Generating Multi-million Data Set using GPGPU Accelerated Models

Ghanshyam Verma
Student, DCEA, NITTTR
NITTTR-Bhopal
India, 462002

Priyanka Tripathi, PhD
Assoc. Professor, NITTTTR
NITTTR-Bhopal
India, 462002

ABSTRACT
Generating synthetic data set which is realistic as well as sufficiently large has been a cumbersome task for researchers in the past. Several models have been proposed previously, all adopting heterogeneous approaches, in this work the emphasis is on speeding up the compute time of the data set distribution. Here, Uniform, Poisson and Zipf distributions have been studied and approaches with parallel computation model have been proposed. The models have been verified for speedup using CUDA based implementation on NVIDIA Quadro 2000 GPU. A speed up in the range of 2x to 6x was observed for various range of data sets.

General Terms
High Performance Computing, General Purpose Graphic Processing Unit, GPU computation, Synthetic dataset.

Keywords
Data set generation, synthetic dataset, Zipf, Poisson, Uniform distribution, GPU, CUDA.

1. INTRODUCTION
Synthetic data though may not sound fancy but is the necessity of the hour. The catch being most of these synthetic data sets need to have certain statistical properties. And there could be several reasons associated with real time data prohibiting its free share or exchange like security concerns or huge size of data. Table 1 demonstrates data sizes of some the popular data sets. As evident from the table, is highly impossible to share or exchange data of such volume. Also, as these data sets are in institute's private domain it's very unlikely to be shared with external research organizations. This makes it more convenient to generate data locally than to exchange it from other sources or institutes. Many research areas namely social studies, public health, epidemiology and cyber security based experiments requires study of large population in nature like scenarios as well as under controlled scenarios. For example, in epidemiology, synthetic datasets are needed for the analysis of disease spread pattern and prevention steps required to be taken, which assists in better implementation of the remedies. In social science also, such datasets may be used to better evaluate which and what policy may affect individual preferences and choices. And, in public health, synthetic populations may be required to capture procedural properties in patient data records without affecting individual confidentialities. [1] Additionally, there are many theoretical researches which may require synthetic data sets to test applicability of proposed models or algorithms. In this paper authors have used certain statistical properties, commonly observed in real world data and proposed a parallel data generation model using these properties. This model ensures the data though generated synthetically has similar properties as real world data and the generation time is relatively low compared to other similar approaches. Another point worth mentioning is most of the techniques proposed earlier, discussed in section 2 of this paper, focused on populating tables in existing relational databases, which though a very efficient method, but many a times the proposed model's efficiency cannot be measure by this method, as most of the relational database management systems are themselves able to handle and transform datasets being populated into tables, thus brining bias in efficiency measurement. Here in this work, the authors have extracted the efficiency to GPU coprocessors to massively parallelize the dataset generation. The work is solely restricted to dataset generation and the model could be easily extended to populate existing databases.

Table 1. Data set size of popular databases

Organization/Project	Table Column Head
Ebay.com	7.5 PB, 40 PB
Amazon.com	7.8 TB, 18.5 TB, 24.5 TB
Walmart	2.5 PB
Large Hedron Collider	200 PB from 0.001% sensor stream
Sloan Digital Sky Survey	140 TB
NASA Center for Climate Simulation	32 PB

PB in above table refers to Petabytes and TB refers to Terabytes. The data in table was accumulated through resources [2], [3], [4], [5] and [6]. Section 2 contains a brief survey of existing models; later sections contain the underlying framework of libraries used and proposed models.

2. PREVIOUS MODELS
As it has been discussed in section 1, synthetic data set generation has been contributing significantly to different research domains. Here, we have enlisted some of the models used for synthetic data set generation.

Hao Wu et. al. [1] proposed the use of maximum entropy principle to form data generation model for statistically correct and unbiased data. The model was validated by the authors against simulated data as well as against US census data. The work's feasibility was further validated using an epidemic simulation application.

Joseph E. Hoag et. al. [7] designed a parallel synthetic data generator to generate industrial sized data sets quickly using cluster computing. Their model depends on SDDL, a synthetic data description language, quite similar to XML, thus making it very flexible in terms of type of data set being generated.

Jim Gray et. al. [8] presented several database generation techniques based upon various statistical distributions. In particular all these techniques discussed parallelism to get generation speedup and scale up. The discussion was focused on generating datasets of the order of billions and used C programs which ran on a shared-nothing computer system consisting of multiple processors and storage discs.

Nathaniel Boggs et. al. [9] presented a framework for generating synthetic datasets with normal and attack data for web applications across multiple layers simultaneously.

As observed in all above cases the focus was to parallelize the data set generation massively. In our work also, we have used the parallel technique to scale up and speed up the generation. However, in place of using several computing nodes or cluster, we have used general purpose graphic processing unit (GPGPU) as computing co-processor with the help of Compute Unified Device Architecture (CUDA). Next section discusses the basic underlying processing flow of CUDA to better understand the parallelism being offered by it.

3. CUDA – PARALLEIZATION AND COMPUTE FLOW

CUDA otherwise known as Compute Unified Device Architecture is a GPU centric library developed by NVIDIA which supports parallel computation. [10] It helps programmers exploit GPUs which support CUDA libraries for executing general purpose computation tasks. CUDA is released in version numbers each version number suggesting certain compute capability supported by that version. For example, CUDA compute capability 2.x supports double precision floating point operations and 3.5 supports dynamic parallelism, which is not supported by 2.x.

Several advantages associated with CUDA are listed below.

a. CUDA supports scattered reads which enables it to access random memory addresses.

b. Unified virtual memory is supported by CUDA 4.0 and unified memory by CUDA 6.0 and above.

c. Provides option for user managed cache, by exposing shared memory region to threads. [11]

d. Efficient read back and downloads between CPU and GPU.

e. Complete support for bitwise and integer functions. Later versions have included support for double precision floating point operations too.

Fig 1 demonstrates basic block architecture of the CUDA. Several components that make CUDA include- CUDA drivers, device level API, CUDA libraries, CUDA runtime and most importantly CUDA parallel compute engines. Apart from these standard CUDA package consist of samples, documentations, NVIDIA C compilers, debuggers and visual profilers. Recently, wrappers for other languages like Java have been developed like JCuda, which makes it easier for the developers to write code in a favored language. Additionally libraries like SWIG has been developed which acts as interface generator for C/C++ with other languages. CUDA provides support for two programming interfaces- device level programming interface which is used to configure and manage GPU and language integration programming interface. [12]

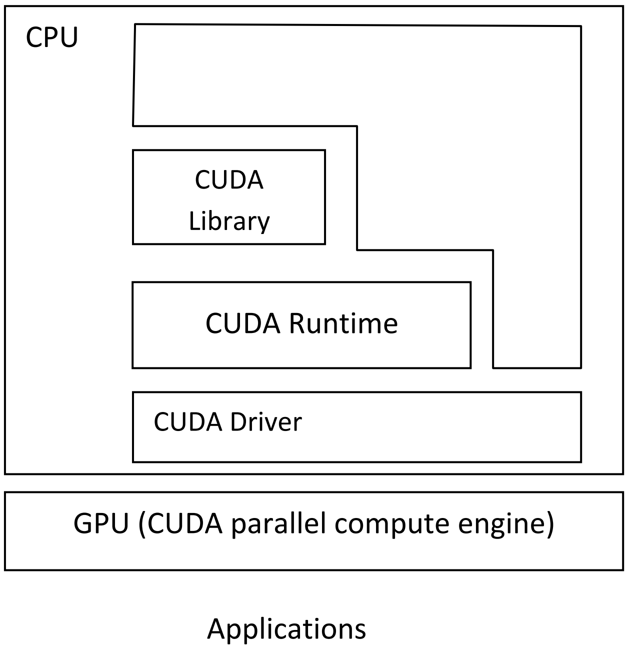

Fig 1: Block architecture of CUDA

Fig 2 represents work flow of CUDA. The steps could be summarized as below.

a. CUDA applications submit code to the NVCC compiler – NVIDIA C compiler.

b. NVCC compilers CPU code onto CPU and loads PTX – parallel code execution- code onto CUDA parallel compute engine.

c. PTX codes are run on GPU cores.

The process could be well depicted using the diagram.

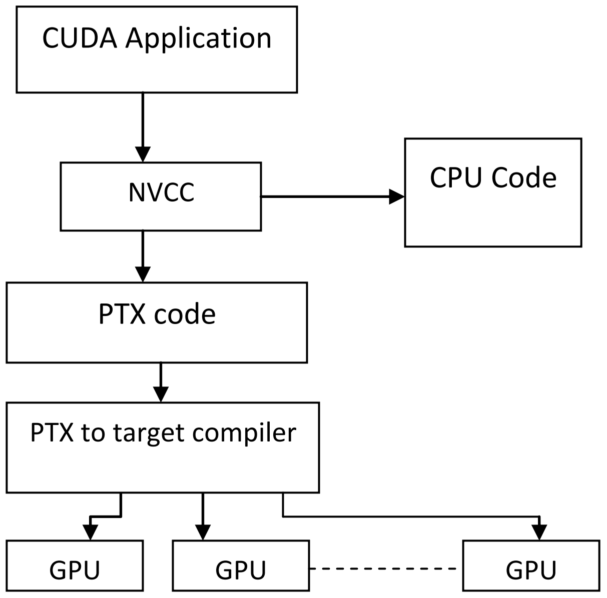

Fig 2: CUDA workflow

Next section discusses some of the statistical models generally observed in the data sets. First we discuss these statistical properties than we propose parallel implementations of the same.

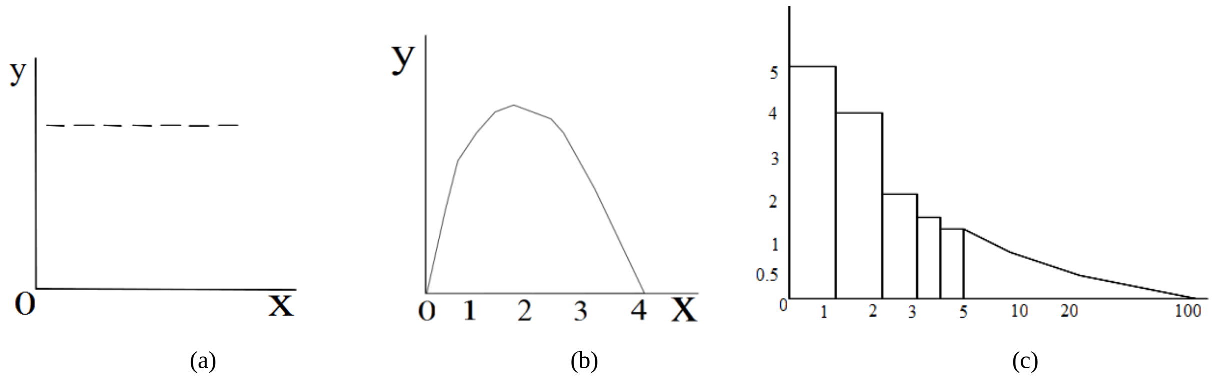

Fig 3: Sample (a) uniform, (b) Poisson, (c) Zipf distributions

4. STATISTICAL PROPERTIES AND PARALLEL MODELS

The size of cities, word lengths, and word frequency in literally works is known to follow a Zipfian distribution. The inter-arrival intervals of events often follow Poisson distribution. Similarly, several other data points could be represented using one or another statistical property. In this section we have discussed some of these distribution to present their overview and later presented parallel implementations of these distribution properties to be implemented using CUDA.

4.1 The uniform distribution

This is the most basic distribution observed in any data set, and most often other complex distributions could be obtained by skewing uniform distribution. Fig 3 (a) represents sample table of uniform distribution. The graph could be obtained by data set specific linear function which may have intervals based upon mean, median or other similar property derived from available dataset. The algorithm for a parallel CUDA implementation is given below.

4.1.1 Algorithm (Parallel Uniform Distribution using CUDA)

1. Generate distribution step value on CPU
2. Upload the value generated in step 1 onto GPU
3. Initialize row matrix on GPU
4. Populate matrix using step value generated in step 1
5. Read back matrix to CPU

The same algorithm steps were tested using CPU-GPU combined model as well as CPU only model to access performance gain. The results have been discussed in the next section.

4.2 Poisson distribution

Poisson distribution could be defined as a quantized probability distribution function which is used to find possibilities of occurrence of events within a fixed range of space and time. Condition being, these events have to occur independently and with a fixed known rate of occurrence. Fig 3(b) represents a sample Poisson distribution graph.

Poisson function could be represented as follows using equation (1).

$$P\ (k\ is\ the\ interval) = {\delta^k e^{-\delta}}/{k!} \qquad (1)$$

Here, δ is the average number of events in each interval, e is the constant 2.718 and k is a natural number interval.

Parallelization of Poisson distribution is achieved indirectly by parallelizing factorial part of the Poisson function and thus saving time on computation time on factorial.

4.2.1 Algorithm (Parallel Poisson distribution using CUDA)

1. Call Poisson function (CPU execution)
2. Call fact() function (CPU execution) to calculate factorial of interval k.
3. Initialize row matrix as → [1, 2, 3….k]
4. Upload row matrix onto GPU
5. Perform row array multiplication, wait for synchronization.
6. Read back value to fact() on CPU.
7. Return value to Poisson function.

The results for this algorithm along with other algorithms have been discussed in section 6.

4.3 Zipf distribution

Zipf distribution could be defined for Integers in the range 1 to N such that the Integer k in the range is given a weight inversely proportional to the index of the integer k , let's call this weight as theta; also, here a parameter called skew is defined in the range 0 < theta < 1. This skew parameter defines the curve of Zipf distribution. Formally, Zipf function could be represented as below by equation (2) mathematically

$$f(k; s; N) = \frac{1/k^{s}}{\sum 1/N^{s}} \qquad (2)$$

Here, N is the total number of elements in the distribution; k is rank of these elements; s is exponent which specifies the curve of the distribution. Fig 3(c) represents a sample Zipf distribution graph. Parallelization of Zipf distribution function was achieved by parallelizing power functions in equation (2).

4.3.1 Algorithm (Parallel Zipf distribution using CUDA)

1. Call function power(base, power) (CPU execution)
2. Initialize all cells of the array with base, where size(array)=power.
3. Upload the array onto GPU.
4. Perform parallel synchronized multiplications, read back value on CPU.
5. Perform remaining operations on CPU.

Similarly, the summation part of Zipf function too was parallelized using similar technique in our model. The model was tested for both CPU/GPU combined execution and CPU only execution like other distribution functions.

5. EXPERIMENTAL SETUP

Following setup was used for performing the stated experiment and to validate the proposed approach for dataset distribution generation.

a. Intel Xeon W3565 CPU with 4 cores, each at 3.20 Ghz
b. NVIDIA Quadro 2000 graphics card with 4 shared memory each with 48 CUDA cores (total 192) and wrap size: 32.

The setup and requites libraries were installed on CentOS 7- an enterprise Linux variant. The codes were compiled using gcc 4.4 and NVCC compliers. The distribution models were tested using both CPU only and CPU-GPU combined processing setups. The detailed results have been discussed in section 6.

6. RESULT AND ANALYSIS

As expected, the compute time on GPU-CPU combined was faster compared to CPU alone for larger data sets. Here, we generated the distribution for three common statistical properties normally observed in large datasets. Out of these, Zipf is of prime importance, as it has been studied and used extensively in big data research earlier. Section 4 apart from introducing these distributions presents the models we evolved for highly parallel GPU architecture. Also, as entire model being ported to GPU was not possible due to GPU memory constraints and loop dependencies and other factors affecting parallel execution results. Only a part of model most suitable for GPU computation was ported to GPU. Table 2 through 4, summarizes test results. Compute time is in milliseconds (ms), which have been calculated with respect to number of elements for uniform and Zipf distribution and number of events for Poisson distribution.

Table 2. CPU/GPU computation speedup comparison for Uniform distribution

Number of Elements	CPU compute time	GPU compute time	Speed up
100	0.003392	0.029568	0.114719
400	0.002016	0.031232	0.064549
2500	0.010144	0.032512	0.312008
10000	0.030752	0.044768	0.686919
40000	0.127456	0.074272	1.716071
250000	2.527296	0.388096	6.512038
1000000	3.346848	2.4102808	1.388614
25000000	84.294243	36.564320	2.305369
100000000	330.299377	141.446945	2.335147
144000000	476.054718	205.950531	2.311500

Table 3. CPU/GPU computation speedup comparison for Poisson distribution

Number of Events	CPU compute time	GPU compute time	Speed up
100	0.002368	0.031008	0.076367
400	0.005728	0.030720	0.186458
2500	0.006208	0.036544	0.169877
10000	0.034688	0.042208	0.821835
40000	0.132960	0.074816	1.777160
250000	0.001984	0.389088	0.005099
1000000	3.222560	1.485152	2.169852
25000000	82.625664	36.749279	2.248362
100000000	328.314667	142.489182	2.304137
144000000	474.054718	207.540253	2.283894

Table 4. CPU/GPU computation speedup comparison for Zipf distribution

Number of Elements	CPU compute time	GPU compute time	Speed up
100	0.005760	0.060576	0.095087
400	0.007744	0.061952	0.125000
2500	0.016352	0.069056	0.236793
10000	0.065440	0.086976	0.752391
40000	0.260416	0.149087	1.746738
250000	2.529280	0.388096	6.517150
1000000	5.569408	3.895152	1.429830
25000000	136.919907	55.313599	2.475338
100000000	658.614044	273.446945	2.408562
144000000	850.109435	413.490784	2.055933

As evident from the above data, the CPU only compute time is better for lower range of data, where as the GPU assisted computation out paces in case of calculations involving larger data sets. However as in this experiment, it was observed there was sharp rise in CPU/GPU compute time ratio, followed by uniformity in this ratio for larger dataset range. Such observations result out of the fact that for smaller values the CPU- GPU data upload and read back time overhead is significant and results in faster compute time on CPU alone computation. However, as the data size increases, the upload and read back time becomes negligible compared to actual compute time, and hence CPU-GPU combined model outperforms CPU only model.

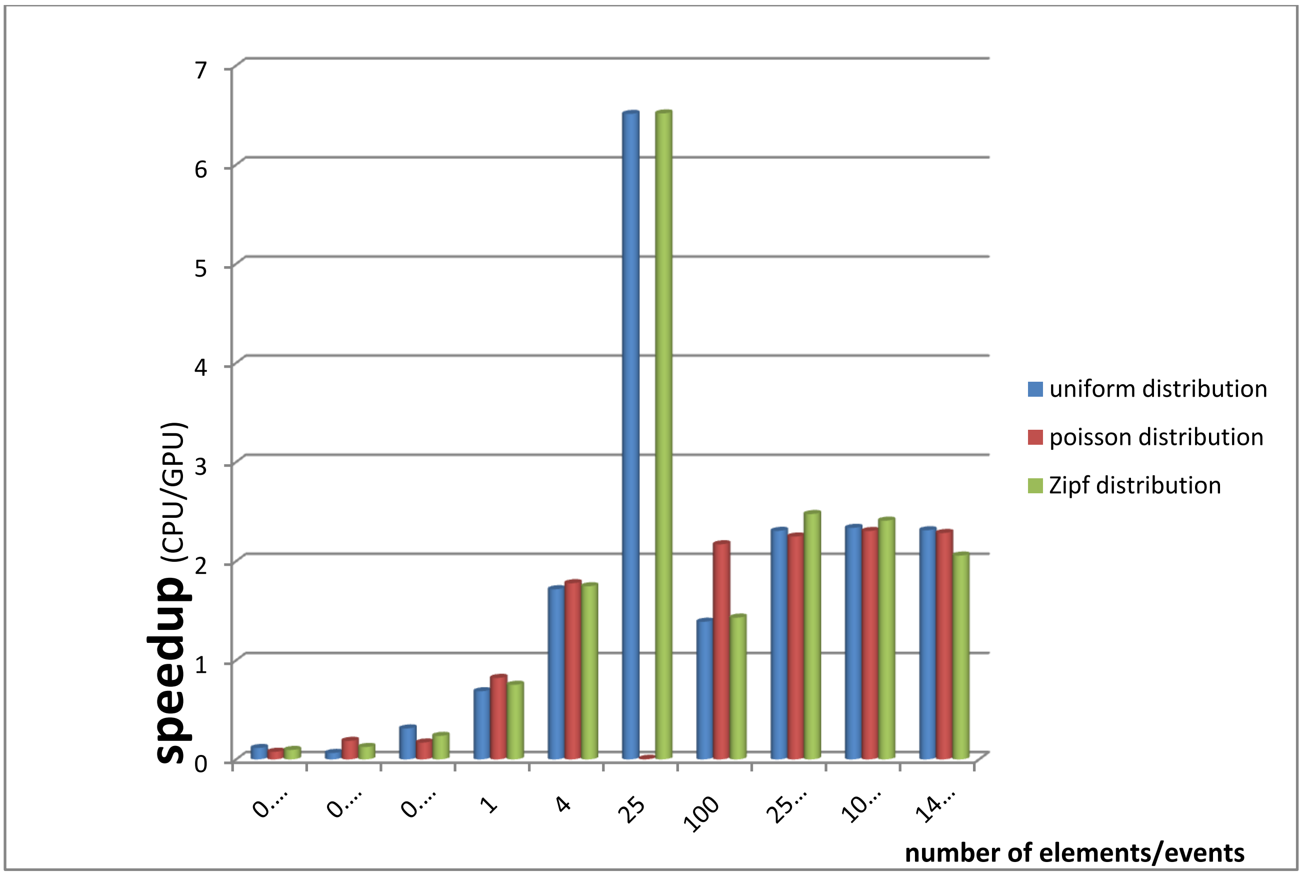

Fig 4: Graphical representation of CPU/GPU speedup ratio (figures are presented as N x 10^4), ratio above 1 represents faster CPU-GPU combined model compared to similar CPU only model.

7. SUMMARY

This paper presents the mechanism to parallel execute parts of synthetic data set generation models on GPGPU, and thus speed up the overall algorithm. Initial sections' deals with the architecture and process flow of the CUDA, followed by a brief introduction of various statistical properties generally observed in the data sets.

Later parallel algorithms have been introduced, followed by experimental results to validate these algorithms. Also, a school of thought prevail that GPU based models are highly optimized compared to CPU only model, hence any result with better GPU result are most often biased. In experiments performed here, the models used in both cases- CPU only and CPU-GPU combined- were kept almost identical. Also, initial results for lower range of data sets was skewed in favor of CPU only model, this was due to high communication overhead between CPU and GPU. Once data sets were large enough, the communication overhead became negligible, and an overall speed up was observed in the GPU compute time over CPU.

We believe a more optimized approach could be built and more statistical models could be parallelized to represent other distribution types. Future work should expand the work in this direction. And a unified framework could be proposed for generating synthetic datasets using these models.

8. REFERENCES

[1] Hao, W., Ning, Y., Chakraborty, P., Vreeken, J., Tatti, N. and Ramakrishnan, N. 2016. Generating Realistic Synthetic Population Datasets. arXiv preprint arXiv:1602.06844.

[2] Cukier, K. 2010. Data, Data Everywhere. Technical Report. The Economist.

[3] Tay, L. 2013. Inside eBay's 90PB data warehouse. Technical Report. ITNews.

[4] Layton, J. 2006. How Amazon Works. Technical Report. HowStuffWorks.com.

[5] Ster, V. D. and Rousseau, H. 2015. Ceph- 30PB Test Report. Test Report. CERN.

[6] DeWitt, S. and Cohen, J. 2010. NASA Goddard Introduces the NASA Center for Climate Simulation. Press Release. Goddard, NASA.

[7] Hoag, J. E. and Thompson, C. W. 2007. A parallel general-purpose synthetic data generator. ACM SIGMOD Record 36, no. 1.

[8] Gray, J., Sundaresan, P., Englert, S., Baclawski, K. and Weinberger, P. J. 1994. Quickly generating billion-record synthetic databases. In ACM SIGMOD Record, vol. 23, no. 2, pp. 243-252.

[9] Nathaniel, B., Zhao, H., Du, S. and Stolfo, S. J. 2014. Synthetic Data Generation and Defense in Depth Measurement of Web Applications. In International Workshop on Recent Advances in Intrusion Detection, pp. 234-254. Springer International Publishing.

[10] Shimpi, A. L. and Wilson, D. 2006. Nvidia's GeForce 8800 (G80): GPUs Re-architected for DirectX 10. Technical Report. AnandTech.

[11] Silberstein, M., Schuster, A., Geiger, D., Patney, A. and Owens, J. D. 2008. Efficient computation of sum/products on GPUs through software-managed cache. In Proceedings of the 22nd annual international conference on Supercomputing - ICS '08.

[12] NVIDIA, CUDA. 2009. Architecture: Introduction & Overview. NVIDIA Corporation.