DIRECT SDI GPU COMMUNICATION

STEFAN DESSENS

Dhr. Gerard Tuk

Dhr. Sjaak v. Peski

Dhr. Peter IJkhout

Dhr. Arjan Houben

Technische Informatica
De Haagse Hogeschool – Delft
Juni 2013 – version 1 (initial release)

VOORWOORD

Tijdens deze afstudeerstage heb ik een hoop geleerd. Over het coördineren van een groot softwarepakket met git & cmake, en over het ontwikkelen van software in het algemeen.

Graag wil ik Arjan Houben bedanken voor het in goede banen leiden van deze opdracht en het geven van antwoorden op alle development gerelateerde vragen. Alsmede Peter IJkhouyt, CTO bij VidiGo, voor het mogelijk maken van deze opdracht.

Tevens wil ik de docenten Gerald Tuk en Sjaak van Peski bedanken voor het beoordelen en feedback leveren op alle documenten. Ten slotte wil ik de Haagse Hogeschool bedanken voor het aanbieden van een opleiding waar ik het de afgelopen jaren erg naar mijn zin heb gehad.

Ik wens de lezer veel plezier bij het doorlezen van dit Afstudeerverslag.

SAMENVATTING

VidiGo is een bedrijf gevestigd in Amsterdam wat software maakt voor o.a. live video streaming. Voor deze taak worden er desktops ingezet met een aantal SDI in- en outputs. SDI is een protocol wat gebruikt wordt om digitale audio en video tussen verschillende systemen (camera's, beeldschermen) te verplaatsen.

Het aantal in en output streams dat per desktop kan aangesloten is beperkt. Onder andere vanwege latency en bandwidth constraints. Er bestaan twee verschillende technieken die deze constraints zouden verlichten: NVIDIA GPUDirect en AMD SDI-Link. Deze technieken zouden een positief effect moeten hebben op onder andere de latency en benodigde memory bandwidth.

Om dat te bewijzen dan wel ontkrachten is er tijdens dit project software ontwikkeld waarbij NVIDIA GPUDirect en AMD SDI-Link kan worden vergeleken met de al bestaande oplossing.

Daarbij merkten we dat, bij de NVIDIA Quadro 4000, het aantal mogelijke input streams licht afnam met ongeveer 10%. En dat het aantal output streams toeneemt met ongeveer 50% tot 100%. Bij testen met een AMD FirePro V7900 nam het aantal mogelijke input streams af met ongeveer 40%, en nam het aantal mogelijke output streams toe met 133%.

De conclusie die we hieruit kunnen trekken is dat beide technieken een grote winst opleveren, bij output streams. In sommige gevallen kan de maximale hoeveelheid gelijktijdige output streams meer dan verdubbelen. Bij de huidige techniek is dat tevens de meest beperkende factor. Of deze techniek daadwerkelijk ingezet kan worden hangt af van het aanbod van grafische kaarten: de enige kaart die AMD aanbiedt met SDI-Link ondersteuning is de FirePro V7900, deze videokaart is te traag om veel streams aan te sturen. AMD bied wel snellere hardware aan, maar deze ondersteunen AMD SDI-Link niet. NVIDIA heeft een breder aanbod aan kaarten, waaronder de Quadro K5000, deze zou wel geschikt zijn voor grote hoeveelheden streams.

Bij het gebruik van deze kaart wordt verwacht, gebaseerd op de tests met de Quadro 4000, dat in elk geval het aantal mogelijke output streams met ongeveer 100% toeneemt. Waardoor dit dus zeker een zinvolle technologie is voor klanten die veel output streams vereisen. Voor input streams kan altijd de oude techniek worden gebruikt.

CONTENTS

1 INLEIDING

VidiGo is een bedrijf gevestigd in amsterdam met ongeveer 20 werknemers. VidiGo ontwikkelt software voor live audio en video streaming applicaties. De klanten van VidiGo zijn bijvoorbeeld SBS6, RTL, Radio 538, NOS, enzovoorts.

Een van de producten van VidiGo is VidiGo Live, deze richt zich op live Video streaming. Daarvoor worden destkops gebruikt met een groot aantal SDI en en outputs, de desktops bewerken het beeld, en sturen het weer door. Het maximaal aan te sturen streams is onderhevig aan bottlenecks, zoals latency, bandwidth, processing power, enzovoorts.

Het doel van deze opdracht is het onderzoeken van verschillende technologiën waarmee deze bottlenecks verlicht kunnen worden, zodat een lagere latency of meer streams tegelijkertijd mogelijk worden.

2

OPDRACHT

De audio en videobeelden worden bewerkt op de GPU. De audio en/of video (AV) komt op het systeem aan door middel van SDI (Serial digital interface) op een aparte PCI-express insteekkaart. De AV moet daardoor van de SDI kaart naar de GPU worden gekopieerd. Bij SDI-output gebeurt hetzelfde, maar dan in omgekeerde volgorde.

De AV kan niet direct van de SDI naar de GPU worden gekopieerd, dat moet via het systeemgeheugen, het heen-en-weer kopiëren van de data levert een flinke bottleneck op, wat de maximale hoeveelheid streams beperkt. Daarnaast leveren alle kopiëerstappen extra latency op.

Het doel van deze opdracht is het onderzoeken van verschillende technologiën waarmee deze bottlenecks verlicht kunnen worden, zodat een lagere latency of meer streams tegelijkertijd mogelijk worden. Namelijk de technologiën AMD SDI-Link en Nvidia GPUDirect. Er moet worden onderzocht of deze technologieën voordeel bieden boven de huidige software.

Hiervoor moeten 3 modellen worden ontwikkeld: één volgens de oude methode, één volgens SDI-Link en één volgens GPUDirect. Hierna moet worden getest welke van de 3 methoden het meest geschikt is en kan een afweging gemaakt worden welke methode het meest geschikt is.

De term meest geschikt kan niet eenduidig worden gedefinieerd, verschillende klanten kunnen verschillende wensen hebben over de hoeveelheid streams en latency. Wel geldt dat meer streams of een lagere latency altijd beter is. Aspecten als de prijs en benodigde devtijd spelen echter ook mee.

3

AANPAK

Er is gekozen voor een iteratieve faseringsmethode. Elke iteratie bestaat uit een planning, het uitzoeken van de requirements, implementeren (optioneel), testen en ten slotte evaluatie.

Hiervoor is gekozen om de volgende redenen: De producten die moeten worden onderzocht en opgeleverd staan grotendeels vast. Het is niet waarschijnlijk dat de requirements tijdens het project significant veranderen. Dat maakt RUP-achtige methodes, waar veel tijd wordt doorgebracht met het uitzoeken van de requirements en belanghebbenden, minder geschikt.

Een mogelijke optie is de watervalmethode. Een nadeel hiervan is dat het project lastig tussentijds veranders kan worden, dat kan gebeuren bij bijvoorbeeld tijdnood. Bij de iteratieve ontwikkelmethode is het mogelijk om, in geval van tijdnood, deelproducten op te leveren.

4

PLANNING

De globale planning is als volgt:

- Plan van aanpak 1 dag
- Onderzoeken technologieën 5 dagen
- Ontwikkelen
  - Oude methode 20 dagen
  - GPUDirect 15 dagen
  - SDI-Link 15 dagen
- Testen (performance) 3 dagen
- Adviseren 2 dagen
- Uitloop 9 dagen

Totaal: 70 dagen plus tijd voor het afstudeerverslag.

Het ontwikkelen zal de meeste tijd in beslag nemen. Er is iets meer tijd gereserveerd voor het ontwikkelen van het beginmodel dan voor de overige modellen, dat komt omdat er nog weinig ervaring is met de te gebruiken technieken, met name de communicatie met de hardware.

5

WAT IS SDI?

SDI staat voor Serial Digital Interface. Een digitale aansluiting voor videobeelden. De gebruikte connector is de 75-ohms BNC interface, die ook wordt toegepast voor analoge videobeelden.

Er bestaan verschillende standaarden, zoals HD-SDI (1.485 of 1.485/1.001 Gbit/s) en 3G-SDI (2.970 of 2.970/1.001 Gbit/s). Er worden verschillende framerates ondersteund, zoals 60 Hz en 24 Hz.

SDI werkt altijd op dezelfde, maximale, bitrate. Toch zijn er verschillende formaten en framerates mogelijk. Dit gebeurt door 'frames' over de lijn te verzenden. Een frame is bijvoorbeeld 2640*1125 pixels. Met een bitrate van 2.970Gb/s ontstaat er dan exact een framerate van 50 'raw' frames per seconde. De frames zijn gecodeerd met YUV 4:2:2, met 32 bits per 2 pixels.[1]
De videobeelden worden verpakt in de raw frames door simpelweg maar een gedeelte van het frame te gebruiken. De data die niet gebruikt wordt door beeld kan worden gebruikt om geluid mee te zenden. Om framerates te ondersteunen die niet exact een veelvoud van de bitrate is, wordt de bitrate gedeeld door 1.001. Zo kunnen ook de gebroken frequenties zoals 23.97hz worden ondersteunt.

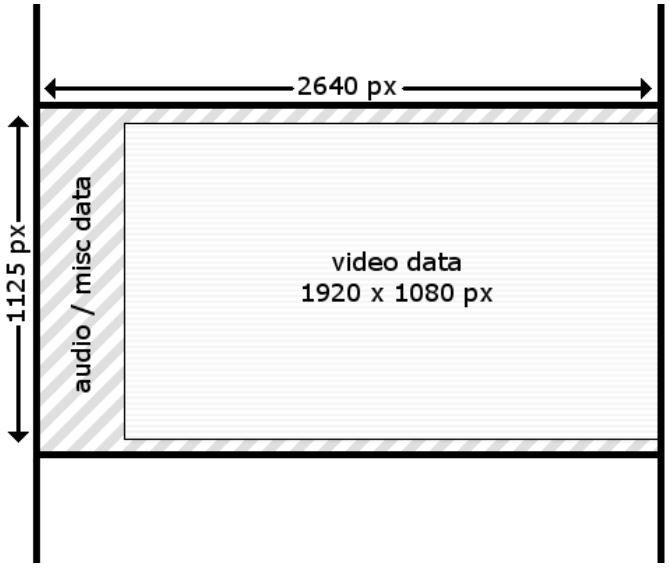

Figure 1: Een iliustatie van een raw frame. In deze iliustratie wordt een 1080p frame verzonden met 50 fps (2640 * 1125 * 20_{bpp} / 2.970Gb/s = 0.02 ms). De bytes wordt serieel (van links naar rechts, van boven naar beneden) verzonden. Lagere framerates zijn mogelijk door het raw frame hoger of breder te maken.

1 documentatie VideoMasterHD SDK

Figure 2: Een SDI connector.

http://en.wikipedia.org/wiki/Serial_digital_interface

6 AMD SDI-LINK

AMD SDI-Link is een technologie van AMD om data van een PCI-e device naar een ander PCI-e device over te zetten zonder dat deze via het systeemgeheugen gekopieerd moet worden. Hiervoor wordt gebruik gemaakt van de mogelijkheid om PCI devices met elkaar te laten communiceren over de PCI-e bus. Zogenaamde P2P (peer-to-peer) communicatie. Meestal praten PCI devices alleen met de host, namelijk de CPU. De datastroom loopt in dat geval altijd via het systeemgeheugen, wat mogelijk CPU load tot gevolg heeft. Daarnaast is de bandbreedte van het systeemgeheugen natuurlijk beperkt. Het snelste single-socket platform wat momenteel beschikbaar is (Sandy Bridge E) heeft een memory bandwidth van ongeveer 37GB/s onder ideale omstandigheden.

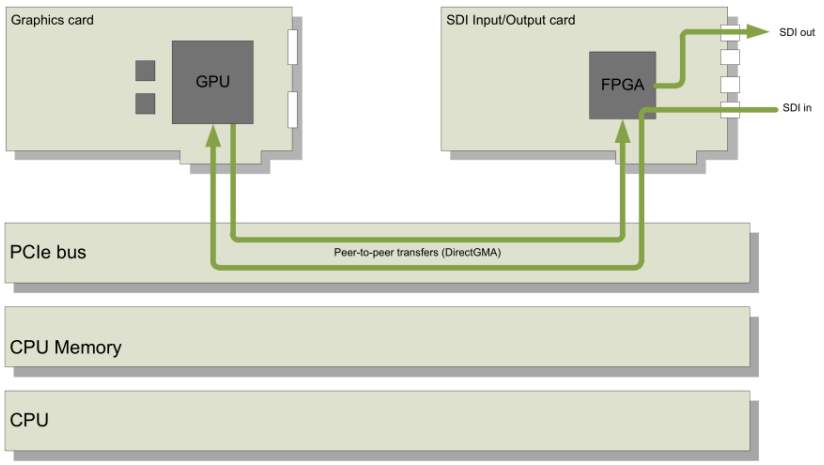

Figure 3: Een diagram het datapad van de SDI-beelden met AMD SDI-Link.

Er bestaat ook een andere oplossing voor het bandwidth probleem: er zijn fabrikanten die een PCI-e kaart uitbrengen met zowel een SDI interface als een GPU. Op deze manier hoeft de informatie niet meer te worden gekopiëerd over de PCI-e bus, en bestaat er ook geen probleem van te weinig bandwidth in het systeemgeheugen. Volgens AMD is deze oplossing echter minder flexibel, wanneer er een nieuwe standaard of snellere grafische chip komt moet je wachten totdat deze combinatie kaarten een update krijgen. [1] Ook zijn er vanzelfsprekend minder concurrenten dan in de losse SDI of GPU markt, waardoor de

1 http://www.amd.com/us/Documents/SDI-tech-brief.pdf

prijzen hoger zullen liggen.

Om AMD SDI-Link te laten werken zijn vier onderdelen benodigd:

- Een PCI-e SDI kaart en een PCI-e GPU aangesloten op dezelfde PCI-e bus.
- support vanuit de drivers van de grafische kaart. Niet alle videokaarten van AMD worden ondersteund.
- support vanuit de drivers van de PCI-e SDI kaart.
- support vanuit de software. De SDK van AMD levert ondersteuning voor OpenGL, DirectX en OpenCL.

Momenteel wordt alleen de AMD V7900 SDI officieel ondersteund door AMD SDI-Link.

Uit latere communicatie met DeltaCast en AMD blijkt dat er meerdere varianten van AMD SDI-Link bestaan. De fabrikant kan ervoor kiezen om de communicatie tussen de PCI-e devices onderling, of via het systeemgeheugen te laten lopen. De fabrikanten AJA, BlueFish en DataPath ondersteunen directe communicatie. Van deze fabrikanten is geen hardware voorhanden binnen VidiGo.

7 NVIDIA GPUDIRECT

Bij het zoeken naar GPUDirect op het internet komen er veel verschillende technieken tevoorschijn.

1. Transfers tussen een GPU en een andere GPU.
2. Transfers tussen een GPU en een andere GPU over een network device (ethernet, infiniband, enz. maar niet SDI). Het systeemgeheugen wordt hierbij niet gepasseerd. Deze feature is alleen beschikbaar onder RHEL (Red Hat Enterprise Linux).
3. Transfers tussen een GPU en een network device of storage device (ook SDI), waarbij de data wel langs het systeemgeheugen gaat, maar er geen buffer-copy's plaatsvinden.
4. Transfers tussen een GPU en een SDI kaart, waarbij het systeemgeheugen niet wordt gepasseerd. (P2P transfer)

Al deze technieken heten 'NVIDIA GPUDirect', dat maakt het zoeken naar informatie er niet makkelijker op. Uit deze opties is optie 4 natuurlijk het meest interessant. Opties 1 en 2 zijn niet van toepassing met een SDI kaart, terwijl directe P2P communicatie natuurlijk te verkiezen is boven een verbinding via het systeemgeheugen, zoals beschreven bij optie 3. Maar er zit een grote kanttekening aan vast, dit werkt namelijk alleen met een NVIDIA SDI-kaart. Deze zijn echter ongeveer 5 keer zo duur (ong. €10.000 p.st) als de hardware van third-party vendors. [1]

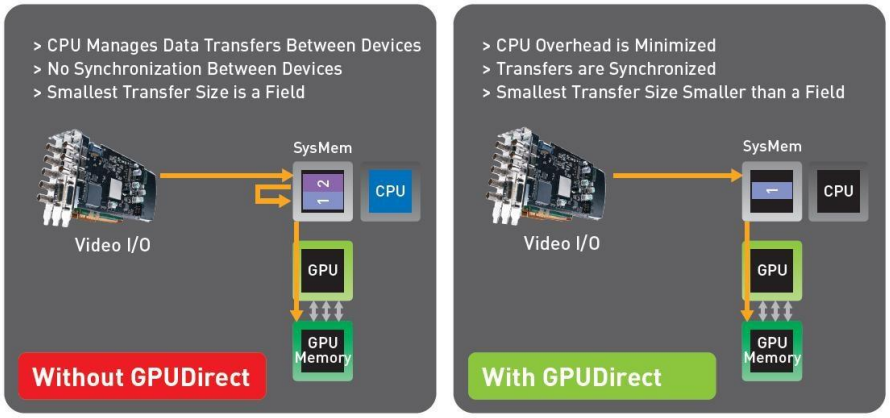

Figure 4: Het datapad bij gebruik van een third-party SDI kaart.[2]

1 Opmerking van Peter Ijkhout

2 http://www.deltacast.com/News/120131-Deltacast-Videomaster-HD-SDK-supports-low-latency-video-NVIDIA-GPUs.asp

Er zijn verschillende verklaringen te vinden waarom third-party SDI kaarten niet worden ondersteund. Volgens NVIDIA zijn de prestaties vergelijkbaar (in termen van latency), en de software veel is makkelijker te ontwikkelen[3]. De vraag is dan waarom NVIDIA wel hun eigen SDI kaarten ondersteund met deze propetaire technologie. Want er is -volgens hunzelf- nauwelijks voordeel.

Een nadeel van de techniek van NVIDIA, waarbij P2P transfers worden gebruikt, is dat het niet mogelijk is om meerdere input of output borden te combineren met één GPU. In situaties waarbij veel SDI in- of outputs vereist zijn is het dus noodzakelijk om meerdere NVIDIA GPU's en SDI kaarten aan te schaffen.

Op moment van schrijven worden de volgende GPU's officieel ondersteund door NVIDIA GPUDirect[4]:

- Quadro 4000
- Quadro 5000
- Quadro 6000
- Quadro FX 3800
- Quadro FX 4800
- Quadro FX 5800

Na de start van de opdracht zijn de kaarten K4000 en K5000 toegevoegd aan de 'supported' lijst. De hardware was toen echter al besteld.

3 http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0049-GTC2012-GPU-Direct-Video.pdf

4 http://www.nvidia.com//object//product_quadro_sdi_capture_us.html

8

ITERATIE 1: VOORBEREIDING

Dit gedeelte beschrijft de eerste iteratie, waarin een standaard test-model wordt ontwikkeld.

8.1 PLANNING

In deze iteratie wordt er een model ontwikkeld op basis van al bestaande software. In het model wordt een frame gelezen van een SDI PCI-e kaart, deze wordt naar de GPU verstuurd, waar vervolgens een simpele bewerking op wordt uitgevoerd. Het doel van dit model is om te leren hoe je met een SDI PCI-e kaart en GPU kan praten, en hoe daar simpele bewerkingen op kunnen worden uitgevoerd. Het model kan aan het einde van dit project getest worden om te kijken hoe deze performt in vergelijking met AMD SDI-Link en NVIDIA GPUdirect.

De planning van deze iteratie is als volgt:

- Planning 0.5 dag
- Uitzoeken requirements 0.5 dag
- implementatie 17 dagen
- testen 1 dag
- evaluatie 1 dag

De verwachting is dat deze iteratie ongeveer 20 dagen gaat duren.

8.2 REQUIREMENTS

Het doel is dat er een model wordt ontwikkeld op basis van de al bestaande software. Dit model moet aan de volgende eisen voldoen:

- Videobeelden kunnen wordt opgehaald vanaf de SDI kaart en naar de GPU gestuurd
- Videobeelden kunnen van de GPU naar de SDI kaart worden verstuurd.

Door het uitvoeren van een simpele bewerking, zoals het inverteren van het beeld, kan worden geobserveerd of de data echt langs de GPU is geweest.

8.3 IMPLEMENTATIE

De producten van VidiGo draaien op een zelf ontwikkeld framework genaamd DAVE. DAVE is gedeeltelijk gebouwd bovenop Qt. Qt is een cross-platform, open source, window management framework en biedt daarnaast grote hoeveelheid standaardklassen. Zoals string manipulatie, threading, graphics, html, enzovoorts. Door de abstractielaag kan Qt ooit worden vervangen door een ander window management framework. Uiteraard moet dan wel een groot deel van de code van DAVE zelf worden herschreven, de applicaties die gebouwd zijn op DAVE hoeven echter niet aangepast te worden.

VidiGo verkoopt diverse producten, zoals Visual Radio, Live en Works. Deze maken allemaal gebruik van het framework DAVE.

Een belangrijke klasse in het framework is `DaveObject`. Een `DaveObject` implementeert een uitgebreide interface. De functie van deze klasse is het leveren van een interface voor een stream. Een stream kan bijvoorbeeld zijn een stilstaand beeld, een bestand van een file, of een SDI input. Zowel in als ouput is mogelijk met een `DaveObject`.

`DaveObject` bevat onder andere de functies `useTexture()` en `screenDataDone()`. Met `useTexture()` wordt er een bitmap naar de GPU gestuurd, zodat deze kan worden bewerkt. Na het bewerken wordt `screenDataDone()` aangeroepen, zodat het `DaveObject` weer beschikt over de bewerkte texture. Het `DaveObject` kan deze texture indien gewenst vervolgens opslaan naar een file, of er nog meer bewerkingen op uitvoeren.

Tevens heeft elk `DaveObject` een grootte, positie en rotatie. Deze attributen worden door DAVE gebruikt om te bepalen op welke plek de texture gerenderd moet worden. Dat renderen gebeurt op de GPU met behulp van OpenGL, een cross-platform graphics API. Het is bijvoorbeeld mogelijk 2 `DaveObjects` als een video stream te gebruiken, deze met een aantal transformaties te renderen op een 3e `DaveObject`, welke de data vervolgens opslaat naar een file.

DAVE zorgt voor een efficiënte communicatie met de grafische kaart: textures worden in batches (een pool) geup- of download om de communicatie zo efficiënt mogelijk te laten verlopen. Vooral de downloads zijn erg traag. Dat komt omdat de standaard videokaarten zijn ontworpen om -met name- games op de draaien. Een game voert normaal gesproken alleen uploads uit. De Quadro 4000 bevat twee DMI-buffers zodat er tegelijkertijd data geüp- en download kan worden zonder dat de grafische pipeline hoeft te stallen. Bij de standaard videokaarten is dat niet mogelijk. Het up- of downloaden van losse textures daarom moet zo veel mogelijk worden vermeden.

Een andere belangrijke klasse is `DavePlugin`. Functionaliteit wordt bij DAVE vaak toegevoegd met behulp van plugins. Een plugin is eigenlijk heel simpel: er bestaat een getter `providesObect()` waarmee

kan worden gecontroleerd of deze plugin een object kan aanleveren. Ook bestaat er een functie `createObject()`, welke een `DaveObject` terug geeft. In feite is een `DavePlugin` niets meer dan een factory. Wanneer er een plugin als externe DLL wordt gebouwd wordt er één methode geëxporteerd waarmee de plugin klasse kan worden opgehaald.

Als oefening is er een simpele plugin gemaakt genaamd `StefanPlugin`. Deze plugin bestaat uit 2 klassen: `StefanPlugin` en `StefanObject`, waar `StefanPlugin` een afgeleide is van `DavePlugin` en `StefanObject` een afgeleide van `DaveObject`. In `StefanPlugin` worden de 2 methoden geimplementeerd die nodig zijn om de factory compleet te maken. `StefanObject` implementeert slechts een functie: `update()`. Deze functie wordt periodiek aangeroepen door DAVE. Binnen deze functie wordt een simpele bitmap aangemaakt en gevuld met een patroontje, dit patroontje wordt in dit voorbeeld door de CPU gegenereerd. Vervolgens wordt de functie `useTexture` aangeroepen. DAVE zorgt er dan voor dat de texture op de juiste plaats op het scherm verschijnt.

Er moet nu een plugin worden gemaakt die beelden, ophaalt van een SDI kaart. Na het ophalen van de frame moet de functie useTexture worden aangeroepen, zodat de texture vervolgens wordt bewerkt op de GPU. De moeilijkheid zit hem natuurlijk in de communicatie met de SDK van de fabrikant van de SDI kaart. Voor deze opdracht wordt er gebruik gemaakt van SDI kaarten van de fabrikant DeltaCast. Detacast is een Belgisch bedrijf wat SDI en DVI capture cards maakt. De SDK van deze fabrikant heeft de naam VideoMasterHD SDK. Hiermee kunnen tot 4 SDI kaarten per systeem worden aangestuurd.

Gelukkig bestaat er al een mooie plugin voor de communicatie met deze SDK. De `DaveDeltaCastPlugin` start een nieuwe thread op. Als de plugin als input is ingesteld wordt de SDK regelmatig gepold om te controleren of ere nieuwe frames zijn gearriveerd. Daarna wordt deze geconverteerd naar een DaveBitmap, vervolgens wordt `useTexture()` gebruikt. Een diagram wat de flow beschrijft is hieronder weergeven:

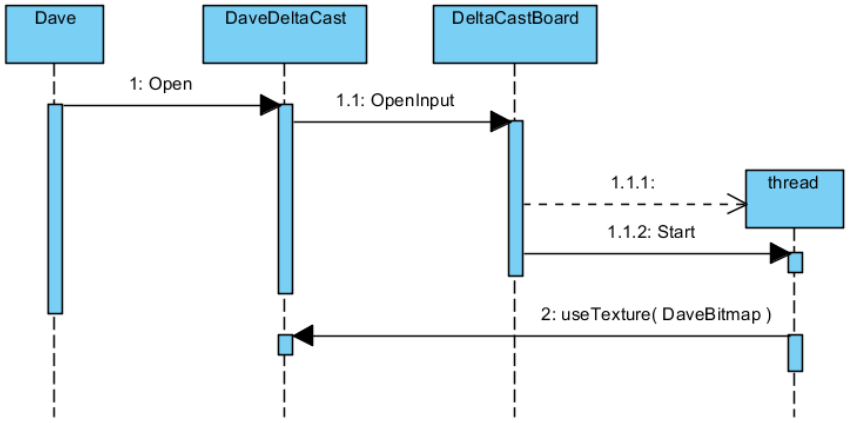

Figure 5: Een sequentiediagram van de input DeltaCastPlugin.

De output is iets lastiger. Doorgegeven frames worden op een stack gezet, de thread controleert regelmatig of er nieuwe frames op de stack staan. Indien dat het geval is wordt de frame doorgegeven aan de VideoMaster HD SDK.

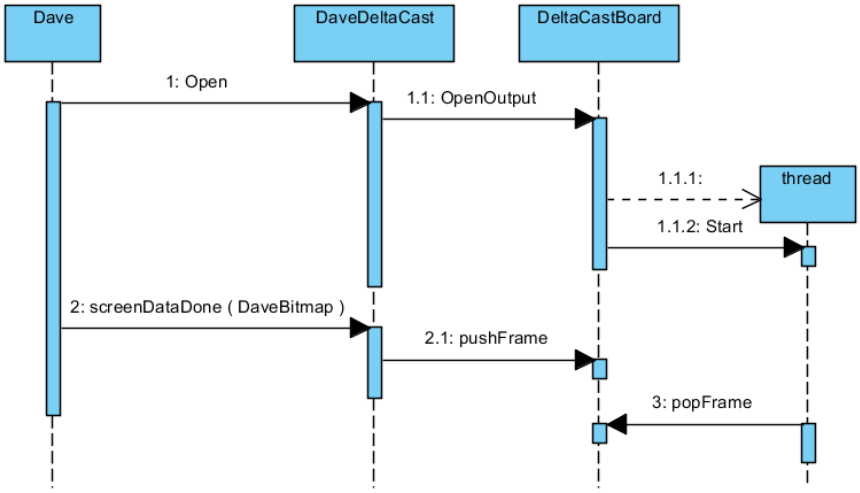

Figure 6: Een sequentiediagram van de output DeltaCastPlugin.

Het doel van deze iteratie was het maken van een model of stuk software waarmee met de SDI kaart gecommuniceerd kan worden, en beeld kan worden getoond op een SDI output. Deze code blijkt echter al te in simpele vorm te bestaan. Het is niet zinvol om het wiel opnieuw uit te vinden door een eigen implementatie te creëren. Daarom is er besloten deze iteratie af te ronden en te beginnen aan de implementatie van NVIDIA GPUDirect.

8.4 TESTEN

Deze iteratie is er geen eigen software ontwikkeld. Er valt om die reden ook weinig te testen. De besproken plugin wordt ook in productie gebrukt, er wordt daarom van uit gegaan dat deze stabiel is.

8.5 EVALUATIE

Het doel van deze iteratie was het ontwikkelen van een model waarmee beelden, binnengekomen vanuit een SDI kaart, worden doorgestuurd naar de GPU. En deze beelden vervolgens weer te tonen op een SDI output. Dat is niet compleet volgens het plan gegaan. De benodigde code bleek namelijk al te bestaan, met behulp van een relatief simpele plugin kunnen er beelden worden ingelezen vanuit de SDI kaart. DAVE, een in-house ontwikkeld framework zorgt er vervolgens voor dat deze beelden worden bewerkt op de GPU. Met behulp van dezelfde plugin kunnen de bewerkte beelden worden uitgezonden over SDI.

Deze iteratie is daarom afgebroken: het is simpelweg niet zinvol om zelf een plugin te ontwikkelen die precies dezelfde functionaliteit heeft als iets wat al lang bestaat. Deze beslissing betekent dat er meer tijd beschikbaar is voor de overige iteraties. Het grootste gedeelte van deze tijd kan worden besteed aan de implementatie van NVIDIA GPUDirect. Er is tijdens deze iteratie namelijk weinig tijd besteed aan het opdoen van ervaring over de communicatie met een SDK van de SDI kaart en communicatie met de GPU.

9

ITERATIE 2: IMPLEMENTATIE

Dit gedeelte bescrijft de tweede iteratie. In deze iteratie wordt een model ontwikkeld voor AMD SDI-Link.

9.1 PLANNING

In deze iteratie wordt er een model ontwikkeld op basis van AMD SDI-Link. Dit wijkt af van de originele planning. De reden hiervoor is dat NVIDIA GPUDirect anders werkt dan oorspronkelijk gedacht. In tegenstelling tot wat toe nu toe werd gedacht is het niet mogelijk om met een third-party kaart P2P transfers uit te voeren. Dat werkt alleen met een NVIDIA SDI kaart.

Bij de oplossing van NVIDIA wordt er 'slechts' een buffer copy bespaard, terwijl bij AMD de frames direct over de PCI-e bus gaan. Alhoewel er binnen VidiGo hoofdzakelijk gebruik wordt gemaakt van NVIDIA hardware wordt de oplossing van AMD kansrijker geacht. Daarom is er besloten AMD SDI-Link vóór NVIDIA GPUDirect te ontwikkelen.

De planning van deze iteratie is als volgt:

- Planning 0.5 dag
- Uitzoeken requirements 0.5 dag
- implementatie 17 dagen
- testen 1 dag
- evaluatie 1 dag

Deze planning is exact hetzelfde als die van de vorige iteratie. Dat is een bewuste keuze

NB: pas na de start van deze iteratie werdt bekend dat de DeltaCast implementatie geen P2P transfers ondersteunt.

9.2 REQUIREMENTS

er moet een model worden ontwikkeld waarmee, met behulp van AMD SDI-Link, beelden kunnen worden verplaatst van de SDI kaart naar de GPU en weer terug. Zonder dat de beelden via het systeemgeheugen lopen. Hieruit volgen dus de volgende eisen:

- De beelden van de SDI kaart worden doorgestuurd naar de GPU, zonder langs het systeemgeheugen te gaan.
- De GPU voert een bewerking uit.
- De beelden worden weer teruggestuurd naar de SDI kaart, wederom zonder dat de beelden via het systeemgeheugen gaan.

9.3 IMPLEMENTATIE

Deltacast levert een extensie op hun SDK waarmee AMD SDI-Link en NVIDIA GPUDirect kunnen worden gebruikt. Deze heeft de orginele naam GPU extension meegekreken en bestaat uit een aantal in C geschreven header files en een aantal .lib en .dll bestanden. Bij de GPU exension wordt documentatie meegeleverd over de werking van alle functies in deze exensie. tevens worden er samples meegeleverd in OpenGL, D3D09 (DirectX) en D3D11. Deze samples zijn geleverd in de vorm van een Visual Studio project file.

De sample waarmee zowel kan worden verzonden als ontvangen lijkt het meest interessant. Deze benodigt wel de nodige aanpassingen voordat hij gebruikt kan worden. De sample gaat er vanuit dat het laagstgenummerde bord zowel een in als output bevat. Het testsysteem bevat echter losse in- en output kaarten. De code moet hiertoe worden aangepast zodat er een apart in en output bord gebruikt kan worden.

Als het programma start worden er 2 streams aangemaakt, een stream is een abstractie van een in of output binnen de VideomasterHD SDK. Zodra de stream is gestart is het niet meer nodig om te weten op welk board deze zit. In het kort werkt het openen van een stream als volgt:

```
HANDLE board_Handle;
ULong board_ID = 0;     // open het laagstgenummerde bord.
ULong result;

/* open een board. board_Id is een integer tussen 0 en de
   hoeveelheid geinstalleerde hardware van DeltaCast.
   Argumenten 3 en 4 zijn gereserveerd */
result = VHD_OpenBoardHandle(board_ID, &board_Handle, NULL, 0);

if (result != VHDERR_NOERR)
    // fout

/* open de stream 'RX0' (receive 0) op het bord
   aangegeven met board_Handle */
result = VDH_OpenStreamHandle(board_Handle, VHD_ST_RX0,
    VHD_SDI_STPROC_DISJOINED_VIDEO, NULL, &inStream, NULL);

if (result != VHDERR_NOERR)
    // fout

// de stream is succesvol geopend.
```

Uiteraard is dit een versimpeld scenario: in de praktijk is het verstandig om te kijken of er wel een bord aanwezig is voordat je deze probeert te openen. Daarnaast is het ook verstandig om pas een stream te openen als je zeker weet dat de het bord beschikt over dit type stream, zodat je de gebruiker een goede foutmelding kan geven. Maar

dat zijn details die niet noodzakelijk zijn voor de werking van een sample, bij een foutmelding wordt het programma domweg afgesloten. Als er eenmaal een stream is geopend kan de stream worden gestart, gepauzeerd, afgesloten, en kan er informatie over de stream worden verzameld binnen verschillende functies in deze SDK.

De GPU extension voegt slechts een klein aantal functies toe aan de SDK. Voor elke ondersteunde graphics API (Dx9, Dx11 en OpenGL) bestaat er één functie om de extensie te initialiseren (bijvoorbeeld `VHD_GpuInitializeOpenGL()`) en één functie om de extensie te deïnitialiseren. Een vergelijkbare verzameling functies is beschikbaar om een OpenGL of DirectX texture kenbaar te maken aan de GPU extensie, en weer te verwijderen.

Minder vanzelfsprekend zijn de functies waarmee data kan worden opgehaald en verzonden. Namelijk `VHD_GpuCopyFromSlot` en `VHD_GpuCopyToSlot`. Slot is een term voor één frame aan data, dat kan zowel interlaced als progressive zijn. De documentatie spreekt over het initiëren van een DMA (Direct Memory Access) transfer. Hoe dat precies werkt volgt niet duidelijk uit de documentatie.

Ten slotte is het belangrijk om te weten dat alle functies uit deze extensie aangeroepen moeten worden vanuit dezelfde thread als waarin de OpenGL context is gemaakt. Dit levert zeer waarschijnlijk problemen op met het gebruik in DAVE. DAVE gebruikt slechts één OpenGL context. De synchronisatie met de SDI kaart wordt momenteel gedaan door middel van een blocking wait in een aparte thread. Het is waarschijnlijk dat dit later voor ernstige latency's gaat zorgen zodra er meerdere streams tegelijk worden afgespeeld zonder gebruik van meerdere threads.

Tijdens deze iteratie is er ook contact opgenomen met AMD, met her verzoek om meer informatie en eventuele code samples. Volgens de contactpersoon bij AMD maakt DeltaCast geen gebruik van P2P transfers, maar wordt er slechts pinned memory gebruikt zodat er geen buffer copy's plaatsvinden wanneer de data wordt verplaatst van de SDI driver naar OpenGL. De fabrikanten JAI, Bluefish en DataPath ondersteunen volgens AMD wel P2P transfers. In principe heeft SDI-Link bij DeltaCast dus exact dezelfde functionaliteit als NVIDIA GPUDirect. Het is erg jammer dat dit soort informatie niet gewoon te vinden is op de website van AMD of andere online te breiken resources.

Het implementen van de GPU extensie in DAVE levert in in de praktijk nogal wat problemen op. Ten eerste vereist de extensie dat alle functies worden aangeroepen vanuit de thread waarin de OpenGL context wordt gedraaid. In een realtime applicatie is dat lastig, de op-

eraties mogen niet te lang duren, anders blokkeert de UI thread. Dat betekent dat er waarschijnlijk een oplossing verzonnen moet worden met meerdere threads. Hier zijn enkele opties voor:

1. De context wordt tussen threads verplaatst met behulp van wglMakeCurrent() (windows), aglSetCurrentContext() (OS X) of glXMakeCurrent() (X11). Slechts een thread kan tegelijkertijd 'current' zijn. Dat vereist enkele aanpassingen aan DAVE, DAVE maakt momenteel de OpenGL en OpenCL context 'current' voor de meeste operaties die hij doet, maar de context wordt nooit weer 'niet-current' gemaakt.

2. Er wordt een 2e openGL context aangemaakt. Met behulp van `wglShareLists()` worden textures tussen de verschillende threads (maar geen processen) gedeeld. Volgens Arjan is de implementatie hiervan platform specifiek en bestaat er een kans dat het via het systeemgeheugen gaat.

Beide opties zijn niet erg aantrekkelijk. Bij de eerste optie moet DAVE worden aangepast zodat de context altijd wordt gedeactiveerd na een OpenGL actie. De OpenGL code moet bovendien worden beschermd met een mutex. Zodat er slechts een thread tegelijkertijd de OpenGL context kan bereiken.

Bij de 2e optie worden de textures gedeeld tussen threads. Doordat beide threads een eigen OpenGL context hebben hoeft DAVE niet te worden aangepast. Een nadeel is dat wglShareLists alleen functioneel is voordat er textures worden aangemaakt of andere OpenGL taken worden uitgevoerd. Aangezien de OpenGL context al vroeg in het programma wordt aangemaakt door het window management van Qt kan dat een flinke opgave worden. Alhoewel deze opdracht zich richt op het maken van een proof of concept is het aanpassen van de libraries van Qt om een functionaliteit in een veel lagere laag te bewerkstelligen ontwerptechnisch natuurlijk geen nette oplossing. Ten slotte is het van belang dat alle openGL contexts worden aangemaakt vanuit dezelfde thread, maar dat is geen probleem: DAVE zelf gebruikt geen threads.

De tweede oplossing lijkt het meest kansrijk op de korte termijn. Het beschermen van de OpenGL context met een mutex en ervoor zorgen dat de context netjes word geunload betekent het aanpassen van een flinke hoeveelheid source files. Er bestaat een kans dat het uitwisselen van textures relatief makkelijk gaat met wglShareLists(), daarom wordt er als eerste gekeken naar deze oplossing. Mocht deze oplossing lastig of onmogelijk blijken, dan kan er worden uitgeweken naar de eerste oplossing.

Er wordt een extra klasse gemaakt die de communicatie met de GPU extensie verzorgt, eventuele variabelen worden opgeslagen in deze klasse.

Eerst wordt de output gemaakt. Hiervoor is gekozen omdat de output gemakkelijker te realiseren dan de input. Bij de output hoeft er alleen een texture ID te worden doorgegeven aan de SDK, DAVE zorgt er zelf voor dat er een geldig texture ID bij een DaveObject hoort. Bij input moet de OpenGL texture zelf worden aangemaakt met behulp van OpenGL.

Het verzenden van de texture naar een SDI kaart met de SDK werkt echter niet. Bij het aanroepen van `VHD_GpuCopyToSlot()` wordt er VHDERR_NOERROR teruggeven, wat de code is voor succes. Er verschijnt echter geen beeld op het scherm wat met SDI is aangesloten. Daarna wordt de NVIDIA kaart verwisseld met een van AMD. Nu komt er wel een error code tevoorschijn: VHDERR_OPERATIONFAILED. Volgens de documentatie betekent dat dat er een unkown error is opgetreden.

Om het probleem om te lossen is er contact opgenomen met DeltaCast. Deze herkenden de problemen niet, maar boden aan een minimum working sample te bekijken als ik deze zou maken. Tijdens het ontwikkelen van deze sample is het probleem zelf gevonden: `VHD_GpuCopyToSlot()` faalt als de grootte van de texture niet exact de helft is als die van de output stream. De helft van de grootte is nodig omdat OpenGL de textures opslaat in RGBA formaat, terwijl SDI gebruik maakt van YUV 4:2:2 waarbij er 2 pixels worden gecodeerd per 32 bits. Op zich is dat logisch, maar dat dit niet juist is gedocumenteerd en de SDK afhankelijk van de fabrikant van de videokaart 'unknown error' teruggeeft of simpelweg faalt, is natuurlijk erg vervelend.

DeltaCast is ook op de hoogte gebracht van dit probleem. Ze schrijven dat het een fout is in de Nvidia dan wel AMD specifieke library, daarop heeft DeltaCast een fix uitgebracht zodat er een zinvolle error code terug wordt gegeven als het formaat van de texture incorrect is.

Na deze aanpassing wou de aangepaste DavePlugin echter nog steeds niet werken. Daarom is er besloten de aanpassingen weg te gooien en opnieuw te beginnen vanaf de zojuist gemaakte minimum working sample. Kort daarna werkte de SDI output wel naar behoren in DAVE, en verscheen ons patroon op het SDI scherm.

Het is nu de bedoeling om ervoor te zorgen dat er concrete beelden worden uitgestuurd over SDI. Dit vereist enkele omwegen: in DAVE wordt er een screen aangemaakt wanneer je beelden beschikbaar wilt maken aan andere plugins. Een screen bestaat uit een OpenGL texture en een formaat (bijvoorbeeld 1080p50). De texture is, zoals de OpenGL standaard voorschrijft, veelvouden van machten van 2 groot. Het is niet mogelijk om in OpenGL een texture te gebruiken met di-

mensies welke geen machten van 2 zijn. Er bestaan wel extensies om dat mogelijk te maken. Maar: dat zijn extensies en behoren niet tot de core OpenGL specificatie. Theoretisch kan het voorkomen dat er een grafische kaart wordt gebruikt die deze feature mist. Als er binnen DAVE een texture wordt aangemaakt, wordt dit ook altijd een macht van 2.

Samenvattend is een texture van een screen altijd een macht van 2 groot, terwijl de GPU extensie op de VideoMasterHD SDk alleen functioneert als de texture exact even groot is als de output (nooit een macht van 2). Hier zal dus een conversiestap voor moeten worden ontworpen. Hier kan dan meteen de RGB naar YUV conversie worden uitgevoerd.

DAVE biedt een functie om een texture naar een texture van een andere groote en formaat om te zetten. In feite wordt hierbij een texture opgezet als framebuffer, naar dit buffer wordt een quad gerenderd. Het is in openGL niet mogelijk om 'simpel' een stuk geheugen te kopiëren.

Deze functie heet `convertTextureToFrameBuffer()`. Het converteren van formaten is erg omslachtig binnen OpenGL. De volgende stappen moeten daarvoor worden uitgevoerd:

- De destination texture wordt als framebuffer ingesteld. (`glBindFramebuffer()`)
- Het viewpoint wordt goed ingesteld. Het viewpoint is gelijk aan het formaat van de texture (vaak 1920x1080).
- De camera wordt ingesteld. In dit geval wordt de projectiematrix ingesteld met een orthogornaal perspectief.
- De brontexture wordt aan de OpenGL context verbonden (`glBindTexture()`)
- de juiste conversieshader wordt geladen. De fragment shader maakt de vertaalslag naar verschillende pixelformaten, in dit geval RGB naar YUV 4:2:2.
- Er wordt een quad gerenderd.

Bij het renderen van een quad (eigenlijk 2 triangles) voert het OpenGL device voor elke pixel de shader uit. Zo verschijnt er uiteindelijk een texture in het gewenste formaat.

Deze functie werkt in deze use case echter niet naar behoren: de kleur rood verandert in de kleur blauw, en omgekeerd. Dit is gemakkelijk te verklaren door het mixen van RGB en BGR data binnen het programma, de kleur groen wordt wel correct weergeven. Bovendien toont alleen de bovenste helft van het SDI scherm beeld: de onderste helft is leeg.

Na enig onderzoek wordt vermoed dat de shader de schuldige is: deze is oorspronkelijk ontworpen voor de oude deltacast drivers zonder de GPU extensie. Die extensie vereise dat de frames in 2 delen werden doorgegeven, zogenaamde fields. De shader zorgt er dus voor dat de texture in 2 delen gesplitst wordt. Na enig onderzoek kan de oorzaak voor het tonen van de helft van het scherm niet worden gevonden, daarom wordt er een simpele shader gemaakt om uit te sluiten dat het probleem daar ligt.

Shaders worden geschreven in GLSL (OpenGL Shader Language). Dit is een variant op C, de manier waarop de shader aan het OpenGL device wordt doorgegeven is redelijk ingenieus: de source code wordt in het programma verpakt, en wordt runtime gecompileerd door de device driver. In dit geval bestaat de device driver uit de driver van de AMD of NVIDIA GPU. Dit is zo gedaan omdat er grote verschillen kunnen zitten in de eigenschappen van de verschillende hardware, door de shader door de driver te laten compileren is het gegarandeerd dat de gecompileerde OpenGL code altijd optimaal is voor de gebruikte hardware.

Er bestaan -in grove lijnen- 3 typen shaders: vertex shaders, geometry shaders en fragment shaders. De vertex shader en geometry shader hebben betrekking op transformaties en manipulaties van 3D objecten, deze zijn voor deze opdracht niet van belang. De fragment shader wordt gebruikt voor het bepalen welke kleur een pixel krijgt. De shader voert een berekening uit op basis van enkele inputs, zoals een 2D texture, normal en/of positie van eventuele lampen. Deze inputs worden overigens door de vertex of geometry shader berekend.

Gelukkig is is de conversie van RGB naar YUV redelijk triviaal: er bestaat een YUV naar RGB conversiematrix. Om de juiste kleur te verkrijgen moet een vector bestaande uit de 3 kleuren worden vermenigvuldigd met deze matrix.

$$\begin{bmatrix} Y' \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.14713 & -0.28886 & 0.436 \\ 0.615 & -0.51499 & -0.10001 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{1}$$

De conversie van YUV naar RGB kan plaatsvinden met de inverse van deze matrix. De exacte getallen in de matrix zijn afhankelijk van het type YUV. Zo bestaat er Ycbcr en YPbPr, YUV en Y'UV, waarbij er weer andere waarden gehanteerd worden afhankelijk of er SD of HD beelden worden gebruikt. Ten slotte wil een fabrikant ook nog wel eens afwijkende getallen in hun hardware gebruiken.

Een GLSL shader lijkt op een simpel C programma. Op het moment dat de shader 'start' wordt de methode `main()` uitgevoerd. Je hebt zelf geen controle over wanneer elke shader precies wordt gestart, daar

zorgt de OpenGL engine voor. OpenGL zorgt ervoor dat de shader voor elke pixel/vertex/... parallel wordt utgevoerd.

In een shader zitten een aantal variabelen, deze worden met 2 verschillende keywords aangeduld: uniform en varying. Uniform betekent dat de waarde voor elke shader instance hetzelfde is, varying betekent dat de waarde per shader instance verschilt. Uniform wordt gebruikt voor het doorgeven van de omgeving, zoals de positie van lichten of de inhoud van een texture. Varying wordt meestal gebruikt om input van een andere shader te accepteren. Zo kan het zijn dat je in de vertex shader een normal van een object laat berekenen, vervolgens kan deze in een varying variabele worden gezet en kan de fragment shader de uiteindelijke kleur berekenen aan de hand van deze normal.

In tegenstelling tot een C programma heeft een shader geen concrete return value. De output wordt simpelweg naar een aantal voorgedefiniëerde variabelen geschreven. Een simpele fragment shader is hieronder weergeven:

```
#verson 120
varying imageSize; // enkele variabelen.
varying textureCoordinates;
uniform textureSize;

main()
{
    gl_FragColor = vec4( 1, 0, 0, 1 ); // maak de pixel rood.
}
```

Na het zelf maken van een shader simpele 'pass through' shader, een shader die alleen maar de kleur doorgeeft en verder geen berekeningen uitvoert, blijft het probleem echter bestaan. De oorzaak moet dus ergens anders zitten. Het blijkt dat de texture die op het scherm verschijnt niet exact de helft van het scherm is, maar net iets meer. Om precies te zijn 1080 / 1920 = 56% van het scherm. Het opsplitsen in aparte fields kan dus niet de oorzaak zijn.

Uiteindelijk blijkt dat je naast het formaat van de texture, ook het formaat van de OpenGL texture moet doorgeven aan (convertFormat()) functie, deze functie wordt gebruikt voor de RGB naar YUV conversie. Als dat niet gebeurt en je geeft de groote van het formaat (1920x1080) aan, dan worden de texture X en Y coordinaten berekend met:

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} \frac{1920}{1920} \\ \frac{1080}{1080} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{2}$$

OpenGL gebruikt een coordinatenstelsel waarbij de coordinaten niet in absoluten worden weergeven, maar in een range van 1 tot 0. In dit

geval wordt dus de complete texture van 2048x2048 op het scherm gerenderd. De correcte berekening is

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} \frac{1920}{2048} \\ \frac{1080}{2048} \end{bmatrix} \approx \begin{bmatrix} 0.94 \\ 0.53 \end{bmatrix} \tag{3}$$

Het probleem van de kleur rood en blauw blijft echter bestaan. Omdat er geen aanknopingspunten gevonden kunnen worden wordt besloten om verder te gaan met de SDI input. Mogelijk wordt er tijdens het maken van de SDI input nog iets gevonden wat tot de oplossing kan leiden tot de verwisseling van rood en blauw.

9.3.1 *input*

Nu de knelpunten in het werkend krijgen van de GPU extensie bekend zijn is het maken van werkende code voor de SDI input niet moeilijk. De bestaande initialisatiecode kan worden hergebruikt met slechts enkele aanpsssingen ten opzichte van de SDI output:

- Er moet een input port worden geopend in plaats van een output poort.
- De texture moet handmatig worden aangemaakt in plaats van op DAVE te vertrouwen dat het object een texture heeft.
- In plaats van `VHD_GpuCopyToSlot()` zal `VHD_GpuCopyFromSlot()` moeten worden aangeroepen om data over te zetten van de SDI kaart naar de GPU.
- De texture moet geconverteerd worden van V208 naar BGRA, in plaats van omgekeerd.

Verassend genoeg werkte de input vrijwel direct. Na de moeite die nodig was om tot een werkende SDI output (waarvan de kleuren niet eens correct zijn) te komen, is dat een grote meevaller. Overigens zijn de kleuren ook hier verwisseld.

Alhoewel de SDI input nu 'werkt' is de robuustheid en kwaliteit van de code laag. Wanneer er een ander formaat stream wordt aangeboden dan het hardcoded formaat werkt de in- of output niet. Ook wil de plugin na een aantal maal op play gedrukt te hebben regelmatig crashen. Verder is het niet mogelijk om meerdere plugins tegelijkertijd te draaien, de stream stopt dan zodra de 2e plugin start.

De oplossing om meerdere streams naast elkaar te kunnen draaien blijkt simpel: alle plugins draaien momenteel in dezelfde thread, en

gebruiken dezelfde functies uit de GPU extensie. Dat gaat fout, de extensie geeft een error als deze meerdere malen wordt geinitialiseerd vanuit dezelfde thread. Schijnbaar bevat de extensie geen mooie foutafhandeling, na de 2e maal initialiseren stoppen de aanwezige streams, en willen deze niet meer opstarten. Dit wordt snel opgelost door een static boolean aan te maken met informatie of de plugin geinitialiseerd is of niet. Het is zinloos om hier nu een mooie oplossing voor te bedenken, terwijl deze binnenkort weer vervangen moet worden indien de plugin wordt uitgebreid met threads.

Het auto-detecten van het formaat is ook relatief triviaal: de orginele DeltacastPlugin bevatte hier al code voor. Bij het veranderen van het formaat moet de groote van de texture opnieuw worden ingesteld. Het is noodzakelijk dat dit goed gebeurt: als de groote van de texture ook maar 1 pixel afwijkt weigert de VideoMasterHD SDK dienst.

Na deze aanpassingen kunnen er meerdere streams tegelijkertijd gedraaid worden met verschillende formaten. Dan komen de nadelen van singlethreaded werken naar boven: de plugin blockt totdat er een nieuwe frame binnen komt. In de praktijk houd dat in dat de CPU usage welliswaar heel laag is, maar er ook een flink aantal frames gedropt worden. De CPU usage is overigens geen ideale manier om de efficientie te meten.

De VideoMasterHD SDK bied mogelijkheden om het aantal gedropte frames exact te meten. Ongeveer elke 7e frame wordt gedropt met een testopstelling bestaande uit 3 streams: PAL 50 input, 1080i50 input, en 1080p25 output. Vermoeddelijk werkt de constructie wel goed als er streams worden gebruikt waarvan de framerate overeenkomt.

Het is nu dus nodig om de plugin zo om te bouwen dat er per stream een thread wordt gestart. Aangezien er door DeltaCast wordt aangegeven dat muti-thread en multi-context omgevingen niet worden ondersteund kan dat nog best een uitdaging worden.

9.3.2 *meerdere streams*

Zoals eerder vermeld zijn er verschillende manieren om de blocking wait te omzeilen:

1. De context wordt tussen threads verplaatst met behulp van `wglMakeCurrent()` (windows), `aglSetCurrentContext()` (OS X) of `glXMakeCurrent()` (X11). Slechts een thread kan tegelijkertijd 'current' zijn. Dat vereist enkele aanpassingen aan DAVE, DAVE maakt momenteel de OpenGL en OpenCL context 'current' voor elke call die hij doet, maar de context wordt nooit weer 'niet-current' gemaakt.

2. Er wordt een 2e openGL context aangemaakt. Met behulp van `wglShareLists()` worden textures tussen de verschillende threads (maar geen processen) gedeeld. Volgens Arjan is de implementatie hiervan platform specifiek en bestaat er een kans dat het via het systeemgeheugen gaat.

Geen van de oplossingen is ideaal. Er bestaat echter nog een 3e oplossing: niet-mulithreaded werken. Momenteel gebruikt DAVE van zichzelf geen threads, wat bewijst dat het mogelijk is om een flink aantal streams tegelijk te bewerken op slechts 1 core. CPU intensieve taken worden door de pluging wel verplaatst naar een aparte thread. Bij de oude versie van de DeltaCast plugin gebeurt dat als volgt (input):

- De video frames worden van de SDI kaart naar het host memory verplaatst. (DMI, In thread DeltaCast driver)
- De video frames worden van het geheugen van de driver naar geheugen van DAVE verplaatst. (In aparte thread)
- De video frames worden doorgegeven aan OpenGL, de werking hiervan is niet gespecificeerd, behalve dat het gegarrandeerd is dat, zodra glTexImage2D terugkeert, de inhoud van de texture veilig kan worden vrijgegeven. (in de main thread)

Omdat multithreading niet wordt ondersteund vanuit DeltaCast en er vaak wordt afgeraden OpenGL vanuit meerdere threads aan te spreken, wordt er eerst gekeken naar een andere oplossing. Het probleem is dat de locken van de slot de thread blockt. Maar het blijkt mogelijk om de timeout daarvan de veranderen. Als de timeout nu naar nul wordt gezet, zou dat dan invloed hebben op de hoeveelheid gedropte slots?

Na een test blijkt inderdaad dat dat een positieve invloed heeft: pas bij 3 input en 3 output streams treden er dropped frames op. Nader onderzoek laat zien dat de bottleneck compleet is verschoven naar de GPU, met behulp van gDEBugger kunnen de OpenGL draw commands worden uitgeschakeld. Als dat gebeurt kunnen er 5 input streams en 4 output streams op 1080i50 worden gedraaid zonder problemen. Meer kunnen niet worden gedraaid met het testsysteem, deze bevat 'slechts' 4 SDI outputs.

Deze oplossing werkt dus voldoende, het is niet nodig om multithreaded te gaan werken. De plugin start overigens wel een aparte thread op om periodiek te controleren of er frames zijn gedropt.

De plugin werkt nu 'goed', en zou in principe al getest op performance getest kunnen worden. Het testen zal waarschijnlijk gebeuren op de testbak bij VidiGo. In deze testbak zijn voldoende SDI in en output aanwezig om de oude DaveDeltaCast plugin op zijn knieen te krijgen. Voordat er wordt begonnen met testen is het echter verstandig

om de plugin helemaal stabiel te krijgen. Met helemaal wordt in deze zin niet bedoeld dat er geen fouten meer in zitten, maar dat de plugin met de meest voorkomende situaties kan omgaan. Het is natuurlijk erg vervelend als de software waarvan de performance getest moet worden, vaak crasht.

Een belangrijk punt is dat het het formaat van de stream tijdens het afspelen in principe kan veranderen. Dat moet worden gedeteceerd, het fromaat van de texture moet anders worden ingesteld, de SDK moet juist worden ingesteld, enzovoorts. Hiervoor is wat al wat code aanwezig in de oude DeltaCast plugin. Dit stukje code detecteert of het formaat is veranderd, als dat zo is, dan wordt de functie `stop()` aangeroepen, het formaat veranderd, en daarna weer gestart met behulp van de `play()` functie. De GPU extensie gedraagt zich echter raar onder deze omstandigheden. Wanneer de stream, na het veranderen van he format, weer aan de GPU extensie gelink wordt wordt er doodleuk een error teruggegeven dat de stream al gelinkt is, terwijl de stream toch echt ge-unlinkt wordt. Niet veel later crasht de SDK met een access violation.

Nadat er contact met DeltaCast werd opgenomen over dit probleem werd er gevraagd om een minimum working sample. Nadat deze werd gegeven kwam er de dag erna een patch welke het probleem oploste. Het wisselen tussen verschillende input formats werkte daarna vrijwel direct.

Het correct wisselen van output format levert vergelijkbare problemen op, zelfs met de patch. bij het opnieuw instellen van het format wordt de error VHDERR_OFFLINEPROPERTY teruggegeven. Bij deltacast is het zo dat bepaalde properties niet meer beschikbaar zijn als de stream draait. Op dit moment is deze error echter onverwacht: enkele regels code eerder is de stream successvol gestopt. Na het opnieuw starten van de stream treed er een acces violation op de eerstvolgende keer dat `VHD_GpuCopyToSlot()` wordt aangeroepen.

Bij het maken van een minimum working sample kon de error VHDERR_OFFLINEPROPERTY niet worden gereproduceerd. Ook is sinds de patch het gedrag van het programma licht veranderd. Na het wisselen van output format wordt error 42 teruggegeven, terwijl er maar 41 errors in de header staan gedefininieerd. Vermoeddelijk beschikken wij over een licht verouderde versie van de SDK.

Na veel zoekwerk is uiteindelijk het probleem opgelost: per abuis werd de stream tweemaal gestart. Dit is schijnbaar een valide scenario, want bij het 2e keer starten werden er geen errors teruggegeven. Dit was niet de enige bug, het gebruikte SDI scherm, een BlackMagic Smartview HD, is schijnbaar slecht getest met DeltaCast hardware. Na het veranderen van output format toont het scherm geen beeld meer. De enige manier om weer beeld te krijgen is om de stream compleet af te sluiten en weer opnieuw op te starten. Met andere

schermen kan dit probleem niet worden gereproduceerd.

Met het oplossen van dit laatste issue wordt de plugin stabiel en compleet genoeg geacht om mee te kunnen performance testen. Na het testen kunnen de vervolgstappen worden besproken.

9.4 TESTEN

In dit hoofdstuk wordt nog kort beschreven wat er wel en niet werkt aan de plugin.

De volgende functionaliteiten werken niet naar behoren:

- AMD grafische kaarten. Ondanks dat de VideoMasterHD SDK in principe onafhankelijk zou moeten zijn van het type of merk grafische kaart, werkt de plugin momenteel niet goed met AMD. Dit vereist nog wat uitzoekwerk om goed te krijgen, terwijl het zinvol is om nu al resultaten te krijgen. Het is natuurlijk ironisch dat deze iteratie juist bedoeld was om SDI-Link werkend te krijgen, terwijl dat nu juist niet het geval is.
- Bij hete switchen naar een input of output format onder de 720p komt er geen zinvolle data op het scherm, bij het teruggaan naar een 720p/i of 1080p/i formaat treed een crash op. Het is voldoende om te testen met de resoluties 1080p/i en 720p/i.
- als de computer in sleep gaat terwijl er streams draaien, verliest de plugin de input. Het is dan niet mogelijk om diezelfde input te gebruiken totdat de computer een reboot krijgt.
- de kleuren rood en blauw zijn omgedraaid. Het oplossen daarvan is een kwestie van het aanpassen van een matrix. Dit heeft geen effect op de performance.
- Geluid werkt niet. Dit heeft geen effect op de performance.

De volgende functionaliteiten werken wel. Het is mogelijk dat er crashes optreden omdat bijvoorbeeld de concurrency niet goed is of ergens foutafhandeling ontbreekt. Maar zijn voldoende stabiel om tests mee te draaien.

- Input streams.
- Output streams streams.
- Input stream waarvan het formaat tijdens het streamen verandert (er wordt automatisch overgeschakeld naar het nieuwe formaat).
- Output stream waarbij het formaat van het screen tijdens het streamen verandert (er wordt automatisch overgeschakeld naar het nieuwe formaat).

9.5 EVALUATIE

Het doel van deze iteratie was het ontwikkelen van een model waarmee beelden via SDI kunnen worden verzonden en ontvangen met de technologieën AMD SDI-Link en NVIDIA GPUDirect. Dat is ten dele geslaagd. Met een NVIDIA grafische kaart werkt de belangrijkste functionaliteit naar behoren. Met een AMD grafische kaart werkt de software nog niet. In theorie zou de VideoMasterHD SDK onafhankelijk moeten zijn van de videokaart, maar de praktijk blijkt anders.

Dat betekent dat er nu, met NVIDIA GPUDirect, getest kan worden op performance. Zodra er een redelijke inschatting gemaakt kan worden van de performance van de software kan er worden bekeken hoe het verder gaat met dit project.

10

ITERATIE 3: TESTEN NVIDIA GPUDIRECT

Dit gedeelte beschrijft de derde iteratie. Hierin wordt de ontwikkelde software aan bepaalde performance tests onderworpen.

10.1 PLANNING

In deze iteratie wordt de ontwikkelde software op basis van DAVE en de VideoMasterHD SDK getest. Het testen zal gebeuren met dezelfde software die in productie wordt gebruikt: VidiGo Live. De planning voor deze iteratie is als volgt:

- Planning 0.25 dag
- Uitzoeken requirements 0.25 dag
- testen 1.5 dagen
- beoordelen resultaten 0.5 dag
- evaluatie 0.5 dag

10.2 REQUIREMENTS

In deze iteratie wordt de performance van de ontwikkende software getest. Er zijn verschillende punten waaraan de performance kan worden gemeten:

- Memory bandwidth
- Latency
- CPU usage
- GPU usage
- hoeveelheid dropped frames

De memory bandwidth kan worden gemeten met Intel VTune. Intel VTune is een zogenaamde profiler. Met een profiler kan je precies zien wat je programma aan het doen is op een bepaalde regel code. Zoals de hoeveelheid branch mispredicts, en het percentage tijd wat in een bepaalde functie is besteed. Onder het Intel Sandy Bridge (de codenaam voor de CPU uit het testsysteem) patform zijn counters voor memory bandwidth aanwezig.

Om de memory bandwidh te isoleren kan gDEBugger worden gebruikt. gDEBugger is een tool waarmee OpenGL kan worden geprofiled en gedebugt. Deze tool kan bepaalde bewerkingen op de GPU uitschakelen, zoals draw calls. Hiermee kan worden gegarandeerd dat de grafische kaart niet de bottleneck is wanneer de memory bandwidth wordt bekeken.

De latency is een stuk lastiger te meten. Hiervoor kan mogelijk een opstelling worden gebruikt die ook toegepast wordt bij het meten van input lag op schermen. Hierbij worden er foto's gemaakt van 2 beeldschermen naast elkaar. Een reference signaal, en een signaal wat je wilt meten. Het reference signaal is vaak een CTR monitor. De reden hiervoor is dat een TFT een bepaalde input lag en processing lag heeft. Veel FTF's bevatten een interne scaler die enkele miliseconden latency toevoegen, zelfs wanneer de resolutie hetzelfde is als die van het paneel. Verder heeft een TFT ook tijd nodig om de pixels van kleur te veranderen. Deze factoren maken een TFT ongeschikt voor het meten van absolute latency. Voor het meten van relatieve latency kan je een TFT natuurlijk wel gebruiken.

Het bedenken en maken van de opstelling voor het correct meten van latency is dus niet makkelijk. Daarom wordt er besloten deze test later uit te voeren, wanneer de AMD implementatie ook gereed is.

Een andere performance metriek is de CPU usage. De CPU usage is met de huidige opzet geen probleem, maar het is natuurlijk altijd

zinvol om te weten of dat nu wel een probleem wordt.

Ook de GPU usage wordt gemeten. Ook hier kan gDEBugger voor worden ingezet. Volgens Arjan trekt de oude continu DeltaCastPlugin ongeveer 20% GPU usage. Volgens enkele korte testen die tijdens de vorige iteratie zijn gedaan, wordt de GPU usage snel een probleem met meer dan een handvol streams.

Ten slotte kan alles worden gecombineerd en er worden gekeken bij welke hoeveelheid streams de totale applicatie het niet meer aan kan. Hiervoor wordt gebruik gemaakt van de dropped frames counter in de plugin. Idealiter treden er geen dropped frames op. Hoe meer streams er gedraaid kunnen worden zonder dropped frames, hoe beter.

De specificaties van het testsysteem zijn als volgt:

Processor	Intel Core i7-3960X (6 cores / 12 threads)
Geheugen	8GB RAM (2 x 4 GB)
OS	Windows 7 x64
Videokaart	HP NVIDIA Quadro 4000 GainWard GTX 680 AMD FirePro V7900

10.3 TESTEN

De eerste taak is om de plugin zo ver te krijgen dat hij daadwerkelijk goed draait binnen VidiGo Live. Dat is een taak waar we ons in de eerste instantie behoorlijk op verkeken hebben.

Om te beginnen maakt DAVE gebruik van een hash om ervoor te zorgen dat er altijd compatibele plugins worden geladen. De hash wordt elke build opnieuw gegenereerd en bestaat uit een checksum van de header dan DAVE.

Met een disassembler kon de checksum uit een bestaande plugin worden gehaald, de hash werdt vervolgens op de juiste plek geplakt, waardoor de plugin kan worden gebouwd met de juiste versie-string. Dat leverde echter niet het gewenste effect op. Het starten van VidiGo Live levert de verhelderende error message 'could not find procedure entry point' op. Dat is een soort van catch-all error message die vaker voorkomt bij het inladen van DLL's, en houd in dat er iets fout is gegaan zonder dit nader te specificeren. Het meest waarschijnlijke is dat er een missende depencancy aanwezig is, maar met Dependancy walker kon dat probleem niet worden geverifieerd.

Een oplossing zou zijn om VidiGo Live en de plugins in dezelfde build te bouwen, met dezelfde headers. Dat blijkt helaas makkelijker gezegd dan gedaan. VidiGo Live draait nog op een ouder build systeem. In de praktijk houd dat in dat deze oplossing ook niet haalbaar is binnen te tijd die gealloceerd is voor het testen. In dat geval worden de testen 'gewoon' gedraaid in de DaveTestApp.

De nadelen hiervan is dat de omgeving van VidiGo live nogal wat overhead met zich mee brengt vanwege de UI die getekend moet worden. Als we willen weten hoe de plugin in de praktijk presteert is dat de meest zinvolle test. De resultaten zijn daarom niet representatief voor gebruik binnen VidiGo Live. Onderling zijn de resultaten natuurlijk wel te vergelijken.

De eerste serie test start een aantal input streams, en kijkt vervolgens hoe goed de performance is. Dat ging minder goed dan verwacht, bij 5 inputs treden er dropped frames. Dat houdt in dat de framerate in ieder geval onder de 50 zakt. Een beeld wat werd bevestigd door gDEBugger.

aantal streams	Videokaart	plugin	fps	opmerkingen
4x 1080i50 in	Quadro 4000	nieuw	> 50	geen dropped frames
5x 1080i50 in	Quadro 4000	nieuw	< 50	ongeveer 1 dropped frame per seconde
8x 1080i50 in	Quadro 4000	nieuw	30	veel dropped frames
8x 1080i50 in	Quadro 4000	oud	30	

Het lijkt er dus op dat de nieuwe plugin -in deze situatie- geen meetbaar effect heeft op de performance. De oude plugin bevat geen dropped frames counter, maar het aantal dropped frames is over het algemeen af te leiden aan de framerate.

Behalve de framerate is ook de gebruikte bandwidth van belang. Intel VTune bevat een optie om de bandwidth te meten, met een precisie van 10 miliseconden. Wanneer de streams eenmaal draaide werd er een sample gemaakt van enkele seconden lang. Intel VTune bevat geen simpele functie om het gemiddelde te berekenen, daarom werd het gemiddelde van handmatig berekend over 10 verschillende samples (totaal 0.1 seconden).

aantal streams	plugin	bandwidth
8x 1080i50 in	nieuw	2.2 GB/s
8x 1080i50 in	oud	7.0 GB/s

De benodigde memory bandwidth neemt dus fors af. Het is echter belangrijk om op te merken dat de memory bandwidth in deze situatie geen bottleneck vormt, de bottleneck lijkt volledig gevormd te worden door de GPU. gDEBugger bevat een optie om een eventuele bottleneck in de fragment shader te elimineren door het forceren van een simpele fragment shader. De framerate stijgt daarbij bij zowel de oude als de nieuwe plugin naar de 200, wat het maximale is wat het metertje kan aangeven. Door de constructie van de plugin wordt er bij framerates boven die van de in of output stream veel werk overgeslagen. Het is dus niet mogelijk om conclusies te trekken aan de hand van een framerate boven die van de stream.

Een ander scenario is door te testen met verschillende outputs. Er zijn 4 outputs aanwezig op de testbak, wat toevallig ook het maximale is wat mogelijk is met deze plugin:

aantal streams	Videokaart	plugin	fps	opmerkingen
4x 1080i50 out	Quadro 4000	nieuw	50	geen dropped frames
2x 1080i50 out	Quadro 4000	oud	60	geen dropped frames

Bij het toevoegen van steeds meer output streams neemt de framerate rechtevenredig af. De limiet wordt bereikt bij 4 streams met de nieuwe plugin, waarbij de framerate nipt boven de 50 blijft. Bij 2 streams kan de oude plugin het nog net bijbenen. Over het algemeen kan worden gezegd dat de nieuwe plugin bij output streams bijna 2 x zo goed presteerd als de oude. Bij het aansturen van 4 output streams is er echter te weinig headroom beschikbaar om nog iets zinvols te doen. Het openen van een enkele input stream of het tonen

van de UI van VidiGo live is dan waarschijnlijk al voldoende om ervoor te zorgen dat er frames beginnen te droppen.

Ten slotte is het ook zinvol om dezelfde tests te draaien met een wat krachtigere grafische kaart. Het testsysteem is normaal gesproken uitgerust met een NVIDIA GTX 680, deze is ongeveer 3 á 4 keer zo snel als de Quadro 4000. De Quadro 4000 betreft immers een niet al te moderne midrange kaart. Vanzelfsprekend is de GTX 680 niet compatibel met de nieuwe plugin.

aantal streams	Videokaart	plugin	fps	opmerkingen
8x 1080i50 in	GTX 680	oud	190	
4x 1080i50 out	GTX 680	oud	90	

De GTX 680m is in staat om beide tests goed te draaien. Wanneer een simpele fragment shader wordt geforceerd stijgt de framerate bij 4 outputs slechts licht, naar ongeveer 130 fps. De conclusie die we daaruit kunnen trekken is dat de bottleneck in deze situatie niet gevormd wordt door de GPU, maar door andere factoren. Zonder GPUDirect is het niet mogelijk om tegelijkertijd te processen en textures te downloaden. Vermoedelijk neemt het downloaden van de textures de meeste tijd in beslag. Het is duidelijk dat de GTX 680m zo snel is dat deze bij input simpelweg geen bottleneck meer vormt bij het maximale aantal streams wat momenteel op de testbak aangesloten kan worden.

10.4 EVALUATIE

In deze iteratie zijn er tests gedraaid met de oude en de nieuwe deltacast plugin, met NVIDIA hardware. Hieruit kunnen de volgende conclusies worden getrokken:

- De Quadro 4000 is te traag om een groot aantal streams aan te sturen.
- Bij meerde input streams valt er geen performancewinst te behalen. De benodigde bandwidth wordt verminderd, maar dat blijkt niet de bottleneck.
- Bij meerdere output streams valt een winst te behalen van ongeveer 80 tot 100%, dat wil zeggen dat er een stuk meer output streams tegelijk aangestuurd kunnen worden met NVIDIA GPUDirect.

Er bestaat dus in ieder geval een situatie waar GPUDirect zinvol is. Mogelijk levert de implementatie van AMD andere resultaten op.

11 ITERATIE 4: AMD SDI-LINK

In deze iteratie wordt een aantal bugs opgelost die momenteel een correcte werking de DeltaCast plugin verhinderen in combinatie met AMD hardware.

11.1 PLANNING

In deze iteratie wordt de DeltaCastPlugin ook geschikt gemaakt voor gebruik met AMD hardware. Ondanks dat de VideomasterHD SDK geen aparte functies bevat voor AMD en NVIDIA hardware werkt het resultaat wel op NVIDIA GPU's, maar niet op AMD GPU's. Het vreemde is dat de minimum working samples die tijdens dit project zijn gemaakt, stuk voor stuk correct werken.

Er wordt verwacht dat er ongeveer een week nodig is om deze iteratie af te ronden. Maar omdat de aard van het probleem nu nog niet in te schatten is, is het mogelijk dat het vinden van een oplossing veel langer duurt.

- Planning 0.25 dag
- Uitzoeken requirements 0.25 dag
- Implementatie 4 dagen
- Testen 0.5 dag
- evaluatie 0.5 dag

11.2 REQUIREMENTS

De requirements voor deze iteratie zijn erg simpel: alles wat nu werkt met NVIDIA, moet ook met AMD werken. Dat houdt in:

- SDI input.
- SDI output.
- Wisselen van input format.
- Wisselen van output format.

11.3 IMPLEMENTATIE

De software is momenteel erg onstabiel met de AMD FirePro V7900. Na een kort onderzoek worden de volgende problemen geconstateerd:

- Bij het openen van een input gaat alles goed totdat `VHD_GpuCopyFromSlot()` wordt aangeroepen. Dan crasht het programma. De call stack leid tot diep in de OpenGL implementatie van AMD.
- Bij het openen van een output gaat alles goed, maar verschijnt er alleen een paars beeld op de output.

Zoals eerder opgemerkt is de SDK in principe onafhankelijk van het type grafische kaart. Er zijn wel aparte DLL's voor AMD en NVIDIA, maar de API blijft hetzelfde.

Wat het nog vreemder maakt is dat er tijdens deze opdracht meerdere minimum working samples zijn gemaakt met Glut (in plaats van Qt + DAVE). Glut is een cross-platform OpenGL window manager. Al deze minimum working samples, zelfs die van het wisselen van de input of output, werken perfect.

Je zou daarom verwachten dat de code voor de minimum working sample niet helemaal hetzelfde werkt als deze in de DeltaCast plugin. Na de plugin goed te bestuderen kon echter geen verschil worden ontdekt in de aansturing van de VideoMasterHD SDK. Zelfs wanneer de code van de minimum working sample letterlijk naar de klassen in de DavePlugin wordt gekopieerd, faalt de software indien er een AMD grafische kaart gebruikt wordt.

De eerste ingeving was dat DAVE 'iets' doet met de OpenGL context die ervoor zorgt dat de aanroep van CopyFromSlot faalt. Om dit te verifiëren werd DAVE gestart in een van de minimum working samples, met als resultaat dat alles nog precies zo werkte als voorheen. Het blijft dus nog een groot mysterie wat er precies fout gaat. Dat de crash diep in een stukje closed-source software gebeurt helpt ook niet mee bij het zoeken naar een oplossing.

Een andere optie is dat Qt op de een of andere manier roet in het eten gooit. Dit kan worden getest door een Qt project op te zetten en daar onze minimum working sample in te draaien.

Na het porten van de minimum working sample werkte de functionaliteit in de eerste instantie. Daarna werd ook DAVE gestart. En jawel: even later treed er een crash op.

Uit nader onderzoek blijkt dat het goed gaat als de DaveContext niet wordt geactiveerd. Bij het activeren wordt de OpenGL context en OpenCL context die bij het window horen 'current' gemaakt. Aangezien DAVE op dat moment nog geen OpenGL of OpenCL context heeft wordt deze aangemaakt. Een van die 2 is dus de schuldige.

Versassend genoeg bleek dat OpenCL te zijn. Na het uitschakelen van de OpenCL mogelijkheid werkt alles naar behoren. Dat is vreemd, de VideoMasterHD SDK doet voor zover wij weten niets met OpenCL. Hoe het aanmaken van een OpenCL context invloed kan hebben op de stabiliteit van een stukje OpenGL code is vooralsnog een raadsel, maar we weten nu hoe het probleem kan worden vermeden: maak geen OpenCL context aan.

Binnen DAVE wordt OpenCL gebruikt voor het up en downloaden van textures. Dit zou met NVIDIA significant beter presteren dan OpenGL, het performanceverschil met AMD is onbekend. OpenCL kan zonder problemen worden uitgeschakeld, DAVE valt dan terug op OpenGL texture uploads. Aangezien de texture up en download functionaliteiten van DAVE niet worden gebruikt in de aangepaste plugin levert dat in simpele testcases geen performanceverlies op.

Hierna werdt er een aparte versie van dave.dll gecompileerd waarin de OpenGL functionaliteit is uitgeschakeld. Met gebruik van deze .dll werkt alles in de DeltaCastPlugin naar behoren, precies zoals dat bij NVIDIA ook het geval is.

11.4 TESTEN

Dezelfde functionaliteit als bij NVIDIA wordt getest:

- Input.
- Output.
- Wisselen van input format.
- Wisselen van output format.

Al deze functionaliteiten blijken prima te werken met een AMD grafische kaart.

11.5 EVALUATIE

In deze iteratie zijn er enkele bugs opgelost in de samenwerking met AMD. De plugin werkt nu ook naar behoren in combinatie met de AMD FirePro V7900.

Dat betekent dat er nu verder kan worden gegaan met de volgende stap, namelijk het testen van de prestaties.

11.6 IMPLEMENTATIE 2

Toen er begonnen was aan het testen bleek dat de correcte werking van de plugin niet goed genoeg gecontroleerd was. In release mode crasht de plugin elke keer dat er een input stream wordt geopend. In de vorige iteratie is er alleen getest met de debug build.

De functie waarop een access violation wordt gegeven is `VHD_GpuCopyFromSlot()`. Het is lastig om er achter te komen wat precies het probleem is. Het is wel mogelijk om een PDB (Program Debug Database) file te genereren zodat ten minste de stack goed bekeken kan worden. De stack trace eindigt diep in de atioglxx.dll, dat is een library die bij de drivers van AMD hoort.

Na enige moeite kan het probleem ook in debug modus worden gereproduceerd: de stack trace eindigt dan in de `delete` methode. Dat wijst op een corrupte heap. In tegenstelling tot de release build crasht het programma niet elke keer, maar alleen bij een specifieke timing optreed. Wanneer de stream na ongeveer 700 a 750 ms na het openen wordt gestart. Met die informatie kunnen we helaas niet zo veel, waardoor er naar andere aanknopingspunten gezocht moet worden.

Zoals vermeld doet het probleem zich niet voor in de minimum working sample. Een van de verschillen tussen de plugin en de minimum working sample is de pauze functionaliteit. Na het inbouwen van de pauze functionaliteit en een delay crasht de sample.

Uit nader onderzoek blijkt dat de crash optreed wanneer `VHD_GpuCopyFromSlot()` wordt aangeroepen na 30 a 40 ms na het unpausen van de stream. Zonder de delay treed het probleem niet op. Dit is dus duidelijk een bug in de VideoMasterHD SDK.

DeltaCast is op de hoogte gebracht van dit probleem. Ze schrijven dat het kan voorkomen dat je wel een slot kan openen met `VHD_LockSlotHandle()`, maar dat er geen data aanwezig is in dat slot. De SDK houdt geen rekening met deze corner case. Er bestaat echter een workaround voor: het is mogelijk om de hoeveelheid data in het slot te controleren. Zit er geen data in het slot, dan weet je dat `VHD_GpuCopyFromSlot()` beter overgeslagen kan worden.

11.7 TESTEN 2

In tegenstelling tot de vorige test-iteratie is er dit maal wel in release mode getest. Daaruit blijkt dat de plugin goed functioneert in basis-somstandigheden.

11.8 EVALUATIE 2

De nieuw ontdekte bug is nu ook opgelost. Daardoor kan er nu, hopelijk zonder verdere verrassingen, verder worden gegaan met het testen van de plugin in combinatie met de AMD V7900.

12

ITERATIE 5: TESTEN AMD SDI-LINK

In deze iteratie wordt de performance van AMD SDI-Link getest.

12.1 PLANNING

In deze iteratie wordt de ontwikkelde software op basis van DAVE en de VideoMasterHD SDK getest. Het testen zal gebeuren met de DaveTestApp De planning voor deze iteratie is als volgt:

- Planning 0.25 dag
- Uitzoeken requirements 0.25 dag
- testen 1.5 dagen
- beoordelen resultaten 0.5 dag
- evaluatie 0.5 dag

12.2 REQUIREMENTS

In deze iteratie wordt de performance van de ontwikkelde software getest in combinatie met de AMD FirePro V7900 getest. Er zijn verschillende punten waaraan de performance kan worden gemeten. Dezelfde tests als bij NVIDIA worden gedraaid, zodat de implementaties goed kunnen worden vergeleken.

De relevante hardware is hetzelfde als voorheen:

Processor	Intel Core i7-3960X (6 cores / 12 threads)
Geheugen	8GB RAM (2 x 4 GB)
OS	Windows 7 x64
Videokaart	HP NVIDIA Quadro 4000 GainWard GTX 680 AMD FirePro V7900

12.3 TESTEN

Er wordt zoals gebruikelijk getest binnen de DaveTestApp. We testen eerst met een aantal input streams

aantal streams	Videokaart	plugin	fps	opmerkingen
5x 1080i50 in	AMD V7900	nieuw	90	geen dropped frames
6x 1080i50 in	AMD V7900	nieuw	60	soms dropped frames
5x 1080i50 in	AMD V7900	oud	60	
6x 1080i50 in	AMD V7900	oud	41	

Bij de nieuwe plugin fluctueert de framerate veel, dat leidt tot meer droped frames dan bij de oude plugin.

Bij nader inzien bleek echter dat de oude plugin wel een dropped frame counter bevat. Maar dat er pas frames beginnen te droppen onder de 25 fps. Misschien is de plugin zo geconstrueerd dat hij, bij een interlaced beeld, beide fields (even en odd) in een keer overzet. Waarbij het dus mogelijk is een 50hz stream aan te sturen met 'maar' 25 updates per seconde. Dat betekent dat de resultaten die eerder getest zijn ongeldig zijn. De framerate is namelijk geen goede indicator voor het feit of de streams wel of niet vloeiend kunnen worden uitgestuurd. Het is daarom noodzakelijk om de tests bij NVIDIA opnieuw uit te voeren. Het metertje voor framerate bij NVIDIA wordt daarbij zo veel mogelijk genegeerd. Het is duidelijk dat de framerate geen goede indicatie geeft over de performance.

De volgende tabellen geven aan of er bij de test succesvol is verlopen. Onder succes wordt verstaan: er komen geen of nauwelijks dropped frames voor.

aantal streams 1080i50 input	oude plugin	nieuwe plugin	nieuwe plugin geen color conversie
1	✓	✓	✓
2	✓	✓	✓
3	✓	✓	✓
4	✓	✓	✓
5	✓	✓	✓
6	✓	soms	✓
7	✓	soms	✓
8	✓	×	×

Verschillende input streams met de AMD FirePro V7900

Het is te zien dat de hoeveelheid streams zelfs iets afneemt. Wat niet in deze tabel te zien is is dat de framerate erg onstabiel is, dat zorgt voor de achteruitgang. De absolute framerate was bij gebruik van de nieuwe plugin hoger dan bij de oude plugin.

aantal streams 1080i50 output	oude plugin	nieuwe plugin	nieuwe plugin geen color conversie
1	✓	✓	✓
2	✓	✓	✓
3	✓	✓	✓
4	×	✓	✓
5	×	✓	✓
6	×	✓	✓
7	×	✓	✓
8	×	×	×

Verschillende output streams met de AMD FirePro V7900

Vergelijkbaar met NVIDIA neemt de hoeveelheid mogelijke output streams toe.

Daarna zijn de tests herhaald met een NVIDA Quadro 4000.

aantal streams 1080i50 input	oude plugin	nieuwe plugin	nieuwe plugin geen color conversie
1	✓	✓	✓
2	✓	✓	✓
3	✓	✓	✓
4	✓	✓	✓
5	✓	✓	✓
6	✓	✓	✓
7	✓	✓	✓
8	✓	×	×

Verschillende input streams met de NVIDIA Quadro 4000

En ook met output streams:

aantal streams 1080i50 input	oude plugin	nieuwe plugin	nieuwe plugin geen color conversie
1	✓	✓	✓
2	✓	✓	✓
3	✓	✓	✓
4	✓	✓	✓
5	×	✓	✓
6	×	✓	✓
7	×	×	✓
8	×	×	✓

Verschillende output streams met de NVIDIA Quadro 4000

Vergelijkbaar met AMD neemt de hoeveelheid streams toe. Het is duidelijk dat in dit geval de grafische kaart een bottleneck vormt. Wanner de color-space conversie wordt overgeslagen neemt de hoeveelheid output streams die kan worden aangestuurd toe.

12.4 EVALUATIE

Na deze testiteratie kunnen de volgende conclusies worden getrokken:

- Zowel bij AMD and NVIDIA neemt de maximale hoeveelheid vloeiend aan te sturen input streams iets af.
- Bij AMD neemt de hoeveelheid output streams met meer dan 133% toe.
- Bij NVIDIA neemt de hoeveelheid output streams met 50 tot 100% toe.
- Bij output streams met een nvidia kaart is de GPU duidelijk de bottleneck.

Nu rest de vraag: In wat voor situaties is deze techniek zinvol?

Over AMD SDI-Link kunnen we kort zijn: de V7900 is momenteel de enige kaart die wordt ondersteund door deze techniek. Uit de tests die tot nu toe zijn gedraaid blijkt dat de V7900 niet significant sneller is dan de GTX 680, op basis van prijs kan AMD ook niet concurreren. Verder zijn er geen echte argumenten te verzinnen die in het voordeel zijn van AMD.

Wat betreft NVIDIA is de conclusie minder eenzijdig. Weer geldt dat de Quadro 4000 het verliest van de veel snellerer GTX 680. Maar in tegenstelling tot AMD biedt NVIDIA ook snellere varianten aan die GPUDirect ondersteunen, zoals de Quadro K5000. Deze kaart biedt ongeveer 70% van de performance van de GTX 680. Op basis daarvan is de verwachting dat een Quadro K5000 voor enige winst ten opzichte van de GTX 680 gaat zorgen. Hoe groot die winst is is zonder het in huis hebben van de hardware erg lastig.

13

ITERATIE 6: BUILDEN VIDIGO LIVE

Uit de testen en verhalen over de performance lijkt het zo te zijn dat er een aanzienlijk performance verschil zit tussen de theorie (DaveTestApp) en de praktijk (VidiGo Live). Het lijkt interessant om dat performanceverschil te kwantificeren.

13.1 PLANNING

De planning van deze iteratie is als volgt:

- Planning 0.25 dag
- Uitzoeken requirements 0.25 dag
- implementatie 5 dagen
- testen 3 dag
- evaluatie 0.5 dag

"testen" is inclusief de performance tests. Mocht het niet binnen de tijd mogelijk zijn om een testsetup te maken met VidiGo live, dan wordt de iteratie afgebroken.

13.2 REQUIREMENTS

Nu de testresultaten van de plugin grotendeels bekend zijn zijn er een aantal opties waar nog tijd aan besteed kan worden. Onder andere:

- Toch nog testen in VidiGo Live.
- De code van de plugin verbeteren

Vanwege het vermeende performanceverschil lijkt het interessant om te testen met VidiGo live. Zoals eerder vermeld draait VidiGo live nog in een ouder build systeem, het is onbekend of het lastig is om de vernieuwde DeltaCastPlugin werkend te krijgen in VidiGo Live.

Het doel van deze iteratie is om een poging te wagen om de vernieuwde DeltaCast plugin werkend te krijgen in VidiGo Live.

13.3 IMPLEMENTATIE

Het 'nieuwe' build systeem van VidiGo draait op CMAKE + git, terwijl VidiGo Live nog draait in een oudere omgeving een omgeving met een python + svn.

Het build systeem omvat het automatisch buildan van alle dependancy's. Bij een project als VidiGo Live zijn dat ongeveer 20, zoals Qt, DAVE, SDL, en SDK's van diverse fabrikanten. Het porten van de packages is een work in progress. Veel packages zijn gedeeltelijk geport naar CMAKE, en werken nog niet optimaal.

Een bijzonder lastig te builden package is net_SDL. Deze wordt gebruikt voorr threads in de aansturing van een stukje externe hardware. De ondersteuning voor het stukje hardware zit in VidiGo Live zelf gebouwd, in plaats van netjes met een interface en een DLL, zoals de plugin uit deze afstudeeropdracht is gemaakt. Het is derhalve niet mogelijk om VidiGo Live te bouwen zonder net_SDL.

De constructie van net_SDL verdient geen schoonheidsprijs: deze bestaat uit een Visual Studio 6 project file. Via de command line wordt het project geupgrade naar de geïnstalleerde versie van visual studio.

Daarna wordt er met sed, een linux stream editor, een aanpassing gemaakt in de geüpgradede project file, wat intern gewoon een XML file is. Deze stap werkte dan ook niet naar behoren. Het project moest handmatig worden geüpgrade, daarna moeten de dependancy's handmatig worden toegevoegd.

Daarna blijkt net_SDL nog steeds niet correct te builden, er blijkt een cmakelists.txt te ontbreken.

13.4 EVALUATIE

Omdat er al enkele dagen wordt gewerkt aan het builden van VidiGo Live wordt deze iteratie gestaakt. Tot op heden zijn ongeveer de helft van de dependancy's gebuild. Het ziet er naar uit dat het compleet werkend krijgen van VidiGo Live nog minstens een week gaat duren, die tijd kan beter besteed worden aan andere dingen.

14

ITERATIE 7: OVERIGEN

Nu dit project ten einde komt wordt het tijd om te kijken wat we tijdens dit project eigenlik hebben geleerd, met de bedoeling dit in een begrijpelijke vorm te presenteren aan de opdrachtgever.

14.1 RAPPORT

Het rapport is bedoeld voor de opdrachtgever en developpers bij VidiGo. Deze moet een korte inleiding geven tot:

- De gebruikte technieken, AMD SDI-Link en NVIDIA GPUDIirect.
- De wijze waarop de tests zijn gedaan.
- De resultaten van die tests.
- De conclusies die daaruit getrokken kunnen worden.
- Eventuele bugs.

De inhoud van dit verslag bestaat voor een groot deel uit het gevolgde process. Dat is voor de opdrachtgever niet van belang.

Tijdens het schrijven van dit rapport bleken er nog een aantal kleine dingen te missen.
Zo zijn er tot 8 output streams getest met de AMD FirePro V7900 en Quadro 4000. Terwijl er met de GTX 680 tot 4 output streams zijn getest.

Daarnaast wordt er in dit verslag met slechts weinig woorden gesproken over de latency, terwijl dat juist een voordeel zou zijn van de nieuwe technieken. De latency is echter lastig te meten, wat de reden is geweest dat dit aspect tot nu toe onbesproken is gebleven. In het rapport is daarom een theoretische uitleg toegevoegd over de theoretische latency winst.

Ten slotte is er ook een theoretische uitleg toegevoegd over de verwachte bandwidth. Het blijkt dat de gemeten bandwidth bij de nieuwe methode vrij nauwkeurig aansluit bij de daadwerkelijk verbruikte bandwidth. Bij de oude methode zit er echter meer dan een factor 2 verschil tussen. Uit nader onderzoek bleek dat DAVE extra buffering toepast. Zoals in het verslag is vermeld worden alle textures in een pool (van 8k bij 8k pixels, RGBA) gezet, welke vervolgens in één keer naar de GPU word geup- of download. Dit levert in de praktijk een wist op qua hoeveelheid aanstuurbare streams. Vermoedelijk omdat het uploaded van losse textures duur is, maar het is evident dat er hierdoor meer bandwidth verbruikt wordt. De gemeten bandwidth wijkt zelfs wanneer dit wordt meegerekend met meer dan een factor 2 af.

Vervolgens werden een aantal pogingen gedaan om er achter te komen waar dat verschil in bandwidth zit. Daarvoor is gemeten hoeveel bandwidth het uploaden van de textures eigenlijk kost, in het

ideale geval is de 'overhead' een factor 3: éénmaal schrijven, tweemaal lezen. De tool die voor het meten wordt gebruikt, Intel VTune, rapporteert alleen de totale bandwidth en de read bandwidth.

Wanneer er een texture van 8k bij 8k pixels wordt geupload resulteerde dit in een totaal verbruikte bandwidth van 1822MB, een overhead van 7.1x vergeleken met de 256MB grote texture. Dat is dus een aanzienlijk grote overhead.

Het is ook mogelijk om een gedeelte van een texture te uploaden. Deze functionaliteit wordt ook door DAVE gebruikt. Wanneer een gebied van 8k bij 1k wordt geüpdate met behulp van `glTexSubImage` wordt er een verkeer gemeten van 730.92MB. 22.8 maal zo veel als de groote van het te updaten gebied. Het ziet ernaar uit dat de oorzaak van het verschil tussen theorie en praktijk is gevonden.

Helaas bleek dat bij nader inzien toch niet zo te zijn. Bij deze testen is gebruik gemaakt van een laptop met NVIDIA optimus. NVIDIA optimus is een techniek om meerdere grafische kaarten in hetzelfde systeem te hebben, zodat daartussen gewisseld kan worden. Schijnbaar zorgt dat voor een enorme achteruitgang in efficientie, na het uitschakelen van NVIDIA optimus kwamen de cijfers erg dicht in de buurt van het theoretisch haalbare. Wat nu daadwerkelijk voor het verschil tussen de bandwidth in theorie en praktijk zorgt, kon helaas niet worden verklaard.

Het resulterende rapport is bijgevoegd als bijlage.

15

CONCLUSIE

Tijdens dit project zijn er implementaties gemaakt voor de software NVIDIA GPUDirect en AMD SDI-Link, met behulp van SDI kaarten van de fabrikant DeltaCast. Hier komt uit dat de hoeveelheid mogelijke output streams ongeveer verdubbelt bij gebruik van een van deze technologiën. Bij NVIDIA meten we een winst van 50% tot 100% in de hoeveelheid gelijktijdig aanstuurbare output streams ten opzichte van de oude implementatie. Bij AMD zien we op dit gebied een winst van 133%. Bij meerdere input streams is de techniek nadelig, en neemt het aantal mogelijke streams licht af. Bij NVIDIA met ongeveer 10%, bij AMD met ongeveer 30%.

De conclusie die hieruit getrokken kan worden is dat NVIDIA GPUDirect en AMD SDI-link zinvolle technologiën zijn, maar slechts voor een beperkte set use cases:

- Klanten die een grote hoeveelheid output streams willen aansturen zullen het meest baat hebben bij een van de ondersteunde NVIDIA kaarten. Er zijn 2 redenen waarom er niet voor een AMD kaart gekozen wordt:
  1. AMD levert slechts één type kaart wat SDI-Link ondersteunt, slechts een gemiddeld snelle kaart.
  2. Er wordt binnen VidiGo al jaren met NVIDIA kaarten wordt gewerkt. Alhoewel de gebruikte technologieën in principe cross platform zijn, gaat er veel tijd in zitten om de correctheid van alle aplicaties bij een non-NVIDIA OpenGL implementatie te controleren.
- Klanten die *niet* meer output streams aan hoeven te sturen dan met de oude techniek mogelijk is, hebben geen voordeel van NVIDIA GPUDirect of AMD SDI-Link.

In feite is er dus maar één use case waar NVIDIA GPUDirect interessant is. Door het beperkte aanbod van kaarten met AMD SDI-Link ondersteuning is deze techniek zelfs in geen enkele relevante use case interessant. Dat laatste kan natuurlijk veranderen als AMD besluit ook high-end kaarten te ondersteunen.

In tegenstelling tot wat bij het begin van dit project werd gedacht is de memory bandwidth niet de bottleneck. Met het testsysteem met een dual-channel memory opstelling van het testsysteem koden tot 16 streams vloeiend worden afgespeeld. Meer SDI in of outputs waren

niet beschikbaar. Bij die opstelling werdt het limiet van memory bandwidth niet gehaald.

Een quad-channel opstelling heeft ongeveer 80% meer memory bandwidth dan het testsysteem, waarmee ten minste 29 streams mogelijk zou moeten zijn voordat de bandwidth een probleem zou kunnen vormen.

Het is echter mogelijk dat er in de toekomst meer bandwidth benodigd is, bijvoorbeeld door de uitrol van 4K of zelfs 8K video. Zowel AMD SDI-Link als NVIDIA GPUDirect zorgen voor een verlaging van de benodigde bandwidth met een factor 3 á 4.

Ten slotte leveren zowel AMD SDI-Link als NVIDIA GPUDirect een theoretische latencywinst op van 4.00ms, van input naar output met 4 input en 4 output streams. Of deze winst ook in de praktijk zichtbaar is, kon helaas niet worden getest.

16 AANBEVELINGEN

Het project is nu tot een einde gekomen. Er zijn enkele vervolgstappen mogelijk. De meest voor de hand liggende is om te gaan kijken of er klanten bestaan die baat hebben bij de specifieke use case van meer output streams, en daar tevens bereid zijn voor te betalen.

Als dat zo is, dan moet:

- De plugin gecontroleerd worden op correctheid.
- De plugin worden getest op performance, in combinatie met een snellere NVIDIA Quadro K5000.

Dat is tevens mijn advies voor het bedrijf: zoek uit of klanten baat hebben bij de winst die door NVIDIA GPUDirect wordt geboden, en ontwikkel vervolgens de plugin door als blijkt dat daar vraag naar is.

Een andere vervolgstap is om de AMD SDI-Link verder te testen met de SDI kaarten van JAI, Bluefish of DataPath. Deze fabrikanten ondersteunen, in tegenstelling tot DeltaCast, directe P2P transfers met AMD SDI-Link. Het is mogelijk dat hierdoor de AMD V7900 wel potent genoeg is voor grote aantallen streams als er met P2P transfers wordt gewerkt.

17

PROCESS EVALUATIE

In de onderstaande tabel staat steeds welke activiteit gepland is, gevolgd door welke activiteit daadwerkelijk is uitgevoerd.

geplance activiteit	duur dagen	uitgevoerde activiteit	duur dagen
Plan van aanpak	1	Idem	1
Onderzoeken	5	Idem	2
Ontw. oude methode	20	IDEM	6
Ontw. GPUDirect	15	Ontw. generieke plugin	37
-	-	testen Qadruo 4000	2
Ontw. SDI-Link	15	bugs oplossen in SDI-Link	10
Testen performance	3	Testen V7900	3
-	-	compileren VidiGo Live	4
Adviseren	2	Rapport & overigen	4

NB: de geplande en uitgevoerde acticiteiten lopen niet synchroon, omdat bij de uitgevoerde activiteiten de dagen gereserveerd voor het verslag niet zijn meegenomen.

Er waren 2 redenen dat het ontwikkelen van de generieke plugin veel langer duurde dan verwacht:

- Er kon bij de start van de iteratie nog niet worden begonnen met ontwikkelen, omdat de SDK (VideoMasterHD GPU extension) nog niet beschikbaar was.
- Er was nog weinig ervaring met de SDK en DAVE, deze ervaring zou tijdens het ontwikkelen van de oude methode worden opgegaan, maar in die iteratie viel er niets te ontwikkelen. Derhalve is de ontwikkeling van de plugin begonnen met minder ervaring met DAVE & de VideoMasterHD SDK dan gepland.

Desondanks heeft dit geen invloed gehad op de looptijd van het project.

18 BEROEPSTAKEN

Bij start van dit project zijn de volgende beroepstaken geselecteerd:

A1 ANALYSEREN VAN HET PROBLEEMDOMEIN
: Aan het begin van deze iteratie is een onderzoek gedaan naar de te gebruiken technologien die voordeel zouden moeten bieden tov. de bestaande software.

A4 KIEZEN VAN EEN ONTWIKKELSTRATEGIE EN ONTWIKKELMETHODIEK
: Aan het begin van dit project is er een gewogen beslissing gemaakt over de ontwikkelstrategie.

C8 ONTWERPEN VAN EEN TECHNISCH INFORMATIE SYSTEEM
: Deze beroepstaak is niet behaald, omdat er slechts een minimale hoeveelheid te ontwerpen viel.

C10 ONTWERPEN VAN EEN SYSTEEMARCHITECTUUR
: Deze broepstaak is niet behaald, omdat er bij dit project slechts een minimale hoeveelheid te ontwerpen viel. De uiteindelijke architectuur lijkt erg op de beginsituatie.

D16 HET REALISEREN VAN SOFTWARE
: In iteratie 2 en 4 is aangetoond dat deze broepstaak is behaald.

D17 TESTEN VAN SOFTWARE SYSTEMEN
: De tussenproducten, zoals gemaakt in iteratie 2 en 4, zijn getest tijdens en na deze iteraties. Eventuele bugs zijn grotendeels opgelost, waarvan enkelen indirect in third-party SDK's.

E24 KWANTITATIEVE ANALYSE MAKEN VAN DE PRESTATIES VAN EEN SYSTEEM
: De testen uitgevoerd in onder andere iteratie 3, 5 en 7 tonen aan dat deze beroepstaak is behaald.

G1 PRAKTISCHE ASPECTEN HANTEREN IN (INTERNATIONALE) PROJECTEN
: Deze beroepstaak is behaald, aan de start van het project is er een opdrachtonschrijving, planning en risicoanalyse opgeleverd

H5 ZELFSTANDIG WERKEN
: Dit project werd grotendeels zelfstandig uitgevoerd.

H7 METHODISCH WERKEN
: Tijdens dit project is de vooraf gemaakte planning met redelijke nauwkeurigheid gevolg. Dit staat mede uitgelegd in het hoofdstuk process evaluatie.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Current Version as of June 6, 2013

DECLARATION

Stefan Dessens, June 6, 2013