Monte Carlo Integration: the VEGAS algorithm for Graphics Processing Units

Luis Biedma[a] and Flavio D. Colavecchia[b]

[a] *FAMAF - Universidad Nacional de Córdoba, CIEM - CONICET, lbiedma@famaf.unc.edu.ar*
[b] *Div. Física Atómica, Molecular y Óptica, Centro Atómico Bariloche, CONICET, flavioc@cab.cnea.gov.ar*

Abstract: This work presents a review and modification of the VEGAS integrating algorithm in Graphics Processing Units. The changes made to the program allow for better performance and use of memory to incorporate more evaluation points, which was an important restriction for the original code on high dimension spaces.

Keywords: *GPU, Monte Carlo methods, VEGAS*
2000 AMS Subject Classification: 65C05 - 68M20

1 Introduction

Monte Carlo methods are a wide and well known class of computational algorithms that rely on repeated random samplings to get numerical results. One of their main applications is the numerical integration of functions in problems ranging from particle physics[1] to cosmology[2]. The usual working pattern for these algorithms is:

- Definition of a possible input domain.
- Generation of random inputs from a *probability distribution* over the domain.
- *Deterministic* computations over the inputs.
- Results gathering.

Since its creation, lots of efforts have been made in order to boost Monte Carlo methods performance, especially concerning the minimization of numerical errors and the algorithm's speed, trying to adapt them to every new coming computing architecture, including graphics processing units (GPUs), by creating different alternatives for them[3].

One of those alternatives is the VEGAS algorithm, created by G.P. Lepage[4]. It is based in the concept of *importance sampling*: It samples points from the probability distribution described by the function $||f||$ to be integrated, so that the sampling points are concentrated in the regions that make the largest contribution to the integral. VEGAS is based on an iterative and adaptive Monte Carlo scheme: each axis of variable is divided into grids, dividing the integration space into hypercubes. Monte Carlo integrations are performed on each hypercube and the variances from hypercubes are used to adapt the shape of the grids, which will be used in the next iteration, reducing the variance of the total integral at each step.

The aim of this work is to improve the code of a multi-dimensional VEGAS algorithm, which uses GPUs, created by J. Kanzaki in 2011[5].

2 The Initial Program

Kanzaki's code can be found at http://madgraph.kek.jp/KEK/GPU/gVEGAS/example/ and is used as the basis for the new program. It is written in CUDA language, adapted from FORTRAN code created originally by Lepage[6]. There is a single and double precision version available in the webpage. The steps followed by the program in each iteration are:

1. Generation of the integration space, dividing it into hypercubes.
2. That data is sent to the GPU, where the function evaluations are performed and the information per hypercube is gathered.

3. These results are sent back to the CPU, where weighted averages, approximation errors and the reallocation of the grid are performed.

4. If the amount of iterations is reached or a minimum value of error is obtained, the program stops.

Although this code has great performance relative to the first stages of VEGAS (evaluating a function in lots of different points in space on GPUs is an embarrasingly parallel task), there are still some issues that are addresed in this work.

There is a considerable amount of data moved between GPU and CPU. The code generates an array of evaluations (one for each point) and also needs to store information regarding the grid box the point is occupying, for each dimension, in the integration space. So, for every evaluation point, the program needs (*dim*+1) * 4 bytes (in single precision) of RAM, where *dim* is the dimension of the integration space. When working on GPUs, memory management is an important issue and it is not properly addressed in this code.

The complete weighted averages are computed in the CPU, but it can be also performed in GPU, because the *reduction* operation is easy to parallelize. The grid evaluation can also work in parallel.

3 Modifications

The new CUDA kernel performs the values (and their squares) reduction, aided by the *atomicAdd* instruction[7] and also takes the data to an histogram array, which is passed to the CPU to obtain the grid rearrangement for the next iteration. In conclusion, this code merges steps 2 and 3 for the original program. This array has a fixed (very small relative to the amount of points generated) size, so there aren't any memory issues.

An important note is that the original code passes single precision numbers to double when it computes the weighted averages for the integrals. The new code uses only single precision since it works with the *atomicAdd* instruction and it is only available for single precision in the Maxwell architecture, which was the one used for this work.

The entire code is hosted at https://github.com/lbiedma/gVegascp.

4 Numerical Results

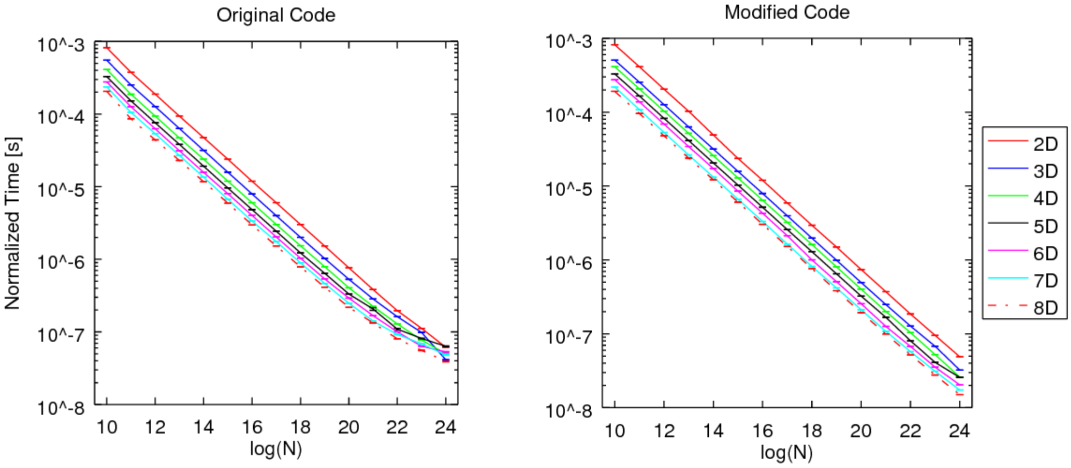
The performance experiments are divided in two stages: pure speed comparison and numerical correctness. They were run on the following hardware:

CPU	2x Intel(R) Xeon(R) CPU E5-2620 (12 cores at 2.40 GHz)
Memory	128 GB DDR3
GPU	NVIDIA GeForce GTX Titan X (12 GB RAM)

Speed Comparison

The termination conditions were set at 1×10^{-6} for the relative error between estimated integrals per step and 10 as the max amount of iterations. The space dimension d was limited to a maximum of 8 and the amount of points used was increased on each execution from 2^{10} to 2^{24} (the memory limitations on the original code made it impossible to reach higher dimension numbers). Only for speed comparison purposes, the multi-dimensional paraboloid ($\|x\|^2$) in the $[-1, 1]^d$ hypercube was chosen.

The following graph shows the performance of the original and the modified programs, with time normalized by the number of points used and the amount of iterations.



It can be observed that both programs perform at almost the same speed at first, but the new program continues to maintain a good performance as the amount of function evaluations increases. More points couldn't be evaluated because of memory restrictions for the original code.

Benchmarking

The numerical correctness tests for the modified program were based on Scilab's testing toolbox for Monte Carlo and Quasi Monte Carlo methods[9]. The integral of the functions is zero for every dimension. 17 integrals from this toolbox were tested, with dimensions ranging from $d = 6, \ldots, 12$, and evaluation points $n = 262144, 524288, 1048576, \ldots, 67108864$ (powers of 2 from 18 up to 26) in the $[0,1]^d$ hypercube. The random number generator used was a version of XORSHIFT available in the repository.

For each amount of evaluation points, the success rate of the program (percentage of integrals along with their standard deviations that are equal to zero) and a normalized average computation time is reported.

Points	Rate	Time [s]
2^{18}	42.0 %	4.133×10^{-9}
2^{19}	52.4 %	2.323×10^{-9}
2^{20}	58.8 %	2.095×10^{-9}
2^{21}	80.7 %	1.463×10^{-9}
2^{22}	68.1 %	1.388×10^{-9}
2^{23}	68.9 %	1.304×10^{-9}
2^{24}	67.2 %	1.298×10^{-9}
2^{25}	77.3 %	1.278×10^{-9}
2^{26}	67.2 %	1.214×10^{-9}

Overall, it can be observed that the success rate doesn't necessarily increase with the amount of evaluation points.

5 Summary and Outlooks

The modifications applied to the original program allowed it to perform at a higher speed for big problems and manage memory in a better way, which is an important issue when working with GPUs, since RAM is highly limited.

NVIDIA's new GPU architecture (Pascal) incorporates the *atomicAdd* instruction for double precision numbers[8], this will be really useful to achieve high performance in double real and complex precision computations, and may be the subject of future work.

Acknowledgments

The first author would like to thank Nicolás Wolovick and Carlos Bederián, of the GPGPU Computing Group in FAMAF, Universidad Nacional de Córdoba, for their Parallel Programming course and its notes[10], where this work began.

References

[1] S. Wertz, *The Matrix Element Method at the LHC: Status and prospects for Run II*, 17th International Workshop on Advanced Computing and Analysis Techniques in Physics Research, 2016.

[2] A. Taruya, *Constructing perturbation theory kernels for large-scale structure in generalized cosmologies*, Physical Review D - Particles, Fields, Gravitation and Cosmology, 2016.

[3] J.H.Halton, *A Retrospective and Prospective Survey of the Monte Carlo Method* SIAM Review, 1970.

[4] G.P. Lepage, *A new algorithm for adaptive multidimensional integration*, Journal of Computational Physics, 1978.

[5] J. Kanzaki, *Monte Carlo integration on GPU*, The European Physical Journal C, 2011.

[6] G.P. Lepage, *VEGAS: An Adaptive Multi-dimensional Integration Program*, Cornell preprint CLNS, 1980.

[7] NVIDIA, *CUDA C Programming Guide*, http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[8] NVIDIA, *CUDA Pascal Tuning Guide* http://docs.nvidia.com/cuda/pascal-tuning-guide.

[9] M. Baudin, *The Integration Test Problems Toolbox, version 0.1*, Scilab, 2010.

[10] Nicolas Wolovick, *Computación Paralela 2016*, https://cs.famaf.unc.edu.ar/ñicolasw/Docencia/CP/2016/index.html, FAMAF - UNC, 2016.