Parallel CE using CUDA

Chinmay Talegaonkar, *IIT Bombay*

Abstract— **Cross Entropy method is a very useful algorithm for estimating and optimizing complicated stochastic processes involving rare events. The mechanism of the algorithm makes it very easy to parallelize. In this report we revisit the problem of speeding up the CE method using an Nvidia GPU and CUDA C/C++. We analyzed the problem of parallelizing the cross-entropy algorithm for maximizing a peak detector function and solving the Max-Cut problem, under the considerations of the CUDA and Nvidia architecture. This report consists of the factors we need to take into account while using CUDA to parallelize the algorithm.**

Keywords— ***Cross Entropy Optimization, CUDA, Parallelization***

I. Introduction

This report talks about using CUDA for speeding up Monte Carlo Simulations. In particular, we look at the implementation of the Cross Entropy (CE) method for solving two well known problems, namely the **Max-Cut** problem and maximizing the output of a **Peak Detector** function. We start with a brief overview of the CE method and its applications for solving the two problems. This is followed by a brief introduction of the CUDA programming model, and the results/inferences we obtained while implementing the CE algorithm using CUDA.

A. The CE Method

The CE method [1] was initially developed to estimate *rare-event* probabilities, but it was soon realized that it can be used for optimization problems as well. One can think of obtaining the maxima or minima of function as being related to estimating the occurrence of a rare event, which can be then estimated using the CE method. Let $\mathcal{X}$ be a finite set of states and S a real-valued performance function on $\mathcal{X}$. We wish to find the maximum value of S over $\mathcal{X}$ and the state(s) which achieve this value. Let $\boldsymbol{\gamma}^*$ be the maximum of S over $\mathcal{X}$ and let $\mathbf{x}^*$ be a state at which this maximum is attained. Then,

$$S(\mathbf{x}^*) = \boldsymbol{\gamma}^* = \max_{\mathbf{x}\in\mathcal{X}} S(\mathbf{x}). \tag{1}$$

The CE method is an iterative optimization algorithm. It starts off with generating the sample states $\mathbf{x}$ according to a parameterized distribution $f(\mathbf{x};\mathbf{u})$. Each of these samples is then ranked according to its performance on a specified optimization problem. In accordance to the performance of all the generated samples for a given optimization function, a given top fraction of these samples is selected as *elite* samples. The parameter vector $\mathbf{u}$ is updated according this set of elite samples. A new set of sample states and elite samples is then generated using the updated parameters. The distribution finally converges to a degenerate one, about the locally optimal which is ideally globally optimal also. The CE method uses the notion of *KL Divergence* (also known as cross entropy) from the domain of information theory, to minimize the distance between the distributions $f(\mathbf{x};\hat{\mathbf{v}}_i)$ and the degenerate distribution about $\mathbf{x}^*$.

Report Sumbitted in July, 2017.

The CE method begins with converting the optimization problem (1) into an estimation problem. Let $\mathbf{I}_{\{S(\mathbf{X})\geq\gamma\}}$ be the set of indicator variables for various output levels γ. Then for the discrete case we can solve (1) by estimating the probability

$$\ell(\boldsymbol{\gamma}) = \mathbb{P}_u(S(\mathbf{X})\geq\gamma) = \sum_{\mathbf{x}} \mathbf{I}_{\{S(\mathbf{x})\geq\gamma\}} f(\mathbf{x};\mathbf{u}) = \mathbb{E}_u[\mathbf{I}_{\{S(X)\geq\gamma\}}].$$

Now we use a two-part iterative approach [6] to obtain $\hat{\gamma}_1, \hat{\gamma}_2, \ldots, \hat{\gamma}_i$ and corresponding parameter vectors $\hat{\mathbf{v}}_1, \hat{\mathbf{v}}_2, \ldots, \hat{\mathbf{v}}_i$ such that $\gamma_i \to \gamma^*$ and $f(\mathbf{x};\hat{\mathbf{v}}_i)$ approaches the degenerate distribution about $\mathbf{x}^*$. Let ρ be a real number between 0 and 1, which represents the fraction of the samples generated to be used as the *elite* samples. For a random sample $\mathbf{X}_1, \ldots, \mathbf{X}_N$ let $S_{(1)} \leq \cdots \leq S_{(N)}$ be the performances of $S(\mathbf{X}_i)$ ordered from smallest to largest. For a fixed $\hat{\gamma}_t$ and $\hat{\mathbf{v}}_{t-1}$, derive $\hat{\mathbf{v}}_t$ from the solution of the following program:

$$\max_{\mathbf{v}} D(\mathbf{v}) := \max \frac{1}{N}\sum_{i=1}^{N} \mathbf{I}_{\{S(\mathbf{X}_i)\ \geq\hat{\gamma}_t\}} \ln f(\mathbf{X}_i;\mathbf{v}). \tag{2}$$

Algorithm 1 Cross Entropy Optimization

```
1) Choose an initial parameter vector $\hat{\mathbf{v}}_0$. Set $t = 1$.
2) Generate a sample $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_N$ from the density
   $f(\cdot;\hat{\mathbf{v}}_{t-1})$ and compute the sample $(1-\rho)$ quantile
   $\hat{\gamma}_t$ of the performance according to $\hat{\gamma}_t = S_{(\lceil 1-\rho)N\rceil)}$.
3) Use the same sample $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_N$ and solve the
   stochastic program (2) to obtain $\hat{\mathbf{v}}_t$.
4) If for some $t \geq d$, say $d = 5$,
       $\hat{\gamma}_t = \hat{\gamma}_{t-1} = \cdots = \hat{\gamma}_{t-d}$.    (3)
   then stop, else set $t = t + 1$ and iterate from Step 2.
```

B. Simple Peak Detector

This is a simple problem of optimizing a function to find its maxima. We characterize the inputs as a bi-variate Gaussian distribution $\mathbf{x} = \{x_0, x_1\}$, and use the CE algorithm to let it converge to a degenerate distribution $\mathcal{N}(\mu^*, \sigma^2)$ about the maxima ($\sigma \to 0$). The updated values of the means $\hat{\mu}_1$ and $\hat{\mu}_2$ after convergence give an estimate of the state $\mathbf{x}^*$ for the chosen function

$$S(\mathbf{x}) = 3(1-x_0)^2 e^{-(x_0^2+(x_1+1)^2)} - 10(\frac{x_0}{5} - x_0^3 - x_1^5)e^{-(x_0^2+x_1^2)} - \frac{1}{3}e^{-((x_0+1)^2+x_1^2)}. \tag{4}$$

The function has multiple peaks and valleys because of which, the maxima we obtain using the CE algorithm depends on the value of $\hat{\mu}_1$ and $\hat{\mu}_2$ used for initializing the algorithm.

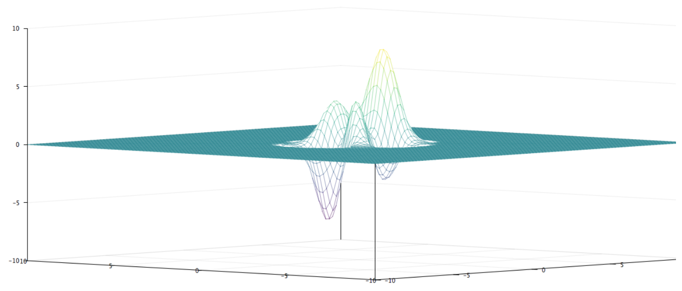

Fig. 1. Plot of the function $S(\mathbf{x}) : \mathbb{R}^2 \to \mathbb{R}^1$

The plot of the function illustrates the above statement more clearly. The advantage of using the CE algorithm here, is that we avoid finding derivatives of the function for a large number of samples, as we would have done in case of gradient descent. Updating parameters using the CE algorithm is computationally much simpler, as we just need to find the average and variance of the elite fraction of the sample values. In the results section, we demonstrate the speed up obtained using a GPU as compared to a CPU for a given function $S(\mathbf{x})$.

C. The Max-Cut Problem

Let $G = (V, E)$ be a weighted and undirected graph. $V = \{1, ..., n\}$ is the set of vertices and E is the set of edges with associated weights c_{ij} between vertices i and j. The Max-Cut problem is to partition V into two distinct subsets V_1 and V_2, for maximizing the sum of the weights $\forall e_{ij} \in E$, where vertices i and j belong to different subsets. The graph G is assumed to be complete and the weights c_{ij} are non-negative. The Max-Cut problem is known to be NP-Complete [7]. The edge weights can be represented via a non-negative, symmetric cost matrix $C = (c_{ij})$ where c_{ij} is the weight of the edge between vertices i and j. The cost of a cut (its score) is then the sum of the weights of the edges with vertices i and j in different partitions. For example, if we had the cut $\{\{1, 2\}, \{3, 4\}\}$ with the following cost matrix

$$\begin{pmatrix} 0 & c_{12} & 0 & c_{14} \\ c_{21} & 0 & c_{23} & 0 \\ 0 & c_{32} & 0 & 0 \\ c_{41} & 0 & 0 & 0 \end{pmatrix}$$

the cost of the cut will be $c_{14} + c_{23}$.

A cut can be written as a vector $\mathbf{x} = (x_2, \dots, x_n)$ where $x_i = 1$ if node i is in the same partition as node 1, and $x_i = 0$ otherwise. For generating our samples we let $X_2, \dots, X_n$ be independent Bernoulli random variables with success probabilities $p_2, \dots, p_n$. Using $p_2, \dots, p_n$ as the parameters, and the Bernoulli distribution according to them, the solution to the stochastic program (2) for updating the sampling distribution in Algorithm (1) becomes:

$$\hat{p}_{t,i} = \frac{\sum_{k=1}^{N} \mathbf{I}_{\{S(\mathbf{X}_k) \geq \hat{\gamma}\}} \mathbf{I}_{\{X_{ki}=1\}}}{\sum_{k=1}^{N} \mathbf{I}_{\{S(\mathbf{X}_k) \geq \hat{\gamma}\}}} \tag{5}$$

For Max-Cut Experiments, one can generate graphs using the method described in the paper by Evans, Keith and Kroese [6]. Also, since the cost of the cut is independent of the direction of the edge, using a symmetric matrix is not mandatory. Other representations like an upper or lower triangular matrix can also be used to solve the problem.

D. A brief overview of CUDA

The CUDA programming model makes use of the parallel computing power of Nvidia GPUs to solve computationally intensive problems more efficiently than everyday CPUs. The programming model is essentially based on thread hierarchies, memory access and allocations on the GPU and synchronizing the executions of operations on threads.

The CPU is referred to as the host, and GPU as the device. The code is composed of two distinct parts, the *host code* and the *device code*. The code in CUDA C++ is compiled using the compiler *nvcc*. Along with compiling the standard C++ code which is executed on the CPU (host code), *nvcc* also compiles the CUDA code fragment to be executed on the GPU (device code). A **thread** on a GPU is the finest granum, or the smallest unit that can execute an operation. A group of threads is called a **block**. A block is a virtual abstraction for a group of threads that interact closely by the means of shared memories. Further extending the hierarchy, a **grid** is a collection of blocks on a GPU. The advantage of introducing this hierarchy instead of just using a huge bunch of threads, is that it allows some degree of control in decomposing a problem. There exists excellent documentation for details about these terminologies [2], but the basic idea is that each block has some memory allotted to it, which can be accessed by all threads in that block. Shared memory on a block differs from the GPUs global memory, as it can be accessed by threads of that particular block only. Also, the time taken to access shared memory is much lower than that required for accessing global memory on a GPU, which makes it capable of increasing the throughput of the GPU. A GPU differs significantly from a CPU in terms of memory access. The memory bus between GPU memory and the GPU cores (multiprocessors) is both wider and has a higher clock rate than a standard bus, enabling many more data to be sent to the cores than the equivalent link on the host allows[5]. A general paradigm for using GPUs for parallel computations is as follows:

- **Device functions and Kernels**: Device functions are the functions which can be invoked as well as executed only on the GPU. They are declared with the qualifier *__device__*. **Kernels** are special functions which are executed on the device, but can be invoked from the code being executed on the host (CPU). Kernels in CUDA are declared using the qualifier *__global__*. The number of blocks and threads to be used for the computations is specified as arguments to the kernel function. E.g. - *add*$<<<30, 500>>>$ *(a, b, c)* calls the kernel or the global function named add, on 30 blocks and allocates 500 threads per block, which is a total of 15000 threads. Kernels are supplied with arguments in the host code, and the outputs generated on the GPU can be then transferred to the host code using pointers.

- **Memory Allocation on the GPU**: Since the CPU and GPU have separate memories, its imperative that memory needs to be allocated on the GPU for the kernels to operate. The Graphics card (GPU) has its own local memory as well, which can be accessed from within the kernel, but for inputs to be passed from the host to the GPU, respective memories have to allocated on the GPUs. This exchange of data between the CPU and GPU memories is achieved through pointers, and the CUDA API functions, *cudaMalloc, cudaMemcpy,* etc[4]. which enable data transfer and memory allocation to the GPU from the host itself.
- **Breaking down the problem**: Essentially any data-parallel problem can be broken down into smaller decoupled and identical problems, which are then executed in parallel on individual threads. Once we get the results from all the threads or blocks, they can be pooled or processed on the CPU to get inferences from the simulation. CUDA allows not just linear data structures, but also 2D and 3D arrays. Exact implementation details are well described in the documentation[4].

Blocks operate independently of each other, and are scheduled using *streaming multiprocessors* (SMs). Streaming multiprocessors are essentially the hardware implementation of the CUDA programming model. A GPU usually has 30 SMs, and each SM has 8 CUDA cores. At a given time each SM executes a group of 32 threads in synchronization. This group of 32 threads, all of which execute the same instruction at a given time, is called a *warp*.

When a CUDA kernel is invoked by the host code, the blocks (which consist of multiple 32-thread warps) are scheduled as per the the availability of resources on a SM. A warp is the physical implementation of threads on a multiprocessor. The warp schedulers can issue warps either from different blocks, or from different places in the same block, if the instructions are independent. Hence warps from multiple blocks can also be processed at the same time. Taking warps into account becomes important for maximizing the performance of an algorithm. It is preferred choosing number of threads in multiples of 32 that divide evenly with the number of threads in a warp. If this is not followed, one ends up launching a block that contains inactive threads. Each warp executes one common instruction at a time, which enables the threads to execute in lockstep, i.e. work in synchronization on one instruction. Divergence in the lockstep will be described in detail, in the Results and Inferences section.

E. Parallel CE using a GPU

The CE method can be easily implemented in a parallel or distributed manner, due to its inherent nature of independently generating samples from a distribution. In this section, we have a look at how to parallelize the CE algorithm

The CE method performs two tasks: the generation of samples from the sampling density $f(\cdot;\hat{\mathbf{v}}_{t-1})$; and updating this density based on the samples. Generating samples can be parallelized using a GPU, since the CUDA programming model allows parallel processing on the threads.

Updating the sampling density can then further be divided in two parts-: Scoring of the generated samples to identify the elite fraction of samples, and updating of the sampling density based on the elite samples. We use a single CPU to update the sampling density and then use it to generate samples in a parallel fashion on the GPU. We generate multiple samples on a thread, and then select the best sample to be forwarded for pooling into the elite sample set. This approach generates more samples using fewer blocks and threads, while ensuring a better quantile of elite samples at the same time. Even though the samples are being generated sequentially on a given thread, since the task of generating samples is divided across threads, the parallel nature of our algorithm is still preserved.

Algorithm 2 Parallel CE optimization on a GPU

```
1) Choose an initial parameter vector $\hat{\mathbf{v}}_0$. Set $t = 1$.
2) Generate on each of the $k$ threads on the GPU, a sample $\mathbf{x}^{\dagger}{}_1, \mathbf{x}^{\dagger}{}_2, \dots, \mathbf{x}^{\dagger}{}_p$ from the density $f(\cdot;\hat{v}_{t-1})$.
3) From the $p$ samples on each thread, select the sample $\mathbf{x}^*$ with the highest performance on the function $S(\mathbf{x})$. Pool the $k$ samples from each of the $k$ threads respectively on the CPU.
4) Using these samples $\mathbf{x}^*{}_1, \mathbf{x}^*{}_2, \dots, \mathbf{x}^*{}_k$ pooled from the GPU, compute the sample $(1-\rho)$ quantile $\hat{\gamma}_t$ of the performance according to $\hat{\gamma}_t = S_{(\lceil 1-\rho)N\rceil)}$.
5) Use the same sample $\mathbf{x}^*{}_1, \mathbf{x}^*{}_2, \dots, \mathbf{x}^*{}_k$ and solve the stochastic program (2) to obtain $\hat{\mathbf{v}}_t$.
6) If for some $t \geq d$, say $d = 5$,
       $$\hat{\gamma}_t = \hat{\gamma}_{t-1} = \dots = \hat{\gamma}_{t-d}. \quad (6)$$
   then stop, else set $t = t + 1$ and iterate from Step 2.
```

II. Parallel CE using CUDA

A. Implementation Overview

We solved the above two problems using both the CPU and CUDA programming on an Nvidia GPU. Lets have a look at the results obtained from implementing the Peak detector optimization for various values of sample sizes (N), both on a CPU and GPU. The execution time for various batch sizes of samples on the CPU increases almost linearly, as seen in 2. This is roughly intuitive, as the execution time is proportional to the number of samples we generate for a simulation. Even though the CE algorithm converges when the variance is less than 10^{-5}, every iteration of the loop in the code on CPU

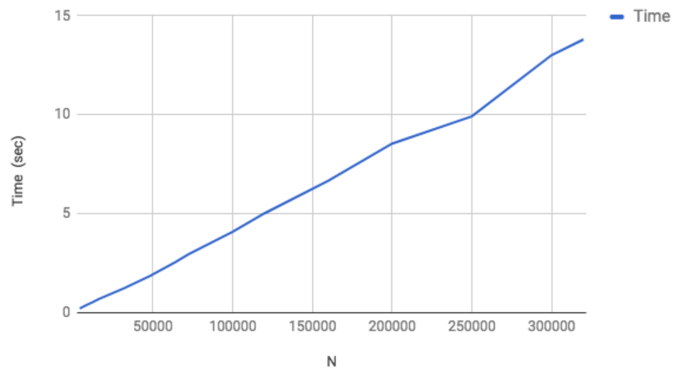

Fig. 2. Time vs N for the peak detector on a CPU

has a direct dependence on the number of samples (N), which explains the almost linear nature of the curve we obtained.

When implementing the peak detector using GPU, we used the CURAND[3] library to generate random numbers. The *curand_init* function initializes the random number generator, using the seeding value supplied as an argument to the kernel invoked in the host code. One important thing to take into account while generating random numbers using CURAND, is that the the function *curand_init*, which initializes the random number generator should preferably used outside the kernel, or the calls to it should be minimized to the extent possible. The call to the *curand_init* function is very expensive, hence multiple calls

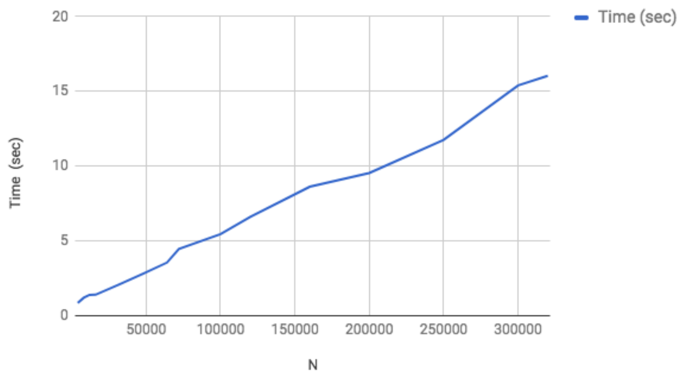

Fig. 3. Time vs N for the peak detector on a GPU with *curand_init* inside the loop on the kernel

to it should be avoided. Once initialized with a seeding value, random numbers can be generated using the CURAND state variables[3]. For taking N samples per iteration on the GPU, we allocate m blocks, n threads per block and p simulations per thread, such that $m*n*p = N$ and $m*n = k$, where k is the total number of threads on which the kernel is executed in algorithm (2). The value of p for the peak detector has a very small range of 40-3200. The difference in the execution times of the kernel due the varying value of p is quite insignificant (the time taken by a simple *for* loop on C++ for 40 iterations is 2ms roughly, same as that for 6400 iterations) . Hence the significant fraction of time required in the execution of the code using GPU is the time taken by the *cudaMemcpy/cudaMalloc* functions for memory access on the GPU . As discussed earlier, memory needs to be allocated on the kernel from the host code, and also the computations done on the kernel need to be copied from the GPU to the host code. The CUDA API functions *cudaMalloc, cudaMemcpy* etc. provide this functionality. The interfacing between the host and GPU is time consuming, and hence it is always preferable to avoid unnecessary calls to CUDA memory function calls to allocate and copy large sized data structures. When *curand_init* is kept inside the *for* loop in the kernel, and is being invoked p times by every thread, the execution time vs N graph in this case is roughly linear. m and n are held constant, only p is being varied to change the value of N. The execution time is quite high, since it is expensive invoking the *curand_init* function this many times. When *curand_init* is kept out of the loop, i.e., when it is invoked only once per thread, the number of curand_init calls is $m * n$ for any value of N and p. We earlier discussed that the there is not much significant time difference when $p = 40$ or 3200 in a *for* loop (the processor speed is of the order of GHz). Hence, the time delay due to invoking that function is roughly constant. Therefore we can see that when *curand_init* is out of the loop, the execution time is almost constant. The increase in the size of the arrays being copied to and from the the GPU account for the slight but not so significant increase in the execution time. Most of it is the time taken by the calls to the functions *cudaMemcpy* and *cudaMalloc*.

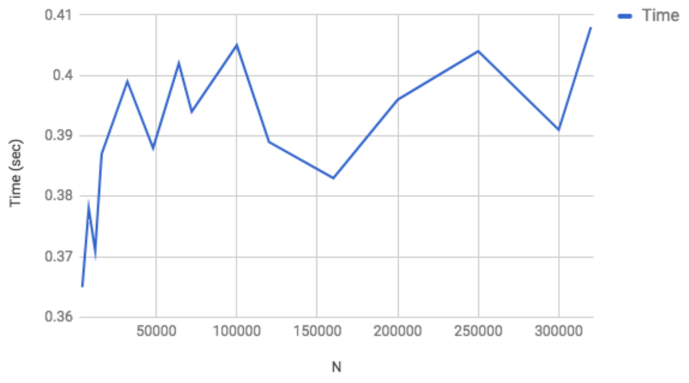

Fig. 4. Time vs N for the peak detector on a GPU with *curand_init* outside the loop on the kernel

B. Optimizing the code

We saw that using a GPU definitely speeds up Monte Carlo Simulations, due the fact that we are dividing the task of generating N random samples among $m * n = k$ threads. Generating N samples in a parallel manner is bound to give a speed up, compared to generating them sequentially. But, there are some factors we need to take into account while using the CUDA programming model.

1) There is a critical number of registers required by a thread for running an instance of a kernel. The registers

or memory required by the threads for a given process determines the number the threads that can be run concurrently[5].

2) Threads in the same block can work together, i.e., can access shared memories. Also, there is no way to sync the working of two different blocks, but threads in the same block execute the same instruction at the same time (SIMT). Hence, it is always preferable to use more number of threads on a block.
3) Keeping the *curand_init* function out of the main CUDA kernel, or at least calling it multiple times should be avoided as we discussed.
4) The data exchanged between the CPU and the GPU should be kept to its minimum, since the memory bus between the GPU and CPU memory has a low bandwidth.
5) The execution time in the graph should be taken as the average of the time taken for executing the same code a couple of times. Also, while executing the output file of the CUDA code for the first time after compiling, the execution time is quite high, due to the delay in setting up the communication links for data transfer between the CPU and the GPU.
6) While using shared memory in the kernel, take into account the use of *__syncthreads* function[2], for synchronizing thread operations within a block.

III. Results and Inferences

A. Peak detector

Keeping these things in mind, on tweaking the values of m and n, along with varying p, we obtained an amortized speed up of roughly **3000**x, for the maximizing the peak detector function. The GPU is able to achieve convergence of the CE algorithm, with $N = 3.2 * 10^8$ in 3.879 seconds, while the CPU takes 13.87 seconds for a value of $N = 320000$. Due to constraint on the size limit of a single array, one cannot increase the size of the array by any other further factor of 10. This roughly gives us an amortized speed up of the order of 3000. This is primarily because of the lockstep mechanism of operation of threads on GPU in a warp. Any divergence from the lockstep, that can be created by the use of *if-then-else* statements in the kernel results in slower speed. Different branches corresponding to different divergence paths are created in a SM. Each of these branches is executed separately, which causes delay. We used the thrust library to sort the arrays on the host, instead of the kernel. Sorting inherently leads to divergences, hence instead of writing our own code for sorting in the kernel, using the compiler optimized function in the thrust library is a much better alternative. Another important factor, which governs the speed up is the value of p. We can see from the table, that for a constant number of simulations, as we increase p, the algorithm converges faster.

B. Max-cut

On solving the max-cut problem for a graph with 6 nodes and 2 non-zero edge weights, we found that convergence is achieved faster using a GPU as compared to a CPU. For the max-cut problem, the condition for convergence, is that

$$||\hat{\mathbf{p}}_t - \hat{\mathbf{p}}_{t-1}|| < 10^{-5}.$$

$\hat{\mathbf{p}}_t$ is the vector of probabilities of the vertices falling in a cut, obtained by the CE method after t-th iteration. On taking a value of $N = 10000$, we obtain a speed up of roughly **140**x. But, this is because, on increasing the value of N, we increase the number of threads, blocks, or number of threads per block we use, due to which the time taken by the GPU for achieving convergence is roughly a constant, as can be seen from the graph, while time taken by the code on CPU increases with the value of N. Hence the speed up we obtain depends heavily on the way we allocate the threads, blocks and simulations per thread on the GPU.

TABLE I. Caption for the table.

No. of Samples (N)	p	m	n	Time
2500000	25	100	1000	4.905
2500000	250	100	100	0.979
2500000	2500	10	100	0.471
25000000	250	100	1000	2.858
25000000	2500	100	100	1.537
25000000	25000	10	100	9.189

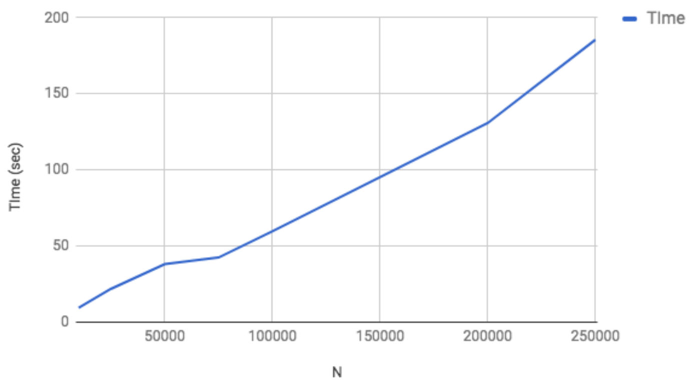

Fig. 5. Time vs N for solving the Max-Cut on a CPU

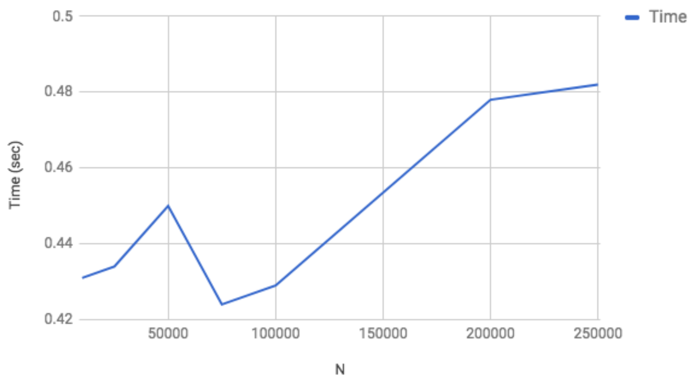

Fig. 6. Time vs N for solving the Max-Cut on a GPU with *curand_init* out of the loop on the kernel

C. *Inferences*

What we observe in Table 1 is primarily due to the working of CE algorithm. We rely on pooling the elite samples, and then updating the value of the parameters accordingly. We use a slightly different approach while using CUDA. Increasing p up to a certain extent, and then taking the sample corresponding to the highest output value from each thread. So on the host code, we get an array of the size $m * n$, which has the best of the samples generated on each thread. From this array, we then again take the top $\rho\%$ of samples and update parameters accordingly. This approach helps in achieving convergence faster, that too with the same order of precision and accuracy. An intuitive reason why this works, is that since the samples we consider for the updating the parameters are being chosen after sampling twice. Hence, the fraction of the samples in this case closer to the final value of the parameters will be higher than that in case of what we obtain using the code on CPU. Also, taking more number of simulations per thread, effectively gives us a more optimal data sample from that thread, as compared to using a lesser number of simulations on that thread. This can be effectively seen in the table. This is what makes the parallel-data implementation converge faster than that on a CPU. So, we obtain a speed up not just because we generating the samples fast, but we are also accelerating the process of convergence of the algorithm along with it, by taking a better pool of samples for updating parameters.

IV. Future Work

Since we saw that we can obtain speed ups of the order of **3000**x using a single GPU, the idea of distributed computing using GPUs is definitely imperative. Updating the parameters locally on the GPU and then using the local updates for updating the parameters on the Host computer, as mentioned in the future work section of [6] still needs to be worked upon. We noted an important observation, that sampling in batches, on each thread by increasing the number of simulations upto a limit leads to faster convergence. Even though the intuition is clear, evaluating this hypothesis for other problems which use the CE algorithm will help modelling this phenomenon mathematically. A mathematical model, which can talk about the dependence of execution time on the value of p based on the sample size can be a significant step towards understanding the convergence of CE algorithm.

Software packages like *BOINC* provide a great framework for volunteer and Grid computing. A setup for distributed Monte Carlo simulations requires setting up a *BOINC* server account, and a database at its backend. After that one just needs to install the *BOINC* manager on the host computers in the cluster, and register them with the desired project, as to allocate their idle time for the same. The details of setting up the project on a *BOINC* server can be looked up in the official tutorials and documentation of *BOINC*. One can also use the framework *PyMW*[8], which essentially acts as wrapper over *BOINC*, to simplify task submission and retrieval process.

V. Conclusion

Hence we saw, that an Nvidia GPU can be effectively used for solving problems pertaining to the domain of Monte Carlo Simulations, due the requirement of generating a large number of random samples. To investigate the computational power of a GPU, we used the CE algorithm for solving the Max-Cut and maximizing the peak detector function. We compared the performance of the GPU and a single CPU for the same. The observations suggest that the GPU performs much better in terms of speed than a CPU, and in case of the Max-Cut problem, we can even make the execution time invariant to that of the sample size by taking more blocks and threads. We also looked upon important factors to consider while optimizing the CUDA program. Also, we obtained some insight on the convergence rate of the CE algorithm using the parallel implementation on CUDA.

Reproducible Research: The cost matrix for the Max-Cut problem and all the code used for this project is uploaded on github, and can be forked from this repository.

Acknowledgment

This project would not have been possible without the guidance of Prof. Kroese. Despite the fact that we were communicating remotely through Emails, his involvement in this project kept me motivated throughout. Dr. Sarat was a constant support and very easily approachable for discussions and doubts. Interacting with him was indeed of great help to me. I would like to thank Alankar Kotwal for helping me approach Prof. Kroese for this project and supporting me throughout the summers. I would also like to thank Shashwat Shukla for helping me out with the basics and queries related to CUDA programming. Last but not the least, a big thanks to Innovation Cell, IIT Bombay, for letting me use their GPU for this project. It was a good learning experience overall!

References

[1] Rubinstein, R. Y., and D. P. Kroese. 2004. *The cross-entropy method*: A unified approach to combinatorial optimization, Monte-Carlo simulation, and machine learning. Springer.

[2] Nvidia, 2012. *CUDA C Programming Guide*

[3] Nvidia, 2012. *CURAND guide*

[4] Jason Sanders, Edward Kandrot. 2011. *CUDA by Example*

[5] Anthony LEE, Christopher YAU, Michael B. GILES, Arnaud DOUCET, and Christopher C. HOLMES. 2010. On the *Utility of Graphics Cards* for Massively Parallel *Monte Carlo Simulations*

[6] Evans, Keith and Kroese. 2007. *Parallel Cross Entropy Optimization*

[7] Karp, R. M. 1972. *Complexity of computer computations*, Reducibility Among Combinatorial Problems, 85103. Springer.

[8] Eric M. Heien, Yusuke Takata, Kenichi Hagihara and Adam Kornafeld. 2009. *PyMW* - a Python Module for Desktop Grid and Volunteer Computing