Ontwerp van een hardware-accelerator voor het real time berekenen van dynamische heatmaps met een FPGA

Ewout Devos

Promotoren: ing. Johan Beke, prof. dr. Steven Verstockt

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: elektronica-ICT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. Rik Van de Walle
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2015-2016

UNIVERSITEIT GENT

Ontwerp van een hardware-accelerator voor het real time berekenen van dynamische heatmaps met een FPGA

Ewout Devos

Promotoren: ing. Johan Beke, prof. dr. Steven Verstockt

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: elektronica-ICT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. Rik Van de Walle
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2015-2016

UNIVERSITEIT GENT

Voorwoord

Alvorens te starten met deze scriptie, zou ik graag iedereen bedanken die bijgedragen heeft tot de verwezenlijking ervan. Als eerste wil ik mijn ouders bedanken die mij de mogelijk gegeven hebben om deze opleiding te volgen. Daarnaast wil ik de promotoren, namelijk ing. Johan Beke en prof. dr. Steven Verstockt, bedanken voor het aanbieden van deze thesis en de hulp die ze mij gegeven hebben tijdens het jaar wanneer ik deze nodig had. Als laatste wil ik mijn collega-studenten en mijn vriendin bedanken voor de steun tijdens het jaar, de nodige ontspanning tussen de werkmomenten en het nalezen van deze scriptie.

Ewout Devos, juni 2016

Toelating tot bruikleen

Ewout Devos, juni 2016

Ontwerp van een hardware accelerator voor het real time berekenen van dynamische heatmaps met een FPGA

door

Ewout Devos

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de industriële wetenschappen: elektronica-ICT

Academiejaar 2015-2016

Promotoren: ing. J. Beke, prof. dr. S. Verstockt
Universiteit Gent: Faculteit Ingenieurswetenschappen en Architectuur
Vakgroep Elektronica en informatiesystemen
Voorzitter: prof. dr. ir. R. Van de Walle

Samenvatting

In dit werk zal onderzocht worden hoe een hardware accelerator kan gebouwd worden om dynamische heatmaps real time te berekenen. Heatmaps zijn een hulpmiddel om grote hoeveelheden data weer te geven. Dit helpt analisten om een betekenis te geven aan die data. Het omzetten van die data naar een heatmap is een rekenintensief proces waardoor enige versnelling vereist is. Vooral bij real time toepassingen zou de versnelling een groot voordeel zijn. Daarom wordt er een hardware accelerator ontwikkeld om die real time berekening te bekomen. Het algoritme wordt aangepast om ze hardware-compatibel te maken. Waar het mogelijk is, wordt er parallellisatie toegepast om de berekening nog meer te versnellen. De uitwerking werd zowel in hardware als in software getest met een dataset van 249 punten. Vanuit de resultaten kon geconcludeerd worden dat de hardware functioneel werkt, maar de versnelling gering is. De grote oorzaak daarvan is dat de parallellisatie voor vertraging zorgt en niet voor versnelling. Een mogelijke reden voor de vertraging is dat de kleine dataset in de cache van de processor opgeslagen kan worden. Daarom werd de grootte van de dataset opgedreven om de invloed van de cache op de berekening in software te testen. De berekeningstijd bleef echter evenredig, net zoals in hardware. Er kan dus geconcludeerd worden dat de cache geen invloed heeft op de berekening.

Trefwoorden: Heatmap, FPGA, Hardware accelerator, Density estimation

Hardware accelerator design of a real time FPGA based heatmap generator

Ewout Devos

Supervisor(s): ing. Johan Beke, prof. dr. Steven Verstockt

Abstract—**The main focus of this thesis is to build a hardware accelerator to calculate dynamic heatmaps for real time applications. To do this, an FPGA will be used. This article will make a comparison between an implementation in hard- and software concerning the calculation speed and the given quality of the heatmap.**

Keywords—**heatmap, density estimation, FPGA, hardware accelerator**

I. Introduction

LARGE amounts of data may be hard to analyse. A heatmap can be used to solve this problem because it gives users the opportunity to process relatively big amounts of data by having a quick look at a simple image. To convert the data into a heatmap, a density estimation should be done. To calculate such an estimation, a lot of calculations are needed when working with large datasets. These calculations might take a long time in software and that's why, in this thesis, the algorithm will be implemented in hardware to see what the possibilities are to calculate a heatmap in real time and how the required quality for a proper visualization can be maintained. The dataset which will be used, was provided by RouteYou and handles the usage of cycling nodes in the 'Vlaamse Ardennen'.

II. Heatmaps

Heatmaps are two-dimensional representations of data in which variables are displayed as colours. To create a heatmap from data, a density estimation algorithm is used. Two density estimation algorithms are investigated and compared in this thesis: the Point Density Estimation (PDE) and the Kernel Density Estimation (KDE).

A. Point Density Esimation

TThe PDE (also known as "The Naive Estimator") can be calculated using equation 1 [2]. In this equation the density estimation depends on the size n of the dataset, the bandwidth h, the weight function $w(x)$, the samples X_i and the current point x from the heatmap. For each point on the heatmap, the equation should be executed on all the samples of the dataset. The function returns $\frac{1}{2}$ if the point lies within the bandwidth range of the sample. If the points lies outside the bandwidth, the given value is zero.

$$\hat{f}(x) = \frac{1}{hn} \sum_{i=1}^{n} w\left(\frac{x - X_i}{h}\right) \tag{1}$$

B. Kernel Density Estimation

A disadvantage of PDE is that the transition between points inside and outside the bandwidth range isn't smooth enough, which may render the heatmap unclear. To avoid this problem, the KDE was introduced: the transitions are smoothened by changing the weight function $w(x)$ into a kernel function $K(x)$. There are several possible kernel functions as shown in figure 1. Notice that the uniform kernel is the same as the PDE as described in part II-A. In figure 1 is shown that the other kernels will cause a smoother transition between points inside and points outside the bandwidth.

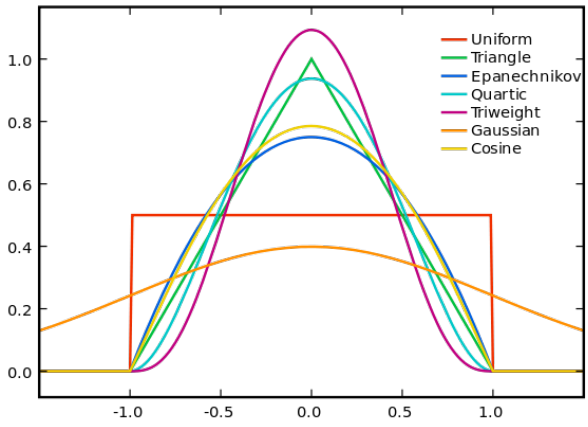


Fig. 1. Possible kernel functions [3]

The kernel functions in figure 1 can be calculated by using the equations in table I. The equations are sorted by complexity, so the uniform kernel is the easiest to implement in software as well as in hardware whereas the gaussian kernel is the most difficult to implement. The gaussian kernel contains the irrational values e and π which aren't beneficial for a calculation in hardware. The more operations the function contains, the less it is suitable for an implementation in hardware.

TABLE I
Kernel functions [1]

Kernel	K(x)
Uniform	$\frac{1}{2}$
Triangle	$1 - \lvert u \rvert$
Epanechnikov	$\frac{3}{4}\left(1 - u^2\right)$
Quartic	$\frac{15}{16}\left(1 - u^2\right)^2$
Triweight	$\frac{35}{32}\left(1 - u^2\right)^3$
Gaussian	$\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}u^2\right)$

III. Implementation in Software

A. Implementation in C

To understand and investigate the algorithm and its kernel functions, the heatmap calculation was first implemented in C. This implementation is independent of the hardware accelerator. The only purpose is to do a study on the different kernels and how they affect the outcome in terms of smoothing and readability of the heatmap.

From the kernel implementation, the triweight kernel came out as the best in terms of smoothing and readability. In Figure 1 shows that this kernel gives the highest values in the center which then quickly and smoothly transitions down to zero. However, the difference between the triangular, quartic and triweight kernel is so minimal at first sight, so the triangle kernel seems the best option for a hardware implementation because it has the least operations.

B. Implementation in Python

The algorithm was also implemented in Python. The purpose of this implementation was to transform the data to the ideal shape for hardware calculations. Python was chosen because the focus was of the implementation was implementation speed and not calculation speed.

IV. Hardware implementation

To implement the density estimation in hardware, a few changes were made to equation 1. This is shown in equation 2 where all the divisions are removed. This way, there is no need for rational values in the implementation which lowers its level of difficulty.

$$\hat{f}(x,y) = \sum_{i=1}^{n} w\left(\frac{x - X_i}{h_1}, \frac{y - Y_i}{h_2}\right) \tag{2}$$

A. First implementation

The equation was transformed into a block diagram to obtain the required functionality (figure 2). A 'data processing' block will process all the incoming data and convert it into two coordinates and a value. The 'add data to heatmap' block will then read the value at the given point of the heatmap and add the given value to it, after which the next sample can be processed. The last part is some output logic which will read the heatmap and display it using an external VGA-screen. This is a PDE where the bandwidth h is 0.

B. Adding the bandwidth

In the second part of the implementation, the bandwidth functionality was added to the density estimator. With this bandwidth, the PDE could be expanded and the KDE implementation would be possible. The concept is that a block is placed between the 'data processing' and the 'Add data to heatmap' blocks. The bandwidth block will transform the coordinates and the value into a series of coordinates and values. The series of coordinates are these in the bandwidth range. The series of values will depend on the distance of the point to the initial sample and the kernel that was used.

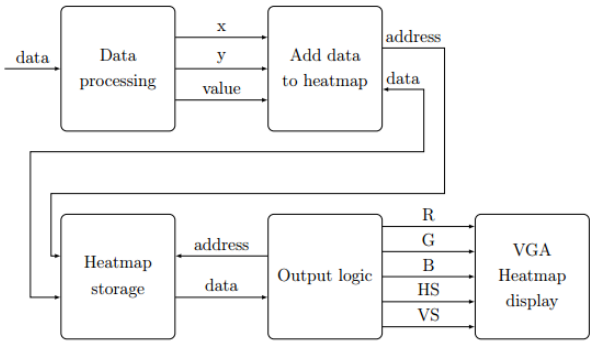


Fig. 2. Block diagram of the system

C. Parallelism

To obtain the highest acceleration, parellelism was added to the system. When a system has a certain degree of parallellisation, the system can execute multiple actions simultaneously. In the density estimator, the parallellisation can be obtained by using multiple 'Add data to heatmap' blocks. This way, multiple points can be processed at the same time. However, the heatmap storage is a true dual-port RAM where one port is used by the 'Add data to heatmap' block and the second port by the output logic. Therefore, there must be multiple instances of the same RAM. Each block will have its own RAM in which it will save the heatmap. The output logic will take a sum of all these memories to display the final heatmap.

So with a degree of parallellisation of two, there is one 'Data processing' block, two bandwidth blocks, two 'Add data to heatmap' blocks, two RAM's to store the heatmap and one 'output logic' block.

V. Results

A. Heatmap in software

When the heatmap calculation was implemented in Python, the main purpose was to have an ideal heatmap which would be used to make comparisons with the hardware generated heatmap. The result is shown in figure 3. The software was tested on three different computers. The timing is shown in table II. To time the software, there were random datasets used with increasing size. Too small datasets could be stored in the local cache memory of the CPU which would decrease the calculation time. By increasing the dataset, the calculation time increases proportionally. The cache memory has no influence on the calculation time in this implementation.

TABLE II
Software timing where bandwidth is 4, tested on different computers.

		Computer (processor)		
		HP (Intel i5)	Dell (Intel i5)	Asus (Intel i7)
Number of samples	249	0,039	0,039	0,013
	249k	29,360	29,360	13,690
	2.49M	314.720		
	24.9M	3264,088		

Fig. 3. Ideal heatmap generated in software with underlying map (bandwidth = 4)

B. Heatmap in hardware

The hardware generated heatmap (figure 4) seems to have the same functional behaviour as the heatmap generated in Python (figure 3). The positions of the points are the same, but the heatmap has an increased red colour intensity. When pushing a button to start the calculation, the calculation was faster than the release time of the button. This way, the heatmap was calculated multiple times and therefore, the heatmap has an increased intensity.

Fig. 4. Heatmap generated in hardware with underlying map (bandwidth = 4)

TABLE III
HARDWARE TIMING WITH DIFFERENT BANDWIDTHS AND DEGREE OF PARALLELLISATION (BANDWIDTH=4).

		Bandwidth			
		2	4	5	10
Degree of parallellisation	1	2,281	7,302	10,89	39,85
	1 met extra func.		60,93		
	2		80,43		
	4		40,22		

The implementation was tested with different bandwidths and degrees of parallellisation (III). There was no acceleration using parallellisation. In table IV is the hardware acceleration shown which is the hardware time compared with the timing from the different computers.

TABLE IV
HARDWARE ACCELERATION COMPARED WITH THE DIFFERENT COMPUTERS.

Computer (processor)	Acceleration
HP (Intel i5)	5,388
Dell (Intel i5)	5,388
Asus (Intel i7)	1,726

VI. CONCLUSION

The use of heatmaps is an interesting way to visualize complex and large amounts of data. Generating such heatmaps may take a long time, therefore a hardware implementation was realised. The acceleration obtained in hardware was relative small. By optimizing the developed hardware and using pipelining, a better acceleration could be obtained.

REFERENCES

[1] *Bandwidth selection in kernel density estimation: A review*
[2] Silverman, B. *Density estimation for statistics and data analysis*
[3] Vanden Eynden, K. *Design of a hardware accelerator for a real time calculation of heat maps with an FPGA*

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Gebruikte afkortingen

AMISE	Asymptotic MISE
BRAM	Block RAM
DFF	Data-Flipflop
FPGA	Field Programmable Gate Array
KDE	Kernel Density Estimation
MISE	Mean Integrated Square Error
PDE	Point Density Estimation
RAM	Random Access Memory
ROM	Read Only Memory
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VGA	Video Graphics Array

Hoofdstuk 1

Inleiding

Bij grote hoeveelheden data is het voor analisten niet eenvoudig om deze te interpreteren. Het gebruik van een heatmap is daarbij een weloverwogen hulpmiddel. Bij het ontwikkelen van zo'n heatmap wordt data omgezet naar kleurcodes. De data kan zo omgezet worden naar kaarten die eenvoudiger te analyseren zijn[6]. Heatmaps vinden hun toepassing in verschillende gebieden waaronder statistiek en sport.

Heatmaps zijn ook voordelig bij real time analyse van data. Neem nu een evenement waar men, om de veiligheid te kunnen garanderen, de mensenstroom wil bepalen. De veiligheidsdiensten willen daarvoor een continu beeld van de hoeveelheid mensen op specifieke plaatsen. Wanneer die hoeveelheden door middel van een heatmap voorgesteld worden, kunnen de veiligheidsdiensten in korte tijd zien waar veel en waar weinig volk aanwezig is. Zo kan men de snelste, lees minst drukke, weg naar plaatsen met een noodsituatie vinden of kan men tijdig ingrijpen bij overbevolking op bepaalde plaatsen.

Het opstellen van een heatmap is echter een zeer tijdrovend proces [12]. Algemeen kan gesteld worden dat er voor een heatmap met breedte b, hoogte h en een dataset met n samples, bhn berekeningen nodig zijn. Bij grote hoeveelheden data zal het voor een klassieke computer onmogelijk zijn om de data real time te verwerken aangezien deze sequentieel instructies uitvoert. Het in hardware uitvoeren van de algoritmes om heatmaps op te stellen, kan daarvoor een oplossing zijn. Veel algoritmes in de beeldverwerking hebben nood aan parallellisatie om ze veel sneller te kunnen uitvoeren. In tegenstelling tot een oplossing in software, leent hardware zich wel om meerdere blokken in parallel te laten werken [11].

In deze thesis zal onderzocht worden hoe het opstellen van heatmaps, met behulp van een hardware-accelerator, in hardware uitgevoerd kan worden. Daarbij zal een studie gedaan worden van heatmaps alsmede de statistische achtergrond die vereist is bij het opstellen ervan. Ook zullen de verschillende methodes om een hardware-accelerator te bouwen onderzocht worden. Aan de hand van een demonstrator zal gepoogd worden het voordeel van het gebruik van hardware aan te tonen. In deze thesis zal verder gewerkt worden aan onderzoek verricht door Kevin Vanden Eynden in diens thesis [13].

Hoofdstuk 2

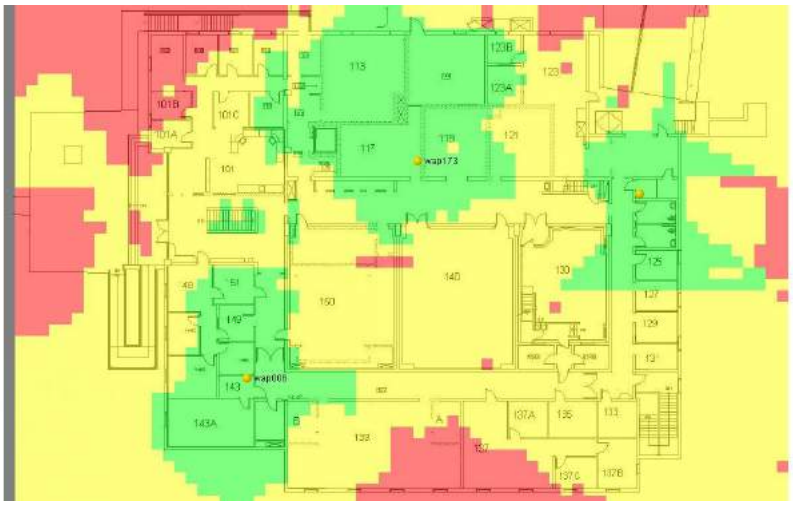
Literatuurstudie

2.1 Het opstellen van heatmaps

2.1.1 Inleiding

Heatmaps zijn tweedimensionale grafische voorstellingen van data waar de variabelen voorgesteld worden als kleuren. Ze worden gebruikt in verschillende toepassingen. Ze maken het mogelijk om grote hoeveelheden data eenvoudig voor te stellen. De weer te geven data wordt eigenlijk driedimensionaal voorgesteld. Twee dimensies om de positie weer te geven en een derde die de dichtheid weergeeft. Dit laatste wordt in een heatmap door kleur weergegeven waardoor de heatmap zelf tweedimensionaal is.

Heatmaps hebben twee grote voordelen. Een eerste voordeel is dat het gebruik van kleuren gelinkt worden met de menselijke intuïtie. Neem nu bij voorbeeld een heatmap waarin de temperatuur op een locatie afgebeeld wordt. In die heatmap kan rood voor een hoge waarde en blauw voor een lage waarde gebruikt worden, analoog als bij temperatuur waar gevoelsmatig blauw koud is en rood warm. Daardoor is een heatmap voor iemand zonder voorkennis gemakkelijk te interpreteren. Een tweede voordeel is dat de data direct in verband staat met de te meten stimulus[8]. Een voorbeeld hiervan is te zien in figuur 2.1 waarbij er een heatmap wordt gebruikt om de sterkte van het Wi-Fi-signaal voor te stellen. Merk hierbij wel op dat de gebruikte kleurschaal het inverse is van wat eerder beschreven werd. Dit komt omdat bij Wi-Fi rood gevoelsmatig gelinkt kan worden aan een slecht bereik. Zonder enige voorkennis van internet-protocollen of draadloze signalen kan



Figuur 2.1: Sterkte van het Wi-Fi-signaal in een gebouw, waarbij rood een lage waarde voorstelt en groen een hoge waarde (http://people.ucsc.edu/ warner/heat-map.html).

nu eenvoudig gezien worden waar het Wi-Fi-signaal het sterkst is.

De omzetting van variabelen naar een kleur kan als een handig hulpmiddel gezien worden. Het gebruik van heatmaps brengt ook sommige nadelen met zich mee. Heatmaps worden louter gebruikt om te communiceren met een gebruiker. Ze worden enkel gebruikt om de data voor te stellen en niet om ze uit te leggen of te analyseren.

2.1.2 Density Estimation

Een belangrijke term bij het opstellen van heatmaps is dichtheidsbepaling. Zoals eerder vermeld, worden waarden door heatmaps voorgesteld als kleuren. Dit kan zeer snel uitgevoerd worden en vraagt weinig rekentijd. De grootste rekenkracht is nodig voor het berekenen van de waarden die voorgesteld moeten worden. Meestal stelt een heatmap een bepaald gebied voor en komt de input uit een dataset. Hoe de heatmap de input zal verwerken is afhankelijk van de dichtheidsbepaling. De dichtheidsbepaling zal betekenis geven aan de data, wat het nemen van conclusies vereenvoudigt [5].

Er zijn verschillende algoritmes die men kan gebruiken om de dichtheid te bepalen. In deze thesis zullen twee algoritmes onderzocht worden, namelijk Point Density Estimation (deel 2.1.2) en Kernel Density Estimation (deel 2.1.2).

Point Density Estimation

Dit algoritme wordt ook wel "The naïve estimator" genoemd. Vergelijking 2.1 toont hoe de point density bepaald kan worden. Het principe is dat men een rechthoek van breedte $2h$ en hoogte $\frac{1}{2nh}$ op ieder punt in de dataset zal plaatsen. Al deze rechthoeken worden dan bij elkaar opgeteld om zo de dichtheidsbepaling te verkrijgen[5].

$$\hat{f}(x) = \frac{1}{hn} \sum_{i=1}^{n} w\left(\frac{x - X_i}{h}\right) \tag{2.1}$$

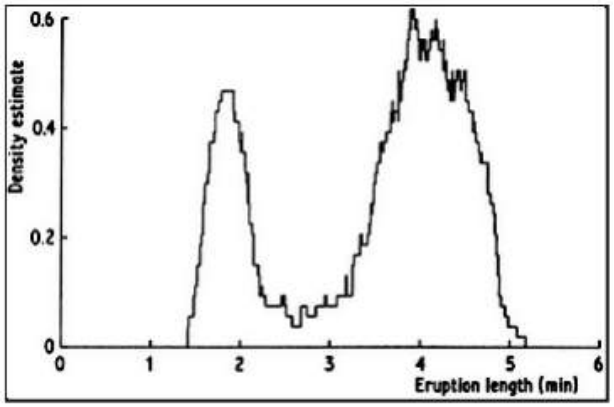
Waarbij:

$$\begin{aligned} h &= \text{Bandbreedte} \\ n &= \text{Grootte van de dataset} \\ w(x) &= \text{Weight function} \\ X_i &= \text{huidige observatie} \end{aligned}$$

$$w(x) = \begin{cases} \frac{1}{2} & als |x| < 1 \\ 0 & anders \end{cases}$$

Merk op dat de bandbreedte een belangrijke functie heeft in deze functie. Hoe hoger de bandbreedte hoe meer velden getriggerd zullen worden door een observatie. Dit zal een minder duidelijk beeld zijn, maar de overgangen zullen minder scherp zijn. Smallere bandbreedtes zullen de dataset beter benaderen, maar zullen ruwe overgangen leveren. Bij deze functie kan ook opgemerkt worden dat de rechthoeken die gecreëerd worden een oppervlakte hebben van $\frac{1}{n}$. Aangezien de dataset uit n observaties bestaat zal de totale oppervlake onder de dichtheidsfunctie gelijk zijn aan 1.

In figuur 2.2 is de data goed gevisualiseerd. Men kan stellen dat de meeste uitbarstingen ofwel twee ofwel vier minuten zullen duren. Dit komt door de twee pieken rond die tijdstippen in de figuur. In de figuur zijn er echter hoekige overgangen die als storend ervaren kunnen worden en tot een verkeerde interpretatie van de data kunnen leiden.



Figuur 2.2: Dichtheidsbepaling berekend voor de lengte van de uitbarstingen van een geiser (in minuten) met een bandbreedte van 0,25 minuten[5].

Vergelijking 2.1 berekent slechts een ééndimensionele dichtheidsbepaling. In deel 2.1.1 staat beschreven dat een heatmap dikwijls een tweedimensionele voorstelling is. Vergelijking 2.1 wordt daardoor uitgebreid naar vergelijking 2.2. Nu wordt er een balk geplaatst over de observatie. De twee bandbreedtes zullen de breedte en de lengte van de balk voorstellen. De factor $\frac{1}{4nh_1h_2}$ bepaalt de hoogte van de balk. In deze vorm zal het volume onder de gehele curve gelijk zijn aan 1.

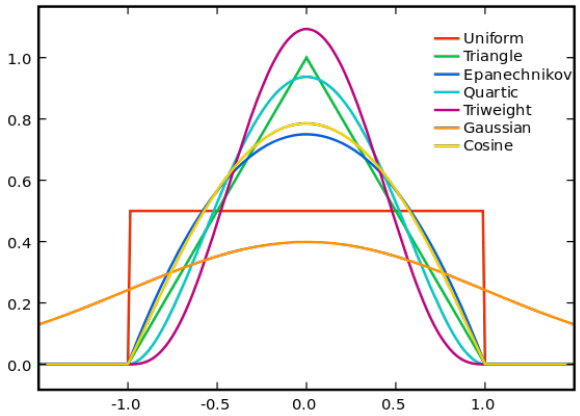
$$\hat{f}(x,y) = \frac{1}{nh_1h_2}\sum_{i=1}^{n} w\left(\frac{x-X_i}{h_1}, \frac{y-Y_i}{h_2}\right) \tag{2.2}$$

$$w(x) = \begin{cases} \frac{1}{4} & als|x| < 1 \\ 0 & anders \end{cases}$$

Kernel Density Estimation

Een nadeel van Point Density is dat het voor ruwe overgangen zorgt. Verhogen van de bandbreedte leidt wel tot vlottere overgangen, maar zorgt daarentegen voor een minder nauwkeurige voorstelling van de data. Voor die problemen biedt Kernel Density een oplossing. In principe verandert er niets aan vergelijking 2.1. De weight function wordt vervangen door een kernel function. In plaats van een rechtoek (of balk in 2D) over een observatie te plaatsen, zal men meestal een symmetrische functie boven de observatie plaatsen die gestaag naar 0 gaat om zo tot een vloeiende overgang te komen met de punten buiten de bandbreedte. Een voorbeeld van zo'n functie is de Gauss-curve (tabel 2.1). Figuur

2.3 toont nog andere mogelijke functies die kunnen worden gebruikt. Als eerste staat de uniforme kernel die dezelfde is als de eerder besproken Point Density Estimation (PDE). Daarom zal die kernel niet apart besproken worden aangezien ze al besproken is onder PDE, deel 2.1.2. Er is te zien dat de andere functies voor een vlottere overgang zorgen.

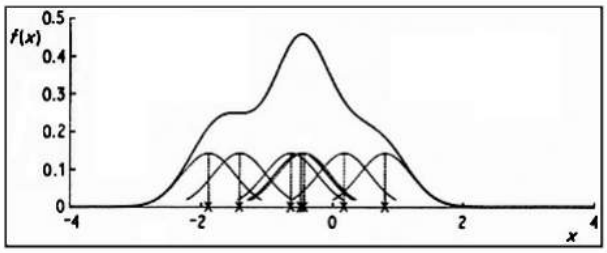


Figuur 2.3: Mogelijke kernel-functies [13].

Figuur 2.4 toont hoe verschillende kernels over observaties zorgen voor een mooie, vlotte dichtheidsbepaling zonder hoekige overgangen. De vergelijkingen van de kernels zijn te zien in tabel 2.1. Daarom is het gebruik van kernel functions een voordeel bij het berekenen van de dichtheidsbepaling. De berekening van de functies brengt wel een extra complexiteit met zich mee. Aangezien deze thesis handelt over het real time berekenen van heatmaps is het zeker aangewezen een goede afweging te maken tussen benodigde rekentijd en nauwkeurigheid.

Tabel 2.1: Kernel functies[4]

Kernel	K(x)
Uniform	$\frac{1}{2}$
Triangle	$1 - \lvert u \rvert$
Epanechnikov	$\frac{3}{4}\left(1 - u^2\right)$
Quartic	$\frac{15}{16}\left(1 - u^2\right)^2$
Triweight	$\frac{35}{32}\left(1 - u^2\right)^3$
Gaussian	$\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}u^2\right)$



Figuur 2.4: Verschillende observaties die samen de dichtheidsbepaling vormen[5].

Bandbreedtebepaling

Het is duidelijk dat de bandbreedte een belangrijke factor speelt in de bepaling van de dichtheid. Een hoge bandbreedte zorgt voor een beter beeld zonder hoekige overgangen. Een lage bandbreedte zorgt voor een beeld dat meer overeenstemt met de data, maar de slechte overgangen kunnen leiden tot verkeerde interpretaties.

Het is gewenst om een mate van afwijking te hebben ten opzichte van de ideale functie die opgesteld kan worden. Twee methodes om dit te doen zijn de Mean Integrated Square Error (MISE) (vergelijking 2.3) en de Asymptotic Mean Integrated Square Error (AMISE) (vergelijking 2.4).

$$\mathrm{MISE}(h) = E \int (f_h - f)^2 \tag{2.3}$$

waarbij:

$E =$ De verwachte waarde

$f_h =$ Opgestelde densiteitsfunctie afhankelijk van de bandbreedte

$f =$ Ideale densiteitsfunctie

$$\mathrm{AMISE}(h) = \frac{1}{hn} \cdot R(K) + h^4 R(f'') \left(\int \frac{x^2 K}{2} \right)^2 \tag{2.4}$$

waarbij:

$$
\begin{aligned}
n &= \text{Aantal observaties} \\
h &= \text{Bandbreedte} \\
R(\phi) &= \int \phi^2(x)dx \\
K &= \text{Kernel functie} \\
f &= \text{Ideale densiteitsfunctie}
\end{aligned}
$$

er wordt een bandbreedte gekozen waar bovenstaande waarden zo klein mogelijk zijn. De complexiteit van bovenstaande vergelijkingen zit hem in de ideale densiteitsfunctie. Deze functie is aan de hand van de data moeilijk te bepalen. Een eerste mogelijkheid is om die ideale densiteitsfunctie te verwijderen uit de functie. Een tweede mogelijkheid is om de ideale densiteitsfunctie te benaderen [2]. Deze technieken worden niet onderzocht in deze thesis.

2.2 Hardware accelerator

Wanneer veel rekenkracht nodig is in een korte tijd, dan gebeurt het vaak dat toepassingen in hardware uitgevoerd worden. Het grote voordeel daarvan is dat acties in hardware parallel kunnen gebeuren [3]. Die parallellisatie zorgt ervoor dat de uitvoertijd aanzielijk verminderd kan worden. Bij een processor is dit niet mogelijk aangezien deze sequentieel instructies uitvoert. Het uitvoeren van algoritmes in hardware brengt wel enige complexiteit met zich mee die er in software niet is. Daarom wordt er in de praktijk vaak gebruik gemaakt van een hardware-accelerator. Bij een hardware-accelerator zal een processor het systeem besturen en de nodige voorbereidingen treffen om de data zo efficiënt mogelijk te sturen naar het hardware-aspect. Daarbij moet er vooral gelet worden op de communicatie tussen beide delen van de accelerator. Een te trage communicatie zou de behaalde hardware versnelling te niet kunnen doen. Na de berekeningen in hardware kan de processor nog verwerking doen van de resultaten vooraleer ze naar buiten gestuurd worden. Meestal zullen delen van het algoritme die parallel uitgevoerd kunnen worden, geïmplementeerd worden in hardware. Het sequentiële deel van het algoritme wordt in software geïmplementeerd.

2.2.1 Algoritmes in hardware

Niet alle algoritmes zijn eenvoudig uit te voeren in hardware. Om de snelheid en hoge parallellisatiegraad te behouden is het aangeraden geen ingewikkelde functies te gebruiken. Ook ingewikkelde variabelen, zoals kommagetallen, worden best vermeden. Die kommagetallen zijn meestal het gevolg van logaritimische functies, vierkantswortels en trigonometrische functies. Dit zorgt dat bepaalde algoritmes benaderd zullen worden en niet volledig gelijk zullen zijn aan diezelfde algoritmes in software. Een te grote benadering van het algoritme kan zorgen voor een veel snellere rekentijd, maar leidt tot incorrecte data.

Tabel 2.2: De verschillende kernels en hun hardware-compatibiliteit [13]

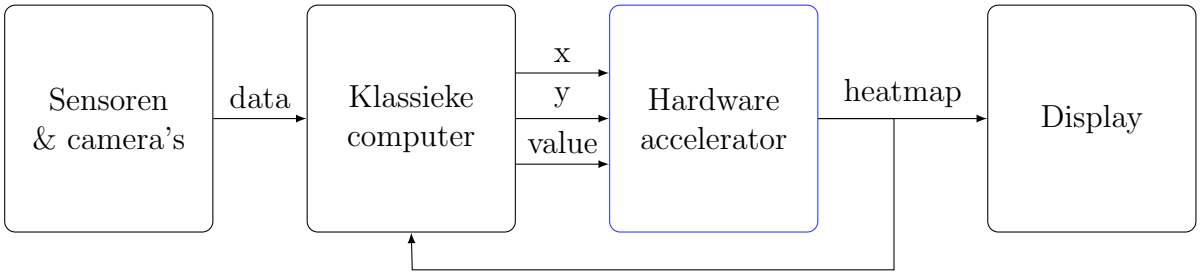
Kernel	Compatibiliteit
Uniform	++
Triangle	+
Epanechnikov	+
Quartic	-
Triweight	-
Gaussian	- -

Tabel 2.2 toont de verschillende kernels uit tabel 2.1 en de mogelijkheid om ze in hardware uit te voeren. Functies met machten zijn niet aan te raden omdat deze tot grote kommagetallen leiden. Hoe meer getallen na de komma, hoe meer geheugen de hardware nodig heeft. Exponentiële functies zijn niet aan te raden vanwege de complexe berekening. Men kan echter de functie benaderen door de waarden op te slaan in een tabel. Dit is slechts effectief wanneer er veel punten opgeslagen worden in de tabel wat tot veel geheugengebruik leidt. Er zijn wel enkele methodes om de kommagetallen in de kernel-functies te omzeilen. Dit kan door bijvoorbeeld zoveel mogelijk delingen uit de formule te halen.

In deze thesis wordt de focus vooral gelegd op de implementatie van de hardware. De algoritmes zullen in hardware op een Field Programmable Gate Array (FPGA) geïmplementeerd worden om de nodige versnelling te bereiken. De bedoeling is dat die hardware-implementatie in de toekomst eenvoudig gecombineerd worden met de software om een gehele hardware accelerator te ontwikkelen.

2.2.2 Communicatie

In het systeem is er natuurlijk communicatie nodig tussen de software en de hardware. Aangezien de focus van deze thesis eerder ligt op de implementatie van het algoritme, zal er geen communicatie gebruikt worden. Wel wordt er bekeken hoe het totale systeem er abstract uit zal zien (2.5). als eerste zijn er de sensoren en camera's die data zullen verzamelen. Die data zal verwerkt worden op een klassieke computer. Vervolgens zullen de coördinaten met waardes verstuurd worden naar de hardware accelerator. Die zal vervolgens de heatmap weergen op een scherm, of zal de heatmap terugsturen naar de computer die dan de weergave zal uitvoeren.



Figuur 2.5: Abstracte uitwerking van het totale systeem

2.2.3 Weergave van de heatmap

Om de heatmap weer te geven wordt er gebruik gemaakt van een VGA-scherm. Dit VGA-scherm kan rechtstreeks op de FPGA aangesloten worden, wat het snel tonen van resultaten eenvoudiger maakt. Het scherm heeft een resolutie van 480x640 pixels en kan aangestuurd worden via 3 kleurwaarden, namelijk rood, groen en blauw. De FPGA heeft voor deze connector respectievelijk vijf, zes en vijf bits beschikbaar. Dit wil zeggen dat er 16 bits aan kleurwaarden gestuurd kunnen worden. In totaal zijn er meer dan 64.000 mogelijke kleuren.

Figuur 2.6: Kleurschaal van de heatmap (http://eds2dspot.blogspot.be/2013/04/color-assignment.html).

Om de heatmap weer te geven wordt er gewerkt met een rood-groen schaal. Initieel is de heatmap volledig groen. Naarmate er meer data verwerkt wordt, worden sommige plaatsen roder. Deze rodere plaatsen staan gelijk aan plaatsen waar er een hogere dichtheid is. Voor de hardware is er 6 bit om de map te schalen. Iedere waarde van de heatmap zal daarom geschaald moeten worden naar een waarde tussen 0 en 63. Dit geldt echter enkel voor de groene kleur aangezien de rode kleur slechts een 5-bit resolutie heeft. Daarom moeten de waarden voor de rode kleur geschaald worden tussen 0 en 31.

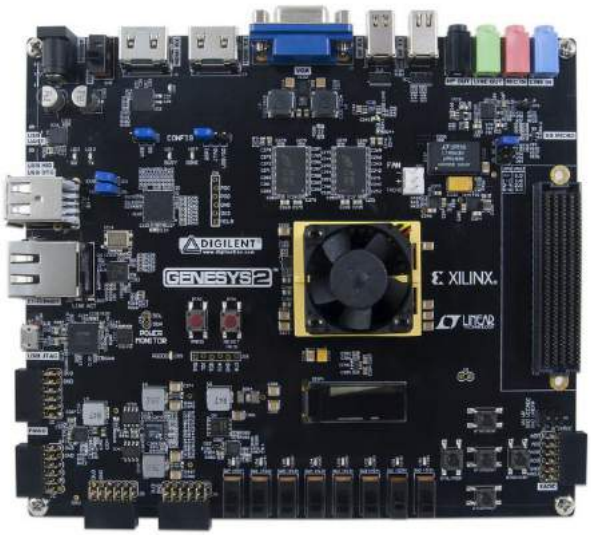
Figuur 2.7: HS bij VGA-scherm (http://fpgacenter.com/examples/vga/sync_module.php).

Om het VGA-scherm aan te sturen zijn er vijf signalen nodig. De eerste drie signalen werden al vermeld, namelijk de drie kleurwaarden rood, groen en blauw. Daarnaast zijn er twee extra signalen die de positie op het scherm regelen, namelijk Horizontal Sync (HS) en Vertical Sync (VS). Deze signalen zijn bits die op '1' staan wanneer ze in het actief gedeelte zitten van het scherm en op '0' wanneer ze erbuiten zitten. HS stuurt de horizontale lijn en door die op '0' te plaatsen weet het scherm dat het einde van een lijn bereikt is. Daarnaast stuurt de VS de verticale lijn en door die op '0' te plaatsen, weet het scherm dat het einde van het scherm bereikt is. Het scherm zal nu terug linksboven starten aan de beginpositie. Het HS-signaal zoals te zien in figuur 2.7 werkt met een front en back porch. De front porch is de tijd tussen het moment waar de kleurwaarden op 0 gezet worden, om aan te geven dat het actief deel van het scherm verlaten wordt, en het moment waar HS op '0' geplaatst wordt. De back porch is de tijd tussen het moment waar HS terug op '1' geplaatst wordt en het moment waar de volgende rij start. Bij de sturing van VS is er ook een front en back porch.

2.2.4 Gebruikte hardware

In deze thesis wordt de berekening van heatmaps ontwikkeld in hardware om een versnelling te bekomen. De daarvoor gebruikte hardware is een FPGA. FPGA's werden ontwikkeld

in de jaren ‘80 met de bedoeling te dienen als medium om aan prototyping te doen. Ze bestaan uit logische blokken die programmeerbaar zijn. Dit maakt het mogelijk voor de gebruiker om de FPGA te herconfigureren naar eigen wens. Die functionaliteit was toen zeer gewenst om ontwerpen te maken en te testen alvorens ze te gaan uitvoeren op chip. Die flexibiliteit maakt het ook een aantrekkelijk medium voor studenten, KMO’s, ...[7].



Figuur 2.8: Digilent Genesys 2 FPGA board (http://store.digilentinc.com/genesys-2-kintex-7-fpga-development-board/).

Voor dit onderzoek zal er gebruik gemaakt worden van het “Genesys 2 FPGA board” van Digilent. Dit is een ontwikkelingsbord waar een FPGA, Kintex-7, van Xilinx op geplaatst is. De FPGA voorzien van 16 Mb Block Random Access Memory (BRAM). Voor deze thesis is dit een zeer interessant type geheugen omdat het een zeer lage toegangstijd heeft [10]. Daarnaast is het voorzien van randapparatuur die ontwikkeling eenvoudiger maakt. Ook zijn er connectoren voorzien die extra randapparatuur kunnen verbinden met de FPGA om zo nog meer functionaliteit toe te voegen. Een voorbeeld daarvan is de VGA-connector die toelaat om verbinding te maken met een extern VGA-scherm. Het bord bevat ook 407.600 d-flipflops (DFF).

Wanneer deze waarden vergeleken worden met het “Digilent Atlys Board” die ook op de universiteit beschikbaar was, dan zit het grote knelpunt hem daar bij de BRAM. Er is slechts 2 Mb BRAM aanwezig op dit ontwikkelingsbord. Dit zal veel te weinig zijn voor deze thesis aangezien er redelijk veel geheugen nodig is. Wanneer parallellisatie zal toegevoegd worden zal het nodige geheugen verdubbelen, dus daarom werd gekozen om

gebruik te maken van het "Genesys 2 FPGA board".

Een FPGA heeft uiteraard ook enige nadelen. Als eerste is er de verhoogde ontwikkelingstijd ten opzichte van ontwikkelen in software. Een eerste moeilijkheid is de schaarste aan datatypes. Tijdens het ontwikkelen van VHDL kunnen er uiteraard zelf datatypes geïmplementeerd worden, maar dit maakt alles nog complexer. Ook is er nood aan bepaalde constructies die het parallel werken bevorderen om de voordelen van het gebruik van hardware te benutten. Het implementeren in hardware staat toe dat de timing van het systeem nauwlettend in de gaten kan worden gehouden. Vooral bij real time toepassingen is dit een voordeel. Dit verhoogt echter de ontwikkelingstijd. Daarnaast is het gebruik van kommagetallen niet gewenst aangezien ze veel resources vragen om ze op te slaan en de tijdsvertraging er relatief groot is. De geheugenbeperking zal ervoor zorgen dat data met zo weinig mogelijk bits voorgesteld moet worden. Daarom zal de ontwikkeling in hardware leiden tot een verlies aan kwaliteit. Uiteraard dient er een afweging gemaakt te worden tussen tijdswinst en kwaliteitsverlies.

2.2.5 Vivado Design Suite

Om toepassingen te ontwikkelen voor de FPGA wordt er gebruik gemaakt van de omgeving "Vivado Design Suite". Dit is een ontwikkelingsomgeving die door Xilinx op de markt gebracht wordt om de door Xilinx geproduceerde FPGA's te configureren. Om de FPGA te configureren wordt gebruik gemaakt van de taal VHDL. Bij het configureren van de FPGA is er een bepaalde workflow die telkens gevolgd zal worden in deze thesis.

1. Implementatie van de VHDL-code om de gewenste functionaliteit te bekomen.
2. Simulatie van de code om eventuele fouten op te lossen.
3. Generatie van een bitstream waarmee de FPGA geprogrammeerd zal worden.
4. Programmeren van de FPGA om de laatste testen te kunnen doen.

Met Vivado Design Suite zijn verschillende simulaties mogelijk:

- Een eerste simulatie is een behavioral simulation die enkel de functionaliteit van de code zal testen.

- Na de synthese kan er een tweede simulatie uitgevoerd worden, namelijk de post-synthesis functional simulation. Deze simulatie zal de functionaliteit na de synthese.
- Een derde simulatie is de post-synthesis timing simulation die rekening zal houden met de vertragingen van de componenten in het ontwerp. Zo kunnen tijdskritische punten aangepast worden.
- Na de implementatie kan er een vierde simulatie uitgevoerd worden, namelijk de post-implementation functional simulation. Deze simulatie zal het systeem functioneel simuleren na de implementatie.
- Als vijfde en laatste simulatie kan er een post-implementation timing simulation uitgevoerd worden. Deze simulatie zal terug rekening houden met de vertragingen van de componenten.

De simulaties dienen om zoveel mogelijk fouten uit het ontwerp te halen vooraleer overgegaan wordt naar de laatste stappen. Die stappen zijn het genereren van een bitstream en deze programmeren op de FPGA om zo de gewenste functionaliteit fysiek op de FPGA te testen. Wanneer er gewerkt wordt met randapparatuur kunnen eventuele fouten kritisch zijn voor die apparatuur, waardoor de simulaties van groot belang zijn.

Hoofdstuk 3

Implementatie in C

Nu de algoritmes onderzocht zijn, kunnen ze geïmplementeerd worden. Een eerste implementatie zal gebeuren in C. Deze implementatie is onafhankelijk van de hardware-accelerator. Ze wordt gebruikt om de algoritmes te verkennen en om de nood aan versnelling aan te tonen. Daarnaast dient ze ook om de correcte werking van de hardware accelerator na te gaan. Via de implementatie kan ook de versnelling van de hardware accelerator bepaald worden. Het programma zal een dichtheidsbepaling uitvoeren op een heatmap van 100x100. De gebruikte dataset bevat 100 observaties. De observaties zijn willekeurige gegenereerde coördinaten. Er zal per kernel nagegaan worden hoe vlot de overgangen zijn naar gebieden met een lagere waarde of een nulwaarde. Ook zal bekeken worden hoe vatbaar de heatmap is voor interpretatie. Die vatbaarheid maakt het mogelijk om gebieden met hoge waarden te herkennen in de figuur om zo toch snel conclusies te kunnen trekken. Die hebben echter geen betekenis aangezien het om een willekeurig gegenereerde testset gaat.

3.1 Analyse van het algoritme

Het algoritme, vergelijking 2.2, moet uitgevoerd worden op ieder punt van de heatmap. Bij een kaart met hoogte h en breedte b zal dit leiden tot bh berekeningen van de vergelijking. In de vergelijking zelf wordt de afstand gemeten tussen het huidige punt en alle observaties. Aan de hand van die afstand krijgt het huidige punt een observatie bij. Al de observaties opgeteld leveren de dichtheid voor dit punt. Indien een dataset uit n observaties bestaat, dan zal in ieder punt n keer de weight function moeten worden berekend. Dit heeft als

gevolg dat het totale algoritme evenredig zal zijn met bhn. Aangezien bij alle kernels, met uizondering van de gaussian kernel, de waarde buiten de bandbreedte gelijk is aan 0, moet de dichtheid voor een specifieke observatie daar niet berekend worden. Dit vereenvoudigd de berekening tot $h_1 h_2 n$ berekeningen om de dichtheid te bepalen. In dit deel zal gewerkt worden met de theoretische beschrijving van de functie aangezien er ook gaussian kernels verwerkt zullen worden. Bij eventuele tijdsmetingen zal de vereenvoudigde versie genomen worden daar deze ook in hardware gebruikt zal worden.

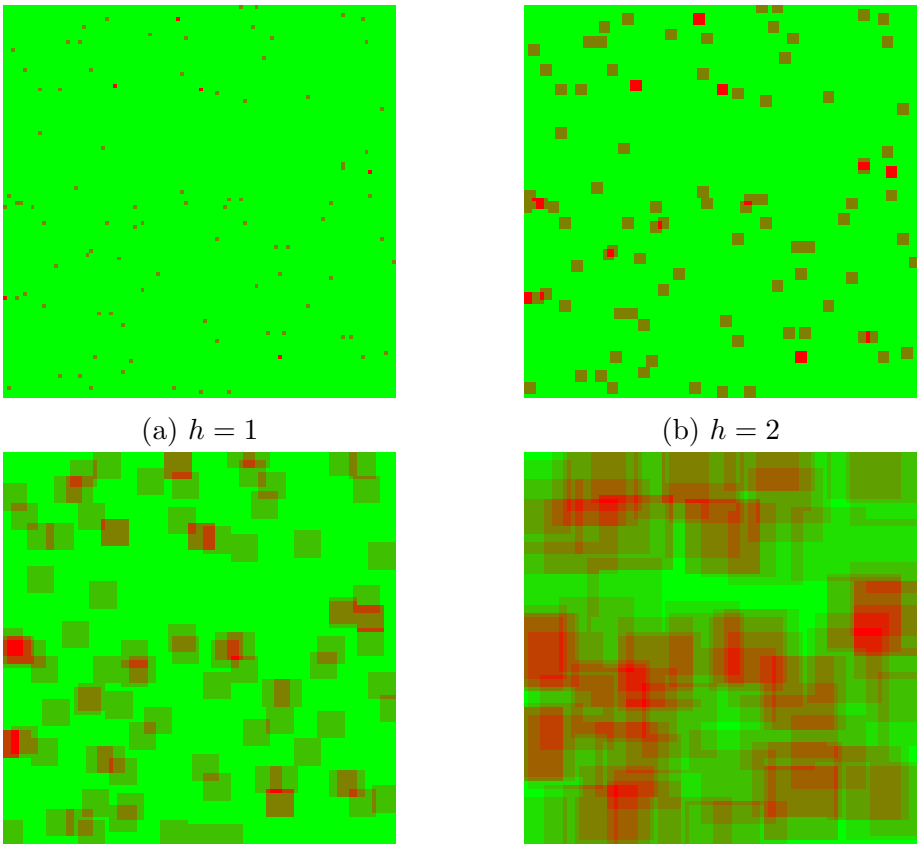
3.2 Omzetten van data naar heatmap

Wanneer het algoritme die de dichtheid bepaalt uitgevoerd is, moeten de verkregen waarden omgezet worden in een kleurcode. In figuur 2.6 is de gebruikte kleurschaal te zien. In de eerste implementaties wordt een schaal gebruikt van 0-255 in tegenstelling tot de in hardware gebruikte schaal van 0-63. Dit om zo nauwkeurigere beelden te kunnen tonen. Bij eventuele tijdsvergelijkingen zullen de specificaties zoals schermresolutie en kleurwaarden gelijk gesteld worden. Aangezien de berekende dichtheid geen waarde tussen 0 en 255 zal zijn, is er nood aan een functie die een waarde in een bepaald bereik zal omzetten naar een nieuwe waarde tussen 0 en 255. Dit is een zeer eenvoudige functie die enkel bestaat uit basisoperaties zoals optelling, aftrekking en vermenigvuldiging. Deze functie zal enkel nodig zijn in software aangezien er in hardware rechtstreeks de gevraagde bits kunnen afgetapt worden van de dichtheid.

3.3 Point Density Estimation

Het PDE algoritme is het meest eenvoudige om te implementeren. Het is gelijk aan de Kernel Density Estimation (KDE) waar de kernel een uniforme vergelijking is. Als basis is er de vergelijking 2.1.2. Als eerste wordt de afstand tussen het punt en de observatie gedeeld door de bandbreedte. Naargelang de waarde van die gedeelde afstand zal de functie een gewicht van $0,25$ of 0 teruggeven. Ieder punt zal bovenstaande functie n keer uitvoeren. Figuur 3.1 toont de resultaten voor verschillende bandbreedtes. Het resultaat ziet er abstract uit, maar dient louter om de correcte werking van het algoritme te testen. Bij realistische datasets zijn er veel meer observaties wat het vermoeilijkt om de juistheid na te gaan. Er is te zien dat een hogere bandbreedte voor vlottere overgangen zorgt. De hogere bandbreedte zorgt wel dat de data minder vatbaar is voor interpretatie. Bij deze

kleine dataset leidt de PDE nog steeds tot een beeld met ruwe overgangen.



(a) $h = 1$ (b) $h = 2$

(c) $h = 4$ (d) $h = 10$

Figuur 3.1: PDE uitgevoerd met verschillende bandbreedtes.

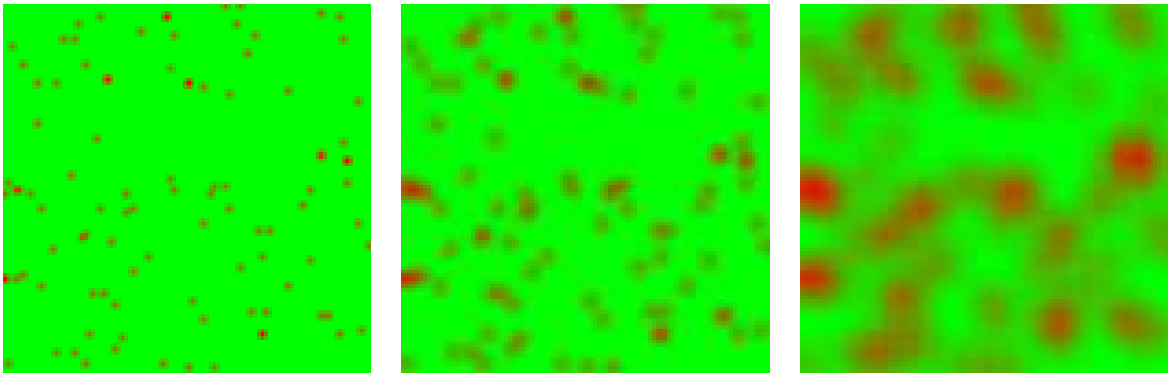
3.4 Kernel Density Estimation

In dit deel zullen de reeds besproken kernel functies geïmplementeerd worden. Door ze één voor één te analyseren zal blijken welke kernels leiden tot een beter resultaat. De resultaten worden hier niet getoetst bij een bandbreedte van 1 aangezien dit tot eenzelfde resultaat zou leiden als in figuur 3.1a.

3.4.1 Triangular Kernel

Bij de implementatie van de triangular kernel daalt de waarde lineair naarmate de afstand van de observatie vergroot. Figuur 3.2 toont de resultaten ervan. Men kan zien dat de

overgangen veel vlotter verlopen door de geleidelijke overgang naar de nulwaarde buiten de bandbreedte. Bij een bandbreedte van 10 is de heatmap nog altijd voor interpretatie vatbaar in tegenstelling tot de PDE in figuur 3.1d waar het tot een wanorde leidt.

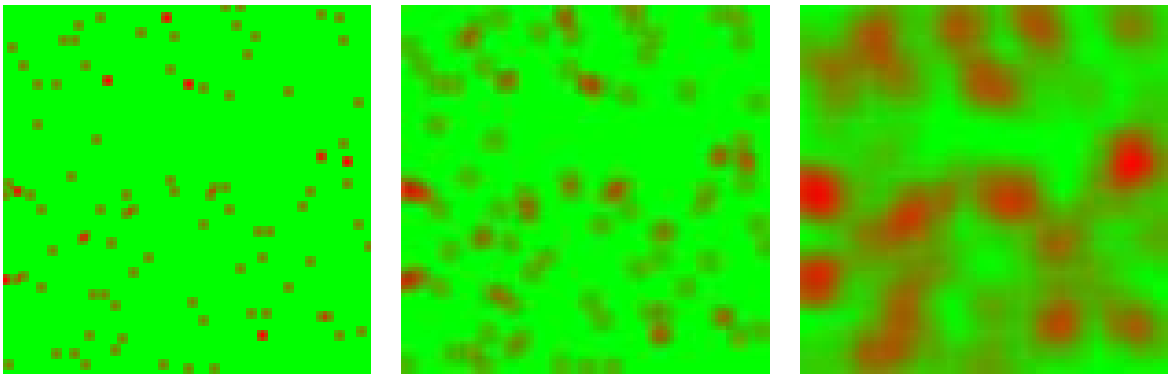


(a) $h = 2$ (b) $h = 4$ (c) $h = 10$

Figuur 3.2: Triangular kernel uitgevoerd met verschillende bandbreedtes.

3.4.2 Epanechnikov Kernel

De Epanechnikov kernel is de eerste die gebruik maakt van kwadraten. Dit kan een implementatie in hardware complexer maken. De kernel is een betere benadering voor de PDE. Er is nog steeds een gestage vermindering van de waarde naarmate de randen van de bandbreedte bereikt worden. In figuur 3.3 kan men zien dat de kernel zorgt voor een vollere kleur binnen de bandbreedte. Bij een bandbreeedte van 2 en 4 lijkt dit een voordeel in tegenstelling tot de bandbreedte van 10 waar het meer naar wanorde leidt.

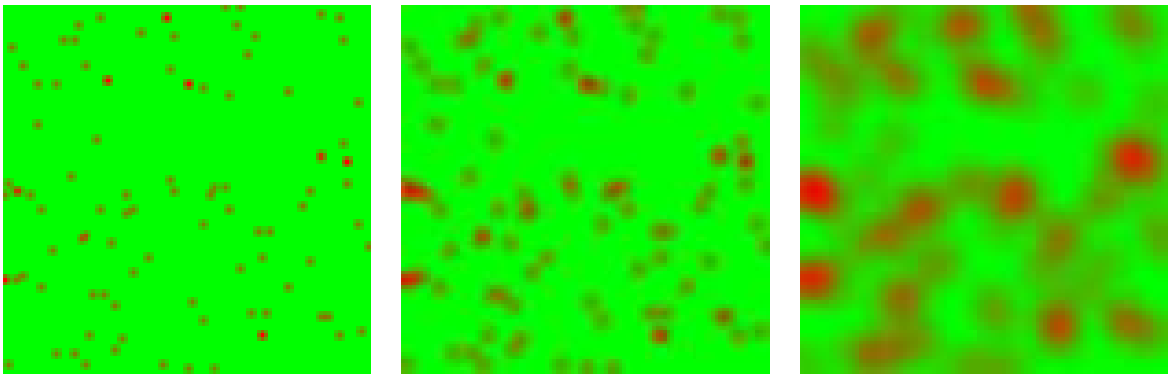


(a) $h = 2$ (b) $h = 4$ (c) $h = 10$

Figuur 3.3: Epanechnikov kernel uitgevoerd met verschillende bandbreedtes.

3.4.3 Quartic Kernel

De Quartic kernel leunt dichter aan bij de triangular kernel van deel 3.4.1. Wel leent de implementatie, hieronder te zien, zich minder voor de hardware omdat er terug kwadraten aanwezig zijn. Bij de resultaten (figuur 3.4) is het merkbaar dat dit algoritme grote gelijkenissen toont met de triangular kernel. De extra complexiteit leidt niet tot een merkbaar beter resultaat.

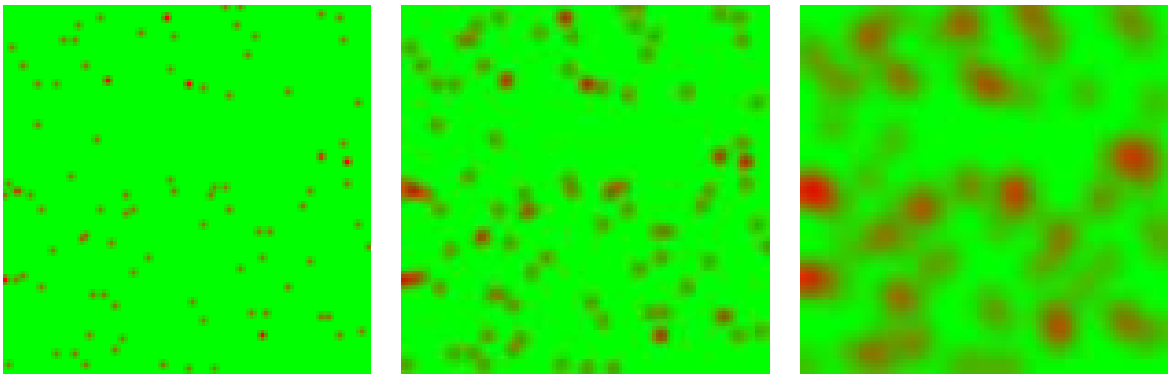


(a) $h = 2$ (b) $h = 4$ (c) $h = 10$

Figuur 3.4: Quartic kernel uitgevoerd met verschillende bandbreedtes.

3.4.4 Triweight Kernel

Deze kernel zal, binnen de bandbreedte, de centrale punten meer gewicht geven dan de buitenliggende. In alle voordien besproken kernels is dit het geval, maar de triweight kernel zal dit extreem implementeren. Het zorgt ervoor dat de overgang tussen punten binnen de bandbreedte en punten buiten de bandbreedte, die geen gewicht meekrijgen, verkleint. Dit zou voor vlottere overgangen moeten zorgen. Deze kernel zal nog meer gebruik maken van machten waardoor de complexiteit stijgt. In figuur 3.5 zijn de resultaten te zien. De aandachtige lezer kan zien dat er kleine nuances zijn die het beeld vlotter en correcter maken. Observaties die uit elkaar liggen gaan hier vlot over in elkaar, maar de verschillende observaties zijn nog zichtbaar. Bij de triangular en quartic kernel zijn de verschillende observaties soms niet meer uit elkaar te halen. Het verschil met de triangular kernel is groter dan het verschil tussen de quartic kernel en de triangular kernel, maar de verschillen zijn hier ook weer zo miniem dat er in hardware voor een triangular kernel geopteerd zal worden.

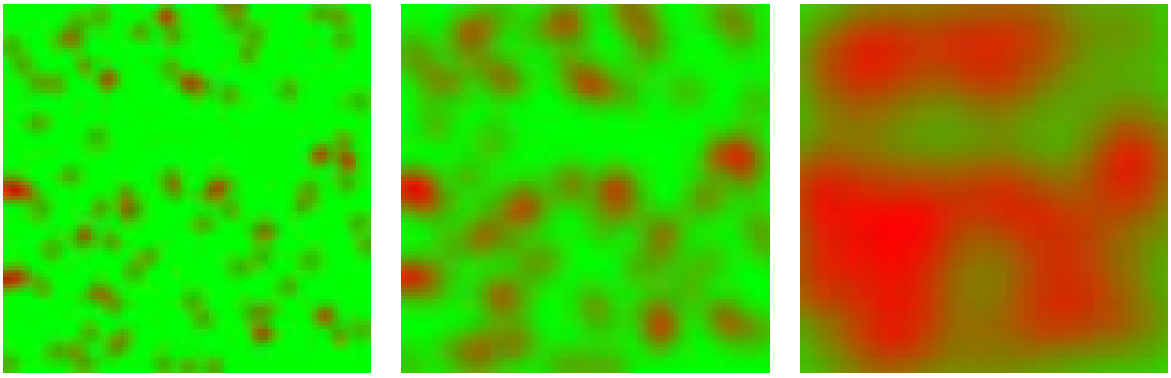


(a) $h = 2$ (b) $h = 4$ (c) $h = 10$

Figuur 3.5: Triweight kernel uitgevoerd met verschillende bandbreedtes.

3.4.5 Gaussian Kernel

De laatste kernel is de meest aparte van de besproken kernels. Waar andere kernels buiten de bandbreedte een 0 toekennen, zal deze kernel een waarde meegeven. Merk hier wel op dat deze waarde nog steeds afhankelijk is van de bandbreedte. Punten die ver liggen van de observatie zullen een waarde krijgen die benaderbaar is met 0. In deze kernel wordt er ook gebruik gemaakt van π. Om nauwkeurig te rekenen zijn er veel cijfers na de komma nodig wat niet gewenst is in hardware. Ook kan er opgemerkt worden dat er gebruik gemaakt wordt van een exponentiële functie. Om deze kernel in hardware te implementeren moeten zowel e als π opgeslagen worden met een grote nauwkeurigheid. Dit zou teveel geheugen vergen waardoor de implementatie van dit algoritme enkel in software uitgevoerd zal worden. Figuur 3.6 laat de resultaten van de gaussian kernel zien. De heatmap met een bandbreedte van 10 is onbruikbaar. De gaussian kernel werkt effectiever bij lagere bandbreedtes. Zo kan de gaussian kernel met een bandbreedte van 4 vergeleken worden met voorgaande kernels bij een bandbreedte van 10.



(a) $h = 2$ (b) $h = 4$ (c) $h = 10$

Figuur 3.6: Triweight kernel uitgevoerd met verschillende bandbreedtes.

Hoofdstuk 4

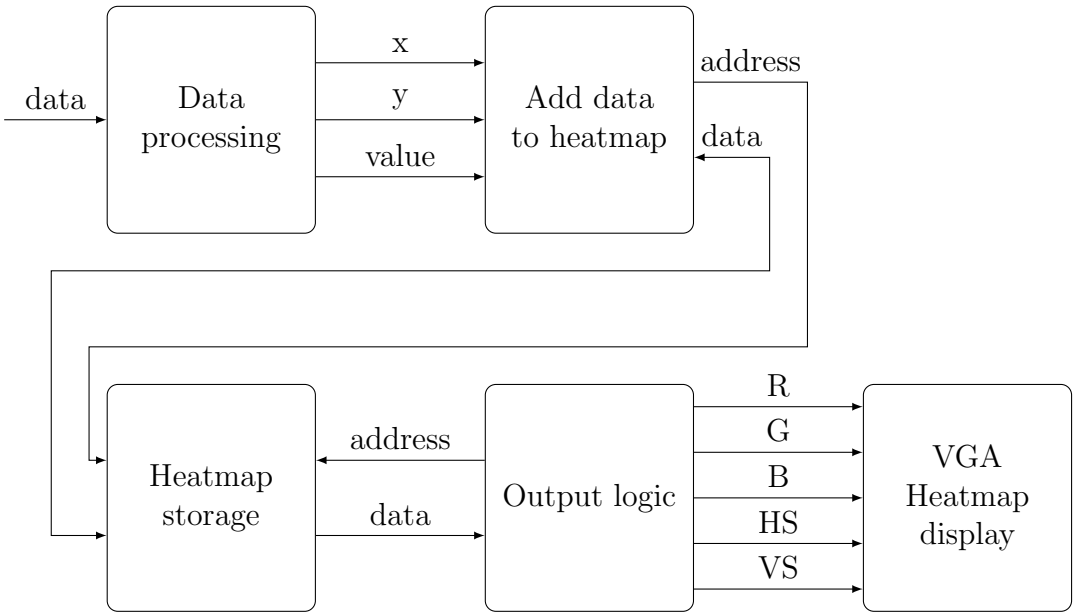
Hardware implementatie

4.1 Ontwerp van het systeem

In deel 2.1.2 werd beschreven dat de oppervlakte onder de dichtheidsfunctie gelijk zal zijn aan 1. Dit wil zeggen dat de dichtheid per punt een getal tussen 0 en 1 zal zijn. Wanneer zo'n zaken uitgevoerd worden in hardware is het gewenst om zo'n kommagetallen te vermijden. Daarom zal vergelijking 2.2 aangepast worden om ze meer hardware-compatibel te maken. Aangezien de dichtheidsbepaling bij het visualiseren geschaald wordt tussen 0 en de maximum waarde die voorgesteld kan worden met de resolutie van het scherm, is het niet nodig om de dichtheid te delen door constante data zoals de bandbreedte en de grootte van de dataset. In hardware is het eenvoudiger om die deling weg te laten en daarna te schalen naar de eventueel beperkende resolutie van het scherm.

$$\hat{f}(x,y) = \sum_{i=1}^{n} w\left(\frac{x - X_i}{h_1}, \frac{y - Y_i}{h_2}\right) \tag{4.1}$$

Om het algoritme (vergelijking 4.1) uit te werken in hardware is het gewenst deze op te delen in verschillende functionele blokken. Als eerste is er nood aan een component die het algoritme gaat uitvoeren, zijnde het opslaan van een bepaalde waarde en daarbij verschillende waarden gaan toevoegen. Daarnaast is er ook nood aan een blok dat de inkomende data gaat verwerken in de gepaste vorm te versturen naar het berekeningsblok. Als laatste is er een blok nodig dat de opgeslagen data gaat verwerken om het zo goed mogelijk te visualiseren voor de gebruiker.



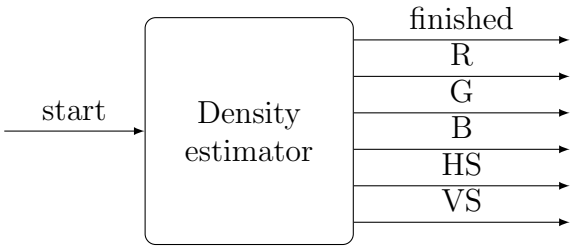
Figuur 4.1: Blokschema van het systeem.

Als eerste is er het blok 'Data processing' dat de inkomende data zal verwerken en in de juiste vorm zal zetten om door te geven aan het volgende blok. In deze thesis zal gewerkt worden met statische data. Dit om de implementatie in hardware te vereenvoudigen en te focussen op de berekening van de dichtheidsbepaling. De gebruikte data is een dataset van RouteYou die informatie geeft over de activiteiten op hun knooppuntennetwerk in de Vlaamse Ardennen [9].

Het essentiële blok van het systeem is het blok 'Add data to heatmap'. Dit blok zal de verkregen coördinaten omzetten naar een geheugenlocatie waar de waarde aan toegevoegd zal worden. Door deze actie op alle punten van de dataset toe te passen, is de dichtheidsbepaling berekend en kan de heatmap in kaart gebracht worden.

Om de data weer te geven wordt er, zoals vermeld in 2.2.3, gebruik gemaakt van een VGA-scherm. Het blok 'Output Logic' zal het beeldscherm besturen en zal zo de gewenste dichtheidsbepaling als kleurwaarde sturen naar het scherm. De gebruiker kan via dit scherm de heatmap in real time zien om zo de nodige conclusies te kunnen trekken.

4.2 Uitwerking van het systeem



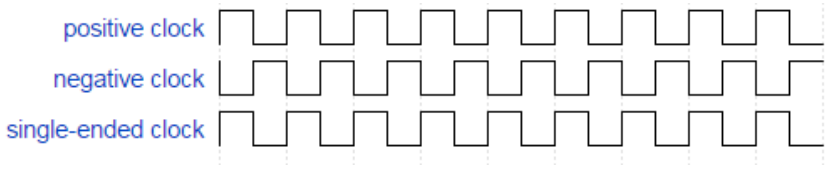
Figuur 4.2: Entity van de Density Estimator.

Het blokschema, zoals besproken in 4.1, is natuurlijk een idealistische aanpak van het systeem. In de praktijk worden er verschillende bijkomende processen en signalen aan het systeem toegevoegd. In dit deel zal de praktische uitwerking en de inhoud van ieder blok besproken worden. Het systeem dat geïmplementeerd zal worden, wordt de 'density estimator' genoemd. Als input zal er een 'start'-signaal, vergezeld van een klok en een reset, zijn. De entiteit van het systeem is in figuur 4.2 te zien. De uitgangen van het systeem zijn de signalen die naar het VGA-scherm gestuurd zullen worden. Daarnaast wordt er ook een 'finished'-signaal gestuurd die aan de gebruiker laat weten wanneer het systeem klaar is met rekenen. Dit is zeer handig bij tijdsmetingen om de versnelling te kunnen bepalen.

4.2.1 Kloksynthese

FPGA's worden veelal als oplossing gebruikt voor real time systemen omdat ze de mogelijkheid bieden om de timing van het systeem onder controle te houden. Idealiter zouden de berekeningen op de maximale werkfrequentie van de FPGA uitgevoerd moeten worden daar de klok het hart van het systeem is. De klokfrequentie is de maat voor het aantal stappen dat het systeem kan ondernemen in een bepaalde tijdseenheid. De gebruikte FPGA levert een differentiële klok van 200 MHz. Een differentiële klok (figuur 4.3) is een klok die bestaat uit twee signalen die elkaars complement zijn. De voordelen van zo'n klok zijn de verminderde gevoeligheid voor ruis en de verlaagde elektro-magnetische straling. De differentiële klok is daardoor geschikt voor hoogfrequente systemen. In deze thesis wordt er met een single-ended klok gewerkt waardoor er nood is aan een component die de differentiële klok zal omzetten naar de single-ended klok. De omzetting kan verwezenlijkt

worden aan de hand van een core die in de IP-catalogus van Vivado zit. Daar kunnen de parameters van de binnenkomende klokken en uitgaande klokken ingesteld worden.

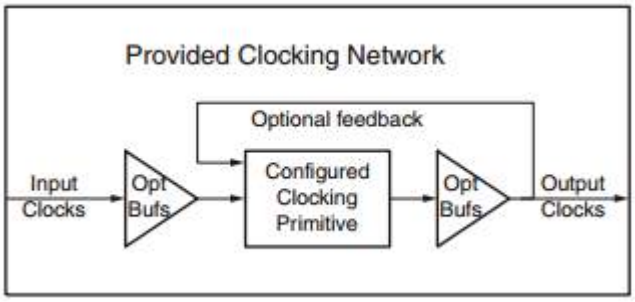


Figuur 4.3: Differentiële klok naar single-ended klok.

Tijdens het ontwerp van een systeem is het beter om eerst op een lagere frequentie te werken om zo de functionaliteit in orde te krijgen. Wanneer het systeem functioneel goed in elkaar zit, kan gekeken worden wat de vertragingen zijn om zo een maximale werkfrequentie te behalen. In deze thesis is er één frequentie die vastligt, namelijk de werkfrequentie van het VGA-scherm. Het scherm heeft een grootte van 480x640 pixels en om met zekerheid vloeiende beelden te leveren moet het scherm vernieuwen aan een frequentie van 60 Hz. Merk daarbij wel dat er in het scherm een niet zichtbaar deel is dat in rekening genomen moet worden als tijd om van het einde van een lijn naar het begin van de volgende lijn te gaan. De effectieve resolutie is 525x800 pixels. De berekeningen in vergelijking 4.2 leiden tot een frequentie van 25,2 Mhz. Dit werd afgerond naar een werkfrequentie van 25 MHz aangezien dit $\frac{1}{8}$ is van de initiële klok van 200 MHz van de FPGA. Om tot een eerste implementatie te komen, wordt er gewerkt met deze relatief trage klok in het gehele systeem. Nadien zal geprobeerd worden om het maximum uit het systeem te halen door de werkfrequentie op te drijven en de traagste blokken te versnellen. Dit zal verder besproken worden in deel 5.3.

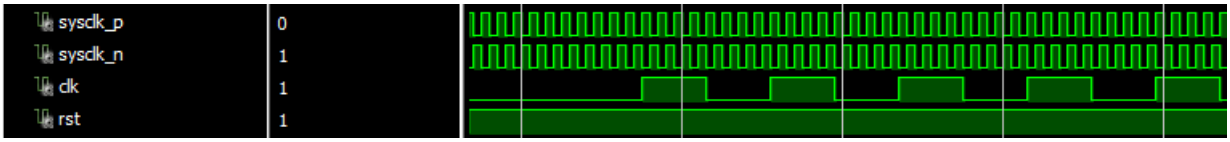
$$\begin{aligned} \text{pixelfrequentie} &= \text{Aantal pixels} * \text{schermfrequentie} \\ &= 525 * 800 * 60 \text{ Hz} \\ &= 25,2 \text{ MHz} \\ &\approx 25 \text{ MHz} \end{aligned} \tag{4.2}$$

De klokdeler wordt eerst gesimuleerd alvorens ze in de praktijk te gebruiken. Zo kunnen er al in een vroege fase van de implementatie eventuele fouten uitgehaald worden wat de implementatietijd verkleint. Zoals vergelijking 4.2 toont, zal er eerst gewerkt worden met een frequentie van 25 MHz. Dit wil zeggen dat er acht klokcycli aan 200 MHz in één periode van de systeemklok zitten. De klok werd via een blok (figuur 4.4) van Vivado



Figuur 4.4: Het door Vivado aangeboden blok voor kloksynthese

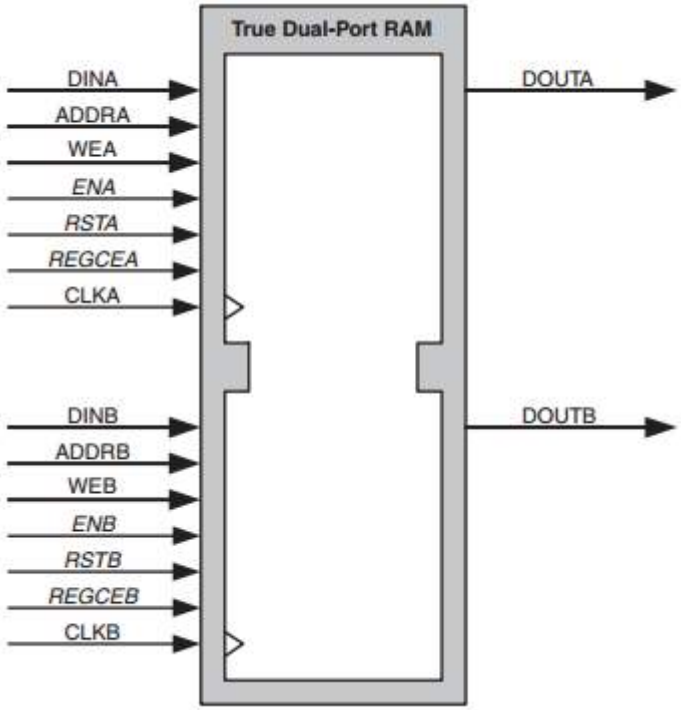
geïmplementeerd en de simulatie ervan is in figuur 4.5 te zien. In het begin is de klok continu '0'. Dit komt omdat het enige tijd duurt vooraleer de uitgang van de klok bekomen wordt na het aanleggen van een klok en het stoppen van de reset. Dit is in het systeem geen probleem aangezien de reset ervoor zorgt dat alle signalen hun startwaarde krijgen en ze enkel wijzigen op het ritme van de uitgangsklok, die hier nu stil ligt. De startwaarden zullen niet wijzigen als de uitgangsklok nog niet stabiel is.



Figuur 4.5: Behavioral simulatie van de klokdeler van 200 MHz naar 25 MHz.

4.2.2 Geheugen

Om de heatmap op te stellen is er snel geheugen nodig. Aangezien de heatmap dynamisch wijzigt, wordt er gebruik gemaakt van RAM-geheugen. Op de FPGA is er BRAM voorzien om snelle geheugens te bouwen [1]. Deze BRAM is echter in beperkte hoeveelheid aanwezig, namelijk 16 Mb. Bij de parallellisatie zal de hoeveelheid BRAM zeker in rekening genomen moeten worden. Voor ieder punt van de heatmap moet een waarde opgeslagen worden. In deel 2.2.3 werd besproken dat er gewerkt wordt met een scherm van 480x640 pixels en dat onze heatmap voor iedere pixel 6 bit aan data nodig heeft om de correcte kleur voor te stellen. Daarvoor is ongeveer 2 Mb BRAM nodig. Het systeem zou met deze hoeveelheid BRAM tot een parallellisatiegraad van 8 kunnen komen. De gebruikte BRAM zal geconfigureerd worden zodat ze functioneert als een true dual-port RAM (figuur 4.6).



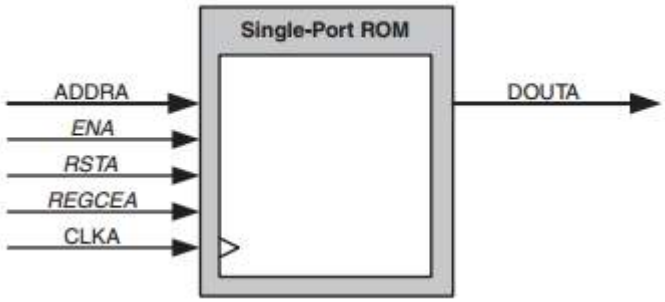
Figuur 4.6: Blokschema van een true dual-port RAM

Bij een tekort aan BRAM zijn er twee opties:

- Een eerste optie is om externe static RAM te gebruiken. Door gebruik te maken van extern geheugen kan er veel meer data opgeslagen worden wat de nauwkeurigheid van het systeem verbetert. Dat externe geheugen zorgt natuurlijk voor extra logica en meer complexiteit en het is niet zo snel toegankelijk als BRAM wat de tijdswinst kan verminderen.

- Een tweede optie is om een reductie te doen van data om zo onder de beperkte hoeveelheid BRAM te blijven. Zo blijft de complexiteit van het programma dezelfde en kan de tijd beter onder controle gehouden worden. Een nadeel is echter dat door de reductie van de data de nauwkeurigheid van de heatmap zal verminderen. Aangezien er in het huidige systeem slechts 6 bit nodig zijn om de data voor te stellen, is het onnodig om met een grotere nauwkeurigheid te werken. Daarom wordt gekozen voor de datareductie.

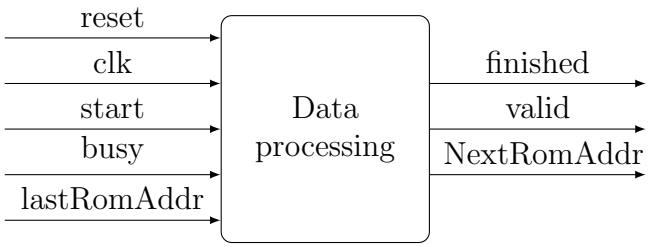
$$\begin{aligned} \text{geheugen} &= \text{Aantal pixels} * \text{bits per pixel} \\ &= 480 * 640 * 6 \text{ bits} \\ &= 1843200 \text{ bits} \\ &\approx 2 \text{ Mb} \end{aligned} \tag{4.3}$$



Figuur 4.7: Blokschema van een single port ROM

In het praktisch gedeelte van deze thesis wordt gewerkt met een vaste dataset. Om deze vanaf de FPGA te kunnen benaderen wordt ze opgeslagen in Read Only Memory (ROM) (figuur 4.7). In de praktijk is dit BRAM waarvan de write-bit standaard op '0' geplaatst wordt om deze te beschermen tegen het schrijven. De dataset bevat 249 observaties bestaande uit 2 coördinaten en een waarde. Iedere observatie neemt 1 geheugenlocatie van de ROM in beslag. De coördinaten van de dataset moeten wel nog geschaald worden naar de afmetingen van het scherm. De waarde moet eveneens geschaald worden aangezien de heatmap geen hogere waarden dan 63 kan verwerken. Voor de x-coördinaat zijn 10 bits nodig aangezien de waarde van 0 tot 640 gaat. Voor de y-coördinaat zijn slechts 9 bits nodig aangezien de hoogte van het scherm slechts 480 pixels is. De waarde zal opgeslagen worden door gebruik te maken van 5 bits. De dataset werd in software geanalyseerd en er werd een heatmap opgesteld. Om de ruimte van 6 bits zo goed mogelijk te berekenen, kan de waarde tot 5 bits groot zijn. De waarde moet geschaald worden van 0 tot 31 om tot een heatmap te leiden waar de maximale waarde kleiner is dan 64, maar groter dan 32 zodat het bereik zo goed mogelijk gebruikt wordt. Eén observatie zal daarom 24 bits in beslag nemen. De totale dataset zal vervolgens $24 * 249 = 5976$ bit BRAM innemen. Vergeleken met het RAM-geheugen is dit een zeer klein geheugen. Om de bovengenoemde geheugens te implementeren werd gebruik gemaakt van de door Vivado aangeboden IP-cores.

4.2.3 Dataverwerking

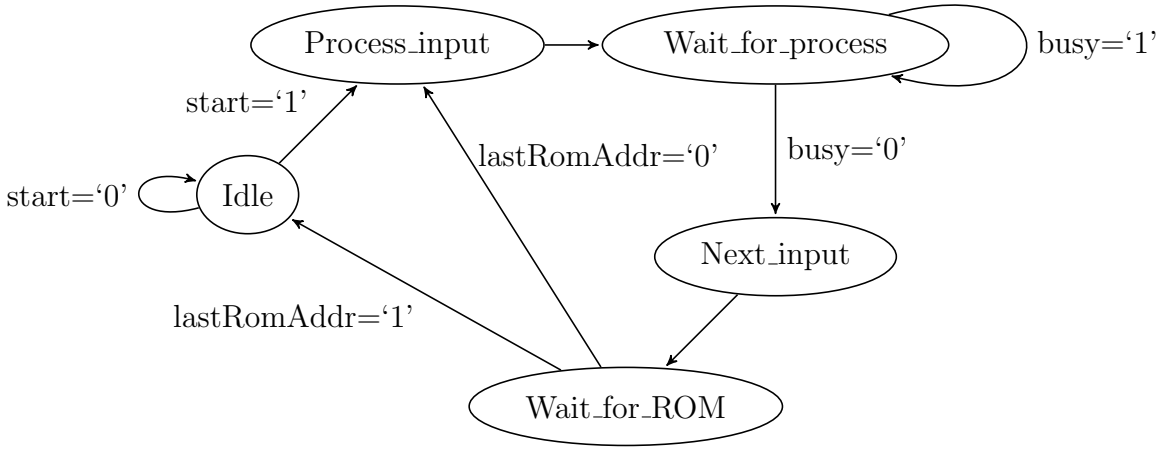


Figuur 4.8: Functionele blok van het dataverwerkingsproces.

Het eerste belangrijke blok in het systeem is het blok dat instaat voor de verwerking van de data. Zoals in deel 4.2.2 beschreven staat, wordt de data voorgesteld door 24 bits per observatie. Uit die 24 bits moeten dan een x- en y-coördinaat en een waarde gehaald worden. Daarvoor dient dit blok. Voor deze functionaliteit wordt in dit proces een FSM geïmplementeerd zoals te zien in figuur 4.9. Op deze figuur is ook te zien dat de FSM wordt gestuurd door een aantal binnenkomende signalen. De states waar er slechts één optie is, gaan sowieso in de volgende klokcyclus naar de volgende state. Het proces stuurt ook een aantal signalen die variëren naargelang de state waar de FSM zit. Tabel 4.1 toont de uitgangen van de FSM voor iedere state. Dit kunnen externe uitgangen zijn voor het gehele systeem of inwendige signalen die gebruikt zullen worden in de volgende processen. De coördinaten en de waarde worden rechtreeks uit het ROM verkregen (vergelijking 4.4). Dit gebeurt onafhankelijk van de state. Bij de correcte werking van de FSM zullen die waarden op het juiste moment de correcte waarde hebben.

$$\begin{aligned} romData_{23-14} &= x_{9-0} \\ romData_{13-5} &= y_{8-0} \\ romData_{4-0} &= value_{4-0} \end{aligned} \tag{4.4}$$

De FSM start in de ‘Idle’-state. Dat is de state waar de FSM rust. Bij de start wordt er gewacht op een ‘start’-signaal van de gebruiker. Dit wordt gesignaleerd door het ‘finished’-signaal hoog te plaatsen. Wanneer het ‘start’-commando gegeven is, gaat de FSM over naar ‘process input’. Hier gaan de coördinaten en de waarde naar het ‘Add data to heatmap’-proces. Via het ‘valid’-signaal laat het proces weten dat er geldige data verstuurd werd. Vervolgens moet het proces, in de ‘wait for process’-state, wachten tot de gestuurde data



Figuur 4.9: FSM van het dataverwerkingsproces.

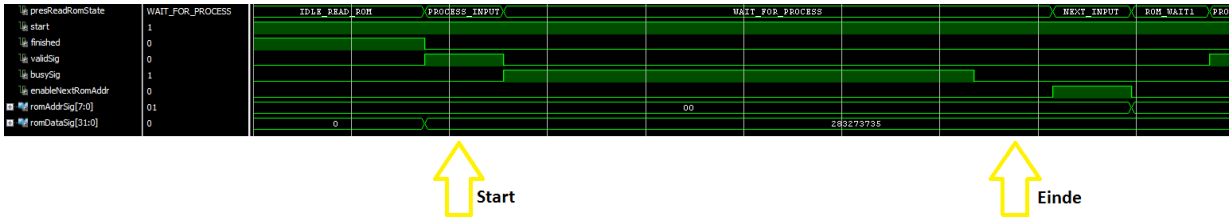
verwerkt is. Daarvoor dient het 'busy'-signaal. Het 'Add data to heatmap' proces moet het 'busy'-signaal op '1' zetten om te laten weten dat het bezig is met zijn berekening. Wanneer de berekening afgerond is, busy op '0', kan er overgegaan worden naar de volgende observatie. In de 'next input'-state wordt 'EnableNextRomAddr' op '1' gezet. Dit signaal zorgt ervoor dat er naar het volgende adres van de ROM gegaan wordt. Dit zit in een apart proces en er wordt op teruggekomen in deel 4.2.4. Als laatste is er de 'wait_for_ROM'-state die wacht tot de nieuwe data, gelezen uit het ROM, stabiel is. In de IP-core van Vivado staat dat het twee klokcycli duurt vooraleer de data stabiel is. Daarom wordt er gebruik gemaakt van twee 'wait'-states, maar dit werd voor de eenvoud in de FSM weergegeven als één state.

Tabel 4.1: De uitgangen van de dataverwerking naargelang de state.

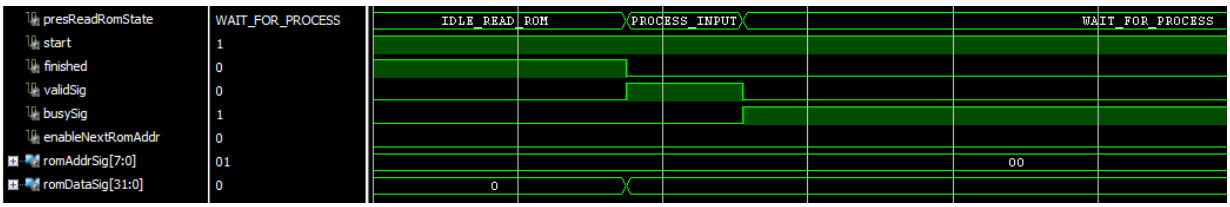
state	**valid**	**finished**	**enableNextRomAddr**
Idle	0	1	0
Process input	1	0	0
Wait for process	0	0	0
Next input	0	0	1
Wait for rom	0	0	0

Net zoals in de vorige delen wordt hier eerst een simulatie gedaan vooraleer de code in de praktijk uitgevoerd wordt. In deze simulatie zijn er wel een paar zaken die anders zijn dan voorheen vermeld werd (vergelijking 4.4)in verband met de definities van de dataset.

Er wordt gewerkt met een ROM van 249 geheugenlocaties, maar slechts het eerste adres heeft een effectieve waarde. Daarnaast waren de geheugens anders beschreven. Er werd eerst met grotere geheugens gewerkt om een betere resolutie te behalen. Dit bleek echter overbodig vanwege de beperkte resolutie van het VGA-scherm. Ook was er met de oude geheugens te weinig BRAM aanwezig om een parallellisatie uit te voeren. Die parallellisatie wordt in deel 4.4 in detail uitgelegd. In de simulatie is er een ROM met een diepte van 249 en een breedte van 32 bits waarvan 19 bits voor de coördinaten, zoals in 4.2.2 beschreven staat. De overige 13 bits worden gebruikt voor de waarde van de observatie.



(a) Het volledige proces



(b) Start van het proces (Start van de berekening)

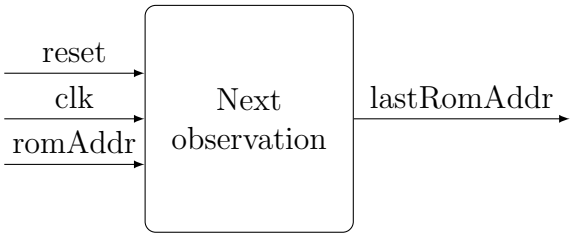
(c) Einde van het proces (Overgang naar de volgende pixel)

Figuur 4.10: Behavioral simulatie van het dataverwerkingsproces.

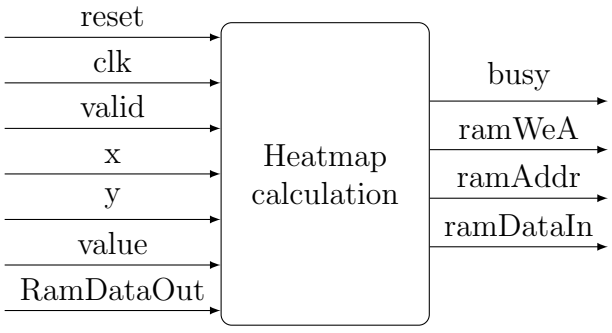
De simulatie is in figuur 4.10 te zien. De uitgangen kunnen bij iedere state nagegaan worden en de sturing van de state in functie van de ingangen kan eveneens nagegaan worden. Er is te zien dat de gewenste functionaliteit behaald wordt. De opeenvolgende toestanden worden correct doorgelopen.

4.2.4 Volgende observatie

Dit proces is in principe een onderdeel van 4.2.3, maar het proces dat de volgende state definieert is niet geklokt. Het enige dat geklokt is, is de overgang naar de volgende state. Dit komt omdat er gebruik gemaakt wordt van een two-process FSM die de overgang naar de volgende state en de bepaling van de volgende state elk in een apart proces uitvoert. In deze thesis zullen die two-process FSM altijd gevisualiseerd worden met slechts één proces. Wanneer de FSM in 'next input' zit, wordt er overgegaan naar het volgende ROM-adres. Als het laatste adres bereikt is en de verwerking ervan is afgerond, moet de FSM terug naar de 'Idle' state. Wanneer die adrestoewijzing en controle puur combinatorisch gebeurt, dan heeft Vivado problemen om dit correct te synthetiseren. Er is nood aan DFF's waardoor de adrestoewijzing onder de vleugels van de klok moet gebeuren. Daarvoor wordt er een apart proces aangemaakt dat dit probleem oplost. Figuur 4.11 toont hoe het volgende adres van het ROM bepaald zal worden onder invloed van de reset, klok en het 'nextPixelEnable'-signaal.



Figuur 4.11: Functionele blok van het 'volgende observatie'-proces.



Figuur 4.12: Functionele blok van de heatmap-berekening.

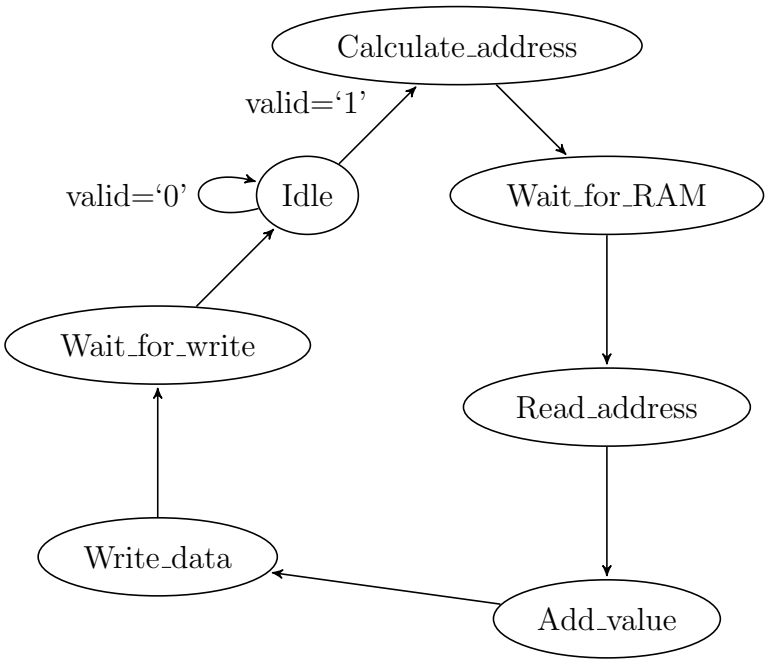
4.2.5 Heatmap-berekening

De berekening van de heatmap is uiteraard het belangrijkste deel van het systeem. Het gaat de verkregen waarde van een observatie op de juiste locatie naargelang de verkregen coördinaten gaan optellen bij de huidige heatmap. Om dit te verwezenlijken werd er opnieuw gebruik gemaakt van een FSM. Dit werd gedaan omdat het bij geheugens altijd een aantal klokcycli duurt tegen dat de data stabiel is en het daarom altijd een voordeel is om de verschillende stappen sequentieel uit te voeren. Zoals in 4.2.3 start de FSM in de 'idle'-state. Het proces is klaar met zijn berekeningen en wacht op een nieuwe geldige observatie van het dataverwerkingsproces. Bij een geldige observatie wordt overgegaan naar de 'Calculate address'-state. Hier wordt aan de hand van de x- en y-coördinaten het specifieke adres van het RAM bepaald (vergelijking 4.5). Vervolgens wordt één klokcyclus gewacht op stabiele data van het geheugen. Daarna wordt de output van het geheugen gelezen in de 'Read address'-state. Deze wordt opgeslagen in signaal 'ramDataInA'. In de 'Add value'-state wordt de gegeven waarde toegevoegd aan de reeds bestaande waarde. Daarna worden in 'Write data' alle parameters gezet zodat de berekende waarde op de correcte locatie in het geheugen geschreven staat. Als laatste wordt er nog een klokcyclus gewacht om te zorgen dat de data zeker goed geschreven werd in het geheugen om daarna terug te keren naar de 'idle'-state. Enkel in die state zal het 'busy'-signaal logisch '0' zijn. Daardoor weet het dataverwerkingsproces dat het component niet aan het rekenen is. Tijdens alle andere states is er een deel van de berekening bezig waardoor het 'busy'-signaal op '1' geplaatst wordt. Het dataverwerkingsproces zal nu wachten.

$$adres = y * breedte + x \tag{4.5}$$

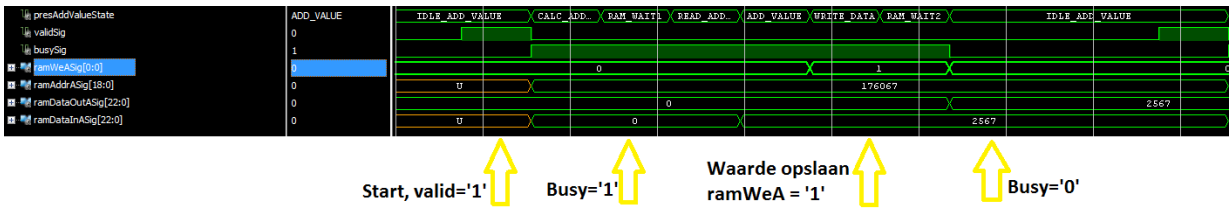
Tabel 4.2: De uitgangen van de heatmap-berekening naargelang de state.

state	**busy**	**ramWriteEnable**	**ramAddrA**	**ramDataInA**
Idle	0	0	0	0
Calculate address	1	0	640*y + x	0
RAM wait	1	0	ramAddrA	0
Read address	1	0	ramAddrA	ramDataOutA
Add value	1	0	ramAddrA	+value
Write data	1	1	ramAddrA	ramDataInA
Wait for write	1	1	ramAddrA	ramDataInA



Figuur 4.13: FSM van de heatmap-berekening

Op dit proces wordt een simulatie uitgevoerd om de implementatie zonder fouten in de praktijk uit te voeren. Ook hier zijn er aanpassingen aan het gebruikte geheugen ten opzichte van de gedefinieerde geheugens in deel 4.2.2. Hier werd een RAM gebruik met diepte 307200 en breedte 23 bit. Die 23 bit werd in het begin van de implementatie zo gekozen omdat er toen gewerkt werd met een dataset van 1000 punten die een 13-bit waarde hadden. Wanneer deze allemaal op hetzelfde punt zouden liggen, zou de maximum waarde van de heatmap ongeveer 23 bit groot moeten zijn. In het begin werd er gefocust op goede functionaliteit waardoor dit in eerste instantie niet gewijzigd werd.



Figuur 4.14: Simulate van de berekening.

In figuur 4.14 is te zien dat de initiële waarde op het adres 176067 op 0 staat. Dit komt omdat bij opstart van het systeem, de volledige heatmap de waarde 0 krijgt. Vervolgens

zal er een waarde 2567 aan toegevoegd worden. Twee klokcycli later is te zien dat die waarde effectief geschreven werd omdat de uitgang van het RAM op dat adres gelijk is aan 2567. Het adres en de waarde werden bepaald uit de uitgang van het ROM die te zien is in figuur 4.10. In tabel 4.3 is te zien hoe de 32-bit data van het ROM omgevormd werd tot een RAM-adres en een waarde die op dit adres toegevoegd moet worden.

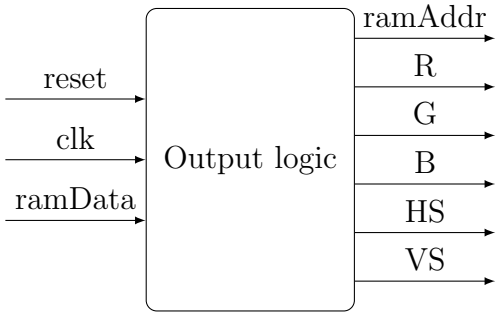
Tabel 4.3: Van ROM-data naar RAM-adres en waarde

283273735		
00010000111000100110101000000111		
10 bits	9 bits	13 bits
0001000011	100010011	0101000000111
67	257	2567
176067		2567

4.2.6 Output logica

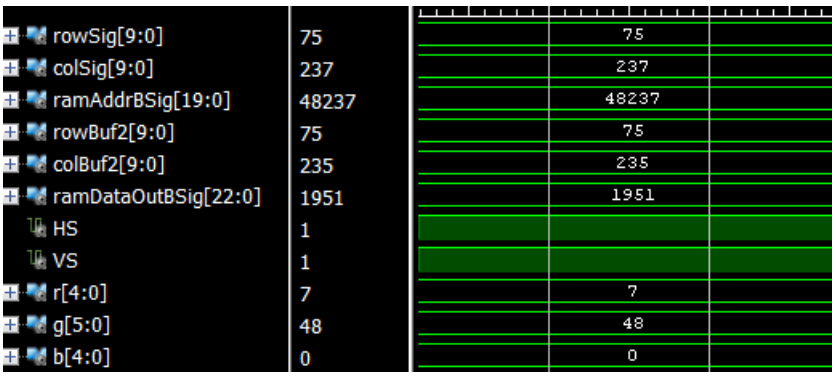
Aan de ene zijde van het systeem wordt het geheugen gewijzigd om de dichtheid te bepalen. Aan de andere zijde wordt de dichtheid gelezen om ze visueel weer te geven. Voor de omzetting van de data in het geheugen naar de sturing van de VGA is er enige logica nodig. In deel 2.2.3 werd al vermeld dat het VGA-scherm initieel 480x640 pixels telt, maar dat er door de HS en VS een 525x800 scherm bekomen wordt. In dit proces zal geteld worden van 0 tot 799 om vervolgens terug te keren naar 0 en over te gaan naar de volgende rij. Wanneer de rij 524 is en de kolom 799 is de volgende pixel de startwaarde (0,0). Aan de hand van de coördinaten wordt het adres bepaald om het geheugen te lezen (vergelijking 4.5). Aangezien het twee klokcycli duurt vooraleer de data van het geheugen stabiel is, moet de sturing van het scherm gebeuren in functie van de data en niet van het adres. Die functionaliteit werd verkregen door de coördinaat, van waaruit het adres berekend wordt, twee klokcycli te bufferen. In functie van dat gebufferde coördinaat wordt de HS en VS gestuurd en worden de kleurwaarden van het scherm bepaald. Voor de eenvoud wordt alle output logica weergegeven met slechts één proces. In de praktijk is er telkens een apart proces voor de HS, VS, buffering en bepaling van de kleurwaarde.

Alvorens de implementatie praktisch te gaan testen wordt deze eerst gesimuleerd. Tijdens de simulatie werden drie zaken bestudeerd, namelijk de kleurwaarden en de synchronisatiesignalen die naar het scherm gestuurd worden. Bij de kleurwaarden wordt er vooral



Figuur 4.15: Functionele blok van de output logica.

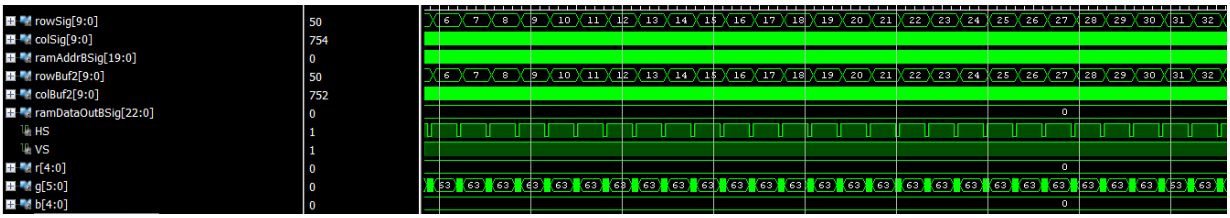
gelet op de omzetting van de data uit het geheugen naar een kleurwaarde en het zorgen dat dit overeenstemt met de positie op het scherm. Eerder werd al beschreven dat er een buffer van twee klokcycli werd toegevoegd aan het systeem om deze aflijning tussen kleurwaarde en positie te verkrijgen. De data in het geheugen is hier opnieuw 23 bit breed. Om de data voor te stellen werd gebruik gemaakt van bit 12 tot en met 7. Op een gegeven moment is de waarde uit het geheugen 1951. Binair kan dit geschreven worden als "000000000000011110011111". De gebruikte bits leveren "001111" of een waarde 15. Zoals gedefinieerd zal de groenwaarde gelijk zijn aan $63 - 15 = 48$. De rode kleur wordt slechts bepaald door 5 bits, namelijk bit 12 tot en met 8. De waarde is "00111" of 7. In figuur 4.16 is te zien dat voor die waarde van 1951 in het geheugen de RGB-waarde correct gedefinieerd wordt.



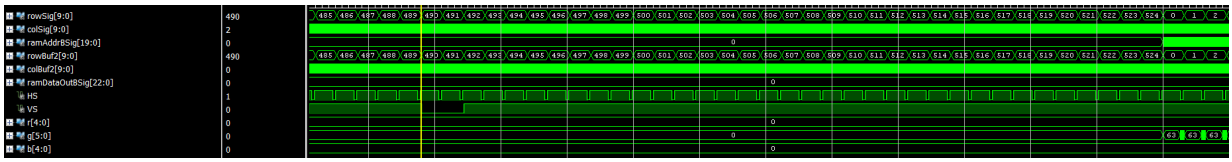
Figuur 4.16: Behavioral simulatie van de sturing van de RGB-waarden in functie van de positie op het scherm.

De sturing van de synchronisatiesignalen hangt enkel af van de gebufferde positie op het scherm. In deel 2.2.3 werd het principe van die signalen al besproken. Hier wordt gekeken in welke mate de simulatie het gewenste gedrag vertoont. Figuur 4.17 toont de gewenste

HS-functionaliteit. Op de figuur is te zien dat wanneer HS logisch '0' wordt, de outputs van de RGB-waarden op 0 gezet worden en dat de positie overgaat naar het begin van de volgende lijn. Figuur 4.18 toont de simulatie wanneeer het einde van het scherm bereikt wordt. Daar is te zien dat het VS-signaal logisch op '0' gezet wordt, wat zorgt voor een continu zwart beeld in die zone en een overgang naar de beginpositie van het scherm. In figuur 4.17 is te zien dat de groene kleur op 63 staat in het actief deel van het scherm en wanneer HS logisch '0' is, is het niet zichtbaar. Dit komt omdat de kleur wisselt naar de waarde 0, maar het is niet te zien. Merk hierbij wel op dat die waarde niet continu 63 zal zijn, maar dit hier nu wel het geval is. Dit komt omdat er slechts een klein deel van de gehele simulatie getoond werd omdat de complexiteit anders te groot zou zijn voor de simulatie.



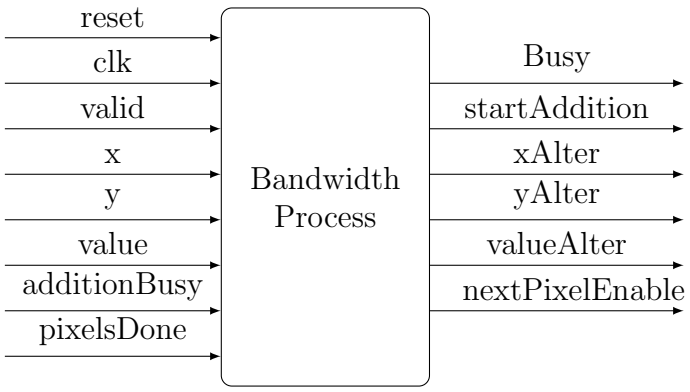
Figuur 4.17: Behavioral simulatie van de sturing van HS in functie van de positie op het scherm.



Figuur 4.18: Behavioral simulatie van de sturing van VS in functie van de positie op het scherm.

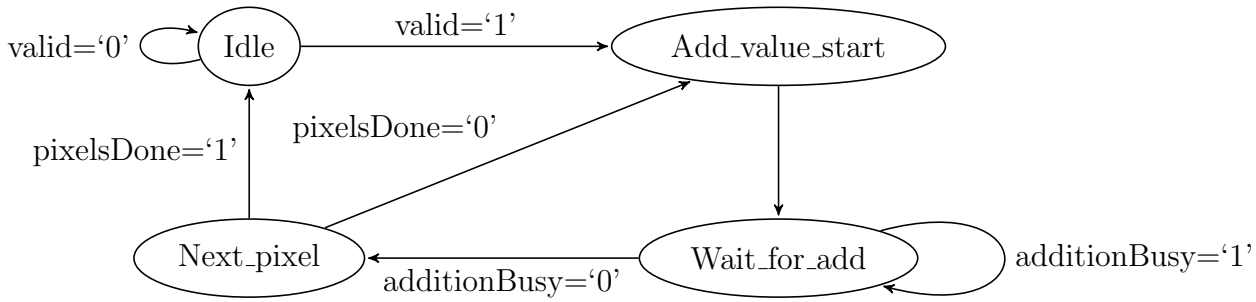
4.3 Toevoegen van de bandbreedte

In de voorgaande realisaties werd enkel gewerkt met PDE bij een bandbreedte van 0. Dit wil zeggen dat een observatie geen invloed heeft op de omliggende punten. Voor het invoeren van de KDE is er nood aan een bandbreedte omdat de gebruikte kernel definieert hoe de observatie over de naburige pixels uitgesmeerd wordt wat een invloed heeft op de 'smoothing' van de heatmap. In deze thesis werd de PDE uitgebreid door er een bandbreedte aan toe te voegen. Dit is eigenlijk KDE door gebruik te maken van



Figuur 4.19: Functionele blok van het bandbreedte-proces.

de uniforme kernel. Om dan de kernel te wijzigen, moet slechts één lijn code gewijzigd worden. Die oplossing is een hard-coded oplossing. In principe is een soft-coded oplossing nodig zodat gebruikers zelf de kernel kunnen kiezen alvorens de berekening te starten. Die soft-coded oplossing werd in deze thesis niet uitgevoerd.



Figuur 4.20: FSM van het bandbreedte-proces.

Om de bandbreedte-functionaliteit te implementeren werd er opnieuw gebruik gemaakt van een FSM. In dit proces was dit een goeie oplossing omdat er concreet een tussenstuk tussen de FSM van het dataverwerkingsproces, zie figuur 4.9, en de FSM van de heatmap-berekening, zie figuur 4.13, moet worden geplaatst. Aangezien het dataverwerkingsproces slechts één observatie doorgeeft, maar er meerdere waarden moeten toegevoegd worden aan het geheugen, zal de geïmplementeerde FSM voor de bandbreedte starten op aangeven van het dataverwerkingsproces. Vervolgens zal hij bijhouden aan welk punt binnen de bandbreedte er gerekend wordt om zo de nodige waarden door te geven aan de heatmap-berekening. Wanneer alle punten berekend zijn, wordt er terug gecommuniceerd naar het

dataverwerkingsproces zodat overgegaan kan worden naar de volgende observatie.

Tabel 4.4: De uitgangen van de heatmap-berekening naargelang de state.

State	**Busy**	**startAddition**	**NextPixelEnable**
Idle	0	0	0
Add value start	1	1	0
Wait for add	1	0	0
Next pixel	1	0	1

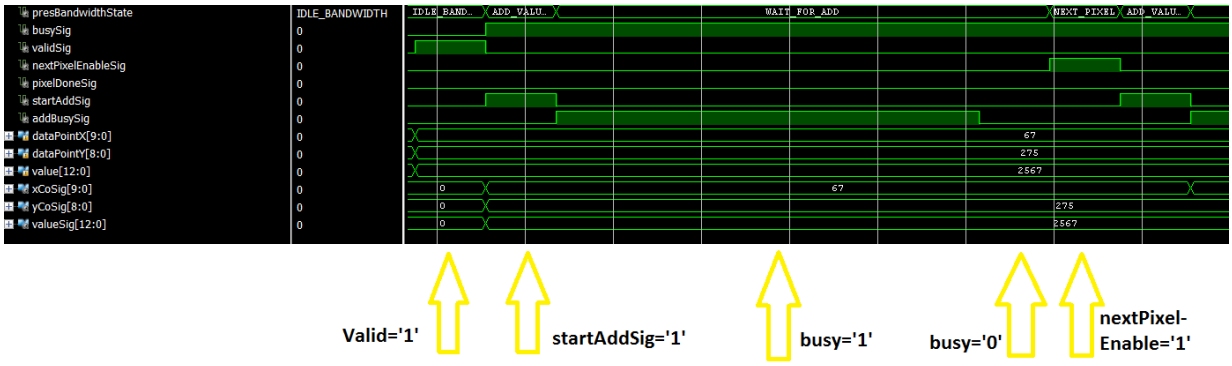
Zoals in iedere geïmplementeerde FSM van deze thesis wordt er gestart in de 'idle'-state. Bij geldige data wordt er overgegaan naar de 'add value start'-state, die de FSM van de heatmap-berekening opstart. Vervolgens wordt er gewacht tot de berekening ten einde is. Daarna wordt overgegaan naar de volgende pixel binnen de bandbreedte. Wanneer alle pixels berekend zijn, wordt terug overgegaan naar de 'idle'-state.

i=-2 **j=-2**	i=-1 j=-2	i=0 j=-2	i=1 j=-2	**i=2** **j=-2**
i=-2 j=-1	i=-1 j=-1	i=0 j=-1	i=1 j=-1	i=2 j=-1
i=-2 j=0	i=-1 j=0	**i=0** **j=0**	i=1 j=0	i=2 j=0
i=-2 j=1	i=-1 j=1	i=0 j=1	i=1 j=1	i=2 j=1
i=-2 **j=2**	i=-1 j=2	i=0 j=2	i=1 j=2	**i=2** **j=2**

Figuur 4.21: Relatief verloop van de pixels bij een bandbreedte van 2[13].

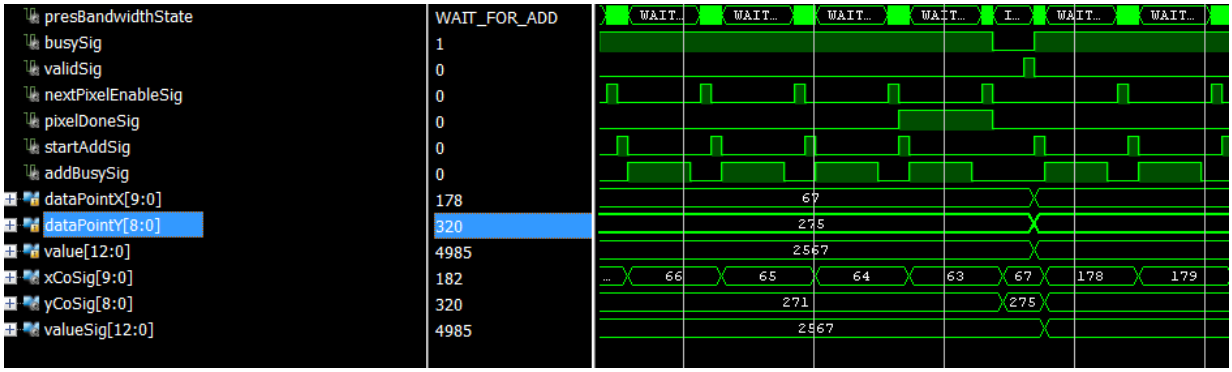
In tabel 4.4 is te zien dat er ook een 'nextPixelEnable'-signaal aangestuurd wordt. In deel 4.2.4 staat al beschreven waarom er soms nood is aan externe processen die geklokt zijn om over te gaan naar een volgend adres of in dit geval pixel. Het daar geschetste probleem is hier ook van toepassing. Daarom is er een extra proces toegevoegd dat de overgang naar de volgende pixel regelt. In figuur 4.21 is een voorbeeld te zien van hoe de pixels relatief verlopen ten opzichte van de centrale pixel bij een bandbreedte van 2. Om dit te implementeren werd eerst gebruik gemaakt van twee variabelen, één voor de horizontale waarde en één voor de verticale waarde. Deze hadden eerst een bereik van $-bandbreedte$ tot $bandbreedte$. Bij het uitvoeren kan er vastgesteld worden dat dit problemen geeft waardoor er een ander aanpak vereist is. Het probleem kan opgelost worden door de variabele een

bereik van 0 tot *bandbreedte* te geven om vervolgens zelf bij te houden ofdat de waarde positief of negatief is. Deze oplossing werd dan ook met succes geïmplementeerd.

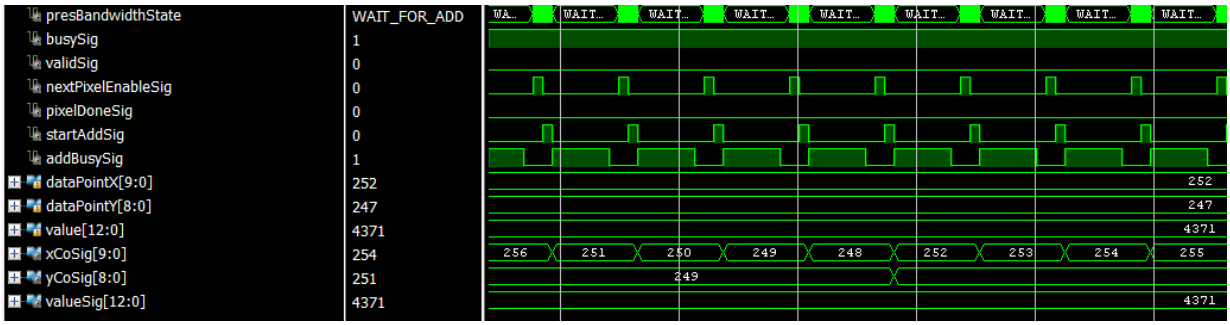


Figuur 4.22: Behavioral simulatie van de FSM van het bandbreedte-proces.

Deze functionaliteit werd eerst gesimuleerd om de correcte werking ervan na te gaan. Bij deze simulaties werden drie zaken getest, namelijk de FSM (figuur 4.22), de overgang naar de volgende observatie (figuur 4.23) en de waarden voor de heatmap (4.24). Bij de simulatie van de FSM is te zien hoe de signalen gestuurd worden bij iedere state en hoe de state wijzigt onder invloed van bepaalde signalen. Bij de overgang naar de volgende observatie is te zien dat ''busy' op '0' geplaatst wordt om de overgang in gang te zetten. Dit gebeurt in de 'idle'-state en daar werd terechtgekomen omdat 'pixelDoneSig' op '1' stond. Als ieder punt binnen de bandbreedte berekend is, zal er naar de 'idle'-state gegaan worden om vervolgens een nieuwe observatie binnen te krijgen.



Figuur 4.23: Behavioral simulatie van de overgang naar de volgende observatie.



Figuur 4.24: Behavioral simulatie van de waarden die naar het berekeningsblok gestuurd worden.

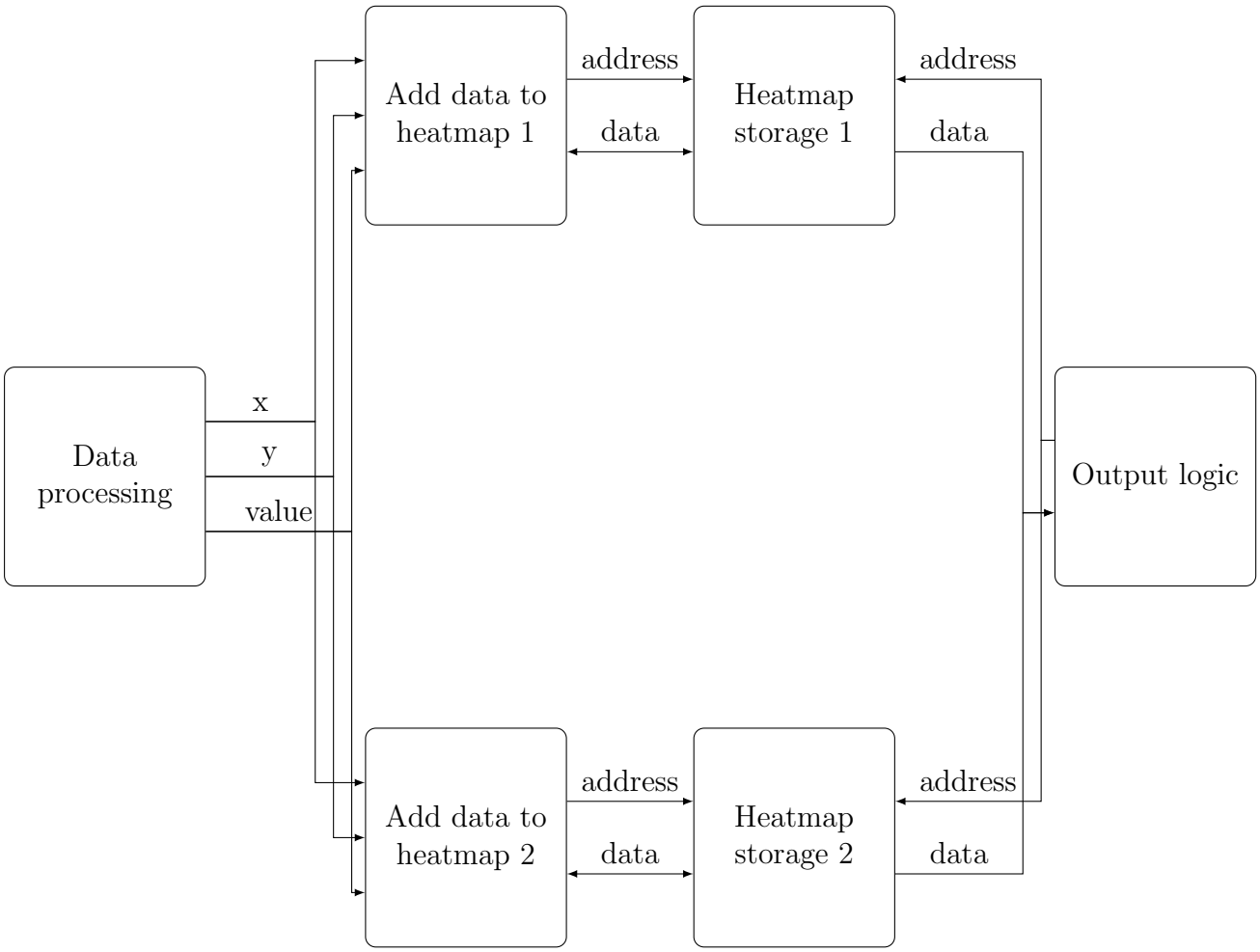
4.4 Parallellisatie

Nu het gehele systeem gebouwd is en wanneer de limieten van de klok bereikt zijn, kan er versnelling bekomen worden door parallellisatie te gebruiken. Hierbij valt wel op te merken dat parallellisatie in de praktijk extra logica vereist wat zorgt dat de versnelling niet altijd evenredig is met het aantal componenten.

4.4.1 Toepassing van de parallellisatie

In deze thesis zal het algoritme toegepast worden op het 'Add data to heatmap'-blok van figuur 4.1. Praktisch zal dit blok bij een parallellisatiegraad van 2 dubbel uitgevoerd worden waardoor er twee observaties tegelijk toegevoegd kunnen worden aan de heatmap. Er is hier echter wel een beperking door het geheugen. Er werd gebruik gemaakt van een true dual-port RAM waarbij aan de ene zijde de dichtheidsbepaling gebeurde en aan de andere zijde het lezen van het geheugen om de data te sturen naar de heatmap. Er zijn geen toegangspunten meer naar het geheugen om de parallellisatie uit te voeren, waardoor de geheugens ook geparallelliseerd worden. Ieder berekeningsblok krijgt een eigen geheugen toegewezen met de grootte van het scherm. Bij het uitlezen van de heatmap zal van ieder geheugen op hetzelfde adres de waarde gelezen en opgeteld worden wat tot de definitieve dichtheid van dat punt leidt.

Om de parallellisatie uit te voeren was er nood aan extra logica. Het dataverwerkingsproces was geïmplementeerd om te werken met slechts één berekeningsblok waarmee de communicatie verliep via het 'valid'- en 'busy'-signaal. De toevoeging van de bandbreedte maakte voor het dataverwerkingsproces niet uit want de communicatie bleef hetzelfde.



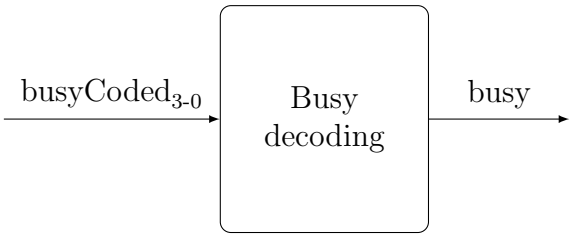
Figuur 4.25: Blokschema van het systeem met een parallellisatiegraad van 2.

Nu moeten er meerdere 'valid'-signalen uitgestuurd worden en krijgt het proces meerdere 'busy'-signalen binnen. Vooral de logica voor de 'valid'-signalen is vermoeilijkt omdat er moet nagegaan worden naar welk blok de observatie gestuurd moet worden. Tijdens het programmeren van code voor hardware is het aangewezen zoveel mogelijk functionaliteiten los te koppelen van elkaar en in aparte processen te steken. Daardoor worden er twee coderingsblokken aangemaakt om het dataverwerkingsproces compatibel te maken met meerdere berekeningsblokken.

4.4.2 Decodering van de 'busy'-signalen

Om de decodering te bespreken wordt uitgegaan van een parallellisatiegraad van vier. Dit wil zeggen dat er vier berekeningsblokken in het systeem zullen zitten die elk één 'busy'-

signaal als uitgang hebben. Het decoderingsblok (4.26) zal die vier signalen, gestuurd als $\text{busyCoded}_{3\text{-}0}$, gaan omzetten naar slechts één signaal dat gestuurd wordt naar het dataverwerkingsproces. Enkel als de vier componenten aan het rekenen zijn, zal het decoderingsblok als uitgang '1' zijn. Enkel dan zal het datavewerkingsproces niet doorgaan naar de volgende observatie, maar wachten tot er een component zijn berekening afrondt.



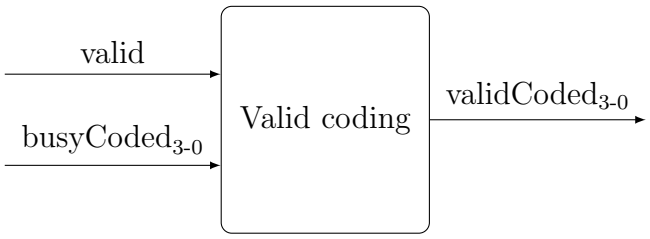
Figuur 4.26: Functionele blok van de 'busy'-decodering.

De beschreven functionaliteit wordt eerst geïmplementeerd en daarna onderworpen aan een simulatie. In die simulatie, zie figuur 4.27, is te zien dat het 'busy'-signaal gestuurd wordt door het $\text{busyComponents}_{3\text{-}0}$-signaal. Dit signaal stelt het 'busyCoded'-signaal van in figuur 4.26 voor. Het 'busy'-signaal staat enkel op '1' als alle componenten aan het rekenen zijn ofwel wanneer 'busyComponents' gelijk is aan "1111".

Figuur 4.27: Behavioral simulatie van de 'busy'-decodering.

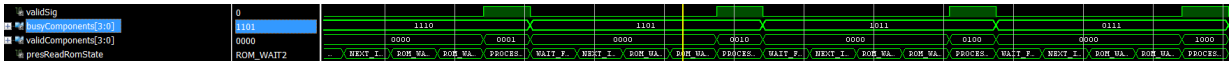
4.4.3 Codering van het 'valid'-signaal

In deel 4.2.3 werd reeds besproken dat het dataverwerkingsproces aangeeft dat er geldige data klaar staat om berekend te worden door gebruik te maken van het 'valid'-signaal. Als er terug met een parallellisatiegraad van 4 gewerkt wordt, dan moeten er vier 'valid'-signalen verstuurd worden. De blokken moeten uiteraard apart gestuurd worden omdat ze anders dezelfde berekening zouden doen. Het coderingsblok dat hiervoor geschreven wordt, heeft als ingangen het 'valid'-signaal van het dataverwerkingsproces en het 'busyCoded'-signaal van de vier componenten. Via 'busyCoded' kan gezien worden welke blokken klaar zijn om een berekening uit te voeren. Wanneer 'valid' daarna logisch '1' is, kan bepaald worden aan welk blok de volgende berekening toegewezen zal worden. Het 'validCoded'-signaal zal zo aangepast worden om de gewenste component aan te sturen.



Figuur 4.28: Functionele blok van de 'valid'-codering.

Ook dit proces werd eerst functioneel gesimuleerd. In deze simulatie, zie figuur 4.29, is te zien dat het 'validComponents'-signaal aangestuurd wordt in functie van het 'valid'-signaal en het 'busyComponents'-signaal. Het proces bekijkt de beschikbare instanties en zal dan het 'validComponents'-signaal gaan wijzigen om de gewenste instantie aan te sturen. Hierbij kan opgemerkt worden dat het uitgangssignaal, bestaande uit vier bits, nooit tweemaal een '1' zal bevatten. Dit proces heeft vijf mogelijke uitgangen, namelijk "0000", "0001", "0010", "0100" en "1000".



Figuur 4.29: Behavioral simulatie van de 'valid'-codering.

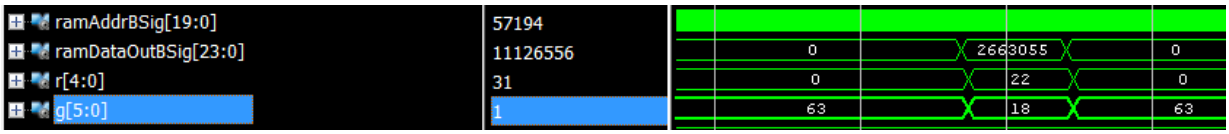
4.4.4 Extra opslag van de observaties

In het systeem zonder parallellisatie wordt de observatie opgeslagen door het dataverwerkingsproces en wordt deze doorgegeven aan het berekeningsblok. De observatie wordt om geheugen te sparen niet lokaal opgeslagen aangezien dit niet vereist is. Door de invoer van de parallellisatie is dit wel nodig omdat de observatie in het dataverwerkingsproces wijzigt terwijl het berekeningsblok nog bezig is aan een oudere observatie. Daarom worden bij de start van de berekening de verschillende waarden opgeslagen waardoor ze tijdens de berekening niet meer wijzigen.

4.4.5 Extra output logica

Aangezien ieder berekeningsblok een eigen geheugen heeft, is er noodzaak aan een blok die de data komende van de aparte geheugens samen neemt en de definitieve dichtheid voor dat punt bepaalt. Dit wordt verwezenlijkt door gebruik te maken van een som. Ieder blok

heeft een deel van de observaties berekend en in zijn toegewezen geheugen opgeslaan. Er kan gesteld worden dat alle blokken alle observaties berekend hebben en dat de som van de geheugens op een specifiek geheugenadres de totale dichtheid voor dit geheugenadres moet zijn. De functionaliteit wordt verwezenlijkt door naar de verschillende geheugens hetzelfde adres te sturen en de verschillende data bij elkaar op te tellen.



Figuur 4.30: Behavioral simulatie van de optelling

De extra logica, met name de sommatie, werd gesimuleerd. In figuur 4.30 is te zien hoe die simulatie verloopt. Aan de hand van de waarden in die simulatie werd berekend in welke mate de kleurwaarden die gestuurd worden overeenstemmen met de data afkomstig van het RAM (tabel 4.5).

Tabel 4.5: Van RAM-data naar RGB bij parallellisatie

2663055			
001010001010001010001111			
6 bits	6 bits	6 bits	6 bits
001010	001010	001010	001111
10	10	10	15
45			
RGB = (22, 18, 0)			

Hoofdstuk 5

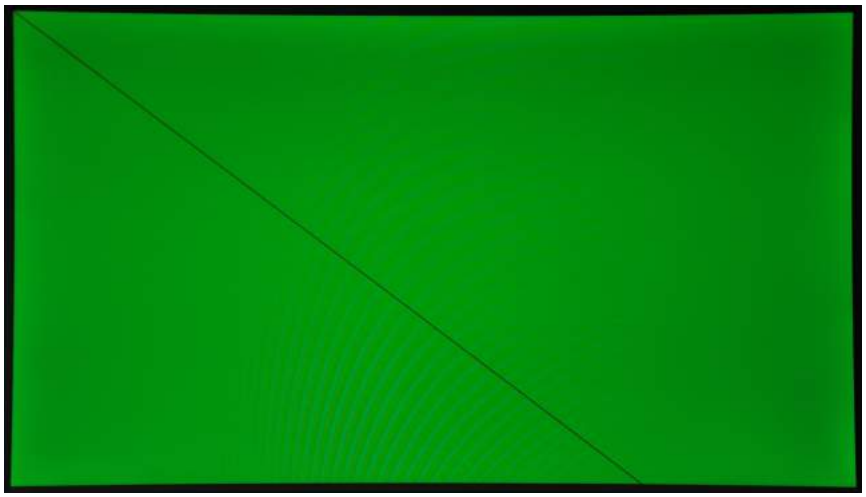
Resultaten

In deel 4 werd beschreven hoe het praktisch deel van deze thesis aangepakt werd. Deze implementaties werden onderworpen aan praktische tests om na te gaan in welke mate de uitwerkingen correct functioneren. Daarnaast werd er ook gekeken naar de berekeningstijd van de implementatie bij verschillende parameters om als laatste eens te kijken naar het geheugengebruik, met name BRAM, van de implementatie.

5.1 VGA

De eerste implementatie was de sturing van een VGA-scherm. In deel 2.2.3 staat beschreven hoe deze implementatie moet gebeuren. De praktische uitwerking werd beschreven in deel 4.2.6. De functionaliteit die vereist is voor de sturing is al zeer complex in de output logica. Daarom wordt de sturing eerst vereenvoudigd uitgevoerd om vertrouwd te geraken met de synchronisatie-signalen. De ontwikkelde toepassing zal een groen scherm weergeven met rode pixels waar de kolom en de rij gelijk zijn aan elkaar. Deze toepassing werd gekozen omdat die een sturing per pixel vereist. Sturing van het scherm in blokken is iets eenvoudiger. Daarnaast is het ook een test om te zien of de rode kleur duidelijk zal zijn in combinatie met een groene achtergrond.

In figuur 5.1 is te zien dat het groene scherm een lijn bevat die overeenstemt met de lijn waarvan de kolom gelijk is aan de rij bij de pixels. Deze lijn zou echter rood moeten zijn, maar door het verlies aan kwaliteit bij het fotograferen, is dit niet te zien. Er kan vastgesteld worden dat de telling van de rijen en kolommen goed gebeurt en de sturing van



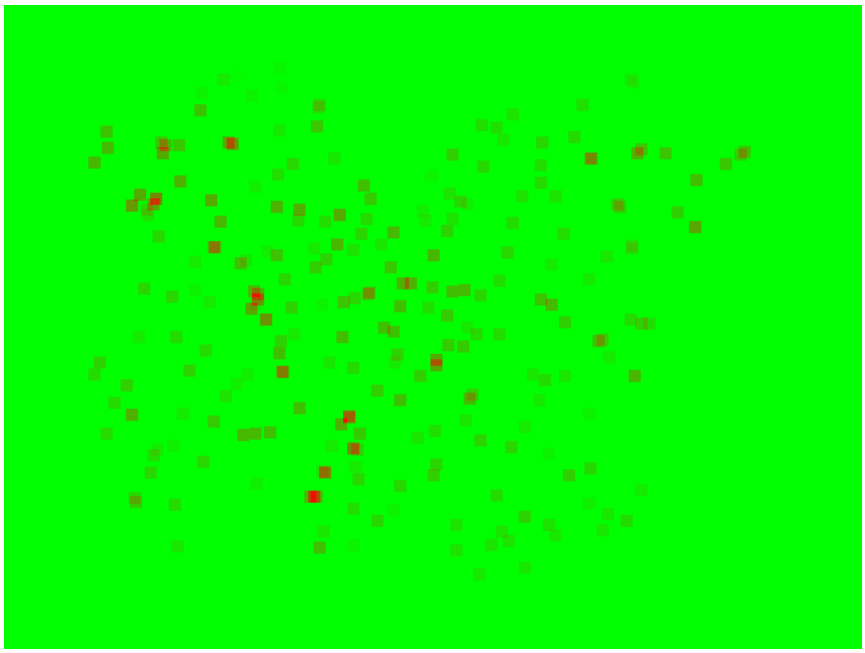
Figuur 5.1: Eenvoudige sturing van het VGA-scherm.

de synchronisatie-signalen goed overeenstemt met de positie op het scherm. Op de figuur zijn er ook nog blauwe strepen te zien. Deze zijn ontstaan door het fotograferen van het VGA-scherm waarbij er af en toe ruis op het scherm komt. Bij sommige figuren zal dit het geval zijn, maar dit heeft niets te maken met de sturing.

5.2 Functionele vergelijking

Om de correcte functionaliteit van de 'Density estimator' te bepalen, wordt de toepassing ook in software geschreven. In deel 3 staat beschreven hoe het algoritme met verschillende kernels geïmplementeerd werd in C om een vergelijking te kunnen maken van de verschillende kernels. Daarvoor is een willekeurig gegenereerde dataset gebruikt waarop het algoritme uitgevoerd werd. De implementatie in hardware maakt gebruik van een dataset omtrent de fietsknooppunten in de Vlaamse Ardennen[9]. In deel 4.2.2 staat beschreven hoe deze dataset gebruikt werd tijdens de implementatie. De 23-bit voorstelling van een observatie werd pas bereikt nadat er enige preprocessing gedaan werd. Die preprocessing werd uitgevoerd in Python. Deze taal werd gekozen vanwege de eenvoudige implementatie, vooral bij het gebruik van arrays. Omdat de focus op het genereren van de dataset lag, was een snelle implementatietijd meer gewenst dan een snelle uitvoertijd. Daarom kreeg Python de voorkeur op C.

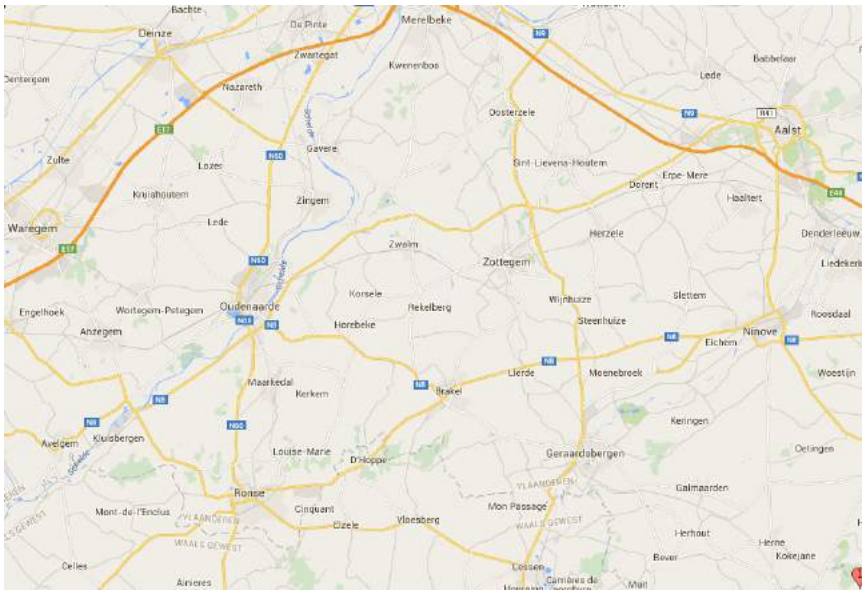
De dataset bestaat uit x- en y-coördinaten en een waarde. Dit is identiek als bij de hardware-implementatie. De coördinaten van de originele dataset waren echter de lengte-



Figuur 5.2: Ideale heatmap van de fietsknooppunten in de Vlaamse Ardennen met een bandbreedte van 4.

en breedtegraden van de verschillende knooppunten. Die lagen in een zeer beperkte range en moesten omgevormd worden naar de juiste pixels op het VGA-scherm. De waarden van de observaties liggen in een bereik van 0 tot 10.000 en moeten omgevormd worden naar een bereik van 0 tot 31. Dit bereik werd gevonden door het maximale bereik te bepalen waarij de maximum waarde van de heatmap kleiner is dan 64. De 'Density estimator' werd dus ook geïmplementeerd in Python. De ideale heatmap wordt dus berekend en is te zien in figuur 5.2.

De verkregen heatmap is redelijk abstract, net zoals de eerder opgestelde heatmaps in deze thesis. Om meer betekenis te kunnen geven aan de data is er nood aan een onderliggende kaart, zie figuur 5.3. Aangezien de coördinaten van de observaties omgevormd werden tot pixellocaties op het scherm, is het bereik van de heatmap op het scherm gekend. Daarom is het mogelijk om het gebied dat overeenkomt met de heatmap te gaan bepalen. Wanneer figuur 5.2 en figuur 5.3 samengebracht worden tot één figuur (figuur 5.4), kan er meer betekenis gegeven worden aan de data. Een eerste zicht op de figuur kan al volstaan om te zien dat de drukste punten rond Oudenaarde liggen. Daarnaast hebben Louise-Marie en Kruishoutem ook een relatief hoge dichtheid. De analyse van de gehele heatmap is echter geen stof voor deze thesis, dus zal er niet verder op ingegaan worden.

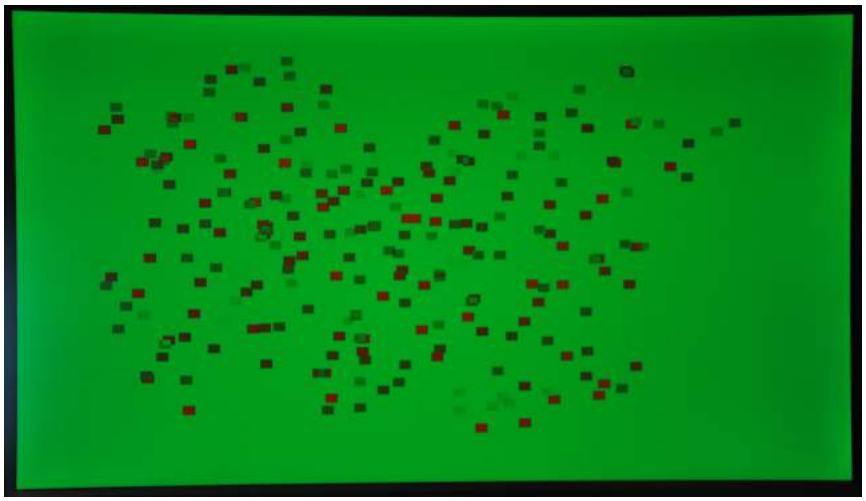


Figuur 5.3: Het gebied dat voorgesteld wordt door de berekende heatmap.

Figuur 5.4: Ideale heatmap met onderliggende kaart bij een bandbreedte van 4.

De bedoeling van deze thesis was om de heatmap in figuur 5.2 te verkrijgen, maar met een implementatie in hardware. In deel 4.2 staat hoe het systeem uitgewerkt werd om de gewenste functionaliteit te bekomen. Bij het bestuderen van het resultaat, zie figuur 5.5, kan er vastgesteld worden dat de positie van de punten overeenkomt met deze in 5.2. Bij

het bekijken van de kleurschaling is er enig verschil op te merken. De rode kleur in figuur 5.5 is donkerder en soms zijn er bij overlappingen van bandbreedtes groene punten. Die gebieden zouden volgens de definitie echter roder moeten zijn dan de individuele observaties vanwege de som van de dichtheden van de overlappende observaties. Die rodere kleur en de groene overlappingen zijn te wijten aan een probleem met het 'start'-signaal. Om de heatmapberekening te starten moet de gebruiker op een knop op de FPGA drukken. De berekening gaat echter zo snel dat de gebruiker de knop nog ingedrukt houdt, wanneer de berekening afgerond wordt. Daarnaast zit er ook dender op de drukknop wat ervoor zorgt dat de knop meer dan één keer ingedrukt wordt. Daardoor wordt de berekening meerdere keren gedaan waardoor observaties roder zijn en de overlapping soms een overflow veroorzaken wat zorgt voor de groene kleur. Er werden verschillende pogingen ondernomen om deze problemen te vermijden. Als eerste werd er gezorgd voor code die slechts één keer een puls gaf als 'start'-signaal, ook al bleef de knop continu ingedrukt. Deze code werkte voor de simpele implementaties, maar bij de parallellisatie leidde het tot onverklaarbare problemen. Daarnaast werd er ook nog geprobeerd om de drukknop te bemonsteren aan een frequentie van 50 Hz. Dit zou zorgen dat zaken zoals dender weggefilterd zouden worden aangezien dit hoogfrequente wijzigingen zijn. Dit was bij de simpele implementaties ook een oplossing, maar had ook problemen bij de parallellisatie.



Figuur 5.5: Heatmap verkregen via de hardware met een bandbreedte van 4 aan 150 MHz.

In de resultaten van de hardware zal dit probleem altijd te zien zijn. Daarom wordt de correcte werking getest op basis van de posities van de punten en de relatieve intensiteit. Het blijft voor de grote meerderheid van de punten mogelijk om te zien dewelke een hogere intensiteit hebben dan andere.

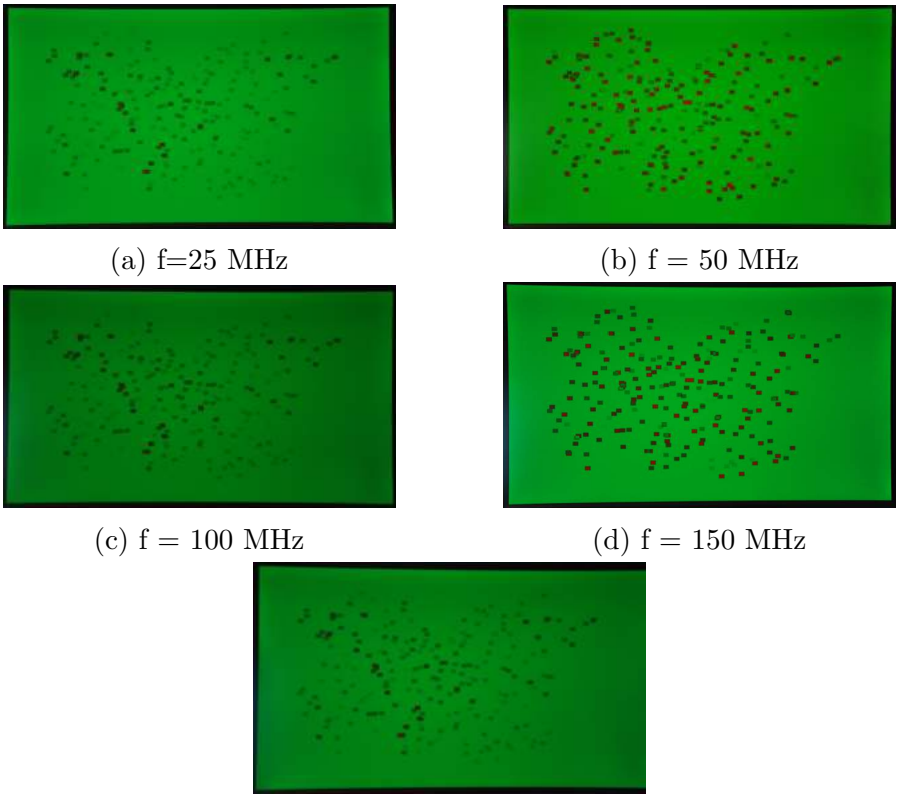
Figuur 5.6: In hardware berekende heatmap met onderliggende kaart.

Om een concrete vergelijking te kunnen maken tussen de in software en in hardware gegenereerde heatmap, werd ook bij de hardware gezorgd voor een onderliggende kaart. Dit werd achteraf handmatig toegevoegd bij de behaalde resultaten. De hardware leent zich niet voor dit soort functionaliteit omdat de gehele afbeelding opgeslagen zou moeten worden, wat teveel geheugen vraagt. Bij studie van de heatmap in figuur 5.6 kan gezien worden dat de pieken rond Oudenaarde, Kruishoutem en Louise-Marie ongeveer overeenstemmen met de pieken uit figuur5.4. Er is wel een lichte verschuiving te zien, maar dit komt waarschijnlijk door de handmatige invoer van de onderliggende kaart. Het VGA-scherm werd altijd gefotografeerd waardoor het scherm nooit ideaal op de foto stond. Daarom werden er wat bewerkingen gedaan om de hoekpunten van het scherm zo goed mogelijk overeen te stemmen met de hoeken van de onderliggende kaart. Daarom kunnen sommige pixels wat verschoven zijn ten opzichte van de ideale heatmap.

5.3 Werkfrequentie

Vooraleer de parallellisatie te implementeren is het gewenst het ontwerp te testen tot aan zijn limieten. De toewijzing van signalen gaat gepaard met enige vertraging. Wanneer die vertraging niet in rekening gebracht wordt, kan het zijn dat de functionaliteit van de toepassing niet gewaarborgd kan worden. Om de limieten van het systeem te testen werd de klokfrequentie geleidelijk aan opgedreven. Er werd gestart met een, zoals in deel 4.2.1 beschreven, frequentie van 25 MHz. Dit was de werkfrequentie van het VGA-scherm en

daarom wordt de implementatie daarop getest. Vervolgens wordt het systeem opgedreven naar veelvouden van die 25 MHZ, namelijk 50, 100, 150 en 200 MHz. Die laatste is de maximale werkfrequentie op de FPGA. Op de resultaten, zie figuur 5.7, is te zien dat de klokfrequentie geen invloed heeft op de functionaliteit van de implementatie. Daarnaast zorgt het evenredig voor de nodige versnelling in het systeem. Als standaard voor de verdere implementaties wordt er met een klok van 150 MHz gewerkt. Dit komt omdat er bij eerdere implementaties aan 200 MHz problemen waren. De maximale werkfrequentie van de FPGA zit net op de valreep van correct werken. Voor de zekerheid wordt daarom de frequentie van 150 MHz gebruikt voor de verdere uiwerking van de 'Density Estimator'.



(a) f=25 MHz (b) f = 50 MHz

(c) f = 100 MHz (d) f = 150 MHz

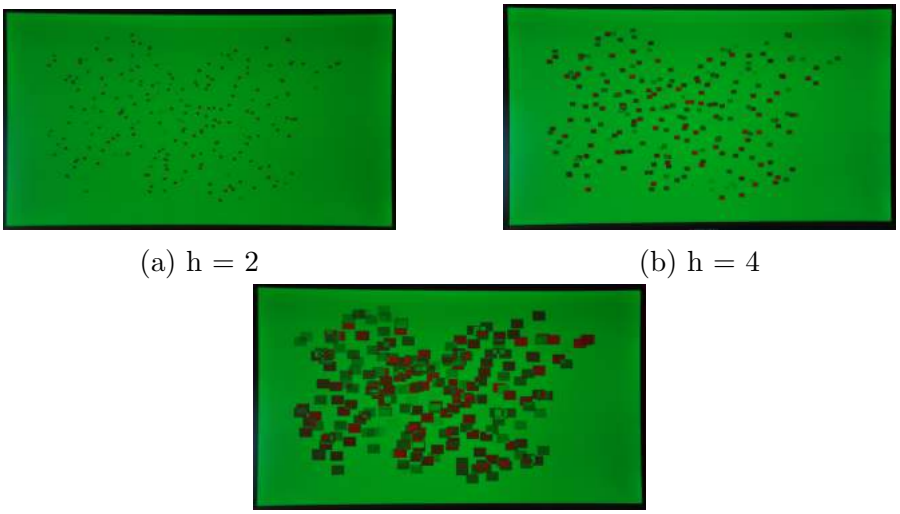
(e) f = 200 MHz

Figuur 5.7: Heatmap bij verschillende klokfrequenties.

5.4 Bandbreedte

Om de KDE te kunnen implementeren, is er nood aan bandbreedte aangezien de KDE anders gelijk zou zijn aan de PDE bij een bandbreedte van 0. In deel 4.3 staat beschreven

hoe die implementatie uitgevoerd werd. Deze implementatie werd getest met verschillende banbreedtes en levert de resultaten zoals te zien in figuur 5.8. De implementatie werd uitgevoerd op de PDE, maar kan door middel van één lijn code omgevormd worden tot een KDE. Die lijn code bevat dan één van de functies uit tabel 2.1, met uitzondering van de gaussian kernel.



(a) h = 2 (b) h = 4

(c) h = 10

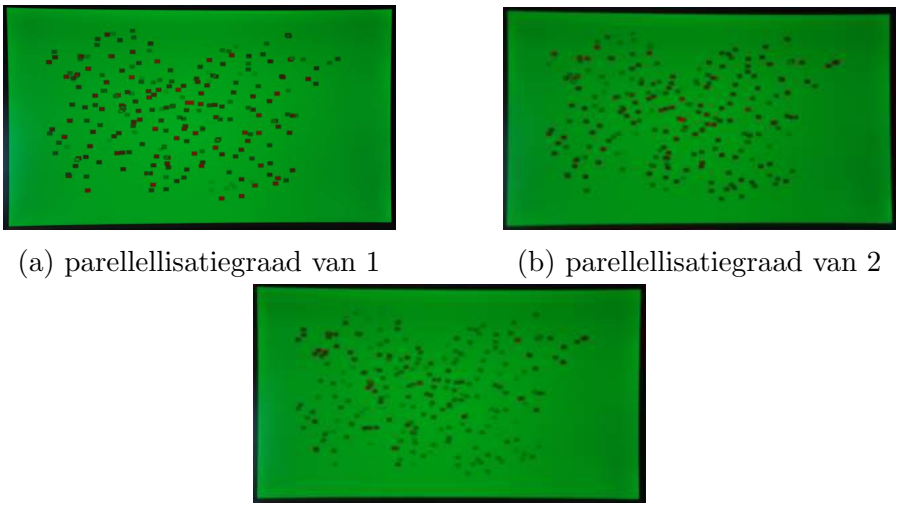
Figuur 5.8: Heatmap bij verschillende bandbreedtes.

In deel 2.1.2 werden de methodes beschreven die de ideale bandbreedte van een dataset bepalen. Een voorwaarde daarvoor is dat de dataset gekend is alvorens de heatmap te berekenen. Bij het opstellen van dynamische heatmaps is dit echter niet het geval, waardoor er een schatting gedaan moet worden van de bandbreedte. Merk hierbij wel op dat er in de hardware gewerkt wordt met een bandbreedte van pixels.

5.5 Parallellisatie

Aangezien er limieten waren op de werkende klokfrequentie, werd er gekeken hoe parallellisatie ingevoerd kon worden om de berekeningstijd nog te verlagen. In deel 4.4 werd al beschreven hoe dit praktisch aangepakt werd. De parallellisatie werd uitgevoerd bij een parallellisatiegraad van 2 en 4. Die eerste is echter gelijk aan de gewone seriële implementatie, maar de extra logica voor de parallellisatie werd eraan toegevoegd. Op de resultaten

in figuur 5.9 is te zien dat de gewenste functionaliteit behouden blijft ongeacht de parallellisatiegraad. Daaruit kan geconcludeerd worden dat de parallellisatie goed geïmplementeerd werd.



(a) parellellisatiegraad van 1 (b) parellellisatiegraad van 2

(c) parellellisatiegraad van 4

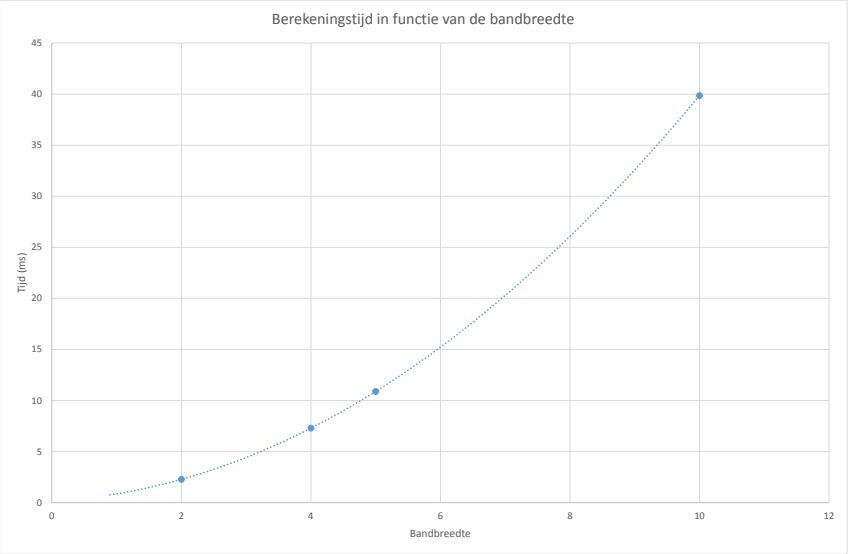
Figuur 5.9: Heatmap bij verschillende parallellisatiegraad.

5.6 Timing

Om de resultaten objectief te kunnen bekijken, werd er ook een tijdsmeting gedaan op de verschillende implementaties. De resltaten van die metingen zijn te zien in tabel 5.1. Als eerste wordt er gekeken naar hoe het tijdsverloop is in functie van de bandbreedte, zoals te zien is in figuur 5.10. Daar is een tweedegraadsfunctie te zien, wat wil zeggen dat de berekeningstijd kwadratisch stijgt met de bandbreedte.

Tabel 5.1: Tijdsmetingen bij verschillende bandbreedte en parallellisatiegraad uitgedrukt in milliseconden, grijze vakken werden niet gemeten.

		Bandbreedte			
		2	4	5	10
Parallellisatiegraad	1	2,281	7,302	10,89	39,85
	1 met extra func.		60,93		
	2		80,43		
	4		40,22		



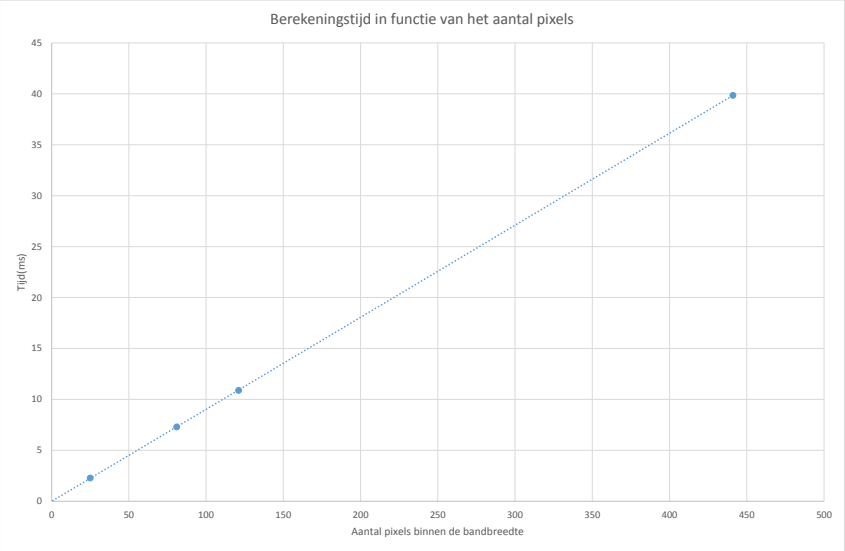
Figuur 5.10: Timing van de heatmapberekening in functie van de bandbreedte.

Om het kwadratisch verband aan te tonen werd dezelfde berekeningstijd geplot in functie van het aantal pixels die berekend moeten worden binnen de bandbreedte. In figuur 5.11 is te zien dat dit verband lineair is. Per pixel die berekend moet worden is er een vaste berekeningstijd en hoe meer pixels hoe groter de berekeningstijd in totaal. Wanneer we de bandbreedte gaan omvormen naar het aantal pixels (vergelijking 5.1) kan worden vastgesteld dat het aantal pixels kwadratisch stijgt met de bandbreedte. Aangezien het aantal pixels lineair zijn met de berekeningstijd, kan geconcludeerd worden dat de berekeningstijd kwadratisch stijgt met de bandbreedte (vergelijking 5.2).

$$\begin{aligned} h &= h_1 = h_2 \\ \text{aantal pixels} &= (2h_1 + 1) * (2h_2 + 1) \\ &= (2h + 1)^2 \end{aligned} \tag{5.1}$$

$$\begin{aligned} a &= \text{constante waarde} \\ b &= \text{constante waarde} \\ \text{tijd} &= a * \text{aantal pixels} + b \\ &= a * (2h + 1)^2 + b \\ &= 4ah^2 + 4ah + (a + b) \end{aligned} \tag{5.2}$$

De berekeningstijd wordt niet alleen beïnvloed door de bandbreedte van de “Density Esti-

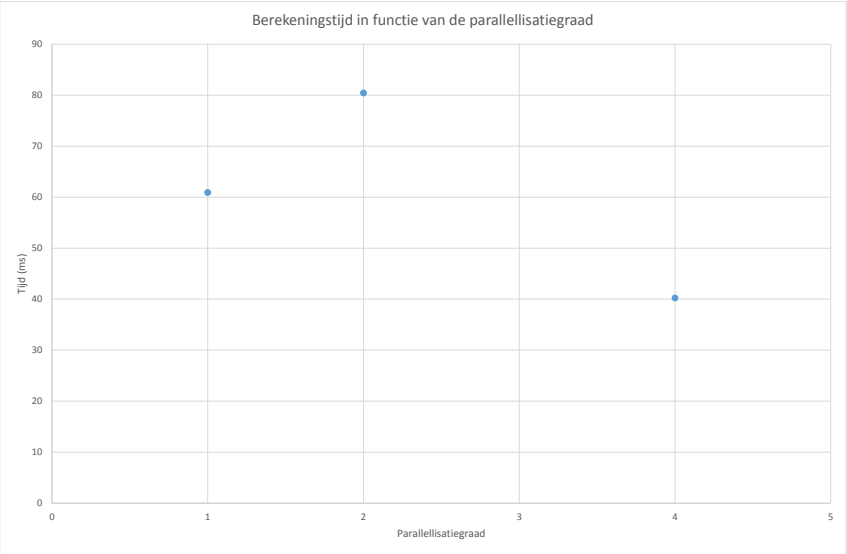


Figuur 5.11: Timing van de heatmapberekening in functie van het aantal pixels binnen de bandbreedte.

mator". De berekeningstijd kan immers verlaagd worden door parallellisatie in te voeren. Op de grafiek in figuur 5.12 is te zien dat de parallellisate pas werkt bij een parallellisatiegraad van vier waar de berekeningstijd de helft is van die tijd bij een parallellisatiegraad van twee. Deze tijd is echter nog veel trager dan de behaalde tijd bij een bandbreedte van vier zonder parallellisatie (tabel 5.1).

Van het in Python geschreven algoritme werd er ook een tijdsmeting gedaan. Het algoritme werd getest op drie afzonderlijke computers. In tabel 5.2 zijn de resultaten te zien. Die punten zijn willekeurig gekozen. Per processor en dataset werden er tien metingen gedaan. De resultaten in de tabel zijn dus de gemiddelden van de tien metingen die er gedaan werden per categorie. Wanneer bij deze tijden de hardware versnelling bepaald wordt, dan kan vastgesteld worden dat die versnelling miniem is. Merk hierbij wel op dat de waarden van die tijdsmetingen sterk schommelen en afhankelijk zijn van welke andere programma's er draaien op de computer. Om die schommelingen tegen te gaan werd er een gemiddelde toegevoegd.

Een mogelijk reden voor de kleine versnelling is de uitzonderlijk hoge rekensnelheid in software. Aangezien er gebruik gemaakt wordt van een kleine dataset, is het mogelijk voor de processor om alle data op te slaan in zijn cache-geheugen. Dit zorgt voor lage toegangstijden bijn het opvragen van de data zodat de software redelijk snel uitgevoerd kan worden. Daarom werd de software getest met een dataset van 249.000 punten. Bij het



Figuur 5.12: Timing van de heatmapberekening in functie van de parallellisatiegraad bij een bandbreedte van 4.

vergelijken van de tijden bij 249 en 249.000 punten, dan is te zien dat de rekentijd evenredig is met het aantal punten. Ook bij een dataset van 2.490.000 en 24.900.000 punten blijft die evenredigheid aanwezig. De versnelling blijft dezelfde. De stelling omtrent het cachegeheugen kan hierdoor ontkracht worden.

Tabel 5.2: Tijdsmetingen bij een bandbreedte van 4 en verschillende dataset op verschillende computers uitgedruk in seconden.

		Computer (processor)		
		HP (Intel i5)	Dell (Intel i5)	Asus (Intel i7)
Aantal punten	249	0,039	0,039	0,013
	249k	29,360	29,360	13,690
	2.49M	314.720		
	24.9M	3264,088		

Tabel 5.3: Bekomen versnelling van de hardware ten opzichte van de verschillende geteste computers

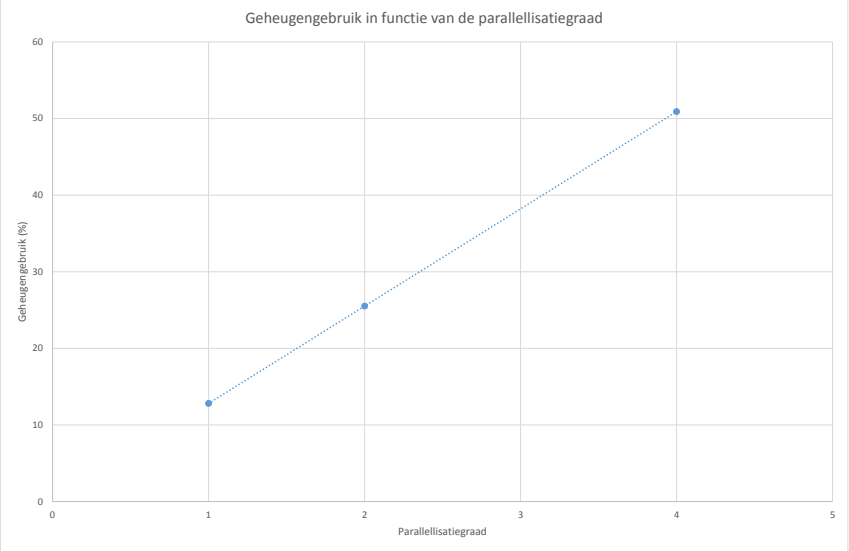
Computer (processor)	Versnelling
HP (Intel i5)	5,388
Dell (Intel i5)	5,388
Asus (Intel i7)	1,726

5.7 Geheugengebruik

Aangezien de FPGA een beperkte hoeveelheid aan BRAM en DFF's heeft, moet er ook eens gekeken worden naar het geheugengebruik bij de parallellisatie (tabel 5.4). Door dit geheugengebruik te analyseren, kunnen betere afwegingen gemaakt worden tussen berekeningstijd en kwaliteit. De grafiek in figuur 5.13 drukt het geheugengebruik procentueel uit ten opzichte van het totale beschikbare geheugen van de FPGA. Er wordt enkel gekeken naar BRAM. Bij een verdubbeling van de parallellisatiegraad kan een verdubbeling van het geheugengebruik beschouwd worden. Het lineaire verband wordt met de rechte op de grafiek aangetoond. Er is de mogelijkheid om naar een parallellisatiegraad van acht te gaan, mits enige wijziging aan het gebruik zodat er bij een parallellisatiegraad van vier minder dan 50% gebruikt wordt. Dit zorgt wel voor extra kwaliteitsverlies.

Tabel 5.4: Gebruik van BRAM en DFF's procentueel uitgedrukt in functie van de parallellisatiegraad

Parallellisatiegraad	BRAM (%)	DFF (%)
1	12,81	0,04
2	25,51	0,05
4	50,90	0,07



Figuur 5.13: Gebruik BRAM in functie van de parallellisatiegraad.

In tabel 5.4 is te zien dat er van de totale hoeveelheid DFF's, er zeer weinig gebruikt worden door het systeem. Een mogelijkheid om de hogere parallellisatiegraad te verkrijgen

zonder kwaliteit te verliezen, is het creëren van distributed RAM met DFF's. Om dit te verwezenlijken zullen DFF's samen genomen worden om één geheugen te vormen. Aangezien er voldoende DFF's over zijn, is dit een mogelijke optie om geheugen te verkrijgen zonder kwaliteit te verliezen.

5.8 Toekomstig werk

De gehele uitwerking van het systeem, zoals te zien in figuur 2.5, is een werk van meerdere jaren. In deze thesis werd de functionaliteit van de 'Density Estimator' met succes geïmplementeerd. Omdat de bekomen versnelling kleiner is dan verwacht, kan in de toekomst geprobeerd worden om het algoritme nog meer te versnellen door pipelining in te voeren. Wanneer er data nodig is uit het geheugen, zal de FSM wachten tot de data stabiel aan de uitgang van het geheugen staat. Telkens wanneer er data opgevraagd wordt uit het geheugen gaan dus twee klokcycli verloren. Door pipelining in te voeren, zal er gezorgd worden dat het proces een andere actie uitvoert terwijl er gewacht wordt op het geheugen. Zo kan er een grotere versnelling bekomen worden.

Daarnaast kan er ook onderzocht worden hoe de geïmplementeerde hardware in een gehele hardware accelerator kan passen. Verder kan er dan onderzoek gedaan worden naar sensoren en camera's en hoe de beeldverwerking zal gebeuren op beelden van camera's om zo de gewenste data naar de hardware accelerator te sturen. De sensoren en camera's zijn wel afhankelijk van de gebruikte toepassing van de heatmaps.

Hoofdstuk 6

Besluit

Om grote hoeveelheden data op een relatief eenvoudige manier voor te stellen, wordt er vaak gebruik gemaakt van heatmaps. Een nadeel is echter dat het bij grote hoeveelheden veel tijd kost om de gehele heatmap op te stellen. Een mogelijkheid om die hoge berekeningstijd tegen te gaan, is om het algoritme uit te voeren in hardware.

Als eerste werd het algoritme in C geïmplementeerd om een eerste analyse te kunnen doen van de verschillende algoritmes en kernels. De daardoor verkregen kennis werd gebruikt om sneller een hardware implementatie te bekomen. Daarnaast werd er ook een tijdsmeting gedaan van het algoritme in Python om een vergelijking te kunnen doen met de hardware.

In deze thesis werd er enkel onderzocht hoe de concrete implementatie gedaan kon worden in hardware. De totale berekening zou moeten gebeuren door een hardware accelerator waar er een co-existentie is van hardware en software om tot een versnelling te leiden. In de software zou de data voorbereid kunnen worden om zo met de ideale vorm te werken in hardware. Daarnaast kan de software de resultaten gaan afbeelden op een onderliggende kaart zodat de heatmap meer betekenis heeft voor gebruikers. De combinatie van de geïmplementeerde hardware met software om een volledige hardware accelerator te bekomen, kan het onderwerp zijn van een volgende thesis.

Tijdens het implementeren van de hardware werd de focus eerst gelegd op de sturing van het VGA-scherm. Het correct sturen van dit scherm zou ervoor zorgen dat de heatmaps eenvoudig afgebeeld konden worden. Daardoor konden er beter fouten uit de implementatie gehaald worden.

Na de sturing van het VGA-scherm werd een eerste implementatie gedaan van het algoritme in hardware. De bekomen versnelling was relatief klein ten opzichte van de berekeningstijd in Python. Het daarbij horende verlies van kwaliteit maakt dat de berekening in hardware amper een voordeel heeft ten opzichte van de software. Merk hierbij wel op dat de berekeningen gedaan werden op een relatief kleine dataset en dat de hardware meer tot zijn recht zal komen bij grotere berekeningen. Indien men eventueel over meer geheugen zou beschikken kan het kwaliteitsverlies opgelost worden.

Als laatste werd er ook parallellisatie toegevoegd om een nog hogere versnelling te bekomen. Die parallellisatie leek echter voor extra berekeningstijd te zorgen, waardoor de versnelling zelf niet behaald werd. Met een groter geheugen kan er een grotere parallellisatiegraad behaald worden, maar de vraag is ofdat die hogere parallellisatiegraad zal leiden tot een versnelling.

Bibliografie

[1] Bruneel K. & Stroobandt D. Davidson t., Merlier M. A dynamically reconfigurable pattern matcher for regular expressions on fpga. 2012.

[2] Marron J.S. & Heather S.J. Jones M.C. A brief survey of bandwidth selection for density estimation. *Journal of the American Statistical Association*, 91(433):401–407, March 1996.

[3] D'Hollander E.H. Touhafi A. Cornelis J.G & Lemeire J. da Silva B, Braeken A. Comparing and combining gpu and fpga accelerators in an image processing context. 2013.

[4] B.A. Turlach. Bandwidth selection in kernel density estimation: A review.

[5] Silverman B.W. Density estimation for statistics & data analysis. 1986.

[6] Wilkinson L. & Friendly M. The history of the cluster heat map. *The American Statistician*, pages 179–184, 2009.

[7] Vanderbauwhede W. & Benkrid K. High-performance computing using fpgas. 798:6, 2014.

[8] Bojko A. Informative or misleading? heatmaps deconstructed.

[9] Baker K. & Brackman P. Routeyou data voor fietsbarometer oost-vlaanderen 2015 (toerisme oost-vlaanderen).

[10] Al Farisi B. Stroobandt D. Kadlcek O. & Pell O. Heyse K., Basteleus J. On the impact of replacing low-speed configuration buses on fpgas with the chip's internal configuration infrastructure. 18, 2010.

[11] Huss S. Rumpf M. & Strzodka R. Klupsch S., Ernst M. Real time image processing based on reconfigurable hardware acceleration.

[12] Konecny M. & Zlatanova S. Bandrova T. *Thematic cartography for the society.* Springer, 2014.

[13] Vanden Eynden K. Ontwerp van een hardwareaccelerator voor het real time berekenen van heat maps met een fpga. Master's thesis, Universiteit Gent, Juni 2015.

Bijlage A

Broncode

A.1 Implementatie in VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity AddValue is
        generic (
                BANDWIDTH : integer := 4
        );
        Port (
                -- Clocking and Reset
                clk     : in    STD_LOGIC;
                reset   : in    STD_LOGIC;

                -- Signals to indicate if component is busy or should start
                valid   : in    STD_LOGIC_VECTOR (0 downto 0);
                busy    : out   STD_LOGIC_VECTOR (0 downto 0);

                -- Data to write in RAM
                dataPointX  : in    STD_LOGIC_VECTOR (9  downto 0);
                dataPointY  : in    STD_LOGIC_VECTOR (8  downto 0);
                value       : in    STD_LOGIC_VECTOR (4 downto 0);

                -- Data to RAM
```

```
                ramWe       : out   STD_LOGIC_VECTOR (0 downto 0);
                ramAddr     : out   STD_LOGIC_VECTOR (18 downto 0);
                ramDataIn   : out   STD_LOGIC_VECTOR (5 downto 0);
                ramDataOut  : in    STD_LOGIC_VECTOR (5 downto 0)
        );
end AddValue;

architecture Behavioral of AddValue is
        --State type definitions
        type AddValueState is (IDLE_ADD_VALUE, CALC_ADDRESS, READ_ADDRESS, RAM_WAIT1,
        signal presAddValueState    : AddValueState;
        signal nextAddValueState    : AddValueState;

        type bandwidthState is (IDLE_BANDWIDTH, ADD_VALUE_START, WAIT_FOR_ADD,
                NEXT_PIXEL);
        signal presBandwidthState   : bandwidthState;
        signal nextBandWidthState   : bandwidthState;

        signal ramAddrSig           : std_logic_vector (18 downto 0);
        signal ramDataInSig         : std_logic_vector (5 downto 0);
        signal xCoSig               : std_logic_vector (9  downto 0);
        signal yCoSig               : std_logic_vector (8  downto 0);
        signal valueSig             : std_logic_vector (4 downto 0);
        signal startAddSig          : std_logic;
        signal addBusySig           : std_logic;
        signal nextPixelEnableSig   : std_logic;
        signal pixelDoneSig         : std_logic;
        signal dataPointXSig        : std_logic_vector (9 downto 0);
        signal dataPointYSig        : std_logic_vector (8 downto 0);
        signal valueInSig           : std_logic_vector (4 downto 0);

begin
        next_state : process(clk, reset)
        begin
                if reset = '1' then
                        presAddValueState   <= IDLE_ADD_VALUE;
                        presBandwidthState  <= IDLE_BANDWIDTH;
                        elsif rising_edge(clk) then
                        presAddValueState   <= nextAddValueState;
                        presBandwidthState  <= nextBandwidthState;
                end if;
        end process next_state;
```

```
define_next_add_value_state : process (presAddValueState, xCoSig, yCoSig,
        valueSig, startAddSig, ramDataOut)
begin
        case presAddValueState is
                when IDLE_ADD_VALUE =>
                        if startAddSig = '1' then
                                nextAddValueState   <= CALC_ADDRESS;
                        else
                                nextAddValueState   <= IDLE_ADD_VALUE;
                        end if;
                        addBusySig      <= '0';
                        ramWe           <= "0";
                        ramAddrSig      <= ramAddrSig;
                        ramDataInSig    <= ramDataInSig;
                when CALC_ADDRESS   =>
                        addBusySig          <= '1';
                        ramWe               <= "0";
                        ramAddrSig          <= (yCoSig * "1010000000")
                                + xCoSig;
                        ramDataInSig        <= (others => '0');
                        nextAddValueState   <= RAM_WAIT1;
                when RAM_WAIT1      =>
                        addBusySig          <= '1';
                        ramWe               <= "0";
                        ramAddrSig          <= ramAddrSig;
                        ramDataInSig        <= (others => '0');
                        nextAddValueState   <= READ_ADDRESS;
                when READ_ADDRESS   =>
                        addBusySig          <= '1';
                        ramWe               <= "0";
                        ramAddrSig          <= ramAddrSig;
                        ramDataInSig        <= ramDataOut;
                        nextAddValueState   <= ADD_VALUE;
                when ADD_VALUE      =>
                        addBusySig          <= '1';
                        ramWe               <= "0";
                        ramAddrSig          <= ramAddrSig;
                        ramDataInSig        <= ramDataOut + valueSig;
                        nextAddValueState   <= WRITE_DATA;
                when WRITE_DATA     =>
                        addBusySig          <= '1';
                        ramWe               <= "1";
                        ramAddrSig          <= ramAddrSig;
```

```
                        ramDataInSig        <= ramDataInSig;
                        nextAddValueState   <= RAM_WAIT2;
                when RAM_WAIT2      =>
                        addBusySig          <= '1';
                        ramWe               <= "1";
                        ramAddrSig          <= ramAddrSig;
                        ramDataInSig        <= ramDataInSig;
                        nextAddValueState   <= IDLE_ADD_VALUE;
        end case;
end process define_next_add_value_state;

define_next_bandwidth_state : process(presBandwidthState, dataPointX,
        dataPointY, value, valid, addBusySig, pixelDoneSig)
begin
        case presBandwidthState is
                when IDLE_BANDWIDTH     =>
                        if valid = "1" then
                                nextBandwidthState  <= ADD_VALUE_START;
                        else
                                nextBandWidthState  <= IDLE_BANDWIDTH;
                        end if;
                        startAddSig         <= '0';
                        busy                <= "0";
                        nextPixelEnableSIg  <= '0';
                        dataPointXSig       <= dataPointX;
                        dataPointYSig       <= dataPointY;
                        valueInSig          <= value;
                when ADD_VALUE_START    =>
                        startAddSig         <= '1';
                        busy                <= "1";
                        nextPixelEnableSIg  <= '0';
                        nextBandwidthState  <= WAIT_FOR_ADD;
                        dataPointXSig       <= dataPointXSig;
                        dataPointYSig       <= dataPointYSig;
                        valueInSig          <= valueInSig;
                when WAIT_FOR_ADD       =>
                        startAddSig         <= '0';
                        busy                <= "1";
                        nextPixelEnableSIg  <= '0';
                        if addBusySig = '0' then
                                nextBandwidthState  <= NEXT_PIXEL;
                        else
                                nextBandwidthState  <= WAIT_FOR_ADD;
```

```
                        end if;
                        dataPointXSig       <= dataPointXSig;
                        dataPointYSig       <= dataPointYSig;
                        valueInSig          <= valueInSig;
                when NEXT_PIXEL         =>
                        startAddSig         <= '0';
                        busy                <= "1";
                        nextPixelEnableSIg  <= '1';
                        if pixelDoneSig = '1' then
                                nextBandwidthState  <= IDLE_BANDWIDTH;
                        else
                                nextBandwidthState  <= ADD_VALUE_START;
                        end if;
                        dataPointXSig       <= dataPointXSig;
                        dataPointYSig       <= dataPointYSig;
                        valueInSig          <= valueInSig;
        end case;
end process define_next_bandwidth_state;

next_pixel_proc : process(clk, nextPixelEnableSig, dataPointXSig,
        dataPointYSig, valueInSig)
        variable h          : integer range 0 to BANDWIDTH + 1 := 0;
        variable v          : integer range 0 to BANDWIDTH + 1 := 0;
        variable dataX      : integer range 0 to 559 := 0;
        variable dataY      : integer range 0 to 479 := 0;
        variable x          : integer range 0 to 559 := 0;
        variable y          : integer range 0 to 479 := 0;
        variable positiveX  : std_logic := '1';
        variable positiveY  : std_logic := '1';
        variable valueIn    : integer range 0 to 8191 := 0;
        variable valueOut   : integer range 0 to 8191 := 0;
begin
        if rising_edge(clk) then
                dataX   := conv_integer(dataPointXSig);
                dataY   := conv_integer(dataPointYSig);
                valueIn := conv_integer(valueInSig);
                if positiveX = '1' then
                        x   := dataX + h;
                else
                        x   := dataX - h;
                end if;
                if positiveY = '1' then
                        y   := dataY + v;
```

```
                        else
                                y    := dataY - v;
                        end if;
                        valueOut     := valueIn; --Uniform Kernel (PDE)
                        if nextPixelEnableSig = '1' then
                                h    := h + 1;
                                if h = (BANDWIDTH + 1) then
                                        if positiveX = '1' then
                                                h    := 1;
                                        else
                                                h    := 0;
                                                v    := v + 1;
                                        end if;
                                        if v = (BANDWIDTH + 1) then
                                                if positiveY = '1' then
                                                        v    := 1;
                                                else
                                                        v    := 0;
                                                end if;
                                                positiveY    := not positiveY;
                                        end if;
                                        positiveX    := not positiveX;
                                end if;
                        end if;
                        if (positiveX = '0' and positiveY = '0' and h = BANDWIDTH
                                and v = BANDWIDTH) then
                                pixelDoneSig     <= '1';
                        else
                                pixelDoneSig     <= '0';
                        end if;
                        xCoSig       <= conv_std_logic_vector (x, 10);
                        yCoSig       <= conv_std_logic_vector (y, 9);
                        valueSig     <= conv_std_logic_vector (valueOut, 5);
                end if;
        end process next_pixel_proc;

        ramAddr      <= ramAddrSig;
        ramDataIn    <= ramDataInSig;
end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
use IEEE.STD_LOGIC_ARITH.ALL;

entity DensityEstimator is
        Generic (
                NUMB_OF_COMP    : integer := 4
        );
        Port (
                HS          : out   std_logic;
                VS          : out   std_logic;
                vga_r       : out   std_logic_vector (4  downto 0);
                vga_g       : out   std_logic_vector (5  downto 0);
                vga_b       : out   std_logic_vector (4  downto 0);
                start       : in    STD_LOGIC;
                finished    : out   STD_LOGIC;
                --clk         : in    STD_LOGIC;
                sysclk_p    : in    STD_LOGIC;
                sysclk_n    : in    STD_LOGIC;
                reset_n       : in    STD_LOGIC
        );
end DensityEstimator;

architecture Behavioral of DensityEstimator is

        --DCM declaration
        component clk200to25MHz
        port (-- Clock in ports
        clk_in1_p    : in     std_logic;
        clk_in1_n    : in     std_logic;
        -- Clock out ports
        clk_out1     : out    std_logic;
        clk_out2     : out   std_logic;
        -- Status and control signals
        reset        : in     std_logic;
        locked       : out    std_logic
        );
        end component;

        --RAM declaration
        COMPONENT estimation_ram
        PORT (
        clka    : IN    STD_LOGIC;
        rsta    : IN    STD_LOGIC;
        wea     : IN    STD_LOGIC_VECTOR(0  DOWNTO 0);
```

```
addra   : IN    STD_LOGIC_VECTOR(18 DOWNTO 0);
dina    : IN    STD_LOGIC_VECTOR(5 DOWNTO 0);
douta   : OUT   STD_LOGIC_VECTOR(5 DOWNTO 0);
clkb    : IN    STD_LOGIC;
rstb    : IN    STD_LOGIC;
web     : IN    STD_LOGIC_VECTOR(0  DOWNTO 0);
addrb   : IN    STD_LOGIC_VECTOR(18 DOWNTO 0);
dinb    : IN    STD_LOGIC_VECTOR(5 DOWNTO 0);
doutb   : OUT   STD_LOGIC_VECTOR(5 DOWNTO 0)
);
END COMPONENT;

COMPONENT init_rom
PORT (
clka    : IN    STD_LOGIC;
addra   : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
douta   : OUT   STD_LOGIC_VECTOR(23 DOWNTO 0)
);
END COMPONENT;

COMPONENT AddValue is
Port (
-- Clocking and Reset
clk     : in    STD_LOGIC;
reset   : in    STD_LOGIC;

-- Signals to indicate if component is busy or should start
valid   : in    STD_LOGIC_VECTOR (0 downto 0);
busy    : out   STD_LOGIC_VECTOR (0 downto 0);

-- Data to write in RAM
dataPointX  : in    STD_LOGIC_VECTOR (9  downto 0);
dataPointY  : in    STD_LOGIC_VECTOR (8  downto 0);
value       : in    STD_LOGIC_VECTOR (4 downto 0);

-- Data to RAM
ramWe       : out   STD_LOGIC_VECTOR (0 downto 0);
ramAddr     : out   STD_LOGIC_VECTOR (18 downto 0);
ramDataIn   : out   STD_LOGIC_VECTOR (5 downto 0);
ramDataOut  : in    STD_LOGIC_VECTOR (5 downto 0)
);
END COMPONENT;
```

```
--RAM signals
signal ramWeASig        : std_logic_vector (NUMB_OF_COMP - 1  downto 0);
signal ramAddrASig      : std_logic_vector ((NUMB_OF_COMP * 19)
        - 1 downto 0);
signal ramDataInASig    : std_logic_vector ((NUMB_OF_COMP * 6)
        - 1 downto 0);
signal ramDataOutASig   : std_logic_vector ((NUMB_OF_COMP * 6)
        - 1 downto 0);
signal ramAddrBSig      : std_logic_vector (19 downto 0);
signal ramDataOutBSig   : std_logic_vector ((NUMB_OF_COMP * 6)
        - 1 downto 0);
signal ramDataAddition  : std_logic_vector (5 downto 0);
alias  vgaDataSig       : std_logic_vector (5  downto 0) is ramDataAddition
        (5 downto 0);

--ROM signals
signal romAddrSig       : std_logic_vector (7  downto 0);
signal romDataSig       : std_logic_vector (23 downto 0);
alias  dataPointXsig    : std_logic_vector (9  downto 0) is romDataSig (23
        downto 14);
alias  dataPointYSig    : std_logic_vector (8  downto 0) is romDatasig (13
        downto 5);
alias  valueSig         : std_logic_vector (4 downto 0) is romDataSig (4
        downto  0);

--DCM signals
signal clk25MHz         : std_logic;
signal clk150MHz        : std_logic;
signal clk50Hz          : std_logic := '0';
signal reset            : std_logic;

--signals to communicate between processes
signal validSig             : std_logic;
signal busySig              : std_logic;
signal vga_clk              : std_logic;
signal enableNextRomAddr    : std_logic;
signal clockedStartSig      : std_logic;
signal validComponents      : std_logic_vector (NUMB_OF_COMP - 1 downto 0);
signal busyComponents       : std_logic_vector (NUMB_OF_COMP - 1 downto 0);

--State type definitions
type ReadRomState is (IDLE_READ_ROM, ROM_WAIT1, ROM_WAIT2, PROCESS_INPUT,
        WAIT_FOR_PROCESS, NEXT_INPUT);
```

```
        signal presReadRomState     : ReadRomState;
        signal nextReadRomState     : ReadRomState;

        --VGA signals
        signal rowSig                       : std_logic_vector (9 downto 0);
        signal colSig                       : std_logic_vector (9 downto 0);
        signal rowBuf1, rowBuf2                             : std_logic_vector (9
                downto 0);
        signal colBuf1, colBuf2                             : std_logic_vector (9
                downto 0);

        --Constants of the system
        constant SIZE_OF_ROM        : std_logic_vector (7 downto 0) := "11111001";
        constant h_pixels   : integer := 640;
        constant h_fp       : integer := 16;
        constant h_pulse    : integer := 96;
        constant h_bp       : integer := 48;
        constant v_pixels   : integer := 480;
        constant v_fp       : integer := 10;
        constant v_pulse    : integer := 2;
        constant v_bp       : integer := 33;

begin
        rom_0       : init_rom port map(clk150MHz, romAddrSig, romDataSig);
        clk_0       : clk200to25MHz port map (sysclk_p, sysclk_n, clk25MHz,
                clk150MHz, reset, open);

        comp_inst : for i in 0 to NUMB_OF_COMP - 1 generate
        inst_ram : estimation_ram port map(clk150MHz, reset, ramWeASig(i
                downto i), ramAddrASig((i+1)*19 - 1 downto (i*19)),
                ramDataInASig((i+1)*6 - 1 downto (i*6)),
                ramDataOutASig((i+1)*6 - 1 downto (i*6)),
                clk150MHz, reset, "0", ramAddrBSig(18 downto 0), (others => '0'),
                ramDataOutBSig((i+1)*6 - 1 downto (i*6)));
        inst_add_value : AddValue port map(clk150MHz, reset, validComponents
                (i downto i), busyComponents(i downto i), dataPointXSig,
                dataPointYSig, ValueSig, ramWeASig(i downto i),
                ramAddrASig((i+1)*19 - 1 downto (i*19)),
                ramDataInASig((i+1)*6 - 1 downto (i*6)),
                ramDataOutASig((i+1)*6 - 1 downto (i*6)));
        end generate comp_inst;

        reset   <= not reset_n;
```

```
clk25MHzTo50Hz: process(reset, clk25MHz)
        variable count  : integer range 0 to 250000 := 0;
begin
        if rising_edge(clk25MHz) then
                count := count + 1;
                if count = 250000 then
                        count := 0;
                        clk50Hz <= not clk50Hz;
                else
                        clk50Hz <= clk50Hz;
                end if;
        end if;
end process clk25MHzTo50Hz;

next_state : process(clk150MHz, reset)
begin
        if reset = '1' then
                presReadRomState    <= IDLE_READ_ROM;
        elsif rising_edge(clk150MHz) then
                presReadRomState    <= nextReadRomState;
        end if;
end process next_state;

define_next_read_rom_state : process(presReadRomState, start, busySig,
        romAddrSig)
begin
        case presReadRomState is
                when IDLE_READ_ROM      =>
                        validSig            <= '0';
                        finished            <= '1';
                        enableNextRomAddr   <= '0';
                        if start = '1' then
                                nextReadRomState    <= ROM_WAIT1;
                        else
                                nextReadRomState    <= IDLE_READ_ROM;
                        end if;
                when ROM_WAIT1          =>
                        validSig            <= '0';
                        finished            <= '0';
                        enableNextRomAddr   <= '0';
                        nextReadRomState    <= ROM_WAIT2;
                when ROM_WAIT2          =>
```

```
                        validSig            <= '0';
                        finished            <= '0';
                        enableNextRomAddr   <= '0';
                        nextReadRomState    <= PROCESS_INPUT;
                when PROCESS_INPUT      =>
                        validSig            <= '1';
                        finished            <= '0';
                        enableNextRomAddr   <= '0';
                        nextReadRomState    <= WAIT_FOR_PROCESS;
                when WAIT_FOR_PROCESS   =>
                        validSig            <= '0';
                        finished            <= '0';
                        enableNextRomAddr   <= '0';
                        if busySig = '1' then
                                nextReadRomState    <= WAIT_FOR_PROCESS;
                        else
                                nextReadRomState    <= NEXT_INPUT;
                        end if;
                when NEXT_INPUT         =>
                        validSig            <= '0';
                        finished            <= '0';
                        enableNextRomAddr   <= '1';
                        if romAddrSig = (SIZE_OF_ROM - 1) then
                                nextReadRomState    <= IDLE_READ_ROM;
                        else
                                nextReadRomState    <= ROM_WAIT1;
                        end if;
        end case;
end process define_next_read_rom_state;

clock_start : process(clk50Hz, start)
begin
        if rising_edge(clk50Hz) then
                if start = '1' then
                        clockedStartSig <= '1';
                else
                        clockedStartSig <= '0';
                end if;
        end if;
end process;

next_rom_addr : process(reset, clk150MHz, enableNextRomAddr)
begin
```

```
        if reset = '1' then
                romAddrSig  <= (others => '0');
        elsif rising_edge(clk150MHz) then
                if enableNextRomAddr = '1' then
                        if romAddrsig = (SIZE_OF_ROM - 1) then
                                romAddrSig  <= (others => '0');
                        else
                                romAddrSig  <= romAddrSig + 1;
                        end if;
                else
                        romAddrsig  <= romAddrSig;
                end if;
        end if;
end process;

valid_encode : process(validSig, busyComponents)
begin
        if validSig = '1' then
                case busyComponents is
                        when "0000"   => validComponents <= "0001";
                        when "0001"   => validComponents <= "0010";
                        when "0010"   => validComponents <= "0001";
                        when "0011"   => validComponents <= "0100";
                        when "0100"   => validComponents <= "0001";
                        when "0101"   => validComponents <= "0010";
                        when "0110"   => validComponents <= "0001";
                        when "0111"   => validComponents <= "1000";
                        when "1000"   => validComponents <= "0001";
                        when "1001"   => validComponents <= "0010";
                        when "1010"   => validComponents <= "0001";
                        when "1011"   => validComponents <= "0100";
                        when "1100"   => validComponents <= "0001";
                        when "1101"   => validComponents <= "0010";
                        when "1110"   => validComponents <= "0001";
                        when "1111"   => validComponents <= "0000";
                        when others => validComponents <= "0000";
                end case;
        else
                validComponents  <= "0000";
        end if;
end process valid_encode;

busy_decode : process(busyComponents)
```

```
begin
        if busyComponents = "1111" then
                busySig <= '1';
        else
                busySig <= '0';
        end if;
end process;

count_pixels : process(reset, clk25MHz)
        variable row    : integer range 0 to 525 := 0;
        variable col    : integer range 0 to 800 := 0;
        variable a      : integer range 0 to 420000 := 0;
begin
        if reset = '1' then
                row := 0;
                col := 0;
        elsif rising_edge(clk25MHz) then
                col := col + 1;
                if col = (h_pixels + h_fp + h_pulse + h_bp) then
                        col := 0;
                        row :=  row + 1;
                        if row = (v_pixels + v_fp + v_pulse + v_bp) then
                                row := 0;
                        end if;
                end if;
                if (row < v_pixels) and (col < h_pixels) then
                        a    := (row * h_pixels) + col;
                else
                        a    := 0;
                end if;
                rowSig      <= conv_std_logic_vector(row, 10);
                colsig      <= conv_std_logic_vector(col, 10);
                ramAddrBSig <= conv_std_logic_vector(a, 20);
        end if;
end process count_pixels;

hs_vs : process (rowBuf2, colbuf2)
        variable row    : integer range 0 to (v_pixels + v_fp + v_pulse
                + v_bp) := 0;
        variable col    : integer range 0 to (h_pixels + h_fp + h_pulse
                + h_bp) := 0;
begin
        row := conv_integer(rowBuf2);
```

```
            col := conv_integer(colBuf2);
            if (row < (v_pixels + v_fp)) or (row >= (v_pixels + v_fp + v_pulse))
                    VS  <= '1';
            else
                    VS  <= '0';
            end if;
            if (col < (h_pixels + h_fp)) or (col >= (h_pixels + h_fp + h_pulse))
                    HS  <= '1';
            else
                    HS  <= '0';
            end if;
    end process hs_vs;

    row_col_wait1 : process(reset, clk25MHz, rowSig, colSig)
    begin
            if reset = '1' then
                    rowBuf1 <= (others => '0');
                    colBuf1 <= (others => '0');
            elsif rising_edge(clk25MHz) then
                    rowBuf1 <= rowSig;
                    colBuf1 <= colSig;
            end if;
    end process row_col_wait1;

    row_col_wait2 : process(reset, clk25MHz, rowBuf1, colBuf1)
    begin
            if reset = '1' then
                    rowBuf2 <= (others => '0');
                    colBuf2 <= (others => '0');
            elsif rising_edge(clk25MHz) then
                    rowBuf2 <= rowBuf1;
                    colBuf2 <= colBuf1;
            end if;
    end process row_col_wait2;

    ram_data_addition : process(ramDataOutBSig)
    begin
            ramDataAddition <= ramDataOutBSig (23 downto 18) + ramDataOutBSig(17
                    downto 12) + ramDataOutBSig(11 downto 6)
                    + ramDataOutBSig(5 downto 0);
    end process ram_data_addition;

    rgb : process (rowBuf2, colBuf2)
```

```
                variable row    : integer := 0;
                variable col    : integer := 0;
        begin
                row := conv_integer(rowBuf2);
                col := conv_integer(colBuf2);
                if (row < v_pixels) and (col < h_pixels) then
                        vga_r   <= vgaDataSig (5 downto 1);
                        vga_g   <= "111111" - vgaDataSig;
                        vga_b   <= (others => '0');
                else
                        vga_r   <= (others => '0');
                        vga_g   <= (others => '0');
                        vga_b   <= (others => '0');
                end if;
        end process rgb;
end Behavioral;
```

A.2 Implementatie in C

```
#include "EstimationFunctions.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double EstimationFunctionsCalculatePoint(int xCo, int yCo,
        long dataSet[][2], long dataSetSize, int bandwidthHorizontal,
        int bandwidthVertical);
double EstimationFunctionsWeightFunction(int xCo, int yCo, int sampleX, int sampleY,
        int bandwidthHorizontal, int bandwidthVertical);

void EstimationFunctionsCalculateAllPoints(double heatmap[HEIGHT][WIDTH],
        long dataSet[][2], long dataSetSize, int bandwidthHorizontal,
        int bandwidthVertical){
        printf("-------- estimation started\n");
        for (long y = 0; y < HEIGHT; y++){
                if ((y % 100) == 0){
                        printf("Estimation Row %ld calculated\n", y);
                }
                for (long x = 0; x < WIDTH; x++){
                        long xCo = ((long) WIDTH_MULTIPLY) * x
                                + (((long) WIDTH_MULTIPLY) / 2)
```

```
                                - OFFSET_X + DATA_OFFSET_X;
                        long yCo = ((long) HEIGHT_MULTIPLY) * y
                                + (((long) HEIGHT_MULTIPLY) / 2)
                                - OFFSET_Y + DATA_OFFSET_Y;
                        heatmap[y][x] = EstimationFunctionsCalculatePoint(xCo, yCo,
                                dataSet, dataSetSize, bandwidthHorizontal,
                                bandwidthVertical);
                }
        }
        printf("-------- estimation finished\n");
}

double EstimationFunctionsCalculatePoint(int xCo, int yCo,
        long dataSet[][2], long dataSetSize, int bandwidthHorizontal,
                int bandwidthVertical){
        double value = 0.0;

        for (int i = 0; i < (dataSetSize/(2*sizeof(long))); i++){
                value += EstimationFunctionsWeightFunction(xCo, yCo,
                        dataSet[i][0], dataSet[i][1], bandwidthHorizontal
                        * (WIDTH_MULTIPLY), bandwidthVertical * (HEIGHT_MULTIPLY))
                        * (1.0/((double)bandwidthHorizontal)
                        * (1.0/((double)bandwidthVertical)));
        }
        return value;
}

double EstimationFunctionsWeightFunction(int xCo, int yCo, int sampleX, int sampleY,
        int bandwidthHorizontal, int bandwidthVertical){
        double funcX = fabs(((double)(xCo - sampleX))/((double)bandwidthHorizontal));
        double funcY = fabs(((double)(yCo - sampleY))/((double)bandwidthVertical));
        double weight = 0.0;

//printf("funcX: %f\t funcY: %f\n", funcX, funcY);
//Point Density
//        if ((funcX < 1.0) && (funcY < 1.0)){
//                weight = 0.25;
//                //printf("-------- POINT IN RANGE\n");
//        }

//Kernel Density: Triangular
//        double weightX = 0.0;
//        double weightY = 0.0;
```

```
//        if ((funcX < 1.0)){
//                weightX = 1 - funcX;
//                //printf("-------- POINT IN RANGE\n");
//        }
//        if ((funcY < 1.0)){
//                weightY = 1 - funcY;
//                //printf("-------- POINT IN RANGE\n");
//        }
//        weight = weightX * weightY;

//Kernel Density: Epanechnikov
//        double weightX = 0.0;
//        double weightY = 0.0;
//        if (funcX < 1.0){
//                weightX = 0.75 * (1 - pow(funcX, 2));
//        }
//        if (funcY < 1.0){
//                weightY = 0.75 * (1 - pow(funcY, 2));
//        }
//        weight = weightX * weightY;

//Kernel Density: Triweight
//        double weightX = 0.0;
//        double weightY = 0.0;
//        if (funcX < 1.0){
//                weightX = (35.0 * pow((1 - pow(funcX, 2)), 3)) / 32.0;
//        }
//        if (funcY < 1.0){
//                weightY = (35.0 * pow((1 - pow(funcY, 2)), 3)) / 32.0;
//        }
//        weight = weightX * weightY;

//Kernel Density: Quartic
        double weightX = 0.0;
        double weightY = 0.0;
        if (funcX < 1.0){
                weightX = (15.0 * pow((1 - pow(funcX, 2)), 2)) / 16.0;
        }
        if (funcY < 1.0){
                weightY = (15.0 * pow((1 - pow(funcY, 2)), 2)) / 16.0;
        }
        weight = weightX * weightY;
```

```
//Kernel Density: Gaussian
//        double weightX = 0.0;
//        double weightY = 0.0;
//        weightX = (1.0 / sqrt(2.0 * PI)) * exp(-0.5*pow(funcX, 2));
//        weightY = (1.0 / sqrt(2.0 * PI)) * exp(-0.5*pow(funcY, 2));
//        weight = weightX * weightY;
        return weight;
}

void EstimationFunctionsCalculateMultipliers(long widthIn, long heightIn,
        long widthOut, long heightOut){
        printf("-------- calculation of multipliers started\n");
        long wMultiplier = 1;
        long hMultiplier = 1;
        long offsetX, offsetY;

        while((wMultiplier * widthOut) < widthIn){
                wMultiplier++;
        }

        while((hMultiplier * heightOut) < heightIn){
                hMultiplier++;
        }

        offsetX = ((wMultiplier * widthOut) - widthIn) / 2;
        offsetY = ((hMultiplier * heightOut) - heightIn) / 2;

        printf("wMultiplier: %ld\thMultiplier: %ld\toffsetX: %ld\toffsetY: %ld\n",
                wMultiplier, hMultiplier, offsetX, offsetY);
        printf("calculation of multipliers finished\n");
}
```

A.3 Implementatie in Python

A.3.1 Verwerking van de data voor de hardware

```
%matplotlib inline
import json
from pprint import pprint
import numpy as np
import matplotlib.pyplot as plt
```

```
import time
import sys
from PIL import Image
import math

def mapping(value, in1, in2, out1, out2):
        output = out1 + ((value - in1)/(in2-in1))*(out2-out1)
        return output

def int2bin(i):
        i = int(i)
        s = ''
        while i > 0:
                if (i % 2) == 1:
                        b = '1'
                else:
                        b = '0'
                s = b + s
                i = int(i / 2)
        return s

with open('knooppunten.json') as data_file:
        data = json.load(data_file)

values = []
for i in range (8,len(data['features'])):
        xCo = float(data['features'][i]['geometry']['coordinates'][0])
        yCo = float(data['features'][i]['geometry']['coordinates'][1])
        value = float(data['features'][i]['properties']['stats_mean'])
        values.append((xCo, yCo, value))

for x, y, v in values:
        if x > maxX:
                maxX = x
        if x < minX:
                minX = x
        if y > maxY:
                maxY = y
        if y < minY:
                minY = y
        if v > maxValue:
                maxValue = v
        if v < minValue:
```

```
                minValue = v

mappedValues = []
for tup in values:
        x = int(mapping(tup[0], 3.4, 4.1, 0, 639))
        y = int(mapping(tup[1], 50.7, 51, 0, 479))
        v = int(mapping(tup[2], 0, 3260, 0, 31))
        mappedValues.append((x, y, v))

binVal = []
for tup in mappedValues:
        x = int2bin(tup[0])
        while len(x) < 10:
                x = '0' + x
                y = int2bin(tup[1])
        while(len(y) < 9):
                y = '0' + y
                v = int2bin(tup[2])
        while len(v) < 5:
                v = '0' + v
                s = x + y + v
        binVal.append(s)

file = open('knooppunten_4.coe', 'w')
file.write('memory_initialization_radix=2;\n')
file.write('memory_initialization_vector=\n')
for i in range(len(binVal)):
        file.write(binVal[i])
        if i == (len(binVal) - 1):
                file.write(';')
        else:
                file.write(',')
        file.write('\n')
file.close()
```

A.3.2 Tijdsmeting van de software

```
%matplotlib inline
import json
from pprint import pprint
import numpy as np
import matplotlib.pyplot as plt
```

```
import time
import sys
from PIL import Image

def create_dataset(numberOfSamples):
        print('generate dataset')
        list = []
        for i in range(numberOfSamples):
        x = np.random.randint(0,640,1)[0]
        y = np.random.randint(0,480,1)[0]
        value = np.random.randint(0,32,1)[0]
        list.append((x,y,value))
        print('dataset generated')
        return list

def calculate_heatmap(dataset, bandwidth):
        print('calculate heatmap')
        heatmap = np.zeros((480, 640))
        for tup in dataset:
                x = tup[0]
                y = tup[1]
                v = tup[2]
                for i in range(y-bandwidth, y+bandwidth+1):
                        if (i > 0) and (i < 480):
                                for j in range (x-bandwidth, x+bandwidth+1):
                                        if (j > 0) and (j < 640):
                                                heatmap[i][j] += v
        print('calculation done')
        return heatmap

ds = create_dataset(249)
time1 = 0
numberOfTests = 10
for i in range(numberOfTests):
        start = time.time()
        calculate_heatmap(ds, 4)
        stop = time.time()
        time1 = time1 + stop - start
time1 = time1 / 10

ds = create_dataset(24900000)
time2 = 0
for i in range(numberOfTests):
```

```
        start = time.time()
        calculate_heatmap(ds, 4)
        stop = time.time()
        time2 = time2 + stop - start
time2 = time2 / 10
print(time1, time2)
```