Degree Project in the Field of Technology Engineering Physics and the Main Field of Study Computer Science

Second cycle, 30 credits

Gamma Random Variable Generation on GPUs using CUDA

JOHAN ERICSSON

Gamma Random Variable Generation on GPUs using CUDA

JOHAN ERICSSON

Date: June 14, 2024

Supervisor: Mohit Daga
Examiner: Arvind Kumar
School of Electrical Engineering and Computer Science
Swedish title: Generering av Gammafördelade Slumptal på GPUer genom CUDA

Abstract

We study how pseudo random gamma distributed random variables can be efficiently generated on graphical processing units. There are many algorithms known today for generating gamma random variables by means of computation that perform well on central processing units of classical computers. In the last 20 years, there has been increasing interest in using graphical processing units and other accelerators to speed up stochastic simulation and machine learning applications. The difference in architecture between graphical processing units and traditional central processing units means that algorithms that perform well on central processing units do not always perform well on graphical processing units. This is especially true for random number generation algorithms from complex distributions. In this work, we show that graphical processing units can be used to efficiently simulate random numbers from a gamma distribution, and that the best performing algorithms for doing so are different than the best performing algorithms on central processing units.

Keywords: Gamma distribution, Random variable generation, Non-uniform, RNG, GPU, CUDA

Sammanfattning

Vi studerar hur gammafördelade slumptal effektivt kan genereras på grafikprocessorer. Det finns många algoritmer kända idag för att generera gammafördelade stokastiska variabler och som presterar bra på processorer för klassiska datorer. Under de senaste 20 åren år har intresset ökat för att använda grafikprocessorer och andra acceleratorer för att accelerera stokastisk simulering och maskininlärningsapplikationer. Skillnaden i arkitektur mellan grafikprocessorer och traditionella processorer innebär att algoritmer som presterar väl på traditionella processorer inte alltid presterar väl på grafiska processorer. Detta gäller särskilt för algoritmer för att generera stokastiska variabler från komplexa distributioner. I detta arbete visar vi att grafikprocessorer kan användas för att effektivt simulera slumptal från en gammafördelning, och att de bäst presterande algoritmerna för att göra det skiljer sig från de bäst presterande algoritmerna på traditionella processorer.

Nyckelord: Gammafördelningen, Slumptalsgenerering, Icke uniform, RNG, GPU, CUDA

Acknowledgements

I would like to thank my thesis supervisor, Mohit Daga for his comments on the drafts and attention to detail. His guidance during the scientific writing process has been very valuable, and his challenging questions have been instrumental in helping me refine the exposition. I am very grateful for his insights and support throughout this journey.

Stockholm, June 2024

Johan Ericsson

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Stochastic simulation techniques are fundamental to modern society, with a wide range of applications such as weather simulations, financial forecasting, computational chemistry and machine learning, among others. The key assumption behind stochastic simulation is that we can generate random variables by means of computation. This was done already in late 1940s as part of the Manhattan Project, when Stanislaw Ulman and John von Neumann invented the Monte Carlo method. Deterministic algorithms that can be used to generate, seemingly random, numbers are called pseudo random numbers generators (PRNGs) and the performance of these algorithms have been studied extensively for classical computers with central processing units (CPUs).

In the early 2000s, graphical processing units (GPUs) started being used for other computing purposes than graphics only, and the term GPGPU: general-purpose computation on graphics hardware was coined. Today GPUs play an important role in the computing landscape and as of November 2023, 9 out of the top 10 supercomputers of the *Top 500 list*[1] partly consist of GPUs. The main difference between GPUs and CPUs is that GPUs have a throughput-oriented architecture whilst CPUs have a latency-oriented architecture. This means that GPUs excel at highly parallel workflows, with little or no control flow or branching. CPUs on the other hand are optimized to handle sequential code with possibly complicated control flow and branching. A consequence of this difference in architecture is that algorithms that perform well on CPUs may not perform well on GPUs.

When it comes to generating random numbers on GPUs, many of the traditional algorithms for generating random numbers on CPUs are not viable from a performance perspective. As a consequence of this, the two leading GPU manufacturers' random number generation libraries (AMD's *rocRAND* library[2] and NVIDIA's *cuRAND* library[3]) only provides the functionality to generate random numbers from five types of distributions: uniform, Poisson, normal, log-normal, and custom discrete distributions. Many scientific disciplines require the possibility to sample from other, oftentimes more complex, distributions. One of the most commonly used distributions among scientific disciplines is the family of gamma distributions. It is also among the most well-studied distributions from a computer simulation perspective and a recent survey paper by Luengo [29] gives a good overview of existing algorithms and a review of their performance on CPUs.

Previous work on random number generation from other distributions has shown that methods that were previously discarded as suboptimal on CPUs performed much better on GPUs. Examples of this for uniform generators and the normal distribution can be found in [14]. These discrepancies in performance between the same algorithm on CPUs and GPUs are more notable in the context of gamma generators. The reason for this is that the only methods used for gamma generation today are based on rejection sampling (and variations thereof), or numerical inversion of the distribution function [29]. The most important factors for highly performant rejection sampling algorithms on GPUs are very different than those on CPUs.

Rejection sampling is a commonly used Monte Carlo technique and the performance on GPUs can be good enough to warrant simulation on GPUs instead of CPUs. Furthermore, there are techniques that can be used to speed up these algorithms, see for instance [43]. In

[1] Top 500 list, ranking the world's fastest supercomputershttps://www.top500.org/lists/top500/2023/11/
[2] ROCm platform version 5.7.1 https://rocm.docs.amd.com/projects/rocRAND/en/docs-5.7.1/index.html
[3] CUDA toolkit version 12.3.1 https://docs.nvidia.com/cuda/archive/12.3.1/curand/index.html

this thesis, we study the existing algorithms for gamma random variable generation from a performance perspective on GPUs. We compare the results of the best-performing gamma generators on GPUs with the gamma generator available in the C++ standard library, and we show that there exist algorithms that make gamma random variable generation on GPUs a viable option from a performance perspective.

1.1 Problem Statement

The gamma distribution is an important family of probability distributions that is widely used in stochastic models, machine learning, and Monte Carlo simulations. Even though there is a large increase in the use of GPUs for stochastic simulation, and machine learning workloads, the preferred algorithms for generating gamma random numbers on the GPU remain unknown. Furthermore, since the majority of algorithms used to generate gamma random numbers are based on rejection sampling, it is not obvious that generation on GPU is viable when compared to using traditional CPUs instead.

1.2 Purpose

The main objective of this thesis is to investigate whether there exists gamma random number generation algorithms that can be implemented efficiently on GPUs and how these perform compared to the state-of-the-art gamma generators available for CPUs.

The answer to these questions will be of interest to researchers and professionals in both academia and industry across a wide range of disciplines where gamma random numbers are used. Knowing what the best methods for generating gamma random numbers on GPUs are will help practitioners make well-informed choices on what hardware to use for their specific purposes. This can help reduce the energy required to perform the computations and may lead to a reduced carbon footprint and lower costs. Moreover, if the best generator on GPUs outperforms the best generator on CPUs this opens up the possibility to perform simulations previously not possible. This can lead to new research in the many disciplines where gamma random numbers are used in computer simulations.

1.3 Research Methodology

We implement a selection of the existing algorithms for gamma random variable generation in CUDA and measure the performance of these algorithms while executed on a NVIDIA GPU. We also compare the time it takes to generate gamma random variables on the GPU with the time it takes on a CPU. For this we use the routine available to generate gamma random variables from the C++ standard library. The selected method of research can be divided into four main components.

1. Algorithm selection.
2. Implementation of selected algorithms in C++ and CUDA.
3. Profiling, timing, and measurements.
4. Analysis of measurements

The first step consists of a literature review which leads to an informed choice of which algorithms to select for development and measurement. An important part of this step is that the selection must be made with the performance characteristics of GPUs in mind and the main challenge is to not dismiss algorithms that may perform well in this early selection step.

The second step involves the implementation of the selected algorithms. This step also includes the implementation of tests for verifying that the algorithms produce gamma random values. The main challenge here is to implement the algorithms efficiently in CUDA.

The main part of the third step is the running time measurements of the implemented algorithms that are used for evaluation. This also includes profiling and optimization of the kernels which is done to ensure that all algorithms are well implemented and that the comparisons of timing results can be fair between kernels.

The fourth and final step consists of the analysis of the timing measurements. It includes a comparison of the performance of the algorithms on GPUs and the reference CPU implementation. There are many challenges associated with measuring the execution times of programs on both CPUs and GPUs. It can especially be hard to ensure that the results are fair between simulations. To reduce the variability we make multiple measurements and report both the mean and variance of the samples collected.

1.4 Method

The algorithm selection is based on a survey of the existing research literature. The reference code is implemented in C++ for two major reasons:

1. C++ allows the user to write highly efficient code without high-level overhead, and the implementations of the C++ STL are generally considered to be of very high standard.

2. CUDA is built on C++ and as such it is possible to reuse common code (e.g. tests, drivers) between the C++ and CUDA implementations.

Statistical tests for verifying the implementations are written in the Python programming language, which has a rich collection of scientific libraries. We used the Kolmogorov-Smirnov (KS) test (see e.g. [18, Section 3.3.1.B]) to verify the output of implemented generators. The KS-test is available for Python through the SciPy[4] Open Source library.

The benchmarking was implemented using timing functionality from the C++ standard library and CUDA events. There are specialized libraries for benchmarking, but this approach gives us more control over measurements and lets us make more precise measurements with respect to data transfers.

1.5 Delimitations

In this work, we have limited ourselves to investigate the performance of existing algorithms for gamma random number generation. This is motivated by the large existing collection of algorithms for generating gamma random variables and a comparison of the performance of

[4] https://scipy.org

these algorithms on GPUs has not been published. We have only compared a selection of the available algorithms, since there are many known algorithms for gamma random number and implementing all of them would be beyond the scope of this thesis.

The implementations of the algorithms do not support exact reproducibility of the pseudo random sequences generated. Parallel and distributed random number generators that support this synchronized splitting into streams are complex to implement and the necessary synchronization has a negative impact on performance. From a theoretical perspective this is not problematic, since stochastic simulation is driven by laws of large numbers. However, this is of course negative from a reproducibility point of view.

We do not analyze the statistical quality or numerical precision of the output of the algorithms. Our focus is on the efficiency and throughput of the generators. That said, all generators we compare are well-known and published in well-respected journals and we are not aware of any flaws in their outputs.

1.6 Outline

The outline of this work is as follows.

In Chapter 2, we provide the necessary background material on probability, random variables and random number generation required to follow this thesis. We also give a brief introduction to GPUs, and the difference in architecture between GPUs and CPUs. We end this chapter by showing how the performance characteristic for rejection sampling algorithms differ between GPUs and CPUs.

In Chapter 3, we present the selected algorithms for generating gamma random variables and discuss their performance characteristics from a GPU perspective. Especially, we present the rejection probabilities of the algorithms with respect to the SIMT architecture of GPUs instead of the rejection probabilities traditionally used to analyze such algorithms.

In Chapter 4, we describe our CUDA implementations of the gamma generators and discuss the benchmark code used for performance measurements. We also provide information about the hardware and systems used for our measurements.

In Chapter 5, we present and analyze the measurement results. Our results show that it is more efficient to generate gamma random numbers on GPUs than on CPUs, even if the algorithms used are rejection sampling algorithms. We show that the best performing algorithm for all shape parameters α (algorithm (GA) from [7]) is close to a normal generator in performance, and that it only takes $1.5 - 2\times$ more time to generate gamma random numbers than normal random numbers on the GPU using this algorithm. This is a $1000\times$ speedup compared to the reference CPU implementation.

2 Background

In this chapter, we review the basic theory behind non uniform random number generation along with the general techniques for generating non-uniform random numbers. We start with a short review of probability and then we introduce the gamma distribution. This is followed by a section covering the basics of pseudo random number generation. We end this chapter with a section covering modern GPU architectures, CUDA programming, and the design of efficient algorithms for GPUs.

2.1 Probability and Random Variables

In this section, we review the basic probability theory which are required to follow thesis. The material is well-known and can be found in standard probability textbooks (see e.g. [16, 13]) and most advanced machine learning textbooks (see e.g. [37]). Formally, a *probability space* is a triple $(\Omega, \mathcal{F}, \mathbb{P})$ consisting of a set Ω called the *sample space*, a σ-algebra $\mathcal{F}$ of subsets of Ω called the *events*, and a probability measure $\mathbb{P}$. A *(real) valued random variable* is a Borel-measurable function $X : \Omega \to \mathbb{R}$. The *(cumulative) distribution function (c.d.f.)* of a random variable X is the function $F : \mathbb{R} \to [0, 1]$ defined by:

$$F(x) := \mathbb{P}(X \leq x) = \mathbb{P}\big(\{\omega \in \Omega : X(\omega) \leq x\}\big).$$

The distribution function of a random variable is always *cadlag*[5] ,which means that

1. F is right continuous on $\mathbb{R}$, i.e. the right limit $\lim_{x \to x_0+} F(x) = F(x_0)$ for every $x_0 \in \mathbb{R}$, and
2. the left limits of F exist on all of $\mathbb{R}$, i.e. the left limit $\lim_{x \to x_0-} F(x)$ exists for every $x_0 \in \mathbb{R}$.

Two random variables have the same distribution if their distribution functions are the same. The following theorem show that there is a one-to-one correspondence between cadlag functions with an additional property and random variables on $\mathbb{R}$.

Theorem 2.1 (see e.g. [16, Theorem 7.2]). *A function $F : \mathbb{R} \to \mathbb{R}$ is a distribution function of a real valued random variable if and only if F is cadlag and satisfy*

$$\lim_{x \to -\infty} F(x) = 0, \quad \lim_{x \to +\infty} F(x) = 1.$$

The random variables we study in this text are all continuous real valued random variables[6] or discrete random variables. A random variable X is *continuous* if there exists a (Borel measurable) real valued function, f, called the *probability density function (p.d.f)* of X such that

$$F(x) = \int_{-\infty}^{x} f(y)\,\mathrm{d}y\,, \quad \text{for every } x \in \mathbb{R}.$$

The *expected value* of a continuous random variable is given by

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} f(x)\,\mathrm{d}x\,,$$

[5] The term cadlag comes from the french expression for right-continuous with left limits: *continue à droite, limite à gauche.*

[6] In measure theoretic terms the continuous real random variables are the ones which have laws that are absolutely continous with respect to Lebesgue measure on $\mathbb{R}$

A random variable is *discrete* if it takes at most countably many values $\{x_1, \ldots, x_n\}$, in which case the expected value is given by

$$\mathbb{E}[X] = \sum_{i=1}^{n} x_j \mathbb{P}(X = x_j).$$

The *variance* of a random variable is defined as

$$\mathbb{V}[X] := \mathbb{E}\Big[\big(X - \mathbb{E}[X]\big)^2\Big] = \mathbb{E}\big[X^2\big] - \mathbb{E}[X]^2.$$

Most random variables of interest to us will be continuous and from now on we will take the term random variable to mean continuous real valued random variable. The probability density function and the distribution are connected by the following result.

Theorem 2.2. *Let X be a random variable with distribution F and probability density function f, then $F'(x) = f(x)$ at every continuity point x of f.*

Especially, this means that if F is differentiable, then $f(x) = F'(x)$. If the random variable is not clear from context we write F_X and f_X to denote the distribution and probability density function of X.

2.1.1 Random Vectors and Independence

If $X_1, \ldots, X_n$ are random variables, then we call $\mathbf{X} = (X_1, \ldots, X_n)$ a random vector. The distribution of a random vector is given by

$$F(x_1, \ldots, x_n) = \mathbb{P}(X_1 \leq x_1, \ldots, X_n \leq x_n).$$

If F is differentiable, then the probability density function the random vector $\mathbf{X}$ is given by

$$f(x_1, \ldots, x_n) = \frac{\partial^n F(x_1, \ldots, x_n)}{\partial x_1 \ldots \partial x_n}.$$

Two events A, B are said to be *independent* if

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B),$$

for every pair of events $A, B \in \mathcal{F}$. Similarly, two random variables are independent if

$$\mathbb{P}(X \in A, Y \in B) = \mathbb{P}(X \in A)\mathbb{P}(Y \in B),$$

for every pair of events $A, B \in \mathcal{F}$. For continuous random variables independence can be characterized in terms of their distribution functions or their probability density functions.

Theorem 2.3. *The random variables $X_1, \ldots, X_n$ are independent if and only if*

$$F_{\mathbf{X}}(x_1, \ldots, x_n) = F_{X_1}(x_1) \ldots F_{X_n}(x_n),$$

or equivalently

$$f_{\mathbf{X}}(x_1, \ldots, x_n) = f_{X_1}(x_1) \ldots f_{X_n}(x_n),$$

2.1.2 Transformations

Given a random vector $\mathbf{X}$ and an integrable function $g : \mathbb{R}^m \to \mathbb{R}^m$ the composition $g \circ X = g(X)$ is also a random vector defined on Ω. There are two important theorems that form the basis of the construction of many non-uniform random number generators. The first result shows how the density of the random vector $g(\mathbf{X})$ can be expressed in terms of the density of $\mathbf{X}$.

Theorem 2.4 (see e.g. [16, Theorem 12.7])**.** *Let $\mathbf{X}$ be a random vector and $g : \mathbb{R}^n \to \mathbb{R}^n$ be an injective differentiable function with continuous non-zero derivative. Then the p.d.f. of $\mathbf{Y} = g(\mathbf{X})$ is given by*

$$f_{\mathbf{Y}}(y) = \begin{cases} f_{\mathbf{X}}\big(g^{-1}y\big)\left|\det J_{g^{-1}}\, y\right|, & y \in g\big(\mathbf{X}(\Omega)\big), \\ 0, & \textit{else}, \end{cases}$$

In the special case when X is a real valued random variable, $g : \mathbb{R} \to \mathbb{R}$ and $Y = g(X)$ this reduces to the formula

$$f_Y(y) = f_X\big(g^{-1}(y)\big)\left|\frac{d}{dy}g^{-1}(y)\right|.$$

An immediate consequence of Theorem 2.4 is that if $\mathbf{X}$ is a random variable with distribution f_X and $Y = aX + b$ for constants $a, b > 0$, then

$$f_Y(y) = \frac{1}{a} f\left(\frac{x-b}{a}\right).$$

More examples of how Theorem 2.4 can be applied to form new distributions from old can be found in [9, §I.4.]. The second transformation, that is often used to generate non-uniform random variables is based on the distribution function.

Theorem 2.5. *Let X be a random variable with distribution function F, and U be a uniform random variable, then the random variable*

$$F^{-1}(U) = \inf\{x : F(x) = U\}$$

also has distribution F.

Theorem 2.5 is the basis for the inverse method for generating non-uniform random variables, which we discuss in section 2.5 of this chapter.

2.1.3 Probability Distributions

In this section, we review the probability distributions which will be of importance for this thesis. The most basic is the Poisson distribution which represents a random variable that is either 1 with probability p or false with probability $1 - p$.

Definition 2.1. Let $p \in [0, 1]$, then a random variable X is *Bernoulli distributed*, $\mathrm{Ber}(p)$, if

$$\mathbb{P}(X = 0) = 1 - p, \quad \mathbb{P}(X = 1) = p.$$

A random variable that is Bernoulli distributed is often called a Bernoulli trial, since it can be used to model an experiment with only two outcomes, one with with probability p, and

the other probability $1-p$. A sequence of random variables $(X_i)_{i\geq 1}$ is called a *Bernoulli process* if the random variables X_i are i.i.d. $\text{Ber}(p)$ distributed. Bernoulli processes are very common and a closely related distribution is the geometric distribution.

Definition 2.2. Let $p \in [0,1]$, then a random variable X is *geometrically distributed*, $\text{Ge}(p)$, if

$$\mathbb{P}(X = k) = (1-p)^{k-1}p, \quad k \geq 1.$$

If X is a geometrically distributed random variable, then the probabilities $\mathbb{P}(X = k)$ correspond to the probability that X_k is the first random variable of a Bernoulli process with value 1. Thus, the geometric can be used to study Bernoulli processes and we will apply it to derive properties of rejection sampling algorithms later in this chapter. The expected value, variance, and characteristic function of some common discrete distributions are listed in Table 1 below.

X	$\mathbb{E}[X]$	$\mathbb{V}[X]$	$\phi_X(t)$
Bernoulli(p)	p	$p(1-p)$	$1-p+pe^{it}$
Geometric(p)	$\frac{1}{p}$	$\frac{1-p}{p^2}$	$\frac{pe^{it}}{1-(1-p)e^{it}}$
Binomial(n, p)	np	$np(1-p)$	$(1-p+pe^{it})^n$

Table 1: Common discrete probability distributions.

Next, we will consider continuous distributions. The most simple of all continuous distributions is the uniform distribution defined below.

Definition 2.3. Let $a < b$ be real numbers, then the *uniform distribution* $U(a,b)$ has p.d.f.

$$f(x) = \begin{cases} \frac{1}{b-a}, & x \in (a,b), \\ 0, & \text{else.} \end{cases}$$

The value of a uniform random variable corresponds to a random sample from the interval $[a,b]$. The most important case is when $a = 0$ and $b = 1$, and we shall generally use U to denote a random variable with distribution $U(0,1)$. Many important distributions belong to location-scale families.

Definition 2.4. A family $\mathcal{F}$ of probability distributions $f(x;\mu,\sigma)$, which are parametrized by two parameters: $\mu, \sigma \in \mathbb{R},\ \sigma > 0$ is called a *location-scale family* of distributions if it holds that

$$f(x;\mu,\sigma) = \frac{1}{\sigma} f\left(\frac{x-\mu}{\sigma}\right), \quad f(x) := f(x;0,1),$$

for every $\mu \in \mathbb{R}$ and $\sigma > 0$.

The probability density function $f(x) = f(x;0,1)$ is called the *base* of the family. The parameter μ is called the *location* and different values on μ shifts the p.d.f. on the real axis. The parameter σ is called the *scale* and it scales the p.d.f by $1/\sigma$. If X has p.d.f. $f(x)$ then it follows from Theorem 2.4 that the random variable $\sigma X + \mu$ has p.d.f. $f(x;\mu,\sigma)$. The normal distribution is the standard example of a location-scale family. Some location-scale families are generally expressed using the reciprocal of σ instead, and we write $\lambda = 1/\sigma$,

for the reciprocal. This is often the case with exponential distribution, which we denote as $\text{Exp}(\lambda)$.

More complex distributions do not fit into the category of location-scale families, and a parameter that does not shift or scale the p.d.f. is generally referred to as a *shape* parameter. Many interesting distributions, which are not location-scale families can be classified by a shape and scale parameter, among them the gamma distribution. The shape and scale parameters are often denoted as (k, θ) or (α, β). If the distribution belong to the location-scale family of distributions we have used μ to denote the location parameter and σ to denote the scale parameter. We have included the the basic properties of the continuous distributions we will use in this thesis in Table 2 below.

X	$\mathbb{E}[X]$	$\mathbb{V}[X]$	$f_X(x)$
$\text{Exp}(\lambda)$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$	$\lambda e^{-\lambda x}$, $x \geq 0$
$\text{N}(\mu, \sigma^2)$	μ	σ^2	$\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
$\text{Be}(\alpha, \beta)$	$\frac{\alpha}{\alpha+\beta}$	$\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$	$\frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$, $0 < x < 1$
χ^2_n	n	$2n$	$\frac{1}{2^{n/2}\Gamma(n/2)} x^{n/2-1} e^{-x/2}$, $x \geq 0$
$\Gamma(\alpha, \beta)$	$\alpha\beta$	$\alpha\beta^2$	$\frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}$, $x \geq 0$
$\text{Cauchy}(\mu, \sigma)$	-	-	$\frac{\sigma}{\pi\left[(x-\mu)^2+\sigma^2\right]}$

Table 2: Common continuous probability distributions.

The expected value and variance of the Cauchy distribution is not defined, because the integrals does not converge.

2.2 The Gamma Distribution

The gamma distribution is family which depends on two parameters: the shape α, and the scale β. Formally, the gamma distribution can be defined through its probability distribution function.

Definition 2.5. Let $\alpha, \beta > 0$ be real numbers, then the *gamma distribution* $\Gamma(\alpha, \beta)$ has p.d.f.

$$f(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^\alpha} x^{\alpha-1} e^{-x/\beta}, & x > 0, \\ 0, & x \leq 0. \end{cases}$$

The factor $\Gamma(\alpha)$ which appears in the denominator of the p.d.f. denotes the gamma function evaluated at α (see definition below). The parameter's α and β are also commonly denoted as and a and b. Furthermore, some authors use the reciprocal of β to denote the gamma distribution (see e.g. [19]), then by taking $\lambda = 1/\beta$ and $\Gamma(\alpha, \lambda)$ the p.d.f. can be expressed as

$$f(x) = \begin{cases} \frac{\lambda^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\lambda x}, & x > 0, \\ 0, & x \leq 0. \end{cases}$$

The two notations $\Gamma(\alpha, \beta)$ and $\Gamma(\alpha, \lambda)$ are the two most common in the literature, but other notations also exist (see e.g. [29]). It is good to note that there is no standard convention for numerical libraries whether to use β or the reciprocal λ as the scale parameter. An example of this is the *SciPy* and *NumPy* libraries for Python which are closely tied but use different representations for their scale parameters in their gamma distribution objects.[7] Thus we recommend that users carefully check which convention is followed by the libraries they use. We will use the location-scale parametrization with shape α and scale β to describe the gamma distribution in all equations and code that follows from now on.

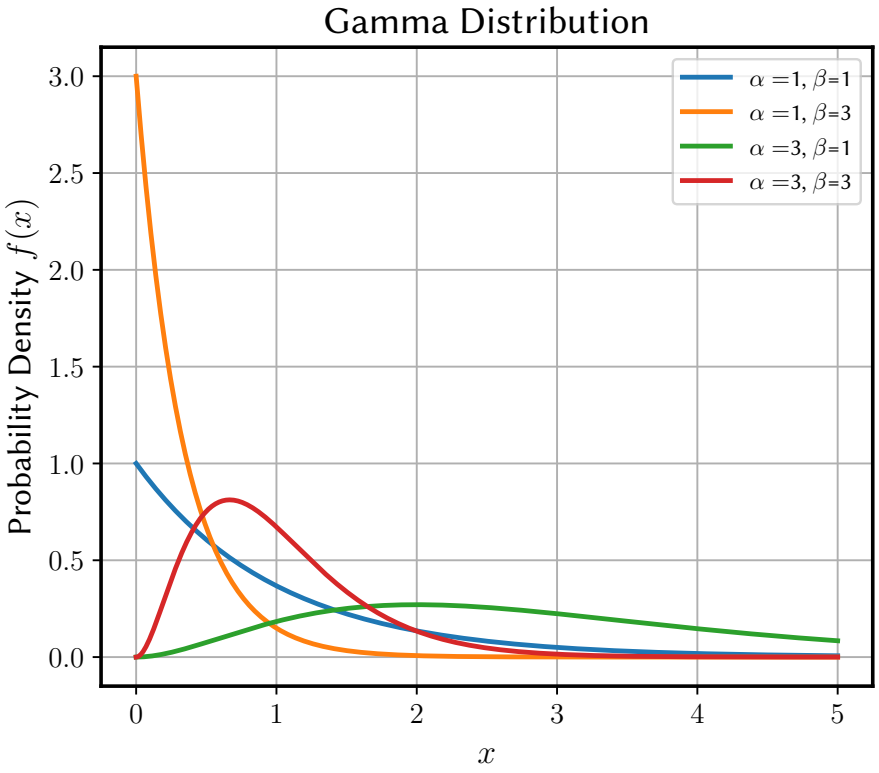

Figure 1: Gamma distributions with different shape and scale parameters.

The *gamma function* $\Gamma : (0, \infty) \to [0, \infty)$ which appears in the quotient of the p.d.f. of the gamma distribution is defined as

$$\Gamma(x) := \int_0^\infty t^{x-1} e^{-t}\, dt.$$

By integration it can be shown that $\Gamma(1/2) = \sqrt{\pi}$ and that $\Gamma(1) = 1$. Using partial integration it follows that

$$\Gamma(x+1) = x\Gamma(x), \quad x > 0, \tag{1}$$

and especially

$$\Gamma(n+1) = n!\,, \quad n \in \mathbb{N}. \tag{2}$$

[7] NumPy's `numpy.random.Generator.gamma` has a scale parameter corresponding to β, while SciPy's `scipy.stats.gamma` object has a scale parameter corresponding to the reciprocal λ.

Even though the gamma distribution does not belong the the class of location-scale families it is still linear in the shape parameter α, and β is a proper scale parameter. We formalize this in Theorem 2.6 below.

Theorem 2.6. *Let $\alpha_1, \ldots, \alpha_n, \beta > 0$ and $c > 0$ be a positive constant. If $X_i \sim \Gamma(\alpha_i, \beta)$, then*

$$cX_1 + \cdots + cX_n \in \Gamma(\alpha_1 + \cdots + \alpha_n, c\beta).$$

This can be proved by using characteristic function (see e.g. [9, p. 402]). Theorem 2.6 is fundamental for efficient generation of gamma random variates as it tells us that in order to generate a $\Gamma(\alpha, \beta)$ variate we can generate a $\Gamma(\alpha, 1)$ variate and multiply it by β. Thus, it is useful to note the density function of a $\Gamma(\alpha, 1)$ distributed random variable is given by

$$f(x) = \frac{e^{-x}x^{\alpha-1}}{\Gamma(\alpha)}.$$

The gamma distribution family is very large and it contains several other families of distributions. Below, we state a useful result that relate the gamma distribution to the exponential and chi-square distribution which can be found in [19] and [42].

Theorem 2.7. *Let X be a gamma distributed random variable.*

1. *If $X \sim \Gamma(1, \beta)$, then $X \sim \text{Exp}(\lambda)$ with $\lambda = \frac{1}{\beta}$.*
2. *If $X \sim \Gamma(n/2, 1)$, then $X \sim \chi^2_n$, i.e. the chi-square distribution with n degrees of freedom.*

Item 1 from Theorem 2.7 combined with the linear property of the gamma distribution give additional insight how the gamma distribution can be used to model stochastic processes. If n is a positive integer, then a $\Gamma(n, \beta)$ random variable can be used to model a sum of exponential random variables.

2.3 Pseudo Random Number Generation

The basic assumption behind all random number generators used for simulation purposes is that we can simulate uniform random numbers. In the most general sense a *random number generator (RNG)* is a method that can produce random numbers according to some probability distribution. Truly random numbers can be generated by measuring physical phenomena, such as radioactive decay or noise in electric circuits. Most CPU vendors include this type of physical random devices in their motherboards and the major operating systems support cryptographically safe random number generation by combining such hardware RNGs with other random sources collected from the system. The RNGs which depend on physical devices and other sources of entropy are generally very robust and secure but not optimal to use for simulations [24]. The main drawbacks of using physical devices for random number generation is that they are costly and complicated to use and the random sequences are not reproducible.

For stochastic simulation purposes, where the speed with which random numbers can be generated matters, algorithms are used to generate random numbers instead. Algorithms that generate random numbers are called *pseudorandom number generators (PRNGs).* A PRNG is deterministic and therefore not truly random. Hence, the word *pseudo* is used to describe such algorithms. The terms RNG and PRNG are often used interchangeably, and from now on we will use RNG to mean a PRNG.

2.3.1 Basic definitions

There are a few different formal definitions of random number generators which appear in the literature (compare for instance [18, 31, 20]). The difference is mainly in the generality of the definition. We will use the following definition of a random number generator which is a slightly modified version of the definition given in [20].

Definition 2.6. A *(pseudo)random number generator* is quintuple (S, s_0, φ, U, u) consisting of:

S is the of states of the generator,

$s \in S$ is the state of the generator,

$\varphi : S \to S$ is the transition function between states,

U is the set of output states, and

$u : S \to U$ is the output function of the generator.

Given a state $s_0 \in S$ the sequence $s_1 = \varphi(s_0), s_2 = \varphi \circ \varphi(s_0), \ldots$ is called the *random (state) sequence* of the generator and s_0 is called the *seed*. Any $s_0 \in S$ can be used as the seed, however for some generators not all states are recommended to be used as seeds. The *period* of a random generator is the smallest integer p for which there exists a sequence $s_0, \ldots, s_p$ such that $s_p = s_0$. Given a generator of period p and a nonnegative integer $\pi \ll p$ it is possible to divide the state sequence into several sequences of length π:

$$s_0, \ldots, s_{\pi-1} \quad s_\pi, \ldots, s_{2\pi-1}, \quad \ldots,$$

and such subsequences are called *streams*. In theory, all RNGs can be divided into multiple streams, however it is not always practically possible. Some RNGs support computationally cheap *look ahead*, i.e. the option to compute s_{n+k} given s_n in constant time (with respect to k), compared to recursively computing the state which is an $O(k)$ operation. The ability to split a generator into multiple streams is very important for practical purposes because multiple threads can use copies of the same generator starting with different seeds.

2.3.2 Categories of random number generators

Random number generators can be broadly classified into two categories:

1. uniform RNGs.
2. non-uniform RNGs.

Uniform RNGs are those where the output sequence simulates i.i.d. random variables U_i over the space U. When the output distribution is non-uniform, the generator is called a non-uniform RNG. All practical algorithms used for generating non-uniform random numbers make use of uniform RNGs as their source of randomness.

Another important aspect of RNGs is whether the state space (and hence the upcoming numbers in a sequence of outputs) can be determined from previous outputs. For cryptographic purposes it is neccessary that the state of a generator can not be determined in a reasonable amount time given an output sequence. *This is not neccessary for simulation purposes and we will not treat cryprographic RNGs. In this thesis all RNGs are non-cryptographic. Given an*

output sequence, the state of the RNGs can generally be recovered and the RNGs we present are not safe to use for cryptographic purposes.

2.3.3 Testing Random Number Generators

There are several test programs that can be used to test a uniform RNG for statistical flaws, notably, the TestU01 test-suite [26] and the PractRand test-suite [10]. These are used as a benchmark for correctness in most publications in the field. The TestU01 paper [26] covers much of the theory for statistically testing the output of a RNG. Another good source that contain a lot of information about RNG testing is [18] which also has a section devoted to randomness from a more philosophical perspective. Another test, that is commonly used, is based on Hamming weight dependencies, and a recent method to test for this can be be found in [6].

There are also many statistical methods which can be used to determine whether it is likely that a sequence of random numbers belong to a non-uniform distribution and that can be used to test non-uniform RNGs. A well-known test that can be used for this purpose is the *Kolmogorov-Smirnov (KS) test* (see e.g. [18] or [42]) , and it is based on the following test statistic.

Definition 2.7. Let $X_1, \ldots, X_n$ be a sample from a population with unknown distribution and F_0 a distribution function. Then, the *Kolmogorov-Smirnov (KS) test statistic* is given by

$$D_n(F_0) := \sup_{x \in \mathbb{R}} \left\| \frac{|\{k : X_k < x\}|}{n} - F_0(x) \right\|.$$

It can be shown that $D_n(F_0)$ should be close to 0 if the sample $X_1, \ldots, X_n$ is distributed according to the distribution F_0. The KS-test is a hypothesis test that can be used to test whether a sample $X_1, \ldots, X_n$ is from a distribution F_0. The KS-test uses the KS-test statistic to test the null hypothesis

$$H_0 : X_1, \ldots, X_n \sim F_0$$

against the alternative

$$H_1 : X_1, \ldots, X_n \nsim F_0.$$

The null hypothesis is accepted if $D_n(F_0)$ is smaller than some threshold, and rejected otherwise. The KS-test is well studied and widely available in software. There are two types of errors commonly discussed when performing hypothesis tests:

Type-I error is the error that occur when the null hypothesis H_0 is true but rejected anyway, i.e. a false positive.

Type-II error is the error that occur when the null hypothesis H_0 is false but not rejected, i.e. a false negative.

The *p-value* associated with test, is the probability of a Type-1 error, given the sample the test is based on. Thus, if the p-value is low, it possible to reject the null hypothesis knowing that the probability of the null hypothesis being true is low. For this reason, the null hypothesis is often rejected if the p-value is lower than some predetermined level of significance. Commonly, a p-value of 0.05 and 0.01 is used when we want to reject the null hypothesis. However, the KS-test is often used to determine whether a sample follows a specified distribution, and in that case we are not looking to reject the null hypothesis. It may not be clear what p-values are considered good enough to not reject the hypothesis H_0, but generally when performing a KS-test we want the p-value to be as high as possible.

2.4 Uniform Random Number Generators

In practice, we are most often interested in uniform distributions over some subset of $\mathbb{R}$ or $\mathbb{Z}$. On computers, we represent numbers by bits, and in this section, we consider uniform generators with an output space consisting of k bits for some nonnegative integer.

Definition 2.8. A *uniform k-bit generator* is a random generator with output space $U = \mathbb{Z}_2^k$, i.e. the finite field of random $k-$bit vectors.

The output sequence $u_i = u(s_i)$ is supposed to mimic a uniform distribution on the output space U, hence the term uniform. Thus, by interpreting the k bits as an unsigned integer, a uniform $k-$ bit generator can be thought of as a generator which generate values for i.i.d. uniform random variables taking values in the set $\{0, 1, 2, \ldots, 2^k - 1\}$. A continuous uniform distribution over $[0, 1)$ is then generally approximated by taking

$$U_i = \frac{u_i}{2^k}$$

We write U(m) to denote a uniform distribution over the set $\{0, 1, 2, \ldots, m\}$. It is easy to convert random numbers in the interval $[0, 1)$ to random numbers in the interval $[a, b)$ by the transformation $X_i = (b - a)U_i + a$. However, in many situations it is desirable to generate uniform random numbers in an open interval (a, b). There are three common methods used to convert a random integer u, from $\{0, \ldots, m - 1\}$ to a random number in an open interval $(0, 1)$:

1. Generate u from U(m) until $u \neq 0$, and set
$$U = \frac{u}{m}. \tag{3}$$
2. Generate u from U(m) and set
$$U = \frac{u + 1}{m + 1}. \tag{4}$$
3. Generate u from U(m) and set
$$U = \begin{cases} \frac{u}{m+1}, & u > 0, \\ \frac{m}{m+1}, & u = 0. \end{cases} \tag{5}$$

This can be used to generate a random number in the interval (a, b) by the transformation $X = (b - a)U + a$, as above. It is very common that software libraries that expose RNGs for uniform distributions generate samples in half-open intervals on the form $[a, b)$, this is the case with `numpy.random.Generator.uniform`[8] from the popular *NumPy* library used for numerical computation in Python and the C++ standard template libraries `std::uniform_real_distribution`[9].

Uniform RNGs have been extensively studied and most books in the field of stochastic simulation and Monte Carlo methods cover the basic principles of uniform random number generation, among them Robert & Casella [44], Asmussen & Glynn [4], and Knuth [18]. There are also review papers by important figures in the field which cover the general theory: [20], and [31]. The rest of this section is devoted to uniform RNGs and we closely follow [25] and [19].

[8] https://numpy.org/doc/stable/reference/random/generated/numpy.random.Generator.uniform.html#numpy.random.Generator.uniform

[9] https://en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution

2.4.1 Finite Fields and Binary Arithmetic

Most pseudo random number generators are based on arithmetic over finite fields. In this section, we introduce some notation and concepts which will be used in the forthcoming sections. A *binary digit* or *bit* is a number $x \in \{0, 1\}$. An ordered collection of bits is called a *bit string/array* and can be written on the form

$$\mathbf{x} = (x_0, \cdots, x_{n-1}), \quad x_j \in \{0, 1\}.$$

In general we consider bit strings as elements of the finite field $\mathbb{F}_2^n \cong (\mathbb{Z}/2\mathbb{Z})^n$, which consists of all n-vectors $\mathbf{x}$ on $\{0, 1\}^n$ under multiplication and addition modulo 2. We use $\oplus$ to denote addition in $\mathbb{F}_2^n$ (i.e. element wise addition modulo 2), and $\otimes$ to denote multiplication in $\mathbb{F}_2^n$ (i.e. element wise multiplication modulo 2). Arithmetic in $\mathbb{F}_2$ is closely connected with the binary operators implemented in computer hardware. The bitwise `XOR` operation corresponds to addition

$$x \texttt{ XOR } y := x \oplus y.$$

Similarly, The bitwise `AND` operation corresponds to multiplication modulo 2

$$x \texttt{ AND } y := x \otimes y.$$

A large family generators can be expressed using arithmetic using matrix arithmetic over $\mathbb{F}_2^n$. These generators are said to be based on matrix linear recurrences modulo 2 (see e.g. [41]).

Definition 2.9. A RNG is said to be a *matrix linear recurrence modulo 2 generator* if it has *state space* $\mathbb{F}_2^k$ and output space $\mathbb{F}_2^w$, for some positive integers w, k, and there exists matrices $A \in \mathbb{F}_2^{k\times k}$, and $B \in \mathbb{F}_2^{w\times k}$ such that

$$\begin{aligned} \mathbf{x}_{n+1} &= A \otimes \mathbf{x}_n, \\ \mathbf{y}_{n+1} &= B \otimes \mathbf{y}_n, \end{aligned}$$

where $\mathbf{x_n}$ is the *state vector*, and $\mathbf{y_n}$ is the *output vector*.

The matrices A, and B are called the *state transition matrix*, and *output transformation matrix* respectively. The output vector can be interpreted as an unsigned w-bit integer, z, in the range $[0, 2^w - 1]$ by setting

$$z = \sum_{i=1}^{w} y_i 2^{w-i}, \quad \mathbf{y} = (y_1, \ldots, y_n). \tag{6}$$

It is common to see the output specified in the interval $[0, 1)$ using a dyadic expansion on the form

$$u = \sum_{i=1} y^i 2^{-i}.$$

Note that this is equivalent to using the integer output from equation (6) combined with the standard transformation from unsigned integers to real values given in equation (3). This can be seen by dividing z by 2^w which yields the identity

$$\frac{z}{2^w} = \sum_{i=1}^{w} y_i 2^{w-i} 2^{-w} = \sum_{i=1} y^i 2^{-i} = u.$$

The choice of the modulus 2 is made for performance reasons and it's possible to extend this method to any finite field $\mathbb{F}$. This is called the *multiple-recursive matrix method for pseudo random number generation* (see e.g [38]) introduced in [39]. Many of the RNG algorithms used today fit into this framework. One advantage of interpreting the generators as linear transformations over finite fields is that there is rich mathematical theory developed for such transformation (see for instance the book [28]) which can be used to reason about the statistical quality of such generators.

2.4.2 Multiple Recursive Generators

The linear congruential generators (LCGs) are among the most simple and well-known uniform random number generators. A *linear congruential generator* is a generator with state $S = [m] = \{0, ..., m-1\}$ and transition function

$$\varphi(s) = as + c \mod m.$$

The constant a is called the *multiplier*, m is called the *modulus*, and c is called the *increment*. Thus a linear congruential generator corresponds to a random sequence of the form

$$s_{n+1} = as_n + c \mod m, \quad u_i = s_i$$

LCGs are simple, fast, and well-studied but too simple for most applications. However, they are still of interest to practitioners because multiple LCGs can be combined to create generators that perform well. There is much published theory about LCGs, and a good overview of the theory can be found in Knuth [18, §3.2.1]. The choice of the values for the constants a, m, c are of huge importance for the statistical performance of LCGs, and good choices for the constants can be found in the papers [22], and [47]. However, the several flaws of LCGs still make them unpractical for large-scale simulations. By comining several LCGs under a recursive relation it is possible to achieve good statistical properties and this is the idea behind multiple recursive generators.

Definition 2.10. A *multiple recursive generator (MRG)* of order $k \geq 1$ and modulus $m \geq 2$, is a generator with state space $\mathbb{Z}_m^k$ and state $\mathbf{x}_n = (x_n, x_{n-1}, x_{n-k})$, and transition function defined by the recurrence relation

$$x_n = a_1 x_{n-1} + \cdots + a_k x_{n-k} \mod m.$$

The output sequence is created from the sequence of integers (x_n), usually by division with m which leads to a sequence which approximately uniform in the half open interval $[0, 1)$. It is common to write a multiple recursive generator on matrix form:

$$\mathbf{x}_{n+1} = A\mathbf{x}_n \mod m,$$

where A is the matrix given by

$$A = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ \vdots & 0 & \ddots & \ddots & \vdots \\ 0 & 0 & \ddots & 1 & 0 \\ a_k & a_{k-1} & \cdots & a_2 & a_1 \end{pmatrix}$$

Note that the special case when $k = 1$ corresponds to a linear congruential generator. The efficiency of MRGs largely depend on two factors, the number of nonzero multiplicative coefficients a_n and their values, and the modulus m. However, efficient choices from a computational perspective does not achieve good-quality random sequences [25, section 3.2.12]. A solution to this problem is to combine multiple lighweight MRGs, this is known as *combined MRGs*. A well-known combined MRG, which pass all the statistical tests in the Test01 test-suite, is the `MRG32k3a` generator from [21]. The `MRG32k3a` consists of two MRGs, both of order 3,

$$x_{i,n} = a_{i,1}x_{i,n-1} + a_{i,2}x_{i,n-2} + a_{i,2}x_{i,n-3} \mod m_i, \quad i = 1, 2.$$

The two MRGs are combined to give an output

$$z_n = x_{1,n} - x_{2,n} \mod m_1.$$

The period is about $3 \cdot 10^{157}$ and the specific values for the coefficients of the `MRG32k3a` generator is tabulated in Table 3 below.

i	$a_{i,1}$	$a_{i,2}$	$a_{i,3}$	m_i
1	0	1403580	-810728	$2^{32} - 209$
2	527612	0	-1370589	$2^{32} - 22853$

Table 3: Coefficients and modulus for the two MRGs in `MRG32k3a`.

2.4.3 Xorshift Generators

Another type of generators introduced by Marsaglia in 2003 [32] are the so called *xorshift* generators. The xorshift generators are a part of the family of the matrix linear recurrence modulo 2 generators. We start by recalling the definition of binary shift operations which are efficiently implemented in modern processors.

Definition 2.11. The *left shift* $\mathbf{x} << 1$ of $\mathbf{x} \in \mathbf{F}_2^k$ is the vector $\mathbf{y}$ given by

$$y_{i-1} = x_i, \quad i = 2, \ldots n, \qquad y_n = 0.$$

Similarly, the *right shift* $1 >> \mathbf{x}$ is the vector $\mathbf{y}$ given by

$$y_{i+1} = x_i, \quad i = 1, \ldots n-1, \qquad y_1 = 0.$$

The left and right shift operators correspond to linear transformations over $\mathbb{F}_2^k$. Especially we can express the left and right shift of $\mathbf{x}$ as

$$\mathbf{x} << 1 = L \otimes \mathbf{x},$$
$$1 >> \mathbf{x} = R \otimes \mathbf{x},$$

Where L and R are $k \times k$ matrices with ones one the super- and sub-diagonal respectively, and zeroes elsewhere. For $a > 0$ we write $\mathbf{x} << a$ and $a >> \mathbf{x}$ to denote a consecutive shift

operations. An operation on $\mathbf{x} \in \mathbb{F}_2^k$ is said to be *xorshift* operation if it can be written on any of the forms:

$$\mathbf{x} \oplus (\mathbf{x} << a),$$
$$(a >> \mathbf{x}) \oplus \mathbf{x}.$$

Using the matrices R and L the left and right xorshift operations can be described as

$$(I + L^a) \otimes \mathbf{x} = \mathbf{x} \oplus (\mathbf{x} << a),$$
$$(I + R^a) \otimes \mathbf{x} = (a >> \mathbf{x}) \oplus \mathbf{x}.$$

Definition 2.12. Let k, r, w be positive integers satisfying $k = wr$. A RNG is said to be an *xorshift generator* if it has state $\mathbf{x}_n = (\mathbf{v}_{n+1-r}, \ldots, \mathbf{v}_\mathbf{n}) \in \mathbb{F}_2^k$, where $\mathbf{v}_i \in \mathbb{F}_2^w$ and the state transition function can be written on the form

$$\mathbf{v}_\mathbf{n} = \sum_{i=1}^{p} A_i \mathbf{v}_{n-m_i},$$

where A_i is either a product of xorshift matrices, the identity matrix, or zero, and $0 < m_i \leq r$.

The transition function of the xorshift operators can be rewritten as a single matrix multiplication modulo 2 over the state vector $\mathbf{x}$ and it can be shown that xorshift RNGs belong the class of matrix linear recurrence modulo 2 generators (see e.g. [40]). Panneton and L'Ecuyer further show in [40] that all the generators presented by Marsaglia in [32] fails the *BigCrush* test from the TestU01 [26] test-suite.

Even if there are other generators with better statistical properties, there is one generator from [32] which is still widely used today and it is known as the `XORWOW` generator. It is an xorshift generator where the integer output is combined with a so called *Weyl sequence* through addition. The pseudocode for the `XORWOW` generator is given in Algorithm 1.

Algorithm 1 Marsaglia's `XORWOW` generator [32]

// word size is 32, i.e. all variables are 32-bit unsigned integers.
Initialization of constants:
$x = 123456789$, $y = 362436069$, $z = 521288629$, $w = 88675123$, $v = 5783321$, $d = 6615241$
$t = x \oplus (x \gg 2)$
$x = y$
$y = z$
$z = w$
$w = v$
$v = (v \oplus (v \ll 4)) \oplus (t \oplus (t \ll 1))$
$d = d + 362437$
return $d + v$ // unsigned integer addition

2.4.4 Mersenne Twister and Feedback Shift Register Generators

Another common family of random number generators, which are also based on recurrences modulo 2, are the so called feedback shift register generators. In his 1965 paper [48] Tausworthe presented a technique for generating random nunbers that is known today as a *linear*

shift feedback register (LFSR) or *Tausworthe* generator. An exentension of this method is the *generalized feedback shift register (GFSR)* generator by Lewis and Payne [27], and later the *twisted GFSR* by Matsumoto and Kurita [34, 35]. It is possible to construct GFSRs with very large periods and good statistical properties and one of the most known twisted GFSRs is the *Mersenne Twister* introduced by Matsumoto and Nishimura in [36]. The Mersenne Twister algorithm is more complex than the RNG algorithms we have discussed in the previous section, but is described in detail in the original paper [36]. The best known Mersenne Twister algorithm is the `MT19937` which has a period of length $2^{19937} - 1$ and a state consisting of 19937 bits. Furthermore, the authors have released C-code, under a BSD-license, for their implementation of the algorithm which can be accessed on the *Mersenne Twister Home Page*[10].

The Mersenne Twister is well-known, available in many software libraries and considered to have good statistical properties (see e.g. [23]). There is also a variant of the Mersenne Twister optimized for the GPU which is called `MTGP` introduced [45]. One drawback of the Mersenne Twister is that the state vector is very large (19937 bits are required to hold the state for `MT19937`). There are some known statistical flaws in the output generated by `MT19937` (see e.g. [51]) but the generator is generally considered to be of high quality [23].

2.4.5 Counter-Based Generators

Another type of RNGs, which are often used in cryptography, are the *counter-based generators*. Cryptographically secure generators tend to be slower than the generators used for simulations. However, there are counter-based generators which are not cryptographically secure but still pass all tests in statistical test suites like TestU01. In [46] the authors present several counter-based generators called *Philox* which are efficient to use in simulations and pass the Bigcrush test suite from TestU01. Philox generators can also be implemented efficiently on GPUs, and the generator `PHILOX4_32_10` from [46] is available in many RNG libraries.

2.5 Non Uniform Random Number Generation

For non-uniform generators the book by Devroye [9] is a standard reference which contains most techniques used in the design of non-uniform generators and it also makes interesting comparisons of generators published before its publication. A more recent and lighter account of non-uniform random number generation, that cover a subset of the material in [9], can be found in [19, Chapter 3 & 4].

Even though the aforementioned texts provide an overview of some of the classical random number generation theory none of them discuss implementation possibilities on modern CPU architectures or accelerators such as GPUs. When it comes to gamma generation, there are many algorithms published and comparisons have been made with respect to their performance on CPUs, for instance in the recent survey paper by Luengo [29]. To the author's knowledge, no performance comparison of these algorithms on GPUs have been published. The known gamma generators use rejection sampling and there are methods

[10] Mersenne Twister Home Page: http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/emt.html accessed 2024-05-01.

which are well-known by practitioners that can speed up rejection sampling on GPUs. An informal account of these techniques is given by David Thomas on his personal web page [50], and a recent paper [43] give a more formal presentation of the same techniques.

All practical methods in use today for generating non-uniform random numbers are based on the assumption that uniform random numbers can be generated efficiently. The general idea is to use transformations that convert uniform random variables to non-uniform random variables. This can be used to generate a samples from non-uniform distributions using samples from a uniform distribution (see e.g. for an overview of these techniques [9, 4, 19, 18, 44]). Two of the most widely used schemes for non-uniform random number generation are the inverse method, and rejection sampling, which we introduce in this section.

2.5.1 Inverse Method

The inverse method is based on Theorem 2.5 and is used to generates a sample x, of a random variable X, by sampling u from U(0, 1) and then use the inverse of the c.d.f. of X to find x. This yields the following algorithm known as the *inverse method* or *inversion sampling*.

Algorithm 2 Inverse Transform Sampling Method

1: Sample $u \sim U(0, 1)$
2: Let $x = F^{-1}(U)$
3: **return** x

If F is the cumulative distribution function of X, then it holds that

$$F_X^{-1}(u) = \inf\{x \in \mathbb{R} \ : \ F(x) \geq u\}.$$

The inverse method is especially useful when the inverse F^{-1} is easy to compute. An example of this is the exponential distribution which has p.d.f. $f(x) = \lambda e^{-\lambda x}$. The c.d.f. is given by $F(x) = 1 - e^{-\lambda x}$ and the inverse is given by

$$F^{-1}(u) = -\frac{1}{\lambda} \log(1 - u). \tag{7}$$

In many cases the inverse of c.d.f. may not have simple analytical form. The inverse method can still be used, but the inverse must be computed through numerical inversion methods (e.g. Newton-Rhapson's method). We will not use any numerical inversion in this work, but we will use the inverse method to generate exponential random variables using the identity in equation (7). Another distribution which can be sampled from efficiently using the inverse method is the Cauchy distribution. The c.d.f. of a Cauchy(μ, σ) distributed random variable is given by

$$F(x) = \frac{1}{2} + \frac{1}{\pi} \arctan\left(\frac{x - \mu}{\sigma}\right).$$

The inverse of the c.d.f is in this case given by

$$F^{-1}(u) = \mu + \sigma \tan\left[\pi \left(u - \frac{1}{2}\right)\right]. \tag{8}$$

2.5.2 Rejection Sampling

A highly versatile technique which can be used to sample from complex distributions or otherwise computationally untractable distributions is rejection sampling. The general idea behind rejection sampling is based on following two mathematical identities.

Theorem 2.8 (see e.g. [25, Proposition 4.1]). *If $U \sim \mathrm{U}[0,1]$ is independent from X, then the random vector $(X, Ucf(X))$ is uniformly distributed on*

$$A = \{(x,u) \in \mathbb{R}^2 : 0 \leq u \leq f(x)\}.$$

Conversely, let (X,Y) be a random vector and $f : \mathbb{R} \to \mathbb{R}$ is an integrable function. If (X,Y) is uniformly distributed on the set A then X has p.d.f. f.

The above idea can be combined with the following result to replace the sampling of a complex distribution to repeated sampling from a a uniform distribution and another distribution.

Theorem 2.9 (see e.g. [9, Theorem 3.2]). *Let (X_i) be a sequence of $i.i.d.$ random vectors taking values in $\mathbb{R}^k$, for some $k \geq 1$, and $A \in \mathcal{B}(\mathbb{R}^{\mathbb{k}})$ with $\mathbb{P}(X_i \in A) > 0$. If $N := \min\{i \in \mathbb{N} : X_i \in A\}$, then the distribution of X_N is given by*

$$\mathbb{P}(X_N \in B) = \frac{\mathbb{P}(X_i \in A \cap B)}{\mathbb{P}(X_i \in A)}, \quad \textit{for every } B \in \mathcal{B}(\mathbb{R}).$$

Note that if the random variables (X_i) are uniformly distributed over the set A, then this implies that X_N is also a random variable which is uniformly distributed over A. We explain below how this can be used to construct an algorithm to generate samples for a random variable X by sampling random variables from another distribution. Let f be the density of X, and assume that we can sample random variables Y_i with density g. If $\mathrm{supp}(f) \subset \mathrm{supp}(g)$ and there exists a constant $c > 1$ such that

$$f(x) \leq Mg(x), \quad \text{for every } x \in \mathbb{R},$$

then we can sample from X by Algorithm 3 below.

Algorithm 3 Rejection Sampling

1: Sample $y \sim Y$ and $u \sim U(0,1)$.
2: **if** $u \leq M \frac{f(y)}{g(y)}$ **then**
3: Let $x = y$.
4: **else**
5: **goto** 2.
6: **end if**

This is known as *rejection sampling* or the *acceptance-rejection (AR)* algorithm. The acceptance probability for rejection sampling is given by

$$p := \mathbb{P}\left(U \leq M \frac{f(Y)}{g(Y)}\right) = \frac{1}{M}.$$

A natural question to ask is how many iterations, N, the rejection sampling algorithm will require to produce a sample X. Each iteration of the rejection sampling procedure produces

two independent random variables Y_i, U_i, which can be thought of as a a random vector (Y_i, U_i). We define the random variable N as

$$N := \min \left\{ i \in \mathbb{N} : U_i \leq M \frac{f(Y_i)}{g(Y_i)} \right\}.$$

Then, N represents the number of iterations the rejection sampling algorithm will take, and it is a stopping time (see e.g. [16]) with respect to the $\sigma-$algebra generated by the process (Y_i, U_i). Fruthermore, since every iteration of the rejection sampling algorithm corresponds to a bernoulli trial we can conclude that $N \sim \text{Ge}(p)$ When analyzing the performance of rejection sampling algorithms it is common to use the rejection probability, $\rho = 1 - p$, instead of the acceptance probability p. In figure 2, the probability mass function of N is plotted for four different values of ρ. Since N is geometrically distributed, the expected

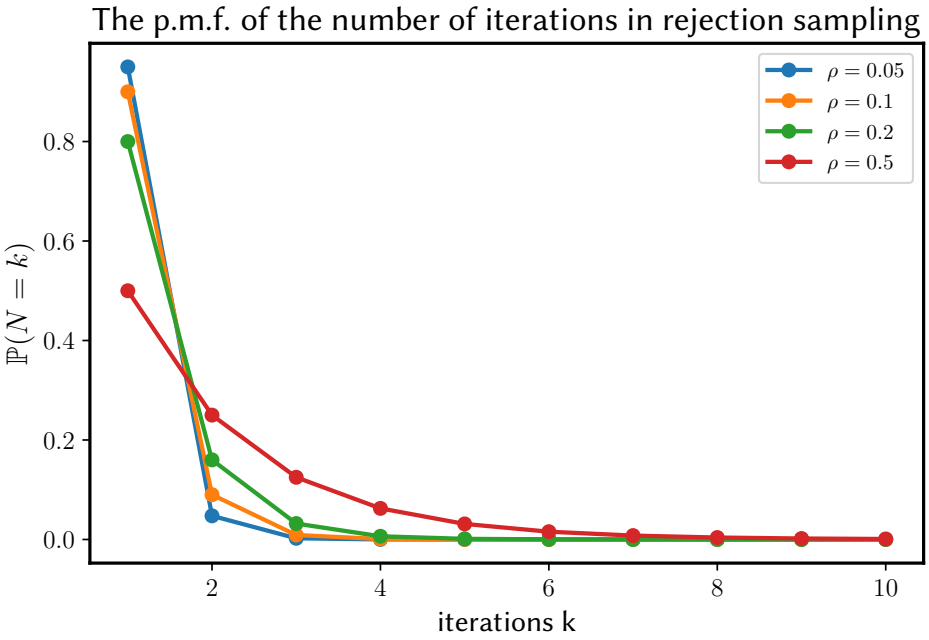

Figure 2: Probability mass function for the number of iterations N required in rejection sampling to generate one sample for different rejection probabilities ρ.

number of iterations for each sample of X is given by

$$\mathbb{E}[N] = \frac{1}{p} = c.$$

Using the relationship between ρ and p we get the identities

$$\mathbb{E}[N] = \frac{1}{1-\rho}, \quad \rho = 1 - \frac{1}{M} = \frac{M-1}{M}.$$

In Figure 3, the expected value of N is displayed for different values of the rejection probability.

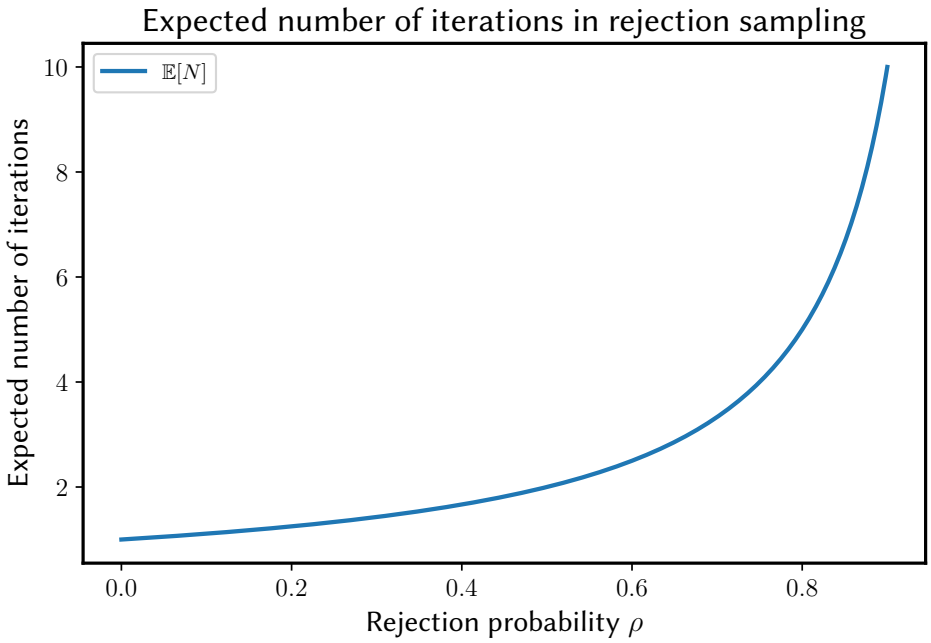

Figure 3: Expected value of the number of iterations N required by the rejection sampling to generate one sample with rejection probability ρ.

2.6 Graphical Processing Units

In this section we review the architecture of NVIDIA GPUs and give an overview of CUDA which is a programming language for programming NVIDIA GPUs. GPUs differ from CPUs in that they have thousands of threads which can execute instructions concurrently. This massive parallelism comes at a cost and GPUs have much less memory per core and share control flow logic among multiple cores.

2.6.1 GPU Architecture

At the highest level a GPU consists of several *streaming multiprocessors (SMs)* which are connected to a L2 cache. The L2 cache is connected to a larger DRAM memory on the GPU. The SMs each have their own L1 cache and instruction units, and they also have their own execution units, sometimes referred to as *CUDA cores*. An illustration of the hardware design of the SMs is given in Figure 4 below. The SMs have a *single instruction, multiple threads (SIMT)* design, also known as an *array processor* in Flynn's taxonomy [11]. This means that different cores have different registers and memory, but are collected in groups which share the same instruction and control unit. In practice this means that the cores operate in groups which all execute the same instructions. We will discuss the programming aspect of this more in detail in the next section.

The number of instructions that can be performed by a CUDA core per clock cycle largely depends on two factors: the instruction and the data type. The NVIDIA consumer GPUs intended for graphics purposes, such as computer games, have worse double precision

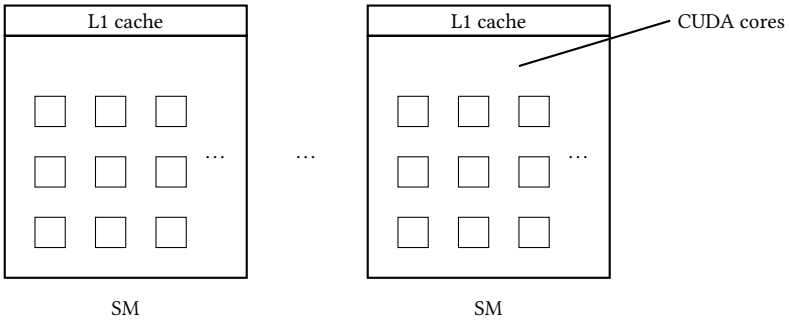

Figure 4: Illustration of GPU architecture showing SMs, CUDA cores, and L1 cache.

performance compared to high-end GPUs aimed at HPC. Newer NVIDIA GPUs also have tensor cores which support complex instructions aimed at machine learning work loads. Figure 2 in the GPU Performance Background User's Guide [11] show how the number of CUDA and Tensor cores per SM differ between the Volta, Turing, and Ampere NVIDIA Architecture for multiply-add operations of various floating point and integer data types.

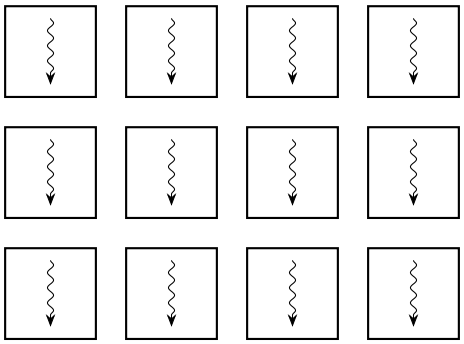

Figure 5: Each CUDA core supports a hardware thread, displayed as an arrow.

2.6.2 Programming NVIDIA GPUs using CUDA

NVIDIA has developed its own programming language intended for programming NVIDIA GPUs, which is called CUDA. CUDA is an extension of C and C++ which support custom functionality for writing code that executes on GPUs. CUDA also exists as extension of the Fortran of programming language, but from now on we will mean CUDA C++ when referring to CUDA. In the CUDA programming model GPUs are referred to as *devices* and CPUs are referred to as *hosts*. Functions which execute on the device are called *CUDA kernels*.

The CUDA programming model exposes the underlying GPU hardware through threads. Threads are executed on the CUDA cores and each CUDA thread executes on a CUDA core (see Figure 5). The SIMT architecture of GPUs means that threads are ordered in small groups that share control logic. In CUDA programming terms, these are called *warps*, and currently the *warp size* (i.e. number of threads in a warp) is 32 but it may change in future GPU generations. If the program control flow leads to different threads in the same warp to take different branches this is handled by first executing the first branch for some threads,

[11] https://docs.nvidia.com/deeplearning/performance/pdf/GPU-Performance-Background-User-Guide.pdf

while leaving the other threads idle, and then executing the second branch for the other threads while leaving the threads which executed the first branch idle. This phenomeonon is called *warp divergence* or *thread divergence* and should be avoided, when possible, for best performance. Figure 6 below illustrate how some cores are left idle while warp divergence occurs. For this reason it is often the case that code with very complicated control flow performs better on CPUs than GPUs. Oftentimes, it may be worth to switch algorithms that include many branches to algorithms with less control flow and more computation to maximimize the performance on GPUs.

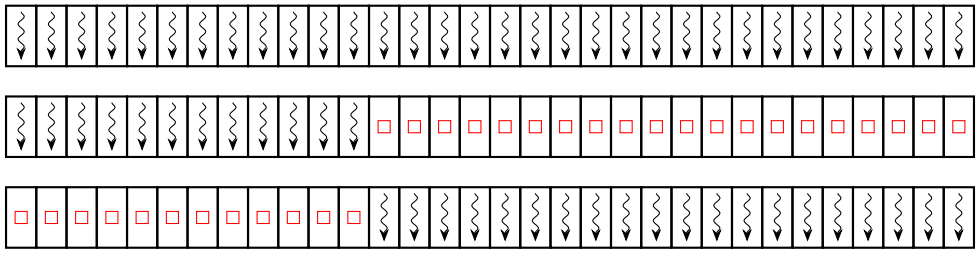

Figure 6: Visualization of warp divergence. The arrow indicates that the thread is doing work and the red square indicates that the thread is idle.

Threads are grouped together in *thread blocks* (also called CUDA blocks) which are all guaranteed to execute on the same SM. The maximum number of threads per block is 1024 which guarantees that the thread block can fit on a single SM. If the block sizes permits, then it possible to have several active blocks on the same SM at the same time. The thread block size is usually chosen as a multiple of the warp size. Since all threads in a block execute on the same SM they can use the L1 cache of that SM to share data between them. The memory shared between the threads is in CUDA terminology called *shared memory* and must be specified either at compile time or as a parameter when launching the CUDA kernel. Thread blocks also supports synchronization of all threads within the same block by calling the intrinsic function `__syncthreads()`. When a thread reaches a `__syncthreads()` call it will wait for all other threads in the thread block to also reach it before it proceeds execution.

Thread blocks can be defined having either one, two, or three dimensions and the thread block size is specified as launch parameter to the CUDA kernel. This parameter can either have type `int` or `dim3` which is a built-in CUDA type that represents a struct consisting of 3 unsigned integer members named `x,y,z`. Furthermore, thread blocks are structured in a *grid* which is also represented by either an `int` or `dim3` variable in CUDA. When CUDA kernels, are launched from a host the thread block and and grid size are specified as launch parameters when the kernel is called using a CUDA specific syntax which consists of three angle brackets <<<>>>. An example of this is shown below in Listing 1.

```
1
2  __global__ void cuda_kernel() {
3  ...
4  }
5
6  int main() {
7  ...
8      // Launch parameters
9      dim3 threads_per_block(16,16);
10     dim3 blocks(20, 20, 20);
```

```
    // kernel launch from host
    cuda_kernel<<<blocks, threads_per_block>>>();
...
}
```

Listing 1: Example code calling a CUDA kernel from host.

The `__global__` specifier used in the definition of the kernel is a CUDA-specific keyword which is used to denote that the kernel is possible to call from a host. We will go more into detail about this in the next subsection.

2.6.3 CUDA Kernels and Memory Transfers Between Device and Host

In CUDA we specify that a kernel is written as device code and is to be executed on the GPU by using the either of the keywords `__device__` or `__global__` when declaring the function. The `__global__` keyword indicates that the kernel is executed on a CUDA device but called from a host (i.e. launched with thread block and grid configuration using triple brackets). The `__device__` keyword is used for kernels which are called from other CUDA kernels. When writing kernel code, the thread block and grid parameters that the kernel is launched with has to be taken into account. There are three variables of type `dim3` which are available in all kernels to simplify the programming:

`threadIdx` holds the index of the current thread inside the current thread block.

`blockIdx` holds the index of the current thread block inside the grid.

`blockDim` holds the thread block dimensions (this is equivalent to the second launch parameter of the kernel).

Listing 2 below show example kernel code for adding two vectors (stored in float arrays) on the device using a one-dimensional grid.

```
__device__ add(float x, float y, int size) {
    return x + y;
}

__global__ vector_add(float* out, float* x, float* y, int size) {
    // calculate thread index of current thread
    int tid = blockIdx.x*blockDim.x + threadIdx.x;

    // if the thread index < size do vector addition
    if (tid < size) {
        out[tid] = add(x[tid], y[tid]);
    }
}
```

Listing 2: Example of vector addition kernel in CUDA.

The CUDA programming model does not allow the device to access memory on the host directly. The programmer is responsible for setting up the memory transfers between the

device host.[12] Device memory can be allocated using the CUDA function `cudaMalloc`, and then memory can be transferred between the host and device using the function `cudaMemcpy`. More details about memory handling in CUDA can be found in [3, 15].

2.6.4 Random Number Generation using CUDA and cuRAND

NVIDIA provides several highly optimized libraries bundled together bundled with the *Cuda-Toolkit*, among them is a library for random number generation, named *cuRAND*. The cuRAND library consists of both a host interface and device interface which provide functions to generate random numbers from some common distributions, including uniform random numbers and normal random numbers. There are five pseudorandom number generators available in cuRAND[13] (version 12.4), which we list below.

`CURAND_RNG_PSEUDO_XORWOW` is an implementation of `XORWOW` from [32] (see section 2.4.3).

`CURAND_RNG_PSEUDO_MRG32K3A` is an implementation of `MRG32k3a` from [21](see section 2.4.2).

`CURAND_RNG_PSEUDO_MT19937` is an implementaion of `MT19937` from [36] (see section 2.4.4).

`CURAND_RNG_PSEUDO_MTGP32` is an implementaion of `MTGP` from [45] (see section 2.4.4).

`CURAND_RNG_PSEUDO_PHILOX4_32_10` is an implementaion of `PHILOX4_32_10` from [46] (see section 2.4.5).

The methods provided by the cuRAND library have multiple overloads based on which RNG the user wants to use. The default pseudorandom number generator in cuRAND is the `CURAND_RNG_PSEUDO_XORWOW` generator. When using the cuRAND device API one must allocate space for and initialize the random number generator states. Each generator type has a corresponding generator state type for storing the state data. The `MRG32k3a`, `XORWOW`, and `PHILOX4_32_10` generators are designed such that each thread can hold its own state. However, the `CURAND_RNG_PSEUDO_MT19937` generator is only possible to use from the host API and the `CURAND_RNG_PSEUDO_MTGP32` generator is designed to be shared by all threads in a block, but the block may not exceed 256 threads. The CUDA states of the generators which can be used on the device are listed in Table 4 below.

The state size is something which the programmer must have in mind when using the cuRAND device API. CUDA kernels are often designed to be launched with a block and grid configuration such that the number of threads are several times higher than what can physically launch concurrently on the device. This, so called, *over-subscription* of compute resources often leads to better hardware utilization. The sizes of the RNG states often make it unpractical to let each thread have its own RNG state, when using many more threads than the hardware supports. There are several solutions to this problem. It is possible to share RNGs between threads or use a queue to reuse states between thread blocks. Another approach is to use thread coarsening and a persistent threads programming style, and we will discuss this more in detail in Chapter 4.

[12] This is not entirely true. It is possible to set up shared memory spaces between the host and device and also to transfer memory between devices using frameworks such as CUDA-Aware MPI. However, none of those methods will be used in this thesis and we will not mention them further.

[13] There are also four quasirandom number generators available in cuRAND.

RNG	cuRAND state	size (bytes)
CURAND_RNG_PSEUDO_XORWOW	curandStateXORWOW_t	48
CURAND_RNG_PSEUDO_MRG32K3A	curandStateMRG32k3a_t	48
CURAND_RNG_PSEUDO_MTGP32	curandStateMtgp32_t	4112
CURAND_RNG_PSEUDO_PHILOX4_32_10	curandStatePhilox4_32_10_t	64

Table 4: State types and state sizes for RNGs available when using cuRAND's device API.

As mentioned previously the RNG states have to be allocated and initialized before they can be used. The set up and usage of the Mersenne Twister CURAND_RNG_PSEUDO_MRG32K3A differ from the other RNGs and we will not describe how to use it. The cuRAND device API accepts a pointer to a generator state, which point to GPU memory. The generator states can be allocated on the device using cudaMalloc, and the size of the states can by queried by the sizeof operator. The generators must be initialized, before use, and this is possible to do on the device through a function named curand_init, which has the following signature:

```
1 __device__ void curand_init(unsigned long long seed, unsigned
      long long subsequence, unsigned long long offset,
      CURAND_STATE* state)
```

where CURAND_STATE should be replaced by the type of the generator state which is used.

2.7 Rejection Sampling on GPUs

The performance characteristics of rejection sampling algorithms differ a lot between CPUs and GPUs. The reason for this is that it suffices that 1 thread in a warp rejects the sample for thread divergence to occur in the warp. Hence, it is not interesting to study required number of iterations to generate a sample from the desired distribution as was done in section 2.5.2. Instead, we ask the question, how many iterations N are required until all threads in a warp has generated a sample. It has been known for a long time that rejection algorithms can scale badly on GPUs for this reason, and that the rejection probability for a warp can be simulated using Markov Chain theory. In a recent paper [43], Ridley and Forget derive an analytical formula for the probability mass function of N given a warp size t and rejection probability ρ:

$$\mathbb{P}(N = k) = (1 - \rho^n)^t - \left(1 - \rho^{n-1}\right)^t$$

This distribution is referred to as the *geometric exponential distribution* in [43]. Figure 7 show the probability mass function of N is for different rejection probabilities. Note that the probability that all threads in a warp would accept their sample in the first iteration is less than 0.2 even for $\rho = 0.05$ and when $\rho = 0.2$ the probability of all threads in the warp accepting the sample is almost 0. Unfortunately, the mean of the geometric exponential distribution has no simple representation or formula. Approximative formulas for the mean

can be found in [43] and they also compute the expected number of iterations, N, numerically. For large rejection probabilities it may beneficial to reduce the number of samples generated per warp, however for small ρ (< 0.2) there is little to none speedup [43, see Figure 8 and Table 1]. Even for values of ρ up to 0.5 it is possible that the overhead of synchronization when generating fewer samples than there are threads in each warp may lead to this approach being suboptimal.

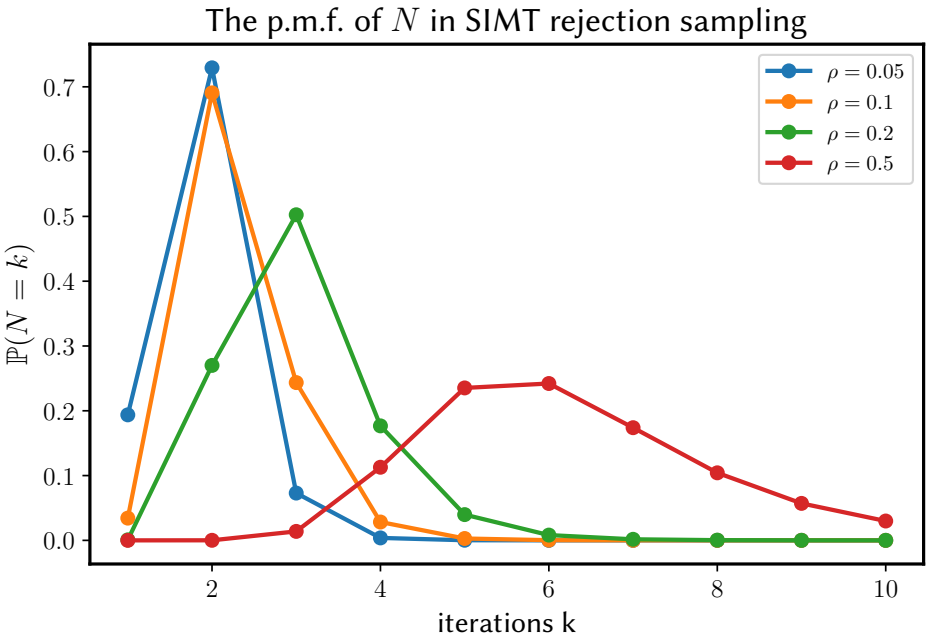

Figure 7: Probability mass function for the number of iterations N required in SIMT rejection sampling to generate one sample for each thread for different rejection probabilities ρ.

3 Gamma Random Number Generation

In this chapter, we present several well-known gamma random number generation algorithms. There are many algorithms for gamma generation published, and we only cover a selection of them. Especially, all algorithms we cover are based on rejection sampling. We focus on algorithms for generating gamma variates for $\alpha > 1$. This is not as limiting as it may seem, because it is always possible to transform a $\Gamma(\alpha + 1, 1)$ random variable to a $\Gamma(\alpha, 1)$ random variable. Thus, with an additional computational step all algorithms presented in this chapter can be modified to generate gamma variates for $\alpha \leq 1$ as well.

3.1 Basics of Gamma Random Generation

As mentioned earlier there are two general approaches generating gamma random variates. The first is by rejection algorithms (see Algorithm 3) and the second is by using the inverse method (see Algorithm 2). All algorithms we cover are based on the rejection method. Since the gamma family of distributions has a scale parameter β it is always possible to generate a random variate $X \sim \Gamma(\alpha, 1)$, and then scale it by the transformation $X \mapsto \beta X$ to get a $\Gamma(\alpha, \beta)$ variate. For this reason, the scale parameter β is usually discarded when looking at gamma generation algorithms and only the parameter α is of interest.

The gamma generation algorithms are broadly divided into two categories, based on whether α exceeds 1 or not. The reason for this is that for $\alpha \leq 1$ the probability density function of the gamma distribution tends to infinity as x tends to 0. On the other hand, when $\alpha > 1$ the density is 0 at $x = 0$ with a peak at some $x > 0$. However, by introducing an independent uniform random variable it is possible to transform a $\Gamma(\alpha + 1, 1)$ random variable to a $\Gamma(\alpha, 1)$ random variable, as is seen in the following lemma ([33, see note on top of page 371]).

Lemma 3.1. *Let $Y \sim \Gamma(\alpha + 1, 1)$ for some $\alpha > 0$ and $U \sim U(0, 1)$ be independent random variables, then*

$$X = YU^{1/\alpha} \sim \Gamma(\alpha, 1).$$

Note that the transformation given in Lemma 3.1 can be used for any $\alpha > 0$, but it is mostly used to convert random variates for α in the range $(1, 2]$ to random variates with α in the range $(0, 1]$. The computations required are also cheap without any branches, which makes it especially suitable for use on GPUs.

There are two special cases for which gamma variables can generated very efficiently. The first concerns the case when $\alpha = n$ for some integer n. Then, by Theorem 2.6 and Theorem 2.7, $\Gamma(n, 1)$ is equivalent to the distribution of a sum of n independent Exp(1) random variables. Exponential random variates can be efficiently generated using the inversion algorithm and equation (7). This results in Algorithm 4, listed below, for generating a $\Gamma(n, 1)$ random variate.

The other case is when $\alpha = n/2$ for some integer n. This corresponds to sampling from the χ^2-distribution with n degrees of freedom. Hence, a $\Gamma(n/2, 1/2)$ variate can be generated from n normal random variates. However, sampling from the exponential is generally more

Algorithm 4 $\Gamma(n, 1)$ random number generation (n integer).

1: $U = 1$
2: **for** $i = 1, \ldots, n$ **do**
3: $\quad V = \mathrm{U}(0, 1)$
4: $\quad U = UV$
5: **end for**
6: **return** $-\log(U)$

efficient than sampling from the normal distribution and $n/2$ is either an integer if n is even or $k + 1/2$, where $k = \lfloor n/2 \rfloor$ if n is odd. By noting that if $N \sim \mathrm{N}(0, 1)$ and $E_i \sim \mathrm{Exp}(1)$, then

$$X = N^2 + \sum_{i=1}^{k} E_i \sim \Gamma(k + 1/2, 1).$$

This identity can be used to generate $\Gamma(k+1/2)$ random variates even more efficiently.

We are mainly interested in algorithms for sampling from the gamma distribution for arbitrary values of α and not only the special cases mentioned above. In the following sections we introduce five known algorithms for gamma random number generation, that we believe can be efficiently implemented on GPUs. What we mean by this is that we have selected algorithms with few branches and only one or two squeeze steps. Many of the algorithms for gamma random number generation which has been published use several different dominating distributions for the different sections of the density. However, for the SIMT architecture this leads to warp divergence and an even higher expected value for N, the number iterations required before accepting a sample.

3.2 Ahrens-Dieter GC

In their paper [1] Ahrens and Dieter present several algorithms for generating gamma random variates. One method, which is often used as a comparison for benchmark in other publications is the GC algorithm, presented below. The GC algorithm is based on rejection sampling from the Cauchy distribution with location $\mu = \alpha - 1$ and scale $\sigma = \sqrt{2\sigma - 1}$. The main idea behind the algorithm is the following mathematical identity (see [1, equation (3.3)])

$$\frac{e^{-x} x^{\alpha-1}}{\Gamma(\alpha)} \leq \frac{1}{\Gamma(\alpha)} \frac{e^{-(\alpha-1)}(\alpha-1)^{\alpha-1}}{1 + \frac{1}{2\alpha-1}\big(x - (\alpha-1)\big)^2}, \quad \alpha > 1, x > 0. \tag{9}$$

The left hand side of equation (9) is the p.d.f. of the $\Gamma(\alpha, 1)$ distribution, and it can be shown the right hand side is is proportional to the p.d.f. of the Cauchy distribution. In order to see this it is easiest to rewrite the p.d.f. of the gamma distribution as

$$g(x) = \frac{\sigma}{\pi\left[(x-\mu)^2 + \sigma^2\right]} = \frac{1}{\pi\sigma\left[1 + \frac{(x-\mu)^2}{\sigma^2}\right]}.$$

Thus, with the location $\mu = \alpha - 1$ and scale $\sigma = \sqrt{2\sigma - 1}$ the density simplifies to

$$g(x) = \frac{1}{\pi\sqrt{2\alpha - 1}\left[1 + \frac{(x-(\alpha-1))^2}{2\alpha-1}\right]}.$$

From equation (9), it follows that densities satisfy

$$f(x) \leq Mg(x), \quad M = \frac{\pi\sqrt{2\alpha-1}(\alpha-1)^{\alpha-1}}{\Gamma(\alpha)},$$

and therefore rejection sampling can be used. The sampling from the Cauchy density is easily done using the rejection method. With the choice of μ and σ introduced above and the identity for the inverse c.d.f. of the Cauchy distribution given in equation (8) we get that

$$x = \alpha - 1 + \sqrt{2\alpha-1}\tan\left[\pi\left(u - \frac{1}{2}\right)\right].$$

Algorithm 5 Ahrens-Dieter (GC) gamma generator [1]

1: $b = \alpha - 1$
2: $A = -\alpha + b$
3: $s = A$
4: **repeat**
5: sample $u \sim \mathrm{U}(0,1)$
6: $t = s \cdot \tan(\pi \cdot (u - 0.5))$
7: $x = b + t$
8: **if** $x < 0$ **then**
9: go to step 2
10: **end if**
11: sample $u^* \sim \mathrm{U}(0,1)$
12: **if** $u^* > \exp\big(b \cdot \log(x/b) - t + \log\big(1 + t^2/A\big)\big)$ **then**
13: accept x
14: **end if**
15: **until** x is accepted
16: **return** x

The GC algorithm follows the general structure of rejection sampling (Algorithm 3). There are two steps of the algorithm which may not be clear from our previous discussion. Firstly, on line 8 and 9 we check if the Cauchy variate generated (x) is less than zero. This is done because the p.d.f. of the gamma distribution is zero for any $x < 0$ which would lead to rejection of the sample. Secondly, the acceptance condition on line 12 has been rewritten using some clever mathematical tricks, see [1] for more details.

We know from section 2.5.2 that the rejection probabilities for Algorithm 5 can be calculated from the expected number of iterations which are given for different values of α in [1]. The rejection probabilities of Algorithm 5, calculated from the values in [1], are presented in Table 5 below.

ρ	$\frac{\pi}{\pi-1}$	0.500	0.47	0.43
α	$1+\varepsilon$	2	3	∞

Table 5: Rejection probabilities (ρ) of the Ahrens-Dieter (GC) gamma generator (Algorithm 5) for different values on α.

3.3 Cheng (GA)

Another simple gamma generator that is based on rejection sampling is the (GA) generator from [7]. The algorithm samples from the Burr XII distribution with parameters μ and λ (see e.g. [9, p. 411]). The probability density of the Burr XII distribution is given by

$$g(x) = \lambda\mu \frac{x^{\lambda-1}}{(\mu + x^{\lambda})^2}, \quad x > 0,$$

and the c.d.f is given by

$$F(x) = \frac{x^{\lambda}}{\mu + x^{\lambda}}.$$

The inverse of the c.d.f can be solved for analytically and it is given by

$$x = F^{-1}(u) = \alpha \exp\left[a \log\left(\frac{u}{1-u}\right)\right].$$

Thus, the inverse method can be used to efficiently generate samples from the Burr XII distribution. In the (GA) algorithm, the choice for the parameters is $\mu = \alpha^{\lambda}$ and $\lambda = \sqrt{2\alpha - 1}$. Then the rejection sampling condition $f(x) \leq Mg(x)$ is satisfied with the constant

$$M = \frac{4\alpha^{\alpha} e^{-\alpha}}{\Gamma(\alpha)\sqrt{2\alpha - 1}}.$$

The full Cheng (GA) algorithm is listed below in Algorithm 6

Algorithm 6 Cheng (GA) gamma generator [7]

1: $a = \sqrt{2\alpha - 1}$, $b = \alpha - \log(4)$, $c = \alpha + \frac{1}{a}$.
2: **repeat**
3: sample $U_1, U_2 \sim \text{Uniform}(0, 1)$.
4: $V = a \log\left(\frac{U_1}{1-U_1}\right)$.
5: $X = \alpha e^{V}$.
6: **if** $b + cV - X \geq \log\left(U_1^2 U_2\right)$ **then**
7: accept X
8: **end if**
9: **until** X is accepted
10: **return** X

There is also a version of Algorithm 6 known as (GB), where an extra squeeze step is added. The version (GB) is commonly used as reference implementation in the literature on gamma generators, but from a theoretical view point the version without a squeeze step, (GA), is more attractive for high performance on SIMT architectures. The rejection probability of the algorithm decreases as α increases. It ranges from $\rho = 0.32$ at $\alpha = 1$ to approximately $\rho = 0.13$ as α tends to infinity, and $\rho < 0.2$ for every $\alpha > 2$.

3.4 Cheng-Feast (GKM3)

There are several non-uniform random number generation methods that can be considered variations of the rejection sampling algorithm. One such method is the the *ratio of uniforms*

(ROU) method, from [17]. Cheng & Feast presents two gamma generators based on this method in [8]. Their proposed algorithms (GKM1) and (GKM2) are efficient for different ranges of α and they recommend combining the algorithms to achieve good performance for all $\alpha > 1$. The resulting algorithm which is referred to as (GKM3) uses (GKM1) to generate the gamma variates if $\alpha < 2.5$ and (GKM2) if $\alpha \geq 2.5$. The (GKM1) algorithm is presented below. The (GKM2) algorithm is similar but also includes a squeeze step and can be found in

Algorithm 7 Cheng-Feast (GKM1) gamma generator [8]

$a = \alpha - 1, b = \frac{\alpha - \frac{1}{6\cdot\alpha}}{a}, c = \frac{2}{a}, d = c + 2$
repeat
  sample $U1, U2 \sim \mathrm{U}(0, 1)$
  $W = b \cdot \frac{U1}{U2}$
  if $(c \cdot \log(U2) - \log(W) + W) < 0$ **then**
    accept W
  end if
until W is accepted
return $a \cdot W$

the original paper [8]. For the theory behind the GKM generators and the ROU method the reader can consult [9], [19], and [17].

3.5 Best (XG)

Another simple gamma generator which is based on rejection sampling was given by Best in [5]. It is based on rejection sampling from Student's t-distribution with two degrees of freedom. A proof correctness of the algorithm can be found in Devroye [9, see Theorem IX.3.3 and surrounding discussion]. The original algorithm contains a squeeze step, which we have removed for increase performance on GPUs. We have listed the algorithm without the squeeze step below in Algorithm 8.

Algorithm 8 Best (XG) gamma generator (without squeeze step) [5]

1: $b = \alpha - 1, c = 3\alpha - \frac{3}{4}$.
2: **repeat**
3: sample $U, V \sim \mathrm{Uniform}(0, 1)$.
4: $W = bU/(1 - U)$.
5: $\beta = (2.71828 + W)/c$.
6: **if** $V \leq 1 - \frac{2}{3}\beta^3$ **or** $\log(V) \leq \frac{1}{2}W + b - b\beta$ **then**
7: accept W
8: **end if**
9: **until** W is accepted
10: **return** W

3.6 Marsaglia-Tsang

In this section we introduce an algorithm for generating gamma random numbers by Marsaglia & Tsang [33]. This generator is commonly used for gamma generation on CPUs and

the code is very simple. One the other hand, the theory behind this generator is more complex than that of the other generators we have presented. The generator is based on the exact-approximation method from [30]. For the theoretical details of this generator we refer the reader to the original paper [33] and to [30] for more details about the exact-approximation method.

The algorithm is a very simple rejection algorithm that require one uniform sample and one sample from the normal distribution in each iteration. In [33] the authors also provide a variant of the algorithm with an extra squeeze step. For usage on a GPU it is generally preferred to avoid introducing squeeze steps, since it is very likely that at least one thread will have to execute the costly branch anyway. Thus the version of the Marsaglia-Tsang generator displayed below in Algorithm 9 is the basic version of the algorithm without the squeeze. The rejection probabilities of the Marsagla-Tsang generator is less than 0.05 for

Algorithm 9 Marsaglia-Tsang gamma generator (without squeeze step) [33]

1: $d = \alpha - \frac{1}{3}$ and $c = \frac{1}{\sqrt{9d}}$.
2: **repeat**
3: sample $Z \sim \text{Normal}(0, 1)$.
4: $V = (1 + cZ)^3$.
5: sample $U \sim \text{Uniform}(0, 1)$.
6: $X = dV$.
7: **if** $V > 0$ **and** $\log(U) < \frac{1}{2}Z^2 + d - dV + d\log(V)$ **then**
8: accept X.
9: **end if**
10: **until** X is accepted
11: **return** X

all $\alpha > 1$ and less than 0.01 for all $\alpha > 4$. This is extremely low compared to most other generators. The exact expression for the rejection probability is given by:

$$\rho = 1 - \frac{\int_{-1/c}^{\infty} e^{g(x)}\,\mathrm{d}x}{\int_{-\infty}^{\infty} e^{-x^2/2}\,\mathrm{d}x}, \quad g(x) = d\log\left[(1 + cx)^3\right],$$

where c and d are the same as on line 1 of Algorithm 9. The low rejection rate, makes the Marsaglia-Tsang an ideal candidate for a gamma generator to be used on the GPU.

4 Methods

In order to investigate whether gamma random number generation on GPUs can be done efficiently we implemented all algorithms from Chapter 3 in CUDA and measured the time to generate gamma variates for varying sample sizes and different shape parameters α. We also measured the time to generate gamma variates on a CPU using a single thread. This was implemented using the the random number generation facilities of the C++ *standard template library (STL)*

As we discussed in Chapter 2, memory transfers between the CPU and the GPU can often be the main bottleneck when using GPUs to accelerate compute work loads. When benchmarking GPU code one must decide whether to include the memory transfer times between the CPU and the GPU. We chose to not include the memory transfer times in our benchmark results and there are several reasons for this. Firstly, the random number generation is usually only a part of a more complex algorithm which also execute on the GPU. Thus by measuring the time it takes to generate the numbers on the GPU, we measure the performance of the kernel for the most common use case. Secondly, if we were to include the data transfer time, it would be harder to compare performance between different gamma generation kernels on the GPU. If a practitioner would like to generate a very large number of gamma random numbers on the GPU and then transfer the random numbers to the CPU, then this is best implemented using CUDA streams to overlap communication and computation, which is architecturally different from the main use case of the random number generators we have in mind.

There is also a cost associated with initializing the pseudo random numbers generator states on the GPU, however this only has to be done once after the generator states has been allocated on the GPU. In most stochastic simulation and Monte Carlo use cases, the initialization is only done once, at the beginning of the application, and for this reason we have chosen not to include the state initialization measurement in our reported generation times. In the next section we present more details about the CUDA implementation of the gamma generation kernels and benchmarking code.

4.1 CUDA Implementation

We implemented the 5 gamma generation algorithms from Chapter 3 in CUDA as `__device__` kernels that can be called from other CUDA kernels. The kernels were all implemented using C++ templates with two template parameters: `RealType` and `CurandState`. `RealType` is the floating point type to be used (`float` or `double`) and `CudaState` is the the CUDA type for the RNG state. The main advantage of using C++ templates is that the each kernel can be used with any combination of floating-point type and cuRAND state but only has to be implemented once. All the algorithms from Chapter 3 are based on rejection sampling and can be efficiently implemented in CUDA using the `do-while` loop construct and by introducing a boolean variable `accept`. The implementations follows the kernel `gamma_kernel` outlined below in Listing 3.

```
1 template<typename RealType, typename CurandState=curandState>
2 __forceinline__ __device__  RealType gamma_kernel(CurandState &
      RNG_state, RealType alpha) {
3     // setup
```

```
4       const RealType a = ...
5
6       // rejection sampling loop
7       RealType X;
8       ...
9       bool accept;
10      do {
11          // update step
12          ...
13          accept =  // acceptance condition
14      } while (!accept);
15      return X;
16  }
```

Listing 3: `do-while` implementation of rejection sampling

The main generator kernel was written using C++ templates with the device kernel passed as a template parameter and is listed in Listing 4 below. The use of C++ templates allow for flexibility in selecting which kernel to use with each benchmark instance, without the overhead of any runtime dispatch. The implementation uses an `auto` template parameter for the gamma kernel and this is a C++ 17 feature. Earlier C++ versions would require more complex workarounds to pass the CUDA kernel as a template parameter.

```
1  template<typename RealType, auto GammaKernel>
2  __global__ void gamma_pt_strided(RealType *gammas, curandState *
       RNG_states,
3                                   const size_t num_rands, RealType
       alpha = 1.0) {
4      uint tid = blockIdx.x * blockDim.x + threadIdx.x;
5      curandState RNG_state = RNG_states[tid];
6      uint nthreads = blockDim.x*gridDim.x;
7
8      // main loop body
9      uint main_iter_per_thread = num_rands/nthreads;
10     for (uint i=0; i<main_iter_per_thread; i++) {
11         uint idx = nthreads*i+tid;
12         gammas[idx] = GammaKernel(RNG_state, alpha);
13     }
14     // tail loop
15     uint tail_rands = num_rands - main_iter_per_thread*nthreads;
16     if (tid < tail_rands) {
17         gammas[main_iter_per_thread*nthreads+tid] = GammaKernel(
       RNG_state, alpha);
18     }
19     RNG_states[tid] = RNG_state;
20 }
```

Listing 4: Gamma RNG PT implementation

There are several challenges that need to be addressed when implementing Monte Carlo code on GPUs. Storing too many RNG states on the GPU can have a negative performance impact and take up much of the memory on the device. This makes it unpractical to oversubscribe on the compute hardware with grids that are many times the size of physical compute capabilities of the GPU, as is otherwise common in GPU applications. A commonly used

solution to this problem is to use *thread coarsening* (see e.g. [15, section 6.3], where the number of threads is reduced but each thread does more work than before. In the gamma random number generation kernel we use a pattern similar to *persistent threads (PT)* [2, 12]. The persistent threads programming pattern on GPUs tries to launch as many threads as can possibly execute simultaneously on the GPU and distribute the work between threads such that all threads remain active throughout the execution. In the case of random number generation this means that we launch the kernel with a fixed number threads which all have a corresponding RNG state and let each thread generate several variates. This is implemented using a `for` loop and a tail iteration performed by a subset of threads if the number of random variates can't be evenly divided by the number threads (see line 8-18 in Listing 4).

A challenge associated with thread coarsening strategies is that they can lead to inefficient memory access patterns. For best performance adjacent threads in the same warp should read to and write from adjacent memory locations (i.e. operate on the same cache-lines). For this reason we use a strided access pattern inside the loop on line 11 of Listing 4. This is a well-known CUDA optimization pattern and is known as memory coalescing (see e.g. [15, section 6.1]).

A potential negative performance impact of the implementation of the kernel in Listing 4 is that the set up of each kernel is called multiple times, even though it would be sufficient to compute these values once and store them. However, this is not a problem when the kernel is compiled with optimization level 3 (`O3`) and the `const` keyword is used for the constants that doesn't have to be computed every time. Expression templates could be used to ensure that the set up of the kernel is computed only once, however, this would increase the complexity of the code substantially and without any benefit when compiling with high optimization levels. Another, much simpler solution if the shape parameter α is known at compile time is to compute the constants in the kernels at compile using `constexpr` variables that are available in C++. However, this requires the usage of experimental compiler features in CUDA and therefore we chose not to do so.

The implementation of the gamma generators in CUDA was profiled using *NVIDIA Nsight Compute*. The results from the profiling show that the implementations use the hardware resources well. We compared the version of the gamma benchmark kernel that uses strided memory access to a naive implementation not using any memory coalescing technique. The Visual Nsight Compute tool report that the naive version have bad memory access patterns, but this is not the case with the strided implementation. The Nsight Compute tool also report that there is thread divergence in the kernel. This is nothing we can optimize away since the algorithms have a stochastic control flow.

4.2 Verification of Implementations

We used a KS-test to verify that the implemented kernels produce gamma distributed random numbers. For this we chose to use the KS-test method available in SciPy. We wrote the data to binary files from C++ and used the `fromfile` method from NumPy to read the binary data into NumPy arrays. The KS-test is based on a sample output consisting of 10^6 random numbers generated for each generator for three different values on the shape parameter.

4.3 CPU Implementation

The reference implementation on CPU was implemented using `std::gamma::distribution` from the the C++ STL `random` header. The timing measurements were made using the `std::chrono::steady_clock` from the `<chrono>` header of the C++ STL. For the C++ code we decided to the mersenne twister `MT19937` as the PRNG, since it is efficient on CPU and available in the C++ STL. We have not included the time it takes to initalize the RNG in the measurements. The RNG state set up and gamma variate generation steps are shown in Listing 5.

```
1     ...
2     std::random_device random_device;
3     std::mt19937 mt(random_device());
4     std::gamma_distribution<float> gamma_generator(alpha);
5     ...
6     auto t = std::chrono::steady_clock::now();
7         for (auto &e: generated_data) {
8             e = gamma_generator(mt);
9         }
10    auto cpu_time = std::chrono::steady_clock::now() - t;
11    ...
```

Listing 5: C++ reference implementation

4.4 Measurements

We measured the execution times for each gamma kernel when generating samples of single precision floating-point numbers. We made the choice to measure the performance for single-precision floating point numbers because a consumer grade GPU was used for measurements and it has significantly worse double precision floating point performance than single precision performance. In the high-end GPUs geared towards scientific computing the double precision performance is usually much closer to the single precision performance. The measurements were done for varying sample sizes between $2^{22} \approx 4 \times 10^6$ and $2^{28} \approx 268 \times 10^6$. The measurements were done for four different values of $\alpha = 1.0001, 2, 4, 10$. The rejection rates of most gamma generation algorithms, including the ones we have used, are very close to their limit value already at $\alpha = 10$. Hence, there is no need for higher values of α to be included in the comparison. The choice of $\alpha = 1.0001$ is because not all algorithms are valid for $\alpha = 1$, and therefore the value $\alpha = 1.0001$ was used instead, which is common in the literature.

For the GPU kernels, the measurements were recorded using `cudaEvents` to measure the execution time of the kernel. Before each measurement a warm up iteration with the kernel was run and then the kernel was executed 10 more times with each execution time recorded. We did this for each sample size and value on α.

4.5 Experimental Setup

4.5.1 Hardware

All benchmarks were performed on a Linux host running Ubuntu 22.04.3 LTS with Linux kernel version 5.15.0-58-generic. The system has a processor of type AMD Ryzen 9 5950X 16-Core with clock frequency 3.4GHz and memory listed in table 6.

L1d cache	512 KiB
L1i cache	512 KiB
L2 cache	8 MiB
L3 cache	64 MiB
RAM	32GiB (2x16 GiB)
SSD	1TB

Table 6: Memory sizes of the system used and cache sizes for the AMD 5950X CPU.

The GPU used for the measurements was a NVIDIA GeForce RTX 4070 GPU. It has the Ada Lovelace GPU architecture and 5888 CUDA cores. The memory sizes and clock speeds are listed in Table 7 below.

GPU Architecture	Ada Lovelace
CUDA Cores	5888
Base Clock	1.92 GHz
Boost Clock	2.48 GHz
RAM	12 GiB
Memory Interface	192-bit
Memory Bandwidth	504.2 GB/s
L1 Cache Size	192 KiB per SM
L2 Cache Size	36 MiB

Table 7: Summary of memory sizes and clock speeds for the NVIDIA GeForce RTX 4070 GPU used for measurements.

4.5.2 Compiler Environment

All code was compiled using NVCC version 12.2 (V12.2.140) and the profiling tools and CUDA libraries used were all from CUDA-Toolkit version 12.2. We used Python version 3.12 for the verification tests with NumPy version 1.26.4 and SciPy version 1.12.0.

4.5.3 Random Number Generators

The performance of the gamma kernels is affected by the choice of uniform random number generator. We used CUDA's default RNG `CURAND_RNG_PSEUDO_XORWOW` for benchmarking the CUDA kernels. This may seem like a strange choice, since it is known to have worst statistical properties of the RNGs available in cuRAND. However, since many benchmarks use the default `CURAND_RNG_PSEUDO_XORWOW` this makes it easier to compare our results with other published work. The quality of the generator is also good enough that the output of all gamma generators can be verified using a KS-test.

4.5.4 Grid and Block Configuration

When using persistent threads one generally wants to maximize the number of active threads per SM. Instead, we chose a grid based on the maximum number of active thread blocks per SM. CUDA provides the function `cudaOccupancyMaxActiveBlocksPerMultiprocessor` to calculate the maximum number of active thread blocks per SM given the thread block size and the CUDA device used. Below is an excerpt from the `GammaBenchmark` class, that show how the grid parameters are computed from the thread block size.

```
1     cudaOccupancyMaxActiveBlocksPerMultiprocessor(&
      max_active_blocks, gamma_pt_strided<RealType, GammaKernel>,
      thread_block_size, device);
2     block.x = thread_block_size;
3     grid.x = max_active_blocks*deviceProperties.
      multiProcessorCount;
4     // verifies kernel dims are supported by initRNG and device
5     verify_kernel_dims(block, grid, deviceProperties, &initRNG);
```

We used a 1-D grid and block and tested the kernels with the thread block sizes: 32, 64, 128, 256, 512, 1024. We found that 64 was the best block size for the kernels, but the other block sizes smaller than 512 also perform well. We used a block size of 64 for the measurements and this results in a a maximum of 24 active blocks on each of the 46 SMs of the NVIDIA 4070 GPU. This means that grid size is 1104, i.e. 70656 active threads, which is equivalent to 12 times the CUDA cores available on the NVIDIA 4070 device used for measurements. This shows that our implementation utilizes the full hardware resources available on the device.

5 Results & Analysis

In this chapter, we present the results of the performance measurements and the verification of the kernels. In section 5.1, the results of the KS-test used for verification are presented. Section 5.2 compares the times it takes to generate gamma random numbers on the GPU with the different kernels. Then, in section 5.3, we compare the performance of the best and worst gamma generator on the GPU with that of the C++ STL on a CPU (single thread). We end this chapter with a discussion of our findings.

5.1 Verification of Output

The KS-test used to verify the correctness of the implemented algorithms is based on sample outputs of size 10^6 for three different shape parameters for each generator. We have chosen to test the generators with the shape parameters $\alpha = 1.0001$, $\alpha = 2$ and $\alpha = 10$. The reason for this choice is that these three values represent different shapes of the gamma distribution. For $\alpha = 1.0001$ the gamma distribution behave close to an exponential distribution, but for $\alpha = 2$ the distribution resembles that of a Poisson distribution, and when $\alpha = 10$ then gamma distribution is closer to a bell curve in shape. We report the value of the KS-test statistic D_n and the p-value of the test (rounded) in Table 8 below. It should be noted that we want the KS-test statistic to be as close to 0 as possible and the p-value to be high. A low p-value, would mean that we could reject the hypothesis H_0 with high probability and that the output of the generator is not likely gamma distributed. The p-value of the KS-test is high for all of the generators except Cheng-Feast (GKM3).

All generators we study should theoretically produce gamma random numbers. The quality of the random numbers generated depends on the quality of the uniform random number generator used. However, even if the mathematics underlying the generators is correct it is possible that numerical approximations make some of the algorithms less stable than others. Based on our tests the Cheng-Feast (GKM3) algorithm seem to be an outlier with worse statistical quality than the other generators.

Algorithm	$\alpha = 1.0001$		$\alpha = 2.0$		$\alpha = 10.0$	
	D_n	*p-value*	D_n	*p-value*	D_n	*p-value*
Cheng-Feast (GKM3)	0.0012	0.11	0.00094	0.34	0.0015	0.018
Marsaglia-Tsang	0.00059	0.88	0.00069	0.72	0.00072	0.67
Cheng (GA)	0.00067	0.76	0.00052	0.95	0.00074	0.64
Best (XG)	0.00069	0.73	0.00059	0.87	0.00062	0.84
Ahrens-Dieter (GC)	0.00063	0.83	0.00059	0.88	0.00064	0.80

Table 8: KS-test results of the algorithms for selected values of α.

Visualizations of the samples used for generating the KS-test statistics are given in Figures 8-12, which show histograms of the generated samples used for the KS-tests. We also include the graph of p.d.f. of the gamma distribution for the value of α used for comparison. The bin

width used to generate each histograms is set such that 40 uniform bins exactly cover the output sample and thus vary between the plots. It is not possible to tell from Figure 10 that the Cheng-Feast (GKM3) generator performs worse than others on the KS-test.

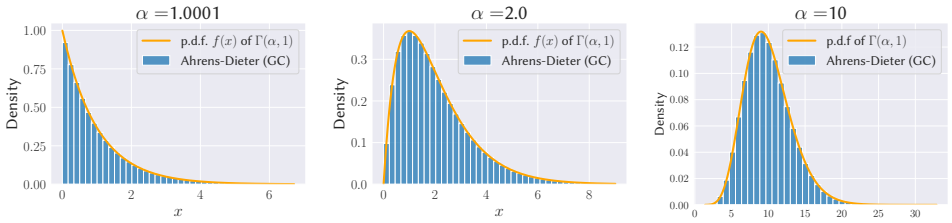

Figure 8: Histograms generated by a sample output of size $n = 10^6$ for the Ahrens-Dieter (GC) generator and the p.d.f of the gamma distribution for three values of α.

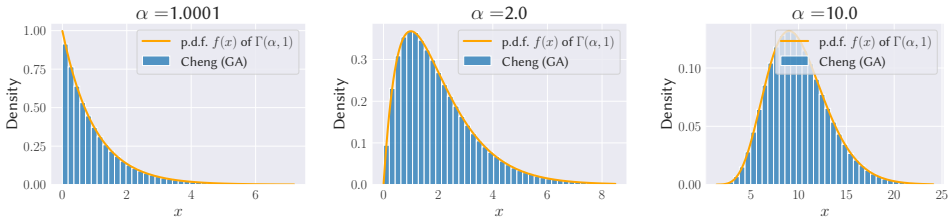

Figure 9: Histograms generated by a sample output of size $n = 10^6$ for the Cheng (GA) generator and the p.d.f of the gamma distribution for three values of α.

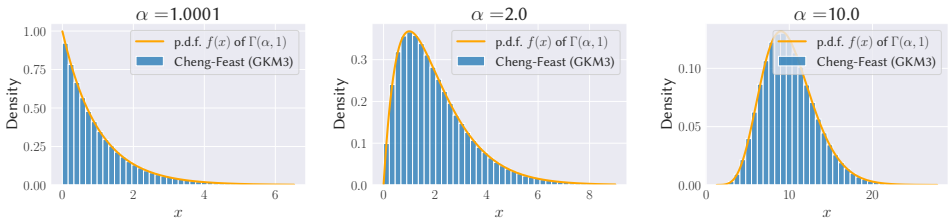

Figure 10: Histograms generated by a sample output of size $n = 10^6$ for the Cheng-Feast (GKM3) generator and the p.d.f of the gamma distribution for three values of α.

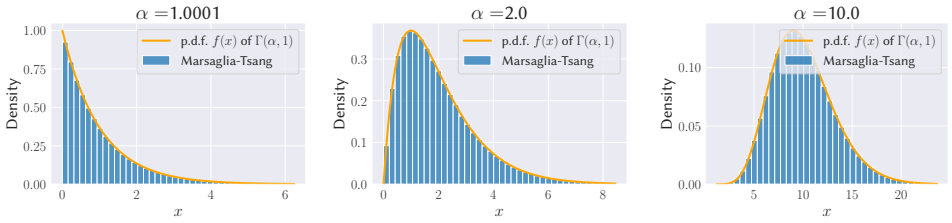

Figure 11: Histograms generated by a sample output of size $n = 10^6$ for the Marsaglia-Tsang generator and the p.d.f of the gamma distribution for three values of α.

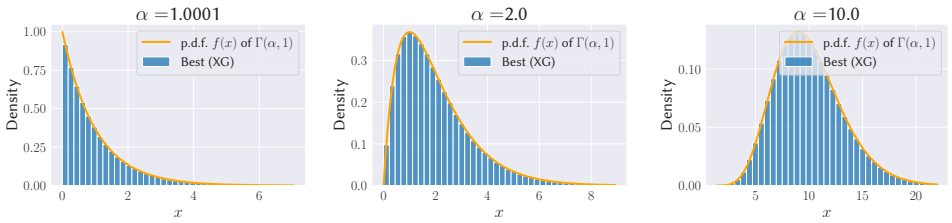

Figure 12: Histograms generated by a sample output of size $n = 10^6$ for the Best (XG) generator and the p.d.f of the gamma distribution for three values of α.

5.2 Comparisons between GPU kernels

In this section, we present and analyze the execution times of the different gamma generation algorithms on a NVIDIA 4070 GPU. For each sample size the time taken to generate the sample was measured a total of 10 times after a warm up iteration. In Figure, 13 the means of our measurements are reported with errorbars corresponding to the sample variance of the measurements. There are four subfigures corresponding to different shape parameters α. In Figure 14, the same data is reported for the best performing gamma kernels for each shape parameter, and we have also included the time it takes to generate normal random numbers using cuRAND's device API with persistent threads. This allows us to compare the time it takes to generate gamma random numbers with the time it takes to generate normal random numbers on the GPU. We shall now analyze the the performance of each kernel more in detail.

5.2.1 Marsaglia-Tsang

The Marsaglia-Tsang generator, is often one the recommended algorithms to use for gamma generation in the literature (see e.g. [19]). It has a very low rejection rate and is among the three best kernels for all values of α we used in our benchmark. It is the second fastest of all kernels tested for $\alpha \geq 2$. Overall the Marsaglia-Tsang generator perform very well and can be used to efficiently generate gamma random numbers on the GPU for all $\alpha > 1$, but it is not the fastest generator. This may be surprising given the low rejection rates of this algorithm, but each rejection sampling iteration of the kernel is expensive, since it requires a normal random variate to be generated.

5.2.2 Ahrens-Dieter (GC)

The Ahrens-Dieter (GC) generator is also among the generators commonly mentioned in the literature for gamma generation (see e.g. [19]). Even though this generator has much higher rejection rates than the Marsaglia-Tsang generator the performance of the two generators is very similar, with Ahrens-Dieter(GC) being the slightly slower of the two generators. The reason for this is likely that the rejection sampling step is much more light-weight than in Marsaglia-Tsang. Still, this generator can be used to efficiently generate gamma random numbers on the GPU for all $\alpha > 1$.

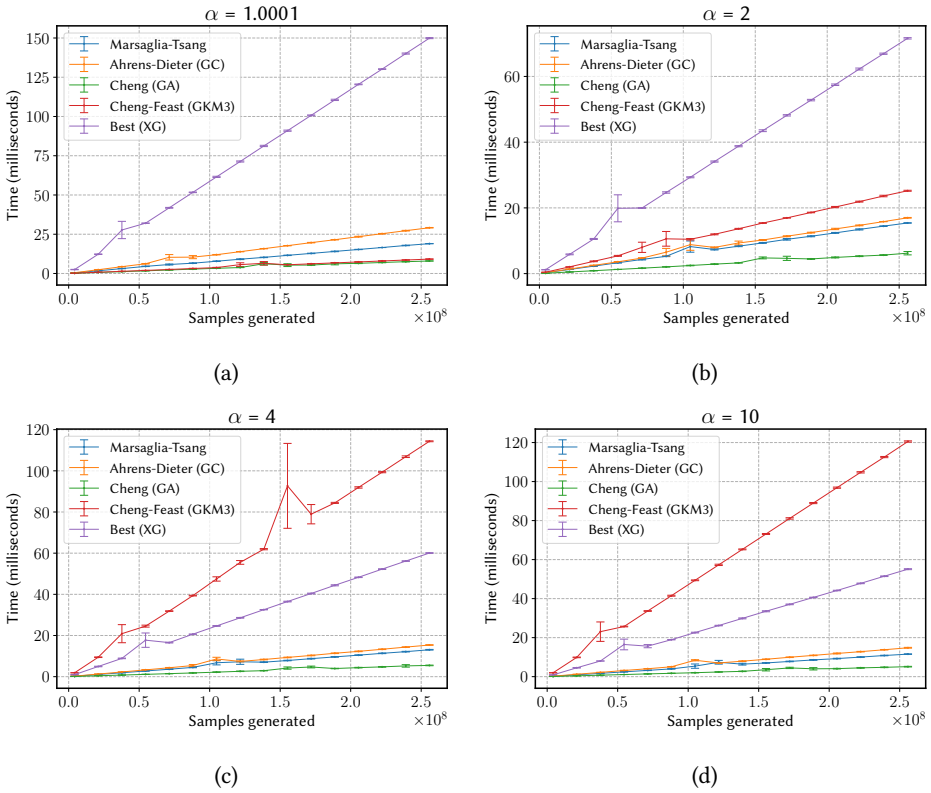

Figure 13: Measured generation times of the gamma kernels for four different shape parameters on a NVIDIA 4070 GPU.

5.2.3 Cheng (GA)

The Cheng (GA) generator is the best performing of all generators we tested, across all values for α. Even if the generator has higher rejection rates that the Marsaglia-Tsang generator the Cheng (GA) generator is very light-weight in terms of computation. It doesn't require any samples from the normal distribution and sampling from the Burr XII distribution is fairly efficient in terms of computation which makes this generator the best of all gamma generators tested across all α. The time it takes to generate gamma random numbers with the Cheng (GA) generator is within $2\times$ the time it takes to generate normal random variates with the cuRAND device API. This shows that the Cheng (GA) algorithm is well suited for generating random numbers on the GPU.

5.2.4 Best (XG)

The Best (XG) algorithm perform poorly compared to the other algorithms across the entire range of α tested. One of the key advantages of Best's algorithm on the CPU, is a rejection step which we have removed in our implementation. Based on our results this generator can not be recommended for use on the GPU.

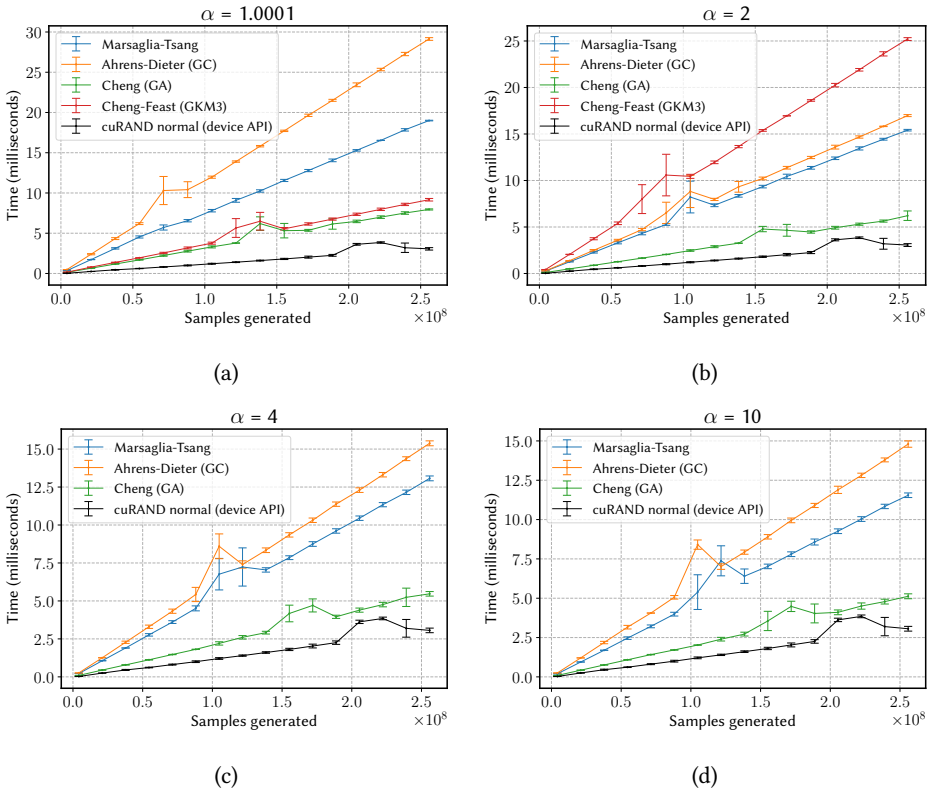

Figure 14: Measured generation times of the best performing gamma kernels for four different shape parameters, and a normal generator using the cuRAND device API as a reference in black, on a NVIDIA 4070 GPU.

5.2.5 Cheng-Feast (GKM3)

The Cheng-Feast (GKM3) generator perform very well for $\alpha = 1.0001$ but the performance quickly deteriorates as α increases and it performs worst of all generators for $\alpha \geq 4$. We believe the reason for this deteriorating performance is is due to an extra branch in the (GKM2) algorithm that is used for $\alpha > 2.5$, which leads to higher thread divergence than the other kernels. However, using the (GKM1) algorithm across the entire range of α is not feasible, because the rejection rate rapidly gets too high for $\alpha > 3$. Thus, based on our results the Cheng-Feast (GKM3) algorithm can not be recommended for use on the GPU.

5.3 Comparisons between GPU and CPU

We also measured the time it takes to generate random numbers on a traditional CPU using the C++ STL. The speedups of the GPU generators compared the C++ version on an AMD Ryzen 5950X CPU is displayed in Figure 15 below. We note that Cheng (GA) is the algorithm displaying the highest speedup. When the shape parameter $\alpha \geq 2$ then the Cheng (GA) GPU kernel is more than 1000× faster than then the reference CPU generator.

It should be noted that these measurements doesn't include the data transfer times between CPU and GPU, but the results clearly show that it can more efficient to perform gamma

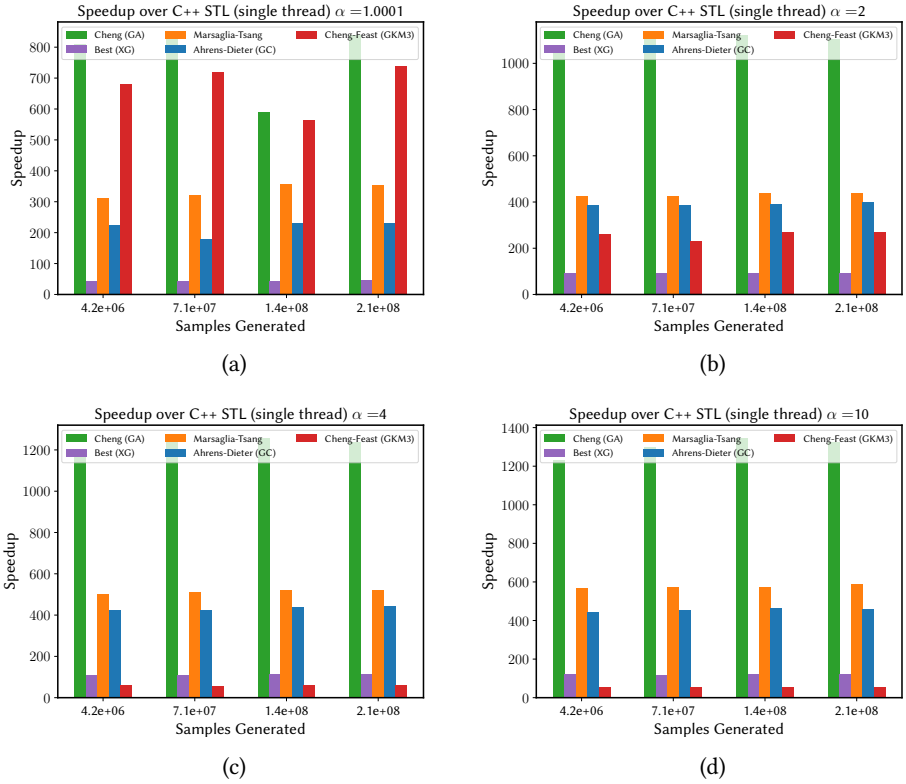

Figure 15: Speedups of the CUDA gamma kernel implementation compared to the CPU reference implementation (C++ STL single thread).

random number generation on GPUs than on CPUs.

5.4 Summary of Results

Three of the gamma generation kernels tested perform well on the GPU with generation times $1 - 10\times$ the time it takes to generate normal random variables using the cuRAND device API and persistent threads. We list them below (fastest to slowest):

1. Cheng (GA)
2. Marsaglia-Tsang
3. Ahrens-Dieter (GC)

Among the three generators listed above, Cheng (GA) is clearly the fastest on GPU. The Marsaglia-Tsang and Ahrens-Dieter (GC) are very similar in performance, but Marsaglia-Tsang is the faster of the two. Using the Cheng (GA) algorithm for generating gamma random numbers gives a $2 - 3\times$ speedup compared to the other two generators, All three generators generate gamma distributed random numbers and have high p-values in the KS-tests performed. Thus, Cheng (GA) should be the preferred generator for anyone who want an efficient and correct algorithm for generating gamma random numbers on GPUs.

6 Conclusions

With the increasing use of GPUs for computation the need for efficient random number generation algorithms on GPUs is as ever important. In this thesis, we have looked at the possibility to efficiently generate gamma random numbers on GPUs. Our results have shown that the best performing algorithm for generating gamma random numbers on GPUs can generate gamma random numbers at more than half the speed normal random numbers can be generated on GPUs. This corresponds to a 1000× speedup when generating gamma random numbers on a NVIDIA 4070 GPU compared to an AMD 5950X CPU (single threaded).

6.1 Conclusions

We have shown that it is possible to efficiently generate gamma random numbers on GPUs. Not only compared to CPUs, but the time it takes to generate gamma random numbers with the best algorithm tested is roughly $1.5 - 2\times$ the time it takes to generate normal random numbers on the GPU. This shows that with the right choice of algorithm gamma generators can be included in Monte Carlo and stochastic simulation code for GPUs without fear of bad performance. Especially, practitioners should not be afraid of using gamma random number generators on the GPU, even if the underlying algorithm is rejection sampling.

Based on our results the preferred gamma generator to use on the GPUs is the Cheng (GA) generator. It performs best of all generators we tested for all shape parameters. The Cheng (GA) algorithm is not often mentioned in the literature, but a variation of this algorithm Cheng (GB), from the same article [7], is more common. The Cheng (GB) algorithm includes an extra squeeze step and performed much worse than (GA) in our initial experiments (due to increased thread divergence). It is interesting to note that the performance of the generators tested on the GPU differ a lot from what has been reported in the literature which is focused on CPUs (see e.g. [29]). This, is not surprising, since GPUs and CPUs are architecturally very different, but it shows that there is a gap in the knowledge of efficient RNG algorithms on GPUs, which the results of this thesis partially fill.

One main take away from our results is that it is generally beneficial to use generators with fewer branches and remove squeeze steps on GPUs. This means that one should trade branches for increased compute loads instead. This should hold true in general for rejection sampling schemes on the GPU, and our findings clearly indicate that simple kernels with less branches outperform the kernels with more branches such as the Best (XG) and Cheng-Feast (GKM3) algorithms which perform much worse than the other algorithms.

6.2 Limitations

In this thesis, we have focused on a selection of the existing algorithms for gamma generation and evaluated their performance on GPUs. We made our selection of algorithm with the architecture and performance characteristics of GPUs in mind, but it is possible that there are other gamma generation algorithms that perform even better on GPUs and was discarded in our selection phase. Furthermore, we focused on generators for shape parameters $\alpha > 1$. As we discussed in Chapter 3, a gamma variate with shape parameter $\alpha = \alpha' + 1 > 0$ can be cheaply transformed into a gamma variate with shape parameter α', and the algorithms evaluated can therefore be used to generate gamma random numbers for shape $\alpha \leq 1$.

However, it is possible that some of the existing algorithms for shape parameters $\alpha \leq 1$ perform better than the algorithms we have tested for α in that range.

We only performed measurements on a NVIDIA 4070 GPU, and there are many other GPU models available, both from NVIDIA and other hardware vendors. One of the main differences between the GPU used and the professional series of GPUs available by several manufacturers is that the professional GPUs tend to have better double precision performance that scale proportionally to the single-precision performance. This is not the case with the NVIDIA 4070 GPU used for measurements and therefore we limited ourselves to single precision floating point numbers in our measurements.

6.3 Future Work

Our work show two promising areas for future research. Firstly, there exists a rich collection of algorithms for generating random numbers for other distributions than the gamma distribution where the performance of the algorithms on GPUs still remain to be tested. We believe that it would be highly valuable for pracitioners to learn which algorithms are best suited for use on the GPU as it often differ from the CPU.

Secondly, traditional CPUs have evolved much during the last 40 years and some of the comparisons between random number generators are based on results which are more than 40 years old. On modern CPUs branches can have a huge negative impact on performance, just like on GPUs and it would be interesting to test whether these results are still valid or if the state of art has changed.

6.4 Reflections

We have shown that using GPUs instead of CPUs can lead to a 1000× increase in performance when generating gamma random numbers. Especially, we have shown that there exist algorithms for gamma random number generation that can be used efficiently on GPUs. Our results can help researchers and GPU programming professionals make better choices when designing and writing code that require gamma generators and help speed up their codes.

There are many examples where our results could be applied to optimize the performance of existing code. In [49], the authors use a generator from [7], but they do not write which one and they cooperate between threads to generate their samples. When using the Cheng (GA) generator that we recommend this approach leads to worse performance in light of the results in [43]. Furthermore, the Marsaglia-Tsang generator is often mentioned for use on the GPUs due the low rejection rates (see e.g. [43]) but based on our results a $2.5 - 3\times$ speedup can be achieved by switching to the Cheng (GA) generator.

In the end, our results can be used optimize current software leading to reduced energy consumption. This is positive from a sustainability perspective and also leads to a reduced cost of running the software. Furthermore, the knowledge that gamma random numbers can be generated efficiently on GPUs open up new possibilities when designing scientific models in a wide range of disciplines where the gamma distribution is used.

References

[1] Joachim H. Ahrens and Ulrich Dieter. "Computer methods for sampling from gamma, beta, poisson and bionomial distributions". In: *Computing* 12.3 (1974), pp. 223–246. DOI: 10.1007/BF02293108. URL: https://doi.org/10.1007/BF02293108.

[2] Timo Aila and Samuli Laine. "Understanding the efficiency of ray traversal on GPUs". In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 145–149. ISBN: 9781605586038. DOI: 10.1145/1572769.1572792. URL: https://doi.org/10.1145/1572769.1572792.

[3] Richard Ansorge. *Programming in parallel with CUDA: a practical guide*. Cambridge University Press, 2022.

[4] Søren Asmussen and Peter W Glynn. *Stochastic simulation: algorithms and analysis*. Vol. 57. Springer, 2007.

[5] D. J. Best. "Letters to the Editors". eng. In: *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 27.2 (1978), pp. 181–182. ISSN: 0035-9254. DOI: 10.1111/j.1467-9876.1978.tb01041.x. URL: https://doi.org/10.1111/j.1467-9876.1978.tb01041.x.

[6] David Blackman and Sebastiano Vigna. "A New Test for Hamming-Weight Dependencies". In: *ACM Trans. Model. Comput. Simul.* 32.3 (2022), 19:1–19:13. DOI: 10.1145/3527582. URL: https://doi.org/10.1145/3527582.

[7] R. C. H. Cheng. "The Generation of Gamma Variables with Non-Integral Shape Parameter". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 26.1 (1977), pp. 71–75. URL: http://www.jstor.org/stable/2346871 (visited on 05/17/2024).

[8] R. C. H. Cheng and G. M. Feast. "Some Simple Gamma Variate Generators". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.3 (1979), pp. 290–295. DOI: 10.2307/2347200. URL: https://doi.org/10.2307/2347200.

[9] Luc Devroye. *Non-uniform random variate generation*. eng. 1st ed. 1986. New York: Springer Science+Business Media, LLC, 1986. ISBN: 1-4613-8643-8.

[10] C Doty-Humphrey. "PractRand official site, 4-2018". In: *URL: http://pracrand.sourceforge.net* (2018).

[11] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Trans. Computers* 21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071. URL: https://doi.org/10.1109/TC.1972.5009071.

[12] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. "A study of Persistent Threads style GPU programming for GPGPU workloads". In: *2012 Innovative Parallel Computing (InPar)*. 2012, pp. 1–14. DOI: 10.1109/InPar.2012.6339596.

[13] Allan. Gut. *An Intermediate Course in Probability*. eng. 2nd ed. 2009. Springer Texts in Statistics. New York, NY: Springer New York, 2009. ISBN: 1-4899-8446-1.

[14] Lee Howes and David Thomas. "Efficient random number generation and application using CUDA". In: *GPU gems* 3 (2007), pp. 805–830. URL: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-37-efficient-random-number-generation-and-application.

[15] Wen-mei W. Hwu, David Kirk, and Izzat El Hajj. *Programming massively parallel processors : A hands-on approach*. eng. Fourth edition. Cambridge, MA: Morgan Kaufmann, 2022. ISBN: 0-323-91231-1.

[16] Jean Jacod and Philip Protter. *Probability essentials*. Springer Science & Business Media, 2004.

[17] Albert J. Kinderman and John F. Monahan. “Computer Generation of Random Variables Using the Ratio of Uniform Deviates”. In: *ACM Trans. Math. Softw.* 3.3 (1977), pp. 257–260. DOI: 10.1145/355744.355750. URL: https://doi.org/10.1145/355744.355750.

[18] Donald Ervin Knuth. *The art of computer programming. Volume 2, Seminumerical algorithms*. eng. 3rd ed.. 1997. ISBN: 0-321-63577-9.

[19] Dirk P Kroese, Thomas Taimre, and Zdravko I Botev. *Handbook of monte carlo methods*. John Wiley & Sons, 2013.

[20] Pierre L’Ecuyer. “Uniform random number generation”. In: *Ann. Oper. Res.* 53.1 (1994), pp. 77–120. DOI: 10.1007/BF02136827. URL: https://doi.org/10.1007/BF02136827.

[21] Pierre L’Ecuyer. “Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators”. In: *Oper. Res.* 47.1 (1999), pp. 159–164. DOI: 10.1287/OPRE.47.1.159. URL: https://doi.org/10.1287/opre.47.1.159.

[22] Pierre L’Ecuyer. “Tables of linear congruential generators of different sizes and good lattice structure”. eng. In: *Mathematics of computation* 68.225 (1999), pp. 249–260. ISSN: 0025-5718.

[23] Pierre L’Ecuyer. “Random number generation with multiple streams for sequential and parallel computing”. In: *Proceedings of the 2015 Winter Simulation Conference, Huntington Beach, CA, USA, December 6-9, 2015*. IEEE/ACM, 2015, pp. 31–44. DOI: 10.1109/WSC.2015.7408151. URL: https://doi.org/10.1109/WSC.2015.7408151.

[24] Pierre L’Ecuyer. “History of uniform random number generation”. In: *2017 Winter Simulation Conference, WSC 2017, Las Vegas, NV, USA, December 3-6, 2017*. IEEE, 2017, pp. 202–230. DOI: 10.1109/WSC.2017.8247790. URL: https://doi.org/10.1109/WSC.2017.8247790.

[25] Pierre L’ecuyer. *Stochastic Simulation and Monte Carlo Methods*. 2024. URL: https://www-labs.iro.umontreal.ca/~lecuyer/ift6561/book.pdf.

[26] Pierre L’ecuyer and Richard Simard. “TestU01: A C library for empirical testing of random number generators”. In: *ACM Transactions on Mathematical Software* 33.4 (2007). ISSN: 00983500. DOI: 10.1145/1268776.1268777.

[27] Ted G. Lewis and William H. Payne. “Generalized Feedback Shift Register Pseudorandom Number Algorithm”. In: *J. ACM* 20.3 (1973), pp. 456–468. DOI: 10.1145/321765.321777. URL: https://doi.org/10.1145/321765.321777.

[28] Rudolf Lidl and Harald Niederreiter. *Finite fields*. 20. Cambridge university press, 1997.

[29] Elena Almaraz Luengo. “Gamma Pseudo Random Number Generators”. In: *ACM Computing Surveys* 55.4 (2022), pp. 1–33.

[30] George Marsaglia. “The Exact-Approximation Method for Generating Random Variables in a Computer”. eng. In: *Journal of the American Statistical Association* 79.385 (1984), pp. 218–221. ISSN: 0162-1459. DOI: 10.2307/2288360. URL: https://doi.org/10.2307/2288360.

[31] George Marsaglia. *Random number generators*. 2003. DOI: 10.22237/jmasm/1051747320.

[32] George Marsaglia. “Xorshift RNGs”. In: *Journal of Statistical Software* 8.14 (2003), pp. 1–6. DOI: 10.18637/jss.v008.i14. URL: https://www.jstatsoft.org/index.php/jss/article/view/v008i14.

[33] George Marsaglia and Wai Wan Tsang. “A simple method for generating gamma variables”. In: *ACM Trans. Math. Softw.* 26.3 (2000), pp. 363–372. DOI: 10.1145/358407.358414. URL: https://doi.org/10.1145/358407.358414.

[34] Makoto Matsumoto and Yoshiharu Kurita. “Twisted GFSR Generators”. In: *ACM Trans. Model. Comput. Simul.* 2.3 (1992), pp. 179–194. DOI: 10.1145/146382.146383. URL: https://doi.org/10.1145/146382.146383.

[35] Makoto Matsumoto and Yoshiharu Kurita. “Twisted GFSR Generators II”. In: *ACM Trans. Model. Comput. Simul.* 4.3 (1994), pp. 254–266. DOI: 10.1145/189443.189445. URL: https://doi.org/10.1145/189443.189445.

[36] Makoto Matsumoto and Takuji Nishimura. “Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator”. In: *ACM Trans. Model. Comput. Simul.* 8.1 (1998), pp. 3–30. DOI: 10.1145/272991.272995. URL: https://doi.org/10.1145/272991.272995.

[37] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. URL: probml.ai.

[38] H. Niederreiter. “The Multiple-Recursive Matrix Method for Pseudorandom Number Generation”. In: *Finite Fields and Their Applications* 1.1 (1995), pp. 3–30. ISSN: 1071-5797. DOI: https://doi.org/10.1006/ffta.1995.1002. URL: https://www.sciencedirect.com/science/article/pii/S1071579785710027.

[39] Harald Niederreiter. “Factorization of polynomials and some linear-algebra problems over finite fields”. In: *Linear Algebra and its Applications* 192 (1993), pp. 301–328. ISSN: 0024-3795. DOI: https://doi.org/10.1016/0024-3795(93)90247-L. URL: https://www.sciencedirect.com/science/article/pii/002437959390247L.

[40] François Panneton and Pierre L’Ecuyer. “On the xorshift random number generators”. In: *ACM Trans. Model. Comput. Simul.* 15.4 (2005), pp. 346–361. DOI: 10.1145/1113316.1113319. URL: https://doi.org/10.1145/1113316.1113319.

[41] François Panneton, Pierre L’Ecuyer, and Makoto Matsumoto. “Improved long-period generators based on linear recurrences modulo 2”. In: *ACM Trans. Math. Softw.* 32.1 (2006), pp. 1–16. DOI: 10.1145/1132973.1132974. URL: https://doi.org/10.1145/1132973.1132974.

[42] Wiebe R. Pestman. *Frontmatter*. Berlin, New York: De Gruyter, 2009, pp. I–VI. ISBN: 9783110208535. DOI: doi:10.1515/9783110208535.fm. URL: https://doi.org/10.1515/9783110208535.fm.

[43] Gavin Ridley and Benoit Forget. “A simple method for rejection sampling efficiency improvement on SIMT architectures”. In: *Stat. Comput.* 31.3 (2021), p. 30. DOI: 10.1007/S11222-021-10003-Z. URL: https://doi.org/10.1007/s11222-021-10003-z.

[44] Christian P Robert and George Casella. *Monte Carlo statistical methods*. Vol. 2. Springer, 1999.

[45] Mutsuo Saito and Makoto Matsumoto. “Variants of Mersenne Twister Suitable for Graphic Processors”. In: *ACM Trans. Math. Softw.* 39.2 (2013), 12:1–12:20. DOI: 10.1145/2427023.2427029. URL: https://doi.org/10.1145/2427023.2427029.

[46] John K. Salmon et al. “Parallel random numbers: as easy as 1, 2, 3”. In: *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*. Ed. by Scott A. Lathrop, Jim Costa, and William Kramer. ACM, 2011, 16:1–16:12. DOI: 10.1145/2063384.2063405. URL: https://doi.org/10.1145/2063384.2063405.

[47] Guy L Steele Jr and Sebastiano Vigna. “Computationally easy, spectrally good multipliers for congruential pseudorandom number generators”. In: *Software: Practice and Experience* 52.2 (2022), pp. 443–458.

[48] Robert C. Tausworthe. “Random Numbers Generated by Linear Recurrence Modulo Two”. In: *Mathematics of Computation* 19.90 (1965), pp. 201–209. ISSN: 00255718, 10886842. URL: http://www.jstor.org/stable/2003345 (visited on 04/29/2024).

[49] Alexander Terenin, Shawfeng Dong, and David Draper. “GPU-accelerated Gibbs sampling: a case study of the Horseshoe Probit model”. In: *Stat. Comput.* 29.2 (2019), pp. 301–310. DOI: 10.1007/S11222-018-9809-3. URL: https://doi.org/10.1007/s11222-018-9809-3.

[50] David Thomas. *Rejection methods on GPUs*. https://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-rejection.html. Accessed: 2024-01-16.

[51] Sebastiano Vigna. "It is high time we let go of the Mersenne Twister". In: *CoRR* abs/1910.06437 (2019). arXiv: 1910.06437. URL: http://arxiv.org/abs/1910.06437.

TRITA – EECS-EX 2024:0000
Stockholm, Sweden 2024

www.kth.se

€€€€ For DIVA €€€€

{
"Author1": { "Last name": "Ericsson",
"First name": "Johan",
"Local User Id": "u1icqs6r",
"E-mail": "joheric@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
}
},
"Cycle": "2",
"Course code": "DA231X",
"Credits": "30.0",
"Degree1": {"Educational program": ""
,"Degree": "Both Degree of Master of Science in Engineering and Master's degree"
,"subjectArea": "Engineering Physics"
},
"Title": {
"Main title": "Gamma Random Variable Generation on GPUs using CUDA",
"Language": "eng" },
"Alternative title": {
"Main title": "Generering av Gammafördelade Slumptal på GPUer genom CUDA",
"Language": "swe"
},
"Supervisor1": { "Last name": "Daga",
"First name": "Mohit",
"Local User Id": "u1tt8moj",
"E-mail": "mdaga@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Computer Science" }
},
"Examiner1": { "Last name": "Kumar",
"First name": "Arvind",
"Local User Id": "u1vukjbl",
"E-mail": "arvkumar@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "Computer Science" }
},
"National Subject Categories": "10105, 10106, 10201, 10206",
"Other information": {"Year": "2024", "Number of pages": "xiv,54"},
"Copyrightleft": "copyright",
"Series": { "Title of series": "TRITA – EECS-EX" , "No. in series": "2024:0000" },
"Opponents": { "Name": "Markus Newton Hedelin"},
"Presentation": { "Date": "2024-06-19 10:00"
,"Language":"eng"
,"Room": "via Zoom https://kth-se.zoom.us/j/68827389570"
,"City": "Stockholm" },
"Number of lang instances": "2",
"Abstract[eng]": €€€€

We study how pseudo random gamma distributed random variables can be efficiently gener- ated on graphical processing units. There are many algorithms known today for generating gamma random variables by means of computation that perform well on central processing units of classical computers. In the last 20 years, there has been increasing interest in using graphical processing units and other accelerators to speed up stochastic simulation and machine learning applications. The difference in architecture between graphical processing units and traditional central processing units means that algorithms that perform well on central processing units do not always perform well on graphical processing units. This is especially true for random number generation algorithms from complex distributions. In this work, we show that graphical processing units can be used to efficiently simulate random numbers from a gamma distribution, and that the best performing algorithms for doing so are different than the best performing algorithms on central processing units.

€€€€,
"Keywords[eng]": €€€€
Gamma distribution, Random variable generation, Non-uniform, RNG, GPU, CUDA €€€€,
"Abstract[swe]": €€€€

Vi studerar hur gammafördelade slumptal effektivt kan genereras på grafikprocessorer. Det finns många algoritmer kända idag för att generera gammafördelade stokastiska variabler och som presterar bra på processorer för klassiska datorer. Under de senaste 20 åren år har intresset ökat för att använda grafikprocessorer och andra acceleratorer för att accelerera stokastisk simulering och maskininlärningsapplikationer. Skillnaden i arkitektur mellan grafikprocessorer och traditionella processorer innebär att algoritmer som presterar väl på traditionella processorer inte alltid presterar väl på grafiska processorer. Detta gäller särskilt för algoritmer för att generera stokastiska variabler från komplexa distributioner. I detta arbete visar vi att grafikprocessorer kan användas för att effektivt simulera slumptal från en gammafördelning, och att de bäst presterande algoritmerna för att göra det skiljer sig från de bäst presterande algoritmerna på traditionella processorer.

€€€€,
"Keywords[swe]": €€€€
Gammafördelningen, Slumptalsgenerering, Icke uniform, RNG, GPU, CUDA €€€€,
}