Adaptive Mixed-Precision Monte Carlo Integration on GPUs

Ferhat Onur Özgan[1†], Berke Kabasakal[1†], Fahreddin Şükrü Torun[1*]

[1*]Computer Engineering Department, Ankara Yıldırım Beyazıt University, Ankara, 06020, Turkiye.

*Corresponding author(s). E-mail(s): fstorun@aybu.edu.tr;
Contributing authors: 22050111040@aybu.edu.tr;
22050111070@aybu.edu.tr;
†These authors contributed equally to this work.

Abstract

Monte Carlo integration (MCI) is widely used for evaluating high-dimensional or non-analytic functions, but its large number of function evaluations can make it computationally demanding. As modern scientific applications increasingly rely on GPU acceleration, balancing numerical accuracy and performance has become a key challenge. To address this, we present a GPU-accelerated mixed-precision MCI framework that adaptively selects precision based on local numerical behavior using heuristics derived from gradient, variance, and average value analysis. Two precision allocation strategies are explored: term-wise and region-wise, both implemented with CUDA batch processing to maximize GPU efficiency. Experimental results on representative test functions of varying dimensionality demonstrate speedups of up to 4.9× compared to full double precision, while maintaining controlled relative error. The proposed framework achieves a practical balance between computational efficiency and numerical reliability. It allows integration tasks on GPUs to run accurately and to adapt their precision automatically for higher performance.

Keywords: Monte Carlo integration, mixed precision, GPU computing, CUDA, high-performance computing

1 Introduction

Numerical integration plays an important role in many applications in physics, engineering, computational finance, and machine learning. These applications rely on integrating high-dimensional or non-analytic functions and they demand both accuracy and computational efficiency. These functions may not have exact formulas or may be costly to evaluate, which makes traditional numerical methods slow or unusable.

Monte Carlo integration (MCI) is one of the most popular methods for numerical integration, especially in high-dimensional settings [1, 2]. It is widely used due to its simplicity and robust performance. MCI estimates the value of an integral using random samples. Its accuracy improves as more samples are used, therefore the rate of improvement is relatively slow. This slow convergence rate requires a large number of evaluations to achieve acceptable accuracy. Therefore, computational cost becomes significant especially when the integrand is complex, discontinuous, or numerically unstable.

Graphics Processing Units (GPUs) offer massive parallelism and are well suited to accelerating Monte Carlo simulations. Their architecture allows thousands of threads to perform independent function evaluations simultaneously. However, achieving high performance on GPUs requires careful design, such as attention to memory access patterns, workload balance, and arithmetic efficiency [3].

One challenge in this setting is how to manage the trade-off between computational performance and numerical accuracy. While single-precision arithmetic executes faster and uses less memory, it may suffer from rounding errors, especially in regions where the integrand exhibits sharp gradients or high variance. Double-precision arithmetic improves robustness but causes higher cost on many GPU architectures.

To address this trade-off, we propose a GPU-accelerated Monte Carlo integration framework that dynamically assigns floating-point precision based on local function behavior. The framework uses heuristics derived from the function's gradient, variance, and local average to decide where higher precision is necessary. Our proposed framework supports two modes of precision assignment which are a term-wise strategy that analyzes components of the integrand expression, and a region-wise strategy that partitions the integration domain.

Our framework is implemented using CUDA and relies on batch processing techniques to maintain high GPU occupancy. By selecting precision levels intelligently based on function behavior, the method improves efficiency without sacrificing accuracy. Experimental results on two-dimensional integrals show up to 2× speedup over fixed double-precision integration, with minimal loss in accuracy. These results demonstrate the potential of adaptive precision strategies to combine performance and numerical robustness.

The remainder of this paper is organized as follows. Section 2 provides background information and reviews related work on Monte Carlo integration, GPU-based numerical computation, and mixed-precision algorithms. Section 3 describes the proposed adaptive precision framework, including heuristic strategies and GPU implementation details. Section 4 presents the experimental setup and results. Section 5 discusses the findings and outlines future research directions. Finally, Section 6 concludes the paper with a summary of key contributions.

2 Background and Related Work

Monte Carlo integration (MCI) is a fundamental numerical method for estimating the result of definite integrals using random sampling. It provides a fast and effective alternative to traditional methods [4]. It is suitable for high dimensional and hard computational problems that can't be integrated by formula. One of MCI's main advantages is that its convergence rate is approximately proportional to $1/\sqrt{N}$, where N is the number of samples, and this rate is independent of the dimension of the problem.

A standard formulation of MCI for a function $f(x)$ over a domain Ω is given by:

$$I \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i, y_i), \quad (x_i, y_i) \sim \text{Uniform}(\Omega) \tag{1}$$

Each point (x_i, y_i) is drawn uniformly from Ω. Although the formulation is simple, MCI can become computationally expensive when the integrand is costly to evaluate or high accuracy is required.

Several studies have targeted MCI on different hardware accelerators. Kanzaki [5] presents one of the earliest GPU-based implementations for MCI that achieves significant speedups over CPU implementation. ZMCintegral [6] extends this line of work with a stratified sampling approach for multi-GPU systems. The m-CUBES framework [7] introduces a region-based adaptive integration scheme that emphasizes portability and scalability. These methods, however, employ fixed-precision arithmetic and do not incorporate adaptive precision control. On the other hand, Chow et al. [8] proposed a mixed-precision Monte Carlo scheme for FPGAs. They use statically assigned precision levels to improve performance. Their method is hardware-specific and lacks runtime adaptivity. In contrast, our GPU-based approach adjusts precision dynamically based on function behavior.

Mixed-precision computing is a numerical strategy that aims at balancing performance and accuracy, particularly in high-performance computing applications. Traditional computations use fixed formats such as single or double precision. This flexibility gives rise to advanced strategies where computations mix different precision levels within the same algorithm to gain speed without compromising accuracy beyond acceptable bounds.

Mixed-precision techniques have been extensively studied and successfully applied in several computational domains. In deep learning, Micikevicius et al. [9] show that mixed-precision arithmetic can significantly accelerate training while maintaining model accuracy. Their models maintain comparable accuracy due to careful control over scaling operations. Katare et al. [10] showed that using mixed-precision with variational inference makes vision transformer models run efficiently on limited-resource devices. In numerical linear algebra, Higham and Mary [11] provided a comprehensive analysis of mixed-precision algorithms. They focus on the design, stability, and error bounds of main operations such as matrix factorizations and linear solvers. Their work describes how mixed-precision arithmetic can be used in numerical algorithms with attention to stability and error bounds.

In contrast to these works, our method is the first to introduce adaptive mixed-precision selection within MCI on GPUs. Unlike FPGA-based static assignments [8], our approach uses runtime heuristics based on gradient, variance, and mean to select between single and double precision. Unlike ZMCintegral [6] and m-CUBES [7], our framework includes precision control in the sampling process. It does not use precision as a fixed setting. Instead, it adjusts the precision during execution.

Our framework supports two strategies for adaptive precision assignment. One selects precision for each term in the integrand. The other assigns precision based on different regions of the integration domain. This adaptivity allows for reduced execution time while preserving numerical accuracy.

3 Proposed Method

In this work, we integrate Monte Carlo integration (MCI) with mixed-precision computation into a unified framework. The main idea is that the system automatically selects the precision based on the characteristics of each expression term combined with effective region splitting and batch processing strategies. The proposed method improves traditional MCI by combining three main ideas which are assigning precision per term, dividing the domain adaptively, and using GPU batch processing. Instead of using the same precision for the entire expression, our system selects suitable precision adaptively.

The process begins with converting the input mathematical expression into postfix notation then it is handled by Shunting Yard Algorithm [12]. Shunting Yard Algorithm is a stack-based method for converting infix expressions to postfix notation with correct precedence and associativity. This transformation simplifies the evaluation of expressions by eliminating the need for parentheses and operator precedence during computation. Then each string expression is evaluated using the stack in GPU kernels.

3.1 Adaptive Precision Selection Heuristics

The key component of the proposed approach is the precision selection algorithm. To appropriately choose precision levels three characteristics of the integrand are analyzed and estimated for each term or region. These include the average function value, the maximum gradient magnitude (approximated using finite differences), and the variance of the function computed through Monte Carlo sampling [13]. These estimates are computed over a fixed random sample space where the same set of randomly drawn points is reused throughout the analysis. This improves consistency and reduces stochastic noise in the decision process. Based on these estimates, the framework selects precision levels that balance computational efficiency and numerical accuracy.

The adaptive framework is driven by a precision selection function, denoted as $P(\text{avg}, \text{grad}, \text{var}, \text{tol})$. This function determines the appropriate arithmetic precision based on estimated numerical characteristics of the integrand. It compares the estimated error against a user-defined tolerance threshold tol.

For each candidate precision level $p \in \{\text{half (FP16)}, \text{float (FP32)}, \text{double (FP64)}\}$, the algorithm estimates the rounding error using:

$$\text{error}_p = \epsilon_p \cdot \max_f \cdot \text{grad} \tag{2}$$

Here, ϵ_p denotes the machine epsilon for precision p, $\max_f$ is the maximum observed function value, and grad is the estimated gradient magnitude of the integrand. The selection strategy begins with single precision and escalates only when the estimated error exceeds the tolerance, thereby favoring efficiency without sacrificing numerical accuracy.

All heuristic quantities are evaluated using 100 random sample points uniformly distributed over the integration domain.

The average function magnitude is estimated as:

$$\text{avg} = \frac{1}{N_s} \sum_{i=1}^{N_s} |f(\mathbf{x}_i)| \tag{3}$$

where $N_s = 100$ and each $\mathbf{x}_i$ is sampled uniformly from Ω.

The gradient magnitude is approximated using finite differences on the same 100 unstructured random sample points:

$$\text{grad} = \max_{i,d} \left| \frac{f(\mathbf{x}_i + h\mathbf{e}_d) - f(\mathbf{x}_i)}{h} \right| \tag{4}$$

where $\mathbf{e}_d$ is the unit vector in dimension d and $h = 10^{-6}$ is the step size. For each base sample $\mathbf{x}_i$, one additional perturbed evaluation is performed per dimension, resulting in a total of $N_s \times (1 + d)$ function evaluations for gradient estimation.

The function variance is estimated using the standard Monte Carlo estimator:

$$\text{var} = \frac{1}{n-1} \left[\sum_{i=1}^{n} f_i^2 - \frac{1}{n} \left(\sum_{i=1}^{n} f_i \right)^2 \right], \tag{5}$$

where n is the number of evaluation points and f_i denotes the value of the function at the i^{th} sample.

The P function selects the lowest-cost precision level for which $\text{error}_p \leq \text{tol}$. The precision selection function can be formally expressed as:

$$P(\text{avg}, \text{grad}, \text{var}, \text{tol}) = \min \left\{ p \in \{\text{half}, \text{float}, \text{double}\} \ : \ \text{total\_error}_p \leq \text{tol} \right\}, \tag{6}$$

where the total estimated error for each precision level p is defined by

$$\text{total\_error}_p = \epsilon_p \cdot (\text{max\_val} \cdot \text{normalized\_grad} \cdot \sqrt{\text{operation\_count}}) + \epsilon_p \cdot \sqrt{\text{var}}. \tag{7}$$

In (7), the quantity max_val, defined as $\max(|\text{avg}|, \sqrt{\text{var}}, 10^{-10})$, provides a conservative measure of the function's magnitude. The small constant 10^{-10} ensures

numerical stability by preventing division by zero and avoiding the underestimation of errors for very small values. The factor $\text{normalized\_grad} = \max(\text{grad}, 1.0)$ ensures that low gradient values do not lead to unrealistically small error estimates. The component $\epsilon_p \cdot (\text{max\_val} \cdot \text{normalized\_grad} \cdot \sqrt{\text{operation\_count}})$ quantifies the rounding error accumulated during arithmetic operations, while $\epsilon_p \cdot \sqrt{\text{var}}$ represents the stochastic uncertainty caused by Monte Carlo sampling. Based on these quantities, the precision selection function P determines the lowest precision level that satisfies $\text{total\_error}_p \leq \text{tol}$, ensuring a balance between numerical reliability and computational efficiency.

The function $P(\text{avg}, \text{grad}, \text{var}, \text{tol})$ requires only $O(N_s)$ function evaluations for input statistics, where $N_s = 100$. Since the Monte Carlo integration phase involves $O(M)$ evaluations with $M \gg N_s$, the overhead of the precision selection is negligible in practice.

Half-Precision (FP16) Arithmetic in Monte Carlo Integration

Although modern GPUs provide substantial hardware support for half-precision arithmetic, FP16 does not yield performance benefits for the Monte Carlo integration workloads considered in this study. Experiments on both NVIDIA RTX 3070 and H100 GPUs show that FP16 kernels are consistently slightly slower than their single-precision counterparts.

This behavior arises from the characteristics of Monte Carlo integration, which is dominated by scalar arithmetic, transcendental function evaluations, random number generation, and control flow rather than dense, tensor-core–friendly operations. Consequently, performance is largely limited by memory access patterns and instruction overhead, making arithmetic precision less relevant to overall throughput. Moreover, FP16 introduces additional type conversions and offers limited native support for transcendental functions, which offsets any potential gains from reduced precision.

While the proposed framework supports FP16 for simple arithmetic, our experiments indicate that FP32 represents the practical performance optimum for Monte Carlo integration on current GPU architectures. Therefore, FP16 is evaluated only in a limited capacity, and the experimental analysis in this work focuses on single and double precision, which provide a more reliable balance between performance and numerical accuracy.

Accuracy Evaluation of the Heuristic

In order to evaluate the reliability of our precision-selection heuristic, we conducted an independent validation study with 210 test cases. These tests include 10 elementary functions listed in Appendix A1, along with 21 parameterized variants evaluated over different domain ranges, as detailed in Appendix A2. Each function was evaluated in three precision settings (float, double and long double). Long double was used on the CPU as the reference. Float and double were used to check the accuracy of the heuristic's predictions.

Algorithm 1 outlines the validation of the proposed heuristic. For each test case, we first computed the reference integral I_{ref} using long double precision. Then, the

proposed precision-selection heuristic P analyzed the numerical behavior of the integrand (its gradient, variance, and mean) to decide whether float or double precision should be used. The predicted precision was then applied to compute I_{pred}, and the absolute error $|I_{\text{ref}} - I_{\text{pred}}|$ was compared with a user-defined tolerance *tol*. If the error was below the tolerance, the prediction was considered acceptable.

Algorithm 1 Validation of the Precision-Selection Heuristic

for each test case: function $f(x, y)$ over domain $[a, b] \times [c, d]$ with tolerance *tol* **do**
    Compute reference integral I_{ref} using long double
    Predict precision p using proposed heuristic $P()$
    Compute integral I_{pred} using precision p
    Compute absolute error $e \leftarrow |I_{\text{ref}} - I_{\text{pred}}|$
    if $e \leq tol$ **then**
        Prediction classified as acceptable
    else
        Prediction classified as too low precision
    end if
end for

Table 1 summarizes the results of the validation. The heuristic selected the correct precision in 182 out of 210 cases. Therefore, it achieves an overall success rate of 86.67%. In 28 cases, it underestimated the required precision, leading to errors that exceeded the tolerance threshold; these cases correspond to false negatives. No false positives were observed. A false positive corresponds to a conservative overestimation, where a higher precision is selected even though a lower precision would satisfy the tolerance threshold.

Table 1 Validation results of the precision-selection heuristic across 210 tests

Metric	Count	Percentage (%)
Total test cases	210	100.00
Correct predictions	182	86.67
False negative (risky underestimation)	28	13.33
False positive (conservative overestimation)	0	0.00

3.2 Term-wise Precision Assignment Strategy

In the proposed approach, we divide the function into smaller additive terms by making use of the linearity property of integration. In this way, each component of the integrand is analyzed independently and assigned a precision level based on individual behavior. For example, a function such as $f(x, y) = x \cdot y + \sin(x) + \cos(y)$ is separated into individual terms: $[x \cdot y]$, $[\sin(x)]$, and $[\cos(y)]$. The precision heuristic is then

applied to each term separately. All terms are evaluated in parallel across Monte Carlo samples. Within each sample, term values are summed in FP64 to form the integrand value, and Monte Carlo sample contributions are accumulated in the same numerical precision as the function evaluations, with single-precision evaluations accumulated in FP32 and double-precision evaluations accumulated in FP64.

This term-wise strategy enables batch processing on GPU architectures. Multiple terms can be evaluated concurrently using separate CUDA thread blocks, allowing independent handling of expression components. The proposed framework maps terms and Monte Carlo samples onto a two-dimensional CUDA grid, which facilitates structured parallel execution and simplifies mixed-precision control. This design preserves term independence and supports flexible precision assignment without introducing sequential dependencies. We note that, for large sample counts, GPU occupancy is primarily determined by sample-level parallelism, and term-wise decomposition is introduced for numerical flexibility and precision control rather than to achieve hardware saturation.

3.3 Region-wise Precision Assignment Strategy

In addition to term-wise decomposition, the proposed framework supports adaptive region-based processing. This approach partitions the integration domain into smaller subregions based on local function complexity. The same precision selection function P is applied independently within each region to determine an appropriate precision level. In contrast to term-wise analysis, which separates the integrand into additive components, region-wise processing operates on the entire function but over spatially distinct subdomains.

Adaptive Partitioning

The partitioning strategy is adaptive and recursive. Starting from the initial domain Ω, each region is analyzed using the Monte Carlo sampling approach described in Section 3.1. If either the variance or gradient exceeds predefined thresholds, the region is subdivided along its largest dimension at the midpoint. This process continues until all regions meet the thresholds or reach minimum size.

The partitioning priority is determined by a combined score $s_i = (\text{var}_i + \epsilon)\,(\|\nabla f\|_i + 1)$, where $\epsilon = 10^{-12}$ is a small constant introduced to ensure numerical stability. Here, var_i denotes the local variance estimated via Monte Carlo sampling, and $\|\nabla f\|_i$ denotes an estimate of the local gradient magnitude obtained using finite differences. This metric accounts for both stochastic sampling uncertainty and numerical sensitivity. Regions with higher scores are subdivided first, allowing computational effort and precision to be focused where they are most needed.

For an N-dimensional region with bounds $[a_1, b_1] \times \cdots \times [a_N, b_N]$, subdivision occurs along dimension k where $(b_k - a_k) = \max_j (b_j - a_j)$, creating two child regions. The integral is computed as the sum over all final regions:

$$I \approx \sum_{i=1}^{N_R} V_i \cdot \frac{1}{M_i} \sum_{j=1}^{M_i} f(\mathbf{x}_{ij}) \tag{8}$$

where N_R is the number of final regions, V_i is the volume of region i, and M_i is the number of samples allocated to that region. Each subregion is evaluated separately with its assigned precision level. This approach is effective for functions with sharp transitions, singularities, or regions of varying numerical stability, where using uniform global precision would result in unnecessary computational overhead.

3.4 GPU Implementation and Optimization Strategies

In our implementation, we use several CUDA-based techniques to improve both computational performance and numerical accuracy. We generate random samples using precision-matching functions. For single precision, we use `curand_uniform`, while double-precision samples are generated with `curand_uniform_double`. Each CUDA thread maintains its own random number generator state, ensuring that the generated samples are statistically independent [14].

To eliminate runtime parsing overhead, mathematical expressions are compiled once during initialization into a compact stack-based bytecode representation. This representation consists of four fixed-size arrays encoding token types (enumerations), numerical constants, variable indices, and operation codes. The compilation is performed on the host CPU using the Shunting Yard algorithm and incurs a negligible one-time cost, typically on the order of milliseconds for expressions containing 5–10 terms. The resulting bytecode, with a memory footprint of approximately 300 bytes per expression, is transferred to GPU device memory once prior to kernel execution.

During kernel execution, expression evaluation is carried out by a lightweight stack-based interpreter implemented using integer-indexed switch statements over operation codes (e.g., addition, trigonometric functions). This design completely avoids string operations, dynamic memory allocation, and runtime tokenization within the kernel, ensuring that expression evaluation overhead remains negligible relative to the computational cost of Monte Carlo sampling. As a result, the one-time compilation and transfer cost is amortized over millions of function evaluations and does not affect overall performance for typical integration workloads. To support concurrent evaluation, device memory is pre-allocated to store multiple expression terms, enabling parallel processing of distinct components of the integrand.

We also include runtime checks to handle numerical instability. Sample evaluations that result in `NaN` or infinity are discarded automatically. Additionally, operations that may lead to undefined behavior, such as division by zero or evaluating functions outside their valid domains, are detected and skipped. These checks strengthen numerical reliability and ensure that invalid outputs do not affect the final result.

We use CUDA's parallel architecture through batch processing design that simultaneously evaluates multiple terms across threads. The batch processing kernel maps mathematical expression terms to a two-dimensional CUDA grid, where each term occupies a separate block dimension (blockIdx.y) while Monte Carlo samples are distributed across thread blocks (blockIdx.x). This idea enables parallel evaluation of all expression terms, with each thread processing multiple samples. Memory allocation is pre-computed to have results from all terms simultaneously. This design

eliminates the need for sequential term evaluation and facilitates structured mixed-precision execution, while overall GPU utilization remains dominated by sample-level parallelism.

The framework supports integration up to 10 dimensions without algorithmic modification. Variable handling uses a unified indexing scheme mapping symbolic names (x, y, z, w, ...) to array positions. Statistical estimates (variance, gradient) scale linearly with dimensionality due to unstructured Monte Carlo sampling. The adaptive partitioning naturally extends to higher dimensions by splitting along the axis with maximum extent, with region count capped at 256 for memory efficiency.

4 Experimental Results

In this section, we evaluate the accuracy and performance of our mixed-precision Monte Carlo integration framework. Experiments were conducted on two different platforms. The first system consists of an NVIDIA GeForce RTX 3070 GPU paired with an Intel Core i7-13700F CPU, which is used to demonstrate baseline performance on a consumer-grade GPU. The second experimental platform is the kolyoz HPC supercomputer at the TÜBİTAK ULAKBİM High Performance and Grid Computing Center (TRUBA), equipped with an NVIDIA H100 GPU and an Intel Xeon Gold 6548 processor. All CUDA kernels on both platforms were compiled using the CUDA Toolkit 13.0.

4.1 Experimental Results on NVIDIA RTX 3070

Table 2 presents MCI results for a test function

$$g(x, y) = \sin(x + y) - \log(x) + \frac{x^4}{y} + 5 \tag{9}$$

which combines trigonometric, logarithmic, and rational polynomial terms. As observed in Table 2, the FP32 configuration delivers the shortest runtime among the evaluated methods, whereas FP16 provides no performance advantage and incurs a slightly larger relative error. The FP64 baseline yields the most accurate estimate but at the highest computational cost.

Among the mixed-precision strategies, the region-wise configuration assigns FP32 to three subregions and FP64 to one numerically sensitive region. This approach reduces runtime by approximately 79.7% relative to the FP64 baseline, while maintaining a relative error of 0.0334. In contrast, the term-wise mixed strategy applies FP32 to two terms, FP64 to one numerically sensitive term, and FP16 to a scalar term with negligible impact on numerical accuracy. This configuration achieves a lower relative error of 0.0103, but provides a more modest runtime reduction of 17.3%.

Overall, these results demonstrate that both mixed-precision strategies substantially reduce computational cost while preserving accuracy close to the FP64 reference. The region-wise approach favors performance by limiting high-precision evaluation to localized regions, whereas the term-wise strategy prioritizes accuracy by selectively elevating precision at the level of individual expression components.

Table 2 MCI results for Equation (9) over $[1, 2] \times [1, 2]$ using 10^8 random samples. The reference integral is 9.0409632.

Method	$\int g(x, y)\, dy\, dx$	R.Err (%)	Time (s)	T.Dec (%)
FP16 batch	9.06003	0.2109	0.0191	92.1
FP32 batch	9.05918	0.2015	0.0177	92.7
FP64 batch	9.04147	0.0056	0.2425	—
Mixed (term-wise)	9.04003	0.0103	0.2005	17.3
Mixed (region-wise)	9.04398	0.0334	0.0492	79.7

R.Err: relative error; T.Dec: percentage decrease in runtime relative to the FP64 batch.

Table 3 shows the term-wise precision allocation for MCI of (9). The rational polynomial term x^4/y exhibits high variance (18.149) and gradient (30.759), necessitating double precision. The remaining terms ($\sin(x + y)$, $-\log(x)$, and the constant 5) are computed efficiently in float precision due to their numerical stability.

Table 3 Precision decisions made by P for each term of Equation (9).

Expression	Avg	Var	Grad	Precision
$\sin(x + y)$	0.361	9.974×10^{-2}	0.999	Float (FP32)
$-\log(x)$	0.166	9.109×10^{-3}	0.423	Float (FP32)
x^4/y	4.516	1.815×10^{1}	30.759	Double (FP64)
5	5.000	0.000×10^{0}	0.000	Half (FP16)

Table 4 reports the region-wise allocation for MCI of (9). In this case, only Region 1 requires double precision due to its high variance (11.507) and gradient (30.167). The other three regions are computed in float precision, as their statistical characteristics indicate sufficient numerical stability. This allocation strategy achieves better efficiency compared to the first function, with 75% of the domain computed in lower precision while maintaining accuracy close to the double precision baseline.

Table 4 Region-wise precision decisions made by P for each region of Equation (9).

Region #	Avg	Var	Grad	Precision	Sub-Integral
0	7.671	1.289	12.064	Float (FP32)	1.879
1	12.907	11.507	30.167	Double (FP64)	3.154
2	6.589	0.510	7.486	Float (FP32)	1.616
3	10.092	4.937	19.756	Float (FP32)	2.467

Figure 1 shows the region-wise precision allocation determined by the P-function for (9) over the domain $[1, 2] \times [1, 2]$. The surface is divided into four regions based on numerical sensitivity analysis. Region 1 is computed in double precision due to its higher curvature and rapid variation, which increase the risk of rounding errors. The remaining regions exhibit smoother behavior and lower variance, for which computations are done in float precision. Assigning higher precision only where it is numerically justified reduces computational cost while maintaining stability.

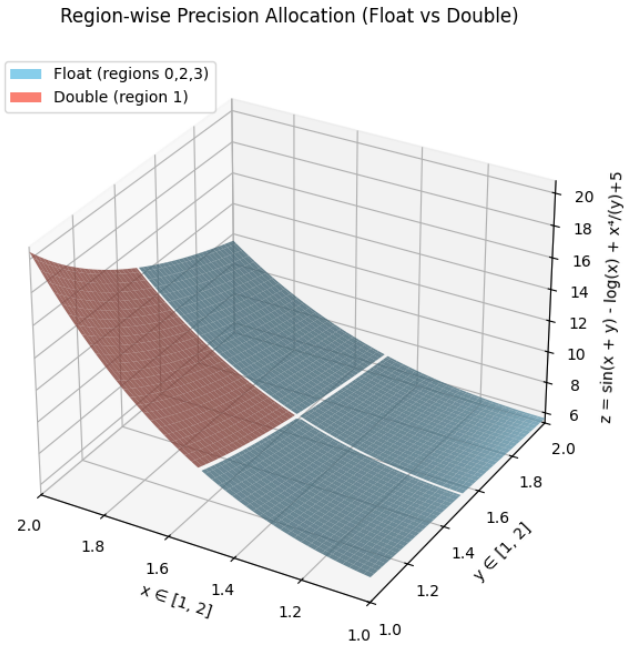

Fig. 1 Region-wise precision allocation for Equation (9). Float precision is applied in regions 0, 2 and 3 (blue), while double precision is used in region 1 (red).

Accuracy as a Function of Sample Count

To examine how numerical precision affects accuracy as the number of Monte Carlo samples increases, we consider an analytically integrable test function,

$$f(x, y, z) = x^2 + y^2 + z^2, \tag{10}$$

evaluated over the domain $[0, 1]^3$. The exact value of this integral is equal to 1, which enables a direct and unbiased assessment of absolute integration error.

Monte Carlo integration is performed using increasing sample counts, and the average and maximum absolute errors are reported for FP16, FP32, and FP64 evaluations. For each configuration, multiple runs with different random seeds are used to compute representative error statistics. Sample contributions are accumulated in the same

precision as the function evaluation, that is, FP32 accumulation for single-precision evaluation and FP64 accumulation for double-precision evaluation.

Table 5 summarizes the observed accuracy trends. For small sample counts, the error is dominated by stochastic Monte Carlo variance, and differences between numerical precisions are limited. As the number of samples increases, the impact of numerical precision becomes more pronounced, with higher precision consistently yielding lower average and maximum absolute error. These results confirm that higher evaluation and accumulation precision primarily improves accuracy in the regime where stochastic sampling error is sufficiently reduced.

Table 5 Average and maximum absolute errors for Monte Carlo integration of $f(x, y, z) = x^2 + y^2 + z^2$ over $[0, 1]^3$ (exact value = 1).

N	FP16		FP32		FP64	
	Avg Err	Max Err	Avg Err	Max Err	Avg Err	Max Err
10^3	7.41×10^{-3}	9.57×10^{-3}	7.18×10^{-3}	8.66×10^{-3}	1.14×10^{-3}	2.80×10^{-3}
10^4	7.39×10^{-3}	8.60×10^{-3}	7.43×10^{-3}	8.65×10^{-3}	2.57×10^{-3}	5.30×10^{-3}
10^5	2.18×10^{-3}	2.87×10^{-3}	2.73×10^{-3}	4.06×10^{-3}	8.09×10^{-4}	1.13×10^{-3}
10^6	4.10×10^{-4}	1.17×10^{-3}	4.37×10^{-4}	8.20×10^{-4}	1.35×10^{-4}	2.44×10^{-4}
10^7	1.51×10^{-4}	2.64×10^{-4}	1.13×10^{-4}	3.18×10^{-4}	9.94×10^{-5}	2.70×10^{-4}

4.1.1 High-Dimensional Integration Experiments

To evaluate the scalability and effectiveness of the proposed mixed-precision framework in higher dimensions, we consider two representative test problems defined over unit hypercubes. The first example examines a four-dimensional integral over $[0, 1]^4$, providing a baseline for moderate dimensionality. The second example extends the evaluation to an eight-dimensional integral over $[0, 1]^8$, allowing us to assess performance and accuracy trends as the dimensionality increases.

The first integrand is defined as

$$\begin{aligned} h(x, y, z, w) = \; & \sin(x + y + z + w) + \cos(xy) - z^2 + x^{12} - w^3 \\ & + \sin(zw^2) - \log(1 + zw)\, x^5 + y^4 e^{-w} - 4 \end{aligned} \tag{11}$$

and combines trigonometric terms, high-degree polynomial components with large dynamic range, and logarithmic expressions that may exhibit increased numerical sensitivity. This mixture of behaviors provides a meaningful test case for evaluating adaptive precision selection in multi-dimensional Monte Carlo integration.

Table 6 presents the Monte Carlo integration results for the four-dimensional test function using 10^8 samples. The FP32 batch configuration achieves the shortest runtime, while the FP64 baseline provides the most accurate reference result at the highest computational cost. Both mixed-precision strategies significantly reduce runtime compared to pure double precision.

Table 6 MCI results for the 4D function (Equation (11)) over $[0,1]^4$ using 10^8 samples.

Method	$\int h\,dw\,dz\,dy\,dx$	R.Err (%)	Time (s)	T.Dec (%)
FP32 batch	-2.519512	0.0027	0.110	89.6
FP64 batch	-2.519581	0.0000	1.055	—
Mixed (term-wise)	-2.519643	0.0024	0.260	75.3
Mixed (region-wise)	-2.519766	0.0070	0.225	78.7

R.Err: relative error; T.Dec: percentage decrease in runtime relative to Double batch.

For the term-wise mixed-precision configuration, the constant term -4 is assigned FP16 by the precision selection function, but is evaluated in FP32 in the implementation, as FP16 was found to provide no performance benefit over FP32 in our study. The remaining smooth and bounded terms, namely $\sin(x+y+z+w)$, $\cos(xy)$, $\sin(zw^2)$, and $-\log(1+zw)\cdot x^5$, are evaluated in FP32, while terms exhibiting higher polynomial growth or stronger numerical sensitivity (y^4e^{-w}, $-z^2$, x^{12}, and $-w^3$) are evaluated in FP64.

The region-wise mixed-precision strategy partitions the domain into 256 subregions, of which 229 are evaluated in FP32 and 27 are promoted to FP64 due to higher numerical sensitivity. As a result, approximately 89.5% of the regions are processed in single precision, while double precision is selectively applied only where necessary. This distribution explains the substantial runtime reduction achieved by the region-wise approach.

Figure 2 compares GPU execution times for FP32, FP64, and two mixed-precision strategies when integrating (11) over $[0,1]^4$ as a function of the number of Monte Carlo samples. Across all methods, runtime increases with sample count, but the mixed-precision strategies exhibit a pronounced fixed overhead at low sample counts. In particular, the region-wise strategy is dominated by its partitioning and region-management overhead for small and moderate N, leading to nearly constant runtimes that are higher than both FP32 and FP64 in this regime. As N increases, this fixed cost is amortized and the region-wise curve becomes competitive, eventually approaching the FP32 curve and remaining well below the FP64 runtime at large sample counts.

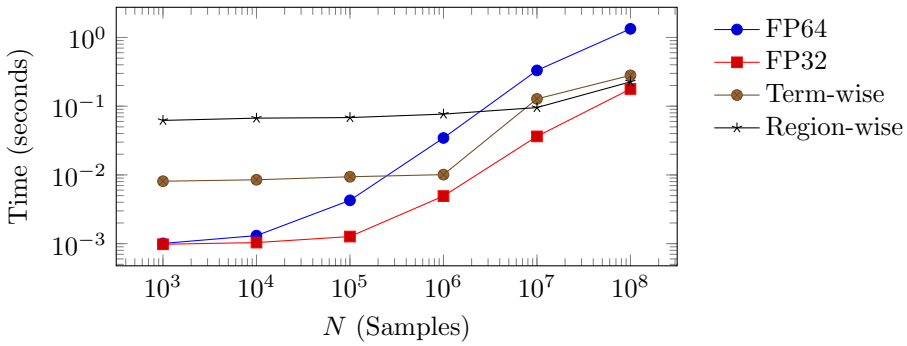

Fig. 2 GPU performance comparison for MCI of (11) over $[0,1]^4$ with increasing sample counts.

The term-wise strategy shows a smaller fixed overhead than the region-wise approach. The term-wise strategy is slower than FP32 for small N, but scales more favorably and remains consistently below FP64 once the sampling workload dominates. Overall, the figure indicates that mixed-precision benefits are most pronounced at large sample counts, where overheads are amortized and selective use of FP64 reduces runtime relative to full FP64 evaluation.

Table 7 reports results for an eight-dimensional integrand defined over the unit hypercube $[0,1]^8$ using 10^8 samples. The integrand,

$$\begin{aligned} h(a,b,c,d,x,v,y,z) = &\sin(2\pi(a+b)) + \cos(5cd) \\ &+ \exp(-0.5(x^2+v^2)) + \exp(-|y|) \\ &+ \frac{1}{1+\exp(-25(z-0.05))} + \ln(1+|ax|) \\ &+ 0.01(\sqrt{1+10^{-5}(bc+dx)}-1) \\ &+ (\ln(1+10^{-5}(v+y)) - 10^{-5}(v+y)) \\ &+ \frac{0.001}{\sqrt{10^{-12}+(y-0.02)^2+(z+0.03)^2}} \\ &+ (\exp(8a) - \exp(8a-10^{-5})), \end{aligned} \tag{12}$$

combines smooth components, such as the trigonometric terms $\sin(2\pi(a+b))$ and $\cos(5cd)$ and the Gaussian-like exponential $\exp(-0.5(x^2+v^2))$, with nonlinear terms that exhibit steeper gradients and localized sensitivity. Sharp transitions arise from the logistic function $\big(1+\exp(-25(z-0.05))\big)^{-1}$ and the near-singular rational term involving $(y-0.02)^2+(z+0.03)^2$ in the denominator. In addition, expressions formed by subtracting nearly equal logarithmic or exponential quantities, such as $\ln(1+10^{-5}(v+y))-10^{-5}(v+y)$ and $\exp(8a)-\exp(8a-10^{-5})$, are prone to numerical cancellation. Such composite integrands are representative of practical Monte Carlo workloads [15], where the quantity of interest is often expressed as the sum of multiple model components with differing numerical characteristics.

Table 7 MCI results for (12) over $[0,1]^8$ using 10^8 samples on NVIDIA RTX 3070.

Method	$\int h\,dz\,dy\,\ldots\,da$	R.Err (%)	Time (s)	T.Dec (%)
FP32 batch	2.745462	0.1765	0.282	60.1
FP64 batch	2.740625	0.0000	0.706	—
Mixed (term-wise)	2.743512	0.0255	0.631	10.6
Mixed (region-wise)	2.741326	0.1053	0.401	43.2

R.Err: relative error as percentage; T.Dec: percentage decrease in runtime relative to FP64.

As shown in Table 7, the FP32 configuration achieves the shortest runtime but incurs the largest relative error, while the FP64 baseline provides the most accurate reference at the highest computational cost. The term-wise mixed-precision strategy

reduces runtime by 10.6% relative to FP64 while maintaining a relative error of 0.0255. In contrast, the region-wise strategy achieves a larger runtime reduction of 43.2% with a relative error of 0.1053, reflecting the trade-off between spatially coarse precision assignment and numerical accuracy. Overall, these results demonstrate that the proposed mixed-precision strategies remain effective as the problem dimensionality increases.

The two mixed-precision strategies exhibit distinct performance–accuracy trade-offs. In the term-wise configuration, five smooth or moderately varying terms are evaluated in FP32, while three numerically sensitive terms are promoted to FP64. Relative to the FP32 baseline, this approach yields a normalized runtime of 2.24×, while reducing the relative error from 1.77×10^{-1} to 2.55×10^{-2}. This behavior reflects the fine-grained precision control at the expression level, which prioritizes numerical robustness at the cost of additional high-precision evaluations.

In contrast, the region-wise strategy assigns FP32 to the majority of spatial subregions, with approximately 93.8% (240 out of 256) of the domain evaluated in single precision and only a small subset promoted to FP64 due to elevated numerical sensitivity. This configuration achieves a normalized runtime of 1.42× relative to FP32, while reducing the relative error to 1.05×10^{-1}. Together, these results indicate that term-wise precision selection favors accuracy through targeted high-precision evaluation, whereas region-wise selection recovers a larger fraction of FP32 performance by limiting double-precision computation to localized regions of the integration domain.

4.2 Experimental Results on NVIDIA H100 (TRUBA HPC)

To assess the portability and robustness of the proposed mixed-precision Monte Carlo integration framework across GPU architectures, we next present experimental results obtained on an NVIDIA H100 accelerator deployed on the TRUBA HPC system. In contrast to the NVIDIA RTX 3070, which is a consumer-class GPU primarily optimized for single-precision throughput, the H100 is a modern HPC-class accelerator featuring substantially higher double-precision performance, increased memory bandwidth, and architectural support tailored for large-scale scientific workloads. As a result, the relative performance gap between FP32 and FP64 computations is expected to be smaller on the H100, potentially reducing the attainable benefits of mixed-precision execution. Evaluating the framework on the H100 therefore provides insight into its effectiveness on contemporary high-performance computing platforms.

Table 8 reports Monte Carlo integration results for the same eight-dimensional integrand (Equation (12)) evaluated over the unit hypercube $[0, 1]^9$ using 10^9 samples. Compared to the RTX 3070 experiments, a larger sample count is employed to fully utilize the higher computational throughput of the H100 and to ensure that fixed overheads are amortized.

As expected, the FP64 configuration provides the most accurate reference result at the highest computational cost, while FP32 achieves the lowest runtime with increased relative error. The term-wise mixed-precision strategy reduces runtime by 13.3% relative to FP64 while maintaining a relative error of 7.35×10^{-2} which reflects the selective use of double precision for numerically sensitive terms. The region-wise strategy achieves a larger runtime reduction of 21.2%, with a relative error of 3.87×10^{-2},

Table 8 MCI results for (12) over $[0,1]^8$ using 10^9 samples on NVIDIA H100.

Method	$\int h\,dz\,dy\,\ldots\,da$	R.Err (%)	Time (s)	T.Dec (%)
FP32 batch	2.744743	0.1503	0.8088	44.8
FP64 batch	2.740624	0.0000	1.4671	—
Mixed (term-wise)	2.742639	0.0735	1.2712	13.3
Mixed (region-wise)	2.741685	0.0387	1.1579	21.2

R.Err: relative error as percentage; T.Dec: percentage decrease in runtime relative to FP64.

by assigning single precision to the majority of spatial subregions and promoting only a limited subset to FP64.

On the NVIDIA H100, the term-wise mixed-precision configuration assigns FP32 to five expression terms and FP64 to three numerically sensitive terms, with no terms executed in FP16. This allocation results in a selection overhead of approximately 2.86% and yields a substantial reduction in runtime relative to a full FP64 evaluation, consistent with the results reported in Table 8. In contrast, the region-wise strategy applies precision selection at the spatial level, evaluating approximately 94.9% of the integration subregions in FP32 and promoting only 13 out of 256 regions to FP64. While this approach incurs a higher selection overhead due to region-level adaptivity, it limits double-precision computation to a small fraction of the domain and thereby achieves a larger overall runtime reduction. In both strategies, half precision is omitted from execution, as FP16 provides no performance benefit for the Monte Carlo workloads considered.

Compared to the RTX 3070 results, the performance differences between FP32, FP64, and mixed-precision configurations are noticeably reduced on the H100. This behavior is consistent with the substantially higher native FP64 throughput of the H100 architecture, which diminishes the relative advantage of mixed-precision execution. Nevertheless, both mixed-precision strategies continue to provide runtime reductions while maintaining controlled accuracy, demonstrating that the proposed framework remains effective on modern HPC-class GPUs.

5 Discussion and Future Work

The experimental results demonstrate that the proposed adaptive precision strategies reduce computational cost relative to full double-precision evaluation while maintaining proximity to the reference solution. Across all tested functions, the mixed-precision configurations achieve lower runtimes than FP64, with relative errors that remain consistent with the applied precision assignments. The term-wise strategy incurs additional overhead due to fine-grained precision selection, but achieves lower relative error by selectively evaluating numerically sensitive expression components in double precision. In contrast, the region-wise strategy assigns single precision to the majority of the integration domain, resulting in larger runtime reductions accompanied by higher relative error for functions exhibiting localized sensitivity. GPU performance analysis further indicates that mixed-precision configurations scale with sample count in

a manner comparable to single precision, whereas full double precision consistently incurs higher execution cost.

Although the experiments include problems up to eight dimensions, the proposed framework is not inherently limited to this scale. Since Monte Carlo convergence is dimension-independent and the heuristic statistics scale linearly with the number of variables, the method remains applicable to higher-dimensional problems.

While the current heuristics are effective for the test cases considered, their behavior remains dependent on the numerical characteristics of the integrand. The precision selection mechanisms capture local numerical sensitivity but may be less effective for functions exhibiting complex global behavior, such as highly oscillatory or discontinuous patterns. In addition, the term-wise and region-wise strategies are currently applied independently. A hybrid approach that combines expression-level and spatial precision selection could provide finer control over numerical behavior, albeit at the cost of increased analytical and computational complexity.

The current implementation evaluates mathematical expressions on the GPU using a precompiled stack-based bytecode representation generated on the host. This design avoids all device-side parsing and enables runtime flexibility, allowing arbitrary expressions and mixed-precision assignments without recompiling CUDA kernels. In the Monte Carlo workloads considered in this study, overall runtime is primarily dominated by random number generation, transcendental evaluations, and memory access rather than expression dispatch. Nevertheless, native kernel generation (e.g., via templates or NVRTC-based runtime compilation) could further reduce instruction overhead in fixed-expression scenarios. Such approaches represent a promising direction for future work, particularly in large-scale production environments where maximum throughput is critical.

Beyond kernel-generation strategies, future work will focus on improving the generality and robustness of precision selection. Machine learning–based models could be developed to automatically determine the most suitable precision strategy—term-wise, region-wise, or hybrid—for a given integrand based on its numerical characteristics. Further extensions may incorporate probabilistic modeling to better handle uncertainty in error estimates and reduce reliance on manually tuned tolerance parameters. Finally, alternative approaches based on unstructured or stochastic sampling for gradient estimation could be explored to mitigate dimensionality-related costs, although such extensions are beyond the scope of the present work.

6 Conclusion

This study introduces a mixed-precision Monte Carlo integration framework that automatically selects appropriate floating-point precision based on the numerical characteristics of the target function. The proposed approach analyzes mathematical expressions and divides the computation by terms and regions. Then it assigns a suitable precision level to each. In doing so, it achieves a balance between computational efficiency and numerical reliability, maintaining acceptable accuracy while significantly improving performance.

The experimental results demonstrate that the proposed adaptive mixed-precision framework consistently reduces computational cost relative to full double-precision evaluation while preserving proximity to the FP64 reference solution. The framework was evaluated using multiple test functions of increasing dimensionality and numerical complexity, combining trigonometric, exponential, polynomial, logarithmic, and rational components that exhibit smooth behavior, steep gradients, and cancellation-prone expressions.

Across all experiments, both term-wise and region-wise precision strategies achieved lower runtimes than pure double precision. The term-wise approach reduced runtime by selectively promoting only numerically sensitive expression components to double precision, resulting in improved numerical accuracy at the cost of moderate overhead. In contrast, the region-wise strategy assigned single precision to the majority of the integration domain and promoted a limited number of subregions to double precision, yielding larger runtime reductions accompanied by higher relative error for functions with localized sensitivity.

The relative performance and accuracy trade-offs observed on the NVIDIA RTX 3070 were also confirmed on the NVIDIA H100 GPU. On the H100, the higher native FP64 throughput reduced the performance gap between single and double precision, leading to smaller but still consistent runtime reductions for mixed-precision configurations. These results indicate that the proposed framework remains effective across both consumer- and HPC-class GPUs and scales robustly with problem dimensionality and sample count.

Acknowledgments

The numerical calculations reported in this paper were partially performed at TUBITAK ULAKBIM, High Performance and Grid Computing Center (TRUBA resources).

The authors thank the anonymous reviewers for their valuable suggestions. The authors acknowledge the use of Grammarly, GPT-4o and GPT-5 to improve the clarity of the manuscript's English-language presentation. The AI tools were used as language aids, with all technical content and interpretations rigorously verified.

Author contributions statement

F. O. Özgan and B. Kabasakal contributed equally to the development of the mixed-precision framework, the design and implementation of experiments, and the analysis of numerical results. They jointly interpreted the findings, prepared the manuscript, and revised it. F. Ş. Torun conceived the initial research idea, supervised the overall study, and provided critical guidance throughout the development of the framework, the execution of experiments on the TRUBA high-performance computing system, and the preparation of the manuscript.

Code Availability

The source code for the mixed-precision Monte Carlo integration framework is available at https://github.com/AYBU-ParLab/MixedPrecisionMCI/, along with example scripts demonstrating its usage and configuration.

Competing interests

No competing interest is declared.

Appendix A Test Functions and Integration Domains

This appendix provides the detailed mathematical definitions of the functions and integration domains used in the heuristic validation study described in Section 3.1.

Table A1 Elementary test functions used for precision-selection validation.

Function $f(x, y)$	Function $f(x, y)$
$\sin(xy)$	$\cos(x + y)$
$\exp(-0.1xy)$	$\log(\lvert x\rvert + 1)$
$\sqrt{x^2 + y^2}$	$x^3 + y^2$
$\arctan(x/y)$	$\dfrac{1}{x^2 + y^2 + 1}$
$\sin(x)\cos(y)$	$\exp(-x^2 - y^2)$

Table A2 Integration domains $[a, b] \times [c, d]$ used in testing.

ID	$[a, b] \times [c, d]$	ID	$[a, b] \times [c, d]$
1	$[0, 1] \times [0, 1]$	12	$[0, 20] \times [0, 20]$
2	$[-1, 1] \times [-1, 1]$	13	$[-3, 3] \times [-3, 3]$
3	$[0, 10] \times [0, 10]$	14	$[0, 0.5] \times [0, 0.5]$
4	$[-5, 5] \times [-5, 5]$	15	$[-100, 100] \times [-100, 100]$
5	$[0, 0.1] \times [0, 0.1]$	16	$[0, 2] \times [0, 2]$
6	$[0, 100] \times [0, 100]$	17	$[-0.5, 0.5] \times [-0.5, 0.5]$
7	$[-10, 10] \times [-10, 10]$	18	$[0, 50] \times [0, 50]$
8	$[0, 1] \times [0.01, 1]$	19	$[-20, 20] \times [-20, 20]$
9	$[-2, 2] \times [-2, 2]$	20	$[0, 0.01] \times [0.01, 0.1]$
10	$[0, 5] \times [0.5, 5]$	21	$[-8, 8] \times [-8, 8]$
11	$[-1, 1] \times [0.1, 1]$		

References

[1] Wang, X., Fang, K.-T.: The effective dimension and quasi-monte carlo integration. Journal of Complexity **19**(2), 101–124 (2003)

[2] Tang, Y.: A note on monte carlo integration in high dimensions. The American Statistician **78**(3), 290–296 (2024)

[3] Kirk, D.B., Hwu, W.-m.W.: Programming Massively Parallel Processors: A Hands-on Approach, 3rd edn. Morgan Kaufmann, Cambridge, MA, USA (2016)

[4] Kalos, M.H., Whitlock, P.A.: Monte Carlo Methods, 2nd edn. WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim, Germany (2008)

[5] Kanzaki, J.: Monte carlo integration on gpu. The European Physical Journal C **71**(2), 1559 (2011)

[6] Zhou, Y., Li, Y., Wang, B., Li, Z.: ZMCintegral: A multi-GPU Monte Carlo integration package. https://github.com/DeepXHub/ZMCintegral (2019)

[7] Sakiotis, I., Arumugam, K., Paterno, M., Ranjan, D., Terzić, B., Zubair, M.: m-cubes: An efficient and portable implementation of multi-dimensional integration for gpus. In: International Conference on High Performance Computing, pp. 192–209 (2022). Springer

[8] Chow, G., Tse, A.H.T., Jin, Q., Luk, W., Leong, P.H.W., Thomas, D.B.: A mixed precision monte carlo methodology for reconfigurable accelerator systems. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), pp. 57–66 (2012)

[9] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al.: Mixed precision training. arXiv preprint arXiv:1710.03740 (2018)

[10] Katare, D., Leroux, S., Janssen, M., Ding, A.Y.: Approximating vision transformers for edge: variational inference and mixed-precision for multi-modal data. Computing **107**(3), 71 (2025)

[11] Higham, N.J., Mary, T.: Mixed precision algorithms in numerical linear algebra. Acta Numerica **28**, 347–414 (2019)

[12] Dijkstra, E.W.: Algol 60 translation: An algol 60 translator for the x1. Technical report, Stichting Mathematisch Centrum (November 1961)

[13] Berntsen, J., Espelid, T.O., Genz, A.: An adaptive algorithm for the approximate calculation of multiple integrals. ACM Transactions on Mathematical Software **17**, 437–451 (1991)

[14] Wilt, N.: The CUDA Handbook: A Comprehensive Guide to GPU Programming, 1st edn. Addison-Wesley Professional, Boston, MA, USA (2013)

[15] Caflisch, R.E.: Monte carlo and quasi-monte carlo methods. Acta Numerica **7**, 1–49 (1998)