Samenvatting

Digitale video is over de jaren heen steeds belangrijker geworden in het dagelijkse leven. De compressie en encoding ervan is bijgevolg dan ook een zeer belangrijk aspect van digitale video geworden. Om video encoders nog verder te versnellen en in de toekomst performant te houden dient gezocht te worden naar algoritmen die optimaal gebruik maken van hedendaagse multi-core processors. Een belangrijk type van deze multi-core processors zijn de GPU's, deze bieden namelijk veel rekenkracht aan maar om deze te benutten dient een zeer hoge graad van parallellisatie bereikt te worden.

Deze thesis onderzoekt de mogelijkheden om video encoding algoritmen te implementeren op de GPU. Hiervoor werd een bundeling gemaakt van gekende literatuur over het onderwerp en in deze thesis gepresenteerd. Er werd ook een eigen implementatie gerealiseerd in CUDA van enkele essentiële componenten. Uit het onderzoek blijkt dat GPU parallellisatie tot 10x snelheids verbeteringen laat voor de typische componenten nodig voor het encoderen van I-frames vergeleken met een multi-threaded CPU implementatie. Voor het encoderen van P-frames laat de GPU implementatie zelfs tot 20x snelheidsverbeteringen zien vergeleken met de multi-threaded CPU implementatie.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Hoofdstuk 1

Inleiding

Digitale video is door de jaren heen een steeds belangrijker aspect geworden in het dagelijkse leven van de mensen. Niet alleen is de hoeveelheid video data dat we consumeren ongelofelijk gestegen maar ook de kwaliteit van de content. De opkomst en groei van services zoals Netflix, Yelo TV (Telenet), ... zullen deze vraag enkel doen toenemen in de komende jaren. Bovendien lijkt de opkomst van 4k video de vraag naar efficiënte encoding algoritmen steeds belangrijker te maken.

Een belangrijk probleem is dan ook deze video data op een zo efficiënt mogelijke manier te transporteren. Het is hiervoor dat traditioneel video encoding algoritmen gebruikt worden, om video data in een zo klein mogelijk formaat te kunnen comprimeren. Door de immer stijgende spatiale resolutie van de te encoden beelden en de vraag om video in steeds betere kwaliteit te encoden stijgt de totale rekenkracht nodig om video te encoden exponentieel.

Het onderzoek naar methodes om meer en beter resources van het systeem te benutten is dan ook van cruciaal belang om ervoor te zorgen dat video encoders ook in de toekomst relevante performantie blijven behouden.

Daar waar één tak van het onderzoek eerder focust op het uitwerken van efficiëntere encoding algoritmen bruikbaar in de traditioneel sequentieel model voor video encoders is er in de laatste jaren steeds meer en meer aandacht aan het ontstaan voor onderzoek naar het beter benutten van de beschikbare rekenkracht van het systeem. De opkomst van de multi-core processors brengt namelijk een grote hoeveelheid extra rekenkracht met zich mee, die wel door de algoritmen correct benut moet worden. Het sequentieel model moet voor optimale benutting van het systeem dus herzien worden om maximaal te beschikbare resources te benutten. De nieuwe generatie encoders zoals H.265 en VP9 proberen zich dan ook te focussen op het introduceren van parallellisme in de video encoding pipeline.

Een belangrijke nieuwe evolutie binnen de multi-core processors is de

opkomst van computing op de GPU. Door de hoge graad van parallellisatie noodzakelijk om optimaal gebruik te kunnen maken van alle resources op de GPU is het onderzoek naar parallellisatie van alle componenten van de video encoder des te belangrijker.

1.1 Doel

In het kader van het onderzoek naar het gebruik van de GPU als mogelijk manier om video encoding te versnellen dienen twee zaken uitgevoerd te worden.

In de eerste plaats moet een grondig overzicht gegeven worden van het al eerder gevoerde onderzoek binnen het domein. Op basis hiervan kan al vastgesteld worden wat de mogelijkheden zijn op het vlak van video encoding op de GPU en waar eventuele problemen zich kunnen voordoen.

Het tweede doel van de thesis is het realiseren van een eigen implementatie van een video encoder op de GPU en de performantie ervan te vergelijken met een CPU implementatie. Op deze manier kan vastgesteld worden vanuit eigen onderzoek welke componenten van de video encoder makkelijk of moeilijk te parallelliseren zijn en welke componenten het meeste baat hebben bij parallellisatie.

1.2 Indeling

In hoofdstuk 2 wordt er eerst gefocust op het introduceren van alle componenten van een typische video encoder. Daarna wordt een overzicht gegeven van algemeen onderzoek naar de parallellisatie van deze componenten en in een laatste sectie wordt een overzicht gegeven van de literatuur rond de implementatie van deze componenten op de GPU.

Hoofdstuk 3 focust op het duiden en argumenteren van de gebruikte technieken in de uiteindelijke implementatie. De focus ligt voornamelijk op NVIDIA CUDA.

In hoofdstuk 4 wordt dan aangehaald welke componenten geïmplementeerd zijn in de uiteindelijke realisatie en de wijze waarop deze implementatie gerealiseerd is.

De resultaten van het onderzoek, zijnde een vergelijking in performantie tussen een CPU implementatie van de eigen video encoder en een GPU implementatie wordt gegeven in hoofdstuk 5.

Tot slot worden in hoofdstuk 6 conclusies van de thesis geformuleerd en worden suggesties gegeven voor eventueel toekomstig onderzoek.

Hoofdstuk 2

Literatuurstudie

2.1 Video encoding

In deze sectie worden enkele basistechnieken beschreven die noodzakelijk zijn om een video encoder te realiseren. Hierbij dient opgemerkt te worden dat er nog vele andere technieken beschikbaar zijn maar deze sectie beperkt zich tot de technieken relevant voor deze thesis.

Video encoders hebben tot doel om raw video data (vaak in YCrCb of RGB formaat) te comprimeren naar een zo klein mogelijk bestandsgrootte met zo weinig mogelijk verlies aan beeldkwaliteit. Hierbij wordt in de regel gebruik gemaakt van een aantal verschillende fases om compressie te realiseren. Een eerste fase, het zogenaamde Prediction model, focust op temporele informatie tussen frames of spatiale informatie binnen hetzelfde frame te gebruiken om compressie te realiseren. Een tweede fase, het Image model, gebruikt een reeks technieken uit de beeldverwerking om compressie op frames te realiseren. In een laatste fase, het Entropy model, worden een reeks traditionele compressietechnieken gebruikt om de overgebleven informatie tot een zo klein mogelijke grootte te reduceren. [1]

In de volgende subsecties wordt er dieper ingegaan op deze verschillende fases, een goed begrip voor de onderliggende technieken van iedere fase is namelijk noodzakelijk om uitspraken te kunnen doen over de effectieve implementatie van de algoritmen.

2.1.1 Prediction model

Zoals eerder al aangehaald bestaat het Prediction model uit twee belangrijke fases. Enerzijds de temporele predictie die zich focust op de relatie tussen twee frames en spatiale predictie die zich focust op de relatie binnen een frame zelf.

De meeste algoritmen binnen het prediction model zijn block-based. In plaats van pixel per pixel frames te vergelijken of te onderzoeken wordt gebruik gemaakt van groepen van pixels, zogenaamde *macroblokken*, om de noodzakelijke algoritmen op uit te voeren.

De temporele predictie bestaat in regel uit twee stappen. Enerzijds de motion estimation, die het doelframe vergelijkt met een bronframe en anderzijds motion compensation die redundante informatie na vergelijking verwijdert om maximale compressie te bekomen. In de volgende secties wordt er dieper ingegaan op deze twee stappen waarna in een laatste sectie het verschil tussen temporele en spatiale predictie wordt aangekaart. [1]

2.1.1.1 Motion Estimation

Zoals eerder al aangehaald wordt tijdens motion estimation blokken uit een doelframe vergeleken met blokken uit een bronframe. Het doel van motion estimation is het vinden van een zo gelijkaardig mogelijk blok uit het doelframe in het bronframe. Hoe kleiner het verschil tussen de 2 blokken hoe meer redundante informatie verwijderd kan worden met als resultaat een hogere graad van compressie.

Om blokken onderling te vergelijken wordt vaak gebruik gemaakt van het zogenaamd Mean of Absolute Differences (MAD). Er wordt eerst een residublok berekent door het vergelijkende blok uit het doelframe af te trekken van een blok uit het bronframe. Hiervan wordt van iedere pixel de absolute waarde gesommeerd en gedeeld door het aantal pixels. De bekomen waarde geeft een goede indicatie van de graad van gelijkenis tussen de 2 gekozen blokken. Het blok uit het bronframe met de kleinste MAD wordt gekozen als *match* voor het blok uit het doelframe. En wordt in de motion compensation fase gebruikt als referentie.

Over de loop van de jaren zijn er een heel aantal algoritmen ontwikkeld om motion estimation uit te voeren. Motion estimation is namelijk de meest rekenintensieve stap binnen video encoders. Alle motion estimation algoritmen maken een keuze tussen enerzijds de snelheid van een match vinden en anderzijds de accuraatheid van de gevonden match. In de volgende paragrafen wordt er dieper ingegaan op 2 veelgebruikte algoritmen, het Full Search algoritme en het Three-step alogoritme. Voor dieper inzicht in andere mogelijke algoritme kan gebruik gemaakt worden van de bronnen aangehaald op het einde van deze sectie.

Het Full Search algorimte is met makkelijkste algoritme om te implementeren en is bovendien het meest accurate. Het grote nadeel is dan weer dat het het minste efficiënte algoritme is van alle mogelijke motion estimation algoritmen. In pseudo-code werkt het algoritme op de volgende manier,

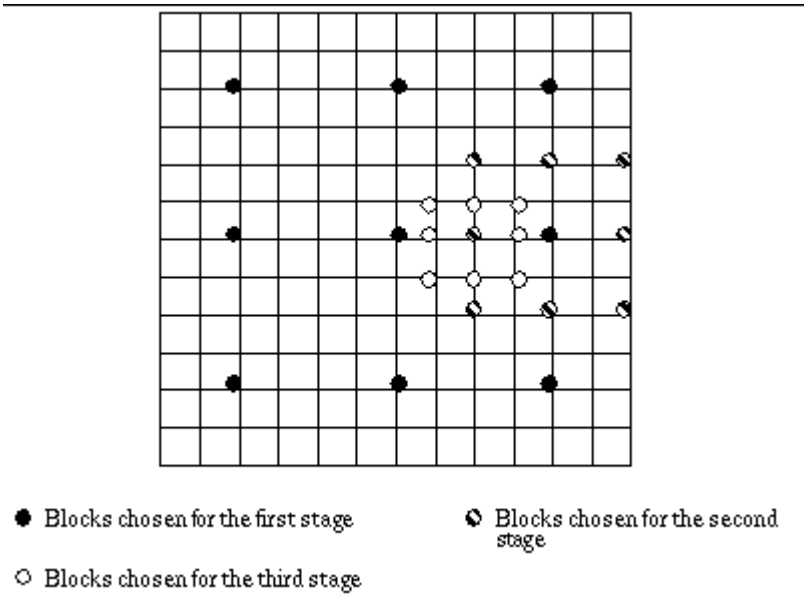
```
int stepSize = getStepSize();
Block target = currentFrame.getBlock();
for x=target.x-stepSize; x < target.x+stepSize; ++x
 for x=target.y-stepSize; y < target.y+stepSize; ++y
  if MAD(target - previousFrame.getBlock(x,y)) < lowestMAD
   match = previousFrame.getBlock(x,y);
```

Door in het bronframe ieder mogelijk blok binnen een bepaalde regio te vergelijken met het gekozen doelblok uit het doelframe, wordt logisch gezien de beste match altijd gevonden (binnen de gekozen regio). Het is makkelijk vast te stellen dat de complexiteit van het algoritme hoog is, namelijk $O(n^2)$. Door de simpelheid van het algoritme is het ideaal om te gebruiken als basis voor andere algoritmen, maar de lage performantie zorgt ervoor dat het weinig praktische waarde heeft als motion estimation algoritme.

Het three-step search algoritme is een meer performant motion estimation algoritme. Het is een zeer populair en veelgebruikt algoritme door zijn simpele implementatie, accurate matching en hoge performantie. In pseudo-code verloopt het algoritme als volgt,

```
int stepSize = getStepSize();
Block target = currentFrame.getBlock();
Position center = target.position;
while stepSize > 1
 Block sourceBlocks[8] = previousFrame.getBlocks(stepSize, center);
 foreach Block b in sourceBlocks
  if MAD(target - b) < lowestMAD
   match = b;
   center = b.position;
 stepSize /= 2;
```

De complexiteit van het algoritme is veel lager dan het Full search algoritme, namelijk $O(log(n))$. Ondanks de lage complexiteit zijn de matches gevonden door het Three-step search algoritme vaak van goede kwaliteit,in vele gevallen zelfs identiek aan de match gevonden door het Full search algoritme. Het is in vele gevallen dan ook preferabel om Three-step search te kiezen over het complexe Full search algoritme.



Figuur 2.1: Voorstelling van het Three-step search algoritme [2]

Er zijn natuurlijk nog vele andere algoritmen beschikbaar die motion estimation realiseren met elk hun eigen voor- en nadelen. Het valt niet binnen de scope van deze thesis iedere algoritmen op te lijsten, voor meer informatie zijn er genoeg bronnen beschikbaar voor alternatieve algoritmen, zoals Four-step search, Diamond search, ... [1] [2].

2.1.1.2 Motion Compensation

Motion compensation is de tweede stap van temporele predictie. Zoals eerder gezegd verwijdert deze stap redundante informatie om maximale compressie te realiseren. Dit wordt gerealiseerd aan de hand van de match gevonden in de motion estimation stap.

Aan de hand van de match wordt net zoals tijdens motion estimation een residu blok berekent waarin enkel de informatie zit die nodig is om aan de hand van het bronblok het doelblok te reconstrueren. Bij compressie wordt dan een vector opgeslagen die verwijst naar het doelblok, de zogenaamde *Motion vector*, en de gecomprimeerde data uit het residu nadat het door het image en entropy model verwerkt is. Omdat het residu blok zeer weinig informatie bevat, bij een goede match is bijna iedere pixel waarde gelijk aan 0, is de gemaakte winst door compressie ervan in vergelijking met volledige compressie van het doelblok veel groter dan het verlies gemaakt door de overhead van het opslaan van de motion vector [1].

Implementatie van een motion compensation algoritme is simpel en invariabel. Het gaat namelijk om een simpele matrix operatie, de rest van het rekenwerk wordt verricht in andere algoritmen.

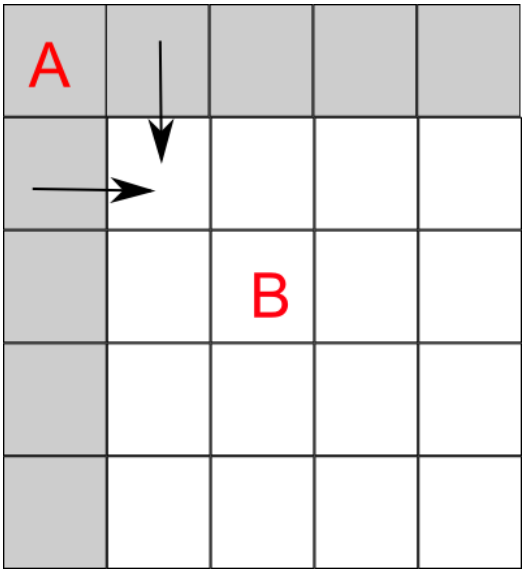
2.1.1.3 Spatiale predictie

Het grote verschil tussen spatiale en temporele predictie is dat het bronframe gelijk is aan het doelframe bij spatiale predictie. Om deze reden wordt spatiale predictie ook wel *Intra predictie* genoemd en temporele predictie *Inter predictie*.

Een belangrijk aandachtspunt bij intra predictie is dat niet ieder blok beschikbaar is om motion estimation en compensation op uit te voeren. Correcte matches kunnen enkel gevonden worden aan de hand van gedecomprimeerde blokken. Dit omdat de decoder enkel kan werken met gedecomprimeerde blokken en niet met originele ongecomprimeerde blokken, moest de encoder motion estimation en compensation verrichten op basis van originele ongecomprimeerde pixel waardes zou dit tot inaccurate resultaten lijden omdat het residu berekent zou worden op basis van deze blokken maar bij decompressie gebruikt zou worden in combinatie van gecomprimeerde blokken. De verwachte output bij encoding is dus anders dan de uiteindelijk gekregen output bij decoding. [1]

Bij inter predictie is dit geen probleem (tenzij er gewerkt wordt met forward prediction maar hier wordt in de thesis niet dieper op ingegaan) omdat alle blokken al gecomprimeerd zijn in een vorige stap van de encoder. Bij intra predictie is dit niet het geval. Blokken moeten namelijk vergeleken worden met blokken die nog gecomprimeerd dienen te worden in dezelfde stap, zie figuur 2.2. Bij het uitvoeren van intra predictie is de volgorde van compressie van de blokken dus van cruciaal belang om een goed resultaat te bekomen.

Implementatie van intra predictie is erg variabel. De meer geavanceerde encoders, zoals H.264, bieden de optie om ieder frame van een video te encoden met een verschillende intra predictie methode, om een zo optimaal mogelijk resultaat te bekomen bij compressie.



Figuur 2.2: Intra predictie van een frame, de grijze blokken (A) zijn al gecomprimeerd, de witte blokken (B) nog niet. Enkel de grijze blokken zijn beschikbaar voor intra predictie

2.1.2 Image model

Normaliter is er een hoge graad van correlatie tussen aangrenzende samples van een frame. Dit maakt het vaak zeer moeilijk om goede compressie te realiseren op frames in hun originele vorm. Het doel van het image model is de correlatie tussen samples zodanig te verminderen dat efficiënte compressie gerealiseerd kan worden op de frame.

In regel wordt gebruik gemaakt van drie stappen, een transformatie map om de correlatie van de data te verlagen en de grootte ervan te verkleinen, een kwantisatie stap om precisie van de data te verlagen en een herordeningsstap om belangrijke data te groeperen. [1]

In de volgende secties wordt dieper ingegaan op ieder van deze drie stappen. Allereerst wordt *Discrecte Cosine Transformation* uitgelegd als belangrijkste onderdeel van de transformatie stap. Hierna wordt er dieper ingegaan op de kwantisatie stap, waarna het herordenen wordt behandeld.

2.1.2.1 Transformatie

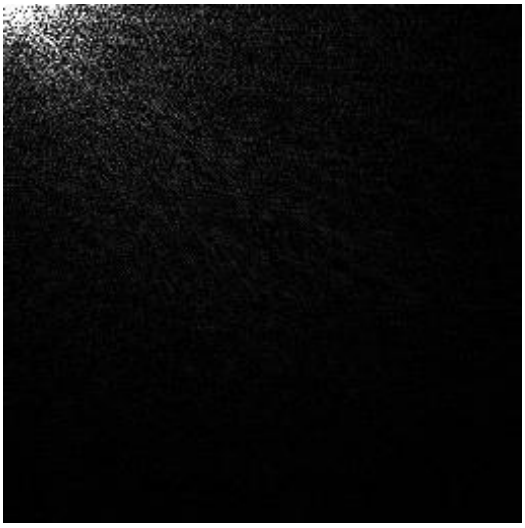
Het belangrijkste doel van de transformatie stap is om de data te converteren naar een gekozen domein opdat de correlatie tussen de data elementen in het nieuwe domein zo laag mogelijk ligt. Bovendien moet de transformatie

inverteerbaar zijn (noodzakelijk voor decompressie) en moet de rekenkundige complexiteit zo laag mogelijk zijn.

De belangrijkste en meest gebruikte techniek om dit te realiseren is de Discrete Cosine Transformation (DCT), een blok gebaseerde methode om afbeeldingen te transformeren. Er zijn nog vele andere technieken beschikbaar, de Karhunen-Loeve Transform, Discrete Wavelet Transform, ... In deze thesis wordt enkel dieper ingegaan op DCT omwille van relevantie voor het verder onderzoek en omwille van zijn populariteit bij veelgebruikte video codes (H.264, MPEG-4).

Discrete Cosine Transformation DCT transformeert een eindig aantal datapunten naar het frequentiedomein. Door data voor te stellen als een sommatie van een reeks cosinussen worden de spatiale eigenschappen van de data losgekoppeld van zijn energetische eigenschappen. Het resultaat hiervan is dat voor afbeeldingen met een hoge graad van correlatie (een extreem voorbeeld zijnde een uniform gekleurd vlak) na toepassing van DCT alle informatie in zeer weinig coëfficiënten samengevoegd is. Het is makkelijk in te zien dat compressie op deze gereduceerde coëfficiënten betere resultaten zal opleveren dan compressie op de gecorreleerde originele data. In figuren 2.3 en 2.4 is een voorbeeld van de DCT transformatie op een afbeelding te zien.

Figuur 2.3: De veelgebruite lena afbeelding [3]



Figuur 2.4: DCT transformatie van de lena afbeelding. Hier is duidelijk te zien hoe de meeste coëfficiënten zich vertonen in de linkerbovenhoek van de afbeelding, de anderen zijn bijna exclusief nulwaarden [3]

.

De DCT van een blok van NxN pixels kan als volgt berekend worden:

$$Y = AXA^T$$

hierin is X de matrix van originele pixelwaarden, Y de coëfficiënten na DCT. Ieder element van A wordt als volgt berekend:

$$A_{ij} = C_i \cos \frac{(2j+1)i\pi}{2N}$$

$$C_i = \sqrt{\frac{1}{N}}(i=0), C_i = \sqrt{\frac{2}{N}}(i>0)$$

Een groot voordeel van deze matrix operatie is dat A en A^T slechts eenmaal berekent moeten worden per blok van dimensies NxN. [1].

2.1.2.2 Kwantisatie

Tijdens kwantisatie (quantization) wordt een reeks waarden gemapt op een een nieuwe reeks waarden. Het doel van de kwantisatie stap is deze mapping zo op te stellen dat de nieuwe reeks waarden efficiënter gecomprimeerd kunnen worden dan de originele set elementen.

Een veelgebruikte methode hiervoor is scalaire kwantisatie. Hierbij wordt ieder sample van de originele samples gemapt op exact 1 waarde in de gekwantiseerde set van waarden. Binnen het image model kan kwantisatie gebruikt worden om de coëfficiënten bekomen uit de DCT transformatie verder te reduceren. De coëfficiënten na DCT stellen zoals al eerder aangehaald een reeks frequenties voor waaruit het originele beeld is opgebouwd, de linkerbovenhoek bevat de lage frequenties en de rechterbenedenhoek de hoge frequenties. Traditioneel is het menselijk oog veel gevoeliger voor de lage frequenties in een beeld dan de hoge. Het is dus perfect mogelijk om hogere frequenties te verwijderen of aan te passen zonder zichtbaar de kwaliteit van het beeld te schaden. Om dit te bereiken wordt scalaire kwantisatie toegepast op de DCT coëfficiënten:

$$Y = round(\frac{X}{QP}) \tag{2.1}$$

waarin Y de gekwantiseerde waarde is, X de originele coëfficiënt en QP de stap grootte van de kwantisatie.

Binnen de video compressie wordt vaak gebruik gemaakt van zogenaamde kwantisatiematrices. Figuur 2.6 toont een voorbeeld van een typische JPEG kwantisatie matrix. Door ieder element van de matrix met DCT coëfficiënten te kwantiseren zoals in berekening 2.1 met QP het overeenkomstige element in de gekozen kwantisatiematrix wordt een nieuwe matrix bekomen die verder gecomprimeerd kan worden. Door de linker boven elementen in de kwantisatiematrix relatief kleine waarden te geven en de rechter beneden elementen relatief hoge waarden te geven worden de hoge frequenties in het beeld naar 0 gekwantiseerd terwijl de belangrijke lage frequenties bewaard blijven [1]. Figuur 2.7 toont het resultaat na kwantisatie van figuur 2.5 met de matrix uit figuur 2.6. [1]

$$\begin{bmatrix} -415 & -33 & -58 & 35 & 58 & -51 & -15 & -12 \\ 5 & -34 & 49 & 18 & 27 & 1 & -5 & 3 \\ -46 & 14 & 80 & -35 & -50 & 19 & 7 & -18 \\ -53 & 21 & 34 & -20 & 2 & 34 & 36 & 12 \\ 9 & -2 & 9 & -5 & -32 & -15 & 45 & 37 \\ -8 & 15 & -16 & 7 & -8 & 11 & 4 & 7 \\ 19 & -28 & -2 & -26 & -2 & 7 & -44 & -21 \\ 18 & 25 & -12 & -44 & 35 & 48 & -37 & -3 \end{bmatrix}$$

Figuur 2.5: Voorbeeld matrix van coëfficiënten bekomen na DCT transformatie [4]

[1]Hierbij dient opgemerkt te worden dat bij kwantisatie informatie verloren gaat. De originele waarde kan nooit exact terug hersteld worden omdat de gekwantiseerde waarde zijn precisie afhankelijk is van de grootte van de kwantisatie stap

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

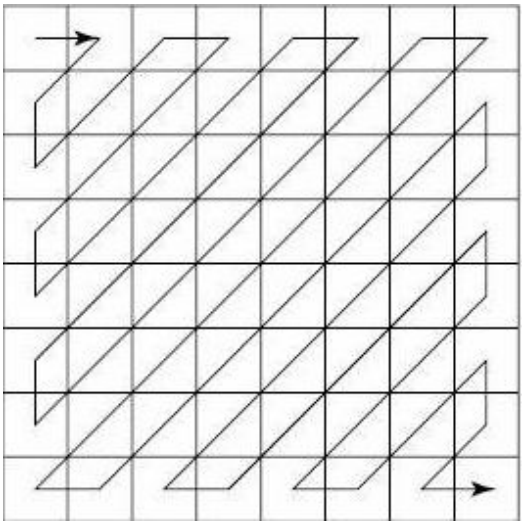
Figuur 2.6: Een typische JPEG kwantisatiematrix [4]

$$\begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -3 & 4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figuur 2.7: Resultaat van kwantisatie van matrix 2.5 door matrix 2.6 [4]

2.1.2.3 Run-length Encoding

Om zo efficiënt mogelijke compressie te bekomen kunnen de coëfficiënten bekomen na toepassing van de DCT transformatie en kwantisatie nog verder worden. Zoals te zien in figuur 2.7 bevat de matrix van coëfficiënten een bijzonder groot aantal nullen. De niet nul elementen zijn sterk gecentreerd in de linkerbovenhoek van de matrix. Een veelgebruikte techniek om deze typische matrices te comprimeren is door het uitvoeren van een zigzag scan op de matrix. Door de elementen van de matrix uit te lezen zoals aangeven in figuur 2.8 wordt er een reeks getallen bekomen bestaande uit een reeks niet-nul elementen gevolgd door nul elementen.



Figuur 2.8: Volgorde waarin elementen van een matrix worden uitgelezen tijdens een zigzag scan [5]

Op deze reeks van getallen kan men dan zogenaamde run-length encoding toepassen (RLE). Bij run-length encoding wordt ieder element van een reeks getallen, het zogenaamde *level*, vooraf gegaan van een waarde die aangeeft hoeveel nul elementen dit element voorafgaan, de zogenaamde *run*. Door het hoge aantal opeenvolgende nul elementen na het uitvoeren van een zigzag scan op het resultaat van de DCT en kwantisatie operaties is RLE een goede manier om de hoeveelheid te comprimeren informatie te reduceren [1].

2.1.3 Entropy coder

De entropy coder converteert een reeks symbolen, in het geval van een video de symbolen gegenereerd door het prediction en image model, naar een bitstream die gebruikt kan worden voor transport en opslag. Hiervoor wordt gebruik gemaakt van traditionele compressietechnieken, zoals predictive coding, variable-length coding, ... [1]

Veel van de gebruikte technieken in deze fase zijn moeilijk te parallelliseren, bovendien valt het onderzoek hiernaar buiten de scope van de thesis. Een paper die parallellisatie van deze algoritmen bestudeert is *Parallel Algorithms for Entropy-Coding Techniques* [6], hierin worden zowel voor arithmetic coding als Huffman coding parallelle algoritmen voorgesteld.

2.2 Parallellisatie van video encoding algoritmen

In deze sectie wordt een overzicht geschetst van de verschillende methodes om eerder genoemde algoritmen te parallelliseren. Allereerst wordt een analyse gemaakt op welke manier parallellisatie opgevat kan worden wanneer men over video spreekt. Vervolgens worden bestaande methodes aangehaald om veel van de bovenstaande algoritmen te parallelliseren.

2.2.1 Soorten van parallellisatie voor video

Als men over parallellisatie van video spreekt kan men de methodes opdelen in twee categorieën, een eerste categorie is parallellisatie op het niveau van frames of Group Of Pictures (GOP). Een tweede categorie zijn algoritmen die parallellisatie implementeren op het macroblok of slice niveau[7]. Elk van deze algoritmen hebben hun eigen voor- en nadelen, waarop in volgende secties dieper ingegaan wordt.

2.2.1.1 GOP-level parallellisatie

Bij GOP-level parallellisatie is iedere rekeneenheid verantwoordelijk voor het encoden van een sequentie van frames. De sequentie van frames is beperkt tot het GOP omdat er op deze manier geen onderlinge communicatie tussen de rekeneenheden nodig is buiten op het einde van de encoding, bij het terug samenvoegen van het uiteindelijk resultaat. Iedere rekeneenheid kan wel verantwoordelijk zijn voor de encoding van verschillende GOPs, op deze manier kan de graad van parallellisatie aangepast worden.

Omdat iedere rekeneenheid verantwoordelijk is voor encoding van 1 I-frame en bijhorende P-frames en eventueel B-frames is het grootste nadeel van deze aanpak dat hij niet bruikbaar is voor alle realtime video. Om namelijk parallellisatie te realiseren moet er een reeks van GOPs beschikbaar zijn om te encoden. Bij realtime video worden de GOPs van de video sequentieel ontvangen en is het moeilijk om encoding ervan te parallelliseren. Indien de requirements voor realtime streaming zeer streng zijn bestaat de mogelijkheid dat GOP parallellisatie niet gebruikt kan worden.

Een groot voordeel van de aanpak is dat er geen communicatie nodig is tussen de verschillende rekeneenheden tijdens de verschillende stappen van de encoding van een GOP. Om deze techniek toe te passen dienen er dus weinig tot geen aanpassingen te gebeuren aan de algoritmen die gebruikt kunnen worden voor sequentiële video encoding.

2.2.1.2 Slice-level parallellisatie

Bij slice-level parallellisatie wordt ieder frame opgedeeld in reeks van macroblokken (zie verder), de zogenaamde slices. Iedere rekeneenheid is verantwoordelijk voor de encoding van een slice. De grootte van de slice is afhankelijk van de wijze waarop de techniek geïmplementeerd is en kan reiken van 1 macroblok per slice tot alle macroblokken van het frame gebundeld in 1 slice.

Deze techniek heeft als grote voordeel dat hij wel bruikbaar is voor realtime video. Parallellisatie wordt namelijk gerealiseerd per frame. Iedere ontvangen frame kan dan onafhankelijk parallel geëncodeerd worden zonder de videostroom te onderbreken.

Het nadeel hiervan is dat er veel meer communicatie nodig is tussen de verschillende stappen van het video encoding proces. Na encoding van een frame dient namelijk de gecomprimeerde datastroom verwerkt worden (opslaan en/of doorsturen) en informatie uitgewisseld te worden om encoding van de volgende frame te starten. Dit vraagt vaak grotere aanpassingen aan de sequentiële algoritmen en voegt meer overhead toe dan bij GOP-level parallellisatie door de toegenomen communicatie.

In de volgende secties worden methodes besproken om parallellisatie te realiseren van de typische stappen het video encoding proces. Omdat bij GOP-level parallellisatie weinig tot geen aanpassingen nodig zijn aan de gebruikte algoritmen ligt de focus in deze secties op de implementatie van slice-level parallellisatie.

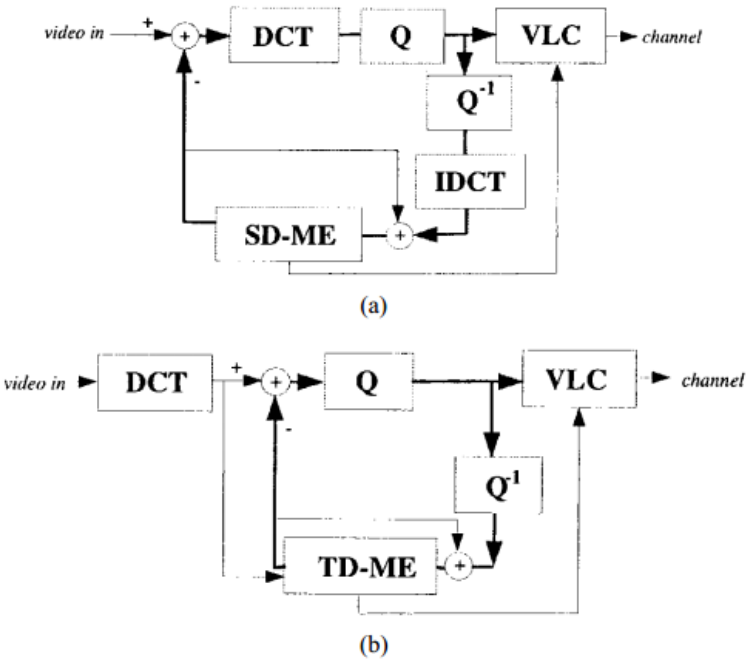
2.2.2 Slice-level parallellisatie van het prediction model

In deze sectie lijsten we een reeks algoritmen en methodes op die gebruikt kunnen worden om de stappen binnen het prediction model te parallelliseren. De voorgesteld technieken kunnen in regel opgedeeld worden in 2 categorieën. Een eerste categorie zijn algoritmen die focussen op implementatie van algoritmen die geschikt zijn voor gebruik in off-the-shelf hardware. Een tweede, veelvoorkomende categorie zijn technieken die hardware samenstellen geoptimaliseerd om het algoritme op uit te voeren. Dit levert in regel betere resultaten op maar is moeilijk toepasbaar in iedere scenario.

Motion estimation In *An Efficient Parallel Motion Estimation Algorithm for Digital Image Processing*[8] wordt een techniek voorgesteld om een match te vinden door de motion vector onafhankelijk te zoeken over 2-assen (X en Y). In de paper wordt het probleem aangehaald dat voor Full Search (FS)

enkel special hardware beschikbaar is om parallellisatie te realiseren. Andere technieken, zoals Three-step search (TSS) hebben als nadeel dat het algoritme vaak onderbroken wordt om te wachten op voltooiing van een vorige stap, wat parallellisatie benadeelt. De voorgestelde oplossing vermijdt het probleem van TSS maar heeft wel voldoende lage complexiteit om nog uitvoerbaar te zijn op off-the-shelf hardware.

A Complete Pipelined Parallel CORDIC Architecture for Motion Estimation[9] stelt een block-matching motion estimation techniek voor die werkt op het DCT transformatie domein in plaats van het spatiale domein. Op deze manier wordt hogere throughput en lagere hardware complexiteit gerealiseerd. In de figuur 2.9 is in (a) een traditionele encoder lus te zien. Hierbij staat het SD-ME component voor *Spatial Domain Motion Estimation*. In (b) is de flow van de geoptimaliseerde oplossing voorgesteld. Hier staat TD-ME voor (Temporal Domain Motion Estimation), hierbij wordt motion estimation gerealiseerd in het temporele domein. Hierdoor is het niet langer noodzakelijk om het IDCT algoritme op te nemen in de motion estimation flow wat de complexiteit van het algoritme aanzienlijk verlaagt. Door deze techniek te combineren met speciale hardware wordt een geoptimaliseerde methode gerealiseerd.



Figuur 2.9: In (a) de traditionele ME lus in het spatiale domein. In (b) de ME lus voorgesteld in [9]

In *Parallel Implementation of the Full Search Block Matching Algorithm for Motion Estimation*[10] wordt een techniek voorgesteld om het FS algoritme te parallelliseren. Dit wordt gerealiseerd door binnen een frame per referentieblok in parallel van alle mogelijk blokken in de search area de MAD te berekenen en te vergelijken. Door gebruik te maken van gespecialiseerde hardware wordt het mogelijk om het grote aantal parallelle berekeningen hiervoor nodig uit te voeren [2]

Motion compensation In vele gevallen wordt motion compensation beschouwd als deel van motion estimation. Apart beschouwd bestaat motion compensation nu eenmaal uit niet meer dan één matrix operatie [3], zie 2.1.1.2. Deze matrix operatie kan dan geparallelliseerd worden op frame niveau als zijnde het parallel berekenen van het residublok van ieder macroblok in een frame of op macroblok niveau als zijnde het parallel berekenen van ieder element van het residublok in parallel. Natuurlijk kan een combinatie van beide gebruikt worden om zo hoog mogelijke parallellisatie te realiseren.

Intra prediction Omdat intra predictie gebruik maakt van de interdependentie tussen macroblokken binnen hetzelfde frame moet er voldoende informatie beschikbaar zijn (in de vorm van eerder geëncodeerde macroblokken) voordat een macroblok geëncodeerd kan worden. Bij parallellisatie van deze operatie resulteert dit in het moeten stallen van parallelle operaties tot meer informatie beschikbaar wordt. De meest parallelle intra predictie algoritmen focussen zich dan ook op het reduceren van de latency geassocieerd met het moeten wachten op het encoden van noodzakelijke macroblokken.

In *A Parallel and Pipelined Execution of H.264/AVC Intra Prediction*[11] en *Pipelined Intra Prediction Using Shuffled Encoding Order for H.264/AVC*[12] worden beiden technieken voorgesteld om de intra predictie modules van de H.264/AVC te parallelliseren. Dit wordt gerealiseerd in beide methodes door de encoding volgorde van de macroblokken op zo een manier te veranderen dat de latency veroorzaakt door interdependentie van macroblokken verlaagd wordt.

2.2.3 Slice-level parallellisatie van het image model

In deze sectie wordt er gefocust op de parallellisatie van de algoritmen binnen het image model. Een groot voordeel van deze algoritmen is dat er geen

[2] Er moeten namelijk voor een ME step size van p in parallel $(2p+1)^2$ berekeningen kunnen gebeuren om deze methode aantrekkelijk te maken.

[3] In de assumptie dat het om block-based motion estimation gaat

interdependentie is tussen de macroblokken van een frame bij uitvoering van deze algoritmen. Bij het parallelliseren hiervan is het dan ook makkelijk om op slice-level parallellisatie te realiseren daar iedere slice onafhankelijk verwerkt kan worden van anders slices.

In de volgende paragrafen ligt de focus dan ook eerder op methodes die specifieke specificaties van deze algoritmen gebruiken om een nog hogere graad van parallellisatie te realiseren.

DCT In *Analytical Evaluation of the 2D-DCT using paralleling processing*[13] wordt een techniek voorgesteld om het DCT algoritme zelf te parallelliseren. Omdat een 2D DCT transformatie voorgesteld kan worden als twee 1D DCT transformaties is het mogelijk het DCT algoritme verder te parallelliseren. Een 1D DCT transformatie kan namelijk als volgt in pseudo-code uitgeschreven worden:

```
for (i=0; i < N; i++)
{
        for (j=0; j < N; j++)
            Y[i] = Y[i] + x[j] * cos((2j+1)*i*π/2N)
        Y[i] = 2 * Y[i] / N
}
```

Deze lussen zijn makkelijk te parallelliseren en een combinatie van geparallelliseerde 1D DCT transformaties kan een geparallelliseerde 2D DCT transformatie gerealiseerd worden.

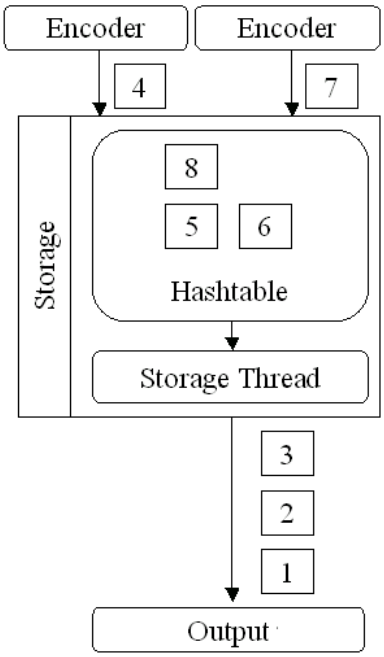
Kwantisatie De aarde van het kwantisatie algoritme, namelijk een per element operatie binnen een macroblok, geeft aan dat een hogere graad van parallellisatie gerealiseerd kan worden door simpelweg ieder element van het macroblok in parallel te kwantiseren. Veel onderzoek is dan ook niet terug te vinden rond parallellisatie van de kwantisatie stap, daar de hoogst mogelijke graad van parallellisatie triviaal af te leiden is.

RLE Het standaard RLE algoritme is geen triviaal algoritme om te parallelliseren. Het grote probleem met parallellisatie van het RLE algoritme is dat het niet triviaal is om vast te stellen op welke manier een datastroom gesplitst moet worden over de parallelle rekeneenheden. Om maximale compressie te realiseren steunt het algoritme nu eenmaal op een volledig sequentiële verwerking van de datastroom.

In *Parallel algorithm for run length encoding*[14] wordt een algoritme beschreven dat voor deze twee problemen een oplossing biedt. Het eerste probleem wordt opgelost door de datastroom op te delen in een aantal even

grote sub-stromen. Dit heeft als resultaat wel dat er geen maximale compressie bereikt wordt maar dit weegt niet op tegen het voordeel van deze opsplitsing. Het tweede probleem is moeilijker op te lossen, het algoritme stelt voor om met een intermediaire hashtable te werken. Wanneer een substroom gecomprimeerd is door het algoritme wordt deze opgeslagen in de table met als key een identifier die zijn volgorde in de originele datastroom aangeeft. Het samenvoegen in het uiteindelijke geheugen dient wel sequentieel te gebeuren maar draagt slechts bij tot een klein gedeelte van de kost van het volledige algoritme. Een flowchart van het algoritme is terug te vinden in figuur 2.10.

Andere algoritmen, zoals hetgeen uitgewerkt in [15] steunen op eenzelfde methodologie als in [14], door de datastroom op te splitsen in delen van een gekozen lengte en zo te verwerken.



Figuur 2.10: De parallelle RLE methode zoals voorgesteld in [14]

2.3 GPU computing voor video encoding

In deze sectie wordt gefocust op de rol van GPU computing binnen video encoding. Allereerst wordt er kort overzicht geschetst wat GPU computing exact is, gevolgd door een overzicht van bestaande implementaties van video encoders op de GPU. Tenslotte wordt er dieper ingegaan op enkele stappen van het video encoding proces en welke algoritmen daarvoor beschikbaar zijn met implementatie op de GPU.

2.3.1 Wat is GPU computing?

Op de website GPGPU.org [16] wordt GPU computing als volgt beschreven:

> GPGPU stands for General-Purpose computation on Graphics Processing Units, also known as GPU Computing. Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput. Once specially designed for computer graphics and difficult to program, today's GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C. Developers who port their applications to GPUs often achieve speedups of orders of magnitude vs. optimized CPU implementations.

In het kort samengevat betekent GPU computing het implementeren van software met als primaire rekeneenheid niet de CPU maar de GPU. Vaak spreekt men vaak pas echt over GPU computing als de software de originele doelen van de GPU overstijgt. Hiermee wordt bedoelt dat de software voor andere doelen ingezet wordt dan bijvoorbeeld vertex berekeningen, texture mapping, ... Voorbeelden van GPU computing houden in maar zijn niet beperkt tot, physics simulation, image processing, molecular dynamics, ...

2.3.2 Bestaande oplossingen

In deze sectie worden enkele totaal oplossingen besproken die video encoding realiseren op de GPU. De bedoeling van deze sectie is niet zozeer een academische achtergrond te scheppen rond de GPU computing implementatie van de verschillende deelstappen van video encoding, maar eerder een overzicht te bieden van oplossingen die al gerealiseerd zijn. Dit heeft als doel aan te tonen dat het niet onmogelijk is om video encoding te realiseren op de GPU en dat dit wel degelijk een commerciële waarde heeft.

In deze sectie worden twee verschillende types van oplossingen besproken, closed-source commerciële oplossingen en open-source oplossingen gerealiseerd in het kader van onderzoek. De meeste oplossingen zijn te vinden in de closed-source categorie.

2.3.2.1 Closed-source oplossingen

Cyberlink MediaEspresso Cyberlink MediaEspresso[17] is één van de bekendere video encoders. Eén van de grote voordelen van MediaEspresso is het grote aantal ondersteunde input en output types en het grote aantal manieren om de encoding te realiseren. Zo wordt ook GPU computing gebaseerd encoderen ondersteunt op basis van AMD APP en NVIDIA CUDA.

FastVideo JPEG encoder FastVideo JPEG encoder[18] realiseert een GPU computing gebaseerde video encoder op het NVIDIA CUDA platform. De encoder implementeert enkel JPEG compressie (en mist op deze manier enkele interessante features zoals motion estimation, ...). Ze claimen compressie ratios tot 10 à 20 maal en encoding snelheden tot 3300 MB/s [4].

NVCUVENC NVCUVENC[19] is de video encoding API van NVIDIA. Deze realiseert H.264 encoding in CUDA. NVUCENC werkt op twee verschillende methodes, partial GPU offload en full GPU offload. Bij partial GPU offload wordt enkel de motion estimation stap uitgevoerd op de GPU, bij full GPU offload worden de meeste berekeningen uitgevoerd op de GPU, enkel de entropy encoding wordt nog uitgevoerd op de CPU.

De partial offload methode kan ongeveer 120 FPS encoderen [5] terwijl de full offload methode ongeveer 60 FPS kan encoderen.

2.3.2.2 Open-source oplossingen

Parallelization of the x264 encoder using OpenCL Deze open-source implementatie van de x264 encoder in OpenCL is gerealiseerd door Erich Marth en Guillermo Marcus in het kader van onderzoek naar het gebruik van de GPU voor video encoding algoritmen[20].

In een eerste stap werd motion estimation gerealiseerd in OpenCL. Dit door een implementatie te voorzien gebaseerd op TSS en FS. Het motion estimation component werd dan geïntegreerd in de x264 encoder waarna optimalisaties aangebracht werden om de encoder beter af te stemmen op de gebruikte architectuur. In een laatste stap werden ook de transformatie (DCT)

[4] Op een NVIDIA GeForce GTX 580

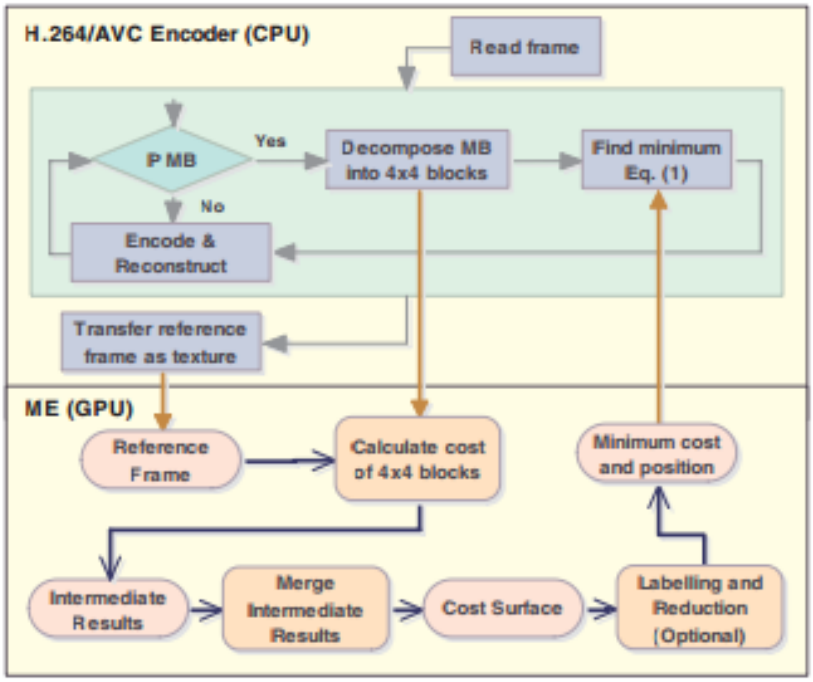
[5] Op een Quadro FX 4800 met als input een 1920x1080 video

en kwantisatie componenten geïmplementeerd in OpenCL en geïntegreerd in de gerealiseerde pipeline.

Het eindresultaat leverde snelheidsverbeteringen op tot 55%, de opmerking werd wel gemaakt dat slechts een beperkt deel van de motion estimation capaciteiten van x264 geport was naar OpenCL.

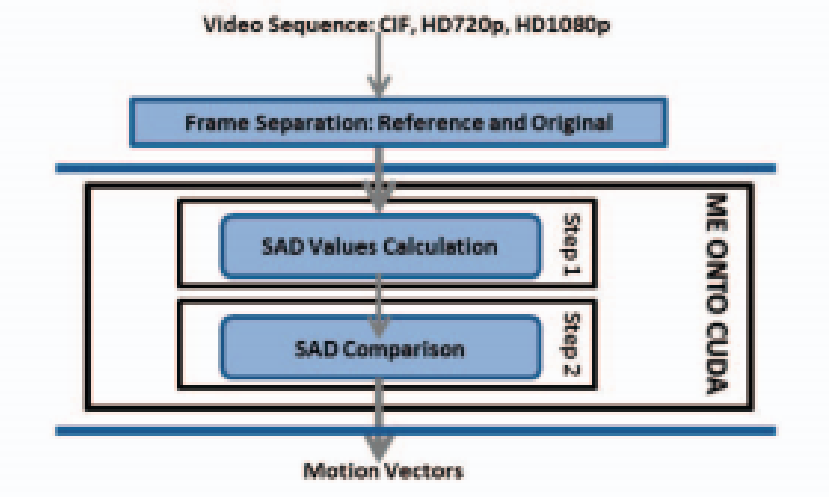
2.3.3 Bestaande algoritmen

Motion estimation In *Motion Estimation for H.264/AVC using Prgrammable Graphics Hardware*[21] wordt een shader gebaseerde oplossing voorgesteld om de motion estimation algoritmen van H.264/AVC te parallelliseren. Dit wordt gerealiseerd door 3 componenten te implementeren. Een eerste component verantwoordelijk voor het berekenen van de MAD van 4x4 blokken. Een tweede component dat de tijdelijke resultaten samenvoegt en de finale kost van het macroblok berekent en ten slotte een component verantwoordelijk voor het bepalen van het blok met minimale kost en zijn bijhorende motion vector. Op deze manier werd een ME algoritme gerealiseerd dat efficiënter was dan de CPU implementatie zowel op consumer als high-end hardware. Een diagram die de uitvoering van het algoritme toont is te zien in figuur 2.11.



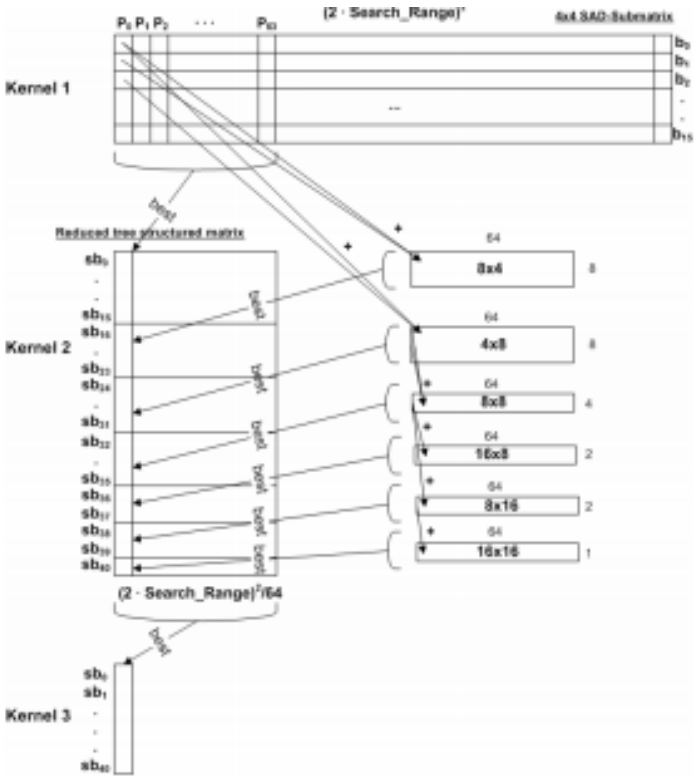
Figuur 2.11: De ME techniek zoals voorgesteld in [21]

In *Real-time Block Matching Motion Estimation onto GPGPU*[22] wordt een andere techniek voorgesteld. De focus in deze paper ligt op het Full Search en het Diamond Search algoritme. Door het algoritme op te splitsen in twee delen, een eerste component verantwoordelijk voor het berekenen van de SAD (Sum Absolute Difference) waarde per blok en een tweede component voor het vergelijken van de eerder berekende SAD waarden. Dit wordt gerealiseerd binnen één kernel waarbinnen alle eerder genoemde componenten opgenomen zijn. Ook hier worden zeer merkbare verbeteringen in de uitvoeringssnelheid van het algoritme vastgesteld tot een factor 10. Een diagram die de uitvoering van het algoritme toont is te zien in figuur 2.12



Figuur 2.12: De ME techniek zoals voorgesteld in [22]

De methode beschreven in *Accelerating H.264 inter prediction in a GPU by using CUDA*[23] steunt op het gebruik van drie kernels. Een eerste kernel die een 16x16 macroblok opdeelt in 4x4 blokken en van deze sub-blokken de SAD berekent. Een tweede kernel die de totale SAD van het 16x16 macroblok berekent door de 4x4 macroblokken samen te voegen in een reeks intermediaire SAD kosten. De derde kernel staat in voor het reduceren van de intermediaire SAD waarden tot 1 SAD waarde, die gebruikt kan worden in het H.264 ME algoritme. Een diagram die de uitvoering van het algoritme toont is te zien in figuur 2.13.



Figuur 2.13: De ME techniek zoals voorgesteld in [23]

Het algoritme voorgesteld in *Parallelizing H.264 Motion Estimation Algorithm using CUDA*[24] stelt een implementatie voor op CUDA die gebruik maakt van een hiërarchisch zoekpatroon om de motion vector per macroblok te bepalen. Een eerste ruwe match wordt gezocht in een zwaar gedownsamplede versie van het originele beeld en de gevonden motion vector wordt steeds verfijnt door de match te verbeteren in minder gedownsamplede beelden, tot men uiteindelijk een match vindt in het originele beeld. Er wordt één kernel uitgevoerd per niveau binnen de hiërarchie, de gevonden motion vectors worden bewaarde op het device (om memory transfers te beperken) en gebruikt in de uitvoering van de volgende kernel. Ook hier wordt een zeer merkbare snelheidsverbetering opgemerkt, maar wordt wel opgemerkt dat het moeilijk is om te vergelijken met bestaande CPU based implementaties (zoals x264) omdat deze meer accurate ME algoritmen gebruiken.

Intra prediction In *Intra Frame Encoding Using Programmable Graphics Hardware*[25] wordt een techniek voorgesteld die de 4x4 intra predictie kan uitvoeren van de H.264/AVC standaard. Door de encoding volgorde van de blokken te veranderen en licht wijzigingen aan te brengen in de architectuur

van de H.264/AVC codec werd een implementatie gerealiseerd die tot 30 keer sneller werkt dan de standaard CPU implementatie.

In de thesis *Video coding on multicore graphics processors (GPUs)*[26], die zich focust op de implementatie van de H.264/AVC intra prediction modules met CUDA, wordt ook aangetoont dat GPU computing een goed alternatief vormt voor de uitvoering van intra predictie. Met vastgesteld verbeteringen in snelheid tot 11 maal en bovendien met betere schaalbaarheid. Wel wordt opgemerkt dat een voldoende hoge spatiale resolutie (minstens 1080p) nodig is om de latency veroorzaakt door memory transfers van host naar device te compenseren.

DCT In *Techniques for Efficient DCT/IDCT Implementation on Generic GPU*[27] wordt geanalyseerd welke methodes geschikt zijn om DCT/IDCT te realiseren geoptimaliseerd voor de GPU. Er wordt geclaimd dat vele methodes die door de jaren heen geoptimaliseerd zijn voor de CPU weinig geschikt zijn voor de GPU. De gerealiseerde implementatie focust dan ook eerder op de directe matrix multiplicatie, omwille van het feit dat de nodige operaties hiervoor erg goed ondersteunt worden door de GPU. Hun verschillende implementaties leveren dan ook elk verbeteringen op het vlak van snelheid op ten opzichte van een standaard CPU implementatie.

2.4 Hardware voor video encoding

Ook al ligt de focus in deze thesis op het realiseren van een video encoder op de GPU wordt er ter volledigheid in deze sectie dieper ingegaan op hardware gebaseerde oplossingen voor video encoding. In een eerste deel worden de standalone oplossingen besproken. In een tweede deel wordt dieper ingegaan op PCIe kaarten gemaakt voor hardware encoding.In een laatste deel wordt NVENC besproken, de nieuwe hardware gebaseerde oplossing aanwezig op de nieuwere NVIDIA kaarten.

2.4.1 Standalone hardware voor video encoding

Standalone hardware kan volledig zelfstandig werken, er is dus geen extra apparatuur nodig. Het nadeel is dan weer dat veel van dit soort hardware vaak beperkt is in zijn functionaliteit. Het grote nadeel is dan wel dat standalone hardware vaak enkel gebruikt kan worden voor video encoding en geen andere zaken, in tegenstelling tot de software gebaseerde oplossingen. Bovendien is er vaak een vrij hoge prijskaart verbonden aan dit soort hardware, zeker omdat het vaak single-purpose hardware is. Als voorbeeld voor dit type

hardware zijn er de *Elemental Server* [28], een dedicated H.264 encoder die ingebouwd kan worden in een server rack en de *Digital Rapids TouchStream* [29], waarvan de focus meer ligt op draagbaarheid en het encoden van live video. Natuurlijk zijn er nog alternatieven beschikbaar binnen hetzelfde type hardware maar deze twee zijn een vrij goede representatie van de meerderheid van de beschikbare toestellen.

2.4.2 PCIe hardware voor video encoding

Een ander type hardware zijn PCIe addon cards dedicated voor video encoding [30] [31] [32]. Deze kaarten worden in regel geleverd met een software pakket dat de video stromen naar de addon kaart stuurt waar deze dan geëncodeerd wordt. Het grote voordeel van dit soort kaart is dat ze weinig tot geen rekentijd innemen op de CPU, deze blijft dus beschikbaar om andere intensieve taken uit te voeren. Bovendien is de prijskaart verbonden aan dit type hardware een pak lager dan bij de standalone hardware. Een nadeel is dat de performantie in regel een pak lager ligt dan bij de standalone hardware. Ook kan men de performantie van de kaart niet verhogen. Als men betere performantie wenst dient een snellere (en vaak ook duurdere) kaart gekocht te worden. Dit neemt niet weg dat het vaak een goede oplossing is voor mensen die frequent video dienen te comprimeren zonder de performantie van de computer te beïnvloeden op een kleiner budget.

2.4.3 NVENC

Een belangrijke nieuwe toevoeging op het vlak van hardware based video encoding is NVENC[33]. NVENC is een extra hardware component op de nieuwe NVIDIA GPU's (Kepler of nieuwer) die het mogelijk maakt om H.264 video stromen te encoderen. De performantie ligt tamelijk hoog, 240 fps voor een 1920x1080 video stroom. Het grote voordeel van NVENC is dat het net zoals bij de PCIe hardware geen CPU of GPU rekentijd ingenomen moet worden voor de encoding van video. Tijdens het encoden zijn dus nog quasi alle resources beschikbaar voor andere rekenwerk. Bovendien is NVENC automatisch aanwezig in de nieuwere NVIDIA grafische kaarten. Wanneer een gebruiker dus opteert voor een high-end consumer level PC zal deze automatisch beschikken over hardware video encoding. In de toekomst zal vermoedelijk zelfs de mid- en low-end NVIDIA grafische kaarten over NVENC beschikken en wordt hardware encoding zeer makkelijk te verkrijgen zonder extra aankopen te doen. Een groot nadeel is dan ook weer dat de performantie vast ligt. Op dit moment is de performantie van alle NVENC chips identiek, dit zal vermoedelijk wel veranderen in de toekomst maar dan

nog dient de volledige grafische kaart vervangen te worden om performantie te upgraden.

Hoofdstuk 3

Gebruikte technieken

In dit hoofdstuk ligt de nadruk op het argumenteren van de gemaakt keuzes tijdens de implementatie van de video encoder.

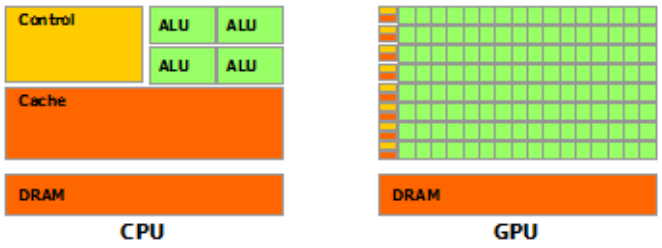
In een eerste sectie wordt beargumenteerd waarom er gekozen is voor NVIDIA CUDA als platform om de GPU computing implementatie te realiseren. Er wordt vergeleken met de andere mogelijkheden en er wordt een grondige uiteenzetting gegeven van de mogelijkheden van CUDA, teneinde hier niet meer op te moeten terugkomen tijdens de uitleg over de feitelijke implementatie.

Hierna wordt uiteengezet welke componenten zijn opgenomen in de basis CPU implementatie, deze is namelijk noodzakelijk om een baseline op te stellen voor het vergelijken van de performantie met een GPU implementatie.

In een laatste sectie wordt aangehaald welke componenten een implementatie op de GPU hebben gekregen en op welke manier dit gerealiseerd is.

3.1 NVIDIA CUDA

Door de constante evolutie van de GPU, gedreven door de vraag naar steeds betere realtime 3D graphics, is deze steeds meer naar de voorgrond gekomen als mogelijke hardware om rekenwerk op te verrichten. Door de manier waarop een GPU ontworpen is, namelijk gefocust op het in parallel uitvoeren van rekenintensieve taken in tegenstelling tot een CPU, die eerder focust op memory caching en flow control, is deze uitermate geschikt om taken uit te voeren die gebaat zijn met een hoge graad van parallellisatie. Voorbeelden hiervan zijn terug te vinden in signaal verwerking, beeldverwerking, ...



Figuur 3.1: Typisch design van een CPU vs een GPU

In 2006 introduceerde NVIDIA daarom CUDA (Compute Unified Device Architecture), een platform dat general purpose computing op NVIDIA GPU's diende te vergemakkelijken door een framework aan te bieden dat programmeurs zonder een al te hoge leercurve konden gebruiken maar nog steeds alle voordelen bood van het implementeren van software op de GPU. CUDA is enkel beschikbaar op NVIDIA GPU's, wat een nadeel is indien een het geschreven programma bruikbaar moet zijn op zoveel mogelijk hardware configuraties. Dit weegt echter niet op tegen de voordelen die CUDA biedt, hier wordt dieper op ingegaan in de volgende sectie[34].

3.1.1 Waarom NVIDIA CUDA?

Om de implementatie te realiseren werd voor deze thesis gekozen voor NVIDIA CUDA. De keuze hiervoor ten nadele van andere alternatieven wordt in deze sectie uitgebreid toegelicht. Dit wordt gedaan door per alternatief de voor- en nadelen af te wegen en aan te halen waarom uiteindelijk CUDA gekozen werd.

Er zijn 2 belangrijke alternatieven beschikbaar voor CUDA, werken met OpenGL/DirectX shaders en textures en OpenCL. Door de jaren heen zijn er een aantal frameworks geweest, zoals AMD FireStream, DirectCompute, ..., maar hedendaags zijn het deze twee alternatieven die overblijven als concurrentie voor CUDA.

3.1.1.1 OpenGL/DirectX

Een eerste alternatief is het werken met shaders in OpenGL/DirectX. Voor het ontstaan van frameworks zoals CUDA en OpenCL was de enige optie om GPU computing te realiseren via het schrijven van custom shaders. Door deze shaders uit te voeren op textures die de data bevatte die verwerkt diende te worden konden zo algoritmen uitgevoerd worden die niet behoorden tot de standaard functionaliteit van frameworks zoals OpenGL/DirectX.

Het voordeel van deze aanpak is dat de geschreven code werkt op alle platformen en configuraties [1]. CUDA is zoals eerder aangehaald beperkt tot hardware configuraties met een NVIDIA GPU. Dit voordeel is echter van niet zo een groot belang in de context van deze thesis. Het programma dat gerealiseerd dient te worden is eerder van de aard van een proof-of-concept. Moest het einddoel van de thesis het converteren naar GPGPU van een open source encoder, zoals x264, zijn geweest dan zou deze requirement veel belangrijker zijn geweest. De focus ligt eerder op het kiezen voor een framework dat een zo efficiënt mogelijke implementatie toelaat van een video encoder, en net hier ligt het grote nadeel van werken met shaders.

Het implementeren van een GPGPU programma via shaders is namelijk een complex gebeuren waarvoor allerhande kunstgrepen uitgehaald dienen te worden. Als men data naar de GPU wilt sturen moet die namelijk in textures geplaatst worden die dan verwerkt kan worden door de shaders. Het is makkelijk in te zien dat voor complexe operaties, waaruit een video encoder bestaat, deze manier van werk snel onoverzichtelijk kan worden. Wanneer men dan de keuze heeft voor frameworks die de programmeur op een meer vertrouwde manier laten werken met GPU computing code is de keuze snel gemaakt voor deze alternatieve.

Bovendien zijn de mogelijkheden van deze aanpak zeer beperkt, CUDA heeft bijvoorbeeld een heleboel extra manieren om geheugen aan te spreken, en ontbreken er vele geavanceerde features die betere frameworks in de loop van de tijd gekregen hebben [35].

3.1.1.2 OpenCL

OpenCL[36] is net zoals CUDA een framework gemaakt om ontwikkeling van software voor de GPU simpeler te maken. Ontwikkelt door de Khronos Group, OpenCL is een volledig open, cross platform framework om aan parallel computing te doen.

Net zoals bij ontwikkeling voor shaders is het voordeel van OpenCL dat het volledig cross platform is. Opgemerkt moet wel worden dat de support voor OpenCL op AMD beter is (AMD APP is namelijk het AMD specifieke alternatief voor CUDA, maar is gebaseerd op OpenCL) dan op andere platformen, maar omdat het een standaard is ondersteunen alle platformen de beschreven functionaliteit uit de standaard. De argumentatie om dit voordeel niet te laten doorwegen op de uiteindelijke keuze is identiek aan de argumentatie gemaakt in 3.1.1.1.

Het nadeel van OpenCL ten opzichte van CUDA is dat de toolset be-

[1] Bij een keuze voor DirectX is enkel Microsoft Windows officieel ondersteund.

schikbaar voor ontwikkeling beter uitgewerkt is in het geval van CUDA. Dit maakt de drempel om software te ontwikkelen en te debuggen een pak lager dan bij OpenCL. Omdat voor de realisatie van deze thesis rekening gehouden moest worden met beperkte tijd en beperkte voorkennis werd dan ook gekozen voor het framework met de betere toolset, teneinde de implementatie zo vlot mogelijk te laten verlopen.

3.1.2 Het CUDA programming model

Voordat er dieper ingegaan wordt op de gerealiseerde implementatie is het nodig een goed overzicht te geven van de manier van werken van CUDA. Dit is namelijk noodzakelijk om veel van de aangehaald concepten in de implementatie te begrijpen. In deze sectie wordt dan ook het programming model van CUDA uitgelegd, beperkt tot de relevante aspecten binnen deze thesis. Alle informatie uit deze sectie is terug te vinden in [38], waar de relevante informatie uit gehaald is.

3.1.3 Kernels

Kernels zijn stukken C code die eens aangeroepen door N verschillende CUDA threads in parallel worden uitgevoerd. Een kernel wordt gedefinieerd door de *__global__* specifier.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Figuur 3.2: Voorbeeld van een CUDA Kernel [38]

Kernels worden aangeroepen met behulp van de <<< ... >>> symbolen. De argumenten tussen deze symbolen geven aan hoeveel threads er aange-

maakt dienen te worden (zie 3.1.4) en optioneel de grootte van het *shared memory* (zie 3.1.5).

Belangrijk om op te merken is dat kernels enkel gelaunched kunnen worden vanuit *host code* (oftewel code die op de CPU runt). Enkel GPU's met compute capability 3.5 of hoger ondersteunen *Dynamic Parallelism* wat nodig is om kernels te launchen vanuit andere kernels of *device code*

Het is ook belangrijk te realiseren dat er een strikte scheiding is tussen code die runt op de CPU of de *host* en code die runt op de GPU of *device*. Omdat er deze strikte scheiding is tussen host en device dient hier bij het ontwerp van de applicatie rekening gehouden te worden, dit heeft namelijk een grote impact op memory access waarop later teruggekomen wordt.

3.1.4 Threads

Wanneer een kernel aangeroepen wordt vanuit de host wordt deze in parallel uitgevoerd in een reeks CUDA threads. Iedere thread is dus verantwoordelijk voor het uitvoeren van de instructies van de kernel op zijn deel van de data.

Om het overzichtelijker te maken welke thread verantwoordelijk is voor de verwerking van welk deel van de data maakt CUDA gebruik van zogenaamde Grids en Blocks.

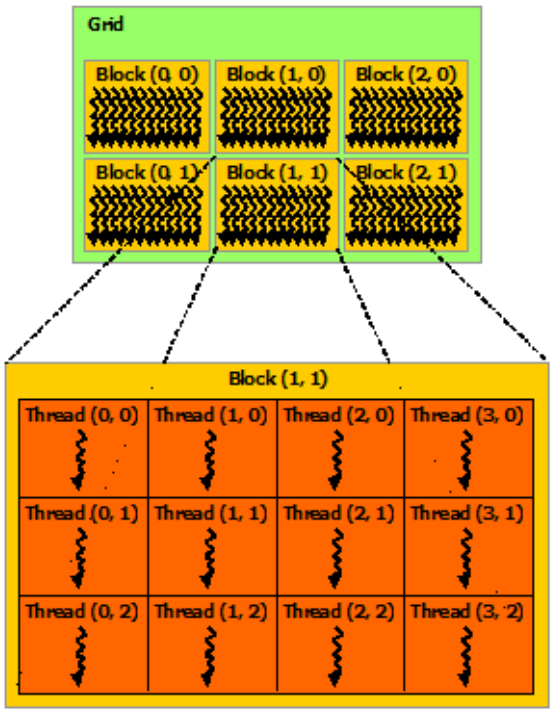
Grids zijn een eerste laag in de structuur die threads overkoepeld. Grids kunnen 1, 2 of 3 dimensionaal zijn. Door bij het aanroepen van de kernel als eerste argument tussen de <<< ... >>> symbolen een *int* of *dim3 object* (dim3 is een CUDA structuur bestaand uit 3 integers, hiermee kan dus de dimensies van grids of blocks mee aangegeven worden) mee te geven worden de dimensies van het grid bepaalt, met andere woorden men geeft aan uit hoeveel blocks het grid bestaat.

De tweede laag in de structuur zijn dan de blocks. Ieder block bestaat uit een reeks threads. Het aantal threads wordt aangegeven door het tweede argument tussen de <<< ... >>> symbolen. Net als bij grids kan dit een *integer* of een *dim3 object* zijn. Dit laat toe om threads nog fijner te structuren binnen het block dat ze bevat. Threads binnen hetzelfde block worden uitgevoerd op dezelfde microprocessor, dit heeft als voordeel dat ze onderling kunnen communiceren via *shared memory*, waarop teruggekomen wordt in 3.1.5. Threads binnen het zelfde block kunnen het uitvoering ook synchroniseren, wat uitvoering van complexe algoritmen mogelijk maakt met behulp van het shared memory.

Het aantal threads per block is op de huidige GPU's beperkt tot 1024, het aantal blocks per grid is dan weer bijzonder groot [2]. Het aantal th-

[2]De block limiet per dimensie van een grid is $2^{31} - 1$ in de recente GPU's

reads dat gestart kunnen worden via een kernel lauch is dus bijgevolg ook bijzonder groot. Om dus te weten welk deel van de data door welke thread verwerkt moeten worden, moet deze thread dus geïdentificeerd kunnen worden. CUDA heeft hiervoor 2 data structuren, *threadIdx* en *blockIdx*. Om de dimensies van het huidige grid en block op te vragen beschikt CUDA over respectievelijk de structuren *gridDim* en *blockDim*. Aan de hand van de waarden van deze structuren kan perfect bepaald worden welke thread van welk block er uitgevoerd wordt en kan zo de juiste data opgevraagd worden.



Figuur 3.3: Structuur van een CUDA Grid [38]

3.1.5 Memory

CUDA kent 4 types memory (5 als men host memory ook als een type beschouwt), registers, local memory, shared memory en global memory. In figuur 3.4 is een overzicht te zien van de types memory dat CUDA kent en hun geassocieerde scope.

Registers Registers zijn per thread memory dat gebruikt wordt om lokale variabelen, buiten arrays, in op te slaan. De scope ervan gaat niet verder dan de thread en de lifetime is die van de thread.

Local memory Local memory wordt gebruikt om lokale arrays binnen een thread in op te slaan. De scope ervan is dus ook beperkt tot de thread en de lifetime is ook die van de thread. Belangrijk om op te ;erken is dat local memory geen fysiek geheugen is. Local memory maakt gebruik van indien mogelijk registers om variabelen op te slaan. Indien dit niet mogelijk is wordt gebruik gemaakt van de L1 cache, vervolgens de L2 cache en tenslotte global memory.

Shared memory Shared memory is geheugen dat gedeeld wordt door een block van threads. De access erop is nog steeds zeer snel (in tegenstelling tot global memory) maar heeft als voordeel dat het toelaat data uit te wisselen tussen threads. Shared memory wordt aangeduid door de __shared__ identifier voor een variabele naam.

Er zijn twee types shared memory, static en dynamic shared memory. Static shared memory is shared memory waarvan de grootte, net zoals een static array, at compile time bekend is. Bij dynamic memory is dit niet het geval. Om de grootte van het dynamic shared memory aan te geven is er een derde argument optioneel beschikbaar tussen de <<< ... >>> symbolen. Dit derde argument van het *integer* type geeft aan uit hoeveel bytes het shared memory bestaat.

Een belangrijke opmerking is dat er bij dynamic shared memory slechts één shared variabele mag zijn, die het adres bevat waar het dynamisch gealloceerd stuk geheugen begint. Als men dus meerdere variabelen op die manier shared wenst te maken moeten deze opgeslagen worden als deel van het gereserveerde geheugen.

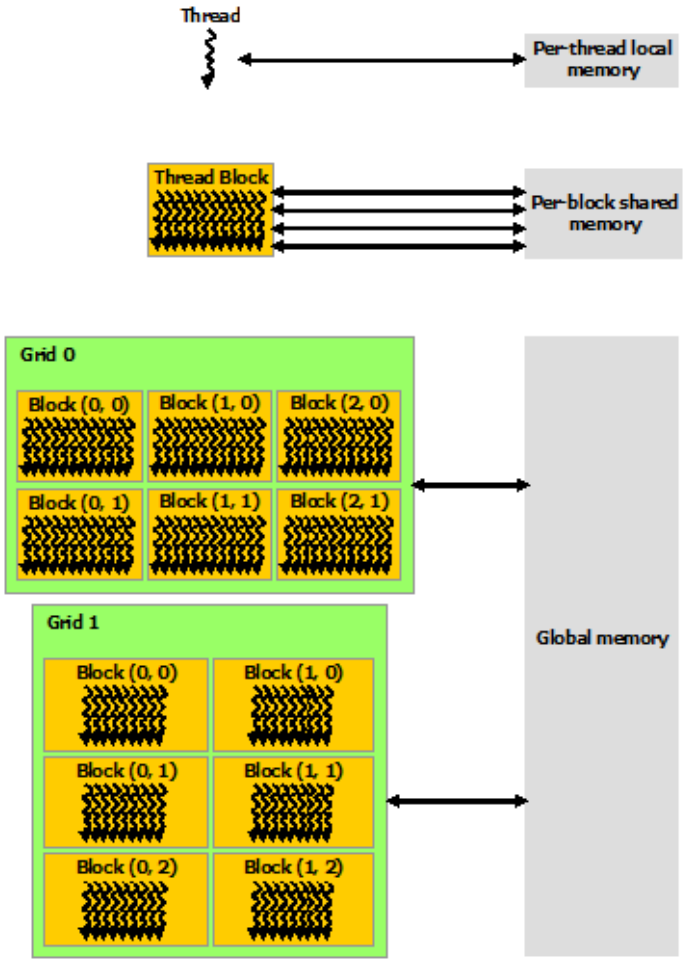
De scope van shared memory is beperkt tot een block van threads, de lifetime ervan is dan ook beperkt tot de lifetime van de kernel.

Global memory Global memory is het traagste van de beschikbare types geheugen binnen CUDA. Het voordeel ervan is wel dat alle threads toegang hebben tot hetzelfde global memory. Het is dus uitermate geschikt voor de opslag van data waar alle threads aan moeten kunnen en om resultaten van berekeningen naar weg te schrijven.

In regel wordt global memory vanuit de host gealloceerd en vrij gegeven. Dit gebeurt via de methodes *cudaMalloc* en *cudaFree*. cudaMalloc alloceert geheugen van een bepaalde grootte op het device en slaat het adres ervan op in een meegegeven pointer. De cudaFree methode geeft gealloceerd geheugen op het device vrij. Een belangrijke opmerking hierbij is dat het adres verkregen via cudaMalloc een adres op het geheugen van de GPU is en men hier vanop de host geen rechtstreekse toegang tot heeft. Als men geheugen van

en naar de GPU wilt overbrengen is er de *cudaMemcpy* methode beschikbaar die host to device en vice versa geheugen operaties ondersteunt.

De scope van global memory is dan ook alle threads en de lifetime wordt bepaald door allocatie en deallocatie (in extremis de application lifetime).



Figuur 3.4: Memory scope in CUDA [38]

Page-locked memory Een laatste type geheugen binnen CUDA is page-locked memory. Page-locked memory is geen geheugen op de GPU, in tegenstelling tot de eerder vernoemde types geheugen, maar is geheugen op de host dat page-locked gealloceerd is. Dit laat toe om via Direct Memory Access (DMA) hier toegang tot te hebben vanop de GPU.

3.1.6 Concurrency en synchronization

CUDA biedt ook een heel aantal mogelijkheden aan om concurrency tussen kernels en threads te implementeren en deze ook onderling te synchroniseren. In deze sectie worden eerste de mogelijkheden bekeken op het vlak van thread synchronization en daarna de mogelijkheden voor concurrency en synchronization tussen kernels

3.1.6.1 Threads

Zoals al aangehaald in 3.1.5 beschikken blocks over gedeeld geheugen, shared memory, dat threads toelaat informatie uit te wisselen. Omdat niet alle threads binnen een block altijd op hetzelfde punt van de code tijdens de uitvoering van de kernel zitten is er synchronisatie tussen de threads nodig om memory conflicts te voorkomen bij het ophalen van geheugen uit het shared memory.

Hiervoor heeft men in CUDA *__syncthreads()* ter beschikking. Deze functie stopt de uitvoering van de thread tot alle threads binnen een block op dit punt aangekomen zijn. Hierna wordt de uitvoering hervat tot het einde van de kernel of tot er een nieuwe __syncthreads() tegengekomen wordt.

3.1.6.2 Kernels

In deze sectie worden concurrentie en synchronisatie mogelijkheden tussen kernels en breder het algemene verloop van de applicatie besproken. CUDA stelt hiervoor een aantal tools ter beschikking, waarvan de belangrijkste hier besproken worden.

Standaard zijn de meeste CUDA calls asynchroon. Dit wil zeggen dat na het uitvoeren van deze CUDA call de host code gewoon verder loopt tot de uitvoering ervan geblokkeerd wordt. Het blokkeren van host code gebeurt meestal wanneer er een volgende CUDA call wordt uitgevoerd, bijvoorbeeld na een kernel launch zal host code uitgevoerd worden tot aan de cudaMemcpy die het resultaat ophaalt. Het is pas nadat deze copy voltooid is dat de uitvoering verder gezet wordt. De volgende calls zijn standaard asynchroon:

- Kernel launches
- Memory copies binnen het device
- Memory copies van host naar device kleiner dan 64 KB
- Memory copies met de affix *Async*

- Memory set function calls

Om de host te dwingen het uitvoeren van code te stoppen is er de functie *cudaDeviceSynchronize()* beschikbaar. Het uitvoeren wordt na deze functie pas hervat wanneer de GPU klaar is met al het huidige werk.

Streams Om concurrentie nog verder uit te breiden beschikt CUDA over een concept genaamd *Streams*. Wanneer een CUDA operatie uitgevoerd wordt, meer specifiek een kernel launch of een cudaMemcpy, is deze uitvoering geassocieerd met een zogenaamde stream. Tenzij specifiek vermeld als extra argument in de functie is dit altijd de default stream. Alle instructies geassocieerd met eenzelfde stream worden gequeued en na elkaar uitgevoerd, dit verklaart waarom standaard gewacht dient te worden tussen verschillende CUDA operaties, deze operaties zijn namelijk geassocieerd met dezelfde stream, namelijk de default stream. Operaties geassocieerd met een verschillende stream worden dus concurrent uitgevoerd en moeten dus niet wachten op voltooiing van elkaar.

Om een stream aan te maken dient de functie *cudaStreamCreate* gebruikt te worden, deze maakt een pointer aan naar een *cudaStream_t* die dan gebruikt kan worden om te associëren met andere CUDA operaties. Streams kunnen verwijderd worden via de functie *cudaStreamDestroy*, deze functie blokkeert dan ook de host code tot alle CUDA operaties voltooid zijn geassocieerd met deze stream.

Om host code te synchroniseren met een specifieke stream beschikt CUDA over de functie *cudaStreamSynchronize*, deze werkt identiek aan cudaDeviceSynchronize met als verschil dat de host code geblokkeerd wordt tot alle operaties geassocieerd met de gekozen stream voltooid zijn.

Events Om de progressie van code te monitoren en timing informatie te verzamelen beschikt CUDA over zogenaamde *Events*. Events kunnen aangeroepen worden op ieder punt in de host code en kunnen hierna gebruikt worden om te controleren of ze compleet zijn. Events zijn compleet wanneer alle CUDA operaties (of alle operaties geassocieerd met een optionele stream) voltooid zijn.

Events worden aangemaakt via de *cudaEventCreate* functie. Deze maakt een pointer naar een *cudaEvent_t* aan dat dan gebruikt kan worden om de progressie van het event te tracken. Events worden verwijderd via de functie *cudaEventDestroy*.

Om host code te synchroniseren beschikt CUDA over de functie *cudaEventSynchronize*, deze blokkeert host code tot het geassocieerde event voltooid is. Om device code te synchroniseren beschikt CUDA over de functie

cudaStreamWaitEvent, deze functie blokkeert alle operaties geassocieerd met een stream (of indien er geen stream gekozen wordt alle streams) tot een gekozen event voltooid is.

Hoofdstuk 4

Implementatie

In dit hoofdstuk wordt dieper ingegaan op de uiteindelijke implementatie van de video encoder. In een eerste sectie wordt algemene informatie gegeven over de eigenschappen van de video encoder, zoals input formaat en settings. In een tweede sectie wordt dan uitgelegd welke componenten van een typische video encoder, zoals beschreven in 2.1, geïmplementeerd zijn en beargumenteerd waarom componenten niet gerealiseerd werden. In een laatste sectie wordt dan ingegaan op de delen van de video encoder die geïmplementeerd zijn in CUDA. Ook wordt beargumenteerd waarom componenten niet gerealiseerd zijn op de GPU.

Opmerking Bij de implementatie van de algoritmen lag de prioriteit op de performance en parallellisatie van de algoritmen. Het implementeren van algoritmen met tot doel het halen van hoge compressieratio's viel dan ook buiten de scope van de thesis en werd ook geen aandacht aan besteed.

4.1 Algemene eigenschappen

De video encoder is gebaseerd op een project gerealiseerd voor het vak *Technologie van multimediasystemen en -software* (TMS) in 2011 aan de Universiteit Hasselt. Het project had tot doel om een eenvoudige video encoder en decoder te realiseren ten einde hands-on ervaring op te doen met de concepten achterliggend aan video compressie. Het encoder component werd grotendeels gerealiseerd door Gert Dubois, het decoder component door Aäron Thijs.

De keuze om de uiteindelijke implementatie te baseren op deze encoder werd gemaakt om de volgende redenen. Een eerste reden was dat binnen de beschikbare tijd en rekening houdend met het feit dat de encoder alleen

geïmplementeerd diende te worden het onmogelijk zou zijn om een encoder te realiseren die voldeed aan meer gangbare standaarden zoals H.264/AVC. De keuze werd gemaakt om een simpelere encoder te realiseren en hierbij viel de logische keuze om deze te baseren op een al gerealiseerde implementatie zodat gebruik gemaakt kon worden van een al bestaande framework.

Input formaat Als input formaat worden video bestanden in het *YUV4MPEG2* (.y4m) formaat geaccepteerd. De beperking hierop is wel dat enkel bestanden van het formaat YCbCr 4:2:0 ondersteund worden door de encoder

Variabele settings van de encoder De encoder laat het aanpassen van een aantal instellingen tot via een configuratie bestand dat meegegeven wordt als argument bij het opstarten van de encoder. De volgende settings kunnen aangepast worden:

- *rawfile*: De file waaruit de raw data gehaald dient te worden
- *encfile*: De file waarin de encoded data opgeslagen dient te worden
- *quantfile*: De file die de kwantisatie matrix bevat
- *gop*: Lengte van de Group Of Pictures
- *merange*: Bepaalt de afstand waarbinnen gezocht moeten worden naar een match bij ME

4.2 CPU based implementatie

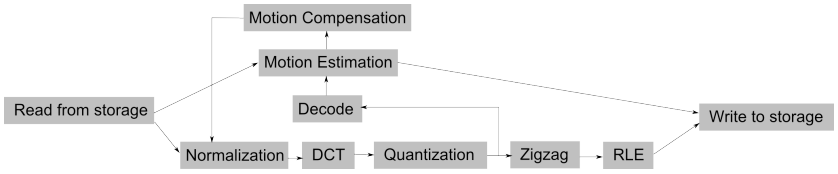
Een eerste stap in de realisatie van de encoder was de implementatie van een CPU gebaseerde encoder. De structuur van deze encoder is zoals al eerder aangehaald gebaseerd op een een eerder gerealiseerd project en een schematische weergave van de werkwijze van de encoder is terug te vinden in figuur 4.1.

Op het eerste zicht kunnen 2 zaken afgeleid worden uit deze flowchart. Ten eerste gebruikt de encoder 2 types frames, I-frames en P-frames, B-frames worden niet gebruikt teneinde de encoder te vereenvoudigen. Een tweede vaststelling is dat er geen intra predictie wordt gebruikt.

Om de encoder te implementeren werd gekozen voor de taal C/C++. CUDA is namelijk het meest geschikt voor gebruikt met C (alhoewel er ook een Python compatibele versie beschikbaar is).

In de volgende subsecties wordt de implementatie van de verschillende componenten van de encoder besproken, hierbij wordt de implementatie van

de verschillende hulpklassen achterwege gelaten omdat deze weinig relevant zijn voor de thesis.



Figuur 4.1: Flowchart die de uitvoering van de encoder weergeeft

4.2.1 Motion Estimation

Het geïmplementeerde motion estimation algoritme is het Full Search algoritme. Hiervoor werd gekozen omdat het algoritme vrij eenvoudig te implementeren is en door zijn structuur erg veel baat heeft bij parallellisatie. De pseudo-code voor het geïmplementeerde algoritme is dan ook quasi identiek aan deze uit 2.1.1.1.

Het algoritme gaat per macroblok een match zoeken binnen de gekozen ME range. De informatie van deze match wordt opgeslagen in een datastructuur genaamd *MADInfoContainer* waarin data zit zoals de motion vector per macroblok, de residu matrix, de MAD, ... De container met de kleinste MAD is het resultaat en bevat alle nodige informatie voor de rest van het algoritme.

Een belangrijke opmerking is dat ME altijd gebeurt op macroblokken van 4x4 pixels in deze encoder. Implementatie van dynamische macroblok grootte is bijzonder complex en valt buiten de scope van deze thesis.

De MAD van een macroblok wordt als volgt berekend:

```
int madTot = 0;

foreach pixel in macroblock
{
        int val = macroblock[i];
        if(val < 0)
                val = -val;

        madTot += val;

}

return (double)madTot/256.0;
```

4.2.2 Motion Compensation

Omdat de thesis zich focust op performantie van de de verschillende componenten van een video encoder en niet op het behalen van een zo hoog mogelijk compressie ratio werd motion compensation op een eenvoudige manier gerealiseerd. Een P-frame wordt in de gecomprimeerde data opgebouwd door eerst de motion vectoren weg te schrijven gevolgd door het gecomprimeerde (identiek aan I-frame compressie) residu weg te schrijven. Omdat er geen gebruik wordt gemaakt van Huffman encoding resulteert dit gemiddeld in een grotere file dan wanneer enkel I-frame compressie zou gebruikt worden.

4.2.3 Intra predictie

Intra predictie is niet gerealiseerd binnen de kader van het thesis en is een mogelijke piste voor verder onderzoek.

4.2.4 DCT

De gebruikte implementatie van het DCT algoritme is eenvoudig in implementatie. Namelijk 2 matrix vermenigvuldigingen in de volgende vorm:

$$A = D * M * D^t$$

Waarin D de DCT matrix is met grootte NxN. In het geval van de geïmplementeerde encoder is N = 4. Omdat de algoritmen uit het image model niet erg complex zijn kunnen kleinere macroblokken gebruikt worden om de kwaliteit van de geëncodeerde video op die manier te verhogen.

Een belangrijke opmerking is dat voor de DCT berekeningen uitgevoerd kunnen worden de data van de macroblokken genormaliseerd dienen te worden. Dit is een eenvoudige operatie die op element basis de pixels van een macroblok centreert rond 0. Voor een typische YCrCb 4:2:0 video worden de pixel waarden gemapt van [0, 255] naar [-128, 127].

4.2.5 Kwantisatie

Het geïmplementeerde algoritme is identiek aan dat beschreven in 2.1.2.2. Er wordt gebruik gemaakt van een kwantisatiematrix, het algoritme deelt dus de elementen van een macroblok door de overeenkomstige elementen van de kwantisatiematrix.

4.2.6 RLE

Het geïmplementeerde RLE algoritme is op te delen in twee componenten. Het eerste component voert de zigzag operatie uit voor het effectieve RLE algoritme en het tweede component is het eigenlijke RLE algoritme.

Het zigzag algoritme wordt hardcoded gerealiseerd. Er wordt altijd gewerkt met macroblokken met grootte 4x4 pixels, de meeste efficiënte aanpak is dan ook de verplaatsingen van de elementen te hardcoden. Het alternatief hiervoor zou een bijzonder complex algoritme zijn wat nutteloos veel rekentijd in beslag neemt.

Het RLE algoritme kan in pseudo-code als volgt uitgedrukt worden:

```
int seriesOf0=0;
foreach macroblock in frame
 foreach pixel in macroblock
  if (currenPixel != 0) {
   if(seriesOf0)
   {
    writeRunAndLength();
    seriesOf0 = 0;
   }
  } else {
   ++seriesOf0;
  }
 }
 if(seriesOf0)
  writeBlockEnd();
```

Kort samengevat bepaald het algoritme wanneer een reeks nullen ophoudt en schrijft dan de run en length weg naar het geheugen.

4.2.7 Entropy Encoder

Net als intra predictie werd er voor gekozen om geen entropy encoder te voorzien in de geïmplementeerde video encoder. De redenering hierachter is dat enerzijds zoals aangehaald in 2.1.3 de meeste entropy encoding algoritmen (zoals Huffman) moeilijk te parallelliseren zijn. Bovendien is de parallellisatie van entropy encoding algoritmen meer een probleem in het domein van de algemene compressie, omdat de thesis focust op parallellisatie van video encoding algoritmen werd dan ook geen entropy coding geïmplementeerd.

4.2.8 Parallellisatie

Parallellisatie van de CPU code wordt gerealiseerd door middel van OpenMP. Dit laat toe om op vrij eenvoudige wijze performante CPU code te schrijven

om de gerealiseerde code te parallelliseren.

Parallellisatie wordt gerealiseerd op macroblok niveau. De redenering hierachter is als volgt, voor bijna alle componenten van een video encoder geldt dat er geen waiting conditions zijn tussen macroblokken onderling (buiten bij intra-predictie). Parallellisatie op het niveau van frames is niet nuttig omwille van de afhankelijk tussen frames voor de compressie van P-frames. Parallellisatie op een dieper niveau (bijvoorbeeld pixel niveau) is vaak moeilijk omwille van waiting conditions binnen de berekeningen van een macroblok, zoals bij het DCT algoritme. Parallellisatie op macroblok niveau is dan ook de methode waarop geen waiting conditions binnen de componenten van de video encoder worden geïntroduceerd en toch nog een voldoende hoge graad van parallellisatie kan bieden.

4.3 Implementatie in CUDA

In deze sectie wordt het belangrijkste deel van het implementatie gedeelte van de thesis behandeld, de implementatie van componenten van het video encoding algoritme in CUDA. De focus ligt op 5 componenten. Het eerste component is motion estimation, het belangrijkste component omdat het het component is dat het meeste rekentijd in beslag neemt. Optimalisaties en snelheidsverbeteringen in dit component hebben onmiddellijk een grote invloed op de totale uitvoeringssnelheid. De 4 andere componenten zijn normalisatie, DCT, kwantisatie en zigzag ordering. Deze algoritmen behoren allen tot de I-frame encoding pipeline en worden dan ook samen behandeld.

4.3.1 Motion Estimation

De implementatie van het CUDA algoritme voor motion estimation is gebaseerd op de Full Search implementatie beschreven in [22]. De grote beperking van de implementatie beschreven in de paper is dat er per pixel van het volledige search window (bepaald door de ME range) een thread gelaunched dient te worden. Op CUDA devices is het zo dat het aantal threads dat per CUDA block gelaunched kan worden beperkt is. Om er nog steeds voor te zorgen dat alle ME ranges uitvoerbaar zijn werd besloten het algoritme licht aan te passen.

De kernel bestaat uit twee grote stappen. In een eerste stap wordt er per pixel van het search window de MAD berekent van het overeenkomstig blok en opgeslagen. In een tweede stap wordt aan de hand van de verkregen tussenresultaten het macroblok met de kleinste MAD bepaald en opgeslagen in global memory.

Het verschil met de origineel beschreven implementatie is dat de wijze van threading anders gebeurt. De aangepaste implementatie gebruikt een instelbaar aantal threads om de MAD van ieder pixel in de search window te berekenen. Iedere thread is dus verantwoordelijk voor het verwerken van 1 of meerdere pixels in de search window, afhankelijk van het aantal pixels in de search window en de gekozen grootte van de threadpool. Iedere thread slaat zijn beste resultaat op in shared memory. In de laatste stap wordt dan aan de hand van de overgehouden resultaten het resultaat met de laagste MAD gezocht en weggeschreven naar het global memory.

Motion Compensation Een opmerking dient gemaakt te worden in verband met motion compensation. De berekening van de motion vectoren gebeurt in de CUDA kernel en is dus geïmplementeerd op de GPU. De berekening van het residu wordt uitgevoerd op de host. Indien deze berekend zou worden op de GPU zouden er veel extra cudaMemcpy's nodig zijn om alle residu data naar de host terug te kopiëren. Om latency te verlagen is beslist om dit deel van de motion compensation op de host uit te voeren.

4.3.2 Normalisatie

De normalisatie kernel is een zeer simpele kernel. Ieder CUDA block is verantwoordelijk voor het berekenen van één macroblok van een frame. Per pixel wordt een thread aangemaakt binnen het CUDA block. Iedere thread moet dan 128 af trekken van een zijn geassocieerde pixel en terug opslaan in het geheugen.

4.3.3 DCT

De DCT kernel is gebaseerd op de officiële implementatie voor DCT voorzien door NVIDIA. Het verschil zit erin dat de gerealiseerde kernel geschikt is voor 4x4 macroblokken in plaats van 8x8 macroblokken zoals in de officiële implementatie.

De kernel is als volgt ingedeeld, ieder CUDA block berekent de DCT transformatie van een macroblok van de frame. Per pixel van het macroblok wordt dan een thread gestart binnen het CUDA block.

De kernel gebruikt shared memory om tussentijdse resultaten van de matrix vermenigvuldigingen op te slaan. Net zoals bij de ME implementatie introduceert dit waiting conditions binnen de kernel waarvan de impact op de performantie in 5.4.4.2 besproken wordt.

4.3.4 Kwantisatie

De implementatie van de kwantisatie kernel is bijna identiek aan die van de normalisatie kernel, alleen moet hier iedere pixel gedeeld worden door een waarde. Het enige verschil is dat de te delen waarde uitgelezen dient te worden uit het geheugen en dat de eindwaarde binnen het interval [-128,127] dient te blijven.

4.3.5 Zigzag

De zigzag kernel is verantwoordelijk voor het herschikken van pixels van ieder macroblok. De indeling in CUDA is dan ook identiek aan die van de andere kernels. Iedere pixel wordt uitgelezen en daarna op een andere index opgeslagen in een nieuwe datastructuur die dan uitgelezen wordt door het RLE algoritme.

Hoofdstuk 5

Resultaten

In dit hoofdstuk worden de resultaten van de uitvoering van de encoder gebundeld. Om deze resultaten te bekomen is van iedere stap van het encoding proces timing informatie verzameld door gebruik te maken van de event library van NSight. In onderstaande tabellen is van iedere stap de uitvoeringstijd uitgedrukt in microseconden. In de tabellen zijn de volgende stappen terug te vinden.

- IFrame encoding: Totale encoding tijd voor een I-frame
- PFrame encoding: Totale encoding tijd voor een P-frame
- DCT, Normalization, Quantization, Zigzag: De I-frame gerelateerde encoding algoritmen
- Perform ME/MC: ME/MC algoritme
- Encode MC data: I-frame encoding van MC residue frame
- RLE encoding: RLE algoritme
- Read from/Write to storage: IO operaties naar de HDD
- Swap context: Memcpy van Host memory naar Device memory of vice versa
- Copying data: Memcpy van Host to Host of Device to Device

Aan de hand van de verkregen data wordt er een grondige analyse gemaakt waarop de uiteindelijke conclusies gebaseerd kunnen worden Voor de I-frame encoding pipeline ligt de focus enkel op de uitvoeringstijden van toepassing op de 4 relevante geïmplementeerde algoritmen, normalisatie, DCT,

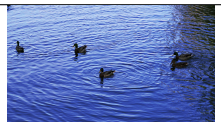
Motherboard	Asus P8Z68-V Pro PCIe 2.0 x16 Intel Socket 1155 with Intel Z68 Chipset
CPU	Intel Core i7 2600K @ 3.4GHz
GPU	Asus GTX 660 DirectCU II OC 2GB DDR5 Memory @ 1502 MHz GPU Base Clock: 1020 MHz GPU Boost Clock: 1085 MHz Compute Capability 3.0 CUDA cores: 960
Memory	2x4GB Kingston Dual Channel memory @ 1333 MHz
Storage	Western Digital Green Caviar 2TB SATA @ 7200 RPM

Tabel 5.1: Hardware specificaties

kwantisatie en zigzag ordering. Er wordt nagekeken of de schaling van de CPU algoritmen voldoen aan het verwachtingspatroon en wat de performantie winsten zijn van de CUDA oplossingen in vergelijking met de CPU implementatie. Bovendien wordt er analyse gemaakt van de verschillende CUDA kernels om in te schatten welk niveau van performantie deze kernels realiseren. Daarna wordt een analyse gemaakt van de performantie van de componenten relevant voor de P-frame encoding pipeline. Eerst wordt een vergelijking gemaakt van de uitvoeringstijden op de CPU en de uitvoeringstijden in CUDA. Daarna wordt een analyse gemaakt van schaling van de ME kernel in CUDA zowel op het vlak van resolutie als motion estimation range. Ten slotte wordt ook gekeken hoe de verschillende componenten onderling schalen bij stijgende resolutie in de CUDA pipeline.

5.1 Testing Hardware

De hardware specificaties van de setup waarop alle tests zijn uitgevoerd staan beschreven in tabel 5.1.

	Flying Ducks	Bumblebee	Bumblebee 4k	Bumblebee 8k
				
Resolution	1280x720	1920x1080	3840x2160	7680x4320
Frames	500	500	500	500
Format	YCrCb 4:2:0	YCrCb 4:2:0	YCrCb 4:2:0	YCrCb 4:2:0

Tabel 5.2: Test sequenties

5.2 Test sequenties

In tabel 5.2 is een overzicht te vinden van de eigenschappen van de gebruikte test bestanden

5.3 Algemeen overzicht

Om het totaal overzicht te maken werd besloten om de gegevens te gebruiken van de uitvoeringen met CUDA enabled met een ME range van 32. De gegevens zelf zijn terug te vinden in secties 5.4.3 en 5.5.2.

In grafiek 5.1 is een volledig overzicht te zien van de belangrijkste componenten van de volledige encoding pipeline. In grafiek 5.2 is dezelfde data te zien maar dan zonder de ME stap. Dezelfde data is ook nog eens te zien in 5.3 maar dan zonder de RLE stap.

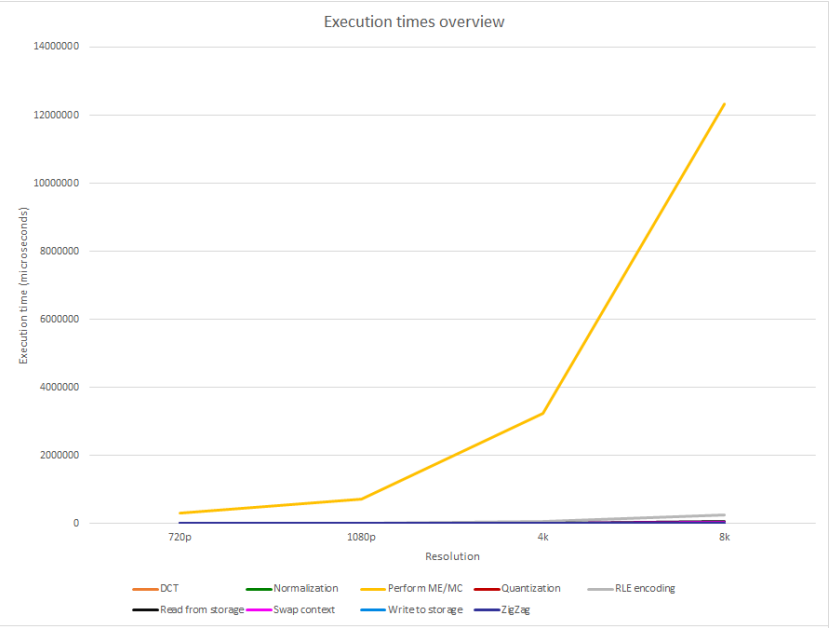
Wat onmiddellijk opvalt is dat het grootste gedeelte van de uitvoeringstijd gespendeerd wordt aan het encoden van P-frames en meer bepaald aan het ME en MC. Dit is volledig naar verwachting, daar dit in courante video encoders ook het geval is. Omdat er gebruik gemaakt wordt van Full Search is dit nog meer het geval dan bij andere video encoders, zelfs bij het gebruik van CUDA blijft dit de grootste performantie bottleneck in de encoding pipeline.

Uit de componenten gebruikt binnen de I-frame encoding blijkt al snel dat RLE encoding een zeer substantieel aandeel uitmaakt van de totale encoding tijd van I-frames. Een belangrijke stap in verder onderzoek zou dan ook de optimalisatie van dit component van de encoder zijn. Hier zijn twee onderzoekspistes voor beschikbaar.

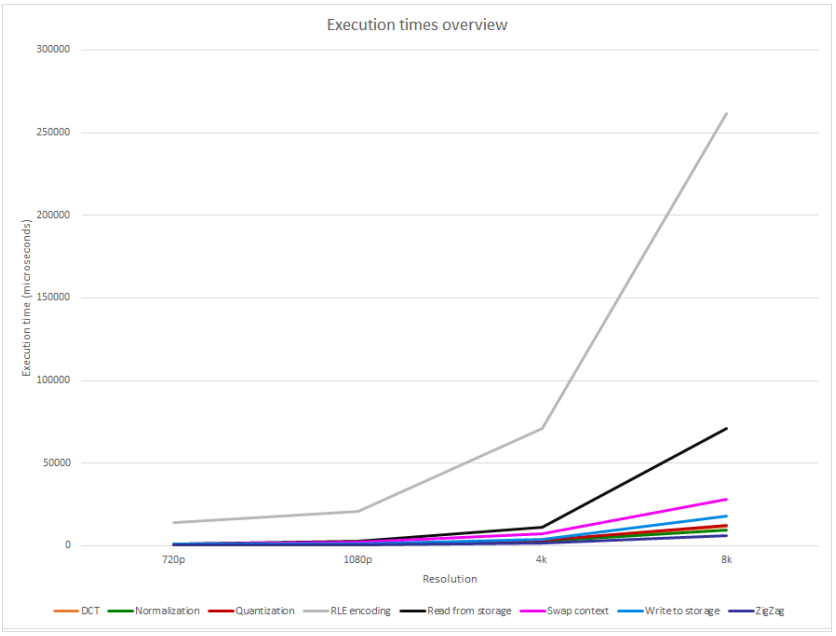
Een eerste piste zou het onderzoeken van de impact van het parallelliseren van het RLE algoritme zijn. Traditionele parallellisatie van het RLE algoritme heeft namelijk ook impact op de bereikbare compressie ratio's en aandacht hiervoor is dan ook van belang tijdens het onderzoek hiernaar.

Een tweede piste is mogelijks de vrijgekomen CPU tijd te gebruiken om

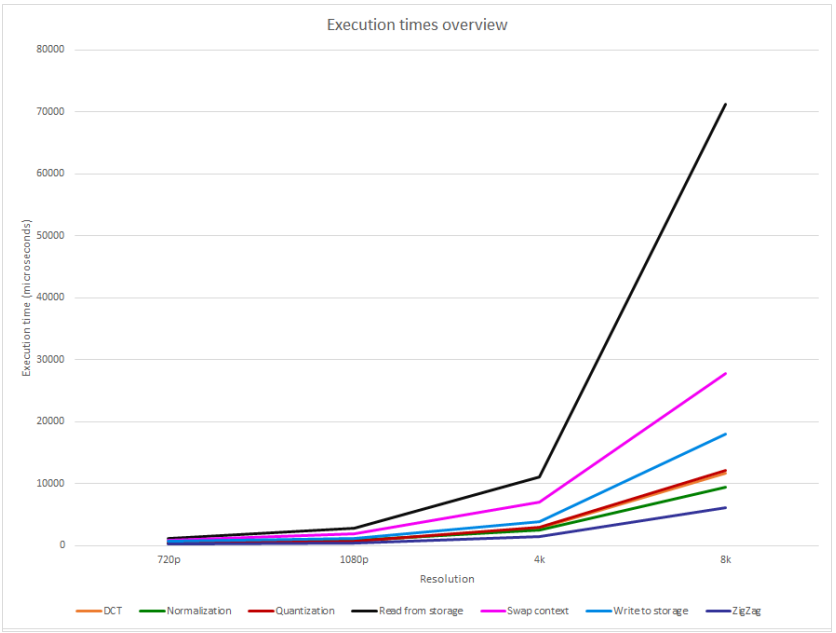
het RLE algoritme uit te voeren tijdens het encoden van een volgende frame op de GPU. RLE is namelijk niet nodig om intern frames te decoderen voor verder gebruik in de pipeline. Het is dus perfect mogelijk om het uitvoeren van het RLE algoritme te beginnen terwijl een volgende frame al verwerkt wordt op de GPU.



Figuur 5.1: Totaaloverzicht



Figuur 5.2: Totaaloverzicht met focus op I-frame encoding + RLE



Figuur 5.3: Totaaloverzicht met focus op I-frame kernels

5.4 I-Frame encoding pipeline

Deze sectie focust op de algoritmen noodzakelijk voor de I-frame encoding pipeline. In een eerste stap wordt onderzocht in welke mate de eerste implementatie van de CUDA kernels performant is en in welke mate de performantie verbeterd kan worden. Daarna wordt de meest optimale implementatie van de CUDA kernels vergeleken met hun respectievelijke CPU code.

5.4.1 Performantie analyse

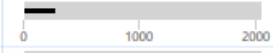
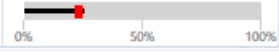
Een van de belangrijkste metrics voor performantie van CUDA kernels is occupancy. Dit geeft aan in welke mate CUDA latency in de uitvoering van de code, zoals wachten op memory transfer, wachten bij synchronisatie punten in de code, ..., kan verbergen en zo optimaal de beschikbare resources van de GPU kan gebruiken.

Als men figuren 5.4, 5.5, 5.6 en 5.7 bekijkt is al snel af te leiden dat de occupancy van de kernels zeer laag is. In de tweede kolom valt te lezen wat de effectief bereikte occupancy van de kernel is. In het geval van al de geïmplementeerde kernels is dit rond de 23%. In de kolom ernaast is de theoretisch maximaal haalbare occupancy te zien en deze is voor al de kernels 25%. Er zit dus een fundamentele redeneringsfout in de implementatie van de kernels die ervoor zorgt dat de resources van de GPU niet voldoende benut worden. De verklaring hiervoor valt af te leiden uit de fysieke maxima die bereikbaar zijn voor iedere mogelijke kernel. Deze waarden vallen af te lezen uit de voorlaatste kolom van de figuren. Deze maxima zijn afhankelijk van de architectuur van de GPU, in het geval van de GTX 660 betekent dit concreet dat er slechts 16 CUDA blocks tegelijk active kunnen zijn. In de eerste implementatie bevat ieder CUDA block slechts 16 threads (namelijk 1 thread per pixel van een macroblok). Omdat CUDA warps launchen in groepen van 32 threads kan er per CUDA block in dit geval slechts 1 warp gelaunched worden. Occupancy wordt rechtstreeks bepaald door de hoeveelheid actieve warps ten opzichte van het fysieke maximum (64 voor de GTX 660), wat de theoretisch maximale occupancy van 25% verklaart.

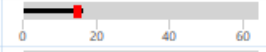
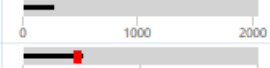
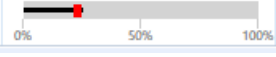
Om de performantie van kernels te verbeteren is het dus cruciaal om het aantal warps dat gelaunched kan worden per macroblock voldoende te verhogen. In de praktijk komt het erop neer dat het aantal threads verhoogt moet worden binnen een CUDA block en binnen de gerealiseerde implementatie kan dit op 2 manieren.

De eerste methode is het vergroten van de gebruikte macroblokken. Deze methode heeft als grote nadeel dat daardoor de kwaliteit van de encoder verlaagt, wat een ongewild effect is. De tweede methode is het aantal macro-

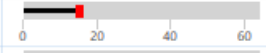
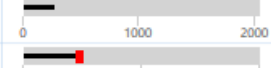
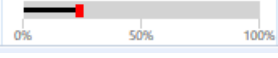
blokken dat een CUDA block verwerkt vergroten. Dit heeft geen invloed op de output van het encoding algoritme en bereikt hetzelfde voor de occupancy als de macroblokken vergroten. De performantie implicaties van deze laatste methode worden nu verder onderzocht.

Active Blocks		16	16	
Active Warps	14.86	16	64	
Active Threads		256	2048	
Occupancy	23.21 %	25.00 %	100.00 %	

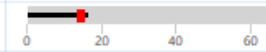
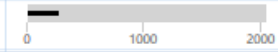
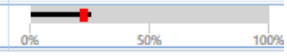
Figuur 5.4: Normalisatie kernel occupancy

Active Blocks		16	16	
Active Warps	15.08	16	64	
Active Threads		256	2048	
Occupancy	23.57 %	25.00 %	100.00 %	

Figuur 5.5: DCT kernel occupancy

Active Blocks		16	16	
Active Warps	15.22	16	64	
Active Threads		256	2048	
Occupancy	23.79 %	25.00 %	100.00 %	

Figuur 5.6: Kwantisatie kernel occupancy

Active Blocks		16	16	
Active Warps	14.39	16	64	
Active Threads		256	2048	
Occupancy	22.48 %	25.00 %	100.00 %	

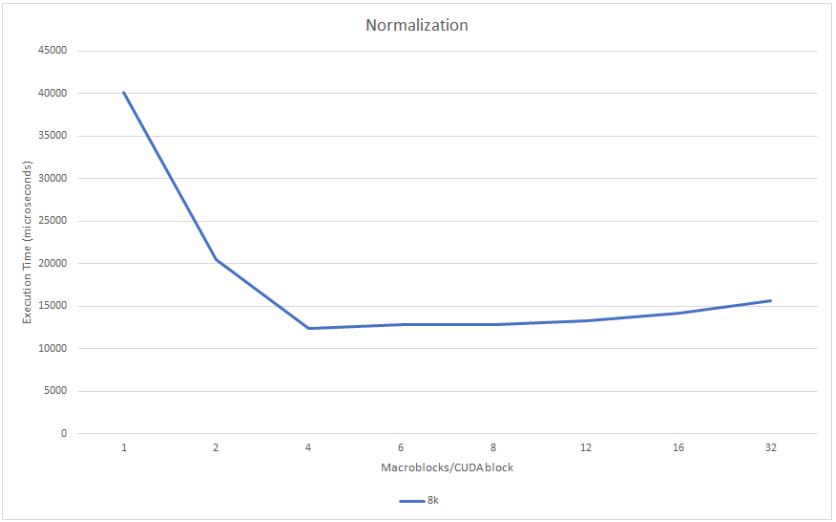
Figuur 5.7: Zigzag kernel occupancy

5.4.2 Kernel performantie tuning

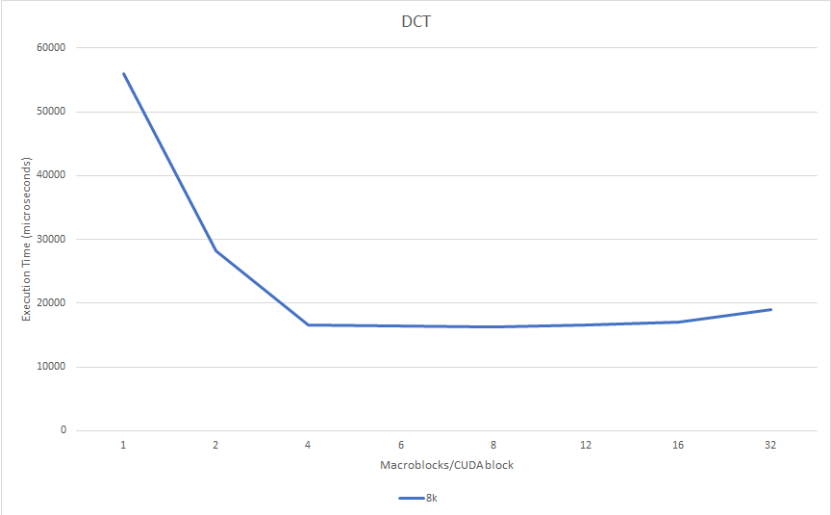
Om vast te stellen hoeveel macroblokken per CUDA block nodig zijn om maximale performantie te bereiken werden de uitvoeringstijden van de vier relevante kernels met toenemende ratio van macroblokken per CUDA block onderzocht. In figuren 5.8, 5.9, 5.10 en 5.11 zijn de uitvoeringstijden te zien van de vier kernels met een toenemend aantal macroblokken per CUDA block en dit voor de 4 geteste resoluties.

Uit de grafieken valt af te leiden dat door het verhogen van het aantal threads per CUDA block, en bijgevolg ook de occupancy de performantie aanzienlijk stijgt. Om goede performantie te bereiken zijn er minimaal 4 macroblokken nodig per CUDA block, meer macroblokken heeft niet in alle kernels veel impact op de performantie. De normalisatie en DCT kernels lijken vanaf 4 macroblokken vrij constante performantie aan te houden. De kwantisatie kernel lijkt wel iets beter te schalen bij het verder opvoeren van het aantal macroblokken per CUDA block. De zigzag kernel daarentegen boet aanzienlijk aan performantie in bij de stap van 4 naar 8 macroblokken per CUDA block en verder.

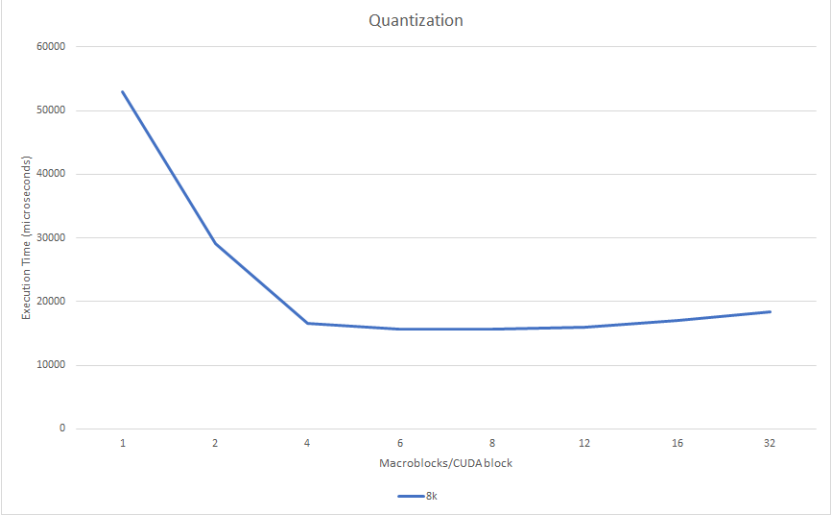
Aan de hand van de bekomen resultaten werd beslist om alle kernels uit te voeren met 4 macroblokken per CUDA block. De performantie van alle kernels is op deze waarde voldoende hoog en is consistent goed over alle kernels heen.



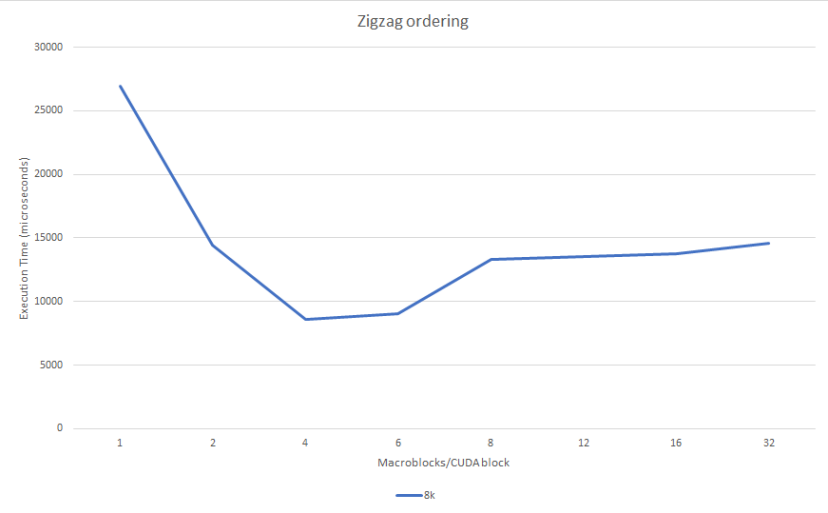
Figuur 5.8: Normalization kernel performance



Figuur 5.9: DCT kernel performance



Figuur 5.10: Quantization kernel performance



Figuur 5.11: Zigzag kernel performance

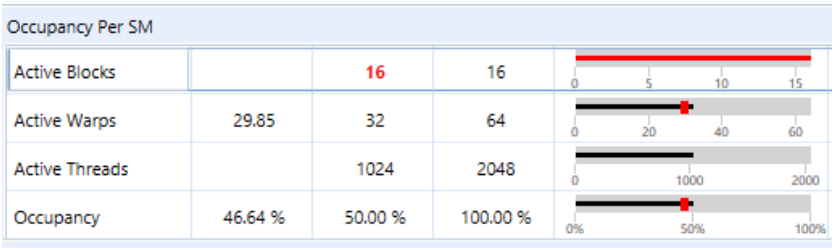
In figuren 5.12, 5.13, 5.14 en 5.15 vindt men een overzicht van de occupancy na de eerste performantie tuning. Een vastelling die men kan doen is dat voor optimale performantie geen 100% occupancy vereist is. De vraag die men dan kan stellen is waarom de performantie, zeker bij de zigzag kernel, van de kernels bij hogere occupancy lager ligt.

De verklaring hiervoor vindt men wanneer men de memory statistics bekijkt van de kernels. In figuur 5.16 vindt men een overzicht van de totale memory transactions in de zigzag kernel uitgevoerd op 4 macroblokken per CUDA block, het optimale scenario. In figuur 5.17 vindt men hetzelfde overzicht voor de zigzag kernel maar dan voor 8 macroblokken per CUDA block. Van belang in deze figuur is de L1 hit rate die men kan aflezen als een percentage in het *L1 Cache* segment. Voor de 4 macroblokken implementatie is deze 91% voor de 8 macroblokken implementatie is dit 41%. Hoe hoger deze hit rate hoe minder een fenomeen zich voordoet genaamd *Register spilling*. Omdat er per CUDA block slechts een beperkt aantal registers beschikbaar is voor operaties is het mogelijk dat CUDA bepaalde operaties dient uit te voeren in het trager cache geheugen. Indien CUDA geen registers beschikbaar heeft voor een operaties wordt deze operatie gespilled naar de L1 cache. Als er geen ruimte beschikbaar is op de L1 cache wordt deze gespilled naar de L2 cache. En als er tenslotte geen ruimte beschikbaar is op de L2 cache wordt deze gespilled naar het global memory. De L1 cache is een onderdeel van iedere Streaming Multiprocessor (SM) net als shared memory. Zolang operaties gespilled worden in de L1 cache is er geen extra memory bandwith nodig omdat alle operaties binnen de SM blijven. De ratio van operaties die

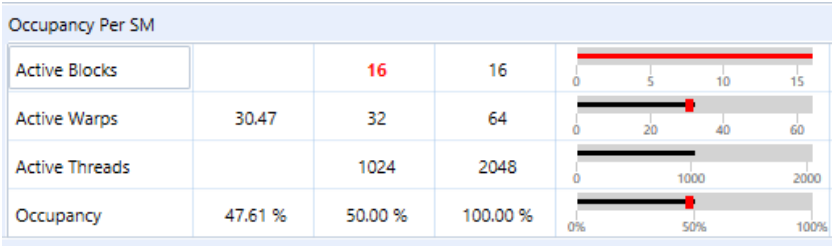
binnen de L1 cache gespilled wordt wordt aangegeven door de L1 hit rate.

Ook al ligt de occupancy hoger van de 8 macroblokken implementatie, omdat er meer operaties per CUDA block worden uitgevoerd, en bijgevolg ook dus meer registers nodig heeft, is er meer spilling van de L1 cache naar de L2 cache. De hierdoor extra toegevoegde memory bandwith is van grotere negatieve impact voor de performantie dan de gewonnen occupancy. Dit fenomeen is van grote impact bij de zigzag kernel. Bij de andere kernels treedt dit fenomeen ook op, maar is de impact op de performantie.

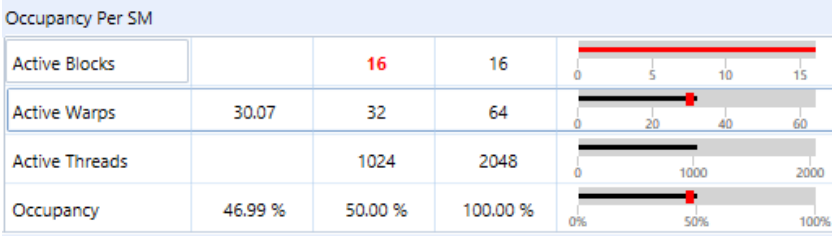
Bij de 4 macroblokken operatie bereikt men dus de balans tussen 2 aspecten. Een zo hoog mogelijke occupancy die ervoor zorgt dat CUDA latency effects kan maskeren en een hoge L1 hit rate zodat er geen extra memory bandwidth nodig is voor local memory operaties.



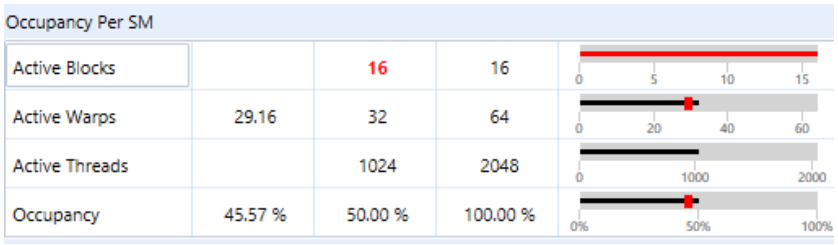
Figuur 5.12: Normalisatie kernel occupancy na optimalisatie



Figuur 5.13: DCT kernel occupancy na optimalisatie



Figuur 5.14: Kwantisatie kernel occupancy na optimalisatie



Figuur 5.15: Zigzag kernel occupancy na optimalisatie

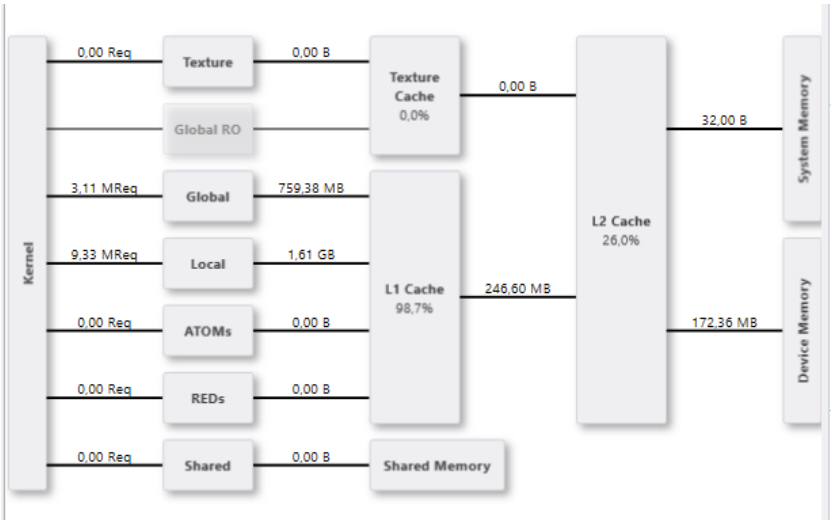
De enige resterende wijze om de kernel verder te optimaliseren is het verbeteren van het memory access pattern. De pixels van een frame zijn opgeslagen in een row-major matrix patroon. In alle kernels wordt gewerkt met macroblokken van 4x4 pixels. Dit heeft tot gevolg dat binnen een warp er pixels opgehaald worden met relatief grote strides tussen de pixels in memory. Bij een perfect coalesced memory access pattern kan CUDA al de requests die mekaar opvolgen in memory samenvoegen tot 1 grote transfer, maar omdat in de kernel non-sequentiële data wordt opgehaald is CUDA genoodzaakt deze transfers te verdelen over een groter aantal dan strikt noodzakelijk aan transfers. Omdat de CUDA transfers van vaste grootte zijn wordt er zo bandwidth verspild aan onbruikbare data.

De logische manier om data voor de I-frame kernels te coalescen zou dus zijn om macroblokken sequentieel in memory te storen. Al deze kernels accessen namelijk vaste macroblokken per CUDA block. Op deze manier zijn er geen strides nodig om data binnen een warp op te halen. In de huidige implementatie zou dit op twee manieren een positieve impact hebben. Ten eerste is dan, zoals eerder aangehaald, de memory access in de kernel perfect coalesced en ten tweede vallen er op die manier een heel aantal indexing berekeningen weg, omdat niet langer het aanspreken van een index in een macroblok vertaald dient te worden naar het ophalen van een pixel in het frame.

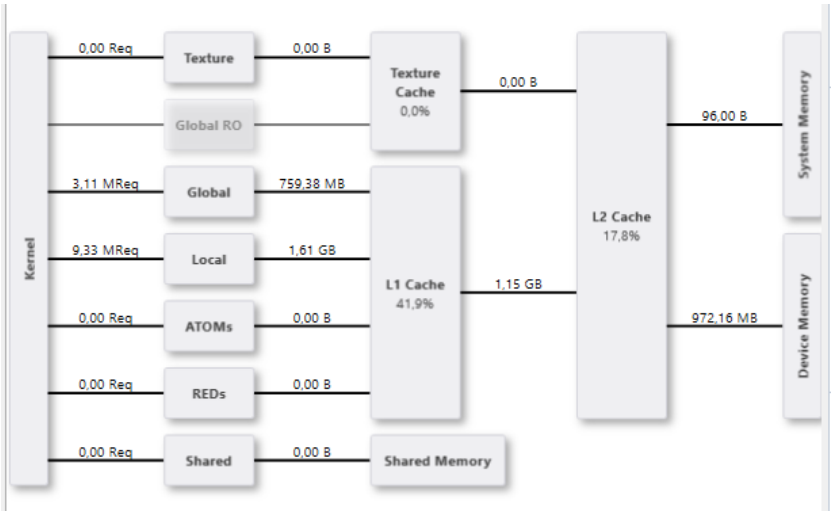
Er is wel een zeer significant nadeel verbonden aan deze optie. De Motion Estimation kernel (zie volgende sectie) heeft een access pattern dat hard steunt op random access van pixels in het frame. Als de pixels niet langer logisch gestructeerd zijn volgens het bron frame introduceert dit minder coalescing in de ME kernel. Sinds deze kernel in absolute termen van grotere impact is op de encoding snelheid van het frame is het best om te opteren van het memory access pattern niet te optimaliseren voor de I-frame kernels.

Belangrijk is wel op te merken dat als performantie van groter belang is dan het bereikte niveau van compressie er voor gekozen kan worden om geen P-frames en bijgevolg geen motion estimation te gebruiken. Hierdoor vervalt het eerder geformuleerde probleem en is het wel aan te raden om de data te

herschikken voor optimale performantie van de I-frame kernels.



Figuur 5.16: Zigzag kernel memory transfers bij 4 macroblokken / CUDA block



Figuur 5.17: Zigzag kernel memory transfers bij 8 macroblokken / CUDA block

5.4.3 Uitvoeringstijden

In tabellen 5.3, 5.4, 5.5 en 5.6 is een overzicht te vinden van de gemeten uitvoeringstijden van de verschillende deelstappen van de I-frame encoding

pipeline. De waarden zijn in microseconden en de gemarkeerde rijen zijn de uitvoeringstijden van de 4 componenten die een geïmplementeerde CUDA kernel hebben.

Opmerking De uitvoeringstijden van de *Read from storage* stap fluctueren bijzonder sterk. Dit is te verklaren omdat de resultaten verzameld zijn op een machine met een HDD. Afhankelijk van achtergrond operaties en caching van het besturingsysteem verschilt het ophalen van de frames sterk van run tot run. De getoonde cijfers zijn louter informatief en moeten niet in rekening gebracht worden bij de analyse van het uiteindelijke resultaat.

5.4.3.1 720p

Encoding Stap	CPU - 1 thread	CPU - 2 threads	CPU - 4 threads	CUDA
IFrame encoding	44625.4	28747.04	25931.12	18473.98
DCT	12737.61	6413.53	4412.25	404.21
Normalization	2302.64	1158.05	891	333.06
Quantization	10797.74	5427.42	4409.98	395.22
ZigZag	5736.53	2859.04	2229.83	413.96
RLE encoding	11957.79	12066.76	13247.94	12206.01
Pre-process frame	875.25	797.52	831.06	974.84
Read from storage	922.42	911.24	1030.77	38413.63
Swap context	0	0	0	657.66
Write to storage	1015.69	774.28	685.81	629.2

Tabel 5.3: 720p I-frame encoding resultaten

5.4.3.2 1080p

Encoding stap	CPU - 1 thread	CPU - 2 threads	CPU - 4 threads	CUDA
IFrame encoding	90943.03	54154.75	42345.65	32224.39
DCT	29079.95	14354.19	8700.03	849.93
Normalization	6353.84	2810.04	1853.49	878.29
Quantization	24033.71	12077.83	8021.87	842.52
ZigZag	12996.81	6599.23	4281.86	738.73
RLE encoding	17284.5	17342.03	18474.76	17425.24
Pre-process frame	2497.7	2224.32	2097.74	2283.64
Read from storage	45232.18	53665.13	44956.71	30147.4
Swap context	0	0	0	1300.78
Write to storage	1079.2	854.63	883.27	3104.48

Tabel 5.4: 1080p I-frame encoding resultaten

5.4.3.3 4k

Encoding Stap	CPU - 1 thread	CPU - 2 threads	CPU - 4 threads	CUDA
IFrame encoding	300023.46	180620.6	136102.14	95150.81
DCT	80214.28	40141.24	25181.45	3213.79
Normalization	22079.54	11993.16	6924.35	2605.55
Quantization	93180.05	49605.84	30379.37	3189.23
ZigZag	52383.84	27009.29	17189.84	2004.56
RLE encoding	49539.89	49291.17	53462.7	48994.74
Pre-process frame	12078.12	10269.5	9764.49	13513.74
Read from storage	191385.45	158097.38	162381.04	171320.48
Swap context	0	0	0	4854.87
Write to storage	2220.04	2166.56	2522.13	2214.17

Tabel 5.5: 4k I-frame encoding resultaten

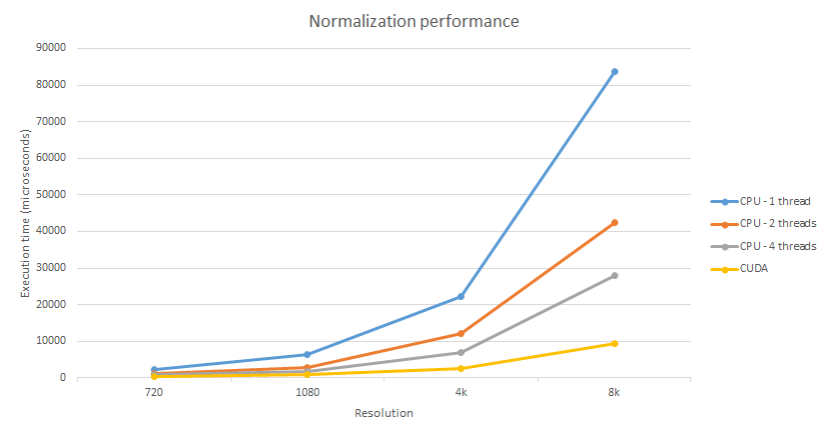
5.4.3.4 8k

Encoding Stap	CPU - 1 thread	CPU - 2 threads	CPU - 4 threads	CUDA
IFrame encoding	1186753.84	707175.15	555773.46	365827.07
DCT	318085.2	161171.89	108315.04	12576.54
Normalization	83675.54	42358.97	27874.2	9393.42
Quantization	368434.44	187216	129832.74	12443.89
ZigZag	208831.5	106783.94	73673.55	6621.0
RLE encoding	197049.15	197572.8	204494.8	196099.75
Pre-process frame	38544.93	38440.17	37684.01	40300.17
Read from storage	648036.04	752162.6	856397.58	661252.47
Swap context	0	0	0	17715.58
Write to storage	8697.44	10209.86	9616.53	8500.38

Tabel 5.6: 8k I-frame encoding resultaten

5.4.4 Analyse

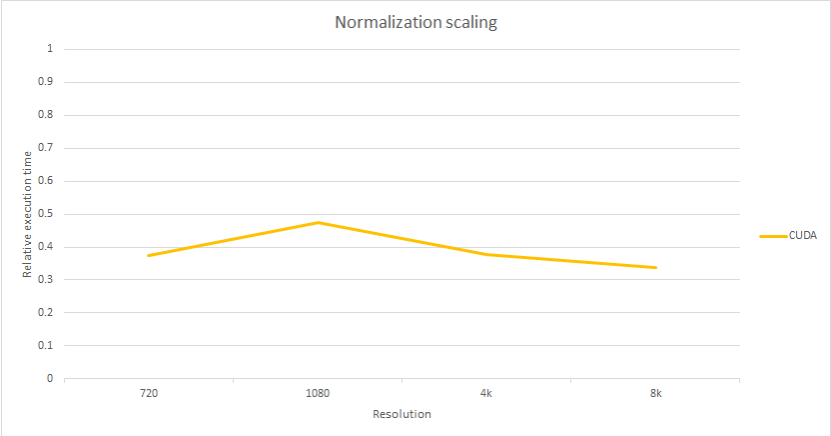
5.4.4.1 Normalisatie



Figuur 5.18: Normalisatie performance

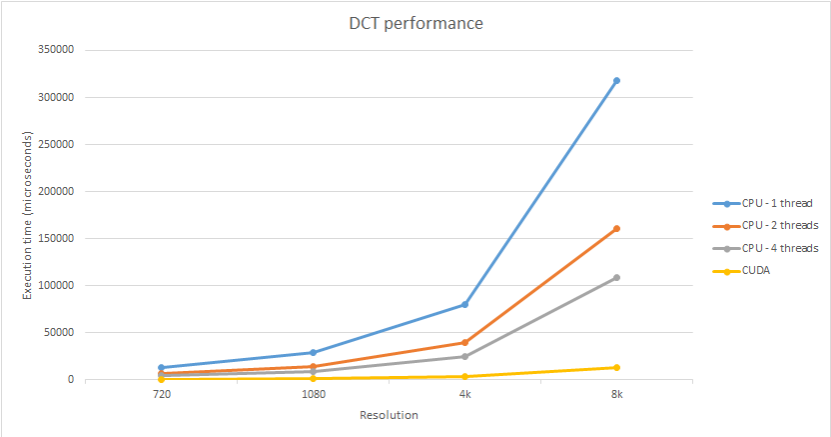
In figuur 5.18 valt de uitvoeringstijd van het normalisatie component te lezen naargelang de resolutie en dit voor zowel de CPU als de GPU code. De CUDA code is duidelijk sneller dan de CPU code maar in mindere mate dan de kernels die later besproken zullen worden. De verklaring hiervoor is dat de kernel bestaat uit 1 single-precision floating (SPF) operation aangevuld met vooral integer operations (IO) om data op te halen uit het geheugen. De overhead van de IO is hierdoor zo groot dat er minder performantie winst gehaald wordt uit de kernel.

In figuur 5.19 is de relatieve uitvoeringstijd van de CUDA code te zien ten opzichte van de 4 threads CPU code. Hierop valt te zien dat de CUDA code wel beter schaalt bij stijgende resolutie. Bij hoge resoluties zal dus ook voor de normalisatie kernel merkbare tijdswinst geboekt worden ten opzichte van de CPU implementatie.



Figuur 5.19: Normalisatie scaling

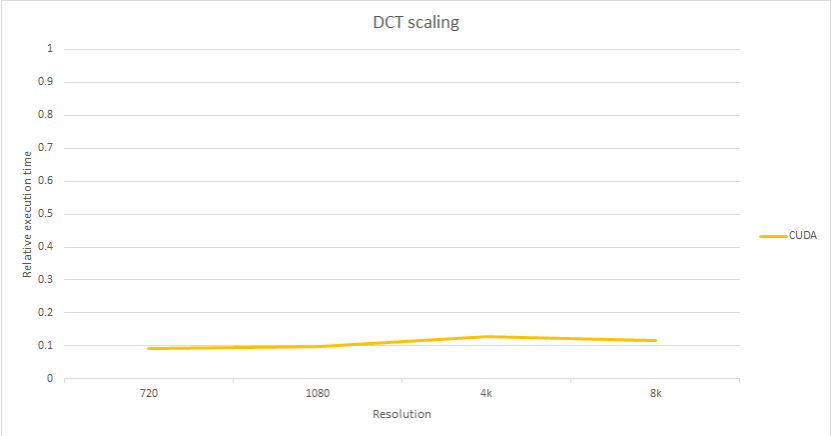
5.4.4.2 DCT



Figuur 5.20: DCT performance

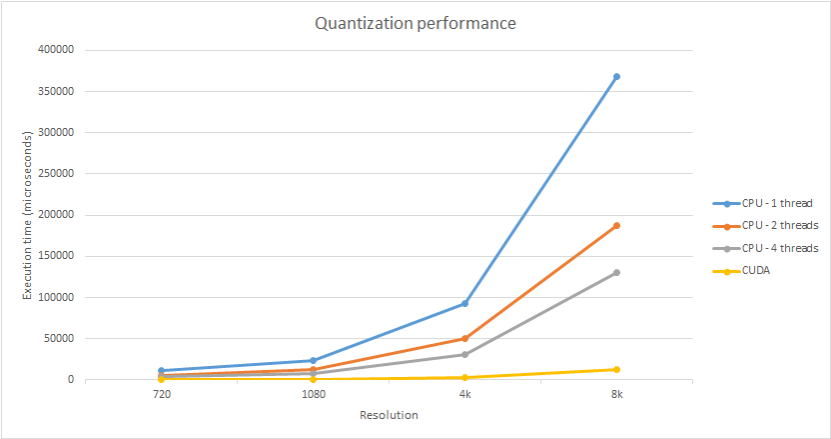
Als men naar de uitvoeringstijden van het DCT component in figuur 5.20 bekijkt is duidelijk dat DCT wel veel baat heeft bij de parallellisatie op de GPU. De hoeveelheid SPF operations is in de kernel voldoende hoog in vergelijking met de minder efficiënte IO operaties. Hierdoor is dit soort code in regel bijzonder snel uit te voeren in CUDA.

In figuur 5.21 is te zien dat bij stijgende resolutie de relatieve performantie van het algoritme min of meer stabiel blijft. Uitvoeren van DCT op de GPU is onafhankelijk van de resolutie een grote tijdswinst in vergelijking met de uitvoering op de CPU.



Figuur 5.21: DCT scaling ten opzichte van 4 threads CPU

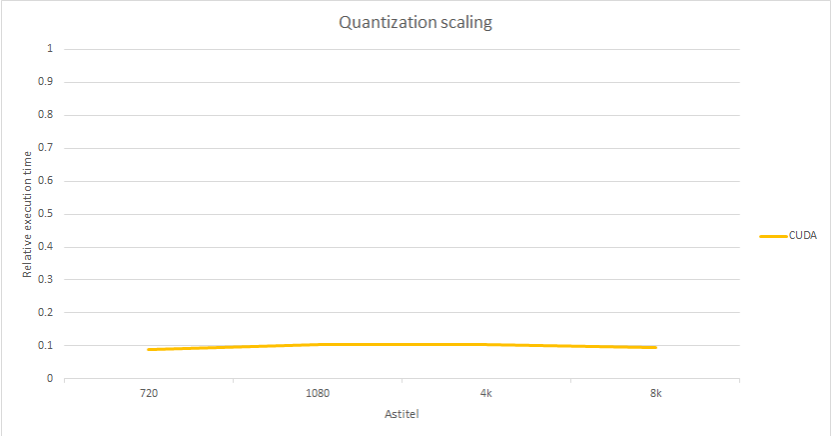
5.4.4.3 Kwantisatie



Figuur 5.22: Quantization performance

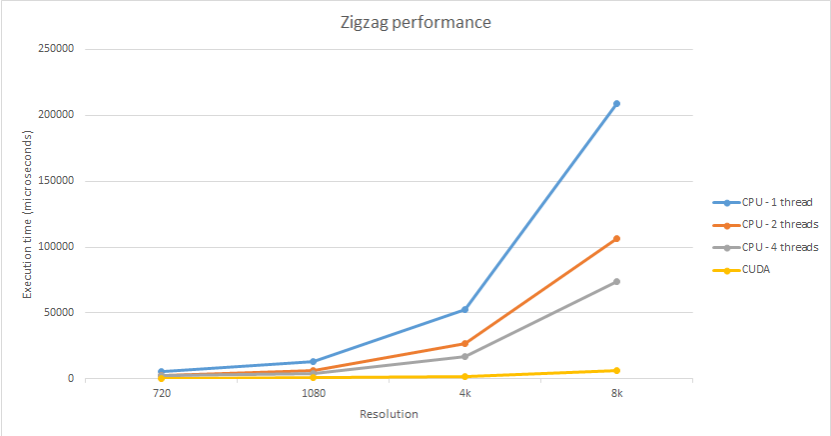
De kwantisatie kernel laat op het vlak van performantie, te zien in figuur 5.22 erg gelijkaardige resultaten optekenen als de DCT kernel. Ook al bevat deze kernel veel minder SPF operations dan de DCT kernel lijken er toch voldoende operaties aanwezig te zijn zodat CUDA significante performantie verschillen kan optekenen ten opzichte van de CPU code.

Net als bij de DCT kernel blijft de performantie van de kernel stabiel bij stijgende performantie (zie figuur 5.23). Ook hier is bij alle resoluties het gebruik van CUDA een grote verbetering ten opzichte van de CPU code.



Figuur 5.23: Kwantisatie schaling ten opzichte van 4 threads CPU

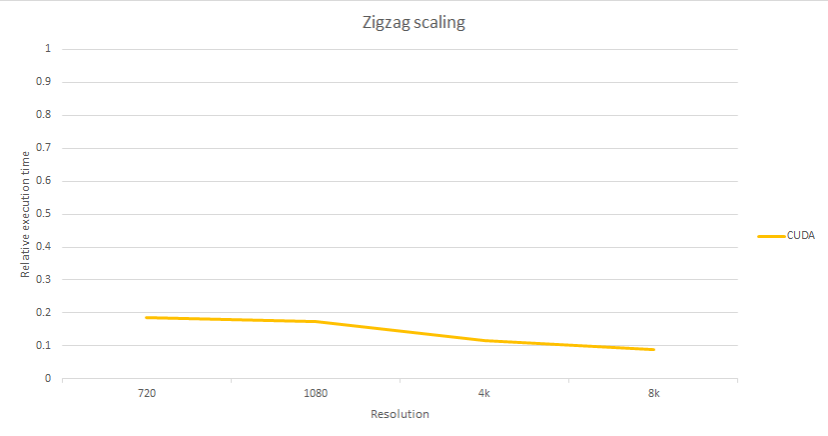
5.4.4.4 Zigzag ordering



Figuur 5.24: Zigzag performance

Uit figuur 5.24 blijkt ook hier dat de CUDA code een pak sneller is dan de CPU code.

Uit de schaling in figuur 5.25 blijkt dat de kernel relatief gezien iets slechtere performantie heeft dan de DCT en kwantisatie kernel. Maar de zigzag kernel schaalt wel zeer goed bij stijgende resoluties en bij 8k is het performantie verschil tussen de DCT en kwantisatie kernel minimaal. De vaststelling is dus ook hier weer dat CUDA altijd zeer grote performantie winsten oplevert ten opzichte van de CPU code.



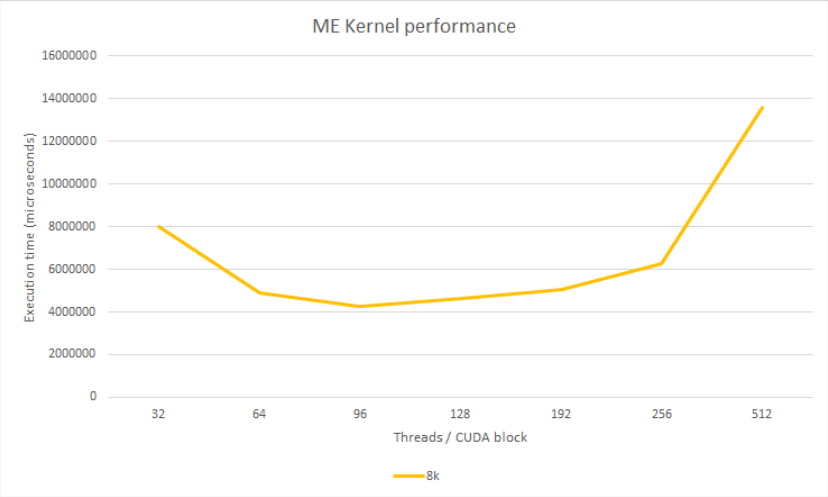
Figuur 5.25: Zigzag schaling ten opzichte van 4 threads CPU

5.5 P-Frame encoding pipeline

In deze sectie wordt de performantie onderzocht van de Motion Estimation kernel. Eerst wordt net als voor de kernels van de I-frame pipeline onderzocht wat het meest optimaal aantal threads is voor de kernel. Daarna wordt de meest optimale implementatie vergeleken met de CPU implementatie en wordt er analyse gemaakt rond de schaling van het algortime zowel op vlak van resolutie als op vlak van ME range.

5.5.1 Kernel performantie tuning

Om de optimale grootte van de threadpool voor de ME kernel te bepalen werd net zoals bij de I-frame kernels op verschillende resoluties de execution time van de kernel getimed bij een reeks logische threadpool groottes. Het resultaat hiervan is te zien in figuur 5.26. Het is vrij duidelijk dat de kernel optimale performantie bereikt bij een threadpool van 96 threads. In figuur 5.27 is te zien dat bij een threadpool grootte van 96 er een occupancy bereikt wordt van 70%. In regel genomen is een occupancy van 50% voldoende om de meeste latency binnen een kernel weg te werken dus werd er gekozen om voor de uitvoering van de volgende experimenten gebruik te maken van een threadpool van 128 threads. In figuur 5.28 is een overzicht van de memory transfers te zien in de kernel, hier is voornamelijk de L1 hit rate van belang.



Figuur 5.26: Kernel performance in functie van het aantal threads

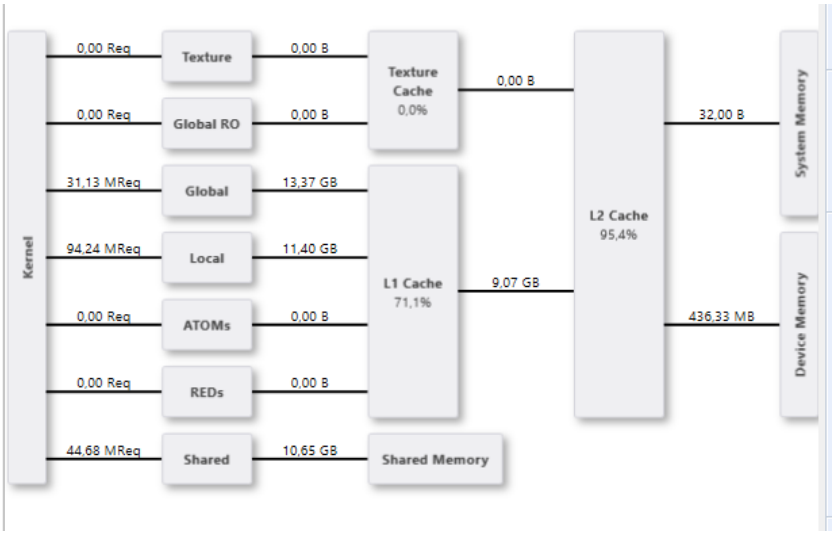
Occupancy Per SM

Active Blocks		16	16	0 5 10 15
Active Warps	44.16	48	64	0 20 40 60
Active Threads		1536	2048	0 1000 2000
Occupancy	69.00 %	75.00 %	100.00 %	0% 50% 100%

Figuur 5.27: Kernel occupancy bij een threadpool grootte van 96

ME-720p-SMEM 32-96 Threads	ME-720p-SNEM 32-128 Threads	ME-720p-SNEM 48-128 Threads
114052	126580	121493

Tabel 5.7: 720p Shared Memory performance impact

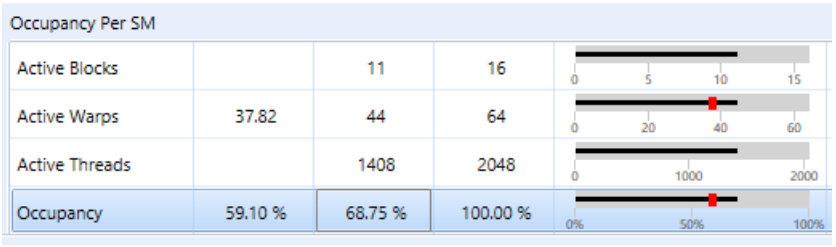


Figuur 5.28: Kernel memory transactions bij een threadpool grootte van 96

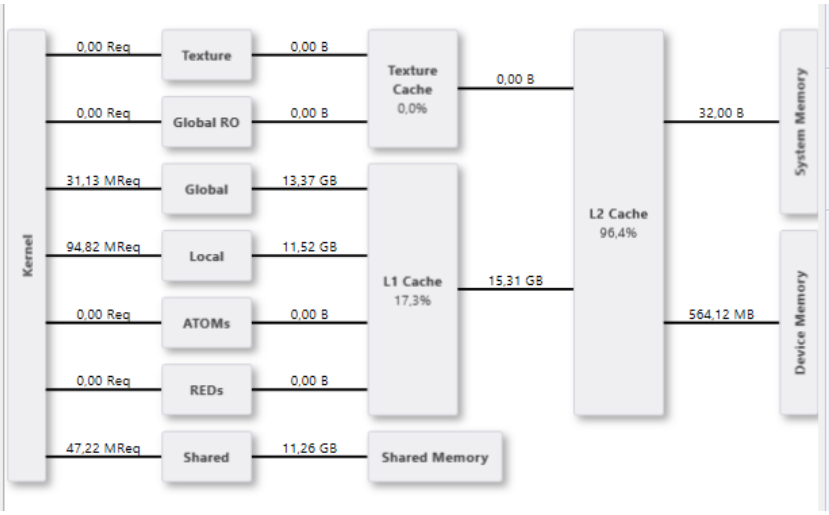
Ook hier kan de vaststelling gedaan worden dat optimale performantie niet bereikt wordt bij een occupancy van 100%. Bij een threadpool van 96 threads wordt er de ideale combinatie bereikt van 2 aspecten. Bij 96 threads is het zo dat de footprint van het benodigde shared memory per CUDA block net onder de 2kB blijft. De device limit van het aantal actieve CUDA blocks per SM is 16. De totale shared memory footprint van 16 CUDA blocks is net kleiner dan 32kB. Dit maakt het mogelijk om in plaat van 48kB te reserveren voor shared memory (de default) er slechts 32kB gereserveerd dient te worden voor shared memory. Het extra vrijgekomen geheugen wordt dan gebruikt door L1 cache, wat ten goede komt van de L1 hit rate. Bij een grotere thread pool wordt men geconfronteerd met het probleem dat ofwel meer shared memory dient gereserveerd te worden voor threads om hoge occupancy te behouden, met als gevolg dat er meer L2 register spilling is. Ofwel een minder hoge occupancy halen omdat CUDA te weinig shared memory per SM ter beschikking heeft om 16 CUDA blocks tegelijk actief te hebben op 1 CUDA block. Beide opties resulteren in een verlies in performantie, wat te zien is tabel 5.7. Hierin is de uitvoeringstijd in microseconden te zien van de 3 beschreven opties.

In figuur 5.29 is te zien wat de impact op de occupancy is bij een threadpool van 128 threads te gebruiken met slechts 32kB shared memory ter

beschikking. De occupancy daalt hierdoor van 69% naar 59%. In figuur 5.30 is dan weer te zien wat de impact is op memory transfers bij het verhogen van de grootte van het shared memory van 32kB naar 48kB. De L1 hit rate daalt van 71% naar 17%



Figuur 5.29: Kernel occupancy bij een threadpool grootte van 128 en shared memory grootte van 32kB



Figuur 5.30: Kernel memory transcations bij een threadpool grootte van 128 en shared memory grootte van 48kB

Bij een kleinere tread pool is onstaat het omgekeerde probleem. Er is een groot verlies aan occupancy zonder dat er ergens anders winst gemaakt kan worden op het vlak van performantie. Pas wanneer de threadpool gereduceerd wordt tot 32 threads is het mogelijk het benodigd shared memory te reduceren tot 16kB. Maar hierdoor is de occupancy gezakt onder 25%. Dit verlaagt de performantie vele malen meer dan de winst geboekt door de hogere L1 hit rate.

De finale conclusie die hieruit genomen kan worden is dat een threadpool van 96 threads met een shared memory grootte van 32kB ideale performantie geeft voor de kernel.

Bij het optimaliseren van het memory access pattern dienen twee vragen gesteld te worden. Ten eerste in welke mate wordt het shared memory gebruikt en ten tweede in welke mate is global memory access in de kernel coalesced.

Wanneer men de kernel naïef analyseert lijkt het logisch om iedere pixel van de ME search window te cachen in shared memory. Op die manier is er geen herhaling van access in het global memory. Het nadeel van deze redenering is dat hierdoor de grootte van het benodigt shared memory afhankelijk wordt van de gekozen ME range. Bovendien verhoogt dit de grootte van het benodigt shared memory bijzonder veel bij hogere ME ranges. Bij een ME range van 16 heeft men al (16+16+1)*(16+16+1)*4bytes nodig oftewel 4.2kB nodig om alle pixel waarden op te slaan van het search window in shared memory. Als men weet dat er per SM slechts 64kB geheugen beschikbaar is om optimaal 16 macroblokken te runnen is het makkelijk in te zien dat de occupancy zwaar beïnvloed wordt door de hoeveelheid shared memory benodigt per CUDA block. Het is beter om de shared memory footprint constant te houden binnen een CUDA block omdat dit het gedrag van de kernel op het vlak van occupancy voorspelbaar maakt.

Omdat er geen shared memory wordt gebruikt wordt voor het ophalen van de pixels binnen een macroblok is het natuurlijk belangrijk dat het ophalen van de data zo coalesced mogelijk is. Dis is in de ME kernel het geval en makkelijk in te zien als men stilstaat bij de werking van het algoritme. Iedere thread van een CUDA block is verantwoordelijk voor het ophalen van het macroblock geassocieerd met zijn pixel. Het ophalen van deze pixels gebeurt iteratief in iedere thread. Dit betekent dat de kernels binnen een warp pixels aanspreken die op elkaar aansluiten in het geheugen. Global memory access is dus coalesced in de ME kernel. Het enige probleem is dat bij het ophalen van data in iedere iteratie pixels worden opgehaald die al door andere threads zijn opgehaald. Het voorkomen van dit patroon is enkel mogelijk door te cachen in shared memory wat de al eerder aangehaalde problemen met zich meebrengt. Voor global memory access is het wel zo dat CUDA gebruik maakt van caching om access naar het global memory te beperken. Wanneer een pixel al in een cache zit (L1 of L2) kan dit hieruit gehaald worden in plaats van uit het global memory wat in het geval van de ME kernel een belangrijke factor is voor de performantie.

	ME 16 - CPU	ME 32 - CPU	ME 16 - CUDA	ME 32 - CUDA
Encode MC data	28,183.39	34,068.64	17,460.33	18,892.18
PFrame encoding	2,969,950.16	13,777,713.21	141,585.35	327,719.23
Perform ME/MC	2,942,409.36	13,739,227.44	114,518.99	300,059.63
Copying data	1,008.38	1,246.31	243.51	261.44
Prepare MC data	3,867.05	4,350.57	2,962.17	3,119.64
Read from storage	1,301.93	25,628.61	1,245.37	1,227.36
Store MC data	1.18	1.28	1.17	1.19
Store Motion Vectors	3,740.70	4,395.63	4,345.01	5,063.65
Swap context	0	0	882.77	868.33
Write to storage	659.99	788.69	749.44	788.3

Tabel 5.8: 720p P-frame pipeline uitvoeringstijden

	ME 16 - CPU	ME 32 - CPU	ME 16 - CUDA	ME 32 - CUDA
Encode MC data	70,567.65	65,194.43	37,209.40	35,915.07
PFrame encoding	7,339,887.24	28,791,201.53	318,265.70	770,057.64
Perform ME/MC	7,268,074.63	28,722,244.85	257,838.47	712,383.70
Copying data	2,723.52	3,288.03	523.54	526.67
Prepare MC data	7,969.38	8,547.02	6,467.62	6,647.83
Read from storage	3,062.50	122,272.83	3,226.14	2,890.28
Store MC data	1.45	1.25	1.25	1.3
Store Motion Vectors	8,924.00	10,511.00	11,429.31	11,827.64
Swap context	0	0	1,904.70	1,888.52
Write to storage	1,253.50	1,167.33	1,187.95	1,204.73

Tabel 5.9: 1080p P-frame pipeline uitvoeringstijden

5.5.2 Uitvoeringstijden

In tabellen 5.8, 5.9, 5.10 en 5.11 tonen de uitvoeringstijden van de verschillende componenten van de P-frame pipeline. De tijden zijn ook hier in microseconden en de gemarkeerde rij is de uitvoeringstijd van het ME component. De CUDA uitvoeringstijden werden vastgesteld met de eerder bepaalde optimale instellingen voor kernel occupancy. De CPU uitvoeringstijden werden vastgesteld door de code te runnen met 4 threads.

Er werd gekozen om de data te tonen bij een ME range van 16 en 32. Dit om de data overzichtelijk te houden en omdat deze twee waarden een vrij goede representatie geven van de te verwachten performance met andere ME ranges.

5.5.3 Schaling

Uit de tabellen is makkelijk vast te stellen dat zoals verwacht motion estimation erg veel baat heeft bij parallellisatie op de GPU. Bij hogere ME range

	ME 16 - CPU	ME 32 - CPU	ME 16 - CUDA	ME 32 - CUDA
Encode MC data	283,044.24	257,895.92	123,966.49	133,919.63
PFrame encoding	24,297,269.63	74,382,502.31	1,235,236.27	3,450,085.34
Perform ME/MC	24,062,362.46	74,216,532.36	1,031,591.04	3,236,202.07
Copying data	11,790.73	12,731.06	636.83	596.63
Prepare MC data	29,908.00	34,448.32	25,230.63	24,572.10
Read from storage	13,694.84	194,144.15	11,589.79	11,156.09
Store MC data	1.25	1.05	1.09	1.1
Store Motion Vectors	30,960.32	33,153.30	36,081.30	36,916.35
Swap context	0	0	7,098.40	6,991.17
Write to storage	4,299.33	3,724.06	3,957.03	3,939.57

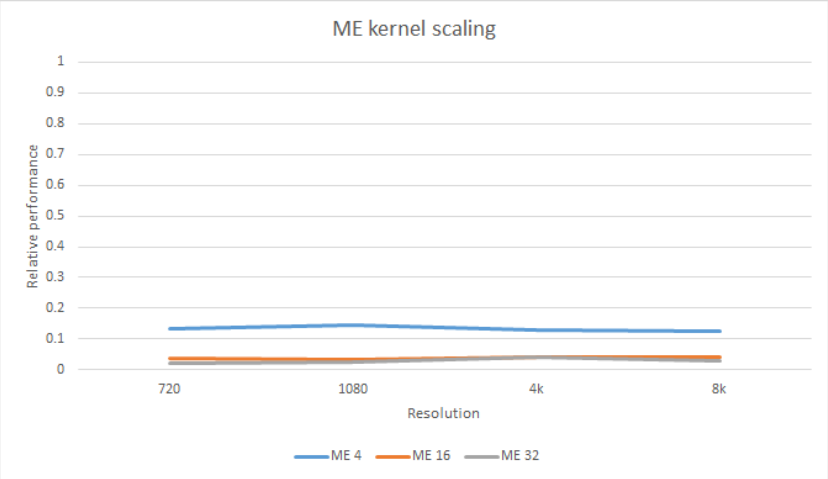
Tabel 5.10: 4k P-frame pipeline uitvoeringstijden

	ME 16 - CPU	ME 32 - CPU	ME 16 - CUDA	ME 32 - CUDA
Encode MC data	899,511.06	925,659.91	479,096.47	761,624.17
PFrame encoding	112,584,081.81	391,785,727.07	5,304,230.63	13,388,493.44
Perform ME/MC	111,646,794.31	390,967,472.28	4,667,893.40	12,321,169.18
Copying data	38,380.43	45,448.09	984.94	967.99
Prepare MC data	124,070.29	140,149.30	93,477.91	93,833.30
Read from storage	48,132.58	326,696.89	44,201.80	71,234.94
Store MC data	1.40	1.11	1.13	1.41
Store Motion Vectors	116,816.56	129,712.79	138,712.69	139,343.36
Swap context	0	0	27,863.73	27,863.73
Write to storage	15,182.21	15,438.28	14,961.31	18,020.51

Tabel 5.11: 8k P-frame pipeline uitvoeringstijden

waarden is de kernel zelfs tot 20 keer sneller dan de CPU code. Als men kijkt naar de grafiek 5.31 is ook goed te zien dat de relatieve performantie van de CUDA code stabiel blijft over de resoluties heen.

Men kan wel vaststellenp dat de performantie van de CUDA code relatief gezien hoger ligt bij ME range 16 en 32 dan bij ME range 4. Er kan dus vastgesteld worden dat de CUDA code beter schaalt bij hogere ME ranges dan de CPU code, wat een belangrijke vaststelling is daar men voor hoge resolutie video vaak genoodzaakt is om hoge ME ranges te gebruiken.



Figuur 5.31: Kernel performance ten opzichte van de CPU bij ME range 4

Hoofdstuk 6

Conclusies en toekomstig werk

Het doel van de thesis was onderzoek te voeren naar de mogelijkheden om een video encoder te realiseren op de GPU. Bovendien werd een implementatie gerealiseerd in CUDA en werd deze implementatie vergeleken met een multi-threaded CPU implementatie.

Voor het onderzoek naar mogelijkheden tot het implementeren van een video encoder werd een uitgebreide literatuurstudie uitgevoerd. In de eerste plaats werd een analyse gemaakt welke onderdelen een typisch video encoder bevat en wat de functionaliteit van ieder van deze onderdelen is. In een tweede stap werd dan onderzocht op welke manier deze onderdelen in parallel geïmplementeerd zouden kunnen worden. Deze implementaties zouden dan gebruikt kunnen worden als basis voor een eventuele implementatie op de GPU. Tenslotte werd onderzocht hoe deze onderdelen op de GPU geïmplementeerd zouden kunnen worden en welke algoritmen hiervoor bekend zijn. De focus van het meeste onderzoek hier rond ligt bij de motion estimation. Omdat deze stap het meest rekenintensief is en ook het meest baat heeft bij parallellisatie op grote schaal focust de meeste literatuur zich op het parallelliseren van allerhande algoritmen voor motion estimation.

Voor het tweede aspect van de thesis werd een eigen implementatie gerealiseerd. Deze implementatie werd gerealiseerd in NVIDIA CUDA. De gerealiseerde encoder kan YCrCb 4:2:0 video data comprimeren, hierbinnen werden de meeste relevante componenten van een typische video encoder gerealiseerd. Er werd gekozen om geen intra predictie te realiseren omwille van de complexiteit die tot zou toevoegen, waardoor de praktische haalbaarheid van de thesis in gedrang zou komen.

6.1 Conclusies

Net als bij de analyse van de resultaten worden de conclusies van de thesis opgedeeld een 2 delen. Het eerste deel focust op conclusies die gemaakt kunnen worden in verband met de I-frame encoding pipeline. Een tweede deel focust op conclusies in verband met de P-frame pipeline.

In de I-frame pipeline kon vastgesteld worden dat alle kernels zeker baat hebben bij de parallellisatie mogelijkheden van de GPU. Zowel DCT, kwantisatie en zigzag toonde grote performantie winsten ten opzichte van de CPU code, normalisatie toonde deze perfomance gains niet maar is nog steeds performanter dan de CPU code. Samengevat kan er gesteld worden dat er geen reden is om deze algoritmen niet uit te voeren op de GPU, de gewonnen performantie is erg hoog en compenseert makkelijk de geïntroduceerde latency van het kopiëren van data van Host naar Device. Bovendien zullen nieuwe toevoegingen aan CUDA zoals Unified Memory de performantie van deze operaties alleen maar gaan reduceren.

Een tweede belangrijke conclusie binnen de I-frame pipeline is de grote impact van RLE encoding op de totale uitvoeringstijd van de pipeline. In de CUDA implementatie is deze namelijk voor 50% van de totale uitvoeringstijd. Verder onderzoek naar optimalisatie van deze stap is dan ook noodzakelijk om nog grote performantie winsten te boeken binnen de I-frame encoding pipeline.

In de P-frame encoding pipeline vallen grotere performance gains te zien dan in de I-frame encoding pipeline. De totale uitvoeringstijd van motion estimation is veel groter dan van de I-frame encoding zodat de absolute snelheidswinsten zeker een pak hoger zijn dan bij I-frame encoding. De CUDA code schaalt ook beter bij stijgende motion estimation range wat bij voor hoge resoluties, waar typisch hogere motion estimation ranges noodzakelijk zijn, een belangrijke vaststelling is.

6.2 Toekomstig werk

Voor toekomstig onderzoek zijn er 2 grote afgetekende pistes. Een eerste piste is het onderzoek focussen op het toepassen van de opgedane kennis in een realistische context. Met de opkomst van 4K en de daaraan gekoppelde nieuwe HEVC standaard (H.265) is het zeker relevant om te onderzoeken welke performantie winsten GPU parallellisatie daar zou kunnen bieden.

Een tweede piste is het huidig onderzoek verder zetten en uitbreiden met de gedane conclusies. Als belangrijkste focuspunt zou dan het onderzoek naar een meer performante ME encoding algoritme gekozen worden, daar

dit de grootste absolute performantie winsten nog kan opleveren. Andere opties zijn het verder optimaliseren van de CUDA kernels en onderzoek naar parallellisatie van het RLE algoritme.

Bovendien zou ook toevoeging van weggelaten features zoals intra predictie en entropy encoding een belangrijke onderzoekspiste kunnen zijn, daar deze algoritmen doorgaans zeer slecht te parallelliseren zijn. Onderzoek naar intra predictie zou zich kunnen focussen op het vergelijken van al bestaande technieken, HEVC definieert 33[40] intra prediction directions en onderzoek naar de parallellisatie van deze technieken zou een logische stap zijn. Voor entropy encoding is een belangrijke volgende stap het onderzoeken in welke mate GPU computing gebruikt zou kunnen worden voor de parallellisatie van Huffman encoding. Deze stap is erg belangrijk voor het bereiken van goede compressie ratio's en onderzoek naar het versnellen van deze stap is dan ook een logische volgende stap.

Bibliografie

[1] Iain E. Richardson. *The H.264 Advanced Video Compression Standard*, chapter 3. Wiley, 2010.

[2] Deepak Turaga and Mohamed Alkanhal. Search algorithms for block-matching in motion estimation. `https://www.ece.cmu.edu/~ee899/project/deepak_mid.htm`, 1998.

[3] `http://en.pudn.com/downloads63/sourcecode/graph/texture_mapping/detail220832_en.html`.

[4] Quantization (image processing). `http://en.wikipedia.org/wiki/Quantization_(image_processing)`.

[5] `http://stackoverflow.com/questions/2292559/zig-zag-scan-algorithm`.

[6] Abdou Youssef. Parallel algorithms for entropy-coding techniques. Technical report, The George Washington University, 1998.

[7] J. C. Fernández and M. P. Malumbres. A parallel implementation of h.26l video encoder. *Euro-Par 2002 Parallel Processing*, 2002.

[8] Liang-Gee Chen, Wai-Ting Chen, Yeu-Shen Jehng, and Tzi-Dar Chiuch. An efficient parallel motion estimation algorithm for digital image processing. *Circuits and Systems for Video Technology*, 1991.

[9] J. Chen and K.J.R. Liu. A complete pipelined parallel cordic architecture for motion estimation. *Circuits and Systems II: Analog and Digital Signal Processing*, 1998.

[10] P. Baglietto, M. Maresca, A. Migliaro, and M. Migliardi. Parallel implementation of the full search block matching algorithm for motion estimation. *International Conference on Application Specific Array Processors*, 1995.

[11] Genhua Jin and Hyuk-Jae Lee. A parallel and pipelined execution of h.264/avc intra prediction. *The Sixth IEEE International Conference on Computer and Information Technology*, 2006.

[12] Wonjae Lee, Seongjoo Lee, and Jaeseok Kim. Pipelined intra prediction using shuffled encoding order for h.264/avc. *TENCON*, 2006.

[13] Angela Di Serio. Analytical evaluation of the 2d-dct using paralleling processing. *CLEI ELECTRONIC JOURNAL*, 1998.

[14] N. Manchev. Parallel algorithm for run length encoding. *Third International Conference on Information Technology: New Generations*, 2006.

[15] Ana Balevic. Fine-grain parallelization of entropy coding on gpgpus. `http://tesla.rcub.bg.ac.rs/~taucet/docs/papers/hipeac_poster_ab.pdf`, 2009.

[16] `gpgpu.org/about`.

[17] `http://www.cyberlink.com/products/mediaespresso/overview_en_EU.html`.

[18] `http://www.fastvideo.ru/english/products/software/cuda-jpeg-encoder.htm`.

[19] Video capture, encoding, and streaming in a multigpu system. `http://www.nvidia.com/docs/IO/40049/TB-Quadro_VideoCaptureStreaming_v01.pdf`, 2010.

[20] Erich Marth and Guillermo Marcus. Parallelization of the x264 encoder using opencl. `http://li5.ziti.uni-heidelberg.de/x264gpu/`.

[21] Chi-Wang Ho, Oscar C. Au, S.-H. Gary Chan, Shu-Kei Yip, and Hoi-Ming Wong. Motion estimation for h.264/avc using prgrammable graphics hardware. *IEEE International Conference on Multimedia and Expo*, 2006.

[22] E. Monteiro and et al. Real-time block matching motion estimation onto gpgpu. *19th IEEE International Conference on Image Processing (ICIP)*, 2012.

[23] R. Rodriguez, J.L. Martinez, G. Fernandez-Escribano, J.M. Claver, and J.L. Sanchez. Accelerating h.264 inter prediction in a gpu by using cuda. *Digest of Technical Papers International Conference on Consumer Electronics (ICCE)*, 2010.

[24] Lawrence Chan, Jae W. Lee, Alex Rothberg, and Paul Weaver. Parallelizing h.264 motion estimation algorithm using cuda. Technical report, Massachusetts Institute of Technology, 2009.

[25] M. C. Kung, Oscar Au, Peter Wong, and Chun-Hung Liu. Intra frame encoding using programmable graphics hardware. *Advances in Multimedia Information Processing*, 2007.

[26] Bruno Alexandre de Medeiros. Video coding on multicore graphics processors (gpus). Master's thesis, Universidade Técnica de Lisboa, 2012.

[27] Bo Fang and et al. Techniques for efficient dct/idct implementation on generic gpu. *IEEE International Symposium on Circuits and Systems*, 2005.

[28] `http://www.elementaltechnologies.com/products/server/professional-video-transcoding`.

[29] `http://www.digitalrapids.com/en/Products/TouchStream.aspx`.

[30] `http://www.digitalrapids.com/en/Products/Flux.aspx`.

[31] `http://www.sensoray.com/products/819.htm`.

[32] `http://www.hauppauge.co.uk/site/products/data_colossus.html`.

[33] Nvenc – nvidia kepler hardware video encoder. `https://developer.nvidia.com/sites/default/files/akamai/cuda/files/CUDADownloads/NVENC_AppNote.pdf`.

[34] What is cuda. `https://developer.nvidia.com/what-cuda`.

[35] Manfred Liebmann Ronan Amorim, Gundolf Haase and Rodrigo Weber dos Santos. Comparing cuda and opengl implementations for a jacobi iteration. *International Conference on High Performance Computing & Simulation*, 2009.

[36] Andreas Klöckner. Cuda vs opencl: Which should i use? `http://wiki.tiker.net/CudaVsOpenCL`.

[37] `http://www.visual-experiments.com/blog/wp-content/uploads/2010/05/opencl_logo.jpg`.

[38] Cuda c programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[39] Cuda. http://en.wikipedia.org/wiki/CUDA.

[40] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding standard. *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY*, 2012.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
GPU Computing voor Video Encoding Algoritmen

Richting: **master in de informatica-multimedia**
Jaar: **2014**

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.

Voor akkoord,

Dubois, Gert

Datum: **3/02/2014**