KU LEUVEN

KU LEUVEN CAMPUS BRUGGE

FACULTEIT INDUSTIËLE INGENIEURSWETENSCHAPPEN

# Computerarchitecturen, geheugenbeheer en interfacing

## Performantie analyse (Jetson Nano) GPU vs CPU via lijn detectie

**Timon Claerhout**

Docent: S. Verslype

Academiejaar 2023-2024

# Inhoudsopgave

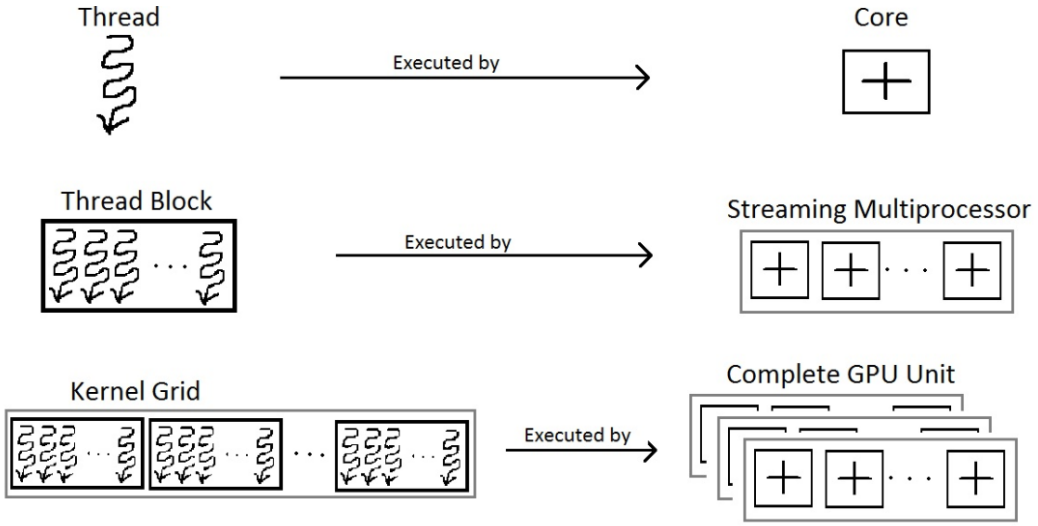
# 1. Inleiding

In deze paper wordt het performantie verschil onderzocht tussen de Jetson Orin Nano Developer kit GPU en de i7-8565U CPU. Dit aan de hand van een lijn detectie project dat onder andere gehanteerd wordt in auto's.

De implementatie van het project wordt getest door een video opname (met een resolutie van 640x480 pixels) van de autobaan af te spelen. Per frame worden allerlei filters op iedere pixel toegepast om uiteindelijk de lijnen te detecteren en weer te geven. Deze rekenintensieve filters worden volledig geparallelliseerd via CUDA op de GPU. Dit niet rechtstreeks doordat de data allereerst moet gekopieerd worden vanuit de Jetson CPU (host) naar de Jetson GPU (device) en terug. Op de CPU wordt alles meteen serieel uitgevoerd waardoor er geen data moet gekopieerd worden. Hoe dit wordt verwezenlijkt en wie de hoogste performantie haalt wordt in deze paper verder beschreven.

# 2. Jetson Orin Nano Developer kit

De Jetson heeft een krachtige Nvidia GPU dat 1024 CUDA cores bevat. Het aantal cores is een belangrijke factor bij het parallelliseren van een filter doordat dit het aantal threads bepaald die tegelijk worden uitgevoerd. Deze threads worden gegroepeerd in een thread block wat zal uitgevoerd worden door de Streaming Multiprocessor. Er worden meerdere thread blocks aangemaakt zodat een grid ontstaat dat parallel uitgevoerd zal worden. Onderstaande figuur biedt verduidelijking:




*Figuur 1 SM diagram, via http://thebeardsage.com/cuda-streaming-multiprocessors/*

De dimensie van deze kernel grid is bepaald door het aantal CUDA cores van de Jetson Nano, waardoor we 32 threads per block genereren met een grid lengte van 32 thread blocks. Deze dimensie is het absolute maximum wat niet mag worden overschreden doordat 32*32=1024, op deze manier benutten we alle beschikbare CUDA cores.

Om deze kernel te compileren moet het keyword '@cuda.jit' boven iedere geparallelliseerde filter. Hierdoor weet de compiler dat de functie moet uitgevoerd worden via CUDA en wordt de Python code vertaald naar PTX code. Dit zal uitgevoerd worden door de Jetson Nano.

Bij het aanroepen van een filter moet de thread- en block dimensie altijd meegegeven worden zodat de gehele grid wordt uitgevoerd met alle cores. Meer gedetailleerde informatie over hoe deze methodiek is toegepast op de geparallelliseerde filters is terug te vinden op de volgende pagina's.

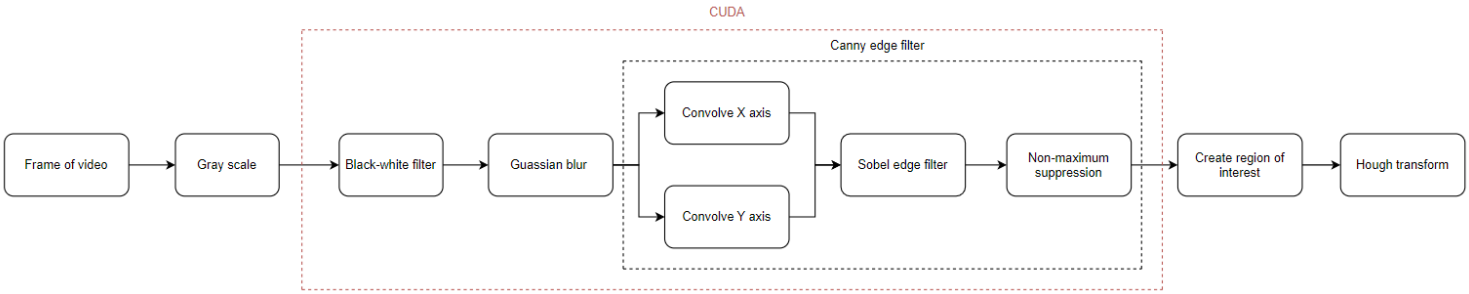
# 3. Lijn detectie

Om de lijnen van een autobaan te detecteren moet onderstaand resultaat bekomen worden:

*Figuur 2 Orginele frame*

*Figuur 3 Resultaat lijn detectie*

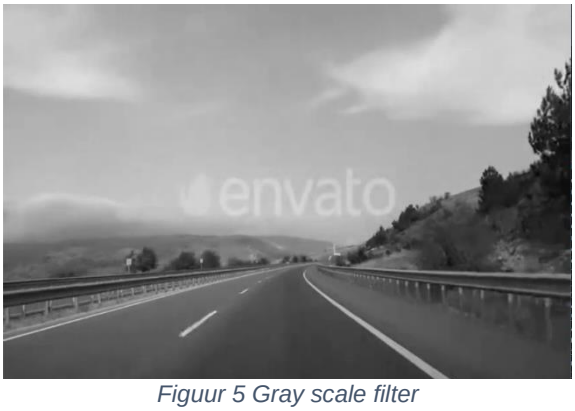
Om dit te verwezenlijken worden allerlei filters toegepast, hiervoor hanteren we volgende image processing pipeline. Waarbij alle filters binnen de CUDA stippellijn volledig geparallelliseerd zijn. De hough transformatie is als enige filter overgelaten door zijn grote complexiteit om CUDA toe te passen. De gray scale filter wordt rechtstreeks toegepast via OpenCV wanneer de frame ontvangen wordt.




## Gray scale filter

Bij het ontvangen van een frame uit de video wordt er allereerst een gray scale filter toegepast. Dit om de algoritmes voor de diverse filters te vereenvoudigen en te versnellen. Op een kleuren frame bestaat iedere pixel uit een RGB waarde (3 dimensies) dat elk een 8 bit waarde bevat. Hierdoor zijn 24 bits nodig zijn om 1 pixelkleur voor te stellen. Via deze filter wordt de 24 bit RGB waarde vereenvoudigd naar een benaderende 8 bit waarde. Hierdoor zal iedere pixel een waarde zal hebben tussen 0 en 255 (1 dimensie). Met 0 volledig zwart en 255 volledig wit.

*Figuur 4 Originele frame*



*Figuur 5 Gray scale filter*

## Zwart-wit filter en gaussian filter

De zwart-wit filter zorgt voor een scherpere overgang tussen de witte lijnen en de rest van de foto, hierdoor zijn ze gemakkelijker detecteerbaar. Door iedere 8 bit pixelintensiteit op te vragen, wordt er gekeken als deze waarde boven/onder 127 is. Bij waarden tussen 0 en 127 wordt een 0 toegekend, alle andere waarden krijgen waarde 255. Hierdoor is er enkel een zwart-wit overgang.

De gaussian filter zorgt voor een vervaging van de foto hierdoor zal er ruis uit de foto weg gefilterd worden doordat de foto vloeiendere beeld overgangen zal bevatten. Dit wordt in het project verwezenlijkt door een benaderende 3x3-gaussian kernel toe te passen op het beeld.


*Figuur 6 Gray scale filter*

*Figuur 7 Zwart-wit filter*

## Convolutie op X-as en Y-as

De sobel edge filter gebruikt twee 3x3-kernels die zijn geconvolueerd met het zwart-wit beeld om benaderingen van de afgeleiden te berekenen, één voor horizontale veranderingen (X-as) en één voor verticale (Y-as). Als we A definiëren als het zwart-wit beeld, en Gx en Gy twee beelden die op elk punt respectievelijk de horizontale en verticale afgeleide benaderingen bevatten, zijn de berekeningen als volgt:

$$\mathbf{G}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

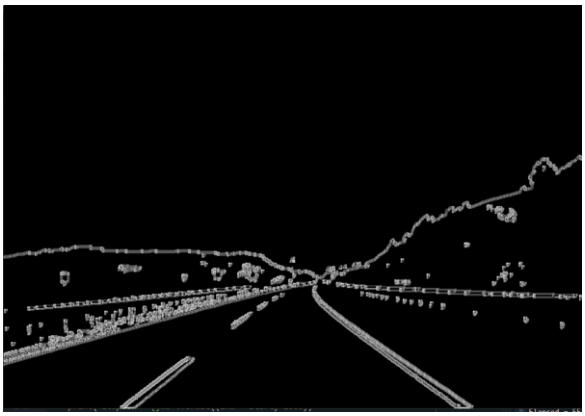
*Figure 2 X-Y convolutie*

*Figuur 8 convolutie X-as*

*Figuur 9 Convolutie Y-as*

## Sobel edge filter en non-max suppression filter

De sobel edge filter combineert de voorgaande 2 convoluties om de totale verandering van het beeld te bekomen via de stelling van Pythagoras (zie figuur 10). Ook de hoek wordt berekend via arctan($\frac{Y}{X}$). De hoek wordt verder gebruikt in de non-max suppression filter zodat de dikte van de lijnen door de sobel edge filter verkleint worden om zo een preciezer resultaat te bekomen. Via deze hoek wordt elke pixel doorlopen en vergelijken we dit met de aangrenzende pixels langs de gradiëntrichting. Als de gradiëntgrootte van de centrale pixel de grootste is onder zijn buren, betekent dit dat deze pixel waarschijnlijk deel uitmaakt van een rand, en dat we deze behouden. Indien dit niet het geval is, onderdrukken we het door de intensiteit ervan op nul te zetten en het buiten beschouwing te laten als randpixel. Hierdoor wordt figuur 11 bekomen.



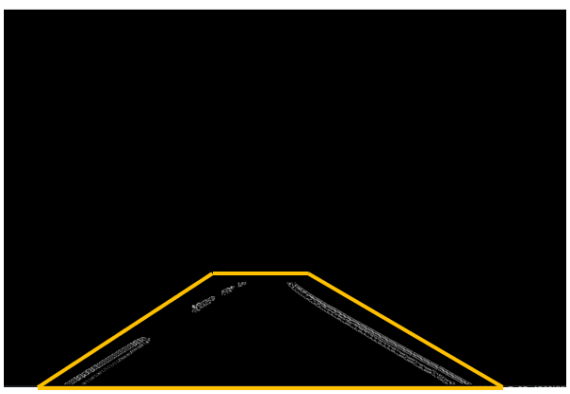
*Figuur 10 Sobel edge filter*

*Figuur 11 Non-max suppression*

## Bepalen Region of interest en hough transformatie

Aangezien enkel de lijnen relevant zijn in de frame, kunnen we een deel hiervan wegsnijden. Via een trapezium vormig figuur worden enkel de lijnen behouden en de rest weg gefilterd (zie figuur 12).

Hierop wordt de hough tranformatie toegepast, deze functie is niet geparallelliseerd in CUDA door zijn complexe implementatie maar wordt al reeds heel snel uitgevoerd in OpenCV (ongeveer 1,8 ms). De werking hiervan staat volledig beschreven in hun documentatie, waarbij dit project gebruik maakt van de Probabilistic Hough Line Transform. Dit zorgt voor een lijst van 4 coördinaten (x0,y0,x1,y1) met steeds de x- en y positie van de 2 uitersten gedetecteerde hoeken. Via deze coördinaten kan een lijn getekend worden op de afbeelding via de Line() functie van OpenCV. Figuur 13 toont al deze lijnen op de originele frame.



*Figuur 12 Regen Of Interest*

*Figuur 13 Hough lines*

Hierop zien we dat er verschillende lijnen zijn getekend, waaronder verticale of erg schuine lijnen die helemaal geen onderdeel kunnen uitmaken van de rechte autobaan lijnen op het frame. Om deze lijnen uit te sluiten moeten we de helling berekenen van alle lijnen en wanneer deze niet tussen ±45° en ±90° liggen, moeten ze niet mee in rekening gebracht worden. De helling wordt berekend via de 4 coördinaten : $helling = \frac{(y1-y0)}{(x1-x0)}$ , waarbij hoe negatiever het resultaat hoe groter de opwaartse helling (linkse lijnen van de autobaan). Hetzelfde voor de rechtse lijnen van de autobaan, hoe positiever het getal hoe groter de neerwaartse helling. Nu alle geschikte lijnen overblijven, moet er een keuze gemaakt worden welke lijn voor de linkse en rechtse kant moet overblijven. Gezien de lijnen soms uiterst links of rechts van de autobaan lijnen getekend zijn is de meest accurate manier om het gemiddelde te nemen van alle lijnen per kant. Hierdoor bekomen we onderstaand beeld in figuur 14.

Deze 2 lijnen worden niet lang genoeg getekend doordat de stippellijn uiteraard even uit het beeld verdwijnt door de lege middenruimte en dus de 2 uitersten opnieuw getekend worden maar niet over de gehele autobaan. Deze 2 lijnen moeten dus verlengd worden. Om dit te doen wordt allereerst de genormaliseerde vector genomen via volgende formule:

$$\widehat{V_x} = \frac{x1 - x0}{\sqrt{(x1 - x0)^2 + (y1 - y0)^2}} \;\&\; \widehat{V_y} = \frac{y1 - y0}{\sqrt{(x1 - x0)^2 + (y1 - y0)^2}}$$

Aan de hand van deze vector kan een verlengd coördinaat gecreëerd worden door:

$$\text{Xverlengd} = \text{x} \pm \text{i} * \widehat{V_x} \;\&\; \text{Yverlengd} = \text{y} \pm \text{i} * \widehat{V_y}$$

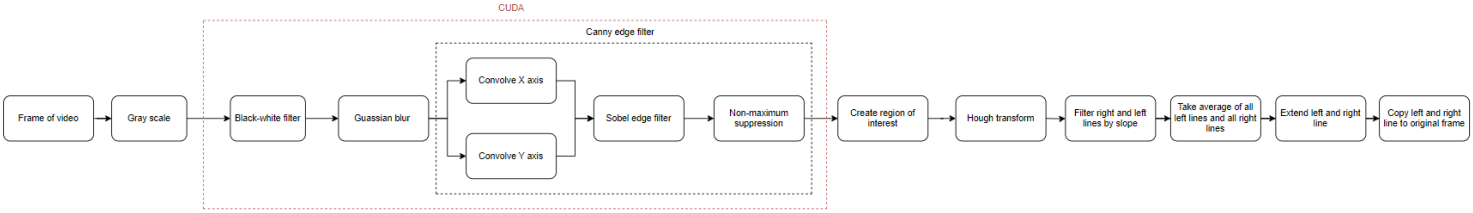
Waarbij ‘i’ de lengte van de extensie bepaald. Deze verlengde lijnen worden weergegeven in figuur 15 en is ook het finaal resultaat dat we nodig hebben.

*Figuur 14 Gemiddelde lijnen*

*Figuur 15 Verlengde lijnen*

## Volledige pipeline

Gezien nu alle stappen doorlopen zijn met de opgeloste problemen doorheen het proces om de lijnen te detecteren op de rijbaan, wordt de volledige pipeline hieronder weergegeven:



# 4. Performantie analyse

Om het performantie verschil te achterhalen tussen de CPU en GPU wordt er getimed hoe lang er over gedaan wordt om zowel de (CUDA) image pipeline (exclusief Hough transformatie) als de gehele pipeline van het project uit te voeren. Deze timing wordt uitgevoerd aan de hand van de time_perf() functie via de time module van Python. Hieruit zijn volgende resultaten bekomen:

Timing CUDA pipeline

- CPU doet er tussen de 18 en 18,8 seconden over
- GPU doet er tussen de 10 en 15 milliseconden over

Bij het nemen van de gemiddelde tijd van de GPU en CPU kan er besloten worden dat de GPU 1471 keer sneller de pipeline uitvoert ten opzichte van de CPU.

Timing volledige pipeline

- CPU doet er tussen de 18,8 en 19 seconden over
- GPU doet er tussen de 17 en 20 milliseconden over

Bij het nemen van de gemiddelde tijd van de GPU en CPU kan er besloten worden dat de GPU 1050 keer sneller het volledige project uitvoert ten opzichte van de CPU.

Om de statistieken van de Jetson te achterhalen wordt de jtop module gebruikt. Hieruit kan bevestigt worden dat de GPU tussen de 25-45% gebruikt wordt om de volledige pipeline uit te voeren. Wat niet eens de helft is van wat hij aankan!

Bij het bekijken van het CPU gebruik via taakbeheer bij het uitvoeren van de volledige pipeline, wordt er ongeveer 25% gebruikt. Dit is logisch gezien de gebruikte CPU 4 cores bevat, waarbij er constant 1 core gebruikt wordt in dit project omdat alles serieel verloopt. Dit omdat de essentie van het project het verschil moet weergeven door parallellisatie. Indien men de volledige CPU wil benutten zodat het sneller uitgevoerd wordt, moeten alle 4 cores aangestuurd worden door het gehele project multithreaded te schrijven.

# 5. Besluit

We zien duidelijk een aanzienlijke performantiewinst bij zowel het uitvoeren van de CUDA pipeline als het gehele project op de GPU. Dit is een verwacht resultaat gezien de CPU niets parallelliseert. Alle pixels op één frame worden via 2 for-loops (voor zowel de x-richting als y-richting) doorlopen per toegepaste filter, dit vraagt heel wat rekenkracht.

Voor grote rekenintensieve toepassingen met ruimte voor parallellisatie is het gebruik van een GPU dus veel performanter.

Maar bij het uitvoeren van kleinere berekeningen is een CPU nog altijd sneller doordat er niets van data gekopieerd moet worden van host (CPU) naar device (GPU) en omgekeerd, wat bij de Jetson wel het geval is. De kleinere data kan op deze manier meteen snel uitgevoerd worden.

Dit verklaart onder andere de performantie daling van de GPU tegenover de CPU wanneer de gehele pipeline wordt uitgevoerd (1050 keer sneller) in plaats van de CUDA pipeline (1471 keer sneller). Hieruit kan er afgeleid worden dat de GPU 1471-1050=421 keer vertraagd wordt bij het uitvoeren van de gehele pipeline. Dit komt voornamelijk door 2 redenen:

1) Het steeds kopiëren van data in de Jetson zorgt voor een zekere overhead (ongeveer 3 milliseconden). Dit zorgt er in het slechtste geval voor dat de CUDA pipeline al voor 20% kan uitgevoerd worden totdat deze overhead is uitgevoerd.

2) De Jetson Nano CPU (Arm Cortex-A78AE) is trager dan de Intel i7-8565U CPU.

   Wanneer er getimed wordt om de mathematische berekeningen voor de helling en de genormaliseerde vectoren te berekenen van de lijnen zijn volgende resultaten waarneembaar:

   - Intel CPU doet er 1,75 tot 2 milliseconden over
   - Jetson Nano CPU doet er 5 tot 6 milliseconden over

   Bij het nemen van de gemiddelde tijd op beide resultaten kan er besloten worden dat de Jetson Nano CPU ongeveer 2.5 tot 3 keer trager is ten opzichte van de Intel CPU. Dit door het verschil in aantal cores, klok frequentie, etc.

Bij dit project zijn de besproken vertragingen natuurlijk verwaarloosbaar door de enorm hoge performantie winst in de CUDA pipeline, waardoor de GPU in dit project er met hoofd en schouders bovenuit steekt.

# 6. Bronnen


https://medium.com/@techreigns/how-does-a-self-driving-car-detect-lanes-on-the-road-fc516c789984

https://www.geeksforgeeks.org/reading-image-opencv-using-python/

https://docs.opencv.org/4.x/dd/d43/tutorial_py_video_display.html

https://www.youtube.com/watch?v=vMZ7tK-RYYc&t=97s

https://www.geeksforgeeks.org/python-opencv-cv2-line-method/?ref=lbp

https://www.youtube.com/watch?v=3dHJ00mAQAY

https://www.youtube.com/watch?v=9bBsvpg-Xlk

https://www.youtube.com/watch?v=BCNnqTFi-Gs

https://stackoverflow.com/questions/56813343/masking-out-a-specific-region-in-opencv-python

https://medium.com/@rohit-krishna/coding-canny-edge-detection-algorithm-from-scratch-in-python-232e1fdceac7

https://en.wikipedia.org/wiki/Sobel_operator

https://indiantechwarrior.com/houghline-method-line-detection-with-opencv/

https://stackoverflow.com/questions/19336778/how-to-extend-a-line-segment-in-python?rq=3