GPGPU Project

Rien Maertens

11 mei 2016

Deel I
Per-pixel Image Filters

Ontwerpkeuzes

De verschillende implementaties van `ImageFilter` hebben een hoop gemeenschappelijke functionaliteit (bijvoorbeeld: RGBA-componenten uit een pixel halen en er terug in steken). Daarom heb ik een paar gemeenschappelijke functies gemaakt die ik in twee categorien heb ingedeeld:

`PixelUtils.h`
: Dit zijn hulpfuncties om pixels te manipuleren. De belangrijkste methodes zijn `getColorData` en `toPixel`. Deze gaan door middel van bitoperaties de RGBA-componenten uit een pixel halen en er terug in steken. De functies hebben het sleutelwoord `restrict(amp,cpu)` in hun declaratie staan zodat ze op de GPU kunnen uitgevoerd worden. In dit bestand is ook de methode `colorDistance` gemplementeerd die de afstand tussen twee kleuren berekent.

`FilterUtils.h`
: Dit zijn functies met betrekking tot het uitvoeren van filter. Mijn eerste idee was om te werken met een abstracte superklasse voor de GPU, voor de CPU en voor iedere verschillende filter. Zo zou de `ImageFilter_Desaturate_GPU` overerven van een GPU-superklasse en een Desaturate-superklasse. Helaas kunnen functies met `restrict(amp)`

niet virtueel zijn of functiepointer aanroepen.
Daarom ben ik overgeschakeld naar het gebruik van lambda's die de manipulaties op de RGBA-componenten voor hun rekening nemen. Deze lambda's worden als argument meegegeven aan de twee functies in dit bestand:

- `process_per_pixel_GPU`:
  Hier wordt de C++AMP `parallel_for_each`-functie gebruikt om de gegeven afbeelding te verwerken door de GPU. De arrays met pixels worden gelezen en bewerkt door middel van `array_view`s.
- `process_per_pixel_CPU`:
  Deze functie doet hetzelfde als hierboven, maar de manipulaties vinden plaats op de CPU. Deze functie heb ik multithreaded gemaakt: er wordt gekeken hoeveel parallelle threads het systeem aan kan en vervolgens wordt het aantal te verwerken pixels gelijk verdeeld over dit aantal threads.

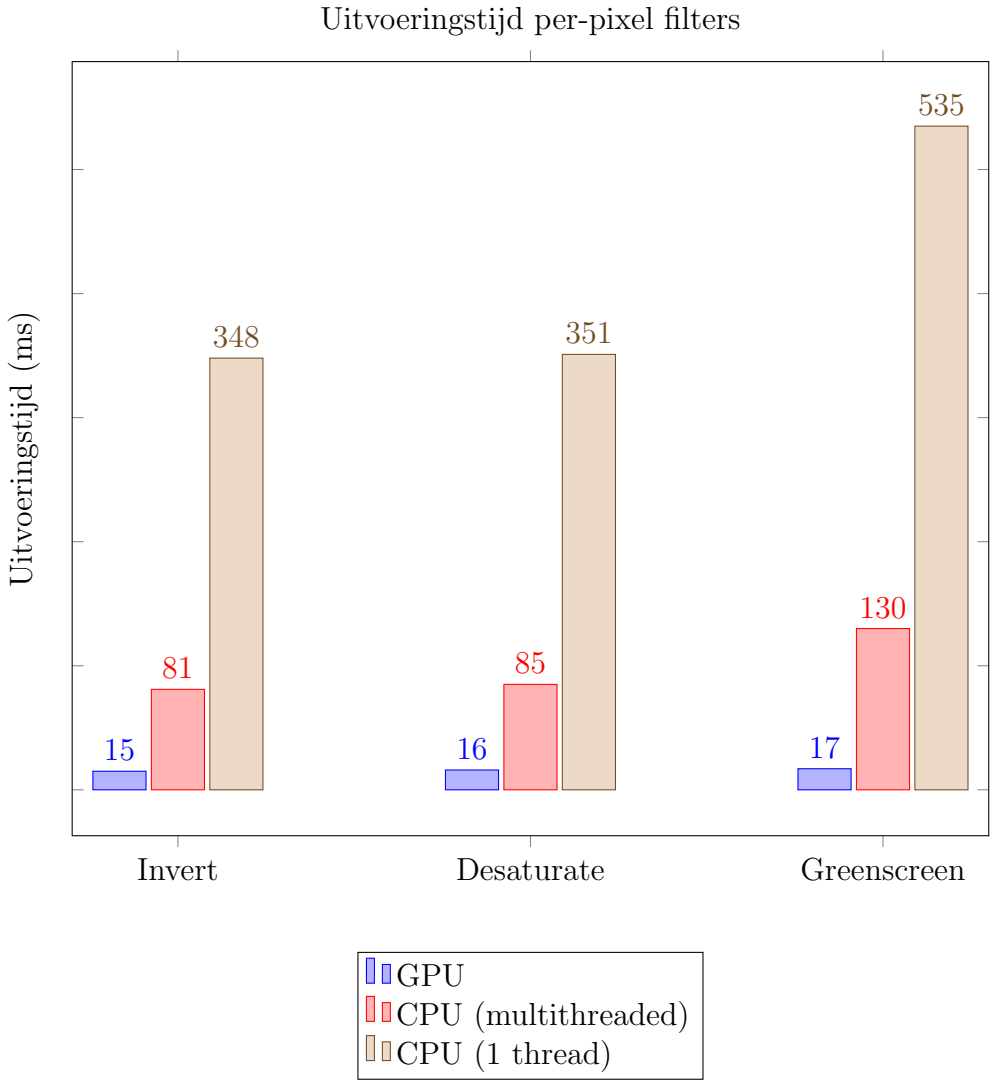
Momenteel wordt exact dezelfde lambda gebruikt voor de CPU als de GPU van een filter, maar heb ik nog geen manier gevonden om deze duplicatie te vermijden. Hier kan er dus eventueel nog een verbetering plaats vinden.

Uitvoeringstijd CPU en GPU

Om de uitvoeringstijd per functie te vergelijken heb ik gekeken naar de logs die naar de console werden geschreven bij het uitvoeren van elke functie. Ik heb opgemerkt dat de eerste functie die op de GPU wordt uitgevoerd significant trager is dan de functies die erna worden uitgevoerd (400 i.p.v 15 milliseconden). Daarom laat ik eerst een filter draaien voor ik de metingen start.

Zoals te zien op figuur 1 is de uitvoeringstijd van de CPU-filters het grootst. Het is logisch dat de multithreaded CPU-filter ongeveer vier keer sneller is dan wanneer alles in serie wordt verwerk: het systeem waarop deze metingen werden uitgevoerd heeft namelijk 4 rekenkernen. We merken op dat de `Greenscreen`-filter het traagst is. De oorzaak hiervan kan zijn dat er meer stappen moeten worden uitgevoerd per pixel.

De GPU-filter is dan weer ongeveer 5 keer sneller dan de parallelle CPU-filter, een enorme snelheidswinst. Onderling is het verschil per filter niet zo groot als bij de CPU-filters. Een mogelijke verklaring hiervan is dat de effectieve verwerkingssnelheid erg klein is in vergelijking met de overhead veroorzaakt door het kopiren van data tussen de CPU en GPU.



Figuur 1: Uitvoeringstijd in milliseconden voor de per-pixel imagefilters.

Deel II
Block-based Image Filters

Ontwerpkeuzes

Voor het tweede deel heb ik twee extra methodes toegevoegd aan de `FilterUtils` die het mogelijk maken om afbeeldingen te verwerken door ze in blokken te splitsen:

- `process_tile_GPU`: Hier wordt gebruik gemaakt van de tiled extents die worden aangeboden door C++AMP. Per tile wordt er per kleurcomponent een array aangemaakt met de
- `process_tile_CPU`: Deze methode gaat eerst multithreaded de kleurcomponenten extraheren uit de afbeelding. Vervolgens wordt van de filter gebruik gemaakt om de nieuwe afbeelding te maken. Deze filter is opnieuw een lambda die wordt meegegeven als argument van de methode.

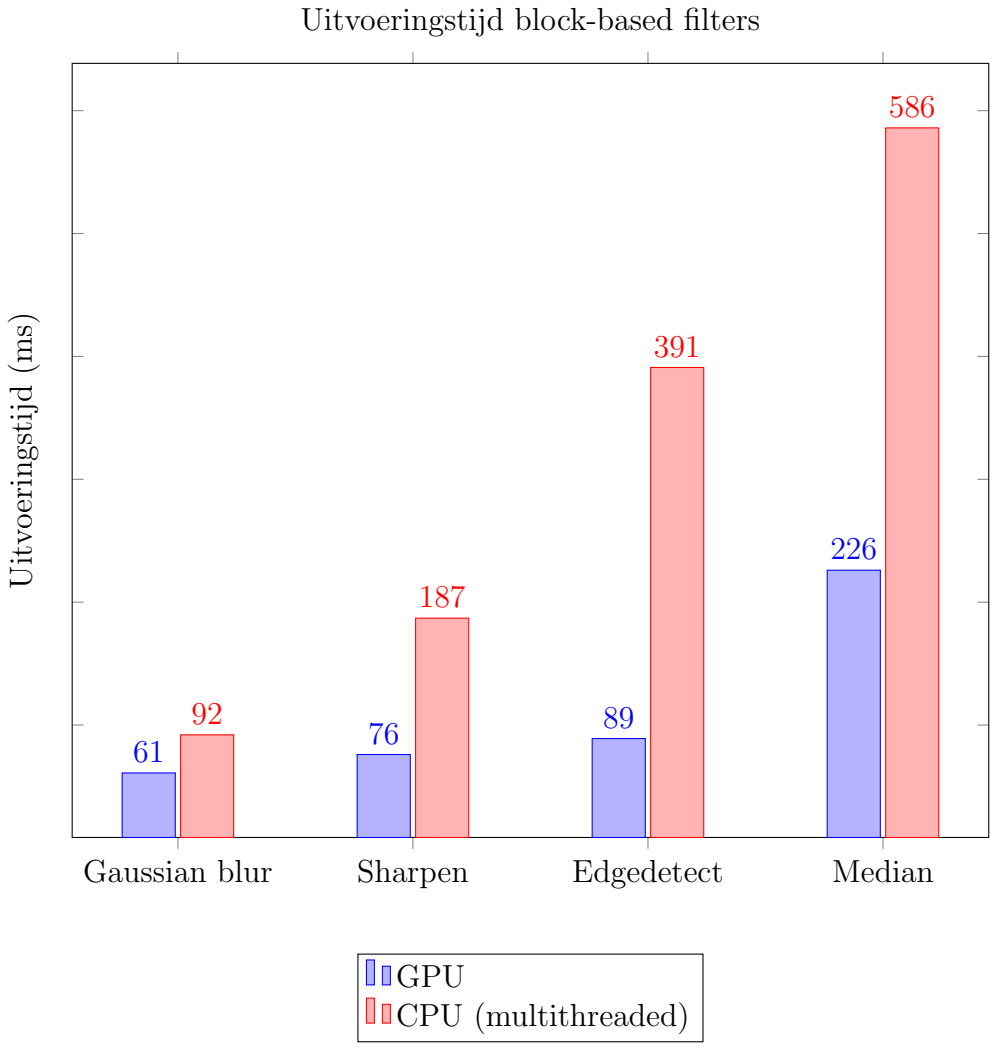
Beide methodes maken gebruik van een nieuwe `WorkBalance`-klasse. Deze klasse is een hulpmethode om werk (bv. het aantal te verwerken pixels) te verdelen over een aantal threads. Deze klasse heeft bovendien een methode `executeParallel` die volgens de berekende verdeling een functie parallel gaat uitvoeren op de CPU. Ik heb de `process_per_pixel_CPU`-methode van in deel I ook aangepast om deze klasse te gebruiken.

De Median-filter werkt door middel van het sorteren van 9 pixels. Voor dit sorteren heb ik gebruik gemaakt van een sorteernetwerk: dit is een vaste opeenvolging van vergelijkingen en (indien nodig) verwisselingen. Ik heb de uitvoeringstijd vergeleken met een insertionsort-implementatie en het sorteernetwerk bleek het snelst te zijn op zowel de GPU als de CPU.

Uitvoeringstijd CPU en GPU

Op figuur 2 is het verschil tussen de GPU- en CPU-implementatie te zien. Hier is de GPU een stuk trager dan bij het vorig deel. Dit is hoogstwaarschijnlijk het gevolg van de vele (dure) geheugenoperaties die plaats vinden.

Bij de Median-filter valt dit het meest op: er wordt gesorteerd, waardoor een veel verplaatsingen in het geheugen plaats vinden. Daarnaast kan het ook zijn dat de tilesize die ik gebruik te klein is om een grote snelheidswinst te hebben.



Figuur 2: Uitvoeringstijd in milliseconden voor de block-based imagefilters.

Deel III
Global Image Filters

Ontwerpkeuzes

Bij dit deel was het niet echt mogelijk om zoals bij de vorige delen te werken met een `process`-methode. Daarnaast heb ik ook maar n opgave gemplementeerd dus had het niet veel nut om dit te doen.
De HistogramEqualization-filter werkt op CPU en GPU ongeveer hetzelfde:

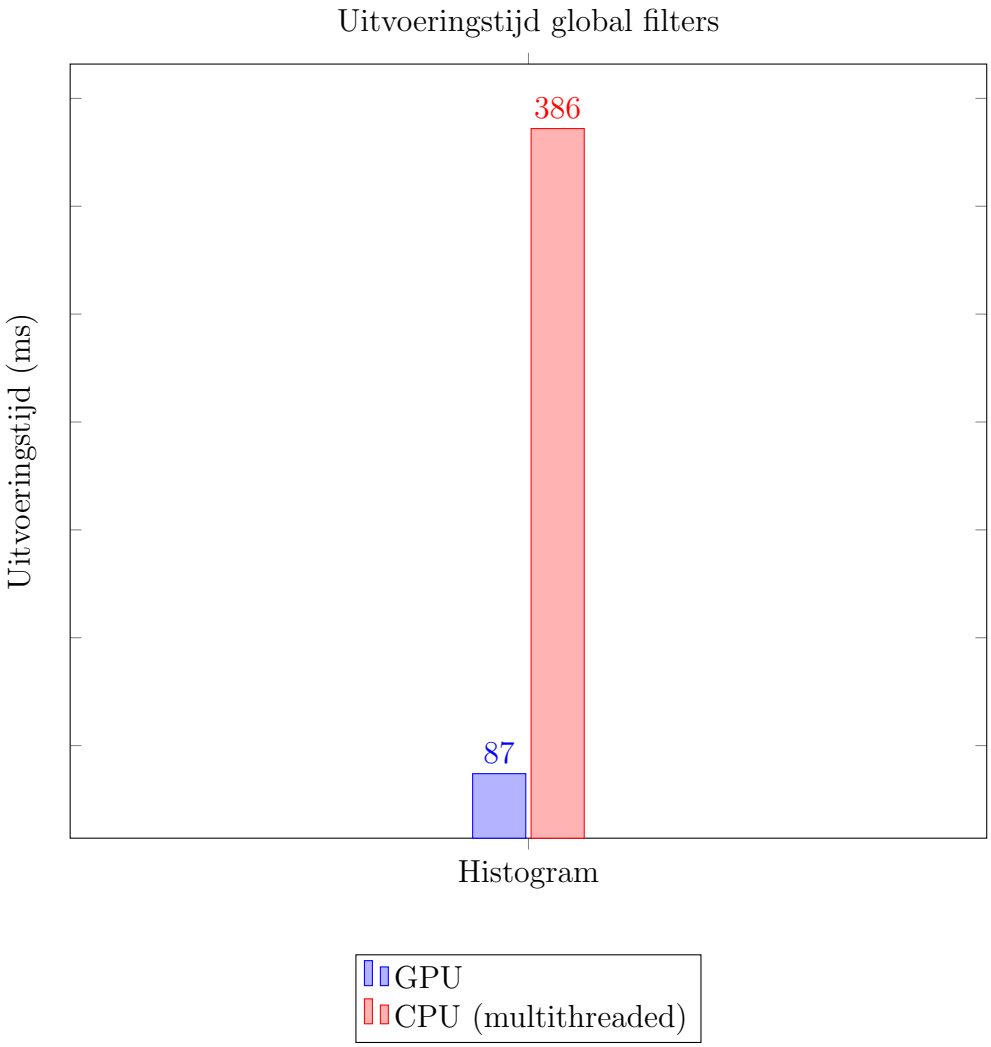
1. Een frequentietabel van alle v-waardes wordt opgesteld.
2. De cumulatieve som wordt bepaald.
3. De nieuwe v-waardes worden berekend.
4. De v-waardes worden vervangen in de afbeelding.

Enkel stap 2, de cumulatieve som bepalen, wordt niet parallel uitgevoerd. Via de `amp_algotithms.h` library heb ik de `scan`-methode even uitgeprobeerd, maar die implementatie was trager dan de serile implementatie. Hoogstwaarschijnlijk is de overhead van het kopiren van data en starten van de threads te groot om een snelheidswinst te verkrijgen. In de CPU- en GPU-implementatie heb ik er dus voor gekozen om dat stuk serieel te houden.

Om de conversie tussen HSV en RGB te kunnen uittesten heb ik ook een GPU-implementatie voor de Hueshift-filter (deel I) gemaakt. Deze conversiemethoden staan in `PixelUtils.h`.

Uitvoeringstijd CPU en GPU

Zoals te zien op figuur 3 is de GPU-implementatie meer dan 4 keer sneller dan de CPU-implementatie. Dit is logisch omdat er, net zoals bij de per-pixel filter, niet veel werk moet gebeuren per pixel. Waar de CPU veel tijd gaat verliezen aan het loopen over alle pixels van de afbeelding kan de GPU veel meer pixels parallel verwerken.



Figuur 3: Uitvoeringstijd in milliseconden voor de globale imagefilter.