ON MODERN OFFLOADING PARALLELIZATION METHODS: A CRITICAL ANALYSIS OF OPENMP

An Undergraduate Research Scholars Thesis

by

SCOTT CARLOS CARRIÓN

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Jeff Huang

May 2021

Major: Computer Engineering (Computer Science Track)

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Scott Carlos Carrión, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

On Modern Offloading Parallelization Methods: A Critical Analysis of OpenMP

Scott Carlos Carrión
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Jeff Huang
Department of Computer Science and Engineering
Texas A&M University

The very concept of offloading computationally complex routines to a graphics processing unit for general-purpose computing is a problem left wide open to the academic community, both in terms of application as well as implementation, with several different and popular interfaces exploding into popularity within the last twenty years. The OpenMP standard is among the elites in this category, standing as a parallelization interface that has stood the test of time. The goals that the inquiry presented herein seeks to answer are twofold: Firstly, we aim to assess the performance of common sorting algorithms parallelized and offloaded using OpenMP, offloaded to NVIDIA GPU hardware, and secondly, to critically analyze the programmer experience in using an implementation of the OpenMP standard (again, with offloading to NVIDIA GPU hardware) to implement these algorithms. For completeness, the empirical analysis contains a comparison to the unparallelized algorithms. From this data and the impression of the programming experience, strengths and weaknesses of usage of OpenMP for parallelizing and offloading sorting algorithms are derived. After discussing each benchmark in depth, as well as the data derived from the parallelized implementations of each, we found that

OpenMP's position as one of the forefront parallel programming standards is well-justified, with few, but notable, pitfalls for the average programmer. In terms of its performance in parallelizing common sorting algorithms with offloading to NVIDIA GPU hardware, it was found that OpenMP fails to deliver viable implementations of the algorithms that are advantageous over their single-threaded counterparts, though, this was found not to be the fault of OpenMP, but rather, of the inherent nature of offloading to NVIDIA GPU hardware.

DEDICATION

To my friends, family, and colleagues who supported me during this inquiry: Dr. Jeff Huang, Gang Zhao, Sadie Preece, Betty Carrión, Leylia Rico CVA, LCpl Elijah Rico USMC, Asani Holmes, Liliana Hernández, and, last but certainly not least, Sayani Shah.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Jeff Huang, my colleague, Gang Zhao, and the rest of the origin-oriented programming lab for their guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University an invaluable experience.

The data analyzed for On Modern Offloading Parallelization Methods: A Critical Analysis of OpenMP were produced Scott Carlos Carrión as original research. The analyses depicted in ON Modern Offloading Parallelization Methods: A Critical Analysis of OpenMP were conducted by Scott Carlos Carrión and are presented herein.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Undergraduate research was supported by Dr. Jeff Huang at Texas A&M University.

NOMENCLATURE

[GPU Graphical Processing Unit]

1. INTRODUCTION

In the modern age of high-performance computing, the race for optimization grows ever-more fervent. As the limits for hardware-driven optimizations are approached, we turn to other methods for optimizing computationally complex routines and algorithms. One such method is the use of graphical processing units (hereafter GPUs) as accelerators for general-purpose computing. Interfaces and implementations for accomplishing this can vary wildly across projects. With a number of these offloading solutions bolstered by research from industry affiliates and the academic community alike, contributors to these projects, who have the most knowledge surrounding their interface, are strongly incentivized to defend and improve their project to the very best of their ability. As such, in the ever-advancing field of high-performance computing, there exists little in the way of critical, impartial analysis of the contemporary interfaces for offloading to graphical processing units. This overall analysis seeks to evaluate one of, if not the most popular interfaces for GPU offloading of C/C++ routines: the OpenMP standard, with offloading to NVIDIA GPU hardware. Criteria for evaluation include: (1) discussion of the programming experience using OpenMP, (2) correctness of output, or conformity to the sequential (control) output, for each algorithmic benchmark, and (3) performance, as compared to the sequential output. The analysis for each interface concludes with an objective assertion of the strengths and weaknesses for the interface.

1.1 Benchmark Selection

The benchmark algorithms selected for these analyses are the classical insertion sort, the classical selection sort, and the comb sort. Unifying the benchmarks by selecting algorithms that all do the same thing (sort values in an array of basic integer primitives) is necessary for the

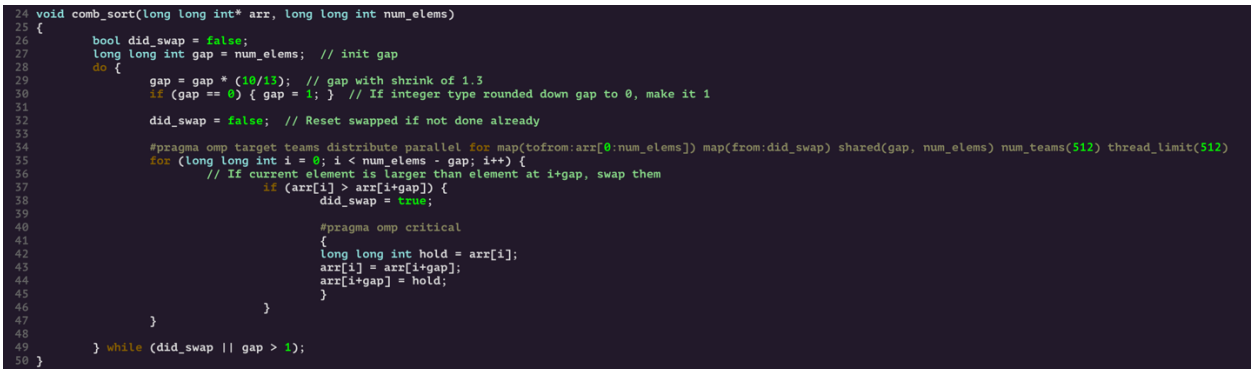
results of the analyses to be attributable only to the selection of parallelization method, rather than the algorithms types chosen. Further, the algorithms chosen as benchmarks for this study, like most sorting algorithms with time complexity of or similar to $O(n^2)$ tend to make a heavy amount of array accesses, even in the average case. This choice was deliberate, as we wish to study the performance of NVIDIA GPU offloading via OpenMP in solving problems which are memory access intensive.

2. METHODS

OpenMP, according to its own documentation, is “a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs.” [1] Practically, OpenMP provides its interface by the use of ‘pragmas’: preprocessor directives that specify to the compiler where and how to parallelize certain code, chiefly revolving around the identification of ‘parallel regions’, a set of instructions the API processes to turn into parallel code during compile-time. For this analysis, the LLVM implementation of the OpenMP 4.0 standard, with support for offloading to NVIDIA (internally referred to as “nvptx” devices) enabled, is configured to the Clang (version 10.0.1) compiler frontend (Figure 2.1, Figure 2.2).

```
23 void comb_sort(long long int* arr, long long int num_elems)
24 {
25         bool did_swap = false;
26         long long int gap = num_elems;  // init gap
27         do {
28                 gap = gap * (10/13);  // gap with shrink of 1.3
29                 if (gap == 0) { gap = 1; }  // If integer type rounded down gap to 0, make it 1
30
31                 did_swap = false;  // Reset swapped if not done already
32
33                 for (long long int i = 0; i < num_elems - gap; i++) {
34                         // If current element is larger than element at i+gap, swap them
35                         if (arr[i] > arr[i+gap]) {
36                                 did_swap = true;
37                                 long long int hold = arr[i];
38                                 arr[i] = arr[i+gap];
39                                 arr[i+gap] = hold;
40                         }
41                 }
42         } while (did_swap || gap > 1);
43 }
44
45
```

Fig 2.1: Excerpt of unparallelized code (from Comb Sort)



```
void comb_sort(long long int* arr, long long int num_elems)
{
        bool did_swap = false;
        long long int gap = num_elems;  // init gap
        do {
                gap = gap * (10/13);  // gap with shrink of 1.3
                if (gap == 0) { gap = 1; }  // If integer type rounded down gap to 0, make it 1

                did_swap = false;  // Reset swapped if not done already

                #pragma omp target teams distribute parallel for map(tofrom:arr[0:num_elems]) map(from:did_swap) shared(gap, num_elems) num_teams(512) thread_limit(512)
                for (long long int i = 0; i < num_elems - gap; i++) {
                        // If current element is larger than element at i+gap, swap them
                                if (arr[i] > arr[i+gap]) {
                                        did_swap = true;

                                        #pragma omp critical
                                        {
                                        long long int hold = arr[i];
                                        arr[i] = arr[i+gap];
                                        arr[i+gap] = hold;
                                        }
                                }
                }

        } while (did_swap || gap > 1);
}
```

Fig 2.2: Excerpt of parallelized code (from Comb Sort)

Parallelization of all algorithms proceeded in a very similar fashion for all benchmarks: A parallelizable region, typically the principal loop of the algorithm, was identified, necessary clauses for the directive itself were chosen, and the procedure within the loop was modified to adhere to constraints introduced by making use of OpenMP to offload to NVIDIA GPU hardware (Figure 2.2). These limitations are discussed more in depth as results, rather than as methods, further in this paper.

3. RESULTS

3.1 Performance and Correctness Assessment

We present a summary of the runtimes and a subsequent assessment of the performance of the OpenMP with GPU offloading implementation of each sorting algorithm below. Timing was performed using the C++ Standard Template Library's "chrono" header; specifically, the high-resolution clock function. [2]

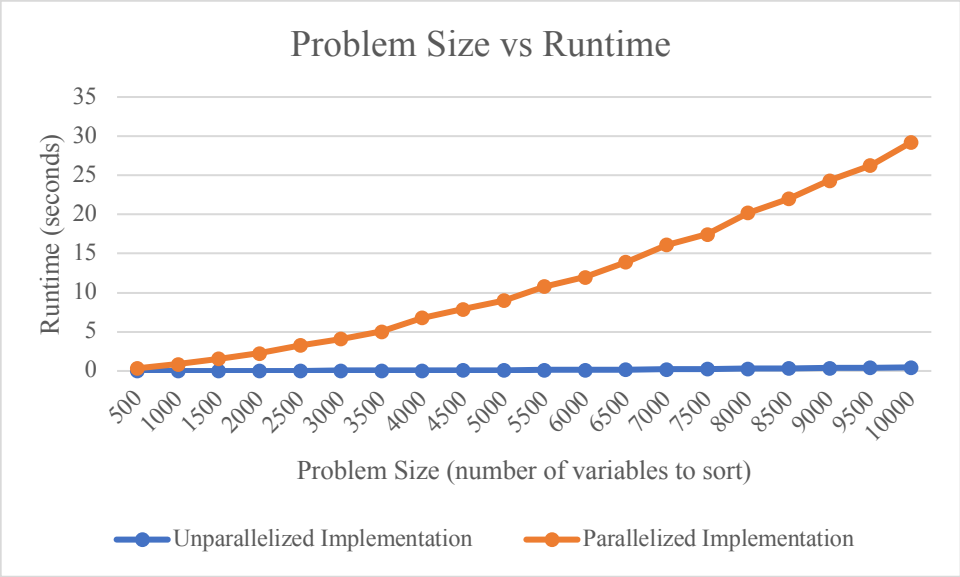


Figure 3.1.1: Visualization of Runtime Data, Comb Sort Benchmark

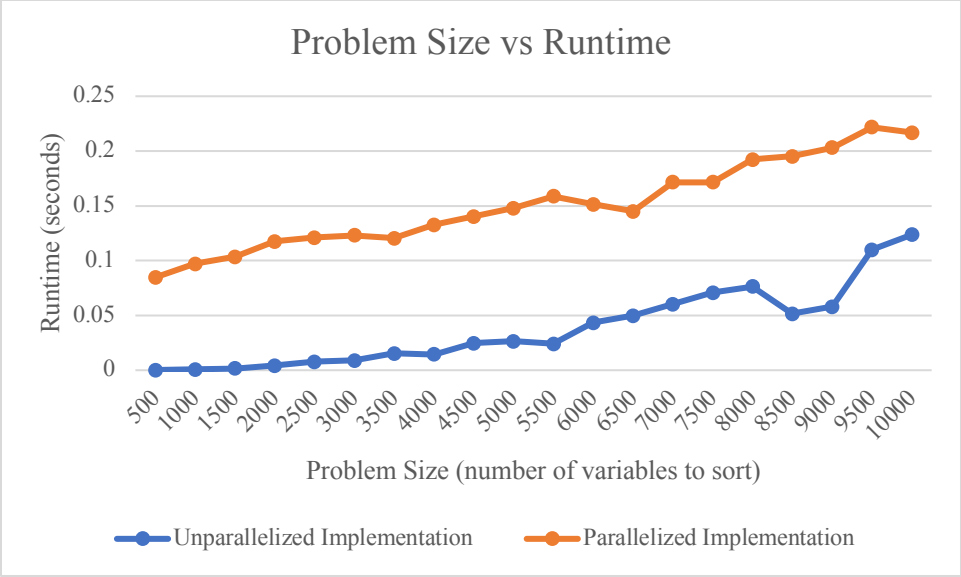


Figure 3.1.2: Visualization of Runtime Data, Insertion Sort Benchmark

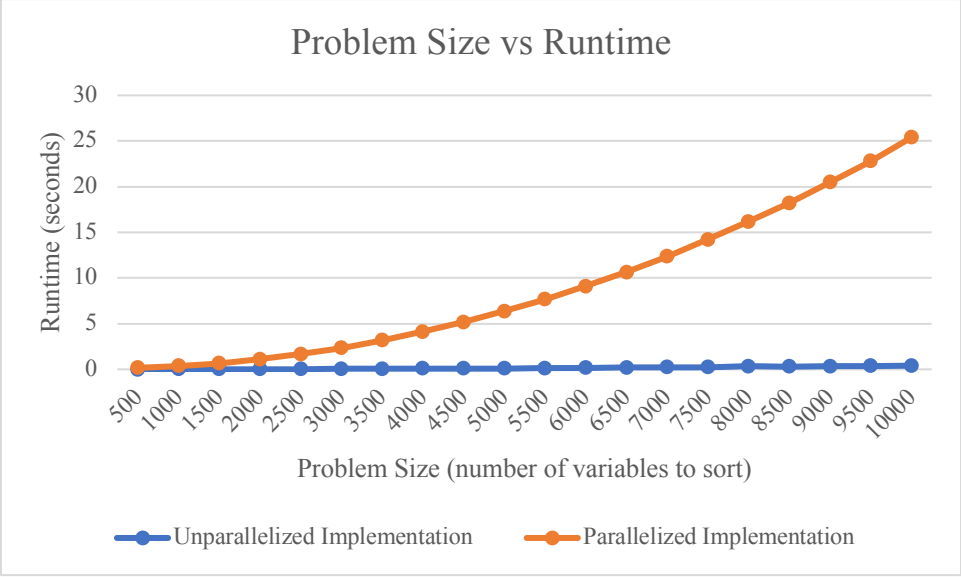


Figure 3.1.3: Visualization of Runtime Data, Selection Sort Benchmark

Table 1: Tabulation of Runtime Data, Comb Sort Benchmark

Problem Size (number of variables to sort)	**Unparallelized Runtime (seconds)**	**Parallelized Runtime (seconds)**
500	0.00087095	0.338751
1000	0.00425457	0.859901
1500	0.00798661	1.56521
2000	0.0145872	2.26183
2500	0.0228375	3.28348
3000	0.0340949	4.12054
3500	0.0485203	5.03266
4000	0.0640665	6.80713
4500	0.0785111	7.89892
5000	0.101147	9.00309
5500	0.122659	10.8212
6000	0.147471	12.0028
6500	0.175818	13.9048
7000	0.203775	16.1392
7500	0.2392	17.4568
8000	0.306832	20.203
8500	0.33647	22.0007
9000	0.394711	24.3303
9500	0.42936	26.2551
10000	0.459501	29.1987

Table 2: Tabulation of Runtime Data, Insertion Sort Benchmark

Problem Size (number of variables to sort)	**Unparallelized Runtime (seconds)**	**Parallelized Runtime (seconds)**
500	0.00017906	0.0847266
1000	0.00069509	0.0970685
1500	0.00154835	0.103393
2000	0.0041255	0.117554
2500	0.0076305	0.121126
3000	0.00875286	0.123181
3500	0.0154178	0.120294
4000	0.0144519	0.132609
4500	0.0245274	0.140412
5000	0.0263155	0.147926
5500	0.0241156	0.158625
6000	0.0434787	0.151437
6500	0.0499099	0.145052
7000	0.0602627	0.171638
7500	0.0708219	0.171523
8000	0.0764472	0.192152
8500	0.0514324	0.195133
9000	0.0580327	0.202965
9500	0.109895	0.221672
10000	0.123627	0.216779

Table 3: Tabulation of Runtime Data, Selection Sort Benchmark

Problem Size (number of variables to sort)	**Unparallelized Runtime (seconds)**	**Parallelized Runtime (seconds)**
500	0.00141071	0.162085
1000	0.00623467	0.363855
1500	0.0144788	0.667205
2000	0.0221421	1.11537
2500	0.0215338	1.67583
3000	0.049337	2.35043
3500	0.0511447	3.17589
4000	0.0745646	4.11176
4500	0.0984962	5.18284
5000	0.0890428	6.35998
5500	0.133128	7.66194
6000	0.159177	9.08676
6500	0.192901	10.6228
7000	0.216681	12.3384
7500	0.246795	14.2102
8000	0.337477	16.1617
8500	0.309093	18.2221
9000	0.352515	20.4817
9500	0.384044	22.8153
10000	0.400469	25.3819

It can be clearly seen in the tables and derived visualization figures that in no instance does an offloaded implementation of any of the benchmark algorithms does any marked improvement occur, though all results were correct. For the comb sort benchmark, the runtime for the parallelized version appears to increase with size almost linearly, and was eminently outperformed by the unparallelized control implementation (Table 1, Figure 3.1.1). The parallelized implementation of the insertion sort benchmark saw the greatest degree of comparability to its unparallelized counterpart, but was still outperformed to a considerable degree for all problem sizes (Table 2, Figure 3.1.2). Finally, much like the comb sort benchmark, the runtime of the parallelized implementation of the selection sort benchmark increased in a nearly linear fashion, and, again, it is clear that the parallelized implementation was outperformed by the unparallelized implementation (Table 3, Figure 3.1.3). These results can be attributed to the overhead related to two circumstances that arise in offloading that do not arise in the unoptimized, single-threaded implementation of the benchmark algorithms. Firstly, swapping the positions of values in the constituent array must be an atomic operation; that is, a directive must be made to OpenMP to treat the swap as a critical section, thus dramatically reducing the performance of the parallelized algorithm. Secondly, in order for the device (GPU) to be able to carry out the procedure of each algorithm, after dividing up the work among its many teams of threads, frequent, synchronized, and expensive two-way (or 'to-from') mapping is necessary to produce correct output. The latency associated with two-way mapping in a procedure where heavy memory access is a requisite necessarily diminishes the optimization potential for offloading of sorting algorithms in general [3].

3.2 Critical Analysis of Programming Experience

For all algorithms tested, this method of abstraction proved relatively simple, although in some cases, refactoring the code to some degree such that an easily identifiable parallel region exists can be beneficial. Use of the interface maintains a much higher-level management of threads, teams of threads, and mapping of memory to the device. In terms of user-end informativeness, however, OpenMP leaves much to be desired. If some runtime error occurs, whether it be a signal being raised by the device, or by the host, there is very little information presented to the programmer. Not only is configuration of the LLVM/Clang 10.0.1 compiler and infrastructure a tedious manual process of hardware specification that leaves much room for error, but, additionally, for the machine tested, attempting to create a debug build of the compiler does not emit error details in the vast majority of cases when such a runtime error occurs. Most of the time, it was found, the dreaded "Libomptarget fatal error 1: failure of target construct while offloading is mandatory" error, which is the only information output upon one of these aforementioned failures, is not due to a shortcoming of the implementation of the interface. Rather, it is a shortcoming of the CUDA language itself. OpenMP, as implemented in the LLVM project, successfully and correctly generates CUDA code for the device (GPU) to execute in accordance with the standards of the C/C++ language as well as the OpenMP 4.0 standard. However, it was found experimentally that many known libraries, including the C++ standard library (as implemented for the host compiler, Clang, that is, not libcu++ [4]), are not supported by CUDA. Both CUDA and OpenMP fail to communicate this information to the programmer. Another instance of this same error occurs when attempting to use a variable that is not mapped on the device within a parallel region. Like before, no documentation surrounding this cryptic

behavior, nor its potential causes, could be located. These lessons were learned by low-level investigation, and vast amounts of time-consuming trial and error.

While it is surprisingly difficult to debug OpenMP as an API, the implementation and parallelization of the routines proved remarkably simple to develop, with the API's clauses providing similarly straightforward logical extensions where necessary. OpenMP's ability to work with the compiler to interpret how variables should be mapped (i.e shared or private), how iterations within the parallel region should be distributed, and the seamless integration with the system's native threading interface (in the case of the Texas A&M High Performance Research Supercomputer, POSIX threads) ought to be duly recognized as well.

4. CONCLUSION AND FUTURE WORK

Among the elite of parallelization offloading methods, the OpenMP standard holds firm as one of the best in terms of its interface, with some debugging difficulties which are rendered inconsequential in the face of the sheer advantages that it provides the programmer over traditional, 'manual' parallelization. After analyzing the performance of algorithms offloaded to NVIDIA GPU hardware via the LLVM Clang 10.0.1 implementation of the OpenMP standard, it was found that using OpenMP to offload memory access intensive, computationally complex sorting algorithms is generally not viable, due almost exclusively to the inherent overhead associated with offloading and frequent two-way mapping. With these findings in mind, we hold that as an interface, OpenMP excels on the whole, and sorting algorithms are poorly parallelized via offloading due to their inherent intensive memory access.

This inquiry proved to be quite informative, shoring up the potential for future work surrounding the limitations of effective parallelization using offloading to NVIDIA hardware in general. Further research into this is a worthy topic of follow-up papers, as it is evidently clear that parallelization via offloading, regardless of means or interface, does not necessarily yield an optimized implementation in general as compared to a host-parallelized, or, as seen in the results presented in this inquiry, even a single-threaded implementation. As explored in this paper, these memory access intensive algorithms proved to not be viable for parallelization via offloading, but, there are still an extremely diverse range of algorithms and other computationally complex, programmatic solutions to vexingly difficult problems for which optimization by means of parallelization via offloading has not yet been explored in-depth as it has in this research.

REFERENCES

[1] OpenMP Architecture Review Board, "OpenMP FAQ," *OpenMP*, 01-Jul-2018. [Online]. Available: https://www.openmp.org/about/openmp-faq/#WhatIs. [Accessed: 09-Feb-2021].

[2] CPPReference Foundation, "std::chrono::high_resolution_clock," *cppreference.com*. [Online]. Available: https://en.cppreference.com/w/cpp/chrono/high_resolution_clock. [Accessed: 09-Feb-2021].

[3]. Free Software Foundation, "Offloading Support in GCC," *Offloading - GCC Wiki*. [Online]. Available: https://gcc.gnu.org/wiki/Offloading. [Accessed: 26-Feb-2021].

[4] NVIDIA Corporation, "CUDA C++ Standard," *CUDA Toolkit Documentation*, 01-Dec-2020. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-std/index.html. [Accessed: 09-Feb-2021].

APPENDIX A: TESTING HARDWARE SPECIFICATIONS

All testing for the results presented of the inquiry herein were performed on the Terra cluster of the Texas A&M University High-Performance Supercomputer (HPRC), with permission. A summary of the specifications follows (see corresponding footnote for cited source)[1].

[1] TAMU HPRC, "Terra: A Lenovo x86 HPC Cluster," *TAMU HPRC Wiki*, 05-Apr-2021. [Online]. Available: https://hprc.tamu.edu/wiki/Terra:Intro. [Accessed: 12-Mar-2021].

Table A: Hardware Specifications of Compute Nodes in the Terra Cluster

Table 1 Details of Compute Nodes

	General 64GB Compute	GPU 128 GB Compute	KNL 96 GB (68 core) Compute	KNL 96 GB (72 core) Compute	V100 GPU 192 GB Compute
Total Nodes	256	48	8	8	4
Processor Type	Intel Xeon E5-2680 v4 (Broadwell), 2.40GHz, 14-core		Intel Xeon Phi CPU 7250 (Knight's Landing), 1.40GHz, 68-core	Intel Xeon Phi CPU 7290 (Knight's Landing), 1.50GHz, 72-core	Intel Xeon Gold 5118 (Skylake), 12-core, 2.30 GHz
Sockets/Node	2		2	2	2
Cores/Node	28		68	72	24
Memory/Node	64 GB DDR4, 2400 MHz	128 GB DDR4, 2400 MHz	96 GB DDR4, 2400 MHz		192 GB, 2400 MHz
Accelerator(s)	N/A	1 NVIDIA K80 Accelerator	N/A		2 NVIDIA 32GB V100 GPUs
Interconnect	Intel Omni-Path Architecture (OPA)		Intel Omni-Path Architecture (OPA)		
Local Disk Space	1TB 7.2K RPM SATA disk		220GB		300GB

Table B: Memory Limits of Nodes in the Terra Cluster

Memory Limits of Nodes				
	64GB Nodes	**128GB Nodes**	**96GB KNL Nodes (68 core)**	**96GB KNL Nodes (72 core)**
Node Count	256	48	8	8
Number of Cores	28 Cores (2 sockets x 14 core)		68 Cores	72 Cores
Memory Limit Per Core	2048 MB 2 GB	4096 MB 4 GB	1300 MB 1.25 GB	1236 MB 1.20 GB
Memory Limit Per Node	57344 MB 56 GB	114688 MB 112 GB	89000 MB 84 GB	89000 MB 84 GB

Table C: Details of Login Nodes in the Terra Cluster

Table 2: Details of Login Nodes		
	No Accelerator	**Two NVIDIA K80 Accelerator**
Host Names	terra1.tamu.edu terra2.tamu.edu	terra3.tamu.edu
Processor Type	Intel Xeon E5-2680 v4 2.40GHz 14-core	
Memory	128 GB DDR4 2400 MHz	
Total Nodes	2	1
Cores/Node	28	
Interconnect	Intel Omni-Path Architecture (OPA)	
Local Disk Space	per node: two 900GB 10K RPM SAS drives	
Notes	Each K80 Accelerator has two GPUs	