m-Cubes: An efficient and portable implementation of Multi-Dimensional Integration for GPUs

Ioannis Sakiotis[1 [0000−0002−1988−0314]], Kamesh Arumugam[2 [0000−0002−6482−6237]], Marc Paterno[3 [0000−0003−0808−8388]], Desh Ranjan[1 [0000−0002−8298−7093]], Balša Terzić[1 [0000−0002−9646−8155]], and Mohammad Zubair[1 [0000−0002−5449−1779]]

[1] Old Dominion University, Norfolk, VA 23529, USA
[2] NVIDIA, Santa Clara, CA 95051-0952, USA
[3] Fermi National Accelerator Laboratory, Batavia, IL 60510

Abstract. The task of multi-dimensional numerical integration is frequently encountered in physics and other scientific fields, e.g., in modeling the effects of systematic uncertainties in physical systems and in Bayesian parameter estimation. Multi-dimensional integration is often time-prohibitive on CPUs. Efficient implementation on many-core architectures is challenging as the workload across the integration space cannot be predicted a priori. We propose m-Cubes, a novel implementation of the well-known Vegas algorithm for execution on GPUs. Vegas transforms integration variables followed by calculation of a Monte Carlo integral estimate using adaptive partitioning of the resulting space. m-Cubes improves performance on GPUs by maintaining relatively uniform workload across the processors. As a result, our optimized Cuda implementation for Nvidia GPUs outperforms parallelization approaches proposed in past literature. We further demonstrate the efficiency of m-Cubes by evaluating a six-dimensional integral from a cosmology application, achieving significant speedup and greater precision than the Cuba library's CPU implementation of Vegas. We also evaluate m-Cubes on a standard integrand test suite. m-Cubes outperforms the serial implementations of the Cuba and Gsl libraries by orders of magnitude speedup while maintaining comparable accuracy. Our approach yields a speedup of at least 10 when compared against publicly available Monte Carlo based GPU implementations. In summary, m-Cubes can solve integrals that are prohibitively expensive using standard libraries and custom implementations. A modern C++ interface header-only implementation makes m-Cubes portable, allowing its utilization in complicated pipelines with easy to define stateful integrals. Compatibility with non-Nvidia GPUs is achieved with our initial implementation of m-Cubes using the Kokkos framework.

1 Introduction

The task of multi-dimensional numerical integration is often encountered in physics and other scientific fields, e.g., in modeling the effects of systematic

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

uncertainties in physical systems and Bayesian parameter estimation. However, multi-dimensional integration is time-prohibitive on CPUs. The emerging high-performance architectures that utilize accelerators such as GPUs can speed up the multi-dimensional integration computation. The GPU device is best suited for computations that can be executed concurrently on multiple data elements. In general, a computation is partitioned into thousands of fine-grained operations, which are assigned to thousands of threads on a GPU device for parallel execution.

A naive way to parallelize the multi-dimensional integration computation is as follows: divide the integration region into "many" (m) smaller sub-regions, estimate the integral in each sub-region individually (I_i) and simply add these estimates to get an estimate for the integral over the entire region ($\Sigma_{i=1}^{m} I_i$). The integral estimate in each sub-region can be computed using any of the traditional techniques such as quadrature or Monte Carlo based algorithms. If we use a simple way of creating the sub-regions, e.g., via dividing each dimension into g equal parts, the boundaries of the sub-regions are easy to calculate, and the estimation of the integral in different sub-regions can be carried out in an "embarrassingly parallel" fashion. Unfortunately, this approach is infeasible for higher dimensions as the number of sub-regions grows exponentially with the number of dimensions d. For example if $d = 10$ and we need to split each dimension into $g = 20$ parts the number of sub-regions created would be $g^d = 20^{10}$ which is roughly 10^{13}. Moreover, uniform division of the integration region is not the best way to estimate the integral. The intuition is that the regions where the integrand is "well-behaved" do not need to be sub-divided finely to get a good estimate of the integral. Regions where it is "ill-behaved" (e.g. sharp peaks, many oscillations) require finer sub-division for a reliable, accurate estimate. However, when devising a general numerical integration method, we cannot assume knowledge of the behavior of the integrand. Hence, we cannot split up the integration region in advance with fewer (but perhaps larger in volume) sub-regions where the integrand is "well-behaved" and a greater number of smaller sub-regions in the region where it is "ill-behaved'. To summarize, efficient implementation of multi-dimensional integration on many-core architectures such as GPUs is challenging due to two reasons: (i) increase in computational complexity as the dimension of the integration space increases, and (ii) the workload across the integration space cannot be predicted.

The first challenge, "curse of dimensionality", can be addressed to some extent by using a Monte Carlo based algorithm for multi-dimensional integration, as the convergence rate of such methods is independent of the dimension d. The convergence rate can sometimes be further improved by utilizing low-discrepancy sequences (Quasi-Monte Carlo) instead of pseudo-random samples [1] [6]. When utilizing Monte Carlo based approaches, the second challenge of consolidating sampling efforts on the "ill-behaved' areas of the integration space, is addressed through "stratified" and/or "importance" sampling, which aim to reduce the variance of the random samples. Stratified sampling involves sampling from disjoint partitions of the integration space, the boundaries of which can be refined

recursively in a manner similar to adaptive quadrature. Importance sampling integration methods, use Monte Carlo samples to approximate behavior of the integrand in order to sample from a distribution which would significantly reduce the variance and accelerate convergence rates. This is accomplished by an initially uniform weight function that is refined across iterations, and results in more samples in the location where the magnitude of the integrand is either large or varies significantly.

The sequential VEGAS algorithm is the most popular Monte Carlo method that makes use of importance sampling [8] [9]. There are several implementations and variants, including Python packages, C++-based implementations in the CUBA and GSL libraries, and the R Cubature package. Unfortunately, while VEGAS can often outperform standard Monte Carlo and deterministic techniques, sequential execution often leads to prohibitively long computation times. A GPU implementation of the original VEGAS algorithm was proposed in [7], but is not packaged as a library and the original implementation is not publicly available. VegasFlow is a Python library based on the TensorFlow framework, providing access to VEGAS and standard Monte Carlo implementations that can execute on both single and multi-GPU systems [2] [3]. Another Python package with support for GPU execution was proposed in [12], incorporating stratified sampling and a heuristic tree search algorithm. All GPU implementations demonstrate significant speedup over serial versions of VEGAS.

We propose m-CUBES, a novel implementation of the well-known VEGAS algorithm for multi-dimensional integration on GPUs. m-CUBES exploits parallelism afforded by GPUs in a way that avoids the potential non-uniform distribution of workload and makes near-optimal use of the hardware resources. Our implementation also modifies VEGAS to make the computation even faster for functions that are "fully symmetric". Our approach demonstrates improved performance, yielding orders-of-magnitude speedup over the CPU implementations and a speedup of 10 when compared against the method presented in [12]. Our goal is to make publicly available a robust and easy-to-use, GPU implementation of VEGAS that will be suitable for the execution of challenging integrands that occur in physics and other scientific fields.

The remainder of the paper is structured as follows. In section II, we describe various Monte Carlo based algorithms. In section III, we describe the VEGAS algorithm. In section IV we describe the m-CUBES algorithm. In section V, we discuss the accuracy and performance of our implementation, comparing its execution time against publicly available Monte Carlo based methods. In section VI we discuss the interface and portability features of m-CUBES and present the results of utilizing those features on an complex integral utilized in parameter estimation in cosmological models of galaxy clusters.

2 Background

We summarize here the previous work related to our research. We first summarize the previously developed sequential Monte Carlo Methods and libraries. Thereafter we summarize the research on parallel VEGAS based methods.

2.1 Monte Carlo Methods

The GSL library provides three Monte Carlo based methods, standard Monte Carlo, MISER, and VEGAS. Standard Monte Carlo iteratively samples the integration space of volume V, at T random points x_i to generate an integral estimate in the form of $\frac{V}{T}\sum_{t=i}^{T} f(x_i)$, whose error-estimate is represented by the standard deviation.

VEGAS is an iterative Monte Carlo based method that utilizes stratified sampling along with importance sampling to reduce standard Monte Carlo variance and accelerate convergence rates. The stratified sampling is done by partitioning the d-dimensional space into sub-cubes and computing Monte Carlo estimates in each sub-cube. For importance sampling, VEGAS samples from a probability distribution that is progressively refined among iterations to approximate the target-integrand. VEGAS uses a piece-wise weight function to model the probability distribution, where the weight corresponds to the magnitude of the integral contribution of each particular partition in the integration space. At each iteration VEGAS adjusts the weights and the corresponding boundaries in the integration space, based on a histogram of weights. The piece-wise weight-function is intentionally separable to keep the number of required bins small even on high-dimensional integrands. Existing implementations of VEGAS, are also found within the CUBA and GSL libraries. A Python package also exists, with support for parallelization through multiple processors.

MISER is another Monte Carlo based method, which utilizes recursive stratified sampling until reaching a user-specified recursion-depth, at which point standard Monte Carlo is used on each of the generated sub-regions. MISER generates sub-regions by sub-dividing regions on a single coordinate-axis and redistributing the number of sampling points dedicated to each partition in order to minimize their combined variance. The variance in each sub-region is estimated at each step with a small fraction of the total points per step. The axis to split for each sub-region, is determined based on which partition/point-redistribution will yield the smallest combined variance.

CUBA is another library that provides numerous Monte Carlo based methods (VEGAS, Suave, Divonne). Suave utilizes importance sampling similar to VEGAS but further utilizes recursive sub-division of the sub-regions like MISER in GSL. The algorithm first samples the integration space based on a separable weight function (mirroring VEGAS) and then partitions the integration space in two similar to MISER. Suave then selects the sub-region with the highest error for further sampling and partitioning. This method requires more memory than both VEGAS and MISER.

Divonne uses stratified sampling, attempting to partition regions such that they have equal difference between their maximum and minimum integrand values. It utilizes numerical optimization techniques to find those minimum/maximum values. Divonne can be faster than Vegas/Suave on many integrands while also providing non-statistically based error-estimates if quadrature rules are used instead of random samples.

2.2 Parallel Methods

The *g*Vegas method, is a Cuda implementation of Vegas that reported a speedup of 50 compared to CPU execution [7]. This method parallelizes the computation over the sub-cubes used in Vegas for stratification. It uses an equal number of samples in each sub-cube as proposed in the original Vegas algorithm. It assigns a single thread to process each sub-cube, which is not very efficient and is discussed in Section IV. Additionally, the importance sampling that requires keeping track of integral values in each bin (explained in the next section) is done on the CPU which slows down the overall computation.

ZMCintegral is the only publicly available library for performing Monte Carlo based multi-dimensional numerical integration on GPU platforms, with support for multiple-GPUs. The algorithm uses stratified sampling in addition to a heuristic tree search that applies Monte Carlo computations on different partitions of the integration space [12].

3 The Vegas Algorithm

Vegas is one of the most popular Monte Carlo based methods. It is an iterative algorithm, that attempts to approximate where the integrand varies the most with a separable function that is refined across iterations. The main steps of a Vegas iteration are listed in Algorithm 1. The input consists of an integrand f which is of some dimensionality d, the number of bins n_b on each dimensional axis, the number of samples p per sub-cube, the bin boundaries stored in an d-dimensional list B, and the d-dimensional list C which contains the contributions of each bin to the cumulative integral estimate.

Initially the integration space is sub-divided to a number of d-dimensional hyper-cubes, which we refer to as sub-cubes. Vegas processes each sub-cube independently with a for-loop at line 2. At each sub-cube, the algorithm generates an equal number of samples[4], which are processed through the for-loop at line 3. To process a sample, Vegas generates d random numbers in the range $(0, 1)$ at line 4, corresponding to one point per dimensional-axis. Then at line 5, we transform the point y from the domain of the unit hyper-cube $(0, 1)$ to actual coordinates in the integration space. At line 6, we evaluate the integrand f at the transformed point x, yielding the value v which contributes to the cumulative

[4] Here, we focus on the original Vegas algorithm which uses equal number of samples in each sub-cube. The later versions of the algorithm deploy adaptive stratification that adjust the number of integral estimates used in each sub-cube.

integral estimate. Before proceeding to the next sample, we identify at line 7 the bins that encompass each of the d coordinates in x. We use the indices of those bins ($b[1:d]$) to increment their contribution (v) to the integral at line 8. Once the samples from all sub-cubes have been processed, we exit the twice-nested for-loop. At line 9, we use the bin contributions stored in d, to adjust the bin boundaries B in such a way that bins of large contributions are smaller. This approach results in many small bins in the areas of the integration space where the integrand is largest or varies significantly, resulting in more samples being generated from those highly contributing bins. Finally, the contribution of each sample must be accumulated at line 10, to produce the Monte Carlo integral estimate and to compute the variance for the iteration.

The most desirable features of VEGAS are its "importance sampling" which occurs by maintaining bin contributions and adjusting the bin boundaries at the end of each iteration. The use of sub-cubes introduces "stratified sampling" which can further reduce the variance of the Monte Carlo samples. Those two variance reduction techniques make VEGAS successful in many practical cases and the independence of the sub-cubes and samples make the algorithm extremely parallelizable.

Algorithm 1 VEGAS

1: **procedure** VEGAS(f, d, n_b, p, B, C) ▷ Each iteration consists of the steps below
2: **for** all sub-cubes **do**
3: **for** $i \leftarrow 1$ to p **do** ▷ f is evaluated at p points in each sub-cube
4: $y_1, y_2, ..., y_d \leftarrow$ generate d points in range $(0, 1)$ uniformly at random
5: $x_1, x_2, ..., x_d \leftarrow$ map vector y to vector x ▷ f is evaluated at x
6: $v \leftarrow f(x_1, x_2, ..., x_d)$
7: let b_i denote the index of the bin to which x_i belongs in dimension i
8: increment $C[1][b_1]$, $C[2][b_2]$, ..., $C[d][b_d]$ by v ▷ Store bin contributions
9: $B[1:d][1:n_b] \leftarrow$ adjust all bin boundaries based on $C[1:d][1:n_b]$
10: $I, E \leftarrow$ compute integral estimate/variance by accumulating v
11: **return** I, E

4 The Algorithm m-CUBES

The main challenges of parallel numerical integrators are the "curse of dimensionality" and workload imbalances along the integration space. While high-dimensionality is made manageable by the use of the Monte Carlo estimate in VEGAS, workload imbalances need to be addressed. This is particularly true for newer variations of the VEGAS algorithm, which involve a non-uniform number of samples per sub-cube. Parallelization of VEGAS poses additional challenges from the need to accumulate the results of multiple samples from different processors. In Algorithm 1, line 10 involves such an accumulation which requires processor

synchronization. Furthermore, a race condition can occur at line 8, where the contributions of a bin may need to be updated by different processors.

To parallelize VEGAS, m-CUBES assigns a batch of sub-cubes to each processor and generates a uniform number of samples per sub-cube. This solves the work-load imbalance issue and further limits the cost of accumulating results from various processors. The integrand contributions from all sub-cubes of each processor (Algorithm 1,line 6), are processed serially. As a result, those values can be accumulated in a single local variable, instead of synchronizing and transferring among processors. This does not eliminate the cost of accumulation, as we still need to collect the contributions from the sub-cube batches in each processor at line 10, but the extent of the required synchronization is reduced significantly.

Algorithm 2 m-CUBES

```
1: procedure m-Cubes(f, d, n_b, maxcalls, L, H, itmax, ita, r)
2:     I, E ← 0                                          ▷ Integral/Error estimate
3:     g ← (maxcalls/2)^{1/d}                            ▷ Number of intervals per axis
4:     m ← g^d                                           ▷ Number of cubes
5:     s ← Set-Batch-Size(maxcalls)                      ▷ Heuristic
6:     B[1 : d][1 : n_b] ← Init-Bins(d, n_b)             ▷ Initialize bin boundaries
7:     C[1 : d][n_b] ← 0                                 ▷ Bin contributions
8:     p ← maxcalls/m                                    ▷ number of samples per cube
9:     for i ← 0 to ita do
10:        r, C ← V-Sample()
11:        I, E ← Weighted-Estimates(r)
12:        B ← Adjust-Bin-Bounds(B, C)
13:        Check-Convergence()
14:    for i ← ita to itmax do
15:        r ← V-Sample-No-Adjust()
16:        I, E ← Weighted-Estimate(r)
17:        Check-Convergence()
```

The input of the m-CUBES algorithm consists of the integrand f and its dimensionality d, the number of bins per coordinate axis n_b, the maximum number of allowed integrand evaluations $maxcalls$, and the upper/lower boundaries of the integration space in each dimension, represented in the form of two arrays L, H. The user must also supply the required number of iterations $itmax$ and the number of iterations that will involve bin adjustments (ita). We also use the array r to store the results which consist of the integral estimate and standard deviation.

In line 2, we initialize the cumulative integral estimate and error-estimate (standard deviation) to zero. In line 3 we compute the number of intervals per axis; the boundaries of the resulting sub-cubes remain constant for the duration of the algorithm. In contrast, the bin boundaries B are adjusted across iterations. At line 4 we determine the number of sub-cubes m, while we also compute

the batch size s, referring to the number of sub-cubes that each thread will process iteratively. Then the bin boundaries are generated on line 6, by equally partitioning each axis into n_b bins, and storing their right boundaries in the list B.

Then we proceed with the m-Cubes iterations. The first step is to compute the result r and bin contributions C, by executing the V-Sample method at line 10. V-Sample produces the Monte Carlo samples, evaluates the integrals and updates the bin contributions. This method requires almost all data-structures and variables as parameters, so we omit them in this description. At line 11, the estimates are weighted by standard Vegas formulas that but can be found in equations 5 and 6 of [9]. We then adjust the bin boundaries B based on the bin contributions C. If the weighted integral estimate and standard deviation produced at line 11, satisfy the user's accuracy requirements, execution stops, otherwise we proceed to the next iteration. Before proceeding to the next iteration, the bin boundaries B are adjusted at line 12. The only difference between an m-Cubes and a Vegas iteration from the original algorithm, are the parallelized accumulation steps and mappings between processors and sub-cubes.

A second loop of iterations (lines 14 to 17) is invoked once *ita* iterations are completed. In this set of iterations, we perform the same computations with the exception of bin adjustments and their supporting computations which are omitted. This distinction is introduced due to the common occurrence of the boundaries B converging after a number of iterations and remaining unchanged. In those cases, the costly operations of keeping track of bin contributions and updating them has no positive effect. As such, the user can mandate a limit of iterations with that will involve bin adjustments, and sub-subsequent iterations will execute faster by avoiding redundant operations.

4.1 The Procedure V-Sample

V-Sample and V-Sample-No-Adjust are the only methods that involve parallelization, encompassing the functionality of lines 2 to 8 from Algorithm 1. To facilitate the accumulation steps needed to yield the integral contributions from multiple sub-cube batches, V-Sample utilizes hierarchical parallelism, where each processor launches parallel 128-sized thread-blocks, requiring a total $\frac{m}{128}$ such blocks. Each thread within a block is independent and processes its own sub-cube batch of size s (see Algorithm 2, line 5). The benefit of this approach, is that block-shared memory and block-synchronization capabilities allow for more efficient accumulation of the integral estimates v local to each thread. The race condition involved with incrementing the bin contributions from multiple threads, is solved through atomic addition operations. The same operation is used to accumulate the integral estimate from all thread-blocks.

The input of the V-Sample method, consists of the integrand f of dimensionality d, the number of sub-cubes m, sub-cube batch size s, number of samples per sub-cube p, bin bounds B, bin contributions C, and result r. Once finished, V-Sample will return an estimate for the integral, variance, and updated bin contributions C.

The for-loop at line 2, indicates the sequential processing of s sub-cubes from each thread. At line 3 we initialize a random number generator. Each thread has local integral and error estimates I and E (line 4) respectively, which encompass the contributions from all s sub-cubes. Each thread processes its assigned sub-cubes with the serial for-loop at line 5. As the sub-cubes are processed, the local estimates I_t and E_t of each sub-cube are accumulated in I and E. This involves yet another for-loop at line 7, to serialize the p samples generated per sub-cube. Similar to the accumulation of I_t to I, we accumulate the estimates I_k and E_k (local to the sample) to I_t and E_t.

For each sample, we generate an d-dimensional point x where we will evaluate the integrand f. This yields estimates for the sample that are used to increment the sub-cubes estimates at lines 10 and 11. Then, based on the bin IDs that are determined in line 12. we update the bin contributions in line 14. The atomic addition guarantees serial access for each thread updating C at each index $b[1:s]$, avoiding race conditions. The actual bin-contribution is the square of the integral estimate I_k. Then, we update the variance at line 16, followed by the updating of the thread-local estimates for the entire batch of sub-cubes in lines 16 and 17.

Once the for-loop at line 5 is finished, we accumulate the I, E from each thread in parallel. This is accomplished by a block-reduction that utilizes shared memory and warp-level primitives. Finally, once each thread-block has accumulated estimates from all its sub-cubes across all its threads, a final atomic addition in lines 23 and 24 accumulates the estimates from all thread-blocks and can return them as the result r.

The V-Sample-No-Adjust method is almost identical to V-Sample, with the distinction that the loop at lines 13-14 are not needed which yields a boost in performance.

5 Experimental Results

We conducted experiments with a standard integrand test suite which consists of several integrals with different characteristics such as corner/product peaks, Gaussian, C^0 form, and oscillations. We used m-Cubes, the Vegas implementations of the Gsl and Cuba libraries, as well as the Vegas 5.0 MPI-aware Python package, to evaluate the following integrals.

$$f_{1,d}(x) = \cos\left(\sum_{i=1}^{d} i\, x_i\right) \tag{1}$$

$$f_{2,d}(x) = \prod_{i=1}^{d} \left(\frac{1}{50^2} + (x_i - 1/2)^2\right)^{-1} \tag{2}$$

$$f_{3,d}(x) = \left(1 + \sum_{i=1}^{d} i\, x_i\right)^{-d-1} \tag{3}$$

Algorithm 3 V-SAMPLE

```
1: procedure V-SAMPLE(f, d, m, s, p, B, C, r)
2:     for m/b threads parallel do
3:         SET-RANDOM-GENERATOR(seed)
4:         I, E ← 0                                   ▷ cumulative estimates of thread
5:         for t = 0 to s do
6:             I_t, E_t ← 0                           ▷ estimates of sub-cube t
7:             for k ← 1 to p do
8:                 x[1 : d] ← GENERATE()
9:                 I_k, E_k ← EVALUATE(f, x)
10:                I_t ← I_t + I_k                    ▷ Accumulate sub-cube contributions
11:                E_t ← E_t + E_k
12:                b[1 : d] ← GET-BIN-ID(x)
13:                for j ← 1 to d do                  ▷ Store bin contributions
14:                    ATOMICADD(C[b[j]], I_k^2)
15:            E_t ← UPDATEVARIANCE(E_t, I_t, p)
16:            I ← I + I_t                            ▷ update cumulative values
17:            E ← E + E_t
18:        I ← BLOCKREDUCE(I)
19:        E ← BLOCKREDUCE(E)
20:        if thread 0 within Block then
21:            ATOMICADD(r[0], I)
22:            ATOMICADD(r[1], E)
```

$$f_{4,d}(x) = \exp\left(-625\sum_{i=1}^{d}(x_i - 1/2)^2\right) \tag{4}$$

$$f_{5,d}(x) = \exp\left(-10\sum_{i=1}^{d}|x_i - 1/2|\right) \tag{5}$$

$$f_{6,d}(x) = \begin{cases} \exp\left(\sum_{i=1}^{d}(i+4)\,x_i\right) & \text{if } x_i < (3+i)/10 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

We evaluated the integrands for various values of dimension d and different levels of user-specified relative error tolerance τ_{rel} in the range $(10^{-3}, 10^{-9})$. The algorithm reports convergence to sufficiently accurate results when the estimated relative error is smaller or equal to τ_{rel}. It is worth noting that the Python package does not accept τ_{rel} as input. Instead, we configured the two parameters: *neval*, *nitn*, until achieving the equivalent relative errors targeted by the other implementations. We conducted all experiments on a node with a 2.4GHz Xeon R Gold 6130 CPU and v100 GPU with 16GB of memory and 7.834 Tflops in double precision floating point arithmetic. The two C++ libraries (CUBA, GSL), were compiled with GCC 9.3.1, while the parallel implementations were compiled with CUDA 11.

5.1 Accuracy

To our knowledge no numerical integration algorithm can claim a zero absolute error on all integrands or even guarantee integral/error estimates that satisfy the various τ_{rel}. As such, it is important to evaluate the degree of correctness for specific challenging integrands whose integral values are known *a priori*. It is equally important to demonstrate how an algorithm adapts to increasingly more demanding precision requirements and whether the yielded integral/error estimates truly satisfy the user's relative error tolerances. This is especially true for Monte Carlo based algorithms, whose randomness and statistically-based error-estimates make them less robust than deterministic, quadrature-based algorithms. In our evaluation of m-CUBES, we adopt the testing procedures of [5] in selecting the target integrands but preselect the various integrand parameter constants as in [10]. We deviate from [10], in that we omit the two box-integrands that were not challenging for VEGAS. We also do not report results on $f_{1,d}$ in our plots, as no VEGAS variant could evaluate it to the satisfactory precision levels.

In the m-CUBES algorithm, we use relative error as a stopping criteria for achieving a specified accuracy, which is the normalized standard deviation (see Algorithm 3 for the error computation). As such, we investigate the quality of the m-CUBES error-estimates in Figure 1, where we display multiple 100-run sets of results on each different level of precision for each integrand. The user's requested digits of precision are represented in the x-axis, while the true relative error is mapped to the y-axis. To make our plots more intuitive, with the x-axis representing increasing accuracy requirements, we perform the $-log_{10}(\tau_{rel})$ transformation on the x-axis of all plots; this translates "roughly" to the required digits-of-precision. We still plot the user's τ_{rel} for each experiment as the orange point. Since we only plot results for which m-CUBES claimed convergence with appropriately small χ^2, comparing against the orange point indicates whether the algorithm is as accurate as it claims.

Due to the randomness of the Monte Carlo samples, there is a wide range of achieved relative error values for the same digits-of-precision. This is to be expected as the error-estimate is interpreted as the standard deviation of the weighted iteration results. Deviation in the results can be more pronounced when generating smaller number of samples which is typical in low-precision runs. In most cases, the number of samples must be increased for higher precisions runs. This leads to a smaller deviation in the results, demonstrated in the figure by the increasingly smaller boxes on the right side of the x-axis. This smaller deviation yields improved accuracy, as we observe the box boundaries encompassing the target relative error. We observed similar behavior from GSL, CUBA, and the VEGAS 5.0 Python package on which we performed single-run experiments.

5.2 Performance

CUBA was generally the fastest method on a CPU platform, while the 32-processor Python execution was typically the slowest. In cases where the CPU

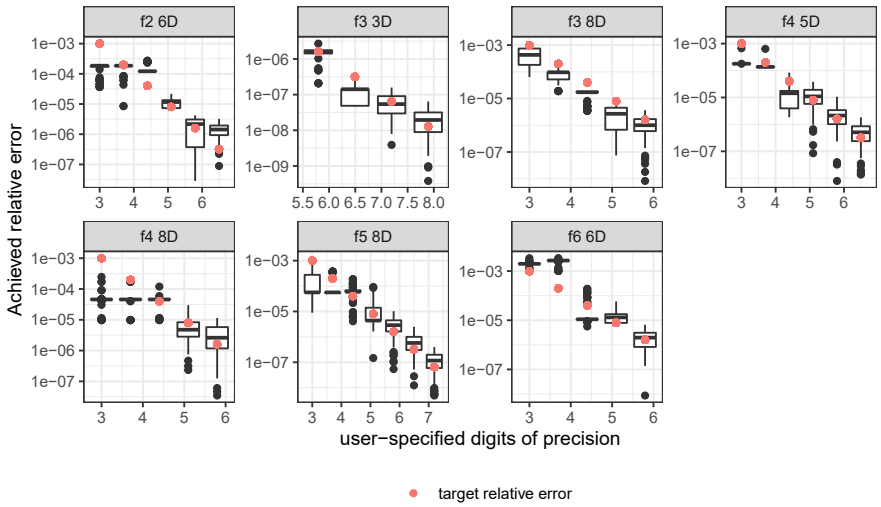

Fig. 1: This box plot displays the user-requested relative error tolerance (orange dot) and the achieved relative errors of m-Cubes algorithm on the y-axis. Each box is a statistical summary of 100 runs. The top and bottom box boundaries indicate the first and third quartiles. The middle line is the value of the median while the vertical lines protruding from the top and bottom box boundaries indicate the minimum and maximum values. The individual points displayed are outliers.

methods would fail to converge on high-precision runs, m-Cubes was successful due to the greater computational capabilities of the GPU. This is evident in Figure 2, where for most integrands, no CPU-method reaches the same digits-of-precision as m-Cubes. We can reach three more digits of precision on the integrand $f_{5,8}$ when using m-Cubes. The remaining integrands display similar behavior with the CPU methods reaching at most one fewer digit of precision than m-Cubes. Furthermore, m-Cubes is orders of magnitude faster than the CPU methods, which is better in illustrated in Figure 3. We also report the execution times of runs with three digits of precision in Table 1, where missing entries indicate that the corresponding algorithm did not convergence to the required τ_{rel}.

m-Cubes generates the random numbers and evaluates the integrand within two GPU kernels, V-Sample which additionally stores bin contributions in order to better approximate the distribution of the integrand, and V-Sample-No-Adjust which does not update bin contributions. The execution time of the two kernels, is directly dependent on the number of required function calls per iteration which in turn determines the workload (number of sub-cubes) assigned to each thread. The required number of iterations tends to increase for higher precision runs. For low-precision runs, the same number of samples and iterations

can result in convergence. This is why for some integrands ($f_{4,8}$, $f_{5,8}$, $f_{3,3}$, $f_{2,6}$), the three, four, and five digits of precision runs display similar execution time.

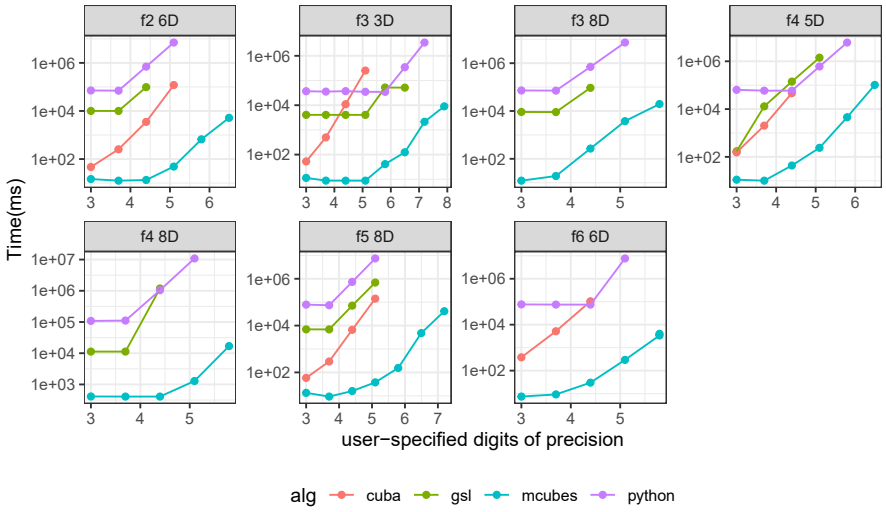

Fig. 2: Execution Times

Table 1: Execution time in milliseconds for integrands on τ_{rel} of .001

id	m-Cubes	Gsl	Cuba	Python
$f_{2,6}$	14.7	1.00×10^4	46.63	7.12×10^4
$f_{3,3}$	11.4	4.06×10^3	52.79	3.69×10^4
$f_{3,8}$	12.2	9.12×10^3		7.23×10^4
$f_{4,5}$	11.1	1.72×10^2	154.11	6.40×10^4
$f_{4,8}$	411.8	1.13×10^4		1.08×10^5
$f_{5,8}$	13.4	6.90×10^3	58.38	7.85×10^4
$f_{6,6}$	7.4		378.90	7.63×10^4

5.3 Comparison with GPU method

To our knowledge, the only publicly available GPU-compatible Monte Carlo based method, is ZMCintegral. In [12], ZMCintegral was compared against the gVegas GPU implementation presented in [12] and reported significant speedup. Unfortunately, the location of the gVegas implementation listed in [7] is no longer publicly available. As such, we cannot perform a comparison with m-Cubes. Nonetheless, the gVegas implementation was reporting a speedup of 50 over serial Vegas. As gVegas was developed during the initial years of the Cuda platform, several design choices are outdated. We expect m-Cubes to

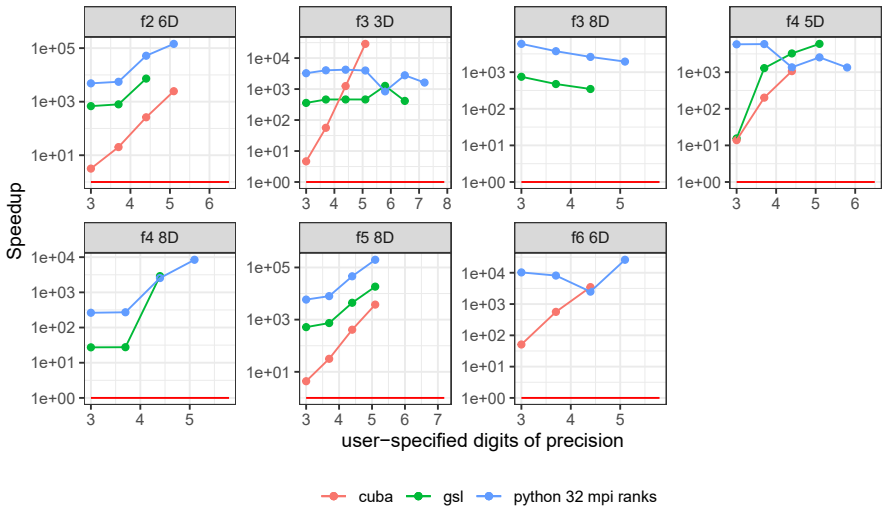

Fig. 3: Achieved speedup over CPU methods.

be consistently faster and we demonstrate as such by reporting speedup over ZMCintegral, which was in turn reporting speedup over gVEGAS.

$$f_A(x) = \sin\left(\sum_{i=1}^{6} x_i\right) \tag{7}$$

$$f_B(x) = \frac{1}{(\sqrt{(2\cdot\pi\cdot.01)})^2} \exp\left(-\frac{1}{2\cdot(.001)^2}\sum_{i=1}^{9}(x_i)^2\right) \tag{8}$$

To compare against ZMCintegral, we executed both methods with the same parameters on the same integrals reported in [12]. The f_A integrand was evaluated over the range $(0, 10)$ on all dimensions, while the integration space of f_B was the range $(-1, 1)$ on all axes. Since ZMCintegral does not accept τ_{rel} as parameter, we try to match the achieved standard deviation of ZMCintegral for a fair comparison. We report our results in Table 2, where we observe a speedup of 45 and 10 respectively, though in both cases m-CUBES reported smaller error-estimates than ZMCintegral. For this series of experiments, we do not report results on the $f(1:6)$ integrals on the grounds of unfair comparison since results on these integrals were not reported in [12]. Our experiments showed execution times slower than serial VEGAS reported but we were not aware of the "best"configuration parameters for ZMCintegral.

5.4 The m-CUBES1D VARIANT

In addition to the m-CUBES algorithm, we also provide the variant m-CUBES1D. m-CUBES1D mirrors m-CUBES, with the distinction that the bin boundaries

Table 2: Comparson with ZMCintegral

integrand	alg	true value	estimate	errorest	time (ms)
f_A	zmc	-49.165073	-48.64740	1.98669	4.75×10^4
f_A	m-CUBES		-49.27284	1.19551	1.07×10^3
f_B	zmc	1.0	0.99939	0.00133	8.30×10^3
f_B	m-CUBES		1.00008	0.00005	9.80×10^2

being updated at line 15 in Algorithm 3, are identical on all coordinate axes, thus not requiring the for-loop at line 14. This is beneficial when the integrand f is fully symmetrical, having the same density across each dimension. Thus, one series of atomic additions are required for dimension $j = 0$ at line 15. When the bins are then adjusted sequentially after the execution of the V-SAMPLE method, the bins at each dimension will have identical boundaries.

Three of the six integrals presented in section IV, are symmetrical. We performed comparisons between m-CUBES and m-CUBES1D, which demonstrate a small performance boost in m-CUBES1D. In Figure 4, we see speedup of various magnitudes depending on the integrand and degree of precision. Theoretically, both implementations would perform the same bin-adjustments on a symmetrical integrand, and m-CUBES1D would require fewer computations for the same effect. We expect that execution of this variant on non-symmetrical integrands, will severely hinder the bin adjustments.

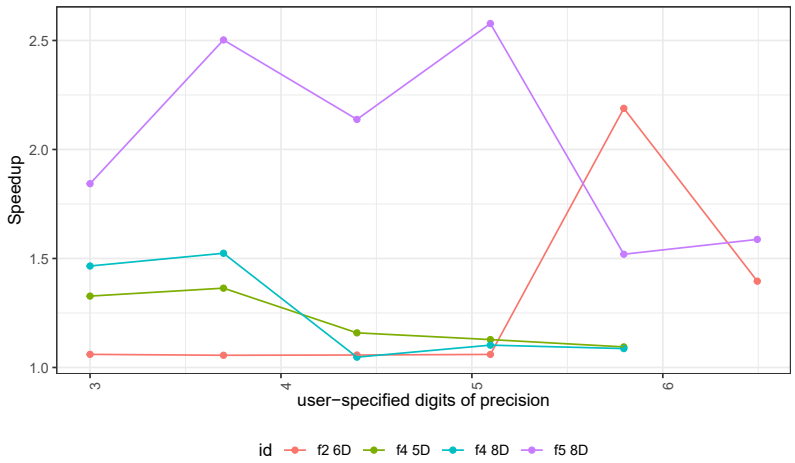

Fig. 4: Speedup of m-CUBES1D over m-CUBES on symmetrical integrands.

6 Portability

There are several challenges related to portability. Ideally, an integrator should be "easy" in incorporate into existing codes and the integrand definitions should be suitable for execution on various platforms, whether that is CPUs or GPUs

regardless of architecture (Nvidia, Intel, Amd, etc.) Additionally, a user should have minimum restrictions when defining the integrand, being allowed to use dynamically created data-structures within the integrand, maintain an integrand state (persistent variables, tabular data, etc.), and define boundaries in the integration space.

The different memory spaces utilized by a GPU pose a challenge in regards to defining integrands with complex states (non-trivial structures). While the user could potentially interface with m-Cubes through the appropriate use of Cuda to handle the different memory spaces, this would severely hinder its ease-of-use and require sufficient knowledge of GPU programming. Additionally, a user who wishes to maintain the option of which platform (CPU, GPU) to execute on, would be forced to write multiple, potentially very different implementations of the same integrand to accommodate the requirements of each platform. To solve this problem, we require the user to define an integrand as a functor to interface with m-Cubes. We also supply our own data-structures such as interpolator objects and array-like structures, that handle the GPU related data manipulations internally, but are set and accessed similar to standard library or Gsl equivalents. This allows the user to initialize such objects and structures in a familiar fashion and use them in their defined integrands without having to worry about allocating and transferring data to GPU memory and without having to write any complicated Cuda code. Finally, in regards to "easily" using integrators in existing code-bases, m-Cubes is implemented as a header-only library.

A use-case demonstrating these features, involves an integrand required for a cosmological study in an astrophysics application. The integrand is six dimensional and requires the utilization of numerous interpolation tables that must be read at run-time and consists of several C++ objects. We evaluate that integrand and compare execution times against the Vegas implementation of Cuba. The results are summarized in Table 3. m-Cubes achieves speedup of 40 and 70 for three and four digits of precision, and reaches one more digit of precision than Cuba. The achieved speedup demonstrates that the overhead of our solutions pertaining to portability does not induce any prohibitive costs.

Table 3: Cosmology integrand results

alg	prec	estimate	errorest	time (ms)
Vegas	3.00	11815.68	11.73	1437
Vegas	4.00	11804.69	1.18	135696
m-Cubes	3.00	11789.72	10.69	35
m-Cubes	4.00	11802.45	1.13	1904
m-Cubes	5.00	11802.88	0.12	7384

6.1 Kokkos implementation

Kokkos is a C++ library that implements an abstract thread parallel programming model which enables writing performance portable applications for major multi- and many-core HPC platforms. It exposes a single programming interface and allows the use of different optimizations for backends such as Cuda,

HIP, SYCL, HPX, OpenMP, and C++ threads. [11], [4]. This was previously infeasible with a CUDA implementation, which is only suitable for execution on NVIDIA GPUs. Parameter tuning in the parallel dispatch of code-segments, allows for both automatic and manual adjustments in order to exploit certain architectural features.

We have completed an initial implementation of m-CUBES in Kokkos with minimal algorithmic changes to the original CUDA version. The hierarchical parallelism constructs of Kokkos, allow the specification of the same thread-block configuration as required by CUDA kernels. This makes "translation" to Kokkos easy to perform but further optimization is required to maintain performance across architectures. Additionally, certain architecture-specific features from the CUDA version could not be ported to Kokkos. For example, the block reduction in lines 18 and 19 of Algorithm 3, requires data exchange between threads in order to sum all individual values. In CUDA, this data exchange can be conducted directly between registers through the use of warp-level primitives that work only on NVIDIA architectures. This offers improved performance over alternatives that accumulate sums by using a shared memory model.Note that in our Kokkos implementation, we have used the shared memory for accumulation.

We present results on the f_A and f_B integrands in Figure 4, which displays the kernel time (time executing on GPU) and total time (CPU and GPU time). We evaluated both integrands with the Kokkos version, for three digits of precision on an NVIDIA V100 GPU. This demonstrates the minimum expected overhead and impact of lacking features that to our knowledge cannot be adopted in Kokkos. We observe an overhead in the range 10-15% in the parallel segments of the code, which are expected to cover the majority of execution time. We note that Kokkos can in some cases be faster on the serial code execution. This leads to the low-precision runs on the two integrands being slightly faster in Kokkos. Additional experiments on other integrands show that this is not the case when computational intensity increases. For example, when we compare the running times for the integrand $f_{4,5}$ with 100 runs for each precision level, Kokkos incurs 20-50% overhead.

Table 4: Kokkos and CUDA m-CUBES Execution Time (ms)

(a) Execution Time (ms) on f_A

platform	kernel	total
CUDA	829.760	1280.318
Kokkos	968.880	1001.035

(b) Execution Time (ms) on f_B

platform	kernel	total
CUDA	664.977	1126.529
Kokkos	726.766	767.343

7 Conclusion

We presented m-CUBES, a new parallel CUDA implementation of the widely used VEGAS multi-dimensional numerical integration algorithm for execution on GPUs. m-Cubes is a portable header-only library, with a modern interface

and features that allow easy interfacing and requires no knowledge of GPU programming to use. We also supply infrastructure to facilitate the definition of complex and stateful integrands. Our experiments on a standard set of challenging integrals and a complex stateful integrand consisted of numerous C++ objects, demonstrate orders of magnitude speedup over existing implementations on both the CPU and GPU. Furthermore, We supply the variant m-Cubes1D to accelerate evaluation of symmetrical integrals. We also provide an initial Kokkos implementation to allow execution on non-Nvidia GPUs.

References

1. Borowka, S., Heinrich, G., Jahn, S., Jones, S., Kerner, M., Schlenk, J.: A gpu compatible quasi-monte carlo integrator interfaced to pysecdec. Computer Physics Communications **240**, 120–137 (Jul 2019). https://doi.org/10.1016/j.cpc.2019.02.015, http://dx.doi.org/10.1016/j.cpc.2019.02.015
2. Carrazza, S., Cruz-Martinez, J.M.: VegasFlow: accelerating Monte Carlo simulation across multiple hardware platforms. Comput. Phys. Commun. **254**, 107376 (2020). https://doi.org/10.1016/j.cpc.2020.107376
3. Cruz-Martinez, J., Carrazza, S.: N3pdf/vegasflow: vegasflow v1.0 (Feb 2020). https://doi.org/10.5281/zenodo.3691926, https://doi.org/10.5281/zenodo.3691926
4. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12), 3202 – 3216 (2014). https://doi.org/https://doi.org/10.1016/j.jpdc.2014.07.003, http://www.sciencedirect.com/science/article/pii/S0743731514001257, domain-Specific Languages and High-Level Frameworks for High-Performance Computing
5. Genz, A.: Testing multidimensional integration routines. In: Proc. of International Conference on Tools, Methods and Languages for Scientific and Engineering Computation. p. 81–94. Elsevier North-Holland, Inc., USA (1984)
6. Goda, T., Suzuki, K.: Recent advances in higher order quasi-monte carlo methods. arXiv: Numerical Analysis (2019)
7. Kanzaki, J.: Monte carlo integration on gpu. Arxiv preprint arXiv:1010.2107 (2010)
8. Lepage, G.P.: Adaptive multidimensional integration: vegas enhanced. Journal of Computational Physics **439**, 110386 (2021). https://doi.org/https://doi.org/10.1016/j.jcp.2021.110386, https://www.sciencedirect.com/science/article/pii/S0021999121002813
9. Peter Lepage, G.: A new algorithm for adaptive multidimensional integration. Journal of Computational Physics **27**(2), 192–203 (1978). https://doi.org/https://doi.org/10.1016/0021-9991(78)90004-9, https://www.sciencedirect.com/science/article/pii/0021999178900049
10. Sakiotis, I., Arumugam, K., Paterno, M., Ranjan, D., Terzić, B., Zubair, M.: PAGANI: A Parallel Adaptive GPU Algorithm for Numerical Integration. Association for Computing Machinery, New York, NY, USA (2021), https://doi.org/10.1145/3458817.3476198
11. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D.S., Ibanez, D., Liber, N., Madsen, J.,

Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., Wilke, J.: Kokkos 3: Programming model extensions for the exascale era. IEEE Transactions on Parallel and Distributed Systems **33**(4), 805–817 (2022). https://doi.org/10.1109/TPDS.2021.3097283

12. Wu, H.Z., Zhang, J.J., Pang, L.G., Wang, Q.: Zmcintegral: A package for multi-dimensional monte carlo integration on multi-gpus. Computer Physics Communications **248**, 106962 (2020). https://doi.org/https://doi.org/10.1016/j.cpc.2019.106962, https://www.sciencedirect.com/science/article/pii/S0010465519303121