CUDA Implementation of Vegas Enhanced Algorithm

Emiliano Tolotti
Mat. 220230
emiliano.tolotti@studenti.unitn.it
University of Trento
Trento, Italy

ABSTRACT

This project aims at the development of a Nvidia CUDA implementation of the Vegas Enhanced Algorithm. This algorithm is a numerical integration method which employs multiple techniques of variance reduction in a Monte Carlo setting. Being a Monte Carlo integration method, the task is well suited for parallelization and for GPU execution. In this report a possible CUDA implementation of the algorithm is proposed, with an analysis of the various choices and optimizations. The performance of the GPU version will be then compared with equivalent sequential and OpenMP CPU implementations.

KEYWORDS

Vegas Enhanced, Monte Carlo, Parallelization, OpenMP, CUDA

1 INTRODUCTION

Numerical integration of functions is a very common task, which has many applications in physics, astronomy and other fields. In particular, Monte Carlo methods aim at estimating the integral value with the usage of random numbers. The estimate is an approximation of the correct value, being an average of multiple evaluations of the integrand function, in random points of the domain. The accuracy of the estimate increases with the number of function evaluations. Monte Carlo methods for integration have been proved to be quite robust and suitable for the calculation of multidimensional integrals, but naive approaches have low performance regarding the convergence rate of the estimate. Many methods and variance reduction techniques have been proposed to boost performance of Monte Carlo integration. In particular, the Vegas Enhanced Algorithm employs importance sampling and adaptive stratified sampling to increase performance in the estimation of multidimensional integrals.

The Monte Carlo integration problem is well suited to parallelization. The task can be distributed among the available cores for the function evaluations, and the results can be later combined to obtain the estimate. Since the number of function evaluations grows for increasing estimation accuracy, in particular in the multidimensional case, GPUs can be exploited, in order to utilize the large number of cores they provide. In this project a CUDA implementation for Nvidia GPUs is considered.

The report is structured as follows: in Section 2, a brief introduction to the Vegas Enhanced algorithm is presented. Section 3 contains the CUDA implementation details, as well as the various implementation choices and optimizations. In Section 4 a performance comparison between GPU and CPU versions for different integrand functions is considered. At the end, in Section 5 some considerations are made.

2 VEGAS ENHANCED ALGORITHM

The *Vegas Enhanced Algorithm* by G.P. Lepage[2], is used to compute Monte Carlo estimates of multidimensional integrals. Monte Carlo integration is robust and makes few assumptions about the integrand function, which doesn't need to be analitical or continuos. The algorithm is adaptive, which means that later iterations are based on information about the integrand collected during the previous steps, in order to improve performance. Each domain axis is divided into grids, such that the integration space is divided into hypercubes. Monte Carlo integration is performed in every hypercube and the variances are used in the next iteration to adapt the grid, in order to decrease the total integral estimation variance. In particular Vegas Enhanced uses two variance reduction methods: *adaptive importance sampling* and *adaptive stratified sampling*.

Adaptive importance sampling aims at sampling points from the probability distribution described by $||f||$, to concentrate samples in the domain regions that make the largest contribution to the integral. The integrand is flattened by means of a Jacobian, and the integral evaluations are made on the transformed variables. In particular (considering the one-dimensional case), the integral:

$$\int_a^b f(x)\,dx \tag{1}$$

is transformed into the equivalent integral:

$$\int_0^1 J(y)f(x(y))\,dy \tag{2}$$

with $J(y)$ being the Jacobian of the transformation, which is chosen in order to minimize integral estimation uncertainty. The integral is then estimated as:

$$I_{MC} = \frac{1}{N_{ev}} \sum_y J(y)f(x(y)) \tag{3}$$

with N_{ev} being the number of uniformly sampled point in the interval $[0, 1]$. I_{MC} is normally distributed in the limit of sufficiently large N_{ev}. The variable map is implemented by dividing each axis in N_g intervals, to map the $[a, b]$ original interval in x space to the new interval $[0, 1]$ in y space, such that intervals of widths Δx_i map to intervals of uniform width $\Delta y = 1/N_g$. The Jacobian of the transformation is:

$$J_{i(y)} = N_g \Delta x_{i(y)} \tag{4}$$

which is a step function in the intervals. It can be shown that the variance of I_{MC} is minimized when $J^2(y(x))f^2(y)$ is the same for every interval Δx_i. The consequence is that:

$$J \propto \frac{1}{|f(x)|} \tag{5}$$

and given that $J_i \propto \Delta x_i$, the jacobian flattens the peaks concentrating samples around them in x space, increasing the number of

intervals Δx_i (and samples) in these regions in y space. The grid is composed by the set of Δx_i, and it is refined at every iteration with a smoothing operation with parameter α to help reducing overreactions of the grid parameters.

Adaptive stratified sampling is used to distribute the number of function evaluations across the hypercubes in order to minimize variance. The y space is divided into hypercubes and the full integral is computed as the sum of the contribution of every hypercube:

$$I_{MC} \approx \sum_h \Delta I_h \tag{6}$$

The number of evaluations per hypercube n_h can be showed to be optimal when:

$$n_h \propto \sigma_h(Jf) \tag{7}$$

constrained by:

$$N_{eval} = \sum_h n_h \tag{8}$$

Where N_{eval} is the total number of evaluations. A damping parameter β is used to reduce fluctuations. This technique greatly improves performance for multiple highly peaked integrand functions.

The integral estimate is then computed combining the estimations of the various iterations of the algorithm, by weighted average considering the estimations variances.

3 IMPLEMENTATION

The code is based on the *CIGAR* [7] implementation in C++ of the Vegas Enhanced Algorithm. In this section the CUDA [4] implementation is described, starting from a base version, commenting the various implementation choices up to the final version. The program follows the structure of the pseudocode in 1. The proposed implementation source code can be found in the repository at [6].

3.1 Program structure

The pseudocode starts with an initialization of the parameters of the algorithm, and the initialization of the random number generators. The default computing of the parameters is:

$$\begin{aligned} N_{it} &= 20 \\ N_{intervals} &= 1024 \\ N_{eval\_it} &= \left\lfloor \frac{N_{eval\_tot}}{N_{it}} \right\rfloor \\ N_{strat} &= \left\lfloor \frac{N_{eval\_it}}{2}^{1/N_{dim}} \right\rfloor \\ N_{cubes} &= N_{strat}^{N_{dim}} \end{aligned}$$

The program then enters the main loop which is repeated for a certain number of algorithm iterations, in which the estimation of the integral is computed and the map is refined to improve the accuracy of the final estimate. For every evaluation in each hypercube, the program generates a random point in the transformed space for which the value of the integrand function is evaluated. Then the jacobian and weights of that evaluation are computed and accumulated for the update of the map and stratification parameters. Then the integral estimation for each cube is computed as well as

Algorithm 1: Program pseudocode

```
initializeParameters();
forall g in generators do
    setupRNG(g);
end
forall it in iterations do
    forall cube in hypercubes do
        forall ev in nh[cube] do
            forall d in dims do
                generateCoordinate(cube, d);
            end
            evaluateIntegrand(ev);
            computeJacobian(ev);
            computeWeights(ev);
            forall d in dims do
                accumulateWeightMap(d);
            end
            accumulateWeightsStrat();
        end
        computeIntegralResult(cube);
        computeIntegralSigma(cube);
        accumulateResults();
        computeDh(cube);
        accumulateDh();
    end
    forall d in dims do
        forall i in intervals do
            normalizeWeight(i, d);
        end
        accumulateWeights();
        forall i in intervals do
            smoothWeight(i, d);
        end
        forall i in intervals do
            backupMap(i, d);
        end
        while condition do
            updateMap(d);
        end
        forall i in intervals do
            resetWeight(i, d);
        end
    end
    forall cube in hypercubes do
        computeNh(cube);
    end
end
if it > skip then
    combineIterationsResults();
end
return results
```

its variance and the values are accumulated to obtain the sum for all hypercubes. Then the weight for the assignment of evaluations the hypercube of the next iteration is computed, and summed to the total of all hypercubes. This part will be called *vegasFill*.

After this section the program enters the update procedure, where the parameters of the map and stratification are updated for the next iteration. The map can be represented as matrix with parameters for every dimension and for every interval. The program iterates in every dimension and as a first procedure normalizes the weights with the counts collected during the *vegasFill* phase. They are then accumulated to sum them and a smoothing operation is applied to each weight. After that, the map is copied into a backup matrix, which is then used to update the new map together with the smoothed weights. The weights are then reset for the next iteration. After that, the stratification parameters are updated, with the computation of the number of evaluations to perform at the next iteration for each hypercube, with the fractions obtained before.

After the computation of the estimate and the update of the parameters, if the iteration index is above a threshold set for the convergence of the map and stratification parameters, the program combines the results of the previous iterations to obtain a weighted estimate for the integral result. After all iterations are competed, the final integral estimate is the output of the program.

3.2 Base version

In the base version of the algorithm the only part which is computed by the GPU is the *vegasFill* section, so the computation of the integral estimate. It is the most expensive section and the one with the most number of work units. In fact, the sample of the point and the relative calculations are performed for every evaluation, and the estimation of the integral results for every hypercube. The total number of evaluations per iteration can be in the order of 10^{10}, and the number of hypercubes about 10^7. In this version the map update is performed by host code. Therefore, the weights data needs to be passed between CPU and GPU. For this task unified memory has been used, with memory allocation performed via the `cudaMallocManaged` function. This lets the driver manage the copying of pages between host and device memory, sharing the addressing space.

The `vegasFill` procedure is performed by a kernel launch, and starts with the random sampling of the point in the transformed space. For the random number generation the *cuRAND* library has been used. In particular an `XORWOW` state is instantiated for each thread that generates random numbers (*setupRNG* in pseudocode), with same seed but different sequence numbers. This initialization is performed by calling a setup kernel that calls `curand_init` passing the thread index as sequence number. The RNG is then performed in the `vegasFill` kernel by means of a `curand_uniform` call, in order to obtain a uniformly distributed random number in the interval $[0, 1]$, from which computing a sample point coordinate. Each thread copies the state from global memory into local memory to improve generation performance as stated by the CUDA documentation, and the copied back to global memory. The state initialization is an expensive operation, so it is unfeasible to prepare a state for each evaluation point, both for speed and memory requirements. To mitigate this fact, the proposed implementation defines a `batch_size` parameter which represents the maximum number of threads for the `vegasFill` kernel, and for which a `XORWOW` state needs to be prepared. Each thread is then presented with a `n_eval/batch_size` number of points to be generated and `n_dim*n_eval/batch_size` calls of `curand_uniform` with the same state. The `batch_size` parameter is a tradeoff between random number quality, memory requirements and speed. As a `curandState` occupies 48 bytes, the batch size limit is in the order of 10^8 for RAM limitations depending on the device. Since the batch size indicates the maximum number of threads for the filling procedure, it needs to be large enough to ensure full utilization of the SMs, depending on the GPU. State initialization is only ran once, so it adds a constant overhead with respect to the total running time of the algorithm. In case of multiple integral estimations or high integrand function complexity, initialization might negligible, but it is not the case with a single or a few evaluations per state and computationally trivial integrands. The second step is the evaluation of the integrand function for the point sampled in the transformed space, and the consequent calculations of jacobian and weights. Then the weights are accumulated for updating map and stratification parameters. The accumulation of weights and result is then carried out with the usage of `AtomicAdd` functions.

After this `vegasFill` procedure, the results are computed for each hypercube. A `results` kernel is launched for every hypercube in order to compute the integrals estimate for every cube, its variance, and the stratification weights. The results are then accumulated with atomic instructions. The rest of the iteration is carried out by host code, applying smoothing to the weights, updating the map and stratification, and computing the estimation at that iteration of the integral value.

3.3 Update on GPU

Until now the update of the parameters has been computed on the host side. It is possible to perform the smoothing of the weights and the update of the stratification on the device, allowing for a reduction in the data copy between GPU and CPU memory. The `vegasFill` and `results` kernels of the previous version remain unchanged, but the code now calls some new kernels. In particular, normalization, smoothing, and reset of the weights can be performed by three different kernels that launch $N_{dim} \cdot N_{intervals}$ threads. The computing of the stratification is also computed by the GPU, with a kernel of N_{cubes} threads.

3.4 Explicit memory copy

The memory management of the previous version was addressed with the usage of unified memory. It is possible to improve the performance by explicitly managing the movement of data between host and device, with the usage of `cudaMalloc` and `cudaMemcpy`, avoiding the driver overhead. The address space is separated between CPU and GPU, and both host memory and device global memory need to be manually allocated.

3.5 Registers and templating of kernels

The `vegasFill` kernel in the previous versions was using global memory arrays for storing the variable map. This situation can

be improved by allowing each thread to allocate it own arrays in memory registers. Registers are implemented physically as on-chip memory, so they are much faster in bandwidth and latency than global memory. In order to achieve this, the size of the arrays needs to be known at compilation time, so it is possible to provide it as a template parameter to the kernel function. In doing so the compiler, can also perform some optimizations in the `vegasFill` kernel such as loop unrolling.

3.6 Reduction and streams

The integral is computed by summation of the contribution of all hypercubes, and the operation has been carried out by atomic instructions in the `vegasFill` kernel. It is possible to compute the reduction of the contributions of all hypercubes (as well as the variances) in parallel with the update of the map in CPU, by means of a `reduce` kernel. The reduction kernel is a version inspired from [1]. The implementation computes block-level reduction with the help of shared memory to store results, with a sequential memory access pattern, and unrolled loops with `blockSize` as template parameter to evaluate expressions at compile time. The warp-level reduction requires no synchronization (SIMD execution inside a warp), and it is performed with a fully unrolled loop. After the block-level reduction, the partial results are accumulated with an `atomicAdd` instruction. The reduction kernel, apart from calculating the integral estimate and variance, is also used for the accumulation of the stratification weights. The update of the map, including the normalization and smoothing of weights as well as their reset, is computed on a different high priority stream with respect to the computation of the integral estimate, variance and stratification update. This allows to parallelize the computations of the device with the update of the map, which is computed by the host.

4 TEST

In this section a performance analysis of the implementation is presented. The GPU implementation will be compared with a sequential CPU implementation as well as a parallel CPU version. A different filling method for the CUDA code will be considered.

4.1 Test methodology

The performance analysis considers 4 test configurations with different integrand functions and evaluations number. Other parameters remain unchanged in the different configurations and are defined below.

```
int max_it=20;
int skip=5;
int max_batch_size=1048576;
int n_intervals=1024;
double alpha=0.5;
double beta=0.75;
```

4.1.1 Configuration 1. This configuration uses the *Roos & Arnold* function [5], which has integral value 1. It is defined as:

$$f(x) = \prod_{i=1}^{d} |4x_i - 2|$$

where the numbers of dimensions considered is `n_dim=10`. A 2-dimensional representation of the function is reported in Figure 1. It is a rather computationally simple integrand, and the number of evaluations is set to `tot_eval=1e8`.

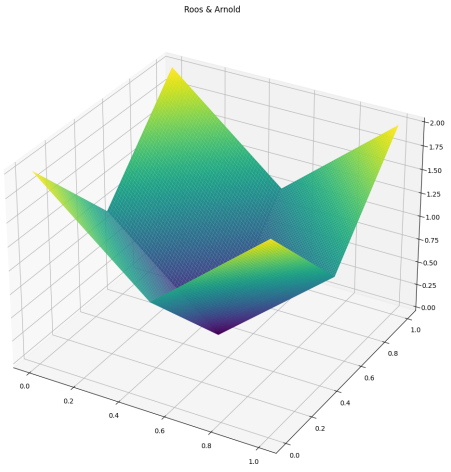

Figure 1: 2-dimensional Roos & Arnold function

4.1.2 Configuration 2. This configuration uses the *Morokoff & Caflisch* function [5], of integral value 1. A 2-dimensional visualization is reported in Figure 2. It is a rather computationally simple integrand, and the number of evaluations is set to `tot_eval=1e8`. The function is defined as:

$$f(x) = (1 + 1/d)^d \prod_{i=1}^{d} x_i^{1/d}$$

where the numbers of dimensions considered in this configuration is `n_dim=8`.

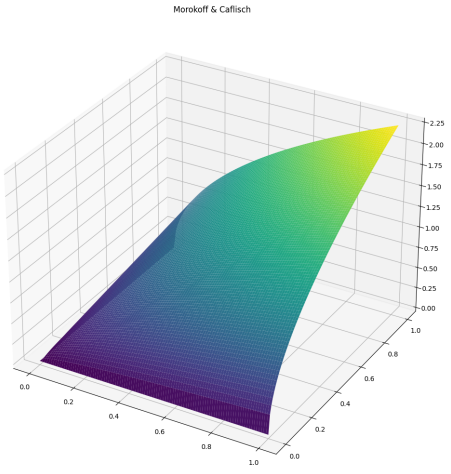

Figure 2: 2-dimensional Morokoff & Caflisch function

4.1.3 Configuration 3. In this configuration a multivariate normal probability density function is considered, centered in the integration domain and with a variance `sigma2=1e-4` in all dimensions. A 2-dimensional visualization of the function is showed in Figure 3. 3. The integral value approaches 1, and the number of dimensions considered is `n_dim=4`. The number of evaluations used is `tot_eval=1e8`.

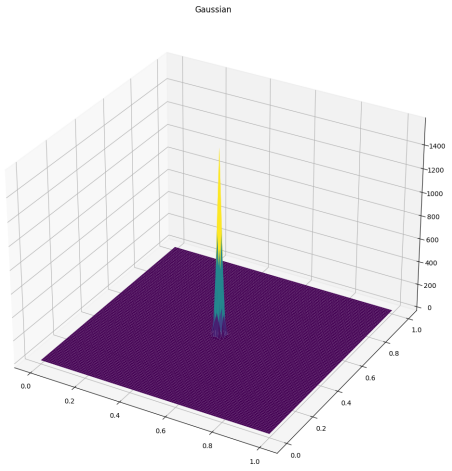

Figure 3: Bivariate normal distribution, $N(\mu = 0.5, \sigma^2 = 0.0001)$

4.1.4 Configuration 4. This configuration computes `tot_eval=1e6` evaluations of a ridge of `N=1000` gaussians as described in the Vegas documentation [3], for which a 2-dimensional case is represented in Figure 4. It is a computationally expensive integrand, and the number of considered dimensions is `n_dim=4`. The integral value is approximately 0.851.

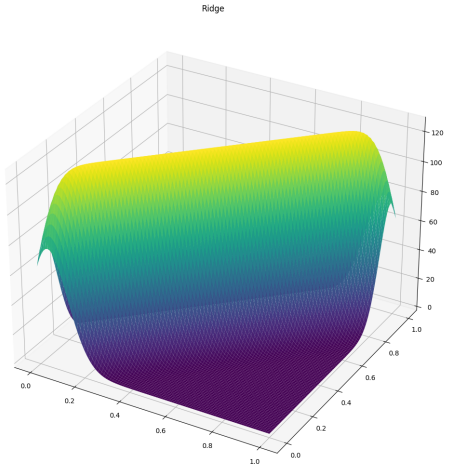

Figure 4: Ridge of 1000 gaussians

All tests are conducted on the system specified in Table 1. The results are reported by collecting 95% confidence intervals.

4.2 Test implementations

The implementations proposed for testing are the following:

- *CPU sequential*: sequential implementation of the program on CPU, with no parallelization, and it is used as a performance baseline for testing.
- *CPU openMP*: parallel CPU implementation, improves on the sequential version by parallelization of the loops with *openMP* over multiple threads.
- *GPU percube*: CUDA implementation that uses a filling procedure with a thread for each hypercube. The number of RNG states to be initialized is the number of hypercubes, and every thread computes every evaluation of the hypercube which is mapped to.
- *GPU batch*: proposed CUDA implementation, after all the optimizations described in Section 3.

4.3 Block size selection

Block size selection is specific to the device architecture and needs some empirical results to be carried out. Block size can not exceed 1024, and should be a multiple of 32 to have fully populated warps, for keeping the possibility to achieve 100% occupancy and for facilitating memory coalescing. Moreover, it is important to have a grid size larger than the number of SMs and multiple active blocks for latency hiding, while avoiding limitation constraints on SM resources. For the `vegasFill` kernel, `blockSize=32` is the optimal configuration, and provides a 3% to 18% performance increase compared to the worst choice (256), depending on the configuration. The `reduce` kernel shows a performance improvement with `blockSize=128` of 25% compared to the worst block size selection (512). The other kernels showed no statistically significant results with different block sizes. Moreover their optimization has little relevancy for the total speedup, because of their little contribution to the total running time, thus intermediate block size values have been used. The total speedup obtained by the best block size configuration compared to the worst case configuration varies from 4% to 12%, depending on the test scenario.

4.4 Floating point precision

The usage of double precision floating point numbers does affect performance, due to the double dimension in memory, and because the GPU architecture used for testing (*Pascal*), has a 1:32 ratio in theoretical double precision performance compared to single precision. Despite this, the performance advantage of using float instead of double is about 8%, averaging the configurations total running time. In fact, the program is not computationally intensive, and memory operations constitute a little fraction of the total time. However, nvprof shows a 400% performance increase for the results kernel, which features floating point calculations. This shows how, at the cost of lower precision in the calculations, the running time could be heavily reduced, for instance with a computationally intensive integrand function. For all implementations double precision has been used.

Table 1: Test system configuration

System specifications	
Processor	Intel Core i7 6700HQ, 4 Core, 8 Threads, base 2.6 GHz, boost 3.5 GHz
Memory	16 GB DDR4, Dual Channel, 2400 MHz
Graphics	Nvidia GeForce GTX 1060 Mobile, 6 GB GDDR5, 1280 cuda, base 1404 MHz, boost 1670 MHz
Software	OS Windows 11 22H2, nvcc V11.5.50, MSVC 19.29

Table 2: Performance comparison

Version	Total					Iteration				
	Running time [ms]				Speedup	Running time [ms]				Speedup
	Conf1	Conf2	Conf3	Conf4		Conf1	Conf2	Conf3	Conf4	
CPU sequential	94362.2 ± 1105.1	101552.7 ± 781.5	82187.3 ± 895.9	348965.5 ± 831.3	1.0x	4718.0 ± 55.3	5077.5 ± 39.1	4109.2 ± 44.8	17448.3 ± 41.6	1.0x
CPU openMP	25241.9 ± 38.1	24914.7 ± 50.8	20283.9 ± 199.6	64330.4 ± 166.5	4.3x	1261.6 ± 1.9	1245.0 ± 2.5	1013.2 ± 10.0	3216.5 ± 8.3	4.3x
GPU percube	573.2 ± 13.0	1602.7 ± 15.0	51357.0 ± 76.3	97871.1 ± 792.0	15.6x	18.3 ± 0.3	67.1 ± 0.3	2552.1 ± 4.0	4886.8 ± 39.7	18.3x
GPU batch	943.6 ± 11.9	1794.4 ± 16.5	1744.4 ± 12.2	8770.3 ± 13.3	57.1x	35.7 ± 0.2	77.1 ± 0.5	75.5 ± 0.3	431.7 ± 0.5	66.4x

4.5 Performance results

The performance comparison is presented in Table 2, which shows the running time of the different implementations for the multiple benchmark configurations. The total running time and the average duration of a single iteration are considered, to show the overhead of the RNG states preparation in the GPU implementations. The speedup is computed as:

$$Speedup(impl) = \prod_{c \in configs} \frac{Time_c(baseline)}{Time_c(impl)}^{1/|configs|}$$

which is the geometric mean of the single speedups, where baseline is the *CPU sequential* version. The parallelization on CPU with openMP shows a speedup of 4.3x, with 8 available threads in the test system. The *GPU percube* version improves with a total speedup of 15.6x and 18.3x when ignoring state preparation time. It is possible to note how performance of this version is unstable depending on the test configuration, being the top performer in some cases, while being slower than the CPU parallel implementation in others. This situation is discussed more in detail in the next section. The *GPU batch* version, which represents the proposed implementation, shows a combined total speedup of 57.1x and 66.4x for a single iteration. Its performance is constant in the multiple test configurations, varying form 40x to 100x.

4.6 Optimizations

In this section, the performance of the various optimizations discussed in Section 3 is considered. As shown in Table 3, an average 4.2x speedup is possible when combining all the described optimizations. In particular, the table lists the optimizations in the same order as described in the previous section. The speedup is computed as the geometric mean of the 4 test configurations, as described above. *Opt1* improves *Opt0* by updating on GPU, and shows a combined performance uplift of 22.9%. The speedup here is limited by the fact that most running time is related to the `vegasFill` part (from 60% to 99% depending on the configuration). On the other hand, the explicit memory copy in *Opt2* introduces a performance increase of 185.6% combined, and shows how unified memory can greatly impact running time under certain circumstances. *Opt3* gains a combined 18.1% speedup, with the usage of registers and compiler optimizations with the help of template parameters. The

Table 3: Optimizations performance

Opt	Running time [ms]				Incr	Tot
	Conf1	Conf2	Conf3	Conf4		
Opt0	6906.2 ± 37.3	9463.7 ± 126.5	12854.2 ± 192.5	9714.6 ± 17.2	+0%	100%
Opt1	5821.5 ± 55.7	6802.0 ± 64.2	9309.0 ± 103.8	9695.7 ± 15.2	+22.9%	123%
Opt2	1762.1 ± 5.0	1848.4 ± 6.5	1814.9 ± 8.3	9086.3 ± 10.9	+185.6%	351%
Opt3	960.7 ± 9.1	1811.2 ± 6.5	1792.6 ± 8.2	8853.1 ± 9.6	+18.1%	415%
Opt4	938.5 ± 6.4	1783.3 ± 6.5	1770.7 ± 11.5	8850.2 ± 11.3	+1.3%	420%

gain is more visible in case of large number of evaluations and computationally simple integrand function, such as in *Conf1*. In this situation, for a `vegasFill` kernel launch, the Nvidia Visual Profiler shows a reduction in global memory requests of 60%, and an average running time reduction from 72 ms to 18 ms. A more modest improvement is given by the parallelization of the map update and integral accumulation in *Opt4*, with a combined 1.3% improvement in total running time. Little speedup is to be expected here, since the map update time is a small fraction of the total program time.

4.7 Fill procedures

In the project, two different procedures for the `vegasFill` kernel have been considered. One, used in the *GPU percube* version, executes the kernel with a thread for every hypercube, filling for all evaluations of the hypercube. The other variant, used in the *GPU batch* version, uses a kernel with a number of threads equal to `min(batch_size,n_eval)`, and filling for all evaluations in the batch. The latter implementation requires the computation of a map that links every evaluation to the respective hypercube it belongs. The map is calculated from the stratification, and requires a sequential computation at every iteration, which is performed by the CPU and copied to GPU memory. This overhead is visible in Table 2, when comparing the performance of the *percube* version and the *batch* for *Conf1*. For this configuration, the batch version running time is almost double of the *percube*. For *Conf1*, *GPU percube* is the best performer, with an iteration speedup over the CPU sequential implementation of more than 250x. Despite the great performance in *Config1*, the running time increases dramatically in other configuration such as *Config3* and *Config4*. In fact, this version suffers from warp divergence, since the number of iterations for every thread depends on the number of evaluations of the hypercubes, which varies. As it is possible to see in Figure 3 and Figure 4, the multivariate normal and ridge integrands are highly peaked functions. This makes the algorithm stratification concentrate many evaluations in a small number of hypercubes, as showed in Figure 5, increasing the divergence effects. The figure shows the distribution of evaluations in hypercubes at stratification weights convergence, and it is possible to note for configuration 3 and 4, how the stratification adapts to the peaks of the integrand function, with the majority of the integration domain with just 2 evaluations points per hypercube. In particular, for the *GPU percube* version in *Conf1* and *Conf2*, Nvidia Visual Profiler shows a warp execution efficiency of more than 70% for all `vegasFill` kernel calls, while in *Conf4* the warp execution efficiency average is about 18%. In *Conf3* the divergence has even more evident consequences, in particular at iteration #2, where the stratification is unstable. It causes 99.98% of the hypercubes having just 2 evaluations, and assignes almost $2 \cdot 10^6$ evaluations points to a single cube, to be computed by a single thread. Warp execution efficiency for this kernel launch is 6.5%, which rapidly gets better in next iterations. This instability reflects to the performance results, where *GPU percube* shows lower performance than *CPU openMP* for *Conf3* and *Conf4*. On the other hand, in the *GPU batch* version, the number of iterations for each thread is constant (differs at most by 1 iteration), so little to none warp divergence is to be expected for the `vegasFill` kernel. In this case, performance is consistent for all test configurations.

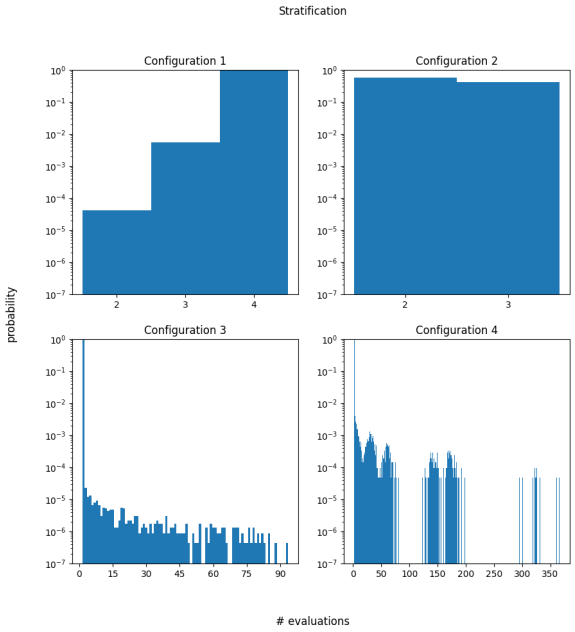

Figure 5: PMF of evaluations per hypercube, at stratification weights convergence for multiple configurations

5 CONCLUSIONS

In this project, a possible CUDA implementation of the *Vegas Enhanced* algorithm has been proposed. It has been shown how the usage of GPU computing can improve performance compared to CPU computation, by exploiting the large number of available multiprocessors. In particular, the proposed implementation has been compared to an equivalent sequential CPU version and an openMP parallel CPU version. The proposed GPU implementation shows a speedup of about 60x and 15x compared the CPU sequential and parallel implementations on the considered test system. It has been shown how fine tuning can massively reduce running time, but many variables need to be considered. In order to perform the optimizations a non trivial testing must be carried out, which considered multiple configurations, and it is crucial in order to find potentially hidden problems and to ensure consistent performance. In general, a possible approach to the parallelization of the algorithm has been presented, with a speedup over sequential computing in the order of ten to hundred times.

REFERENCES

[1] Mark Harris. 2022. Optimizing Parallel Reduction in CUDA. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[2] G. Peter Lepage. 2021. Adaptive multidimensional integration: vegas enhanced. *J. Comput. Phys.* 439 (aug 2021), 110386. https://doi.org/10.1016/j.jcp.2021.110386

[3] Peter Lepage. 2022. *gplepage/vegas: vegas version 5.3.* https://doi.org/10.5281/zenodo.7315964

[4] Nvidia. 2022. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/index.html

[5] S. Surjanovic and D. Bingham. 2022. Virtual Library of Simulation Experiments: Test Functions and Datasets. http://www.sfu.ca/~ssurjano.

[6] Emiliano Tolotti. 2022. Vegas Enhanced CUDA. https://github.com/emiliantolo/vegas-enhanced-cuda.

[7] Yongcheng Wu. 2020. CIGAR. https://github.com/ycwu1030/CIGAR.