n-Queens minimale dominerende verzamelingen

Chessboard Domination on Programmable Graphics Hardware
door Nathan Cournia

Rick van der Zwet
<hvdzwet@liacs.nl>

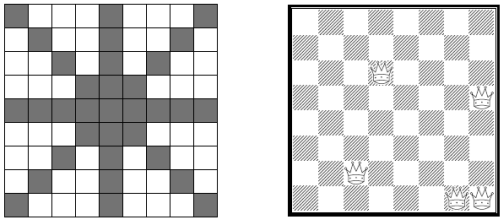
8 september 2010

Samenvatting

Dit schrijven zal het paper van Nathan Cournia *Chessboard Domination on Programmable Graphics Hardware* [CDGPU2006] —en in het bijzonder de gepresenteerde n-Queens oplossing— vrij samenvatten in het Nederlands en zal de mening van de ondergetekende op het geheel geven.

1 Inleiding

Vanwege het feit dat *n-Queens* een algemeen geaccepteerde terminologie is voor het plaatsen van n koninginnen op een $n \times n$ schaakbord zodanig dat de koninginnen elkaar niet kunnen slaan, zal dit in het schrijven niet vertaald worden. Verder zal in de tekst op diverse plekken de originele Engelse



Figuur 1: Links: koningin kan dit patroon slaan. Rechts: ongeldige *n-Queens* oplossing, omdat de koninginnen rechtsonder elkaar kunnen slaan.

bewoording staan om zo terugzoeken in en refereren naar het originele paper [CDGPU2006] makkelijker te maken.

2 Minimale dominerende verzameling

Een dominerende set is een verzameling koninginnen, die tesamen alle velden bereiken. Een minimale dominerende verzameling (*Minimum domination set*) is een opstelling waarbij met zo weinig mogelijk koninginnen elk vakje van het schaakbord a) door ten minste 1 koningin direct bereikt kan worden of b) bezet is door een koningin. Omdat een koningin een *karakteristiek patroon* kan slaan (zie Figuur 1) kan het minimale aantal koninginnen dat nodig is bepaald worden door middel van Formule 1[1]:

$$y(Q_n) \geq \frac{n-1}{2}, n \geq 1 \tag{1}$$

Hierbij is n de hoogte van het bord. De knopen van Q_n zijn genomen van een $n \times n$ bord met n^2 knopen. Als knoop a door middel van het *karakteristiek patroon* van een koningin vanuit b bereikt kan worden, wordt er een tak tussen deze knopen gevormd. Dit geheel (de knopen en takken) is Q_n. De $y(Q_n)$ is dan het minimale aantal koninginnen dat men kan plaatsen om zo een minimale dominerende verzameling te bereiken.

3 Grafische Verwerkings Eenheid

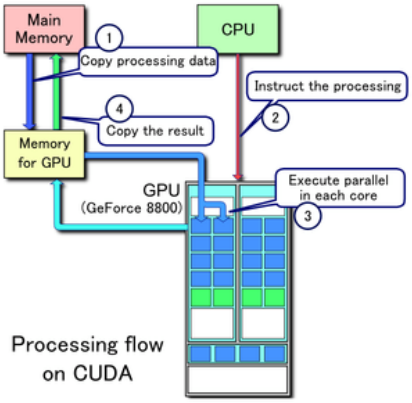
De Grafische Verwerkings Eenheid (*Graphics Processing Unit*, ook bekend als *GPU*) heeft speciale electronica om ervoor te zorgen dat deze snel de *RGB* waardes van alle beeldpunten kan berekenen in complexe beeldsystemen met bijvoorbeeld ingewikkelde (lees: tijdrovende) berekeningen voor schaduw, reflectie en intensiteit. Om deze grote hoeveelheid gegevens te verwerken maakt de *GPU* gebruik van een grote hoeveelheid parallelle processoren, welke alle individueel een deel van de berekeningen op zich nemen.

Recente (elektronica) ontwikkelingen zoals *CUDA*[2] en een meer generieke implementatie *OpenCL*[3] hebben ertoe geleid dat op de *GPU* in plaats van enkel beeldverwerkingen te doen nu ook geprogrammeerd kan worden om specifieke berekeningen uit te voeren. Figuur 2 laat de verschillende stappen zien die uitgevoerd moeten worden om de *GPU* aan te sturen. Het is belangrijk om te beseffen dat de *GPU* een heel simpele processor is en (dus)

[1] Bewezen in [DM2003].

[2] `http://en.wikipedia.org/wiki/CUDA`

[3] `http://en.wikipedia.org/wiki/OpenCL`



Figuur 2: *GPU* werking. De verschillende *GPU* processor-blokken worden *kernels* genoemd. De data wordt getransporteerd door verschillende data-kanalen (*streams*) en heeft de vorm van een *framebuffer*, welke intern als $n \times m$ array gezien kan worden. Een *kernel* kan toegepast worden op een (deel van de) *framebuffer*. Illustratie uit [WPCUDA].

zeer kleine buffers en instructieset heeft. Verder is het ook cruciaal te weten dat de *GPU niet* direct gebruik gebruik kan maken van het hoofd-geheugen van de *CPU*, maar dat de invoer en uitvoer altijd eerst naar/van de *GPU* verplaatst zal moeten worden.

4 Aanpak n-Queens probleem op de *GPU*

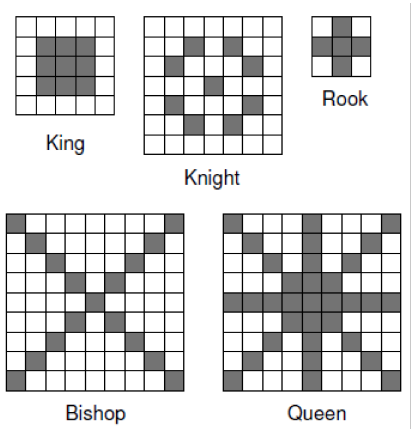
Het zoeken van geldige minimale dominerende verzamelingen is een stevige klus, waarvoor minimaal (in het optimale geval) n koninginnen nodig zijn die we op een $n \times n$ schaakbord kunnen plaatsen. Dat levert $(n*n)! - ((n*n) - n)!$ mogelijkheden op. De *GPU* zal gebruikt worden om oplossingen sneller te controleren en zal niet gebruikt worden om efficiënte oplossingen te vinden. De kracht zit hem in het feit dat meer oplossingen getest kunnen worden en er dus potentieel betere oplossingen tussen kunnen zitten, wat ook te zien is in Algoritme 1. De genereerde potentiële oplossingen die aan de *GPU* ter controle aangeboden worden respecteren eventuele bekende boven- en ondergrenzen [4] zoals Formule 1.

[4] Zie voor meer bekende boven- en ondergrenzen onder andere de papers genoemd in [CDGPU2006][sectie 2, pagina 62].

Algoritme 1 evalueren minimale dominerende set

```
01: klaar=nee
02: doe
03: ..bereken potentieel minimale dominerende verzamelingen
04: ..plaats in framebuffer
05: ..als (alle pixels zijn gemarkeerd) dan
06: ....klaar=ja
08: totdat (klaar=ja)
```

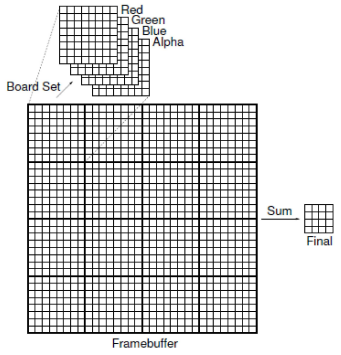
4.1 Detectie algoritme



Figuur 3: Stempels van verschillende schaakstukken, de verschillende groottes zijn noodzakelijk om aan te geven hoe het stempel vergroot of verkleind moet worden.

Om snelle detectie mogelijk te maken, wordt er gebruikt gemaakt van een eigenschap waar een *GPU* in uitblinkt: het "*stempelen*" van objecten in een raster (welke in traditionele beeldbewerking gebruikt wordt om texturen te maken). Elk schaakstuk heeft zijn eigen stempel-patroon zoals te zien in figuur 3. Hierbij moet opgemerkt worden dat de stempels allemaal op hun eigen manier schalen.

Er zijn nog twee eigenschappen van de *GPU* waar dankbaar gebruik van gemaakt wordt, namelijk kleur en het verschil in grootte van het bord en de geaccepteerde invoer. Door slim te combineren —zie Figuur 4 op pagina 5— kan het aantal potentiële oplossingen dat getest kan worden gemaximaliseerd worden.



Figuur 4: Door slim te coderen kunnen meerdere potentiële oplossingen tegelijk bekeken worden. Hier wordt gebruik gemaakt van (a) het feit dat de ruimte groter is dan het "schaakbord" dat bekeken wordt en (b) een pixel gecodeerd is uit vier onafhankelijke kleuren.

De individuele *kernels* volgen Algoritme 2. De test of alle punten gemarkeerd zijn lijkt op het eerste gezicht een lus/loop die test over alle beeldpunten, echter de *GPU* heeft specifieke instructies om dit efficiënter uit te voeren. Voor het plaatsen van de stempels is er een een grafische operatie die equivalent is aan een `OF` operatie. Als een `INVERSE` operatie gebruikt zou worden om de stempel te plaatsen zou een tweede overlappende stempel onterecht als niet geraakt gemarkeerd worden.

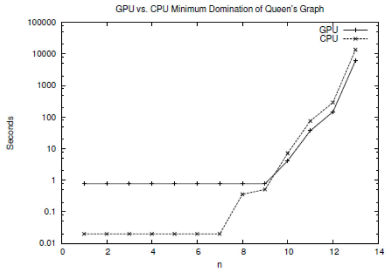
Algoritme 2 evalueren minimale dominerende set door individuele kernel

```
01: voor alle stempels in stempel locatie
02: ..plaats stempel
03: als (alle punten gemarkeerd) dan
04: ..oplossing=ja
```

5 Conclusie

Door het toepassen van de *GPU* op het *n-Queens* probleem kunnen winsten geboekt worden zoals te zien is in Figuur 5[5]. Het toepassen van *GPU* dominerende textuur technieken lijkt voor dit specifieke geval een goede vertaling van de traditionele *CPU* wereld naar een implementatie in de *GPU* wereld.

[5]Gegevens direct uit [CDGPU2006] overgenomen, experiment niet opnieuw uitgevoerd.



Figuur 5: Uitvoertijden (logaritmische schaal) van de *CPU* en *GPU* gebaseerde minimale dominerende implementaties welke $y(Q_n)$ uitrekenen. Hoe groter n wordt des te beter de *GPU* gaat presteren in vergelijking met de *CPU*.

De stempels bieden tevens meer vrijheden om alternatieve "schaakstukken" te onderzoeken.

5.1 Verder werk

Er zal gekeken worden of de generatie van nieuwe oplossingen ook op de *GPU* gedaan kan worden om zo het probleem van de langzame contextwisselingen op te lossen. Verder zal er gekeken worden of de codering van de borden op nog een slimmere manier aangepakt kan worden: in plaats van de kleur in 4 basis-kleuren uit te splitsen zouden ook de volledige 32 bits (elke kleur heeft 8 bits) kunnen worden om nog meer (combinaties) van oplossingen te coderen.

5.2 Discussie

De claim dat de *GPU* "veel" sneller is lijkt niet gefundeerd in de grafieken. Beide lijken erg dicht bij elkaar te blijven en ik zie niet waarom dit plots veel beter zou worden bij grotere n waarden. De "tegel-methode" van Figuur 4 lijkt in theorie leuk, maar als n groter wordt is er grote kans dat de tegels niet meer (goed) passen. Als n groter wordt dan in mogelijke invoer —de maximale grootte van een bord die in in het geheugen van de de *GPU* past is gelimiteerd aan de hoeveelheid geheugen iin de *GPU* en op welke manier dit geheugen ingedeeld is— is het helemaal niet meer mogelijk.

Referenties

[DM2003] E. J. Cockayne, *Chessboard domination problems*, *Discrete Math*, 86:13–20, 1990.

[CDGPU2006] Nathan Cournia, *Chessboard Domination on Programmable Graphics Hardware*, *Proceedings of the 44th annual Southeast Regional Conferencee*, pp. 62–67, 2006.

[WPCUDA] Wikipedia, CUDA, `http://en.wikipedia.org/wiki/CUDA` (as of Sept. 7, 2010, 12:03 GMT).