Parallel Romberg Integration

Iresh Agrawal, Swayambhu N R
Department of Computational and Data Sciences
Indian Institute of Science, Bangalore, India
{ireshagrawal, swayambhunath93}@gmail.com

***Abstract*—In numerical analysis, Romberg's method is used to estimate definite integrals by applying Richardson extrapolation repeatedly on the trapezoidal rule. The estimates generate a triangular array. Highly accurate integrals can be computed using Romberg integration which requires huge computations. This paper focuses on efficient parallelisation of Romberg integration using shared memory and message passing paradigms.**

I. Introduction

Numerical integration has a wide range of application in modeling various physical systems. Hence computing accurate integral is the key for modeling various complex systems. Among various numerical integration techniques, Romberg integration is one of the most accurate and efficient method. For desired accuracy, the order of computation is fairly high. Using efficient parallelism of computation, the runtime of the integration can be curbed down without trading off with the accuracy (theoretically).

In this paper, an efficient parallel algorithm for large range integration using Romberg Integration method with an order of relative error about 10^{-10} is proposed. We have implemented the proposed algorithm in various parallel platforms (GPU using CUDA, MPI and openMP-CUDA hybrid) and compared the run-time with a parallel Simpson's method, MATLAB and sequential version by keeping the order of relative error same for all the methods.

II. Related Work

[3] implements Simpson Cumulative Integration on GPU, which is a parallel work in this field. [3] calculates cumulative integrals, which is not exactly the main purpose of this paper. So we have used their algorithm and optimized it for computing integrals. We have compared our algorithm with the optimized version of the algorithm presented in [3] keeping the relative error of the same order.

III. Methodology

In this paper we have developed two parallel algorithms for numerical integration using Romberg method. The Romberg method essentially uses Richardson extrapolation repeatedly on the trapezoidal rule. For a definite integration of the form:

$$S(b) = \int_a^b f(x)\,dx$$

the trapezoidal rule approximates the integral as:

$$S(b) = \frac{h}{2}[f(a)+2\sum_{j=1}^{n-1} f(x_j)+f(b)]+k_1h^2+k_2h^4+k_3h^6+\cdots$$

$$\Rightarrow S(b) = R_n + k_1h^2 + k_2h^4 + k_3h^6 + \cdots ....(i)$$

By doubling the number of panels:

$$S(b) = R_{2n} + k_1(\frac{h}{2})^2 + k_2(\frac{h}{2})^4 + k_3(\frac{h}{2})^6 + \cdots ....(ii)$$

4*(ii) - (i) gives:

$$3S(b) = 4R_{2n} - R_n - k_2\frac{3h^4}{4} - k_3\frac{15h^6}{16} - \cdots$$

Doing some mathematical manipulations the order of accuracy is increased. In general the Romberg integration is done using Romberg table which can be generalized as follows:

$$R_{m,j} = \frac{4^jR_{m,j-1} - R_{m-1,j-1}}{4^j - 1} + O(\frac{h}{2m})^{2j+2}......(iii)$$

where m = number of panel doublings and j = number of error removals.

A typical Romberg triangle goes as follows:

j / m	0	1	2	3
0	$R_{0,0}$			
1	$R_{1,0}$	$R_{1,1}$		
2	$R_{2,0}$	$R_{2,1}$	$R_{2,2}$	
3	$R_{3,0}$	$R_{3,1}$	$R_{3,2}$	$R_{3,3}$

[1] describes the Romberg integration in details.

The first column of the Romberg table uses the trapezoidal rule repeatedly with decreasing step size. With less step size the number of function evaluations using trapezoidal rule increases exponentially, typically requires $2^{rowsize-1} + 1$ function evaluations which is crucial for the accuracy desired. The following diagram explains the computations required:

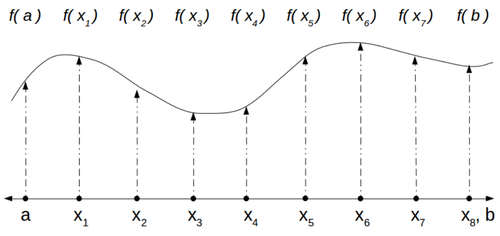

Fig. 1. Sample points from the function f(x)

For the above function the first column of the Romberg table will be as follows:

$$\begin{array}{|c|l}
\hline
 & {}^{h}/_{2}(f(a)+f(b)) \\
\hline
 & {}^{h}/_{4}(f(a)+2\times f(x_4)+f(b)) \\
\hline
 & {}^{h}/_{8}(f(a)+2\times [f(x_2)+f(x_4)+f(x_6)]+f(b)) \\
\hline
 & {}^{h}/_{16}\left(f(a)+2\times \left[\sum_{k=1}^{7} f(x_k)\right]+f(b)\right) \\
\hline
\end{array}$$

Thus the number of function evaluations increases exponentially with the number of rows. Also, from the above figures, it is clear that function evaluations needed for the last row of the first column is maximum and is equal to $2^{rowsize-1}+1$.

Hence the main bottleneck of the process lies in this large number of function evaluations for the first coloumn. By efficiently parallelizing the function evaluations and with efficient distribution of tasks, the run-time of Romberg integration (with reasonable accuracy) can be reduced.

We propose 2 algorithms for computing Romberg integration. *In order to integrate over a large range the entire range is divided into small intervals and each interval is integrated using Romberg's method separately. The result for all the intervals are added to get the final result.*

CUDA Algorithm-1

Each block calculates a part of integration using Romberg's method. Function evaluations are divided among the threads in a round-robin fashion and are stored in shared memory (minimizing bank conflicts). The following flowchart explains the algorithm in detail:

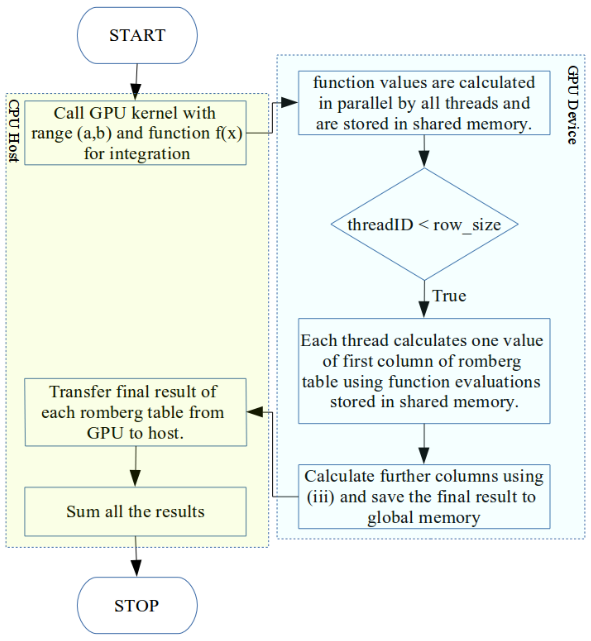
START
CPU Host
Call GPU kernel with range (a,b) and function f(x) for integration
GPU Device
function values are calculated in parallel by all threads and are stored in shared memory.
threadID < row_size
True
Each thread calculates one value of first column of romberg table using function evaluations stored in shared memory.
Calculate further columns using (iii) and save the final result to global memory
Transfer final result of each romberg table from GPU to host.
Sum all the results
STOP

Fig. 2. Flowchart for Algorithm-1

Use of shared memory to store function evaluations limits the row size for the first column of Romberg table. Here, load (to calculate the first column of Romberg table) is not uniform among threads, due to which runtime is not satisfactory. These problems have been addressed in the 2^{nd} algorithm.

CUDA Algorithm-2

Each block calculates a part of integration using Romberg's method. Function evaluations are divided among the threads in a round-robin fashion.

Each thread has its own array in register memory, which is of same size as the first column of Romberg table. The following diagram shows how each thread populates its own array and in the end all the arrays are summed in the shared memory.

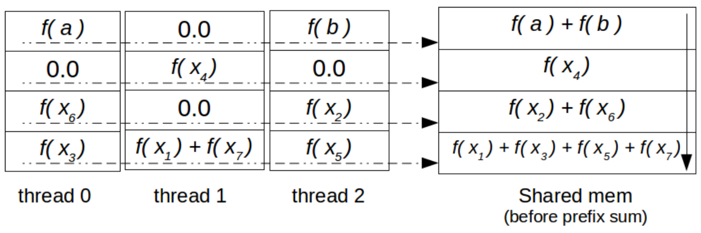

Fig. 3. Function evaluation in Algorithm-2

After summing up all the arrays in the shared memory, a prefix sum is performed to get the final first column of Romberg table. The following flowchart explains the algorithm in detail:

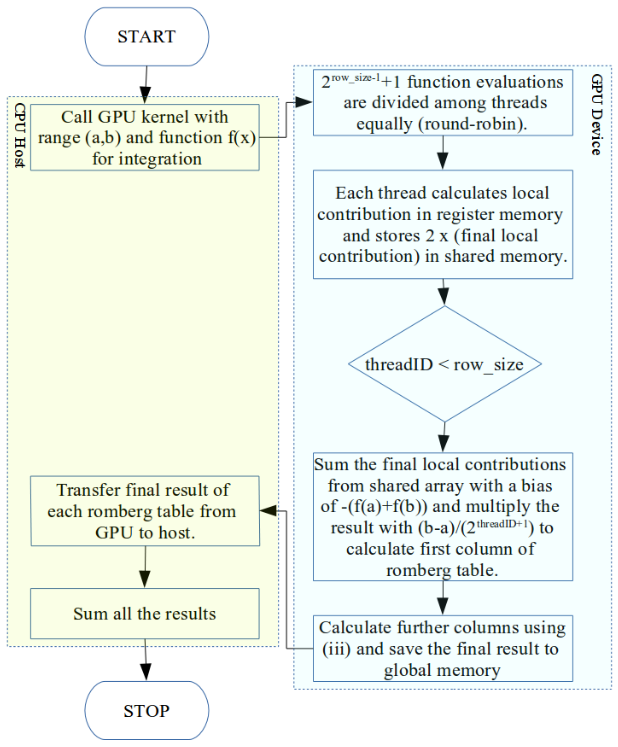
START
CPU Host
Call GPU kernel with range (a,b) and function f(x) for integration
GPU Device
2^row_size-1+1 function evaluations are divided among threads equally (round-robin).
Each thread calculates local contribution in register memory and stores 2 x (final local contribution) in shared memory.
threadID < row_size
Sum the final local contributions from shared array with a bias of -(f(a)+f(b)) and multiply the result with (b-a)/(2^threadID+1) to calculate first column of romberg table.
Calculate further columns using (iii) and save the final result to global memory
Transfer final result of each romberg table from GPU to host.
Sum all the results
STOP

Fig. 4. Flowchart of Algorithm-2

Hybrid CUDA-OpenMP Implementation

Number of romberg tables to be calculated is divided in two parts (1:3).

We have experimentally observed that function evaluations are much faster in GPU due to large number of cores available. By taking advantage of this, first column of $1/4^{th}$ of total romberg tables are calculated in GPU global memory using algorithm-2's logic.

Using two CUDA streams,

1) calculation of $3/4$ of romberg tables in GPU and
2) transfer of first columns of $1/4^{th}$ of total romberg tables to CPU

is done concurently.

While GPU calculates results for $3/4^{th}$ of romberg tables, OpenMP is used on host to compute result via (iii) using columns received by GPU.

MPI Implementation

Algorithm-2 has been implemented in MPI. Each processor computes a part of integration using romberg's method. The final result is obtained by reducing the result (using MPI_Reduce) from each processor.

IV. Experiments and Results

The experiments are performed keeping the worst case relative error of the order of 10^{-10} for all the methods. We also implemented reference and the sequential method for comparison purposes. CUDA, MPI and hybrid CUDA-OpenMP implementations has been compared with the reference method, MATLAB and sequential implementation for the following type of integration using various values for x:

$$S(x) = \int_0^x f(x)\,dx$$

A. *Experiment Setup*

All codes were ran on the Turing cluster at CDS. The CPU and GPU specifications are as given below

- turing-gpu (for CUDA and hybrid CUDA-OpenMP):
  - CPU : Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
  - GPU : NVIDIA Tesla K40m with 2880 CUDA cores divided among 15 SMXs.
- turing (for MPI):
  - CPU : Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
  - 7 more CPUs with same configuration are connected via message passing interface.

B. *Results*

The time required for different methods is given below:

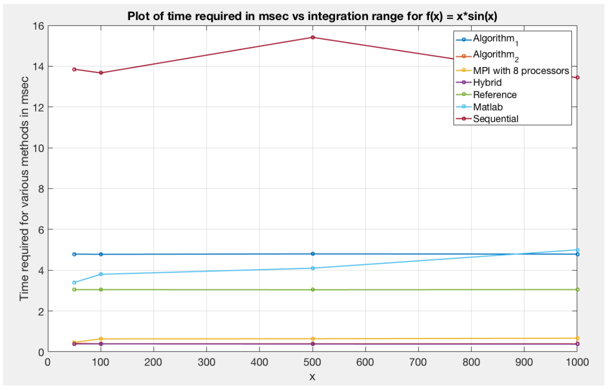

Fig. 5. Time required for various methods.

Speed-up achieved for our methods compared to the reference method:

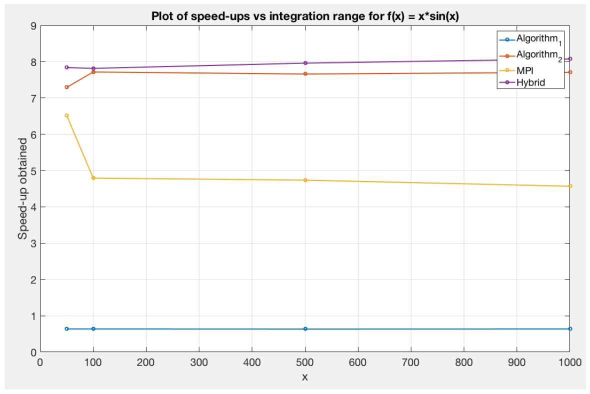

Fig. 6. Speed-up obtained for various methods.

V. Conclusions

- Keeping order of relative error same (10^{-10}) for all methods, considerable speed-up is achieved with second algorithm in all its parallel implementation due load balancing techniques presented in this paper.
- For MPI implementation, the speed-up is not satisfactory. This is mainly because the communication is the dominant factor over computation here. Lowest runtime was acheived with 8 cores running on a single node.
- The maximum speed-up is obtained from the hybrid (CPU+GPU) implementation.
- Order of accuracy can be easily increased in Simpson's method by increasing the number of function evaluations as there is no error prone numerical divisions; whereas in our implementation increasing number of rows of romberg table (i.e., more function evaluations) does not necessarily increase the accuracy due to the divisions (by $4^j - 1$) involved at every step of the algorithm.

VI. References

[1] Numerical Analysis by Richard L. Burden and J. Douglas Faires, sec-4.5 PP-213-220, 9th edition.

[2] Ali YAZICI, ”The Romberg-like Parallel Numerical Integration on a Cluster System”, 2009 24th International Symposium on Computer and Information Sciences

[3] Wayan Aditya Swardiana, Taufiq Wirahman and Rifki Sadikin, ”An Efficient Parallel Algorithm for Simpson Cumulative Integration on GPU” - 2015 Third International Symposium on Computing and Networking (CANDAR).