Scalable Real-Time Hardware for 2D Empirical Mode Decomposition

Eelko van Breda

Master of Science Thesis

TU Delft Delft University of Technology

Computer Engineering Group

Scalable Real-Time Hardware for 2D Empirical Mode Decomposition

Eelko van Breda

December 9, 2013

For the degree of Master of Science in Embedded Systems
Delft University of Technology

Supervisors:
Prof. dr.ir. Pieter Jonker
Dr.ir. Arjan van Genderen

Committee chairman:
Dr.ir. Stephan Wong

Thesis number:
CE-MS-2013-17

Computer Engineering Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Contents

1 Introduction

For stationary signals the Fourier Transform is an efficient tool for signal and frequency analysis, splitting a complex signal up in a number of sinusoids. In physical phenomena, however, many signals are non-stationary because they have a drift, which can compromises the analysis of the Fourier Transform results, because the drift is also interpreted as a sum of sinusoids and can not be distinguished from the sinusoids making up the original frequency components. To overcome this limitation, Huang *et al.* [11] proposed a data driven filter approach called Empirical Mode Decomposition (EMD). The EMD algorithm aims at building a set of orthogonal basis functions called *Intrinsic Mode Functions* (IMFs) and a residue that represents the original non-stationary component. It will thus split a signal up in a number of natural frequencies that are independent of each other[1] and that do not include any drift or non-stationary component. After subtracting all the found IMFs from the original signal, we recover this non-stationary component. Because this method works without a predefined basis function, like Fourier functions or Wavelets, it is well suited to extract frequency components that have a physical meaning, to study physical phenomena like the analysis of rainfall [19][18], or the analysis of brain waves recorded with EEG[21]. Although this decomposition method was originally developed for one dimensional signals, an extension was made by Nunes et.al[16], to do the same analysis on images. This has led to applications in image compression[14] and illumination correction[17][2]. Figure 1.1 gives an example of an EMD[2] preformed on an image of the Delft university campus.

Figure 1.1: Empirical Mode Decomposition of the Delft University Campus, from left to right we see the original image, followed by three IMFs and a residue.

In this figure we see the original image on the left, which is the luminance component of a color image. To the right of the original image, we see three IMFs, which split the image up in high, medium and low frequencies. On the right side we see, what is called

[1] Independent in the statistical sense.

[2] To be able to have a visual appreciation of this decomposition, the IMFs and the residue are visually enhanced, i.e. stretched, for this image. More information on this stretching will be given in the following chapters.

the residue, which is the drift plus the mean of the original image. If we add up the residue and the three IMFs, we get the original image back.

In our lab EMD has been used for the research on depth extraction from single lens images, called depth from luminance (DfL)[3]. This research has led to the idea of using DfL for the processing of video streams, preferably in a high definition format. The EMD algorithm, however, suffers from excessive computational complexity for large images, and at this moment, no method exists to perform EMD on video streams in real-time. This has led to the question if it would be possible to make a hardware implementation of the EMD algorithm that would be capable of real-time EMD on video streams.

Since this project has been part of a subsidized research with industry, called the ICAF[4] project, which involves a camera manufacturer[5] that makes broadcasting cameras, an additional requirement was requested for the implementation to fit on an FPGA[6]. The research question for this thesis thus became: *Is it possible to implement streaming real-time EMD for image processing on an FPGA?*

It was recognized that such a system should be of a scalable design, to be capable of growing with the ever increasing demands of the broadcast industry as well as the constantly increasing performance and size of FPGAs. A study was made on the current algorithms for EMD image processing as described in chapter 2, which lead to the conclusion that a fundamental step in the EMD process takes the whole image data into account, hampering a scalable solution for this design. In chapter 3 an alternative mathematical solution is proposed, that solves this problem, and enables a parallel and scalable processing approach. Using this alternative a hardware implementation is proposed that exploits these parallel and scalable features as given in chapter 4. To test the behavior of this new architecture and parameterize it, a software simulation was written, as described in chapter 5. This simulator, which is capable of processing EMD on images, was then extensively tested to research its properties and parameters. Some parts of the algorithm were implemented in an FPGA to research the area consumption and performance, as discussed in chapter 6. Given all this information, it is concluded in chapter 7, that this design is still too complex to fit in a currently available FPGA. It is also noted which parts of the design can be improved to reduce area, and how much work this would involve. If the FPGA requirement is lifted, e.g. the system would be implemented in an ASIC a real-time streaming design would theoretically be possible in hardware.

[3]Research by Boris Lenseigne, Delft University of Technology.

[4]ICAF (Image CApture of the Future), part of CATRENE (Cluster for Application and Technology Research in Europe on NanoElectronics).

[5]Grass Valley[9]

[6]FPGA – Field Programmable Gate Array, i.e. programmable hardware.

2 Empirical Mode Decomposition

This section first describes the classical EMD for frequency analysis of 1D signals. In the next section, the extension to a two dimensional EMD, also called Bi-dimensional EMD (BEMD) is given. This BEMD is especially focussed on the analysis of spatial frequencies in 2D images.

2.1 One-dimensional EMD

As stated, an EMD is a data driven decomposition that splits a signal into a number of IMFs and a residue, which should sum up to the original signal. As shown in equation 2.1.

$$S_0 = \sum_{i=1}^{n} c_i + r \tag{2.1}$$

Where S_0=signal, n=number of IMFs, c_i=IMF, r=residue.

The original EMD requires these IMFs to adhere to the following properties:

1. The number of extrema and zero crossings must differ at most by one.
2. There is only one mode of oscillation (*ie.* only one maximum or minimum) between two zero crossings.
3. At any point, the local mean of the signal must be zero.
4. IMFs are orthogonal to each other and as a whole.

The first to rules will cause the maxima to be positive and the minima to be negative, and as such that the oscillation is centered around the zero axis. For the local mean to be zero, means that if you take a window of an arbitrary size big enough to catch at least one oscillation of a signal, and slide it over this signal, the the mean of that window is zero. The orthogonality of the IMFs basically implies that the IMFs form basis functions for the stationary part of the signal.

2.1.1 The Sifting Process

To find the IMFs, an algorithm was created by Huang et. al., called the *sifting process*, which is illustrated here by a flowchart (figure 2.1) followed by an example sift of a signal (figure 2.2 to 2.6).

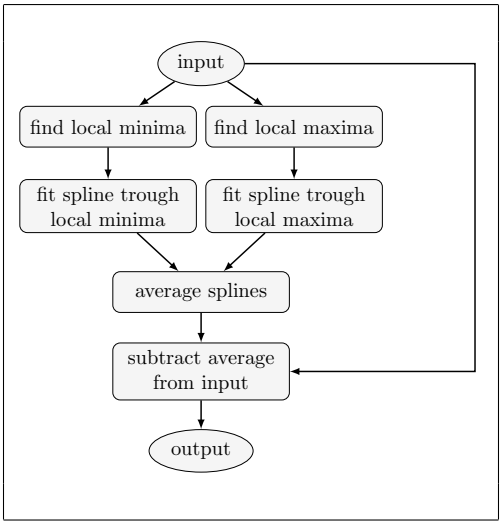


Figure 2.1: Flowchart of the Sifting Process.

First, in the original signal, the local minima and maxima are identified as can be seen in figure 2.2. These local extrema are then used to fit two envelopes, using a cubic spline interpolation method, as shown in figure 2.3. An average envelope is then calculated, which is an approximation of the local mean of the signal. This trend is then subtracted from the original signal, taking with it (a part of) the non-stationary characteristics that were embedded in the original signal. The resulting signal, should be an *IMF*, and satisfying the original properties we want the IMF to have. Equation 2.2 shows the averaging of the envelopes, and equation 2.3 shows the creation of the IMF.

$$m_1 = \frac{1}{2}(Env_{min} + Env_{max}) \tag{2.2}$$

Where Env_{min} and Env_{max} are the minima and maxima envelopes.

$$h_1 = S - m_1 \tag{2.3}$$

Where m_1 is the averaged result from the previous equation that is subtracted from the original signal(S), creating the first (partial[1]) IMF (h_1).

2.1.2 Iterating the sifting process

Because the *sifting process* is not ideal, it can create overshoots, possibly introducing new extrema in the signal. Moreover, it is also possible that the envelope mean differs from the true local mean and that the cubic spline fitting creates large swings at the borders of the signal, which is visible in figure 2.4. We will therefore call an IMF created with this process a *partial IMF*.

[1] see next section

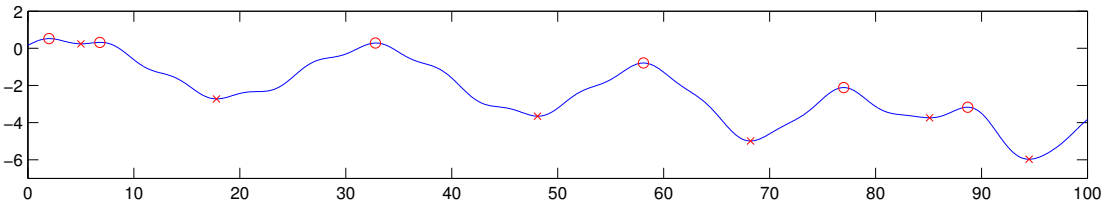


Figure 2.2: Example signal with the local maxima depicted as circles and the local minima depicted as crosses. Note that the signal has a downwards drift.

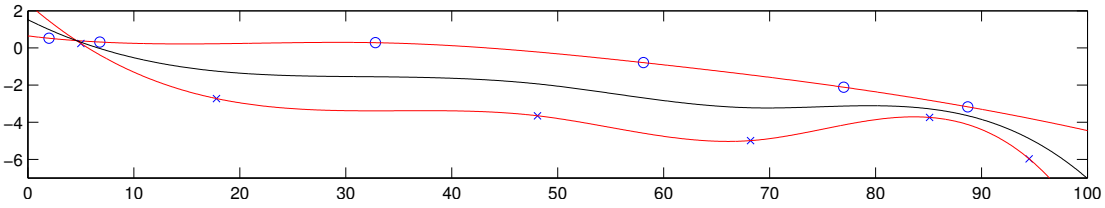


Figure 2.3: The line trough the circels represents the maxima envelope and the line through the crosses the minima envelope. The line in the middle is the average of the two and if we subtract this from the original signal we get a partial IMF.

To overcome these problems, it is necessary to repeat the sifting process until the properties of the IMF are satisfied.

$$\begin{aligned} h_{10} &= S_0 - m_{10} \\ h_{11} &= h_1 - m_{11} \\ &\dots \\ h_{1k} &= h_{1k-1} - m_{1k} \end{aligned} \tag{2.4}$$

Where S_0 is the original signal, m_{1k} is the k^{th} envelope of the first IMF (see equation:2.2) and h_{1k} is the k^{th} (partial)IMF result for the first IMF (see equation:2.3).

To find the iteration at which we converge to an IMF, we need a stop criterium, which would most logically be a test that shows that we found an IMF. This would, however, be too difficult or even impossible because the local mean itself (as given in the definition of the IMF) is ill defined, i.e. there is no definite way to determine a range for locality that would hold for all frequencies. The signal will however not converge completely because the repeating of the sifting does have side effects, like the smoothing of uneven amplitudes. We should therefore eventually decide, when we have come to an optimal result among the sifts. Huang et.al. proposes to look at the convergence of the IMF using the standard deviation (SD) of two consecutive siftings as given in equation 2.5.

$$SD = \sum_{t=0}^{T} \left[\frac{|(h_{1(k-1)}(t) - h_{1k}(t))|^2}{h^2_{1(k-1)}(t)} \right] \tag{2.5}$$

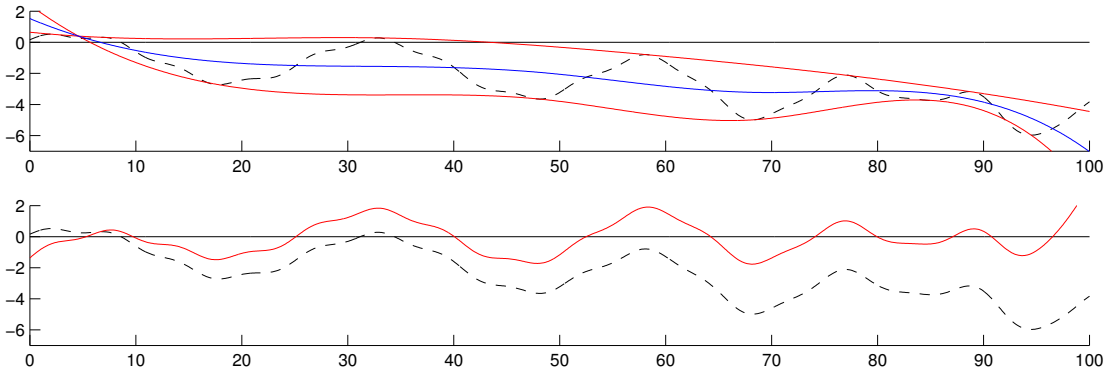


Figure 2.4: In the top graph we see the original signal (dashed), the envelopes and the average signal. In the bottom graph we see the original signal (also dashed) and the found IMF. We can see that this IMF now misses the non-stationary component represented by the average line in the top graph.

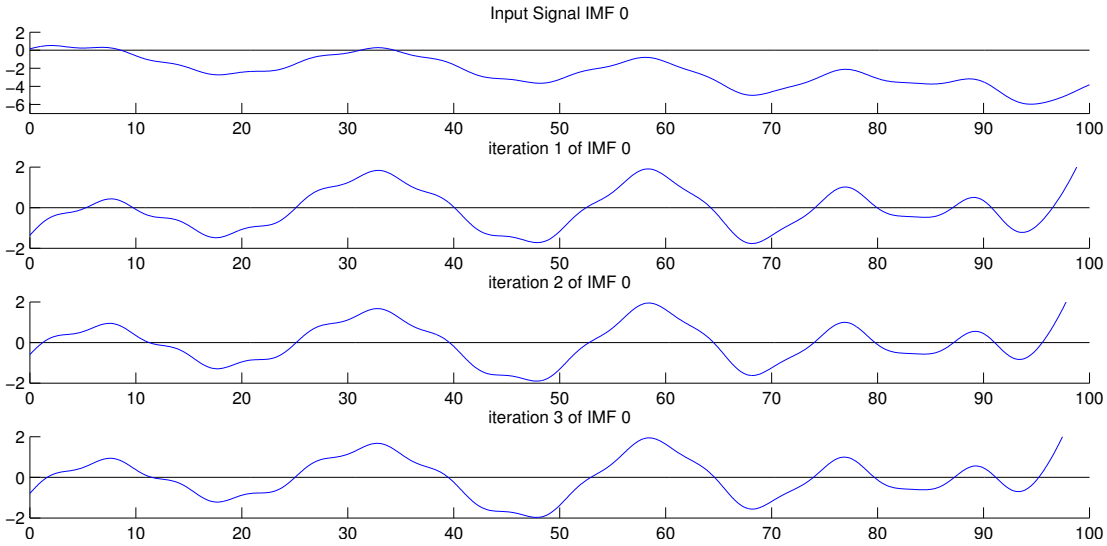


Figure 2.5: The iterative steps of the first IMF. The top graph depicts the signal at the start. The second graph is the result after the first iteration etc. We can see that the first iteration removes the non stationary component almost completely. The second iteration improves on this by lifting the envelope at the sides, and third iteration does not contribute as much as the second. Note that the top graph (original signal) has a larger range on the y axis than the other graphs. The other graphs are *zoomed in* to emphasize their differences.

If the SD is smaller than a threshold value, which was experimentally found to be between 0.2 and 0.3, the *pseudo*[2] IMF has been found. Figure 2.5 shows all the intermediate partial IMFs that were created from the test signal. Basically we see that, each iteration, the local mean becomes closer zero, i.e. all the areas between the line

[2]We can call this a pseudo IMF because we do not have a mathematical test that can prove the strict IMF-ness of a signal. In the original paper of Huang et.al. it is therefore called a pseudo IMF, we will refer to it as an IMF for the rest of this thesis.

and the zero axis will cancel each other out when summed. .

2.1.3 EMD

If the first IMF is found, it can be subtracted from the original signal, leaving a residue. This residue usually contains other oscillatory components of lower frequencies that can also be extracted as IMFs. So *if* the residual signal contains more than one minima and one maxima, new envelopes can be generated and we can keep on extracting IMFs until the residue has no oscillatory mode of its own anymore as shown here:

$$\begin{aligned} c_n &= h_{nk} \\ S_1 &= S_0 - c_1 \\ S_2 &= S_1 - c_2 \\ &\dots \\ r &= S_{n-1} - c_n \end{aligned} \tag{2.6}$$

Where c_n is the n^{th} IMF (which is the last result of the iterative sifting process), S_n is the signal that is left after subtracting the n^{th} IMF and is also called the n^{th} residue. The last residue (that does not contain oscillatory modes) is called *the residue* r.

The residual signal then represents the general trend, or non-stationary component, that was embedded in the original signal. The full EMD of the test signal is shown in figure 2.6 and a flow chart is given in figure 2.7.

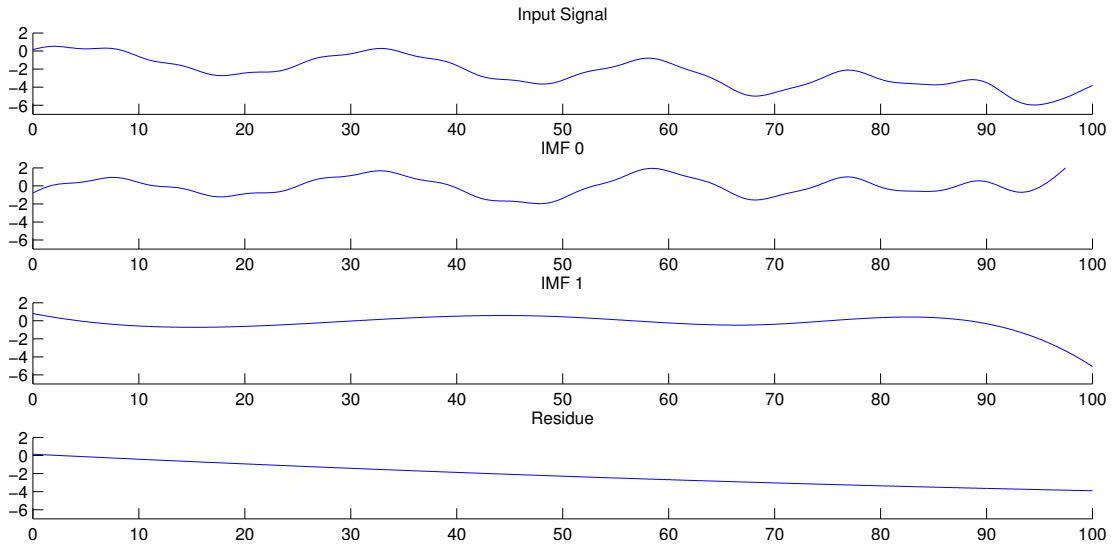


Figure 2.6: The complete EMD of a signal. The top graph depicts the original signal, the second graph is the first IMF holding the high frequency components, the third graph is the second IMF holding the low frequency components and the bottom graph is the residue representing the non-stationary part of the signal.

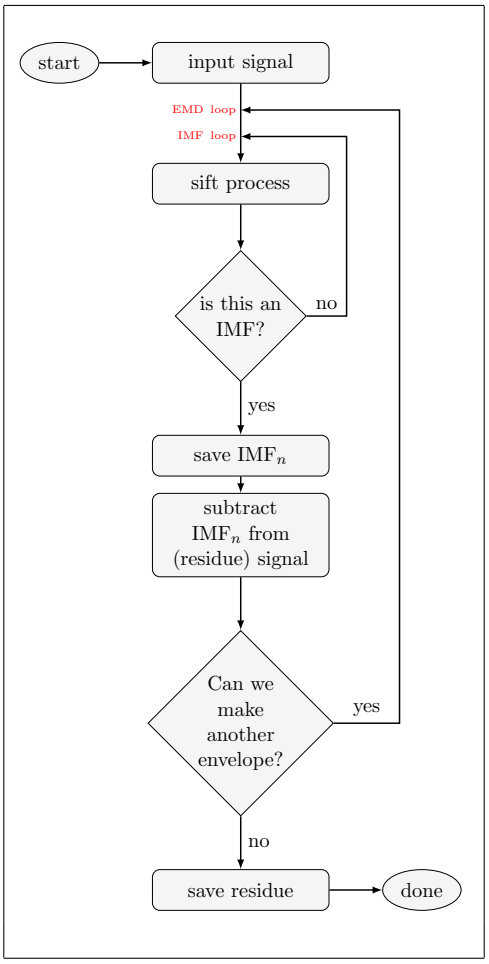


Figure 2.7: Flowchart of the complete EMD process. The inner loop creates an IMF, which is then subtracted from the signal. The remainder is then reprocessed (the outer loop) and IMFs are subtracted until non are left.

2.1.4 Alternative ways of doing the 1D EMD

Given the 1D EMD algorithm of Huang et.al [11], it becomes clear that the two main components that determine the quality of the results, are the way the interpolation of the envelopes is done and the way the stopping condition is calculated. Since no mathematical grounding exists for either of these two, alternative ways of doing the interpolation and calculating the stop condition have been studied by various groups. This especially holds for the two-dimensional extensions discussed in the next section.

2.2 Bidimensional (2D) EMD

In this section we will first discuss the spatial frequencies, which are of interest for image analysis. We will then look at what this means for the extension of the EMD process with regard to extrema, envelopes and stop conditions.

2.2.1 Spatial Frequencies

While the original EMD algorithm was designed for 1D signals, it is possible to extend this same idea to a two-dimensional signal, which in literature is called a Bidimensional EMD (BEMD). In contrast to a one dimensional signal that is changing in one dimension over time, a bidimensional signal has two dimensions in space and no time component. This is the typical domain of image processing and in this field we speak about spatial frequencies, which means, the occurrence of an oscillation in space. With a two dimensional image, we can thus have oscillations in x and y directions. Typical high frequencies in an image are fluctuations in the intensity signal that happen within a relatively small area. For example, figure 2.8 shows an image of a processor cooler which has higher frequencies along the vertical axes than along the horizontal axes. The EMD principle can therefore offer an interesting decomposition over the natural oscillations within an image. An example of a BEMD is given in figure 2.9. Extending the original EMD algorithm for *two spacial dimensions*, is not trivial however, as we shall see in the next section.

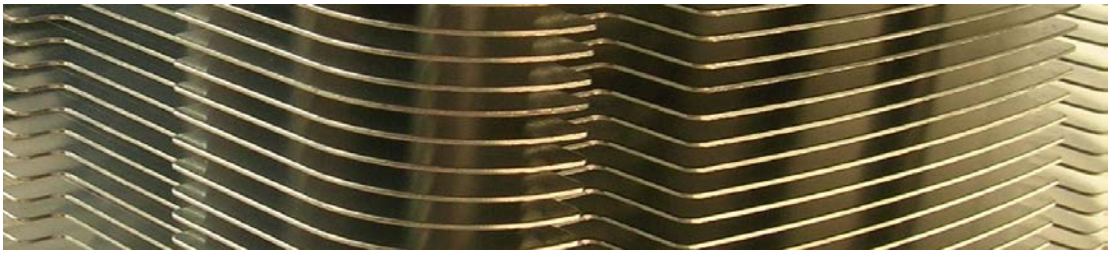


Figure 2.8: Vertical spacial frequencies in a photo of processor cooling fins.

Figure 2.9: BEMD of Lena. On the left we see the original image, followed by three IMFs and a residue. Note that the IMFs and the residue are contrast stretched to make them clearly visible. Underneath the images the minimal and maximal pixel value are depicted. Note also that the average of the IMFs is close to zero.

2.2.2 2D IMF

Since the original EMD process is based on the definition of an IMF in one dimension, we must first define the criteria for a two dimensional IMF. These 2D criteria can be built directly on top of the 1D criteria from section 2.1 The definition of a two dimensional IMF as proposed by Damerval et.al. [7] is quoted below and it will be the basis for our two dimensional EMD.

An image is a bidimensional IMF, if it has a zero mean, if the maxima are positive and the minima are negative and if the number of maxima equals the number of minima.

Note that the neither orthogonality criterium, nor the *local* mean, as we saw them in the 1D defenition, is part of the definition Damerval et. al. proposes. Furthermore, the number of maxima and the number of minima in a two dimensional IMF will not always be exactly equal to each other because in the 2D case we can not exactly define a rule for the relation between minima and maxima. Figure 2.10 shows such a non-trivial situation. Is the elevated ring in this figure a local max? If so, where is it located? If every pixel in the ring is a local max, then what happens to the relation between local maxima and minima? There does not exist a satisfying *correct* solution for this problem and, although the way we define a local extremum in practice will effect the result, the iterative nature of the sifting process allows these errors to be mostly eliminated.

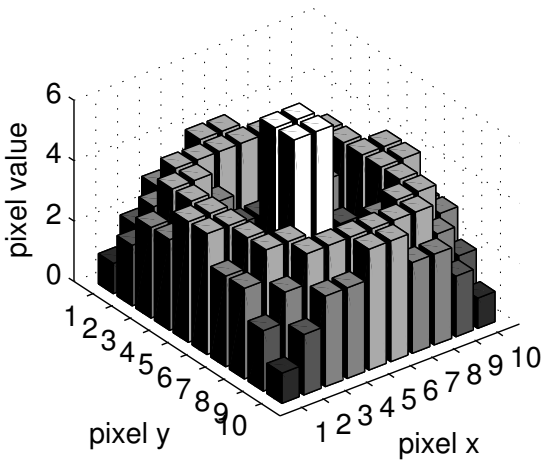


Figure 2.10: Where is the local extreme on the ring surrounding the center?

2.2.3 2D local extrema

A direct result from the definition of the 2D-IMF, is the question of how to define the local extrema in two dimensions. A 2D local extremum in the smallest sense is a pixel that has a higher (or lower) value than all its neighbors, using an 8-adjacency $N_8(p)$ neighborhood. This idea works fine for natural signals with a high enough level of quantization, so that no flat planes exist in the tops or valleys of the extrema, as depicted in figure 2.10 and 2.11. But if the level of quantization is low, or if saturation occurred, this definition will not be practical. If this happens, a number of extrema might not be detected. It is arguable, as mentioned above, to say that the iterative fashion in which the IMFs are found solves that problem, because after each subtraction of a partial IMF, which is a non flat surface, the orientation of flat surfaces will change and so new local extrema are formed.

To find the local extrema there are basically two possibilities.

1. Use data with a high quantization level, for instance high dynamic range (HDR) images, and require high precision in the intermediate results, for instance using floating point, which enables easy detection of the local extrema, using the simple definition.

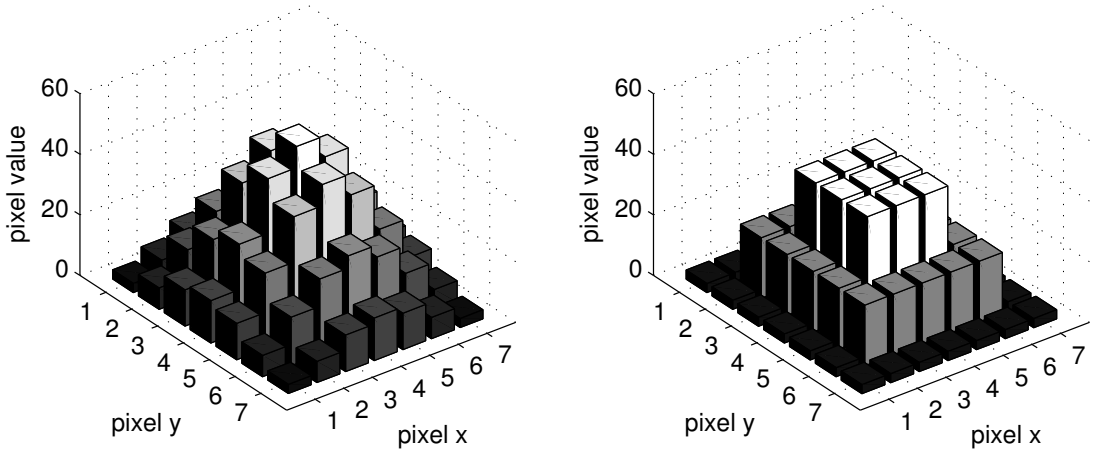


Figure 2.11: Pointy local max and flat local max

2. Have a more elaborate way of finding the local extrema that are lager than 1 pixel, for instance using mathematical morphology like Nunes [16] and Vincent [20], posing less requirements on the data and the intermediate results.

2.2.4 Building the 2D envelopes

Since Huang's original EMD algorithm uses cubic spline interpolation in 1D, the most popular extension is to use thin-plate splines for 2D, as demonstrated in the work of Linderhed [14] and Nunes [16].

In table 2.1 a number of studies is given that have researched BEMD using different kinds of approaches for estimating the envelopes.

Study	Priciple used
Damerval [7]	Delaunay + cubic interpolation of the triangles
Linderhed [14]	Spline interpolation
Nunes [16]	Morphological Operators
Bhuiyan [3]	Order-Statistics Filters
Xu [22]	Finite elements method
Liu [15]	1D EMDs and tensor products
Bhagavatula [2]	Data vectorization

Table 2.1: BEMD studies

Most of these BEMD algorithms solve the interpolation of the 2D envelopes based on Radial Basis Functions (RBF) for which the most straight forward example now follows. An RBF is a function that expresses a value based only on the magnitude of a radius and is therefore isotropic[3] .

$$\phi(||(\delta x, \delta y)||) = \phi(r) \tag{2.7}$$

In table 2.2 a small list of common RBFs used for 2D interpolation is given.

Interpolation using the RBFs is done by expressing each pixel as a linear combination of the RBFs of the known points with their appropriate weights. Say that we have a

[3]i.e. the size of the filter is only dependent on the length of the vector defined by δx and δy.

RBF	$\phi(r)$
Linear	$\phi(r) = r$
Thin plate spline	$\phi(r) = r^2 logr$
Gaussian	$\phi(r) = e^{-\frac{r^2}{2\sigma^2}}$

Table 2.2: Common radial basis functions

sparse set of points $(\xi_i, I(\xi_i))$, where ξ_i are the coordinates of the points and $I(\xi_i))$ are the values of these points. We can then state that the interpolated value of any point ξ is the linear combination of the RBFs and their appropriate weights λ_i, mathematically denoted as:

$$U(\xi) = \sum_{i=1}^{n} \lambda_i \phi(||\xi - \xi_i||) \tag{2.8}$$

To find these weights, we can define all the known relations as:

$$\sum_{i=1}^{n} \lambda_i \phi(||\xi_j - \xi_i||) = I(\xi_i) \tag{2.9}$$

Which can also be written as a matrix:

$$\begin{bmatrix} \phi(||\xi_1 - \xi_1||) & \phi(||\xi_2 - \xi_1||) & \cdots & \phi(||\xi_n - \xi_1||) \\ \phi(||\xi_1 - \xi_2||) & \phi(||\xi_2 - \xi_2||) & \cdots & \phi(||\xi_n - \xi_2||) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(||\xi_1 - \xi_n||) & \phi(||\xi_2 - \xi_n||) & \cdots & \phi(||\xi_n - \xi_n||) \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{bmatrix} = \begin{bmatrix} I(\xi_1) \\ I(\xi_2) \\ \vdots \\ I(\xi_n) \end{bmatrix} \tag{2.10}$$

To find the appropriate weights we will thus have to solve this distance matrix and then for each pixel calculate the value based on all the contributions as given by equation 2.8. Since the number of entries in the matrix for one envelope is equal to the number of minima (or maxima), this will be very complex for large images. In this case the computational complexity of this calculation becomes problematic for a streaming video application, since it involves the inversion of a linear system, requiring $O(Q^3)$ calculations per frame, where Q is the total number of extrema. If we for instance have an image of VGA resolution (640x480) with high frequency content, for which 10% of the data is a local min, we obtain 640x480/10 = 30720 local minima. This would mean that we would have to solve a matrix of size 30720x30720, which is unfeasible for a streaming application at a frame rate of 25 frames per second.

2.2.5 The stop condition for 2D IMFs

As in the 1D case we can use the standard deviation rule as given by equation 2.5. However, some groups like Damerval et al. [7] use a fixed number of iterations, because images seem to be quite stable in the number of IMFs found. It must be noted, that the number of IMFs is dependent on the number of frequencies in the image, so that larger images usually have a larger frequency range and so can possibly contain more IMFs.

2.3 Facilitating a Streaming Solution

As stated in the original paper from Huang et.al. [11], the EMD is not mathematically grounded and is defined by the algorithm producing it. We saw that several flavors of this algorithm exist for 2D, which lead to different results, following from the way the extrema are defined and interpolated, as well as the amount of iterations an IMF is sifted. However, none of the above mentions algorithms is suited for a streaming implementation, because they need to take the whole image data in to account before they can start building the envelopes for the IMFs. To facilitate a scalable streaming solution, we will need a way of processing that localizes the envelope generation to a subset of the image, reducing computational complexity and enabling pipelining. In the next chapter we will investigate if we can change the mathematical structure behind the interpolation step to support such a streaming solution.

3 Streaming BEMD

The main problem with BEMD[1] algorithms is the fitting of a plane though a point cloud, which is usually done by a spline interpolation technique that takes the whole image into account. Such a solution is not scalable and a streaming solution is not feasible for the sizes of images used in modern day imaging hardware in High Definition Television. A mathematical technique called Normalized Convolution (NC), first proposed by Knutsson et. al. [13], enables a interpolation like technique that can localize the interpolation operation. Using this technique we can do a BEMD that leads to satisfying results for the purposes of our application, while enabling a streaming solution. Furthermore, using NC for interpolation offers a scalable solution where the quantity of hardware resources relates to the quality of the result.

3.1 Introduction

In our application, a flavor of BEMD is used for the purpose of decomposing images for further image processing. Processing is done on images from a camera system containing a 1920x1200, 14 bit prototype High Dynamic Range (HDR) image sensor. At the start of this project, a Matlab algorithm was used to calculate the BEMD components. This algorithm typically took a couple of hours to calculate the BEMD for a single image. This slow processing sparked the need for a faster implementation that would preferably perform the BEMD in real-time for video processing in HDTV applications. This chapter describes how we can do the BEMD in such a way that it facilitates streaming processing. Basically, the resulting algorithm is inspired on empirical mode decomposition but is not strictly an EMD. It is a data driven filter bank that produces IMF like components and a non stationary residue, which is suitable for our purposes, i.e. a data driven decomposition of images in frequency and non-stationary components.

[1] Bidimensional Empirical Mode Decomposition, see section 2.2

3.2 Speeding up BEMD

The main problem of the algorithms described in the previous chapter is that they need the complete image data to be taken into account when envelopes are made. For large images, this turns out to be cumbersome. A real time solution should preferably be streaming, which would in turn require some sort of locality in the processing of the data, so that sensible memory and bandwidth requirements can be made. In the following sections we will look at the key parts of the algorithm that require speedup and discuss solutions that enable local processing. The key parts of the algorithm are:

1. Detecting local extrema
2. Building envelopes
3. Averaging envelopes
4. Subtracting envelopes to form the IMF
5. Calculating a stop condition
6. Subtracting the IMFs from the (residual) signal

Because averaging and subtracting is trivial in a streaming manner, given that the streams are synchronized, we will not discuss them further in this chapter. As also mentioned in section 2.2.5, the stop condition does not necessarily need to be calculated, if a fixed number of iterations is chosen that fits the data. Experiments have shown that for the purposes for which our group uses BEMD, this is good enough. We can therefore limit our research on how to make a streaming version of the BEMD; i.e. finding a streaming way to detect the local extrema and fitting the envelopes to these extrema.

3.3 Detecting Local Extrema

As described in section 2.2.3, there are two approaches to the detection of the local extrema. The first is to use a simple detection method requiring an 8-adjacency $N_8(p)$ neighborhood to detect a local extreme, but this requires natural HDR images. A second method uses a more elaborate local extrema search algorithm which is more complex, but less restrictive on the input data. Because of the simple local extrema extraction of the first approach and the fact that we process HDR images, we have chosen to use this method as the basis for our real-time streaming algorithm.

3.4 Building Envelopes

In section 2.2.4, an overview is given of the known methods that are used to calculate the envelopes that fit through the local maxima and minima point clouds. They basically all do an interpolation through the local extrema. Since non of these methods would be ideally suited for a direct implementation in a real-time streaming system, because they take the whole image data into account, we will have to look for a way to localize the problem to a fraction of the image so that a streaming solution can be build on top of that. Section 2.2.4 already describes the basic way of doing the original interpolation and its computational complexity, so we will now explore a method to localize the problem and reduce the complexity. Afterwards we will then look at the possibilities of creating a streaming algorithm from this localized version of the interpolation.

3.4.1 Using Normalized Convolution as a streaming framework for the interpolation of the envelopes

One of the approaches for this interpolation problem is to use normalized convolution (NC), as described in section 3.2 of [13]. In this approach an envelope is calculated in a weighted least square way, where the value and the certainty (distance) of a local extreme in relation to every point in the image is considered. As we shall see, this approach offers a way to do the interpolation using radial basis functions in a local way that enables a streaming solution.

Convolution in two dimensions

To introduce the concept of normalized convolution, first the difinition of a two-dimensional convolution is given. A convolution operation for window $W(s,t)$ of size (m,n) with an image $I(x,y)$ is denoted as:

$$U(x,y) = W * I(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} W(s,t) \cdot I(x-s, y-t) \tag{3.1}$$

Where $a = (m-1)/2$ and $b = (n-1)/2$.

For the simplification of the reasoning about convolutions we can also express the two dimensional coordinates using a spatial operator so that:
ξ is the global spatial coordinate in the image representing (x,y).
χ is the local spatial coordinate in the filter window representing (s,t).
So we can write equation 3.1 for the two-dimensional convolution as:

$$U(\xi) = W * I(\xi) = \sum_{\chi} W(\chi) \cdot I(\xi - \chi) \tag{3.2}$$

Normalized Convolution

The idea of *normalized convolution*[13] is to add a certainty measure to every pixel in the image and an applicability for every filter coordinate, so that:
$c(\xi)$ is a positive scalar representing the certainty of $I(\xi)$
$a(\chi)$ is a positive scalar representing the applicability of $W(\chi)$
for which the normalized convolution is defined as:

$$U_N = \{aW * cI\}_N = \frac{aW * cI}{aW\overline{W} * c} \tag{3.3}$$

Where $\overline{W}$ is the complex conjucate of W

We can now use the normalized convolution to do a interpolation on an image with missing samples, as demonstrated by Knutsson et.al. To do this we set the window filter operation W to a constant where $W = [1, .., 1]$ and express the interpolation as:

$$U_N = \{a * cI\}_N = \frac{a * cI}{a * c} \tag{3.4}$$

If we write out this interpolation we get:

$$U_N(\xi) = \frac{a * c(\xi)I(\xi)}{a * c} = \frac{\sum_\chi a(\chi) \cdot c(\xi - \chi)I(\xi - \chi)}{\sum_\chi a(\chi) \cdot c(\xi - \chi)} \tag{3.5}$$

Normalized Convolution for estimating the envelopes

If we now apply this equation to the envelope estimation problem, we can note the certainty map $c(\xi)$ is one for each coordinate where there exists an extreme and zero everywhere else

$$c(\xi) = \begin{cases} 1 & \text{if } I(\xi) \text{ is an extreme} \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

If we now define the extrema signal in this problem to be:

$$I_{ext}(\xi) = \begin{cases} I(\xi) & \text{if } I(\xi) \text{ is an extreme} \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

We can rewrite equation 3.5 as:

$$U_{env}(\xi) = \frac{\sum_\chi a(\chi) \cdot I_{ext}(\xi - \chi)}{\sum_\chi a(\chi) \cdot c(\xi - \chi)} \tag{3.8}$$

We can further conclude that the outcome of the calculations inside the summations, i.e. $(a(\chi) \cdot I_{ext}(\xi - \chi))$ and $(a(\chi) \cdot c(\xi - \chi))$ will produce zero if $(\xi - \chi)$ is not the location of a local extreme, and so the computational complexity can be reduced by only considering the locations for which the extrema are defined. This makes it possible to rewrite equation in a simpler form, with summations as function of the extrema in stead of the window size. To be able to write the equation 3.8 in terms of the extrema, we will start to express the numerator in terms of extrema. The denominator will then follow in the same fashion since it has the same construction. So the question becomes how to express 3.9 in terms of the extrema?

$$\sum_\chi a(\chi) \cdot I_{ext}(\xi - \chi) \tag{3.9}$$

Now let us assume that the window operator χ is infinitely large, so the convolutions cover all the extrema, and let us define the coordinates of the extrema as ξ_i,where i is the index of the extrema.

Since χ goes from $-\infty$ to ∞, but the summation only produces an other addition if $\xi - \chi = \xi_i$, we can rewrite χ in terms of ξ_i:

$$\chi = \xi - \xi_i \tag{3.10}$$

So the operation inside the summation becomes:

$$a(\xi - \xi_i) \cdot I(\xi_i) \tag{3.11}$$

Since this operation only produces non-zero values for the known points ξ_i, we only have to iterate over i, resulting in:

$$\sum_i a(\xi - \xi_i) \cdot I(\xi_i) \tag{3.12}$$

If we now do the same thing for the denominator we can write equation 3.8 as:

$$U_{env}(\xi) = \frac{\sum_i a(\xi - \xi_i) \cdot I(\xi_i)}{\sum_i a(\xi - \xi_i) \cdot c(\xi_i)} \tag{3.13}$$

Where i is the index over the extrema and ξ_i is the locations of an extreme.

This leaves us with the question of the applicability function. We will use a Gaussian Radial Basis Function for the applicability function when estimating the IMF envelopes because of isotropic and smoothness properties. It is similar in shape to the applicability function used by Kenneth Andersson et. al.[1].
The Gaussian RBF is given by

$$\phi(r) = e^{-\frac{r^2}{2\sigma^2}} \tag{3.14}$$

Where r is the distance and σ is the size operator

3.4.2 RBF Filter Size

For the envelope estimation to work using the normalized convolution technique, we need the size of the gaussion curve (defined by σ) to be matched to the average distance of the local extrema. Too big, and the envelope will not follow the extrema, and too small, the envelope will not be smooth, as depicted in figure 3.1.

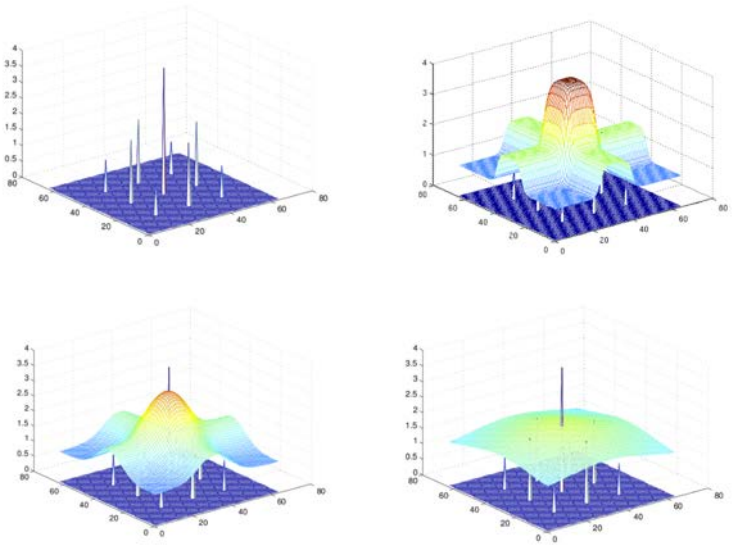


Figure 3.1: Pseudo interpolation using normalized convolution, data domain is (64x64), applicability function is (128x128) with $\sigma = 4$ (top right), $\sigma = 8$ (bottom left), $\sigma = 16$ (bottom right).

Research outside the scope of this thesis showed that the average distances of the Delaunay triangulation edges from the extrema would work well for the purpose of this EMD. A complete Delaunay triangulation of all the extrema would however not be preferred for a streaming hardware solution. Further research[2] showed that this value can be estimated using the average length of the three nearest neighbors, looking in just one direction, which would be suitable for a streaming solution. We therefore decided that a streaming solution for the average distance of the three nearest neighbors of each extrema would have to be found.

[2]Boris Lenseigne, personal communication.

3.4.3 How Normalized Convolution offers a localized solution

The normalized convolution interpolation equation 3.13 makes it possible to define the envelope locally in stead of globally which was the case in RBF interpolation described in section 2.2.4. This opens the door for calculating the envelopes in a streaming fashion. To illustrate the localized processing solution and the rules that govern the parameters, we will first look at a 1D example.

Imagine we have a one dimensional signal with data points $x_1(1,1)$, $x_2(4,0.4)$ and $x_3(7,1.5)$, and we want to use equation 3.13 to interpolate these points. As depicted in figure 3.2, we can see the convolutions (in the numerator and the denominator) as a multiplication of the applicability function with the impulses of these points and certainties.

In figure 3.2 you can see how each point in the image contributes to the eventual interpolated signal. Here the individual contributions of each point are given by the striped lines and their sums are given by the solid line. The NC does not produce interpolations that always go trough the data points, but this does not pose a problem for creation of the IMFs, due to the iterative nature of finding them. Furthermore, we do not suffer from the border conditions that we see when we use the cubic spline interpolation.

Looking further at how the individual points contribute to the interpolated signal, it seems intuitive that point x_3 does not contribute a lot to the shape of the resulting signal in the range $[0,..,4]$. This suggests that we might as well not consider it until its contribution starts to influence the end result. To illustrate this, figure 3.3 presents the results if we only consider points [x_1 and x_2] or [x_2 and x_3] as shown with the dashed (red and blue rep.) lines. The full interpolation is depicted by the solid (green) line.

The reason why our intuition seems to mach the reality of figure 3.3 is that the NC equation considers the value of each point according to its applicability. Since the applicability function that was used is a Gaussian function, the applicability of the point drops exponentially to zero as we travel away from that point. However, if no other point exists on the interpolated line traveling away from the last considered point, it will keep its influence on this signal, but the values in the numerator and denominator of the NC equation will become very small. It is in fact their ratio that does not change and which defines the value of the interpolation at that point. If we should encounter another point, it will eventually have much more influence on the ratio in the NC equation and so the new signal will drown out the influence of the previous point. As we will see in chapter 6, there are manageable problems calculating the envelope if the numbers become too small for the arithmetic units to handle; e.g. dividing by zero.

Given the previous discussion, we can express the contribution of a single point k to the total contribution of all the points at coordinate ξ as:

$$\omega_k(\xi) = \frac{a(\xi - \xi_k)}{\sum_i a(\xi - \xi_i)} \tag{3.15}$$

Now if we would for instance calculate the influence of point x_3 to the total range of the example we obtain figure 3.4, where it is clearly shown that the contribution of x_3 is fading quickly in the presence of point x_2 at the left side, but it is everlasting at the right side, where no other points are present. As figure 3.4 and table 3.5 show, the contribution of x_3 at the location of x_1 is neglectable.

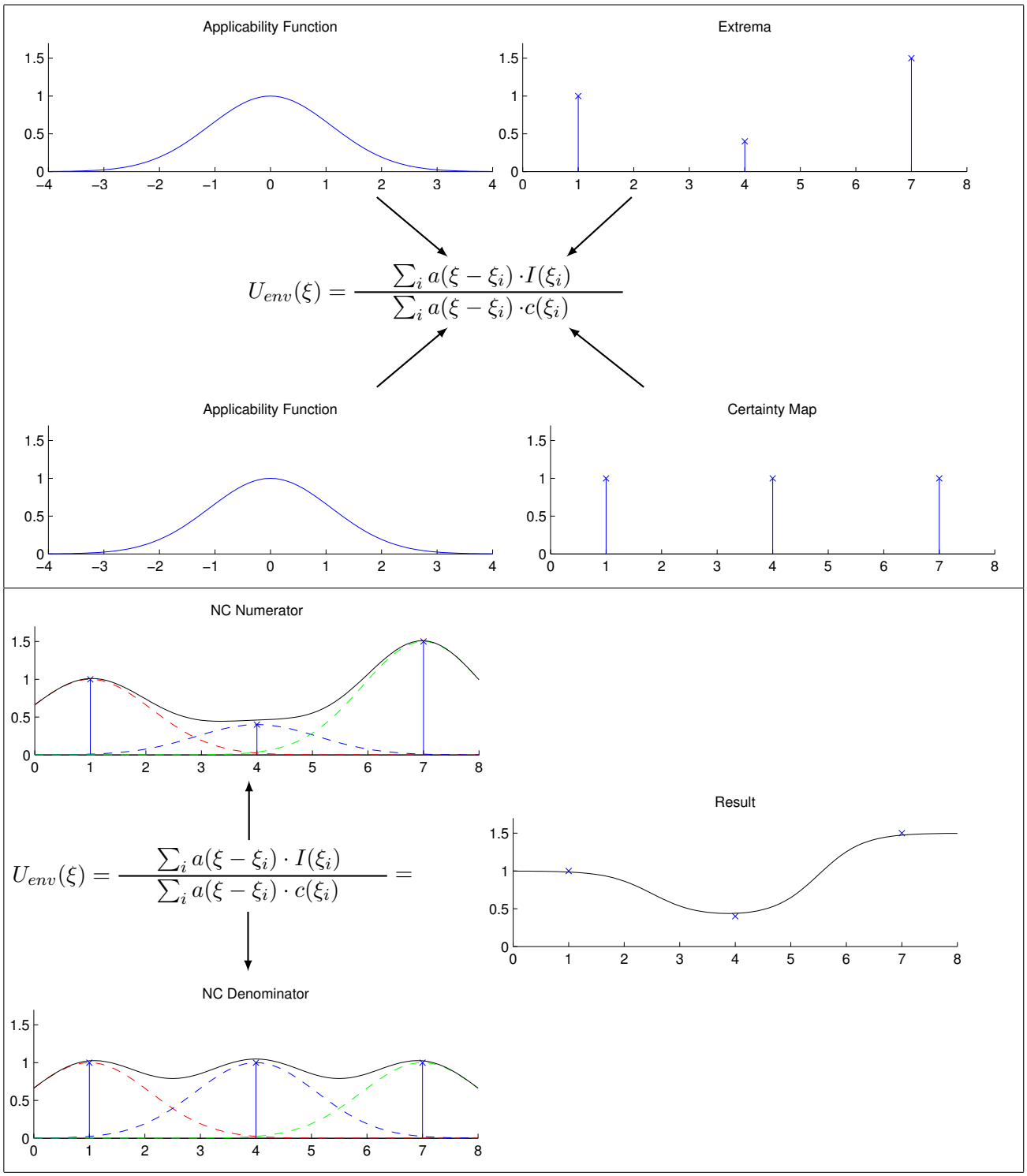


Figure 3.2: The graphical representation of the Normalized Convolution for one dimension

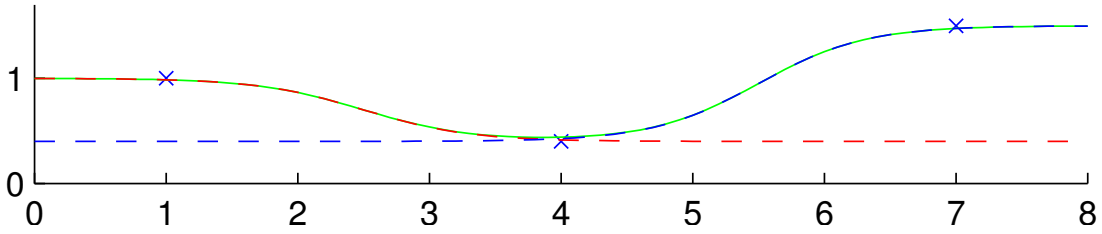


Figure 3.3: Here we see three lines created with normalized convolution. The dashed red line represents the envelope created with $x = 1$ and $x = 4$, the blue dashed line represents the envelope created with $x = 4$ and $x = 7$, and the solid green line represents the envelope created with all three points. This indicates that a local pice wise construction of the envelope is possible.

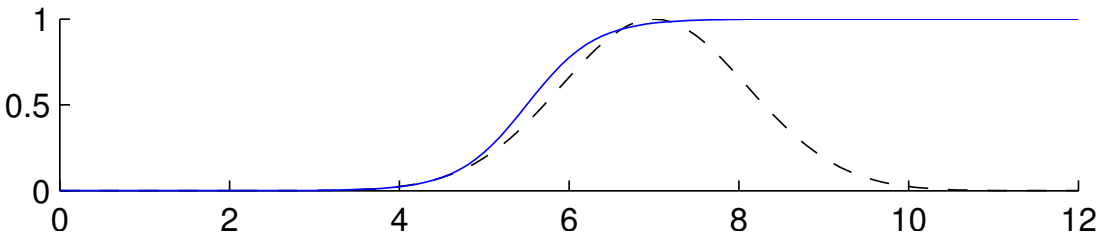


Figure 3.4: If we display the applicability for $x = 7$ get the dashed line representing the Gaussian curve, but if we display the contribution we get the solid line showing that the the extremum at $x = 7$ contributes a lot at the envelope at $x = 12$, but not al all at $x = 2$. This is due to the absence of a competing extrema for $x > 7$. Note that the scale on the horizontal axes has changed with regard to figure 3.3.

For the purpose of making the envelopes for the EMD we use a Gaussian RBF with a fixed sigma for all the points[3]. To illustrate the effects of the size of the RBF, figure 3.6 shows the effect on the confidence values with varying sigmas. The solid lines are the contributions of the separate points, the dashed lines are the original RBFs. As is clearly visible, the size of the RBF and the intermediate distance of the points both have an opposite effect.

If we take the concept of the extrema to a 2 dimensional situation and we superimpose the Gaussian applicability curves, we obtain figure 3.7. Although only the applicability (and not the contribution) is depicted, one can clearly see that if we want to calculate the value of the envelope at the position of the blue square, we will only need to consider the surrounding extrema depicted in blue, and we can thus leave out the extrema depicted in red. This shows that it is possible to calculate the value of the normalized convolution using only a small number of neighbors in a window like operation.

[3]Boris Lenseigne, personal communication

0	1	2	3	4	5	6	7	8	9	10	11	12
2.4267e-09	3.3807e-07	3.824e-05	0.001574	0.023136	0.22414	0.77548	0.97632	0.99797	0.99983	0.99999	1	1

Figure 3.5: The values of the contribution of x_3

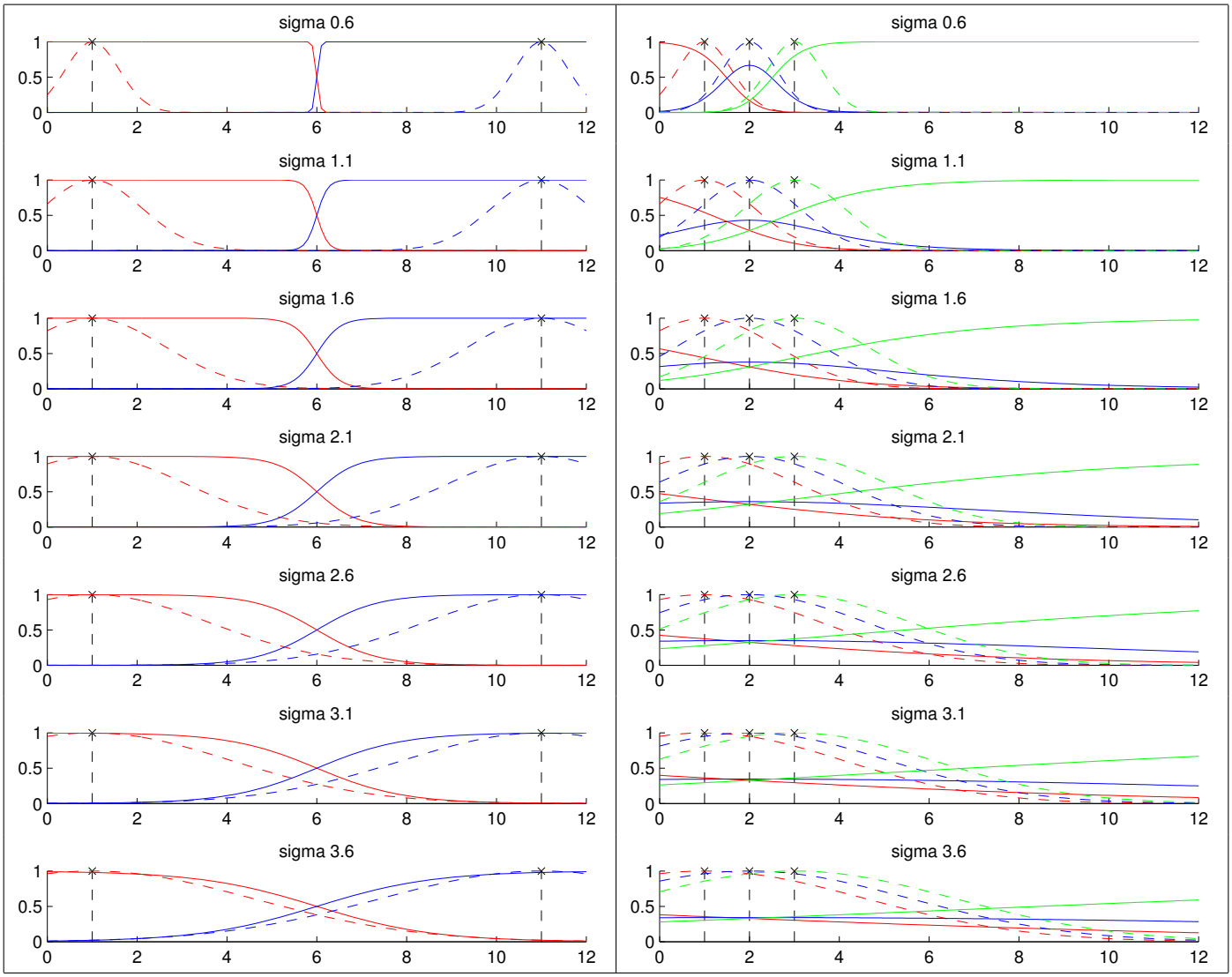


Figure 3.6: The left side shows the behavior of the contribution between two points if we increase the value of sigma. For larger sigmas the contribution of the points is more shared and both points should always be considered. On the right side we see the effect of increasing sigma for three point outside their commonly shared area. Here we see that the closest point always has the most influence, but for large sigmas, the second and even the third closest point has to be considered. Especially for envelope values close to these points.

Minimal Contribution The total contribution of a subset of considered points should preferably generate the same results as considering the whole set, and should at least make a smooth surface. Calculating an identical result with fewer points considered, should therefore ideally have an error that is smaller than the quantization error.

Since a jump in the output is related to the number of discrete levels of the output,

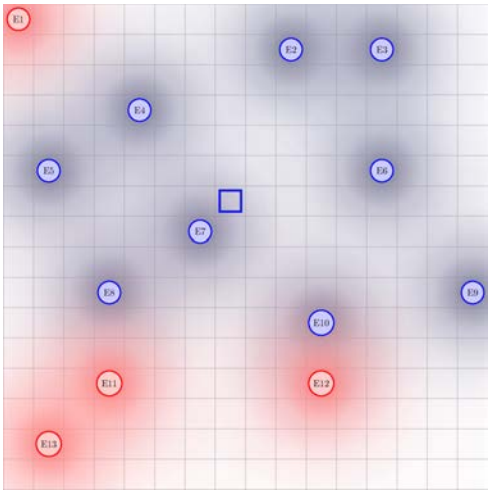


Figure 3.7: Here we see the a bi-dimensional representation of the Normalized Convolution envelope creation. For each of the pixels, a gaussian curve is projected on the scene, in this case showing their applicability. If we want to calculate the value of the envelope for the blue square, we will only need to consider the blue extrema, because the red extrema are too far away, and behind a layer of closer by extrema that are contributing.

let us look what needs to happen to get a difference of one quantum in the result. If we, for instance, use an 8 bit image, we have 256 quanta. The max difference between the lowest and the highest value is 255 and such a difference can influence a step in the end result if the contribution of the added point is larger than $\frac{1}{255}$. To illustrate this, let us take an example, where we have one (or more) points that have the minimal value of 0 with an applicability of 254 and one point with the max value of 255 and an applicability of 1. If we would put this in the NC interpolation equation of 3.13, we will get $\frac{254 \cdot 0}{254} = 0$ if we do not include the point and $\frac{254 \cdot 0 + 1 \cdot 255}{254+1} = 1$ if we include it.

If we assume that rounding takes place at half a quantum, we can say that, if the contribution $\omega_k(\xi)$ is larger than $\frac{1}{2Q-1}$, where Q is the number of quantization levels, it can influence the end result.

What exactly should be defined as a limit for the contribution of a point to be included in the NC interpolation is strongly related to the quality and the computational complexity. Basically we can define the following cases:

1. *Lossless* - The result is the same as with all the points included, but many points can be discarded because their contribution is so small that they will not influence the end result at all.

2. *Qualitative* - The result might contain errors, but the errors are small, generally not more that one quantum, but less points need to be considered than in the lossless case.

3. *Speed* - The result contains larger than single quantum errors, but the speed of the calculation can be dramatically increased

4. *Anyarea* - Based on the concept of an anytime algorithm[4]; the quality of the

[4]An anytime algorithm is an algorithm that produces a result *at any time* during the calculation,

system is based on the hardware resources available expressed in area, always producing the maximal quality for the given recourses. [5]

Note that this anyarea algorithm tries to include as many points as possible for calculation given the parallel recourses available. As a concept this is very scalable, in that it will allow to get better performance from the algorithm by adding resources. This is a very nice property which fits very well with the purpose of our application in HDTV image processing, which is targeted to a visually lossless experience rather than a correct experience. Furthermore, a scalable solution would enjoy the benefits of progress in FPGA growth over the coming years without the need for rewriting the code. We will therefore look closely at the anyarea case.

Which points to include The question of which points to include, in the case of our accelerator, translates in: "What is the minimal amount of points that we need to include to get an acceptable error". This entirely depends on the question of what an acceptable error is. Because the purpose of the accelerator is to create visually lossless results, the best way is to assess this empirically. We can, however, study the principles that govern this question a bit more to get a good appreciation of the problem.

If we would like to have a lossless result, this would mean that the contribution $\omega_k(\xi)$ should be smaller than $\frac{1}{2Q-1}$

$$\omega_k(\xi) < \frac{1}{2Q-1} \tag{3.16}$$

If we state that the region of interest that includes all the extrema that should be considered for the envelope calculation of a given coordinate is called a *Confidence Window*(CW) $\mathbf{J}$, we can see if we can find a equation that defines which points should be included. This confidence window inclusion equation expresses the confidence $\omega(\xi)$ as a function of the confidence window $\mathbf{J}$.

$$\omega_{\mathbf{J}}(\xi) = \frac{\sum_j^{\mathbf{J}} a(\xi - \xi_j)}{\sum_i a(\xi - \xi_i)} > 1 - \frac{1}{2Q-1} \tag{3.17}$$

The problem from the complexity point of view is that this equation loops over all the known points in the dataset for each coordinate that we calculate the envelope for, resulting in a very complex calculation which is not suitable for a realtime solution.

A slight improvement could be to look at the added value of including a point in the confidence window. We can for instance define $\Delta\omega\mathbf{J}_k(\xi)$ in equation 3.18 that calculates the effect of adding an additional point k to the points already considered in the CW

$$\Delta\omega\mathbf{J}_k(\xi) = \frac{a(\xi - \xi_k)}{\sum_j^{\mathbf{J}} a(\xi - \xi_j) + a(\xi - \xi_k)} \tag{3.18}$$

To illustrate the proposed concepts, figure 3.8, and tables 3.1 and 3.2 show an example. Figure 3.8 shows four points with their applicability and their contribution depicted. Table 3.1 shows the values for the contribution at point $x = 4$. And table 3.2 shows the absolute as well as the relative contribution for different window sizes. Note that the relative contribution is the change in contribution due to the addition of a point given an existing CW.

where the results usually improve the longer the algorithm runs.

[5] In this case the term *anytime* would be false because we now express the resource *area* as a scalable unit. Thus the quality of the result is varied by the amount of resources used and processing time in an absolute sense is not varied.

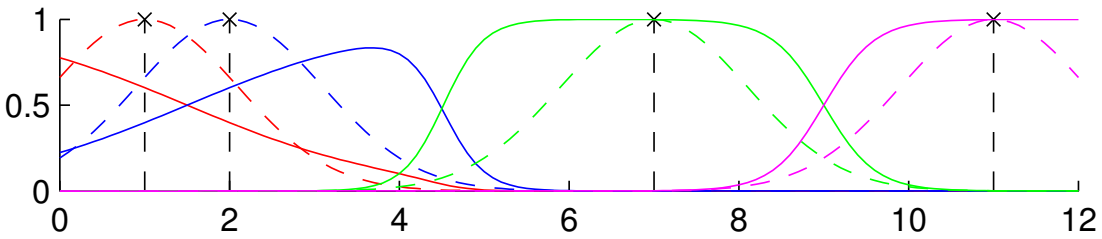


Figure 3.8: Example of 4 points with their applicability (dashed) and their contribution (solid), showing their common relation.

1	2	3	4
0.10107	0.79786	0.10107	6.702e-09

Table 3.1: The values of the contribution of all the points in figure 3.8 for $x = 4$

$\omega J\{x_1, x_2\}(4) =$	8.989297e-01
$\omega J\{x_2, x_3\}(4) =$	8.989297e-01
$\omega J\{x_1, x_2, x_3\}(4) =$	1.000000e+00
$\omega J\{x_1, x_2, x_3, x_4\}(4) =$	1.000000e+00
$\Delta\omega J\{x_2\}_{x_1}(4) =$	1.124341e-01
$\Delta\omega J\{x_2\}_{x_3}(4) =$	1.124341e-01
$\Delta\omega J\{x_1, x_2\}_{x_3}(4) =$	1.010703e-01
$\Delta\omega J\{x_2, x_3\}_{x_1}(4) =$	1.010703e-01
$\Delta\omega J\{x_1, x_2, x_3\}_{x_4}(4) =$	6.701970e-09

Table 3.2: Absolute Contributions and the Delta contributions of **J**

As stated before, these equations can supply us with a proper numerical description of which points to include, but calculating so would considerably increase the computational complexity of the system. So what other ways could be explored to estimate the correct CW?

Fixed CW size A fixed region of interest (ROI) for deciding if a point should be included in the CW or not, would be a very simple way of dealing with this problem. Because of the data driven nature of the algorithm however, this fixed ROI would have a largely varying number of extrema which would not produce a fixed quality for the result. Basically we want a variable size ROI that adjust itself to the density of the extrema population. This would be optimal for hardware purposes if it would always have the same number of extrema which brings us to the next option.

Fixed number of points in CW As was shown in figure 3.6, the distance between points and the size of their Gaussian curves are each others inverses with respect to the contribution. Because we know that the size of the Gaussian curves are dependent on the average distance between the points (the larger the average distance, the larger the Gaussian curves), it follows that a decrease in point density will lead to an increase in the area of the CW. That suggests that roughly the same amount of neighbors can be considered regardless of the density of the local extrema points, i.e. regardless of the data.

The most straight forward way to implement this is to take a ROI, centered at the coordinate of interest, that has a fixed size that is related to the density of the point cloud (or the size of the Gaussians). However, this would still mean that we would need to find which extrema fall into the ROI of the CW for that coordinate.

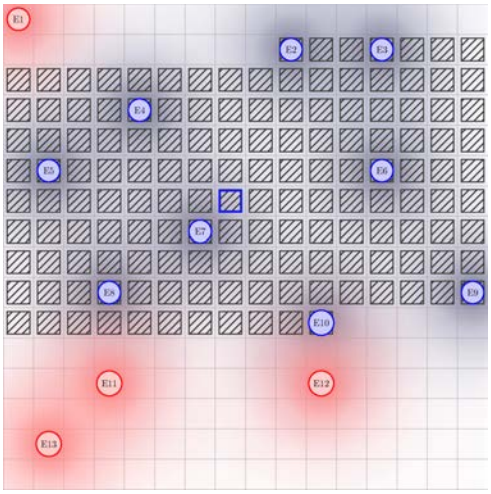


Figure 3.9: This is Figure 3.7 with the streaming confidence window superimposed upon it. The streaming confidence window will slide over the image with the same speed as the pixel for which we calculate the envelope (blue square). The extrema contributions for all the extrema in the confidence window can be calculated by separate *Workers* in parallel.

The Streaming Confidence Window We know that in our system the image data is streamed into the system pixel by pixel, line by line. This means that if we discover the extrema in a streaming fashion, we get a stream of extrema coordinates. Given the idea of a fixed number of extrema in the CW, we can envision a rectangular CW the width of the image, as depicted in figure 3.9. For such a window we can calculate the contributions to it's central line of coordinates. We could then add new extrema to the top of the list of extrema that contribute to future calculations and remove extrema from the bottom that do not contribute anymore in a (First In First Out) FIFO manner and effectively create a streaming solution. A discussion on the size of the CW is given in section 6.3.1, where we look at the effect of different sizes for the CW.

Anyarea The idea of using a FIFO to represent the currently active extrema points that contribute to a streaming calculation of the envelope is very powerful. The contributions of each of these extrema can be calculated independently offering a parallel solution. If we assume that the work for calculating the contribution of one extrema can be done by a set of resources we call a *Worker*, the FIFO will be nothing else than a set of workers taking turns in processing points in the CW. If we now grow the size of the FIFO, i.e. we add *Workers* to the pool of actively processing hardware, and we grow the size of the CW and improve the results. This would theoretically even be possible in a dynamic manner at run-time. The concept of *Workers* working in a pool as parallel elements will be called the *Worker Model* in the remainder of this thesis.

3.5 Conclusion

In this chapter the parts of the BEMD algorithm that need to change to enable a streaming real-time solution were identified. The main bottle neck being the interpolation step in creating the envelopes. As an alternative, the principle of the Normalized Convolution was discussed, which has the ability to do an interpolation on a point cloud with irregularly spaced points. Consequently, this technique is able to limit this interpolation to a fraction of the whole image which enables a localized operation. Next, the rules governing the size of the locality as well as ways to decide which points should be included were discussed. As a result we found that a fixed number of points considered in a streaming FIFO manner, would probably serve the goal of streaming hardware best. Finally, this concept opens the door for a scalable solution using *Workers* as processing elements in the FIFO, which can be added or removed to change the quality of normalized convolution. In this way, an alternative is offered for processing BEMDs, that facilitates streaming localized processing, where the amount of resources used, scales directly to the quality of the result.

4 Streaming BEMD Architecture

4.1 Introduction

The purpose of this research is to see if it is possible to make a streaming version of a BEMD like algorithm for processing video sequences. With the information from chapter 2 and 3 we should be able to design a hardware architecture that can handle and process a stream of video images. Most of the implementation questions that arise are straight forward, except for the way we calculate the envelope, as extensively described in chapter 3, and the way we calculate the filter size as noted in section 3.4.2. This chapter gives a modular decomposition in a top down fashion, after which, for each of these modules a data- and control path description is given

4.2 Modular Decomposition

4.2.1 Top Level

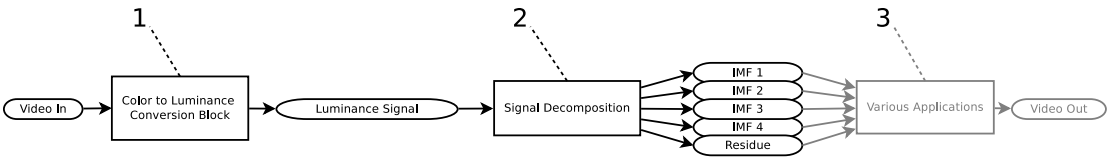


Figure 4.1: EMD streaming process

At the top level of the streaming EMD we have a streaming video source that needs to be processed and that will have to be split up in several streaming IMFs. The general structure of this concept is depicted in figure 4.1. The incoming video stream is a pixel stream (one pixel each pixel clock tick), where the pixels are ordered in a raster scan fashion; i.e. from top left, row by row, to bottom right. Furthermore, we ant to keep the order in this stream so that buffers will generally be of the FIFO type. Since the EMD in our application is based on the luminance representation of the color data, we first convert the *input signal to a luminance signal (1)*, as described in section 4.3.1.

Then the luminance signal is streamed to the *Signal Decomposition Block (2)*, which is further explained in section 4.2.2.

4.2.2 Signal Decomposition Block

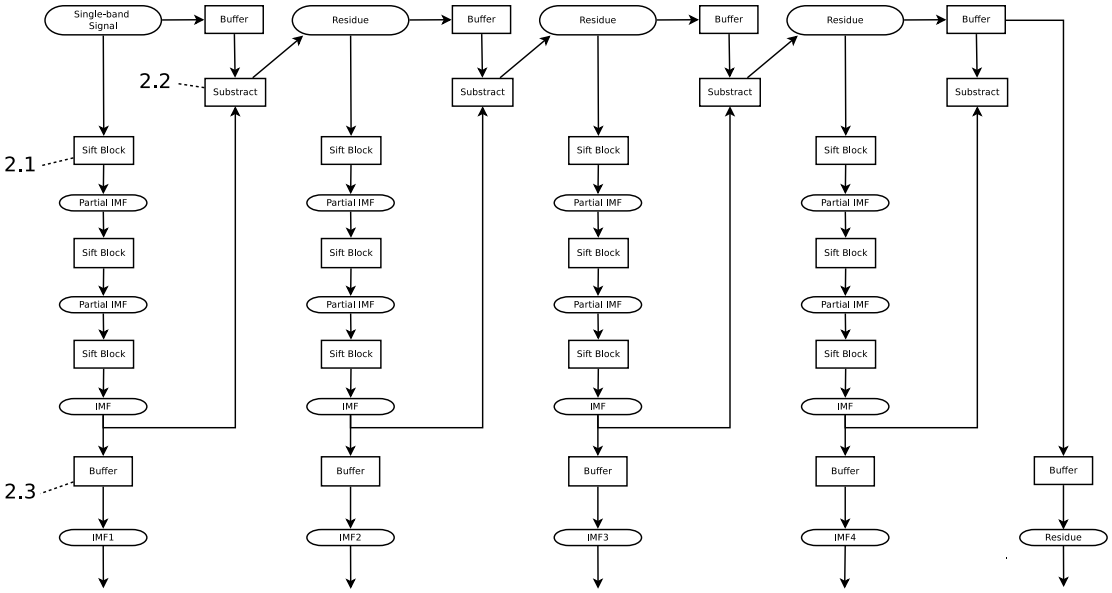


Figure 4.2: Signal Decomposition

The solution for the signal decomposition is split up in a fixed number of sifts. As discussed in section 2.2.5, we can vary the number of sift blocks per IMF as well as the number of IMFs to tune it to a desired end result. Figure 4.2 shows a possible design for creating four IMFs with 3 sifts per IMF. A flexible solution can be realized by designing for the worst case and then use as much hardware as needed. Since our application has a fixed resolution, this can be accomplished relatively easily. This level of the design only consists of three unique processing blocks, replicated a number of times in a pipelined fashion to enable a streaming solution. This way of pipelining the process is basically a form of loop-unrolling. To be able to produce the results in a synchronized manner, a number of *Delay Buffers (block 2.3)* is needed to delay (intermediate) results. More details on the delay buffers is given in section 4.3.10. Furthermore, a *streaming adder block (2.2)* as described in section 4.3.2 is used to subtract the IMFs from the (intermediate) signal. The *Sift Block (2.1)* that is responsible of one sift in the EMD sifting process (as was described in section 2.1.1) is further decomposed in section 4.2.3

4.2.3 Sift Block

This paragraph discusses figure 4.3. The input to a *Sift Block* is either an image- or a (partial) IMF stream. From this signal we extract the local minima and maxima into two streams using a *Local Extrema Extraction Block (2.1.1)*, as further described in section 4.3.3. Through each local extrema point cloud, we then fit an envelope using a *Streaming Envelope Generator(2.1.2)*, as further decomposed in section 4.2.4, from which we then calculate the average using an *Averaging Block*(block 2.1.3; section 4.3.4). This average is then subtracted from the original input stream using a *streaming adder*

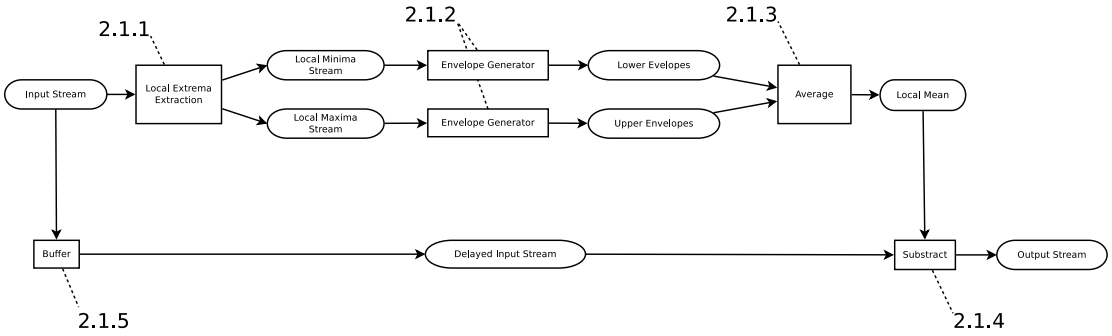


Figure 4.3: Sift Block

block (2.1.4), identical to the streaming adder used before and described in section 4.3.2. To compensate for the delay induced by blocks(2.1.1), (2.1.2) and (2.1.3) we need to add a delay to the input stream that enters the streaming adder block. This *Buffer(block 2.1.5)* is identical to the previously described *Delay Buffers (block 2.3)* which are explained in section 4.3.10.

4.2.4 Envelope generation

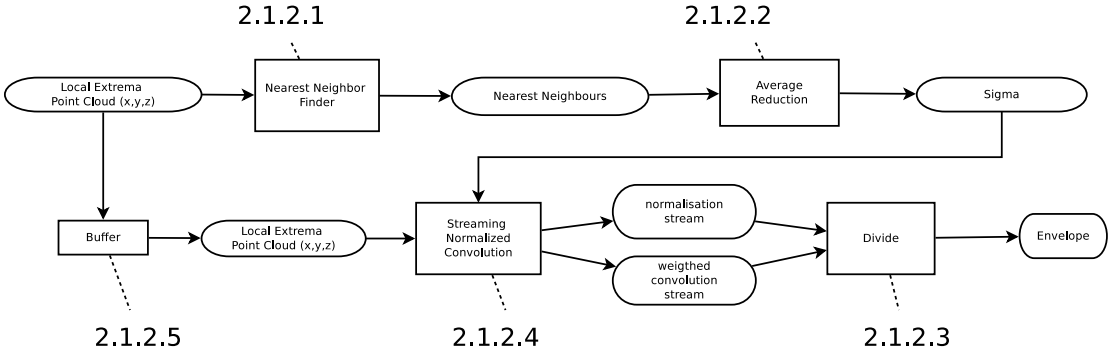


Figure 4.4: Envelope generation

In the EMD algorithm, interpolation is used to create a curved surface through the point cloud. As explained in section 3.4.3 we choose the normalized convolution to do a streaming approximation of the envelopes, using a big Gaussian filter for each local extreme. As depicted in figure 4.4, the input of the envelope generator is a point cloud, consisting of the found local minima or maxima. The size of the filter is first calculated by measuring the average distance between the local extrema using a nearest neighbor algorithm. To do so, we first convert the point cloud to a stream of neighbors (block 2.1.2.1; section 4.3.6), of which we then calculate the average distance (block 2.1.2.2; section 4.3.7). We then use the resulting value (one value for each frame) as an input for our streaming normalized convolution block (2.1.2.4), which produces a weighted result and a normalization factor for each pixel in the resulting image. The streaming normalized convolution block is further described in section 4.2.5. We then normalize the results using a streaming divider block (2.1.2.3) and effectively create the envelope. To synchronize the results of the filter-size generation with the start of the

NC calculation, we theoretically have to delay the stream for a full frame size (block 2.1.2.5). We can again reuse the streaming buffer block as described in section 4.3.10. This delay is a nice candidate to be further optimized by guessing the filter size, for instance based on the previous frame, which will dramatically improve the latency of the system.

4.2.5 Streaming Normalized Convolution

The basic flow diagram of the streaming normalized convolution interpolation is given in figure 4.5. The basic input is the streaming point cloud of local extrema. Because we need an output of interpolated values per pixel coordinate, we need a source of pixel coordinates (block 2.1.2.4.2) to feed into the *Worker Model* as well. The *Worker Model* doing the normalized convolution interpolation (block 2.1.2.4.1), described in section 4.3.8, then produces two streams of values for each active worker, representing the values of the contribution and its normalization factor for each pixel coordinate. A reduction operation, that sums op these values for each pixel coordinate, is implemented by a streaming summation block (2.1.2.4.3) which is further described in section 4.3.9

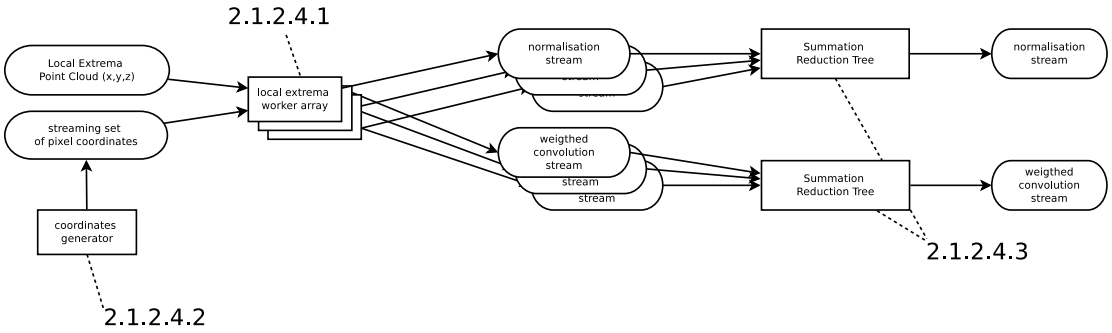


Figure 4.5: Streaming Normalized Convolution Interpolation

4.3 Module Data-path and Control

4.3.1 Color to Luminance Conversion Block

This paragraph is referenced by section 4.2.1

The color to luminance conversion block has the responsibility of extracting the most useful luminance component for the purpose of the decomposition. If the input signal is in YUV mode, we can just take the Y component. If we have an RGB signal, we can use a conversion equation which would typically[8] be:

$$Y = 0.299R + 0.587G + 0.114B \tag{4.1}$$

The calculation can be done using an ALU in a streaming fashion. The three color values enter the system in parallel with a pixel clock of maximal 164 MHz for 1080p60[1]. Lower resolutions have subsequently lower pixel clocks. In our case, for the targeted FPGA platform that can run up to 400MHz, a pipelined multiply and accumulate is the most economical implementation strategy. An other strategy would be to split the stream in to a couple of parallel streams handling consequent pixels in the stream and use dedicated non-pipelined ALUs.

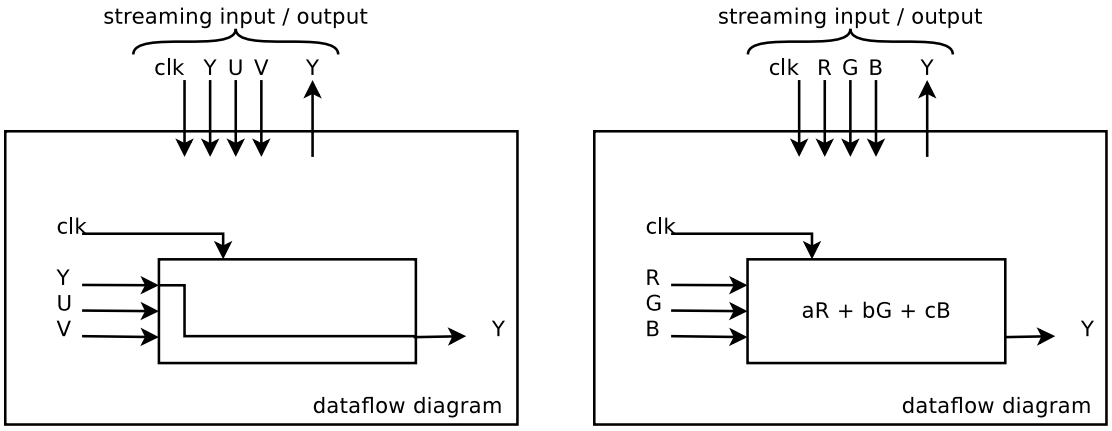


Figure 4.6: Data Path Diagram - Color Luminance Conversion Block

4.3.2 Streaming Adder

This paragraph is referenced by section 4.2.2 and section 4.2.3

The streaming adder block is used to subtract two streams from each other. This ALU shares the same properties as the one described in section 4.3.1 in that it is clocked using the pixel clock and its implementation depends on the ratio between the pixel clock and the system clock. It is however much simpler and it is very probable that a single addition or subtraction can be done in the time span of one pixel clock. If the precision would require an addition that needs more time, a pipelined implementation can be considered. A data flow representation is given in figure 4.7.

[1] 1080p60 is non-interlaced HD television at 60 frames per second

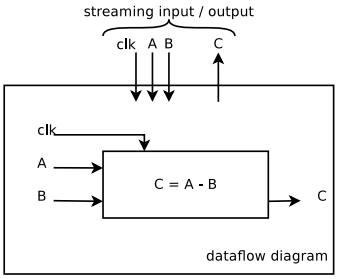


Figure 4.7: Data Path Diagram - Streaming Adder

4.3.3 Local Extrema Extraction Block

This paragraph is referenced by section 4.2.3

The local extrema extraction block has the responsibility to extract the 8-adjacency $N_8(p)$ neighborhood from the streaming image data as described in section 2.2.3. Because the data in the input stream is ordered in a row-wise fashion, we need to buffer three rows of data and slide a 3x3 window over the buffered stream. In the sliding window we check for each pixel clock step if the middle pixel is either larger or smaller than all other pixels in the window. A coordinate generator block (detailed description in section 4.3.5), generates the corresponding pixel coordinates for the middle pixel. If an extreme is found, the value and the coordinate of that extreme is streamed to the appropriate output. In a skeleton implementation this would result in the communication of a coordinate for a found extrema only if valid data is present. The 3x3 window slider is implemented using FIFOs with a total length of 2 lines and 3 pixels. A data-flow diagram is given in figure 4.8.

4.3.4 Averaging Block

This paragraph is referenced by section 4.2.3

The averaging block needs to calculate the piecewise average of two streams of data. This ALU shares the same properties as the one described in section 4.3.1 and 4.3.2, in that it is clocked using the pixel clock, and that its implementation depends on the ratio between the pixel clock and the system clock. Since this operation exists of adding two values and then dividing them by two, we can use a simple adder and a division by two. The division can either be implemented by a bit shift for a fixed point solution, or decreasing the exponent by one in case of a floating point solution. This would enable a quick and efficient implementation with the latency and area equal to one or two additions for the given precision. A data flow representation is given in figure 4.9.

4.3.5 Coordinate Generator

This paragraph is referenced by section 4.2.5 and section 4.3.3. The coordinate generator is effectively a counter that counts the row and column position of the current pixel coordinate. At reset, a count down is started from an offset to synchronize the timing. If the offset reaches zero, two counters start counting the pixel position, which is pushed to the output bus. Since coordinates are given in integers, we have the same version of the coordinate generator for all possible implementations. Also, latency is not of interest in this function block. The control diagram is given in figure 4.10.

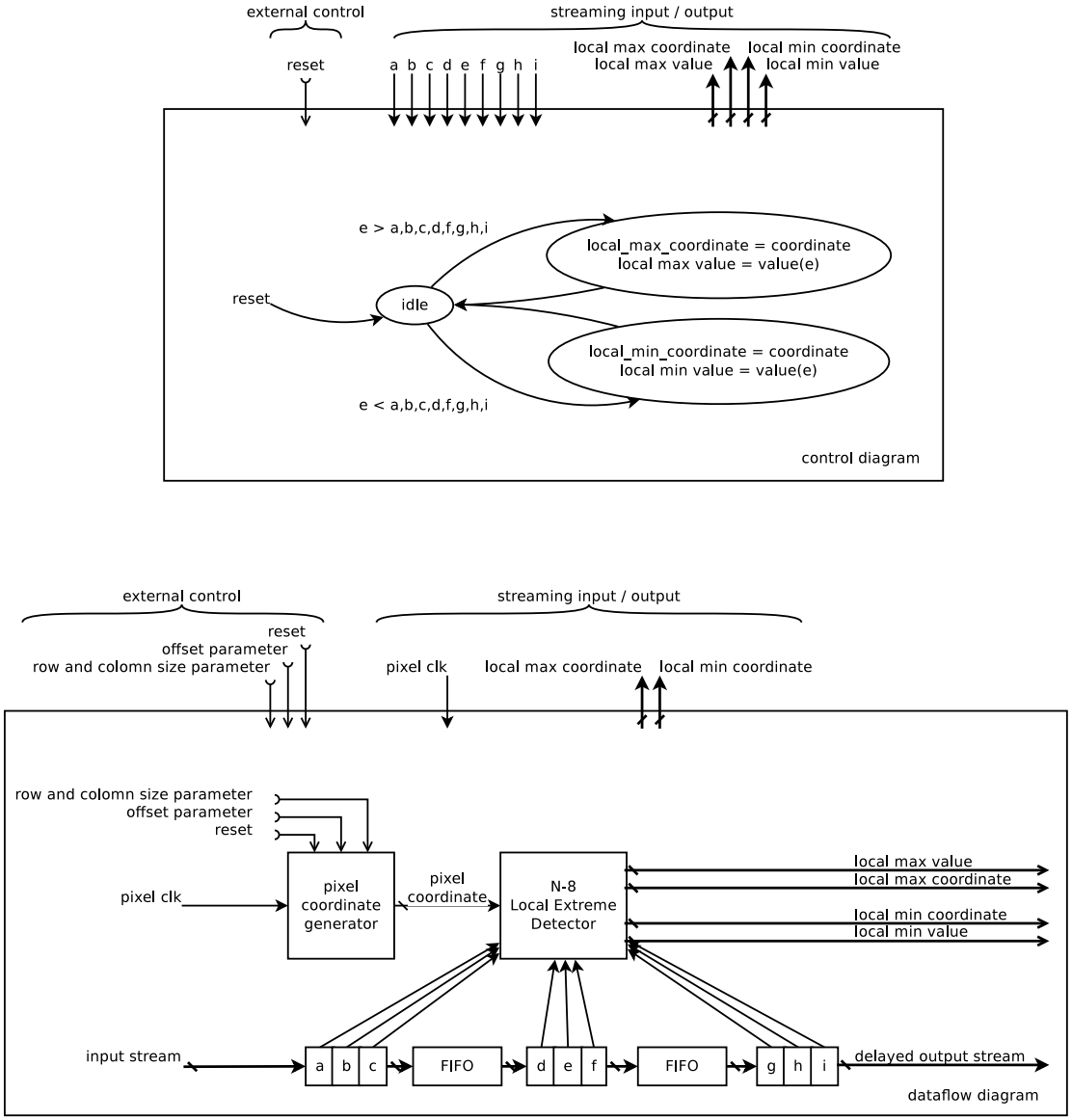


Figure 4.8: Data Path - Control - Diagram - 8-Adjacency Local Extreme Detector

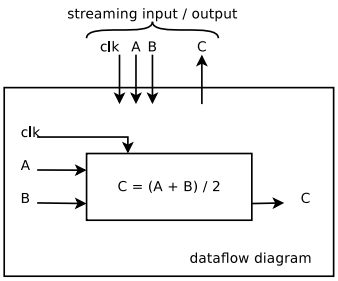


Figure 4.9: Data Path Diagram - Streaming Averager

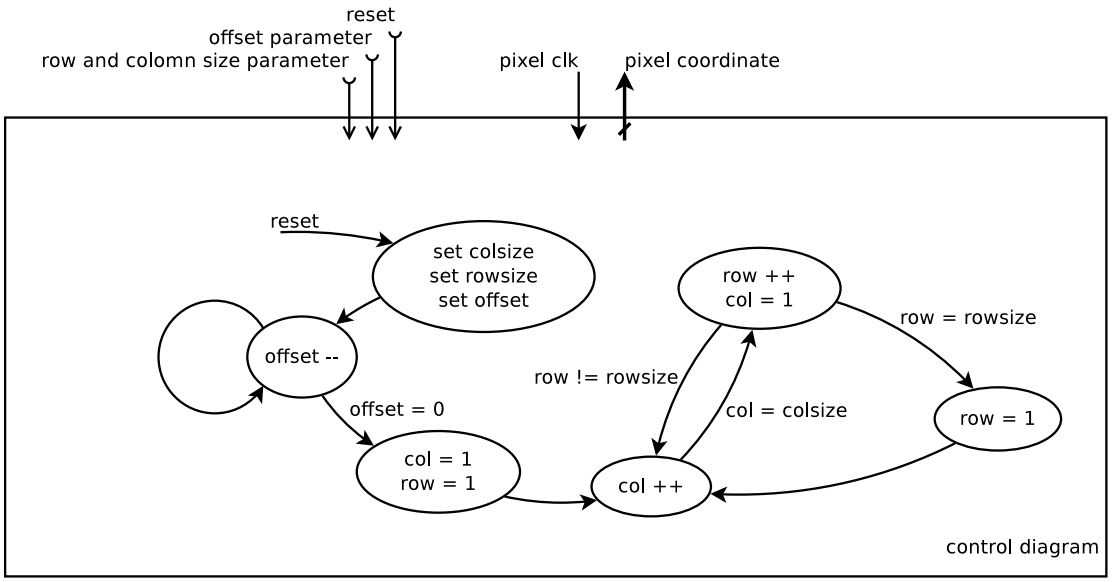


Figure 4.10: Control Diagram - Pixel Coordinate Generator

4.3.6 Nearest Neighbor average distance finder

As described in section 3.4.2, the size of the Gaussian filter can be estimated by calculating the average length of the nearest neighbors for each extrema in the cloud. The implementation approach is that each local extremum, when first encountered in the stream, will be appointed a worker. This worker then listens to the ongoing stream of local extrema coming in. Every time a new coordinate arrives, it calculates its own distance to the current coordinate. If the distance is smaller than the three known distances already found, it will remember the new distance and forget the largest one. It will continue to monitor the stream of incoming coordinates until the row distance of the stream is equal or larger than the distance to its three closest neighbors. In this case it will not be possible to find a closer neighbor. The worker will then report its nearest neighbor distances for its assigned extremum to an averaging unit or reduction tree, and then recycle by attaching to a new extremum.

The worker essentially does three additions two squares and a squared root[2] to calculate the distance. Two comparisons are needed to test for the shortest distance and the stop condition. It is probably worthwhile to match the needed precision of the calculation to the specific implementation to optimize the used area here.

The following possible optimizations can reduce the required area:

1. Only calculate the filter size for one of the two extrema point clouds.
2. Use a simplified way of calculating or estimating the distance with a LUT or specialized arithmetic.
3. Share a fast pipelined ALU among workers (if pixel clock << system clock).
4. Use less workers than strictly needed for the worst case and accept a more rough estimate.

[2]Taking the square root of the distance can actually be omitted inside the worker to reduce worker area. The square root can then be calculated in the reduction tree after the final distances are found, which is more effective. This change was also used inside the tested implementation.

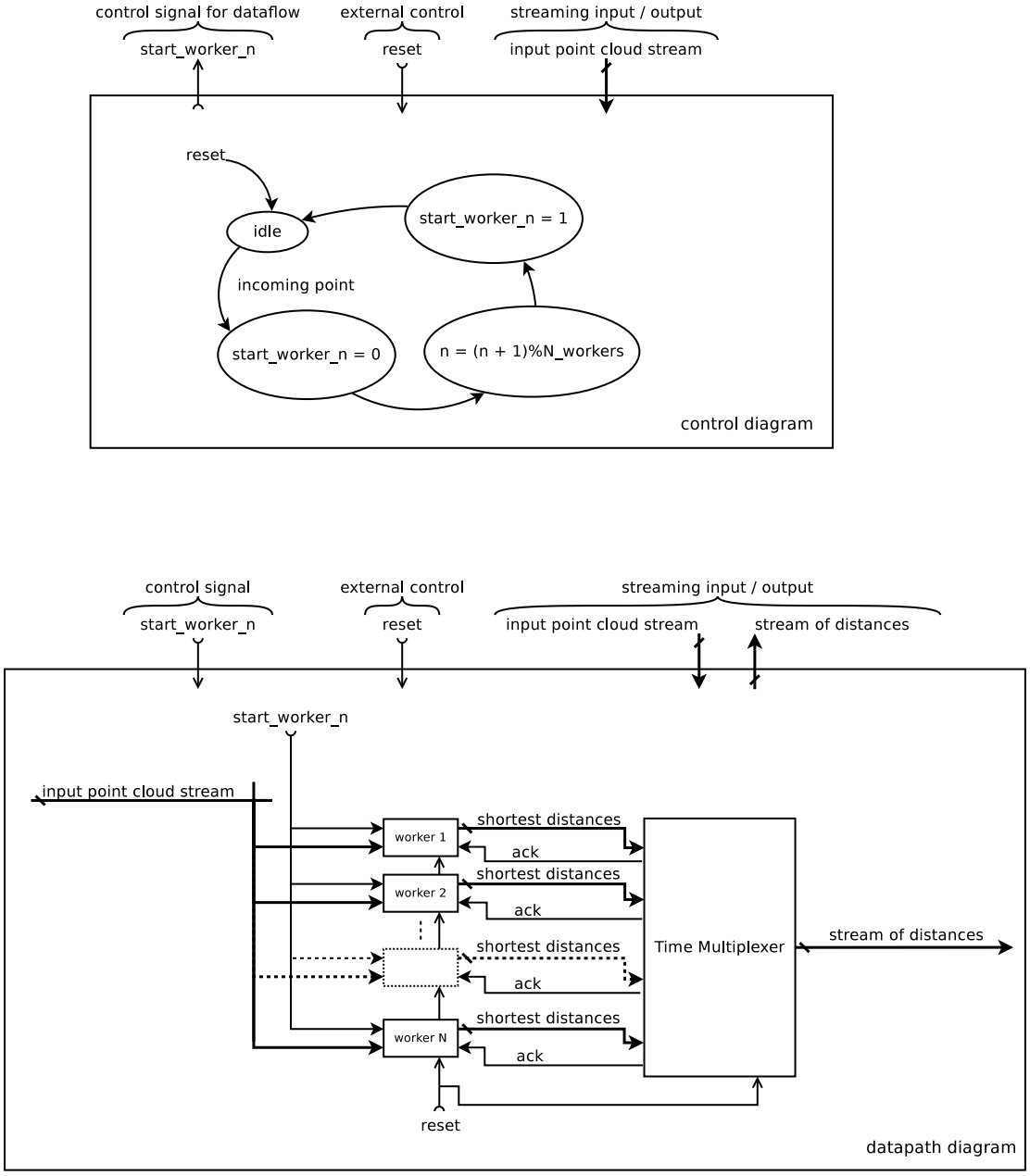


Figure 4.11: Data Path and Control - Streaming Average Nearest Neighbor Block

5. Only calculate the filter size for the first IMF and for subsequent IMFs guess it based on the first filter size.

The following optimizations can reduce the overall latency of the system.

1. Use the filter size calculated in the previous frame, which saves a frame delay but gives a large error for the first frame after a scene change.

2. Use an other method to get or guess the filter size.

Figure 4.11 gives a control flow diagram of the worker model that indicates how the workers work together as a system. Figure 4.12 gives the control flow diagram of a single worker.

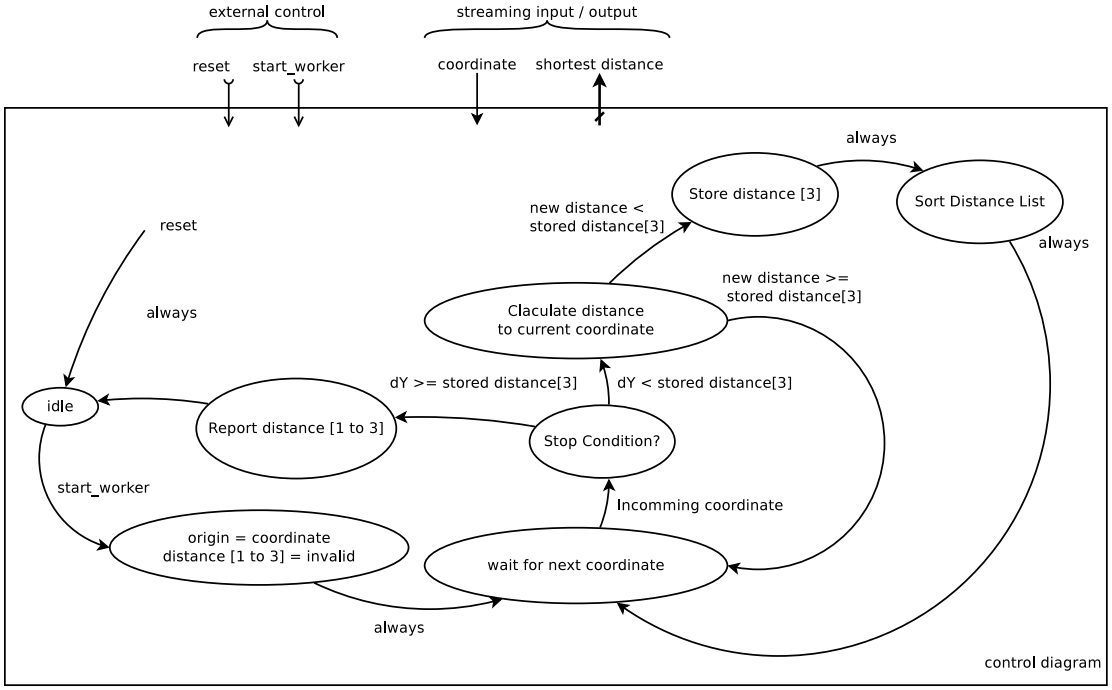


Figure 4.12: Control Diagram - 3 Nearest Neighbors Worker

4.3.7 Average Reduction

This paragraph is referenced by section 4.2.4

The nearest neighbor worker model (section 4.3.6), produces a set of parallel outputs that need to be reduced to one average. It is probable that more than one worker will terminate and report at the same pixel clock. This is especially the case when a new line is started since the Y coordinate is then increased, triggering the stop conditions of some workers. This congestion will however be limited to a single line, since the maximal number of extrema on one line is half of the number of pixels. This means that if we use the pixel clock as a baseline, we can have the congestion resolved within one line. With a handshaking system between the workers and a multiplexer we can produce a single stream of distances, with a slight delay in the stop logic of the worker as depicted in figure 4.11. This stream of distances as well as the number of distances can then be summed to produce the input for the devision that will calculate the average when the frame is done as depicted in figure 4.13. Since the numerator can potentially overflow, it should be big enough to hold the largest accumulated number.

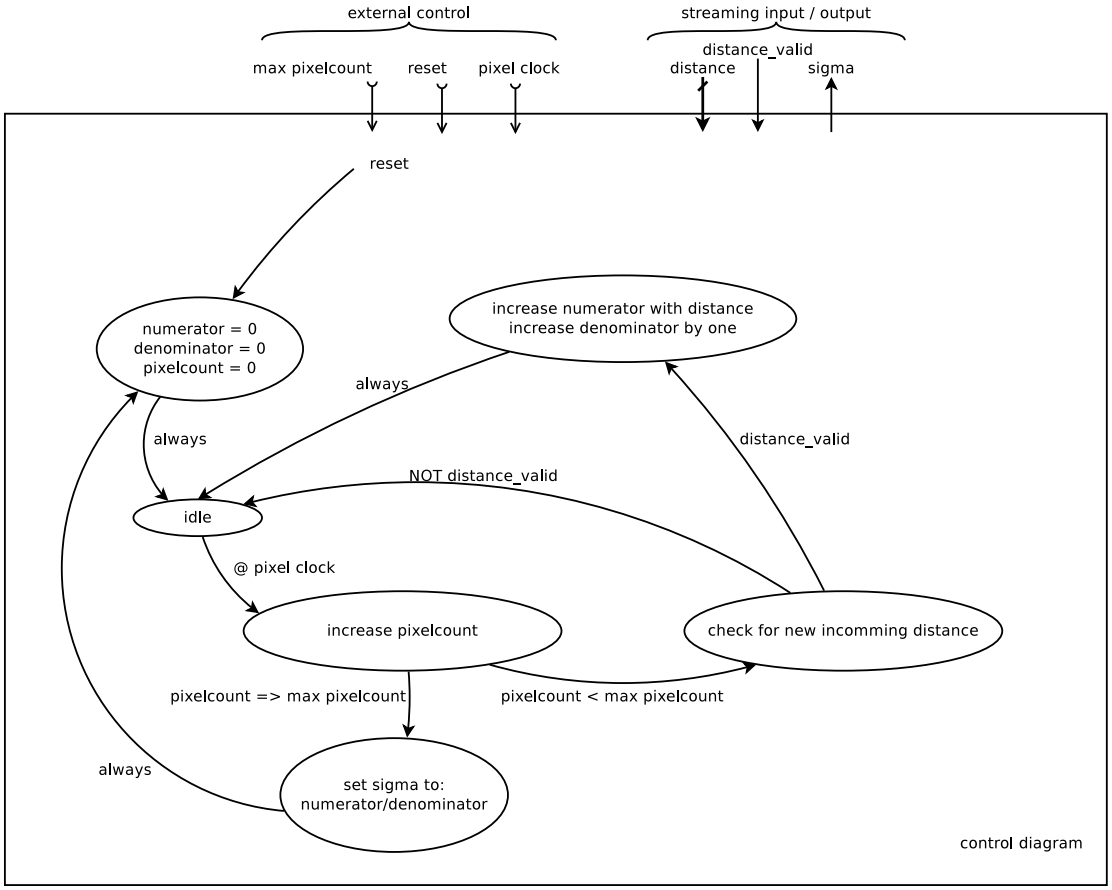


Figure 4.13: Control Diagram - Stream to Average Block

4.3.8 Normalized Convolution Interpolation Worker Farm

This paragraph is referenced by section 4.2.5

As described in section 3.4.3, a logical way of parallelizing the envelope estimation is to split the work into individual contributions of local extrema. As the coordinates themselves are streaming, the contributions to the coordinates will be streaming too. We also saw in section 3.4.3 that there exist a minimal number of extrema that will need to contribute to the envelope, to accomplish a predefined envelope quality. Adding or removing extrema for contribution will in this way influence the quality of the result. But since the streaming coordinate is moving as the stream progresses, we need to update which extrema populate the contributing group.

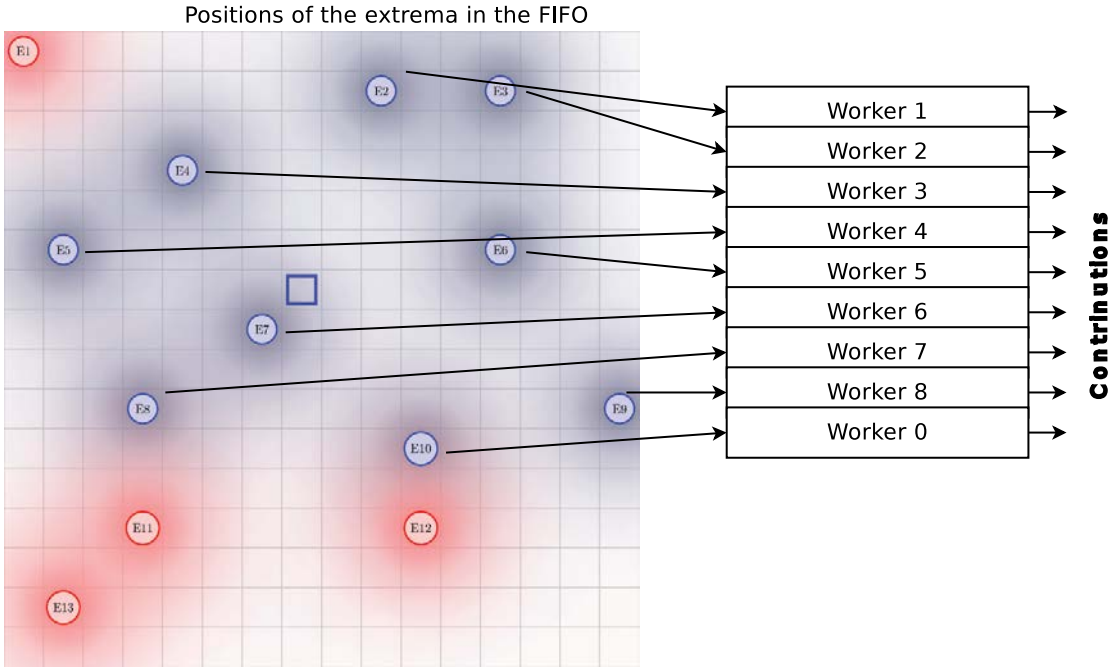


Figure 4.14: The Workers as FIFO for the extrema

As was also described in section 3.4.3, a FIFO collection containing a number of extrema surrounding the coordinate would give a simple solution to this problem. This concept is depicted in figure 4.14. Here we see that extrema $E1$ in the top left corner which was first connected to $Worker$ 0 is now outside of the confidence window and extrema $E10$ is now the new extrema for $Worker$ 0. We can thus replace extrema in the FIFO collection in such a way that the coordinate is always surrounded by enough contributing extrema.

To build the envelope creating hardware we can use a similar architecture to the nearest neighbor solution as described in section 4.3.6. We do this by attaching a worker to each incoming point in the stream of local extrema. Once a worker is attached it will start listening to a different stream that streams pixel coordinates in an increasing fashion. For each point in the coordinate stream it will then calculate its contribution and normalization factor. These two values are then fed into two reduction trees that receive all the contributions or normalization factors of all the workers for that coordinate. Each reduction tree will be implemented as a binary tree and will sum the received values in

an efficient way. The final sum of the contributions and the normalization factor will then be divided to produce the value of the envelope for the original coordinate that was streamed into the workers. A graphical representation of the control and data-path diagrams can be found in figures 4.15 and 4.16.

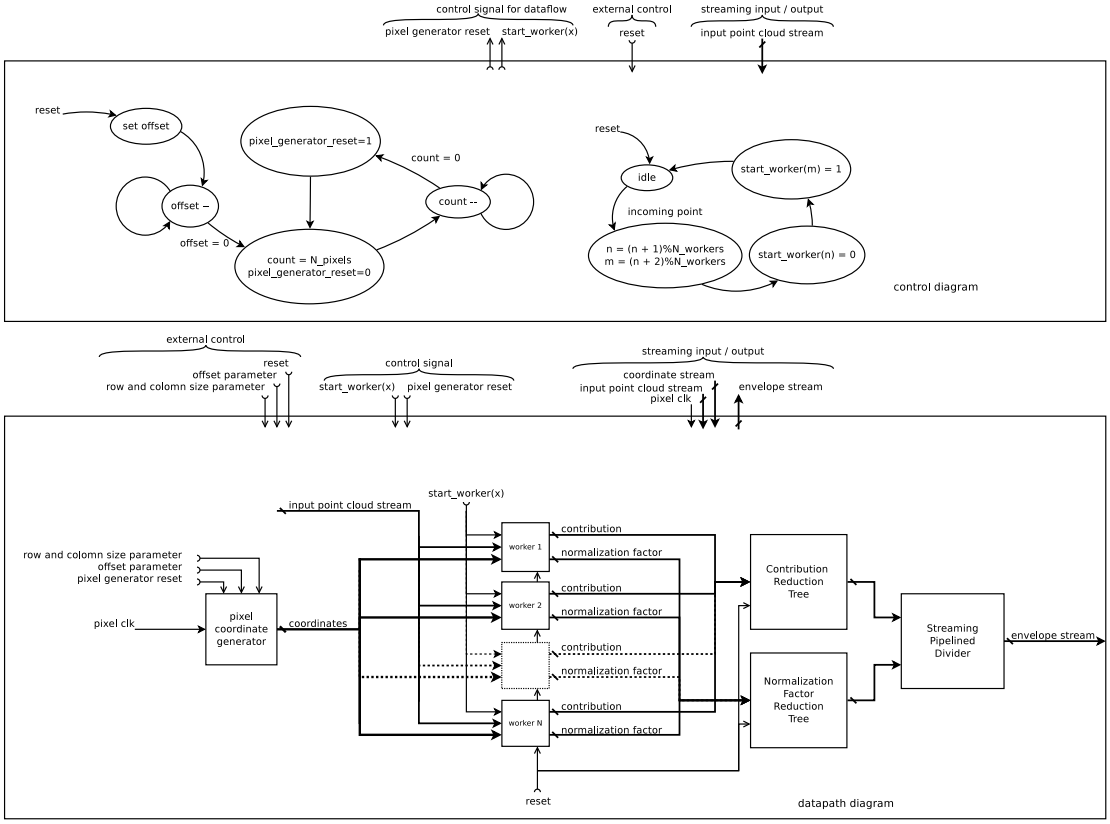


Figure 4.15: Data Path and Control - Normalized Interpolation Worker Farm

4.3.9 Streaming Summation Block

This paragraph is referenced by section 4.2.5

As described in sections 4.2.5 and 4.3.8, the workers produce a number of parrallel contributions that all belong to the same pixel coordinate. These contributions all need to be summed to enable the final envelope calculation. Since we know that all workers will always contribute simultaneously, we can make a binary tree of adders, or even use multi-operand adders. The exact implementation will again depend on the properties of the hardware platform and the maximal pixel clock. Because we want to have a flexible and scaling solution it is probably best to initially use a binary tree of normal adders.

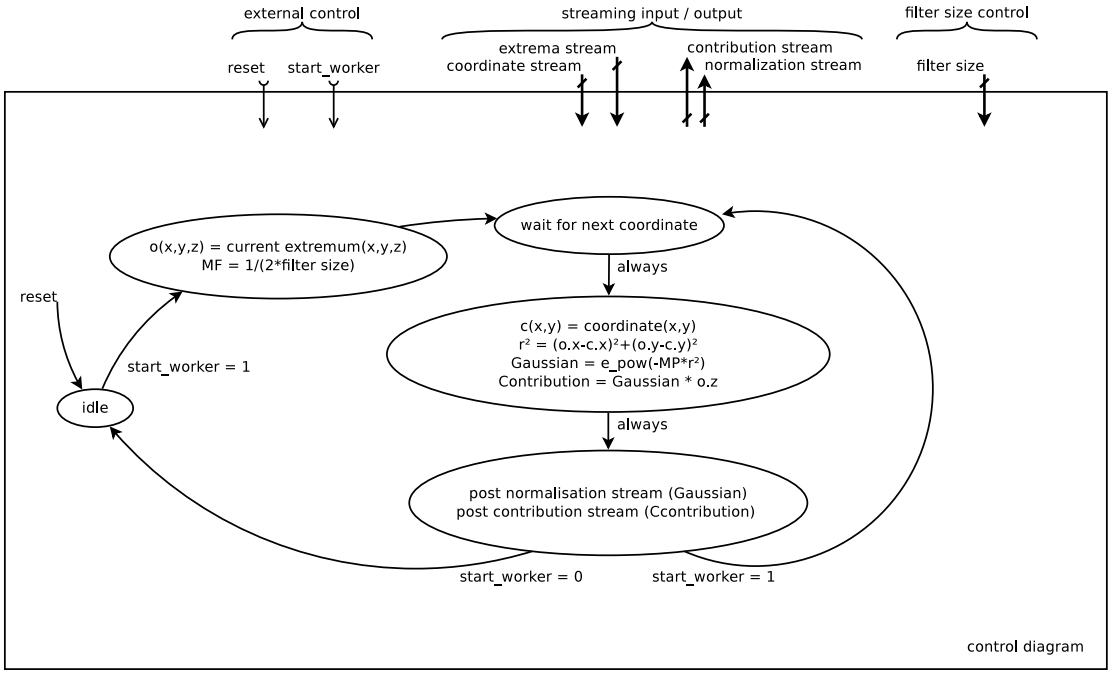


Figure 4.16: Control Diagram - Normalized Interpolation Worker

4.3.10 Streaming Buffer

This paragraph is referenced by section4.2.2 , section 4.2.3 and section 4.2.4

The streaming buffer has the responsibility to store a stream of data of any width for a specified length of time and behave as a FIFO. Because of the size of the data we want to store with this component, we can not use a normal FIFO. If we for instance take a quite large FPGA like the Altra Stratix IV GX530, that is used in our tests, we can only store about 26kB in the local caches. If we take all the recources of this chip into account we can eventually fit about one grayscale full HD image($\pm$ 2MB) on the FPGA and have a small portion of the logic elements left. To handle large delays in these big streams properly we need external memory.

The streaming buffer therefore is basically a FIFO connected to an external memory which stores the major part of this data. As can be seen in figure 4.17 the basic functionality of this block is to take streaming data and store it for a specified number of clock ticks in an external memory. This can be implemented as a circular buffer with a fixed size (loopsize parameter). The location of the data in the memory can be set using an offset (offset parameter). The way we use the circular buffer is to do a read after write on consecutive addresses. This is preferable because the DDR2 memory controller pre charges the consecutive address blocks (and rows and banks) while addressing memory locations, enabling fast burst transfers. The data-path diagram, figure 4.18, shows a more detailed representation of the dataflow. The memory addresses for reading and writing are controlled by counters in the address generator. The (maximal) size of the stored data and the offset in the memory are defined at compile time.

A more detailed description of this process now follows:

When data enters the the system, a shift register concatenates the data to the word size of the memory. If the shift register is full, it enables the read on the connected input-lane FIFO and latches the data. This process continues until the FIFO is half

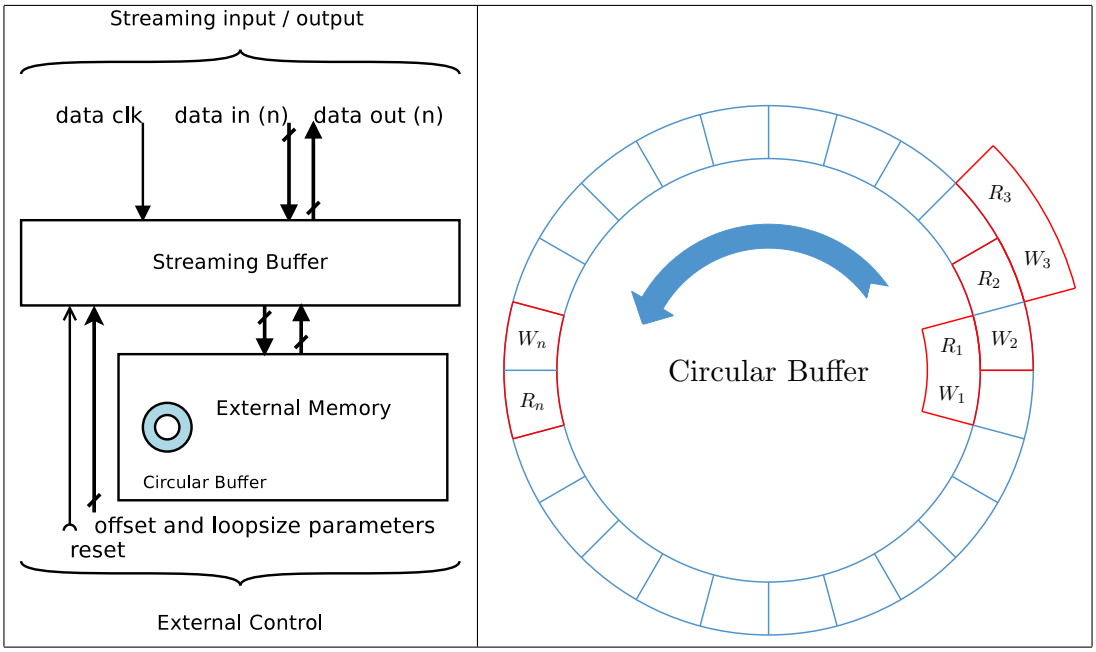


Figure 4.17: Streaming Buffer

full, and the half_full_1 signal is enabled. Now the controller (figure 4.18), requests the internal address bus of the memory subsystem. The system now waits for the memory to become available, but it keeps processing the streaming data using the FIFOs as buffers. This can take some time, but no more than the time that is needed to fill the FIFO completely. When the memory becomes available, the address generator generates the next write address and half of the FIFO's content if written to the memory. Because we read from the streaming buffer at the same rate as we write to it, we can now also read the contents of the next memory location and write it to the output-lane FIFO. The controller sets the correct signals to accomplish this and when the data is read, it releases the memory sub-system. At the same time that data is clocked in, it is also clocked out. This is done using the reversed process. A shift register is filled using the contents of the output-lane FIFO, where the data is split up in the width of the output data, and clocked out. Every time the shift register is empty, it reads a new chunk of data from the output-lane FIFO. By construction the system has always enough data in the output-lane FIFO to be read. This system does not take any special actions to flag data as stale, as would be the case when the system is started. This is left to the components using this system, and can be accomplished by using a counter, if needed.

If we know the number of streaming buffers, their maximal buffer sizes and the rates of the data and memory bus, we can define the scheduling statically and calculate the worst case memory utilization for real-time operation. A study was made to this design and bachelor graduation project[3] was done[12] on this proposed design, which resulted in a version that works in simulation. This implementation was written in *System Verilog* under supervision of the author of this thesis. During tests it showed that the hardware implementation suffers from glitches, which will have to be solved to get a practical solution.

To get an idea about the used hardware resources, table 4.1 gives an overview of the recourses used by our test bench implementation.

[3]HRO, thesis project by Raul James

Block	*Detail*	*ALUTs*	*Register*	*Memory*	*DSPs*
Streaming Memory Controller	Test Version	6168(1%)	6864(2%)	0	0

Table 4.1: Area needed for the basic Streaming Memory Controller block serving one stream. Additional streams will require a little bit more area, but most of the recourses are now in the memory controller. (Altera DDR2 controller)

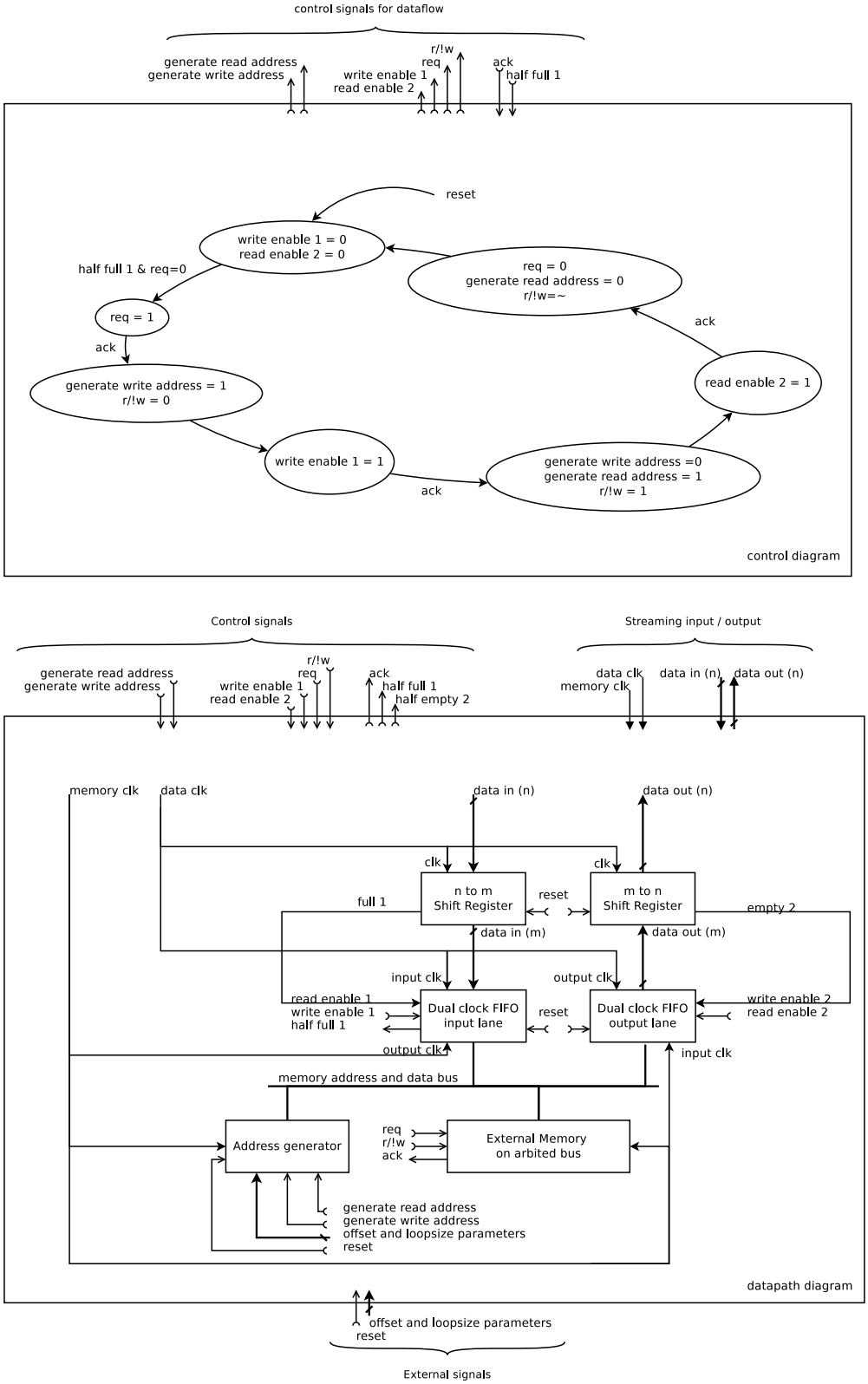


Figure 4.18: Dataflow and Control - Streaming Buffer

5 Worker Model Simulation

Before we can start making hardware for the proposed EMD implementation we have to test the behavior of the worker model as discussed in section 4.3.6 and 4.3.8, to see how many workers we need and how precise the operations will have to be. Given this information we will be able to estimate the feasibility of this approach for a real implementation. To test the behavior of the implementation of the EMD hardware, a simulator in C++ has been made that imitates the proposed hardware implementation, so we can judge the proposed architecture for the folloing topics:

1. Look at the consequences of the worker model on the timing
2. Research the effect of scaling the number of workers with regard to the quality
3. Investigate the effect of the recycling of workers
4. Have a platform to quickly test architectural changes and implementation specific solutions

For the implementation of the simulator and the hardware we will use an abstraction technique called *Algorithmic Skeletons*, to separate structure from computation, enabling us to streamline the implementation process and enable reuse of components. In the next section a short explanation on *Algorithmic Skeletons* is given, after which a number of skeletons in our system is discussed. We end this chapter with a more specific description of the implementation, explaining how specific parts are working and what data is recorded.

5.1 Algorithmic Skeletons

In the previous chapters we introduced the concept of a dynamic load balancing system with *workers*. This concept of workers is basically an architectural concept, in that it describes an algorithmic architecture rather than a specific calculation. The worker model therefore can be described as an *algorithmic skeleton* [6].

In the Delft Bio-Robotics Lab, research is ongoing on automated design of digital hardware for robot vision and visual servoing, using algorithmic skeletons and stream processing [5]. Currently, the work focusses on the implementation of skeleton frameworks for FPGAs, which brings the opportunity of using the benefits of algorithmic skeletons for this project as well.

Algorithmic skeletons are a template system based on functional programming and generally specify a method of processing, such as *for all pixels in the image do:"..."*, or *for all pixels in the neighborhood do:"..."*. Skeletons form an interface that is independent of the parallel system that implements them. In this way skeletons are used to hide data parallelism. A specific skeleton describes a specific algorithmic construction and separates architecture and functionality in, respectively, a *skeleton* and a *kernel*. Because each specific skeleton describes a specific algorithmic construction like a template, a different optimal implementation can be made for a number of hardware solutions, e.g. single processors, multi processors, GPUs or FPGAs. With the availability of implementations on a variety of hardware platforms, design space exploration can be performed such that the skeleton is executed on the most efficient processor structure. Furthermore, the same skeleton may be instantiated multiple times with different kernels, leading to the reuse of components. Implementations of skeletons can be made – also for FPGAs – using code generators, however for this project we will not implement a complete hardware solution, but only research the feasibility of such an implementation using skeletons.

5.2 Skeletons in the proposed architecture

To illustrate the functionality of skeletons shortly, let us look at the most widely used skeleton which is called the *map* skeleton. The type[1] for this skeleton is:

```
map :: (a->b) -> ([a]->[b])
```

In an image processing setting can this be described as *for all pixels in the image do...*. To start with we have a function for an element `a` that produces an element `b` written as `(a->b)`, which in our case would be the pixel operation. The map skeleton then takes this function and applies it to a set of pixels `[a]` and produces a set of pixels `[b]`, e.g. it takes an image and produces an image.

A skeleton can take any function that has the same structure as described in its definition. The function that we put into the skeleton is called the *kernel*. To show the flexibility of this we will look at two kernels that can fit in a map skeleton. First of all, in the design of this thesis, we use `convert` kernel to calculate the luminance value out of the red, green and blue pixels as described in section 4.3.1. the Haskell notation for this kernel is:

```
convert :: (Integer, Integer, Integer) -> Double
```

Here we see that this kernel takes a tuple of three arguments as an input and produces one output, but its essential structure is still defined as `(a->b)` and it can thus be used in a map skeleton. An other example is the kernel for the streaming adder as described in section 4.3.2

```
add :: (Double, Double) -> Double
```

Here we see that we have the same structure, but the kernel takes a tuple of two arguments in stead of three, which does not matter for the skeleton.

[1] We use Haskell notation for type and function composition. See [10] for details.

The kernel and the skeleton together form an *operation*, but because we already use this word for the pixel operation we will call it a *frame operation* to emphasize the difference. . We can then thus describe the frame operations for the for examples above as:

```
color_conversion = map convert
streaming_adder = map add . zip
```

Here we see that there is an extra function called `zip`, which does nothing more than combining two sets of data to one set of tuples, which in our case means that it merges two streams of pixels; i.e. `zip :: ([a],[b])-> [(a,b)]`.

If we now look at the design of the previous chapter, we can recognize the following skeletons, kernels and frame operations in the system, as shown here in algorithm 1.

Algorithm 1 Haskell description of all modules in the Worker Model design for Empirical Mode Decomposition using Normalized Convolution.

```
1  -- Skeletons
2
3  map :: (a->b) -> ([a]->[b])
4  filtermap :: (a->(b, Bool)) -> ([a]->[b])
5  iworker :: (a -> (b -> c)) -> (([a], [b]) -> [[c]])
6  eworker :: (a -> ([b] -> c)) -> (([a], [b]) -> [c])
7  reduce :: ((a, a) -> a) -> ([a] -> a)
8
9  -- Kernels
10
11 convert :: (Integer, Integer, Integer) -> Double
12 add :: (Double, Double) -> Double
13 extract :: [Double] -> ((Integer, Double), Bool)
14 average :: (Double, Double) -> Double
15 interpolate :: (Integer, Double) -> (Integer -> (Double,Double))
16 add2 :: ((Double, Double), (Double, Double)) -> (Double, Double)
17 div :: (Double, Double) -> Double
18 nndf :: Integer -> ([Integer] -> (Integer,Integer,Integer))
19
20 -- Frame Operations
21
22 color_conversion = map convert
23 streaming_adder = map add . zip
24 streaming_averager = map average . zip
25 extrema_extractor = filtermap extract . stencil
26 interpolation = map (div . reduce add2) . iworker interpolate
27 nearest_neighbors = reduce average . eworker nndf
```

To clarify this cryptic Haskell description, we will discuss each of the *Frame Operations* separately. Note that in the definition of the frame operations we will read the Haskell notation from right to left if we reason from input to output. Since we already discussed the `color_conversion` and the `streaming_adder`, let us now look at the `streaming_averager`.

streaming_averager

```
-- Frame Operation
streaming_averager = map average . zip
```

Here we see that we merge two streams using the `zip` function, creating a stream of tuples, which we feed into the kernel `average`.

```
-- Kernel
average :: (Double, Double) -> Double
```

This kernel creates an average value out of the tuple, which for our case will be a pixel operation averaging two pixels. We then use the `map` skeleton to apply this operation on a whole frame, for which we will use a streaming implementation.

extrema_extractor

```
-- Frame Operation
extrema_extractor = filtermap extract . stencil
-- Skeleton
filtermap :: (a->(b, Bool)) -> ([a]->[b])
-- Kernel
extract :: [Double] -> ((Integer, Double), Bool)
```

Here we see that the frame operation `extrema_extractor` starts with the `stencil` function. This function creates a set of subsets of the set that is put in, `stencil :: [a] -> [[a]]` which in our case represents a sliding 3x3 window over the image. Because this is also the most practical implementation in a streaming environment we will implement the `stencil` as a sliding window. The stream of neighborhoods created by the `stencil` is then fed to an `extract` kernel in a `filtermap` skeleton. The `extract` kernel takes a set, which in this case is the set of values from the sliding window, and produces a coordinate, a value, and a boolean that indicates whether the middle of the window was an extremum. The `filtermap` skeleton is basically a combination of a map skeleton and a filter that will filter out all results that have a false boolean, while performing the operation on a whole frame. We thus get a set of extrema, described by coordinates and values.

interpolation

```
-- Frame Operation
interpolation = map (div . reduce add2) . iworker interpolate
```

This frame operation describes the normalized convolution worker model. To start, let us first look at the `iworker` skeleton.

```
-- Skeleton
iworker :: (a -> (b -> c)) -> (([a], [b]) -> [[c]])
```

Here `a` is a task, `b` is a piece of work, and `c` is the result of that piece of work-task combination. The combination of task, work and outcome can be run in parallel over the tasks and the work, and creates a set of sets as outcome `[[c]]`, with each inner set being the outcome of all tasks for a certain piece of work. This skeleton runs the kernel `interpolate`, that takes an extremum (coordinate,value) as task, a coordinate as work and produces a tuple containing the *contribution* and the *normalization factor* of the normalized convolution equation, as is also depicted in figure 4.15

```
-- Kernel
interpolate :: (Integer, Double) -> (Integer -> (Double,Double))
```

The parallel results then need to be reduced using adders which is done by the skeleton `reduce` using the kernel `add2`

```
-- Skeleton
reduce :: ((a, a) -> a) -> ([a] -> a)
-- Kernel
add2 :: ((Double, Double), (Double, Double)) -> (Double, Double)
```

The `add2` kernel adds two tuples to form one tuple and the reduction skeleton will execute the kernel on the inner set of results from the `iworker` skeleton, thus generating a single *(contribution, normalization)* pair per pixel. This is followed by the `div` kernel that divides the two. A map skeleton is then used to do this on all outer sets of results from the `iworker` skeleton, producing the envelope.

nearest_neighbors

```
-- Frame Operation
nearest_neighbors = reduce average . eworker nndf
```

This frame operation describes the nearest neighbor worker model. To start, let us first look at the `eworker` skeleton.

```
-- Skeleton
eworker :: (a -> ([b] -> c)) -> (([a], [b]) -> [c])
```

Here `a` is a task, `[b]` is all the work, and `c` is the result of that task on all the work. The combination of task, work, and outcome can be run in parallel over the tasks, and creates a set as outcome `[c]`. In our situation, the tasks are represented by the individual extrema, the work is the set of all extrema left in the stream and the results are tuples of nearest neighbor distances. In the implementation as described in this thesis, we use the natural order in the stream, which is a raster scan, as input for `[b]`. But this is a specialization of the general case as described here and does actually differ slightly in results, because the general case takes the whole stream into account, while we only use the remainder of the stream. The details on this difference are however outside of the scope of this report. The skeleton uses the `nndf` kernel, which stands for nearest neighbor distance finder.

```
-- Kernel
nndf :: Integer -> ([Integer] -> (Integer,Integer,Integer))
```

This kernel takes one coordinate as task, for which it will look at a set of data, also containing coordinates, to look for the three nearest neighbors of which it will report the distances in a tuple. Finally we reduce the set of distances to a single distance with a reduction skeleton using the kernel `average`

```
- Skeleton
reduce :: ((a, a) -> a) -> ([a] -> a)

-- Kernel
average :: (Double, Double) -> Double
```

5.2.1 Parallelism

The skeletons described above reveal that parallel processing is possible, but it hides the specific parallel implementation. Because, in our specific case, we use streams, we

basically have two forms of parallelism in our implementation. First of all, because we have streams, a frame takes a certain amount of time to pass through the system, giving the system time to do sequential processing on pieces of the stream, i.e. stream processing, which, in the sense of a frame time period, is parallel processing. Then we also have the traditional parallelism i.e. the execution of tasks in the same time step. These *two dimensions* of parallelism give the opportunity to optimize the system in time and in space. Given the `iworker` skeleton, we see that we can parallelize over the tasks `[a]` as well as over the work `[b]`.

```
iworker :: (a -> (b -> c)) -> (([a], [b]) -> [[c]])
```

In our implementation we take the streaming temporal parallelism into account for the work `[b]` and the spatial parallelism for the tasks `[a]`. We then use the `reduce` skeleton to reduce the results back to the streaming domain. For the `eworker` skeleton, we practically use the same solution, even though the skeleton is different because the worker requires knowledge of the entire set of work to come to a result.

```
eworker :: (a -> ([b] -> c)) -> (([a], [b]) -> [c])
```

In our implementation we solve this by giving the worker a sense of state, giving it the possibility to decide for itself when it is ready to produce a result.

5.2.2 Reuse

The skeleton description shows that a large number of structures in the design can be reused. `map` and `reduce` are used multiple times and also the `worker` skeletons share much of the same structure. Even if automated code generation is not used, this can significantly reduce costs for implementing this design in hardware.

5.3 FPGA Skeleton Structure

Parallel to developing this hardware architecture, our lab is busy developing a streaming FPGA skeleton framework. This means that we can use this basic structure to implement our skeletons in hardware in the future. Figure 5.1 gives a graphical representation of the streaming FPGA skeleton structure.

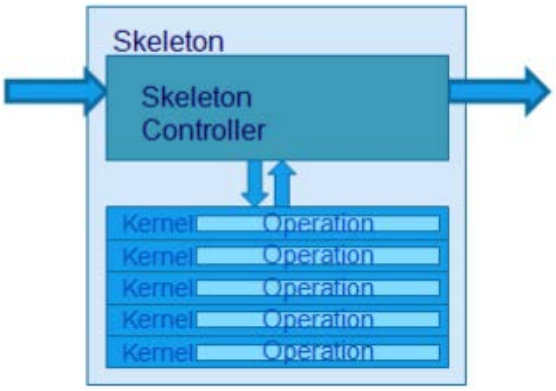


Figure 5.1: Skeleton Controller and Kernel

In this figure we see a skeleton controller, which describes the specific functionality of that skeleton, and we see a number of kernels, which can be chosen for different types of operation, or which can be used in a *worker skeleton* way. The skeletons modules can be placed in a pipeline and skeletons can themselves be used as kernel. The skeletons have standardized streaming inputs and outputs, and they can be daisy chained as long as the types of output and input match. An automated code generation tool is available and at this moment, only a small number of skeletons is available. Since this is work in progress in our lab, there are no publications available at this moment. We will however use the available hardware descriptions to test one of the workers in our design; more details are given in chapter 6 and the code for the worker operation tested can be found in appendix A.

5.4 Simulating the Worker Model Hardware

C++ has been chosen to write a simulator of the worker model implentation, because we can easily implement the skeletons using the object oriented nature of C++, as there is a strong resemblance between the class concept of C++ and skeletons. In terms of information hiding, the skeleton hides how a function is implemented, independently of the function itself. This means that we can represent workers as objects, so we can create a flexible framework in which we can test the influence of the number of workers as well as the dynamic behavior of the proposed FIFO structure for workers.

Streaming Worker Skeleton Class Structure Since C++ offers inheritance between classes, we use this feature to define the principle relation between skeletons. As depicted in figure 5.2, our system consists of four basic classes. In principle the *Skeleton Class* is never directly used and just acts as the common ancestor[2] for the *Worker* and the *WorkerSkeleton*. In this system a *Job* object acts as a single data element in a stream, the *Worker Skeleton* represents the architecture – e.g. it acts as a data-path representation[3]– and the *Worker* represents the operation. The *WorkerSkeleton* class thus describes how the *Worker* objects are initiated, recycled, and how streams are connected to the workers, and furthermore acts as a container for a group of *Worker* objects, hence if a stream is given to a *WorkerSkeleton* it will distribute the stream among its *Workers*, which themselves only process information that is posted to them.

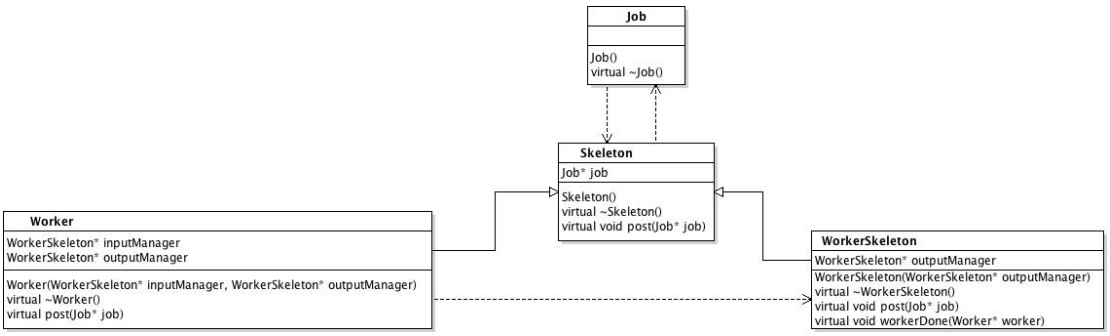


Figure 5.2: Worker Skeleton Class Diagram

To implement the streaming nature, we register the *outputSkeleton* in both constructors to define the stream flow structure; i.e. the output skeleton is the destination of the streams that are produced by the workers. We also register an *inputSkeleton* in the constructor of each worker, representing the worker-skeleton that handles that worker and enabling that worker to directly call functions of its parent worker-skeleton; i.e. interacting with the control of the skeleton.

We will use this structure as a framework for the worker models described in the following two sections.

[2]Note that in theory the kernel (the kernel in our case is a worker) of a skeleton can be another skeleton, so if they have the same type, we can interchange them.

[3]Some control is included so this is not entirely correct.

5.5 Filter Size Estimation

The filter size calculation as described in section 4.3.6, uses workers to find distances in a stream of extrema. As discussed in the previous section, this process is divided in two parts, i.e. a worker that does the actual distance calculations and the skeleton that represents the architectural layout of the workers and their overall control logic. We will first look at the worker, and then discuss the skeleton.

The Worker Implementation To simulate this architecture, we have an object, called a *FilterSizeWorker*. This worker will be appointed a local extremum that is stored as its *_origin*, and it will then monitor the incoming extrema. If an extremum is posted, it will execute the control process as described in figure 4.12, which is shown in the code snippet given in algorithm 2.

Algorithm 2 The pixel operation description of the Nearest Neighbor Distance Finder Worker, this code snippet is only for illustrating the functionality of the worker and is a simplified version of the simulator code.

```
class NNDF_Worker
/* calculate distance to posted point, sort distances and check stop condition */
void post(Job* job){
        Point* point = (Point*)job;      //create a Point pointer from the job
        /* if the max_nr_of_neighbours_in_distance is not reached,
        *  just add the point to the nearest neighbor list */
        if(_nr_of_neighbours < max_nr_of_neighbours_in_distance){
        addNeighbour(_nr_of_neighbours,point, distance(point));
        _nr_of_neighbours ++;
        }
        /* else see if the termination condition is met */
        else if( point->y()-_origin->y() > _distance[_max_distance_index])
        {
        //add the latency based on the current Y distance
        _origin->incLatency((point->y()-_origin->y())*_cols + (point->x()-_origin->x()) );
        done(point);
        }
        /* else check if the new point is closer than one of the already
        *  known neighbours */
        else
        {
        unsigned int d =  distance(point);
        if (d < _distance[_max_distance_index]){
                addNeighbour(_max_distance_index, point, d);
                updateMaxDistance(); // sort distances
                }
        }
}}
```

To calculate the latency that the system introduces to each produced distance in the stream, we keep track of the latency. The latency introduced by this worker consists of doing the specific calculations plus the time the worker listens to the stream (i.e. before recycling). The part of the latency that is caused by the calculation is dependent on the implementation details and will be a constant in the simulator. The data dependent latency behavior of the worker is registered at the moment of termination by measuring

the pixel stream distance[4] between the current coordinate and the workers origin, as depicted in the following code example, where we see that we register the value of the shortest distances, the current position of termination, the original start position of the worker (originDistance), and the latency recorded by the origin. The code is shown in algorithm 3.

Algorithm 3 Function for terminating the worker. Note that this code snippet is only for illustrating the functionality of the worker and is a simplified version of the simulator code.

```
void done(Point* currentPosition){
        //calculate the distance in pixlelclks from framestart to origin.
        int originDistance = _origin->y()*_cols + _origin->x();
        //add data dependent latency to origin
        _origin->incLatency((currentPosition->y()-_origin->y())*_cols +
         (currentPosition->x()-_origin->x()) );
        //post the three shortest nn distances (if) found
        //and the current position as absolute latency indicator.
        for(unsigned int i=0; i<_nr_of_neighbours; i++){
        Value* d = new Value(_distance[i],this,currentPosition->x(),
                        currentPosition->y(),
                        originDistance);//create a value with the distance
        d->incLatency(_origin->latency());//add the latency for this value
        outputManager->post((Job*)d,true);//post
        }
        //reset and recycle
        reset();
        inputManager->workerDone(this);
}
```

Additionally we add a latency of one pixel clock to the initialization procedure in which the origin is set at the start (not shown).

With regard to the latency we assume that the distance calculation and sorting as described in the control diagram of figure 4.12 will be able to cope with the rate at which extrema enter the system, either by being quick enough or by being implemented in a way that handles multiple simultaneous calculations, e.g. being pipelined. A more in depth discussion on this topic will follow in chapter 6.

The eworker Skeleton Implementation The functionality of this implementation is described in the data- and control-flow diagram of figure 4.11. Actually, this figure describes two skeleton implementations. The first is the implementation of the `eworker` skeleton, managing the instantiation of the workers and the control of dividing the work among the workers. The second is the implementation of the `reduce` skeleton, which will handle the reduction tree.

We will start with the the `eworker` skeleton implementation, which is quite simple. Basically it has to attach a worker to every new extremum entering the stream. Its second responsibility is to connect the incoming data stream to all workers. A part of the code is given in algorithm 4 to illustrate the way the skeleton was implemented. Because all latencies are managed by the workers, we do not add additional latencies here.

[4]This is the number of pixel clock ticks that will elapse between two given pixel coordinates in the stream.

Algorithm 4 Function that handles the management of workers in the implementation of the eworker skeleton. Note that this code snippet is only for illustrating the functionality of the worker and is a simplified version of the simulator code.

```
class FilterSizeWorkerModel { /*<The following code is only a part of the whole class>*/
virtual void post(Job* job){
        //assign a worker with this point as origin
        //check if we have a contracted idle worker, if so, use that one
        if(_idleWorkers.size()>0){
                NNDF_Worker* worker = (NNDF_Worker*)_idleWorkers.back();
                _idleWorkers.pop_back();
                worker->setOrigin(job);
        }
        //else, contract a new worker
        else if(_uncontractesWorkers) {
                  _workerArray[_contracted_workers] =
new NNDF_Worker(this, _outputManager, _rows, _cols );
                _workerArray[_contracted_workers]->setOrigin(job);
                  _contracted_workers++;
                _uncontractesWorkers--;
        }
        //else, quit with error
        else{
                printf("can not process job, no available workers left\n");
                assert(0);
        }
//post this extremum to the existing active workers to do their work
        for (unsigned int i = 0; i < _contracted_workers ; i++){
        _workerArray[i]->post(job, false);
        }
}}
```

The reduction skeleton implementation (see code snippet in algorithm 5) relies on the time multiplexing of the parallel results to create a single stream. In practice we get a number of parallel results at the start of a next line; i.e. the stop condition is triggered by a line increase. The most parallel results will probably become available when the last line is reached and all remaining workers finish simultaneously. To see what happens in practice, we record the number of parallel results that come available at the start of each line in the simulator. We can use this to check if the timing of the multiplexer is possible. The results are then streamed into a streaming averaging unit, that has a fixed delay. The `_relativeLatency` is the latency between the moment the worker starts and the moment the worker ends. The `_absoluteLatency` is the time between the frame start and the moment the worker ends. The `_frameAddittiveLatency` is the time between the end of the frame and the moment the worker ends, if the worker ends after a frame end. This last latency is actually the most interesting latency for calculating the total latency, because we already know that the worker model will have to process the complete contence of a frame.

Algorithm 5 Function of filter size implementation for the `reduce` skeleton. Note that this code snippet is only for illustrating the functionality of the worker and is a simplified version of the simulator code.

```
class FilterSizeReduction : public Skeleton{
virtual void post(Job* job)
{
        Value* dist = (Value*)job;
        //add distance value to our averager
        _distance->addValue(dist->x());
        //check contributions per line
        if (dist->lastY() == _previousY){
                _counter++;
                if (_maxCounter<_counter) //record max
                        _maxCounter = _counter;
        } else {
                _previousY = dist->lastY();
                _counter = 0;
        }
        //add latency value to our latency statistics
        _relativeLatency->addValue(dist->latency() + _counter + AVERAGINGLATENCY);
        _absoluteLatency->addValue(dist->baseDistance() + dist->latency() + _counter +
                                   AVERAGINGLATENCY);
        if ((dist->lastX() >= _cols) && (dist->lastY() >= _rows)){
                _frameAddittiveLatency->addValue(_counter + AVERAGINGLATENCY);
        }
        _nr_of_workers++;
}}
```

5.6 Envelope Generation

The normalized convolution as described in section 4.3.8, uses workers to calculate the contribution for each extremum. Similar to the previous section, this process is also divided in two parts, i.e. a worker that does the actual contribution calculations and a skeleton that represents the architectural layout of the workers and their overall control logic. We will first look at the workers, and then discuss the skeleton.

The Worker Implementation As described in section 4.3.8, each worker will be initialized with the coordinates of a single extremum. Once the worker is initialized it will listen to a stream of coordinates and for each coordinate produce a contribution and a normalization factor. These two streams will then stream out to a reduction skeleton.

So to start, we have a function that initializes the worker and sets its *_origin* and the size of the Gaussian, as shown in algorithm 6.

Algorithm 6 Function for setting the origin and sigma for the normalized convolution. The sigma is stored as $\frac{1}{2\sigma^2}$ for efficiency.

```
class ConvolutionWorker : public Skeleton{
void setGaussParameters(Point* origin, double multiplyfact){
        _origin = origin;
        _multiplyfact = multiplyfact; // = 1/(2*_variance)
        _initialized = true;
        }
}
```

The worker then listens to a stream that is delivered to the worker using the *post()* function as depicted in algorithm 7.

Algorithm 7 Operation of the normalized convolution worker. Note that this code snippet is only for illustrating the functionality of the worker and is a simplified version of the simulator code.

```
class ConvolutionWorker : public Skeleton{
void post(Job* job){
        Point* point = (Point*)job;      //create a Point pointer from the job
        int dx = _origin->x()-point->x();
        int dy = _origin->y()-point->y();
        //calculate distance to the given point
        long square_distance = pow((float)dx,2)+pow((float)dy,2);
        //_maxSquaredDistance controls Exponent Precision
        if (square_distance < _maxSquaredDistance){
          //calculate the weights
          double exponent = square_distance * _multiplyfact;
          double gaussianWeight = pow(M_E,-exponent);
          double filterValue = _origin->z() * gaussianWeight;
          D4Point* contributionPoint = new D4Point(
                point->x(),point->y(),filterValue,gaussianWeight,outputManager);
          //post the results
          outputManager->post((Job*)contributionPoint,true);
        }
}}
```

As can be seen, the calculation is quite complex, in arithmetic terms. The way this calculation is implemented will influence the area and latency of the solution, as well as the quality of the end result, if rounding and/or estimation errors start to play a role. For the worker model, however, the important aspect is that the latencies of all the workers is equal (or within equal bounds), and that the rate at which new contributions need to be calculated is dependent on the data. For the simulator we assume a fixed latency for calculating a contribution. We also assume that the calculations are pipelined so that the system is (theoretically) not bounded by the pixel frequency. Since the data dependent latency[5] is not part of the worker but of the skeleton, we will keep the latency administration for the worker limited to a constant.

The *_maxSquaredDistance* variable is used to control the exponential precision, so that we can simulate different implementations of the exponential calculation. *_maxSquaredDistance* is calculated in such a way that it represents the maximal distance for which we can calculate a value with a limited exponential calculation. A more in-depth discussion on this topic follows in chapter 6.

The iworker Skeleton Implementation The implementation of the `iworker` has the same structure as the `eworker` implementation, in that the skeleton has the control that starts the workers and recycles workers that are done. In this specific case, however, the recycling of the workers is initiated by the skeleton implementation, i.e. the workers are in a FIFO and the oldest worker is replaced when a new extrema comes in. Furthermore, the skeleton controls the timing at which the coordinate generator needs to start so that the pixel stream is in the middle of the considered *confidence window*, as explained in section 3.4.3. In the data-control diagram depicted in figure 4.15 this timing is set by an

[5]The data dependent latency in this case is the time difference between the start of the extrema stream and the start of the coordinate stream.

offset parameter which was not further explained. In the simulator we have the freedom to experiment with different solutions for controlling this parameter and we start with a construction that enables the coordinate streaming when half the buffer is full. We will see in chapter 6 (results), that this is not an optimal solution.

The timing delay which the skeleton imposes on the resulting stream is therefore data dependent and should be considered when testing the system. We know that there is a clear relation between this delay and the filter size (σ) of the Gaussians because they are both related to the individual distances of the extrema.

Algorithm 8 The implementation of the `iworker` skeleton for the normalized convolution worker model. Note that this code snippet is only for illustrating the functionality of the worker and is a simplified version of the simulator code.

```
class ConvolutionWorkerModel : public Skeleton{
virtual void post(Job* job){
        Point* point = (Point*)job;

        /* PART 1: appoint new worker or recycle old one*/
        //if we have uncontracted workers, give them a job
        if(_uncontractesWorkers) {
          _workerArray[_contracted_workers] =
                new ConvolutionWorker(false ,this, _outputManager );
          _workerArray[_contracted_workers]->
                setGaussParameters(point,_multiplyfactor);
          _contracted_workers++;
          _workerIterator++;
          _uncontractesWorkers--;
        }
        else{ //if all workers are already busy, recycle the oldest one
          if(_workerIterator >= _nrOfProcessingElements){
            _workerIterator=0;
          }
          _workerArray[_workerIterator]->done(); //have the worker finish its job
          _workerArray[_workerIterator]->setGaussParameters(point,_multiplyfactor);
          _workerIterator++;
        }

        /* PART 2: stream the appropriate coordinates*/
        //If we did not start the coordinate generation
        if(_coordinate_generator_delay == 0){
          //we start generating coordinates if at least half of the workers has a job
          if (_contracted_workers >= _nrOfProcessingElements/2){
            _coordinate_generator_delay = _cols*point->y() + point->x() + 1;
          }
        }
        else{
        /*if we did start the generation we stream all the pixel coordinates that
          * should have been processed between the old and the new shore line.
          * calculate the new shore line*/
          unsigned int new_shore_line = _cols*point->y() + point->x()+
                1 - _coordinate_generator_delay;
          generateCoordinates(new_shore_line);
        }
}
}
```

Algorithm 8 shows how the skeleton is implemented in the simulator. In the first

part of the loop we manage initialization and the recycling of the workers. Note that we set the origin and the size of the Gaussian in the same function. In the second part we simulate the coordinate generator. Because we run this function[6] only when a new extremum arrives, we have to stream the intermediate coordinates each time to keep the simulation in sync. We therefore introduce the concept of a shoreline in the simulator, representing the position that the coordinate generator is at, in between the different iterations of this function. We first establish the moment at which the worker FIFO is half full, represented by the term *_nrOfProcessingElements/2*. When this happens we set the variable *_coordinate_generator_delay*, which represents the offset of the coordinate generator for this frame. We then have a function *generateCoordinates(new_shore_line)* that streams all the coordinates to the workers that are between the previous and the new shoreline.

The *_coordinate_generator_delay* is recorded and represents the delay between the coordinate stream of the incoming extrema and the coordinate stream of the outgoing envelope.

As shown in figure 4.15, there are two reduction trees in the system that each reduce one of the two streams of all the workers to a single value. As stated in section 4.3.8, each reduction tree will be implemented as a binary tree and will sum the received values in an efficient way. The final sum of the contributions and the normalization factor will then be divided to produce the value of the envelope for the original coordinate that was streamed into the workers. In the simulation we will just sum all values that enter the system and give the divided results at the end. Additionally, the number of contributing workers for each pixel coordinate is recorded.

[6] officially called a method in C++ jargon.

5.7 The Simulator

Using the the Worker Skeletons as described above as a basis, a complete simulator was build that mimics the behavior of the design as described in chapter 4. With this simulator we are able to do a complete EMD of any image, while we record the following data:

1. The file name of the processed image
2. Number of columns
3. Number of rows
4. Number of the IMF for which the following data holds
5. Number of the Sift for which the following data holds
6. Maximum used number of workers in the filter size estimator
7. The added latency at the end of a frame before sigma was calculated
8. The value of sigma
9. The number of workers involved in the maxima stream of the envelope generation
10. The number of workers involved in the minima stream of the envelope generation
11. The data dependent latency of the maxima confidence window (*_coordinate_generator_delay*)
12. The data dependent latency of the minima confidence window (*_coordinate_generator_delay*)
13. The number of local maxima in the frame
14. The number of local minima in the frame
15. The minimal contribution in the envelope generation
16. The Confidence Window Sigma Multiplier[7]
17. The maximal range of the exponent function
18. The dynamic range of the image

For every processed image this data was saved, and this information was used to analyze the performance and behavior of the intended hardware architecture, as described in chapter 6.

[7]This is the normalized indicator for the number of workers used, as will be explained in section 6.3.1

6 Results and Discussion

6.1 Quality of the simulated EMD Results.

In this chapter we will first look at the total results from the EMD process. Since BEMD lacks a golden standard, as previously explained in chapter 2, we will make an assessment on the quality of the EMD results with the most comparable algorithm, which is the version we developed in our group, and which stood model for the hardware implementation.

The original decomposition made with Matlab code

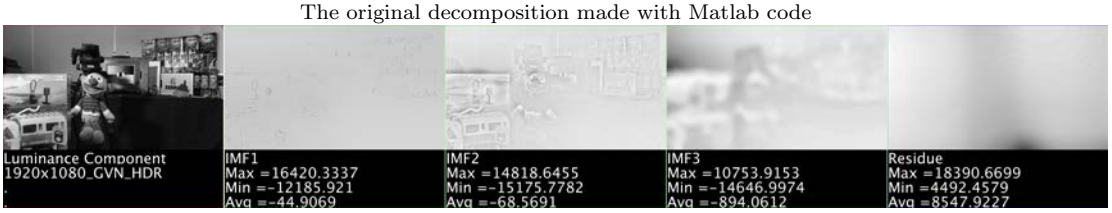
The decomposition made with the simulated hardware code

Figure 6.1: Decomposition of Lena with the same settings, calculated by the original algorithm and the hardware simulation. Underneath the results we see name and the minimal, maximal and average value of the pixels in that result.

To start with we maximized the number of used workers, as well as the possible ranges for calculating the Gaussian curves. This is to see if the worker model solution is capable of correct results. These settings are however not feasible for an FPGA implementation at the current moment, so we will investigate the places where we can reduce precision and used number of workers, further on in this chapter. Figure 6.1, shows an example decomposition of Lena[1] (512 by 512 pixels), processed with two sifts. The top decomposition is made with the original Matlab script and the bottom decomposition is the result of the hardware simulator as described in the previous chapters. The input image is shown on the outer left side, then followed by the three subsequent IMFs with the residue at the far right. Please note that the IMFs and the residue are displayed with a stretched range to make them easily visible. The stretch factor is indicated by the minimal and maximal figures under the results.

The most important feature of an IMF is the average value, which is also displayed under the results and which should be close to zero. If we look at the high dynamic range, Full HD version of the GVN[2] scene, processed with 2 sifts per IMF, we get the EMD as depicted in figure 6.2.

The original decomposition made with Matlab code



The decomposition made with the simulated hardware code

Figure 6.2: Decomposition of HDR GVN scene with the same setting, calculated by the original algorithm and the simulated hardware.

Here we see again that we get visually similar results. There are absolute differences in the pixel values though. If we compare the average between the top and bottom IMFs, which is an error indication of the IMF, we see that for both algorithms, they are of the same order of magnitude. This shows that there is no buildup of error in between the iterations of the different sifts, because they would produce additive errors leading to more differences each iteration. One possible explanation for differences, is that the calculated sigmas differ slightly due to the structurally different way of calculation as described in chapter 3.4.2. An other reason can be that the way the pixel data is interpreted in C++ can be different from the way we interpret it in Matlab, e.g. the conversion from color to luminance has slight differences. A third reason that can influence the difference in the results is, that in the Matlab scripts, we only use doubles, while the simulator uses integers where possible. If we look at the quality of the EMDs in general, the larger absolute nonzero averages in the high dynamic range

[1] http://www.ee.cityu.edu.hk/ lmpo/lenna/Lenna97.html

[2] The GVN scene is a test image recorded by the camera manufacturer of our test camera:Grass Valley[9]

should be appreciated in the perspective of their dynamic range. If we look at the third IMF, where the absolute average value is largest, we see that it is still within 4% of the absolute range; i.e. $\frac{avg}{min-max} \cdot 100$. The size of the non zero average value is related to the number of sifts per iteration. This can be improved by using more sifts as shown in figure 6.3, where we see that the average value of the third IMF is now halved using four sifts per IMF, compared to the decomposition of figure 6.2 that used two sifts per IMF. Using more sifts will have large affects on the used number of resources as we will see later in this chapter.

Figure 6.3: The decomposition made with the simulated hardware code using 4 sifts per IMF in stead of 2. Here we see that compared to figure 6.2 we now have only halve the non zero average for the third IMF.

Figure 6.4: If we add up the IMFs and the residue form the simulator results and subtract that from the original image, we get the difference. Note that the Min, Max and Average values are extremely small.

An other property of the EMD is that if we add up all the IMFs and the residue, we should get the original image back. If we do this, there is no visual difference to the original; so to highlight the small differences there are, figure 6.4 shows the differences after subtracting the combined image from the original image, for the HDR GVN scene processed by the worker model simulator. This difference is displayed in a stretched

format, and the min, max and average values, which are in the order of 10^{-12}, shows that these differences are most probably caused by rounding errors.

We can conclude from these initial tests that the worker model architecture for the BEMD decomposition is structurally working. We will now look into detail at a number of aspects for the implementation that can be tuned and see what their effect is on timing, scaling and area. We will then make an estimate for the number of resources that will be required for implementing this architecture on an FPGA for a number of resolutions and frame rates.

We will look at the timing, scaling and resource usage of the *filter size estimation* and the *normalized convolution* separately, as shown in figure 6.5, where we see the two different worker models as used inside the envelope generating block.

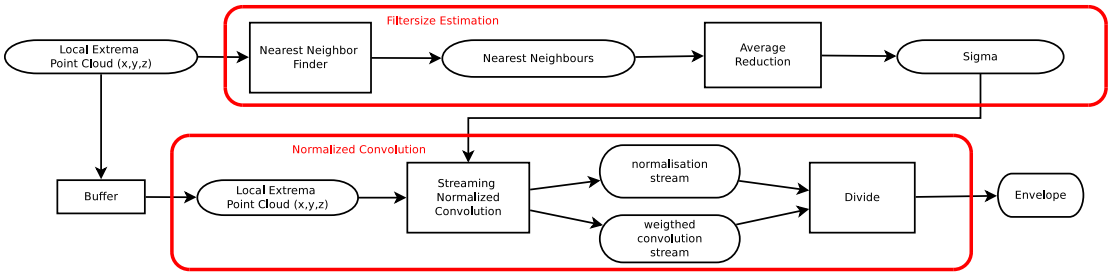


Figure 6.5: Timing of the two worker models

6.2 Filter Size Estimation

The simulator was used to test a number of real life images, as well as some artificial images to see their effect on the added latency, for different resolutions and for varying IMFs. For statistical analysis we took a set of 90 natural High Dynamic Range (HDR) images of outdoor scenes from the University of Texas [4] and a set of 40 random noise images[3], which were all cropped and sub-sampeld to 1920x1080, 960x540, 480x270 and 240x135. We also looked at Lena in three resolutions and at a low dynamic range (LDR) and HDR version of the GVN test scene, which was recored by the camera manufacturer for this purpose.

Because the number of workers determines the needed area in the implementation, it is important to know the maximal number of workers that will be needed for a given resolution. Furthermore, for the filter size estimation, the latency is related to the number of workers in the image, so it is also important for the latency to know the maximal amount of workers. The amount of workers is itself directly related to the number of extrema in the image. To maximize the number of workers, we need to maximize the number of extrema, so an artificial image was constructed to maximize the number of extrema. In absolute terms, an image can contain a maximum of one 8-adjacency $N_8(p)$ neighborhood local max per grid of four pixels as shown in figure 6.6

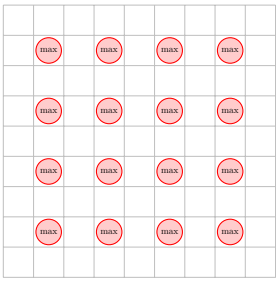


Figure 6.6: There can be only one 8-adjacency $N_8(p)$ neighborhood local max per grid of four pixels.

6.2.1 Used Number of Workers

If we look at the used number of workers for the filter size estimation, and plot them as a function of the number of columns in the image we get the graphs shown in figure 6.7.

Note that the number of workers will generally be less for higher order IMFs. We can also see that there is a linear relation between the resolution and the used number of workers, in the case of the *maximal extrema test image*. If we look at the results for the natural images at the right side of figure 6.7, we see that the percentage of workers per number of columns drops for higher resolutions. This suggests that the maximal recorded frequencies in a scene will not grow as quickly as the maximal frequencies that can be encoded when we increase resolution. This could for instance be due to a limited number of frequencies in the scene, defocus or optical limitation of the camera system.

[3]Images created with the random function of Matlab

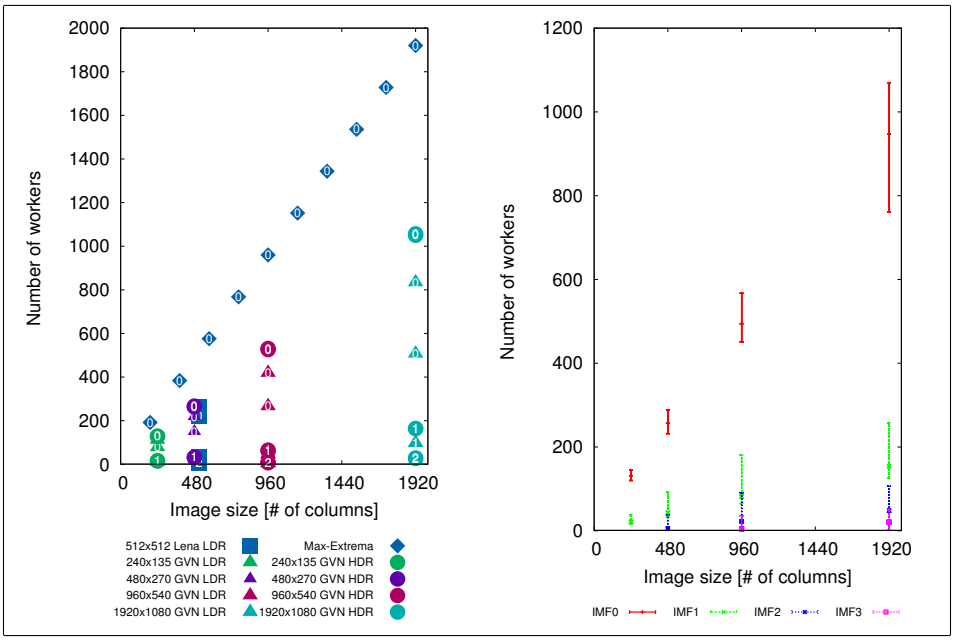


Figure 6.7: Number of workers used in filter size estimation for a variety of resolutions and images. The left graph shows the actual values for Lena, a max extrema image and the GVN scene in different resolutions and two dynamic ranges. The right graph shows the statistical analysis of two sets of images (described in text), depicting the min max and average value for different IMFs. The number of the IMF is depicted in the symbol for the left graph. The number of the IMF is depicted as a color for the right graph. IMF 0 is the original image, IMF 1 is the first IMF etc.

In absolute terms, an image can contain a maximum of one 8-adjacency $N_8(p)$ neighborhood local max per grid of four pixels. This means that a full HD image of 1920x1080 pixels can contain up to $\frac{1920 \cdot 1080}{4} = 518400$ extrema. The maximal number of workers used measured in this case is 1920 (see figure 6.7), equal to the amount of column pixels in the image, which means that the average worker was recycled 270 times in the streaming of one frame. In general terms we get:

$$Worker\ Reuse = \frac{columns \cdot rows}{4} \cdot \frac{1}{columns} = \frac{rows}{4} \tag{6.1}$$

Which means that the efficiency of the system will actually increase for higher resolutions; i.e. for a higher number of rows in the image.
This also shows that the used number of workers scales relatively to the number of columns in the image, and that the number of rows only has an influence on the amount of reuse of the workers. This suggests that the number of workers can be minimized if the orientation of the raster scan is changed so that the number of rows in the image is larger than the number of columns in the image.

6.2.2 Area

To see what the costs are in terms of area, some pieces of the worker model were implemented on an FPGA platform. In this case the worker implementation was completely build in Verilog, and is given in appendix A. A good judgement of the costs, however, is poorly expressed in numbers, because modern FPGAs have a large range of building

blocks with overlapping functionality, and so the metric for the used area is dependent on the available hardware resources, like memory blocks, slices, DSPs, dedicated arithmetic blocks, etc. Furthermore, the use of these building blocks is primarily decided by the synthesis software, so a deterministic outcome is impossible between different designs. We can however use a couple of these numbers to give an indication on the used area for this implementation. In our case we used an Altera® Stratix 4 FPGA[4] with the Altera Quartus® 2 software to implement our worker model structures. The figures for used *Combinational ALUTs*, *Logic Registers*, *Block Memory Bits* and *DSP elements* were taken as the metric to express the used area.

If we look at the resources as given by Altera for the intended FPGA we have the figures of table 6.1 to our disposal:

ALUTs	*Register*	*Memory (Kbit)*	*DSPs*
424960	424960	27.376	1024

Table 6.1: Resource availability for the FPGA used in out tests, which is the Altera EP4SGX530KH40C2.

As stated before, for the purpose of studying the area consumption, a functioning worker model kernel was created in *Verilog*, which we tested using a *hardware in the loop* construction in which a serial port is used to stream information to and from the skeletons. If we look at the used area for the filter size calculation we get the figures of table 6.2. An overview of the area needed for processing a given resolution is shown in 6.3 It is interesting to note that the synthesis software starts consuming the DSPs and when they are all gone, it models the same functionality with the other resources available. The difference can be seen in the table for the two worker operations presented. As can be seen we can approximately house 500 workers excluding, their reduction, on one FPGA. For more than 500 workers, synthesis was not possible, and the tested max frequency of the workers with or without DSPs was 209MHz. Taking the reduction into account, it is feasible that we can implement one filter size estimation for a resolution of 240x135. An implementation for 480x270 pixels would reach the boundary of our FPGA resources for the workers alone and is probably not achievable if we include the supporting logic, like a complete reduction tree. Note that we also need one filter size estimator per IMF.

[4]Stratix IV GX EP4SGX530

Block	*Detail*	*ALUTs*	*Register*	*Memory*	*DSPs*
Empty Skeleton	part	2	2	0	0
Empty Kernel	part	5	5	0	0
Empty Worker	part	12	10	0	0
Empty Worker Skeleton	TOTAL	19	17	0	0
Worker Operation inc. dsp		356	121	0	10
Worker Operation no dsp		900	178	0	0
Worker Skeleton 1 worker	TOTAL	375(0.09%)	138(0.03%)	0(0%)	10(0.98%)
Worker Skeleton 10 workers	TOTAL	2940(0.69%)	1550(0.36%)	0(0%)	100(9.77%)
Worker Skeleton 100 workers	TOTAL	29787(7.01%)	15500(3.65%)	0(0%)	1000(97.65%)
Worker Skeleton 150 workers	TOTAL	77776(18.3%)	26700(6.28%)	0(0%)	1024(100%)
Worker Skeleton 500 workers	TOTAL	378073(89.0%)	89000(20.9%)	0(0%)	1024(100%)

Table 6.2: Area of the Filter Size Calculation Worker Skeleton excluding the reduction skeleton

Cols	Rows	max workers	*ALUTs*	*Register*	*Memory*	*DSPs*
240	135	240	169252(40%)	44500 (10%)	0	1024(100%)
480	270	480	376252(89%)	85440 (21%)	0	1024(100%)
960	540	960	808252(190%)	170880 (42%)	0	1024(100%)
1920	1080	1920	1672252(394%)	341760 (84%)	0	1024(100%)

Table 6.3: Estimated recourses of the Filter Size Calculation Worker Skeleton, excluding reduction for different resolutions. Figures for 960x540 and 1920x1080 are estimated.

6.2.3 Timing

Because we need to loop over a whole image frame before we can produce a result for the filtersize estimation, the latency of this block is always larger than one frame. As described in section 5.5, a number of workers is assigned to each extrema in the image as they stream in, and each worker will monitor the distance for its extremum to the following extrema in the stream. During the streaming of the frame, workers are recycled if they have found their results, and thus at the end of the frame a number of workers will have been recycled while others will be active. To finish calculating the result at the end of a frame, we will therefore still need to stop all the active workers, handle their results and then calculate the sigma. This means that we will first have to serialize the output of the workers, stream it to the accumulators and then divide the result.

The left side of figure 6.8 shows the latencies introduced for *Lena*, the *LDR version of GVN*, the *HDR version of GVN*, and the *artificial image* that maximizes the number of local extrema, while the right side of figure 6.8 shows the statistical ranges for the 90 HDR images from the Texas Campus set [4] and 40 random noise images created with MATLAB®.

The additive latency for the worst case extrema image, shown on the left of figure 6.8, as the top diamond shaped blocks, is linear to the number of pixel columns. This is to be expected because the worst case number of extrema will lead to the worst case number of active parallel workers, and thus to the worst case congestion in the parallel to serial conversion at the end of a frame. Given this result we can see that the timing of the filter size estimation will not be a problem, and given the assumptions[5]that were

[5]As described in section 5.5; i.e. the atomic calculations take less time than a pixel clock or are

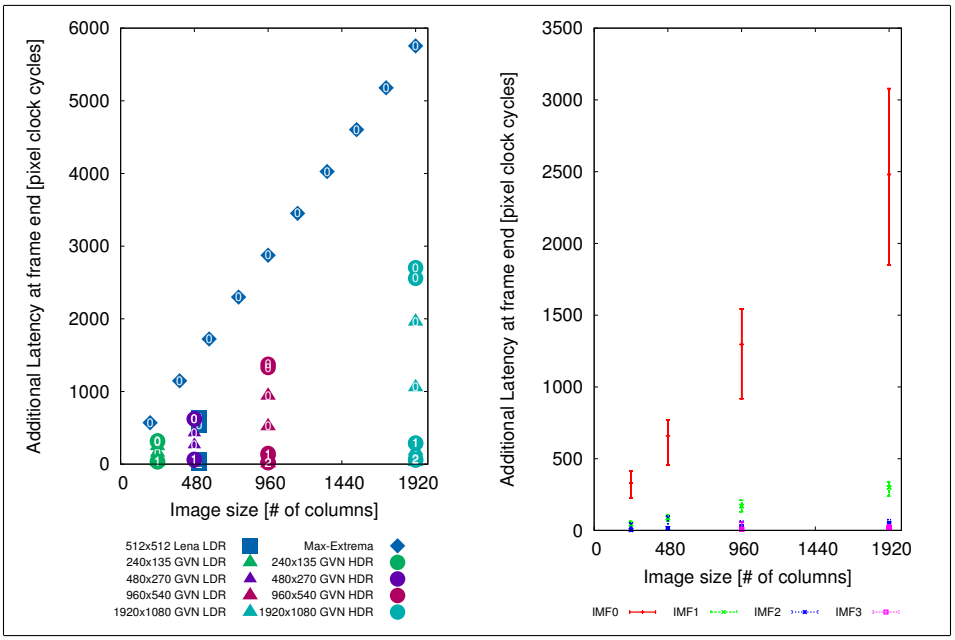


Figure 6.8: Additional latency at end of a frame. The left graph shows the actual values for Lena, a max extrema image and the GVN scene in different resolutions and two dynamic ranges. The right graph shows the statistical analysis of two sets of images (described in text), depicting the min max and average value for different IMFs. The number of the IMF is depicted in the symbol for the left graph. The number of the IMF is depicted as a color for the right graph. IMF 0 is the original image, IMF 1 is the first IMF etc.

taken for the simulation the additional latency will be the latency of two lines for the worst case.

As both graphs show, the higher order IMFs will usually require less additional latency. This is explained by the fact that empirical mode decomposition will filter out the highest frequencies in the first IMF iteration, leaving less extrema in the residue for next iteration[6]. Table 6.4 shows a summary of the latency for the filter size estimation. If we assume that it takes one pixel clock to calculate the distance, which is not unreasonable if the pixel clock is much smaller that the system clock, we can generate a

Block	*Dependency*	*Latancy in pixel clock*
Walk all pixels	resolution	columns x rows (frame)
Assign local extrema to worker	fixed	1
Calculate distance to next point	implementation	1 (only if pclk << clk)
Sort known nearest neighbors	fixed	1
Finish active worker	fixed	1
Reduction	resolution	$3 \cdot$ columns
Total Filtersize Estimation	columns & impl.	$frame + 4 + 3 \cdot columns$

Table 6.4: Latency of the Filter Size Calculation step(s)

pipelined.

[6]It must be noted however, that if artifacts are created in the envelope generation, the number of extrema can actually grow between IMFs.

table for the induced latencies as given in table 6.5. At very high pixel clock rates, the calculation time for the distance might be higher, but this will not substantially alter these figures, because the additional latency will still be orders of magnitude smaller than the total frame time.

Cols	Rows	Latency (pclk)	Latency (frame)
240	135	33124	1.022
480	270	131044	1.011
960	540	521284	1.006
1920	1080	2079364	1.003

Table 6.5: Worst Case Latency of the Filter Size Calculation for different resolutions. Here we see that the additional latency per frame becomes less for higher resolutions.

6.3 Envelope Generation

We will now have a look at the envelope generation. To start with, we will look at the amount of workers needed, which is data dependent.

6.3.1 Varying the number of workers

Because the quality of the result will directly be influenced by the number of used workers, it is interesting to see what happens to the quality if we vary the number of workers. Figure 6.9, shows a decomposition of *Lena* with only 32 workers available

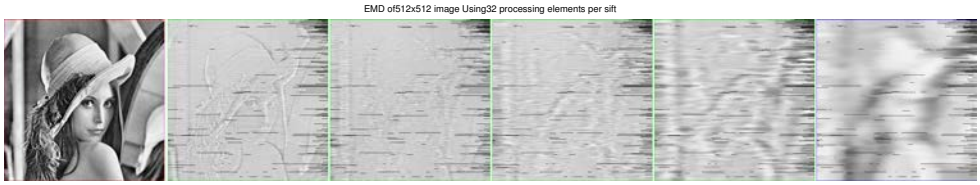


Figure 6.9: EMD of Lena (512x512) with 32 workers and 2 sifts per IMF. In this case, artifacts prohibit the EMD form functioning correctly.

As can be seen, 32 workers is not enough to correctly decompose this image. In the first IMF we see horizontal artifacts and even black bars. The gray horizontal artifacts (difficult to see in the printed version of this thesis) are caused by recycling workers while they were still contributing, resulting in a large step in the envelope contribution. The black bars are pixels for which no extrema was contributing at all, due to the workers being recycled for extrema at the other side of the image. The artifacts of the first IMF spill over to the next IMF iterations and will accumulate with new errors, making a proper decomposition impossible.

Confidence Window Multiplier

The (minimum) required number of workers for a correct decomposition is not a fixed value, since it depends on the sigma. Also, if we change the resolution of a processed image, the number of workers has to change as well, to preserve quality level. To get comparable results between different resolutions and different contents (e.g. different scenes or IMFs), it is necessary to create a equation that will calculate a normalized number of workers for a given image size and extrema density.

If we look at how the confidence window is made as described in section 3.4.3, we see that the confidence window is stretched over a consecutive part of the stream, which is streamed in a raster scan fashion; i.e. from top left to bottom right, line by line. This means that the confidence window, as depicted in figure 6.10, will have an aspect ratio (will always contain the full width of the image) and will only vary in height. Since the sigma calculated with the nearest neighbor search algorithm gives an average distance between the extrema, we can use the sigma to express a normalized region of interest (ROI), which we will only have to compensate for the aspect ratio of the confidence window. To do this, let us consider a square ROI with sides the length of one sigma, which would on average contain n local extrema[7]. A ROI the width of the image and

[7] n thus stands for the average number of extrema in a square with sides sigma. n will not be exactly

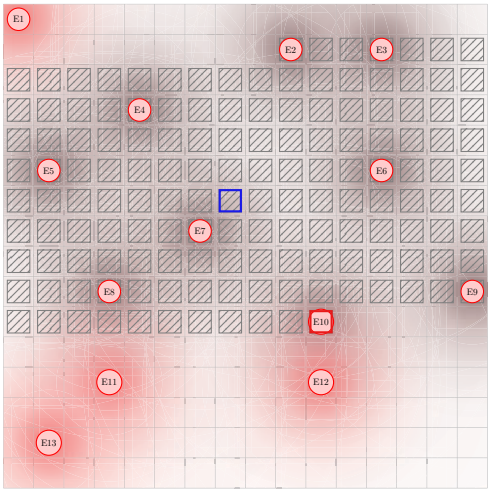


Figure 6.10: The Streaming Confidence Window

the height of one sigma will contain $\frac{\#columns}{sigma} \cdot n$ extrema. This ROI now describes a confidence window the with of the image and a height of one sigma, and thus gives us a way of calculating the number of workers used for any resolution and sigma, if we want to keep the same number of extrema included in the vertical direction. The number of workers in the confidence window normalized to the sigma and the image size will thus become:

$$CW_{norm} = \frac{\#columns}{sigma} \cdot n \tag{6.2}$$

To asses the quality of the used number of workers we will use a multiplier called K_{CW} to express the size of the confidence window. Since the average number of workers per square with side sigma, (n), is a constant with a value approximately one, we will leave this out of the equation[8]. The normalized equation for the number of workers in the FIFO will thus become:

$$Number\ of\ Workers = \frac{\#columns}{sigma} \cdot K_{CW} \tag{6.3}$$

Theoretically we can thus say, that if the middle pixel of the confidence window should on average see one extreme above and below in an image with evenly distributed extrema, the K_{CW} should be slightly larger than two.

To study the effect on changes in resolution and dynamic range a number of images was processed with $K_{CW} = 1$ to compare the artifacts, as shown in figure 6.11. If we look GVN scene, there seems to be a consistency in the produced artifacts between different resolutions for $K_{CW} = 1$. Most of these artifacts are produced by the sudden stop in contributions for workers that are recycled, as explained before. Since the effect is line based it scales to the size of the pixels, i.e. it becomes visually smaller for a higher resolution image. The dynamic range does not seem to effect the artifacts, as would be expected. Note that the artifacts are hard to see if the results are printed on paper.

1 because the ROI is square and not round, and because the sigma is an estimated average.

[8]If we change the way sigma is calculated, we should compensate the results of tests with K_{CW} in the same way that n was changed, if we want to compare results.

IMF0 of 240x135 LDR GVN scene with $K_{cw} = 1$.

IMF0 of 240x135 HDR GVN scene with $K_{cw} = 1$.

IMF0 of 480x270 LDR GVN scene with $K_{cw} = 1$.

IMF0 of 480x270 HDR GVN scene with $K_{cw} = 1$.

IMF0 of 960x540 LDR GVN scene with $K_{cw} = 1$.

IMF0 of 960x540 HDR GVN scene with $K_{cw} = 1$.

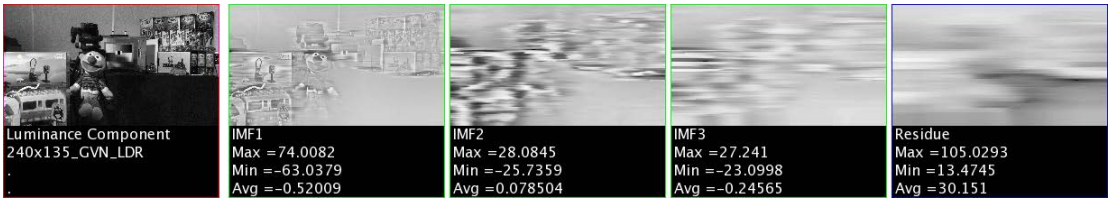
IMF0 of 1920x1080 LDR GVN scene with $K_{cw} = 1$.

IMF0 of 1920x1080 HDR GVN scene with $K_{cw} = 1$.

Figure 6.11: IMF 0 with $K_{CW} = 1$ for different resolutions and dynamic ranges of the GVN scene. This figure shows that the artifacts produced by different resolutions and dynamic ranges, using different numbers of workers, but all with $K_{CW} = 1$, lead to consistent artifacts. This shows K_{CW} is a good measure for the used number of workers if we want to compare images of different resolution and dynamic range. Note that the errors are very small so they will not clearly show if this page is printed on paper.

Varying the K_{CW}

If we increase the number of workers, by increasing K_{CW} from one, we increase the precision of the result because we take more surrounding extrema into account. Figure 6.12 shows a set of decompositions of a low resolution, low dynamic range version of the GVN scene and 6.13 and 6.14 show the decompositions for a high resolution, in low and high dynamic range.



EMD of GVN scene with $K_{CW} = 1$

EMD of GVN scene with $K_{CW} = 3$

EMD of GVN scene with $K_{CW} = 5$

EMD of GVN scene with $K_{CW} = 10$

EMD of GVN scene with $K_{CW} = 60$

Figure 6.12: EMDs varying KCW of the LDR 240x135 GNV Scene

As we can see, the low resolution version decomposed with $K_{CW} = 1$ is full of artifacts and a version decomposed with $K_{CW} = 60$ is (more) correct. We also see that the "correct" EMD has only two IMFs. This is due to the fact that a low resolution image contains less high frequencies by definition. It is only because of the artifacts at $K_{CW} = 1$ that we get an additional IMF. If we only look at the first IMF, we already see quite good results at $K_{CW} = 3$, but the residue still looks erroneous, which can have

multiple reasons. First of all, the iterative nature of the IMFs, introduces the errors of the first iteration into the second iteration. This especially holds when we do multiple sifts. Secondy, if we have a small K_{CW}, we run more risk of the K_{CW} not being centered around the pixel we create the envelope for. This will cause clear artifacts. At last we see that the decomposition at $K_{CW} = 10$ is already almost similar to the decomposition of $K_{CW} = 60$, including the residue.

EMD of GVN scene with $K_{CW} = 10$

EMD of GVN scene with $K_{CW} = 15$

EMD of GVN scene with $K_{CW} = 60$

Figure 6.13: EMDs varying KCW of the LDR 1920x1080 GNV Scene

In the high resolution, low dynamic range decomposition we see that an acceptable decomposition, including the residue, starts to be possible with a K_{CW} of 15 and at high dynamic range with a K_{CW} of 20. Interestingly, it seems that if we only need the residue, we might as well process the image at a much lower resolution, because this does not seem to change the results for the residue that much. This also suggests that it might be possible to subsample in between IMFs to reduce the number of workers used. Even more so because the higher order IMFs should contain less high frequencies. Although this is not researched as a part of this thesis, this might be one of the things we should look at in the future.

EMD of GVN scene with $K_{CW} = 15$

EMD of GVN scene with $K_{CW} = 20$

EMD of GVN scene with $K_{CW} = 60$

Figure 6.14: EMDs varying KCW of the HDR 1920x1080 GNV Scene

If we plot the standard deviation[9] between the produced IMF for a given K_{CW} and one made with a very large number of workers (i.e. $K_{CW} = 60$) we get the graph of figure 6.15, which shows that for an increasing K_{CW} the difference between the images drops exponentially for the first IMF(please note the graph has a log scale).

[9]This is the SD function originally used by Huang et.al. as given in equation 2.5

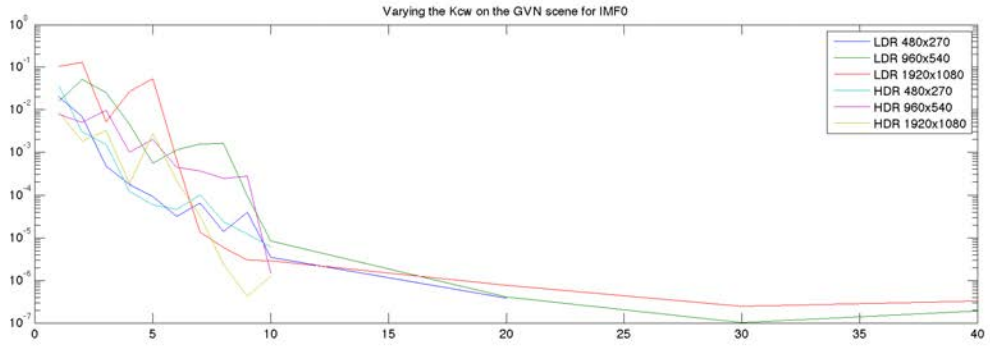


Figure 6.15: Varying K_{CW}, showing the standard deviation between an IMF produced with a given K_{CW} and one with $K_{CW} = 60$

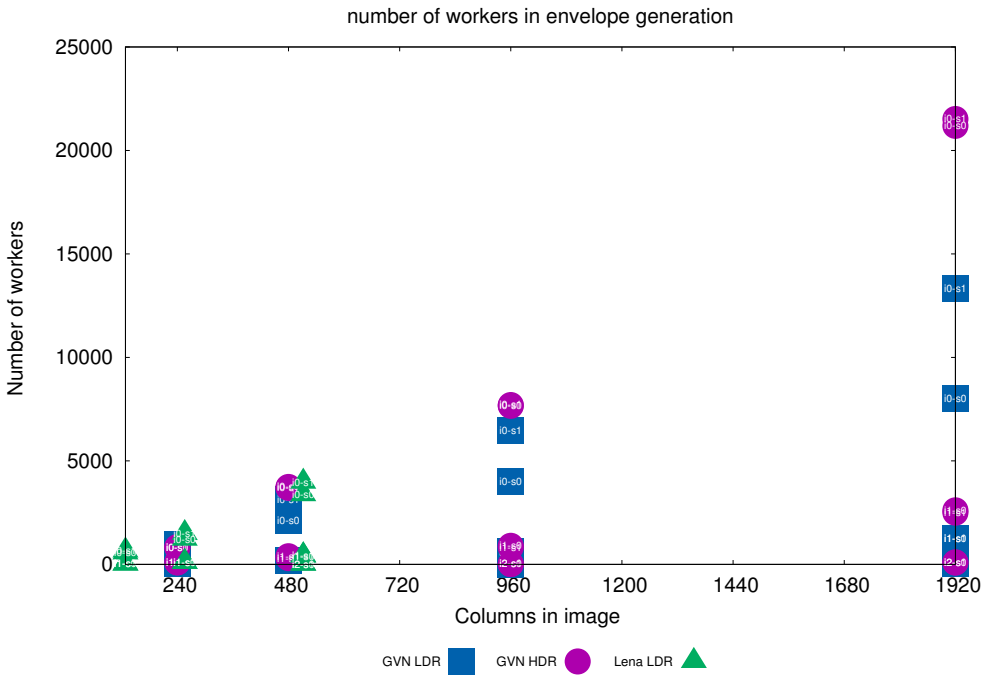


Figure 6.16: The number of workers needed for generating an IMF of comparable results for different resolutions

6.3.2 The needed amount of workers

If we look at the number of physical concurrent workers that are needed for the decomposition of a streaming video signal, and if we take the above knowledge for K_{CW} into account, i.e. most of the decompositions we can do with $K_{CW} = 15$ [10], we can create the graph of figure 6.16, and table 6.6.

Here we see that the first IMF iteration will need the most workers, and subsequent IMFs will need less. Furthermore, the number of workers will grow linearly to the number of columns, with exception of the cases where we changed the K_{CW}. The table below gives an overview for the total number of workers needed for a complete EMD.

Image	K_{CW}	*IMF0-S0*	*IMF0-S1*	*IMF1-S0*	*IMF1-S1*	*IMF2-S0*	*IMF2-S1*	total
GVN LDR 240	9	636	942	51	46	0	0	1675
GVN LDR 480	15	2100	3154	211	172	0	0	5637
GVN LDR 960	15	4008	6454	614	566	0	0	11652
GVN LDR 1920	15	7998	13338	1258	1248	73	61	23976
GVN HDR 240	7	834	814	96	97	0	0	1841
GVN HDR 480	15	3742	3694	394	304	0	0	8134
GVN HDR 960	15	7660	7698	896	826	16	16	17112
GVN HDR 1920	20	21188	21518	2610	2498	91	94	47999
Lena 128	15	552	646	18	17	0	0	1233
Lena 256	15	1218	1492	92	93	0	0	2895
Lena 512	15	3368	3972	400	420	10	15	8185

Table 6.6: Total amount of physical workers needed for the implementation of a full EMD video processing pipeline.

The amount of workers needed for a high quality Full HD HDR EMD will be in the order of 50000 workers, without any additional optimization. We can see that the second sift of an IMF, will cost about the same, so if we can half the number of sifts, i.e. use one sift in stead of two, we can gain a factor of two in the used workers.

If we look at the reuse of the workers, i.e. the number of local extrema divided by the number of used workers, we get table 6.7. Here we see that the efficiency gained by reusing workers is mostly present in the first IMF iterations and at higher resolutions. This is logical because the larger resolution images with a large number of extrema will have a relatively small confidence window, which enables more reuse. Interestingly the current K_{CW} is so large that this reuse is less than was initially expected by the author. If we want to improve the reuse we can primarily focus on finding a way for using a smaller confidence window without the loss of quality.

Next to the efficiency by reusing workers that fall outside the confidence window, there is a potential to reuse workers inside the confidence window; e.g. we could have a single worker calculate the result of multiple extrema, within the time there exists between two pixel clock ticks. For now this was not included in the simulation or the results, but a theoretical estimation can be made, using the known pixel frequency and the maximal frequency at which we can calculate the exponential contribution, for the intended FPGA, as will be discussed in section 6.3.3. Note that these figure are estimates, because they do not take into account any other limiting factor for the max frequency, then the suggested exponent calculation. The results are shown in table 6.8.

Here we see that the temporal reuse with the suggested exponent calculation, will not help for Full HD images at a frequency of 60 frames per second. However, it shows

[10] Full HD HDR will need $K_{CW} = 20$ and 240x480 will need $K_{CW} = 9$ for LDR and ,$K_{CW} = 7$ for HDR. Quality was visually estimated based on the residues.

Image	*IMF0-S0*	*IMF0-S1*	*IMF1-S0*	*IMF1-S1*	*IMF2-S0*	*IMF2-S1*
GVN LDR 240	2.49	4.09	1	1	-	-
GVN LDR 480	3.19	4.73	1	1	-	-
GVN LDR 960	6.05	9.35	1.43	1.32	1	1
GVN LDR 1920	11.98	18.79	3.22	3.01	1	1
GVN HDR 240	6.00	5.64	1	1	-	-
GVN HDR 480	5.54	5.36	1	1	-	-
GVN HDR 960	11.00	10.92	2.05	1.86	1	1
GVN HDR 1920	16.80	17.13	3.44	3.17	1	1
Lena 128	1.74	1.90	1	1	-	-
Lena 256	3.57	4.09	1	1	-	-
Lena 512	8.44	9.98	1.53	1.40	1	1

Table 6.7: Worker reuse for a full EMD; the figures in this table show how often a physical worker was reused during the processing of a specific frame.

Rows	Cols	Freq	Pixelclk(MHz)	Max Freq Exp	Max Temporal Reuse
240	135	25	0.81	248	306
240	135	30	0.972	248	255
240	135	60	1.944	248	127
480	270	25	3.24	248	76
480	270	30	3.888	248	63
480	270	60	7.776	248	31
960	540	25	12.96	248	19
960	540	30	15.552	248	15
960	540	60	31.104	248	7
1920	1080	25	51.84	248	4
1920	1080	30	62.208	248	3
1920	1080	60	124.416	248	1

Table 6.8: Theoretical Temporal Reuse. This table shows that for low pixel clock rates, workers can be reused in time for the amount of times shown in this table.

that if we lower the frame rate, or if we reduce the resolution, we could gain significant reuse in time multiplexing the workers.

6.3.3 Precision of the exponential function

As shown in figure 3.2, the envelope is calculated by dividing the accumulation of contributions for each pixel by the accumulative applicability for each pixel. They are both based on the gaussian value, which exponentially drops to zero for large distances. If the calculated value for the gaussian does become zero, we get into problems; i.e. we will start dividing by zero. The gaussian is calculated with the equation:

$$\phi(r) = e^{-\frac{r^2}{2\sigma^2}} \tag{6.4}$$

This means that the maximal distance r for which we can calculate a contribution is directly related to the smallest value for which we can calculate the exponent. A quick test in the c++ simulator showed that the standard double result from the $exp()$ function in $math.h$ becomes zero for $exp(-745)$. If we thus express the distance in terms of σ we get:

$$r < \sqrt{2 \cdot 745 \cdot \sigma^2} \approx 38.6\sigma \tag{6.5}$$

If we have al relatively large amount of workers and a small sigma, this will happen with increasing probability, because the confidence window is stretched over the full width of the image. A full HD image has 1920 columns, which means that we would have this situation for sigmas smaller than approximately 50 in the given example. The sigma in the initial image is usually below 10, so this will be a very common problem for high resolution images. If we catch these situations and just generate a zero as contribution, we might not experience any consequences, if enough extrema are around to give a minimal contribution for each coordinate in the image. This is due to the fact that *close by* extrema contribute exponentially more to level of the envelope and thus it is only the few closest extrema that will actually matter. It might however be possible that, in a very unevenly distributed point cloud, a situation would arise that a coordinate does not get a contribution at all. This would then lead to holes in the envelope, with clear visual artifacts. To analyze the effect of the precision on the results, the number of contributions for each pixel in the image was measured for different ranges of the exponent function, as explained in the next paragraph.

Effect of the exponential precision on EMDs

If a pixel obtains zero contributions, as shown in figure 6.17, a correct decomposition becomes impossible, because we get holes in the envelopes of the IMFs. It is therefore evident that a minimal range and precision of the exponent function is required. We will thus investigate the relation of the range handled by the exponent function to the resolution and the dynamic range of an image.

Figure 6.17: First IMF of 240x135 HDR image of the GVN scene with the exponent range $(\inf, -1]$, with holes in the envelope.

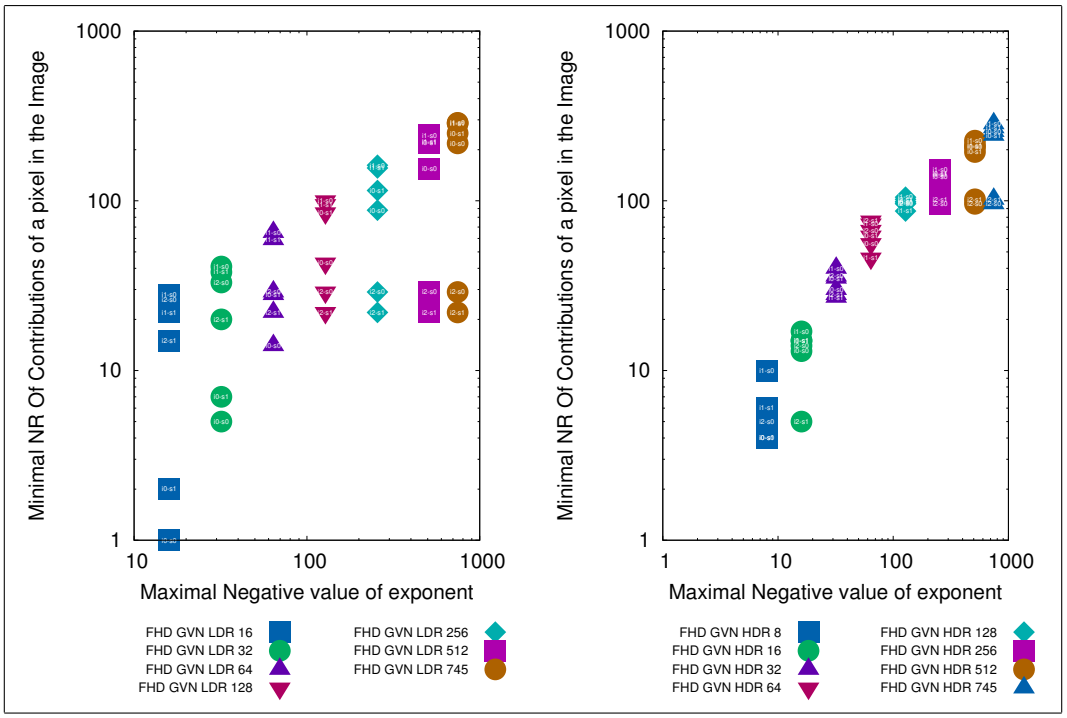
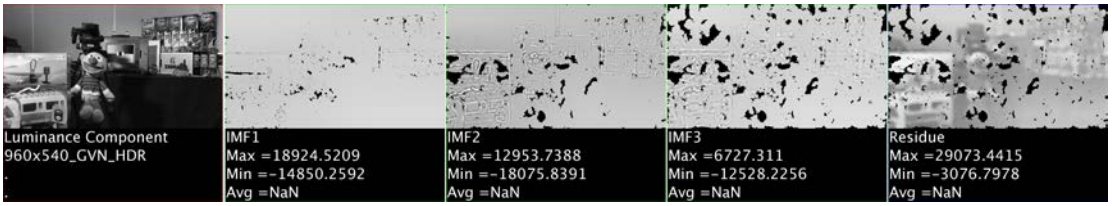


Figure 6.18: Number of contributions for the pixel with the minimal number of contributions in the image, for varying exponent ranges at FullHD (1920x1080) resolution. The left graph shows the LDR test results and the right graph shows the HDR test results.

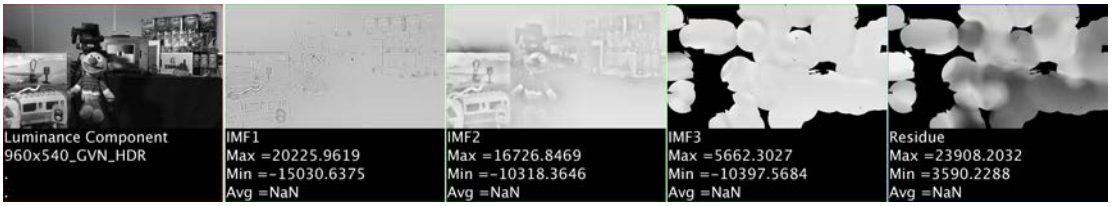
In figure 6.18 we see two graphs that show the relation between range of the exponent function and the number of contributions for the pixel that has the minimal number of contributions in the image. For the maximal negative exponent value, expressed as $e^{-(value)}$, the value was limited to the figure at the x axis. For all the pixels in the image, the number of contributing workers was recorded and the minimal value is given on the Y axis. So we tested exponent ranges where we stopped processing exponents at the levels of $e^{-1}, e^{-2}, e^{-4}, e^{-8}, e^{-16}, e^{-32}, e^{-64}, e^{-128}, e^{-256}, e^{-512}$, and e^{-745} as shown on the x-axes. For this test, the confidence multiplier [11] was fixed to 15 and we processed three IMFs with two sifts per IMF. In the icon of the graphs the number of the IMF and the Sift are depicted. We see that the minimal contribution grows roughly linear with the exponent range for IMF0, Sift0, which is the first iteration. For higher order IMFs and Sifts we see a less strict but up going trend. The leveling of the number of minimal contributions for higher order IMFs and large exponent ranges is due to the maximal number of possible contributions for these IMFs. We were only able to process images if the minimal contribution was greater then zero, which was at e^{-16} for the LDR image and e^{-8} for the HDR image. This is logical if we consider that the first sigma for the LDR version is 7.2 and the first sigma for the HDR version is 3.6. If we look at equation 6.4, this means that the value of $e^{-(value)}$ will be smaller for the HDR case. It must be noted that for the minimal case, the quality of the results is not good enough, as we will see next. The effect does not appear to change if we vary the resolutions, so the above also holds for the low resolution version of the GNV image.

In figure 6.19 we see a number of decompositions for different ranges of the exponent calculation. Remarkably the quality of the residue is stable for the whole EMD from e^{-32} onwards. This can be directly seen in the figures for average, min and max value as depicted under the IMFs and residues. Of course, the needed precision will be dependent on the data in the image, but this will not effect the range in orders of magnitude. Effectively this means that we can limit the precision of the exponent calculation to reduce the area costs and latency.

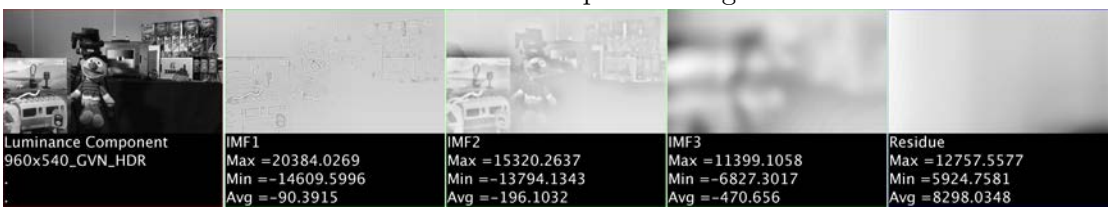
[11] See section 6.3.1



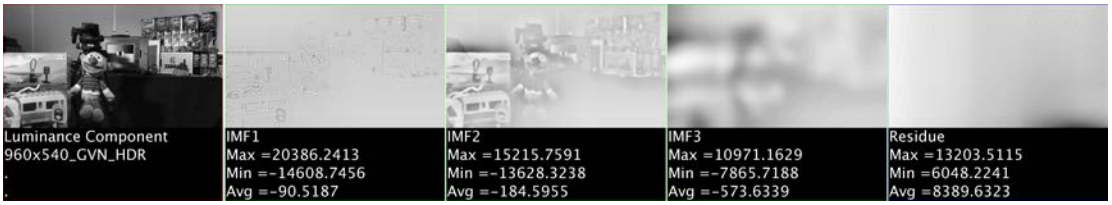
EMD of GVN scene with the exponent range limited to e^{-1}



EMD of GVN scene with the exponent range limited to e^{-4}



EMD of GVN scene with the exponent range limited to e^{-8}



EMD of GVN scene with the exponent range limited to e^{-16}

EMD of GVN scene with the exponent range limited to e^{-32}

EMD of GVN scene with the exponent range limited to e^{-64}

Figure 6.19: EMDs of HDR 960x540 GNV Scene with varying precision in the exponent function

Effect of the exponential precision on area

Altera offers a floating point exponent function which can be instantiated using the MegaFunction Wizard. With this MegaFuction, it is possible to build a exponential function of which the information is showed in table 6.9, 6.10 and 6.11 as given by the

documentation[12].

Precision	*Mantissa Width*	*Latency (in clock cycles)*
Single	23	17
Double	52	25

Table 6.9: Latency Options for Each Precision Format

For the Stratix IV that we intend to use the, the following logic utilization[13] is given:

Precision	*Latancy*	*ALUTs*	*DLRs*	*ALMs*	*DSPs*	f_{max} *(MHz)*
Single	17	631	521	448	19	248
Double	25	4104	2007	2939	46	279

Table 6.10: Resource Utilization and Performance

ALUTs	*DLRs*	*ALMs*	*DSPs*
424960	424960	212480	1024

Table 6.11: Resource Availability EP4SGX530KH40C2

If we look at the resources as given by Altera for the intended FPGA (table 6.11), we can conclude that the number of these functions we can implement for our FPGA is primarily limited by the number of DSPs. We can thus implement either 53 single precision exponential or 22 double precision exponential arithmetic units, before we run out of DSPs on this platform. Tests on the FPGA show that a single precision exponent calculation on the Altera platform becomes zero for $exp(-87)$ and the double precision exponent becomes zero for $exp(-708)$.

These results suggest that it is worthwhile to investigate an alternative way of calculating the exponential function. Since the range of the exponential needs to be large but the precision is not equally important (not over the full range, but only locally), it is very probable that a much cheaper exponent implementation would deliver equal results but facilitate much more workers in the FPGA.

[12] http://www.altera.com/literature/ug/ug_altfp_mfug.pdf

[13] ALUT = Adaptive Lookup Table, DLR = Dedicated Logic Register, ALM = Adaptive Logic Module, DSP = Digital Signal Processor

6.3.4 Total area requirements of the Envelope Generator Skeleton

The envelope generation worker has the following expensive operations:

1. A geometric distance calculation in integers
2. An exponent calculation in floating point (FP)
3. A FP multiplication

For the previous worker that calculates the nearest neighbor distance for the filter-size estimation, a hardware implementation was made to investigate the resources used and the effect of scaling on the synthesis software. Because we already know that for the envelope generation worker the proposed implementation is going to be orders of magnitude too large for fitting our FPGA in the numbers required, and because we can choose to develop cheap(er) exponent estimator that could solve this, an implementation of this worker was not made yet. We can however give a rough estimate on how much resources would be consumed if we would take the previous knowledge into account.

Since both workers do a distance calculation in integers, we can use the figures of the previous worker as a basis. If we take this information into account we can generate the following table of estimates:

Block	*Detail*	*ALUTs*	*Register*	*Memory*	*DSPs*
Empty Skeleton	part	2	2	0	0
Empty Kernel	part	5	5	0	0
Empty Worker	part	12	10	0	0
Empty Worker Skeleton	TOTAL	19	17	0	0
Worker Operation		987	642	0	33
1 Worker	TOTAL	1006(0.24%)	659(0.15%)	0(0%)	33(3.2%)
30 Workers	TOTAL	30180(7.1%)	19770(4.65%)	0(0%)	990(97%)
1675 Workers	TOTAL	1685050(397%)	1103825(259%)	0(0%)	55275(5397%)
47999 Workers	TOTAL	48286994(11362%)	31631341(7448%)	0(0%)	1583967(154684%)

Table 6.12: Guessed Area of the Envelope Generation Workers for 1 Sift

Note that the area estimated is without the reduction or the extra logic needed to feed and handle the workers. It is obvious that this implementation will be infeasible for even the smallest resolution that was tested.

6.3.5 Timing

As previously described in section 5.6, we have a data dependent latency in the envelope generation. Figure 6.20 gives a graphical representation of the streaming confidence window (striped boxes) and the local extrema (circles). The position at which extrema coordinates stream in, is depicted by the red square at the position of extrema ($E10$). The position where the coordinate generator is currently producing coordinates is depicted by the blue square at the center of the confidence window. The latency introduced is depicted by the blue section with the stripes from the top left to the bottom right.

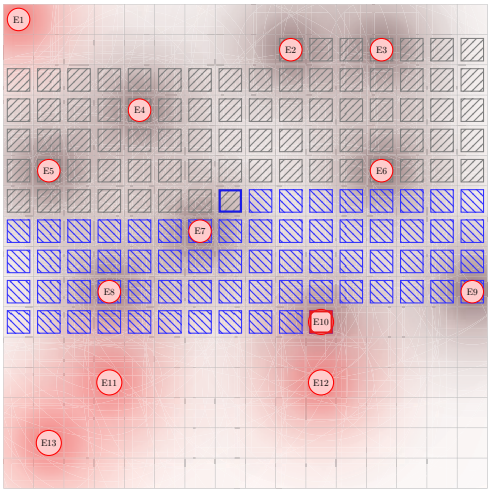


Figure 6.20: The Streaming Confidence Window

The latency between the extrema streaming in at the front of the confidence window and the middle of the confidence window where new coordinates stream in, is ideally fixed to to keep the streams in the system synchronized. In the simulator, the start of the coordinate stream is triggered by filling the worker FIFO for 50%. This basically means that the data in the top left corner of an image will determine the latency for that particular frame. After running the tests for the minimal K_{CW}, this actually showed to be quite a poor solution, because it will mean that the pixel generated might not be in the middle of the confidence window for some images. This showed itself in images that start with a blue sky, for which there were relatively few local extrema at the top of the image. To compensate for this error the K_{CW} had to be bigger than would otherwise have been necessary to keep quality to a good level for all possible images. Future tests should include an alternative for calculating the latency, which will probably lead to a reduction of the necessary number of workers in the confidence window. With hindsight it is probably best to start the coordinate stream when $\frac{1}{2} \cdot K_{CW} \cdot \sigma$ lines have processed, because we know that the height of the confidence window is $K_{CW} \cdot \sigma$. It was however not feasible to run new tests for the deadline of this thesis, because the quality tests involve the processing of large amounts of images which takes considerable time to process with the simulation.

The following results for the latency are based on the current implementation that fills the FIFO for 50% before starting the coordinate stream.

As can be seen in figure 6.21, higher order IMFs have larger sigmas and higher latencies. The maximum data dependent latency is bounded by the image size and can not grow larger than one frame.

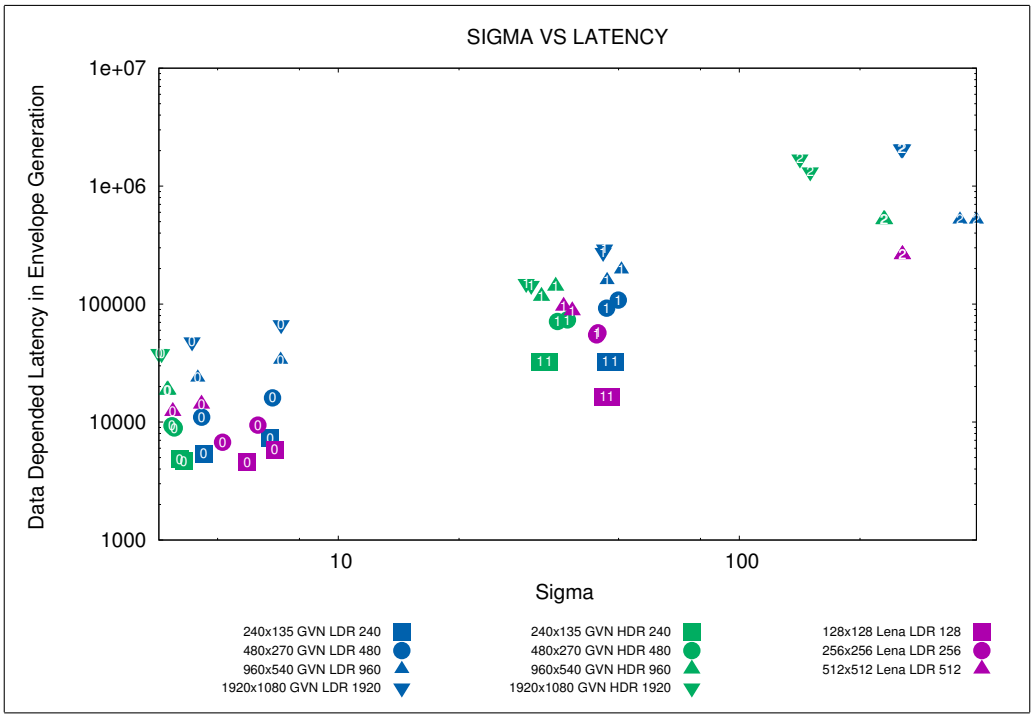


Figure 6.21: Data Dependent Latency in the Envelope Generation. The latency shown is the latency induced by the streaming confidence window for different resolutions of three images. This graph shows the relation between the sigma and the latency. Note that the sigma is measured, and the images were processed with a fixed K_{CW} of 15. The order of the IMF is shown as a number in the symbol, where 0 stands for the processing of the original image, and 1 for the first IMF etc.

If we look at the total induced confidence window latency summed over all the iterations to do a full EMD, we get table 6.13. Here we see that all images in the table stay below an added latency of three frames for a full EMD with two sifts per IMF. Note that this is purely the added latency caused by the confidence window, and the number of workers used for each iteration is based on the equation of K_{CW}, using values for the workers that produce results with comparable quality, as explained in section 6.3.1

6.4 Total System

6.4.1 Timing

If we take the whole sift block hardware model into account and make a table with all latency inducing operations except the confidence window latency, we get table 6.14. If the the resolution increases, so does the pixel clock, so for a number of operations, e.g. dividing, multiplying and exponential, we need more pixel clock ticks to get a result. These values are safe estimates based on the documentation of Altera.

Image	Columns	Rows	IMFs	Sifts	L_{maxima}	L_{minima}	Latency (frames)
GVN HDR	240	135	2	2	33352	39405	1.22
GVN HDR	480	270	2	2	162844	158283	1.26
GVN HDR	960	540	3	2	1331425	1308134	2.57
GVN HDR	1920	1080	3	2	4651156	4658603	2.25
GVN LDR	240	135	2	2	63553	72864	2.25
GVN LDR	480	270	2	2	227614	211955	1.76
GVN LDR	960	540	3	2	1449006	1443088	2.80
GVN LDR	1920	1080	3	2	4831036	4844994	2.34
Lena LDR	128	128	2	2	43192	42005	2.64
Lena LDR	256	256	2	2	128136	138552	2.11
Lena LDR	512	512	3	2	733855	749547	2.86

Table 6.13: Total measured Confidence Window latency. L_{maxima} and L_{minima} are the latencies in pixel clocks for the maxima and minima envelopes respectively. This table shows that there is a sleight difference between the two. For the latency expressed in frames the maximal value of these two was used.

Latency Type			240x135	480x270	960x540	1920x1080
Extrema Extr.			483	963	1923	3843
Filter Size						
	Walk Pixels		32400	129600	518400	2073600
	Assign Worker		1	1	1	1
	Worker Run	Distance	1	1	2	10
		Sort	1	1	1	1
		Finish	1	1	1	1
	Reduction		720	1440	2880	5760
	Averaging		1	1	2	10
Envelope gen.						
	Assign Worker		1	1	1	1
	Worker iteration	Distance	1	1	2	10
		Exponent	1	1	3	17
		Multiplicaton	1	1	2	10
	Binary Sum.		1	1	1	1
	Divider		1	1	3	14
Averaging			1	1	3	14
Subtraction			1	1	1	1
TOTAL (pclk)			33616	132016	523226	2083294
TOTAL (frame)			1.04	1.02	1.01	1.00

Table 6.14: Total Sift Latency in pixel clock ticks unless otherwise stated, excluding the confidence window induced latency from table 6.13

As we can see in the latency expressed in frames, the total latency of a sift does not change with resolution (of course the frame time is itself scaling to the resolution). We can thus say that the latency scales (roughly) linearly to the number of pixels. If we add up these figures for a full decomposition, we get table 6.15. The largest part of the total latency is caused by the sigma calculation for each sift. This can be optimized by only calculating one sigma per IMF, because the sigmas do not change much among

different sifts within one IMFs[14] If we process videos, we can also take the sigma of the previous frame. This would induce temporal artifacts in large scene changes, but it would reduce the latency further because we then do not have to wait a whole frame before we can start processing. If these ideas were to be implemented the frame latency would drop to under 3 frames for a total EMD for all the tested resolutions.

Latency Type		240x135	480x270	960x540	1920x1080
EMD	nr of sifts per IMF	2	2	2	2
	nr of IMFS	2	2	3	3
	total nr of sifts	4	4	6	6
	Total IMF Latency	134464	528064	3139356	12499764
EMD CW Latency		72864	227614	1449006	4844994
Residue Subtraction		1	1	1	1
TOTAL EMD (pixelclk)		207329	755679	4588363	17344759
TOTAL EMD (frame)		6.40	5.83	8.85	8.36

Table 6.15: Total EMD Latency. The values of table 6.14 are multiplied by the number of sifts, and then added with the maximal confidence window latency for that resolution found in table 6.13

6.4.2 Area

Concerning the area we can say that most of the hardware in the system will be consumed by the workers. As described in table 6.3 and section 6.3.4, the total needed area is much larger than what fits on our FPGA. Table 6.16 gives an overview of the biggest components needed for the total system. If we add them up taking the number of sifts into account, for the smallest and the largest resolution, we get the totals also depicted in table 6.16. Although these figures seem overwhelmingly large, they should be viewed with the reservation that they hold for the non optimal solution. They give a good benchmark to work with. As will be discussed in the section on future work, there might be the possibility to reduce the recourses with a couple of tricks that will bring a workable implementation for the smallest resolution into the order of magnitude we need for this FPGA.

[14]Boris Lenseigne, personal communication.

		ALUTs	%	Registers	%	Memory bits	%	DSPs	%
Streaming Memory Controller	Test Version	6168	1.5	6864	1.6	0	0	0	0
Extra Buffers	Working	274	0.1	74	0.0	5341184	19.5	0	0
SDI Interface	Working	1460	0.3	1619	0.4	11520	0.0	0	0
Total Basic System		7902	1.9	8557	2.0	5352704	19.6	0	0
Extrema Extraction Module	Working	119	0.0	128	0.0	30688	0.1	0	0
FilterSize Generation 1 IMF	240x135	169252	40	44500	10	0	0	1024	100
Envelope Generation 1 Sift	240x135	1685050	397	1103825	260	0	0	55275	5398
Total for 4 sifts + basic system	240x135	7425586	1747	4602369	1083	5475456	20	225196	21992
FilterSize Generation 1 IMF	Full HD	1672252	394	341760	80	0	0	1024	100
Envelope Generation 1 Sift	Full HD	48286994	11363	31631341	7443	0	0	1583967	154684
Total for 6 sifts + basic system	Full HD	299764092	70539	191847931	45145	5536832	20	9509946	928706

Table 6.16: Total EMD Area. Here we see that for a full EMD of 240x480, we need need in the order of 250 of our FPGAs and for full HD we will need in the order of 2000 FPGAs. These are very rough estimates because the figures in the table exclude a number of components like the reduction trees, and also do not include any benefits or drawbacks that can be gained using synthesis software on the whole design. Also note that the DSPs are not a good estimator, because they can be substituted with the other components.

7 Conclusions

7.1 Conclusions

This thesis started with the question if it was possible to implement a given Empirical Mode Decomposition (EMD) algorithm on an FPGA. To answer this question, a study was made in the way the EMD and BEMD (2D-EMD) algorithms work. As stated in the original paper of Huang et.al. [10], the EMD is not mathematically grounded and is defined by the algorithm producing it. Different implementations lead to (slightly) different results. A number of existing solutions were discussed and it was concluded that none of these approaches can directly lead to a real time streaming implementation. The question then became if it is possible to make a streaming hardware implementation if we change the algorithm slightly.

In chapter 3, a study was made on how to change the mathematical approach so that a streaming solution would be possible. The main bottleneck was found to be the interpolation step in creating the envelopes. As an alternative, the principle of the Normalized Convolution was investigated, which has the ability to do interpolation on a point cloud with irregularly spaced points, which is exactly the problem we faced. Moreover, this technique is able to limit this interpolation to a fraction of the whole image which enables a localized operation. Next, the rules governing the size of the locality as well as ways to decide which points should be included were investigated. As a result we found that a fixed number of points considered in a streaming FIFO manner, would probably serve the goal of streaming hardware best. Finally, this option also serves the notion of an *anyarea* solution, where the size of the FIFO is related to the hardware resources. In this way an alternative is offered for processing BEMDs that facilitates streaming localized processing, where the quality of the result scales directly with the amount of resources used.

In chapter 4, an architecture was proposed that facilitates the local operations found. For all the pixel based streaming operations, a straightforward implementation is suggested. For the filter size estimation, the envelope generation and the streaming memory buffering, a dynamic approach is suggested. The memory controller was built and tested in a separate project with support of a bachelor student that made his bachelors thesis[12] on the subject. The other two dynamic designs are based on a new design

that was called the *worker model*, which presented itself as a natural solution in the previous chapters. Since this concept of the worker model is dynamic with data dependent results, it would have to be tested, so a simulator was written that simulates the worker model structure as described in chapter 5.

It was recognized that the worker model represents a new species of Algorithmic Skeleton[6], so Skeletons were used to investigate reusability and build an abstract design description. Using the skeleton description and the design from chapter 5, the simulator was written in C++. This simulator tests the functionality of the worker model approach to explore the feasibility of the idea and the effects of varying the number of workers as well as their precision for different resolutions of an image. Chapter 5 also explains how the simulator was made and how test data is collected.

The results of the simulator on real images are discussed in chapter 6. First the simulator was used with extreme settings to optimize for quality. The results of the produced EMD were compared to the closest known EMD algorithm in Matlab, which is used in our lab. Concluding from these results we can say that the quality of this EMD approach is very good. However, because the resources in the simulator were maximized, this approach would not be implementable in an FPGA (at this moment). Therefore, aspects of the system that influence latency and area were investigated, to see how the number of workers and their precision would affect quality, latency and area. Furthermore, one worker was implemented in Verilog, and tested on the FPGA, to investigate the area usage and latency on our FPGA.

From this we can conclude that with the current design, it will not be possible to make a streaming EMD implementation on the proposed FPGA[1], for any of the tested resolutions[2], at a feasible frame rate starting with 25fps. For a 240x135 resolution it is estimated that we would at least need in the order of 250 of our FPGAs to get to a working result. For full HD this would be in the order of several thousands. There are, however, some interesting ideas to drastically reduce the number of workers, some of which are suggested in the previous chapters, which might allow an implementation in the future. These ideas are further discussed in section 7.2.

7.2 Algorithm Optimizations

First of all, a number of easy improvements can be made to optimize the used number of workers in the system. Since the solution scales mostly to the number of columns in the image, we can reduce the number of columns by rotating the image by 90 degrees before processing, and rotating it back when it streams out. This will approximately reduce the used resources by half (depending on the aspect ratio), while adding some for the rotations. Secondly we can optimize the way the center of the Confidence Window is calculated, by basing it on the filter size instead of on the fullness of the buffer. This will probably reduce the number of workers needed for the same quality by a factor of two. Next, the exponential function that is now proposed in the envelope generation is too precise for its purpose. We can probably reduce the used recourses of this function, which would probably reduce the used resources for this worker by by a factor of three. Since we use so many of them, they will have a large impact on the total used area, which will roughly be halved.

So if we implement these three optimizations we can have a total reduction in the order of $2^3 = 8$.

[1] Altra Stratix IV GX530

[2] 240x135, 480x270, 960x540, 1920x1080

If the results of the first IMF are not important for the application, it is possible to subsample the image to a lower resolution before processing. If we only need the residue, a very low resolution image will already be sufficient to get to the desired result.

In the same fashion, it might be worthwhile to filter the image with a low pass filter to reduce the number of extrema in the image. The effects of the used filter on the residue should be considered carefully however, to see if they do not filter out part of the non-stationary signal of which the residue is primarily composed.

As suggested in section 6.3.2, depending on the clock rate differences between the pixel clock and the system clock there is the possibility to have workers, or their ALUs, reused in a temporal fashion. This would benefit the envelope generation the best, because it has an expensive calculation. As shown in table 6.8, this will not really help for Full HD 60fps video, but for 240x135 at 25 fps it is theoretically possible to do 300 calculations with one worker. The downside if this approach is that the workers will be more complex, so they will require more logic, reducing the gain.

7.3 Remaining Work Analysis

If we assume that the minimum viable product should be an implementation on an FPGA that can process an EMD without the need of an external computer, and if we assume that a big enough FPGA exists to hold this design, we can estimate how much work is still left to get the current design implemented and running. This is a useful baseline for calculating the development costs for a product containing any of the above mentioned optimizations. To implement the current system on an FPGA, the author expects that about 0.5 to 1 man-year of work will be needed to get the streaming memory implementation working and tested on the FPGA hardware. For the implementation of the remaining skeleton structures, 0.5 man-year of work by a experienced person should be enough. If the person is not experienced with the skeleton structure this will, at least, be 1 man-year instead. To get the system integrated with the other modules, i.e. the SDI input and the DVI output, as well as the memory controller, another 0.5 man-year is needed. This means that, to get the non optimal system running, about 1.5 to 2.5 man-year is to be invested, assuming no major set backs are encountered. To implement the proposed optimizations, some extra research with the simulator will have to be done, plus the additional implementations. The needed time for this is more difficult to judge and depends on the results from the simulator. A rough guess is that 0.25 to 0.5 man-year is needed for each simple optimization (again by an experienced programmer) and at least 0.5 man-year for making a time shared worker.

7.4 Alternative Approaches

The above approach is based on a stand alone implementation on an FPGA. The Skeleton based structures, however, enables us to run the design on multiple types of hardware. For instance, the currently implemented worker in the FPGA can be used in a *hardware in the loop* fashion. We now use a serial port to communicate with the worker, but a higher bandwidth port would be easily implemented. If we make a complete working skeleton implementation for one sift on an FPGA, we can use it as an accelerator component in the current software implementation. This will mean that the computer can do the buffering and relieves us of the need of implementing the streaming memory buffer in the FPGA. Using Skeletons, an automated design space exploration can be done in the way as was proposed by Caarls et.al. [5], if the Worker Model Skeleton is

implemented on more hardware architectures. For instance, a super computer would ideally fit the proposed worker skeleton structure.

8 Acknowledgements

It is my pleasure to thank Pieter Jonker and Martijn Wisse for having given me the opportunity of combining my work and studies for the past few years. The combination of work and study has really made my job much fun!

Furthermore I want to thank Anja, my wife, for supporting me in this busy period of working and studying.

I am also very pleased to thank all my colleagues with whom I worked with the past years. Special thanks to all the members of the team working on EMDs: Pieter Jonker, Boris Lenseigne, Mark Wijtvliet and Raul James. Also special thanks to my direct colleagues: Jan van Frankenhuyzen, Guus Liquilung and Cory Meijneke, for giving me a great work environment. And, of course, thanks to all other colleagues and PHDs from the BioRobotics lab with whom I have worked with on our robots in the past few years.

Finally I also want to thank the members of the committee: Arjan van Genderen, Stephan Wong, Wouter Caarls and Pieter Jonker, for their work reviewing this thesis.

Bibliography

[1] K. Andersson, C.-F. Westin, and H. Knutsson. Prediction from off-grid samples using continuous normalized convolution. *Signal Processing*, 87(3):353–365, 2007.

[2] R. Bhagavatula and M. Savvides. Analyzing facial images using empirical mode decomposition for illumination artifact removal and improved face recognition. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 1, pages I–505. IEEE, 2007.

[3] S. Bhuiyan, R. Adhami, and J. Khan. Fast and adaptive bidimensional empirical mode decomposition using order-statistics filter based envelope estimation. *EURASIP Journal on Advances in Signal Processing*, 2008:164, 2008.

[4] J. Burge and W. S. Geisler. Optimal defocus estimation in individual natural images. *Proceedings of the National Academy of Sciences*, 108(40):16849–16854, 2011.

[5] W. Caarls. *Automated Design of Application-Specific Smart Camera Architectures*. PhD thesis, University of Delft, Delft, The Netherlands, 2007.

[6] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman, 1989.

[7] C. Damerval, S. Meignen, and V. Perrier. A fast algorithm for bidimensional emd. *Signal Processing Letters, IEEE*, 12(10):701–704, 2005.

[8] A. Ford and A. Roberts. Colour space conversions. *Westminster University, London*, 1998:1–31, 1998.

[9] GrassValleyNederland. http://www.grassvalley.com/.

[10] Haskell. Information repository http://www.haskell.org/.

[11] N. Huang, Z. Shen, S. Long, M. Wu, H. Shih, Q. Zheng, N. Yen, C. Tung, and H. Liu. The empirical mode decomposition and the hilbert spectrum for nonlinear and non-stationary time series analysis. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 454(1971):903–995, 1998.

[12] R. James. Streaming ddr2 memory controller design in fpga. Bachelor Thesis HRO, May 2012.

[13] H. Knutsson and C. Westin. Normalized and differential convolution. In *Computer Vision and Pattern Recognition, 1993. Proceedings CVPR'93., 1993 IEEE Computer Society Conference on*, pages 515–523. IEEE, 1993.

[14] A. Linderhed. Compression by image empirical mode decomposition. In *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, volume 1, pages I–553. IEEE, 2005.

[15] Z. Liu and S. Peng. Boundary processing of bidimensional emd using texture synthesis. *Signal Processing Letters, IEEE*, 12(1):33–36, 2005.

[16] J. Nunes, Y. Bouaoune, E. Delechelle, O. Niang, and P. Bunel. Image analysis by bidimensional empirical mode decomposition. *Image and vision computing*, 21(12):1019–1026, 2003.

[17] A. Ogier, T. Dorval, and A. Genovesio. Biased image correction based on empirical mode decomposition. In *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, volume 1, pages I–533. IEEE, 2007.

[18] M. Peel, T. McMahon, and G. Pegram. Assessing the performance of rational spline-based empirical mode decomposition using a global annual precipitation dataset. *Proceedings*

of the Royal Society A: Mathematical, Physical and Engineering Science, 465(2106):1919–1937, 2009.

[19] S. Sinclair, G. Pegram, et al. Empirical mode decomposition in 2-d space and time: a tool for space-time rainfall analysis and nowcasting. *Hydrology and Earth System Sciences*, 9(3):127–137, 2005.

[20] L. Vincent. Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms. *Image Processing, IEEE Transactions on*, 2(2):176–201, 1993.

[21] Y. Washizawa, T. Tanaka, D. Mandic, and A. Cichocki. A flexible method for envelope estimation in empirical mode decomposition. In *Knowledge-Based Intelligent Information and Engineering Systems*, pages 1248–1255. Springer, 2006.

[22] Y. Xu, B. Liu, J. Liu, and S. Riemenschneider. Two-dimensional empirical mode decomposition by finite elements. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science*, 462(2074):3081–3096, 2006.

A NNSD operation in Verilog

The following code is the *nearest neighbor searcher worker operation* implementation as described in section 4.3.6 and section 5.2, implemented for the existing FPGA skeleton framework developed in our lab by *Mark Wijtvliet*. This code was written and tested in a joined effort between Mark Wijtvliet and the author of this thesis.

```
1  module Operation_WORKER_NN(Clk, Enable, OperationRunning, CycleCounter,Data_IN,Data_OUT,
       StartNextEnable);
2
3    //---------------------------------------
4    //|              Parameters             |
5    //---------------------------------------
6
7  `include "./Skeletons/Config/GlobalParameters.inc"
8
9    parameter O_CYCLECOUNTERWIDTH = 1;
10   parameter O_DATAWIDTH_IN = 8;
11   parameter O_DATAWIDTH_OUT = 8;
12   parameter SC_NEIGHBOURHOODSIZE = 1;
13   parameter SC_REDUCTION = 0;
14
15   //---------------------------------------
16   //|    Input/output declarations        |
17   //---------------------------------------
18   input Clk;
19   input Enable;
20   input OperationRunning;
21   input [O_CYCLECOUNTERWIDTH-1:0] CycleCounter;
22
23   input [(O_DATAWIDTH_IN*SC_NEIGHBOURHOODSIZE*(SC_REDUCTION+1))-1:0] Data_IN;
24   output [O_DATAWIDTH_OUT-1:0] Data_OUT;
25   output StartNextEnable;
26
27   //---------------------------------------
28   //|          Wire declarations          |
29   //---------------------------------------
30   wire [(O_DATAWIDTH_IN*SC_NEIGHBOURHOODSIZE*(SC_REDUCTION+1))-1:0] Data_IN;
31   wire [O_DATAWIDTH_OUT-1:0] Data_OUT;
32
33   wire StartNextEnable;
34
35   wire [O_DATAWIDTH_IN-1:0] Data_IN_Array [SC_REDUCTION:0][SC_NEIGHBOURHOODSIZE:0];
36
37   genvar CurrReduction;
38   genvar CurrNB;
39   generate
40     for (CurrReduction = 0; CurrReduction < (SC_REDUCTION+1); CurrReduction = CurrReduction +
         1) begin : AA_RED
41       for (CurrNB = 0; CurrNB <SC_NEIGHBOURHOODSIZE; CurrNB = CurrNB + 1) begin : AA_NB
42         assign Data_IN_Array[CurrReduction][CurrNB] = Data_IN[((CurrReduction+1)*O_DATAWIDTH_IN
             )+(CurrNB*(SC_REDUCTION+1)*O_DATAWIDTH_IN)-1:(CurrReduction*O_DATAWIDTH_IN)+(CurrNB
             *(SC_REDUCTION+1)*O_DATAWIDTH_IN)];
43       end
44       assign Data_IN_Array[CurrReduction][SC_NEIGHBOURHOODSIZE] = 0;
45     end
46   endgenerate
47
48   //-------------------------------- OPERATION SPECIFIC --------------------------
49
50     wire ExternalStopCondition = Data_IN_Array[0][0][G_COORDINATEWIDTH+G_ANNOTATIONWIDTH+
         G_VIDEOWIDTH:G_COORDINATEWIDTH+G_ANNOTATIONWIDTH+G_VIDEOWIDTH];
51     wire [G_VIDEOWIDTH-1:0] Video = Data_IN_Array[0][0][G_VIDEOWIDTH-1:0];
52     wire [G_ANNOTATIONWIDTH-1:0] Annotation = Data_IN_Array[0][0][G_VIDEOWIDTH+
         G_ANNOTATIONWIDTH-1:G_VIDEOWIDTH];
```

```
53
54      wire [(G_COORDINATEWIDTH/2)-1:0] CoordinateX = Data_IN_Array[0][0][(G_COORDINATEWIDTH/2)+
            G_ANNOTATIONWIDTH+G_VIDEOWIDTH-1:G_ANNOTATIONWIDTH+G_VIDEOWIDTH];
55      wire [(G_COORDINATEWIDTH/2)-1:0] CoordinateY = Data_IN_Array[0][0][G_COORDINATEWIDTH+
            G_ANNOTATIONWIDTH+G_VIDEOWIDTH-1:(G_COORDINATEWIDTH/2)+G_ANNOTATIONWIDTH+G_VIDEOWIDTH];
56
57      assign StartNextEnable = StopConditionReached;// 1'b1;
58      assign Data_OUT = {7'b0,StopConditionReached,DistanceTable[0],DistanceTable[1],
            DistanceTable[2]};
59
60      reg FirstCoordinate;
61      reg [(G_COORDINATEWIDTH/2)-1:0] OriginX;
62      reg [(G_COORDINATEWIDTH/2)-1:0] OriginY;
63
64      parameter DISTANCE_WIDTH = 23;          //width is ceil(log2(1920^2+1080^2)) = 23
65      parameter DISTANCE_MAXVAL = 4852800;        //1920^2+1080^2 = 4852800
66      parameter DISTANCE_TABLE_DEPTH = 3;       //how many entries required
67      parameter DISTANCE_TABLE_DEPTH_WIDTH = 3;   //how many entries required
68
69      reg [DISTANCE_WIDTH-1:0] CalculatedDistance;
70      reg [DISTANCE_WIDTH-1:0] DistanceY;
71      reg [DISTANCE_WIDTH:0] DistanceTable[DISTANCE_TABLE_DEPTH-1:0];
72      reg StopConditionReached;
73
74      wire CompareWires [DISTANCE_TABLE_DEPTH-1:0];
75
76      genvar CurrEntry;
77      generate
78        for (CurrEntry = 0; CurrEntry < DISTANCE_TABLE_DEPTH; CurrEntry = CurrEntry + 1) begin:
              GenCompareWires
79        assign CompareWires[CurrEntry] =  CalculatedDistance < DistanceTable[CurrEntry];
80        end
81      endgenerate
82
83      reg [DISTANCE_TABLE_DEPTH_WIDTH-1:0] TableInit;
84
85      initial begin
86        FirstCoordinate = 1;
87        OriginX = 12'd0;
88        OriginY = 12'd0;
89        TableInit = 0;
90        StopConditionReached = 1'b0;
91
92        for (TableInit = 0; TableInit < DISTANCE_TABLE_DEPTH; TableInit = TableInit + 1) begin
93          DistanceTable[TableInit][DISTANCE_WIDTH-1:0] = DISTANCE_MAXVAL;
94          DistanceTable[TableInit][DISTANCE_WIDTH] = 1'b1; //mark value as invalid (1 = invalid,
                0 = valid)
95        end
96      end
97
98      always @ (negedge Clk) begin
99        if (Enable) begin
100         if (OperationRunning) begin
101
102           if (CycleCounter == 1 & !ExternalStopCondition) begin
103             if (FirstCoordinate) begin
104               OriginY <= CoordinateY;
105               OriginX <= CoordinateX;
106
107               StopConditionReached <= 0;
108
109               //reset table on receiving first coordinate
110               for (TableInit = 0; TableInit < DISTANCE_TABLE_DEPTH; TableInit = TableInit + 1)
                      begin
111                 DistanceTable[TableInit][DISTANCE_WIDTH-1:0] <= DISTANCE_MAXVAL;
112                 DistanceTable[TableInit][DISTANCE_WIDTH] <= 1'b1; //mark value as invalid (1 =
                        invalid, 0 = valid)
113               end
114             end else begin
115               CalculatedDistance <= (CoordinateY - OriginY)*(CoordinateY - OriginY)+(
                      CoordinateX - OriginX)*(CoordinateX - OriginX);
116               DistanceY <= (CoordinateY - OriginY)*(CoordinateY - OriginY);
117             end
118           end
119
120           if (CycleCounter == 2) begin
121
122             //================================== determine where to write
123             if      (DistanceTable[0] >= DistanceTable[1] &
124                      DistanceTable[0] >= DistanceTable[2])
125             begin //write to entry 0
126               if (CompareWires[0] & !FirstCoordinate & !ExternalStopCondition) begin
127                 DistanceTable[0] <= CalculatedDistance;
128               end
129               StopConditionReached <= (DistanceY > DistanceTable[0]) | ExternalStopCondition;
130               FirstCoordinate <= (DistanceY > DistanceTable[0]) | ExternalStopCondition;
131             end else if (DistanceTable[1] >= DistanceTable[0] &
```

```
132                     DistanceTable[1] >= DistanceTable[2])
133             begin //write to entry 1
134               if (CompareWires[1] & !FirstCoordinate & !ExternalStopCondition) begin
135                 DistanceTable[1] <= CalculatedDistance;
136               end
137               StopConditionReached <= (DistanceY > DistanceTable[1]) | ExternalStopCondition;
138               FirstCoordinate <= (DistanceY > DistanceTable[0]) | ExternalStopCondition;
139             end else if (DistanceTable[2] >= DistanceTable[0] &
140                     DistanceTable[2] >= DistanceTable[1])
141             begin
142               if (CompareWires[2] & !FirstCoordinate & !ExternalStopCondition) begin
143                 DistanceTable[2] <= CalculatedDistance;
144               end
145               StopConditionReached <= (DistanceY > DistanceTable[2]) | ExternalStopCondition;
146               FirstCoordinate <= (DistanceY > DistanceTable[0]) | ExternalStopCondition;
147             end else begin
148               StopConditionReached <= ExternalStopCondition;
149               FirstCoordinate <= ExternalStopCondition;
150             end
151           end
152         end
153       end
154     end
155
156 endmodule
```