Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen, de gevraagde informatie in te vullen (en de overeenkomst te ondertekenen en af te geven).

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling met

Titel: Real-time detectie en tracking voor sportanalyse met behulp van CUDA
Richting: master in de informatica - multimedia Jaar: 2008

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Ik ga akkoord,

VANLOFFELD, Rembert

Datum: 5.11.2008

Real-time detectie en tracking voor sportanalyse met behulp van CUDA

Rembert Vanloffeld

promotor :
Prof. dr. Philippe BEKAERT

Eindverhandeling voorgedragen tot het bekomen van de graad
master in de informatica multimedia

universiteit hasselt

Dankwoord

Een eerste dankwoord gaat uit naar mijn promotor Prof. dr. Philippe Bekaert voor het mogelijk maken van deze thesis. Ik zou ook mijn twee begeleiders, Maarten Dumont en Yannick Franken willen danken voor hun hulp bij deze thesis. Zonder hun voortdurende hulp en aanmoedigingen zou deze thesis er heel anders uitgezien hebben.

Ook wil ik mijn vrienden Anthony, Bram, Servaas en Jeroen bedanken omdat ik steeds bij hun terecht kon voor hulp en vragen over taalgebruik in mijn thesis.

Het laatste maar zeker niet het minste dankwoord is voor mijn moeder, zonder haar steun en geduld was deze thesis er waarschijnlijk nooit gekomen.

Rembert Vanloffeld
Hasselt 28 Mei 2008

Abstract

De vooruitgang van de computertechnologie maakt het mogelijk om steeds meer en ingewikkeldere berekeningen uit te voeren. In de sportwereld wordt deze trend gebruikt om steeds meer aan automatische analyse van sportsituaties te doen. Het doel van deze thesis is een aantal technieken bespreken om tot de detectie van balletjes te komen voor sportanalyse.

We beschrijven vervolgens een methode om via een temporele wavelet transformatie tot de detectie van een klein, snel bewegend balletje te komen. We zullen de recentelijk ontwikkelde programmeertaal CUDA gebruiken om deze applicatie deels op een recente grafische processor te implementeren. De grafische processor is een relatief goedkope maar zeer krachtige parallelle processor. Een wavelet transformatie is een rekenintensieve operatie waarbij veel data moet verwerkt worden en is dus uitermate geschikt om parallel geïmplementeerd te worden.

Inhoudsopgave

1 Inleiding **1**
- 1.1 Sportinformatica . . . 1
  - 1.1.1 Toepassingen . . . 2
- 1.2 Computervisie . . . 5
- 1.3 CUDA . . . 5
- 1.4 Doelstelling . . . 6

2 General purpose graphics processing unit **7**
- 2.1 Evolutie van een GPGPU . . . 7
- 2.2 Huidige generatie GPUs . . . 9
- 2.3 CUDA . . . 10
  - 2.3.1 Architectuur . . . 10
  - 2.3.2 Programmeermodel . . . 11
  - 2.3.3 Programmeren in CUDA . . . 12
  - 2.3.4 Geheugen optimalisatie . . . 14
  - 2.3.5 CUDA in de praktijk . . . 15
  - 2.3.6 Scan primitieven in CUDA . . . 17
  - 2.3.7 Conclusie . . . 25

3 Detectie **28**
- 3.1 Beeld differentiatie . . . 28
- 3.2 Achtergrond differentiatie . . . 28
  - 3.2.1 Gaussiaanse mix model . . . 29
- 3.3 Kleur differentiatie . . . 32
- 3.4 Wavelets . . . 34
  - 3.4.1 Eéndimensionale wavelet tranformatie . . . 35
  - 3.4.2 Tweedimensionale wavelet transformatie . . . 36
  - 3.4.3 Wavelets en detectie . . . 37
  - 3.4.4 Wavelet Analyse . . . 40
- 3.5 Binair masker analyse . . . 42
  - 3.5.1 Opening . . . 42
  - 3.5.2 Positie van bewegende delen . . . 44
  - 3.5.3 Object-specifieke analyse . . . 45

4 Tracking **47**
4.1 Discrete Kalman filter . . . 47
4.1.1 Algoritme . . . 49
4.1.2 Voorbeeld . . . 50
4.2 Uitgebreide Kalman filter . . . 52
4.2.1 Algoritme . . . 54
4.3 Particle filter . . . 54
4.3.1 Monte Carlo simulatie . . . 56

5 Implementatie **58**
5.1 Grijswaarden . . . 58
5.2 Wavelet transformatie . . . 58
5.3 Analyse . . . 60
5.4 Opening . . . 61
5.4.1 Erosie . . . 62
5.4.2 Dilatie . . . 62
5.5 Detectie . . . 62
5.5.1 Geconnecteerde componenten . . . 62
5.5.2 Traject detectie . . . 63
5.6 Tracking . . . 64
5.6.1 Lineaire tracking . . . 64
5.6.2 Kalman filter . . . 65
5.6.3 Virtuele posities . . . 65
5.6.4 Randvoorwaarden . . . 66
5.7 Afwerking . . . 66

6 Resultaten **67**
6.1 Opstelling . . . 67
6.2 Detectie . . . 67
6.3 Tracking . . . 70
6.4 Real-time . . . 71
6.4.1 Grijswaarde transformatie . . . 71
6.4.2 Wavelet transformatie en analyse . . . 71
6.4.3 Detectie en Tracking . . . 72
6.4.4 Mogelijke Optimalisaties . . . 73
6.5 CUDA . . . 73
6.5.1 Programmeren in CUDA . . . 74

7 Conclusies **75**

8 Toekomstig werk **76**

Bibliografie **77**

Hoofdstuk 1

Inleiding

Een recente ontwikkeling in de grafische hardware heeft het mogelijk gemaakt om de enorme parallelle rekenkracht van een grafische kaart beschikbaar te stellen voor algemene doeleinden. Met behulp van die ontwikkeling zal deze thesis een manier beschrijven om tot de real-time detectie van kleine, snel bewegende objecten te komen gebruik makende van videobeelden.

1.1 Sportinformatica

Net als in de meeste andere vakgebieden worden computers meer en meer een hulpmiddel voor sportwetenschappen [23]. De reden hiervoor is dat een groot deel van de nodige tools en algoritmes om aan dataverwerking in sport te doen reeds voor handen zijn in de computerwetenschappen [3], dit is meteen de reden dat we een snelle vooruitgang kunnen waarnemen in de sportinformatica. Computeranalyse wordt in de sport gebruikt om data samen te vatten op een manier waarop deze makkelijker geanalyseerd kan worden. Deze informatie kan vervolgens gebruikt worden om bijvoorbeeld te zien waar de prestatie van de speler nog kan verbeterd worden of kan door sportverslaggevers gebruikt worden om een wedstrijd nauwkeuriger te bespreken.

Het idee om computertoepassingen te maken die helpen bij de analyse van sport is niets nieuws, al in 1976 werd de term "sportinformatik", de Duitse term voor "sportinformatica" gebruikt door de Duitse sportgeleerde Herbert Haag. Het is echter pas recentelijk haalbaar geworden om aan computeranalyse van sport te doen. Dit omdat de hardware die voordien voorhanden was nog niet krachtig genoeg bleek te zijn om op een praktische manier gebruikt te kunnen worden [41].

1.1.1 Toepassingen

Computerwetenschappen kunnen in de sportwereld voor verschillende doeleinden worden gebruikt, hierop volgt een overzichtje hiervan.

Training

De meest voor de hand liggende toepassing van computerwetenschappen in de sport is het gebruik hiervan bij de training van de athleet. Analyse van het spel van een team of van het individu kan voor onschatbare waarde zijn bij het opstellen van een trainingsregime, zo kan een trainer bijvoorbeeld probleempunten identificeren en hier meer aandacht aan schenken tijdens de training.

Baca [2] beschrijft een model om een tafeltennismatch te analyseren aan de hand van verschillende attributen zoals het type van de slag, positie en impact van het balletje of een overzichtje van de fouten en de resultaten. Door deze data hebben speler en coach toegang tot een schat aan informatie die kan mogelijk kan gebruikt worden om tekortkominingen in de speler te detecteren.

Ook bij de training zelf kan de computer een hulp zijn. Een virtuele omgeving kan gebruikt worden als omgeving van de training. Deze aanpak wordt voorlopig vooral gebruikt bij rehabilitatie [44] maar is in theorie ook uitermate geschikt voor algemene trainingsdoeleinden. Het systeem laat immpers toe om bepaalde acties te herhalen die in het echte leven moeilijk te bekomen zijn, denk aan een doelwachter die wil trainen op een bal die exact in de kruising vliegt. Een situatie die niet altijd even herhaalbaar is door de gemiddelde trainer maar waarmee een virtueel systeem amper problemen heeft.

CAREN [10] is een dergelijk systeem. CAREN laat toe om bijvoorbeeld een loper een ideale tred aan te meten door de loper deze tred te fysiek te laten voelen. Doormiddel van metalen omhulsels waarin de athleet zijn voeten kan plaatsen wordt de loper aangeleerd wat de bewegingen zijn om tot zo een ideale tred te komen.

Op een andere manier kan een athleet onder laboratoriumomstandigheden handelingen uitvoeren waarna deze door de sportwetenschappers geanalyseerd kunnen worden op efficiëntie, performantie, etc... waarna een wetenschappelijk verantwoord beeld wordt gevormd van de sporter en zijn training. Een voorbeeld hiervan word gegeven door Arrellano [1], Arellano combineert een model van het gedrag van vloeistoffen en een video-analyse van wedstrijden om de performantie van zwemmers te bestuderen. Hij maakt gebruik van een visualisatie van de bewegingen die het water maakt rond de zwemmer en een berekening van wat de krachten zijn die tussen de zwemmer en het water aan het werk zijn.

Verslaggeving

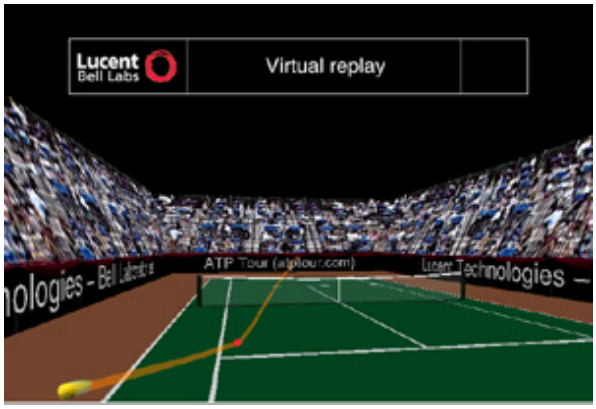
Ook de media maakt gebruik van de vorderingen in het vakgebied van de sportinformatica, hoe meer informatie de presentator en de kijker ter beschikking hebben, hoe makkelijker ze zich een beeld kunnen vormen van de wedstrijd.

Interessante informatie over een wedstrijd die uitgezonden wordt kan door een televisiezender weergegeven worden op het scherm, denk vooral aan tijdsinformatie bij een Formule 1 wedstrijd of een wielerwedstrijd. Het hoeft echter niet alleen met tijd te maken te hebben. Zo kan een tenniswedstrijd toegelicht worden door een weergave van de servicepercentages en tijdens een voetbalmatch is het balbezit of het aantal schoten op doel geen overbodige informatie. Dit blijkt niet alleen nuttig te zijn voor televisieuitzendingen, ook op het internet kunnen websites die live scores aanbieden van sportwedstrijden gebruik maken van dezelfde technologie.

Het feit dat er goede en betrouwbare systemen zijn om dergelijke informatie automatisch te vergaren maakt het voor de verslaggevers makkelijker om duidelijke en volledige verslaggeving te verstrekken aan het publiek.

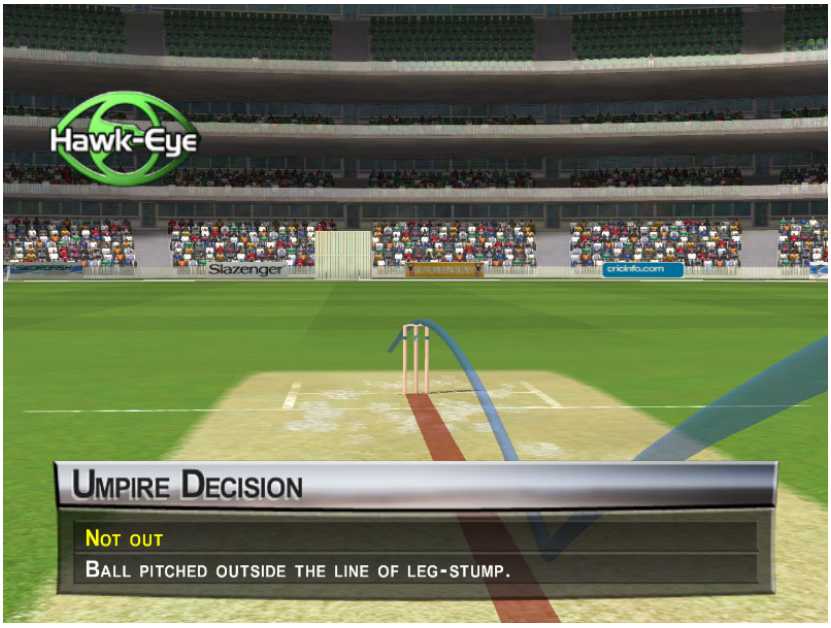
Virtuele herhalingen

Een ander voordeel van de vorderingen in de sportinformatie is de mogelijkheid om aan zogenaamde virtuele herhalingen te doen, een manier om wedstrijdsituatie vanuit verschillende standpunten in een virtuele setting te kunnen observeren, zo kan er bijvoorbeeld bij een voetbalwedstrijd beoordeeld worden of een speler al dan niet buitenspel stond of kan er bekeken worden of een bal al dan niet buiten was. Pingali et al. [42] beschrijft zo een virtueel replay systeem voor tennis, een afbeelding hieruit is gegeven in figuur 1.1.



Figuur 1.1: Een voorbeeld van een virtual replay systeem voor tennis [42].

Een bekender voorbeeld hiervan is het Hawk-Eye [26] systeem dat vooral gekend is van tenniswedstrijden. Hawkeye kan een virtueel beeld geven van een spel en werkt voor sporten als cricket, tennis, snooker, etc... In figuur 1.2 wordt een voorbeeld van de Hawkeye implementatie voor het analyseren van een cricket wedstrijd gegeven.



Figuur 1.2: Een voorbeeld van Hawkeye [26] toegepast op de analyse van een cricket wedstrijd.

Arbitrage

Ook als hulpmiddel voor scheidsrechters kan een sportinformatica systeem dienst doen, een scheidsrechter is vaak beperkt in zijn mogelijkheden om wedstrijdsituaties te beoordelen omdat ze bijvoorbeeld niet zichtbaar waren of omdat ze te snel gebeurden. Daarom kijken meer en meer sporten naar assistentie van de computerwetenschappen om arbitrage betrouwbaarder te maken.

Het Hawk-Eye[26] systeem dat in de vorige sectie al werd besproken is zo betrouwbaar dat het wordt gebruikt als aanvulling van de scheidrechters of "umpires" bij bepaalde tennistoernooien. Spelers kunnen een beslissing van de umpire aanvechten waarna Hawk-Eye kijkt of de beslissing al dan niet correct was.

1.2 Computervisie

We kiezen in deze thesis om gebruik te maken van een computervisie techniek en wel om de volgende redenen:

Niet-invasief

In theorie zou het mogelijk zijn om zowel spelers als attributen van sensoren te voorzien om zo hun positie en eigenschappen te detecteren, en terwijl dit voor bepaalde sporten mogelijk, zoals een GPS ontvanger die in een helm zit ingebouwd om een skiër te volgen [53], is dit voor andere sporten niet aangeraden. Een professionele sporter zal waarschijnlijk de sensoren als storend waarnemen waardoor het normale spelverloop in het gedrang komt, iets dat ten aller koste vermeden moet worden.

Daarom is computervisie een goede oplossing hiervoor, het is namelijk niet-invasief, het beïnvloedt de sport op geen enkele manier en is dus te verkiezen boven andere methodes die potentieel storend kunnen zijn voor de sporter.

Video

Videocameras zijn relatief goedkoop, zeker als je ze vergelijkt met tracking-apparatuur. Dikwijls is computervisie dus een goedkopere oplossing voor het probleem van sport analyse. Ook kunnen de bekomen videobeelden gebruikt worden voor andere doeleinden, zo kan men bijvoorbeeld de bekomen beelden nog gebruiken om later de wedstrijd opnieuw te bekijken. Omgekeerd kan computervisie ook potentieel toegepast worden op opnames van televisiecameras zodat ook wedstrijden die een lange tijd geleden gespeeld zijn mogelijk kunnen geanalyseerd worden.

1.3 CUDA

CUDA [38] is de nieuwe programmeertaal van nVidia die de kracht van een grafische kaart ter beschikking stelt van de programmeur voor algemene doeleinden. In eerdere generaties van grafische kaarten was deze kracht enkel toegankelijk voor grafische doeleinden.

Zoals we in het volgende hoofdstuk zullen bespreken wordt het gebruik van grafische hardware meer en meer geïntroduceerd in andere vakgebieden omdat de huidige generatie grafische hardware goedkope maar zeer krachtige multiprocessoren bevat. Deze multiprocessoren maken het mogelijk om applicaties die op een klassieke CPU typisch zeer traag werken ettelijke malen te versnellen door gebruik te maken van parallelle computatie.

1.4 Doelstelling

In deze thesis zullen we ons toespitsen op de detectie van kleine en snel bewegende objecten, dit komt in sport omgevingen meestal neer op het detecteren van balletjes. Kleine, snel bewegende balletjes zijn vaak moeilijk te detecteren wegens hun geringe grootte en het feit dat ze vaak onderhevig zijn aan een grote hoeveelheid motion blur. Om deze redenen is een het detecteren van deze objecten nog weinig beschreven in de literatuur.

We zullen aantonen dat CUDA een zeer geschikte oplossing is om het detecteren van een bal doormiddel van verschillende computervisie technieken te versnellen. De methode waarvoor we kiezen is een vorm van wavelet transformatie, een methode die door zijn grote aantal berekeningen typisch niet op een real-time manier kan berekend worden. We tonen eveneens aan dat de grote rekenkracht van de laatste generatie grafische hardware en de mogelijkheid om wavelets in parallel te berekenen ervoor zal zorgen dat we deze vorm van detectie in real-time kunnen berekenen.

We kiezen voor een wavelet transformatie methode om een tot een realistisch beeld over het nut van CUDA te kunnen komen. Een wavelet transformatie is een zeer reken intensieve methode die daarenboven een zeer hoge doorvoersnelheid vereist. Indien we een methode gekozen hadden die weinig rekenintensief is of die op een kleine hoeveelheid data werkt, zoals de methoden die typisch voor de CPU gebruikt worden, kan dit ons weinig leren over de mogelijkheden van programmeren voor de GPU.

Hoofdstuk 2

General purpose graphics processing unit

Bijna elke recente pc heeft wel een grafische processor aan boord, zonder deze hardware accelleratie zou het voor veel grafische applicaties onmogelijk zijn om aanvaardbaar vlot te kunnen werken. Dit hoofdstuk volgt de evolutie van deze grafische processoren van de primitieve versies in de jaren 1980 tot de zeer krachtige en zeer veelzijdige versies van tegenwoordig en bespreekt een nieuwe programmeertaal die toelaat om de kracht van de grafische processor voor algemene doeleinden te gebruiken.

2.1 Evolutie van een GPGPU

Sinds het prille begin van de computer graphics was het duidelijk dat de klassieke computerarchitectuur niet echt geschikt was om aan de eisen van de computer graphics te voldoen. Er was nood aan een chip, specifiek ontworpen voor grafisch renderen; een Graphics Processing Unit of GPU. In het midden van de jaren 1980 kwamen dan ook de eerste, zij het zeer rudimentaire, grafische chips op de markt. Chips als de MARIA [63], die in de vroege Atari systemen waren verwerkt, moesten basis 2D operaties versnellen zodat de programmeurs mooiere en vlottere beelden op het scherm konden tonen. Chips als deze werden gebruikt om de processor wat te ontlasten van grafische berekeningen om zo snellere uitvoering toe te laten.

Met de jaren 1990 kwam de opmars van de desktop pc's met hun bijbehorende 2D gebruiksinterfaces. Hierdoor kwam er een grote vraag naar 2D hardware accelleratie van de hoge resolutie bitmap beelden die gewoonlijk bij zulke interfaces horen. S3 Graphics [65] was het eerste bedrijf dat met zo een chip op de markt kwam. Het duurde echter niet lang eer elke grote chipfabrikant zijn eigen versie van een dergelijke chip op de markt had.

Met de opkomst van 3D rendering kon het niet anders dan dat de hardware accelleratie weeral

volgde. Enkele vroege voorbeelden hiervan waren de Playstation [58] en de Nintendo 64 [33], beide kwamen met een ingebouwde chip voor 3D hardware accelleratie. Niet veel later volgde de PC wereld met zijn eigen 3D hardware accelleratie, eerst in de vorm van hardware die bovenop de 2D-accelleratie werkte en vervolgens als alleenstaande hardware.

Het probleem met het ontwikkelen van software voor deze hardware was dat elke fabrikant wel zijn eigen API had. Het ontwikkelen van een applicatie voor een breed publiek bracht dus een hele hoop werk met zich mee, aangezien elke API grondig verschillend was. Om deze reden ontwikkelde Microsoft DirectX[30]. DirectX was een universele API die voor een abstractielaag zorgde bovenop al de API's van de verschillende fabrikanten. Dit gaf aan programmeurs de mogelijkheid om voor een hele waaier aan verschillende chipsets te programmeren zonder al te veel moeite. In 1998 kwam ook OpenGL [57] met zijn equivalente API op de markt waarmee programmeurs een waardig alternatief aangeboden kregen voor DirectX.

Met de jaren werden GPU's steeds krachtiger en met de introductie van pixel en vertex shaders werd de GPU voor het eerst programmeerbaar. Zowel OpenGL als DirectX kwamen met hun eigen scripting taal om rechtstreeks op de GPU te kunnen werken, respectievelijk GLSL [51][22] en HLSL [29].

De GPU evolueerde stilaan van een rudimentair versnellingsapparaat naar een parallel rekenwonder en bleef aan een razend snel tempo evolueren. Het werd duidelijk dat de GPU het als een parallelle processor nog niet zo slecht zou doen. Hierdoor werd het idee van een General Purpose GPU geboren. Het idee was om algemene applicaties toegang te verlenen tot de rekenkracht van de GPU via de programmeerbare delen hiervan. Deze aanpak was relatief succesvol maar het kampte nog met enkele ernstige problemen waardoor het nooit erg populair is geworden [38]:

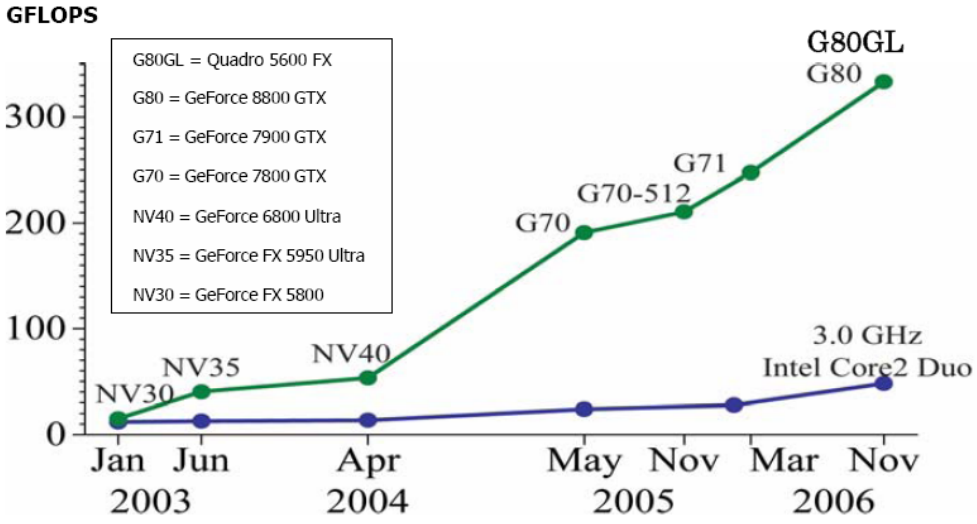
- De GPU was alleen beschikbaar via een grafische API. Dit zorgde voor een onnodig hoge leercurve en complexiteit voor een programmeur die hiermee niet vertrouwd was.
- De DRAM van de GPU was niet van die aard dat die kon beschreven worden met een flexibiliteit die programmeurs van de CPU gewoon waren.
- Sommige applicaties hadden niet genoeg aan de geheugenbandbreedte van de DRAM op de GPU, waardoor de rekenkracht GPU niet ten volste werd benut.
- De nood om via de grafische API te werken zorgde voor een ongewenste overhead.

Om deze problemen het hoofd te bieden kwam nVidia in november 2006 met zijn G80 chipset op de markt. Deze chipset ondersteunde CUDA, een programmeertaal die specifiek ontwikkeld was om op als een algemene programmeertaal op deze chip te werken. Voor het eerst waren programmeurs in staat om op een intuitieve manier te programmeren voor de GPU.

2.2 Huidige generatie GPUs

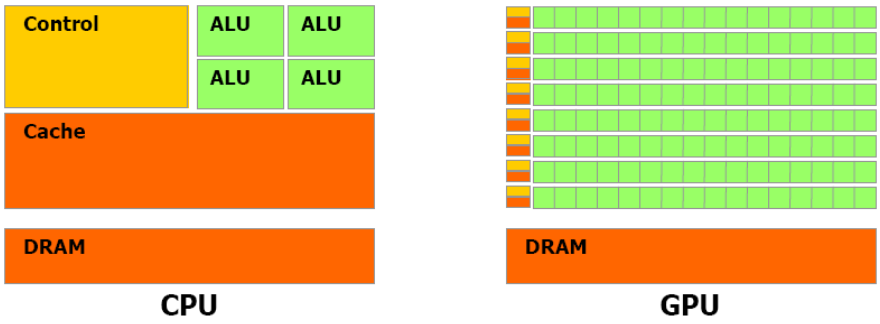
De strenge eisen van de grafische software en de sterke competitie op de markt van de grafische hardware zorgen ervoor dat de GPUs zowel betaalbaar als zeer performant worden. zie Fig. 2.1 waar de evolutie in kracht van de GPU vergeleken wordt met die van de CPU, let op het feit dat de snelheid van de CPU in de laatste 3 jaar nauwelijks veranderd is in vergelijking met de zeer sterke opmars van de GPU.

Merk op dat de wet van Moore hier nog steeds geldt, maar dat de performantiewinst niet meer op de traditionele wijze verkregen wordt. In plaats van steeds sneller worden CPUs en GPUs steeds breder. Dit wil zeggen dat we de snelheidswinst kunnen waarnemen op het gebied van parallellisme en minder op het gebied van hogere rekensnelheid[55]. Deze trend is waarneembaar bij zowel de GPUs als de CPUs, denk maar aan CPUs die reeds verkrijgbaar zijn met twee of zelfs vier processoren op een chip. Ook bij GPUs kunnen we deze evolutie waarnemen, zo heeft de G80 16 multiprocessoren [38] en 128 stream processoren [37] die in parallel werken.



Figuur 2.1: De evolutie van de rekenkracht van GPU's (bovenste lijn) en CPU's (onderste lijn). Weergegeven in GFLOPS = 1 miljoen floating point operaties per seconde [38].

De hoofdreden waarom de GPU op zo een manier evolueerde is natuurlijk een nood naar steeds snellere en betere grafische berekeningen. Dit resulteert in een GPU die geoptimaliseerd is voor zeer parallelle en zeer intensieve berekeningen met meer aandacht voor rekenkracht dan voor caching en flow control, wat natuurlijk exact datgene is dat grafische applicaties nodig hebben. Zie Fig. 2.2 voor een schematische voorstelling van de functies van zowel de CPU als de GPU. De CPU heeft meer aandacht voor caching en flow control terwijl de GPU meer de nadruk legt op pure rekenkracht door parallelisme.



Figuur 2.2: Een schematische voorstelling van de taken van een CPU ten opzichte van een GPU, let op de grote hoeveelheid caching en controle bij de CPU ten opzichte van het grote parallellisme en de grote nadruk op rekenkracht bij de GPU [38].

Een GPU is dus uitermate geschikt voor data-parallelle berekeningen met een nadruk op arithmetische complexiteit. Met andere woorden, de GPU blinkt uit bij operaties die kunnen uitgedrukt worden als de parallelle uitvoering van steeds hetzelfde programma op heel veel data elementen tegelijk en als er veel meer wiskundige berekeningen dan geheugenoperaties nodig zijn.

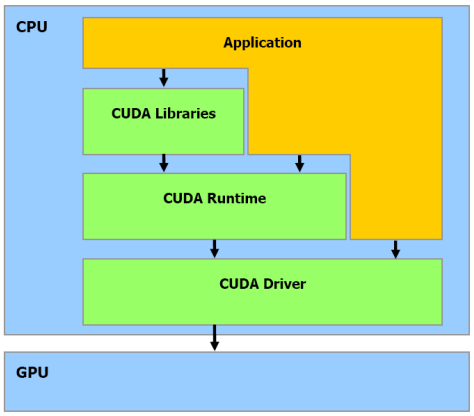
Het is duidelijk dat de GPU hoofdzakelijk een zeer performante grafische processor vormt, dit wil echter niet zeggen dat hij alleen geschikt is voor grafische applicaties. Elk probleem dat als een data-parallel probleem kan worden uitgedrukt is een kandidaat om op de GPU te werken.

2.3 CUDA

De term CUDA [34] staat voor Compute Unified Device Architecture en is een programmeertaal die ontwikkeld is door nVidia bovenop C. CUDA is specifiek ontwikkeld om algemene applicaties te kunnen programmeren voor de G80 GPU en zijn opvolgers, beter bekend als de nVidia GeForce 8 reeks.

2.3.1 Architectuur

De CUDA software bestaat uit verschillende lagen, zoals geïllustreerd in Fig. 2.3. CUDA wordt gevormd uit een CUDA API en de CUDA bibliotheken CUFFT en CUBLAS. Dit alles communiceert op een zeer performante manier met de hardware door gebruik te maken van een lichtgewicht CUDA driver.



Figuur 2.3: De Cuda architectuur schematisch voorgesteld [38].

De geheugenarchitectuur van CUDA laat toe om op elke plaats in de DRAM te schrijven, net als bij een CPU, wat zich vertaalt in een meer flexibele programmeerervaring ten opzichte van de eerdere GPGPU projecten. Ook is er de mogelijkheid om gebruik te maken van een gedeeld geheugen op de chip met zeer snelle lees- en schrijfsnelheden. Dit minimaliseert het probleem van de beperkte snelheden van het DRAM geheugen.

2.3.2 Programmeermodel

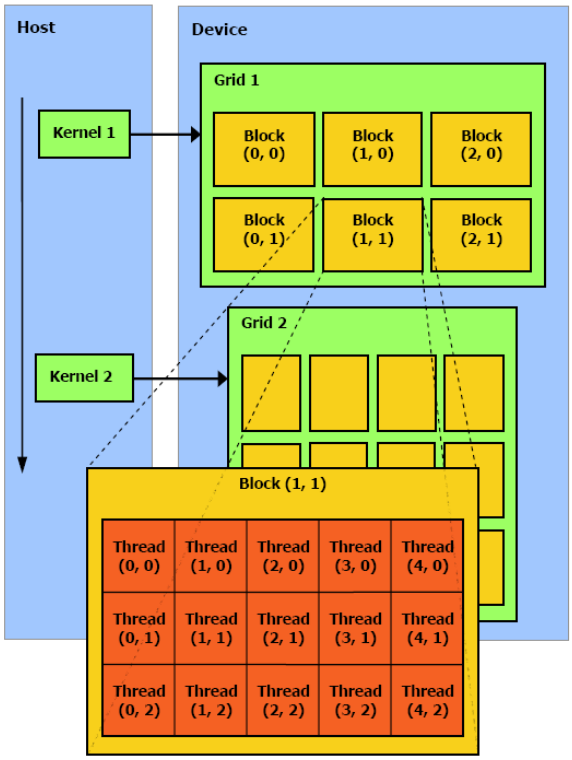
Het CUDA programmeermodel beziet de GPU als een processor die in staat is om een zeer grote hoeveelheid aan *threads* in parallel af te werken, zulke *threads* worden door de CUDA compiler vertaald naar de instructieset van de GPU, het hieruit bekomen programma noemen we een *kernel*. Zowel de CPU als de GPU blijven over hun eigen beschikken zodat programmas op de CPU niet aan geheugen op de GPU kunnen en omgekeerd kan een programma op de GPU onmogelijk rechtstreeks aan de data op de RAM van de CPU. Data kan echter op een snelle manier overgezet worden van het ene naar het andere geheugen [38].

Om het starten van *threads* te versnellen kunnen ze gebatched worden, *threads* met eenzelfde kernel kunnen verzameld worden in een *thread block* en elk van deze thread blocks kan verzameld worden in een *grid*. Met andere woorden, *threads* worden gegroepeerd in *grids* van *thread blocks*.

Een *thread block* is een verzameling van *threads* die efficient kunnen samenwerken door toegang tot het zeer snelle gedeelde geheugen. Parallelle executie op dezelfde data is echter niet wenselijk zonder een vorm van synchronisatie, daarom is het dan ook mogelijk om binnen een *thread block* voor synchronisatie te zorgen. Een *thread block* kan bestaan uit 2 of 3 dimensies en elk *thread* binnen dit block heeft zijn eigen uniek *thread ID* bestaande uit x, y en mogelijks een z waarde.

Omdat de grootte van een *thread block* beperkt is kan men gebruik maken van een *grid* om het uitvoeren van *thread blocks* te batchen, binnen een *grid* heeft elk *thread block* zijn eigen *block ID*. Het nadeel hiervan is dat *threads* in verschillende *thread blocks* geen toegang hebben tot het gedeeld geheugen waar de *threads* binnen dit *block* wel toegang tot hebben. *Grids* zijn tweedimensionaal, dus een *block ID* bestaat uit een x en een y waarde.

Het hierarchisch programmeermodel van CUDA wordt voor alle duidelijkheid in figuur 2.4 nog eens schematisch weergegeven.



Figuur 2.4: Het CUDA programmeermodel schematisch voorgesteld [38].

2.3.3 Programmeren in CUDA

De syntax van CUDA is zeer vergelijkbaar met die van C, dit is niet verbazingwekkend als je beseft dat CUDA een extensie is van de programmeertaal C. Het grootste verschil met pure C is het parallelle executiemodel van CUDA, dat voor programmeurs die hier niet mee vertrouwd zijn in eerste instantie wat onwennig zal zijn.

De CUDA compiler nvcc is geen volwaardige compiler en heeft de compiler vanop het systeem

nodig om correct te functioneren. De nvcc compiler compileert de data die nodig is om de kernels voor de GPU te kunnen starten en laat de rest over aan een systeem compiler zoals gcc [38].

Voor functies in CUDA is het nodig om te specificeren voor welke processor een methode bestemd is, ofwel voor de GPU ofwel voor de CPU. Daarvoor zijn de keywords `__host__` , `__global__` en `__device__` gedefinieerd die voor de functienaam komen. Het standaard keyword `__host__` betekent dat de methode op de CPU zal werken. Een volgende type methode is `__global__`, dit definieert de methode als een *kernel* en deze wordt op de GPU uitgevoerd. Een `__global__` methode kan enkel vanuit een `__host__` methode opgeroepen worden op een zeer specifieke manier, meer hierover later. Het laatste type methode is `__device__`, deze wordt uitgevoerd op de GPU en kan enkel van daaruit aangeroepen worden. Voorbeeldjes van deze keywords vind je in listing 2.1.

```
__host__ void hostFunc();        // werkt op de CPU
void hostFunc2();                // werkt op de CPU
__global__ void globalFunc();    // werkt op de GPU
__device__ void deviceFunc();    // werkt op de GPU
```

Listing 2.1: Voorbeelden van de keywords voor het definieren van het type methode.

CUDA definieert bovendien een `dim3` vector type dat gebaseerd is op 3 `unsigned int` waarden, toegankelijk als een structure waarbij de member variabelen toegankelijk zijn als x, y en z. het `dim3` type wordt gebruikt bij initialisatie en controle over individuele *threads* en is beschikbaar als een set van ingebouwde variabelen binnen een *thread* die het *thread id* en het *block id* bevatten evenals de dimensies van de *grids* en de *blocks*.

Een andere uitbreiding van de standaard C taal is de manier waarop een kernel gestart moet worden, het aantal *threads* en *blocks* moet gedefinieerd worden alsook de hoeveelheid gedeeld geheugen dat nodig is per *block*, in listing 2.2 zie je een voorbeeld van de CUDA syntax om een kernel te lanceren. Dit gebeurt door de lijn `kernel<<<dimGrid, dimBlock, memory_size>>>();` waarbij `kernel` de naam van de starten kernel is en `<<<dimGrid, dimBlock, memory_size>>>` het aantal *threads* en de hoeveelheid gedeeld geheugen bepaalt dat voor deze kernel zal worden gebruikt.

```
__global__ void kernel();

// werkt op de CPU
void launchKernel()
{
  ...//initialiseer CUDA
  ...//kopieer de nodige data naar het device geheugen.
```

```
 // bepaal het aantal threads en het aantal blocks
 // beiden hebben hier een dimensie van 10 x 10
 dim3 dimBlock(10,10), dimGrid(10,10);

 // bepaal de grootte
 size_t memory_size = 20 * sizeof(float);

 // lanceer 100 blocks met telkens 100 threads
 // en geef elk block 20 floats aan gedeeld geheugen
 kernel<<<dimGrid, dimBlock, memory_size>>>();

 ... // kopieer de berekende data terug naar het host geheugen

}

// werkt op de GPU
__global__ void kernel()
{
 // bepaal het threadId en het blockId
 // met andere woorden bepaal de virtuele
 // positie van deze thread op de GPU
 unsigned int tidX = threadId.x;
 unsigned int tidY = threadId.y;
 unsigned int bidX = blockId.x;
 unsigned int bidY = blockId.y;

 ... // doe de nodige berekeningen
}
```

Listing 2.2: Een simpel voorbeeldje van het typische gebruik van CUDA.

2.3.4 Geheugen optimalisatie

Een CUDA programma optimaliseren is een hele taak, er is dan ook zeer veel dat kan geoptimaliseerd worden, we zullen ons hier beperken tot optimalisatie van het globaal geheugengebruik doormiddel van *coalesced* lezen en schrijven en het optimaal gebruik van *gedeeld geheugen*. Gedeeld geheugen wordt best zo vaak mogelijk gebruikt voor communicatie tussen *threads* binnen hetzelfde *block* aangezien dit geheugen veel sneller is dan het globale geheugen. De snelheid van het gedeeld geheugen wordt immers beperkt door *bank conflicten*. Het vermijden van *bank conflicten* is een belangrijke taak in het versnellen van het gebruik van het gedeeld geheugen. De geïnteresseerde lezer wordt aangeraden om Harris [14] na te lezen voor een zeer goed en uitgebreid overzicht van CUDA optimalisatie in het algemeen.

Globaal geheugen op de GPU is niet gecached, het is dus extra belangrijk om ervoor te zorgen dat de bandbreedte van het geheugen optimaal gebruikt wordt om opstroppingen te vermijden, zeker gezien de relatieve traagheid van globaal geheugen ten opzichte van de registers en

het gedeeld geheugen.

De GPU voorziet speciale operaties om 32,64 en 128bit woorden in 1 instructie uit het geheugen te halen en in de registers van de multiprocessor te stoppen, met andere woorden, elke variabele waarvoor geldt dat sizeof(variabele) = 4,8 of 16 kan in 1 instructie geladen worden. Dit houdt bijvoorbeeld in dat een char, een 8 bit woord, trager ingeladen zal worden dan een float, hetgeen een 32bit woord is. Minder data is dus niet per definitie sneller geladen.

Een andere voorzorg die genomen zou moeten worden is een techniek die *coalesced* lezen en schrijven genoemd wordt. Het idee is dat *threads* binnen een halve warp (16 threads) een aaneensluitende regio in het globaal geheugen zouden moeten adresseren. Dit zorgt ervoor dat deze data in ëën brok naar de multiprocessor kan gestuurd worden door een hardware ondersteunde snelle kopieeroperatie en dit minimaliseert dus de tijd dat er moet gewacht worden op het geheugen. Een *warp* is een verzameling van 32 *threads* die tegelijk op een multiprocessor aan het werk zijn.

Nog een optimalisatie is het gebruik van het gedeeld geheugen. Gedeeld geheugen is bijna zo snel als de registers indien er zich geen *bank conflicten* voordoen. Gedeeld geheugen is opgedeeld in 16 *banks*, deze *banks* zijn telkens 32 bit groot. Als elke *thread* binnen een halve warp een aparte *bank* binnen het gedeeld geheugen aanspreekt kunnen deze in parallel afgehandeld worden, hetgeen voor n *threads* resulteert in n maal de geheugenbandbreedte. Indien er echter binnen ëën *bank* meer dan ëën request tegelijk gebeurt moeten deze na elkaar afgehandeld worden, hetgeen voor een onwenselijke vertraging zorgt.

Een uitzondering hierop is als al de *threads* van een *halve warp* dezelfde *bank* aanspreken, voor dat geval is er een broadcast mechanisme dat de data naar elke *thread* kan broadcasten.

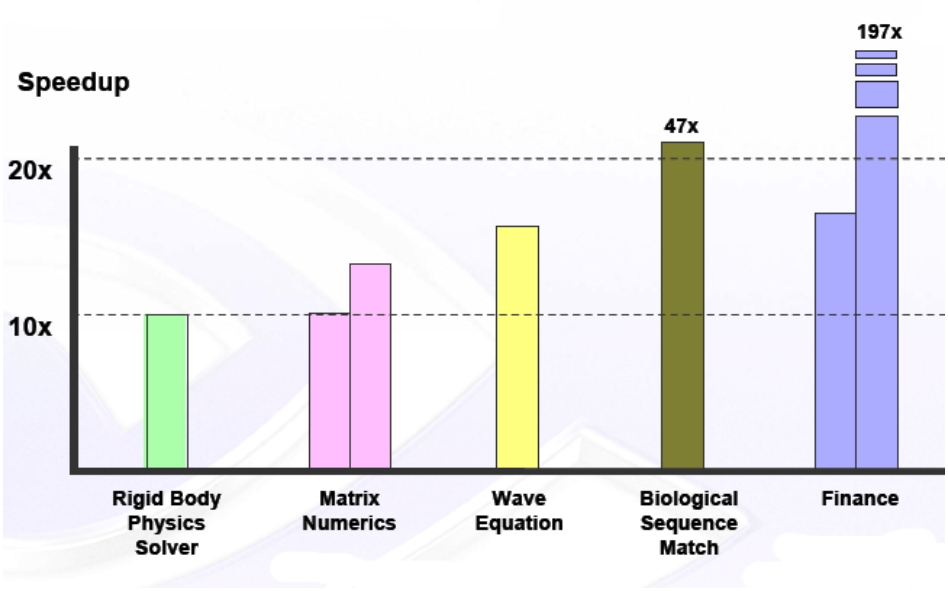
Adressering binnen het gedeeld geheugen werkt als een circulaire array. Daarom worden *bank conflicten* typisch vermeden door lineaire toegang tot de *banks* met een opvulling van s*32bit, waarbij s een oneven getal is, dit zal in een latere sectie in meer detail worden uitgelegd.

2.3.5 CUDA in de praktijk

Buiten de overduidelijke grafische toepassingen van CUDA wordt CUDA in een wijd gamma van vakgebieden gebruikt. Er zijn er te veel om hier op te sommen, de geïnteresseerde lezer kan steeds de CUDA showcase [36] eens nakijken voor een uitgebreider en een up-to-date overzicht van projecten die CUDA gebruiken om hun applicatie te versnellen. Figuur 2.5 toont een overzichtje van verschillende vakgebieden die CUDA al gebruikt hebben voor hun applicatie en hoe veel versnelling ze hadden ten opzichte van het origineel. Merk op dat de

financiële applicatie bijna 200 maal sneller was dan origineel na het gebruik van CUDA.

Hieronder worden nog een paar voorbeelden besproken van projecten die CUDA gebruiken en hun resultaten.



Figuur 2.5: CUDA performantie en mogelijke versnelling van de applicaties, Let vooral op de financiële applicatie die een versnelling tot 197 maal liet optekenen. Afbeelding eigendom van nVidia.

Moleculaire dynamiek

Moleculaire Dynamiek of MD gebruikt computer simulatie om het gedrag van moleculen en atomen te voorspellen. Het wordt gebruikt voor toepassingen die varieren van statistische mechanica over materiaalkunde tot virologie en is dus van onschatbare waarde voor wetenschappelijk onderzoek in allerlei vakgebieden.

van Meel et al. [64] beschrijft een manier om MD-interacties op lange afstand tot 80 maal te versnellen en interacties op korte afstand tot 40 maal. Ook beschrijven ze een MD-specifieke random nummer generator die tot 150 maal sneller werkt dan zijn klassieke voorganger.

Medische en Biologische Toepassingen

Ook de medische wereld is gebaat met CUDA, Jeong et al. [19] beschrijft een manier om connectiviteit van witte hersencellen te modelleren gebruik makende van de GPU en dit model vervolgens te visualiseren. Het gebruikte Hamilton-Jacobi algoritme is 50 tot 100 maal sneller op de GPU dan op de CPU.

CMATCH [52] is een bio-informatica applicatie dat de kracht van de GPU gebruikt om aan string matching te doen op een manier die tot 35 maal sneller is dan zijn CPU tegenhanger.

Meteorologie

Ook modellen om het weer te voorspellen kunnen versneld worden door het gebruik van een GPU, Michalakes en Vachharajani [28] beschrijven een snelheidswinst van 20 maal de originele snelheid van een rekenintensief deel van het Weather Research and Forecast model, wat resulteerde in een algemene snelheidswinst van ongeveer 130%.

2.3.6 Scan primitieven in CUDA

Een implementatie van een scan algoritme, als beschreven door Harris [15] en Sengupta et al. [54] is een goed voorbeeld en een perfecte introductie voor het gebruik van CUDA in de praktijk. Er wordt aandacht besteed aan het vermijden van *bank conflicten* doormiddel van adressering met extra opvulling.

Een *scan* operatie is de operatie waar een vector v als invoer genomen wordt, v wordt bepaald door:

$$v = [a_0, a_1, .., a_{n-1}] \tag{2.1}$$

De uitvoer van de exclusieve scan operatie is een vector v' waarbij elk element in v vervangen is door een binaire operatie $\oplus$ op al de voorgaande elementen:

$$v' = [0, a_0, (a_0 \oplus a_1).., (a_0 \oplus a_1 \oplus .... \oplus a_{n-2})] \tag{2.2}$$

Een inclusieve scan operatie wordt vervolgens gedefinieerd als v''

$$v'' = [a_0, (a_0 \oplus a_1), .., (a_0 \oplus a_1 \oplus .... \oplus a_{n-1})] \tag{2.3}$$

Dus een scan operatie op de vector met als binaire operator de optelling $(+)$

$$u = [1, 2, 3, 4, 5, 6] \tag{2.4}$$

geeft als uitvoer

$$u' = [1, 3, 6, 10, 15, 21] \tag{2.5}$$

In de rest van het voorbeeld zullen we de binaire operator $\oplus$ vervangen door de optelling $(+)$ maar het is evengoed mogelijk met andere binaire operatoren zoals bijvoorbeeld max en min.

Een sequentiële implementatie van een scan algoritme voor een enkele thread op de CPU is een triviale taak, implementatie hiervan in C code wordt gegeven in listing 2.3.

```
void scan ( float * vector, int length )
{
 for( int i = 1; i < length; i++)
  vector[i] += vector[i-1];
}
```

Listing 2.3: Een implementatie van een inclusief scan algoritme in C

Deze code heeft exact n-1 optellingen nodig om de scan operatie te berekenen. Een doelstelling voor onze parallelle implementatie wordt daarom dat de parallelle versie werk-efficiënt is, met andere woorden er mogen niet meer operaties gebeuren dan in de sequentiele versie en we mikken dus op een werk-complexititeit van $O(n)$.

Eerste poging

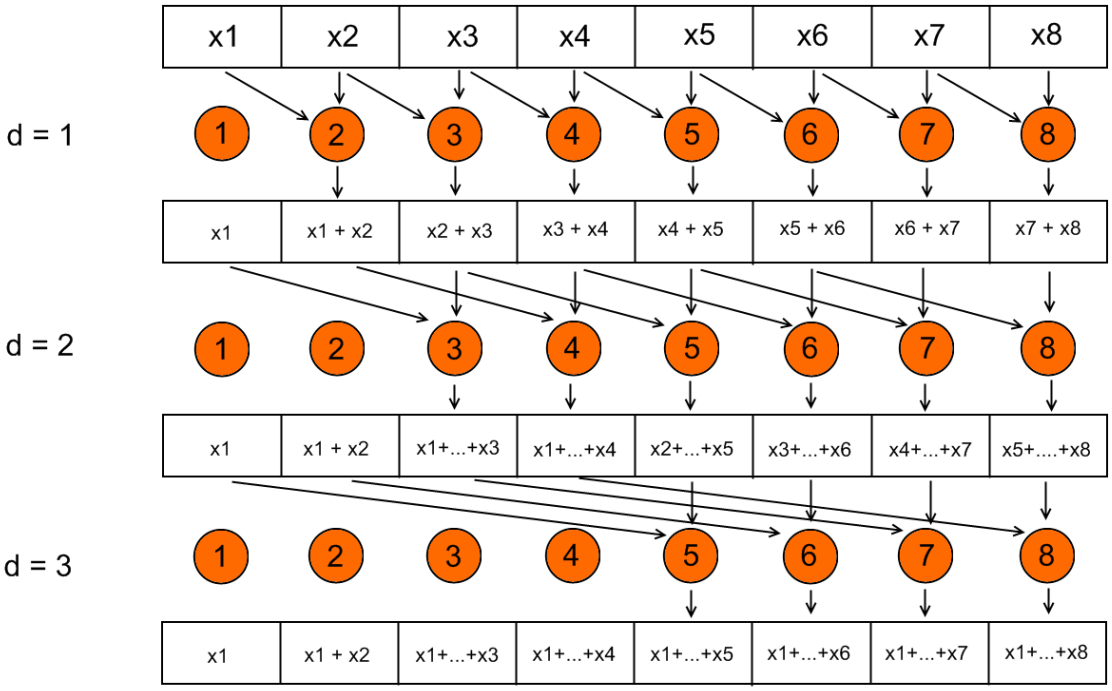
Een eerste naïve poging om tot een parallel algoritme te komen voor een inclusieve scan is hier gegeven in pseudocode.

1. Voor alle k in parallel:
  2. Voor alle d $= 1 \rightarrow \log_2 n$:
    3. Als $k > 2^{d-1}$ dan $x[k] = x[k - 2^{d-1}] + x[k]$

Wat dit algoritme doet is voor elke thread apart recursief telkens 2 waarden op te tellen. Na elke iteratie van d worden de eerste 2^{d-1} threads op non-actief gezet omdat hun werk gedaan is. Bekijk figuur 2.6 voor een voorbeeld, elke cirkel stelt een thread voor en de vakjes stellen de array voor. Elke thread krijgt 2 waarden binnen, voorgesteld door inkomende pijlen. De thread telt deze twee waarden op een zet ze op de positie waar de uitkomende pijl eindigt. Merk op dat dit algoritme, omdat het parallel loopt een tijdscomplexiteit van $O(\log n)$ heeft. Omdat tijdens en puur parallelle uitvoering de tijdscomplexiteit van de langst uitvoerende thread overweegt. Hetgeen al beter is dan de sequentiële versie waarvan de tijdscomplexiteit $O(n)$ bedraagt.

Dit algoritme is echter niet werk-efficiënt, het vereist $\sum_{d=1}^{\log_2 n} n - 2^{d-1} = O(n \log_2 n)$ optellingen nodig om tot een resultaat te komen, dit terwijl de sequentiële operatie een resultaat van $O(n)$ had. Een optimalisatie is hier dus nog nodig.

Een ander probleem met dit algoritme is dat het aanneemt dat er zoveel parallelle processoren zijn als er data elementen zijn . In CUDA is dit echter niet zo, het aantal parallelle processoren is beperkt. Dit wil zeggen dat op een CUDA GPU deze berekeningen zouden opgesplitst worden in kleinere parallelle batch executies die dan sequentiëel naar de multiprocessor gestuurd worden. Omdat dit algoritme hier geen rekening mee houdt zouden resultaten van 1



Figuur 2.6: Een grafische voorstelling van de naïeve scan methode, de cirkels stellen aparte threads voor, en de vector wordt weergegeven als opeenvolgende vakjes.

batch de anderen overschrijven.

Tweede poging

Om het vorige algoritme te kunnen gebruiken op een CUDA GPU zijn er enkele aanpassingen nodig, het nieuwe algoritme heeft een dubbele tijdelijke array ($x[2][n]$) nodig die het dan gebruikt als een dubbele buffer om overschrijving te vermijden.

1. Voor alle k in parallel:
  2. Voor alle d $= 1 \rightarrow \log_2 n$:
    3. Als $k > 2^{d-1}$ dan $x[uit][k] = x[in][k - 2^{d-1}] + x[in][k]$
    4. Anders $x[uit][k] = x[in][k]$
  5. Verwissel(in, uit)

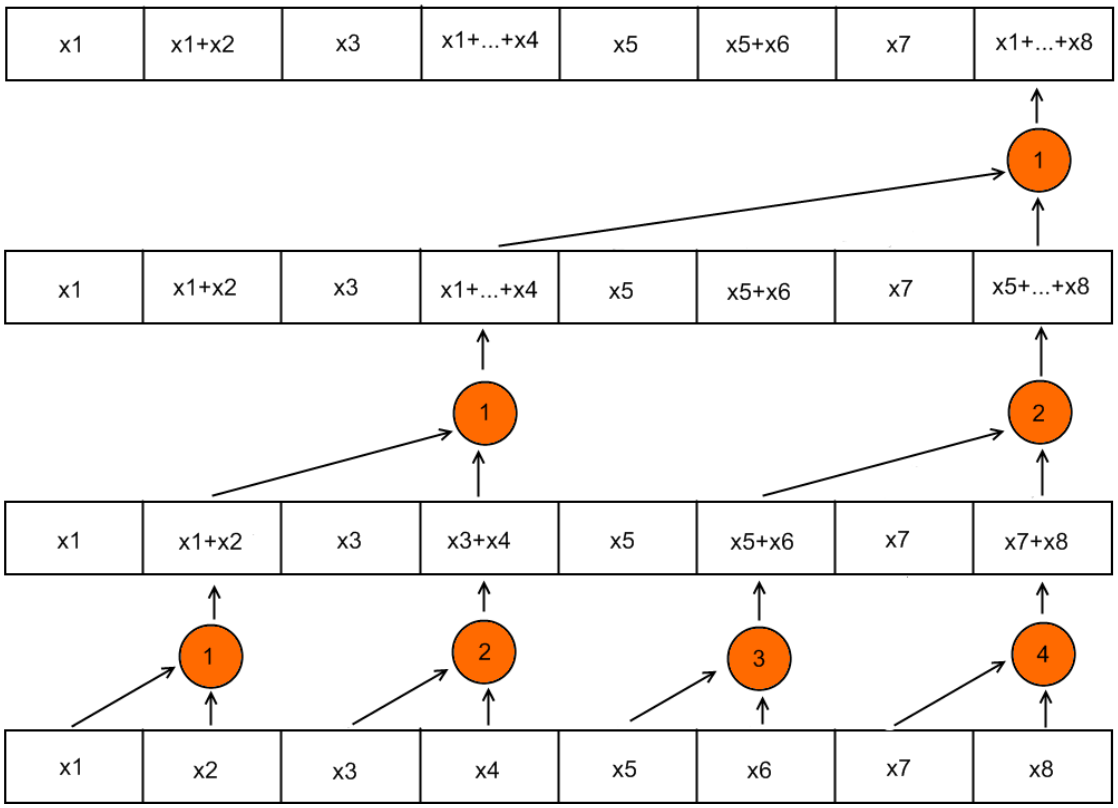
Aan het basis idee en dus de prestaties van de eerste poging is er niets veranderd, het is alleen aangepast aan de eisen van een CUDA implementatie.

Een derde, finale versie wordt nu geïntroduceerd om de werkcomplexiteit op het niveau van die van de sequentiële implementatie te krijgen, namelijk $O(n)$

Een finale versie

Het doel voor deze finale versie is dus een werk-efficiënt scan algoritme te beschrijven. Ter illustratie berekenen we hier de exclusieve scan, een conversie naar een inclusieve scan is immers triviaal en een oplossing voor een inclusieve scan is net iets minder ingewikkeld dan zijn exclusieve tegenhanger.

We bekijken de originele vector als een gebalanceerde binaire boom. Het algoritme bestaat uit 2 delen, een *up-sweep* fase en een *down-sweep* fase. De up-sweep fase gaat van de bladeren van de boom naar de wortel toe terwijl de down-sweep fase terug van de wortel naar de bladeren gaat.



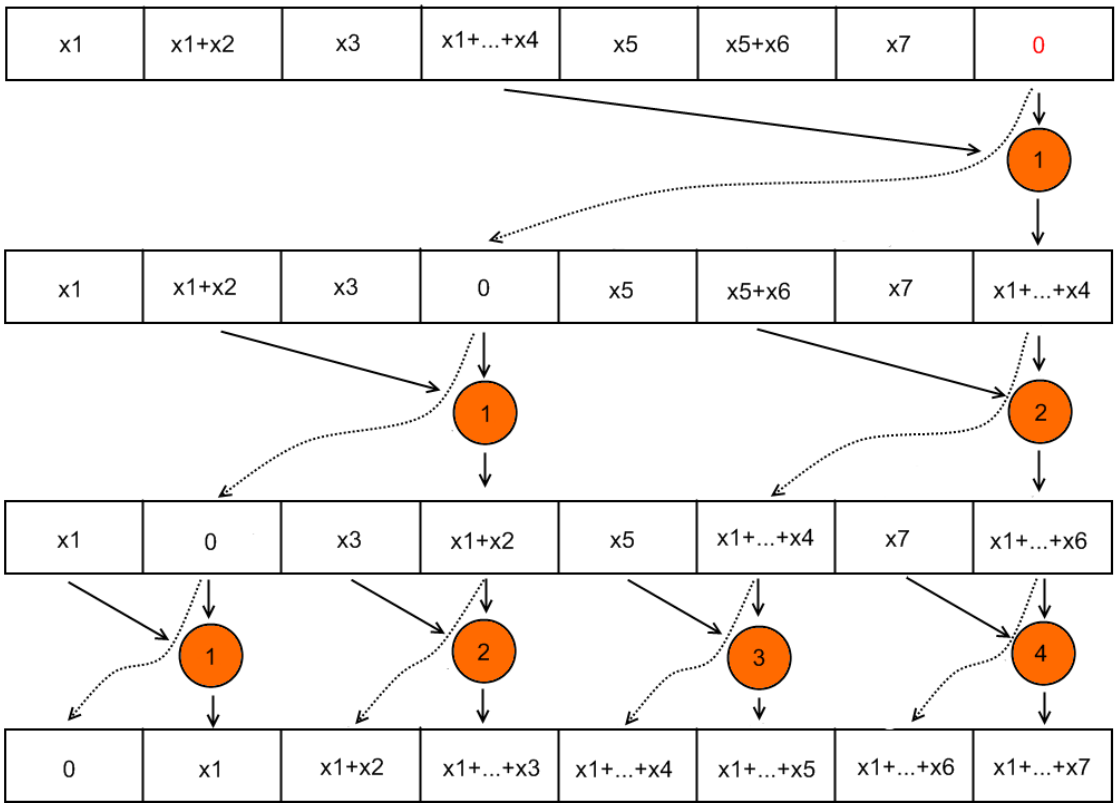
Figuur 2.7: Een schematische weergave van de up-sweep fase van het werk-efficiënte scan algoritme.

Het algoritme voor de up-sweep wordt

1. Voor alle k =1 $\rightarrow n/2$ in parallel:
  2. Voor alle d = 1 $\rightarrow \log_2 n$:
    3. Als $k \leq log_2 n - d + 1$ dan $x[k + 2^d - 1] = x[k + 2^{d-1} - 1] + x[k + 2^d - 1]$

Er wordt sequentieel een tussensom opgebouwd van de vector en uiteindelijk houdt het laatste element van de vector alle waarden, voor een grafische voorstelling van de up-sweep kijk naar

figuur 2.7.



Figuur 2.8: Een schematische weergave van de down-sweep fase van het werk-efficiënte scan algoritme.

Omdat we met de exclusieve scan bezig zijn zetten we nu de laatste waarde van de vector op 0, deze 0 waarde zal zich in de down-sweep fase naar het begin van de vector bewegen. De pseudocode voor de down-sweep fase wordt dan:

1. $x[n] = 0$
2. Voor alle k =1 $\rightarrow n/1$ in parallel:
  3. Voor alle d = $\log_2 n \rightarrow 1$:
    4. Als $k \leq log_2 n - d + 1$ dan
      5. $t = x[2^d k]$
      6. $x[2^d k] = t + x[2^{d-1} k]$
      7. $x[2^{d-1} k] = t$

De down-sweep fase wordt schematisch voorgesteld in figuur 2.8 en de CUDA code van het hele algoritme wordt gegeven in listing 2.4

```
__global__ void prescan(float * g_odata, float * g_idata, int n)
{
  extern __shared__ float temp[];

  int tid = threadIdx.x;
  int offset = 1;

  // laad invoer in het snelle gedeeld geheugen
  // ------------ A ------------------------
  temp[2*tid] = g_idata[2*tid];
  temp[2*tid+1] = g_idata[2*tid+1];
  //---------------------------------------

  // sweep up
  for(int d = n >> 1; d > 0; d >>= 1)
  {
    __suncthreads();

    if(tid < d)
    {
      // ------------ B ------------------------
      int ai = offset*(2*tid+1)-1;
      int bi = offset*(2*tid+2)-1;
      //---------------------------------------
      temp[bi] += temp[ai];
    }

        offset *= 2;
  }
  // zet het laatste element op 0
  //------------------ C --------------------
  if(tid == 0) {temp[n-1] = 0;}
  //-----------------------------------------

  // sweep down
  for ( int d = 1; d < n; d*=2)
  {
    offset >>=1;
    __syncthreads();

    if(tid < d)
    {
      //----------------- D -----------------
      int ai = offset*(2*tid+1)-1;
      int bi = offset*(2*tid+2)-1;
      //-------------------------------------

      float t = temp[ai];
      temp[ai] = temp[bi];
      temp[bi] += t;
    }
  }
  __syncthreads();
```

```
    // schrijf de data weg
    //------------------------ E ------------------------
    g_odata[2*tid] = temp[2*tid];
    g_odata[2*tid+1] = temp[2*tid+1];
    //---------------------------------------------------
}
```

Listing 2.4: Een efficiënt scan algoritme in CUDA, de delen gelabeld door A, B, C , D en E worden later nog besproken.[15]

Dit algoritme mag dan wel in theorie optimaal zijn, als je programmeert voor een CUDA GPU kan er nog veel geoptimaliseerd worden aan deze kernel met betrekking tot geheugentoegang.

In de sectie over optimalisatie hebben we het kort gehad over het probleem van *bank conflicten*. Deze code is zeer vatbaar voor *bank conflicten*, met tot zestienvoudige *bank conflicten*, hetgeen een significant effect op de performantie heeft. Een zestienvoudig *bank conflict* zorgt er immers voor dat een kernel tot 16 maal langer dan verwacht op de data uit het gedeeld geheugen moet wachten, een zeer ongewenste situatie.

We lossen dit hier op door volgende macro's te definiëren [15]:

```
#define NUM_BANKS 16
#define CONFLICT_FREE_OFFSET(n)    ((n) >> NUM_BANKS )
```

Listing 2.5: Definitie van macro's om bank conflicten te vermijden

Hier heeft `n >> NUM_BANKS` dezelfde betekenis als `n / NUM_BANKS` maar aangezien `NUM_BANKS` een macht van 2 is wordt het efficiënter om een bitshift te gebruiken.

We gebruiken in dit voorbeeld de waarden voor `ai` om de problemen te illustreren maar de berekeningen zijn equivalent voor `bi` en de conclusies zijn dezelfde, we bekijken dus voor de rest van deze sectie enkel de *bank conflicten* veroorzaakt door `ai`.

Zoals al vermeld is het gedeeld geheugen opgebouwd als een circulaire array, dit wil zeggen dat als we een adres over de grens van de array aanspreken dat CUDA terug begint te tellen aan het begin van deze array. In dit voorbeeld gebruiken we een array van `32 int` waarden waarbij verschillende *threads* een *bank conflict* veroorzaken indien ze `temp[i]` en `temp[i+16]` aanspreken. Een `int` type is immers een 16 bit woord en een *bank* bestaat uit een 32 bit woord, hetgeen wil zeggen dat er twee int waarden per *bank* worden opgeslaan.

Om een voorbeeld te geven van waar dit *bank conflicten* zal opleveren, kijk naar de `int ai` waarde gemarkeerd als blokken B en D in listing 2.4. `ai` wordt bepaald door `offset*(2*tid+1)-1`.

We rekenen dit uit voor `tid = 0` en `offset = 1` en we komen bij `ai = 0`. Als we nu hetzelfde doen voor `tid = 8` komen uit bij `ai = 16` en zoals we net gezien hebben is dit een *bank conflict*. *Bank conflicten* zijn echter niet beperkt tot alleen deze twee *threads* maar elke *thread* i zal een *bank conflict* genereren met *thread* $i + 8$.
Bij een hogere offset gaan de *bank conflicten* alleen maar erger worden, bij een offset van 2 worden het viervoudige *bank conflicten* en bij een offset 8 zijn we zelfs aan het worst-case scenario, een zestienvoudig *bank conflict*. In figuur 2.9 wordt het probleem schematisch weergegeven.
Elke *thread* in het scan algoritme leest 2 waarden in van de array g_idata naar het gedeeld geheugen. Dit is meteen een eerste plaats waar tweevoudige *bank conflicten* zullen optreden. Dit is echter makkelijk op te lossen, namelijk door elke *thread* 2 elementen aan andere kanten van de array te laten inlezen wat onmiddellijk al *bank conflicten* vermijdt.

```
int ai = tid;
int bi = tid + (n/2);
int bankOffsetA = CONFLICT_FREE_OFFSET(ai);
int bankOffsetB = CONFLICT_FREE_OFFSET(bi);
temp[ai + bankOffsetA] = g_idata[ai];
temp[bi + bankOffsetB] = g_idata[bi];
```

Listing 2.6: Blok A [15]

Iets moeilijker om op te lossen zijn de bank conflicten tijdens het doorlopen van de boom, we moeten hier om de 16 elementen wat plaats toevoegen om bank conflicten te vermijden.

We herschrijven nu blokken B en D om gebruik te maken van de nieuwe adressering, we voegen simpelweg de nieuw berekende offset toe aan de vooraf berekende positie.

```
int ai = offset*(2*tid+1)-1;
int bi = offset*(2*tid+2)-1;
ai += CONFLICT_FREE_OFFSET(ai);
bi += CONFLICT_FREE_OFFSET(bi);
```

Listing 2.7: Blokken B en D [15]

Blok C en D moeten vervolgens worden aangepast om gebruik te maken van het nieuwe soort adressering anders lezen ze de verkeerde waarden in.

```
if(tid==0) {temp[n-1 + CONFLICT_FREE_OFFSET(n-1)] = 0; }
```

Listing 2.8: Blok C [15]

```
g_odata[ai] = temp[ai + bankOffsetA];
g_odata[bi] = temp[bi + bankOffsetB];
```

Listing 2.9: Blok E [15]

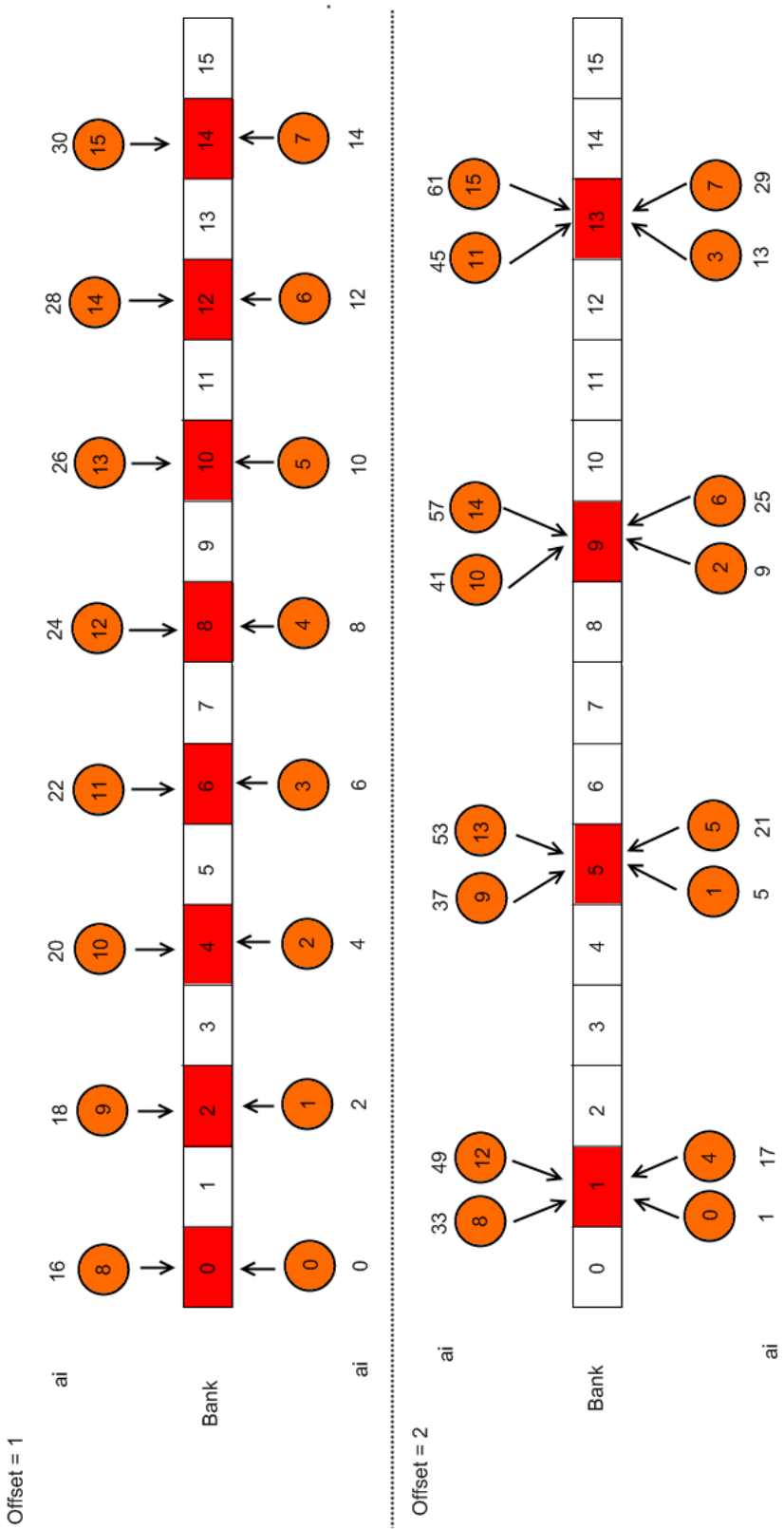
We voegen dus steeds wat extra opvulling bij de adressering. Door voor elke 16 elementen een extra opvulling toe te voegen zorgen we ervoor dat de adressering zonder bank conflicten gebeurt zoals voorgesteld in figuur 2.10.

Door deze vorm van adressering kunnen we de bank conflicten volledig vermijden.

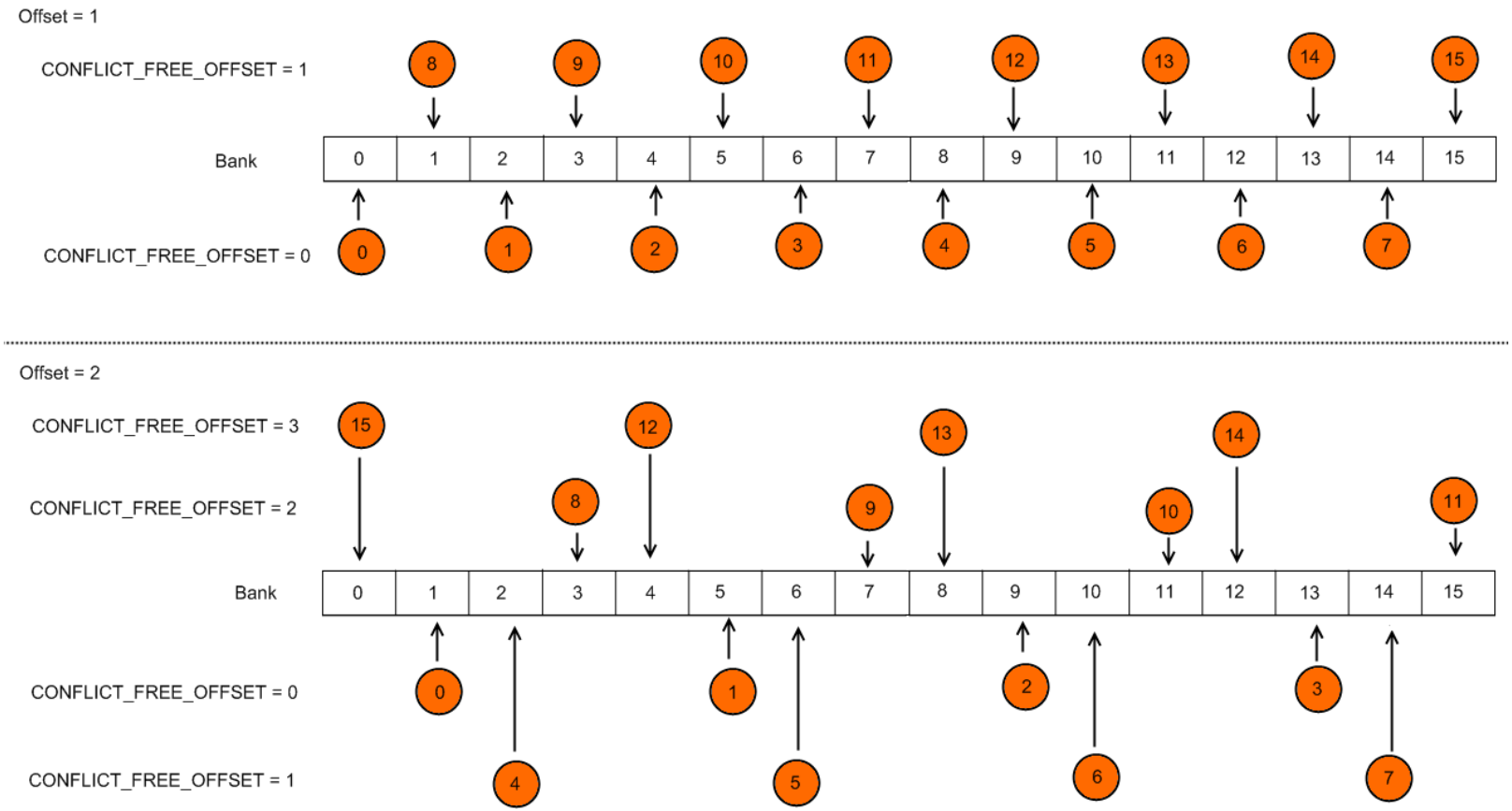
2.3.7 Conclusie

De laatste jaren hebben aangetoond dat de GPU aan een veel snellere evolutie bezig is dan de CPU en het einde van deze trend is nog niet in zicht. GPUs hebben al ruwweg 10 maal de rekencapaciteit en doorvoersnelheid van en CPU en gaan alleen maar sneller worden. Bovendien is een GPU een relatief goedkope uitbreiding van systeem en meerdere GPUs kunnen perfect samenwerken om nog meer performantie te bekomen binnen hetzelfde systeem. Een nieuwe, extreem parallelle manier van programmeren dringt zich dus op en met CUDA krijgt de programmeur de mogelijkheid om deze manier van programmeren toe te passen.

In de vorige secties toonden we aan aan dat niet alleen grafische applicaties van deze evolutie kunnen gebruik maken maar dat er enorm veel snelheidswinst te halen is uit het gebruik van CUDA. Er was sprake van applicaties die tot bijna 200 maal sneller waren na het gebruiken van CUDA. Het loont dus de moeite om te bekijken of een applicatie geen nut kan hebben bij het gebruik van CUDA.



Figuur 2.9: Een schematische weergave van bank conflicten voor de code in listing 2.4. De treads worden voorgesteld door de cirkels en adresseren de banks voorgesteld door een pijl naar de bank. Bank conflicten komen voor bij de gekleurde banks. Ter illustratie is de waarde van ai bij elke *thread* weergegeven. Merk op dat een hogere offset een hogere graad van bank conflicten veroorzaakt.



Figuur 2.10: Een schematische weergave van het gebruik van de banks zonder conflicten door toevoeging van opvulling met CONFLICT_FREE_OFFSET(ai) waarbij ai = offset*(2*tid+1)-1

Hoofdstuk 3

Detectie

In dit hoofdstuk zullen we een aantal technieken bespreken om tot de detectie van objecten, meer specifiek de detectie van een bal te komen uit één of meerdere video's.

3.1 Beeld differentiatie

Veruit de meest gebruikte methode om tot detectie te komen is de methode van beeld differentiatie. Deze methode vergelijkt steeds weer twee opeenvolgende beelden in een videoreeks om te bepalen waar de beweging plaatsvond. Deze methode is dan ook de meest eenvoudige methode om tot kandidaten te komen voor de detectie van bewegende objecten.

Beeld differentiatie detecteert de bewegende delen van een beeld doormiddel van drempelwaarde. Gegeven een afbeelding X^t op tijdstip t, het differentiatiebeeld D^t met drempelwaarde T in de vorm van een binair masker wordt gegeven als:

$$D_t = \begin{cases} 1 & X^t - X^{t-1} > T \\ 0 & X^t - X^{t-1} \leq T \end{cases} \tag{3.1}$$

De drempelwaarde hangt af van de scene waarop deze wordt toegepast en wordt meestal experimenteel bepaald. Het resultaat van deze methode is het binair masker D^t dat de bewegende delen van het beeld in het wit weergeeft en de niet-bewegende delen, de achtergrond in het zwart.

3.2 Achtergrond differentiatie

Bij een achtergrond differentiatie wordt een onderscheid gemaakt tussen de bewegende voorgrond pixels en de relatief statische achtergrond pixels om hieruit de bewegende onderdelen te detecteren. In tegenstelling tot de vorige sectie wordt een nieuw beeld niet vergeleken met

het vorige beeld maar met een bepaalde waarde voor de achtergrond.

Een mogelijke implementatie zou zijn om een statisch beeld op te stellen in het begin van de detectie en deze achtergrond voor de rest van de scene te gebruiken. Deze aanpak brengt echter verschillende problemen met zich mee. Zo kan een veranderende scene door deze methode onmogelijk op een voldoende accurate manier onderzocht worden. De achtergrond wordt als statisch aanzien zodat bijvoorbeeld een nieuw statisch object dat in de scene wordt geïntroduceerd steeds als een bewegend object zal gedetecteerd worden. Ook als er zich een fout zou voordoen bij de berekening van de originele achtergrond kunnen de fouten zich over de rest van de scene opstapelen en tot een onnauwkeurige detectie leiden.

Een betere methode is duidelijk een adaptieve achtergrond, een methode waarbij de achtergrondinformatie steeds aangepast wordt aan de huidige scene door het uitmiddellen van gedetecteerde achtergronden over verschillende beelden heen. Maar ook dit heeft zijn gebreken. Zo is het moeilijk om de achtergrond te onderscheiden van traag bewegende objecten, en erger nog, indien de achtergrond weinig zichtbaar is door bijvoorbeeld veel beweging van grote objecten is het quasi onmogelijk om deze te detecteren.

Een zeer robuuste methode om aan achtergrond differentiatie te doen is de methode beschreven door Stauffer en Grimson [59] en gebruikt wordt in Ren et al. [46][48], etc... Hier wordt elke pixel beschreven als een mix van statistische waarden. Elke pixel wordt initieel als een deel van de voorgrond aanzien tot er genoeg bewijs is om te kunnen beslissen dat de pixel tot de achtergrond behoort. Elke statistische waarde, of gaussiaan, bepaalt voor een pixel de kans om een deel van de voorgrond of de achtergrond te zijn onder varierende lichtcondities. Verschillende lichtcondities vereisen immers verschillende interpretaties.

De methode kent twee belangrijke parameters, α een leerconstante en T de drempelwaarde voor de proportie van het beeld dat gedetecteerd wordt als achtergrond. Na elke update van de gaussianen wordt een simpele heuristiek gebruikt om te bepalen welke pixels een deel van de achtergrond zijn en worden de pixels gegroepeerd in geconnecteerde delen. Tenslotte worden de geconnecteerde delen gevolgd van beeld tot beeld.

3.2.1 Gaussiaanse mix model

In deze sectie gebruiken we de notatie als gedefinieerd door Stauffer en Grimson [59] met betrekking tot het gaussianse mix model.

Een *pixel proces* is de informatie over de geschiedenis van een pixel. Het is een vector die

de staat van de pixel in chronologische volgorde bewaart. Elk element van de vector bestaat uit ofwel een vector van rgb waarden of één enkele waarde in het geval van een grijswaarde beeld. Dus het *pixel proces* van de pixel $X^t(x, y)$ met X^t net als in de vorige sectie het beeld op tijdstip t is gegeven als:

$$\{X_1, ..., X_t\} = \{X^i(x, y) : 1 \leq i \leq t\} \tag{3.2}$$

Het model wordt nu gegeven door de volgende vergelijking, het *pixel proces* van elke pixel is gemodelleerd als een mix van K gaussiaanse distributies. De waarschijnlijkheid van het observeren van pixel X_t is gegeven als:

$$P(X_t) = \sum_{i=1}^{K} \omega_{i,t} \eta(X_t, \mu_{i,t} \Sigma_{i,t}) \tag{3.3}$$

Hierbij is K het aantal verdelingen en hangt deze af van de gewenste prestaties, hoe groter de K waarde hoe accurater de differentiatie maar een grote K is kostelijk voor zowel CPU als geheugen. $\omega_{i,t}$ stelt een schatting van het gewicht voor van de i^{de} gaussiaan in de mix op tijdstip t, een relatieve verhouding van de hoeveelheid pixeldata waarvoor deze gaussiaan geldig is en η wordt later gegeven.

$\Sigma_{i,t}$ is de covariantie matrix van de i^{de} gaussiaan in de mix op tijdstip t en wordt benaderd door vergelijking (3.4), we stellen deze gelijk met de variantie van het beeld X, let op het feit dat er geen superscript gebruikt wordt bij het beeld, dit omdat Stauffer en Grimson [59] aannemen dat de variantie voor elk beeld X^t identiek is en dat de pixelwaarden voor rood, groen en blauw onafhankelijk zijn maar met dezelfde variantie. Alhoewel dit niet waar is, zorgt deze aanname ervoor dat we een kostelijke matrix inversie kunnen vermijden dus we winnen hiermee snelheid ten koste van nauwkeurigheid. Deze aanpak werkt echter alleen goed bij lineaire kleurvoorstellingen zoals RGB, andere voorstellingen zoals HSV, waarbij de verschillende kanalen zeer verschillende varianties kunnen hebben zijn hier niet zo mee gebaat [43] en gebruiken best de nauwkeurigere methode.

$$\Sigma_{k,t} = \sigma_k^2 X \tag{3.4}$$

$\mu_{i,t}$ is dan weer de mediaan van de i^{de} gaussiaan in de mix op tijdstip t en η is gegeven als in (3.5):

$$\eta(X_t, \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(X_t - \mu_t)^T \Sigma^{-1} (X_t - \mu_t)} \tag{3.5}$$

We komen dus bij een formule waarbij de distributie van recentelijk bepaalde pixels gedefinieerd wordt door een mix van gaussianen. De nieuwe pixelwaarde zal in het algemeen gerepresenteerd worden door één van de gaussianen en zal gebruikt worden om het model aan

te passen voor de volgende stap.

Elke nieuwe pixel wordt vervolgens aanzien als een set van 1 en moet berekend worden volgens het model gegeven in (3.3), het probleem hierbij is dat dit doen voor elke nieuwe pixel een zeer kostelijke operatie is. Daarom gebruiken we een on-line benadering. Elke nieuwe pixelwaarde wordt vergeleken met de bestaande distributies tot er een gelijkwaardige pixel gevonden wordt binnen 2.5 standaarddeviaties van een normaaldistributie.

Indien er geen van deze waarden gevonden wordt vervangen we de minst waarschijnlijke distributie door een distributie waarbij de huidige pixelwaarde de mediaan is met een hoge variantie en een klein gewicht. Het gewicht wordt aangepast aan de hand van de volgende formule:

$$\omega_{k,t} = (1 - \alpha)\omega_{k,t-1} + \alpha(M_{k,t}) \tag{3.6}$$

Met α nog steeds de leerconstante en $M_{k,t}$ is 1 voor het gevonden model en 0 voor al de andere modellen. Na deze benadering worden de gewichten opnieuw genormaliseerd. De μ en σ parameters voor de nieuwe distributie worden aangepast als volgt:

$$\mu_t = (1 - \rho)\mu_{t-1} + \rho X_t \tag{3.7}$$

$$\sigma_t^2 = (1 - \rho)\sigma_{t-1}^2 + \rho(X_t - \mu_t)^T(X_t - \mu_t) \tag{3.8}$$

$$\tag{3.9}$$

Met

$$\rho = \alpha\eta(X_t|\mu_k, \sigma_k) \tag{3.10}$$

Om nu tot een detectie te komen van de achtergrondparameters gebruiken een heuristiek waarbij we geïnteresseerd zijn in de gaussianen met de meeste ondersteunende data, een hoge ω_k, en de minste variantie, een lage σ^2. Een achtergrond heeft namelijk typisch een zeer lage variantie indien een statisch object zichtbaar is, indien nu een bewegend object hier voorbij komt zal er ofwel een nieuwe distributie begonnen worden, ofwel zal een huidige distributie aangepast worden, maar telkens zal de variantie hiervan hoog liggen.

Een eerste stap in bepalen welke gaussianen hiervoor in aanmerking komen is het ordenen van deze gaussianen op de verhouding ω/σ Deze waarde zal toenemen als ofwel een lage variantie hebben, ofwel een veel ondersteunende data, of beide.

Na de ordening worden de B eerste gaussianen gekozen als het achtergrondmodel, waarbij

$$B = min_b\left(\sum_{k=1}^{b} \omega_k > T\right) \tag{3.11}$$

Met T een drempelwaarde voor de minimum portie van de data dat als een achtergrondobject kan aanzien worden. Met andere woorden wat dit doet is het vergelijken van mogelijke achtergrond kandidaten met een voorop bepaalde waarde van de fractie van het beeld dat een object minimaal moet zijn om als achtergrond in aanmerking te komen.

Na deze stap is de detectie van bewegende onderdelen compleet, we hebben een onderscheid tussen voor-en achtergrond pixels en we geven dit weer in een binair masker waar een bewegende voorgrond pixel in het wit wordt weergegeven en een statische achtergrond pixel in het zwart.

3.3 Kleur differentiatie

Een andere detectiemethode is het detecteren van een object doormiddel van zijn kleurwaarde. Deze methode werkt bovenop een achtergrond differentiatie methode, bijvoorbeeld de hierboven beschreven methode, door op de gedetecteerde voorgrondpixels voort te werken en de objecten te detecteren door hun kleurwaarden te analyseren.

Een eerste stap in dit algoritme is het uitvoeren van een geconnecteerde componenten algoritme op de voorgrond pixels van de scene. We delen met andere woorden de voorgrond op in geconnecteerde regionen. Voor elk van deze regionen zullen we een kleurmodel opstellen en bepalen of dit al dan niet een gezocht object is.

De termen in de volgende sectie zijn gebaseerd op de notatie beschreven door Orwell et al. [39].

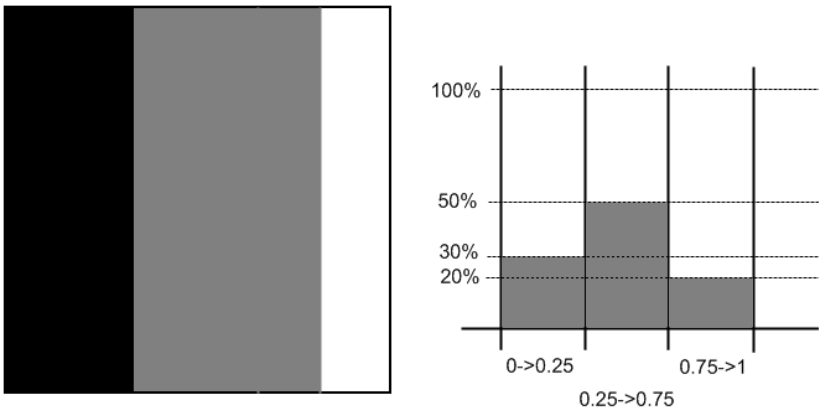
Elk object in een scene heeft zijn eigen bepaalde set van kleurwaarden waardoor het kan geidentificeerd worden, zo zal bijvoorbeeld een tennisbal grotendeels geel zijn met enkele witte pixels waar de naden zitten, een klassieke voetbal zal dan eerder half wit en half zwart zijn. In dit algoritme worden deze kleureigenschappen samengevat in een vector m als gegeven in (3.12). m is gebaseerd op een onderliggend signaal s met een ruis factor ν . De ruisterm ν is een term die gebruikt wordt meestal om verschil in belichting van een object te kwantificeren maar ook om observatieruis en foutjes in de achtergrond differentiatie te kunnen opvangen.

$$m = s + \nu \tag{3.12}$$

Een histogram wordt opgesteld voor de kleurenruimte. Elke vector m bestaat nu uit zijn histogram waarden s^j als gegeven in (3.13), meer specifiek elke s^j is een genormaliseerde voorstelling van hoeveel pixels er binnen deze histogram onderverdeling vallen.

$$m = \{s^j | j = 1...N\} \tag{3.13}$$

Zie figuur 3.1 voor een voorbeeldje van een simpele histogram ruimte, de ruimte is opgesteld voor een grijswaarde beeld en het histogram kent 3 onderverdelingen van de kleurenruimte, in het geval van het voorbeeld zou de vector $m = \{0.3, 0.5, 0.2\}$. In de praktijk is zo een histogram echter multidimensionaal, zo zal voor een RGB of een HSV verdeling het histogram uit drie dimensies bestaan.



Figuur 3.1: Een simpele afbeelding(links) met zijn bijbehorend histogram(rechts), de grijswaarden kleurenruimte wordt onderverdeeld in 3 verdelingen waarvoor de eigenschappen onder het histogram genoteerd worden.

De methode modelleert de *a posteriori* kans $P(M|m_i)$ hetgeen de kans weergeeft dan een meting m_i een model M definieert en gebruikt wordt als een indicator dat m_i en M hetzelfde object representeren. Een *model* van een object O wordt opgesteld aan de hand van geobserveerde waarden voor dit object en continu aangepast om tot een *a posteriori* kans $P(M|m_i)$ te komen.

De meest eenvoudige vorm om tot een model te komen zou zijn om één enkele observatie te gebruiken als een model, hierbij wordt $M \equiv m_1$, dit wil zeggen dat een beslissing of m_2 voorgesteld kan worden door model M hetzelfde is als beslissen of m_1 en m_2 hetzelfde signaal voorstellen. In de praktijk wordt een model echter eerder als een gemiddelde van verschillende waardes weergegeven zodat een beslissing of een signaal tot een model behoort in een statistisch kader kan bestudeerd worden.

Gegeven de meting m_1 van een regio gebaseerd op het signaal s_1 onder invloed van ruis in de observatie. Door de ruisterm zal de meting m_1 dikwijls andere waarden bevatten dan s_1 en soms zelfs als een ander signaal aanzien worden. De kans op een dergelijke fout in de classificatie van een model is gegeven als p_i. Als we aannemen dat zo een misclassificatie enkel kan gebeuren tussen naburige modellen dan wordt verwacht dat een meting m_1 onder invloed staat van $\frac{1}{8}s^j p^n(j)$ voor elke s^j die buren van het signaal s^i voorstellen in een kleurvoorstelling waarbij elke s^i 8 buren heeft. De totale invloed zal een gemiddelde en variantie van $\langle s^j p^n(j) \rangle$

hebben waarbij j de naburige componenten van i voorstelt [39].

We kunnen nu de verwachte variantie van elk element m^i van de vector m definieren als:

$$\sigma^2(m^i) = p_i s^i + \langle s^j p_j \rangle \tag{3.14}$$

De variantie van een element m_i kan dus weergegeven worden als de kans p_i dat een meting s_i bij een naburige histogram onderverdeling is beland plus de kans dat één van de buren van s^i een meting fout naar s_i klasseert.

Gebruik makende van een statische camera is het niet moeilijk om een waarde voor p_i te berekenen. We gebruiken hiervoor de variantie op een aantal beelden van eenzelfde statisch signaal uit de camera. De geobserveerde varianties geven een zeer goed model om de misclassificatie van elke observatie weer te geven.

We kunnen nu de vectoren m_1 en m_2 vergelijken en een kans berekenen of de vectoren eenzelfde signaal s als basis hebben. Een tweevoudige hypothese wordt opgesteld om te zien of deze 2 vectoren significant van elkaar verschillen aan de hand van een χ^2 verdeling, deze hypothese bepaalt of de vectoren al dan niet eenzelfde signaal bepalen.

Tenslotte, om een object te detecteren is het dan enkel noodzakelijk om een kleursignaal op te stellen voor het gewenste object, waarna deze methode kan gebruikt worden om te kijken of een waargenomen object al dan niet het gezochte object is.

3.4 Wavelets

Wavelets zijn een mathematische tool om een functie op een hiërarchische manier te decomposeren in een detailcoëfficiënt en een approximatiecoëfficiënt. Wavelets hebben vooral toepassingen in gebieden zoals beeldcompressie maar hebben binnen de computer graphics en computervisie een groot aantal applicaties [60], hier gebruiken we de eigenschappen van wavelets, zoals bijvoorbeeld hun robuustheid ten opzichte van ruis, om tot detectie van objecten te komen.

De meest eenvoudige wavelet basis is de Haar basis [18], de Haar basis definieert een manier om een reeks waarden op een andere manier te noteren. In zijn discrete vorm definieert de Haar basis een approximatiecoëfficiënt simpelweg als het gemiddelde van twee opeenvolgende waardes. Vervolgens kan de detail coëfficiënt berekend worden door deze te definieren als het verschil tussen de eerste van de twee opeenvolgende waardes en de benaderende coëfficiënt

[13]. Door deze methode hierarchisch toe te passen bekomen we een Haar basis waarin we de detailcoëfficiënten in verschillende resoluties kunnen waarnemen.

De Haar basis is niet de enige wavelet basis, het is echter de meest eenvoudige en meest efficiënte wavelet basis om te berekenen maar bezit toch veel van de goede eigenschappen van meer ingewikkeldere wavelet basissen zoals de Daubechies [18] wavelet basis .

Merk op dat het mogelijk is om uit de Haar basis terug de originele vector te reconstrueren. Gegeven een detail- en een approximatiecoëfficiënt is het mogelijk om de twee originele waarden terug te berekenen. De eerste waarde is simpelweg de som van de approximatiecoëfficiënt en de detailcoëfficiënt terwijl de tweede waarde het verschil is.

3.4.1 Eéndimensionale wavelet tranformatie

Een voorbeeldje van deze methode in 1 dimensie volgt. De originele vector wordt weergegeven in (3.15).

$$[\,5 \qquad 9 \qquad 4 \qquad 6\,] \tag{3.15}$$

We berekenen eerst de gemiddelden of approximatiecoëfficiënten van elke 2 opeenvolgende getallen, namelijk $\frac{5+9}{2} = 7$ en $\frac{4+6}{2} = 5$, we komen dus uit bij (3.16)

$$[\,7 \qquad 5\,] \tag{3.16}$$

Vervolgens berekenen we de detail coëfficiënten voor deze stap: $5 - 7 = -2$ en $4 - 5 = -1$, weergegeven in (3.17)

$$[\,-2 \qquad -1\,] \tag{3.17}$$

We hebben nu een Haar basis van resolutie 2, want zowel vectoren (3.16) als (3.17) hebben 2 elementen. We passen hetzelfde proces nogmaals toe op (3.16) en bekomen (3.18) als gemiddelde en (3.19) als detail.

$$[\,6\,] \tag{3.18}$$

$$[\,1\,] \tag{3.19}$$

Aangezien we nu op resolutie 1 gekomen zijn moet hier de recursie stoppen en hebben we de Haar basis berekend van de originele vector (3.15).

Nogmaals een beknopt overzichtje van de vorige methode in de volgende tabel:

Resolutie	Approximatiecoëfficiënten	Detailcoëfficiënten
4	[5 9 4 6]	
2	[7 5]	[−2 − 1]
1	[6]	[1]

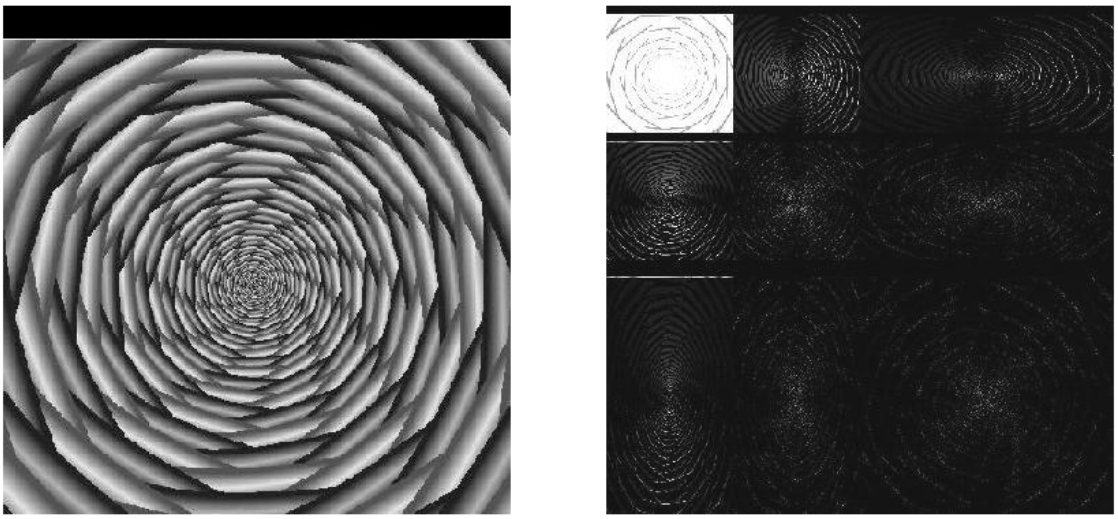
We kunnen nu de wavelet transformatie definieren als 1 enkele benaderende coëfficiënt die in de Haar basis de vorm aanneemt van het gemiddelde van de hele vector, gevolgd door al de detailcoëfficiënten in oplopende volgorde van resolutie [60]. De wavelet transformatie van de vector (3.15) is gegeven in (3.20).

$$[\ 6 \quad 1 \quad -2 \quad -1\ ] \tag{3.20}$$

In de praktijk wordt de wavelet transformatie nog genormaliseerd door de detailcoëffciënte te vermenigvuldigen met een factor $\frac{1}{\sqrt{2}}$ maar dit is in dit voorbeeld weggelaten voor de duidelijkheid.

3.4.2 Tweedimensionale wavelet transformatie

De standaard manier om aan wavelet decompositie te doen van tweedimensionale afbeeldingen is om eerst op de rijen en vervolgens op de kolommen één voor één een wavelet tranformatie toe te passen [60], zie figuur 3.2 voor een voorbeeld van van een tweedimensionale wavelet tranformatie. Let op de detailcoëfficiënten van de tranformatie die aangeven op welke plaatsen het beeld de grootste veranderingen ondergaat.



Figuur 3.2: Een voorbeeld van een tweedimensionale wavelet transformatie na 3 iteraties (rechts) met de originele afbeelding links. Merk op hoe de wavelet transformatie de gradiënten in het beeld weergeeft[61].

3.4.3 Wavelets en detectie

Wavelets zijn uitermate geschikt om discontinuïteiten te berekenen in data, zo kunnen ze gebruikt worden om randen en overgangen te detecteren in beelden. Een andere goede eigenschap is hun relatieve ongevoeligheid voor ruis. Door meerdere waarden te combineren in hun multiresolutie aanpak zijn ze zeer robuust met betrekking tot ruis. Deze eigenschappen maken wavelets zeer geschikt om te gebruiken bij de detectie van objecten.

Wavelet transformatie voor cirkel detectie

Door D'Orazio et al. [8] en Leo et al.[24] wordt een methode beschreven waarbij wavelet transformaties gebruikt worden om tot de detectie te komen van een voetbal door eerst de wavelets transformatie te gebruiken om circulaire patronen in het beeld te detecteren.

Een tweedimensionale wavelet transformatie wordt toegepast op een beeld om de gradiënten hieruit te halen, de bedoeling is om uit de textuur informatie van de beelden de randen van objecten te detecteren, kijk naar figuur 3.3 voor een voorbeeld hiervan. Dit geeft ons een beeld van de detailcoëfficiënten van een beeld in verschillende resoluties, dit wil zeggen dat in de wavelet transformatie vooral de randen van de objecten duidelijk zullen zijn, meer specifiek de plotse overgangen tussen verschillende kleuren.

Figuur 3.3: Een tweedimensionale wavelet transformatie van een televisiebeeld geeft de gradiënten van het beeld weer. Figuur uit [8]

We definiëren de Circulaire Hough Transformatie (CHT)[8], een methode om circulaire patronen te detecteren van een gegeven grootte op de tweedimensionale wavelet transformatie van een beeld als in (3.21) met de straal R van de cirkel die we willen zoeken $R \in [R_{min}, R_{max}]$.

$$u(x,u) = \frac{\int\int_{D(x,y)} \vec{e}(\alpha,\beta) \cdot \vec{O}(\alpha - x, \beta - y) d\alpha d\beta}{2\Pi(R_{max} - R_{min})} \tag{3.21}$$

waarbij $\vec{e}$ de gradiënt normalisatie term en $\vec{O}$ de kernel vector is, gegeven door:

$$\vec{e}(x,y) = \left[\frac{E_x(x,y)}{|E|}, \frac{E_y(x,y)}{|E|}\right]^T \tag{3.22}$$

$$\vec{O}(x,y) = \left[\frac{\cos(\tan^{-1}(\frac{y}{x}))}{\sqrt{x^2+y^2}}, \frac{\sin(\tan^{-1}(\frac{y}{x}))}{\sqrt{x^2+y^2}}\right]^T \tag{3.23}$$

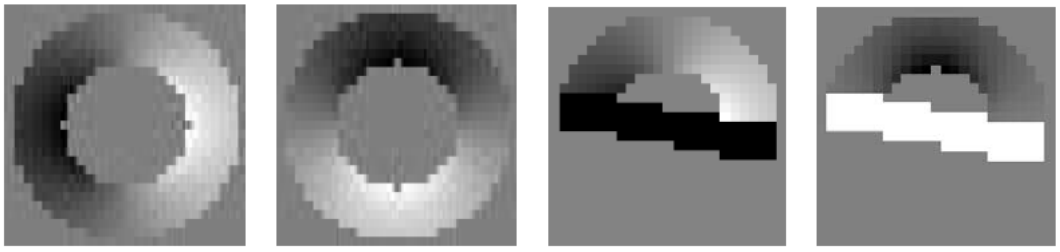
Tenslotte definiëren we het domein $D(x,y)$

$$D(x,y) = \{(\alpha,\beta) \in \Re^2 | R_{min}^2 \leq (\alpha - x)^2 + (\beta - y)^2 \leq R_{max}^2\} \tag{3.24}$$

In (3.21) wordt er dus voor elk punt in een afbeelding berekend of het al dan niet het middelpunt van een cirkel van detailcoëfficiënten voorstelt. Dit gebeurt doormiddel van een kernel vector $\vec{O}$ die bepaalt of er al dan niet een rand pixel binnen het interval van de straal ligt. Indien er genoeg pixels binnen dit interval liggen kan aangenomen worden dat dit punt het middelpunt van een cirkel beschrijft.

De gradiënt normalisatie term is nodig om ervoor te zorgen dat de meest complete cirkel gekozen wordt, en niet simpelweg de cirkel met het meeste contrast. Ook de kernel vector kent een normalisatie term $\sqrt{x^2+y^2}$, deze is toegevoegd om er voor te zorgen dat cirkels van verschillende grootte hetzelfde gewogen worden indien ze tussen de minimum en maximum waardes vallen. Door tenslotte in (3.21) alles te delen door $2\Pi(R_{max} - R_{min})$ wordt ervoor gezorgd dat de resultaten van $u(x,y)$ steeds in het interval $[-1,1]$ liggen onafhankelijk van de grootte van de cirkel.

Zowel $\vec{O}$ als $\vec{e}$ worden geïmplementeerd als convolutiemaskers , als weergegeven in figuur 3.4.



Figuur 3.4: Links de twee convolutiemaskers die worden gebruikt om de kernel vector te implementeren, rechts die van de gradiënt vector. Figuren uit [8]

Het resultaat van deze analyse is een lijst van mogelijke kandidaten voor de bal weergegeven door het middelpunt en de straal van de kandidaat.

Temporele wavelet transformatie

Een temporele wavelet transformatie laat toe om de dynamische aspecten van videobeelden te kunnen weergeven, met name welke pixel bewogen heeft en welke niet. Een temporele wavelet transformatie is, zoals de naam zegt een wavelet transformatie over de tijd. In zijn discrete vorm wil dit zeggen dat het een wavelet transformatie is van een aantal opeenvolgende beelden. Merk op dat differentiatie tussen twee opeenvolgende beelden, als in een vorige sectie besproken, een speciale vorm van temporele wavelet transformatie is, namelijk eentje met een resolutie van 2.

Een temporele wavelet transformatie heeft het voordeel ten opzichte van de klassieke wavelet transformatie van nog minder gevoelig te zijn voor ruis artefacten, door de informatie over verschillende beelden te combineren in de typische multiresolutie decompositie van wavelets wordt het effect van ruis nog verminderd en komt het effect van beweging in de pixels nog meer bovendrijven [56].

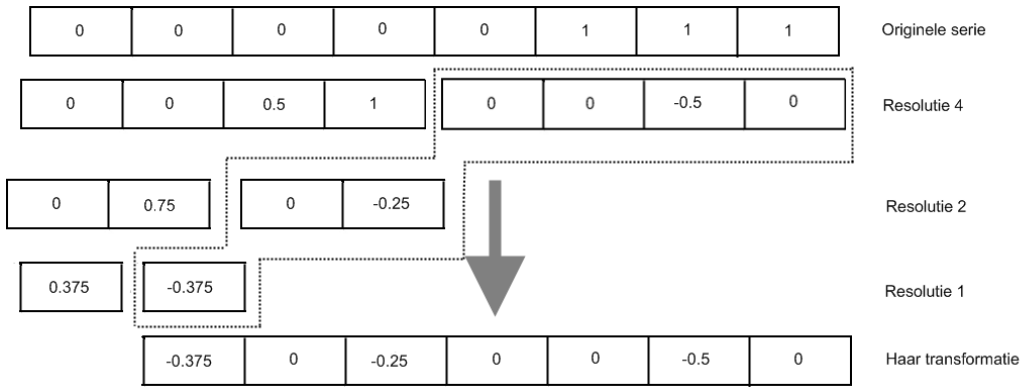
Het nadeel van de methode die in de vorige sectie beschreven is wordt duidelijk als hij wordt toegepast op kleine, snel bewegende balletjes zoals tafeltennis of squash balletjes. Zulke objecten hebben typisch een zeer grote vervorming door *motion blur* die optreedt door hun hoge snelheid waardoor de vorm niet meer per definitie als rond zal waargenomen worden, bekijk figuur 3.5 als voorbeeld van de voorstelling van dezelfde bal onder invloed van verschillende hoeveelheden motion blur. Dit probleem kan vermeden worden door camera's die opnemen tegen een zeer hoge snelheid te gebruiken, maar dit is meestal een zeer dure oplossing en dus ongewenst.

Figuur 3.5: Een voorstelling van dezelfde voetbal op verschillende momenten in een reeks beelden, merk op hoe motion blur steeds weer de vorm van de bal verandert [49]

Tenslotte, als we naar zeer kleine balletjes gaan kijken hebben deze zeer dikwijls een zeer lage signaal/ruis verhouding, simpelweg omdat het "signaal" van het balletje zeer klein is, waardoor de verhouding bij een constante ruis over heel het beeld typisch groot wordt. Desondanks de robuustheid van wavelets ten opzichte van ruis is dit nog steeds een beperkende factor omdat het moeilijk wordt om te differentieren tussen ruis en het kleine object [62].

Een voorbeeldje van zo een temporele wavelet transformatie wordt gegeven in figuur 3.6. De

originele serie stelt hier de verschillende waardes van een pixel voor. Stel je de scene voor als een zwarte achtergrond (0) waarop een wit object (1) rond beweegt, zoals je kan zien in de figuur wordt in beeld 6 de pixel plots wit, een teken dat het witte object door deze pixel kan gezien worden. De Haar transformatie hiervan geeft nu een aantal pieken weer. Aan de hand van deze waardes kan men veel afleiden. De waarde op positie 1 geeft aan dat er een beweging heeft plaatsgevonden in 1 van de 8 opeenvolgende frames van de transformatie. De waarde op positie 3 geeft aan dat die in de laatste 4 frames gebeurd is en de waarde op positie 6 geeft aan dat de beweging zich voordeed tussen frame 5 en 6.



Figuur 3.6: Een voorbeeldje van een haar transformatie op een simpele tijd serie.

Indien men na de temporele wavelet transformatie de pixels terug op hun originele plaats zet bekomen we een reeks wavelet beelden, elk van deze beelden geeft een beeld van de temporele veranderingen en aan de hand van zijn buren kunnen er allerlei voorspellingen gedaan worden.

3.4.4 Wavelet Analyse

Uit de wavelet informatie van de vorige secties moet vervolgens een beslissing genomen worden over welke pixels een beweging voorstellen en welke pixels niet. We bespreken hier enkele mogelijke oplossingen hiervoor.

Drempelwaarde analyse

De simpelste en meest gebruikte methode om uit een wavelet transformatie te beslissen of een pixel al dan niet een beweging ondervindt is om deze te vergelijken met een drempelwaarde. Gedefinieerd equivalent aan (3.1) in de sectie over beelddifferentiatie. Waarbij de waarde van een pixel in het masker $D_{tk}(x, y)$ op tijdstip t en op de wavelet diepte k gegeven wordt door:

$$D_{tk}(x,y) = \begin{cases} 255 & W_{tk}(x,y) > T \\ 0 & W_{tk}(x,y) \leq T \end{cases} \tag{3.25}$$

Davies et al. [6] beschrijven een robuustere methode hiervoor door voor elke pixel niet zijn eigen waarde maar een gemiddelde van de waarde van zijn buren te nemen. Dit is vergelijkbaar met een low-pass filter toepassen voor het binaire masker wordt bepaald. En zorgt ervoor dat ruis iets minder invloed heeft maar zorgt er evengoed voor dat zeer kleine objecten met een laag contrast ongedetecteerd kunnen blijven.

De vergelijking wordt dan gegeven door:

$$D_{tk}(x,y) = \begin{cases} 255 & O_{tk}(x,y) > T \\ 0 & O_{tk}(x,y) \leq T \end{cases} \tag{3.26}$$

Waarbij

$$O_{tk}(x,y) = \frac{1}{N^2} \sum_{i=-\frac{-N}{2}}^{\frac{N}{2}} \sum_{j=\frac{-N}{2}}^{\frac{N}{2}} W_{tk}(x+i, y+j) \tag{3.27}$$

Met N de grootte van het venster waarmee we de waarde uitmiddellen.

Statistische Analyse

In [67] wordt de analyse van de wavelet transformatie op een statistische manier gedaan. De bedoeling is de detailcoëfficiënten die in een vorige stap bekomen zijn om te zetten in een binair masker. Het binair masker dient nog steeds aan te duiden in welke pixels er een temporele verandering plaatsvindt en in welke niet.

Temporele verandering van een pixel wordt bepaald binnen een statistische hypothese. Er worden 2 hypotheses H_0 en H_1 opgesteld voor elke pixel en in elke resolutie. De H_0 hypothese stelt temporele verandering voor in de betrokken pixel en H_1 wordt gebruikt om aan te duiden dat er geen temporele verandering geweest is op deze pixel. Een beslissing wordt genomen aan de hand van (3.28), met λ een waarde die opgezocht kan worden in statistische tabellen, ϕ volgt een χ^2 verdeling met 3 vrijheidsgraden.

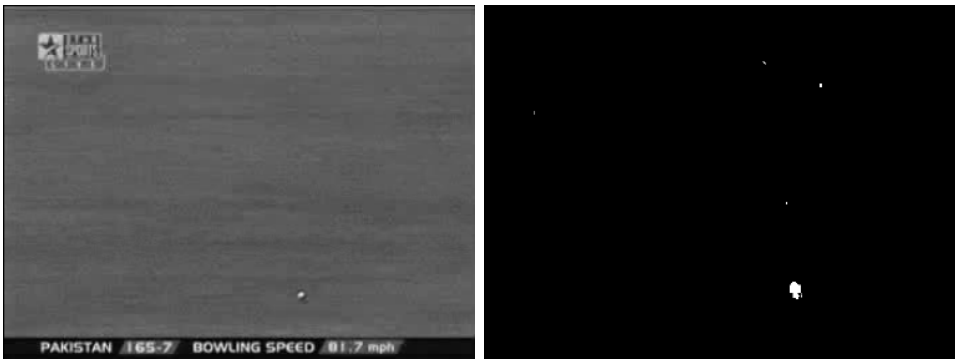
$$Verwerp\ H_0\ indien\ \phi^k(p) \geq \lambda \tag{3.28}$$

De waarde van ϕ wordt bepaald door formule (3.29) [67].

$$\phi^k(p) = \frac{1}{2\sigma_k^2}\left[\frac{1}{N}\left(\sum_{i=1}^{N} D^k(p_i)\right)^2 + \frac{1}{\sum_{i=1}^{N} x_i^2}\left(\sum_{i=1}^{N} x_i D^k(p_i)\right)^2 + \frac{1}{\sum_{i=1}^{N} y_i^2}\left(\sum_{i=1}^{N} y_i D^k(p_i)\right)^2\right] \tag{3.29}$$

Waarbij D^k de detailcoëfficiënten voorstellen op niveau k, N is de grootte van het venster met betrekking tot de pixels gecentreerd in punt p en (x_i, y_i) duidt de relatieve locatie aan van de pixels ten opzichte van het midden van het venster p. Ten slotte is σ_k^2 de variante van de waardes van D^k binnen het venster.

Het resultaat van deze operatie is een binair masker waarop de pixels die gedetecteerd werden als "bewogen" pixels in het wit weergegeven worden, kijk naar figuur 3.7 voor een voorbeeld van een dergelijk binair masker.



Figuur 3.7: Een voorbeeld van een binair masker (rechts), bekomen door statistische analyse van een temporele wavelet transformatie op het originele beeld (links) [67].

3.5 Binair masker analyse

In al de secties in dit hoofdstuk buiten 3.4.3 kwamen we tot een binair masker dat op één of andere manier de beweging van de scene bepaalde. Een binair masker is een stap in de goede richting maar dit is nog geen detectie. Daarom bespreken we in deze sectie enkele methodes om van een binair masker naar de detectie van een object te gaan.

3.5.1 Opening

Een binair masker heeft vaak nog last van ruis. Het is daarom aangewezen om voor er verdere operaties op te doen eerst het resultaat wat op te kuisen. Een typische manier om dit te doen is een opening operatie, een opening heeft meestal het effect om in het binair masker de randen van de witte regionen vlakker te maken, het verwijdert dunne uitstulpingen en smalle verbindingen tussen twee witte regionen [13]. Met andere woorden, het verwijdert wat ruis aan de randen van gedetecteerde bewegende objecten waardoor de detectie ervan op een nauwkeurigere manier kan gebeuren.

Een opening wordt gedefinieerd door een dilatie na een erosie met hetzelfde structurerend element ofwel:

$$A \circ B = (A \ominus B) \oplus B \tag{3.30}$$

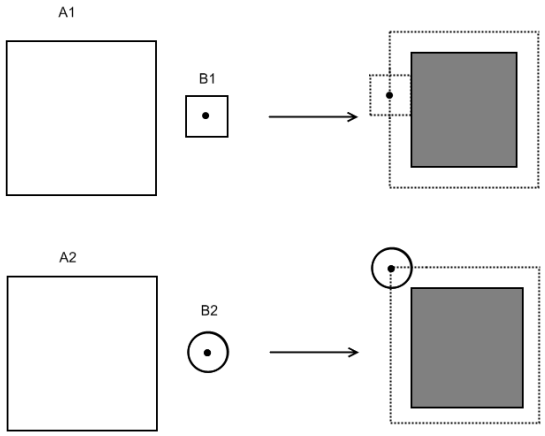
Erosie

Erosie op een binair masker A met structurerend element B wordt gedefiniëerd als in formule (3.31) [13].

$$A \ominus B = \{z|(B)_z \subseteq A\} \tag{3.31}$$

Of in duidelijkere taal, de erosie is de verzameling van alle punten z waarvoor geldt dat B na translatie met z nog steeds binnen A ligt. Je kan je het structurerend element B voorstellen als een gom die rond de randen van A gaat en al de delen van A waar hij over gaat uitwist.

Het resultaat van de erosie van een masker A hangt af van de vorm van het structurerend element, als geïllustreerd in figuur 3.8



Figuur 3.8: Twee voorbeeldjes van erosie, telkens met een ander structurerend element, het punt in het midden van de structurerende elementen B1 en B2 geeft hun oorsprong aan, de stippellijnen geven de originele vorm van A weer.

Erosie wordt vaak gebruikt om irrelevant detail te verwijderen, stel men wil alle kleine details uit een binair masker halen, een erosie met een structurerend element dat net iets kleiner is dan het kleinste detail dat je wil behouden zal ervoor zorgen dat alle details die kleiner dan het structurerend element zijn verwijderd worden uit het binair masker.

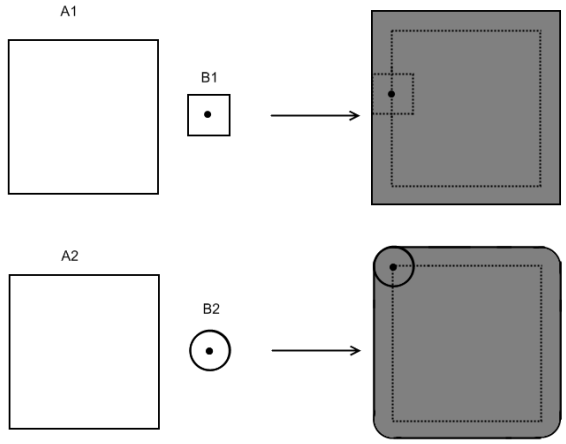
Dilatie

Dilatie op een binair masker A met structurerend element B wordt vervolgens gedefinieerd in formule (3.32) met $\hat{B}$ de reflectie van B rond zijn eigen oorsprong [13].

$$A \oplus B = \{z|(\hat{B})_z \cap A \neq \emptyset\} \tag{3.32}$$

Of nog: de dilatie van A door B is de verzameing van alle z zodat $\hat{B}$ en A overlappen in ten minste 1 element. Visueel komt dit neer op het structurerend element B rond de randen van A te bewegen en de grenzen van A te verleggen naar de nieuwe grenzen die B aftekent.

Twee voorbeeldjes van een dergelijke dilatie kan je vinden in figuur 3.9.



Figuur 3.9: Twee voorbeeldjes van dilatie, telkens met een ander structurerend element, het punt in het midden van de structurerende elementen B1 en B2 geeft hun oorsprong aan , de stippellijnen geven de originele vorm van A weer.

Dilatie wordt typisch gebruikt om kleine gaten in een binair masker te dichten. Stel je wil een geconnecteerde componenten analyse uitvoeren van je binair masker, met andere woorden, je wil zien welke componenten met elkaar verbonden zijn. Een probleem is echter dat ruis voor gaten in je masker gezorgd heeft waar er geen zouden mogen zijn. Een dilatie met een structurerend element van de juiste grootte, namelijk de helft van de grootte van de verwachte gaten in je masker zal ervoor zorgen dat de gaten verdwijnen.

3.5.2 Positie van bewegende delen

Om de positie en de grootte van een bewegend onderdeel te kunnen bepalen onderzoeken berekenen we voor elke regio geconnecteerde witte pixels hun midden (x_m, y_m) aan de hand

van de grootte N in pixels waarbij (x_i, y_i) pixels zijn in de huidige witte regio.

$$(x_m, y_m) = \frac{1}{N} \sum_{i=1}^{N} (x_i, y_i) \tag{3.33}$$

Naargelang de noden van het algoritme kunnen we nu het middelpunt (x_m, y_m) afronden naar de dichtstbijzijnde pixel of kunnen we in een float precisie laten staan.

3.5.3 Object-specifieke analyse

De volgende stap in de detectie is dikwijls zeer specifiek aan het object dat gedetecteerd moet worden. Dit is omdat verschillende objecten onvermijdelijk verschillende eigenschappen gaan hebben. Zo zal een algoritme dat een voetbal detecteert waarschijnlijk slechte resultaten geven als het toegepast wordt om een badminton pluimpje te detecteren. Verschillende objecten bewegen dikwijls op totaal verschillende manieren dus een andere aanpak is vaak vereist voor een ander object.

Meestal worden bepaalde eigenschappen van een object zoals zijn snelheid, grootte, afgelegde afstand of kleur gemodelleerd om te beslissen of een object al dan niet het gewenste object is om te detecteren.

Ren et al. [46] beschrijven de detectie van een voetbal aan de hand van zijn grootte, de kans $l_i \in [0, 1]$ dat een object b_i een bal is wordt hier gegeven als:

$$l_i = \frac{1}{2} + k_1 \frac{||v_i|| - ||\overline{v}||}{||\overline{v}||} + k_2 n_i \tag{3.34}$$

Met k_1 en k_2 parameters om de vooraf gedetecteerde snelheid en leeftijd van een traject mee in rekening te brengen, v_i de snelheid van de bal over zijn gedetecteerd traject en n_i de lengte van het traject. $\bar{v}$ is de gemiddelde snelheidsvector van alle gedetecteerde objecten en wordt gebruikt als een normalisatieterm.

Chen en Wang [5] bepalen of een gedetecteerd object een badmintonpluimpje is aan de hand van zijn traject gegevens. Een traject vector wordt bepaald door een viertal aan gegevens:

1. De coordinaten van het traject in het huidige beeld
2. De hoek tussen de X-as en de lijn tussen de twee laatste punten in het traject
3. De afstand tussen de twee laatste punten op het traject
4. De richting van het traject

Aan de hand van het vergelijken van deze gegegevens met het verwachte resultaat krijgt elk van deze parameters een score die aangeeft hoe goed hij de verwachte waarde benadert. Het uiteindelijke traject wordt dan gekozen als het traject dat de hoogste gezamenlijke score heeft maar alleen indien deze score boven een drempelwaarde ligt.

Zo heeft elk type object wel zijn eigenschappen en methodes die belangrijk zijn om detectie mogelijk te maken, het is aan de programmeur om het juiste model op te stellen voor het te detecteren object.

Hoofdstuk 4

Tracking

In het vorige hoofdstuk hebben we methodes besproken om aan detectie van objecten te doen, maar eenmaal deze objecten gevonden zijn is het nodig om hun bewegingen te volgen over het scherm, daarom dit hoofdstuk. De bedoeling van tracking is om een de bewegingen van een object te kunnen voorspellen zodat in de volgende stap van de detectie, het object binnen een bepaalde regio kan gezocht worden. In dit hoofdstuk zullen we enkele voorbeeldjes van tracking bespreken.

4.1 Discrete Kalman filter

De Kalman filter is een veel gebruikte methode om aan object tracking te doen. Origineel ontwikkeld door R.E. Kalman in 1960 [20] had de Kalman filter zijn eerste toepassing in navigatiesystemen van ruimteschepen. Een Kalman filter is een set van mathematische vergelijkeningen die op een efficiënte manier de staat van een proces kunnen voorspellen met een minimale fout zelfs als de precieze aard van het systeem ongekend is [66]. Om deze redenen werd het nut van de Kalman filter al snel ingezien in andere vakgebieden.

We gebruiken in deze sectie een notatie gebaseerd op de beschrijvingen van Welch en Bishop [66].

Een discrete Kalman filter probeert de staat van een discrete variabele $x \in \Re^n$ te voorspellen van een proces dat bepaald is door de staat vergelijking:

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \tag{4.1}$$

Met A, B matrices, x_k de staat van x op tijdstip k, w_k een term om de ruis in de voorspelling te kwantificeren en u_k een gekende invoer in het systeem. Stel we zijn de positie van een voertuig aan het volgen met een discrete Kalman filter. De staat x_k stelt dan de positie van

het voertuig voor in deze tijdstap, x_{k-1} is de positie van het voertuig bij de vorige observatie en de invoer u_k zijn de gekende invoeren in het systeem zoals bijvoorbeeld de bestuurder die gas geeft, remt of aan haar stuur draait.

De observatie variabele $z \in \Re^m$
wordt vervolgens gedefiniëerd als:

$$z_k = Hx_k + v_k \tag{4.2}$$

Hier is H een matrix en v_k een willekeurige term om ruis in de observatie te representeren. In equivalentie met het vorige voorbeeld is de observatie variabele z_k hier de echte staat van het voertuig x_k onder invloed van de ruisterm v_k, een term die observatieruis voorstelt door bijvoorbeeld onnauwkeurige metingen.

De willekeurige variabelen w_k en v_k zijn onafhankelijk van elkaar, hun waarden zijn normaal verdeeld met een gemiddelde van 0. Met andere woorden, de variabelen w_k en v_k zijn witte ruis. Met de matrices Q_k en R_k die respectievelijk de ruis van het proces en de ruis van de meting bepalen in een covariantie matrix.

$$p(w_k) \backsim N(0, Q_{k-1}) \tag{4.3}$$

$$p(v_k) \backsim N(0, R_k) \tag{4.4}$$

De staat vector x is een vector die de huidige staat van een systeem beschrijft, deze staat kan echter niet rechtstreeks uitgelezen worden omdat de observatie onder invloed staat van ruis. Daarom gebruiken we de observatie waarde z die onder invloed van de ruis waarde v staat om tot een benadering van x te komen.

We definiëren nu de *a priori* schatting $\hat{x}_k^- \in \Re^n$ op tijdstip k en de *a posteriori* schatting $\hat{x}_k \in \Re^n$. De *a priori* schatting is de berekende staat op tijdstip k met kennis van de vorige staat $k-1$. De *a posteriori* schatting is de berekende staat op basis van de observatiewaarde z_k.

Nu kunnen we de *a priori* en *a posteriori* fouten definieren als respectievelijk:

$$e_k^- \equiv x_k - \hat{x}_k^- \tag{4.5}$$

$$e_k \equiv x_k - \hat{x}_k \tag{4.6}$$

Dit maakt van respectievelijk de *a priori* en *a posteriori schattingsfout covariantie* matrices:

$$P_k^- = E[e_k^- e_k^{-T}] \tag{4.7}$$

$$P_k = E[e_k e_k^T] \tag{4.8}$$

Hier is $E[i]$ de verwachte waarde van een variabele i. De matrices P_k^- en P_k zullen worden gebruikt om de fout in zowel de voorspelling als de observatie te minimaliseren.

De *a posteriori* schatting wordt nu bepaald doormiddel van de *a priori* waarde en een gewogen verschil van de eigenlijke waargenomen waarde z_k en de voorspelde waarde $H\hat{x}_k^-$.

$$\hat{x}_k = \hat{x}_k^- + K(z_k - H\hat{x}_k^-) \tag{4.9}$$

De term $(z_k - H\hat{x}_k^-)$ noemen we het *observatie residu*. Indien dit 0 is komt de geöbserveerde waarde overeen met de voorspelde, dit is natuurlijk het ideale scenario omdat dit wil zeggen dat de Kalman filter perfect werkt voor het systeem. De matrix K in vergelijking (4.9) noemen we de Kalman gain matrix, deze $n \times m$ matrix wordt gekozen om P_k als voorgesteld in (4.8) te minimaliseren. Een van de meest populaire manieren om K te berekenen is gegeven in (4.10).

$$K_k = \frac{P_k^- H^T}{H P_k^- H^T + R} \tag{4.10}$$

Als we kijken naar (4.10) zien we dat naarmate de waarde van de *observatiefout covariantie* R dichter bij de 0 ligt, de waarde van K omhoog gaat, en dat het *observatie residu* dus een groter gewicht krijgt in (4.9). Anders als de *a priori schattingsfout covariantie* P_k^- de 0 nadert wordt het *observatie residu* minder gewogen. Meer specifiek [66]:

$$\lim_{R_k \to 0} K_k = H^{-1} \tag{4.11}$$

$$\lim_{P_k^- \to 0} K_k = 0 \tag{4.12}$$

Een gevolg hiervan is dat als de *a posteriori* observatiefout klein genoeg wordt de Kalman filter de observatiewaarde meer en meer zal vertrouwen, als aan de andere kant de *a priori* voorspellingsfout klein genoeg is zal de voorspelling meer en meer vertrouwd worden.

4.1.1 Algoritme

Om een Kalman filter te gebruiken in een algoritme zijn twee stappen nodig, een *a priori voorspelling* en een *a posteriori correctie*.

Het algoritme wordt beschreven als volgt:

1. Initialiseer x_0 en P_0
2. Voorspel de volgende staat $\hat{x}_k^-$ aan de hand van de vorige $\hat{x}_{k-1}$
3. Corrigeer de schatting $\hat{x}_k^-$ aan de hand van de geobserveerde waarde z_k
4. Incrementeer k en ga terug naar (2)

Voorspelling

De voorspellingsfase van het discrete Kalman filter algoritme gebruikt waarden uit de vorige stap om een voorspelling te doen van de huidige stap. De formules die moeten uitgerekend worden in de voorspellingsfase zijn:

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1} \tag{4.13}$$
$$P_k^- = AP_{k-1}A^T + Q_{k-1} \tag{4.14}$$

De matrices A en B zijn gedefiniëerd in (4.1) en zijn afhankelijk van het systeem dat voorspeld wordt terwijl Q_{k-1} werd gedefiniëerd in (4.3)

Correctie

De correctie gebeurt als de nieuwe observatiewaarde binnen is, namelijk z_k. De waarde uit de observatie wordt gebruikt om de *a priori* schatting bij te werken met de *a posteriori* informatie.

De eerste stap in de correctiefase is het berekenen van de Kalman gain matrix als gegeven in (4.10) maar hier herhaald om duidelijkheidsredenen.

$$K_k = \frac{P_k^- H^T}{HP_k^- H^T + R_k} \tag{4.15}$$

Vervolgens wordt de *a posteriori* schatting gedaan aan de hand van (4.9), die ook hier herhaald wordt.

$$\hat{x}_k = \hat{x}_k^- + K(z_k - H\hat{x}_k^-) \tag{4.16}$$

De laatste stap bestaat er nu in om de *a posteriori schattingsfout covariantie* te berekenen voor stap k.

$$P_k = (I - K_k H)P_k^- \tag{4.17}$$

4.1.2 Voorbeeld

Een voorbeeld van het gebruik van een Kalman filter is het voorspellen van een vrachtwagen die op een eendimensionale weg voortbeweegt [17], de vrachtwagen begint op de weg op po-

sitie 0 en wordt voortgedreven door willekeurige accelleratie. Om de zoveel tijd meten we de positie van de vrachtwagen. Het probleem met deze metingen is dat ze niet nauwkeurig zijn, er is met andere woorden een ruis effect op de metingen. De opgave van de Kalman filter wordt nu om telkens de positie en de snelheid van de vrachtwagen te voorspellen.

De staat van de vrachtwagen op tijdstip k, x_k ,met positie pos_k en snelheid vel_k wordt dan :

$$x_k = \begin{bmatrix} pos_k \\ vel_k \end{bmatrix} \tag{4.18}$$

De vrachtwagen ondervindt een constante accelleratie a_k. We geven eerst ter illustratie de formules van Newton voor een model met constante accelleratie.

$$x_t = x_0 + v_0 t + \frac{1}{2} a t^2 \tag{4.19}$$

Met x_0 een positie aan het begin van de meting op tijdstip 0, t de tijd, v_0 de snelheid aan het begin van de meting en a de accelleratie.

Als we dit vertalen dan naar een Kalman filter implementatie bekomen we:

$$x_k = Ax_{k-1} + Ba_k \tag{4.20}$$

waarbij A en B gegeven als:

$$A = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \tag{4.21}$$

$$B = \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix} \tag{4.22}$$

Waarbij Q de variantieσ_a^2 van a over de tijd modelleert, we laten hier het subscript bij Q achterwege aangezien deze constant blijft over heel het model :

$$Q = \sigma_a^2 B B^T \tag{4.23}$$

De observatie z_k onder invloed van ruis wordt dan:

$$z_k = Hx_{k-1} + v_k \tag{4.24}$$

met

$$H = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{4.25}$$

We nemen ook aan dat R constant is dus we laten ook hier het subscript achterwege:

$$R = \sigma_z^2 \tag{4.26}$$

4.2 Uitgebreide Kalman filter

Een beperking van de gewone Kalman filter is dat hij veronderstelt dat het systeem kan worden voorgesteld als een lineaire vergelijking. Een niet-lineair systeem zou mogelijk door een gewone Kalman filter niet op een voldoende accurate manier kunnen worden voorgesteld.

Daarom introduceren we hier de uitgebreide Kalman filter gebruik makende van de notatie van Welch en Bishop [66], een filter die gebruik maakt van linearisatie rond een huidig gemiddelde en covariantie. We definieren net als in (4.1) een staat vector $x \in \Re^n$ maar in tegenstelling tot (4.1) gebruiken we hier een niet-lineaire stochastische vergelijking.

$$x_k = f(x_{k-1}, u_{k-1}, w_{k-1}) \tag{4.27}$$

Waarbij u_k en w_k nog steeds respectievelijk de invoerterm en de ruis term zijn op tijdstip k. De observatieterm $z \in \Re^m$ wordt dan gegeven door:

$$z_k = h(x_k, v_k) \tag{4.28}$$

Met v_k nog steeds een ruisterm. Zowel f als h stellen in dit geval niet-lineaire functies voor.

Omdat de ruis termen vaak niet gekend zijn is het niet praktisch om ze te gebruiken in een methode. Een benadering voor de x en z termen kan berekend worden door aan te nemen dat er geen ruis op het systeem zit:

$$\tilde{x}_k = f(\hat{x}_{k-1}, u_{k-1}, 0) \tag{4.29}$$

$$\tilde{z}_k = h(\tilde{x}_k, 0) \tag{4.30}$$

We kunnen nu een afleiding geven van een schatting voor de staat x_k.

$$x_k \approx \tilde{x}_k + A_k(x_{k-1} - \hat{x}_{k-1}) + W_k w_{k-1} \tag{4.31}$$

Waarbij $\tilde{x}_k$ de benaderende waarde voor x_k is zonder de ruisterm, gegeven in (4.29), $\hat{x}_{k-1}$ een *a posteriori* schatting is van de staat in de vorige stap en w_k een lineaire ruisterm voorstelt.

A_k en W_k zijn de Jacobianen van gedeeltelijke afgeleiden van f in respectievelijk de termen x

en w en worden aldus weergegeven als :

$$A_k(i,j) = \frac{\partial f_i}{\partial x_j}(\hat{x}_{k-1}, u_{k-1}, 0) \tag{4.32}$$

$$W_k(i,j) = \frac{\partial f_i}{\partial w_j}(\hat{x}_{k-1}, u_{k-1}, 0) \tag{4.33}$$

De observatievector z_k wordt dan gegeven door:

$$z_k \approx \tilde{z}_k + H_k(x_k - \hat{x}_k) + V_k v_k \tag{4.34}$$

Hier is $\tilde{z}_k$ als gegeven in (4.30) en v_k een ruis term. Jacobianen H_k en V_k van partiele afgeleiden in respectievelijk x en v worden gedefinieerd als:

$$H_k(i,j) = \frac{\partial h_i}{\partial x_j}(\tilde{x}_k) \tag{4.35}$$

$$V_k(i,j) = \frac{\partial h_i}{\partial v_j}(\tilde{x}_k) \tag{4.36}$$

Vervolgens kunnen we de schattingsfout $\tilde{e}_{x_k}$ en het observatieresidu e_{z_k} definieren als:

$$\tilde{e}_{x_k} \equiv x_k - \tilde{x}_k \tag{4.37}$$

$$\tilde{e}_{z_k} \equiv z_k - \tilde{z}_k \tag{4.38}$$

Waarbij de uiteindelijke waarden van het *foutproces* gegeven worden door:

$$\tilde{e}_{x_k} \approx A_{k-1}(x_{k-1} - \hat{x}_{k-1}) + \epsilon_k \tag{4.39}$$

$$\tilde{e}_{z_k} \approx H_k \tilde{e}_{x_k} + \eta_k \tag{4.40}$$

Hier zijn ϵ_k en η_k onafhankelijke variabelen met een mediaan van 0 en covariantie matrices $W_k Q_{k-1} W_k^T$ en $V_k R_k V_k^T$ met Q en R zoals in (4.3) en (4.4). Of anders:

$$p(\epsilon_k) \backsim N(0, W_k Q_{k-1} W_k^T) \tag{4.41}$$

$$p(\eta_k) \backsim N(0, V_k R_k V_k^T) \tag{4.42}$$

De gelijkenis tussen vergelijkingen (4.39) (4.40) van het *foutproces* en de originele vergelijkingen van de discrete Kalman filter (4.1) (4.2) zijn opvallend. Deze overeenkomst laat ons toe om het *observatieresidu* $\tilde{e}_{z_k}$ te gebruiken als een tweede Kalman filter om de *schattingsfout* $\tilde{e}_{x_k}$ te voorspellen. We noemen deze schatting $\hat{e}_k$ en gebruiken deze om de *a posteriori* staat $\hat{x}_k$ te bepalen.

$$\hat{x}_k = \tilde{x}_k + \hat{e}_k \tag{4.43}$$

De vergelijking voor $\hat{e}_k$ wordt dan:

$$\hat{e}_k = K_k \tilde{e}_{z_k} \tag{4.44}$$

En we komen dus bij:

$$\begin{aligned} \hat{x}_k &= \tilde{x}_k + K_k \tilde{e}_{z_k} \\ &= \tilde{x}_k + K_k(z_k - \tilde{z}_k) \end{aligned} \tag{4.45}$$

4.2.1 Algoritme

De volledige set van vergelijkingen van de uitgebreide Kalman filter kan vervolgens op een equivalente manier als de discrete filter in een algoritme gegoten worden, doormiddel van een voorspelling een een correctie fase.

Voorspelling

De benodigde berekeningen in de voorspellingsfase zijn dan:

$$\hat{x}_k^- = f(\hat{x}_{k-1}, u_{k-1}, 0) \tag{4.46}$$

$$P_k^- = A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T \tag{4.47}$$

Correctie

En de correctie fase wordt:

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1} \tag{4.48}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, 0)) \tag{4.49}$$

$$P_k = (I - K_k H_k) P_k^- \tag{4.50}$$

4.3 Particle filter

Een particle filter heeft als doel een reeks parameters X_k te schatten met alleen de invoer waarden Y_k. We definiëren nu de densiteiten als gedefiniëerd door Li en Zhang [25]:

$$p(X_k|X_{k-1}) \tag{4.51}$$

$$p(Y_k|X_k) \tag{4.52}$$

Waarbij de *a priori* densiteit $p(X_k|X_{k-1})$ een Markov proces weergeeft. Een Markov proces is een model voor willekeurige evolutie van een systeem zonder geheugen, met andere woorden een systeem waarbij de toekomst en het verleden onafhankelijk van elkaar zijn.

De bedoeling is om op een recursieve manier tot een schatting van de *a posteriori densiteit* $p(X_k|Y_{1:k})$ en een integraal functie $f_k(X_k)$ te komen. De *a posteriori densiteit* wordt net als de Kalman filter in 2 stappen geschat, een voorspelling en een correctie.

De voorspelling projecteert de *a posteriori densiteit* $p(X_{k-1}|Y_{1:k-1})$ een stap vooruit door gebruik te maken van de de *a priori densiteit* $p(X_k|X_{k-1})$ op de volgende manier:

$$p(X_k|Y_{1:k-1}) = \int p(X_k|X_{k-1})p(X_{k-1}|Y_{1:k-1})dX_{k-1} \tag{4.53}$$

En de correctie gebruikt de regel van Bayes om nieuwe observaties te integreren in de schatting als volgt:

$$p(X_k|Y_{1:k}) = \frac{p(Y_k|X_k)p(X_k|Y_{1:k-1})}{p(Y_k|Y_{1:k-1})} \tag{4.54}$$

Met de verwachting:

$$E_p(f_k(X_k)) = \int f_k(X_k)p(X_k|Y_{1:k})dX_k \tag{4.55}$$

Omdat de voorspelling (4.53) en correctie (4.54) vaak een hoogdimensionale integraal beschrijven is het ongewenst om deze analytisch te gaan berekenen. Deze formules zouden kunnen aangepast worden om met de Kalman filter geschat te worden maar omdat de Kalman filter lineaire schattingen uitvoert zal dit enkel accuraat zijn in voor een lineair model, een veel betere manier is om Monte Carlo methodes toe te passen, deze kunnen tot een veel nauwkeuriger resultaat komen door een hoogdimensionale integreaal te benaderen door een reeks van steekproeven.

Rechtstreeks waarden bepalen van de *a posteriori densiteit* $p(X_{k-1}|Y_{1:k-1})$ is echter niet mogelijk, daarom wordt er een *propositie densiteit* gedefiniëerd $\pi(X_k|Y_{1:k})$ die de meest recente observaties gebruikt om tot een schatting van de *a posteriori densiteit* te komen, deze is vergelijkbaar met de *a posteriori densiteit* maar kan makkelijker gebruikt worden om Monte Carlo Integratie op toe te passen:

$$\pi(X_k|Y_{1:k}) = \pi(X_k|X_{k-1}, y1:k)\pi(X_{k-1}|Y_{1:k-1}) \tag{4.56}$$

De *a posteriori* densiteit (4.55) kan dus benaderd worden door de verwachting van $f_k(X_k)$:

$$E_p(f_k(X_k)) = \frac{E_\pi(\omega_k(X_k))}{E_\pi(\omega_k)} \tag{4.57}$$

waarbij

$$\omega_k = \omega_{k-1}\frac{p(Y_k|X_k)p(X_k|X_{k-1})}{\pi(X_k|X_{k-1}, Y_{1:k})} \tag{4.58}$$

4.3.1 Monte Carlo simulatie

De bedoeling is nu om de *a posteriori* densiteit te gaan schatten doormiddel van Monte Carlo integratie. We gebruiken een set van particles $(X^i_{k-1}, \omega^i_{k-1})$ die we op willekeurige plaatsen uit de *propositie verdeling* $\pi(X^i_k|X^i_{k-1}, Y_{1:k})$ halen. We bekomen dan de *a posteriori* densiteit $\{(X^i_k, \omega^i_k)|i \in [1, N]\}$.

Het algoritme werkt als volgt [25]:

1. Initialisatie:
   Haal een set van N particles (X^i_0, ω^i_0) uit de *a priori* staat $p(X_0)$
2. Steekproef:
   (a) Voor $i = 1 \rightarrow N$
   Haal een waarde voor X^i_k uit de propositie distributie $\pi(X^i_k|X^i_{k-1}, Y_{1:k})$ als in (4.56)
   Bereken de nieuwe gewichten aan de hand van (4.58)

$$\omega^i_k = \omega^i_{k-1} \frac{p(Y_k|X^i_k)p(X^i_k|X^i_{k-1})}{\pi(X^i_k|X^i_{k-1}, Y_{1:k})}$$

   (b) Voor $i = 1 \rightarrow N$
   Normaliseer de gewichten

$$\omega^i_k = \frac{\omega^i_k}{\sum_{j=1}^{N} \omega^j_k}$$

3. Uitvoer:
   Bereken een set van particles (X^i_k, ω^i_k) die we kunnen gebruiken als een schatting voor de *a posteriori* distributie:

$$p(X_k|Y_{1:k}) \approx \sum_{j=1}^{N} \omega^j_k \delta(X_k = X^j_k)$$

   Waarbij $\delta(*)$ de Dirac functie is.
   Bereken ook de verwachting:

$$E_p(f_k(X_k)) \approx \sum_{j=1}^{N} \omega^j_k f_k(X^j_k)$$

4. Selectie:
   Herhaal de steekproef op particles X^i_k met gewicht ω^i_k om N onafhankelijke willekeurige particles te detecteren die ongeveer verdeeld zijn als in $p(X_k|Y_{1:k})$
   Zet al de gewichten ω^i_k op $\frac{1}{N}$

5. Incrementeer k, ga naar stap 2

Hoofdstuk 5

Implementatie

De implementatie deze thesis is gebeurd in Visual Studio 2005 op Windows XP, het programmeren voor de GPU gebeurde doormiddel van CUDA 1.1 [38]. Intel OpenCV [16] samen met Microsoft DirectShow [31] worden gebruikt om de gebruikte beelden weer te geven. Een temporele wavelet transformatie, volledig werkend op de GPU wordt gebruikt voor de real-time detectie van kleine, snel bewegende objecten alleen gebruik makende van de beweging van de bal.

5.1 Grijswaarden

Voor we aan de wavelet transformatie kunnen beginnen moeten we eerst het originele beeld omzetten naar grijswaarden. Dit is een typische applicatie waarvoor de GPU zich perfect leent, we moeten namelijk dezelfde instructies, namelijk $Y = 0.3R + 0.59G + 0.11B$, herhalen voor elke pixel. Dit deel werd dan ook geïmplementeerd in CUDA. Deze opgave leende zich zeer goed om met CUDA te beginnen wegens zijn lage moeilijkheidsgraad.

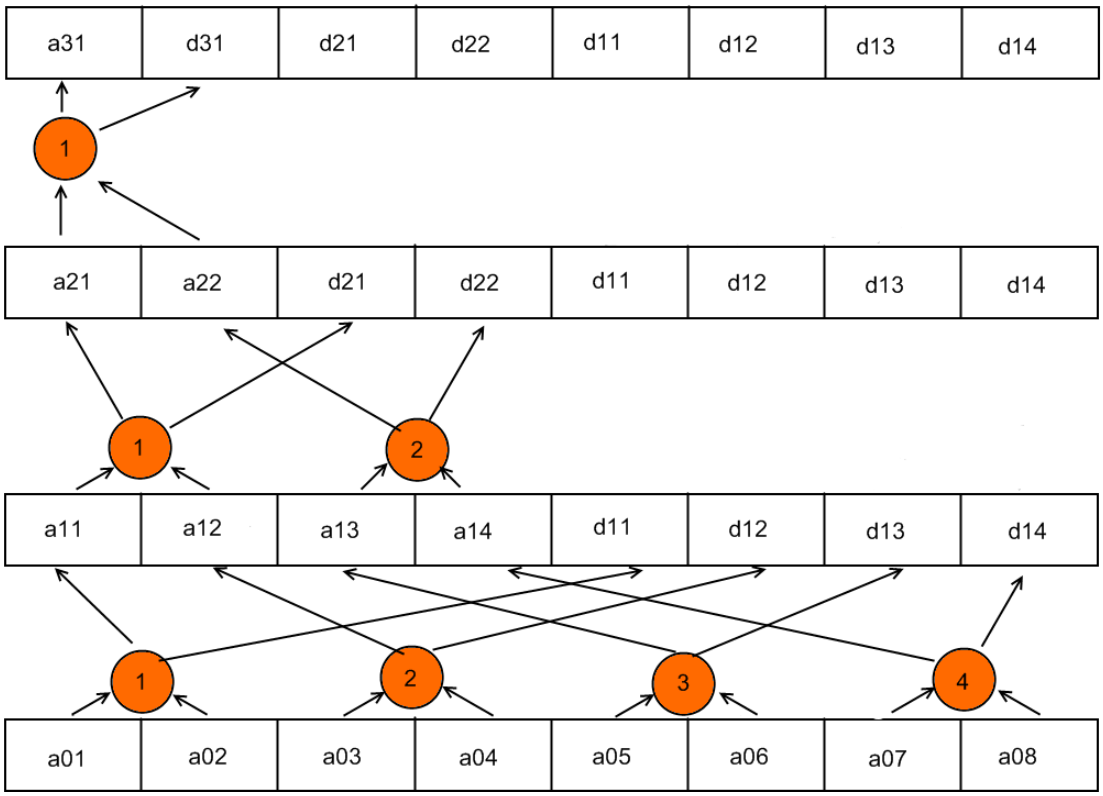
5.2 Wavelet transformatie

Nu we het beeld in grijswaarden hebben kunnen we beginnen aan de wavelet tranformatie. Als invoer krijgen we 8 opeenvolgende afbeeldingen in 1 grote buffer. De beelden worden in volgorde toegevoegd aan de buffer, die circulair gemaakt is om efficiëntie redenen en aan de wavelet transformatie methode doorgegeven.

Deze circulaire datastructuur zorgt voor problemen met betrekking tot *coalesced* geheugentoegang omdat de in te lezen pixels telkens de afmetingen van een beeld uit elkaar liggen in het geheugen. Bovendien is de in te lezen data van het type char, een type dat niet van snelle overdracht kan genieten die bijvoorbeeld floats wel hebben. Het was echter een afweging die moest gemaakt worden, een datastructuur gebruiken die wel coalesced ingelezen kan worden

zou de CPU zwaar belasten zodat al de snelheidswinst die hieruit gehaald wordt op de GPU teniet gedaan wordt door het verlies aan snelheid aan de CPU kant. Uiteindelijk werd er voor gekozen om het performantieverlies aan de GPU kant te laten zoals het is omdat de optimalisatie potentieel veel meer snelheidsverlies aan de CPU kant kan optekenen dan het wint aan de GPU kant.

De wavelet transformatie methode is gemaakt om 4 threads, oftewel n/2 threads per wavelet te gebruiken om de wavelet transformatie te berekenen. Elke kernel leest 2 waarden in in het geheugen op een manier die bank conflicten vermijdt doormiddel van sequentiële addressering. Deze twee waarden worden vervolgens omgezet naar floating point precisie en worden genormaliseerd. Een char waarde is namelijk niet geschikt voor een wavelet transformatie wegens te onnauwkeurig en daarenboven kunnen float waarden op een snelle manier van en naar het globale geheugen gekopiëerd worden.



Figuur 5.1: De schematische voorstelling van de berekening van een wavelet vector van 8 lang. Van onder naar boven, a_{ij} en d_{ij} zijn respectievelijk de approximatie- en detailcoëfficiënten op niveau i plaats j.

Figuur 5.1 geeft schematisch de stappen weer om tot de uiteindelijke wavelet transformatie te komen. We gaan van onder naar boven en beginnen met 4 threads die telkens 2 opeen-

volgende waarden uit het gedeeld geheugen inlezen. Deze figuur is niet helemaal correct aangezien de detailcoëfficiënten rechtstreeks naar het tragere globaal geheugen geschreven worden. Detailcoëfficiënten zijn niet meer nodig voor verdere berekeningen dus het schrijven van detailcoëfficiënten naar het gedeeld geheugen zou een extra kopieeroperatie vereisen om de data later naar het globaal geheugen over te zetten hetgeen natuurlijk ongewenst is. Voor de duidelijkheid werden de waarden in dit schema echternaar het gedeeld geheugen geschreven.

Na elke decompositie stap wordt het aantal threads gehalveerd en herhalen we de berekeningen voor de volgende stap. Omdat we de approximatiewaarde a_{31} niet nodig hebben wordt deze ook niet bijgehouden.

Als uitvoer hebben we nu 7 opeenvolgende beelden waarbij het eerste beeld de wavelet informatie van al de 8 beelden bevat.

5.3 Analyse

Een eerste poging om aan analyse van deze wavelet informatie te doen kwam uit [67], een statistische analyse van de informatie van al de pixels zou een antwoord moeten geven op de vraag of een pixel al dan niet een beweging detecteerde.

De invoer voor deze methode was de eerste van de 7 opeenvolgende beelden die we uit de vorige stap kregen en de uitvoer was een binair masker in de vorm van een char array waarbij zwart de achtergrond voorstelt en wit de voorgrond oftewel de bewegende delen van de video. De formule zoals beschreven in (3.29) bestaat uit het analyseren van een venster rond elke pixel en de bekomen waarde te vergelijken met een waarde uit een statistische tabel. Het venster moest groot genoeg zijn om al de bewegingen te detecteren dus ik schatte de maximale grootte van een beweging van een balletje manueel in door de videobeelden te bestuderen.

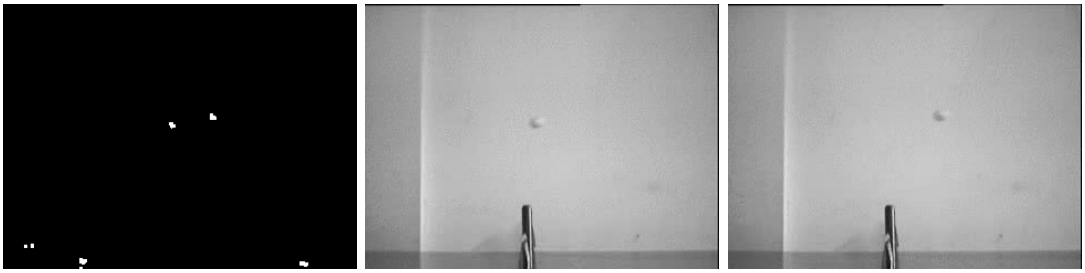
Het werk van een kernel voor deze methode bestaat erin om al de pixels in de buurt van de te onderzoeken pixel na te gaan. Er moesten speciale voorzorgen genomen worden om te zien dat een venster niet buiten de randen van het beeld zou bewegen. Deze kernel maakte geen gebruik van gedeeld geheugen, elke kernem moest namelijk alleen data voor zichzelf berekenen.

Het werd echter snel duidelijk dat deze methode niet geschikt zou zijn voor onze doeleinden. Eerst en vooral was het overduidelijk dat deze methode ongeschikt was voor real-time berekeningen. Een doelstelling vooraf was de analyse in real-time doen, maar zelfs na strenge optimalisaties kwam deze methode niet verder dan ongeveer 2 of 3 frames per seconde, met andere woorden, ruim 10 maal te traag om nog real-time genoemd te kunnen worden. Het grootste probleem van de snelheid was de grootte van het venster, hoe kleiner het venster

waarmee we rekenen hoe sneller de berekeningen gaan. Een kleiner venster was helaas geen optie, een kleiner venster wil zeggen dat grote bewegingen, dus snel bewegende ballen niet gedetecteerd worden als een beweging.

Een tweede probleem was het feit dat bij het testen van de wavelet analyse op een eerder moment al betrouwbaardere maskers verkregen door simpelweg de wavelet informatie te vergelijken met een drempelwaarde. Er werd besloten om deze vorm van analyse achterwege te laten en te beginnen met een vorm van analyse die meer aansloot bij Davies et al. [6].

Een drempelwaarde analyse werd toegepast en het typische signaal dat een bewegend balletje achterlaat op de maskers werd gebruikt als basis om tot detectie te komen. Als getoond in figuur 5.2 laten de balletjes een kenmerkend pad na op een wavelet transformatie, hier gedemonstreerd op een wavelet transformatie van het allerhoofste niveau. Merk op dat figuur 5.2 equivalent is aan een gewone differentiatie tussen 2 beelden aangezien het de wavelet transformatie op het allerhoogste niveau is. Meer bepaald de wavelet transformatie tussen het huidige en het vorige beeld.



Figuur 5.2: Een binair masker bekomen door de wavelet transformatie op het hoogste niveau, dwz. tussen de 2 laatste beelden, te vergelijken met een drempelwaarde. Links het masker, midden en rechts de originele beelden waarop het masker gebaseerd is.

De drempelwaarde analyse wordt eveneens volledig op de GPU uitgevoerd, de detailcoëfficiënten van elk wavelet beeld worden vergeleken met een drempelwaarde en aan de hand van deze drempelwaarde op zwart of wit gezet. We bekomen dus een binair masker waarbij de witte delen een beweging en de zwarte pixels de statische achtergrond voorstellen.

5.4 Opening

De bekomen maskers uit de vorige sectie worden wat opgepoetst doormiddel van een opening. We bekomen een opening door simpelweg eerst een erosie toe te passen op het binaire masker en vervolgens een dilatie op het resultaat hiervan.

Het structurerend element is een vierkant van grootte $N \times N$ gevuld met 1 waarden.

5.4.1 Erosie

Een CUDA kernel wordt gestart voor elke pixel in het te eroderen beeld. De pseudocode voor de kernel die pixel p berekent op masker m wordt gegeven als volgt:

1. Voor alle $i = 1 \rightarrow N$
  2. Voor alle $j = 1 \rightarrow N$
    3. Als alle $m[i - (N-1)/2][j - (N-1)/2] == 255$ dan $p = 255$
    4. Anders: $p = 0$

Met andere woorden als er 1 pixel van het structererend element dat over de pixel gelegd wordt een achtergrondpixel bevat wordt de pixel een achtergrond pixel, indien dit niet zo is, indien al de elementen voorgrond pixels zijn wordt de pixel een voorgrond pixel.

5.4.2 Dilatie

De CUDA kernel voor de dilatie wordt eveneens voor elke pixel p opgestart en de pseudocode hiervoor wordt gegeven door:

1. Voor alle $i = 1 \rightarrow N$
  2. Voor alle $j = 1 \rightarrow N$
    3. Als alle $m[i - (N-1)/2][j - (N-1)/2] == 0$ dan $p = 0$
    4. Anders: $p = 255$

Deze werkt op een methode die net andersom gaat dan bij de erosie, als er 1 voorgrond pixel wordt gedefinieerd wordt de pixel een voorgrond pixel, indien er geen gevonden wordt blijft de pixel een achtergrond pixel.

5.5 Detectie

We hebben nu onze opgepoetste maskers uit de vorige stap en we detecteren elke mogelijke kandidaat voor een balletje door op het allerlaagste niveau naar het dubbele voorkomen van het balletje te zoeken. Maar voor we dit kunnen doen moeten we eerst elke witte regio kunnen detecteren als een lijst van aaneensluitende witte pixels.

5.5.1 Geconnecteerde componenten

We voeren de analyse van de geconnecteerde componenten uit door op een masker het volgende algoritme toe te passen. Naast het masker definieren we een array van dezelfde grootte.

De array A bestaat uit allemaal pointers naar lijsten van coordinaten. De bedoeling is om voor elk element een lijst van zijn geconnecteerde componenten te gaan bijhouden.

We doen dit door rij per rij het masker m af te gaan en voor elke voorgrond pixel p een eventuele lijst bij te werken. Pseudocode wordt hieronder gegeven:

1. Initialiseer A en zet al de pointers op 0
2. Voor alle $i = 1 \rightarrow N$
  3. Voor alle $j = 1 \rightarrow N$
    4. Als $m[i][j] == 255$ dan $p = A[i][j]$
      5. Als alleen de linkerbuur een voorgrond pixel is :
      Voeg p toe aan de lijst van zijn linkerbuur en voeg de lijst toe aan p
      6. Als alleen de bovenbuur een voorgrond pixel is :
      Voeg p toe aan de lijst van zijn bovenbuur en voeg de lijst toe aan p
      7. Als beide buren een voorgrond pixel zijn :
      Voeg beide lijsten samen en voeg de lijst toe aan p
      8. Als p geen voorgrond buren heeft: initialiseer een nieuwe lijst voor p

We verkrijgen zo een array A met een lijsten in. Het enige dat ons nu rest is om elke unieke lijst uit A bij te houden en hiervoor de middenpunten te berekenen. We hebben dan een lijst van kandidaten van bewegende pixels.

5.5.2 Traject detectie

We passen de geconnecteerde componenten analyse uit de vorige sectie toe op de meest oppervlakkige van de 7 wavelet transformatie maskers die we hebben. Diegene die de waardes van de laatste 2 beelden bevat. Als resultaat krijgen we een lijst van coordinaten van bewegende pixels. Hierin zoeken we naar de typische dubbele voorkomens van een balletje die we eerder observeerden. We doen dit door elk paar naburige posities apart op te slaan in een lijst en we sorteren de lijst op hun onderlinge afstand, de langste eerst.

Tijdens experimentatie is er ondervonden dat de snelste objecten, met andere woorden de objecten met het meeste beweging tussen beelden in, meestal de balletjes zijn. Daarom sorteren we de lijst met buren op basis van afstand zodat we meteen het meest waarschijnlijke paar kunnen selecteren.

We herhalen vervolgens de detectie van de posities van de witte regionen maar deze keer op de eerste afbeelding in de reeks wavelets. Diegene die die detailcoëfficiënten van 8 beelden bevat met andere woorden, zie figuur 5.3 en let telken op de 8 duidelijke voorkomens van een

balletje. We schatten met het gedetecteerde paar als basis, in beide richtingen en met een lineaire voorspelling de regio waar zich normaal gezien nog een bal zou moeten bevinden. Indien zich daar een kandidaat bevindt blijven we in die richting zoeken tot we het pad hebben met maximaal 8 balletjes.

Figuur 5.3: Enkele voorbeelden van binaire maskers bekomen door een volledige wavelet transformatie, let op de 8 duidelijke voorkomens van het balletje in de maskers.

5.6 Tracking

Eenmaal we een pad hebben kunnen we beginnen aan het volgen van de bewegingen van het balletje. We detecteren voor elke nieuwe wavelet transformatie telkens de nieuwe punten en we vergelijken die met de voorspelde positie om te bepalen of het het balletje voorstelt of niet.

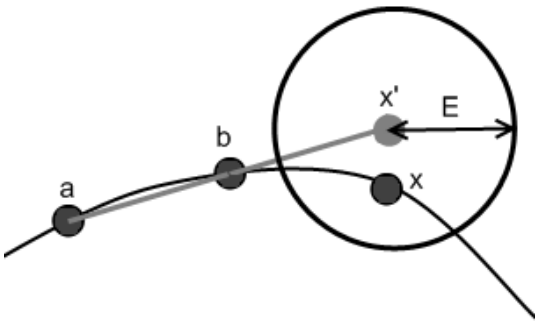
Twee methoden werden gebruikt, een zeer rudimentaire lineaire tracking en een meer robuuste Kalman filter.

5.6.1 Lineaire tracking

Deze methode veronderstelt dat een balletje zich lineair zal bewegen ten opzichte van zijn vorige 2 posities. Wat deze methode doet is de positie van het balletje voorspellen door aan te nemen dat een balletje dezelfde beweging zal doen als hij in het vorige pad gedaan heeft.

Dit benadert de baan van het balletje op een lineaire manier en is niet erg accuraat, daarom moeten we een E foutwaarde bepalen die groot genoeg is om de fout te kunnen opvangen.

Het algoritme werkt door een vector van de twee laatste waarden op het pad $\overrightarrow{ab}$ op te tellen bij de laatste waarde van het pad. We bekomen nu een schatting x' voor de te detecteren waarde x. We zoeken vervolgens in de buurt van x' met een straal E naar kandidaten voor de bal. De dichtstbijzijnde kandidaat wordt gekozen als uiteindelijke waarde. Voor een grafische voorstelling van de methode zie naar figuur 5.4 .



Figuur 5.4: Een voorstelling van de gebruikte lineaire tracking, een bal x wordt benaderd door x' doormiddel van een lineaire predictie gebruik makend van de twee laatst gekende waardes voor de bal a en b.

5.6.2 Kalman filter

Een tweede manier van tracking werd geïmplementeerd omdat het al snel duidelijk was dat de lineaire tracking maar een beperkt nut zou hebben. Er werd gekozen voor een Kalman filter. De Kalman filter implementatie in OpenCV werd gebruikt om dit te realiseren.

Equivalent aan de lineaire tracking wordt de Kalman filter gebruikt om gegeven het pad als invoer een nieuwe waarde te schatten die we dan kunnen gebruiken als startpunt voor een zoektocht naar het eigenlijke voorkomen van de bal.

5.6.3 Virtuele posities

Indien een het algoritme tijdens de tracking niet erin slaagt om het balletje te vinden wordt de geschatte positie als een virtuele positie toegevoegd. Een balletje verdwijnt immers niet zomaar. Het probleem ligt hierbij meestal bij het feit dat er zich een obstructie tussen het balletje en de camera bevindt of dat het balletje beweegt door een regio waarvan de textuur quasi identiek is aan die van het balletje.

De virtuele positie wordt bij het volgende beeld als basis gebruikt voor het vinden van de volgende positie en als hiermee wel een balletje wordt gevonden voegen we de virtuele positie toe aan het traject. Indien er geen balletje wordt gevonden na deze stap veronderstellen we dat het balletje verdwenen is en beginnen we terug aan de detectiefase.

5.6.4 Randvoorwaarden

Om de tracking robuuster te maken gebruiken we bepaalde randvoorwaarden om bijvoorbeeld te bepalen of een balletje al dan niet gemaneuvreerd heeft en bepalen we of een balletje of een balletje buiten het scherm is beland.

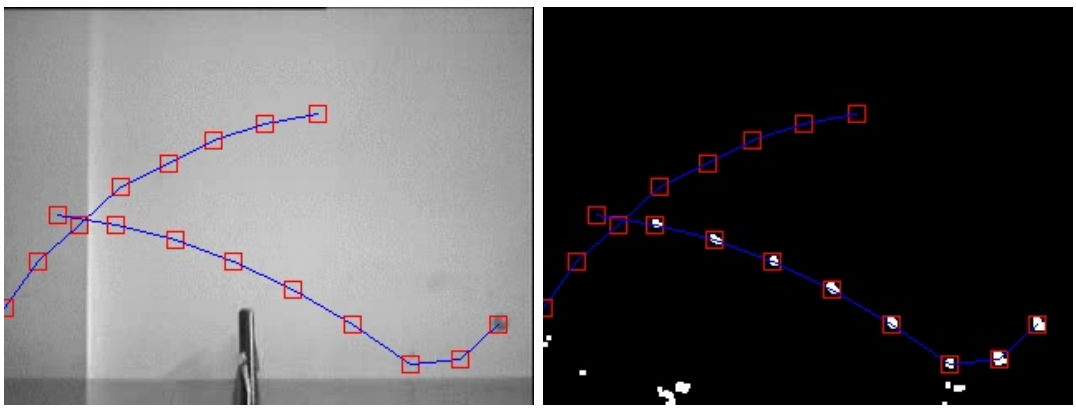
Indien een bal zich buiten het beeld begeeft zetten we al de waardes om te tracken weer op hun initiele positie en beginnen we weer aan een detectie fase.

Indien de bal gebotst heeft berekenen we eerst het waarschijnlijke punt van de botsing, we berekenen dan de waarschijnlijke positie van het contact van het balletje met de grond en doormiddel van dit punt berekenen we de meest waarschijnlijke positie van het balletje. Omdat deze berekeningen maar zeer ruwe schattingen zijn zoeken we in het geval van een maneuvrerend object in een grotere straal rond de voorspelde positie.

Na het detecteren van het gebotste balletje herstarten we het tracking proces en voegen we de berekende contactpositie van het balletje met de grond en de gedetecteerde positie van het balletje toe als eerste twee waardes voor een nieuwe tracking.

5.7 Afwerking

Om de detectie vervolgens te illustreren tonen we op het scherm zowel het originele beeld als het berekende masker met daar bovenop de gedetecteerde trajecten getekend. Een voorbeeld van zo een uitvoer wordt gegeven in figuur 5.5.



Figuur 5.5: Links het originele beeld en rechts het masker voor dit beeld en de 7 voorafgaande beelden. Op beide afbeeldingen werd het gedetecteerde pad van het balletje aangeduid.

Hoofdstuk 6

Resultaten

6.1 Opstelling

Als primaire test data gebruiken we een video opname van een pingpongballetje [67] dat heen en weer geslaan wordt over een pingpongtafel. Een statische camera staat ter hoogte van het net van de tafel.

De videodata heeft een resolutie van 320 x 240 pixels en bestaat uit 100 frames tegen 24 frames per seconde. De moeilijkheid van detectie in deze scene ligt bij het feit dat het balletje een laag contrast heeft ten opzichte van de achtergrond, het onderscheiden van het balletje. Ook beweegt het balletje met momenten zeer snel waardoor motion blur zeker en vast een probleem is.

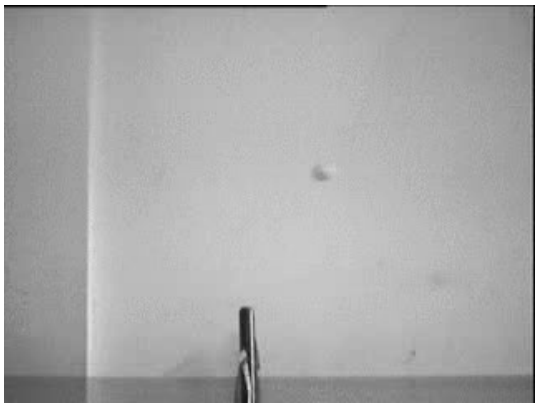
In figuur 6.1 wordt een willekeurig beeld uit de scene gegeven ter illustratie, in dit geval beeld 38.

Het systeem waarop getest werd was een Dual Core Pentium D 820 aan 2.8Ghz met 2GB aan ram en een nVidia GF8800GT als videokaart. We zijn erin geslaagd real-time detectie te bekomen door zo veel mogelijk werk te laten doen door de GPU.

6.2 Detectie

Het algoritme slaagt erin om voor de testscene van de 58 beelden dat er een bal in de scene te zien is de bal voor 55 van deze beelden rechstreeks te detecteren. Voor 2 van deze wordt het probleem echter opgelost in een latere stap door de methode van de virtuele posities, 6.2 toont de situatie waarbij deze problemen optreden.

In beeld 23 is het balletje net terug in beeld na even uit beeld verdwenen te zijn, het probleem

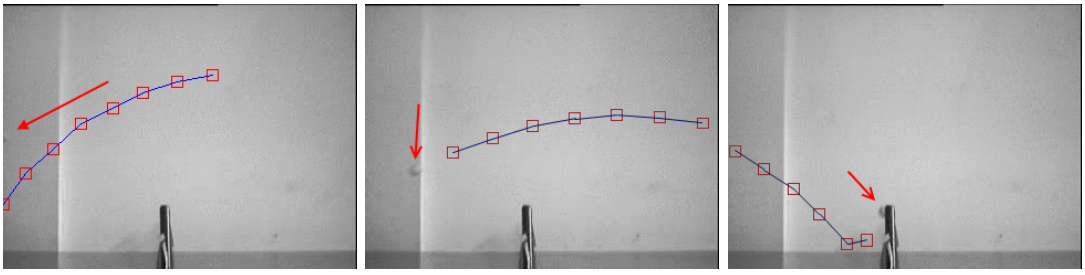


Figuur 6.1: De testscene.

hier is dat het balletje veel te klein is om gedetecteerd te worden, het is maar net in beeld en is zelfs voor het blote oog zeer slecht zichtbaar. De virtuele positie methode heeft hier geen effect omdat er nog geen tracking data beschikbaar is.

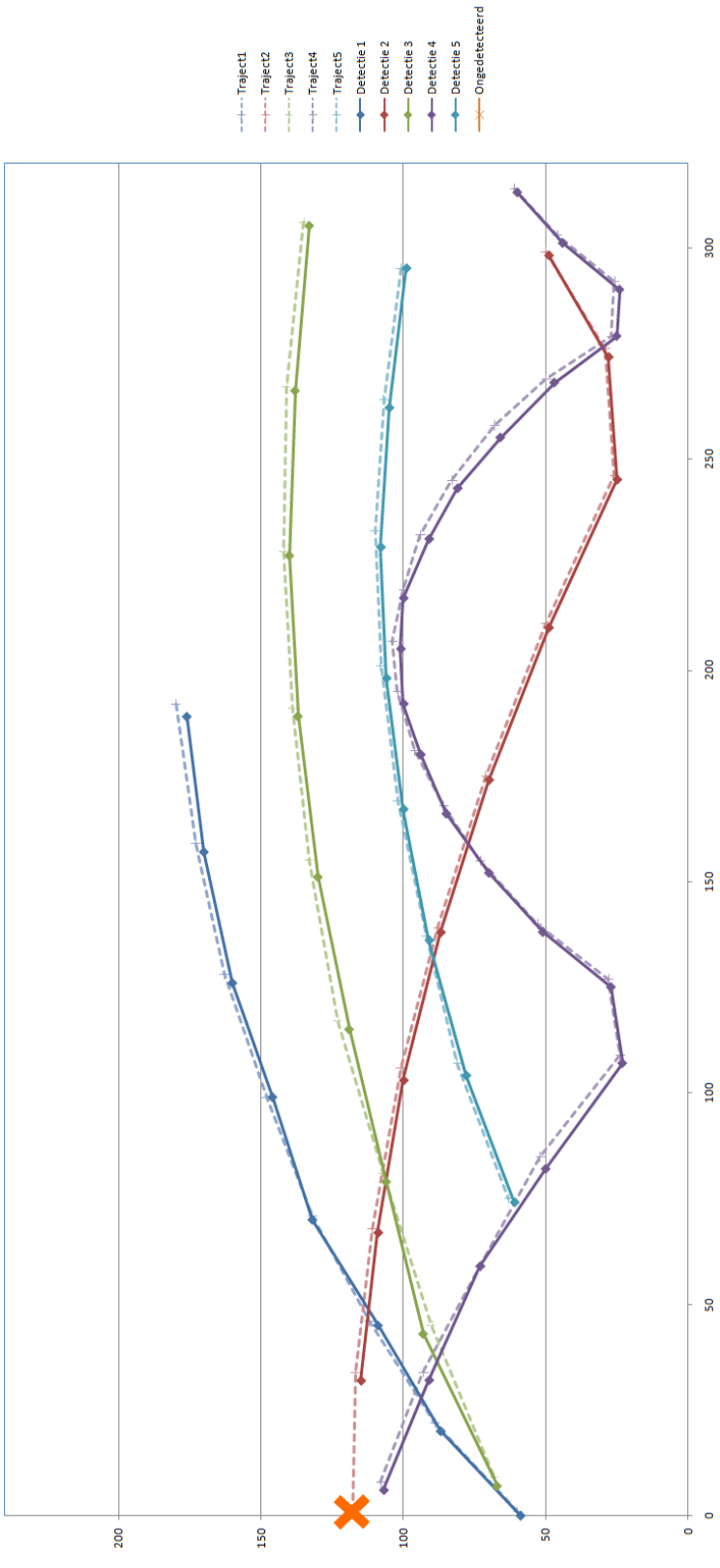
Het volgende problematische punt in het traject is beeld 49, hier gaat het balletje over een discontinuiteit in de achtergrond met een zeer gelijkaardige textuur als het balletje, op het masker wordt dit niet als een beweging gedetecteerd en hierdoor wordt deze positie niet gedetecteerd als een balletje. Dit is echter niet dramatisch aangezien een virtuele positie wordt toegevoegd in de volgende stap.

Tenslotte verdwijnt het balletje in beeld 71 gedeeltelijk achter het net. Ook hiermee heeft de detectie een probleem maar dit wordt opgelost door de tracking in de volgende stap.



Figuur 6.2: Ongedetecteerde balletjes in vlnr. beelden 23, 42 en 64. Het virtuele positie algoritme detecteert in een volgende stap de waarden van beeld 42 en 64 nog maar het balletje in beeld 23 blijft ongedetecteerd. Het balletje wordt telkens aangeduid met een rode pijl.

Om de nauwkeurigheid van de detectie te testen gebruiken we een set van data waarbij de detectie manueel gebeurde, we plotten deze posities op een grafiek en vervolgens plotten we de gedetecteerde waarden erbij. Aan de hand van deze analyse bepalen we hoe dicht de detectie de "juiste" waarde benadert. Zie figuur 6.3 voor de plot hiervan, de gedetecteerde paden worden met een volle lijn weergegeven en de manueel ingevoerde waarden krijgen een



Figuur 6.3: Een schematische voorstelling van de detectie van een pingpong balletje aangeduid op een grafiek. De stippellijnen stellen de correcte data voor en de volle lijnen de gedetecteerde data doormiddel van het beschreven algoritme. Het ongedetecteerde balletje aan de linkerkant is weergegeven met een kruis.

stippellijn toegewezen. Het oranje kruis stelt het enige ongedetecteerde balletje voor.

Indien we geen rekening houden met het ongedetecteerde balletje, een balletje dat zelfs met het blote oog heel moeilijk te detecteren is, worden de balletjes gedetecteerd met een gemiddelde fout van 2,7pixels en een maximale fout van 5 pixels. We zijn er dus in geslaagd om een zeer nauwkeurige detectiemethode te ontwikkelen.

Het probleem dat typisch optreedt bij de detectie van kleine objecten en gevoelige is de gevoeligheid voor ruis. De typische methoden om ruis weg te werken in een beeld zouden echter ook het kleine object kunnen weg werken omdat filters niet differentiëren tussen ruis en nuttige objecten. Daarom is deze methode zeer geschikt om met deze ruis om te gaan, het werkt rechtstreeks op de ruishoudende beelden maar de robuustheid van een wavelet transformatie ten opzichte van ruis zorgt ervoor dat ruis weinig effect heeft op het uiteindelijke resultaat.

De beperking van de detectie methode ligt echter bij het feit dat het de beweging van pixels gebruikt om tot de detectie te komen, dit wil zeggen dat alleen beelden uit een statische camera geschikt zullen zijn. Aangezien bij een bewegende camera de meeste pixels als bewegend zullen gedetecteerd worden zal een eventueel balletje opgaan in de algemene beweging van de scene. Ook zullen scenes met veel beweging, bijvoorbeeld met een bewegende achtergrond zoals een tribune waar publiek in zit, de detectie van het balletje hinderen.

De beschreven methode toont aan dat real-time detectie met behulp van wavelets een realistisch haalbare methode is om tot een robuuste detectie van kleine en snel bewegende objecten te komen. Omdat deze methode geen textuur informatie vereist om tot detectie te komen, enkel bewegings informatie is het uitermate geschikt voor scenes waarbij het balletje en de achtergrond een zeer laag contrast hebben ten opzichte van elkaar.

6.3 Tracking

Het is als we naar de tracking methode gaan kijken die gebruikt werd dat we de gebreken van de methode kunnen analyseren. De tracking met behulp van de Kalman filter is accuraat tot op het punt dat een balletje maneuvreert, bijvoorbeeld door te kaatsen op het grondoppervlak of doordat de speler het balletje van koers doet veranderen. Het model om maneuvers te voorspellen in het algoritme is beperkt tot een voorspelling van de botsing in twee dimensies, waardoor het enkel een ruwe benadering voorstelt van de botsing.
Een andere beperking kan geöbserveerd worden bij traag bewegende balletjes, deze balletjes zullen in een masker als 1 aaneensluitende regio gedetecteerd worden en zullen hierdoor waarschijnlijk genegeerd worden door de tracking.

6.4 Real-time

De typisch zeer rekenintensieve wavelet tranformatie wordt doormiddel van CUDA en de grafische hardware versneld waardoor er elke $\frac{1}{24}$ste seconde een wavelet transformatie van $320 \times 240 \times 8 = 614.400$ pixels plaatsvindt, hetgeen resulteert in 14.745.600 pixels per seconde waarvoor berekeningen uitgevoerd worden alleen al voor de wavelet transformatie en nog eens zoveel voor de berekening van de maskers. Door deze grote hoeveelheid data en berekeningen is het belangrijk om te weten welke operaties snel verlopen en welke de dure operaties zijn.

Om die reden werd een tijdsanalyse uitgevoerd, de applicatie werd 100 maal uitgevoerd met de originele testscene en de prestaties van verschillende delen van de applicatie werden geanalyseerd.

6.4.1 Grijswaarde transformatie

Een eerste analyse werd uitgevoerd op de transformatie van een kleurenafbeelding van 320×240 pixels naar een grijswaarden afbeelding, de resultaten zijn samengevat in de volgende tabel:

Meting	Gemiddelde (ms)	Variantie (ms)	Maximum (ms)
Totaal	$1,8227248$	0.134859	3.222376
Transformatie	$0,4535001$	$0,0015547$	$1,255715$
Kopieeroperaties	$1,241024$	$0,26985672$	$2,4233988$

Hier stellen kopieeroperaties de operaties voor waarbij de data van en naar de GPU moet getransfereerd worden, merk op dat kopieer operaties hier het overgrote deel van de tijd innemen terwijl de eigenlijke transformaties maar een fractie van die tijd nodig hebben.

6.4.2 Wavelet transformatie en analyse

De volgende reeks van tijdwaarden komt uit de wavelet transformatie van het algoritme. Deze methode bestaat uit 3 delen, namelijk de wavelet transformatie (Wavelet), het berekenen van het masker (Masker) en het uitvoeren van de opening (Opening) op het binaire masker. Net zoals in de vorige sectie werd ook hier de benodigde tijd voor de kopieer operaties gemeten. De resultaten worden opgesomd in de volgende tabel:

Meting	Gemiddelde (ms)	Variantie (ms)	Maximum (ms)
Totaal	$16,471665$	$0,335425776$	$18,568207$
Wavelet	$0,161571648$	$0,007357183$	$0,753571$
Masker	$0,484147154$	$0,016733001$	$1,059736$
Opening	$0,775763099$	$0,020650459$	$1,32856$
Kopieeroperaties	$12,04055601$	$0,293597856$	$14,088066$

Ook hier bestaat het grootste deel van de uitvoertijd van de methode uit kopieer operaties terwijl de wavelet transformatie, de opening en de berekening van het masker elk in gemiddeld minder dan een miliseconde uitgevoerd worden.

Merk op dat voor elk van deze operaties de variantie zeer laag ligt, dit komt doordat de waarde van de invoer helemaal geen invloed heeft op de prestatie van het algoritme. Deze observatie is logisch aangezien er voor elk apart beeld exact dezelfde berekeningen gebeuren, het enige dat verschilt is de data zelf. De enige manier waarop de invoer invloed kan hebben op de prestaties is als er meer data binnen komt, bijvoorbeeld voor een video van hogere resolutie of een wavelet transformatie op 16 in plaats van 8 beelden. Tijdens de executie kunnen we er echter op vertrouwen dat deze berekeningen min of meer in dezelfde tijd zulen klaar zijn voor elke stap.

6.4.3 Detectie en Tracking

Deze fase gebeurt op de CPU dus we hebben hier geen overhead voor het kopieren van de data. In de eerste fase hierbij proberen we een balletje te detecteren, hiervoor moeten eerst de middelpunten van elke kandidaat bal berekend worden(Kandidaat detectie). Vervolgens worden dubbele voorkomens van punten geanalyseerd (Paar detectie). En tenslotte wordt er een voorspelling gedaan van mogelijke posities van balletjes door aan te nemen dat trajecten lineair zijn (Lineaire predictie).

Meting	Gemiddelde (ms)	Variantie (ms)	Maximum (ms)
Kandidaat detectie	$6,682533048$	$2,250648382$	$11,752391$
Paar detectie	$0,046796677$	$0,00579362$	$0,398665$
Lineaire predictie	$0,090945258$	$0,016661188$	$0,519891$

Hier is de kandidaat detectie de belangrijkste reden voor een eventuele vertraging van het algoritme, merk vooral op dat de variantie van de kandidaat detectie zeer hoog is, dit volgt uit het feit dat er meer berekeningen moeten gebeuren voor maskers waar er meer beweging op is gedetecteerd. Een drukke scene met veel beweging zal dus extra rekentijd vereisen.

Eenmaal een traject opgesteld is moet het balletje gevolgd worden, er moet eveneens een detectie van bewegende punten gebeuren en deze punten moeten gevolgd worden met een Kalman filter, geimplementeerd door OpenCV [16].

Meting	Gemiddelde (ms)	Variantie (ms)	Maximum (ms)
Kandidaat detectie	$5,6612809$	$3,122203132$	$10,551264$
Kalman filter	$2,07623908$	$0,01547892$	$2,72948072$

Ook hier ligt de variantie van de kandidaat detectie hoog terwijl de Kalman filter binnen een relatief betrouwbare tijd uitgevoerd wordt, eveneens omdat de invoerdata geen invloed heeft op de performantie van deze methode.

6.4.4 Mogelijke Optimalisaties

Zoals de vorige secties uitwijzen zijn er twee grote oorzaken voor een mogelijke vertraging van het algoritme. Een eerste hiervan zijn de kopieeroperaties die data van en naar de GPU verplaatsen, de operaties zijn echter noodzakelijk en zijn niet rechtstreeks voor optimalisatie vatbaar. Onrechtstreeks zijn er wel optimalisaties hiervoor mogelijk, zoals bijvoorbeeld het beperken van deze dure kopieer operaties, meer hierover later.

De tweede optie wordt dan de optimalisatie van de andere grote vertraging en dat is het kandidaat detectie algoritme. Zeker voor grotere beelden en beelden met meer beweging zal deze methode een optimalisatie kunnen gebruiken. Een mogelijkheid voor de optimalisatie hiervan wordt een uitvoering van deze taak op de GPU. In plaats van het middelpunt van de geconnecteerde witte regionen te berekenen zou een analyse van elke aparte pixel een betere oplossing zijn voor een GPU implementatie. Deze analyse zou dan voor elke pixel apart kunnen bepalen of het al dan niet een bal voorstelt. Op deze manier zouden ook traag bewegende objecten mogelijks gedetecteerd kunnen worden, een mogelijkheid die het huidige algoritme niet heeft.

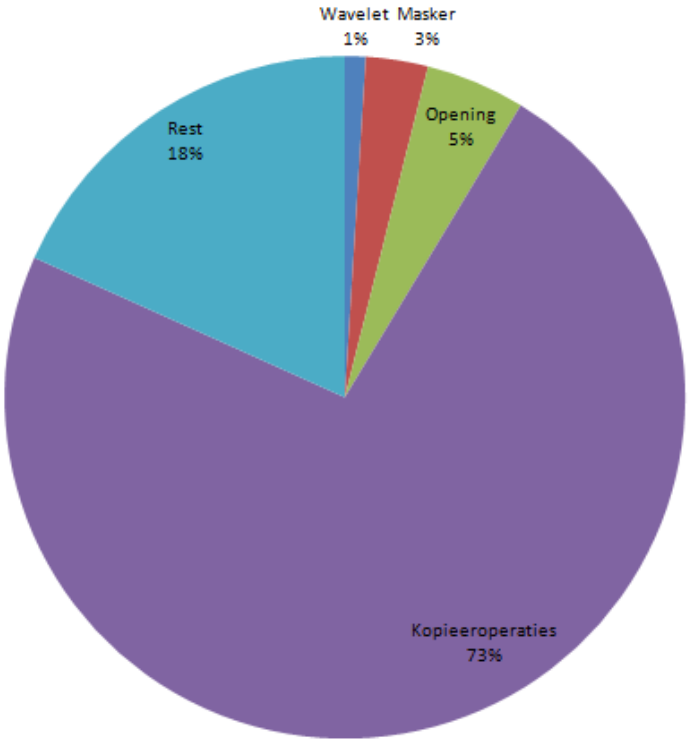
Een ander voordeel van deze laatste aanpak is dat de wavelet data op de GPU kan gelaten worden. Dit zou een dure kopieer operatie vervangen door een operatie waarbij enkel een lijst van kandidaten doorgegeven moet worden.

6.5 CUDA

Zoals al voorspeld in het hoofdstuk over de GPGPU is een GPU dus een zeer krachtige parallelle processor die sinds kort zijn snelheid ter beschikking stelt voor applicaties buiten de grafische rendering alleen. Een zeer breed gamma aan applicaties zouden de kracht van een GPU kunnen gebruiken om tijd-kritieke berekeningen uit te voeren.

Er moet echter wel rekening gehouden worden met de overhead die gegenereerd wordt door het kopieren van de data, dit kan een significante vertraging in uitvoering teweeg brengen. Een afweging moet dus steeds gemaakt worden of de tijd die gewonnen wordt door parallelle executie niet verloren wordt aan data transfers. Om de grootte van de invloed van kopieeroperaties op de prestatie van een algoritme te illustreren geven we figuur 6.4. De figuur toont een verdeling waarop aangeduid is welke fractie tijd elk deel van een algoritme inneemt. We gebruiken hiervoor de waarden uit sectie 6.4.2 *Wavelet transformatie en analyse*. De blauwe

Rest waarde somt alles op dat niet onder de andere categoriën valt maar toch bij het volledige algoritme hoort, zoals onder andere initialisatie en debugging berekeningen.



Figuur 6.4: Een verdeling die aangeeft welke delen binnen de wavelet transformatie en analyse het meeste tijd innemen. Merk op dat kopieeroperaties het grootste deel van de tijd vereisen.

6.5.1 Programmeren in CUDA

CUDA heeft een relatief lage instapdrempel voor een programmeur die al vertrouwd is met de programmeertaal C. Het programmeren in parallel is intuitief en relatief eenvoudig. CUDA is echter meestal vatbaar tot zeer strenge optimalisaties en om deze optimalisaties uit te voeren is een grondige kennis van de architectuur van CUDA en heel wat ervaring nodig. Ook ligt de meest efficiënte parallelle versie van een algoritme meestal niet voor de hand en is het nodig om de literatuur te raadplegen, als deze voorhanden is.

CUDA is echter nog beperkt in zijn ondersteuning, zo was er voor het maken van deze thesis nog geen debugger of profiler voorhanden voor CUDA. Met de recentelijk uitgebrachte beta van CUDA 2.0 is er echter wel een profiler uitgebracht en nVidia [35] heeft al aangekondigd dat het aan een debugger aan het werken is.

Hoofdstuk 7

Conclusies

In dit werk werd een methode beschreven om tot de detectie en tracking van snel bewegende objecten te komen. We gebruikten hiervoor de kracht van de recente grafische hardware om de wavelet transformatie en de detectie in real-time te kunnen realiseren.

We zijn erin geslaagd om een nauwkeurige detectie en tracking methode te ontwikkelen. We detecteerden voor de test data 53 van de 54 balletjes die in beeld kwamen met een gemiddelde afwijking van $2,7$ pixels.

De beschreven methode toont aan dat de implementatie gebaseerd op een temporele wavelet transformatie een realistisch haalbare methode is om tot een robuuste real-time detectie van kleine en snel bewegende objecten te komen. Omdat deze methode geen textuur informatie vereist om tot detectie te komen, enkel bewegingsinformatie, is ze zelfs geschikt voor scenes waarbij het balletje en de achtergrond een zeer laag contrast hebben ten opzichte van elkaar, iets waar andere detectiemethodes dikwijls moeite mee hebben.

We toonden tenslotte aan dat de GPU een geschikt platform is om bepaalde rekenintensieve, data-parallelle berekeningen zoals bijvoorbeeld een wavelet transformatie op een zeer snelle manier uit te voeren.

Hoofdstuk 8

Toekomstig werk

De besproken methode in deze thesis is redelijk beperkt in zijn mogelijkheden om tot de detectie van een bal te komen. Een algemene methode om tot detectie van een bal te komen dringt zich op. Sinds we bewezen hebben dat het mogelijk is om met behulp van grafische hardware de detectie van balletjes betrekkelijk te versnellen is het onderzoek naar andere methodes die met behulp van deze grafische hardware versneld kunnen worden aan te raden. Zo kan bijvoorbeeld de wavelet transformatie methode gecombineerd worden met andere detectie methodes om tot een volledig sport analyse systeem te komen.

Een Daubechies wavelet basis heeft eigenschappen die superieur zijn aan die van de Haar basis, zoals een betere impuls respons, daarom is het aangewezen om de invloed van een Daubechies wavelet basis te analyseren voor deze applicatie.

Een uitbreiding van deze implementatie naar een multi-camera systeem om de detectie en tracking uit te breiden van 2D naar 3D zou een noodzakelijke stap zijn om tot een realistisch sport analyse systeem te komen. Ook zou de informatie uit verschillende cameras gebruikt kunnen worden om de tracking en detectie te verbeteren voor het hele systeem.

Een laatste mogelijke optimalisatie zou zijn om de berekeningen wat te verdelen tussen GPU en CPU. Op dit moment wordt quasi al het werk gedaan door de GPU terwijl de CPU maar weinig werk voorgeschoteld krijgt. Waarschijnlijk kan de hele uitvoering nog versnellen als de CPU ook wat te doen krijgt. Het probleem hiermee is dat sommige berekeningen niet in parallel kunnen gebeuren omdat elke nieuwe stap voortbouwt op de uitvoer van een vorige stap. Wat echter wel kan gebeuren is bijvoorbeeld de Kalman filter voorspelling te laten berekenen door CPU terwijl de GPU aan de wavelet transformatie bezig is. Aan de andere kant is een GPU uitermate geschikt om aan matrix vermenigvuldigingen te doen, dus een GPU implementatie van een Kalman filter zal waarschijnlijk veel sneller werken dan zijn CPU tegenhanger. Ook hier is er dus nog ruimte voor onderzoek naar optimalisaties.

Bibliografie

[1] Raúl Arellano. Computer science applied to competitive swimming: Analysis of swimming performance and fluid mechanics. *International Journal of Computer Science in Sport*, 2(1):9–19, 2003.

[2] Arnold Baca. Computer science based feedback systems on sports performance. *International Journal of Computer Science in Sport*, 2(1):20–30, 2003.

[3] Arnold Baca. Computer science in sport: An overview of history, present fields and future applications(part i). *International Journal of Computer Science in Sport*, 2006.

[4] Domingo Blázquez and Jordi Calvo. E-learning experience of the "virtual campus of sport". *International Journal of Computer Science in Sport*, 2003.

[5] Bingqi Chen and Zhiquang Wang. A statistical method for analysis of technical data of a badminton match based on 2-d seriate images. *Tshingua science and technology*, 2007.

[6] D. Davies, P. L. Palmer, and M. Mirmehdi. Detection and tracking of very small low-contrast objects. In *Proceedings of the 9th British Machine Vision Conference*, pages 599–608. BMVA Press, 1998.

[7] Neville de Mestre. Mathematics in sport down under. *International Journal of Computer Science in Sport*, 1(1), 2002.

[8] T. D'Orazio, N. Ancona, G. Cicirelli, and M. Nitti. A ball detection algorithm for real soccer image sequences. *International Conference on Pattern Recognition*, 2002.

[9] Pogalin E., Thean A.H.C., Baan J., Schipper N.W., and A.W.M. Smeulders. Video-based training registration for swimmers. *International Journal of Computer Science in Sport*, 2006.

[10] E-motek. Caren. `http://www.e-motek.com/medical/applications/index.htm`, 2008.

[11] Nathan Funk. A study of the kalman filter applied to visual tracking. *Project for CMPUT 652*, 2003.

[12] GLDynamics. Teamscout. `http://gldynamics.com/index.html`, 2008.

[13] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Prentice Hall, 2001.

[14] Mark Harris. Optimizing cuda. *SC07*, 2007.

[15] Mark Harris. Parallel prefix sum (scan) with CUDA. *Nvidia*, 2007.

[16] Intel. Opencv. `http://opencvlibrary.sourceforge.net/`, 2008.

[17] The Internet. Kalman filter - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Kalman_filter`, 2008.

[18] Jensen and la Cour-Harbo. *Ripples in Mathematics*. Springer, 2001.

[19] Won-Ki Jeong, P. Thomas Fletcher, Ran Tao, and Ross T. Whitaker. Interactive visualization of volumetric white matter connectivity in DT-MRI using a parallel-hardware hamilton-jacobi solver. In *IEEE Conference on Visualization*, 10 2007.

[20] R.E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 1960.

[21] Larry Katz. Multimedia and the internet for sport sciences: Applications and innovations. *International Journal of Computer Science in Sport*, 1(1), 2002.

[22] John Kessenich, Dave Baldwin, and Randi Rost. *The OpenGL® Shading Language*. OpenGL, 1.20 edition, 10 2006.

[23] Martin Lames. Computer science for top level team sports. *International Journal of Computer Science in Sport*, 2(1):57–72, 2003.

[24] M. Leo, T. D'Orazio, P. Spagnolo, and A. Distante. Wavelet and ica preprocessing for ball recognition in soccer images. *ICGST International Journal on Graphics, Vision and Image Processing*, SI1, May 2005.

[25] Peihua Li and Tianwen Zhang. Visual contour tracking based on particle filters. *Image and Vision Computing*, 2004.

[26] Hawk-Eye Innovations Ltd. Hawk-eye. `http://www.hawkeyeinnovations.co.uk/`, 2008.

[27] Joachim Mester, Florian Seifriz, and Ulrike Wigger. Scientific basics and commercial interests in sport information technology. *International Journal of Computer Science in Sport*, 1(1), 2002.

[28] John Michalakes and Manish Vachharajani. Gpu acceleration of numerical weather prediction. *Workshop on Large Scale Parallel Processing*, 2008.

[29] Microsoft. MSDN DirectX Developer Center : HLSL. `http://msdn2.microsoft.com/en-us/library/bb509561.aspx`, 11 2007.

[30] Microsoft. `http://www.gamesforwindows.com/en-US/AboutGFW/Pages/DirectX10.aspx`, 2008.

[31] Microsoft. Directshow. `http://msdn.microsoft.com/en-us/library/ms783323(VS.85).aspx`, 2008.

[32] Bart De Moor and Emil-Mihal Muresan. Soccer mining: voetbal, tactiek. *Het Ingenieursblad*, 2005.

[33] Nintendo. Nintendo 64. `http://www.nintendo.nl/NOE/nl_NL/systems/nintendo_64_1152.html`, 2008.

[34] nVidia. Cuda. `http://www.nvidia.com/object/cuda_home.html`, 2007.

[35] nVidia. nvidia. `http://www.nvidia.com/`, 2007.

[36] nVidia. Cuda showcase. `http://www.nvidia.com/object/cuda_showcase.html`, 2008.

[37] nVidia. Geforce 8800. `http://www.nvidia.com/page/geforce_8800.html`, 2008.

[38] NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA 95050. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 1.1 edition, 2007.

[39] James Orwell, P. Remagnino, and G.A. Jones. Multi-camera colour tracking. *Second IEEE Worksop on Visual Surveillance*, 1999.

[40] John Owens and U.C. Davis. Data-parallel algorithms and data structures. *High Performance Computing with CUDA*, 2005.

[41] Jürgen Perl. Computer science in sport: An overview of history, present fields and future applications(part ii). *International Journal of Computer Science in Sport*, 2006.

[42] Gopal Pingali, Agata Opalach, and Yves Jean. Ball tracking and virtual replays for innovative tennis broadcasts. *icpr*, 04:4152, 2000.

[43] P. Wayne Power and Johann A. Schoonees. Understanding background mixture models for foreground segmentation. *Imaging and Vision Computing New Zealand,*, 2002.

[44] Thomas Reilly. Selected impacts of computers in the sports sciences. *International Journal of Computer Science in Sport*, 2006.

[45] Jinchang Ren, James Orwell, Graeme Jones, and Ming Xu. Real-time 3d soccer ball tracking from multiple cameras. *Workshop on Computer Vision Based Analysis in Sport Environments*, 2004.

[46] Jinchang Ren, James Orwell, Graeme Jones, and Ming Xu. Real-time modeling of 3d soccer ball trajectories from multiple fixed cameras. *IEEE Transactions on Circuits and Systems for Video Technology*, 2007.

[47] Jinchang Ren, James Orwell, and Graeme A. Jones. Estimating the position of a football from multiple image sequences. *BMVA One Day Symposium on Spatiotemporal Image Processing, March 24th, Royal Statistical Society, London*, 2004.

[48] Jinchang Ren, James Orwell, and Graeme A. Jones. Generating ball trajectory in soccer video sequences. *Workshop on Computer Vision Based Analysis in Sport Environments*, 2006.

[49] Jinchang Ren, James Orwell, Graeme A. Jones, and Ming Xu. A general framework for 3d soccer ball estimation and tracking. In *ICIP*, pages 1935–1938, 2004.

[50] Ulrike Rockmann, Stafan Thielke, and Miriam Seyda. E-learning experiments - the software race and the general design. *International Journal of Computer Science in Sport*, 2003.

[51] Randi J. Rost. *OpenGL Shading Language*. Addison Wesley, 2006.

[52] Michael C. Schatz and Cole Trapnell. Fast exact string matching on the gpu. *Center for Bioinformatics and Computational Biology*, 2007.

[53] Florian Seifriz, Jochen Mester, Alexander Krämer, and Ralf Roth. The use of gps for continuous measurement of kinematic data and for the validation of a model in alpine skiing. *International Journal of Computer Science in Sport*, 2003.

[54] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. *Graphics Hardware*, 2007.

[55] John Shalf. The new landscape of parallel computer architecture. In *Journal of Physics: Conference Series*, volume 78, 2007.

[56] Mehdi Sharifzadeh, Farnaz Azmoodeh, and Cyrus Shahabi. Change detection in time series data using wavelet footprints. *Lecture Notes in Computer Science*, 2005.

[57] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Addison Wesley, 2005.

[58] Sony. Playstation. `http://be.playstation.com/?site_locale=nl_BE`, 2008.

[59] Chris Stauffer and W.E.L. Grimson. Adaptive background mixture models for real-time tracking. *Computer Vision and Pattern Recognition*, 1998.

[60] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. Wavelets for computer graphics: A primer part 1. *Computer Graphics and Applications, IEEE*, 1995.

[61] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. Wavelets for computer graphics: A primer part 2. *Computer Graphics and Applications, IEEE*, 1995.

[62] R. N. Strickland and He Il Hahn. Wavelet transform methods for object detection and recovery. *IEEE Transactions on Image Processing*, 1997.

[63] The Atari Museum. `http://www.atarimuseum.com/videogames/consoles/7800/7800menu`, 1995.

[64] J.A. van Meel, a. Arnold, D. Frenkel, S.F. Portegies Zwart, and R.G. Belleman. Harvesting graphics power for md simulations. *ArXiv e-prints*, 2008.

[65] Virge. S3 graphics. `http://www.s3graphics.com/en/index.jsp`, 2008.

[66] Greg Welch and Gary Bishop. An introduction to the kalman filter. *Department of Computer Science University of North Carolina*, 2006.

[67] Mukesh A. Zaveri, S.N. Merchant, and Uday B. Desai. Small and fast moving object detection and tracking in sports video sequences. *IEEE International Conference on Multimedia*, 2004.