Parallelization for Quadrature **Kyle Webster, 68017623**

Consider numerically evaluating the following, accurately, and quickly:

$$F = \int_{\alpha}^{\beta} f(x)\mathrm{d}x$$

Background and Motivation

There are three primary motivations for considering the parallelization of numerical integration, speed-up capabilities offered by parallel computation, the prevalence of quadrature in numerical methods, and the clear path to parallelizing the algorithms. Foregoing super-computers (or high-power computing facilities), modern consumer computer systems are more frequently containing high end Graphics Processing Units (GPUs), enabling massively parallel computations for methods that can take advantage of it. As the name suggests, GPUs are typically used for processing and displaying graphics and particularly enjoy the taste of linear algebra; however, thinking of GPUs as hierarchical parallel processing units, the applications of GPUs widens. These parallel computation units can house upwards of 2000 cores and execute tens of thousands of threads simultaneously which offers tremendous speed-up in processes, if high a degree of parallelization is possible. Quadrature is such a process with a high degree of parallelization. Additionally, the use of quadrature in other numerical methods, and prevalence of them, such as in FEM solvers and separable ODEs, increases the desirability for quick accurate numerical evaluations of integrals. Lastly, quadrature is a particularly nice target for parallelization as the algorithms depend solely upon function evaluations on a defined grid. Meaning, a GPU can simultaneously compute the expensive part of quadrature leading to high speed-up.

The hardware used for computations is relevant when considering speed-up in methods. Here, a computer running 64 bit Windows 11, an AMD Ryzen 9 4900HS (8-cores at max 3GHz), and an NVIDIA GeForce RTX 2060 (1920 CUDA Cores at max 1.185GHz) was used. These specifications represent a high quality modern computer, but lacks the huge computing capabilities of super-computing facilities. Hence, the results acquired are considered as a demonstration of the potential speed-up for larger systems as the total time for the methods to complete are nonetheless reasonable enough for the computations to be done in parallel or not.

Method and Parallelization

The chosen method to initially investigate is the compound Trapezoidal method. It approximates F by splitting the interval $[\alpha, \beta]$ into N subintervals with $N + 1$ points. On each subinterval, the area under the function is approximated to be the trapezoid formed by the x-axis, and the function values at the edges of the subinterval. That is,

$$F = \sum_{i=1}^{N} \frac{x_i - x_{i-1}}{2}(f(x_i) + f(x_{i-1})) + \frac{h^2 K_2}{12}$$

Where K_2 is a value of the second derivative on the entire interval. This mean, Trapezoidal Quadrature is a second order method (which will be important later). Simplifying by considering a grid of constant spacing $h = x_i - x_{i-1}$ for all i, the approximation can be compactly written as

$$F \approx h\left(\frac{f(\alpha) + f(\beta)}{2} + \sum_{i=1}^{N-1} f(x_i)\right)$$

The parallelization is done by assigning the GPU to compute the function evaluations in parallel.

In the implementation here, the summation is still done serially.[1] Therefore, the speed-up potential of parallelizing Trapezoidal quadrature in this implementation is a result of reducing the time spent on each loop iteration in exchange for an overhead of computing the function values and memory access.

An application to demonstrate the potential use of such a speed-up is Richardson extrapolation (which must be done on a regular grid). Considering the error of Trapezoidal Quadrature is second order, a forth order accurate method can be achieved by comparing the results that were computed at two different grid refinements. Denoting the Trapezoidal Quadrature on a grid of size h by $T(h)$, we have

$$F = \frac{4}{3}T(h/2) - \frac{1}{3}T(h) + \mathrm{O}(h^4)$$

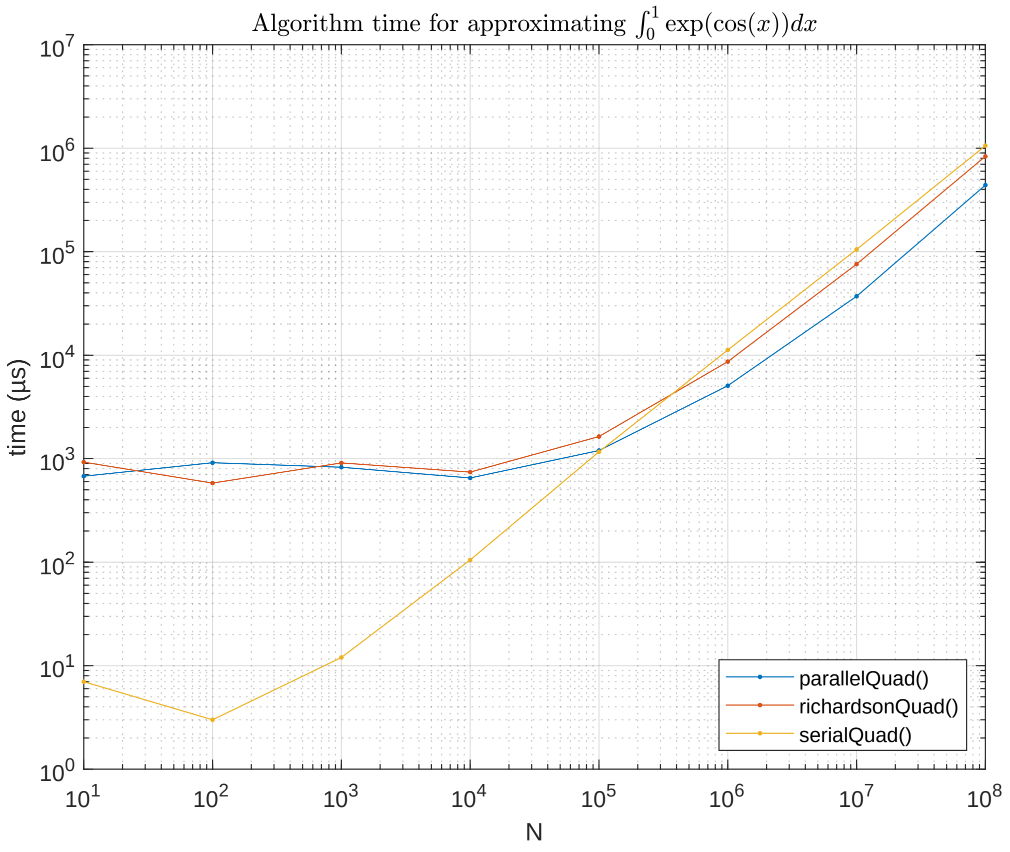
Hence, a higher order approximation can be computed for a single set of function evaluations on a grid of size $h/2$, and summed over $i = 0, ..., 2N$. For loop durations that are negligible compared to the parallel function evaluation time, the Richardson Extrapolation method should take comparable time to the simple Trapezoidal case. However, when the loop duration dominates, it is expected to take twice as long.

Results

The parallel trapezoidal, parallel Richardson Extrapolation, and serial trapezoidal methods were implemented and run for grids of number of intervals $\texttt{N} = [\texttt{1e1}, \texttt{1e2}, \texttt{1e3}, \texttt{1e4}, \texttt{1e5}, \texttt{1e6}, \texttt{1e7}, \texttt{1e8}]$ and two functions. First,

$$F = \int_0^1 e^{\cos(x)} \mathrm{d}x$$

Producing,

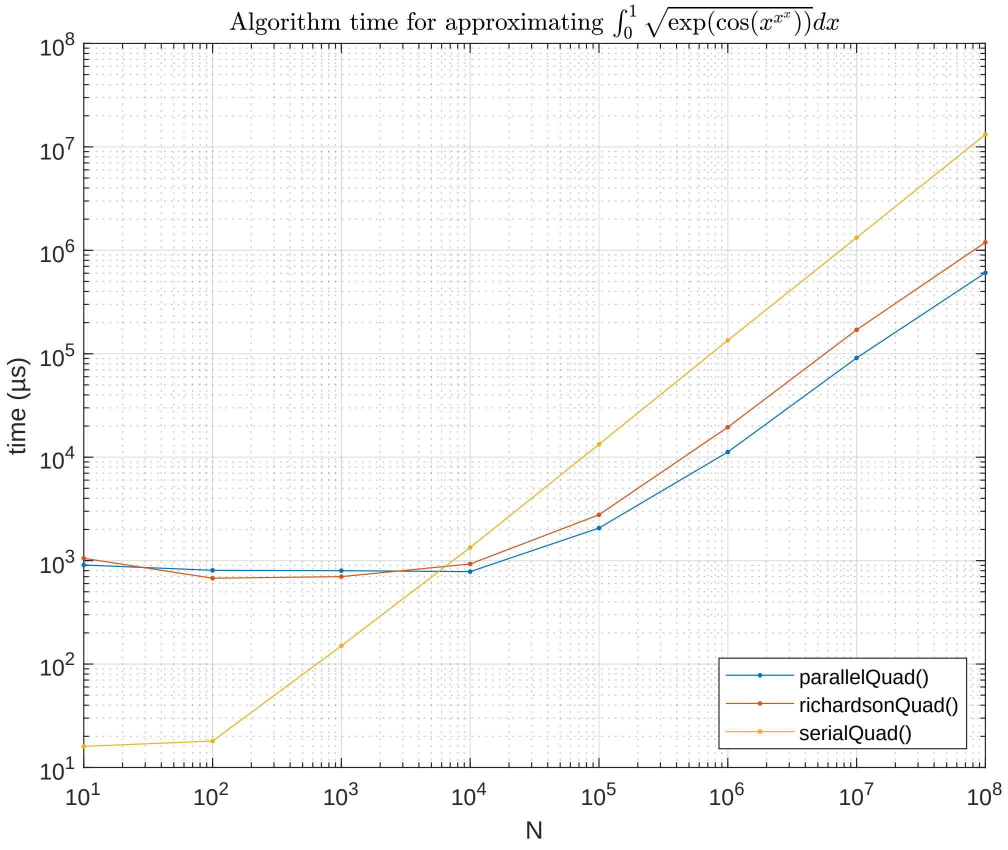


[1]This is because the installation of CUDA did not recognize the `atomicadd`$(\cdot,\cdot)$ function for parameters of type `double`, which is what would be used to allow for correct memory writing in parallel.

Which demonstrates the expected behavior of the parallel trapezoidal and parallel Richardson Extrapolation methods. It is clear that the serial implementation is more desirable in this case so long as less than 8 digits of precision are required – 8 digits of precision was achieved at approximately $N = 1 \cdot 10^5$. Secondly, a much more complex function was considered:

$$F = \int_0^1 \sqrt{\exp(\cos(x^{x^x}))}\mathrm{d}x$$

Producing,



Due to the much higher computational cost of the integrand, the speed-up is larger, on the order of 10 times faster for large N.

The conclusion that should be drawn from this investigation is that parallelization of quadrature should be used when the integrand is complicated and/or a high degree of accuracy (12 or more digits) is desired.

The important code used to implement the above methods is included in the appendix below, while the entire code is uploaded with the submission and available on github.

— Appendix —

paratrap.cu

```
#include<iostream>
#include<iomanip>
#include<math.h>
#include<chrono>

// CUDA includes
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <cuda.h>
#include <cuda_runtime_api.h>

// Number of threads per block
int BLOCK_SIZE = 256;

// GPU code for computing the function values on the grid points
__global__ void f_eval(double* f, double h, int n)
{
  // thread number
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i <= n) {
    // set x value for i
    double x = i * h;

    // compute the function value at x
    f[i] = exp(cos(x));
  }
}


// Trapezoidal Rule in parallel
double parallelQuad(const double alpha, const double beta, const int N)
{
  std::cout << "Computing integral in parallel\n";

  // Set timer start
  auto start = std::chrono::high_resolution_clock::now();

  // Compute interval size
  const double h = (beta - alpha) / N;

  // allocate host memory for function evaluations on grid
  const int ARRAY_BYTES = sizeof(double) * (N + 1);
  double * h_farray = new double [N + 1]; // limited to size < 1 million bc of stack size

  // allocate GPU memory for function evaluations on grid
  double* d_farray;
  cudaMalloc(&d_farray, ARRAY_BYTES);
  cudaMemset(d_farray, 0, ARRAY_BYTES);

  // compute number of blocks required and launch kernal for each point in the grid
  int num_blocks = (N / BLOCK_SIZE) + 1;
  f_eval <<<num_blocks, BLOCK_SIZE>>> (d_farray, h, N);

  // copy back the array from GPU memory, then free GPU memory
  cudaMemcpy(h_farray, d_farray, ARRAY_BYTES, cudaMemcpyDeviceToHost);
  cudaFree(d_farray);
  // Compute approximation and remove memory allocation
  double intgrl = 0.5 * (h_farray[0] + h_farray[N]);
  for (int k = 1; k < N; k++) {
    intgrl += h_farray[k];
  }
  intgrl *= h;
  delete[] h_farray;

  // Set timer end, compute duration
  auto end = std::chrono::high_resolution_clock::now();
  auto duration = std::chrono::duration_cast<std::chrono::microseconds> (end - start);


  // Print results to console
  std::cout << "Integral␣is␣approximately:␣" << std::fixed << std::setprecision(16) << intgrl << "\n
  std::cout << "Computed␣in:␣" << duration.count() << "␣(microseconds)\n\n";

  // Return intgrl value
  return intgrl;
}

// Richardson Extrapolation in parallel
double richardsonQuad(const double alpha, const double beta, const int N)
{
  std::cout << "Computing␣in␣parallel␣with␣Richardson␣Extrapolation\n";

  // Set timer start
  auto start = std::chrono::high_resolution_clock::now();

  // Compute interval size and half interval size
  const double h = (beta - alpha) / N;
  const double h_upon2 = 0.5 * h;

  // allocate host memory for function evaluations on h_upon2 sized grid
  const int ARRAY_BYTES = sizeof(double) * (2 * N + 1);
  double* h_farray = new double[2 * N + 1];

  // allocate gpu memory for function evaluations on grid
  double* d_farray;
  cudaMalloc(&d_farray, ARRAY_BYTES);
  cudaMemset(d_farray, 0, ARRAY_BYTES);

  // compute number of blocks required and launch kernal for each point in the grid
  int num_blocks = (2 * N / BLOCK_SIZE) + 1;
  f_eval<<<num_blocks, BLOCK_SIZE>>>(d_farray, h_upon2, 2 * N);

  // copy back the array from GPU memory, then free GPU memory
  cudaMemcpy(h_farray, d_farray, ARRAY_BYTES, cudaMemcpyDeviceToHost);
  cudaFree(d_farray);

  // Compute approximations on both grid levels and remove memory allocation
  double trap_h_upon2 = 0.5 * (h_farray[0] + h_farray[2 * N]);
  double trap_h = trap_h_upon2;
  for (int k = 1; k < 2 * N; k++) {
    trap_h_upon2 += h_farray[k];
    if (k % 2 == 0) {
      trap_h += h_farray[k];
    }
  }
  double intgrl = (4 / 3) * (h_upon2 * trap_h_upon2) - (1 / 3) * (h * trap_h);
  delete[] h_farray;
  // Set timer end, compute duration
  auto end = std::chrono::high_resolution_clock::now();
  auto duration = std::chrono::duration_cast<std::chrono::microseconds> (end - start);

  // Print results to console
  std::cout << "Integral is approximately: " << std::fixed << std::setprecision(16) << intgrl << "\n'
  std::cout << "Computed in: " << duration.count() << " (microseconds)\n\n";

  return intgrl;
}
```

serialtrap.cpp

```
#include<iostream>
#include<iomanip>
#include<chrono>
#include<math.h>


// Nonparallelized Trapezoidal rule quadrature for f(x) = exp(cos(x)) on alpha,beta with N intervals
double serialQuad(const double alpha, const double beta, const int N)
{

  std::cout << "Computing integral in series\n";

  // Set timer start
  auto start = std::chrono::high_resolution_clock::now();

  // Compute h
  double h = (beta - alpha) / N;

  // Compute approximation
  double intgrl = 0.5 * (exp(cos(alpha)) + exp(cos(beta)));
  for (int k = 1; k < N; k++) {
    intgrl += (exp(cos(h * k)));
  }
  intgrl *= h;

  // Set timer end, computer duration
  auto end = std::chrono::high_resolution_clock::now();
  auto duration = std::chrono::duration_cast<std::chrono::microseconds > (end - start);

  // Print results to console
  std::cout << "Integral is approximately: " << std::fixed << std::setprecision(16) << intgrl << "\n'
  std::cout << "Computed in: " << duration.count() << " (microseconds)\n\n";

  return intgrl;
}
```