PAGANI & MCUBES: PARALLEL ADAPTIVE GPU ALGORITHMS FOR NUMERICAL INTEGRATION

Ioannis Sakiotis (isaki001@odu.edu), Kamesh Arumugam, Marc Paterno, Desh Ranjan, Balsa Terzic, Mohammad Zubair
INTRODUCTION

- Applications: parameter estimation, simulation of beam dynamics, risk management, ray-tracing
- Probabilistic, deterministic
- Monte Carlo, Quadrature
- No algorithm guarantees the accuracy of its estimated results
- Specify desired accuracy (relative error tolerance, digits of precision, etc.)
- Goal: Bring efficient adaptive quadrature and Monte Carlo to GPUs
OUTLINE

• Two integration algorithms
  • PAGANI
  • mCubes
• Performance
• Interface/ease of use
• Execution platform portability
  • Kokkos, intel oneAPI
• Algorithm details
• Parallelization approach
• CUDA
• Deterministic using adaptive quadrature
• Weighted summation $\sum (w_i f(x_i))$
• Error-estimate
• Expect large initial error
• Apply multiple weighted summations in disjoint sub-regions to reduce error
• Based on the **CUHRE** integrator from the **CUBA** library
• Parallelize function evaluations
• Parallelize sub-region evaluation
• Parallelization targeted algorithmic changes
MCUBES

- CUDA
- Probabilistic using Monte Carlo integration
- Evaluate the integrand at random points
- Based on VEGAS integrator from the CUBA library
- Accelerate converge through importance and stratified sampling
- Parallelize function evaluations, random generation
- Algorithm is very parallelizable
- Targeted batching of operations
EXPERIMENTS

- Evaluate on various precision levels
- maximum relative error tolerance: 1.0e-3
- GPU: v100
- CUDA 11
- m-Cubes: plot averages from 100 runs per experiment
PERFORMANCE

- Limited libraries for parallel multi-dimensional numerical integration
- Cuba library for single threaded execution on CPU
- Parallelization yields orders of magnitude speedup
- One order of magnitude faster than existing GPU methods
PERFORMANCE

• Fastest GPU implementation
• Parallel deterministic
  • Two-phase CUHRE
• Parallel probabilistic
• gVegas
  • CUDA C++
  • original GPU-accelerated VEGAS
• ZMC
  • Python
  • Monte Carlo based
  • Stratified sampling + heuristic tree search

TEST SUITE

\[ f_1(x) = \cos \left( \sum_{i=1}^{8} i x_i \right) \]
\[ f_2(x) = \prod_{i=1}^{6} \left( \frac{1}{502} + (x_i - 1/2)^2 \right)^{-1} \]
\[ f_3(x) = \left( 1 + \sum_{i=1}^{d} i x_i \right)^{-d-1} \]
\[ f_4(x) = \exp \left( -625 \sum_{i=1}^{d} (x_i - 1/2)^2 \right) \]
\[ f_5(x) = \exp \left( -10 \sum_{i=1}^{d} |x_i - 1/2| \right) \]
\[ f_6(x) = \begin{cases} 
\exp \left( \sum_{i=1}^{d} (i + 4 x_i) \right) & \text{if } x_i < (3 + i) / 10 \\
0 & \text{otherwise}
\end{cases} \]

\[ f_7(x) = \left( \sum_{i=1}^{d} x_i^2 \right)^{11} \]
\[ f_8(x) = \left( \sum_{i=1}^{d} x_i^2 \right)^{15/2} \]
• Different algorithm than cuhre
• “Easy” integrals not worth the overhead
• Pagani generates more sub-regions
• More digits-of-precision
• More robust than two-phase cuhre
MCUBES

• Very similar to VEGAS
• Higher resolution importance sampling
• Consistently faster than gVegas and ZMC
• More digits-of-precision
• \( f_A(x) = \sin\left(\sum_{i=1}^{6} x_i\right) \)
• \( f_B(x) = \frac{1}{(x.02 \pi)^2} \exp\left(-\frac{1}{2e-6} \sum_{i=1}^{9} (x_i)^2\right) \)

<table>
<thead>
<tr>
<th>( f(x) )</th>
<th>algorithm</th>
<th>true value</th>
<th>Integral estimate</th>
<th>error estimate</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_A )</td>
<td>zmc</td>
<td>-49.165</td>
<td>-48.64740</td>
<td>1.98669</td>
<td>4.75e4</td>
</tr>
<tr>
<td></td>
<td>mcubes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( f_B )</td>
<td>zmc</td>
<td>1.0</td>
<td>0.99939</td>
<td>0.00133</td>
<td>8.3e3</td>
</tr>
<tr>
<td></td>
<td>mcubes</td>
<td></td>
<td></td>
<td></td>
<td>9.8e2</td>
</tr>
</tbody>
</table>
- f1: oscillatory
- f2: product peak
- f3: corner peak
- f4: gaussian
- f5: $C^0$ function
- f6: discontinuous
• Easy to use without parallel computing expertise
• Use functors to define integrands
• GPU compatible containers
HOW TO USE

• Available at https://github.com/marcpaterno/gpuintegration
• Pagani, mCubes are header only libraries
  • Optional build of tests utilizing the catch2 testing framework
  • Optional build of demos, executing closed form integrands on various accuracies
  • Required software: cMake
• Usage requirements
  • CUDA: nvcc to replace gcc
  • Kokkos: kokkos-kernels library
  • C++17
class Gaussian {
public:
  __device__ __host__
  double 
  operator()(double x, double y, double z, double w)
  {
    double beta = .5;
    return exp(-1.0 *
                (pow(25., 2.) * (pow(x - beta, 2) + pow(y - beta, 2) + pow(z - beta, 2) + pow(w - beta, 2))));
  }
};

Pagani<double, ndim> workspace;
Gaussian integrand;
double epsrel = 1.e-3, epsabs = 1.e-20;
workspace.integrate<Gaussian>(integrand, epsrel, epsabs);
double lows[] = {0., 0., 0., 0., 0., 0.};
double highs[] = {10., 10., 10., 10., 10., 10.};
quad::Volume<double, ndim> volume(lows, highs);

double epsrel = 1.e-3;
double epsabs = 1.e-20;

workspace.integrate< integrand_type >(integrand, epsrel, epsabs, volume);
STATEFUL INTEGRANDS

- Integrand is evaluated on the GPU
- Integrand is copied to GPU memory
- Integrand is invoked by multiple threads in parallel
- CUDA compatible computations and data-structures
- Use our types
  - quad::cudaArray
  - quad::Interp1D
  - quad::Interp2D
  - quad::Volume
CUSTOM TYPE FOR 2D INTERPOLATION

```cpp
std::array<double, 3> xs = {1., 2., 3.};
std::array<double, 2> ys = {4., 5.};
std::array<double, ny * nx> zs; // ← VALUES

std::array<double, 3 * 2> zs;
quad::Interp2D f(xs, ys, zs);

class Gaussian {
public:
    __device__ __host__
    double operator()(double x, double y,  double z,  double w) {
        return (f(x,y) * f(z,w));
    }

    quad::Interp2D f;
};
```
ARCHITECTURE PORTABILITY

• Initial implementation in CUDA
  • Best performance on NVIDIA GPUs
  • Limited to NVIDIA GPUs
  • Requires use of CUDA qualifiers
• Support for alternative platforms
  • Easier to maintain single code-base
  • Kokkos: completed initial implementation
  • oneAPI: in development
class GENZ_3_3D {
    public:
        double operator()(double x, double y, double z) {
            return pow(1 + 3 * x + 2 * y + z, -4);
        }
};
MCUBES IN KOKKOS

- Can maintain identical hierarchical parallelism as CUDA
- No major overhead
- Greater consistency in computationally intense parallel code
- Slightly different number of registers
- Slightly different cache configurations
- ~15% overhead in kernel code
- Sequential code can vary
## MCUBES (KOKKOS)

<table>
<thead>
<tr>
<th>integrand</th>
<th>CUDA time (ms)</th>
<th>Kokkos time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_A(x) = \sin(\sum_{i=1}^{6} x_i)$</td>
<td>829.760</td>
<td>968.880</td>
</tr>
<tr>
<td>$f_B(x) = \frac{1}{(\sqrt{0.02} \pi)^2} \exp\left( -\frac{1}{2e^{-6}} \sum_{i=1}^{9} (x_i)^2 \right)$</td>
<td>664.977</td>
<td>726.766</td>
</tr>
</tbody>
</table>
KOKKOS OVERHEAD IN PAGANI
ACCURACY

- Integrals known a priori
- Mathematica provided true integral
- Investigate results with various relative errors tolerances
- Evaluated the achieved relative error
INTEGRAND TEST SUITE

- Integrands represent challenging integrand families
- Product and corner peaks, oscillatory, Gaussian, etc.
- Typically, randomized parameters
- Fixed parameters

\[ f_1(x) = \cos \left( \sum_{i=1}^{8} i x_i \right) \]
\[ f_2(x) = \prod_{i=1}^{6} \left( \frac{1}{50^2} + (x_i - 1/2)^2 \right)^{-1} \]
\[ f_3(x) = \left( 1 + \sum_{i=1}^{d} i x_i \right)^{-d-1} \]
\[ f_4(x) = \exp \left( -625 \sum_{i=1}^{d} (x_i - 1/2)^2 \right) \]
\[ f_5(x) = \exp \left( -10 \sum_{i=1}^{d} |x_i - 1/2| \right) \]
\[ f_6(x) = \begin{cases} 
\exp \left( \sum_{i=1}^{6} (i + 4) x_i \right) & \text{if } x_i < (3 + i) / 10 \\
0 & \text{otherwise}
\end{cases} \]
\[ f_7(x) = \left( \sum_{i=1}^{d} x_i^2 \right)^{11} \]
\[ f_8(x) = \left( \sum_{i=1}^{d} x_i^2 \right)^{15/2} \]
• Evaluate error-estimate
• Claimed vs true relative-error
• Proximity to line indicates subdivision efficiency
MCUBES ACCURACY

- Error-estimate: standard deviation of integral estimate
- More samples and iterations required for more digits of precision
- Expected deviation in results
- Achieved relative error is consistent with user’s target relative error
- 100 runs per experiment
• Convergence status
• Integral and error estimates per iteration
• PAGANI verbose mode
  • Generated sub-region bounds
  • Generated sub-region integral and error estimates
• m-Cubes verbose mode
  • Function evaluations and generated numbers per sample
  • Importance sampling bin bounds per iteration
PER-ITERATION EVALUATION

• Estimates as measurement
• Consistency across iterations
• Overlapping error-bars
• 3-digits of precision
• Red line = true value
PER-ITERATION EVALUATION

- Estimates as measurement
- Consistency across iterations
- Overlapping error-bars
- 6-digits of precision
- Red line = true value
- More iterations
A CLOSER LOOK TO THE ALGORITHMS

- High level quadrature
- Parallelization schemes for parallel quadrature
- What other methods did differently
BACKGROUND: QUADRATURE

- Weighted summation $\sum(w_i f(x_i))$
- Error-estimate
- Expect large initial error
- Apply weighted summation in sub-region to reduce error
- Highly parallelizable
- Number of points grows exponentially with the number of dimensions
- Uniform split and sub-region evaluation infeasible
- Cuhre
  - $2^n + \Theta(n^3)$ functions evaluations (n-dimensions)
  - Attractive option for low/mid dimensional integrands
  - Priority-queue

https://mathematica.stackexchange.com/questions/222184/whats-the-most-difficult-multidimensional-integral-that-mathematica-has-solved
NAÏVE PARALLELIZATION
BACKGROUND: TWO-PHASE CUHRE

- GPU-targeted algorithm
- Assign each sub-region to a processor
- **Applies the Cuhre algorithm on each sub-region in parallel**
- Utilizes pre-processing Phase 1
  - Generate sufficiently large workload (# of sub-regions)
  - Load-balancing
- No synchronization between processors
- Local termination
TWO-PHASE CUHRE

- Utilization of sequential algorithm by parallel processors
- Poor load-balancing when high-precision
- Local data-structures
- Unknown global state (unless global sync.)
- Local termination
Parallel algorithm designed for massively parallel architectures
Avoid sequential scheme
Sub-divide all sub-regions
Filtering instead of sorting
Green/yellow = accurate enough
Avoid synchronization after isolated processing
Uniform workload
Bound by memory
Algorithm Description

- Initial uniform-split
- Parallel sub-region evaluation
- Two-level error-estimate
- Relative error classification
  - Finished/active
- Summations
- Termination conditions
- Conditional threshold classification
- Filtering
- Split all active regions

Algorithm 2 PAGANI Algorithm

```plaintext
1: procedure PAGANI(f, n, b[n], τ_rel, τ_abs) 
2:   R₀ ← b 
3:   s ← d^n 
4:   H ← UNIFORM-SPLIT(R₀, d) 
5:   A[1 : s] ← 1 
6:   V[1 : s], E[1 : L], K[1 : s] ← 0 
7:   V_p[1 : s], E_p[1 : s] ← 0 
8:   v, e, v_f, e_f ← 0 
9:   for it : it max do 
10:      V, E, K ← EVALUATE(H) 
11:         E ← TWO-LEVEL-ERROR(V, E, V_p, E_p) 
12:         A ← REL-ERR-CLASSIFY(V, E, A) 
13:         v ← SUM(V) 
14:         e ← SUM(E) 
15:         if \( \frac{v+e_f}{v+v_f} \leq \tau_{rel} \) or \( e + e_f \leq \tau_{abs} \) then 
16:            return v + v_f, e + e_f 
17:         A ← THRESHOLD-CLASSIFY (A, E, v + v_f, e + e_f, v, e, s) 
18:         v_f ← v - SUM(V · A) + v_f 
19:         e_f ← e - SUM(E · A) + e_f 
20:         H, V, E, L ← FILTER(H, V, E, A) 
21:         V_p ← V, E_p ← E 
22:         H ← SPLIT(H, K) 
23:         s ← 2s 
```
SUB-REGION CLASSIFICATION

- Why classify finished/active?
  - Only keep active regions in memory
  - Use relative error for global termination and finished/active classification
  - Regions with small estimates may not satisfy relative error termination
  - Remove regions that don’t contribute “significantly”

- Aggressive filtering
  - How to define “significantly”?  
  - Finished regions irrecoverable for performance
  - Balance finished and active estimates
  - Pick a threshold (initially the average) and adapt until criteria are met
  - Criteria: conserved memory, finished vs. active ratio
  - Perform if memory exhaustion or convergence of significant digits
PERFORMANCE: HEURISTIC SEARCH FILTERING

![Graphs showing time (ms) vs. user-specified digits of precision for 5D f4, 8D f4, and 8D f5.]

- Filtering
  - Mem-exhaustion
  - No filtering
  - PAGANI
PAGANI CHARACTERISTICS

- All operations utilize parallelization
  - Function-evaluations within sub-region
  - Sub-region evaluations
- No use of sequential algorithm
- No persistence in region-processor mapping
- Global data-structure
- Global state through reduction
- Implicit synchronization
- Limited by memory
MCUBES

- Parallel Vegas
- Importance sampling
  - Uses bins on each dimensional axis to draw sample
  - Adjust bin bounds based on sample point contribution
  - High contribution leads to smaller bins
  - Identical update contribution in each dimensional axis
- Stratified sampling
  - Samples separate parts of integration space
  - Helps issues with high contributions along dimensional axis diagonals
• 4D integrand
• Peaks are at \( x = [0.33, 0.33, 0.33, 0.33], y = [0.67, 0.67, 0.67, 0.67] \)
• Bins concentrate on all peaks
• Additional phantom peaks lead to wasted sampling

https://vegas.readthedocs.io/en/latest/background.html#adaptive-stratified-sampling
IMPORTANCE SAMPLING

BIN 0  BIN 1  BIN 2  BIN 3

larger intervals

bins
**MONTE CARLO – IMPORTANCE SAMPLING**

- mCubes uses constant memory
- Limited by maximum samples and iterations
- Step-wise function approximates distribution
- Difficult to capture when no alignment with axis
- Greater efficiency when contribution can be localized
MONTE CARLO – IMPORTANCE SAMPLING

- mCubes uses constant memory
- Limited by maximum samples and iterations
- Step-wise function approximates distribution
- Difficult to capture when no alignment with axis
- Greater efficiency when contribution can be localized
MCUBES-1D

- Applicable for fully symmetrical functions
- Maintains a single set of bins for all axes
- Reduces the number of expensive atomic operations
Algorithm 2 \(m\)-CUBES

1: procedure \(m\)-CUBES(\(f, d, n_b, maxcalls, L, H, itmax, ita, r\))
2: \(I, E \leftarrow 0\) \textcolor{red}{\triangleright} Integral/Error estimate
3: \(g \leftarrow (\text{maxcalls}/2)^{1/d}\) \textcolor{red}{\triangleright} Number of intervals per axis
4: \(m \leftarrow g^d\) \textcolor{red}{\triangleright} Number of cubes
5: \(s \leftarrow \text{SET-BATCH-SIZE}(	ext{maxcalls})\) \textcolor{red}{\triangleright} Heuristic
6: \(B[1 : d][1 : n_b] \leftarrow \text{INIT-BINS}(d, n_b)\) \textcolor{red}{\triangleright} Initialize bin boundaries
7: \(C[1 : d][n_b] \leftarrow 0\) \textcolor{red}{\triangleright} Bin contributions
8: \(p \leftarrow \text{maxcalls}/m\) \textcolor{red}{\triangleright} number of samples per cube
9: for \(i \leftarrow 0\) to \(ita\) do
10: \(r, C \leftarrow \text{V-SAMPLE()}\)
11: \(I, E \leftarrow \text{WEIGHTED-ESTIMATES}(r)\)
12: \(B \leftarrow \text{ADJUST-BIN-BOUNDS}(B, C)\)
13: \(\text{CHECK-CONVERGENCE()}\)
14: for \(i \leftarrow ita\) to \(itmax\) do
15: \(r \leftarrow \text{V-SAMPLE-NO-ADJUST()}\)
16: \(I, E \leftarrow \text{WEIGHTED-ESTIMATE}(r)\)
17: \(\text{CHECK-CONVERGENCE()}\)
- Thousands of stateful integrands
- Lookup-tables set at run-time
- State includes objects with Bilinear interpolation: 1D, 2D cases
- Prior use of CUBA
- Slow executions times, lack of convergence
- More than 2 orders of magnitude speedup compared to single-threaded CPU
- Multiple mpi-ranks, each with invoking an independent integration call
CONCLUSION

- Two algorithms for multi-dimensional numerical integration
- Most performant versions in CUDA for use with NVIDIA GPUs
- Kokkos, Intel oneAPI
- Designed for challenging intergrands
- Easy to use header only libraries
- Use requires no parallel programming expertise
- Seamless to transition from VEGAS/CUHRE
- Improved robustness and execution time
- Reliable error-estimation