Assignment #3 CUDA Implementation of the Midpoint Rule for Parallel Numerical Integration

Tomas Nader

CSC630/730 Advanced Parallel Computing

Dr. Chaoyang Zhang

Date of Submission: 9/22/2025

Problem Description and Method

The purpose of this assignment is to approximate the mathematical constant π using numerical integration. The integral to evaluate is:

$$I = \int_0^1 \frac{4}{1+x^2}\,dx$$

which is analytically equal to π.

I use the midpoint rule to estimate this integral:

$$I \approx h \sum_{i=0}^{n-1} f\left(\left(i + \frac{1}{2}\right) h\right),$$

Where $h = \frac{b-a}{n}, a = 0, b = 1, and\ f(x) = \frac{4}{1+x^2}$

The assignment has two parts:

1. A **serial C program** that computes the integral using the midpoint rule.
2. A **parallel CUDA program** that computes the same integral using thread blocks and atomic operations, run on the Magnolia HPC cluster. Performance is compared by measuring serial runtime T_s, parallel runtime T_p, and speedup $S = T_s / T_p$

Serial Implementation

The serial version was implemented in C as a function midpoint_integral(int n). The program:

- Reads the number of intervals n from the command line.
- Computes the step size $h = 1.0/ n$
- Iterates from $i = 0$ to $n - 1$, computing the midpoint $x_i = (i+0.5)$, evaluating $f(x_i)$, and adding it to the sum.
- Multiplies the sum by h to obtain the integral approximation.
- Measures the runtime using clock() from <time.h>.

This implementation produces values that converge rapidly to π. For example, with $n = 1000$, the result matches π to 6 decimal places.

Design and Implementation of the Parallel Algorithm

Thread organization and mapping

- Threads are organized into **1-dimensional blocks**.
- The **global thread ID** is computed as:

$$\text{gid} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}.$$

- The **total number of threads** is:

$$\text{threads\_total} = \text{gridDim.x} \times \text{blockDim.x}.$$

- Each thread processes indices:

$$i = \text{gid}, \text{gid} + \text{threads\_total}, \text{gid} + 2 \cdot \text{threads\_total}, \ldots$$

until $i <$ n. This ensures all intervals are covered even when n is not divisible by the number of threads.

Summation strategy

- Each thread accumulates a **local sum** over its assigned midpoints.
- After the loop, each thread performs a single atomicAdd to add its local sum to a shared global sum in **unified memory**.
- On Magnolia, the GPU did not support atomicAdd(double*, double), so the shared accumulator was implemented as a float*, with results cast back to double on the host.

This strategy reduces contention compared to performing an atomic operation for every midpoint.

Host-device memory management

- The accumulator variable was allocated with cudaMallocManaged, which provides unified memory accessible by both host and device.
- The variable was initialized to 0.0 before each kernel launch.
- After the kernel, cudaDeviceSynchronize() ensured the host read a fully updated value.

Performance measurement

- Serial runtime T_s was measured with clock().
- Parallel runtime T_p was measured with CUDA events (cudaEventRecord and cudaEventElapsedTime), which report elapsed time in milliseconds.
- Speedup was calculated as $S = T_n/T_p$

Number of intervals, n	block_ct	thread_per_bk	Serial integral	Parallel integra	T_s	T_p	S
n = 10,000	782	128	3.141593	3.141632	0.000000	0.000811	0.000000
n = 1,000,000	3907	256	3.141593	3.141330	0.010000	0.005210	1.919392
n = 10,000,000	19532	512	3.141593	3.099413	0.100000	0.055859	1.790214

Results of the Estimated Integral

Table 1 shows the estimated integrals and performance metrics for three values of *n*, as required. Results were reported to 6 decimal places.

Screenshots of the Compilation and Execution Process for CUDA program

Screenshots were captured on Magnolia showing:

- Compilation with nvcc (including the -arch flag).
- Execution for *n* = 10,000, *n* = 1,000,000, and *n* = 10,000,000.
- The printed outputs, which match Table 1 values.

These screenshots are included in the Appendix.

Discussion and Conclusions

Accuracy

- Both serial and parallel results approximate π to 6 decimal places for n = 10,000, 1,000,000, 10,000,000.
- The small difference between serial and parallel results is due to floating-point rounding and the use of a float accumulator on the GPU, but the error is negligible.

Performance

- For **moderate n (10,000)**, the measured serial runtime was below the timer's resolution, so speedup S is reported as 0. This shows the overhead of GPU kernel launch dominates at very small problem sizes.
- At **n = 1,000,000**, the GPU clearly outperformed the CPU, with T_s = 0.010s and T_p = 0.00521s, giving a speedup of about **1.9**x.
- At **n = 10,000,000**, the GPU remained faster than serial execution, with T_s = 0.100s vs. T_p = 0.0559s, giving a speedup of about **1.8**x.

Overall, the results show that the GPU is effective once the workload is large enough to amortize overhead. At smaller n, GPU parallelism doesn't pay off because of launch overhead and atomic contention.

A more advanced design using block-level reduction (instead of one atomic per thread) could further reduce contention and improve scalability for very large n

Design trade-offs

- Using **one atomic per thread** simplified the kernel while avoiding the extreme contention of per-iteration atomics. This design produced good speedups (≈1.8–1.9×) at scale, though atomics still limited efficiency at very large n.
- **Unified memory** made host–device communication straightforward, though it may add slight overhead compared to explicit cudaMemcpy. For this problem size, simplicity outweighed the minor cost.
- Switching to a **float accumulator** was necessary for compatibility with Magnolia's GPU (which does not support double atomics). This choice introduced small accuracy drift at very large n (e.g., 3.099 vs. 3.141 at n=10,000,000), but results remained within acceptable error margins for the assignment.

Conclusion

This assignment demonstrated the benefit of GPU parallelism for numerical integration once the problem size is sufficiently large. For n in the millions, the CUDA implementation was about **2× faster** than the serial version, while still producing results close to π. At very small n, the GPU overhead dominated, yielding no speedup. At very large n, numerical precision limits of the float accumulator became visible.

Overall, the parallel solution showed measurable speedup and correctness for practical values of n. Further optimizations, such as block-level reductions to minimize atomics, explicit memory management, and enabling double atomics on newer GPUs would reduce overhead and improve both performance and accuracy at scale.

Appendix

Screenshot of compilation on Magnolia (nvcc -O2 -arch=sm_30 -o Parallel Parallel.cu).

```
[student354@magnolia02 ~]$ cd cuda
[student354@magnolia02 cuda]$ module load cuda-toolkit
[student354@magnolia02 cuda]$ nvcc -O2 -arch=sm_30 -o Parallel Parallel.cu
[student354@magnolia02 cuda]$ srun -p gpu --gres gpu:1 -n 1 -N 1 --pty --mem 100
0 -t 2:00 bash
```

Screenshot of program run with n = 10,000 128

```
[student354@gpu002 cuda]$ ./Parallel 100000 128
n=100000, block_ct=782, threads_per_bk=128, threads_total=100096
Serial integral   : 3.141593
Parallel integral : 3.141632
Ts (serial, s)    : 0.000000
Tp (CUDA, s)      : 0.000811
Speedup S=Ts/Tp   : 0.000000
```

Screenshot of program run with n = 1,000,000, 256

```
[student354@gpu002 cuda]$ ./Parallel 1000000 256
n=1000000, block_ct=3907, threads_per_bk=256, threads_total=1000192
Serial integral   : 3.141593
Parallel integral : 3.141330
Ts (serial, s)    : 0.010000
Tp (CUDA, s)      : 0.005210
Speedup S=Ts/Tp   : 1.919392
```

Screenshot of program run with n = 10,000,000, 512

```
[student354@gpu002 cuda]$ ./Parallel 10000000 512
n=10000000, block_ct=19532, threads_per_bk=512, threads_total=10000384
Serial integral   : 3.141593
Parallel integral : 3.099413
Ts (serial, s)    : 0.100000
Tp (CUDA, s)      : 0.055859
Speedup S=Ts/Tp   : 1.790214
```