Lattice Cubature Using GPUs and Matlab

Michael Bonilla

December 4, 2018

Abstract

This work is in effort to provide some GPU capabilities to the Guaranteed Automatic Integration Library (GAIL) [1] namely in the Quasi-Monte Carlo integration techniques using lattice points [2]. Using Matlab we will attempt to integrate functions over high dimensional regions using lattices and high dimensional cubature. First we will generate the lattices in parallel and using the work of the contributors of GAIL we will transform these points using Fast Fourier Transforms (FFT) and come up with an approximation of the integral. We will also attempt to incorporate the automatic integration features provided by GAIL into our GPU implementation. To illustrate our findings we will implement the methods on various suitable problems of many dimensions.

1 Introduction

Integration techniques in low dimensions (1,2,3) are extensively used in many computing libraries such as trapezoidal rule, Gaussian quadrature, etc. Once we head to higher dimensional problems such as 10 or more dimensions these techniques become rather cumbersome and are not efficient or reliable. For higher dimensional problems we often utilize Monte-Carlo methods to approximate the definite integral of high dimensional regions and functions. A branch of these Monte-Carlo methods includes Quasi-Monte Carlo methods which instead of using pseudo random numbers we use quasi-random numbers. Using such numbers can gives us faster convergence over usual pseudo random number methods.

2 Background and Preliminary Theory

Traditional Monte Carlo uses independent and identically distributed (IID) random numbers to represent samples x_i where x_i can be multidimensional.

Therefore we approximate the integral

$$\int_\Omega f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

for Ω is the domain we wish to integrate over. Most of the time Ω is either $\mathbb{R}^d$ or $[0,1]^d$, the d dimensional hypercube, for d is the dimension of the domain. The x_i are generated based on the distribution we need for the appropriate domain. For example if we work on the entire domain $\mathbb{R}^d$ we would use normally distributed x_i in d dimensions. Similarly for a hypercube we will use uniformly distributed x_i on $[0,1]^d$. Once we obtain the x_i they are used to evaluate the function f and are averaged over the N number of x_i. This is essentially the traditional pseudo-random Monte Carlo method and it has an order of convergence of about $\mathcal{O}(1/\sqrt{N})$. Meaning if we would like to half the error tolerance we would need to increase the sample size by a factor of 4.

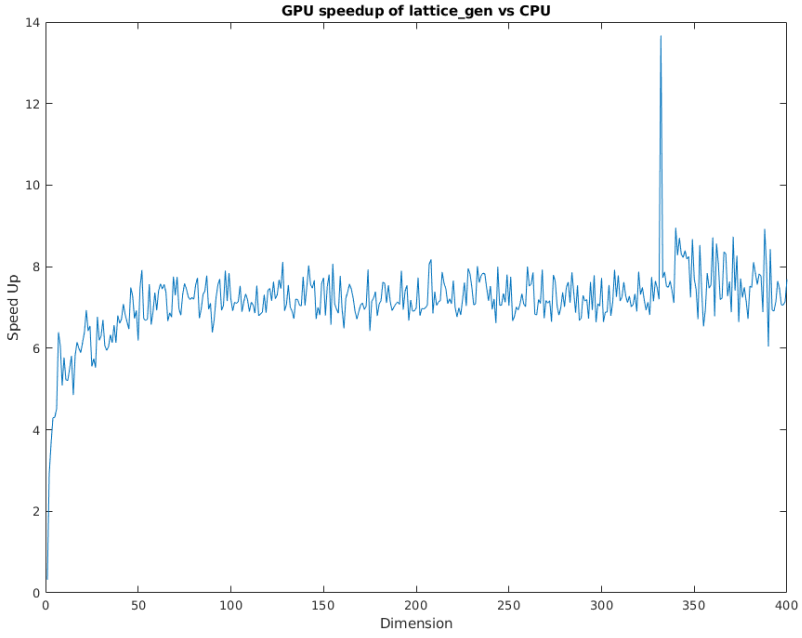
On the other hand Quasi-Monte Carlo is the same as Monte Carlo except of how we choose the randomized x_i. The x_i are no longer chosen in an IID manner but are now actually chosen using low-discrepancy sampling [2]. Essentially the x_i are spread across the d dimensional hyper cube in a clever way and are essentially shifted by a uniformly distributed shift Δ. Using this method of choosing x_i gives us a order of convergence of nearly $\mathcal{O}(1/N)$ [2]. The result is we get a much faster convergence by using fewer samples and the disadvantages can range from having a biased estimator or larger variance depending on the problem.

We have gone over the basics of Monte Carlo and Quasi Monte Carlo and if one were to implement these methods the user will have to set the number of samples themselves based on the problem for a need error tolerance. The work done by Hickernell and his group [1] created the Guaranteed Automatic Integration Library (GAIL) allows the user to numerically integrate functions in high dimensions using Monte Carlo methods automatically. Automatically in this context means the user provides a function, domain, and error tolerances at the very least and the library and its functions will determine the work needed automatically to achieve such error tolerances. Work done by Ruguma and Hickernell [2] is incorporated in GAIL in the function *cub_lattice*. The Quasi-Monte Carlo is done as previously described and the automatic part of the algorithm is based on the work in [2]. For the most part the algorithm will converge to the desired tolerance if the Fourier coefficients of the evaluated $f(x_i)$ diminish quickly enough otherwise more

samples are used until the algorithm uses up the prescribed budget given by the user allowed by the theory.

3 Results and Analysis

After analyzing the algorithm made by Rugama and Hickernell [2] we have identified the parts of the algorithm that will benefit most for parallel implementation on a GPU. First is in the generation of the lattice points themselves despite not being a process that is embarrassingly parallel. Using the GAIL library to choose the points in a clever way we can apply the random shift to the points on the GPU to gain a speed up in this process of 6-8 times depending on the dimension of the problem.

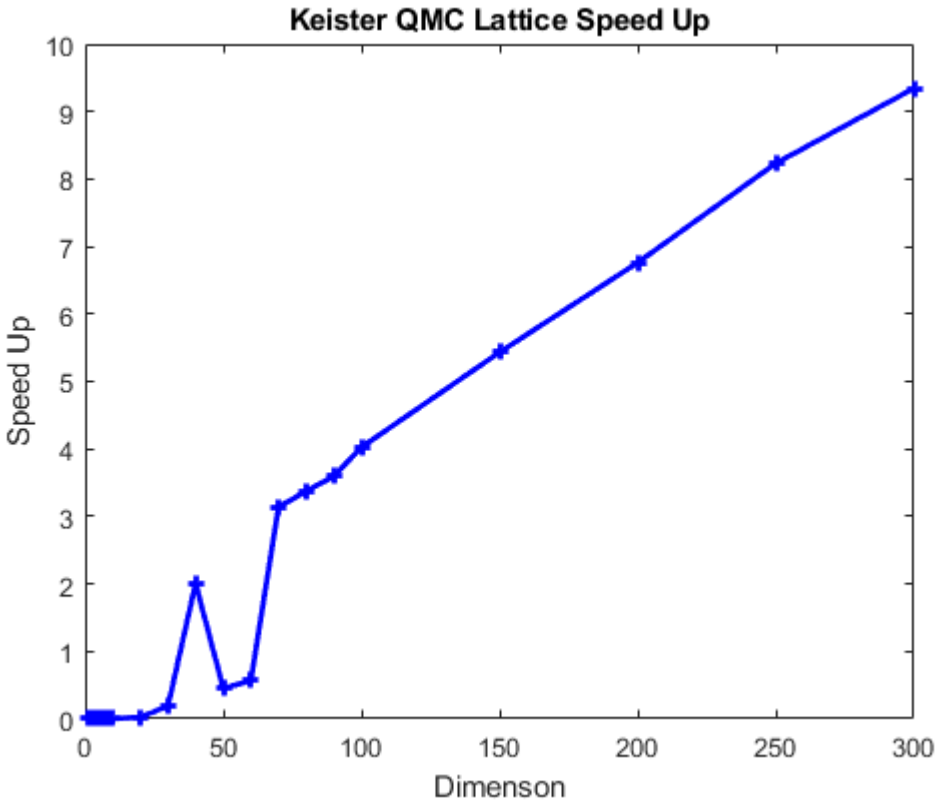


Then next speed up we can achieve is in the evaluation of the function over the x_i generated by the lattice process and the overall mean achieved from these function evaluations. Speed ups in this manner really depend on the problem at hand whether the functions themselves are easy to evaluate themselves. In order to test the speed of the GPU we use a benchmark using

in various literature the Keister function given by

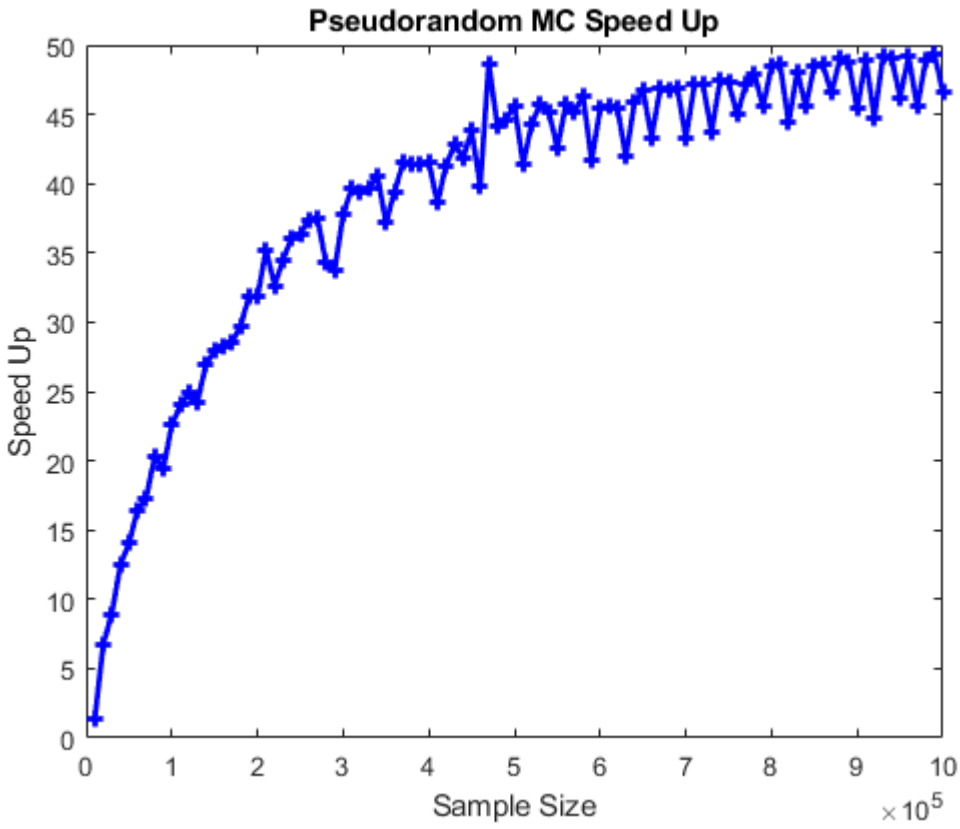
$$\int_{\mathbb{R}^d} \cos(\|x\|) e^{\frac{\|x\|}{2}}$$

This examples is a vector function over a multidimensional domain and the numerical solution can be ill conditioned if done in high dimensions. Using this example we can see the speed ups given over various dimensions compared to using a CPU over a GPU.



For this particular example we achieve a speed up of 3 to 11 times faster depending on the dimension. Again this is due to generating the lattices in parallel and the evaluation and averaging lattice points in parallel. When it comes to the automatic integration feature of GAIL it is explained later that it is more beneficial to leave the algorithm checks on the CPU for implementation due to the nature of GPU computing and the algorithm. Despite this we do achieve some nice speed ups but this is very much dependent on

the algorithm used and the problem. On the other hand if we were to implement the same speed ups in a regular pseudo-random Monte Carlo method for IID samples we can obtain even faster speed ups.



In the preceding plot we have an implementation of IID Monte Carlo for pricing a European stock option using various samples sizes. As we stated before an IID implementation is an example of an embarrassingly parallel problem. From here the GPU demonstrates its most potent uses and we achieve speed ups form 10-50 times depending on the number of samples used. In examining the speed ups in this examples one may notice that the speed ups do not scale one might expect in fact as the samples increase the speed up we achieve begins to decelerate. This might occur for any problem using this many sample size but the most contributing factor is in fact the hardware being used to implement the algorithm. In our case the GPU is indeed a limiting factor in the speed ups achieved and will in fact vary

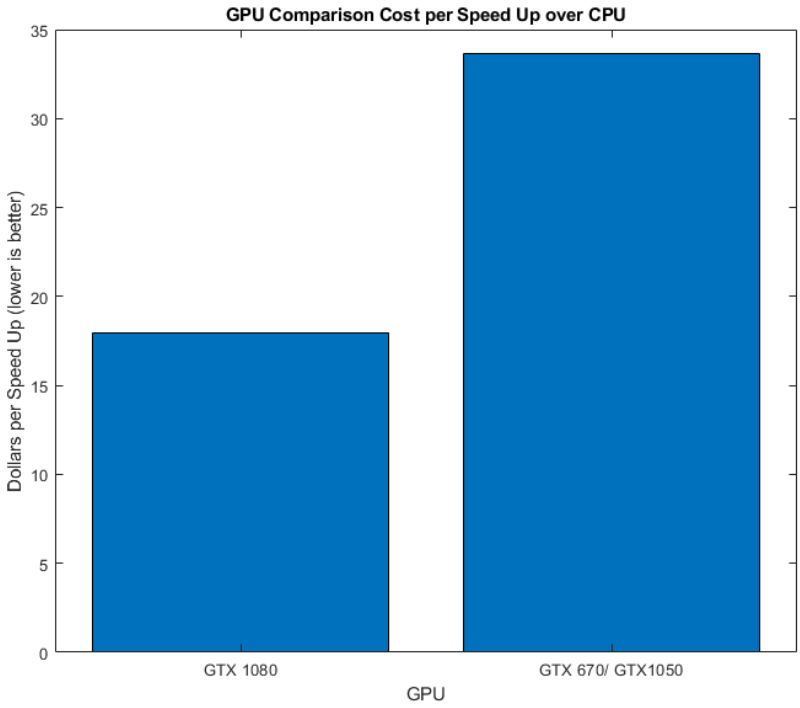
from user to user depending on their hardware. GPU's can vary in various categories most importantly in memory, core count, and memory bandwidth. All three of these factors contribute very much into the price of a GPU now a days therefore we would like to know if paying more for a GPU can be beneficial achieving meaningful speed ups over a CPU implementation. In the proceeding plot we have the same option pricing problem used for pseudo-random Monte Carlo implemented on two different GPU's. From this plot we observe the more expense GPU gives us more value per speed up, meaning we pay less per speed up over a CPU implementation. In this simple comparison between two GPU's one might conclude that paying more will yield better speed ups but there is some caution to be taken this approach. For instance this example is on a example that is embarrassingly parallel, where GPU's benefit greatly. If one were not working on a problem as such it may be beneficial to explore options in deciding which GPU to purchase/use in attempting to speed up an algorithm and perhaps a CPU implementation might be more viable due to the synchronization overhead, which is discussed later.

4 Shortfalls and Future Work

There were some issues along the way in trying to implement GPU routines of functions into GAIL. For the most part the GPU integration into the function *cub_lattice* works well but when GAIL uses *cub_lattice* in other routines like option pricing we run into so bugs. With enough time and knowledge of the GAIL classes I am sure these issues can be resolved for a complete working GPU implementation of *cub_lattice*. Besides the technical issue in our proposal we said we would need to do some FFT routines in parallel. After investigating the algorithm and work in [2] and [1] we actually only need to compute the Fourier coefficients. This can be done rather quickly and the routine in *cub_lattice* is very efficient. We could have done this on the GPU but the gains in speed are nullified by communication and synchronization overhead between GPU and CPU.

For future work we use the GPU on other cubature methods in GAIL especially those involving IID sampling which works exceptionally well on GPUs. Measuring performance on GPU is also something we can work on in the future. In this project we measured only speed up performance in time versus a single core implementation compared to a GPU. Other performance metrics such as efficiency is a bit more complicated for a GPU because the hardware is not quite the same compared to a CPU. For instance measuring

performance on a multi-core machine vs single core is easier to do because for the most part a single core is identical to any core in a multi-core machine. On the other hand GPU's can vary in a number of hardware ways for example clock speed, memory, core count, memory transfer speeds, and memory bandwidth. Without a more in depth knowledge of GPU architecture measuring performance on a per core basis versus CPU is difficult. Hence we actually compare the performance of a GPU with the price of one might pay and measure how much we actually pay for each speed up over a CPU implementation. We do this on a desktop GPU specifically a GTX 1080 and a latop GPU comparable to a GTX 670. The example we used is an IID Monte Carlo simulation to show off the speed ups on an embarrassingly parallel environment. The following plot essentially shows the more expensive GPU gives us the better value per speed up.



We pay only $17 per speed up versus $34 per speed up for the cheaper GPU. This is so in general since the overall benefit in GPU's is the number of physical cores and the memory transfer bandwidth and speeds which all are expensive and contribute to the price of a GPU. Hence if one were to

use GPU's to speed up an algorithm or routine there must be some caution in how we choose our hardware since price and performance are impacted heavily if one decides to spend to little or much.

References

[1] S.-C. T. Choi, Y. Ding, F. J. Hickernell, L. Jiang, L. A. J. Rugama, D. Li, J. Rathinavel, X. Tong, K. Zhang, Y. Zhang, and X. Zhou. Gail: Guaranteed automatic integration library (version 2.2). `Availablefromhttp://gailgithub.github.io/GAIL_Dev/`, 2017.

[2] L. A. J. Rugama and F. J. Hickernell. "adaptive multidimensional integration based on rank-1 lattices,". In R. Cools and e. D. Nuyens, editors, *Monte Carlo and Quasi-Monte Carlo Methods: MCQMC*, volume 163, pages 407–422, Leuven, Belgium, April 2016. Springer Proceedings in Mathematics and Statistics.