Using GPU Simulation to Accurately Fit to the Power-Law Distribution

Efstratios Rappos and Stephan Robert

Institut des technologies de l’information et de la communication
Haute Ecole d’Ingénierie et de Gestion du Canton de Vaud (HEIG-VD)
Haute Ecole Spécialisée de Suisse occidentale

Institute for Information and Communication Technologies
School of Business and Engineering Vaud (HEIG-VD)
University of Applied Sciences of Western Switzerland

Abstract This article describes a methodology for fitting experimental data to the discrete power-law distribution and provides the results of a detailed simulation exercise used to calculate accurate cutoff values used to assess the fit to a power-law distribution when using the maximum likelihood estimation for the exponent of the distribution. Using massively parallel programming computing, we were able to accelerate by a factor of 60 the computational time required for these calculations across a range of parameters and construct a series of detailed tables containing the test values to be used in a Kolmogorov-Smirnov goodness-of-fit test, allowing for an accurate assessment of the power-law fit from empirical data.

Keywords: Zipf distribution; power law; Kolmogorov-Smirnov test; parallel computing simulation; graphics processing unit programming

PACS: 02.50.Ng–Distribution theory and Monte Carlo studies; 05.10.Ln–Monte Carlo methods; 89.75.–k–Complex systems

MSC 2010: 62F03, 68W10, 62P10, 62P30, 62P35, 62Q05

May 2013

1 Introduction

Power-law distributions and their extensions characterize many physical, biological and social phenomena [2, 8, 12, 7, 9, 11] but the process of accurately fitting a power-law distribution to empirical data is not straightforward, and in some cases very imprecise methods are known to be used, namely ‘estimating’ the power-law exponent and fit via linear regression on a log-log plot [2].

A popular method to fit a power-law is by calculating the maximum likelihood estimator (MLE) for the distribution exponent and then using the Kolmogorov-Smirnov (KS) test to assess the goodness-of-fit by comparing against simulation-derived cutoff values. The practicalities of this approach are described in [2] and [3].

To produce these cutoff values, a large number of statistical simulations needs to be run. However, generic tables cannot always be used accurately, as the cutoff values depend on the sample size and the estimated value of the exponent of the data.

Producing such tables for the power-law is computationally challenging. The most complete set of tables to date was produced by [8]; however, presumably due to limitations of the computer technology of the time, aggregate values were obtained across a range of values for the estimated exponent. We extend this work by providing the calculated cutoff tables for a variety of sample sizes and values for the exponent, a task that would require over 2.5 years of computational time on a typical PC. We also describe the methodology and provide computer code which enables researchers to calculate the corresponding tables for values of the exponent other than the ones we considered.

Recent technological developments in the field of Graphics Processing Units (GPU), have resulted in consumer-level graphical cards being able to assist with computationally intensive tasks, because their massively parallel design can outperform traditional CPU algorithms. The use of graphics cards to improve the computational power for simulation methods has been studied in many areas such as Monte Carlo techniques [6] and Bayesian estimation [10].

We demonstrate the use of GPU algorithms for the estimation of the KS cutoff values for assessing the goodness-of-fit of power-law data. The use of parallel methods allows much larger simulations to be produced
in a shorter time, producing more accurate results and higher precision.

Furthermore, we consider the case of the truncated power-law distribution where there is an upper limit to the distribution values. This variation allows for cases where the exponent $\gamma < 1$ to be fitted, as is the case in some phenomena such as the world-wide-web [1].

We consider two versions of the discrete power-low distribution, known as the Zipf distribution, described by:

$$\displaystyle p(k) = \frac{k^{-\gamma}}{\zeta(\gamma)} \quad (1)$$

where

- $k$ is a positive integer $1, 2, 3, \ldots$;
- $p(k)$ is the probability of observing the value $k$;
- $\gamma > 1$ is the power-law exponent;
- $\zeta(\gamma)$ is an appropriate scaling factor.

In the traditional version of the power-law the value of the integer $k$ is unbounded ($k \geq 1$) and in that case the scaling factor is the Riemann zeta function $\zeta(\gamma) = \sum_{k=1}^{\infty} k^{-\gamma}$ and for convergence we must have $\gamma > 1$.

If we assume that the range of values for $k$ is finite i.e., $k = 1, 2, \ldots, K$, then in this truncated Zipf distribution the scaling factor is $\zeta(\gamma) = \sum_{k=1}^{K} k^{-\gamma}$ and we only require the exponent to be $\gamma > 0$ for convergence.

2 Estimating the power-law exponent from the data

The maximum likelihood estimator for the power-law parameter is described in [3] and applies to both variations of the Zipf distribution. If the observed dataset consists of $N$ observations $x_1, x_2, \ldots, x_N$, the best estimate for $\gamma$ is the value that satisfies the equation

$$\frac{\zeta'(\gamma)}{\zeta(\gamma)} = -\frac{1}{N} \sum_{i=1}^{N} \log(x_i) \quad (2)$$

where $\zeta(\gamma)$ is either the scaling factor described in the previous section. The above differential equation can easily be solved for $\gamma$ using the standard Newton-Raphson method.

3 A KS goodness-of-fit test for power-law distributions

The Kolmogorov-Smirnov test is a traditional statistical test for goodness-of-fit, relying on calculating the statistic

$$\displaystyle K = \sup_{x} |F^*(x) - S(x)| \quad (3)$$

where $F^*$ is the hypothesized cumulative distribution function and $S$ is the empirical cumulative distribution based on the sample data, which is then compared with specific cutoff values. There are alternative approaches, such as the general Khmaladze transformation [4] [5], but are outside of the scope of this article. The standard tables of cutoff values for the KS test cannot be directly used when the model parameters (the $\gamma$ in our case) have been estimated from the data, and bespoke tables have to be created using Monte-Carlo simulation. Moreover, the tables to be used also depend on the estimated value of $\gamma$ and the sample size.

Cutoff values provided in [3] were obtained by simulating 10,000 Zipf distributions with a random exponent $\gamma = 1.5$ to 4.0, for 14 logarithmically-spaced choices of the sample size. Whilst this method produces reasonable results, we cannot ignore the fact that the KS cutoff values depend on the calculated value of $\gamma$ and therefore average values do not work well for cases where the power-law fit is marginal.

We extend the results by providing the corresponding test values, simulating 50,000 Zipf distributions for 15 similar choices of sample size, and in each case for 12 possible values of $\gamma$. In addition, we consider the case of the truncated distribution where observations are bounded at $K = 20, 50, 100, 500$ and 1000. We repeat each experiment 10 times for each case and tabulate the average value obtained in each case. In total, this results to a total of over 10,000 separate simulations compared to the 14 used in the above-mentioned research, each one containing five times the number of points.

4 A CUDA algorithm for the calculation of the KS test values

To achieve this level of experimentation, the simulations were performed in a parallel computing environment consisting of two GTX590 graphics processing units (GPU) on a PC using the CUDA/C programming language. This approach carries out the calculations in a high-end computer graphics card rather than in the CPU and the inherent parallel architecture of the GPU makes it well suited for simulation experimentation, allowing for a 60 times faster program execution speed compared to CPU calculations. Indeed, we were able to produce these simulation results in just over 373 hours of computational time; using traditional CPU programming this would have taken 2.5 years.

The algorithm, available as a supplementary material to this article, separates the simulations into 782 blocks of 64 simulations (threads) each. The last 48 simulations are discarded to give the required 50,000 simulations. The program is repeated for the different values of $N$, $K$ and $\gamma$. Care is taken in the code to ensure an efficient execution, for example, the natural logarithms of the first $K$ integers are pre-computed and stored in an array: this speeds up considerably the calculation since the terms $k^{-\gamma}$, which
appear in $\zeta(\gamma)$ and its derivatives, can be calculated as $e^{-\gamma \ln k}$. Care should also be taken, as explained in the attached code, to adjust a compiler parameter when running the code in order to ensure all calculations are carried out in double-precision rather than single-precision by default and avoid numerical underflow in the calculations.

Table 1 presents the test values to use for the pure Zipf distribution (which corresponds to a truncated Zipf distribution with $K = \infty$) for various choices of the estimated value of the exponent $\gamma$. Tables 2 to 6 present the corresponding tables for the truncated power-law distribution with $K = 20, 50, 100, 500$ and 1000 respectively.

These refinements extend the accuracy of the implementation. We note the variation in the cutoff values of Table 1 depending on the exponent $\gamma$: for example, the 90% cutoff value for a sample size of 1,000 ranges from 0.0056 when $\gamma = 4$ to 0.0569 when $\gamma = 1.25$, a difference of a factor of 10. In contrast, the corresponding figure in Table 2, calculated for an ‘average’ exponent is reported to be 0.0186. This demonstrates the importance of using cutoff tables that are particular not only to the specific sample size but also the value of the exponent $\gamma$.

In practice, the value of $\gamma$ calculated from the data will probably not be an exact match with any of the tabulated values. Ideally, to achieve the best level of accuracy, a meticulous researcher would have to create a bespoke table containing the cutoff values that correspond to the exact value of $\gamma$ as calculated from the sample. Nevertheless, our tables provide a useful approximation for cases where this level of precision is not required, and a simple gauge of how good the power-law fit is required. In any case, marginal cases aside, using these tables with a close approximate value for $\gamma$ can be a lot more precise than log-log plots or the Pearson’s test.

Finally, it is worth noting that the tables presented apply only when the exponent $\gamma$ has been calculated using the MLE method described in Section 2 and would not be relevant if a different method was used instead.

The way to use these tables in practice is described in Section 2 and 3. Assuming one has a set of discrete observations and wishes to test if they follow the Zipf distribution, they would first calculate the maximum likelihood estimator for the exponent $\gamma$ using (2). Then, they would calculate the test statistic (3) by determining the maximum deviation of the empirical cumulative distribution function against the theoretical Zipf one.

This test statistic will then be compared with the cutoff value in the tables that corresponds to the values of $N, K$ and estimated $\gamma$ of the observed dataset. If the test value is less than the tabulated value, there is insufficient evidence to reject the hypothesis that the data follow a Zipf distribution, at the required level of significance. As mentioned earlier, for maximum accuracy a bespoke cutoff value would ideally need to be calculated matching exactly the values of $N, K, \gamma$ of the sample. This can be achieved using the accompanying code.

5 Conclusions

We presented the results of a detailed simulation to calculate the cutoff values of the Kolmogorov-Smirnov test when used to assess the fit of empirical data to the discrete Zipf or power-law distribution. We carry out a much larger set of simulations that the state-of-the-art and further extend previous research by breaking down the cutoff tables according to the estimated value of the Zipf exponent and further consider two versions of the Zipf distribution.

This level of complexity was only possible using Graphical Processing Unit (GPU) algorithms to massively parallelize the simulations. In doing so, we produced a 60-fold faster simulation algorithm compared with traditional programming techniques, which demonstrates the huge potential value of GPU techniques in improving the performance of statistical simulations and other complex algorithms. The provided computer code is also of benefit to any researcher who needs, for more accuracy, to create their own Kolmogorov-Smirnov cutoff value which is specific to the sample size and estimated exponent of their datasets.

6 Supplementary Materials

CUDA/C code: The annex contains the CUDA program that can be used to replicate the results presented in this article. The instructions for compilation and use are included in the code.

References

[1] L Breslau, P Cao, L Fan, G Phillips, and S Shenker. Web caching and Zipf-like distributions: Evidence and implications. In Proceedings of the IEEE INFOCOM 99 Conference on Computer Communications: the Future is Now, volume 1–3, pages 126–134, 1999.
[2] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. SIAM Review, 51(4):661–703, December 2009.
[3] M. L. Goldstein, S. A. Morris, and G. G. Yen. Problems with fitting to the power-law distribution. European Physical Journal B, 41:255–258, 2004.
[4] E.V. Khmaladze. Martingale approach to the theory of goodness of fit tests. Theory of Probability and its Applications, 26(2):240–257, 1981.
[5] E.V. Khmaladze. Goodness of fit problem and scanning innovation martingales. The Annals of Statistics, 21(2):798–829, 1993.
[6] Anthony Lee, Christopher Yau, Michael B. Giles, Arnaud Doucet, and Christopher C. Holmes. On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods.
Journal of Computational and Graphical Statistics, 19(4):769–789, 2010.

[7] Joanne Lee, Wendy K. Tam Cho, and George G. Judge. Stigler’s approach to recovering the distribution of first significant digits in natural data sets. Statistics & Probability Letters, 80(2):82–88, January 2010.

[8] M. E. J. Newman. Power laws, Pareto distributions and Zipf’s law. Contemporary Physics, 46(5):323–351, 2005.

[9] Andrew I. Su and John B. Hogenesch. Power-law-like distributions in biomedical publications and research funding. Genome Biology, 8:404, April 2007.

[10] Marc A. Suchard, Quanli Wang, Ciburn Chan, Jacob Frelinger, Andrew Cron, and Mike West. Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures. Journal of Computational and Graphical Statistics, 19(2):419–438, 2010.

[11] Hsiaw-Chan Yeh. Six multivariate Zipf distributions and their related properties. Statistics & Probability Letters, 56(2):131–141, January 2002.

[12] Peter Zörnig and Gabriel Altmann. Unified representation of Zipf distributions. Computational Statistics and Data Analysis, 19:461–473, 1995.
| N  | γ = 0.25 | γ = 0.5 | γ = 0.75 | γ = 1.0 |
|----|---------|---------|----------|--------|
| 10 | 0.001   | 0.001   | 0.001    | 0.001  |
| 20 | 0.002   | 0.002   | 0.002    | 0.002  |
| 50 | 0.005   | 0.005   | 0.005    | 0.005  |
| 100| 0.01    | 0.01    | 0.01     | 0.01   |
| 500| 0.05    | 0.05    | 0.05     | 0.05   |

Table 2: KS test statistic for the truncated power-law distribution with $K = 20$. The table provides quantiles for different values of $γ$. The entries represent the quantiles at various significance levels (0.9, 0.95, 0.99, 0.999) for different sample sizes (N). The table helps in assessing the goodness-of-fit of the truncated power-law distribution to data sets with varying tail weights ($γ$).
Table 3: KS test statistic for the truncated power-law distribution with $K = 50$.

| $N$   | $K = 50$ | $γ = 0.25$ Quantiles | $γ = 0.5$ Quantiles | $γ = 0.75$ Quantiles | $γ = 1.0$ Quantiles |
|-------|---------|----------------------|---------------------|----------------------|---------------------|
|       | 0.9     | 0.95                 | 0.99                | 0.9                  | 0.95                |
| 10    | 0.2655  | 0.2660               | 0.2662              | 0.2662               | 0.2662              |
| 20    | 0.1917  | 0.1955               | 0.1974              | 0.1995               | 0.2015              |
| 30    | 0.1567  | 0.1703               | 0.1833              | 0.1995               | 0.2139              |
| 40    | 0.1357  | 0.1498               | 0.1679              | 0.1904               | 0.2149              |
| 50    | 0.1208  | 0.1357               | 0.1506              | 0.1769               | 0.2021              |
| 100   | 0.0857  | 0.0948               | 0.1135              | 0.1367               | 0.1672              |
| 500   | 0.0532  | 0.0624               | 0.0776              | 0.0985               | 0.1249              |
| 1000  | 0.0310  | 0.0360               | 0.0471              | 0.0640               | 0.0890              |
| 2000  | 0.0192  | 0.0212               | 0.0254              | 0.0303               | 0.0390              |
| 3000  | 0.0156  | 0.0173               | 0.0207              | 0.0249               | 0.0309              |

... (remaining rows...)

| 5000  | 0.0038  | 0.0042               | 0.0051              | 0.0061               | 0.0078              |
| 10000 | 0.0004  | 0.0005               | 0.0006              | 0.0006               | 0.0006              |

... (remaining columns...)

[Table continued...]

| $N$   | $K = 50$ | $γ = 0.25$ Quantiles | $γ = 0.5$ Quantiles | $γ = 0.75$ Quantiles | $γ = 1.0$ Quantiles |
|-------|---------|----------------------|---------------------|----------------------|---------------------|
|       | 0.9     | 0.95                 | 0.99                | 0.9                  | 0.95                |
| 20    | 0.1946  | 0.2323               | 0.2775              | 0.3335               | 0.3335              |
| 30    | 0.1219  | 0.1487               | 0.1833              | 0.2139               | 0.2139              |
| 40    | 0.1058  | 0.1165               | 0.1388              | 0.1672               | 0.1672              |
| 50    | 0.0943  | 0.1041               | 0.1241              | 0.1467               | 0.1467              |
| 100   | 0.0666  | 0.0737               | 0.0879              | 0.1061               | 0.1061              |
| 500   | 0.0298  | 0.0329               | 0.0393              | 0.0475               | 0.0475              |
| 1000  | 0.0112  | 0.0123               | 0.0127              | 0.0127               | 0.0127              |
| 2000  | 0.0047  | 0.0047               | 0.0047              | 0.0047               | 0.0047              |
| 3000  | 0.0015  | 0.0015               | 0.0015              | 0.0015               | 0.0015              |
| 5000  | 0.0007  | 0.0008               | 0.0008              | 0.0008               | 0.0008              |
| 10000 | 0.0003  | 0.0003               | 0.0003              | 0.0003               | 0.0003              |

... (remaining rows...)

| 50000 | 0.0000  | 0.0000               | 0.0000              | 0.0000               | 0.0000              |
| 100000| 0.0000  | 0.0000               | 0.0000              | 0.0000               | 0.0000              |

... (remaining columns...)

[Table continued...]
| $\gamma$ | 0.9 | 0.95 | 0.99 | 0.999 | 0.9 | 0.95 | 0.99 | 0.999 | 0.9 | 0.95 | 0.99 | 0.999 |
|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 10    | .973 | .872 | .724 | .516 | .973 | .872 | .724 | .516 | .973 | .872 | .724 | .516 |
| 20    | .0078 | .0045 | .0023 | .0009 | .0078 | .0045 | .0023 | .0009 | .0078 | .0045 | .0023 | .0009 |
| 30    | .0031 | .0018 | .0010 | .0005 | .0031 | .0018 | .0010 | .0005 | .0031 | .0018 | .0010 | .0005 |
| 50    | .0016 | .0009 | .0005 | .0003 | .0016 | .0009 | .0005 | .0003 | .0016 | .0009 | .0005 | .0003 |
| 100   | .0008 | .0004 | .0002 | .0001 | .0008 | .0004 | .0002 | .0001 | .0008 | .0004 | .0002 | .0001 |
| 200   | .0004 | .0002 | .0001 | .0001 | .0004 | .0002 | .0001 | .0001 | .0004 | .0002 | .0001 | .0001 |
| 300   | .0002 | .0001 | .0001 | .0001 | .0002 | .0001 | .0001 | .0001 | .0002 | .0001 | .0001 | .0001 |
| 500   | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 |
| 1000  | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 | .0001 |

Table 4: KS test statistic for the truncated power-law distribution with $\gamma = 1$.
| N | γ = 0.9 | γ = 0.95 | γ = 0.99 | γ = 0.999 |
|---|---|---|---|---|
| 50 | 0.00 | 0.00 | 0.00 | 0.00 |
| 100 | 0.00 | 0.00 | 0.00 | 0.00 |
| 200 | 0.00 | 0.00 | 0.00 | 0.00 |
| 500 | 0.00 | 0.00 | 0.00 | 0.00 |
| 1000 | 0.00 | 0.00 | 0.00 | 0.00 |
| 10000 | 0.00 | 0.00 | 0.00 | 0.00 |

Table 5: KS test statistic for the truncated power-law distribution with $K = 500$.
Table 6: KS test statistic for the truncated power-law distribution with $K = 1000$.

| $N$ | $K = 1000$: $\gamma = 0.25$, Quantiles | $\gamma = 0.5$, Quantiles | $\gamma = 0.75$, Quantiles | $\gamma = 1.0$, Quantiles |
|-----|-----------------------------------------|---------------------------|----------------------------|---------------------------|
| 10  | 0.0043 0.0047 0.0056 0.0067           | 0.0042 0.0046 0.0054 0.0065 | 0.0039 0.0042 0.0050 0.0059 | 0.0035 0.0038 0.0044 0.0052 |
| 20  | 0.0088 0.0097 0.0106 0.0115           | 0.0086 0.0095 0.0102 0.0112 | 0.0086 0.0095 0.0102 0.0112 | 0.0086 0.0095 0.0102 0.0112 |
| 50  | 0.0122 0.0134 0.0147 0.0158           | 0.0122 0.0134 0.0147 0.0158 | 0.0122 0.0134 0.0147 0.0158 | 0.0122 0.0134 0.0147 0.0158 |
| 100 | 0.0165 0.0182 0.0207 0.0232           | 0.0165 0.0182 0.0207 0.0232 | 0.0165 0.0182 0.0207 0.0232 | 0.0165 0.0182 0.0207 0.0232 |
| 250 | 0.0209 0.0236 0.0266 0.0301           | 0.0209 0.0236 0.0266 0.0301 | 0.0209 0.0236 0.0266 0.0301 | 0.0209 0.0236 0.0266 0.0301 |
| 500 | 0.0253 0.0286 0.0325 0.0371           | 0.0253 0.0286 0.0325 0.0371 | 0.0253 0.0286 0.0325 0.0371 | 0.0253 0.0286 0.0325 0.0371 |

| $N$ | $K = 1000$: $\gamma = 2.5$, Quantiles | $\gamma = 3.0$, Quantiles | $\gamma = 3.5$, Quantiles | $\gamma = 4.0$, Quantiles |
|-----|-----------------------------------------|---------------------------|----------------------------|---------------------------|
| 10  | 0.0043 0.0047 0.0056 0.0067           | 0.0042 0.0046 0.0054 0.0065 | 0.0039 0.0042 0.0050 0.0059 | 0.0035 0.0038 0.0044 0.0052 |
| 20  | 0.0088 0.0097 0.0106 0.0115           | 0.0086 0.0095 0.0102 0.0112 | 0.0086 0.0095 0.0102 0.0112 | 0.0086 0.0095 0.0102 0.0112 |
| 50  | 0.0122 0.0134 0.0147 0.0158           | 0.0122 0.0134 0.0147 0.0158 | 0.0122 0.0134 0.0147 0.0158 | 0.0122 0.0134 0.0147 0.0158 |
| 100 | 0.0165 0.0182 0.0207 0.0232           | 0.0165 0.0182 0.0207 0.0232 | 0.0165 0.0182 0.0207 0.0232 | 0.0165 0.0182 0.0207 0.0232 |
| 250 | 0.0209 0.0236 0.0266 0.0301           | 0.0209 0.0236 0.0266 0.0301 | 0.0209 0.0236 0.0266 0.0301 | 0.0209 0.0236 0.0266 0.0301 |
| 500 | 0.0253 0.0286 0.0325 0.0371           | 0.0253 0.0286 0.0325 0.0371 | 0.0253 0.0286 0.0325 0.0371 | 0.0253 0.0286 0.0325 0.0371 |
Listing 1: CUDA/C code

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda_runtime_api.h>
#include <curand.h>
#include <curand_kernel.h>

/*
 CUDA C program used for the results of the article

DISCRETE TRUNCATED ZIPF DISTRIBUTION:

Calculates the quantiles for a given value of K, gamma, and random seed.

Syntax is:
program.exe K Gamma Random Seed

The value of N is fixed in the code.

− K must be less than 32766 (in the paper, it’s 20, 30, 50, 100, 500, 1000).
− The value of N is fixed at the start of the code below.
− In the paper, it’s 10, 20, 30, 40, 50, 100, 500, 1000, 2000, 3000, 4000, 5000, 10000, 20000.
− Gamma should be > 0.25 (for meaningful results)

Technical note: Important when compiling this CUDA program:
− The program requires a GPU that supports CUDA, and the (freely downloadable) CUDA developer software installed
  (for Visual Studio it is available as an add-on)
− The program requires a CUDA GPU that supports ‘double’ floating-point numbers. Some GPU only support ‘float’ — this is not good enough
  and will produce incorrect results (zeros, infinities) due to the accumulation of rounding errors
− By default, CUDA may demote ‘double’ to ‘float’ to conserve resources. If a compilation warning:
  ‘double is not supported, demoting to float’ is produced, the following compilation parameters need to be adjusted:
  code generation = compute_20, sm_20
  compiler options = --arch=sm_20
  (20 refers to the cuda computational ability level of the card; level 13 or more supports ‘double’)
− The standard CUDA library curand.lib must be included (used for random number generation)

Dr. Efstratios Rappos and Prof. Stephan Robert
HEIG–VD
Switzerland
Efstratios Rappos at heig–vd <dot> ch
Stephan.Robert at heig–vd <dot> ch
May 2013
*/

// The number of points in each simulation (sample size N)
#define N 2000

#define CUDA_GPU_DEVICE 2 // If you have multiple NVIDIA cards, specify which to use. Start with 0 = "first card", 1 = "second card" etc.
#define BLOCKS 782
#define THREADS_PER_BLOCK 64
#define SIMULATIONS BLOCKS * THREADS_PER_BLOCK //number of simulations (a multiple of NTHREADS)
#define SIMULATIONS_REQUIRED 50000

/*
 SIM = 1 * 64 = 64
 SIM = 2 * 64 = 128
 SIM = 4 * 64 = 256
 SIM = 5 * 64 = 320
 SIM = 7 * 64 = 448
 SIM = 8 * 64 = 512
 SIM = 16 * 64 = 1024
 SIM = 32 * 64 = 2048
 SIM = 63 * 64 = 4032
 SIM = 79 * 64 = 5056
 SIM = 157 * 64 = 10048
*/
```
SIM = 313 * 64 = 20032
SIM = 782 * 64 = 50048 ∗

87 // Nothing really to change from here on.
88 // #define MAX_POINTS 32767
89 #define MAX_POINTS 32767
90 int sort_dbl(const void *x, const void *y) {
91 double t = (+(double)x) - (+(double)y);
92 return (int) (t > 0) - (t < 0);
93 }
94
95 //device double NewtonRaphson(double initial_guess, double RHS_data, int K, const double *LOGS);
96 device double NewtonRaphson(double initial_guess, double RHS_data, int K, const double *LOGS);
97
98 //host void check_cuda(cudaError_t cudaStatus, char *message, bool &fail)
99 host void check_cuda(cudaError_t cudaStatus, char *message, bool &fail)
100 {
101 if (cudaStatus) {
102 printf("Error in %s (%d) \n", message, cudaStatus, cudaGetErrorString(cudaStatus));
103 fail = true;
104 system("pause");
105 }
106 }
107
108 //device void setup_kernel (int seed, curandState *state)
109 device void setup_kernel (int seed, curandState *state)
110 {
111 int id = threadIdx.x + blockIdx.x * THREADS_PER_BLOCK;
112 unsigned long long seed1 = seed;
113 // Each thread gets the same seed, but a different sequence number, no offset
114 curand_init (seed1, id, 0, &state[id]);
115 }
116
117 //device void generate_kernel (curandState *state, double *dev_results, const int K, const double gamma, const double *LOGS)
118 device void generate_kernel (curandState *state, double *dev_results, const int K, const double gamma, const double *LOGS)
119 {
120 int id = threadIdx.x + blockIdx.x * THREADS_PER_BLOCK;
121 curandState localState = state[id]; // Copy state to local memory for efficiency
122 unsigned short points[N];
123 int i, t;
124 double x, c;
125 for (i = 0; i < N; i++)
126 points[i] = 0;
127 int KMAX = 0;
128 c = 0.0;
129 for (i = 1; i < K; i++)
130 c = c + exp(-gamma * LOGS[i]); // c = c + (1.0 / pow(double i, (double) gamma));
131 c = 1.0 / c;
132 for (t = 0; t < N; t++) {
133 x = curand_uniform_double (&localState);
134 double sum_prob = 0;
135 for (i = 1; i < MAX_POINTS; )
136 if (sum_prob >= x) {
137 points[i]++; // Should never happen
138 if (i > KMAX) KMAX = i;
139 break;
140 }  // We store the value of KMAX, the max observation in the current generated series.
141 // As all observations are <=K anyway (an input parameter), we will have KMAX <= K,
142 // However, when using loops 1 to K, we can loop up to KMAX only, rather than K, as there are no observations in the range KMAX to K (more efficient).
143 // Copy state back to global memory
144 state[id] = localState;
145 // We now have points[]
146
147 // FIRST FIND Maximum Likelihood Estimator
148 // RHS
149 int NPOINTS = 0;
150 double RHS_data = 0.0;
151 for (t = 0; t < N; t++) {
152 // check: should never happen
153 // copy points to RHS
154 RHS_data += points[t];
155 NPOINTS++;
156 }
if (points[t] < 1 || points[t] > 32766) { printf(\"n\n ERROR points[%d] < 1 (=%d) \n\n\n", t, points[t]); dev_results[id] = -1.; return; }

RHSdata += LOGS[points[t]] ;
NPOINTS++;

// If all points are = 1, adjust RHS so that it's >0. (for RHS=0 the estimated gamma is infinity).
// adjust by a factor of ln(2) -- as if one point was 2 instead of 1. This given a max gamma of ~15 for 50,000 total points
// This only affects very small values of N, eg 10, 20, 30, 40.
if (RHSdata <= 0) { printf("RHS is <=0 ! (%f) NPOINTS=",RHSdata);
for(t=0;t<N;t++) printf("%d,",points[t]); printf(".
") } } if(RHSdata < 0) { printf("RHS is <=0 ! (%f)",RHSdata);
RHSdata += LOGS[2];
RHSdata = RHSdata / (double) NPOINTS;
// Newton–Raphson to obtain estimated value for gamma
double estimated\gamma = NewtonRaphson( 0.5 , RHSdata , K, LOGS); } //must be K, not KMAX
//Kolmogorov–Smirnoff test statistic
double KStest = KolmogorovSmirnoff(points, K, estimated\gamma, LOGS);
dev_results[id] = KStest;
}

int main(int arg, char* argv[]){
if(arg != 4){
 printf("syntax is program.exe K GAMMA SEED\nbye\n"); system("pause"); return 1; }
const int K = atoi(argv[1]);
if(K==0){
 printf("Cannot read value for K\nbye\n");
 system("pause"); return 1; }
const double gamma = atof(argv[2]);
if(gamma<=0.1){
 printf("Cannot read value for Gamma, or Gamma<0.1\nbye\n");
 system("pause"); return 1; }
const int seed = atoi(argv[3]);
if(seed<1){
 printf("Cannot read value for SEED, or SEED<1\nbye\n");
 system("pause"); return 1; }
if(K>32766){
 printf("Value for K must be < 32766\nbye\n"); } //must be < 32,767 as with the fast implementation, the CUDA sample points are coded 'short'

system("pause"); return 1; }
if(SIMULATIONS_REQUIRED > SIMULATIONS){
 printf("SIMULATIONS_REQUIRED must be <= SIMULATIONS\nbye\n");
 system("pause");
 return 1; }
cudaError cudaStatus ;
int i;
// Pre—compute Logarithms for 1 -- K, K < 32,767, for faster execution
double + LOGS;
LOGS = new double[K+2];
for(i=0;i<=(K+1);i++)
LOGS[i] = log((double) i);
cudaGetDeviceCount(&i);
printf("Found %d Graphics cards that support CUDA\n",i);
printf("Checking capabilities of chosen GPU device (CUDA_GPU_DEVICE = %s)\n",CUDA_GPU_DEVICE);

cudaDeviceProp properties;
cudaGetDeviceProperties(&properties, CUDA_GPU_DEVICE);
printf("Name: %s\n", properties.name);
printf("Total global Memory: %d\n", (int)properties.totalGlobalMem);
printf("Shared Memory per block: %d\n", (int)properties.sharedMemPerBlock);
printf("Total Const Memory: %d\n", (int)properties.totalConstMem);
printf("Multiprocessors: %d\n", (int)properties.multiProcessorCount);
printf("Max # threads per multiprocessor: %d\n", (int)properties.maxThreadsPerMultiProcessor);
printf("Max # threads per block: %d\n", (int)properties.maxThreadsPerBlock);
printf("Compute capability: %d.%d\n", properties.major, properties.minor);
printf("Kernel timeout enabled: %d\n", properties.kernelExecTimeoutEnabled);

bool fail = false;
cudaStatus = cudaSetDevice(CUDA_GPU_DEVICE); check cuda(cudaStatus, "setdevice", fail);
time_t tl = clock();
// Copy logarithms to CUDA device
double *devLogs = 0;
cudaStatus = cudaMalloc((void**)&devLogs, (K+2) * sizeof(double));
cudaStatus = cudaMemcpy(devLogs, LOGS, (K+2) * sizeof(double), cudaMemcpyHostToDevice);
cudaStatus = cudaGetLastError(); check cuda(cudaStatus, "lastError ii", fail);

delete[] LOGS;

// Simulation Setup
double KStest_sim[SIMULATIONS]; // stores the K-Smirnov statistic
for(i=0;i<SIMULATIONS;i++) KStest_sim[i] = -1.0;
printf("Generating %d power-law distributions, with N=%d, K=%d, gamma=%f\n", SIMULATIONS, N, K, gamma );
#endif
// generate @SIMULATIONS random seed values using the CUDA random generator
curandState = devStates ;
cudaMalloc((void**)&devStates, SIMULATIONS * sizeof(curandState));

setupKernel <<< BLOCKS, THREADS_PER_BLOCK >>>( seed, devStates );
cudaStatus = cudaDeviceSynchronize(); check cuda(cudaStatus, "cudaMemcpy 2", fail);
cudaStatus = cudaGetLastError(); check cuda(cudaStatus, "lastError i", fail);
double *devResults;
cudaMalloc((void**)&devResults, SIMULATIONS * sizeof(double));
cudaMemcpy(devResults, 0, SIMULATIONS * sizeof(double));
cudaMemset(devResults, 0, SIMULATIONS * sizeof(double));
cudaStatus = cudaMemcpy(devResults, 0, SIMULATIONS * sizeof(double));
cudaStatus = cudaGetLastError(); check cuda(cudaStatus, "cudaMemcpy 5", fail);
cudaStatus = cudaDeviceSynchronize();
cudaStatus = cudaGetLastError(); check cuda(cudaStatus, "lastError ii", fail);
cudaStatus = cudaGetLastError();
cudaStatus = cudaGetLastError(); check cuda(cudaStatus, "lastError iv 2", fail);
cudaMemcpy(KStest_sim, devResults, SIMULATIONS * sizeof(double), cudaMemcpyDeviceToHost);
cudaFree(cudaStatus, "cudaMemcpy 7", fail);
cudaFree(devLogs);
cudaFree(devResults);
cudaFree(devStates);

debug ("error catching – should never happen")
#endif
// As # simulations is a multiple of 64, we must discard some simulations to have the required number
double KStest[SIMULATIONS_REQUIRED];
for(i=0;i<SIMULATIONS_REQUIRED;i++)
KStest2[i] = KStest_sim[i];
qsort(KStest2, SIMULATIONS_REQUIRED, sizeof(double), sortdbl);
printf("Quantile 90 %%% is at %6.4f \n", KStest2[SIMULATIONS_REQUIRED+9/10]);
printf("Quantile 95 %%% is at %6.4f \n", KStest2[SIMULATIONS_REQUIRED+95/100]);
printf("Quantile 99.00% is at %6.4f n", KStest2[ SIMULATIONS_REQUIRED+99/100 ]);  
printf("Quantile 99.99% is at %6.4f n", KStest2[ SIMULATIONS_REQUIRED+999/1000 ]);  

time_t t2 = clock();  
double duration = (double)(t2−t1) / CLOCKS_PER_SEC;  

// if output to a text file is desired  
FILE *fout;  
fout = fopen("output.txt", "a+");  

fprintf(fout, "%d & %d & %6.2f & %6.4f & %6.4f & %6.4f & %6.4f & %10.4f \\
",K, N, gamma,  
KStest2[ SIMULATIONS_REQUIRED∗9/10 ],  
KStest2[ SIMULATIONS_REQUIRED∗95/100 ],  
KStest2[ SIMULATIONS_REQUIRED∗99/100 ],  
KStest2[ SIMULATIONS_REQUIRED+999/1000 ],  
duration);  

close(fout);  
printf("Time taken: %10.4f seconds n", duration);  
return 0;  

// Newton−Raphson algorithm: produces the estimate the power−law exponent gamma from the data  

\_device\_double NewtonRaphson\(\) \{  
const double absolute\_tolerance = 0.00001; // the required level of accuracy in the estimation of gamma  

int t;  
double x, xnew;  
x = xnew = initial\_guess; // initial guesses for gamma  

double A, B, C;  

do{  
  x = xnew;  
  double f, f1;  
  f = 0.0;  
  A=0.0; B=0.0; C=0.0;  
  for(t=1;t<K;t++)  
    double powt = exp(−x∗LOGS[t]);  
    A += (−powt + LOGS[t] );  
    B += powt; // C  
    C += powt + LOGS[t]∗LOGS[t] ;  
  }  

  //f(x)  
  f = A/B + RHS\_data ;  

  //f'(x) − the derivative  
  f1 = C/B − A+B∗A/B;  
  xnew = x − f / f1;  

  while(\( abs(x − xnew ) > absolute\_tolerance )\);  

  return xnew;  
}  

// Kolmogorov−Smirnov test: returns the test value of the test  
\_device\_double KolmogorovSmirnoff\(\) \{  

double c = 0.0;  

int i;  

int t;  
double xnew = gamma;  

do{  
  c += exp(−xnew∗LOGS[t]);  
  c = 1.0 / c;  

  double actual\_prev, theoretical\_prev;
int actual;
double KTest = -2.0;

int NPOINTS = N;

for(t=1;t<=K;t++) { //K here is the max observation
    if(t==1){
        theoretical_prev = c * exp(-xnew + LOGS[t]) ;
        actual = 0;
        for(i=0;i<NPOINTS;i++){
            if(data[i] == t)
                actual++;
        }
        actual_prev = (double) actual / (double) NPOINTS;
    }
    else{
        theoretical_prev += c * exp( -xnew + LOGS[t] ) ;
        actual = 0;
        for(i=0;i<NPOINTS;i++){
            if(data[i] == t)
                actual++;
        }
        actual_prev += (double) actual / (double) NPOINTS;
    }

    // Find SUP
    if(abs(theoretical_prev - actual_prev) > KTest )
        KTest = abs(theoretical_prev - actual_prev);
}
return KTest;