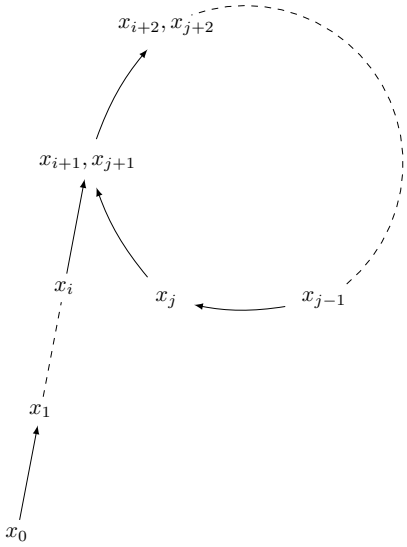
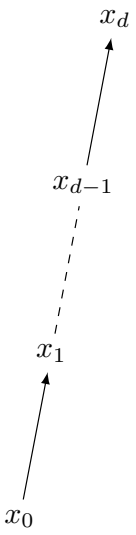
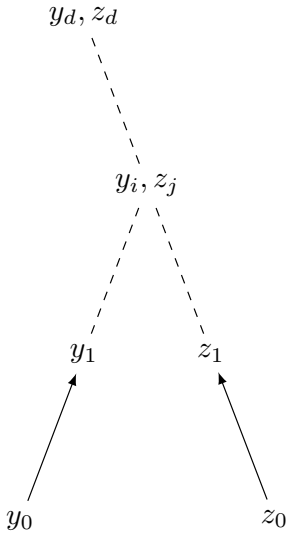
Universiteit Utrecht

Faculteit Bètawetenschappen
Departement Wiskunde
Bachelorscriptie

Pollards Rho algoritme voor het discrete logaritme, geïmplementeerd op de GPU







Auteur:

Thijs van der Horst

Begeleider:

Dr. G. Tel
Department of Information and Computing Sciences

10 juni 2019

Samenvatting

Het discrete logaritme probleem wordt gezien als onmogelijk op te lossen, voor de juiste instanties van het probleem. Dit terwijl de inverse operatie dat niet is. Hierdoor wordt het discrete logaritme probleem veelal gebruikt in de cryptografie. In deze scriptie wordt Pollards Rho algoritme voor het oplossen van discrete logaritme problemen onderzocht en geïmplementeerd voor gebruik met een GPU, met een parallellisatiemethode die lineaire snelheidswinst biedt in het aantal threads.

In sectie 2 wordt allereerst het bestaande algoritme uitgelegd, samen met optimalisaties die het sequentiele algoritme in $O(\sqrt{q})$ tijd en $O(1)$ ruimte laten lopen, waarbij q de orde van de gebruikte generator is. Vervolgens zullen wij in sectie 3 een methode behandelen voor het parallelliseren van dit algoritme, die met genoeg geheugen zorgt voor een lineaire snelheidswinst. Daarna volgt een korte introductie van het OpenCL standaard, dat gebruikt wordt om te kunnen werken met een GPU. Dit gebeurt in sectie 4. Verder bespreken wij in sectie 5 hoe wij kunnen werken met de grote getallen die in de cryptografie gebruikt worden, waarbij vooral wordt gekeken naar methoden voor het uitvoeren van een modulaire vermenigvuldiging. In sectie 6 wordt vervolgens een globaal beeld geschetst van het gemaakte programma, waarna in sectie 7 verscheidene testresultaten worden gegeven en behandeld.

Inhoudsopgave

1 Inleiding **1**

2 Pollards Rho algoritme **1**
2.1 De functie f . . . 2
2.2 Botsingsdetectie zonder alle gevonden punten . . . 2
2.3 Het antwoord verkrijgen uit de cykel . . . 2

3 Het parallelliseren van Pollards Rho algoritme **3**
3.1 Het kiezen van de distinguished point conditie . . . 4
3.2 De waarde van k . . . 5

4 GPGPU en OpenCL **6**

5 Grote getallen op de GPU **7**
5.1 Representatie van de getallen . . . 7
5.2 Modulaire vermenigvuldiging van de getallen . . . 8
5.2.1 Radix-b deling . . . 9
5.3 Montgomery vermenigvuldiging . . . 11
5.3.1 Coarsely Integrated Operand Scanning . . . 13

6 Opzet van het programma **14**
6.1 De opdeling van het programma . . . 14
6.2 Onvoorziene problemen . . . 15
6.2.1 De watchdog timer . . . 15
6.2.2 Global memory tegelijk gebruiken op de CPU en GPU . . . 16

7 Resultaten **16**

References **I**

1 Inleiding

Het beveiligen van informatie is niets nieuws. Het versleuten van belangrijke berichten is een eeuwenoud concept. Maar met de opkomst van het internet wordt het beveiligen van allerlei soorten informatie steeds meer van cruciaal belang. Denk bijvoorbeeld aan bank transacties, inloggegevens of gesprekken over de telefoon. Allerlei informatie waarvan niemand wil dat het zomaar op straat komt te liggen. De cryptografie houdt zich bezig met methoden om dit soort informatie te beveiligen.

In de cryptografie wordt doorgaans gebruik gemaakt van *sleutels*, stukjes informatie waaruit de oorspronkelijke inhoud gehaald kan worden. Doorgaans wordt een algoritme voor sleutelgeneratie beoordeeld op hoe veilig de gegenereerde sleutels zijn, dat is, hoe lastig het is om het oorspronkelijke bericht terug te krijgen uit de sleutel, zonder extra informatie.

Een bekend algoritme voor het genereren van sleutels is het Diffie-Hellman-sleuteluitwisselingsprotocol [DH76]. Dit protocol gebruikt de aanname dat het *discrete logaritme probleem* (DLP) voor de juiste gebruikte groepen onmogelijk op te lossen is. Het DLP is als volgt gedefinieerd:

Definitie 1.0.1 (Het discrete logaritme probleem)**.** Gegeven een groep G, een element $g \in G$ en een element $y \in \langle g \rangle$. Zij $q = |\langle g \rangle|$ de orde van de groep voortgebracht door g. Vind het getal $x \in \{0, 1, \ldots q-1\}$ zodanig dat $g^x = y$.

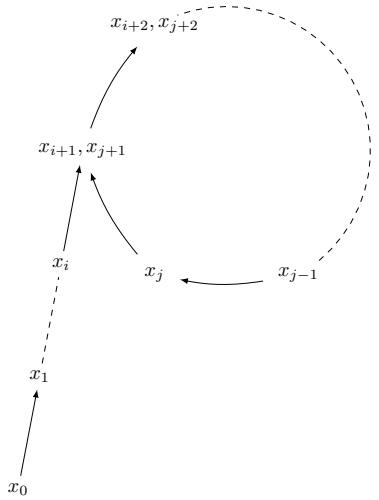
In deze scriptie zullen wij een bekend algoritme voor het oplossen van deze DLP's, *Pollards Rho* algoritme, gaan onderzoeken en implementeren voor de GPU, de *Graphics Processing Unit*. Wij zullen vervolgens deze implementatie gaan vergelijken met het oorspronkelijke Rho algoritme, dat zal worden geimplementeerd op de CPU, de *Central Processing Unit*. In deze scriptie zullen wij werken met de groep $G = \mathbb{Z}_p^*$, voor p een oneven getal, waarbij het element $g \in \mathbb{Z}_p^*$ van priemorde q is.

2 Pollards Rho algoritme

Wij zullen in dit stuk Pollards Rho algoritme beschrijven, aan de hand van het dictaat "Cryptografie, Beveiliging van de digitale maatschappij" [Cry06], tenzij anders aangegeven.

Pollards Rho algoritme is hedendaags een veelgebruikt algoritme voor het oplossen van DLP's. Een reden hiervoor is dat het een zogeheten groepsalgoritme is. Dit wil zeggen dat het algoritme voor iedere groep gebruikt kan worden, in tegenstelling tot bijvoorbeeld het *Function Field Sieve* algoritme [AH99], dat slechts met eindige lichamen $\mathbb{F}_{p^n}$ kan worden gebruikt.

Pollards Rho algoritme is gebaseerd op het genereren van een rij (x_i), met alle elementen $x_i \in \langle g \rangle$. Deze rij wordt ook wel een keten genoemd. De keten wordt gemaakt door een startpunt $x_0 \in \langle g \rangle$ te kiezen. De keten wordt verder voortgezet via een functie $f : \langle g \rangle \to \langle g \rangle$, waarbij $x_{i+1} = f(x_i)$.
Omdat de groep $\langle g \rangle$ een eindig aantal elementen bevat, zullen er een $i, j \in \mathbb{N}$ bestaan waarvoor $x_{i+1} = x_{j+1}$, maar $x_i \neq x_j$. In het punt x_{i+1} vindt dan een *botsing* plaats. Verder geldt dat $x_{i+k} = x_{j+k}$ voor alle $k \in \mathbb{Z}_{\geq 1}$, in andere woorden, de keten loopt in een cykel na stap i. Het doel van het algoritme is om een punt op deze cykel te vinden. Zie voor een visualisatie van deze cykel figuur 1.



Figuur 1: Een visualisatie van de keten (x_i). Er vindt een botsing plaats in punt x_{i+1}, waarna de keten in een cykel terecht komt.

Omdat er een cykel zit in de keten zal de keten na maximaal q iteraties een keer botsen. Er zijn alleen nog een aantal overwegingen die gemaakt moeten worden om met het algoritme DLP's op te kunnen lossen. Zo moet de functie f gekozen worden, moet er een manier bedacht worden om niet ieder punt op te slaan en

moet er bedacht worden hoe het uiteindelijke antwoord op het DLP geconstrueerd kan worden uit de botsing. Hieronder worden opties besproken voor deze overwegingen, die ervoor zorgen dat het algoritme in $O(\sqrt{q})$ tijd en $O(1)$ ruimte DLP's kan oplossen.

2.1 De functie f

Door f slim te kiezen, kunnen wij zorgen dat er naar verwachting al na $O(\sqrt{q})$ stappen een botsing optreedt. Hiervoor is het nodig dat f een voldoende willekeurige functie is. Wanneer dit zo is, zal het verwachte aantal benodigde stappen voordat een botsing optreedt ongeveer gelijk zijn aan $\sqrt{\pi q/2} \approx 1.25\sqrt{q}$ [VW99, pp. 24–25].
Naast dat f voldoende willekeurig moet zijn om de looptijd van het algoritme te beperken, moet f ook goedkoop zijn om uit te rekenen. De functie wordt namelijk in iedere iteratie gebruikt. Een goede constructie voor f blijkt te zijn om een hashfunctie $k : G \to \{1, 2, 3\}$ te gebruiken, die de groep G op een zo goed als willekeurige wijze in drie delen verdeelt. Vervolgens wordt f door middel van de hashfunctie k geconstrueerd als

$$f(x) = \begin{cases} x \circ g & \text{als } k(x) = 1 \\ x \circ y & \text{als } k(x) = 2 \\ x \circ x & \text{als } k(x) = 3 \end{cases}.$$

2.2 Botsingsdetectie zonder alle gevonden punten

Een makkelijke manier om te zoeken naar botsingen is om de keten (x_i) volledig bij te houden en in iedere iteratie de keten te doorzoeken voor een botsing. Dit kost echter $O(\sqrt{q})$ geheugenplekken, wat voor grote q te veel wordt. Pollards Rho algoritme maakt daarom gebruik van een tweede keten (y_i), die in hetzelfde startpunt begint, maar die elke iteratie de functie f tweemaal toegepast krijgt. Er geldt dus dat $x_{2i} = y_i$ voor alle $i \geq 0$.
Stel nu dat er een botsing plaatsvindt in het punt x_{i_0}, waarbij $x_{i_0} = x_{j_0}$ en $x_{i_0-1} \neq x_{j_0-1}$ voor een $j_0 > i_0$. Noem de periodelengte $l = j_0 - i_0$, dan geldt voor iedere $i \geq i_0$ en voor iedere $k \in \mathbb{N}$ dat $x_i = x_{i+kl}$. Zij $i = kl$ nu het kleinste veelvoud van de periodelengte dat ten minste zo groot is als i_0. Dit geeft dat $x_i = x_{i+kl} = x_{2i} = y_i$, wat geeft dat de laatst gevonden elementen in de ketens botsen in iteratie i. Dit laat zien dat alleen de laatst gevonden elementen van de ketens bijgehouden moeten worden om een punt op de cykel te vinden.

2.3 Het antwoord verkrijgen uit de cykel

Om het antwoord op het DLP te verkrijgen uit het gevonden punt, worden de startpunten x_0 en y_0 zo gekozen, dat ze beiden in de vorm $g^a \circ y^b$ staan. Dit kan gemakkelijk door de machten a en b te kiezen, in plaats van de startpunten zelf. Merk op dat, omdat de ketens binnen $\langle g \rangle$ worden gemaakt, ieder punt te schrijven is als product van machten van g en y. Doordat de machten a en b van de startpunten bekend zijn, en doordat er in een iteratie alleen wordt vermenigvuldigd met g, y of het huidige punt zelf, waarvan allemaal de machten a en b bekend zijn, kan van ieder punt op de ketens de machten worden bijgehouden. Als wij deze machten ook nog reduceren modulo q, dan kan ieder punt op de ketens gerepresenteerd worden door een paar a en b uniek aan dat punt.
De functie f wordt vervolgens aangepast tot een functie

$$f : \langle g \rangle \times \mathbb{Z}_q \times \mathbb{Z}_q \to \langle g \rangle \times \mathbb{Z}_q \times \mathbb{Z}_q,$$

zodat deze de bijgehouden machten kan updaten. Deze nieuwe f neemt dan naast het punt x dus nog een macht a en een macht b, en zal er als volgt uitzien:

$$f(x, a, b) = \begin{cases} (x \circ g, a + 1, b) & \text{als } k(x) = 1 \\ (x \circ y, a, b + 1) & \text{als } k(x) = 2 \\ (x \circ x, 2a, 2b) & \text{als } k(x) = 3 \end{cases}.$$

In het geval van een botsing zal dus gelden dat $x_i = y_i$ voor een $i \in \mathbb{Z}_{\geq 1}$. Zij (a_i, b_i) de representatie van x_i en (c_i, d_i) de representatie van y_i. Dan geldt dus dat $g^{a_i} \circ y^{b_i} = g^{c_i} \circ y^{d_i}$, waaruit volgt dat $y =$

$g^{(a_i-c_i)(d_i-b_i)^{-1}}$. Hiermee volgt dat indien (d_i-b_i) een inverse heeft in $\mathbb{Z}_q$, het antwoord op het DLP gelijk is aan $(a_i-c_i)(d_i-b_i)^{-1}$. Indien er geen inverse bestaat voor (d_i-b_i), zal er met een nieuw startpunt begonnen moeten worden, waarna alles herhaald wordt. Aangezien q priem is, zal (d_i-b_i) een inverse hebben in $\mathbb{Z}_q$ zolang $b_i \neq d_i \pmod q$. Dit geeft dat de kans dat er geen antwoord gehaald kan worden uit de botsing gelijk is aan $1/q$, wat verwaarloosbaar is voor grote q.
Het uiteindelijke Pollards Rho algoritme, met al deze overwegingen meegenomen, is nu gegeven in algoritme 1.

Algorithm 1 Pollards Rho

```
1: do
2:     a, b ← random
3:     z ← g^a ∘ y^b
4:     (t, c, d) ← (z, a, b)
5:     do
6:         (z, a, b) ← f(z, a, b)
7:         (t, c, d) ← f(f(t, c, d))
8:     while z ≠ t
9: while (a − c) ∉ Z_q^*
10: return (d − b)(a − c)^{−1}
```

3 Het parallelliseren van Pollards Rho algoritme

Wij zullen in dit hoofdstuk aan de hand van het artikel "Parallel Collision Search with Cryptanalytic Applications" [VW99] beschrijven hoe wij Pollards Rho algoritme parallelliseren in ons programma.

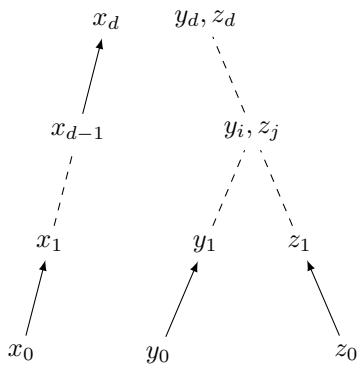
Een voor de hand liggend idee voor het parallelliseren van Pollards Rho algoritme is het uitvoeren van het algoritme op meerdere threads, elk met een ander startpunt. Met deze methode is het erg makkelijk om het algoritme te parallelliseren, aangezien de threads geen enkele communicatie onderling nodig hebben tijdens het algoritme, op eventueel een signaal dat een antwoord is gevonden na. De snelheidswinst van deze methode is echter slechts $O(\sqrt{m})$.

Er bestaat een betere aanpak, dat een lineaire snelheidswinst oplevert ten opzichte van het aantal threads. Deze aanpak gebruikt het idee dat ons doel niet is om een cykel te vinden, maar slechts een botsing. Met deze methode worden er meerdere ketens opgezet, verspreid over de verschillende threads. Het verschil met de eerste methode is echter dat er niet binnen één keten wordt gezocht naar een botsing, maar dat er ook naar botsingen wordt gezocht die tussen ketens plaatsvinden.

Aan het begin wordt er een aantal startpunten berekend, die samen met hun representaties worden bijgehouden. Vervolgens krijgt elke thread een startpunt, waarmee het een keten genereert. Dit keer worden de representaties van de individuele punten echter niet bijgehouden, alleen die van de startpunten. Dit scheelt wat tijd met het updaten, aangezien een hoop threads geen keten zullen maken die botst met een andere keten. De representaties van de botsing zullen later gevonden worden. Om te kijken of er een botsing heeft plaatsgevonden tussen twee ketens, moeten de punten van de ketens worden bijgehouden. Dit kost echter veel geheugen. Vandaar dat er een *distinguished point* conditie wordt gedefinieerd. Deze conditie is een functie $d : \langle g \rangle \to \{0, 1\}$, waarbij punten die worden afgebeeld op 1 *distinguished points* worden genoemd.

Bij ieder gegenereerd punt wordt gekeken of het een distinguished point is. Indien dit zo is, wordt het punt opgeslagen, samen met het startpunt van de keten. Het doel van de distinguished point conditie is om het aantal punten dat wordt opgeslagen te beperken. Dit zorgt er namelijk niet alleen voor dat de hoeveelheid nodige opslag kleiner wordt, maar ook dat de tijd die het kost om te kijken of er een botsing heeft plaatsgevonden wordt verkleind.

Zodra er twee ketens hetzelfde distinguished point zijn tegengekomen, weet het programma dat er een botsing heeft plaatsgevonden tussen de twee ketens. Vervolgens kunnen de twee representaties van het distinguished point berekend worden door de ketens opnieuw te genereren vanuit de bijbehorende startpunten, waarna het antwoord op het DLP gevonden kan worden. Zie voor een visualisatie van deze parallellisatiemethode figuur 2.



Figuur 2: Een visualisatie van de parallelle ketengeneratie met distinguished points. De punten x_d, y_d en z_d zijn allen de eerste distinguished points op de bijbehorende ketens. De ketens (y_i) en (z_i) botsen, waarna zij hetzelfde distinguished point vinden.

3.1 Het kiezen van de distinguished point conditie

Aan het kiezen van de distinguished point conditie zitten nog wat overwegingen vast. Ten eerste moet de conditie makkelijk te evalueren zijn, bij ieder punt wordt er immers gekeken of het een distinguished point is. Verder beïnvloedt de conditie de lengte van de ketens, een strictere conditie zal er namelijk voor zorgen dat het gemiddeld langer duurt om een distinguished point tegen te komen. Overigens moet er ook rekening mee gehouden worden dat er niet al te veel distinguished points gevonden worden. Dit omdat de distinguished points opgeslagen moeten worden, waarvoor slechts beperkte ruimte is. Bovendien zal de overhead van het zoeken naar botsingen groter worden naarmate het aantal gevonden distinguished points groeit.
Als distinguished point conditie zullen wij de functie

$$d_k(x) = \begin{cases} 1 & \text{als } x \bmod 2^k = 0 \\ 0 & \text{anders} \end{cases}$$

gebruiken. Met deze functie wordt een punt x gezien als distinguished zodra de bitrepresentatie van x tenminste k *trailing zeros* bevat.
Door gebruik te maken van de bitrepresentatie van getallen bij de evaluatie van d_k, kunnen wij d_k implementeren via snelle bitoperaties. Hiermee is dus rekening gehouden met de eerste overweging. Het fijne aan deze keuze voor d_k is echter vooral dat de kans dat d_k evalueert naar 1 nauwelijks afhangt van de grootte van de punten. Sterker nog, wij kunnen de kans beschouwen als een constante, gelijk aan 2^{-k}. Zie hiervoor stelling 3.1.1 en gevolg 3.1.2 (beiden eigen werk).

Stelling 3.1.1. *Gegeven een getal $k \in \mathbb{N}$ en een modulus $m \geq 1$. Laat nu $n, r \in \mathbb{N}$ zodat $m = n \cdot 2^k + r$, waarbij $1 \leq r \leq 2^k$. Zij $x \in \{0, 1, \ldots, m-1\}$ willekeurig. Dan is de kans dat $x \bmod 2^k = 0$ gelijk aan*

$$\Pr(x \bmod 2^k = 0) = \frac{n+1}{n \cdot 2^k + r}.$$

Bewijs. Zij $k \in \mathbb{N}$ en een modulus $m \geq 1$ gegeven. Zij nu $n, r \in \mathbb{N}$ zodat $m = n \cdot 2^k + r$, met $1 \leq r \leq 2^k$. Merk op dat $0 \leq i \cdot 2^k \leq m-1$ voor $i = 0, 1, \ldots, n$. Dit zijn de enige elementen in $\{0, 1, \ldots, m-1\}$ die een veelvoud zijn van 2^k. Zij x nu een willekeurig element in $\{0, 1, \ldots, m-1\}$. Dan is de kans dat $x \bmod 2^k = 0$ dus gelijk aan

$$\Pr(x \bmod 2^k = 0) = \frac{n+1}{m} = \frac{n+1}{n \cdot 2^k + r},$$

waarmee de stelling is bewezen. □

Gevolg 3.1.2. Merk op dat, voor $k, n, r \in \mathbb{N}$, $n > 0$ en $1 \leq r \leq 2^k$, geldt dat

$$(1 + \frac{1}{n}) \cdot 2^{-k} > \frac{n+1}{n \cdot 2^k + r} \geq 2^{-k}.$$

Indien $k < \log_2(p)$ kan de gebruikte modulus p geschreven worden als $p = n \cdot 2^k + r$, waarbij $n > 0$ en $1 \leq r \leq 2^k$. Wanneer wij nu aannemen dat de kans op een distinguished point, gegeven de conditie functie d_k als boven, gelijk is aan 2^{-k}, zullen wij deze kans dus met maximaal $1/(n \cdot 2^k)$ onderschatten.
Deze bovengrens op de grootte van onze onderschatting zal dalen naarmate de gebruikte modulus groter wordt, aangezien hierdoor de waarde van n zal stijgen. Hiermee beargumenteren wij dat het geen probleem is om de kans af te beschouwen als de constante waarde 2^{-k}, zolang de gebruikte modulus p groot genoeg is.

3.2 De waarde van k

Met de distinguished point conditie als gedefinieerd in sectie 3.1 hierboven, is het nu de vraag welke waarde van k het beste gebruikt kan worden. Het gebruik van een distinguished point conditie neemt namelijk wel wat overwegingen met zich mee. In dit stuk zullen wij deze overwegingen gaan behandelen.

Het gebruik van een distinguished point conditie verandert niets aan het verwachte aantal nodige stappen voordat er een botsing plaatsvindt, deze is dus nog steeds $\sqrt{\pi q/2}$. Het is echter niet altijd zo dat een botsing plaatsvindt op een distinguished point. In dit geval zal de botsing pas later gedetecteerd worden. Zij θ de kans dat een gevonden punt distinguished is. Bij ons is dus $\theta = 2^{-k}$. Het verwachte aantal extra stappen dat moet worden uitgevoerd voordat twee botsende ketens in een distinguished point belanden is geometrisch verdeeld, met verwachtingswaarde $1/\theta$. Dit is misschien wat vreemd, gezien de verwachte lengte van een keten voordat het in een distinguished point beland dezelfde verdeling volgt en dus ook gelijk is aan $1/\theta$. Een verklaring hiervoor is dat kortere ketens minder vaak botsen dan langere ketens. Deze kortere ketens worden dan wel meegenomen in de gemiddelde ketenlengte, maar niet in het verwachte aantal extra stappen.

Nadat een botsing is gedetecteerd, moeten de bijbehorende ketens nogmaals worden gegenereerd, om de representaties te vinden. De twee ketens kunnen parallel aan elkaar worden gegenereerd, waardoor dit proces gemiddeld $3/(2\theta)$ stappen kost. Hierbij is $3/(2\theta)$ het verwachte maximum van twee geometrische verdelingen met verwachtingswaarden gelijk aan $1/\theta$, zie appendix B in [VW99]. Bij elkaar komt de verwachte looptijd van het algoritme dus uit op

$$\mathrm{E}_t = \frac{\sqrt{\frac{\pi q}{2}}}{m} + \frac{5}{2\theta}$$

iteraties, wanneer uitgevoerd met m threads.

Het geheugengebruik moet natuurlijk ook in gedachten genomen worden bij de bepaling van de kans θ. Het geheugengebruik hangt af van het verwachte aantal nodige stappen voor het algoritme, aangezien dit het aantal gevonden distinguished points bepaald. Het verwachte aantal gevonden distinguished points is gelijk aan

$$\mathrm{E}_d = m \cdot \theta \cdot \left(\mathrm{E}_t - \frac{3}{2\theta}\right) = \theta\sqrt{\frac{\pi q}{2}} + m.$$

De term $3/(2\theta)$ wordt eerst van E_t afgehaald, omdat dit stond voor het hergenereren van twee ketens, waarvan de distinguished points al waren gevonden en opgeslagen.
Per distinguished point zal ook nog het startpunt en een representatiepaar (a, b) opgeslagen moeten worden, dus zal het uiteindelijke geheugengebruik in $O(\mathrm{E}_d)$ liggen. Wij zullen verder alleen kijken naar het aantal opgeslagen distinguished points.

In [KS01, p. 216] wordt θ gelijkgesteld aan $\alpha m/\sqrt{\pi q/2}$, met $0 < \alpha \leq \sqrt{\pi q/2}/m$ een constante, om de looptijd in $O(\sqrt{q}/m)$ te houden. De verwachte looptijd wordt dan

$$\mathrm{E}_t = \left(1 + \frac{5}{2\alpha}\right) \frac{\sqrt{\frac{\pi q}{2}}}{m}$$

iteraties en het verwachte aantal gevonden distinguished points wordt dan

$$\mathrm{E}_d = (1+\alpha)\, m.$$

Zoals is te zien neemt de looptijd van het algoritme af naarmate de waarde van α groeit, echter zal het geheugengebruik dan juist groeien. Er zal dus een afweging gemaakt moeten worden tussen de gewenste looptijd en de aanwezige hoeveelheid geheugen. Zodra een waarde voor α is gekozen, kan de waarde k hieruit gehaald worden, aangezien $2^{-k} = \theta = \alpha m/\sqrt{\pi q/2}$. De waarde van k wordt dan

$$\frac{1}{2}\log_2(q) + \log_2\left(\frac{\sqrt{\frac{\pi}{2}}}{\alpha m}\right),$$

afgerond naar een geheel getal. Zie voor een aantal testresultaten met betrekking tot verschillende waarden van k sectie 7.

4 GPGPU en OpenCL

In dit hoofdstuk zullen wij aan de hand van "OpenCL programming guide" [Mun+11] een uitleg geven over GPGPU en de OpenCL standaard.

GPGPU, of *General-Purpose computing on Graphics Processing Units*, is het process waarbij de GPU wordt gebruikt om bepaalde taken van de CPU over te nemen. GPGPU wordt veelal gebruikt bij problemen die een grote mate aan *data parallellisme* bevatten. Dit zijn problemen waarbij eenzelfde taak over meerdere verschillende datasets moet worden uitgevoerd. Het parallelliseren van Pollards Rho algoritme is een voorbeeld van een probleem met een grote mate van data parallellisme. De ketens worden onafhankelijk van elkaar gemaakt met dezelfde iteratiefunctie, waarbij voor iedere keten een ander startpunt wordt gebruikt. Het ligt dan ook voor de hand om de GPU een groot aantal ketens parallel te laten genereren, waarbij iedere thread zijn eigen keten maakt. In deze sectie zullen wij een korte uitleg geven over de programmeerstandaard die wij zullen gebruiken.

Om de GPU te kunnen gebruiken voor het berekenen van deze ketens, zullen wij een standaard moeten gebruiken waarmee wij taken aan de GPU kunnen doorsturen vanaf de CPU. Onze keuze is OpenCL, of *Open Computing Language*. OpenCL is namelijk een open standaard, wat betekent dat wij niet gebonden zijn aan een bepaald merk GPU, zoals bijvoorbeeld wel het geval zou zijn met NVIDIAs CUDA standaard. OpenCL werkt zelfs wanneer er een ander soort aparaat dan een GPU wordt gebruikt, echter zullen wij het altijd hebben over een GPU.

Met OpenCL zijn er twee soorten apparaten. Er is altijd een *host*, die op de CPU draait, en er zijn één of meerdere *devices*, zoals dus een GPU. Deze devices bestaan uit een aantal *compute units*, die op hun beurt weer een aantal *processing elements* bevatten. Deze processing elements werken met het SIMD patroon, wat staat voor *Single Instruction, Multiple Data*. Dit betekent dat deze processing elements dezelfde instructie uitvoeren, echter op verschillende datasets.

In OpenCL kunnen wij zogeheten *compute kernels* schrijven. Deze worden geschreven in een programmeertaal die is afgeleid van de programmeertaal C, of C++ voor latere versies van OpenCL. Deze kernels kunnen in een wachtrij worden gezet vanaf de host, waarbij wordt aangegeven hoeveel threads de kernel moeten draaien. Deze threads worden ook wel *work-items* genoemd. De work-items kunnen eventueel worden opgesplitst in 2 of 3 dimensies, wat het werken met 2- of 3-dimensionale datasets makkelijker maakt. Ook kan het aantal work-items per *work-group* aangegeven worden. Alleen tussen work-items binnen een work-group kan gesynchroniseerd worden, work-items in verschillende work-groups kunnen niet direct met elkaar communiceren.

Er bestaat ook nog het concept *warp* of *wavefront*. Tegenwoordig bestaat elke warp gewoonlijk uit 32 work-items op NVIDIA GPU's en 64 work-items op AMD GPU's. Work-items in dezelfde warp werken in *lockstep*, wat betekent dat de work-items wachten totdat ieder work-item in de warp klaar is met een instructie, voordat

zij verder gaan naar de volgende instructie. De huidige instructie wordt verder wel parallel uitgevoerd. Deze warps hebben door het werken in lockstep echter wel een klein nadeel:
zodra er aftakkende paden zitten binnen een stuk code kan het zijn dat deze paden allemaal sequentieel worden uitgevoerd, ook al hoeft een individueel work-item slechts een enkel pad uit te voeren. Dit gebeurt wanneer work-items uit een warp verschillende paden in gaan. Aangezien executie in lockstep gebeurt, kunnen de work-items niet tegelijkertijd verschillende paden uitvoeren, omdat de instructie dan niet hetzelfde is. In dit geval zullen de paden achter elkaar worden uitgevoerd, waarbij work-items die het huidige pad niet nemen tijdelijk worden uitgeschakeld. Het loont dus om zo weinig mogelijk aftakkende paden te maken in het programma, of om in ieder geval deze aftakkingen zo klein mogelijk te houden. Indien alle work-items in een warp hetzelfde pad nemen is er ook geen probleem, alleen dat pad wordt dan afgelegd.

Als laatste nog iets over het geheugen in OpenCL. OpenCL heeft vier soorten geheugen:

- **Private memory**: stukken Private memory horen bij een work-item. Deze stukken zijn dan ook alleen toegankelijk vanaf dit work-item. Er is weinig Private memory beschikbaar per work-item, maar het geheugen zit wel het dichtst bij. Hierdoor is Private memory over het algemeen het snelste type geheugen.
- **Local memory**: stukken Local memory horen bij een work-group. Deze stukken worden gedeeld tussen alle work-items in de work-group. Local memory wordt dan ook vaak gebruikt wanneer data gedeeld moet worden binnen een work-group, maar niet erbuiten.
- **Global memory**: dit geheugen is beschikbaar voor zowel de host als alle work-items, en zowel de host als de work-items kunnen lezen en schrijven in het geheugen. Er is altijd veel Global memory op een GPU vergeleken met de andere types, maar het geheugen zit wel ver van de compute units af. Hierdoor is Global memory het traagste om te gebruiken van alle soorten geheugen. Echter, vanwege de grote hoeveelheid die aanwezig is en omdat het de enige manier is om data terug te sturen naar de host, wordt Global memory veel gebruikt voor de invoer en uitvoer van data.
- **Constant memory**: Constant memory is ongeveer hetzelfde als Global memory, aangezien het een deel van de Global memory ruimte gebruikt. Het verschil is dat Constant memory constant blijft gedurende de uitvoering van een kernel, en alleen de host kan schrijven naar dit geheugen. De host en alle work-items mogen nog wel lezen van dit geheugen.

Data kan niet rechtstreeks tussen de host en de devices verstuurd worden. OpenCL moet de data eerst in het DRAM geheugen zetten, een process dat voor veel overhead zorgt. Het is dan ook een goed idee om communicatie tussen de host en de devices zoveel mogelijk te beperken.

5 Grote getallen op de GPU

In de cryptografie wordt veel gewerkt met getallen die groter zijn dan wat standaard wordt ondersteund. Zo wordt hedendaags door het NIST, het National Institute of Standards and Technology, aanbevolen om met sleutels van 2048 bits te werken voor algoritmen die ervan uitgaan dat het DLP niet oplosbaar is. Voor veel programmeertalen bestaan library's bestemd voor deze grote getallen. Zo bestaat voor C en C++ de GNU MP library, en zullen wij voor ons programma de BigInteger klasse van C# gaan gebruiken. Voor de GPU zijn er echter nog niet zulke openbare library's, wij moeten dus zelf al het nodige implementeren. In dit hooftstuk wordt besproken hoe wij met grote getallen om zullen gaan op de GPU.

5.1 Representatie van de getallen

Om met getallen groter dan wat standaard wordt ondersteund te kunnen werken op de GPU, zullen wij de getallen representeren als polynomen. Een getal x kan zo, gegeven een *radix* b, gerepresenteerd worden als het polynoom

$$x = \sum_{i=0}^{m} x_i b^i,$$

voor een $m \in \mathbb{N}$. De coëfficiënten van het polynoom worden dan de *cijfers* of *woorden* van x genoemd. Indien x geschreven kan worden met m radix-b cijfers, noemen wij x ook wel een m-cijferig radix-b getal. Als notatie hiervoor gebruiken wij $x = (x_{m-1}, \ldots, x_1, x_0)_b$. Hierbij mogen de cijfers gelijk zijn aan 0. Het getal 34 kan bijvoorbeeld geschreven worden als $(3, 4)_{10}$, maar ook als $(0, 3, 4)_{10}$, waarbij het respectievelijk 2 en 3 radix-10 cijfers heeft.

Op de GPU zijn de *registers* tegenwoordig vaak 32 bits groot. Dat wil zeggen dat getallen van maximaal 32 bits in grootte in een enkel register kunnen worden opgeslagen. Vandaar dat wij voor ons programma zullen werken met radix $b = 2^{32}$, dit geeft ieder woord 32 bits. Aangezien wij in $\mathbb{Z}_p^*$ werken, kunnen wij overigens volledig met de standaardrepresentaties werken, ofwel getallen in $\{0, 1, \ldots, p-1\}$. Hierdoor hoeven wij alleen met niet-negatieve getallen te werken, wat ervoor zorgt dat het teken niet hoeft worden opgeslagen.

Het makkelijkst is om het aantal woorden van de getallen niet onderling te laten variëren, maar om de getallen een vaste lengte te geven. Aangezien wij alleen met getallen in $\{0, 1, \ldots p-1\}$ zullen werken, kunnen wij deze lengte vastzetten op het minimum aantal woorden dat nodig is om de modulus te representeren. Dit aantal is gelijk aan $\lceil m/32 \rceil$, waarbij $m = \lfloor \log_2(p) \rfloor + 1$ de grootte van de bitrepresentatie van p is. Wij zullen ieder getal x dus representeren als

$$x = \sum_{i=0}^{\lceil m/32 \rceil - 1} x_i \cdot 2^{32i},$$

waarbij de coëfficiënten x_i in een array van *unsigned int* datatypes worden opgeslagen.

5.2 Modulaire vermenigvuldiging van de getallen

Een operatie die iedere iteratie wordt gedaan is modulaire vermenigvuldiging. De twee getallen die vermenigvuldigd worden staan beiden in arrays opgeslagen, waarvoor de GPU geen standaard vermenigvuldigings- of modulaire delings-operator kent. Hier zijn echter wel algoritmen voor, die wij aan de hand van "GPUMP: a Multiple-Precision Integer Library for GPUs" [ZC10] zullen behandelen, tenzij anders aangegeven. Af en toe wordt in de algoritmen de notatie (u, v) gebruikt. Gegeven een radix b wordt met deze notatie het getal $u \cdot b + v$ bedoeld.

Een voor de hand liggende aanpak van het implementeren van deze operatie is om het vermenigvuldigen en modulo rekenen te scheiden. Voor het vermenigvuldigen kan vervolgens bijvoorbeeld de *Schoolboek* methode worden gebruikt. Deze methode heeft veel weg van het vermenigvuldigen van twee eindige sommaties. De methode loopt een van de operanden woord voor woord door, waarbij het elk woord met de hele andere operand vermenigvuldigd en het resultaat voldoende "opschuift". Het algoritme is te vinden als algoritme 2. Het Schoolboek algoritme, wanneer gebruikt met twee m-cijferige getallen, heeft een looptijd van $O(m^2)$. Er bestaan betere algoritmen voor vermenigvuldiging, echter zijn deze niet heel geschikt voor gebruik op de GPU. Zo heeft Karatsubas methode een looptijd van $O(m^{\log_2(3)}) \approx O(m^{1,58})$ [Gro+05], alleen is het een recursief algoritme, iets wat op de GPU niet goed zal werken. Vandaar dat wij de Schoolboek methode zullen gebruiken.

Algorithm 2 Schoolboek methode voor vermenigvuldiging

Input: Twee m- respectievelijk n-cijferige, niet-negatieve getallen $x = (x_{m-1}, \ldots, x_1, x_0)_b$ en $y = (y_{n-1}, \ldots, y_1, y_0)_b$.
Output: Het $(m+n)$-cijferige product $z = x \cdot y = (z_{m+n-1}, \ldots, z_1, z_0)_b$.

```
1:  z ← 0
2:  for i from 0 to n − 1 do
3:      u ← 0
4:      for j from 0 to m − 1 do
5:          (u, v) ← x_j · y_i + z_{i+j} + u
6:          z_{i+j} ← v
7:      end for
8:      z_{m+i} ← u
9:  end for
10: return z
```

5.2.1 Radix-b deling

Voor de modulaire deling kunnen wij een algoritme gebruiken dat een staartdeling uitvoert. Hiervoor zijn echter nog een aantal operaties nodig voor grote getallen, namelijk aftrekking en vergelijking. Deze worden hieronder gegeven in respectievelijk algoritmen 4 en 5. Voor de volledigheid en omdat het algoritme later ook gebruikt zal worden, wordt hiernaast nog een algoritme voor het optellen van grote getallen gegeven, zie algoritme 3.

Algorithm 3 Radix-b optelling

Input: Twee m-cijferige, niet-negatieve getallen $x = (x_{m-1}, \ldots, x_1, x_0)_b$ en $y = (y_{m-1}, \ldots, y_1, y_0)_b$.
Output: De $(m+1)$-cijferige som $z = x + y = (z_m, \ldots, z_1, z_0)_b$.

1: $c \leftarrow 0$
2: **for** i **from** 0 **to** $m-1$ **do**
3: $\quad (c, z_i) \leftarrow x_i + y_i + c$
4: **end for**
5: $z_m \leftarrow c$
6: **return** z

Algorithm 4 Radix-b aftrekking

Input: Twee m-cijferige, niet-negatieve getallen $x = (x_{m-1}, \ldots, x_1, x_0)_b$ en $y = (y_{m-1}, \ldots, y_1, y_0)_b$, waarbij $x \geq y$.
Output: Het m-cijferige getal $z = x - y = (z_{m-1}, \ldots, z_1, z_0)_b$.

1: $c \leftarrow 0$
2: **for** i **from** 0 **to** $m-1$ **do**
3: $\quad z_i \leftarrow (x_i - y_i + c) \mod b$
4: $\quad$ **if** $x_i + c \geq y_i$ **then**
5: $\quad\quad c \leftarrow 0$
6: $\quad$ **else**
7: $\quad\quad c \leftarrow -1$
8: $\quad$ **end if**
9: **end for**
10: **return** z

Algorithm 5 Radix-b vergelijking

Input: Twee m-cijferige, niet-negatieve getallen $x = (x_{m-1}, \ldots, x_1, x_0)_b$ en $y = (y_{m-1}, \ldots, y_1, y_0)_b$.
Output: Het getal 1, 0 of -1, wanneer $x > y$, $x = y$ of $x < y$ respectievelijk.

1: $i \leftarrow m-1$
2: **while** $x_i = y_i$ **and** $i > 0$ **do**
3: $\quad i \leftarrow i - 1$
4: **end while**
5: **if** $x_i > y_i$ **then**
6: $\quad$ **return** 1
7: **else if** $x_i = y_i$ **then**
8: $\quad$ **return** 0
9: **else**
10: $\quad$ **return** -1
11: **end if**

Met deze operaties gedefinieerd voor grote getallen kan nu een algorite voor staartdelingen worden opgesteld. Het algoritme in 6 geeft naast het quotiënt van een deling ook de rest van deze deling. Voor een modulaire vermenigvuldiging wordt alleen de rest van deze deling gebruikt.

Algorithm 6 Radix-b staartdeling

Input: Twee m- respectievelijk n- cijferige, niet-negatieve getallen $x = (x_{m-1}, \ldots, x_1, x_0)_b$ en $y = (y_{n-1}, \ldots, y_1, y_0)_b$, waarbij $m \geq n \geq 1$ en $y_{n-1} \neq 0$.
Output: Het $(m - n + 1)$-cijferige quotiënt $q = \lfloor x/y \rfloor = (q_{m-n}, \ldots, q_1, q_0)_b$ en de n-cijferige rest $r = x \bmod y = (r_{n-1}, \ldots, r_1, r_0)_b$.

1: $q \leftarrow 0$
2: **for** i **from** $m - n$ **down to** 0 **do**
3: **while** $x \geq y \cdot b^i$ **do**
4: $q_i \leftarrow q_i + 1$
5: $x \leftarrow x - y \cdot b^i$
6: **end while**
7: **end for**
8: $r \leftarrow x$
9: **return** q, r

Het is goed te zien dat het algoritme een staartdeling uitvoert. In iedere stap i van de for-loop worden de n meest significante cijfers van x, gezien als een getal, gedeeld door y. Noem dit resultaat z. Vervolgens wordt x veranderd naar $x - yz$ en wordt q_i gezet op z. Dit algoritme is erg eenvoudig en dus makkelijk te implementeren, echter kan het wel wat sneller gemaakt worden. Zie voor een sneller delingsalgoritme algoritme 7.

Dit nieuwe algoritme gebruikt voor de eerste stap nog dezelfde techniek als het staartdelingsalgoritme, echter zodra de significantste n cijfers van x niet meer door y gedeeld kunnen worden verandert het algoritme. Het algoritme voert nog steeds een soort staartdeling uit, alleen werkt het voornamelijk met radix-b cijfers in plaats van getallen. Dit is fijn, omdat operaties op grote getallen trager zijn dan dezelfde operaties op cijfers van deze getallen. Zo worden in het staartdelingsalgoritme, in het slechtste geval, in alle $m - n + 1$ iteraties $b - 1$ vergelijkingen, $2(b - 1)$ verschuivingen (vermenigvuldigingen met de radix b) en $b - 1$ aftrekkingen gedaan met een radix-b getal. Dit is niet echt een probleem voor ons decimale systeem met $b = 10$, maar omdat wij werken met $b = 2^{32}$ wordt dit toch echt te traag. Het nieuwe delingsalgoritme gebruikt, in het slechtste geval, slechts 1 vergelijking, 2 verschuivingen, 1 aftrekking en 1 optelling met een radix-b getal, voor $m - n - 1$ iteraties.

Voorbeeld 5.2.1. Om het algoritme uit te leggen volgt nu een voorbeeld. In het voorbeeld zullen wij werken met het decimale getallensysteem, ofwel $b = 10$. Laat $x = 5227$ en $y = 18$, waarbij wij x als 4-cijferig en y als 2-cijferig getal beschouwen.

Het algoritme begint met het vinden van het meest significante cijfer van q. Deze stap gaat net zo als een staartdeling, waarbij de $m - n = 2$ significantste cijfers van x worden gedeeld door y. Dit geeft als resultaat $\lfloor 52/18 \rfloor = 2$. Vervolgens wordt q_2 gezet op 2 en wordt x gelijk gesteld aan $5227 - 2 \cdot 18 \cdot 10^2 = 1627$. Dit gebeurt allemaal in regels 2 tot en met 5.

In de for-loop die volgt, in regels 7 tot en met 14, wordt q_{i-n} afgeschat op het grootste cijfer waarvoor $q_i \cdot Y_2 \leq X_3$, waarbij X_3 bestaat uit de 3 significantste cijfers van x en Y_2 bestaat uit de 2 significantste cijfers van y. In ons geval is $X_3 = 162$ en $Y_2 = 18$, waarmee $q_{m-1} = q_3$ op 9 wordt afgeschat. In regel 15 wordt x vervolgens geüpdate als gewoonlijk, dus x wordt gezet op $1627 - 9 \cdot 18 \cdot 10^1 = 7$.

Vervolgens kan het zijn dat x nu negatief is. Dit omdat er niet naar de volledige waarde van y was gekeken, maar slechts naar de laatste 2 cijfers. Dit is alleen het geval wanneer q_i met 1 is overschat. Dit wordt

Algorithm 7 Radix-b deling

Input: Twee m- respectievelijk n- cijferige, niet-negatieve getallen $x = (x_{m-1}, \ldots, x_1, x_0)_b$ en $y = (y_{n-1}, \ldots, y_1, y_0)_b$, waarbij $m \geq n \geq 2$ en $y_{n-1} > 0$.
Output: Het $(m-n+1)$-cijferige quotiënt $q = \lfloor x/y \rfloor = (q_{m-n}, \ldots, q_1, q_0)_b$ en de n-cijferige rest $r = x \bmod y = (r_{n-1}, \ldots, r_1, r_0)_b$.

1: $q \leftarrow 0$
2: **while** $x \geq y \cdot b^{m-n}$ **do**
3: $\quad q_{m-n} \leftarrow q_{m-n} + 1$
4: $\quad x \leftarrow x - y \cdot b^{m-n}$
5: **end while**
6: **for** i **from** $m-1$ **down to** n **do**
7: $\quad$ **if** $x_i = y_{n-1}$ **then**
8: $\quad\quad q_{i-n} \leftarrow b - 1$
9: $\quad$ **else**
10: $\quad\quad q_{i-n} \leftarrow \lfloor (x_i \cdot b + x_{i-1})/y_{n-1} \rfloor$
11: $\quad$ **end if**
12: $\quad$ **while** $q_{i-n} \cdot (y_{n-1} \cdot b + y_{n-2}) > x_i \cdot b^2 + x_{i-1} \cdot b + x_{i-2}$ **do**
13: $\quad\quad q_{i-n} \leftarrow q_{i-n} - 1$
14: $\quad$ **end while**
15: $\quad x \leftarrow x - q_{i-n} \cdot y \cdot b^{i-n}$
16: $\quad$ **if** $x < 0$ **then**
17: $\quad\quad x \leftarrow x + y \cdot b^{i-n}$
18: $\quad\quad q_{i-n} \leftarrow q_{i-n} - 1$
19: $\quad$ **end if**
20: **end for**
21: $r \leftarrow x$
22: **return** q, r

gecorrigeerd in regels 16 tot en met 19. In ons geval is x gelijk aan 7, dus is er geen correctie nodig.

Na de for-loop staat het quotiënt in q en de rest in x. In ons voorbeeld is er in de for-loop na de eerste iteratie niks veranderd, waarmee q op 290 en r op 7 staat, en inderdaad, $290 \cdot 18 + 7 = 5227$. △

Hoewel dit nieuwe delingsalgoritme een grote verbetering is ten opzichte van de simpele staartdeling, is het niet geschikt voor gebruik op een GPU. Zo staan er een aantal if-statements en een while-loop, die in iedere iteratie van de for-loop doorlopen zullen worden. Dit geeft een hoop branching, wat op de GPU juist vermeden dient te worden. In de sectie die volgt zullen wij een alternatieve manier van modulaire vermenigvuldiging behandelen, die wel goed op de GPU zal werken.

5.3 Montgomery vermenigvuldiging

In dit stuk behandelen wij een andere manier van modulair vermenigvuldigen. Dit doen wij aan de hand van "Analyzing and Comparing Montgomery Multiplication Algorithms" [KAK96], tenzij anders aangegeven.

Een andere manier van modulaire vermenigvuldiging is door *Montgomery vermenigvuldiging* te gebruiken. Dit algoritme gebruikt het *Montgomery reduction* algoritme [Mon85] om de modulaire deling na vermenigvuldiging uit te voeren. In het algoritme wordt alleen gedeeld door en modulo gerekend met een getal R, waarbij R ieder geheel getal mag zijn dat copriem is aan de modulus p en groter is dan p. In ons programma kunnen wij dus R als een macht van de radix 2^{32} kiezen, aangezien p een oneven getal is. Hiermee worden delingen door R slechts verschuivingen van cijfers en wordt een deling modulo R slechts het "weggooien" van een aantal cijfers. In dit stuk zullen wij uitleggen hoe deze manier van vermenigvuldigen werkt.

Allereerst definieren wij het p-residu van een getal $x \in \{0, 1, \ldots, p-1\}$:

Definitie 5.3.1 (p-residu)**.** Gegeven een modulus p en een geheel getal R waarvoor geldt dat $\text{ggd}(R, p) = 1$ en $R > p$. Het p-residu $\bar{x}$ van een getal $x \in \{0, 1, \ldots, p-1\}$ is dan $\bar{x} = xR \mod p$.

Doordat R copriem is aan p, bestaat er een getal R^{-1} waarvoor geldt dat $RR^{-1} \mod p = 1$. Er kan dus vanuit een p-residu $\bar{x}$ terug naar het oorspronkelijke getal x gerekend worden door dit p-residu met R^{-1} te vermenigvuldigen, modulo p.

Het optellen en aftrekken van twee p-residuen $\bar{x}$ en $\bar{y}$ kan volgens de standaard modulaire optelling en aftrekking. Immers is $\bar{x} + \bar{y} = xR + yR = (x+y)R \equiv \overline{x+y} \pmod p$ en $\bar{x} - \bar{y} = xR - yR = (x-y)R \equiv \overline{x-y} \pmod p$. Vermenigvuldiging heeft echter een extra stap nodig. Het product $\bar{x} \cdot \bar{y}$ is namelijk gelijk aan $xR \cdot yR = xyR^2 \equiv \overline{xyR} \pmod p$. Om op het juiste resultaat te komen moet er dus nog met R^{-1} worden vermenigvuldigd.
Montgomery vermenigvuldiging voert deze volledige vermenigvuldiging uit. Het rekent dus voor twee p-residuen $\bar{x}$ en $\bar{y}$ het product $\bar{x} \cdot \bar{y} \cdot R^{-1} \mod p$ uit. Hiervoor heeft het algoritme wel het getal p' nodig, waarvoor geldt dat $RR^{-1} - pp' = 1$. Dit getal, samen met het (ongebruikte) getal R^{-1}, kan gevonden worden via het *Extended Euclides* algoritme [Cor+09, p. 937], gegeven in algoritme 8. Dit hoeft slechts één keer gedaan te worden, zolang er met dezelfde modulus p wordt gerekend. De reden dat niet direct met R^{-1} wordt vermenigvuldigd, waardoor R^{-1} onnodig is voor het algoritme, is om de deling modulo p te vermijden. Het Montgomery vermenigvuldigingsalgoritme is gegeven in algoritme 9.

Algorithm 8 Extended Euclides

Input: Twee niet-negatieve gehele getallen x, y.
Output: De grootste gemeenschappelijke deler $d = \text{ggd}(x, y)$ en een paar getallen a, b waarvoor geldt dat $ax + by = d$.

1: **if** y $= 0$ **then**
2: **return** $(x, 1, 0)$
3: **else**
4: $(d', a', b') =$ Extended Euclides$(y, x \mod y)$
5: $(d, a, b) = (d', b', a' - \lfloor x/y \rfloor \cdot b')$
6: **return** (d, a, b)
7: **end if**

Algorithm 9 Montgomery vermenigvuldiging

Input: Een modulus p, een getal $R > p$ met $\text{ggd}(R, p) = 1$, twee p-residuen $\bar{x}$ en $\bar{y}$, en het getal p' waarvoor geldt dat $RR^{-1} - pp' = 1$.
Output: Het p-residu $\bar{z} = \bar{x} \cdot \bar{y} \cdot R^{-1} \mod p$, waarbij $z = xy \mod p$.

1: $u \leftarrow \bar{x} \cdot \bar{y}$
2: $v \leftarrow ((u \mod R) \cdot p') \mod R$
3: $\bar{z} \leftarrow (u + v \cdot p)/R$
4: **if** $\bar{z} \geq p$ **then**
5: $\bar{z} \leftarrow \bar{z} - p$
6: **end if**
7: **return** $\bar{z}$

Het Montgomery vermenigvuldigingsalgoritme werkt door een specifiek gekozen veelvoud van de modulus p bij het product $\bar{x} \cdot \bar{y}$ op te tellen, waardoor het resultaat een veelvoud wordt van R. Hierdoor kan door R gedeeld worden, wat met een goed gekozen R erg snel zal zijn. Na deze deling zal het getal kleiner zijn dan

$2p$, wat betekent dat er in het ergste geval alleen nog een aftrekking van de modulus moet plaatsvinden. Hieronder volgt een bewijs dat laat zien dat het werkt. Het bewijs is afgeleid uit [Mon85].

Stelling 5.3.2. *Zij p een gegeven modulus en $R > p$ een gekozen getal met $ggd(R, p) = 1$. Zij nu $x, y \in \{0, 1, \ldots, p-1\}$, met $\bar{x}$ en $\bar{y}$ de p-residuen van respectievelijk x en y. Het Montgomery vermenigvuldiginsalgoritme berekend $\bar{z} = \bar{x} \cdot \bar{y} \cdot R^{-1} \mod p$, waarbij $\bar{z}$ het p-residu is van $z = xy$.*

Bewijs. Zij $u = \bar{x} \cdot \bar{y}$. Laat nu $v = ((u \mod R) \cdot p') \mod R$ en $w = (u + v \cdot p)/R$. Eerst laten wij zien dat w een geheel getal is.

Zie hiervoor dat $u + v \cdot p \equiv u + (((u \mod R) \cdot p') \mod R) \cdot p \equiv u + u \cdot pp' \pmod R$.
Omdat $p' \equiv -p \pmod R$ volgt hiermee dat $u + v \cdot p \equiv u - u \equiv 0 \pmod R$. Dit laat zien dat $u + v \cdot p$ deelbaar is door R, dus $w = (u + v \cdot p)/R$ is een geheel getal.

Nu zullen wij aantonen dat het resultaat van het algoritme congruent is aan $\bar{z} = \bar{x} \cdot \bar{y} \cdot R^{-1} \mod p$. Hiervoor gebruiken wij dat $wR = u + v \cdot p$, dus $w \equiv uR^{-1} \pmod p$. Overigens geldt hiermee ook $w - p \equiv uR^{-1} \pmod p$, dus beide mogelijke uitkomsten van het algoritme zijn congruent aan $uR^{-1} \equiv \bar{x} \cdot \bar{y} \cdot R^{-1} \equiv \bar{z} \pmod p$.

Het enige wat nu nog bewezen moet worden is dat het resultaat van het algoritme in het interval $[0, p-1]$ ligt. Merk hiervoor allereerst op dat $u = \bar{x} \cdot \bar{y} < p^2 < pR$ en dat $v < R$. Hiermee volgt dat $u + v \cdot p < pR + pR = 2pR$, waaruit volgt dat $w = (u + v \cdot p)/R < 2p$. Omdat er in de laatste stap p van w afgetrokken wordt, indien $w \geq p$, geldt hiermee dat het resultaat kleiner is dan p. Het is duidelijk dat het resultaat niet negatief kan zijn, wat laat zien dat het resultaat inderdaad in het interval $[0, p-1]$ ligt.

Het algoritme berekent dus een geheel getal, in het interval $[0, p-1]$, wat congruent is aan $\bar{x} \cdot \bar{y} \cdot R^{-1}$, modulo p. Dit laat zien dat het algoritme inderdaad $\bar{z}$ berekent, met $z = xy$. □

Montgomery vermenigvuldiging zelf is erg snel bij een goede keuze van R. Echter kost het omzetten naar de p-residuen nog steeds een deling modulo p. Montgomery vermenigvuldiging wordt dan ook niet aanbevolen wanneer er maar een enkele vermenigvuldiging moet worden uitgevoerd. Des te meer vermenigvuldigingen uitgevoerd moeten worden achter elkaar, des te minder de *overhead* wordt van het omrekenen naar de p-residuen. Aangezien de p-residuen uniek zijn voor ieder element in $\{0, 1, \ldots, p-1\}$, kunnen wij in ons programma echter volledig met deze p-residuen werken, waardoor p-residuen en Montgomery vermenigvuldiging een goede uitkomst bieden voor de modulaire vermenigvuldiging.

5.3.1 Coarsely Integrated Operand Scanning

Er zijn veel algoritmen die Montgomery vermenigvuldiging implementeren voor grotere getallen. Een van de snelste algoritmen is de Coarsely Integrated Operand Scanning (CIOS) methode. Deze methode verweeft de reductiestappen met de vermenigvuldigings stappen. Dit geeft naast de snelheid als voordeel dat deze methode, gegeven twee m-cijferige operanden, slechts $m + 3$ extra geheugenplekken nodig heeft. Dit is een grote verbetering ten opzichte van andere methoden waarbij er apart wordt vermenigvuldigd en gereduceerd, aangezien het product van twee m-cijferige getallen een $2m$-cijferig getal wordt. Het CIOS algoritme is gegeven in algoritme 10. In het algoritme wordt af en toe de notatie (u, v) gebruikt. Gegeven een radix b staat dit voor $(u, v) = u \cdot b + v$.

Hieronder volgt een korte uitleg van het CIOS algoritme, aan de hand van [LG14, p. 221]. De CIOS methode is ongeveer hetzelfde als Montgomery vermenigvuldiging, echter gebruikt het wat kleinere stappen, als het ware.

Het CIOS algoritme bestaat uit een for-loop met daarbinnen twee andere for-loops. De buitenste for-loop gaat over de cijfers van $\bar{y}$. Regels 4 tot en met 7 zorgen vervolgens voor de vermenigvuldiging van $\bar{x}$ met het huidige cijfer $\bar{y}_i$. Vervolgens wordt in regel 11 een cijfer q van $\bar{x} \cdot \bar{y} \cdot p' \mod R$ berekend.
In regels 12 tot en met 16 wordt dit cijfer vervolgens gebruikt om een veelvoud van p bij het huidige resultaat op te tellen. Het minst significante cijfer van het huidige resultaat zal hierdoor gelijk zijn aan 0, waarna het getal wordt opgeschoven om dit cijfer weg te werken. Deze verschuiving gebeurt door het getal dat bestemd was voor plek j, te schrijven naar plek $j - 1$.

Algorithm 10 Coarsely Integrated Operand Scanning

Input: Twee m-cijferige p-residuen $\bar{x} = (\bar{x}_{m-1}, \ldots, \bar{x}_1, \bar{x}_0)_b$ en $\bar{y} = (\bar{y}_{m-1}, \ldots, \bar{y}_1, \bar{y}_0)_b$ en de constante $p'_0 = -p^{-1} \mod b$
Output: Het m-cijferige p-residu $\bar{z} = (\bar{z}_{m-1}, \ldots, \bar{z}_1, \bar{z}_0)_b$, waarbij $z = xy \mod p$.

$\bar{z} \leftarrow 0$
for i **from** 0 **to** $m-1$ **do**
    $u \leftarrow 0$
    for j **from** 0 **to** $m-1$ **do**
        $(u, v) \leftarrow \bar{x}_j \cdot \bar{y}_i + \bar{z}_j + u$
        $\bar{z}_j \leftarrow v$
    end for
    $(u, v) \leftarrow \bar{z}_m + u$
    $\bar{z}_m \leftarrow v$
    $\bar{z}_{m+1} \leftarrow u$
    $q \leftarrow \bar{z}_0 \cdot p'_0 \mod b$
    $(u, v) \leftarrow \bar{z}_0 + p_0 \cdot q$
    for j **from** 1 **to** $m-1$ **do**
        $(u, v) \leftarrow p_j \cdot q + \bar{z}_j + u$
        $\bar{z}_{j-1} \leftarrow v$
    end for
    $(u, v) \leftarrow \bar{z}_m + u$
    $\bar{z}_{m-1} \leftarrow v$
    $\bar{z}_m \leftarrow \bar{z}_{m+1} + u$
end for
if $\bar{z} \geq p$ **then**
    $\bar{z} \leftarrow \bar{z} - p$
end if
return $\bar{z}$

Na de buitenste for-loop zal het resultaat, net als bij het oorspronkelijke Montgomery vermenigvuldigingsalgoritme, tussen 0 en $2p - 1$ liggen, waardoor er dus eventueel nog een aftrekking van p moet plaatsvinden.

6 Opzet van het programma

In dit hoofdstuk beschrijven wij de uiteindelijke structuur van ons programma, samen met een aantal problemen die tijdens het implementeren van het programma zijn opgedoken. Voor de geïnteresseerden is het programma overigens online te vinden op GitLab, via de volgende link: `https://git.science.uu.nl/t.w.j.vanderhorst/parallel-pollard-rho`.

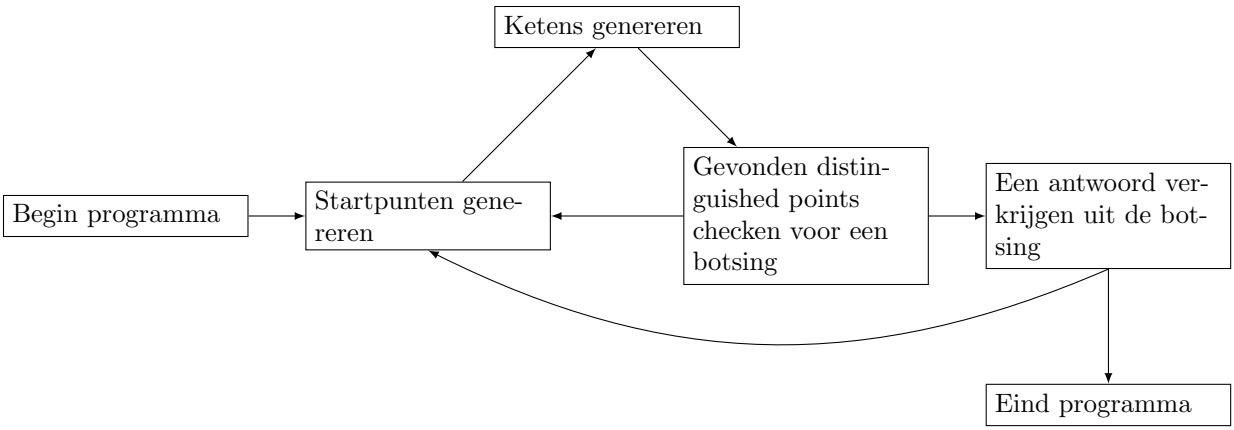
6.1 De opdeling van het programma

Het programma bestaat in feite uit vijf delen, namelijk:

1. De startpunt generator, een deel dat startpunten genereert. Dit deel zal altijd een aantal startpunten klaar hebben liggen die gebruikt kunnen worden. Zodra er startpunten uit deze voorraad worden gehaald, zullen er nieuwe startpunten worden gegenereerd om de voorraad weer bij te vullen. De startpunt generator houdt ook bij welke startpunten zijn gecreëerd, samen met hun representaties.

2. Een deel dat de ketens construeert. Dit deel krijgt startpunten binnen en stuurt paren van distinguished points en hun bijbehorende startpunten terug. Dit is het deel dat op de GPU geïmplementeerd is.

3. Een deel dat gebruikt kan worden om ketens opnieuw te construeren, samen met het berekenen van de uiteindelijke oplossing voor het DLP.



Figuur 3: De opzet van het programma.

4. De distinguished point verzamelaar, een deel dat distinguished points ophaalt en kijkt of er een botsing heeft opgetreden. Dit deel houdt bij welke distinguished points er al gevonden zijn, samen met de startpunten waarmee deze distinguished points zijn gevonden. In het geval van een botsing zal het proberen een antwoord op het DLP te vinden via deel 3.

5. Een deel dat al deze voorgaande delen aanstuurt. Dit deel zal zorgen voor de interactie met de gebruiker en het draaiend houden van deel 2. Ook zal dit deel checken of er nieuwe startpunten nodig zijn en of het nodig is om de gevonden distinguished points op te halen. In deze gevallen zal het de startpunt generator en distinguished point verzamelaar gebruiken om te zorgen dat het programma kan blijven doorgaan.

Zie voor een beeld van hoe het programma in elkaar steekt figuur 3. Het programma loopt eigenlijk in een lus. Het genereert steeds startpunten en daaruit ketens, totdat het distinguished points vindt. Daarna wordt er gekeken of een botsing heeft plaatsgevonden, waarna het een antwoord uit die botsing probeert te halen. Zolang er geen antwoord bekend is, zal het programma opnieuw deze stappen doorlopen.

6.2 Onvoorziene problemen

Tijdens het implementeren van het programma zijn er een aantal onverwachte problemen opgedoken. In deze sectie zullen wij deze problemen benoemen, samen met de manier waarop ze zijn opgelost.

6.2.1 De watchdog timer

Het voornaamste onverwachte probleem was dat OpenCL na een aantal seconden met de GPU gewerkt te hebben een error gaf. Afhankelijk van hoeveel iteraties wij de GPU lieten doen, dook deze error op of niet. Dit bleek te liggen aan de *watchdog* timer van Windows, die ervoor zorgt dat programma's die dezelfde GPU gebruiken als het display niet teveel tijd van de GPU gebruiken. Het volgende citaat komt uit de CUDA FAQ op de website van NVIDIA:

"**Q: What is the maximum kernel execution time?**
On Windows, individual GPU program launches have a maximum run time of around 5 seconds. Exceeding this time limit usually will cause a launch failure reported through the CUDA driver or the CUDA runtime, but in some cases can hang the entire machine, requiring a hard reset.
This is caused by the Windows "watchdog" timer that causes programs using the primary graphics adapter to time out if they run longer than the maximum allowed time. [...]" [NVI].

Het programma moest dus aangepast worden zodat de kernels niet langer dan een aantal seconden draaiden. Hiervoor was het nodig dat de huidige gevonden punten aan het eind van een kernel in Global memory gezet konden worden, zodat ze aan het begin van de volgende kernel weer opgehaald konden worden. Global

memory is namelijk, naast Constant memory, het enige soort geheugen dat niet geleegd wordt wanneer de kernel klaar is. Met deze aanpassing moest er verder alleen nog voor gezorgd worden dat de kernels steeds weer opgestart werden.

6.2.2 Global memory tegelijk gebruiken op de CPU en GPU

Een ander probleem dat opdook had te maken met het continu moeten opsturen van nieuwe startpunten naar de GPU, samen met het ophalen van de gevonden distinguished points van de GPU. Deze operaties zouden namelijk moeten gebeuren terwijl de GPU druk bezig is met het maken van ketens. Het kopiëren van data tussen de CPU en GPU, terwijl delen van deze data gebruikt worden, zorgde voor veel onvoorspelbaar gedrag. Dit onvoorspelbare gedrag was meestal de data die verkeerd aankwam, maar het kwam ook voor dat OpenCL een error gaf die het hele programma stopte. Dit probleem kon echter makkelijk worden opgelost door het te combineren met het probleem van de watchdog timer. Het programma is namelijk aangepast zodat de CPU alleen het stuk Global memory gebruikt tussen de kernels door. Hierdoor is er geen risico dat de CPU en GPU tegelijkertijd bij een stuk geheugen moeten.

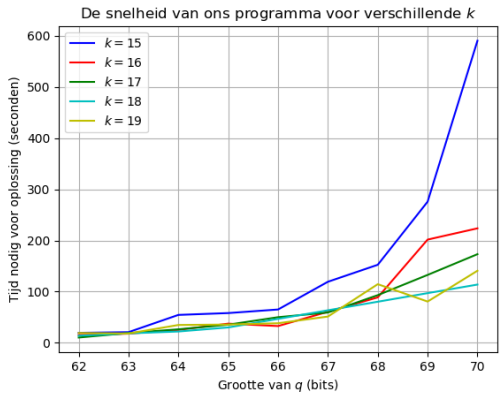
7 Resultaten

Hieronder volgen alle resultaten van uitgevoerde tests. Bij het testen was p altijd een oneven priemgetal. Het programma werkt zolang p oneven is, maar door p als priemgetal te kiezen is de orde van $\mathbb{Z}_p^*$ bekend. Dit maakt het vinden van elementen met priemorde een stuk gemakkelijker, omdat wij weten dat de orde van ieder element een deler is van $p - 1$.

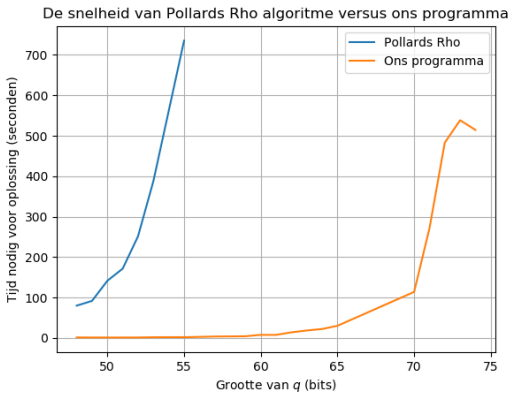
De machine waarop de tests gedraaid zijn bestaat uit de voldende onderdelen:

- **CPU**: een Intel Core i7-4770K, met een base clock van 3,5GHz en een boost clock van 3,9GHz;
- **GPU**: een NVIDIA GTX 1080, met een base clock van 1607MHz en een boost clock van 1733MHz;
- **Geheugen**: 16 GB DDR3 RAM geheugen, met een snelheid van 1600MHz.

Het programma gebruikt veel willekeurige elementen, net als Pollards Rho algoritme. Om hier toch nog goed mee te kunnen testen werd iedere test 10 keer herhaald. Bovendien werden er, per groep, steeds 5 problemen gegenereerd, waardoor iedere groep die getest werd in totaal 50 keer getest is. Hieronder volgen de resultaten.



Figuur 4: De snelheid van ons programma voor verschillende waarden van k. Het programma werd getest met 2048 GPU threads.



Figuur 5: De snelheid van Pollards Rho algoritme vergelijken met dat van ons programma, voor verschillende groottes van q. Ons programma werd getest met de waarde $k = 18$ en 2048 GPU threads.

Allereerst hebben wij uitgezocht wat de beste waarde voor k zou zijn voor ons programma. Herinner dat wij een distinguished point conditie gebruiken die de kans dat een punt distinguished is gelijk maakt aan 2^{-k}. Zie voor de resultaten figuur 4.
Hoewel er verwacht werd dat een kleinere k beter was voor de snelheid van het programma, ten koste van het geheugengebruik, is het voor ons programma beter om een iets hogere waarde voor k te kiezen, zeg 18. De reden dat een kleinere k niet altijd tot betere prestaties leidt, heeft waarschijnlijk te maken met de overhead van het ophalen van de distinguished points.
De distinguished points moeten zo nu en dan van de GPU worden opgehaald. Dit gebeurt slechts wanneer het aantal gevonden distinguished points een bepaalde grens heeft bereikt, omdat het versturen van data vanaf de GPU terug naar de CPU veel overhead geeft. Een kleinere waarde van k zorgt ervoor dat deze grens sneller is overschreden, waardoor er vaker data verstuurd moet worden tussen de CPU en GPU, wat voor meer overhead zorgt. Hierdoor zal voor kleine waarden van k een grotere k nog voor snelheidswinst zorgen.

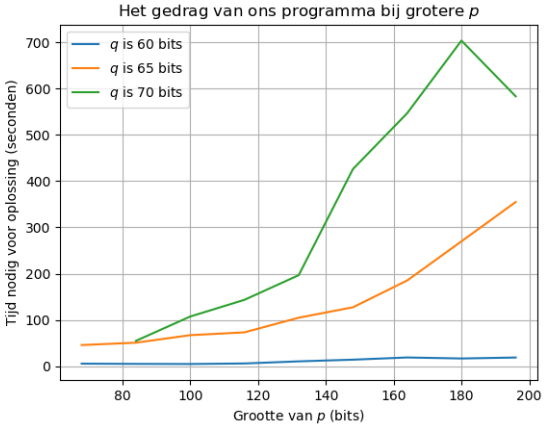
In figuur 5 staat de tijd die Pollards Rho algoritme nodig heeft voor verschillende DLPs, uitgezet tegenover de tijd die ons programma nodig heeft voor deze DLPs. Hierbij gebruikt ons programma de waarde $k = 18$ en worden 2048 GPU threads gebruikt.
De test gebruikte waarden van q tussen de 48 en 74 bits, waarbij Pollards Rho alleen voor waarden van q tot en met 55 bits getest is. Voor iedere waarde van q werd een modulus gebruikt van ongeveer 5 keer zo groot gebruikt.
Zoals is te zien in de figuur doet ons programma ongeveer even lang over een DLP als Pollards Rho algoritme, wanneer de gebruikte q uit ongeveer 19 bits meer bestaat dan de gebruikte q voor Pollards Rho. Dit komt overeen met de verwachte lineaire snelheidswinst, zoals wij nu zullen laten zien.

Zij q de orde voor de gebruikte generator bij Pollards Rho algoritme. Laat de orde van de gebruikte generator bij ons programma dan ongeveer $q \cdot 2^{19}$ zijn. De verwachte looptijd van ons programma is dan

$$\frac{\sqrt{\frac{\pi q \cdot 2^{19}}{2}}}{2048} + 5 \cdot 2^{17} = \frac{\sqrt{\frac{\pi q}{2}} \cdot 2^9 \sqrt{2}}{2^{11}} + 5 \cdot 2^{17} = \frac{\sqrt{\frac{\pi q}{2}}}{2\sqrt{2}} + 5 \cdot 2^{17}$$

iteraties. Ons programma zou dus bijna 3 keer sneller moeten zijn dan Pollards Rho algoritme, bij deze groottes van q. Er zijn een aantal redenen waarom deze snelheidswinst er niet is.
Zo is een GPU over het algemeen langzamer dan een CPU als het gaat om een enkele thread. Dit is ook te zien in onze test setup, waarbij de kloksnelheid van de CPU ruim het dubbele is dan die van de GPU.
Bovendien is er een hoop overhead door met de GPU te werken. Zo moeten kernels continu opgestart worden en moet geheugen steeds tussen de CPU en GPU verstuurd worden.
Als laatste zit er tussen de gebruikte q voor Pollards Rho en die voor ons programma een verschil tussen de grootte, in radix-2^{32} representatie. Dit zorgt ervoor dat alle operaties net iets langer nodig hebben in ons programma, wat met een klein aantal operaties niet te merken zal zijn, maar met het aantal uitgevoerde iteraties in de tests toch wel significant is geweest.
Hiermee beargumenteren wij dat het resultaat binnen de verwachting ligt van een lineaire snelheidswinst.

Als laatste testten wij de prestaties van het programma bij verschillende groottes van de modulus p. Een grotere modulus betekent namelijk dat de getallen in meer woorden worden opgeslagen, waardoor operaties als vermenigvuldigen langer zullen duren. Hier is geen rekening mee gehouden voor de verwachte looptijd van het programma, die slechts kijkt naar het aantal nodige iteraties. Wij testten het gedrag met drie verschillende groottes voor q en wederom met $k = 18$ en 2048 GPU threads. De resultaten zijn te zien in figuur 6.
Uit de resultaten lijkt het dat een grotere p meer impact heeft wanneer ook een grotere q wordt gebruikt. Dit is waarschijnlijk omdat een kleinere q ervoor zorgt dat er minder iteraties nodig zijn, dus ook minder operaties op grote getallen.



Figuur 6: De snelheid van ons programma voor verschillende waarden van p. Het programma werd getest met een q van 65 bits, $k = 18$ en 2048 GPU threads.

De looptijd van het programma bij een grotere p lijkt lichtelijk kwadratisch of exponentieel te stijgen, zeker naarmate de waarde van q groeit. Naar ons idee groeit de tijd kwadratisch in p, aangezien de meest gebruikte operatie, namelijk Montgomery vermenigvuldiging, ook kwadratisch is in p.

Referenties

[AH99] Leonard M Adleman en Ming-Deh A Huang. „Function field sieve method for discrete logarithms over finite fields". In: *Information and Computation* 151.1-2 (1999), p. 5–16.

[Cor+09] Thomas H Cormen e.a. *Introduction to algorithms*. MIT press, 2009.

[Cry06] Tel G Cryptografie. „Beveiliging van de digitale maatschappij". In: *Instituut voor Informatica en Informatiekunde, Universiteit Utrecht* (2006), p. 305–307.

[DH76] Whitfield Diffie en Martin Hellman. „New directions in cryptography". In: *IEEE transactions on Information Theory* 22.6 (1976), p. 644–654.

[Gro+05] Johann Großschädl e.a. „Energy-efficient software implementation of long integer modular arithmetic". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2005, p. 7–8.

[KAK96] C Kaya Koc, Tolga Acar en Burton S Kaliski. „Analyzing and comparing Montgomery multiplication algorithms". In: *IEEE micro* 16.3 (1996), p. 26–33.

[KS01] Fabian Kuhn en René Struik. „Random walks revisited: Extensions of Pollard's rho algorithm for computing multiple discrete logarithms". In: *International Workshop on Selected Areas in Cryptography*. Springer. 2001, p. 212–229.

[LG14] Zhe Liu en Johann Großschädl. „New speed records for Montgomery modular multiplication on 8-bit AVR microcontrollers". In: *International Conference on Cryptology in Africa*. Springer. 2014, p. 215–234.

[Mon85] Peter L Montgomery. „Modular multiplication without trial division". In: *Mathematics of computation* 44.170 (1985), p. 519–521.

[Mun+11] Aaftab Munshi e.a. *OpenCL programming guide*. Pearson Education, 2011.

[NVI] NVIDIA. *CUDA FAQ*. `https://developer.nvidia.com/cuda-faq`. Gebruikt: 01-06-2019.

[VW99] Paul C Van Oorschot en Michael J Wiener. „Parallel collision search with cryptanalytic applications". In: *Journal of cryptology* 12.1 (1999), p. 2–8.

[ZC10] Kaiyong Zhao en Xiaowen Chu. „GPUMP: A multiple-precision integer library for GPUs". In: *2010 10th IEEE International Conference on Computer and Information Technology*. IEEE. 2010, p. 1164–1168.