CHALMERS

Effektiv parallell slumptalsgenerering på GPU:er

Kandidatarbete RRYX02-17-01

JAKOB AASA, TKTEM
ROBIN JOHANSSON, GU-FY
MARCUS LUNDBERG, TKDAT

Institutionen för Rymd-, geo- och miljövetenskap
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, 2017

Kandidatarbete RRYX02-17-01

Effektiv parallell slumptalsgenerering på GPU:er

JAKOB AASA, ROBIN JOHANSSON, MARCUS LUNDBERG

CHALMERS

Institutionen för Rymd-, geo- och miljövetenskap
Chalmers tekniska högskola
Göteborg, 2017

Effektiv parallell slumptalsgenerering på GPU:er
JAKOB AASA, ROBIN JOHANSSON, MARCUS LUNDBERG

Handledare: Markus Billeter, institutionen för Rymd-, geo- och miljövetenskap
Handledare: Magnus Röding, RISE Research Institute of Sweden, Bioscience and Materials, Agrifood and Bioscience
Examinator: Magnus Thomasson, institutionen för Rymd-, geo- och miljövetenskap

Kandidatarbete RRYX02-17-01
Institutionen för Rymd-, geo- och miljövetenskap
Chalmers tekniska högskola
SE-412 96 Gothenburg

Omslag: Bilden föreställer ett konsumentgrafikkort tillverkat av Nvidia.

Typsatt i LaTeX
Göteborg, 2017

Effektiv parallell slumptalsgenerering på GPU:er
Jakob Aasa, Robin Johansson, Marcus Lundberg
Institutionen för Rymd-, geo- och miljövetenskap
Chalmers University of Technology

Abstract

This work compares several popular pseudo-random number generators implemented on a graphics processing unit (GPU). We consider generation for both uniform and normal distributions. The generators have also been tested using a selection of test algorithms to assess the quality of the output. As a final verification the generators have been tested in-situ on a simulation code.
We chose to implement and test five different algorithms for generating uniform distributed numbers and three for generating normal distributed numbers. The generators were implemented with an object oriented programming approach in C++ using Nvidia's CUDA framework. We have also included generators from Nvidia's own random number generator library, cuRAND, to compare with our own. The test algorithms were implemented in C++ and CUDA as well.
Our results show that some algorithms are not suited for use on GPUs, while other more GPU customized algorithms perform admirably. The results from the simulation code show that all of the generators except Wallace provide good output. The running time of the simulation code is about 100 to 250 times faster on the GPU depending on implementation compared to CPU. From our results we can recommend the Linear Congruental Generator (LCG) for generating uniform numbers if perfomance is a priority, and combining it with the Box-Muller Transform for generating normal distributed numbers.

Sammandrag

Detta arbete jämför prestanda och kvalitet för några populära slumptalsalgoritmer implementerade på GPU:er. Både likformigt fördelade (uniformt) och normalfördelade pseudoslumptalsalgoritmer har implementerats i generatorer och jämförts. Generatorerna har även blivit testade med ett urval av testalgoritmer för att granska kvalitén på de producerade slumptalen. Som ett slutgiltigt test har generatorerna körts på simuleringskod.
Vi har implementerat och testat fem olika algoritmer för att generera uniformfördelat distribuerade tal, och tre stycken för att generera normalfördelat distribuerade tal. Generatorerna är implementerade med ett objektorienterat tillvägagångssätt i C++ Nvidias Cuda-ramverk. Några av Nvidias egna implementationer har kapslats in för att jämföra med arbetets egna. Testalgoritmerna har även de varit implementerade i C++ och Cuda. Resultaten visar att några algoritmer inte är lämpade för att köras på GPUer, medan andra mer GPU-anpassade sådana presterar betydligt bättre. Resultaten från simuleringskoden visar att alla generatorer förutom Wallace ger bra slumptal. Körtiden för simuleringskoden är ungefär 100 till 250 gånger snabbare på GPU jämfört med CPU, beroende på implementation. Vi rekommenderar en linjär kongruensgenerator (LCG) för att generera uniformfördelat distribuerade tal. För normalfördelade sådana rekommenderar vi en kombination av LCG och Box-Muller-transformen.

Keywords: gpu, prng, lcg, wallacegenerator, boxmuller, mtgp.

Förord

Denna rapport är resultat av ett kandidatarbete som utförts över två läsperioder på Chalmers tekniska högskola. Vi skulle först och främst vilja tacka Markus Billeter för ovärderlig feedback på arbetet. Vi skulle även vilja tacka Magnus Röding för simuleringskoden samt feedback på rapporten.
Vissa beräkningar gjordes på resurser tillhandahållna av Swedish National Infrastructure for Computing (SNIC) på Chalmers Centre for Computational Science and Engineering (C3SE).

Jakob Aasa, Robin Johansson och Marcus Lundberg, Göteborg, Maj 2017

Innehåll

1

Inledning

Pseudoslumptal, eller deterministiska följder av tal som framstår som slumpmässiga, har många användningsområden inom bland annat datorgrafik, kryptografi och simuleringar. Dessa tal har under lång tid för det mesta genererats på vanliga få- eller enkeltrådade processorer. På senare tid, i och med den stadigt ökande beräkningskraften i grafikprocessorer (GPU) gentemot traditionella processorer (CPU), har det framkommit en önskan av att köra olika sorters beräkningar på GPU:er istället, och ett av dessa är just generering av pseudoslumptal [1]. Idag görs detta genom ramverk speciellt anpassade för grafikkort.

En begränsning av GPU:er är att beräkningarna behöver vara massivt parallelliserbara (se Avsnitt 3) vilket också slumptalsberäkningar i många fall är. Arbetet undersöker ett antal olika slumptalsalgoritmers anpassningsbarhet för GPU:er och deras prestanda.

1.1 Diffusionssimulering

Ett specifikt behov av pseudoslumptal uppkommer i stokastiska simuleringar, där mycket av simuleringstiden kan gå åt att generera pseudoslumptal. Vi applicerar algoritmerna på en simulering för hur partiklar sprider sig i vätska [2], så kallad partikeldiffusion, för en mer praktisk indikator på eventuella prestandavinster.

En körning består av ett antal experiment där varje experiment innehåller ett stort antal oberoende partiklar. Partiklarna rör sig helt slumpmässigt i en volym och kan simuleras oberoende av varandra.

Idén är att simulera hela volymen, men endast samla in data från ett litet utsnitt av den, detta skulle likna verkligheten där ett mikroskop endast samlar in data från en liten mängd av provet. Informationen som sparas är avståndet partiklarna färdats (r) samt antalet tidssteg (N) inuti volymsnittet, då partikeln lämnar volymsnittet beräknas ett MSD-värde

$$\mathrm{MSD}_{\mathrm{approx}} = \frac{1}{N}\sum_{i}^{N} dr_i^2 \tag{1.1}$$

där dr_i är längden på den i'te förflyttningen inuti volymsnittet. Det teoretiska MSD-värdet är

$$\mathrm{MSD}_{\mathrm{teori}} = 2dD\delta t \tag{1.2}$$

där d är antalet dimmensioner, D är diffusionkoefficienten, och dt är tidssteget i simulationen. Utifrån Ekvation 1.1 och 1.2 kan den approximerade diffusionskoefficienten beräknas som

$$D_{\text{approx}} = \frac{\text{MSD}_{\text{approx}}}{2d\delta t}.$$

Tiden partikeln befinner sig i snittet bör vara ungefär geometriskt fördelat.

En stor del av implementationens beräkningstid används för att generera både uniform- och normalfördelade slumptal vilket är varför simuleringen är relevant för detta arbete.

1.2 Syfte och mål

Syftet med arbetet är att undersöka för- och nackdelar med olika slumptalsgeneratorer och jämföra dem med varandra med inriktning mot både ren prestanda men också för en exempelimplementation för mer praktiska tester.

Målet är att beskriva ett antal olika färdiga slumptalsgeneratorer för normal- och likformigt fördelade slumptal och implementera dem för GPU:er. Dessa ska sedan analyseras med avseende på prestanda, minnesanvändning och slumptalkvalité men också användas i kod för diffusionssimuleringen. Som slutsats av vår analys ska det tas fram rekommendationer på slumptalsgeneratorer för olika användningsområden.

1.3 Avgränsningar

Arbetet kommer att beröra fem olika uniformfördelade generatorer och tre stycken normalfördelade. Dessa kommer testas med ett antal olika egenskrivna implementationer ur TestU01-biblioteket [3] samt ENT-programmet [4]. Generatorerna appliceras på diffusionssimuleringen. Arbetet innehåller prestandatester och analyser över samtliga generatorer. Arbetet behandlar endast GPU:er från grafikkortstillverkaren Nvidia.

1.4 Upplägg

Arbetet ger en omfattande teorigenomgång av hur slumptalsalgoritmerna som behandlas fungerar, hur man kvalitetstestar dessa, samt hur en grafikprocessor är uppbyggd. I Avsnitt 4 presenteras mer praktiska implementationsdetaljer och vilka optimeringar som genomförts. Resultat från kvalitet- och prestandamätningar redovisas i Avsnitt 5, med en påföljande diskussion i Avsnitt 6.

2

Slumptalsteori

Många användningsområden inom simulering och grafik använder sig av någon form av slumptal i sin kod, ett exempel är ray tracing. Den snabbaste och mest vanliga versionen av dessa är så kallade pseudoslumptal, tal som genereras deterministiskt genom algoritmer, och vars mål är att i största mån efterlikna äkta slumpmässighet. Alternativen till dessa är sanna slumptal som tas från mätdata, vilka har användningsområden men är avsevärt mycket långsammare [5].

Bra slumptalskvalitet definieras av hur slumpmässigt följder av slumptalen verkar, och hur bra de följer den givna fördelningen. Om man till exempel genererar slumptal med en normalfördelning N(0,1) så väntas talen följa fördelningen i största mån. Slumpmässighet är extra viktig då hela meningen med pseudoslumptal är att de ska efterlikna äkta slumptal. Hur man analyserar slumptalskvalitet beskrivs närmare i Avsnitt 2.5.

Under åren har ett antal algoritmer med olika egenskaper tagits fram. För olika användningsområden är vissa egenskaper mer viktiga än andra. Det finns de applikationer där god prestanda är mest viktigt, för andra är det god slumptalskvalité. I säkerhetskritiska program är det särskilt viktigt att nästa slumptal inte kan förutspås, och det finns då kryptografiskt säkra algoritmer som är specialiserade på just detta [6].

En slumptalsgenerator (PseudoRandom Number Generator, PRNG), kan ses som två funktioner: f och g, där f förändrar tillståndet som generatorn befinner sig i, och g använder nuvarande tillstånd för att producera ett resultat: själva slumptalet i sig. I många fall är $g(n) = n$ (identitetsfunktionen), vilket betyder att generatorns tillstånd direkt blir det producerade slumptalet [6].

Starttillståndet för att påbörja slumptalsgenereringen brukar kallas *seed* eller frö. En generators *period* definieras som antalet tillstånd som produceras innan generatorn når tillbaka till starttillstånd [7].

I följande avsnitt finns beskrivningar av några olika uniformfördelade generatorer, som är utvalda för deras popularitet (LCG, Mersenne twister), anpassning för exekvering på GPU:er (MTGP, Philox) eller utlovad prestanda (Threefry). Arbetet täcker också några populära normalfördelade generatorer: Box-Muller (se Avsnitt 2.4.1), Ziggurat (Avsnitt 2.4.2) samt Wallace Gaussian Generator (Avsnitt 2.4.3), där både Box-Muller och Ziggurat är vanliga i applikationer, och Wallace är ett annorlunda angreppssätt på normalfördelning.

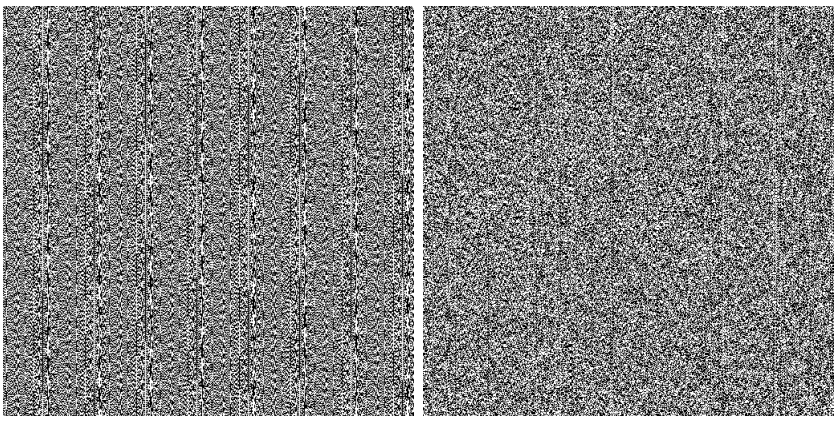
2.1 Linjär kongruensgenerator

Den kanske enklaste slumptalsgeneratorn är en linjär kongruensgenerator (LCG), som använder en given faktor a, konstant c samt modulusoperatorn, modulus m, för att från ett tidigare tal generera ett nytt slumptal - denna definition visas i Ekvation 2.1. LCG används som standardgenerator i bland annat programspråken Java [8] och ANSI C [9].

$$\begin{aligned} n_{i+1} &= f(n_i) = (a \times n_i + c) \mod m \\ g(n) &= n \end{aligned} \tag{2.1}$$

En fördel med LCG är att algoritmen kompileras till relativt få instruktioner då den i princip bara består av enkel aritmetik. Därför blir LCG prestandaeffektiv jämfört med många andra mer komplicerade PRNG:er. Dessutom beror det nya tillståndet endast på det förra slumptalet, så varje generator behöver minimalt minne för att spara tillståndet.

Ett problem med algoritmen är dock bristande slumptalskvalité, där lägre bitar i talet inte är lika slumpmässiga som högre. Detta fenomen kan ses i Figur 2.1 där mönstret för bit 0 (Figur 2.1a) är betydligt mindre slumpmässigt än mönstret för bit 5 (Figur 2.1b).



(a) Bit 0 **(b)** Bit 5

Figur 2.1: Mönstret skapas genom att för varje pixel generera ett slumptal, maska ut bit x, och om den är 1 så blir pixeln vit, annars svart.

Kvalitetsproblemet löses enklast genom att justera g till att skifta ner högre bitar. Detta ökar kvalitén men det innebär också att det resulterande slumptalet är mer begränsat i sin storlek. Ett exempel är Javas PRNG som använder sig av 48-bitars tillstånd och skiftar bort de lägsta 16 bitarna för att producera ett 32-bitars tal [8]. Generatorns period är 2^48. Denna variant, med Javas egna konstantvärden, visas i Ekvation 2.2. Operatorn >> är bitvis högershift.

$$\begin{aligned} n_{i+1} &= f(n_i) = (25214903917 \times n_i + 11) \mod 2^{48} \\ g(n) &= n >> 16 \end{aligned} \tag{2.2}$$

2.2 Mersenne twister och MTGP

En annan vanlig generator är *Mersenne twister* [10] som används av de flesta mjukvarusystem och kodbibliotek, och är dessutom standardimplementationen i ett flertal språk som Python [11], Ruby [12] och Julia [13]. Generatorns period är ett mersenneprimtal, ett primtal som kan skrivas på formen $2^k - 1, k \in \mathbb{N}$, vilket också är inspirationen till namnet på generatorn. Makoto Matsumoto och Takuji Nishimura introducerade Mersenne twister 1998 med en föreslagen period $P = 2^{19937} - 1$, vilket har lett till att de flesta implementationer använder sig av just detta primtal. En viktig aspekt som talar för denna implementation, MT19937, är att den långa perioden leder till att slumptalsgenerationen håller god kvalité och den är dessutom relativt snabb. En nackdel är dock den stora tillståndsbufferten, hela 2.5 kB[1], vilket gör denna implementation problematisk på GPU:er där minnet är begränsat.

Det finns andra versioner av Mersenne twister, till exempel en variant som har mindre minnesavtryck. En mer intressant typ för detta arbete är MTGP [14], som är speciellt anpassad för parallell exekvering. MTGP fungerar liknande som Mersenne Twister, men där den stora skillnaden är att tillståndet delas mellan flera parallellt exekverande enheter. Denna algoritm finns beskriven i Ekvation 2.3, där f är en rekursiv funktion som utför ett antal specifika bitoperationer på parametrarna. Funktionen h är en del av g och används för att bättra kvalitén på slumptalen med ytterligare bitoperationer. Konstanten m är specificerad av implementationen, men kan exempelvis vara $n/2$. Operatorn $\oplus$ är bitvis XOR (exclusive OR).

$$\begin{aligned} x_{n+i} &= f(x_{m+i}, x_{i+1}, x_i), 1 < m < n \\ g &= x_{n+i} \oplus h(x_{m-1+i}) \end{aligned} \tag{2.3}$$

Ett möjligt problem med att exekvera f parallellt är om ett eller flera av argumenten inte räknats klart än av en annan tråd eller trådar. MTGP-algoritmen är dock utformad på ett sådant sätt att om beräkningsbufferten är tillräckligt stor så förändrar trådarna inte varandras argument och de kan då exekveras parallellt. Det krävs att tillståndsbufferten är minst $2n - m$ stor för att $n - m$ trådar ska kunna parallellt exekvera algoritmen utan att påverka varandra [14].

2.3 Counter based random number generators

En nackdel med konventionella slumptalsgeneratorer är att de kräver ett tillstånd (*state*) för att generera ett nytt pseudoslumptal och implicit innebär det att för att generera det n:te slumptalet krävs det beräkning av det $n - 1$:ta. Salmon et al. presenterar istället *Counter based random number generators* (CBRNG) [6], eller räknare-baserade slumptalsgeneratorer som undviker denna begränsning genom att vara konstruerad så att genera-

[1] Artikeln nämner storleken 624 ord (word), och definierar ett ord som 32 bitar, 4 byte. Då blir minneavtrycket: 624 ord = 624∗4 bytes ≈ 2,5 kB. På ett 64-bitars system blir det dubbelt så stort, 5 kB

torerna tar in två värden; en *counter*, räknare, samt en *key*, nyckel. Nyckeln kan sättas så att varje ström av generatorn har en egen och sedan inkrementeras räknaren, som då är tillståndet för varje gång ett nytt slumptal behövs. Detta innebär att tillståndsfunktionen f är en enkel addition till räknaren och att g istället görs mer komplicerad. Fördelen blir att det är enkelt att starta upp ett stort antal generatorer med samma tillstånd och ge varje en egen nyckel.

Konceptet härstammar från hur blockchiffer krypterar, där en förändring av en bit i klartexten bör ge 50% chans att förändra en godtycklig bit i blockchiffertexten. Enligt Salmon et al. ger algoritmerna som används till blockchifferna extremt bra slumptal men är också för långsamma för de flesta ändamål där endast slumptalen är av intresse. Genom att framför allt förenkla hur nycklar genereras för varje steg (se Figur 2.2) av krypteringen lyckades författarna åstadkomma stora prestandavinster. I detta arbete implementeras två av dessa algoritmer, *Philox* samt *Threefry*.

Den konceptuella skillnaden mellan Threefry och Philox ligger i att den förstnämnda bygger på att en större mängd steg körs och där nycklarna inte appliceras lika ofta men å andra sidan genereras på ett mer avancerat sätt. Philox använder sig istället av färre steg, men där beräkningarna i varje steg är av en mer komplicerad karaktär. Enligt skaparna har båda generatorena en period på åtminstone 2^{128} för varje nyckel, och totalt finns 2^{64} stycken nycklar [6].

2.3.1 Philox

Philox är en CBRNG som använder sig av ett Substitution-Permutation Network (SP-nätverk) (se Figur 2.2). Philox finns i några olika utföranden beroende på hur många slumptal som ska genereras åt gången, samt storleken i bitar på talen som ska genereras. Varianten som genererar fyra heltal åt gången använder en modifierad Feistel-funktion

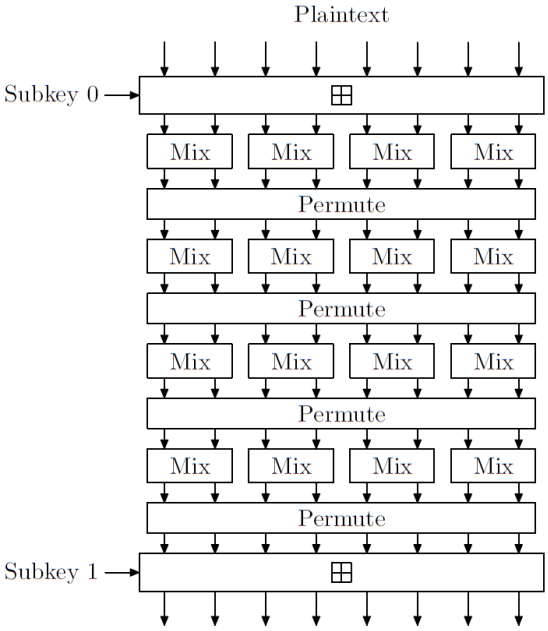
$$\begin{aligned} L_n &= B_k(R_{n-1}) = \texttt{mullo}(R_{n-1}, M) \\ R_n &= F_k(R_{n-1}) \oplus L_{n-1} = \texttt{mulhi}(R_{n-1}, M) \oplus k_{n-1} \oplus L_{n-1}, \end{aligned} \tag{2.4}$$

för substitutionsdelen för varje par (S-box) Där $\oplus$ är en bitvis XOR-operation, `mullo` tar de lägre W bitarna av en $W \times W$-bitarsmultiplikation och `mulhi` tar de högre. Dessa par permuteras sedan med varandra (P-box). Två kopior av L_0, R_0 är då de ursprungliga räknarna som skickas in i generatorn och genom ett antal rundor itereras tills en tillräckligt bra slumpnivå har nåtts.

M är en fast konstant för varje S-box. Nyckeln k adderas med en konstant varje runda, på formen

$$\begin{aligned} k_0 &= \text{Från användaren} \\ k_n &= k_{n-1} + C \mod 2^W \end{aligned}$$

Salmon et al. har tagit fram lämpliga konstanter genom att undersöka när förändringen blir störst på så få rundor som möjligt.



Figur 2.2: Figuren visar en överblick över ett SP-nätverk. En klartext tas in och i ett flertal steg mixas och permuteras till en krypterad variant. Mellan vissa steg appliceras även en nyckel för att förhindra avkryptering.
(Public Domain)
[CC0 3.0 (https://creativecommons.org/publicdomain/zero/1.0/deed.en)], via Wikimedia Commons

2.3.2 Threefry

Threefry är en CBRNG [6] som baseras på en förenklad variant av Threefish-chiffret som används till Skein-hashfunktionen [15]. CBRNGn använder samma typ av SP-nätverk (se Figur 2.2) som Threefish men förenklar genom att ta bort en extra modifiering av nycklarna. Threefry finns i modifierade varianter som kan variera antalet iterationsrundor, antalet slumptal som genereras åt gången samt ord-storleken mellan 32 och 64 bitar.

Threefry använder sig av enkla matematiska operationer för att modifiera sitt tillstånd: XOR, addition samt roterande skift-operationer av bitar. S-boxen använder sig av två ord i taget, där det ena resultat är en addition av de två orden, medan det andra resultatet ges av att det först bitroteras med en konstant för varje runda innan en XOR-operation görs med det första resultat. Efter detta permuteras alla orden och fortsätter köras. Var fjärde runda adderas en nyckel till alla orden.

Threefrys algoritm för en ord-storlek på 32 bitar som genererar 4 tal åt gången är som följer:

- För den nuvarande rundan d av totala antalet rundor $0, \ldots, N_r$ ges

$$e_{d,i} := \begin{cases} (v_{d,i} + k_{d/4,i}) \mod 2^{32}, & \text{om } d \mod 4 = 0 \\ v_{d,i}, & \text{annars.} \end{cases}$$

  Här är $v_{d,i}$ det i:te slumptalet under d:te rundan och $k_{d/4,i}$ är nyckeln som appliceras var fjärde runda.

- Varje par mixas i varsin S-box genom funktionen

$$(f_{d,2j}, f_{d,2j+1}) := \text{MIX}_{d,j}(e_{d,2j}, e_{d,2j+1})$$

  med $j = 1, 2$ där

$$(x_0, x_1) = \text{MIX}(y_0, y_1) = \begin{cases} y_0 := (x_0 + x_1) \mod 2^{32} \\ y_1 := (x_1 <<< R_{(d \mod 8),j}) \oplus y_0, \end{cases}$$

  R är en samling konstanter, $<<<$ är bitvis vänsterrotation.

- Slumptalen för nästa runda permuteras till slut genom permuteringsfunktionen $\pi(i)$ som definieras enligt $\pi(0) = 0$, $\pi(1) = 3$, $\pi(2) = 2$, $\pi(3) = 1$. Slumptalet till nästa runda blir då $v_{d+1,i} := f_{d,\pi(i)}$.

- Denna process upprepas tills dess att den önskade mängden rundor uppnåtts och slutligen returneras $e_{d,i}$.

Nycklarna $k_{0,\ldots,3}$ skapar hjälpnyckeln k_4 enligt $k_4 = \texttt{0x1BD11BDA} \oplus k_0 \oplus \cdots \oplus k_3$. Efter detta roteras nycklarna enligt

$$k_{d/4,i} := \begin{cases} k_{(d/4+i) \mod 5}, & \text{för } i = 0, 1, 2 \\ k_{(d/4+i) \mod 5} + d/4, & \text{för } i = 3. \end{cases}$$

Konstanten som används för att skapa hjälpnyckeln k_4 syfte är enligt skaparna att se till så att k_4 inte kan bli 0.

2.4 Normalfördelade slumptal

Nästan alla pseudoslumptalsgeneratorer genererar uniformfördelat distribuerade tal. I många statistiska och fysikaliska tillämpningar behövs det istället normalfördelade sådana. Det finns en generator (se Wallace Gaussian Generator i Avsnitt 2.4.3) som skapar normalfördelade men den behöver en distribution av normalfördelade tal som frö vilket innebär att behovet av att på något sätt transformera om en uniformtfördelat distribution av tal till normalfördelade sådana kvarstår.

2.4.1 Box-Muller-transformen

Box-Muller-transformen [16] är en metod som använder sig av två oberoende uniformfördelade slumptal, $U_1, U_2 \in (0, 1)$, för att skapa två oberoende normalfördelade slumptal X_1, X_2 där

$$\begin{aligned} X_1 &= (-2 \ln U_1)^{1/2} \cos(2\pi \times U_2) \\ X_2 &= (-2 \ln U_1)^{1/2} \sin(2\pi \times U_2). \end{aligned} \tag{2.5}$$

Att detta ger normalfördelade slumptal visar Box och Muller med att hänvisa till inverserna som blir

$$\begin{aligned} U_1 &= e^{\frac{-(X_1^2+X_2^2)}{2}} \\ U_2 &= -\frac{1}{2\pi} \arctan \frac{X_2}{X_1}. \end{aligned} \tag{2.6}$$

Ur detta kan den gemensamma täthetsfunktionen

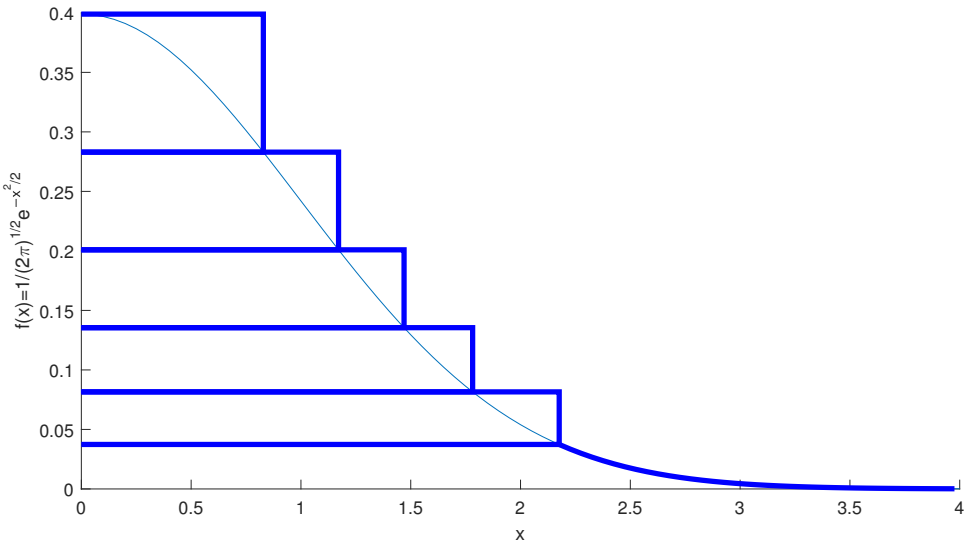
$$f(X_1, X_2) = \frac{1}{2\pi} e^{\frac{-(X_1^2+X_2^2)}{2}} = \frac{1}{\sqrt{2\pi}} e^{\frac{-X_1^2}{2}} \times \frac{1}{\sqrt{2\pi}} e^{\frac{-X_2^2}{2}} = f(X_1)(X_2) \tag{2.7}$$

beräknas, som förutom att visa att X_1, X_2 är normalfördelade även visar att de är oberoende av varandra. Box-Muller-transformen är en tidig och ganska långsam metod på den är implementerad på CPU:er jämfört med modernare varianter som Ziggurat [17] och Polar-metoden [18]. Box-Muller har dock en utmärkande fördel jämfört med andra metoder i det att det alltid är samma kod som körs, det vill säga den grenar sig inte och inga specialfall behöver tas hänsyn till jämfört med exempelvis Ziggurat.

2.4.2 Ziggurat

Ziggurat[17] är en förkastande metod för distributioner med avtagande täthetsfunktion f, eller unimodal symmetrisk sådana där båda sidor är avtagande (exempelvis normalfördelning). En enkel men extremt ineffektiv förkastande metod hade genererat två uniforma tal X, Y i ett rektangulärt område som innefattar hela distributionens täthetsfunktion (för halva $N(0, 1)$ skulle alltså $X \in [0, \infty), Y \in [0, 1/\sqrt{2\pi}]$. Slumptalet X accepteras förutsatt att $Y \leq f(X)$ vilket resulterar i att de accepterade talen följer distributionen.

Ziggurat skapar istället $n - 1$ rektanglar som tillsammans med en basregion av samma area A täcker distributionens fördelningsfunktion (se Figur 2.3). Då alla regioner har samma area kan en region slumpmässigt väljas där ett par av uniforma slumptal som formar en punkt inom rektangeln kan testas. Ziggurat utnyttjar att fördelningsfunktionen är avtagande för att nästan alltid behöva generera endast ett tal. Ett specialfall uppstår för basregionen som då består av en rektangel samt 'svansen'. När ett slumptal placeras i basregionen kan det inte förkastas då hela regionen ligger under fördelningsfunktionens kurva och det genererade talet behöver därmed hanteras annorlunda.



Figur 2.3: Figuren visar sex rektanglar samt basregion med 'svans' som ziggurat-metoden kan använda sig av för en normalfördelning. Rektanglarna och basregionen är av identisk area.

Ziggurats algoritm är sedan som följer:

- en slumpmässig region $0 \leq i \leq n$ av n rektanglar samt basregion väljs ut.
- Ett värde $x = U_0 x_i$ genereras där $U_0 \in [0, 1)$ är en uniform slumpmässig variabel och $x_i, i \neq 0$ är den nedre punkten på rektangeln, då $i = 0$, beräkna $x = U_0 A / f(x_0)$.
- Om $x \leq x_{i+1}$ är slumptalet accepterat och algoritmen är klar. Notera att om $i = 0, x > x_{i+1}$ så krävs ett specialfall, se nedan.
- Annars genereras ett nytt värde $y = y_i + U_1(y_{i+1} - y_i)$ som jämförs med fördelningsfunktionen $f(x)$. Om $y \leq f(x)$ accepteras slumptalet och algoritmen är klar.
- Annars börja om från början då den genererade (x, y)-punkten inte ligger inom distributionens fördelningsfunktion.

För specialfallet då $i = 0$ och $x > x_{i+1}$ behöver en variabel från 'svansen' genereras. För det föreslår Marsaglia [19] följande:

- Generera $x = -\ln(U_1)/x_0,\ y = -\ln(U_2)$ tills $y + y > x \times x$.
- Returnera $x_0 + x$.

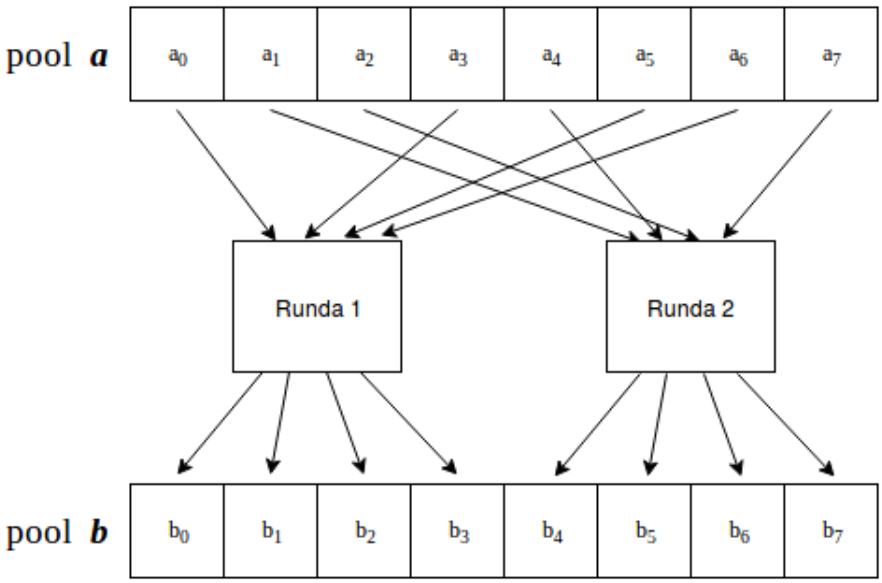
Algoritmen transformerar om svansen till enhetsintervallet och genererar slumptal fram tills det att man hittar ett utfall som ligger inom svansen, som då accepteras.

Genom att låta $x_0 = 3,442619855899$ för en 128 remsor hög Ziggurat visar Marsaglia att arean för varje rektangel och basregion kan beräknas och sannolikheten att endast ett slumptal behöver genereras blir $97,98\%$.

I Marsaglias originalimplementation av Ziggurat finns också några brister med slumptalens kvalité: En för enkel slumptalsgenerator används för de ursprungliga uniforma talen samt att de minst signifikanta bitarna användes för att välja det vertikala indexet vilket medför en korrelation i ett tals storlek och dess minst signifikanta sista siffror. Mathworks, som använder Ziggurat-algoritmen för sin normalfördelade slumpgenerator `randn` i Matlab använder istället två stycken Mersenne twister-genererade 32-bitars slumptal där 8 bitar används för det vertikala indexet och 52 av de resterande 56 bitarna används för att konstruera ett *double*-flyttal [20].

2.4.3 Wallace Gaussian Generator

Wallace Gaussian Generator [21], förkortat Wallace, är en algoritm som använder sig av en redan tidigare genererad samling (pool) av n normalfördelat distribuerade slumptal för att generera n nya. Normalfördelningen behålls i den nya poolen genom matris-multiplikation på den gamla. I en artikel från 1996 presenterar C. S. Wallace denna metod, som namngivits efter honom. Hans process består av att låta poolen vara $K \times L$ stor och i en *runda* tas K tal ur poolen till en vektor X_i, som multipliceras med en ortogonal matris A_j, $X_{i+1} = A_j X_i$. Då samlingen har storleken $K \times L$ så är en *iteration* av algoritmen L rundor. Ett exempel på algoritmen visas i Figur 2.4. Matrisen A_j väljs slumpmässigt för varje runda ur mängden $A_1, ..., A_k$, där k är antalet iterationer som körs. Fördelen med att dela upp iterationer i rundor på detta sätt är att varje runda kan exekveras oberoende av de andra då ett tal i den gamla poolen endast används i en runda.

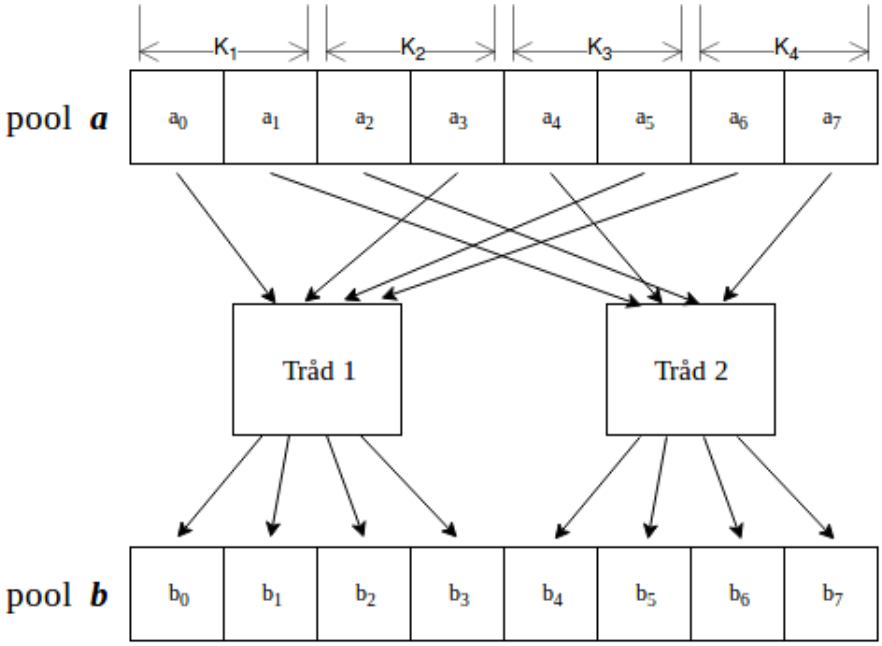


Figur 2.4: Denna figur föreställer en *iteration* av Wallace-algoritmen, där $K = 4$, och antalet rundor: $L = 2$. Rundorna hämtar, efter varandra, de K talen från slumpmässiga platser i poolen $\boldsymbol{a}$, utför matrismultiplikation och skickar resulterande tal till $\boldsymbol{b}$.

Anledningen till att man skulle vilja köra flera iterationer innan man returnerar sina slumptal till användaren är för att öka slumptalskvaliteten. För att den ska vara tillräckligt bra

så bör varje tal i ursprungspoolen till slut påverka alla tal i slutpoolen. Wallace löser detta genom att tolka poolen som en matris med K rader och L kolumner. Efter varje iteration transponeras poolen genom att byta plats på K och L. En annan metod som föreslås är att samlingen permuteras vid varje iteration, vilket tillsammans med transponeringen leder till tillräckligt bra kvalitet enligt C. S. Wallace [21].

Lee Howes och David Thomas visar i en artikel en variant av Wallaces generator som är anpassad för GPU-exekvering [5]. De låter K vara 4 och varje runda exekveras av en tråd som parallellt hämtar in 4 tal från en pool och utför beräkningarna. Detta sätt att exekvera algoritmen visas i Figur 2.5. Denna metod är möjlig eftersom varje tal i den äldre samlingen endast används en gång per iteration. Det innebär att varje runda kan arbeta oberoende av de andra.



Figur 2.5: Detta exempel använder parametrarna $K = 4$, och antalet steg $L = 2$. Varje *runda* i Wallaces ursprungsalgoritm exekveras här parallellt av en tråd. Till skillnad från ursprungsalgoritmen plockas inte de K talen per tråd från godtycklig plats i poolen utan varje tråd hämtar ett tal var per K_i i $K_1..K_4$. Anledningen till detta finns beskrivet i Avsnitt 4.2.3.

2.5 Test av pseudoslumptal

I detta avsnitt beskrivs teorin bakom de statistiska tester som används för att avgöra kvaliteten av slumptalsgeneratorerna. För varje test ges en överblickande beskrivning, en kort härledning samt en sammanfattning.

De tester som har valts är χ^2-testet och Kolmogorov-Smirnov-testet. Testerna har valts för att de är bra på att avgöra hur väl mätdata följer en underliggande distribution. I fallet för simulationskoden och uppskattningen av diffusionskonstanten är det viktigt

att slumptalen följer normalfördelningen väl, det är inte lika viktigt att slumptalen är oberoende sinsemellan.

Testerna kan användas separat eller tillsammans med varandra vilket rekommenderas av Knuth i *The Art of Computer Programming* [22], kombinationen av testerna beskrivs i Avsnitt 4.3.

2.5.1 Chi-kvadrattestet

χ^2-testet är ett vanligt statistiskt test som använder sig av en teststatistika och som avgör hur väl en mängd data följer en underliggande distribution. Utifrån diskret data (eller kontinuerlig data som diskretiserats) beräknas en teststatistika som jämförs med tabellvärden. Jämförelse med tabell ger information om hur sannolikt det erhållna värdet är. Om värdet är för osannolikt kan detta användas som motivation för att en hypotes skall motbevisas.

Grunden för testet ligger i att en underliggande distribution kan representeras av K stycken intervall med sannolikhetstätheterna $p_1, p_2, p_3, \ldots\ldots, p_K$, där $\sum_i^K p_i = 1$. Givet N stycken diskreta oberoende mätpunkter sorteras dessa in i de K intervallen vilket ger $x_1, x_2, x_3, \ldots\ldots, x_K$, där $\sum_i^K x_i = N$, och x_i anger antalet mätpunkter inom det i:te intervallet.

Teststatistikan bygger på skillnaden mellan antalet mätpunkter i ett specifikt intervall och det förväntade antalet mätpunkter för samma intervall, skillnaden kvadreras och normaliseras vilket ger teststatistikan

$$\chi^2 = \sum_{i=1}^{K} \frac{(x_i - Np_i)^2}{Np_i} = \frac{1}{N} \sum_{i=1}^{K} \left(\frac{x_i^2}{p_i} \right) - N \tag{2.8}$$

där den andra formen är en kompaktare och mer lättanvänd form som erhållits efter användning av att $\sum_i^K p_i = 1$ och att $\sum_i^K x_i = N$ [22].

Den erhållna teststatistikan i Ekvation 2.8 beror på hur bra mätdatan följer den underliggande distributionen. Mätdata som följer distributionen väl kommer att ha ett relativt lågt värde på χ^2 (skalan på χ^2 beror på ν), teststatistikan kommer att ha ett relativt högt värde för data som avviker mycket från distributionen. I fallet då slumptal studeras ska teststatistikan varken vara för hög eller för låg[2]. En låg teststatistika tyder på att datan inte är tillräckligt slumpartad och en för hög teststatistika tyder på att slumptalen inte följer den underliggande distributionentillräckligt väl [22].

För att avgöra om den erhållna teststatistikan är sannolik eller ej behövs sannolikhetstäthetsfunktionen (PDF) $\mathrm{f}(X, \nu)$, samt den kumulativa fördelningsfunktionen (CDF) $\mathrm{F}(X, \nu)$, för χ^2-fördelingen. PDF för χ^2-fördelningen [23] är

$$f(X, \nu) = \frac{1}{2^{\nu/2}\Gamma\left(\frac{\nu}{2}\right)} X^{\frac{\nu}{2}-1} \mathrm{e}^{-\frac{X}{2}} \tag{2.9}$$

där ν är antalet frihetsgrader ($\nu = K - 1$) och där $\Gamma\left(\frac{\nu}{2}\right)$ är gammafunktionen med parametern $\frac{\nu}{2}$.

[2]Vad som anses vara högt och lågt avgörs av en signifikansnivå som oftast är 5%.

CDF:en beräknas utifrån PDF:en given i Ekvation 2.9 på följande sätt:

$$\begin{aligned}
\mathrm{F}\left(X,\nu\right) &= \int_0^X f\left(x,\nu\right)dx \\
&= \frac{1}{2^{\nu/2}\Gamma\left(\frac{\nu}{2}\right)}\int_0^X x^{\frac{\nu}{2}-1}\mathrm{e}^{-\frac{x}{2}}dx \\
&= \frac{1}{2^{\nu/2}\Gamma\left(\frac{\nu}{2}\right)}\sum_{i=0}^{\frac{\nu}{2}-1}\left[\frac{\left(-1\right)^i x^{\frac{\nu}{2}-1-i}\left(\frac{\nu}{2}-1\right)!\mathrm{e}^{-\frac{x}{2}}}{\left(\frac{\nu}{2}-1-i\right)!\left(-\frac{1}{2}\right)^{i+1}}\right]_{x=0}^{X} \\
&= \frac{1}{2^{\nu/2}\Gamma\left(\frac{\nu}{2}\right)}\left(2^{\frac{\nu}{2}}\left(\frac{\nu}{2}-1\right)! - \sum_{i=0}^{\frac{\nu}{2}-1}\frac{2^{i+1}X^{\frac{\nu}{2}-1-i}\left(\frac{\nu}{2}-1\right)!\mathrm{e}^{-\frac{X}{2}}}{\left(\frac{\nu}{2}-1-i\right)!}\right) \\
&= \frac{1}{\Gamma\left(\frac{\nu}{2}\right)}\left(\left(\frac{\nu}{2}-1\right)! - \frac{\mathrm{e}^{-\frac{X}{2}}}{2^{\nu/2}}\sum_{i=0}^{\frac{\nu}{2}-1}\frac{2^{i+1}X^{\frac{\nu}{2}-1-i}\left(\frac{\nu}{2}-1\right)!}{\left(\frac{\nu}{2}-1-i\right)!}\right)
\end{aligned} \tag{2.10}$$

där de tre sista likheterna endast gäller för jämna värden på ν.

CDF-funktionen i Ekvation 2.10 kan approximeras [23] av

$$\mathrm{F}\left(X,\nu\right) \approx \mathrm{H}\left(\frac{\mathrm{A}\left(X,\nu\right)}{\mathrm{B}\left(\nu\right)}\right) \tag{2.11}$$

där funktionerna $\mathrm{A}\left(x,r\right)$, $\mathrm{B}\left(r\right)$, och $\mathrm{H}\left(x\right)$ ges som

$$\begin{aligned}
\mathrm{A}\left(x,r\right) &= \left(\frac{x}{r}\right)^{\frac{1}{3}} + \frac{2}{9r} - 1 \\
\mathrm{B}\left(r\right) &= \sqrt{\frac{2}{9r}} \\
\mathrm{H}\left(x\right) &= \frac{1}{1+\mathrm{e}^{-\mathrm{p}(x)}} \\
\mathrm{p}\left(x\right) &= 1.59145x + 0.01095x^2 + 0.06651x^3,\ x \geq 0.
\end{aligned}$$

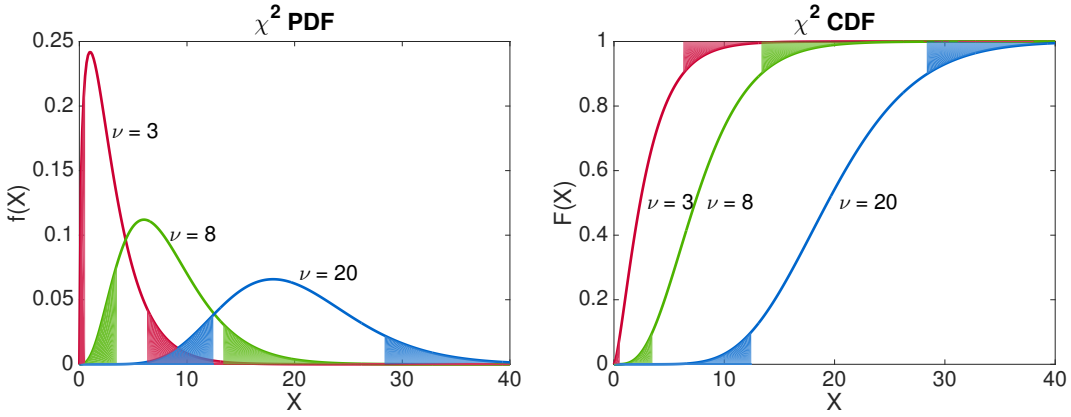
Approximationen i Ekvation 2.11 stämmer bra överens med CDF-funktionen i Ekvation 2.10 för stora värden på ν.

I Figur 2.6 visas PDF:en samt CDF:en för tre χ^2-fördelningar med olika ν. I varje delfigur visas också vilka värden på teststatistikan som ligger under 10% eller över 90%[3].

Det finns nackdelar med χ^2-testet. Det kräver CDF-värden vid signifikansgränserna, och tar inte hänsyn till all data då information om mätdatan förloras när det sorteras in i intervallen. Andra problem är att valet av intervall kan påverka styrkan av testet [22], och enstaka tester kan ge missvisande resultat. Därför är det viktigt att ett stort antal tester genomförs för pålitliga resultat. N måste också vara tillräckligt stort för ett starkt test ($Np_i > 5\,\forall\, i$).

En fördel är att testet är intuitivt, då den bygger på skillnaden mellan antalet element per intervall och det förväntade antalet element för respektive intervall. Dessutom är χ^2-testet lätt att implementera. Beräkning av teststatistikan i Ekvation 2.8 är lätt givet simpla p_i och Ekvation 2.11 innehåller inga komplicerade uttryck.

[3]Detta skulle motsvara de kritiska X värdena för ett test med signifikansnivån 10% där de övre och undre X värdena betraktas separat.



(a) (b)

Figur 2.6: Figuren visar PDF samt CDF för tre χ^2-fördelningar med olika frihetsgrader i intervallet $[0, 40]$. Teststatistikor inom de skuggade områdena har CDF värden under 10% eller över 90%.

2.5.2 Kolmogorov-Smirnov-testet

Kolmogorov-Smirnov-testet, förkortat KS-testet, är ett statistiskt test som förekommer i två versioner. Den första versionen använder två teststatistikor $\left(\mathrm{K_N^+}, \mathrm{K_N^-}\right)$ och den andra använder en teststatistika $(\mathrm{D_N})$, i båda fallen bygger teststatistikorna på kontinuerlig mätdata[4]. Teststatistikorna avgör hur väl en mängd data följer en underliggande distribution. Jämförelse med CDF värden för KS-distributionen avgör hur sannolika de erhållna statistikorna är.

Grunden för testet ligger i att studera skillnaden mellan den empiriska CDF:en för mätdatan samt CDF:en för den underliggande distributionen. Givet N kontinuerliga mätpunkter $x_1, x_2, x_3, \ldots\ldots, x_N$, kan CDF:en för mätdatan beräknas som

$$\mathrm{F}_N(X) = \frac{\#x \leq X}{N}. \tag{2.12}$$

där $\#x \leq X$ är antalet x värden mindre än X. K_N^+ och K_N^--statistikorna bygger på de största skillnaderna mellan den empiriska CDF:en för mätdatan samt den underliggande distributionens CDF. För kontinuerlig mätdata är teststatistikorna definierade [22] som

$$\mathrm{K}_N^+ = \sqrt{N} \max_{-\infty \leq x \leq \infty} \left(\mathrm{F}_N(x) - \mathrm{F}(x)\right) \tag{2.13}$$

$$\mathrm{K}_N^- = \sqrt{N} \max_{-\infty \leq x \leq \infty} \left(\mathrm{F}(x) - \mathrm{F}_N(x)\right) \tag{2.14}$$

där $\mathrm{F}(x)$ är CDF:en för den underliggande distributionen och där $\mathrm{F}_N(x)$ är CDF:en för mätdatan (se Ekvation 2.12).

I en datorimplementation kan inte x anta alla värden i intervallet $[-\infty, \infty]$ så Ekvation 2.13 och 2.14 kan därmed inte användas som dem är utan behöver skrivas om för att kunna

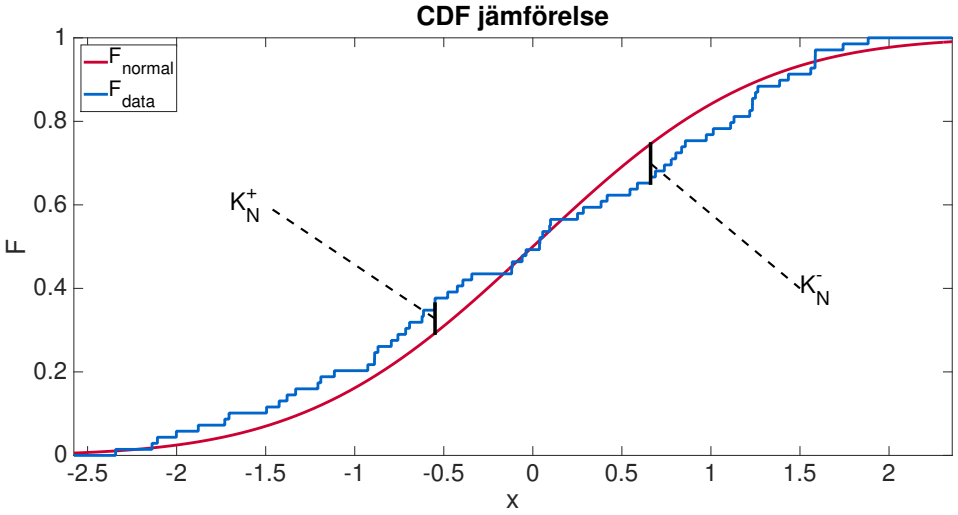
[4]Till skillnad från χ^2-testet där teststatistikan bygger på diskret data.

hantera ett fixt antal x värden. De nya ekvationerna för att beräkna teststatistikorna [22] blir

$$\mathrm{K}_N^+ = \sqrt{N} \max_{1 \leq j \leq N} \left(\frac{j}{N} - \mathrm{F}\left(x_j\right) \right) \tag{2.15}$$

$$\mathrm{K}_N^- = \sqrt{N} \max_{1 \leq j \leq N} \left(\mathrm{F}\left(x_j\right) - \frac{j-1}{N} \right) \tag{2.16}$$

där x-värdena sorterats i stigande ordning.



Figur 2.7: Figuren visar CDF-funktionen för normalfördelningen $N\left(0,1\right)$ ($\mathrm{F}_{\mathrm{normal}}$) samt CDF-funktionen för 70 normalfördelade slumptal ($\mathrm{F}_{\mathrm{data}}$) genererade med Matlabs funktion `randn`.

I Figur 2.7 visas den underliggande normalfördelningens CDF ($\mathrm{F}_{\mathrm{normal}}$) samt den empiriska CDF:en ($\mathrm{F}_{\mathrm{data}}$) för 70 stycken normalfördelade slumptal genererade med Matlabs funktion `randn`. Teststatistikorna K_N^+ samt K_N^- är markerade i grafen.

Den andra versionen av testet använder sig av statistikan D_N [22] som är definierad som

$$\mathrm{D}_N = \sqrt{N} \max_{-\infty \leq x \leq \infty} \left| \mathrm{F}\left(x\right) - \mathrm{F}_N\left(x\right) \right| = \max\left(\mathrm{K}_N^+, \mathrm{K}_N^- \right) \tag{2.17}$$

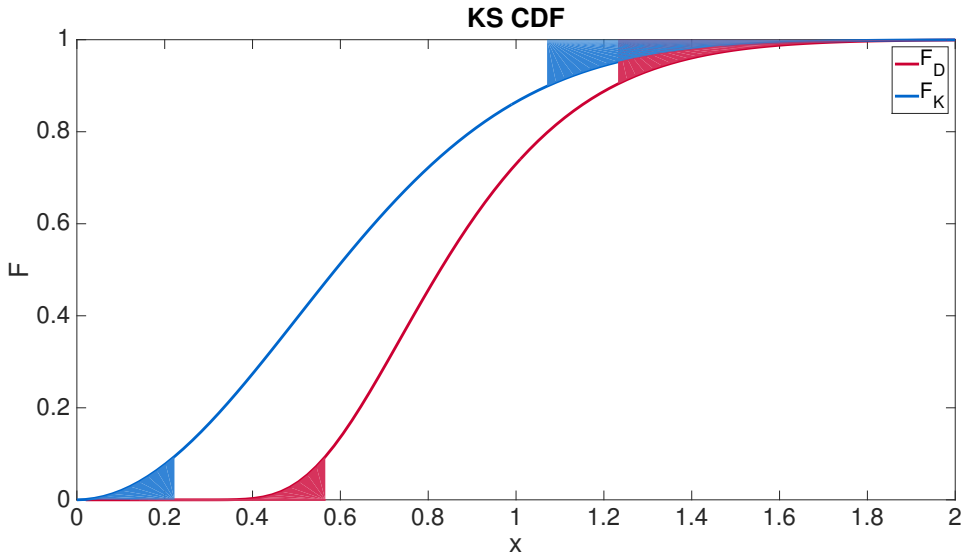
De erhållna teststatistikorna beror på hur bra mätdatan följer den underliggande distributionen. Data som följer den underliggande distributionen väl kommer att ha låga teststatistikor. På samma sätt som för statistikan i χ^2-testet skall inte KS-statistikorna vara för höga eller för låga då slumptal studeras, för höga värden innebär att mätdatan inte följer den underliggande distributionen tillräckligt bra och för låga värden innebär att mätdatan inte är tillräckligt slumpfördelad. För att avgöra vilka värden på statistikorna som är godtagbara behövs CDF:erna för de två versionerna av testet. För den första versionen [22] är CDF:en

$$\mathrm{F}_K\left(X\right) = 1 - \mathrm{e}^{-2X^2}, \; x \geq 0 \tag{2.18}$$

för tillräckligt stora värden på N. För den andra versionen av testet [24] är CDF:en

$$\mathrm{F}_D(X) = \frac{\sqrt{2\pi}}{X} \sum_{i=1}^{\infty} \mathrm{e}^{-(2i-1)^2\pi^2/\left(8X^2\right)}. \tag{2.19}$$

Summan konvergerar fort — vid användning av $\mathrm{F}_D(X)$ bör de 10 första iterationerna räcka.



Figur 2.8: Figuren visar CDF-funktionen av D respektive K-statistikan för KS-fördelningen i intervallet $[0, 2]$. Teststatistikor inom de skuggade områdena har CDF-värden under 10% eller över 90%.

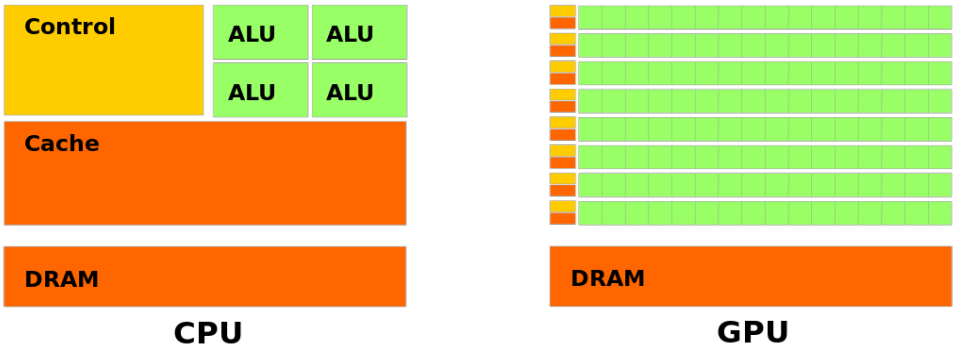
I Figur 2.8 visas CDF:en för K-teststatistikorna (F_K) samt D-statistikan (F_D), det är även markerat vilka statistika värden som motsvarar CDF-värden under 10% respektive över 90%.

Precis som χ^2-testet så har finns det även för- och nackdelar med KS-testet. Testet är intuitivt då den bygger på den maximala skillnaden mellan mätdatans CDF och den underliggande distributionens CDF. Den tar också hänsyn till all mätdata, till skillnad från χ^2-testet. CDF-funktionen beror endast på teststatistikan, till skillnad från χ^2-testet där fördelningen även beror på ν. KS-testets styrka är oberoende av N, och är också enkelt att implementera. CDF-funktionerna i Ekvation 2.18 och 2.19 är korta och lätta att beräkna. Teststatistikorna är också simpla att beräkna. Den enda stora nackdelen med KS-testet är att den kräver CDF-värden vid signifikansgränserna.

3

GPU-teori

En grafikprocessor (GPU) är en typ av processor som specialiserar sig på parallella beräkningar med fokus på flyttal (decimala tal). Den används främst till grafik, bildbehandling och andra aritmetiskt tunga applikationer där ett stort antal beräkningar kan ske oberoende av varandra, så kallad massivt parallelliserbara arbeten. Program inom dessa områden kan få en betydlig prestandaförbättring genom att köras på en GPU jämfört med en CPU, som har fokus på generell exekvering och snabb minnesåtkomst i form av ett höghastighetsminne som ligger på processorn (cache). En GPU har betydligt fler beräkningsenheter jämfört med en CPU, där mycket plats istället tas upp av höghastighetsminnet [25]. Skissen i 3.1 visar skillnaden i design mellan dessa processorer. Detta avsnitt behandlar endast GPU-tillverkaren Nvidias GPU-arkitektur, då det är deras ramverk och grafikkort vi använder i detta arbete.



Figur 3.1: Bilden visar den konceptuella skillnaden i arkitekturen av en CPU kontra GPU. Beräkningsenheter är gröna och minnesenheter röd-orange. ALU står för "Arithmetic Logic Unit", vilket är en enhet som utför aritmetiska beräkningar.

Från början var GPU:er endast anpassade för beräkningar relaterade till grafik, men intresset växte för andra användningsområden som simulering och bildbehandling. Det visade sig att sådana beräkningar var mycket parallelliserbara. Detta ledde till att Nvidia anpassade sina grafikkort för generella beräkningar och introducerade CUDA, en arkitektur och medföljande ramverk som möjliggör generell programmering för exekvering på Nvidias GPU:er, så kallad *General-purpose computing on graphics processing units* (GPGPU).

Den minsta enheten som exekveras på en GPU kallas *tråd*, och dessa grupperas tillsammans i så kallade *block*. Trådar i samma block har tillgång till ett gemensamt minne vilket

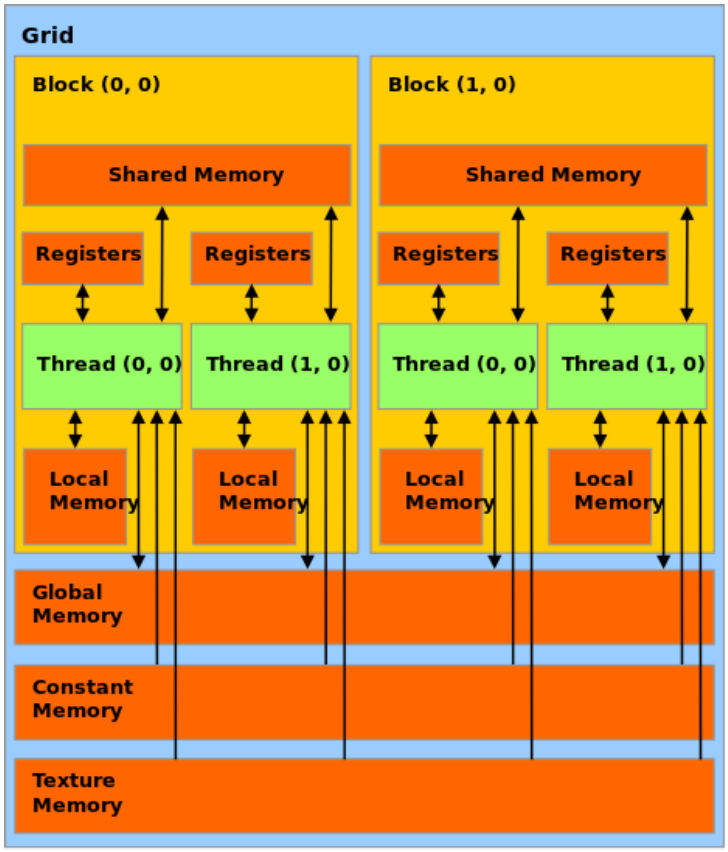
underlättar för samarbete mellan trådarna. Dessutom finns det ett max antal trådar som får vara i ett block, så om man vill använda fler trådar måste man också dela upp dem i fler block.

För att programmera en GPU så behöver speciella ramverk användas, som Cuda eller OpenCL. Dessa ger funktionaliteten att skriva och exekvera *kernels*, vilket är funktioner som kör på en GPU. GPU kallas även *Device* i Cuda-världen, och CPU kallas *Host*. Cuda använder sig av en Nvidias egna kompilator, NVCC, som kompilerar den GPU-specifika koden och låter en annan kompilator, exempelvis GCC, omvandla CPU-koden. Det betyder alltså att programmeraren tydligt måste markera vilka funktioner som är *kernels*, vilket görs genom att skriva `__global__` i början av typsignaturer. Hjälpfunktioner till en kernel taggas med `__device__`. Dessa kan även exekveras på CPU:n om inga Cuda-specifika kod används och taggen `__host__` läggs till i typsignaturen [25]. Anledningar till att man skulle vilja exekvera en funktion på både CPU och GPU är om man vill jämföra prestanda mellan dem, och då vill man använda samma algoritm.

Nvidias GPU:er har egna arkitekturspecifika assemblerspråk, där nyare arkitekturer stödjer fler funktioner. Cuda definierar därför också ett arkitekturoberoende assemblerspråk, PTX, som kan översättas till assemblerkod specifik för GPU:n man kör på [26].

GPU och CPU använder olika minnen så om en kernel behöver startvärden så behöver dessa flyttas från CPU- till GPU-minne först. Cuda är designat så att en pekare till minne på CPU ser exakt likadan ut som om den hade pekat på GPU-minne. Man måste därför vara extra försiktig med hur man använder sina pekare så man inte råkar använda fel minne. En CPU-pekare allokeras precis som i vanlig C/C++, för allokering av minne på GPU:n behövs Cuda-specifika funktioner som exempelvis `cudaMalloc`.

När kod exekveras är det extra viktigt att minska minnesanvändning då det i jämförelse med beräkningshastighet tar lång tid att transportera data mellan minnesenheter och beräkningsenheter. Detta är också anledningen till att minnet är delat upp i en hierarki, där snabbare, men mindre minnesenheter ligger närmare beräkningsenheterna och större längre bort. Denna minneshierarki kan ses i Figur 3.2. *Register* ligger närmast trådarna och är också snabbast, med nackdelen att mängden minne är väldigt begränsat. *Shared Memory* används för att dela på data mellan flera trådar i ett block - variabler av denna typ taggas med `__shared__`. När minne allokeras med funktionen `cudaMalloc` så hamnar datan i *Global Memory* som alla trådar kan läsa från och skriva till. En variant som är lik Global Memory är *Local Memory*, som används när lokala variabler inte får plats i register. Det finns också *Constant Memory* och *Texture Memory* som trådar endast kan läsa från (*read only*) med effekten att flera trådar asynkront kan läsa samma data utan problem [25]. På moderna Nvidia-GPUer ligger Global, Local, Texture och Constant memory i samma fysiska minne, *Device Memory*.



Figur 3.2: Denna figur visar den minneshiearki som finns i en GPU. Minnen som *Register* och *Shared Memory* har snabbare minnesåtkomst än till exempel det större *Global Memory*. Dessutom kan man se vilka minnen som olika trådar har åtkomst till, där alla trådar i ett block kommer åt *Shared Memory*, men inte varandras *Register*. Trådar är gröna och minnesenheter orange. Den ljusare orange avgränsar varje block.

3.1 Hårdvarudetaljer

En GPU består av ett flertal *Streaming multiprocessors* (SM) som exekverar ett eller flera block samtidigt. När ett block har kört klart så körs ett nytt. Arkitekturen för en SM kallas SIMT (Single-Instruction, Multiple-Thread) som möjliggör exekvering av hundratals trådar för varje SM. SM delar upp varje block i *warps*, vilket är en samling av 32 trådar. Om ett block är av storleken 128 trådar så delas den upp i 4 warps [25].

Ett warp exekverar en instruktion i taget, vilket innebär att alla trådar som ska exekvera instruktionen kör samtidigt, medan resten väntar. Om till exempel kroppen i en if-sats exekveras av hälften av trådarna så väntar resten. Det är därför mest effektivt att undvika förgrening (branching) eller att låta alla trådar i ett warp ta samma gren [25].

SM:s har ett maximalt antal warps som beror på GPU-arkitekturen. *Occupancy* defineras som antalet aktiva warps i ett SM delat med hur många den maximalt har. När ett warp pausar och inte kan få nya instruktioner från SM så är det viktigt att det finns ett annat warp att ge instruktioner till. Det betyder att ju fler aktiva warps det finns, desto mer effektivt blir utdelningen av instruktioner och därmed fås bättre prestanda [27].

3.2 Slumptalsberäkningar på grafikprocessorer

Slumptalsgenerering på GPU:er är intressant då beräkningar för tidigare nämnda användningsområden (se Avsnitt 2) ofta sker på denna hårdvara, och det är då gynnsamt att även producera slumptalen där. Vissa PRNG:er är anpassade för parallell exekvering, vilket är särskilt användbart på GPU:er då de är speciellt anpassade för just sådan exekvering. En GPU kan köra tusentals trådar samtidigt, som då alla kör sin egen generator.

Idag används ett flertal olika generatorer på GPU:er, som lämpar sig för olika ändamål beroende på önskad prestanda, kvalité och minnesanvändning. Det finns generatorer som specialiserar sig på kryptografisk säkerhet, där det ska vara så svårt som möjligt att kunna förutspå slumptalsgenerationen och därmed utnyttja denna information till sin fördel. Detta arbete fokuserar dock inte på slumptalsgenerering av den kvalitén, men använder sig av några algoritmer inspirerade av kryptografiska tekniker.

Nvidia har ett antal olika färdiga generatorer i sitt `curand`-bibliotek [28] som kan användas beroende på ändamål och krav. Bland dessa finns MTGP (se Avsnitt 2.2) och Philox (Avsnitt 2.3).

4

Metod

Ett antal utvalda slumptalsalgoritmer implementerades i ett egetbyggt objektorienterat ramverk, dessa har sedan testats med egenimplementerade tester för slumptalskvalitén och prestandaanalyserats med Nvidias verktyg `nvprof` och `nvvp`.

4.1 Engine-Factory-ramverket

För att enkelt kunna testa alla generatorer och enkelt kunna byta mellan olika har ett ramverk använts där varje tråd skapar ett egen *engine*-objekt (*generator*) ur en huvudklass som är en *fabrik* (*factory*).

I början av varje programkörning allokeras plats i grafikkortets globala minne (*Global Memory*) för tillstånd och andra variabler som behövs lokalt vilket görs genom en initialiseringsfunktion. Varje fabrik har en `seed`-funktion för att fylla det globala minnet med starttillstånd för varje generatorobjekt. Denna funktion visas i Listing 4.1 med LCG som exempelgenerator. Sedan kan varje tråd hämta data genom att ladda in en referens till den globala datan och använda sitt tråd-ID för att hämta sitt tillstånd.

Listing 4.1: Funktionen `seed` för LCG. Skriver i Cuda C++.

```
template<class tSeq> inline
void seed(LCGGlobalData& aData, tSeq& aSeq, std::size_t aThreadCount){

  // Temporary buffer
  std::vector<std::uint64_t> buf(aThreadCount);

  // State size
  const std::size_t bytes = aThreadCount * sizeof(std::uint64_t);

  // Generate new state and move to GPU memory
  std::generate(buf.begin(), buf.end(), aSeq);
  cudaMemcpy(aData.states, buf.data(), bytes, cudaMemcpyHostToDevice);
}
```

Varje kernel behöver ladda in en egen kopia av generatorn genom att kalla på en *factory* `engine_state_load`-funktion som returnerar ett generatorobjekt. Dessa objekt kan sedan enkelt kallas på för att returnera ett slumptal genom en definierad `operator()`-funktion

där algoritmen för generatorn är implementerad. En exempelimplementation i Cuda C++ för denna generator finns i Listing 4.2.

Listing 4.2: Funktionen `operator()` för LCG.

```
__device__ inline
auto LCGEngine::operator() (const LCGGlobalData&) -> result_type {

  // Update state
  this->state = (LCG::a * this->state + LCG::c) & LCG::m;

  // Return number
  return (std::uint32_t)(this->state >> 16u);

}
```

Vid slutet av en kernels körning kan tillståndet som generatorn har sparas genom att en `engine_state_store`-funktion används. Denna funktion flyttar över tillståndet som ligger internt hos tråden till det globala minnet. Vid en ny körning av kerneln kan ett nytt engine-objekt använda den globala datavariablen och kernelns tråd-ID för att fortsätta körning från det sparade tillståndet.

Genom att låta definiera ett `typename result_type` kan datatypen som returneras snabbt ändras för exempelvis 32-bitars respektive 64-bitars heltal för de generatorer som stödjer det.

4.2 Implementationsdetaljer för generatorer

Då generatorerna skiljer sig åt och på olika sätt använder grafikprocessorns resurser följer nedan detaljer för de olika implementeringarna. Alla uniformt fördelade generatorer producerar 32-bitars tal i våra implementationer.

4.2.1 Philox och Threefry

Då både dessa generatorerna är räknebaserade CBRNG, så ser dess implementationer väldigt lika ut bortsett från själva slumptalsalgoritmen. Vår implementation stödjer endast 32-bitars integer-generering av fyra tal åt gången för att totalt generera 128 bitar slumptal vid varje anrop till generatorn. Generatorobjekt seedas från början med samma tillstånd (ett heltal) till alla generatorer, vilket fungerar eftersom varje generator använder sitt tråd-ID för att skapa nycklar som resulterar i att varje tal blir unikt [6]. Alla konstanter sparas som `const` för att kompilatorn ska kunna optimera bort minnesintruktioner och använda talen direkt.

Eftersom båda algoritmerna genererar fyra tal åt gången kan efter ett anrop tre stycken sparas undan och genom kontrollsatser returneras direkt vid följande anrop. Detta ger inga problem med warps, eftersom så länge en multipel av 32 tal returneras kommer alla trådar gå genom samma kodgren. När fyra tal sedan har returnerats körs ett nytt pass igen.

Philox-generatorn använder *intrinsic*-funktionen `_umulhi` för att plocka de övre 32 bitarna av en 32-bitars integer-multiplikation, detta resulterar i att en onödig konvertering till 64 bitar och tillbaka kan undvikas.

4.2.2 Ziggurat

Ziggurat-implementeringen är en rak implementering av algoritmen beskriven i Avsnitt 2.4.2 med få optimeringar, för mer detaljer se Marsaglias originalpapper [17].

Ziggurat-metoden är mycket snabbare än både Box-Muller-transformen då den implementeras på traditionella processorer [17] men dock ej här [5]. En avgörande nackdel är som tidigare nämnt att algoritmen bara går optimalt $97,98\%$ av fallen och i resterande fall behöver ta en längre väg för att generera ett tal. Prestandavinsten försvinner på grund av grafikkortets *warp*-struktur då sannolikheten är $1 - 0,9798^{32} = 47,95\%$ att en tråd i ett warp tar en längre väg genom algoritmen, så att resten av warpet behöver vänta.

4.2.3 Wallace Gaussian Generator

Wallace-implementationen har sitt tillstånd i *Shared Memory* vilket innebär att varje block har en egen pool. Denna variant är betydligt snabbare än en implementation där alla trådar delar på poolen som i sådana fall skulle använda sig av *Global Memory* (se Avsnitt 3). Dessutom sparas talen i grupper om fyra, för att effektivisera hämtningen från minnet.

För matris-transformeringen finns fyra olika matriser som väljs slumpmässigt av trådarna. Viktigt att notera är att alla trådar i ett warp alltid väljer samma matris. Det leder till hälften så lång körtid jämfört med en variant där varje tråd själv väljer matris.

Dessa matrisoperationer innebär tyvärr prestandaproblem om man inte är försiktig. Skaparen av algoritmen, C. S. Wallace, föreslår att sätta dimensionen $K = 4$ och använda Walsh-Hadamard-matriser för att minska beräkningarna. Sådana matriser har egenskaperna att de är ortogonala, dimensionerna är tvåpotenser, och alla element är ± 1. Med matrisdimensionerna 4×4 så blir en matris-vektor-multiplikation endast 7 additioner/-subtraktioner och en multiplikation med 0.5 [21] - ett exempel visas på nästa sida.

$$
\begin{aligned}
[ht]A_1 &= \frac{1}{2}\begin{bmatrix} +1 & +1 & -1 & +1 \\ +1 & -1 & +1 & +1 \\ +1 & -1 & -1 & -1 \\ -1 & -1 & -1 & +1 \end{bmatrix} \\
X_i &= \begin{bmatrix} a_i & b_i & c_i & d_i \end{bmatrix}^t \\
X_{i+1} = A_1 X_i &= \begin{bmatrix} a_{i+1} & b_{i+1} & c_{i+1} & d_{i+1} \end{bmatrix}^t \\
s &= \frac{1}{2}(a_i + b_i + c_i + d_i) \\
a_{i+1} &= s - c_i \\
b_{i+1} &= s - b_i \\
c_{i+1} &= a_i - s \\
d_{i+1} &= d_i - s
\end{aligned}
$$

Denna metod innebär alltså att de fyra matriserna i implementationen är varianter av matrisen A_1 i exemplet ovan. Eftersom $K = 4$ så tar varje tråd i implementationen 4 tal från poolen för att applicera på någon av matriserna.

Ett problem är hur talpoolen permuteras på ett effektivt sätt. Om man flyttar runt alla talen på ett slumpmässigt sätt för att få en permutation så skulle det vara väldigt ineffektivt. Implementationen av Lee Howes och David Thomas [5] löser detta med en annan teknik som grundar sig på att alla trådar har unika id. Om $K = 4$ så har poolen storleken $4 \times$ antal trådar, och man låter då alla trådars id förändras med en slumptalsgenerator, i detta fall en LCG (Avsnitt 2.1) med perioden lika med antal trådar. Det är då garanterat att varje tråd inom detta intervall får ett unikt index, vilket tas ur poolen. Detta upprepas fyra gånger för att hämta de fyra slumptal varje tråd behöver, och vid varje gång så anpassas indexet också till vilken del av poolen man hämtar från. Om man till exempel tolkar poolen som K rader och L kolumner så skulle beräkningen se ut som följande: LCG(id) + ($i \times$ antal trådar), där i är nuvarande omgång i hämtningen [5].

Med denna metod så kan man dock inte nå alla möjliga permutationer, för om poolen har storlek n så finns det $n!$ permutationer. Metoden kan dock bara nå: $4! \times$ antalet trådar. Detta beror på att för varje omgång får en tråd endast ta ett av talen, men för att kunna nå alla permutationer så måste en tråd kunna ta flera tal i följd. Det enklaste exemplet är när talen för tråd 1 är de på plats 0, 1, 2, och 3 i poolen, och fortsatt för nästa tråd. Denna metod används därför tillsammans med transponeringen för att säkra slumptalskvalitén.

C. S. Wallace påpekar i sin publikation [21] att poolens egenskap att den håller N(0,1) efter tid börjar försvinna. I hans implementation så är det på grund av att heltalsaritmentiken inte är helt exakt, och i detta fallet är det liknande problem med flyttalsaritmetik. Därför rekommenderar han att man skapar en ny pool var 2000:e passering för att behålla egenskaperna. Denna pool kommer från en annan generator, i vårt fall en Mersenne Twister + Box Muller på CPU:n.

Han föreslår dessutom att man använder χ^2-korrektion för att behålla fördelningen. Denna korrektion beräknas enligt ekvationen nedan [29], där n är ett slumptal ur N(0,1) och ν är storleken på poolen. Variabeln s multipliceras sedan med varje tal i poolen. Detta utförs varje genomgång.

$$\chi^2_\nu = \frac{(n + \sqrt{2\nu - 1})^2}{2}$$
$$s = \sqrt{(\chi^2_\nu / \nu)}$$

Det finns ett antal skillnader mellan implementationen som Lee Howes och David Thomas har gjort och vår. Den mest signifikanta är att deras implementation endast transformerar med en matris medan fyra olika matriser används här, vilket teoretiskt ska leda till bättre slumptalskvalité. Eftersom deras variant är mer pseudokod än en riktig implementation så saknas vissa detaljer som omöjliggör en ordentlig jämförelse mellan implementationerna.

4.2.4 Curands generatorer

De Curand-generatorer vi har valt att inkludera är *MTGP* och *Philox*. De har anpassats till det ramverk som resterande är skrivna för (se Avsnitt 4.1), för att få egenskapen att tillståndet bevaras mellan kernelkörningar även för dessa.

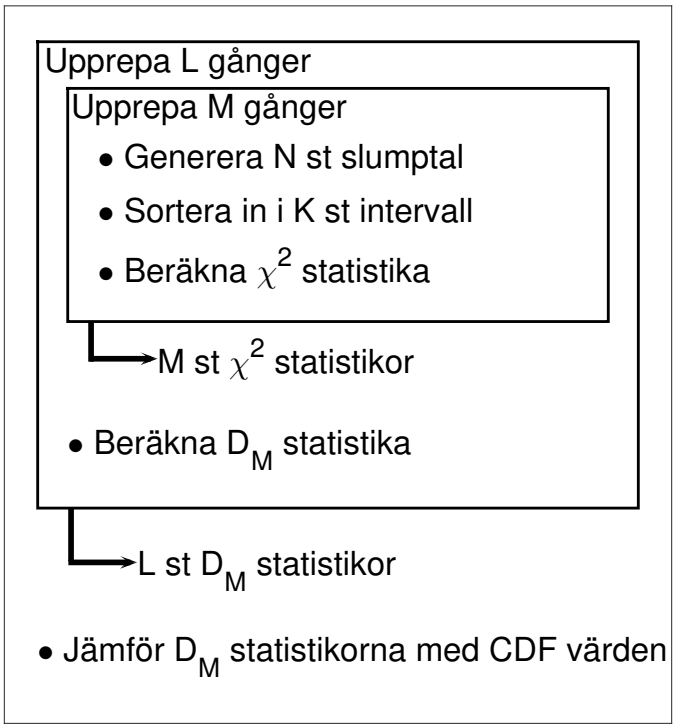
Curands egna ramverk använder sig inte av den objektorienterade paradigmen, som EngineFactory, utan består istället av en samling funktioner, där den viktigaste är funktionen `curand`, som returnerar ett slumptal utifrån det tillstånd som anges som argument. De båda generatorerna initialiseras på olika sätt, där Philox använder sig av funktionen `curand_init` medan MTGP behöver andra funktioner där man specificerar långa strukturer av konstanter.

Den implementation av MTGP som Curand har gjort bör inte användas på fler än 200 block och 256 trådar per block som standard. Detta beror på storleken på de parametrar som följer med biblioteket. Det är möjligt att skapa sina egna parametrar och använda istället [28].

4.3 Implementation av tester

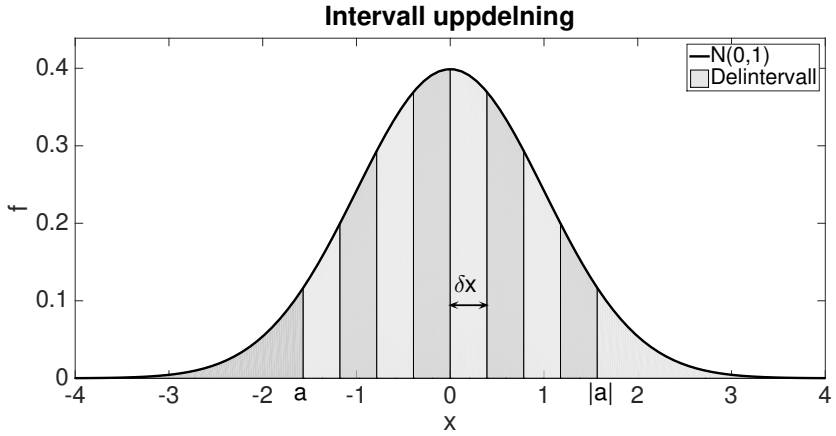
I följande stycke beskrivs implementationen av de statistiska tester som utförts samt vilka approximationer som gjorts. Algoritmerna för testerna gås igenom mer noggrant i detta stycke än vad de gjordes tidigare i Avsnitt 2.5.1 och 2.5.2.

4.3.1 Chi-kvadrat kombinerat med Kolmogorov-Smirnov



Figur 4.1: En förenklad schematisk beskrivning av algoritmen för ett χ^2-KS-test.

Figur 4.1 visar en schematisk bild över de viktigaste delarna av algoritmen för det kombinerade χ^2-KS-testet.



Figur 4.2: Grafen visualiserar hur intervallen valts. För intervallen i figuren är $K = 10$.

Val av intervall över vilket slumptalen sorteras är inte specifikt för algoritmen och kan väljas godtyckligt. För detta projekt har ett intervall valts vars utseende motsvaras av det i Figur 4.2. Axeln delas in i $K - 2$ delintervall vardera med bredden δx, de två resterande

delintervallen sträcker sig mellan $\pm a$ och $\pm\infty$ på respektive sida. δx väljs så att $p_1 = p_2 = p_{K-1} = p_K$. För att finna δx används en iterativ metod för att lösa ekvationen

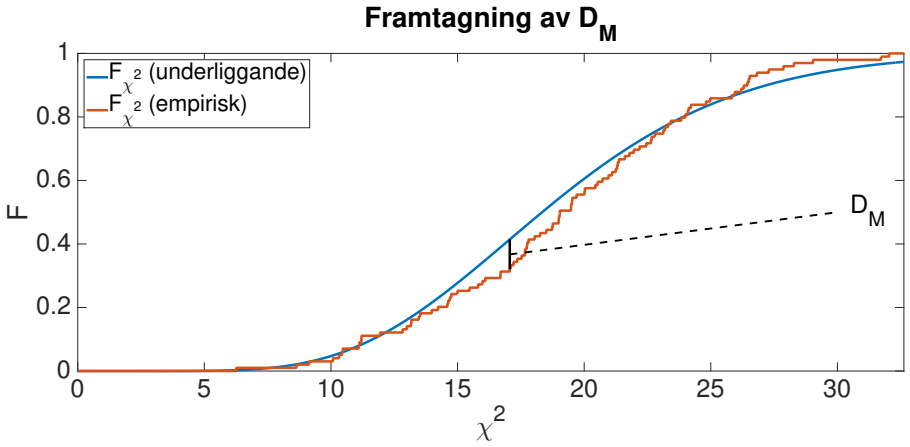
$$2F_N\left(a|\mu,\nu\right) - F_N\left(a+\delta x|\mu,\nu\right)dx = 0$$
$$a = \frac{\delta x\left(2-K\right)}{2}$$

där $F_N\left(x|\mu,\nu\right)$ är CDF-funktionen för en normalfördelning med parametrarna μ och ν. I implementationen har en approximation av CDF-funktionen använts

$$F\left(x|0,1\right) = \frac{1}{1+\mathrm{e}^{-\mathrm{p}(x)}}$$
$$\mathrm{p}\left(x\right) = 1.59145x + 0.01095x^2 + 0.06651x^3$$

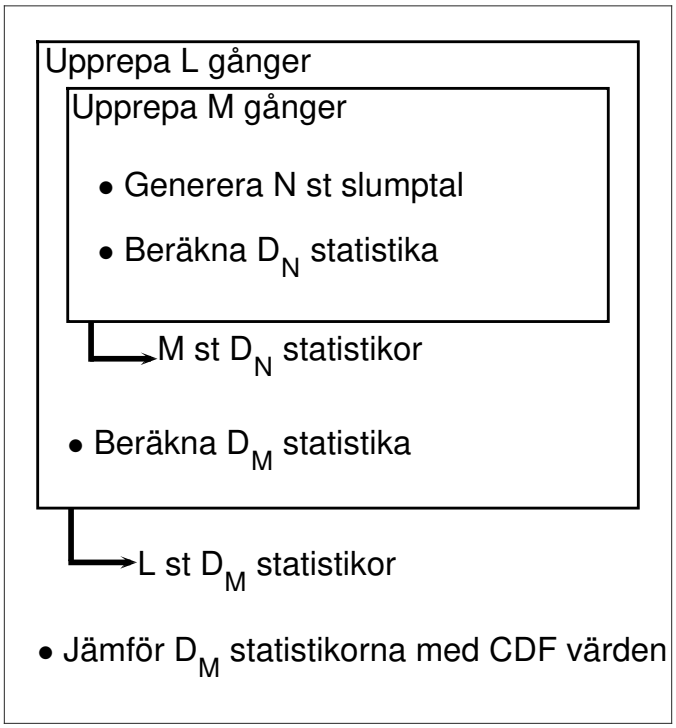
som gäller för $\mu = 0$ och $\nu = 1$.

Efter indelning av slumptalen i delintervallen beräknas χ^2-statistikan uifrån Ekvation 2.8. Då M χ^2-statistikor beräknats tas den empiriska CDF-funktionen fram och jämförs med den underliggande χ^2-fördelningens CDF-funktion, detta ses i Figur 4.3. D_M-statistikan beräknas och proceduren upprepas till L D_M-statistikor tagits fram. Dessa jämföres med CDF-funktionen för KS-fördelningen och det avgörs hur många D_M-statistikor som ligger utanför den satta signifikansnivån. När testet är klart erhålls en procentsats för hur stor andel av de L stycken D_M-statistikorna som ligger utanför signifikansnivån, denna procent används för att avgöra kvalitén av generatorn testet körts för. En annan kvalitetskontroll är att visuellt granska jämförelsen mellan en mängd av χ^2-statistikors empiriska CDF-funktion med den underliggande fördelningens CDF-funktion så som gjorts i Figur 4.3.



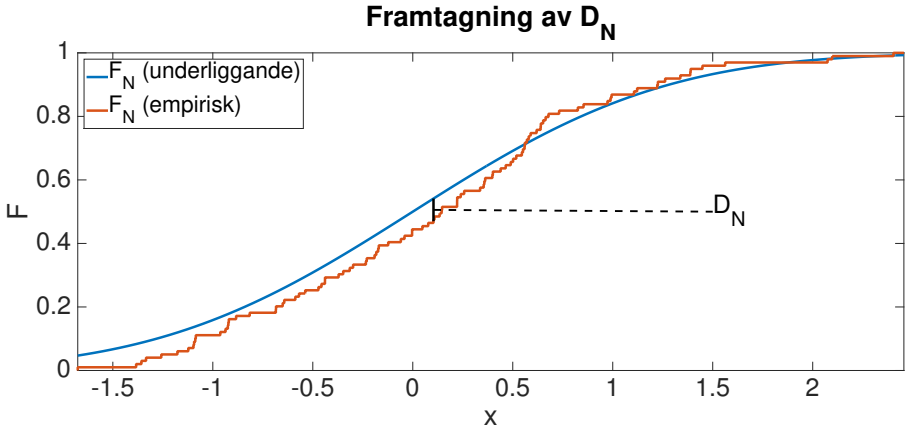
Figur 4.3: En plot över den empiriska χ^2-CDF-funktionen och den underliggande CDF-funktionen. χ^2-statistikorna är skapade utifrån slumptal genererade med *minstd_rand*0 från C++ biblioteket `<random>`.

4.3.2 Dubbelt Kolmogorov Smirnov



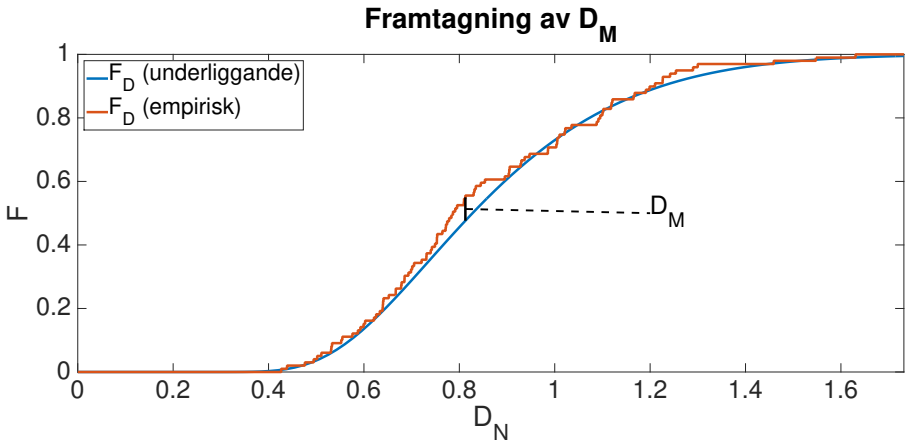
Figur 4.4: En förenklad schematisk beskrivning av algoritmen för ett KS-KS test.

KS-KS-testet är likt χ^2-KS-testet på många sätt, skillnaden är att slumptalen inte sorteras in i intervall för vilka en χ^2-statistika beräknas utan istället tas en D_N statstika fram för varje set av N slumptal, detta ses i Figur 4.5.



Figur 4.5: Grafen visar en jämförelse mellan den empiriska CDF-funktionen (röd linje) för 100 normalfördelade slumptal genererade med *minstd_rand*0 från `<random>` biblioteket, samt CDF-funktionen för den underliggande normalfördelningen (blå linje). Den framtagna D_N-statistikan är utmärkt.

Då M D_N-statistikor tagits fram beräknas en D_M-statistika (se Figur 4.6). På samma sätt som för χ^2-KS-testet upprepas processen tills L stycken D_M-statistikor erhållits. När testet är klart används andelen D_M-statistikor som ligger utanför det godtagbara området som ett mått på slumpgeneratorns kvalité. På samma sätt som för χ^2-KS-testet kan viss information ges ifrån att granska jämförelsen mellan den empiriska CDF-funktionen för slumptalen och CDF-funktionen för den underliggande normalfördelningen så som gjorts i Figur 4.5.



Figur 4.6: Grafen visar jämförelsen mellan den empiriska CDF-funktionen (röd linje) för 100 D_N-statistikor, samt CDF-funktionen för den underliggande normalfördelningen (blå linje). Den framtagna D_M-statistikan är utmärkt.

5

Resultat och Analys

Prestandamätningar har gjorts på tre olika grafikkort: GTX 1060, GTX 780 och Tesla K40. Det sistnämnda grafikkortet ligger på C3SE:s (Chalmers Centre for Computational Science and Engineering) beräkningskluster *Hebbe*. Mätningarna med GTX 1060 samt Tesla K40 har körts på GNU/Linux-baserade system medan GTX 780 har körts på Windows 10. Kompilatorinställningarna har valts för maximal prestanda (flaggan -O2 har getts vid kompilering). Alla mätningar har gjorts ett antal gånger för att ge pålitliga resultat.

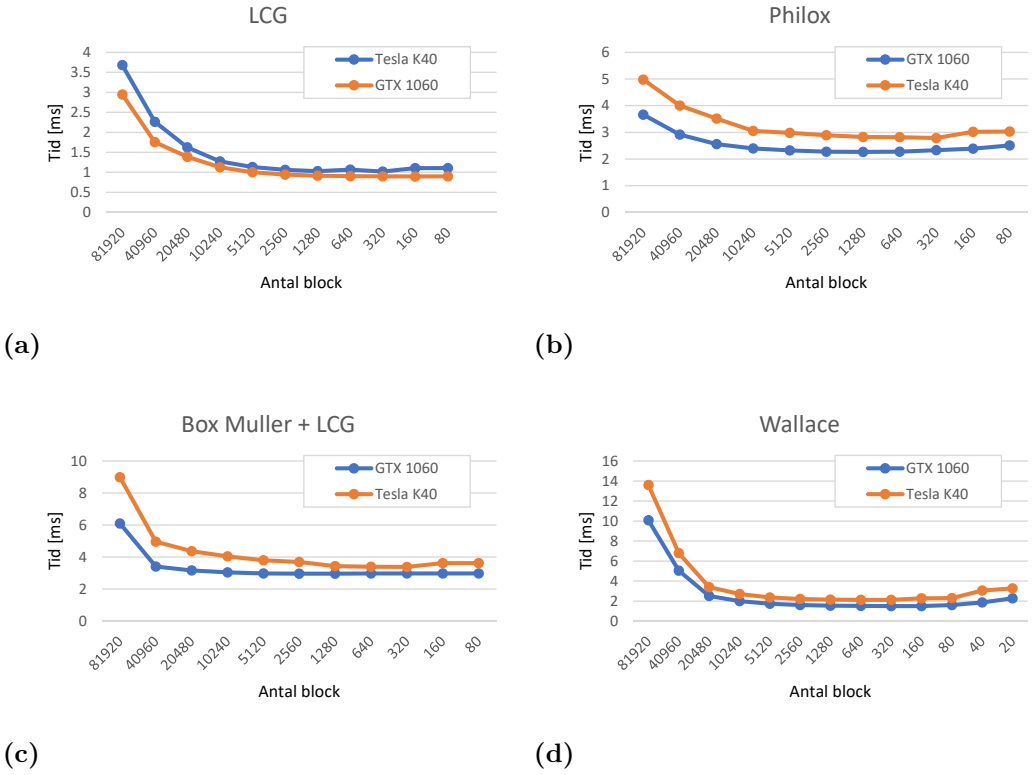
Grafikkort	**Kompilator**	**NVCC-version**
GTX 1060	GCC g++ 4.9.4	8.0.44
Tesla K40	GCC g++ 5.4.0	8.0.44
GTX 780	Visual C++ 19.0	8.0.44

För att reflektera över olika generatorers prestanda så behövs mer information. Denna kan fås genom att köra koden med Nvidias profileringsprogram *nvprof* samt *nvvp* (Nvidia visual profiler) [30]. *nvvp* låter användaren analysera prestanda för en enskild kernel och avläsa parametrar som occupancy, antal minnesåtkomster, anledningar till att exekveringen av kerneln stannar upp (stalls), och fördelningen av olika sorters instruktioner. Det är också utifrån denna information som vi har optimerat generatorerna.

5.1 Val av konfiguration

Prestanda för de olika generatorerna uppskattas med ett flertal olika mätningar med olika konfigurationer. Generatorer som använder shared memory (Wallace) är de enda som påverkas av blockstorleken relativt mängden block. Detta förutsätter dock att blockstoleken är en multipel av 32, så att hela warps används. Vi har därför valt att köra 1024 trådar per block och sedan anpassa antalet block för prestandamätningar. För MTGP finns det dock en övre begränsning i antalet block och trådar (se Avsnitt 4.2.4 så vi låter varje generatorinstans arbeta extra för att totalt generera lika många tal på en körning.

För att jämföra generatorernas prestanda med varandra mäter vi när endast slumptalsgeneratorna körs på kerneln, detta för att eliminera andra faktorer som skulle kunna påverka mätresultaten. Eftersom slumptalsgeneratorer oftast används inbäddade i annan kod har vi även mätt körningstiden på en simulationskod för att se hur snabba våra implementeringar är i ett mer praktiskt scenario.



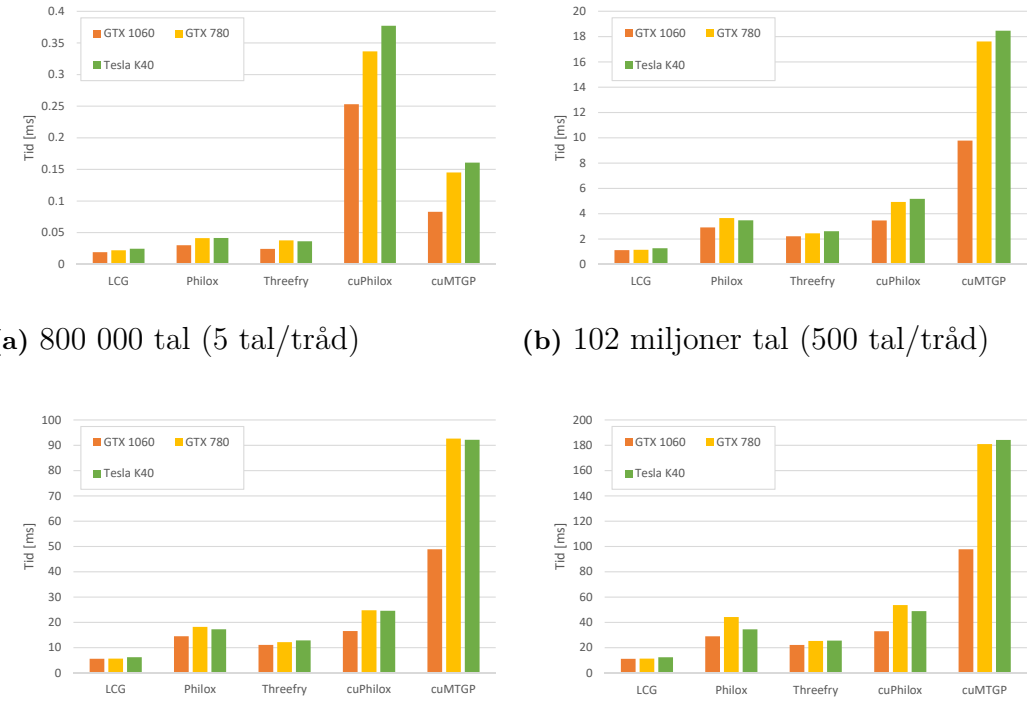
Figur 5.1: Graferna visar hur prestandan skiljer sig när mängden trådar jämfört med mängden tal varje tråd genererar har ett inverst förhållande. Varje punkt i varje graf representerar tiden det tar att generera $1024 \times 81920/x \times x = 83886080$ trådar, där x är antalet block.

Vi mäter kernel-körtid i alla mätningar, mätningen börjar precis innan kernel startar och avslutar när den är klar. Alla mätningar har gjort flera gånger och resultaten som visas är ett medelvärde av dessa.

I Figur 5.1 mäts tiden det tar att generera en fast mängd tal med ett inverst förhållande mellan totala mängden trådar och antal tal varje tråd genererar (antalet iterationer). Från detta kan vi se hur körningstiden påverkas av blockmängden kontra hur mycket arbete varje enskild tråd måste jobba för att generera totalt sett samma mängd slumptal. Detta görs för några olika generatorer för att få en uppfattning om vilken konfiguration är rimlig att köra de följande testerna i.

Graferna i Figur 5.1 visar att så länge inte ett program startar väldigt många trådar som gör genererar väldigt få tal så spelar det ingen större roll för prestandan vilken konfiguration man väljer. Det omvända gäller också, där grafikkortet behöver en viss mängd trådar att jobba med för optimal prestanda. Den nedre gränsen av mängden trådar som behövs för bra prestanda beror även på vilken grafikprocessor som används då olika GPU-er kan exekvera olika mängder trådar samtidigt. Detta eftersom grafikkort på marknaden presterar olika beroende på segment, ofta genom att ha färre antal SM för svagare kort.

Utifrån resultatet i Figur 5.1 har vi valt att göra de fortsatta mätningarna i en konfiguration om 200 block och 1024 trådar per block, förutom Curands MTGP som på grund av



(a) 800 000 tal (5 tal/tråd)

(b) 102 miljoner tal (500 tal/tråd)

(c) 512 miljoner tal (2500 tal/tråd)

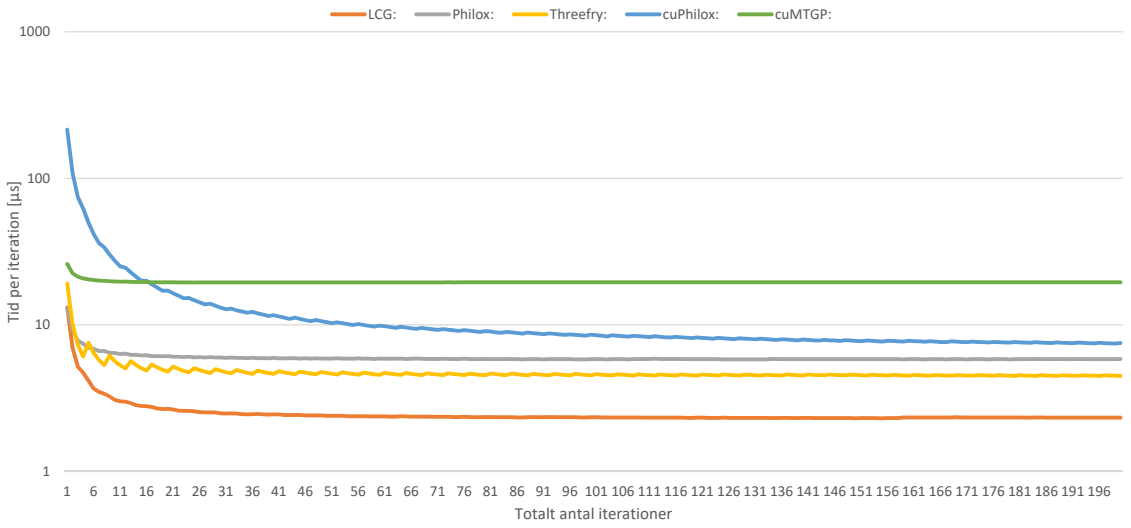
(d) 1 miljard tal (5000 tal/tråd)

Figur 5.2: Graferna visar hur lång körningstid det är för att generera respektive mängd slumptal för en kernel. Konfigurationen är 200 block och 1024 trådar per block. Resultaten är medelvärden över 5000 mätningar.

dess implementation istället kommer köra 256 trådar per block och generera fyra gånger så många tal per tråd. Valet av 200 block gjordes eftersom prestandan för alla generatorerna överlag varierar väldigt lite med mängden block och MTGP-implementationen inte bör köras med fler [28]. Vi kommer istället variera antalet slumptal varje tråd genererar för att jämföra prestandan i olika scenarion.

5.2 Uniformfördelade generatorer

De inledande testerna som ses i Figur 5.2 visar att alla grafikkorten har liknande prestanda, där GTX 1060 i genomsnitt presterar något bättre än de övriga. GTX 780 presterar i sin tur något bättre än Tesla K40, som den delar arkitektur med. Vi kan se att Curands implementation av Philox presterar väldigt dåligt när få tal genereras per tråd men verkar bli mer prestandaeffektiv med fler tal per tråd. På grund av detta väljer vi att analysera hur mängden tal per tråd påverkar prestandan. Eftersom prestandaskillnaderna är liknande för de olika generatorerna har vi valt att göra följande prestandanalyser och köra simulationskoden på GTX 1060. Mätningarna för GTX 780 har varit något opålitliga medan de andra GPU:erna har varit väldigt stabila över mätningarna, vi tror det har att göra med att Windows använder grafikkortet samtidigt.



Figur 5.3: Grafen visar tid per iteration för uniformt fördelade generatorer när mängden iterationer varieras. Tiden t för ett antal n iterationer är justerat med t/n. Tiden för varje punkt på grafen är ett medelvärde av 5000 körningar.

Som man kan se i Figur 5.3 så är Curand/Philox minst prestandaeffektiv i början, men kommer ikapp MTGP ju fler tal som genereras. Detta tyder på att startkostnaden för att börja producera tal med MTGP är lägre än Curand/Philox men den arbetar mer långsamt och blir därför av med sitt prestandaövertag med tiden. Man kan se att Threefry är hackig, det beror på att algoritmen är extra effektiv vid multiplar av fyra. Detta är en konsekvens av implementationen som beskrivits närmare i Avsnitt 4.2.1. Det är intressant varför Philox inte visar samma beteende då den fungerar på samma sätt, vi har dock ingen bra förklaring till detta.

Den snabbaste generatorn är LCG, som både har lägst startkostnad och är fortsatt snabbast oavsett mängden tal som genereras. Detta beror delvis på att tillståndet utgörs av endast ett tal, vilket får plats i ett register. Dessutom är det väldigt få aritmetiska instruktioner. För extra prestanda kan man byta till en annan LCG-variant som inte shiftar bort bitar, och då bli av med en instruktion, men med bieffekten att kvalitén påverkas negativt av de lägre bitarna, som får vara kvar (se Avsnitt 2.1).

Bland de snabbare generatorerna finner vi dessutom Philox och Threefry. Det beror på att de är aritmetiskt tunga, och arbetar med minimalt minne. För Philox är 85% av alla instruktioner heltalsaritmetik, och för Threefry är det 75%. Båda har också hög *occupancy*, där Philox har 89,2% och Threefry 91,5%. Det är så bra att en av de stora anledningarna till att ett warp pausar (väntetid), är att det är redo att exekvera men SM ger order till ett annat warp just då. För Philox är detta anledningen till hela 18% av all väntetid. I beskrivningen av denna typ av väntetid (*not selected*) så påpekar Nvidia att man till och med kan sänka occupancy för att höja prestanda [30].

Det finns en betydande prestandaskillnad mellan Curands Philox-implementation och den vi har gjort för ett mindre antal iterationer. Det är samma algoritm, så resultatet är vid en första blick förvånande. Vid en djupare analys för 100 iterationer kan man se att Curand-

versionen använder sig av betydligt mer Local Memory än vår implementation. Hela 16,2 miljoner åtkomster till L2 Cache görs av Curand-MTGP vilket är en stor skillnad jämfört med vår Philox, som endast gör 51 tusen åtkomster. I den genererade assembler-koden kan man också se betydligt fler minnesinstruktioner för Curands version. Detta innebär en prestandaförlust — *nvvp* visar på att 29% av all väntetid (stalls) är trådar som väntar på att minnet ska hämtas (*memory dependency* [30]). Detta är operationer som görs i början när Curands tillstånd laddas in vilket medför att ju fler iterationer som körs desto närmre i prestanda kommer den vår implementation av generatorn vilket också kan ses i Figur 5.3.

Mersenne Twister är inte med i graferna då den är flera storleksordningar långsammare än de andra generatorerna. Det tar hela $38,8$ ms att generera 100 slumptal på GTX 1060. Det är ungefär 46 gånger längre tid än för Curands Philox-generator. Under exekveringen sker det 293 miljoner minnesåtkomster till *Local Memory*. Anledningen till detta är att varje generator har 2,5kB minne att hålla reda på, vilket inte får plats i lokala register och sparas därför i Local Memory istället, som har betydligt högre väntetid än register.

Bortsett från Mersenne twister så är MTGP den generator som har sämst prestanda. Detta syns i Figur 5.3, där MTGP efter ett antal iterationer med marginal blir minst prestandaeffektiv. Implementationen är dessutom direkt opassande för applikationer med mer än 200 block och 256 trådar per block. Precis som Mersenne Twister så har MTGP ett stort tillstånd, vilket innebär många minneshämtningar. Detta saktar ner generatorn då ungefär 67% av all väntetid är memory dependency. *Synchronization*, vilket är när trådar väntar på att andra ska hinna ikapp [30], står för cirka 15%. Denna aspekt är dock direkt kopplad till själva algoritmen, där man inte vill att trådar ska börja läsa på tillstånd som inte räknats fram än, så ingen optimering kan ske i det området.

5.3 Normalfördelade generatorer

För normalfördelade slumptal så har vi valt att testa kombinationerna Box-Muller + LCG, Box-Muller + Philox, Ziggurat + LCG, Ziggurat + Philox samt Wallace med olika antal iterationer (se Figur 5.4). Som man kan se är Wallace hackig i grafen, vilket beror på att den alltid genererar multipler av 4 slumptal. Wallace har en viss uppstartskostnad men blir snabbt den mest prestandaeffektiva generatorn när fler iterationer körs. Långsammast är kombinationen av Ziggurat och Philox.

Det är tydliga skillnader mellan Box-Muller + Philox och Ziggurat + Philox. Den förstnämnda är två till tre gånger snabbare, och problemet ligger huvudsakligen i själva Ziggurat-algoritmen. Det nämns i Avsnitt 4.2.2 att det i teorin är 47,95% risk att en tråd i ett warp måste ta en alternativ exekveringsväg (gren, branch), vilket leder till att resten av trådarna i warpet måste vänta. I praktiken har *nvvp* mätt hela 59% alternativa exekveringar, och 75% av alla instruktioner exekverade är inaktiva/väntande och därmed bortkastade. Endast 18% av exekveringen är heltalsoperationer. Box-Muller + Philox, å andra sidan, har inga alternativa grenar och kör därför på 100% effektivitet. Dessutom består generatorn av 20% flyttal-instruktioner, vilket en GPU är speciellt optimerad för.

En liknande jämförelse kan göras mellan Box-Muller + LCG och Ziggurat + LCG. För Ziggurat-implementationen så är det 59% alternativa exekveringar, samma som för Philox.

Figur 5.4: Grafen visar tid per iteration för normalfördelade generatorer när mängden iterationer varieras. Tiden t för ett antal n iterationer är justerat med t/n. Tiden för varje punkt på grafen är ett medelvärde av 5000 körningar.

Detta är ett klart bevis för att det är Ziggurat-delen av algoritmen som står för alla alternativa exekveringar i dessa fallen.

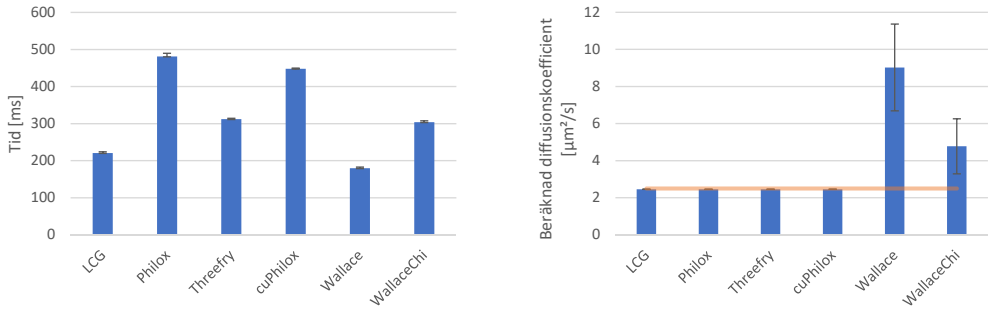
Den snabbaste av alla varianterna, Wallace, har inga alternativa exekveringar och 99% *occupancy*. Detta tillsammans med det stora antalet flyttalsinstruktioner, 33%, bidrar till väldigt bra prestanda. Mätningarna av enskild generator-prestanda (Figur 5.3 samt 5.4) har visat att Wallace är snabbare än vissa av de uniformt fördelade generatorerna, exempelvis Threefry och Philox.

Den största anledningen till att trådarna som kör Wallace-implementatinen måste vänta är synkroniseringen. Likt MTGP så måste trådarna i vissa delar av algoritmen vänta på att alla andra trådar kommit ikapp så att de exempelvis inte försöker hämta gamla värden som inte uppdaterats än. Detta står för 39% av all väntetid. Denna aspekt går inte att optimera då den är djupt förankrad till algoritmen i sig. Till skillnad från många av de andra generatorerna så påverkas Wallace inte alls av någon *memory dependency* - det är endast 1,61% av all väntetid.

5.4 Simulationskod

Som man kan se i Figur 5.5a är Wallace, utan korrektion, snabbaste generatorn med LCG tätt efter. Med χ^2-korrektion (se Avsnitt 4.2.3) blir dock algoritmen nästan dubbelt så långsam och lyckas ändå inte möta referensvärdet (se Figur 5.5b). Vi valde att inte testa Ziggurat då den alltid presterar sämre än Box-Muller-transformen i tidigare resultat.

Man kan också se att osäkerheten för Wallace är stor, vilket indikeras av de svarta staplarna som visar standardavvikelsen av mätdatan. Alla de andra generatorerna ger en mycket nära approximation med minimal osäkerhet vilket tyder på bra kvalitet..



(a) **(b)**

Figur 5.5: Dessa figurer visar prestanda (Figur 5.5a) och precision (Figur 5.5b) för diffusionssimuleringen. Referensvärdet på diffusionen är i detta fallet $2,5$ och indikeras med orange linje. Wallace har två mätningar, en gång med χ^2-korrektion (WallaceChi).

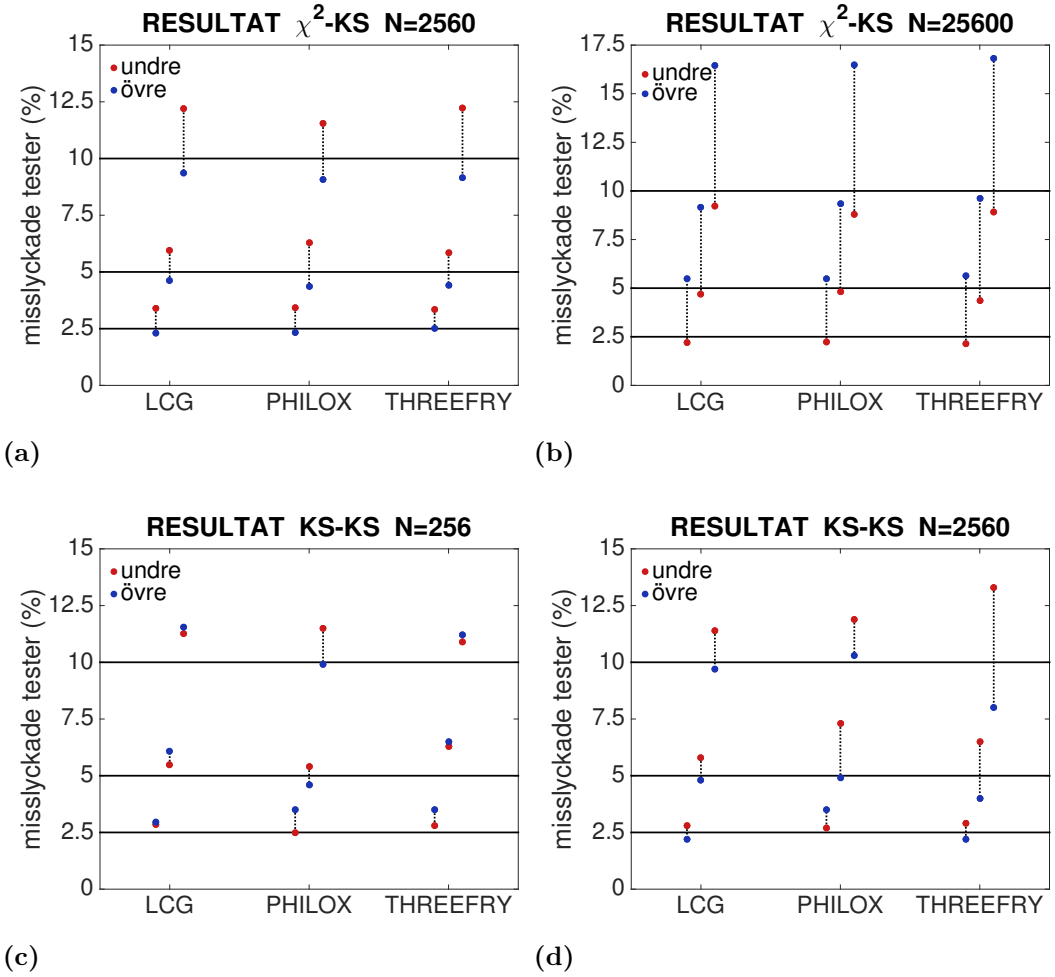
Intressant nog är Curands Philox något snabbare än vår Philox. Detta beror förmodligen på det stora antal iterationer som genomförs i simulationen, då Curands implementation har större startkostnad men kommer ikapp Philox ju fler tal varje generator genererar. Detta kan ses i Figur 5.3.

För en jämförelse med traditionella processorer så gjordes även mätningar med samma parametrar där för simuleringskoden, då på en Intel i5-6500. När vi använde oss av C++ standardbiblioteks slumptalsgeneratorer tog det i snitt, med liten osäkerhet, runt 54300 ms. Detta är att jämföra med endast 220ms för en kombination av Philox och Box Muller-transformen på GPU.

5.5 Statistiska resultat

Resultaten för Wallace har inte inkluderats i Figur 5.6 av den anledningen att de inte är i närheten av de förväntade resultaten (se Tabell 5.1). Från Figur 5.6 framgår det att ingen av generatorerna (LCG, Philox och Threefry) har mycket sämre kvalité än de andra. De två χ^2-KS-testerna ger liknande resultat för alla generatorerna; stor andel misslyckade tester vid undre gränsen för $N = 2560$ och stor andel misslyckade tester vid övre gränsen för $N = 25600$. Slutsatser som kan dras från χ^2-KS-testerna är att de genererade slumptalen inte är tillräckligt slumpmässiga då 2560 tal studeras men att de ändå följer normalfördelningen bra. När 25600 tal studeras följer talen inte normalfördelningen väl men de har bra slumpmässighet.

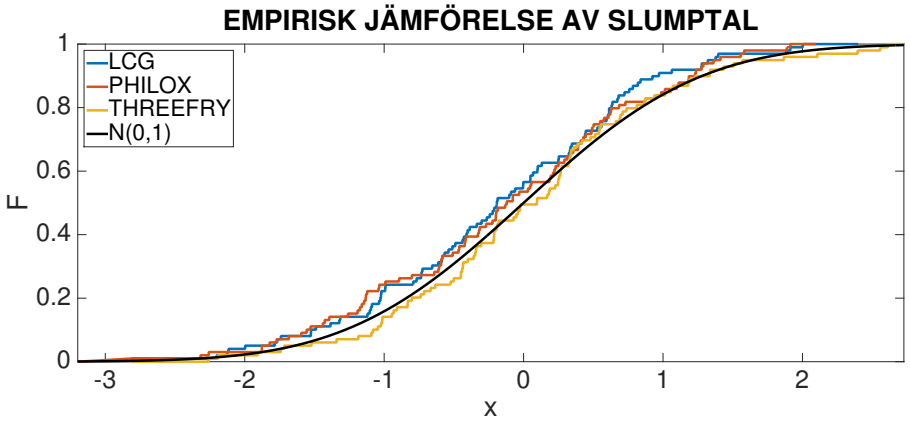
För KS-KS-testet med $N = 256$ tycks Philox ha lite bättre kvalité än de andra generatorerna, som överlag har en stor andel misslyckade tester både vid övre och undre gränsen. För det andra KS-KS-testet med $N = 2560$ är Threefry den generator som visar sämst resultat. Utifrån alla fyra testerna går det inte att dra någon definitiv slutsats om vilken generator som har bäst slumpmässig kvalité. Vad som kan sägas är att enligt KS-KS-testerna



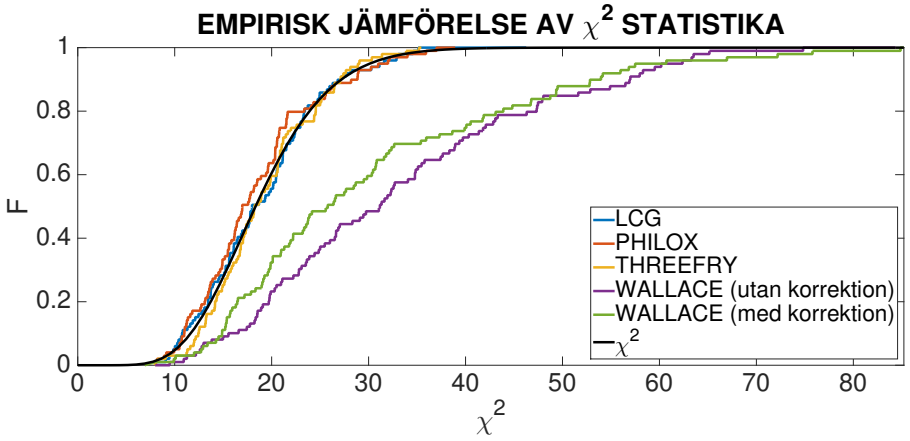
Figur 5.6: De fyra plottarna visar andelen misslyckade körningar för de 4 olika testerna som utförts för LCG, Philox och Threefry. De svarta horisontella linjerna indikerar de förväntade antalet misslyckade körningar (2.5, 5.0 respektive 10%).

följer generatorerna normalfördelningen väl och slumpkvalitén är bra. χ^2-KS-testerna ger motstridiga resultat — en möjlig anledning diskuteras i kapitlet 6.

Anledningen till att inte data från Wallace generatorn inkluderats i Figur 5.7 är att standardavvikelsen växer då generatorn används utan att ett nytt seed används som initialtillstånd [21], därmed är det klart att Wallace inte följer normalfördelningen $N(0,1)$ efter ett antal kernel-anrop. Vad som kan ses i figuren är att de tre generatorerna tycks följa normalfördelningen bra lokalt, slutsatsen från χ^2-KS testet är dock att generatorerna inte följer normalfördelningen bra globalt.



Figur 5.7: Plotten visar de empiriska CDF funktionerna för 100 slumptal genererade av LCG, Philox och Threefry.



Figur 5.8: Plotten visar de empiriska CDF funktionerna för 100 χ^2 statistikor erhållna från Lcg, Philox, Threefry samt Wallace med och utan χ^2 korrektionen. Wallace χ^2 statistikorna är skapade från slumptal genererade efter 100-200 kernel-anrop.

I figur 5.8 syns täcken på att chi^2 statistikor från Wallace genererade slumptal efter 100-200 kernel-anrop inte längre följer χ^2 fördelningen, den χ^2 korrigerade Wallace generatorn följer fördelningen lite bättre. Det tyder på att Wallace inte heller följer $N(0,1)$ vilket också kan ses i datan i tabell 5.1. Det ses att χ^2 statistikor från de resterande generatorerna (LCG, Philox och Threefry) följer distributionen väl.

				GENERATORER			
TEST	N	P		LCG	Philox	Threefry	Wallace
χ^2-KS	2560	2.5	UNDRE	3.4	3.42	3.34	0 0
			ÖVRE	2.3	2.34	2.51	98.69 99.51
		5.0	UNDRE	5.94	6.28	5.84	0.01 0
			ÖVRE	4.62	4.35	4.41	98.96 99.65
		10.0	UNDRE	12.19	11.54	12.23	0 0.01
			ÖVRE	9.35	9.08	9.16	100 99.86
χ^2-KS	25600	2.5	UNDRE	2.21	2.23	2.13	0 0
			ÖVRE	5.49	5.47	5.65	100 100
		5.0	UNDRE	4.69	4.81	4.37	0 0
			ÖVRE	9.15	9.34	9.61	100 100
		10.0	UNDRE	9.22	8.79	8.93	0 0
			ÖVRE	16.45	16.47	16.83	100 100
KS-KS	256	2.5	UNDRE	2.84	2.5	2.8	0 0
			ÖVRE	2.95	3.5	3.5	98 99.9
		5.0	UNDRE	5.47	5.4	6.3	0 0
			ÖVRE	6.09	4.6	6.5	98.4 99.9
		10.0	UNDRE	11.27	11.5	10.9	0 0
			ÖVRE	11.54	9.9	11.2	98.9 100
KS-KS	2560	2.5	UNDRE	2.8	2.7	2.9	
			ÖVRE	2.2	3.5	2.2	
		5.0	UNDRE	5.8	7.3	6.5	
			ÖVRE	4.8	4.9	4.0	
		10.0	UNDRE	11.4	11.9	13.3	
			ÖVRE	9.7	10.3	8.0	

Tabell 5.1: Tabellen innehåller resultat för χ^2-KS- och KS-KS-testerna för samtliga generatorer. Angivit är andelen misslyckade test i %. För χ^2-KS-testen gäller: $K = 20$, $M = 100$ och $L = 10000$. För KS-KS-testen gäller: $M = 100$ och $L = 1000$. I kolonnen markerad P finns signifikansgränserna för varje test. *Under* och *Över* syftar på andelen misslyckade test vid den undre respektive övre signifikansgränsen. För Wallace är två värden angivna, med och utan χ^2-korrektion. Wallace har bara testats med ett KS-KS-test, i det testet användes $N = 1024$.

6

Diskussion och Slutsats

Vi har i denna rapporten presenterat olika algoritmer som genererar pseudoslumptal, och gett en teknisk och matematisk bakrund till dessa. Dessutom har GPU:n som plattform beskrivits, med fokus på de delar som har påverkat våra implementationers prestanda. Analysen av de olika generatorerna har bestått av prestanda- och kvalitémätningar och jämförelsen mellan dessa har varit med avsikten att rekommendera olika generatorer för olika användningsområden.

Att de två χ^2-KS-testerna ger samma resultat oavsett generator kan vara relaterat till val av intervall (se Figur 4.2). Knuth skriver i sin artikel *The Art Of Computer Programming* [22] att χ^2-testets styrka beror på val av intervall och antalet mätdata. Han nämner specifikt att för ett starkt test ska intervallen vara av samma storleksordning ($p_1 \approx p_2 \approx p_3 \cdot \cdots \cdot p_{K-1} \approx p_K$). För $K = 20$ erhålls med indelningen beskriven i Avsnitt 4.3.1 intervall sannolikheterna

```
p[ 1] = 0.008352
p[ 2] = 0.008353
p[ 3] = 0.014655
p[ 4] = 0.023967
p[ 5] = 0.036536
p[ 6] = 0.051916
p[ 7] = 0.068766
p[ 8] = 0.084903
p[ 9] = 0.097714
p[10] = 0.104827
p[11] = 0.104827
p[12] = 0.097714
p[13] = 0.084903
p[14] = 0.068766
p[15] = 0.051916
p[16] = 0.036536
p[17] = 0.023967
p[18] = 0.014655
p[19] = 0.008353
p[20] = 0.008352
```

som inte alla är av samma storleksordning. En annan förklaring till resultaten för χ^2-KS-testerna kan var låga antalet slumptal. Knuth rekommenderar att $N \times p_i >> 5 \forall i$ [22] vilket uppfylls för $N > 600$ då $K = 20$. För det första χ^2-KS-testet är $N = 2560$ vilket uppfyller $N \times p_i >> 5 \forall i$ men kan ändå vara en aning lågt.

Vår implementation av Philox presterar ungefär likvärdigt jämfört med Curands version då mängden tal som varje tråd genererar ökar. Detta gör att vi kan rekommendera Curands

version av Philox för en färdig implementation av en ganska komplicerad generator. Philox är långsam jämfört med LCG enligt våra mätningar men har å andra sidan en betydligt längre period. Detta gör att vi kan rekommendera Curands version, om LCG:s korta period inte räcker för en applikation och den extra prestandan inte behövs.

Den generator som har visat sig vara absolut mest prestandaeffektiv i alla prestandatester är LCG, som har väldigt låg startkostnad och är snabbaste algoritmen vid många iterationer. Enligt våra tester är dessutom LCG ungefär lika slumpmässig som de andra generatorena. Vi kan därför rekommendera den LCG vi har testat till den användare som prioriterar hög prestanda. Vid användning av andra parametrar, som de i ANSI C [9], så gäller dock inte denna rekommendation då slumptalskvaliteten påverkas.

Resultaten för Mersenne Twister tyder på dess olämplighet för parallella applikationer. Den har stor startkostnad på grund av sitt stora minnestillstånd, vilket är opassande när en stor mängd generatorer behövs. Mersenne Twister passar bra på en CPU för att man endast brukar skapa en eller ett fåtal instanser av generatorn och bieffekterna minskar därmed. Även den GPU-anpassade varianten MTGP visar på bristande prestanda i parallella applikationer. Eftersom implementationen också är begränsad av antalet trådar och block så kommer inte MTGP vara användbar för större parallella applikationer. MTGP är dessutom krånglig att använda då den inte följer resten av ramverket som Curand [28] använder. Andra generatorer använder samma funktioner för att initialisera tillståndet, medan MTGP använder sig av en egen variant. Detta beror på alla extra konstanta parametrar som algoritmen kräver. Det är därför inte en generator som bör användas om någon annan finns tillgänglig, varav det finns flera i Curand.

Av de olika metoderna för normalfördelade slumptal visar sig Box-Muller + LCG vara snabbast, tätt följt av Wallace Gaussian Generator. Analysen av Box-Muller har påvisat att den är utmärkt som metod i parallella applikationer på GPU då den saknar alternativa exekveringar som Ziggurat. Det är alltså ett resultat som är motsatsen till liknande jämförelse på CPU, där Ziggurat anses vara den snabbare av algoritmerna [5].

En intressant metod är Wallace, som har lyckats hålla väldigt god prestanda och samtidigt vara en relativt komplicerad algoritm. Ett problem med den är dock att den är beroende av att det finns *Shared Memory* tillgängligt, och passar därför dåligt i applikationer som använder sig mycket av denna minnestyp, något som också påpekas av Lee Howes och David Thomas i deras artikel [5]. Om man dessutom tar i åtanke vilken av dessa metoder som är enklast att implementera för en användare, så är Box-Muller ett säkert kort för bra prestanda och enkelhet. Ziggurat bör i allmänhet undvikas på GPUer, och Wallace är ur prestandasynvinkel ett bra alternativ om man inte använder LCG som sin uniformt fördelade generator, då Wallace är snabbare än de andra alternativen vi presenterat (se Figur 5.4).

Nackdelen med Wallace är att resultaten i diffusionssimuleringen var katastrofalt dåliga. Detta beror sannolikt på två faktorer. Dels att vår implementation seedar om på CPU-sidan. Wallace behöver seedas om efter ett antal pass genom algoritmen och simuleringen som körs helt på grafikkortet kan inte göra detta. Vi har också funnit att även med χ^2-korrigering och ett färre antal pass så blir resultat trots det fortfarande dåligt om än något bättre, men då också med betydligt sämre prestanda. Detta innebär att vi inte kan rekommendera Wallace-generatorn för något användningsområde.

6.1 Framtida arbete

För ett mer tillförlitligt χ^2-KS-test bör ett annat val av intervalluppdelning göras, den bästa intervalluppdelningen skulle vara det som uppfyllde ($p_1 = p_2 = p_3 \cdot \cdots \cdot p_{K-1} = p_K$). Att utföra χ^2-KS-testerna för fler slumptal (större N) skulle också kunna ge mer tillförlitliga resultat.

De tester som utförts är primärt anpassade för att avgöra huruvida mätdata tillhör en underliggande distribution, i vårt fall om slumptalen kan anses komma från en normalfördelning. Fler tester skulle behöva utföras som är mer anpassade att studera slumpmässigheten av mätdatan, t.ex olika former av korrelationstester som testar huruvida slumptalen är oberoende varandra som Knuth's *Serial correlation test* [22] eller *smultin* från TESTU01 [3]. För att upptäcka periodiciteter i slumptalen skulle en spektralanalys (fouriertransform) behöva utföras på de uniformt distribuerade slumptalen, periodiciteter i LCG generatorn samt dess korta period skulle därmed kunna upptäckas.

Det finns ett antal algoritmer vi inte har analyserat i denna rapport, som skulle vara intressanta för vidare analys inom området. En sorts generatorer som inte har undersökts är kombinerade algoritmer - där man samlar ihop flera olika algoritmer för att ge bättre slumptalskvalité. Ett exempel på en sådan generator är den *kombinerade Tausworthe*-generatorn [5]. Algoritmer som är implementerade i Curand [28] men som vi inte analyserat är *XORWOW*, och *MRG32*. Det finns även en variant av *Wallace Gaussian Generator* som är specialiserad för vektorprocessorer [31].

Det är möjligt att utveckla de olika algoritmerna för att passa in i det ramverk som Curand är byggt på, för att enkelt kunna integrera dessa med existerande kod i industrin. Då kan man publicera koden som ett bibliotek, med en prestandaanalys som passar in väl på det ramverket. En djupare jämförelse mellan vår Philox och Curands motsvarighet hade också varit av intresse.

Polar-metoden är en algoritm för generering av normalfördelade slumptal av Marsaglia [18]. Den ersätter ofta kostsamma `sin` och `cos`-funktioner med logaritmer för prestandavinster. Nackdelen är dock densamma som Ziggurat, att änden av svansen fortfarande behöver genereras separat och därmed tappas prestandavinsten i GPU-sammanhanget. Därför anser vi att en analys av Ziggurat var tillräcklig för detta arbete, men en framtida analys kan gynnas av att ta hänsyn till fler metoder som denna.

En typ av programmeringsstil för GPGPU som undersöks i dagsläget är *persistent threads* [32]. Nuvarande praxis för arbete som utförs på GPU:er är att data som ska beräknas skickas som en mängd jobb till grafikkort, där kortet sedan själv startar upp jobben i form av trådar. Dessa trådar stängs sedan ner direkt när de är klara för att göra plats för nya som scheduleras av grafikkortet. *Persistent Threads* kringgår istället denna paradigm genom att man startar upp en fast mängd trådar som jobbar tills hela arbetet är slutfört. Detta skulle kunna vara av intresse inom pseudoslumptal-generering men enligt K. Gupta [32] finns ej något praktiskt stöd inom Cuda för det.

Litteraturförteckning

[1] Myles Sussman, William Crutchfield, and Matthew Papakipos. Pseudorandom number generation on the GPU. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Graphics Hardware*, pages 87–94, 2006.

[2] Magnus Röding, Hendrik Deschout, Kevin Braeckmans, and Mats Rudemo. Measuring absolute number concentrations of nanoparticles using single-particle tracking. *Phys. Rev. E*, 84:031920, Sep 2011.

[3] Pierre L'Ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4), August 2007.

[4] John Walker. ENT - a pseudorandom number sequence test program. http://www.fourmilab.ch/, 2008. (läst 3 februari 2017).

[5] David Thomas Lee Howes. Efficient random number generation and application using CUDA. In *GPU Gems 3*, chapter 37. Pearson Education, Inc.

[6] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 16:1–16:12, 2011.

[7] Melissa O'Neill. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. (Submitted to ACM Transactions on Mathematical Software), 2015.

[8] Oracle. Class random. https://docs.oracle.com/, 2017. (läst 20 mars 2017).

[9] Karl Entacher. Bad subsequences of well-known linear congruential pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation*, 8(1), January 1998.

[10] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1), January 1998.

[11] Python Software Foundation. Python documentation | 9.6 random - generate pseudo-random numbers. https://docs.python.org/, 2017. (läst 22 februari 2017).

[12] James Britt and Neurogami. Class: Random (Ruby 1.9.3). https://ruby-doc.org/, 2017. (läst 22 februari 2017).

[13] NumFOCUS. Julia - Docs » Numbers - Random Numbers. https://docs.julialang.org, 2017. (läst 22 februari 2017).

[14] Mutsuo Saito and Makoto Matsumoto. Variants of Mersenne Twister suitable for graphic processors. *ACM Transactions on Mathematical Software*, 39(2), February 2013.

[15] Niels Ferguson and Stefan Lucks. The Skein hash function family. 2010.

[16] G. E. P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 06 1958.

[17] George Marsaglia and Wai Wan Tsang. The Ziggurat method for generating random variables. *Journal of Statistical Software*, 5(1):1–7, 2000.

[18] G. Marsaglia and T. A. Bray. A convenient method for generating normal variables. *SIAM Review*, 6(3):260–264, 1964.

[19] George Marsaglia. Generating a variable from the tail of the normal distribution. *Technometrics*, 6(1):101–102, 1964.

[20] Cleve Moler. The Ziggurat random normal generator. blogs.mathworks.com/cleve/2015/05/18/the-ziggurat-random-normal-generator/, 2015. (läst 8 maj 2017).

[21] C. S. Wallace. Fast pseudorandom generators for normal and exponential variates. *ACM Transactions on Mathematical Software*, 22(1), March 1996.

[22] Donald E. Knuth. *The art of computer programming*, volume 2. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 1997.

[23] Bertil W. Lennart, R. *Mathematics Handbook for Science and Engineering*. Studentlitteratur AB, 5:15 edition, 2004.

[24] Tsang WW. Wang J. Marsaglia, G. Evaluating Kolmogorov's distribution. *Journal of Statistical Software*, 8, 2003.

[25] NVIDIA Corporation. CUDA C Programming guide. http://docs.nvidia.com, 2017. (läst 28 Mars 2017).

[26] NVIDIA Corporation. PTX ISA. http://docs.nvidia.com, 2017. (läst 28 Mars 2017).

[27] NVIDIA Corporation. Achieved Occupancy. http://docs.nvidia.com, 2015. (läst 26 April 2017).

[28] NVIDIA Corporation. cuRAND. https://developer.nvidia.com, 2017. (läst 28 februari 2017).

[29] Richard P. Brent. Some comments on C. S. Wallace's random number generators. *The Computer Journal*, 51(5):579, 2008.

[30] NVIDIA Corporation. Profiler User's Guide. http://docs.nvidia.com, 2017. (läst 7 Maj 2017).

[31] Richard P Brent. A fast vectorised implementation of wallace's normal random number generator. Technical report, 1997.

[32] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *2012 Innovative Parallel Computing (InPar)*, 2012.

Ordlista

CBRNG

Counter based random number generator.

CDF

Cumulative Distribution Function.

CPU

Central Processing Unit.

CUDA

Tidigare akronym för Compute Unified Device Architecture men numera endast *CUDA*..

DRAM

Dynamic Random Access Memory.

GCC

GNU Compiler Collection.

GPGPU

General-purpose computing on graphics processing units.

GPU

Graphics Processing Unit.

MSD

Mean Square Distance.

NVCC

Nvidia CUDA Compiler.

OpenCL

Open Computing Language.

PDF

Probability Density Function.

PRNG

PseudoRandom Number Generator..

SP-nätverk

Substitution-Permutation Network.

Bidragsrapport

Här följer en beskrivning av hur vi har delat upp arbetet och arbetat under detta projekt. Jakob Aasa har lagt 260 timmar, Robin Johansson har lagt 300 timmar och Marcus Lundberg har lagt 310 timmar.

Ansvarsområden

Arbetet har varit uppdelat som sådant att Robin Johansson primärt fokuserat på testalgoritmerna för slumptalen och deras implementation. Jakob Aasa och Marcus Lundberg har läst artiklar och valt ut vilka algoritmer för slumptalsgenerering som skulle behandlas samt implementerat dessa. Jakob Aasa har i större utsträckning jobbat med att ta fram resultat och presentera dessa medan Marcus Lundberg har analyserat resultaten. Rapportskrivandet har till största del varit uppdelat på liknande sätt där gruppmedlemmarna har skrivit om den del de jobbat med.

Planering och informationshämtning

Första veckorna av arbetet lades mycket tid på att läsa på och komma underfund med vilka algoritmer som fanns och var intressanta att implementera. Mycket tid lades även på att bekanta sig med C++ och Cuda då medlemmarna i gruppen hade lite eller ingen tidigare erfarenhet av dessa.

Planeringen var strukturerad så att testalgoritmerna kunde arbetas på parallellt med implementering av generatorer.

Resultat och analys hamnade efter implementeringarna var klara, då det visade sig vara svårt att göra dessa parallellt med implementeringarna.

Metoder

Metoderna som använts har, förutom mycket läsning av artiklar knutna till ämnet, varit att skriva kod och testa den skrivna koden samt analys. Kodskrivandet har följt resterande delar av rapporten, där varje medlem har skrivit för det den varit ansvarig för. Se rubrik Huvudansvarig författare för avsnitt nedan för mer detaljerad uppdelning.

Genomförande

I genomförandet har vår handledare Markus Billeter skapat det ursprungliga ramverket som våra implementationer använder sig av. Jakob har implementerat Philox, Threefry, samt Ziggurat-generatorerna medan Marcus har gjort LCG, Mersenne Twister, Wallace och Curand-anpassningarna.

Testerna av slumptalens slumpmässighet har valts ut och skrivits av Robin.

Jakob har kört prestandastester och tagit fram mätdata och sedan tolkat datan i Excel för att ta fram grafer för visuell analys.

Marcus har genom att använda programmet nvvp analyserat körningar av generatorerna.

Jakob och Marcus har jobbat mycket gemensamt med att diskutera vilka prestandaresultat som är relevanta för analysering. Robin har på egen hand valt vilka samt implementerat kvalitétsanalyser av generatorerna och analyserat den resulterande datan.

Arbetsprocessen har varit mycket kreativ då en en ganska vag projektformulering behövdes konkretiserats i flera steg. Gruppen har gemensamt bestämt mycket med en del handledning av handledaren vad som ska fokuserats på.

Huvudansvarig författare av avsnitt

Avsnitt	**Ev. underavsnitt**	**Huvudansvar**
Inledning		JA
Slumptalsteori	LCG	ML
	Mersenne twister	ML
	CBRNG	JA
	Box-Muller + Ziggurat	JA
	Wallace Gaussian	ML
	Test av pseudoslumptal	RJ
GPU-teori	Arkitektur...	ML
	Slumptalsberäkningar..	ML
Metod	Engine Factory	JA
	Philox & Threefry	JA
	Ziggurat	JA
	Wallace	ML
	Curands	ML
	Tester	RJ
Resultat och Analys		JA, ML
	Uniformt	JA, ML
	Normalfördelade	JA, ML
	Statistiska tester	RJ
Diskussion och Slutsats		Gemensamt

Jakob Aasa, Robin Johansson och Marcus Lundberg, Göteborg, Maj 2017