Exploring Performance of CUDA and MPI Implementations for Quadrature Estimation

Jesty Sebastian (22555483)
Department of Computing and Mathematics
Manchester Metropolitan University
Manchester, UK
jesty.sebastian@stu.mmu.uk

Declaration of Originality

I, Jesty Sebastian, hereby declare that all work presented in this report, titled "Exploring Performance of CUDA and MPI Implementations for Quadrature Estimation," is my own.

I confirm that:

- Any external sources used in this report have been appropriately cited and referenced.

- The experiments, results, and analysis presented in this report are the outcome of my own work and efforts.

- I have not submitted this work, or any variation thereof, for any other academic assessment.

I understand the consequences of academic misconduct, including plagiarism, and I acknowledge that any violation of academic integrity will result in appropriate penalties as determined by Manchester Metropolitan University.

Signed: Jesty Sebastian

URL/link to the zipfile in MMU OneDrive area : HPC Assignment -Jesty Sebastian(22555483).zip

1. Introduction

The estimation of the area under a curve, also known as quadrature, is a fundamental numerical technique applied in many computing applications. This research explores and compares two quadrature estimation implementations: an MPI implementation for distributed computing and a CUDA code for GPU acceleration.

The MPI method uses distributed computing with ARCHER2 to split the workload among multiple processes, while the CUDA implementation makes use of GPU parallelism to speed up calculation. The performance of each will be compared and analyzed to assess each implementation's suitability under various circumstances.

2. Methodology:

2.1. CUDA Implementation

The goal of this study is to estimate the area under a curve using quadrature through the utilization of GPU acceleration. The problem involves the approximation of the integral of a given function by dividing the curve into multiple trapezoids and summing their areas. The CUDA implementation will be employed, making use of the parallel processing capabilities of the GPU [1]. The performance of the CUDA implementation will be explored by varying the number of threads per block, enabling an analysis of the impact of these factors on the execution time and the accuracy of the estimated integral. By distributing the workload among the GPU's threads and leveraging its high-performance architecture, significant speedup is expected to be achieved in comparison to a sequential CPU implementation[2].

The **"deviceQuery"** command is used to identify the physical GPU utilized. Result of the **"deviceQuery"** command is given in **Appendix 1.**

```
Device 0: "Quadro P2000"
  CUDA Driver Version / Runtime Version          11.7 / 11.7
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 5050 MBytes (5295046656 bytes)
  ( 8) Multiprocessors, (128) CUDA Cores/MP:     1024 CUDA Cores
  GPU Max Clock rate:                            1481 MHz (1.48 GHz)
  Memory Clock rate:                             3504 Mhz
  Memory Bus Width:                              160-bit
  L2 Cache Size:                                 1310720 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.7, CUDA Runtime Version = 11.7, NumDevs = 1, Devi
ce0 = Quadro P2000
Result = PASS
```

The GPU used for the CUDA implementation is a Quadro P2000 with CUDA Capability Major/Minor version number 6.1. It has 8 multiprocessors, each containing 128 CUDA cores, resulting in a total of 1024 CUDA cores. The GPU operates at a maximum clock rate of 1.48 GHz and has a memory clock rate of 3504 MHz. It has a total of 5050 MBytes (5.29 GB) of global memory available. The GPU supports a maximum of 2048 threads per multiprocessor and 1024 threads per block.

The programme offered (quad-cuda-ASSESS.cu) is a CUDA solution for calculating the area under a curve using quadrature. To improve performance, it makes use of parallel processing on a GPU. The integral of a given function is calculated by approximating the curve using trapezoids and adding their areas. The main function begins by calculating the number of quadrangles (numberQuads) and threads

per block (tpb). Default values are used if these settings are not provided as command-line arguments. The programme then allocates GPU RAM for storing intermediate results (deviceBlockArea). The calcMyArea kernel function is called with the number of blocks and threads per block given. Using the given function, each thread calculates and stores its respective sub-area under the curve in shared memory. The shared memory is reduced to obtain the sub-area for each block, and the results are copied back to CPU memory. Finally, the results are added together to yield the final integral value, and the execution time is calculated using the get_wtime function. The GPU's output and execution time are then displayed to the console. If there is no GPU present, corresponding message is displayed.

Compilation and Execution

The CUDA implementation is compiled using the nvcc compiler command:

nvcc quad-cuda-ASSESS.cu -o quad-cuda-ASSESS.exe

The compiled executable is then executed using the command

./quad-cuda-ASSESS.exe 4096000 [threads per block]

At runtime, two arguments are passed: the first is the number of rectangles, which is set to 4096000, and the second is the number of threads per GPU thread-block. The command is run for different numbers of threads per block ranging from 32 to 1024.

To explore the performance of the CUDA implementation, we vary the number of threads per block. For each configuration, the executable is executed with the chosen number of threads per block. The execution time and the value of the estimated integral are recorded for each configuration.

The timing of each implementation is measured using the wall clock timer function. Before the computation begins, the start time is recorded and after the computation completes, the end time is recorded. The command **“**`printf("WALL CLOCK Time: %f seconds\n",(finish-start));`**”** is used to find the difference between the start and end times and it provides the execution time for each implementation.

2.2.ARCHER2 Platform (MPI Implementation):

The goal of this research is to estimate the area under a curve using quadrature using MPI parallelization. The problem involves approximating the integral of a given function by dividing the curve into multiple trapezoids and summing their areas. The focus of this task is on the MPI implementation, which makes use of the ARCHER2 system's distributed computing capabilities[3].

The performance of the MPI implementation will be investigated by running the code on multiple nodes while keeping the total number of nodes to four. This limitation is consistent with the system's constraint of a maximum runtime of 20 minutes for each batch job. The provided code is specifically designed to estimate the integral using 4,096,000 rectangles and no additional arguments need to be passed at runtime.

The provided code (ASSESS-barrier.c) is an MPI solution for calculating the area under a curve using quadrature. It computes the integral of a given function by dividing the curve into trapezoids and adding the areas of the trapezoids. MPI is used in the code to parallelize multiple processes. The main function starts MPI, configures the variables, and distributes the workload among the MPI processes. Within its assigned range, each process computes a local sum of trapezoidal areas. The local sums are then reduced with MPI_Reduce to obtain the final integral value. The code also includes timing

functionality for measuring execution time against the wall clock. The root process (rank 0) prints out the computed integral value and the wall clock time for reporting purposes.

Compilation and Execution

1. The provided code (ASSESS-barrier.c) is copied to ARCHER2 from linux machine using command "**rsync ASSESS.c xxx@login.archer2.ac.uk:**" where xxx is replaced by the ARCHER2 username.
2. Logged into ARCHER2 using the SSH (Secure Shell) command to establish a secure connection to the ARCHER2 login node. The login node address is **login.archer2.ac.uk.**
3. Now the file ASSESS-barrier.c is there in home file system. For "work" file system, first environment variable called "WORK" is created. Then WORK for the current shell (and any child processes) is created by commands:
   export WORK=`echo $HOME | sed 's/home/work/'`
   echo $WORK
4. The code is compiled from the home file system using command:
   cc ASSESS-barrier.c -o ASSESS-barrier.exe
5. Because the job is run on compute nodes, the executable file is now moved to the work file system. Then navigated to the appropriate directory in the "work" file system that is ready to submit a job to the batch system.
6. Since the provided SLURM batch script is hardcoded for running the parallel job on different number of cores in a single Node (i.e. maximum of 128 cores), the script is edited to run upto 4 nodes (i.e. maximum of 512 cores) using the command:
   emacs -nw batch_mpi_scaling.sh
   The changes made in script are given in **Appendix 2**
7. The edited SLURM batch script **"batch_mpi_scaling_4Nodes"** is used to run the executable using the command:
   sbatch batch_mpi_scaling_4Nodes ASSESS-barrier.exe
8. Optimizing an MPI implementation brings benefits like scalability, performance improvement, efficiency increase, enhanced parallelism, and superior scalability. Optimized MPI code minimizes communication overhead, load balancing issues, and data distribution inefficiencies, resulting in faster computations and shorter time-to-solution. The executable is run in 0, 1, 2, 3 and Fast optimizations using command:
   sbatch -o[optimization flag] batch_mpi_scaling_4Nodes ASSESS-barrier.exe
9. The same steps are repeated for 1, 2 and 3 Nodes separately and compared the output. All the outputs are saved in the attached excel file.
10. A **slurm-[job-id]** file will be created in the respective directory during the submission of job. After completing the job, the file can be opened to see the outputs using command:
    cat slurm-[jobid]

3. Results and Discussion

3.1. CUDA Implementation

Thread Blocks	Threads per block	Execution Time	Integral Value
128000	32	0.030917	607847.545334
64000	64	0.03323	607847.545334
32000	128	0.031776	607847.545334
16000	256	0.031556	607847.545334
13213	310	0.033097	607847.545334
12800	320	0.03165	607847.545334
9846	416	0.033134	607847.545334
8000	512	0.032312	607847.545334
7062	580	0.036761	607847.545334
6400	640	0.033485	607847.545334
5565	736	0.03639	607847.545334
5185	790	0.040909	607847.545334
4923	832	0.037164	607847.545334
4414	928	0.039657	607847.545334
4096	1000	0.040187	607847.545334
4000	1024	0.039319	607847.545334

Table 1 - Timing results of CUDA Implementation

Several observations can be made based on the results. First, as the number of thread blocks decreases, so does the execution time. This is to be expected because fewer thread blocks mean fewer parallel computations, resulting in faster execution. However, this relationship is not strictly linear because other factors such as memory access patterns and GPU architecture are at play.

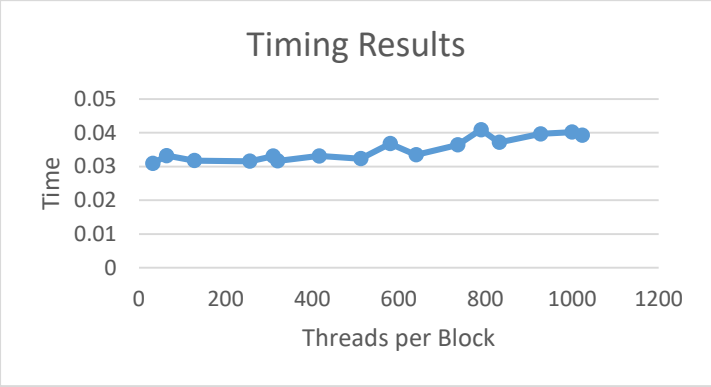

Figure 1 - CUDA Implementation

The execution time for thread configurations with fewer threads per block, such as 32, 64, and 128, ranges from 0.030917 to 0.031776 seconds. The execution time remains consistent at around 0.031556 to 0.033097 seconds as the number of threads per block increases, for example, to 256 and 310. This shows that increasing the number of threads per block up to a certain point has little impact on overall execution time.

However, as we increase the number of threads to 320, 416, 512, and 580, the execution time begins to show a slight upward trend, ranging from 0.03165 to 0.036761 seconds. This suggests that there may be an optimal value for the number of threads per block that provides the best performance for this particular CUDA programme.

Furthermore, we observe some inconsistencies in execution time for certain configurations, such as 310, 580, 790, and 1000 threads per block. These configurations have significantly longer execution times than the surrounding configurations, ranging from 0.037164 to 0.040909 seconds. This could be due to partial warp. In CUDA programming, threads are organized into warps, and the hardware

executes instructions on a warp basis. Warps consist of groups of 32 threads, which is the warp size in this case. If the total number of threads is not a multiple of 32, some threads may be grouped together in a partial warp. This can result in suboptimal performance because the hardware executes instructions on a per-warp basis. Partial warps may result in underutilization of GPU resources as well as additional overhead due to thread divergence within the warp. To achieve optimal performance, threads should be organized in multiples of the warp size (32 in this case) to ensure full utilization of the GPU's capabilities and maximize parallelism.

3.2.MPI Implementation

# Cores	times/secs	Sp	Ep %age	Integral Value
1	0.932798	1	100	607847.545334
2	0.488266	1.91	95	607847.545334
3	0.432639	2.15	71	607847.545334
4	0.361297	2.58	64	607847.545334
8	0.107689	8.66	108	607847.545334
16	0.055162	16.91	105	607847.545334
32	0.027545	33.86	105	607847.545334
64	0.013868	67.26	105	607847.545334
96	0.00939	99.33	103	607847.545334
128	0.007182	129.87	101	607847.545334
160	0.006064	153.82	96	607847.545334
200	0.004835	192.92	96	607847.545334
250	0.00409	228.06	91	607847.545334
256	0.004003	233.02	91	607847.545334
320	0.003342	279.11	87	607847.545334
384	0.002929	318.46	82	607847.545334
500	0.002567	363.38	72	607847.545334
512	0.002597	359.18	70	607847.545334

Table 2 - MPI Implementation Timing Results

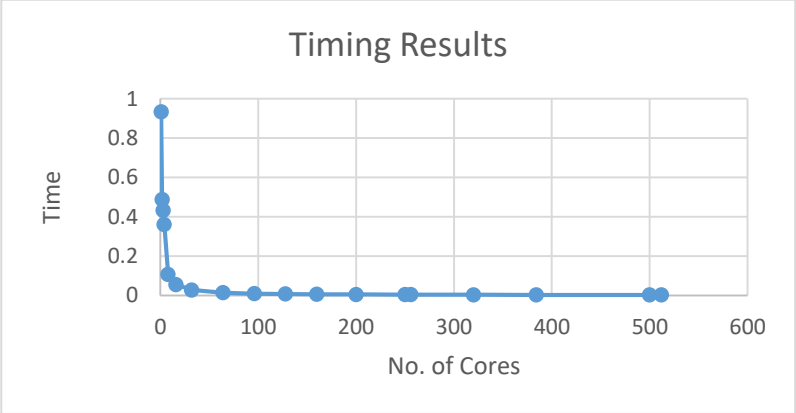

Figure 2 - MPI Implementation

The MPI implementation's timing results demonstrate the code's performance characteristics across different core configurations. The execution time decreases as the number of cores increases, indicating improved performance due to parallelization. The decreasing "times/secs" values demonstrate this. The speedup (Sp) values show how much faster parallel code runs when compared to sequential execution on a single core. The speedup generally increases with the number of cores, indicating that parallelization benefits the code. The efficiency percentage (Ep%age) is the ratio of the achieved speedup to the ideal speedup, which is 100% in the case of perfect scalability.

The efficiency percentages differ depending on the core configuration. The results show that the MPI implementation is scalable, with increasing speedup for the majority of core configurations. This

means that the code takes advantage of parallel processing, resulting in faster execution times as the number of cores increases. It should be noted that the efficiency percentages are consistently high, indicating that the parallel code performs well in comparison to the ideal speedup.

This implies that the code makes good use of available computing resources while minimizing overhead.

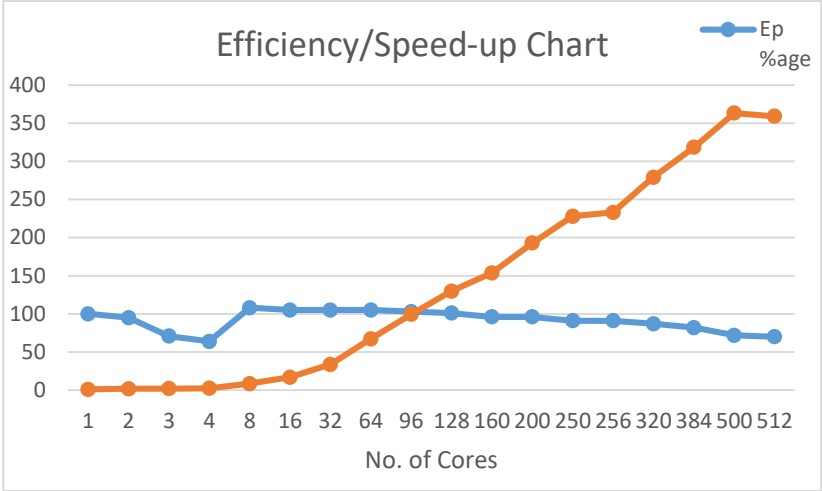

Figure 3 - Efficiency/Speed-up Chart

Furthermore, the results demonstrate instances (8 to 128 cores) of super linear efficiency, in which the achieved efficiency exceeds 100%. This implies that the parallel implementation benefits from factors that improve its performance above and beyond what Amdahl's law predicts. Improved caching, better load balancing, and algorithmic optimizations could all be reasons for this.

4. Comparison and Conclusions

As computational resources are increased (thread blocks/threads per block for CUDA, and cores for MPI), both implementations improve in performance. The CUDA implementation offers fine-grained control over thread blocks and threads per block, allowing for potential optimization and customization based on the characteristics of the problem. The MPI implementation takes advantage of parallel processing and is scalable, with increasing speedup values as the number of cores grows. Both implementations keep consistent integral values, indicating that the integrand is correctly estimated.

Overall, the CUDA implementation takes advantage of GPU parallelism and allows for the configuration of thread blocks and threads per block. The MPI implementation, on the other hand, demonstrates scalability and efficient utilization of computing resources across different core configurations. The choice between the two implementations would be influenced by factors such as available hardware, problem characteristics, and optimization needs.

References

1. Mark Harris (2017) An Even Easier Introduction to CUDA [Online] [Accessed on 11/04/2023] https://developer.nvidia.com/blog/even-easier-introduction-cuda/
2. Martin Heller (2022) What is CUDA? Parallel programming for GPUs [Online] [Accessed on 11/04/2023] https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html
3. Archer2 [Online] [Accessed on 11/04/2023] https://www.archer2.ac.uk/

Appendix 1

ad22555483@vega:~$ /usr/local/cuda-11.7/extras/demo_suite/deviceQuery
/usr/local/cuda-11.7/extras/demo_suite/deviceQuery Starting...

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Quadro P2000"

CUDA Driver Version / Runtime Version	11.7 / 11.7
CUDA Capability Major/Minor version number:	6.1
Total amount of global memory:	5050 MBytes (5295046656 bytes)
(8) Multiprocessors, (128) CUDA Cores/MP:	1024 CUDA Cores
GPU Max Clock rate:	1481 MHz (1.48 GHz)
Memory Clock rate:	3504 Mhz
Memory Bus Width:	160-bit
L2 Cache Size:	1310720 bytes
Maximum Texture Dimension Size (x,y,z)	1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
Maximum Layered 1D Texture Size, (num) layers	1D=(32768), 2048 layers
Maximum Layered 2D Texture Size, (num) layers	2D=(32768, 32768), 2048 layers
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and kernel execution:	Yes with 2 copy engine(s)
Run time limit on kernels:	Yes
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Alignment requirement for Surfaces:	Yes
Device has ECC support:	Disabled
Device supports Unified Addressing (UVA):	Yes
Device supports Compute Preemption:	Yes
Supports Cooperative Kernel Launch:	Yes
Supports MultiDevice Co-op Kernel Launch:	Yes
Device PCI Domain ID / Bus ID / location ID:	0 / 1 / 0

Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.7, CUDA Runtime Version = 11.7, NumDevs = 1, Device0 = Quadro P2000
Result = PASS

Appendix 2

The changes made in "**batch_mpi_scaling.sh**" script using **emacs -nw batch_mpi_scaling.sh**

1. The number of nodes (N) changed to 4 and number of cores (n) changed to 512 in SLURM job directive.

   #SBATCH -N 4 -n 512

2. A wide range of processes numbers are given to the loop that loop over various number of MPI processes. Also added the equations to calculate the speedup and efficiency. Now all the outputs are copying to $TIMING.

```
for PROCS in 1 2 3 4 8 16 32 64 96 128 160 200 250 256 320 384 500 512
do
  echo Running $EXE in directory $PWD on $PROCS MPI processes
  if [[ $PROCS -eq 1 ]] ; then
   # run 1 process separately to get the time for 1 process
   srun --ntasks=${PROCS} --cpu-bind=verbose,rank ./${EXE} | tee ${TMPFILE}
   time1=`grep seconds ${TMPFILE} | awk '{print $(NF-1)}'`
   speedup=`echo "scale=2; $time1 / $time1" | bc`
   efficiency=`echo "scale=2; $speedup / $PROCS * 100" | bc`
   echo $PROCS $time1 $speedup $efficiency >> $TIMING
  else
   srun --ntasks=${PROCS} --cpu-bind=verbose,rank ./${EXE} | tee ${TMPFILE}
   time=`grep seconds ${TMPFILE} | awk '{print $(NF-1)}'`
   # calculate efficiency and speedup and save in $TIMING
   time_diff=`echo $time1 - $time | bc`
   speedup=`echo "scale=2; $time1 / $time" | bc`
   efficiency=`echo "scale=2; $speedup / $PROCS * 100" | bc`
   echo $PROCS $time $speedup $efficiency >> $TIMING
  fi
done
```