightarrow 0} e^{-\sum_k \frac{(\epsilon_k + S_k^{ideal} - S_k^{ideal})^2}{\delta^2}} = \lim_{\epsilon \rightarrow 0} e^{-\sum_k \frac{\epsilon_k^2}{\delta^2}} = 1 \quad (3.5)$$

Since the absolute value of equation 3.4 is not important - the goal is to find the point where the likelihood is highest - this expression can be simplified by removing the natural logarithm. Because e^{-x} is *monotonically decreasing* over the entire domain, the following property can be used:

$$\text{global maximum of } e^{-x} = \text{global minimum of } x \quad (3.6)$$

The equation can also be multiplied with δ^2 to remove that constant. Maximizing the likelihood can be done by minimizing equation 3.7.

$$\Phi(S_k^{ideal}) \approx \sum_k (S_k - S_k^{ideal})^2 \quad (3.7)$$

The outcome of equation 3.7 is that the ideal signal S_k^{ideal} should be chosen to be equal to the measured signal S_k . In reality this is usually not possible and the goal is to find the ideal signal which has the least cumulative error across all samples. It can also be noted that, because the range of this expression excludes negative numbers, equation 3.4 has range [0..1].

3.2.2 Ideal response of a linear, time-invariant system

The ideal output signal of a system is an output which is not corrupted by noise or finite resolution, it is only corrupted by the system response function. While there are maximum likelihood based signal recovery algorithms for *non-linear systems*, the scope of this research is limited to *linear, time-invariant systems* as this covers the majority of applications (Soliman and Srinath, 1998).

A system is linear if it adheres to the *superposition* principle; the response to a complex input can be described as the sum of responses to simpler inputs. For more information about linear systems please refer to chapter 2 of "Continuous and discrete signals and systems" by Soliman and Srinath (1998).

A system is time-invariant when the response does not depend on the time when it is calculated, a shift of the input signal results in an equally shifted and otherwise identical output signal (Soliman and Srinath, 1998). These two properties together result in a linear, time-invariant system, which has the property that it can be completely described by the *impulse response*. The impulse response is the response of a system to the *Dirac delta function*, a generalized function which is explained in chapter 1 of "Continuous and discrete signals and systems" (Soliman and Srinath, 1998).

The response function of a *linear* and *time-invariant* system with *impulse response* $H(t)$, to input signal $x(t)$, is shown in equation 3.8 (Soliman and Srinath, 1998). It is also known as the *convolution* of the input signal and the impulse response.

$$y(t) = \int_{-\infty}^{\infty} x(\tau)H(t - \tau) d\tau \quad (3.8)$$

For Maximum Likelihood, the impulse response of the system should be a known function.

Quantizing equation 3.8 using Euler the *discrete time* approximation can be obtained, which is shown in equation 3.9.

$$y(t) = \lim_{N \rightarrow \infty} \sum_{n=0}^N X(n\Delta_t)H(t - n\Delta_t)\Delta_t, \Delta_t = \frac{t}{N} \quad (3.9)$$

In this approximation there is an infinite amount of samples. Since that is not possible for a physical sensor the number of samples is finite, which reduces the accuracy of this approximation. Rewriting the equation for a finite number of samples results in:

$$Y(k) = \sum_{n=0}^k H((k - n), \Delta_t)X(n)\Delta_t \quad (3.10)$$

The attentive reader has already noticed that the discrete time impulse response now has two input arguments. An impulse response is something that only works in continuous time and in discrete time an approximation is used instead, which needs a notion of the time steps.

This equation can be written in matrix form if the response matrix R is defined.

$$R_{nk} = H'((k - n), \Delta_t)\Delta_t \quad (3.11)$$

$$Y_k(X_n) = \sum_n R_{nk}X_n \quad (3.12)$$

If the input and output signals are renamed, S_k^{ideal} is defined, as is shown in equations 3.13 through 3.15.

$$S_k^{ideal} = Y_k \quad (3.13)$$

$$I_n = X_n \quad (3.14)$$

$$S_k^{ideal}(I_n) = \sum_n R_{nk}I_n \quad (3.15)$$

3.2.3 Optimization problem

An optimization problem is when the best solution has to be found in a set of feasible solutions to a problem. In discrete time optimization problems the set of feasible solutions is countable but usually very large, making an exhaustive search not feasible. In section 3.2.4 the conjugate gradient method will be introduced to solve certain optimization problems. Before that, the maximum likelihood signal recovery problem has to be written in the general form of an optimization problem. This form, a system of linear equations, is shown in equation 3.16.

$$A\vec{x} - \vec{b} = 0 \quad (3.16)$$

Putting equations 3.7 and 3.12 together, equation 3.17 is obtained. This equation will be rewritten in the form of an optimization problem by expanding it and then defining the A matrix and \vec{b} vector, which can be seen in equations 3.18 through 3.21.

$$\Phi(I_n) = \sum_k \left(S_k - \left(\sum_n R_{nk} I_n \right) \right)^2 \quad (3.17)$$

$$\Phi(I_n) = \sum_k \left(S_k^2 - 2S_k \sum_n (R_{nk} I_n) + \sum_n (R_{nk} I_n) \sum_m (R_{mk} I_m) \right) \quad (3.18)$$

$$\Phi(I_n) = \sum_k (S_k^2) - \sum_k (2S_k \sum_n (R_{nk} I_n)) + \sum_k \left(\sum_n (R_{nk} I_n) \sum_m (R_{mk} I_m) \right) \quad (3.19)$$

$$\Phi(I_n) = \sum_k (S_k^2) - \sum_n (2I_n \sum_k (S_k R_{nk})) + \sum_m \sum_n I_n I_m \sum_k (R_{nk} R_{mk}) \quad (3.20)$$

In equation 3.21 the A matrix and \vec{b} vector are defined, to obtain a function in *quadratic form*. The term $(\sum_k S_k^2)$ from equation 3.20 can be left out because it is a constant scalar and the eventual goal is to minimize this function.

$$\Phi(I_n) = \cancel{\sum_k (S_k^2)} - \sum_n (2I_n \underbrace{\sum_k (S_k R_{nk})}_{\vec{b}}) + \sum_m \sum_n I_n I_m \underbrace{\sum_k (R_{nk} R_{mk})}_A \quad (3.21)$$

Which means the definitions of the A matrix and \vec{b} vector are:

$$A = R^T R \quad (3.22)$$

$$\vec{b} = R \vec{S} \quad (3.23)$$

$$\vec{i} = I_n \quad (3.24)$$

Rewriting the function in vector notation leads to equation 3.25

$$\Phi(\vec{i}) = -2\vec{i} \vec{b} + \vec{i}^T \vec{i} A \quad (3.25)$$

To find the input signal with maximum likelihood, this expression should be minimized. A minimum of a multi-variable function is a point where *the gradient* is zero. The gradient of a function is the vector containing all the *partial derivatives* of the function. By applying the gradient, equation 3.26 is obtained.

$$\nabla \Phi(\vec{i}) = \frac{1}{2} A^T \vec{i} + \frac{1}{2} A \vec{i} - \vec{b} \quad (3.26)$$

For a *positive-definite* and *symmetric* A matrix, this can be reduced to equation 3.27, which has the correct form as shown in equation 3.16.

$$A \vec{i} - \vec{b} = 0 \quad (3.27)$$

This means that, given some constraints on the A matrix, a maximum likelihood signal recovery problem can be reduced to a solvable system of linear equations. The equations can be solved using conjugate gradient (Fletcher and Powell, 1963).

Positive-definite is a hard to grasp property which is excellently explained in the MIT OpenCourseWare video lecture by Strang and Moler (2015). If the definition of the A matrix from equation 3.22 is recalled, the A matrix is at least positive semi-definite as by the proof of Dr. Strang. He also states that the matrix is positive-definite when the base matrix has linearly independent columns. This can be translated to the requirement that the Response matrix should have linearly independent columns.

3.2.4 Projected gradient

Projected gradient algorithms are iterative algorithms mostly used to solve optimization problems.

The global minimum of a multi-variable function, where the gradient is zero, equals the solution to a system of linear equations with a *square* and *positive-definite* A-matrix (Shewchuk, 1994). This quality means that if the local minimum is found the solution to the optimization problem is also found. Since it has already been established that the A-matrix is positive-definite and it is square by definition, a projected gradient method can be used to solve the signal recovery problem.

Projected gradient methods utilize the *gradient* of a function to find a *critical point* of that function. The gradient is a vector in the direction which has the steepest slope, it is composed of the directional derivatives of all free variables. A critical point is either a maximum, minimum or saddle point. Using a symmetric A-matrix, the minimum found can only be the global minimum, as proven by Shewchuk (1994).

The most intuitive projected gradient method is the *method of steepest descent* (Curry, 1944), which attempts to find the minimum of a multi-variable function by always going in the direction of steepest descent. The algorithm calculates the gradient and then takes a step in the direction opposite to the gradient, down the steepest slope. The size of the step is determined by a line search, which determines where the directional derivative in the direction of steepest descent is zero. The method has not seen much use because the computational complexity is high and it converges slowly.

The *method of conjugate gradients* is an improved projected gradient algorithm (Hestenes and Stiefel, 1952). The method of conjugate gradients converges just as fast or faster than the method of steepest descent (Gilbert and Nocedal, 1990), and has an upper bound on the amount of computations required to converge (Fletcher and Powell, 1963). In figure 3.4 the difference between conjugate gradient and steepest descent is visualised.

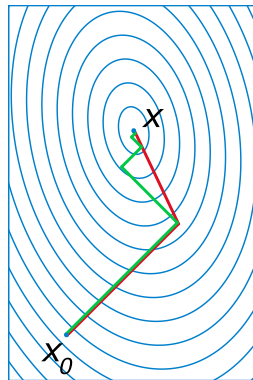


Figure 3.4: Comparison between Conjugate Gradient (Red) and Steepest Descent. The Red line converges much faster.

The first iteration of conjugate gradient is the same as with steepest descent, the algorithm starts by stepping in the direction of steepest descent. In the next iterations it chooses *search directions* which are *A-orthogonal*, or *conjugate*, to all earlier directions. *Conjugate directions* are vectors which behave similar to orthogonal vectors in the space for which they are defined. Because of the conjugacy, each step minimizes the error in a dimension of the system, until the global minimum is reached. Conjugate gradients, in contrast to steepest descent, does not 'travel' in the same direction twice. The nice properties of the method of conjugate gradients are caused by the search directions being conjugate. The name of the algorithm can be misleading, as the gradients are not conjugate and most of the search directions are not gradients.

The method of conjugate gradients can be used to solve systems of the form shown in equation 3.27. This means an A-matrix and b-vector are known at the start of the algorithm. The starting point $\vec{x}_{(0)}$ is an arbitrarily chosen vector. In later iterations, the *point* \vec{x} represents the best solution found so far.

In the first iteration conjugate gradient, just as steepest descent, takes a step in the direction of steepest descent. This direction, the search direction \vec{d} , is equal to the residual, \vec{r} , which is defined as the inverse of the gradient. All vector and matrix dimensions in the conjugate gradient algorithm have a dimension equal to the amount of samples.

$$\vec{d}_{(0)} = \vec{r}_{(0)} = \vec{b} - A\vec{x}_{(0)} \quad (3.28)$$

After which an iteration consists of calculating equations 3.29, 3.30, 3.31, 3.32 and 3.33. These equations will be explained below.

$$\alpha_{(i)} = \frac{\vec{r}_{(i)}^T \vec{r}_{(i)}}{\vec{d}_{(i)}^T A \vec{d}_{(i)}} \quad (3.29)$$

$$\vec{r}_{(i+1)} = \vec{r}_{(i)} - \alpha_{(i)} A \vec{d}_{(i)} \quad (3.30)$$

$$\vec{d}_{(i+1)} = \vec{r}_{(i+1)} + \beta_{(i+1)} \vec{d}_{(i)} \quad (3.31)$$

$$\beta_{(i+1)} = \frac{\vec{r}_{(i+1)}^T \vec{r}_{(i+1)}}{\vec{r}_{(i)}^T \vec{r}_{(i)}} \quad (3.32)$$

$$\vec{x}_{(i+1)} = \vec{x}_{(i)} + \alpha_{(i)} \vec{d}_{(i)} \quad (3.33)$$

The calculation of step size α is shown in equation 3.29. The calculation consists of dividing the magnitude of the residual by a scaled version of the magnitude of the search direction. The step size is a non-negative number which is used in determining the next point and next residual.

After iteration zero, the search direction is calculated using the conjugate Gram-Schmidt process. Since in iteration zero the gradient is used, the new search directions will be conjugate to the first gradient as well as all other earlier search directions. The conjugate Gram-Schmidt works by creating a residual which is orthogonal to all earlier residuals, and construction the search direction from is. Equation 3.30 shows the construction of an orthogonal residual, equation 3.31 is the definition of the new conjugate search direction and in equation 3.32, the *optimization parameter* or *Gram-Schmidt constant* β is calculated.

The next point $\vec{x}_{(i+1)}$ can be calculated by taking a step with size α along the search direction \vec{d} from the last point, as is shown in equation 3.33.

These steps conclude one iteration of the theoretical conjugate gradient algorithm (Shewchuk, 1994).

3.2.5 Conjugate gradient converges

Conjugate gradient theoretically converges in just as much iterations as there are independent variables (Gilbert and Nocedal, 1990). This useful property is caused by the conjugacy of the directions. Since each search direction is linearly independent, n conjugate directions form a basis of an n dimensional subspace. This means that in n iterations, each direction is traversed exactly once.

If we recall that the A-matrix is positive definite and combine that with the energy definition of a positive definite matrix (Johnson, 1970), which is shown in equation 3.34, it is proven that the denominator of equation 3.29 is always a positive number. Since the magnitude of a vector, the

nominator in the equation, is non-negative that means that α is non-negative. This means that the conjugate gradient algorithm always takes a step towards the minimum in the direction traversed.

$$x^T Ax > 0 \text{ for } x \neq 0 \quad (3.34)$$

Since each possible direction is travelled towards the minimum, it can be concluded that conjugate gradient traverses to a minimum. Using a symmetric A-matrix, the minimum found can only be the global minimum, as proven by Shewchuk (1994).

In the previous chapters it is explained that the global minimum of the optimization problem resulting from the maximum likelihood principle corresponds with the input signal which causes the least cumulative error. Any step away from the global minimum will increase the error and decrease the accuracy of the results.

3.2.6 Computer implementations

In mathematics, infinite precision is assumed in calculations. With infinite precision, it is guaranteed that after n iterations the global minimum is reached. In computer hardware this is not the case, and the finite precision deteriorates the results.

The big problem with conjugating the search directions is that each iteration is based solely on the previous iterations, which causes errors to accumulate. The solution is to combine conjugate gradient with the method of steepest descent, and every \sqrt{n} iterations reset the residual to the inverse of the gradient. This causes the algorithm to be more precise than conjugate gradient and faster than steepest descent, but loses the guarantee to converge within n iterations.

Since the result can never get better than the maximum precision of the hardware, it also does not make sense to continue iterating when near the minimum. When near the minimum, the step size δ becomes very low. In computer implementations an early exit mechanism which stops iterating once δ reaches below a certain threshold is desirable.

4 Computer architecture background

4.1 FPGA

In the past 50 years, computer processors have been growing in size and capabilities. Processor designers have found ingenious ways to increase the frequency and instructions per clock in order to do more calculations per second. The amount and complexity of hardware that can fit in a chip depends on the transistor budget, which has been growing according to *Moore's law* (Mack, 2011). In the last decade however, limits on clock frequency, instructions per clock and transistor budget have been reached (Kumar, 2015). The next way of increasing processing power is to use parallel hardware, such as multi-core processors and graphics programming units (GPU), which can be more time and power efficient for most computational loads. Regardless of implementation, general-purpose processors are a source of inefficiency (Hameed et al., 2010)

An alternative is to use *fixed-function* or *reconfigurable* hardware. These types of hardware can be used to create dedicated architecture for the intended application, which can be more time and power efficient. *Field Programmable Gate Arrays* (FPGA) are reconfigurable chips which can be programmed using a *Hardware Description Language* (HDL). An FPGA consists of a large amount of *programmable logical blocks* which can be wired together to form a digital circuit. The advantage over fixed-function hardware is that the architecture is not fixed, the FPGA can be reconfigured to function as a different digital circuit. In addition to that most FPGA's currently available also contain DSP blocks, embedded processor cores and/or peripherals.

On a processor core, the amount of resources is fixed, and the software consists of a sequence of instructions which execute sequentially. On an FPGA the resources can be configured, and functions can either execute parallel, pipelined or sequential. Because no transistors are reserved for instructions which are not used, an FPGA architecture usually uses less energy for the same computations. An FPGA architecture can be considered as a trade-off between the use of time and area.

4.2 CLaSH

FPGA's are mostly programmed using a low-level hardware description language (HDL) such as VHDL and Verilog. These languages differ from traditional software programming languages in that they include an explicit notation of time. In an HDL *functionality* is assigned to *chip area*, and *memory assignments* are done at a *clock edge*. This corresponds with how hardware works, which means a design written with a HDL can be translated to digital circuit.

Since a design written using an HDL is intrinsically parallel and needs to be timed manually, many software developers used to writing sequential programs struggle with HDL's. A lot of research is done on high-level HDL's to design hardware more easily.

There are high-level hardware design languages, such as SystemC, which are based on traditional imperative programming languages. The problem with these high-level hardware design languages is that they are intrinsically designed for a sequential order of execution, while the distinction between time and area is very important in designing an efficient FPGA architecture. Another drawback is that the translation from an imperative language to VHDL is not straightforward and can lead to inefficient designs.

The emphasis on the order in which statements are being executed in programming is called *control flow*, whereas FPGA's are better suited to *data flow* applications. In data flow there is a fixed set of functions and the flexibility comes from the routing of the data.

CLaSH is a set of compilation tools and language extensions for the functional programming language Haskell to enable hardware description using the Haskell syntax (Baaij, 2009). Clash is developed by the Computer Architecture for Embedded Systems group at the University of Twente. As CLaSH is based on the Haskell syntax, it is a functional programming language. A functional programming language sacrifices describing the sequential order of execution to describe software as a collection of functions. Both mathematics and hardware can also be described as a collection of functions, which makes functional programming an ideal method to program a mathematical algorithm in hardware.

4.2.1 Mathematics in Haskell/CLaSH

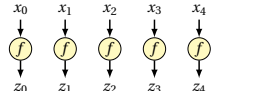
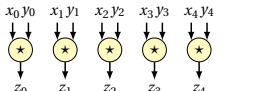
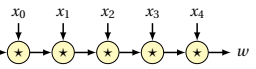
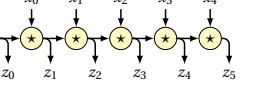
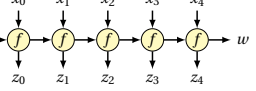
Haskell is a pure language, which means that calling a function has no side-effects. Every result from executing the function is captured in its return argument. Mathematical algorithms are also pure, a function cannot affect anything besides its output value. Languages like C are not pure, every function can change the state, for example by changing a global variable. Pure programs and functions are sometimes called stateless. Purity is an important concept in mathematical algorithms and proofs.

The next concept that makes Haskell nice for mathematics is the addition of higher-order functions. Higher order functions can have functions as an argument or return value. A higher-order function could, for example, scale a function with a constant. Higher order functions are important to mathematicians, as it allows them to describe derivatives, integrals, gradients and other functions which operate on functions.

4.2.2 CLaSH for FPGA design

CLaSH uses higher-order functions and full type inference to describe fully parametric hardware structures in a very concise manner (Gerards et al., 2011). Because of this CLaSH maps very well to parallel hardware such as an FPGA. Programs written using CLaSH transform directly into VHDL or Verilog, which can be synthesized to hardware.

For FPGA design higher order functions are also very important. Higher order functions can be used to describe structure. In this way large parallel structures can be created using simple keywords. Examples of higher order functions and the structure they describe are listed below.

<i>map</i>		$f x \Rightarrow z$	$zs = \widehat{f} xs$	$zs = map f xs$
<i>zipWith</i>		$x \star y \Rightarrow z$	$zs = xs \widehat{\star} ys$	$zs = zipWith (\star) xs ys$
<i>foldl</i>		$a \star x \Rightarrow a'$	$w = a \widehat{\star} xs$	$w = foldl (\star) a xs$
<i>scanl</i>		$a \star x \Rightarrow a'$ $z = a$	$w = a \overline{\widehat{\star}} xs$	$zs = scanl (\star) a xs$
<i>mapAccumL</i>		$f a x \Rightarrow (a', z)$?	$(w, zs) = mapAccumL f a xs$

Independent parallel operations can be done using *map* and *zipWith*, which execute a one or two argument function for all elements of a list in parallel. Another higher-order function

important for hardware design is the *fold* function, which represents a *reducing* operation such as a summation.

Another useful property for hardware design is *full type inference*. Full type inference means that the type of a functions return argument can be described as a function of the input argument type. This means that the output type can be *inferred* from the input type, not only for a single function but also for a composition of functions. In hardware this property is used to determine the bit-width of interconnects, the signs of operands and is used by the compiler to determine whether the descriptions are physically possible. It also stops seemingly compatible types such as floating point and fixed-point numbers from being mixed without manually specifying a conversion step.

4.2.3 Matrix vector multiplication in CLaSH

An example of how CLaSH is used to generate hardware for mathematics is the implementation of the Matrix vector multiplication. A mathematical matrix vector multiplication is shown in equation 4.1.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{pmatrix} \quad (4.1)$$

The highlighted part corresponds to a single output value. The corresponding algebraic equation is included as equation 4.2.

$$\forall_i v_i = \sum_j^n a_{i,j} * d_j \quad (4.2)$$

In CLaSH this can be modelled using the following code snippet:

```
1 mvMult a d = map (v) a
2   where
3     v an = foldl1 (+) $ zipWith (*) an d
```

Where the first line specifies the input types, the "map" operation corresponds with \forall_i , the "zipWith (*)" adds the multipliers and "foldl1 (+)" results in a summation (Σ). This small piece of code corresponds to the hardware architecture shown in figure 4.1.

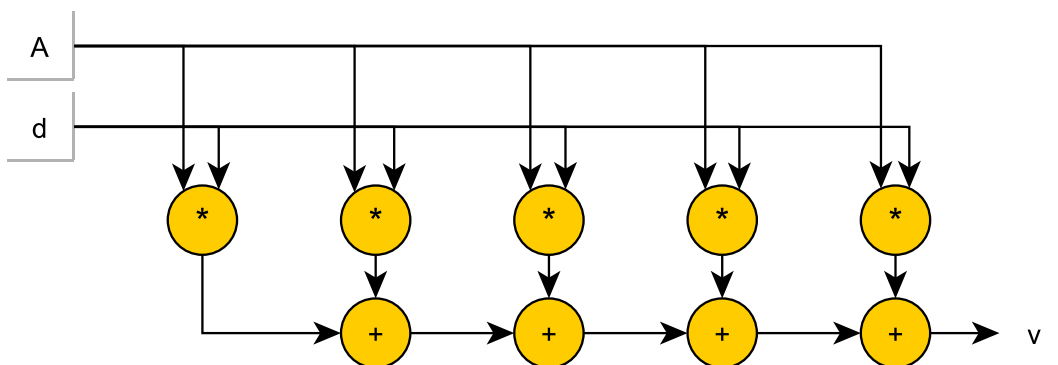


Figure 4.1: Multiplication of multi-dimensional arguments using ZipWith and Fold

Although the description is very short, the resulting hardware architecture has a lot of nice properties. The architecture is described for any amount of input arguments, whether the matrix is 10×20 or 100000×2 , and can be used for any numeric type.

5 Design space exploration

For the signal recovery a combination of algorithm, platform and implementation needs to be chosen which is energy efficient and fast.

5.1 Optimization algorithm

- Projected gradient (Steepest decent, conjugate gradient)
- Newtons method
- Simulated annealing

5.1.1 Steepest descent

Steepest descent is a simple but effective method to solve an optimization problem. The problem with steepest descent is that there is no limit to the amount of iterations required to converge. Conjugate gradient is also a projected gradient algorithm, but is proven to converge towards the optimal solution in a limited amount of steps. A downside of Conjugate Gradient is that a large amount of iterations is required and these iterations are computationally complex. Conjugate Gradient is not very robust against floating point round-off errors.

5.1.2 Newtons method

Newtons method uses Hessians instead of gradients in order to converge in fewer iterations than Conjugate Gradient. The downside is that the iterations in Newtons method are more complex. For problems with a large amount of samples Newtons method is faster. To program on an FPGA Newtons method will probably require more logical cells than Conjugate gradient.

5.1.3 Simulated annealing

Simulated annealing is not guaranteed to converge to the global optimum in a fixed amount of computations. The end result is an approximation.

5.1.4 Algorithm choice

Conjugate gradient has been chosen as optimization algorithm because it is guaranteed to find the optimal solution in a limited amount of iterations. The iterations of Conjugate gradient are less computationally complex than Newton's method, which makes it easier to implement on an FPGA. Simulated annealing is usually faster, but it is never certain if a feasible solution is reached.

5.2 Platform

Because the algorithms are quite complex, a powerful computational platform has to be chosen in order to run the algorithm in a reasonable amount of time for a decent amount of samples. The requirement for a powerful platform is almost contrary to the requirement that the platform should not use a lot of energy.

The definition of portable is vague, so the goal is to minimize energy consumption while still doing the required computations per second. The platform should be able to process 2000 samples within one second.

At 2000 samples, the algorithm requires at most 2000 iterations with at least one matrix vector product per iteration. There are 45 iterations where the residual has to be calculated another time. The computational complexity of a matrix vector product is $2n^2$ when additions and multiplications are counted separately. The complexity of the other arithmetic operations

combined is approximately $10n^2$. One execution of the algorithm can result in over 16 Giga-computations to be done in one second.

$$(n + \sqrt{n} + 5) * 2n^2 = (2000 + 45 + 5) * 2000^2 = 16,5 * 10^9$$

- Field-programmable gate array
- ARM Processor
- Digital signal processor
- Graphics processor

5.2.1 Field-programmable gate array

A field-programmable gate array (FPGA) is an integrated circuit consisting of programmable logic blocks which can be configured using a hardware description language (HDL). Modern FPGA's contain DSP blocks, memory blocks and sometimes even complete processors in addition to standard logic cells, and can be configured to do massively parallel computations.

The advantage of using a FPGA is that the architecture can be highly efficient, because it is wired to compute the algorithm provided and do nothing else. With the amount of parallelism available on FPGA's the computations can also be done quickly without requiring a high clock frequency, which would increase the power draw. FPGA's are also very scalable, with a peak processing performance ranging from a few million FLOPS (Floating-point operations per second) to several trillion for the fastest FPGA's (Vishwanath, 2016).

The disadvantage of a FPGA is that it is considered hard to program for, HDL developers are therefore scarce and expensive. Another disadvantage is that powerful FPGA's are very expensive.

5.2.2 ARM Processor

The last decade the amount of powerful ARM processors available has increased significantly, benefiting from rising sales of tablets, smart tv's, smart-phones and other smart devices. Modern ARM multi-cores are able to reach multiple GFLOPS (Giga-FLOPS) while retaining a manageable power envelope (Reed et al., 2015).

ARM processors have as advantage that they are easy to program, and developers are relatively easy to find. ARM processors are mass-produced and therefore very cheap.

Although recent developments have seen processing power increased significantly in ARM processors, they are still the slowest and least scalable option in this comparison. ARM processors also are not very efficient, as they use a general-purpose architecture which is not optimized for computations.

5.2.3 Digital signal processor

Digital signal processors (DSP) are general-purpose processors which have been optimized to do calculations. When compared to ARM processors they usually have more complex arithmetic operations, a deeper pipeline stage and a memory architecture better suited for batch processing. The throughput of a DSP is between that of an ARM processor and a FPGA or GPU (Texas Instruments Incorporated., 2017).

As a more specific product, DSP's are more expensive than ARM processors. Like ARM processors, DSP's can be programmed in C and C++ making programming easier than for a FPGA. To effectively use the processing power of the DSP complex multi-threaded programs are required, making them more difficult than normal processors.

Type	Max (GFLOPS)	Typical (GFLOPS)	GFLOPS/Watt	Price
FPGA	>10000	100	40	Very expensive
ARM	8	4	0.8	Cheap
DSP	200	20	10	Expensive
GPU	>10000	2000	20	Cheap

Table 5.1: Platform characteristics.

5.2.4 Graphics processor

Graphic processing units (GPU) are massively parallel processors optimized to do image processing. Using a large memory bandwidth, a lot of threads and vector arithmetic instructions GPU's are very fast and quite efficient (NVIDIA Corporation, 2016).

The efficiency of GPU's decreases drastically for non-parallel workloads. Gpu computation libraries such as Cuda and OpenCL are difficult to grasp. Programs using GPU's for general purpose computations almost never reach the efficiency and power a GPU can reach. Dedicated GPU's require a separate CPU and don't fit the power budget. The GPU's integrated into embedded processors are a lot less powerful and usually not that well supported for computations.

5.2.5 Platform choice

In table 5.1 several characteristics of the platforms have been listed.

FPGA's have been chosen as the development platform, as they are the best solution in terms of computational efficiency. Ease of programming, traditionally a downside of using an FPGA, is solved by choosing a high level language, as will be explained in the next section. Scarcity of developers as a downside of using an FPGA is not relevant in the context of education.

The University of Twente has provided a Terasic SoCKit development board with an Altera Cyclone 5 FPGA for testing. Since the cyclone 5 is already very old and does not meet the requirements, theoretical comparisons will be made with more modern Cyclone 10 and Aria 10 FPGA's.

5.3 Development language

The development language is of importance for the ease of programming and maintaining the code. Programming language can also have a large effect on efficiency and speed.

- Low level hardware description language
- Haskell/CLaSH
- SystemC
- Matlab/Simulink

5.3.1 Low level hardware description language

The leading hardware description languages in the industry are VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) and Verilog. The languages are similar in abstraction level and capabilities, but differ in syntax (Sandstrom, 1995).

VHDL and Verilog can be used to program time-, energy- and space- efficient hardware designs, but are very hard to program for. For programming large mathematical algorithms these low-level hardware description languages are ill suited, since the interconnections and timing all have to be done manually.

5.3.2 CLaSH

CLaSH is a set of compilation tools and language extensions for the functional programming language Haskell to enable hardware description using the Haskell syntax. Programs written using CLaSH transform directly into VHDL or Verilog (Baaij, 2009).

The advantage of CLaSH is that its high-level descriptions generally speaking require less programming for the same amount of hardware when compared to VHDL or Verilog. CLaSH uses higher-order functions and full type inference to describe fully parametric hardware structures in a very concise manner (Gerards et al., 2011).

As CLaSH is based on the Haskell syntax, it is a functional programming language. The advantage of this is that CLaSH maps very well to parallel hardware such as an FPGA. The disadvantage of using a functional language is that it does not map very well to a processor and is therefore slow to simulate. The other disadvantage is that functional programmers are scarce because of the dominance of imperative programming languages.

CLaSH is better suited for mathematical algorithms, but does not synthesize directly. CLaSH as a higher-level alternative abstracts away the registers, interconnects and clock generation, and creating complex designs is easy with CLaSH. CLaSH translates to VHDL or Verilog for synthesis, but the translation is significantly more straightforward than it is with SystemC.

5.3.3 SystemC

SystemC is a C++ class library which enables high-level synthesis and system-level modelling. SystemC can be used as hardware description language. SystemC does not synthesize directly but, like CLaSH, translates to an intermediate HDL. The translation from SystemC to VHDL is not straightforward and can lead to inefficient designs.

As SystemC is based on C++ it is easy to learn for most programmers. When compared with VHDL and Verilog, SystemC uses a higher level of abstraction to simplify programming. SystemC is well suited for simulation on a traditional computer. Because SystemC is intrinsically not-parallel, complex dependency analysis is required to generate parallel code (Nayak et al., 2001). Some inefficiency is added in translating SystemC to a synthesizable hardware description language and not all parts of SystemC are synthesizable.

5.3.4 MATLAB Simulink

MATLAB is a suite of applications to program, simulate or evaluate mathematical expressions and algorithms. Simulink is a graphical environment for the creation and simulation of models and algorithms. MATLAB contains "HDL Coder", a compiler which is able to generate VHDL or Verilog code from MATLAB programs and Simulink models.

MATLAB is close to mathematics and easy to use. MATLAB natively supports matrix and vector operations and a lot of the operators have parallel implementations, which makes it well suited for FPGA's and GPU's (Nayak et al., 2001).

Automatic analysis of fixed point parameters is not always efficient (Banerjee et al., 2003). Efficiency of a Matlab program is very dependent on how Matlab is being used, built-in Matlab functions often have efficient implementations, whereas hand-written code usually loses efficiency in translation.

5.3.5 Algorithm choice

CLaSH has been chosen as the programming language for the project, as it is well-suited for both mathematics and hardware description. The low level HDLs require too much effort to get the mathematics working. SystemC and Matlab try to abstract away from hardware, which makes it hard to create efficient architectures.

5.4 Architecture

Since a naive implementation of the algorithm does not fit on a modern FPGA, an architecture has to be designed which reuses hardware in a smart way. For the architecture multiple topologies are available, which can be used as a design guideline. Since matrix-vector products take up most of the computations, it makes sense to create an architecture which can do matrix-vector calculations as fast as possible.

The choices are:

- Processor with vector ALU
- Multicore processor
- VLIW processor
- Streaming architecture

5.4.1 Processor with vector ALU

One type of architecture is to create a simple processor with a small instruction set and then tailor it specifically for the algorithm. An example of such a system is the vector ALU processor with dual blockram created by Appel and Folmer (2016), which achieves a factor 3 performance-per-joule increase.

A processor is a machine which is able to process data using a predetermined set of instructions. A program is a list of instructions which should be executed sequentially. A processor consists of memory, an arithmetic logic unit (ALU) and a control unit.

The instruction set should be designed in such a way that the hardware is utilized to its fullest potential. Several techniques can be used to improve the performance of the architecture, such as pipelining, parallel processing and unfolding. One of the more obvious improvements is a parallel arithmetic unit, given that most FPGAs have many dedicated DSP blocks which should be used in parallel.

The processor like structure closely resembles a GPU, which reduces the chances of increasing efficiency over a GPU-based solution.

5.4.2 VLIW processor

A Very Large Instruction Word (VLIW) processor uses a complex large instruction which contains a lot of instructions in parallel. A VLIW processor aims to use the available hardware more efficiently by increasing the amount of work that can be specified by one instruction.

A processor can be split into several parts. There is a part for memory operations, control operations and there are one-or-more parts for arithmetic operations. In a standard processor each instruction uses one part of the processor to do one simple task. A VLIW instruction instructs multiple processor parts at the same time, for example loading the next input value while computing the current one. This principle, called *instruction level parallelism*, improves performance significantly at the cost of program complexity.

5.4.3 Stepwise 'static' algorithm architecture

Just as the algorithm can be split into iterations, which can be split into equations, equations can be split into steps and sub-steps. A type of architecture models these steps and substeps into a large state-machine and steps through them statically.

Compared with the vector processor or VLIW processor, a static architecture is not that much different. It sacrifices a great deal of accuracy in order to gain more speed. Some things are a lot harder to model without control instructions like Branch and Jump.

5.4.4 Streaming architecture

Instead of splitting the problem into several steps to reduce the computational complexity, it can also be reduced by looking at individual samples. Looking at one sample instead of n samples can give a factor n reduction in computational complexity. In a streaming architecture samples are fed into the architecture one-at-a-time, and the parallelism comes from *pipelining*. Pipelining is the process of adding memory elements so multiple otherwise sequential steps are simultaneously done for several samples.

A streaming architecture requires less control logic, especially for the memory. The amount of arithmetic operations per clock cycle can be really high once the pipeline is *filled*. If the implementation is good, the clock speed can also be very high.

Not all algorithms are well suited for streaming architectures. For a streaming architecture to work, the calculation for one samples needs to rely solely on its data and a state obtained from previous samples. If future samples are required to process the current sample, a static architecture can't be done. The amount of logic required to process one sample can be more than there is available, which would require a sub-architecture in the architecture.

5.4.5 Architecture choice

A processor-like architecture with a large vector processing unit will be made. The VPU is scalable, flexible and it is easier to implement the 'off' cases (each 40 iterations do the residual calculation different). The CPU will be improved with characteristics from the other architecture possibilities.

A VLIW processor would require a complex program. The normal processor architecture has been chosen because its instruction set can resemble normal assembly.

The static algorithm is probably highly efficient and somewhat scalable if done well, but hard to program. The instruction set from the architecture will be kept small, as to resemble the static architecture machine more.

The streaming architecture, if possible, is a very efficient architecture. Due to different sizes (scalar, vector, matrix) and the different complexities of the corresponding mathematical equations, a streaming architecture is very hard to implement. The matrix-vector calculations require information from future samples, which means a streaming architecture can not be done. From the streaming architecture the idea of auto-incrementing memory addresses is taken, as to increase the amount of arithmetic instructions done per cycle.

5.5 Memory

Feeding a parallel computation structure with data requires memory with high bandwidth and enough storage to store all the vectors and matrices. In addition it would be nice to have a predictable low latency memory, as it simplifies the control structure.

The amount of memory required is defined by the sample size n and the data width m with the relation $m(n^2 + 4n + 2)$.

Since the multipliers on the platform chosen are 27 bits wide (Altera Corporation, 2011), the data width will be 27 or lower. Multiplier pairing, where multiple DSP blocks are combined to achieve a larger data width, is not being considered.

Interesting memory technologies are:

- DDR3 SDRAM
- SRAM
- BlockRam
- QSPI Flash

The technologies are explained below.

5.5.1 DDR3 SDRAM

Double Data Rate Type Three (DDR3) SDRAM is a high-throughput and high-density memory technology which is widely used in computers, phones and other electric devices. The SocKit evaluation platform has 1GB of DDR3 memory available for the FPGA, which corresponds with a sample size of over 16000 samples. The RAM can be controlled in various ways, ranging from low-level electrical control of all the signals to ready-made IP blocks.

The advantage of DDR3 RAM is its large size and decent throughput. The complexity of the control structure is the largest downside of using DDR3 RAM. The control structure is complex because the latency is high and not static. The memory is organized into rows and columns, where switching rows is slower than switching columns. The not-constant memory latency of up to 14 memory clock cycles requires extensive planning in the software or architecture.

5.5.2 SRAM

Static RAM is a more expensive type of RAM which has a lower and more predictable latency, while retaining the high throughput of DDR type memory. A lot of newer FPGA's are equipped with multiple megabytes of SRAM, but the platform chosen for this thesis does not support it. A possible improvement in the future might be the selection of a platform which supports SRAM.

5.5.3 BlockRam

Every FPGA comes with internal memory block structures commonly referred to as BlockRam. BlockRam is essentially on-chip SRAM with a 1 cycle latency and a very wide bus. The performance of BlockRam is better than the other technologies and the control structure is simpler. The downside of using BlockRam is that it has a limited size.

The FPGA used contains a little over 5MB of BlockRam, resulting in a maximum sample size of 1200.

5.5.4 QSPI Flash

QSPI Flash is a simple to use, predictable memory technology which is non-volatile.

5.5.5 Memory choice

Disregarding complexity, the best solution is a combination of DDR3 memory for storage and BlockRam as a buffer, so that large matrices can be used and the datapath never has to wait for memory reads. Unfortunately such a memory controller would take a significant amount of programming effort while not being the main focus of the project. It is therefore a better choice to use BlockRam, and leave a more advanced memory controller for future improvements.

6 Implementation

This chapter aims to describe the various implementations made during the research, and the research results that have been obtained from them.

6.1 Software implementation of algorithm

In order to effectively test the algorithm and determining the hardware requirements, several software implementations have been made before attempting hardware design. The algorithm, as described in section 3.2.1, consists of several parts. These parts are consistent in all the software implementations and will be explained below.

- Calculation of the A matrix and b-vector
- Iteration zero
- Iterate

At the start it is assumed that the set of samples and the response function are known. From this point the response matrix can be calculated and from it the A-matrix and b-vector.

The conjugate gradient algorithm requires an A-matrix and b-vector, which are both constructed from the response matrix. The first step is therefore calculating the response matrix. The response matrix contains a calculated value of the response function for every sample at every time unit. It is a square matrix with a size equalling the amount of samples.

The A-matrix is, by the definition in section 3.2.1, the product of the transposed response matrix and the normal response matrix. This means calculating the A-matrix has a complexity of n^3 , where n is the sample size. Fortunately the response of the machine does not change much over time, so the A-matrix does not need to be calculated for each running of the algorithm.

The b vector is, by the definition in section 3.2.1, constructed from the input samples and the response matrix. An implication of this is that all the samples must be present before the b-vector can be calculated.

6.1.1 Haskell implementation of algorithm

As Haskell is a more abstract and powerful superset of CLaSH, it is better suited for writing and testing a mathematical algorithm. Therefore the first versions of the algorithm have been written in Haskell. Since the Haskell implementation runs on a CPU it is comparatively slow.

The main body of the algorithm is the iteration function, which has been included below. The source code can be found in appendix A.

```

1 --Iteration function. Calculate the values for one iteration and then make a
  recursive call to calculate the next iteration.
2 --The recursion ends at maxN samples or when  $\delta$  get's too low.
3 --Note: ' denotes "next value of" or (i+1)
4 iteration (x, r, d,  $\delta$ ,  $\alpha$ ,  $\beta$ , i, maxN, min $\delta$ )
5   | i >= maxN
6   |  $\delta!$ (0,0) < min $\delta$  = (x', r', d',  $\delta'$ ,  $\alpha'$ ,  $\beta'$ , i', maxN, min $\delta$ )
7   | otherwise = iteration (x', r', d',  $\delta'$ ,  $\alpha'$ ,  $\beta'$ , i', maxN, min $\delta$ )
8   where
9     x' = x /+ (d // *  $\alpha$ )
10    r' | (i `mod` 40 == 0) = b /- / a_mn // * / x'
11    | otherwise = r /- / a_mn // *  $\alpha$  // * / d
12    d' = r' /+ / d // *  $\beta'$ 
13     $\delta'$  = transp r' // * / r'
14     $\alpha'$  =  $\delta'$  /// (transp d' // * / a_mn // * / d')
15     $\beta'$  =  $\delta'$  ///  $\delta$ 
16    i' = i+1

```

The main advantages of using Haskell are immediately clear when comparing the code with the mathematical formulas, the structure is identical to the mathematics explained in section 3.2.4.

The upper lines of code are the function definition with its arguments and the output definitions corresponding to the condition in the *guards*. The guards check whether the end condition has been reached as to stop calculating. If the end conditions have not been reached the function does a recursive call to calculate the next iteration using the output values calculated in the *where clause*.

The where clause contains the mathematics. In the formulas, custom infix operators are used for scalar, matrix and vector operations. The infix operators can be read without the edge slashes as just a mathematical operator (e.g. + for addition or / for division), but the slashes also serve a purpose. A single slash denotes an array with a single dimension such as a column vector or a scalar, while a double slash operates on double dimensioned arrays such as row vectors or matrices.

Testing

The algorithm has been verified using a testbench also written in Haskell. The testbench generates an input signal and adds Gaussian noise using a random number generator and a distribution function.

The testbench uses a simple RC network configured as a low-pass filter. The simulated resistance is 8200Ω and the capacitance $1\mu F$. Gaussian noise with standard deviation 0.1 has been added, corresponding to about 25% noise. For a sample size of 200 the input signal and recovered signals have been plotted and are shown in figure 6.1.

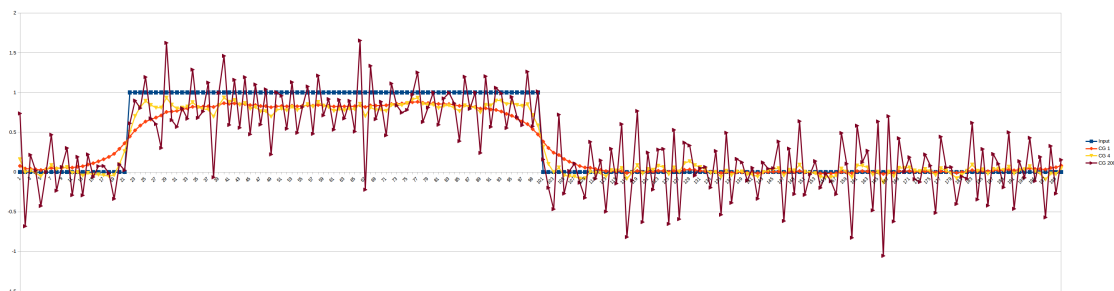


Figure 6.1: Input (Blue) and recovered signals

The orange line is the signal after one iteration, it shows a small resemblance to the input signal given in blue. After only five iterations the yellow signal is obtained, which resembles the input well enough to use the early escape. The dark red signal is what happens after iterating far beyond the early escape, a signal is obtained which is severely corrupted by accumulated errors, as predicted in section 3.2.6.

6.1.2 CLaSH implementation of Haskell algorithm

The Haskell algorithm is not directly compatible with the CLaSH compiler. The algorithm had to be rewritten using the CLaSH prelude library. Using CLaSH meant several concepts, such as lists with unknown size or recursion with unknown depth, are not possible since they cannot be compiled.

The first part of writing CLaSH code from Haskell was defining the mathematical operators used for Vectors instead of lists. The module "matrixMath.hs" contains definitions for matrix vector products, dot products and vector scaling.

The second part was converting the maximum likelihood program to use Vectors and the new functions. The CLaSH version of the iteration function is included below. The full program can be found in appendix B.

```

1 iteration :: (Vect, Vect, Vect, Number, Number, Unsigned 27, Vect)
2   -> ()
3   -> ((Vect, Vect, Vect, Number, Number, Unsigned 27, Vect), Maybe (Vect))
4 iteration (x_n, r_n, d_n, r_n2, alpha_n, n, b_m) _
5 | stop      = ((x_n, r_n, d_n, r_n2, alpha_n, n, b_m), Just x_n)
6 | otherwise = ((x_n', r_n', d_n', r_n2', alpha_n', n', b_m), Nothing)
7   where
8     x_n' = zipWith (+) x_n (map (* alpha_n) d_n)
9     r_n' = zipWith (-) r_n (mvMult a_mn (map (* alpha_n) d_n))
10    d_n' = zipWith (+) r_n' (map (* beta_n') d_n)
11    r_n2' = rcMult r_n' r_n'
12    alpha_n' = r_n2' / (rcMult d_n' (mvMult a_mn d_n'))
13    beta_n' | r_n2 == 0 = 0
14             | otherwise = r_n2' / r_n2
15    n' = n+1
16    stop = (n >= iterations) || (r_n2' == 0)

```

As it is unknown at compilation time how many iterations are required, the iteration function contains recursion with unknown depth. To remove the recursion from the iteration formula, the *mealy* instruction is used. The mealy instruction tells the compiler a mealy machine can be used for a specific part of the design, in this case the iteration function. A mealy machine is a machine with an input and a state, which produces output and a next state. In the case of the iteration it takes the current state, values like x , r and α , and produces an output and next state. The next state is then fed back to the same mealy machine at the next clock cycle. This solves the recursion problem, and also has other benefits which are explained in section 6.2.

Testing

The naive CLaSH implementation has been compared to the Haskell implementation and gives similar results. Since the CLaSH simulation takes more time, comparisons are done with smaller sample sizes. A comparison between the Haskell results and CLaSH results can be found in figure 6.2. The differences can be attributed to fixed-point round-off versus floating point round-off. These differences can be resolved using different fixed-point word sizes or by switching to floating points.

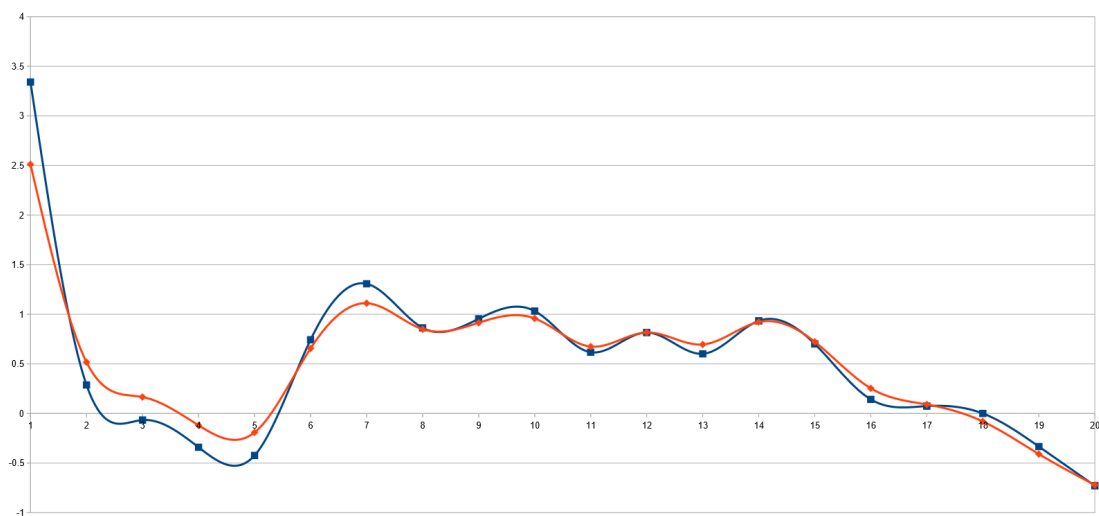


Figure 6.2: Haskell (Blue) and CLaSH (Red) recovered signals

Synthesizing

The architecture is synthesizable, but requires too much resources to fit inside a FPGA. The largest design that can be fit onto the Cyclone V FPGA provided can be calculated.

Considering the off-case where the residual is re-calculated, an iteration consists of at most two matrix-vector multiplications, two dot products, two vector scalar multiplications, two vector additions, 1 vector subtraction and 2 scalar divisions. If each scalar arithmetic operation can be done in one clock cycle using one DSP block this results in a complexity of $2n^2 + 7n + 2$. This means that, using the naive algorithm, an FPGA with 112 DPS blocks can do one iteration in one cycle for a sample size of 5, from equation 6.1

$$2n^2 + 7n + 2 = 112 \Rightarrow n = \frac{\sqrt{929} - 7}{4} \quad (6.1)$$

6.1.3 Dividing the algorithm

The first division across time has already been done in the naive algorithm. Instead of a *recursive call*, which is problematic for the CLaSH compiler, a mealy machine was used to divide the algorithm in several iterations. A recursive call would have resulted in a long chain of large complex hardware blocks, which each compute a single iteration. Using a mealy machine architecture results in a factor n increase in clock cycles used but also a factor n decrease in hardware resources. Another upside to using a mealy machine is that the early stop condition can be modelled much more easily.

Despite the implementation of a mealy machine for the iterations, the complexity of the naive implementation is still too much. To further split the algorithm across time, the iterations need to be split into several calculations which are done sequentially. A process graph has been made to calculate the complexity, find possible time domains and determine memory requirements. The process graph is included in figure 6.3.

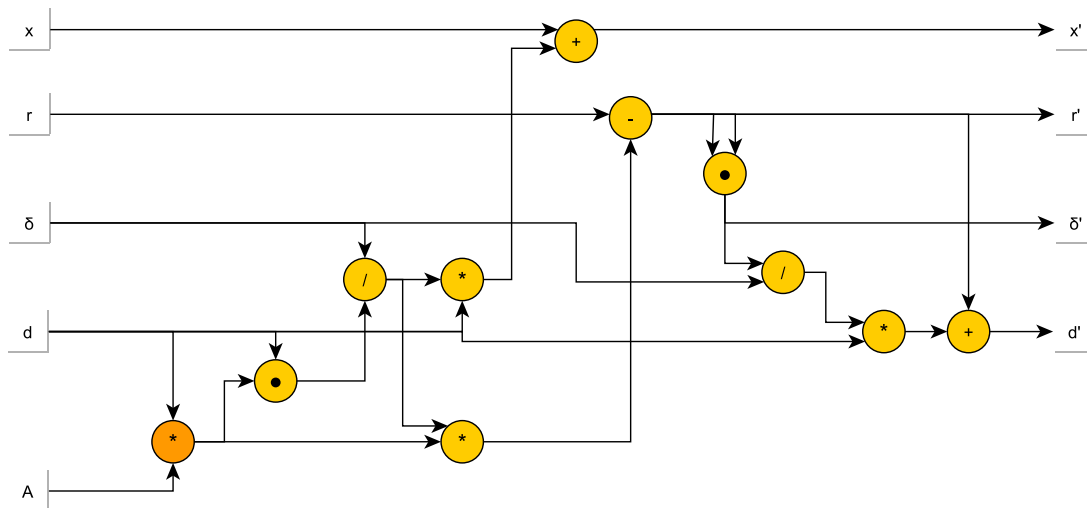


Figure 6.3: Algorithm process graph

From the process graph a list of sequential steps can be constructed, which is inserted below.

```

1  i ← 0
2  r ← ∑i bi - ∑k Ai,k xk
3  d ← r
4  δ = ∑i ri2
5
6  do:

```



```

7    $q \leftarrow \forall_i \sum_k A_{i,k} d_k$ 
8    $\alpha \leftarrow \frac{\delta}{\sum_i q_i d_i}$ 
9    $x \leftarrow \forall_i x_i + \alpha d_i$ 
10  if (i % 45) == 0
11     $r \leftarrow \forall_i b_i - \sum_k A_{i,k} x_k$ 
12  else
13     $r \leftarrow \forall_i r_i - \alpha q_i$ 
14   $new\delta \leftarrow \sum_i r_i^2$ 
15   $\beta \leftarrow \frac{new\delta}{\delta}$ 
16   $d \leftarrow \forall_i r_i + \beta d_i$ 
17   $\delta \leftarrow new\delta$ 
18   $i \leftarrow i + 1$ 
19 while  $i < 2000$  and  $\delta > min\delta$ 

```

Splitting each iteration in several mathematical operations further reduces the arithmetic complexity, but the FPGA area requirement of a single matrix-vector product is still too high. To reduce the arithmetic complexity beyond n^2 , the matrix vector product has to be split in several vector operations. The other operations are dot products, vector scaling, vector addition, vector subtraction and scalar division. With the exception of the scalar division, all these operations can be executed highly parallel. In section 5.4 several architectures are explored which can be used to further split the algorithm.

6.2 Processor design using CLaSH

The problem with designing mathematics with CLaSH is that an FPGA does not have unlimited resources. The naive algorithm from section 6.1.2 does not fit on any presently available FPGA. This is because for every multiplication, addition or other mathematical operator in the algorithm, extra hardware elements are reserved. To create synthesizable hardware which runs the algorithm, it has to be split across time. By using multiple clock cycles, each arithmetic unit can be used multiple times.

To be able to process hundreds or more samples, a very large reduction in area is required. Several architectures which promote hardware re-usage have been described in section 5.4, after which a single-purpose vector processor architecture has been chosen.

In a processor architecture the problem is split into a *program*, a series of sequential operations, and a *processor*, hardware which is able to execute those operations. The relation between processor and program is defined by the *instruction set*, the set of possible operations.

Generally speaking processors are multi-purpose, which means the instruction set and processor are designed to run a wide variety of programs. Since there is only one program, the instruction set and architecture are designed to be as efficient as possible for the pseudo-program described in section 6.1.3.

The processor hardware is described in sections 6.2.1, 6.2.2 and 6.2.3. In section 6.2.4 the memory architecture is explained in greater detail. The source code has been included in appendix C.

The instruction set is described in section 6.2.5. The program created using the instruction set can be found in section 6.2.6. Appendix D contains the program.

6.2.1 Processor-like architecture

To maximize energy efficiency a single-purpose processor has been designed. The instruction set is designed to utilize as much hardware to be as fast as possible, while minimizing unused hardware by implementing unused instructions. The optimizations are described in section 6.2.2.

The structure of the processor is based on a *Harvard Architecture*, an architecture with separate access to the instruction memory and the data memory. In a minimalistic Harvard processor

the sections that can be distinguished are the *Datapath*, *Control Unit*, *Instruction memory* and *Data memory*.

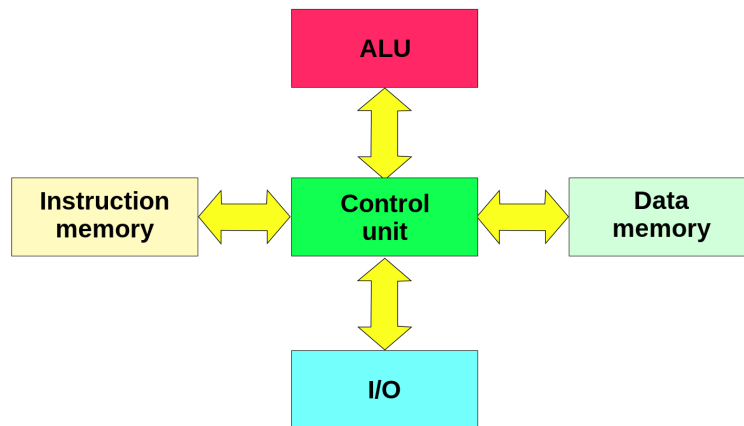


Figure 6.4: Harvard architecture

The Datapath consists of all the logic and registers required to process arithmetic and logic instructions. The heart is the *Arithmetic Logic Unit (ALU)*, which is responsible for all the comparisons and arithmetic operations.

In the control unit all the logic to control the datapath and memories is contained. The Control unit contains a *decoder* and *program counter*. The decoder processes the current instruction, while the program counter calculates which instruction should be fetched next.

The instruction memory contains the program, and the data memory contains the variables and vectors from the algorithm. In a Harvard architecture the memories are treated as physically separated and do not share an address bus, data bus and address space. The advantage is that instructions and data can be fetched simultaneously. The processor is designed to fetch the next instruction each cycle regardless of what the datapath is doing. Since the program is static, the instruction memory can be read-only.

6.2.2 Optimization: a revisit of the matrix vector multiplication

Since the largest computational load comes from matrix vector multiplications, it makes sense to design the architecture to be good at matrix vector multiplications. In section 4.2.2 the parallelization of matrix-vector multiplications has already been discussed. In this section several optimizations will be presented to make the architecture faster and more efficient when doing matrix vector multiplications.

- Vector ALU
- Short critical path
- Multiply-accumulate
- Memory address auto-increment

Vector processing

The biggest speed improvement comes from using a vector *arithmetic logic unit (ALU)*. Instead of processing a single value each cycle, multiple values are processed simultaneously. This optimization is called parallel processing (Parhi, 1995).

Short critical path

The maximum clock speed of the final hardware design is determined by the critical path. A path consists of all the logic elements from an input or memory element to an output or memory element. The path whose logical elements need the most time to produce a valid result is called the critical path. The maximum clock speed is the inverse of the time before valid data is produced of the critical path, therefore a shorter critical path increases the maximum clock frequency. Critical path length can be reduced in several ways, for example by pipelining, unfolding and retiming (Parhi, 1995).

As noted before, the biggest computational load comes from matrix-vector calculations. The matrix vector multiplication from section 6.1.2 can be split in two dimensions, across the rows or the columns. This results in the architectures shown in figure 6.5 and 6.6. It is clearly visible that the critical path in the horizontal one is far from optimal, as the critical path contains the entire chain of additions. The vertically parallelized architecture is therefore the better choice. Shifting memory elements to reduce critical path length is called retiming.

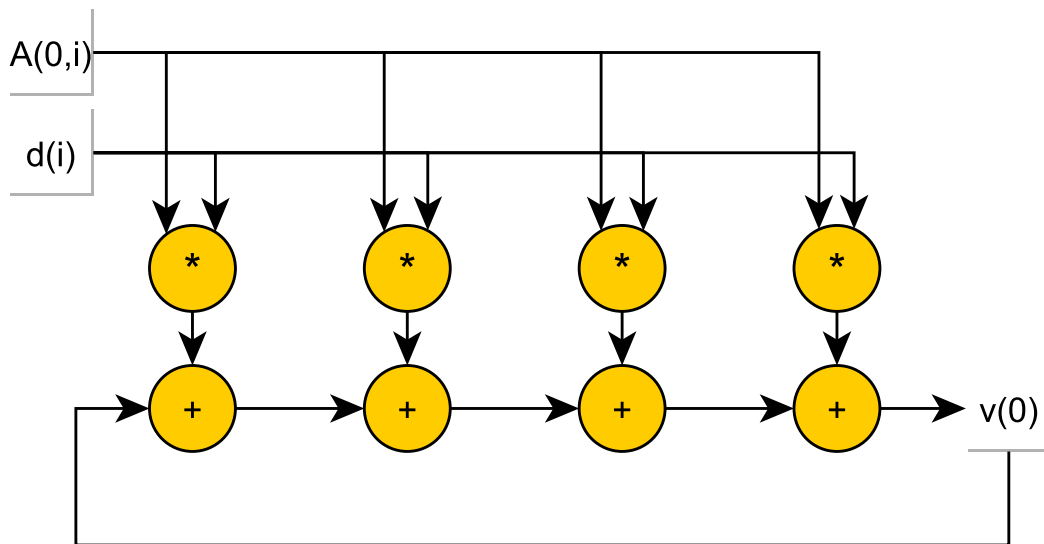


Figure 6.5: "Horizontal" parallelization

If the inputs are chosen correctly, the summation part at the bottom is not required to do a Matrix-Vector multiplication. The summation unit has been added for doing dot products. The summation unit will be described in section 6.2.3

Another solution to decrease critical path length is pipelining. In this situation, pipelining can be done by putting memory elements between the multiplications and additions. This optimization might seem useful, but has not been done because the FPGA is actually really good at doing multiplication and accumulation at the same time, which will be explained in the next subsection.

Multiply-accumulate

From the architecture shown in image 6.6, it can be seen that each matrix-vector product requires n^2 multiplications and just as much additions. For each sample a distinct *multiplier* and *accumulator* are required. This arithmetic structure is not uncommon, and a lot of hardware is specifically designed for these computations.

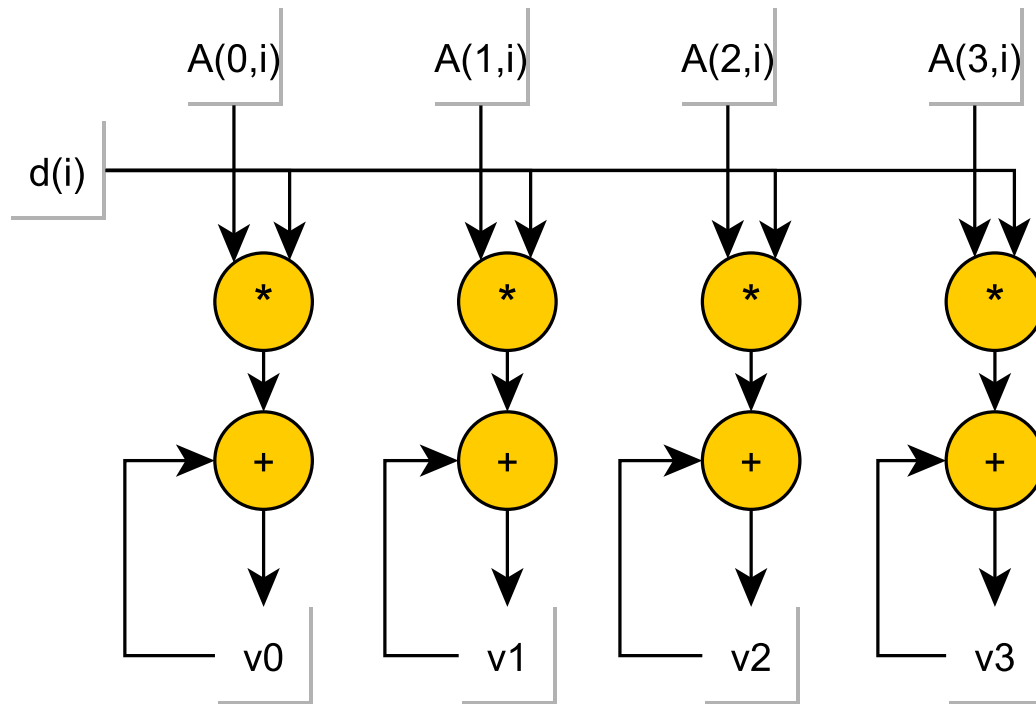


Figure 6.6: "Vertical" parallelization

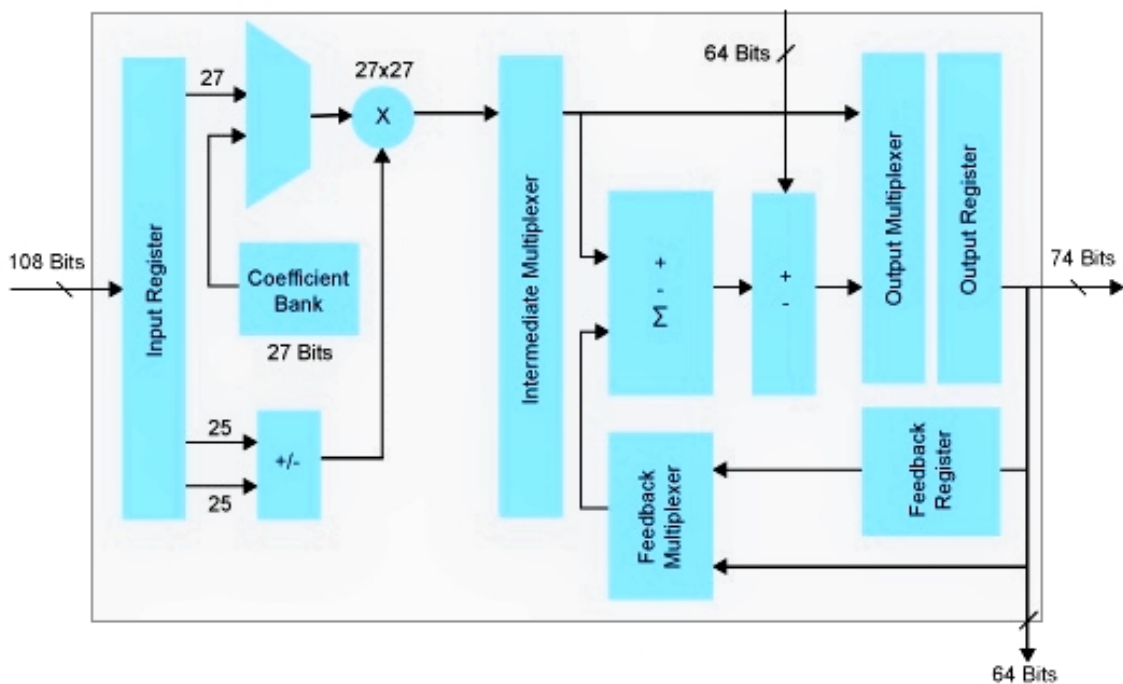


Figure 6.7: Cyclone V DSP Block

In image 6.7 the shape of a single Cyclone V DSP block is shown (Altera Corporation, 2011). More modern FPGA's use similar structures (Vishwanath, 2016). It can be seen that these DSP blocks contain a dedicated multiplier and accumulator. Since the DSP blocks are designed to run at the same clock speed as the logic, doing both computations in the same cycle comes at

almost no cost. Using this multiply-accumulate construction reduces the amount of clock cycles required for matrix-vector products by a factor 2, without increasing energy consumption and critical path delay by a comparable amount.

Memory address auto-increment

The last addition is a memory auto increment. In a normal simple processor, each computation consists of several steps. Before the actual computation, the input values are loaded into registers, requiring one or more "load" instructions. After the computation instruction, the result is stored in memory using a "store" instruction. This means for each computation, generally four instructions are required. In the case of multiply-accumulate instructions, three instructions are required for each multiply-accumulate since no "store" instruction is required.

One of the improvements made is the parallelization of memory instructions and computations. The data for the next computation is fetched in the computation instruction. The advantage is that after each multiply-accumulate the next one can be done immediately, giving a factor 3 speed improvement. The downside is that this only works with *alligned memory access*, meaning that the data for the next computation needs to be at the next memory address.

6.2.3 Other arithmetic operations

Apart from matrix vector multiplications the processor also has to be able to other vector operations, dot products and scalar division. In order to do this, several distinguishable hardware components have been added.

Vector ALU

The vector arithmetic logic unit described in section 6.2.2 is not only used for matrix-vector multiplications. The Vector ALU is able to do addition, multiplication, subtraction and multiply-accumulation. In the algorithm these are all the operations done on vectors. Vector division is not used.

Summation unit

In the dot product a summation is used, for which a summation unit is included. The summation unit is implemented as a state machine which sums a subvector in several steps. The width of the summation unit is determined by the size of the subsubvector. Smaller subsubvectors require less hardware for the summation unit, but more cycles to compute the sum of a subvector. The summation unit runs independent of the rest of the hardware, but a special branch instruction can be used to wait for the summation unit to finish.

In a more generic architecture at least one multiplexer would be present, but since the summation unit is only used for dot products, it can be directly connected to the accumulator registers of the vector ALU.

The architecture of the summation unit is a "line" summation instead of a "tree" summation because of programming complexity. The line summation is easily scalable and simple to program. A tree summation is a better choice for larger subsubvector sizes, as it has a shorter critical path.

Division unit

The algorithm contains two scalar divisions every iteration. Division is more complex than the other basic arithmetic operations in hardware, so choosing a basic single-cycle divisor would drastically reduce the maximum clock speed. Instead of doing so, a division algorithm which spans multiple cycles has been chosen. It has been implemented as a state machine which runs in parallel to the rest of the machine. A special case of branch can be used to wait for the division unit to finish before the results are used.

Scalar division is implemented using the non-restoring division algorithm. The non-restoring division algorithm is similar to the restoring division algorithm, but uses negative residuals to skip the restoring step (Galal and Pham, 2000). In the non-restoring division algorithm, the quotient and remainder are calculated in n steps, where n is the bit-width of the input scalars. The non-restoring division algorithm is explained in the lecture by Galal and Pham (2000).

In hardware the division unit consists of a bit-shifter, an adder and a few registers, which makes it very compact.

6.2.4 Memory architecture

In section 5.5 several options for the memory storage have been considered. In implementing the memory several decisions have been made to increase memory bandwidth and calculate faster.

In order to access the memory more quickly, it has been split into three memory sections with large bus widths, as can be seen in figure 6.8. The first memory contains the current point \vec{x} , search direction \vec{d} and a section for intermediate values \vec{m} . The second memory contains the residual \vec{r} , input vector \vec{b} and a section for intermediate values \vec{q} . The third memory contains the A-matrix.

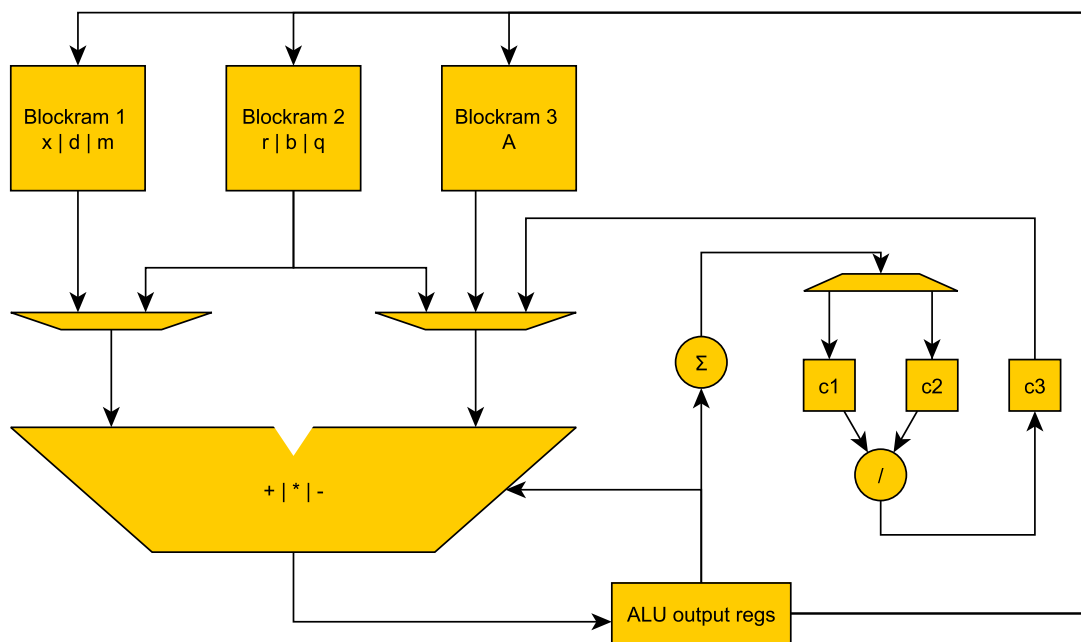


Figure 6.8: Memory architecture.

In section 6.2.2, the concept of memory address auto-incrementation has already been explained. The auto-incrementer has several operating modes, since it has to control 3 memory addresses. Apart from automatically setting memory addresses, immediate values can be set with the "Load" instruction and relative addresses can be used with the "MemIncr" instruction. The relative addresses are very useful in loops.

Between the arithmetic units and the memory there are several multiplexers. These multiplexers can be controlled using the "Mux" instruction. It has been decided to use direct connections between the memory and the vector ALU inputs, which reduces the amount of registers required. At the ALU outputs there are registers to reduce the critical path.

The memory is dual-ported so theoretically it is possible to read and store simultaneously. Defining instructions which store simultaneously would give a small reduction of cycles re-

quired at the cost of energy, added complexity and possibly a longer critical path. It has been decided that dual-port memory access will not be used, in order to keep the memory architecture simple and efficient.

From the definition of the A-matrix it follows that it must be symmetric. In a symmetric matrix the amount of storage space required can be reduced by not storing the duplicate elements above or under the main diagonal. It was chosen not to do this to reduce complexity and to have *aligned memory access*. Most types of memory, especially with larger memory sizes, are slower at obtaining individual memory elements compared to larger sections of memory. Aligned memory access means that the elements necessary at one time are stored at adjacent memory locations. The memory types that have been considered for this project all require less time and power to read aligned memory elements.

Blockram contents from file

In section 5.5 it has been decided to test with blockram. Because of the limited size of the blockram, it has been decided to calculate the A-matrix and b-vector off-line. Removing these preparations means the response matrix does not need to be saved in memory. In addition the performance metrics are more realistic, since the calculation of the A-matrix is typically not done for every measurement.

The offline calculation is done with a Matlab script, which outputs two text files containing the b-vector and the A-matrix. The formatting is done according to the specifications in the prelude library API.

6.2.5 Instruction set

The instruction set of the processor is loosely based on the assembly syntax. Several instructions, such as Jump and Branch, are very standard while others, such as Mux and VecCompute, are designed specifically for this architecture. Table 6.1 contains the instruction set.

Instruction	Parameters	Description
VecCompute	VecOp	Compute a VecOp using the vector ALU
AluCompute	AluOp	Compute a summation or division using the scalar ALU.
Mux	MuxSetting, AutoIncr	Reroute the inputs and outputs of the vector ALU. Also selects settings for the memory auto incrementer.
Jump	Address	Relative jump
Branch	BranchType, Register, CompareVal, Address	Relative jump if a condition is met. The conditions vary by branchtype. Sometimes a comparison between Register and CompareVal is required.
Load	Read addresses, Write Address	Selects absolute memory addresses for the three memories. Also sets a write address.
MIncr	Read addresses, Write Address	Selects relative memory addresses.
StoreI	Store address	Store the vector ALU output at the address specified.
Store		Store the vector ALU output.
Nop		Do nothing for one cycle.
Reset		Reset the registers to zero and restart the program.

Table 6.1: Instruction set of the custom vector processing unit.

As specified in section 6.2.3, the vector ALU can do additions, subtractions, multiplications and multiply-accumulates. The VecOp parameter instructs the vector ALU to do one of these computations, hold a value or set its registers to zero.

The AluOp is used to start the other arithmetic units, the division unit and summation unit. There are also AluOp codes to do nothing and to swap the two constant registers.

6.2.6 Program

A program is a list of instructions which are executed by a processor. The program created starts from the point where the A-matrix and b-vector are known and computes the corresponding input signal by executing the conjugate gradient algorithm. The program is split into a preparation phase and several iterations, and it will keep iterating until finished.

Before creating the program pseudo-code has been written, which is included below. The pseudo-code shows the general structure and order of execution.

```

1 :preparation
2   i ← 0
3   x ← repeat1
4   q ← ∑k Ai,kxk
5   r ← ∑i bi - q
6   d ← r
7   num ← ∑i riri
8   minDelta ← num
9
10 :iteration
11   q ← ∑k Ai,kdk
12   den ← ∑i qidi
13   res ←  $\frac{num}{den}$ 
14   m ← ∑i dires
15   x ← ∑i xi + mi
16   cmp i (something???)
17   brne :normalR
18     q ← ∑k Ai,kxk
19     r ← ∑i bi - q
20     jmp :readyR
21 :normalR
22   q ← ∑i qires
23   r ← ∑i ri - qi
24 :readyR
25   den ← num
26   num ← ∑i riri
27   res ←  $\frac{num}{den}$ 
28   m ← ∑i dires
29   d ← ∑i ri + mi
30   i ← i + 1
31   cmp num minDelta
32   brlt :end
33   cpm i 2000
34   brlt :iteration
35 :end

```

The full code is included in appendix D.

6.2.7 Testing

Unfortunately, there has not been time to create and test a physical test setup. The working of the algorithm and determination of the amount of cycles required have been done by simulating the processor and program.

Simulation

In simulation, the algorithm does not work with fixed-point values, as there is not enough dynamic range. A solution is to shift the point for each computation. A different solution is to use floating-point numbers. In simulation the algorithm runs with both fixed- and floating-point numbers.

The response matrix used for this implementation is different from the one used in the Haskell implementation, so the results cannot be compared. The script that generates a response matrix in BlockRamFile format does not work with floating point numbers.

Using the simulator, the total amount of clock cycles required for the worst case execution can be determined, as can be seen in chapter 6.2.8.

Synthesis

When using fixed-point numbers, the CLaSH compiler is able to generate synthesizable code for a sample size of 400 and a sample size of 2048. Since the Cyclone 5 and Cyclone 10 platforms do not have enough BlockRam for the 2048 sample implementation, the architecture only synthesizes for the Arria 10. The specific Arria 10 FPGA chosen is the "10AX115U4F45I3SGES". The results have been included in table 6.2.

FPGA	ALU Width	DSP Slices	Clock speed	BlockRam usage
Arria 10	512	198	1 MHz*	22 Mb
Arria 10	128	198	50.2 MHz	18 Mb
Arria 10	100	170	50.2 MHz	18 Mb
Cyclone 10	128	D.N.S.	D.N.S.	D.N.F.
Cyclone V	128	D.N.S.	D.N.S.	D.N.F.

Table 6.2: FPGA Synthesis results

In synthesizing the code using a 512 scalars wide ALU, only 198 DSP slices are used, indicating something went wrong. This also reflects in the maximum clock frequency, which is very low. The RTL viewer does list 512 independent multipliers in the RTL viewer, so they are lost in synthesis.

For the 128 wide implementation and the 100 wide implementation, the length of the response matrix has been reduced by 20% in order to be able to synthesize. Although the Arria 10 does have enough BlockRam cells, it is not able to address them all individually.

Instead of using 27-bit fixed numbers, IEEE 754 floating point numbers can be used to obtain more dynamic range. On modern Altera FPGA's, such as the Cyclone 10 and Arria 10, IEEE 754 single precision floating point DSP slices have been added (Vishwanath, 2016). The CLaSH compiler is not able to generate code for floating point numbers.

6.2.8 Energy efficiency comparison

The goal of this research is to improve the energy efficiency for signal recovery. Currently the platform to beat is a NVidia Geforce GTX1050 Ti Graphics Processing Unit (GPU).

The GPU has a maximum power usage of 75 Watt (NVIDIA Corporation, 2017). To test the worst case execution time, the early escape clause has been removed, so that all iterations will be calculated. It takes the GPU 740 milliseconds to run the algorithm with a sample size of 2048. This means about 56 Joules are required for the calculation. A Quadro M1000 has also been tested, and requires more energy. The results are included in in table 6.4.

To get the corresponding performance numbers from FPGA's, Altera tooling has been used. The Timequest analyser and Powerplay analyser can be used to get the maximum clock frequency and estimated maximum power usage respectively. The GPU is compared against the Cyclone V, Cyclone 10 and Arria 10 FPGA platforms.

For the algorithm running on the FPGA, the exact amount of cycles required for the worst-case execution can be determined. These results are included in table 6.3. The Altera Timequest analyser has been used to determine the maximum clock speed for the FPGA platforms. In table 6.3 the Maximum clock speed and corresponding worst case execution time have been listed.

The Altera Powerplay analyser has been used to determine the maximum power draw of the FPGA's running the algorithm. These results are included in table 6.4. The time and power requirements have been combined into theoretical energy requirements.

FPGA	DSP Slices	Clock cycles	Clock speed	Worst-case execution time
Arria 10	512	17978064	1 MHz*	17978 ms
Arria 10	128	69470608	50.2 MHz	1384 ms
Arria 10	100	82935050	50.2 MHz	1653 ms
Cyclone 10	128	69470608	D.N.S.	
Cyclone V	128	69470608	D.N.S.	

Table 6.3: FPGA Execution times

Platform	Worst-case execution time	Load power usage	Worst-case energy
Geforce GTX1050 Ti	740 ms	75 W	55.5 J
Quadro M1000	6900 ms	45 W	310.5 J
Arria 10 (512)	17978 ms	2707.3 mW	48.67 J
Arria 10 (128)	1384 ms	2294.64 mW	3.18 J

Table 6.4: Energy comparison

In this results it has to be taken into account that the GPU estimations are more pessimistic than the estimates provided by the Altera tools. Nevertheless the energy efficiency advantage of the Arria 10 FPGA with a 128 numbers wide ALU is at least a factor 8.

What further strengthens this conclusion is that the Arria 10 can be used standalone, while the GPU needs a desktop computer in order to work.

7 Conclusion

By studying the algorithm, it has been found that the computational complexity of maximum-likelihood signal recovery comes from matrix-vector multiplications, which add up to a complexity of $O(n^3)$. Calculating the A-matrix using a matrix product also has a high complexity, but it is not necessary to calculate it for every measurement.

In software a naive implementation of the algorithm is made, and it is shown to effectively recover input signals. The CLaSH implementation is synthesizable, but any implementation with a useful amount of samples does not fit on an FPGA.

A processor architecture has been designed to provide hardware re-usage and reduce the amount of FPGA resources required. The architecture is optimized for matrix-vector computations in order to calculate the conjugate gradient part of the algorithm very fast. In simulations the arithmetic operations work as intended, but running the entire algorithm causes fixed-point overflow errors. Using floating point numbers works in simulation, but does not synthesize.

The processor architecture is synthesized for an Arria 10 FPGA and the relevant performance figures are listed in chapter 6.2.8. Theoretically on a modern FPGA a factor 15 increase in energy efficiency over the given GPU implementation is possible, for a conjugate gradient architecture using blockram for the storage of matrices and vectors.

Using Clash for programming a conjugate gradient algorithm resulted in an architecture which is fast and energy efficient, but did require extensive knowledge about functional programming, FPGA internals and processor design. Commercial exploitation of the architecture is not advised, as developers able to maintain, improve or change the architecture are almost impossible to come by.

7.1 Future work

The architecture does not synthesize for a Cyclone FPGA because of the amount of BlockRam cells required. An improvement would be to utilize different types of memory in order to work with larger problems or smaller FPGA's, for example DDR3 memory could be used. Switching memory architecture does have consequences for the speed and energy efficiency, and new measurements should be made.

To make the processor architecture and program suitable for commercial exploitation, it should be easier to program with. Expecting future programmers to have a deep understanding of processor architecture is not realistic. At the University of Twente research is being done on developing mathematical algorithms with a generic architecture.

The current architecture does calculations from and to memory, where the memory is filled at compile time. To be used with an IMS device the FPGA also needs to be able to sample or receive input data, and output the restored signal. An option is to implement the current design as a signal recovery co-processor and use other parts of the FPGA for sampling and further processing.

Bibliography

- Altera Corporation (2011), Enabling High-Performance DSP Applications with Arria V or Cyclone V Variable-Precision DSP Blocks.
- Appel, R. and H. Folmer (2016), Analysis, optimization, and design of a SLAM solution for an implementation on reconfigurable hardware (FPGA) using CLaSH.
- Baaij, C. (2009), C[∞]SH : from Haskell to hardware.
<http://essay.utwente.nl/59482/>
- Banerjee, P., D. Bagchi, M. Haldar, A. Nayak, V. Kim and R. Uribe (2003), Automatic Conversion of Floating Point MATLAB Programs into Fixed Point FPGA Based Hardware Design, *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)*.
www.accelchip.com
- Cumeras, R., E. Figueras, C. E. Davis, J. I. Baumbach and I. Gràcia (2014), Review on Ion Mobility Spectrometry. Part 1: current instrumentation, *The Analyst*, **vol. 140**, doi:10.1039/c4an01100g.
<http://pubs.rsc.org/en/Content/ArticleLanding/2015/AN/C4AN01100G>
- Curry, H. B. (1944), -NOTES- THE METHOD OF STEEPEST DESCENT FOR NON-LINEAR MINIMIZATION PROBLEMS*.
<https://cs.uwaterloo.ca/~y328yu/classics/curry.pdf>
- Eiceman, G. A. and Z. Karpas (1995), *Ion Mobility Spectrometry*.
- Espy, C. and J. Lim (1983), Effects of additive noise on signal reconstruction from Fourier transform phase, **vol. 31**, no.4, pp. 894 – 898.
- Fletcher, R. and M. J. D. Powell (1963), A Rapidly Convergent Descent Method for Minimization, **vol. 6**, no.2, pp. 163–168, doi:10.1093/comjnl/6.2.163.
<https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/6.2.163>
- Galal, S. and D. Pham (2000), Division Algorithms and Hardware Implementations EE 213A: Advanced DSP Circuit Design.
[http://www.seas.ucla.edu/~ingrid/ee213a/lectures/division\[_\]presentV2.pdf](http://www.seas.ucla.edu/~ingrid/ee213a/lectures/division[_]presentV2.pdf)
- Gerards, M., C. Baaij, J. Kuper and M. Kooijman (2011), Higher-Order Abstraction in Hardware Descriptions with C[∞]SH, in *14th EUROMICRO Conference on Digital System Design, DSD 2011*, Ed. P. Kitsos, IEEE Computer Society, pp. 495–502, ISBN 9780769544946.
<https://doi.org/10.1109/DSD.2011.69>
- Gilbert, J. C. and J. Nocedal (1990), Global convergence properties of conjugate gradient methods for optimization, , no.1268.
<https://hal.inria.fr/inria-00075291>
- Hameed, R., W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis and M. Horowitz (2010), Understanding Sources of Inefficiency in General-Purpose Chips, *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 37–47.
http://csl.stanford.edu/~christos/publications/2010_efficiency.isca.pdf
- Hayes, M. H. I. (1981), Signal reconstruction from phase or magnitude.

- Hestenes, M. R. and E. Stiefel (1952), Methods of Conjugate Gradients for Solving Linear Systems, *Journal of Research of the National Bureau of Standards*, **vol. 49**.
<https://www.fing.edu.uy/inco/cursos/numerico/aln/hes{ }stief1952.pdf>
- Holmes, T. J. and Y.-H. Liu (1991), Acceleration of maximum-likelihood image restoration for fluorescence microscopy and other noncoherent imagery, **vol. 8**, no.6, p. 893, ISSN 1084-7529, doi:10.1364/JOSAA.8.000893.
<https://www.osapublishing.org/abstract.cfm?URI=josaa-8-6-893>
- Johnson, C. R. (1970), Positive Definite Matrices, **vol. 77**, no.3, p. 259, ISSN 00029890, doi:10.2307/2317709.
<http://www.jstor.org/stable/2317709?origin=crossref>
- Karasek, F. W. (1970), The plasma chromatograph, *Research and Development*, **vol. 21**, pp. 34 – 37.
- Kosarev, E. L. (2002), Shannon's superresolution limit for signal recovery, **vol. 6**, no.3, pp. 479–479, ISSN 02665611, doi:10.1088/0266-5611/6/3/515.
- Kumar, S. (2015), Fundamental Limits to Moore ' s Law, pp. 1–3.
- Langevin, P. (1905), Une formule fondamentale de théorie cinétique, *Annales de chimie et de physique*, **vol. 5**, pp. 245 – 288.
- Lanteh Ri, H., M. Roche, O. Cuevas and C. Aime (2001), A general method to devise maximum-likelihood signal restoration multiplicative algorithms with non-negativity constraints, *Signal Processing*, **vol. 81**, pp. 945–974.
<http://dunand.northwestern.edu/refs/files/Lanteri.pdf>
- Loo, S. M., J. P. Cole and M. M. Gribb (2008), Hardware / Software Codesign in a Compact Ion Mobility Spectrometer Sensor System for Subsurface Contaminant Detection, *EURASIP Journal on Embedded Systems*, **vol. 2008**, pp. 1–8, doi:10.1155/2008/137295.
- Louis, R. and H. Hill (1990), Ion mobility spectrometry in analytical chemistry, *Analytical Chemistry*, **vol. 21**, pp. 321 – 355.
- Mack, C. A. (2011), Fifty years of Moore's law, **vol. 24**, no.2, pp. 202–207, ISSN 08946507, doi:10.1109/TSM.2010.2096437.
- Michalewicz, Z. (1995), Genetic Algorithms, Numerical Optimization, and Constraints.
<https://cs.adelaide.edu.au/users/zbyszek/Papers/pl6.pdf>
- Nayak, A., M. Haldar, A. Choudhary and P. Banerjee (2001), Precision and Error Analysis of Matlab Applications During Automated Hardware Synthesis for Fpgas, *Proceedings of the conference on Design, automation and test in Europe*.
<https://pdfs.semanticscholar.org/0b39/78e6177b5844d9aa42caa9199fd957fe2201.pdf>
- NVIDIA Corporation (2016), NVIDIA ® TESLA ® P100: INFINITE COMPUTE POWER FOR THE MODERN DATA CENTER.
<http://images.nvidia.com/content/tesla/pdf/nvidia-teslap100-techoverview.pdf>
- NVIDIA Corporation (2017), GeForce GTX 1050 Ti | Specifications | GeForce.
<https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-1050-ti/specifications>
- Parhi, K. K. (1995), High-Level Algorithm and Architecture Transformations for DSP Synthesis, **vol. 9**, no.9, pp. 121–143.
<https://link.springer.com/content/pdf/10.1007%}2F02406474.pdf>

- Reed, R. G., M. A. Cox, G. T. Wrigley and B. Mellado (2015), A CPU benchmarking characterization of ARM based processors, **vol. 7**, no.3, pp. 581–586.
<http://crm-en.ics.org.ru/uploads/crmissues/crm{ }2015{ }3/15728.pdf>
- Sandstrom, J. (1995), Comparing Verilog to VHDL Syntactically and Semantically.
<http://www.sandstrom.org/systemde.htm>
- Saxena, R. and K. Singh (2005), Fractional Fourier transform : A novel tool for signal processing, *Journal of The Indian Institute of Science*, **vol. 85**, pp. 11–26, ISSN 09704140.
- Shewchuk, J. R. (1994), An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, **vol. 49**, no.CS-94-125, p. 64, ISSN 14708728, doi:10.1.1.110.418.
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.418{&}rep=rep1{&}type=pdf{ }5Cnhttp://www.cs.cmu.edu/{~}quake-papers/painless-conjugate-gradient.pdf>
- Soliman, S. S. and M. D. Srinath (1998), *Continuous and discrete signals and systems*, Pearson Education (Us), 2nd edition, ISBN ISBN 0-13-518473-8.
- Soussen, C., J. Idier, D. Brie, J. Duan, C. Soussen, D. Brie and J. Duan (2010), From Bernoulli-Gaussian deconvolution to sparse signal restoration.
<https://hal.archives-ouvertes.fr/file/index/docid/443842/filename/part1.pdf>
- Strang, G. and C. Moler (2015), RES.18-009 Learn Differential Equations: Up Close with Gilbert Strang and Cleve Moler: Positive Definite Matrices.
<https://ocw.mit.edu/resources/res-18-009-learn-differential-equations-up-close-with-gilbert-strang-and-cleve-moler-differential-equations-and-linear-algebra/applied-mathematics-and-ata/positive-definite-matrices/>
- Texas Instruments Incorporated. (2017), C66x | C6000 | Overview | DSP | TI.com.
<http://www.ti.com/processors/dsp/c6000-dsp/c66x/overview.html>
- Vishwanath, A. (2016), Enabling High-Performance Floating-Point Designs.
<https://www.altera.com/content/dam/altera-www/global/en{ }US/pdfs/literature/wp/wp-01267-fpgas-enable-high-performance-floating-point.pdf>
- Zolotov, Y. A. (2006), Ion mobility spectrometry, **vol. 61**, no.6, p. 519, ISSN 1608-3199, doi:10.1134/S1061934806060013.
<http://dx.doi.org/10.1134/S1061934806060013>

Appendix A: Haskell implementation

```

1
2 module MaxLikelihood where
3
4 import Data.Array
5 import MatrixMath
6 import ResponseFunction
7
8 --Settings
9 sampleSize = 200
10
11 --Todo: This can be done in a more efficient way with only O(n) calls to kernel
12 responseMatrix :: Int -> Array (Int, Int) Double
13 responseMatrix m = array ((0,0),(m-1, m-1)) [(i, j), response i j | i <- [0..(m-1)]
14   , j <- [0..(m-1)]]
15   where response i j | j >= i = kernel j i
16   | otherwise = 0
17
18 --Now we can create the A-matrix and B-Vector used in the matrix notation of the
19   problem
20 --Likelihood(i_m) = -2 * i_m * b + i_m * i_n * a_mn
21 mkAMatrix :: Int -> Array (Int, Int) Double
22 mkAMatrix m = (transp (responseMatrix m) /**/ (responseMatrix m))
23
24 mkBVector :: Int -> Array (Int, Int) Double
25 mkBVector m = (responseMatrix m /**/ s_k)
26   where
27     s_k = listArray ((0,0),(m-1, 0)) (corruptedOutputSet m)
28     --s_k = listArray ((0,0),(m-1, 0)) (outputSet m)
29
30 a_mn = mkAMatrix sampleSize --A Matrix
31 b = mkBVector sampleSize --b vector
32
33 --Conjugate gradient algorithm
34 --Source: Shewchuk, Jonathan Richard. "An introduction to the conjugate gradient
35   method without the agonizing pain." (1994).
36 --Obtain value '' from tuple of output values:
37 obt_x (x, r, d,  $\delta$ ,  $\alpha$ ,  $\beta$ , i, maxN, min $\delta$ ) = x -- Current point
38 obt_r (x, r, d,  $\delta$ ,  $\alpha$ ,  $\beta$ , i, maxN, min $\delta$ ) = r -- Residual, Negative gradient
39 obt_d (x, r, d,  $\delta$ ,  $\alpha$ ,  $\beta$ , i, maxN, min $\delta$ ) = d -- Search direction
40 obt_ $\delta$  (x, r, d,  $\delta$ ,  $\alpha$ ,  $\beta$ , i, maxN, min $\delta$ ) =  $\delta$  -- Magnitude of Residual/Gradient
41 obt_ $\alpha$  (x, r, d,  $\delta$ ,  $\alpha$ ,  $\beta$ , i, maxN, min $\delta$ ) =  $\alpha$  -- Step Length, Line search result
42 obt_ $\beta$  (x, r, d,  $\delta$ ,  $\alpha$ ,  $\beta$ , i, maxN, min $\delta$ ) =  $\beta$  -- Optimization parameter
43 obt_i (x, r, d,  $\delta$ ,  $\alpha$ ,  $\beta$ , i, maxN, min $\delta$ ) = i -- Iteration number
44 obt_N (x, r, d,  $\delta$ ,  $\alpha$ ,  $\beta$ , i, maxN, min $\delta$ ) = maxN -- Maximum amount of iteration
45
46 --Iteration zero: Choose a starting point and go in the direction of steepest descent
47
48 -- Note: /-/, /**/, /**/ and other strange symbols are infix operations for vector (/)
49   or matrix (//) math
50 -- So for example /**/ means "matrix multiplied with vector"
51 conjugateGradient = iteration (x0, r0, d0,  $\delta$ 0,  $\alpha$ 0,  $\beta$ 0, 1, sampleSize, min $\delta$ )
52   where
53     x0 = listArray((0,0), ((sampleSize-1),0)) (repeat 0)
54     r0 = b /-/ a_mn /**/ x0
55     d0 = r0
56      $\delta$ 0 = transp r0 /**/ r0
57      $\alpha$ 0 =  $\delta$ 0 /// (transp d0 /**/ a_mn /**/ d0)
58      $\beta$ 0 = listArray((0,0), (0,0)) [0]
59     min $\delta$  =  $\delta$ 0!(0,0) * (noiseStdDev^2)
60
61 --Iteration function. Calculate the values for one iteration and then make a
62   recursive call to calculate the next iteration.
63 --The recursion ends at maxN samples or when  $\delta$  get's too low.
64 --Note: ' denotes "next value of" or (i+1)
65 iteration (x, r, d,  $\delta$ ,  $\alpha$ ,  $\beta$ , i, maxN, min $\delta$ )
66   | i >= maxN
67   ||  $\delta$ !(0,0) < min $\delta$  = (x', r', d',  $\delta'$ ,  $\alpha'$ ,  $\beta'$ , i', maxN, min $\delta$ )
68   | otherwise = iteration (x', r', d',  $\delta'$ ,  $\alpha'$ ,  $\beta'$ , i', maxN, min $\delta$ )

```

```
63 where
64   x' = x /+ (d //*  $\alpha$ )
65   r' | (i `mod` 40 == 0) = b /- a_mn //* x'
66     | otherwise       = r /- a_mn //*  $\alpha$  //* d
67   d' = r' /+ d //*  $\beta'$ 
68    $\delta'$  = transp r' //* r'
69    $\alpha'$  =  $\delta'$  /// (transp d' //* a_mn //* d')
70    $\beta'$  =  $\delta'$  ///  $\delta$ 
71   i' = i+1
```


Appendix B: Naive CLaSH implementation

```

1
2 module MaxLikelihood where
3
4 import CLaSH.Prelude
5
6 type Number = SFixed 3 24
7 type Vect = Vec 1000 Number
8 type Matr = Vec 1000 (Vect)
9 --All matrices are square...
10
11 --Settings
12 sampleSize = d1000
13 iterations = 1000
14
15 kernel :: Integer -> Integer -> Number
16 kernel limitN k | k<= limitN = 0.2 * exp (-(fromIntegral (limitN-k)*0.2))
17               | otherwise = 0
18
19 samples = map (sin) index
20 responseMatrix = map row index
21   where row limitN = map (kernel limitN) index
22
23 index = iterate d1000 (+1) 1
24
25 a_mn = mmMult (transpose responseMatrix) responseMatrix
26
27 --Conjugate gradient algorithm
28 --Source: Shewchuk, Jonathan Richard. "An introduction to the conjugate gradient
29   method without the agonizing pain." (1994).
30 conjugateGradient :: Vect -> Signal () -> Signal (Maybe (Vect))
31 conjugateGradient samples = mealy iteration (x0, r0, d0, r_n2, alpha0, 0, b_m)
32   where
33     x0 = replicate sampleSize 0
34     r0 = zipWith (-) b_m (mvMult a_mn x0)
35     d0 = r0
36     r_n2 = rcMult r0 r0
37     alpha0 = r_n2 / (rcMult d0 (mvMult a_mn d0))
38     b_m = mvMult responseMatrix samples
39
40 iteration :: (Vect, Vect, Vect, Number, Number, Unsigned 27, Vect)
41   -> ()
42   -> ((Vect, Vect, Vect, Number, Number, Unsigned 27, Vect), Maybe (Vect))
43 iteration (x_n, r_n, d_n, r_n2, alpha_n, n, b_m) _
44   | stop      = ((x_n, r_n, d_n, r_n2, alpha_n, n, b_m), Just x_n)
45   | otherwise = ((x_n', r_n', d_n', r_n2', alpha_n', n', b_m), Nothing)
46   where
47     x_n' = zipWith (+) x_n (map (* alpha_n) d_n)
48     r_n' = zipWith (-) r_n (mvMult a_mn (map (* alpha_n) d_n))
49     d_n' = zipWith (+) r_n' (map (* beta_n') d_n)
50     r_n2' = rcMult r_n' r_n'
51     alpha_n' = r_n2' / (rcMult d_n' (mvMult a_mn d_n'))
52     beta_n' | r_n2 == 0 = 0
53             | otherwise = r_n2' / r_n2
54     n' = n+1
55     stop = (n >= iterations) || (r_n2' == 0)
56     --redirect = n == (sqrt maxN)
57
58
59 --MATRIX MATH
60 rcMult :: Vect -> Vect -> Number
61 rcMult a b = foldl (+) 0 $ zipWith (*) a b
62
63 crMult :: Vect -> Vect -> Matr
64 crMult a b = map row a
65   where
66     row a_n = map (* a_n) b
67

```

```
68 vvMult :: Vect -> Vect -> Vect
69 vvMult a b = zipWith (*) a b
70
71 mmMult :: Matr -> Matr -> Matr
72 mmMult a b = map row a
73   where
74     b' = transpose b
75     row a_n = map (rcMult a_n) b'
76
77
78 mvMult :: Matr -> Vect -> Vect
79 mvMult a b = map (row) a
80   where
81     row a_n = foldl (+) 0 $ zipWith (*) a_n b
82
83 msMult :: Matr -> Number -> Matr
84 msMult a b = map (map (* b)) a
85
86
87 --TOP ENTITY
88 topEntity = (conjugateGradient samples)
```

Appendix C: Processor architecture

```

1 {-# LANGUAGE RecordWildCards, TupleSections #-}
2 module VPU where
3
4 import CLaSH.Prelude
5 import Data.Maybe
6
7 $(decLiteralD 40000)
8 $(decLiteralD 1999)
9
10 type MemAddr    = Signed 32
11 type VecIndex   = Unsigned 6
12 type Value      = Signed 32
13
14 type DivScalar  = SFixed 11 17
15 type Scalar     = SFixed 10 17
16 type Vect       = Vec 400 Scalar
17 type SubVect    = Vec 100 Scalar
18 type SubSubVect = Vec 4 Scalar
19
20 --VPU instruction set
21 data Instruction
22   = VecCompute VecOp
23   | AluCompute AluOp
24   | Mux MuxLeft MuxRight MuxOut MuxConst AutoIncr
25   | Jump Value
26   | Branch BranchOp Identifier Value Value
27   | Load MemAddr MemAddr MemAddr MemAddr
28   | MIncr Value Value Value Value
29   | StoreI MemAddr
30   | Store
31   | Nop
32   | Reset
33   deriving (Eq, Show)
34
35 --Computation OpCodes
36 data VecOp = Add | Sub | Mult | MACC | Cp | VNop
37   deriving (Eq, Show)
38
39 data AluOp = Sum | SetZero | Div | Swap | Imm
40   deriving (Eq, Show)
41
42 data BranchOp = NB | JMP | BLT | BGT | BEQ | BNE | BDIV | BSUM
43   deriving (Eq, Show)
44
45 --Mux settings
46 data MuxLeft = ML_Ram2 | ML_Ram3 | ML_Const
47   deriving (Eq, Show)
48 data MuxRight = MR_Ram1 | MR_Ram2 | MR_Const
49   deriving (Eq, Show)
50 data MuxOut = MO_Ram1 | MO_Ram2
51   deriving (Eq, Show)
52 data MuxConst = MC_NUM | MC_DEN
53   deriving (Eq, Show)
54
55 --Auto increment settings : Which memory addresses should AI
56 data AutoIncr = AI_Ram1 | AI_Ram2 | AI_Ram3 | AI_Ram12
57   deriving (Eq, Show)
58
59 emptyVec :: SubVect
60 emptyVec = replicate d100 0
61
62 --Register file
63 data Mem
64   = Mem
65   { pc      :: Value
66   , ic      :: Value
67   , aluRegs :: SubVect
68   , divRegs :: (Scalar, Scalar, DivScalar, Value)

```

```

69 , sumRegs      :: (Scalar, VecIndex)
70 , divNum      :: Scalar
71 , divDen      :: Scalar
72 , c3          :: Scalar
73 , muxL        :: MuxLeft
74 , muxR        :: MuxRight
75 , muxO        :: MuxOut
76 , muxC        :: MuxConst
77 , autoI       :: AutoIncr
78 , vecCnt      :: VecIndex
79 , addrR1      :: MemAddr
80 , addrR2      :: MemAddr
81 , addrR3      :: MemAddr
82 , addrR4      :: MemAddr
83 } deriving (Eq, Show)
84
85 emptyRegs = Mem 0 0 emptyVec (0, 0, 0, 0) (0,0) 0 0 0 ML_Ram3 MR_Ram2 MO_Ram2 MC_NUM
      AI_Ram3 0 0 0 0 0
86
87 data Identifier = RPC | RAD1 | RAD2 | RAD3 | RAD4 | SumR | DivR
88   deriving (Eq, Show)
89
90 --Instruction machine code
91 data MachCode
92   = MachCode
93   { mcAluOp      :: AluOp
94   , mcVecOp      :: VecOp
95   , mcBrOp       :: BranchOp
96   , mcMuxL       :: Maybe MuxLeft
97   , mcMuxR       :: Maybe MuxRight
98   , mcMuxO       :: Maybe MuxOut
99   , mcMuxC       :: Maybe MuxConst
100  , mcAutoI      :: Maybe AutoIncr
101  , mcAddr1      :: Maybe MemAddr
102  , mcAddr2      :: Maybe MemAddr
103  , mcAddr3      :: Maybe MemAddr
104  , mcAddr4      :: Maybe MemAddr
105  , mcJumpVal    :: Maybe Value
106  , mcCmpr       :: Maybe Value
107  , mcRegId      :: Maybe Identifier
108  } deriving (Eq, Show)
109
110 --The instruction for doing absolutely nothing at all
111 nullCode = MachCode
112   { mcAluOp      = Imm
113   , mcVecOp      = VNop
114   , mcBrOp       = NB
115   , mcMuxL       = Nothing
116   , mcMuxR       = Nothing
117   , mcMuxO       = Nothing
118   , mcMuxC       = Nothing
119   , mcAutoI      = Nothing
120   , mcAddr1      = Nothing
121   , mcAddr2      = Nothing
122   , mcAddr3      = Nothing
123   , mcAddr4      = Nothing
124   , mcJumpVal    = Nothing
125   , mcCmpr       = Nothing
126   , mcRegId      = Nothing
127   }
128
129
130 --The CPU/VPU: {Decoder, Vector ALU, ALU, Program counter, Multiplexers and Registers
      }
131 --Cpu :: State -> (Inputs) ->
132 --      (State, (Outputs))
133 cpu :: Mem -> (SubVect, SubVect, SubVect, Instruction) ->
134   (Mem, (Scalar, MemAddr, Maybe (MemAddr, SubVect),
135         MemAddr, Maybe (MemAddr, SubVect),
136         MemAddr, Maybe (MemAddr, SubVect), Value))
137 cpu regFile (m1, m2, m3, instr) = (regFile', (output, rdAddr1, dout1, rdAddr2, dout2,
138         rdAddr3, dout3, ipntr))
138   where

```

```

139  --Decode the instruction and label the registers
140  (MachCode {..}) = case instr of
141    VecCompute op  -> nullCode {mcVecOp = op}
142    AluCompute op  -> nullCode {mcAluOp = op}
143    Mux a b c d e  -> nullCode {mcMuxL = Just a, mcMuxR = Just b, mcMuxO = Just c,
      mcMuxC = Just d, mcAutoI = Just e}
144    Jump addr     -> nullCode {mcJumpVal = Just addr, mcBrOp = JMP}
145    Branch a b c d -> nullCode {mcBrOp = a, mcRegId = Just b, mcCmpr = Just c,
      mcJumpVal = Just d}
146    Load a b c d  -> nullCode {mcAddr1 = Just a, mcAddr2 = Just b, mcAddr3 = Just
      c, mcAddr4 = Just d}
147    MIncr a b c d -> nullCode
148    StoreI addr   -> nullCode {mcAddr4 = Just addr}
149    Store         -> nullCode
150    Nop           -> nullCode
151    Reset        -> nullCode
152  (Mem {..}) = regFile
153
154  --Vector ALU with output registers
155  result = vAlu mcVecOp vAluL vAluR aluRegs
156  aluRegs' = result
157  output = head result
158  --Normal ALU (Not really an ALU, just the calculations not done by vALU)
159
160
161  c1' | mcAluOp == Swap = divDen
162      | muxC == MC_NUM = sum
163      | otherwise = divNum
164  c2' | mcAluOp == Swap = divNum
165      | muxC == MC_DEN = sum
166      | otherwise = divDen
167
168  sumRegs'@(sum, sumCnt) | mcAluOp == Sum = summator aluRegs (0, 2)      --sum
      <- sum(aluRegs)
169                          | otherwise = summator aluRegs sumRegs      --
      continue summation
170
171  divRegs'@(dA, dB, dP, dCnt) | mcAluOp == Div = divider (divNum, divDen, 0, 27) --
      dA <- c1 / c2
172                          | otherwise = divider divRegs      --
      continue division
173
174  --Program counter w/ Jumps
175  regCmpr = selReg (fromMaybe RAd1 mcRegId) regFile
176  branchBool = case mcBrOp of
177    NB -> False
178    JMP -> True
179    BLT -> regCmpr < fromMaybe 0 mcCmpr
180    BGT -> regCmpr > fromMaybe 0 mcCmpr
181    BEQ -> regCmpr == fromMaybe 0 mcCmpr
182    BNE -> not $ regCmpr == fromMaybe 0 mcCmpr
183    BDIV -> dCnt /= 0
184    BSUM -> sumCnt /= 0
185
186  pc' | branchBool = ipntr + (fromMaybe 1 mcJumpVal)
187      | otherwise = ipntr + 1
188
189  ipntr = pc
190
191  --Vector Multiplex registers
192  muxL' = fromMaybe muxL mcMuxL
193  muxR' = fromMaybe muxR mcMuxR
194  muxC' = fromMaybe muxC mcMuxC
195  muxO' = fromMaybe muxO mcMuxO
196  autoI' = fromMaybe autoI mcAutoI
197
198  --Change address, autoincrement, nothing
199  (ai_1, ai_2, ai_3, vecCnt') = case autoI of
200    AI_Ram1 -> (1, 0, 0, 0)
201    AI_Ram2 -> (0, 1, 0, 0)
202    AI_Ram12 -> (1, 1, 0, 0)
203    AI_Ram3 -> if vecCnt == 19 then (1, 1, 1, 0)
204              else (0, 0, 1, vecCnt + 1)

```

```

205
206   addrR1' = case instr of
207     VecCompute v -> addrR1 + ai_1
208     MIncr a b c d -> addrR1 + a
209     otherwise -> fromMaybe addrR1 mcAddr1
210   addrR2' = case instr of
211     VecCompute v -> addrR2 + ai_2
212     --MIncr a b c d -> addrR1 + a
213     otherwise -> fromMaybe addrR2 mcAddr2
214   addrR3' = case instr of
215     VecCompute v -> addrR3 + ai_3
216     --MIncr a b c d -> addrR1 + a
217     otherwise -> fromMaybe addrR3 mcAddr3
218   addrR4' = case instr of
219     Store -> addrR4 + 1
220     --MIncr a b c d -> addrR1 + a
221     otherwise -> fromMaybe addrR4 mcAddr4
222
223   --Vector Multiplexers
224   vAluL | muxL == ML_Ram2 = m2
225         | muxL == ML_Ram3 = m3
226         | muxL == ML_Const = repeat dA
227   vAluR | muxR == MR_Ram1 = m1
228         | muxR == MR_Ram2 = m2
229         | muxR == MR_Const = repeat $ m1 !! vecCnt
230
231   --Output stuff
232   writeMem = case instr of
233     Store -> Just (addrR4, result)
234     otherwise -> (, result) <$> mcAddr4
235   dout1 | muxO == MO_Ram1 = writeMem
236         | otherwise = Nothing
237   dout2 | muxO == MO_Ram2 = writeMem
238         | otherwise = Nothing
239   dout3 = Nothing
240
241   rdAddr1 = bound 0 59 addrR1'
242   rdAddr2 = bound 0 59 addrR2'
243   rdAddr3 = bound 0 39999 addrR3'
244
245   -- update registers
246   regFile' = regFile {
247     pc      = pc',
248     aluRegs = aluRegs',
249     divRegs = divRegs',
250     sumRegs = sumRegs',
251     divNum  = c1',
252     divDen  = c2',
253     c3      = c3,
254     muxL    = muxL',
255     muxR    = muxR',
256     muxO    = muxO',
257     muxC    = muxC',
258     autoI   = autoI',
259     vecCnt  = vecCnt',
260     addrR1  = addrR1',
261     addrR2  = addrR2',
262     addrR3  = addrR3',
263     addrR4  = addrR4'
264   }
265
266   --Arithmetic units--
267
268   --Vector Alu : takes two vectors and a state, produces one vector
269   vAlu :: VecOp -> SubVect -> SubVect -> SubVect -> SubVect
270   vAlu Add  xs ys zs = zipWith (+) xs ys
271   vAlu Sub  xs ys zs = zipWith (-) xs ys
272   vAlu Mult xs ys zs = zipWith (*) xs ys
273   vAlu MACC xs ys zs = zipWith (+) zs $ zipWith (*) xs ys
274   vAlu Cp   xs ys zs = xs
275   vAlu VNop xs ys zs = zs
276

```

```

277 --Summator : takes one vector, one scalar and a state as input, produces scalar and
      state
278 summator :: SubVect -> (Scalar, VecIndex) -> (Scalar, VecIndex)
279 summator inp (total, cnt) | cnt == 0 = (total, cnt)
280                               | otherwise = (total', cnt')
281   where
282     cnt' = cnt - 1
283     toSum = splitSubVector cnt' inp
284     total' = foldl (+) total toSum -- todo tree summation to reduce critical path
285
286
287 --divider : takes a state as input and produces a state
288 divider :: (Scalar, Scalar, DivScalar, Value) -> (Scalar, Scalar, DivScalar, Value)
289 divider (a, b, p, cnt) | cnt == 0 = (a, b, p', cnt)
290                               | otherwise = (a', b, p', cnt')
291   where
292     pa = (pack p) ++# (pack a)
293     pa_s = shift pa 1
294
295     (pb, ab) = split pa_s
296     pi = unpack pb
297     ai = unpack ab
298
299     b' = unpack $ low ++# pack b
300     --b' = (fromInteger . toInteger) b
301
302     p' :: DivScalar
303     p' | p >= 0 && (cnt == 0) = pi
304         | p >= 0             = pi - b'
305         | otherwise          = pi + b'
306     a' | p' >= 0 = ai `setBit` 0
307         | otherwise = ai `clearBit` 0
308     cnt' = cnt - 1
309
310
311 --Helper functions: Multiplexers, type conversion
312
313 selReg :: Identifier -> Mem -> Value
314 selReg RPC regs = pc regs
315 selReg RAD1 regs = (fromInteger . toInteger) $ addrR1 regs
316 selReg RAD2 regs = (fromInteger . toInteger) $ addrR2 regs
317 selReg RAD3 regs = (fromInteger . toInteger) $ addrR3 regs
318 selReg RAD4 regs = (fromInteger . toInteger) $ addrR4 regs
319
320 --TODO
321 splitSubVector :: (Num a, Eq a) => a -> SubVect -> SubSubVect
322 splitSubVector 24 xs = selectI d95 d1 xs
323 splitSubVector 23 xs = selectI d91 d1 xs
324 splitSubVector 22 xs = selectI d87 d1 xs
325 splitSubVector 21 xs = selectI d83 d1 xs
326 splitSubVector 20 xs = selectI d79 d1 xs
327 splitSubVector 19 xs = selectI d75 d1 xs
328 splitSubVector 18 xs = selectI d71 d1 xs
329 splitSubVector 17 xs = selectI d67 d1 xs
330 splitSubVector 16 xs = selectI d63 d1 xs
331 splitSubVector 15 xs = selectI d59 d1 xs
332 splitSubVector 14 xs = selectI d55 d1 xs
333 splitSubVector 13 xs = selectI d51 d1 xs
334 splitSubVector 12 xs = selectI d47 d1 xs
335 splitSubVector 11 xs = selectI d43 d1 xs
336 splitSubVector 10 xs = selectI d39 d1 xs
337 splitSubVector 9 xs = selectI d35 d1 xs
338 splitSubVector 8 xs = selectI d31 d1 xs
339 splitSubVector 7 xs = selectI d27 d1 xs
340 splitSubVector 6 xs = selectI d23 d1 xs
341 splitSubVector 5 xs = selectI d19 d1 xs
342 splitSubVector 4 xs = selectI d15 d1 xs
343 splitSubVector 3 xs = selectI d11 d1 xs
344 splitSubVector 2 xs = selectI d7 d1 xs
345 splitSubVector 1 xs = selectI d3 d1 xs
346 splitSubVector _ xs = selectI d0 d1 xs
347
348 bound lb ub val | val <= lb = lb

```

```

349         | val >= ub = ub
350         | otherwise = val
351
352
353 --Processor "entity"
354 topEntity = system prog
355
356 maybeBundle :: (Maybe a, Maybe b) -> Maybe (a, b)
357 maybeBundle ((Just a), (Just b)) = Just (a, b)
358 maybeBundle _ = Nothing
359
360 system :: KnownNat n => Vec n Instruction -> Signal Scalar
361 system instrs = output
362   where
363     (output, rdAddr1, dout1, rdAddr2, dout2, rdAddr3, dout3, ipntr) = mealyB cpu regs
364       (ram1Out, ram2Out, ram3Out, romOut)
365     regs = emptyRegs
366     romOut = asyncRom instrs <$> ipntr
367     ram1Out = blockRam (replicate d60 emptyVec) rdAddr1 dout1
368     _dout2 = liftA maybeBundle $ bundle (addr2, data2)
369     addr2 = fmap (liftA fst) dout2
370     data2 = fmap (liftA pack) $ fmap (liftA snd) dout2
371
372     ram2Out :: Signal (SubVect)
373     ram2Out = unpack <$> blockRamFile d60 "fixb.txt" rdAddr2 _dout2
374
375     ram3Out :: Signal (SubVect)
376     ram3Out = unpack <$> blockRamFile d40000 "fixa.txt" rdAddr3 (signal Nothing)

```


Appendix D: Processor program

```

1  --Program
2  prog = preparation ++ iteration
3  preparation = Reset  r0d0 ++ delta0
4  iteration = mv0 ++ dot0 ++ div0 ++ scale0 ++ add0 ++ normal_r ++ sub0 ++ div1 ++ dot1
      ++ scale1 ++ add1 ++ eoIt
5
6  -----
7  --Subprograms--
8  --Future work:
9  --Calculate A matrix
10 --Calculate B vector
11 --Wfe
12 --Read inputs
13 --Compare minDelta
14 --r0d0 for x!= 0
15 --Recalculate r
16
17 --r0 = d0 = b - A x = b
18 r0d0 = --mv1 ++ sub0 ++
19   Mux ML_Ram2 MR_Ram1 MO_Ram1 MC_DEN AI_Ram12
20   Load 0 0 0 0
21   Nop
22   (concat $ replicate d20 (
23     VecCompute Cp
24     Store
25     Nil))          ++
26   Nil
27
28 --d0 = transp ?0  ?0
29 delta0 = dot1 -- copy to minDelta
30
31 --mv0 d A - q
32 mv0 =
33   Mux ML_Ram3 MR_Ram1 MO_Ram2 MC_DEN AI_Ram3  --Memory mux for MV0
34   Load 0 0 0 20  --{d_0, dc, A_00, q}
35   Nop  --Wait for memory
36   VecCompute Mult  --Multiply & Reset accumulators, auto-
      increment addr
37   replicate d1999 (VecCompute MACC)  ++ --Macc & addr++
38   Store  --Store subvector at q & addr++
39   MIncr (-20) 0 0 0  --Reset d vector
40   Branch BLT RAd3 39999 (-2002)  --Loop untill end address reached
41   Nil
42
43 --dot0 q . d
44 dot0 =
45   Mux ML_Ram2 MR_Ram1 MO_Ram1 MC_DEN AI_Ram12  --
46   Load 0 20 0 20
47   Nop
48   VecCompute Mult  --Multiply & Reset accumulators, auto-
      increment addr
49   replicate d19 (VecCompute MACC)  ++
50   AluCompute Sum
51   --Branch BSUM RAd2 0 0  --Wait (jmp 0) untill Sum is ready
52   Nil
53
54 --div0
55 div0 =
56   Mux ML_Ram2 MR_Ram1 MO_Ram1 MC_DEN AI_Ram12
57   AluCompute Div
58   Nil
59
60 --scale0 alpha d - m
61 scale0 =
62   Mux ML_Const MR_Ram1 MO_Ram1 MC_DEN AI_Ram1
63   Load 0 0 0 20  --Load d, _, _, m
64   Nop
65   (concat $ replicate d20 (

```

```

66     VecCompute Mult
67     Store
68     Nil))                                ++
69     Nil
70
71     --add0  x + m - x'
72 add0 =
73     Mux ML_Ram2 MR_Ram1 MO_Ram1 MC_DEN AI_Ram12
74     Load 40 20 0 40                    --{x, m, dc, x}
75     Nop
76     (concat $ replicate d20 (
77         VecCompute Add
78         Store
79         Nil))                            ++
80     Nil
81
82     --mvl  --TODO Swap with previous for the every iteration thingy
83 odd_r =
84     Mux ML_Ram3 MR_Ram1 MO_Ram2 MC_DEN AI_Ram3
85     Load 40 20 0 20                    --{x, q, A_00, q}
86     Branch NB RAd2 (10) (1)            --Todo Every 40 iterations calculate
87         Ax instead of Am2
87     VecCompute Mult                    --Multiply & Reset accumulators, auto-
88         increment addr
88     replicate d1999 (VecCompute MACC)    ++ --Macc & addr++
89     Store                                --Store subvector at m1 & addr++
90     MIncr (-20) (-20) 0 0              --Reset x and m2
91     Branch BLT RAd3 39999 (-2002)      --Loop untill end address reached
92     Nil
93
94 normal_r =
95     Mux ML_Const MR_Ram2 MO_Ram2 MC_DEN AI_Ram3
96     Load 40 20 0 20                    --{x, q, A_00, q}
97     Nop                                  --Wait for memory
98     (concat $ replicate d20 (
99         VecCompute Mult
100        Store
101        Nil))                            ++
102    Nil
103
104    --sub0  (r b) - q - r' --todo
105 sub0 =
106    Mux ML_Ram2 MR_Ram1 MO_Ram2 MC_DEN AI_Ram12
107    Load 20 0 0 0                        --{m1, r, dc, r}
108    Nop
109    (concat $ replicate d20 (
110        VecCompute Sub
111        Store
112        Nil))                            ++
113    Nil
114
115    --dot1  r r - const
116 dot1 =
117    AluCompute Swap
118    Mux ML_Ram2 MR_Ram2 MO_Ram1 MC_NUM AI_Ram12  --
119    Load 0 0 0 0                          --{r, r, dc, dc}
120    Nop
121    VecCompute Mult                        --Multiply & Reset accumulators, auto-
122        increment addr
122    replicate d19 (VecCompute MACC)        ++
123    AluCompute Sum
124    Branch BSUM RAd2 0 0                  --Wait (jmp 0) untill Sum is ready
125    Nil
126
127    --div1  beta - delta old delta
128 div1 =
129    Mux ML_Ram2 MR_Ram1 MO_Ram1 MC_DEN AI_Ram12
130    AluCompute Div
131    Branch BDIV RAd2 0 0
132    Nil
133
134    --scale1
135 scale1 = scale0

```

```
136
137  --add1
138 add1 =
139  Mux ML_Ram2 MR_Ram1 MO_Ram1 MC_DEN AI_Ram12
140  Load 0 20 0 0                                --{d, m, dc, d}
141  Nop
142  (concat $ replicate d20 (
143    VecCompute Add
144    Store
145    Nil))                                        ++
146  Nil
147
148  --end of iteration
149 eoIt =
150  Branch NB RAd2 (10) (1)                       --Todo Compare delta_n to delta_0, if
151  Nil                                           --Todo otherwise increase iteration
152  count and goto start
153  --end of program
154 eop =
155  --Signal                                     --Signal avalon bus
156  --Wfe                                       --If memory copy has finished
157  Reset                                       --Reset the system
158  Nil
```