RaiderSTREAM: Adapting the STREAM Benchmark to Modern HPC Systems

Michael Beebe*, Brody Williams*†, Stephen Devaney*, John Leidel*‡, Yong Chen*, Steve Poole§

*Texas Tech University, Lubbock, Texas
†NVIDIA Corporation, Santa Clara, California
‡Tactical Computing Laboratories, Muenster, Texas
§Los Alamos National Laboratory, Los Alamos, New Mexico

2022 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-9786-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPEC55821.2022.9926292

***Abstract*—Sustaining high memory bandwidth utilization is a common bottleneck to maximizing the performance of scientific applications, with the dominating factor of the runtime being the speed at which data can be loaded from memory into the CPU and results can be written back to memory, particularly for increasingly critical data-intensive workloads. The prevalence of irregular memory access patterns within these applications, exemplified by kernels such as those found in sparse matrix and graph applications, significantly degrade the achievable performance of a system's memory hierarchy. As such, it is highly desirable to be able to accurately measure a given memory hierarchy's sustainable memory bandwidth when designing applications as well as future high-performance computing (HPC) systems. STREAM is a de facto standard benchmark for measuring sustained memory bandwidth and has garnered widespread adoption. In this work, we discuss current limitations of the STREAM benchmark in the context of high-performance and scientific computing. We then introduce a new version of STREAM, called RaiderSTREAM, built on the OpenSHMEM and MPI programming models in tandem with OpenMP, that include additional kernels which better model irregular memory access patterns in order to address these shortcomings.**

***Index Terms*—Benchmarking, Memory Bandwidth, Irregular Memory Access Patterns, High-Performance Computing**

I. Introduction

In recent decades, the STREAM benchmark [16], [17] has become the industry standard for measuring sustainable memory bandwidth. The STREAM benchmark is a simple, yet powerful synthetic benchmark program that measures memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels [16]. However, there are two primary limitations to the current STREAM benchmark that make it less applicable to modern high-performance computing. First, the STREAM vector kernels only demonstrate sequential memory access patterns and therefore provide an estimation of theoretical peak performance only in "ideal conditions", where memory accesses are sequential and consecutive. Such conditions often do not hold nowadays on HPC systems because a wide spectrum of workloads, such as sparse matrix, graph computing, data mining, data analysis, etc., make irregular memory accesses a new norm. Furthermore, STREAM was originally designed to measure the achievable memory bandwidth of a single-node system and is thus parallelized using only OpenMP [19]. Arguably, given the increasing scale and complexity of HPC systems, we are often much more interested in understanding the memory subsystem performance as a coherent whole infrastructure, instead of disjoint, individual nodes.

In this work, we introduce a new design and associated implementation of the STREAM memory bandwidth benchmark [7] that better models real-world HPC scenarios (called RaiderSTREAM). The fundamental idea is two-fold. First, the new benchmark considers irregular memory accesses as a norm, characterizes such behaviors as part of its DNA, and measures the memory subsystem performance given such behaviors. Second, the new RaiderSTREAM benchmark focuses on measuring the collective memory bandwidth across many nodes to mimic real-world HPC workloads. The end result could be a much more realistic and useful benchmark tool for HPC systems. Given this new design philosophy, we provide a new implementation that specifically focuses on two representative memory access patterns, *gather* and *scatter*. In other words, our newly introduced benchmark contains *gather* and *scatter* kernels that are constructed to mimic the irregular memory accesses patterns characteristic of scientific computing. Moreover, we have added benchmark variants built upon the OpenSHMEM [21] and MPI [11] programming models, together with OpenMP [19], to gauge aggregate memory bandwidth performance as a collective, coherent memory subsystem instead of separate and individual nodes.

The contribution of this research and development is summarized as follows. First, we review and identify limitations of the existing, widely-used STREAM benchmark in the context of modern HPC workloads. Second, we provide a new benchmark design with considering both irregular and regular memory accesses, as well as considering memory subsystem collectively. Third, we provide an implementation for gather and scatter patterns and their variants. The new implementation is publicly available from [7].

II. Background and Motivation

For decades, performance improvements in CPUs have been realized at a much higher rate than those to memory subsystems. This phenomenon, wherein application performance is limited by a given system's sustainable memory bandwidth capabilities, is commonly referred to as the *memory wall* problem [25]. When the so-called *memory wall* is reached,

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

application execution time becomes almost entirely dependent on the speed at which the memory system can send data to the CPU, as opposed to the computational capabilities of the CPU itself. Many applications today demonstrate this effect. As a result, continued CPU performance improvements alone do not necessarily yield system-wide performance boosts. This is also known as the Von Neumann bottleneck effect [12]. Due to the memory wall problem and its ramifications on system performance, continued improvement of sustainable memory bandwidth is critical for continued system performance increases. As a consequence of this need, we have seen a rise in different memory technologies that seek to improve memory bandwidth, particularly in high-performance computing environments. Some examples of these technologies include 3D stacked memory devices such as high bandwidth memory (HBM) [14] and hybrid memory cubes (HMC) [20].

The desire to design improved memory technologies implies the need for benchmarks that are capable of accurately measuring "real-world" sustainable memory bandwidth when evaluating a system's performance capabilities and identifying bottlenecks, particularly for memory-intensive applications. Benchmarks such as High-Performance LINPACK (HPL) [10] and its variants can be considered misleading with respect to memory behaviors, and thus, a system's overall performance because they apply sequential kernels that result in few cache misses and fail to stress the system's memory hierarchy or indicate DRAM performance. HPCG (High Performance Conjugate Gradients) benchmark has been introduced to overcome the limitations of HPL and has attracted increasing attention since its debut. However, the HPC community still lacks a powerful benchmark for memory subsystems that can characterize a variety of real-world HPC workloads' access patterns.

The STREAM benchmark has become the de facto memory bandwidth benchmark and garnered widespread adoption. However, the original STREAM benchmark has two primary limitations that make it unsuitable for modern high-performance computing systems and scientific applications. First, the original STREAM kernels demonstrate only sequential memory access patterns and therefore provide an estimation of theoretical peak performance only in "ideal conditions". However, modern scientific codes most often exhibit real-world behavior that is characterized by irregular memory access patterns, which are not well modeled by the current benchmark implementation. As a memory subsystem's bandwidth performance can be expected to vary significantly different across memory access patterns, benchmark kernels that replicate these different behaviors would provide a more realistic understanding of a given system's performance capabilities. Second, STREAM was originally designed to measure the achievable memory bandwidth of a single-node system and is thus parallelized using OpenMP. This is a limitation with respect to modern HPC as most large-scale HPC applications are run across multiple nodes in tandem. It is therefore desirable to have to have a variation of STREAM that can measure aggregate memory bandwidth performance across nodes in an HPC cluster. In section III, we detail the contributions we have made to the STREAM benchmark in order to rectify these limitations.

III. Design and Implementation

In this section, we introduce the design and implementation of RaiderSTREAM in order to make a memory subsystem benchmark more applicable to modern HPC systems.

A. Irregular Memory Access Patterns

The first contribution of RaiderSTREAM is the introduction of new benchmark kernels that mimic irregular memory access patterns. One of the limitations of the STREAM benchmark is that the kernels are sequential, giving users of the benchmark a picture of their system's "best case scenario" memory bandwidth. In our new design, we introduce irregularity by way of *gather* and *scatter* [8] variations of the original STREAM kernels. Our *gather* and *scatter* kernels are implemented in a nondeterministic fashion through utilization of randomized index arrays. The index arrays are populated with non-repeating values in order to avert race conditions. This nondeterminism, in turn, is used to simulate sparsity, which is a common characteristic of data-intensive HPC workloads. Herein, we utilize these randomly populated *IDX arrays*, passed as array subscripts, to read from and write randomly to otherwise contiguous memory spaces.

The *gather* and *scatter* benchmark kernels are similar in that they both provide insight into the real-world performance one can expect from a given system in a scientific computing setting. However, there are differences between these two memory access patterns that should be understood. The *gather* memory access pattern is characterized by randomly indexed loads coupled with sequential stores. This can help give us an understanding of *read* performance from sparse datasets such as arrays or matrices. The *scatter* memory access pattern can be considered the inverse of its *gather* counterpart, and is characterized by the combination of sequential loads coupled together with randomly indexed stores. This pattern can give us an understanding of *write* performance to sparse datasets. Fig. 1 illustrates the gather and scatter irregular memory access patterns through an example, where the index arrays have values of 3, 7, 2, and 0.

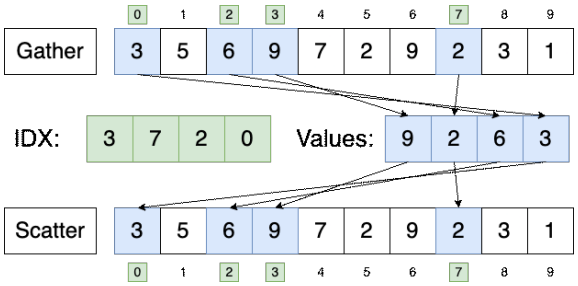

Fig. 1: Gather and Scatter Irregular Memory Access Patterns

B. Multi-Node Support

The second contribution of RaiderSTREAM is the introduction of multi-node support for the STREAM benchmark. One

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

of the limitations of the original STREAM implementation is that it was parallelized only using OpenMP and is thus limited to running on a single node. However, HPC systems are becoming increasingly heterogeneous and consist of many CPU sockets across many nodes. As such, when evaluating an HPC system's performance capabilities, it is highly desirable for system administrators and architects to be able to measure the scalability of memory bandwidth utilization for a given system as the number of nodes being used for a particular problem is increased.In order to address this need, we introduce two additional kernels that incorporate the MPI and OpenSHMEM parallel programming paradigms, allowing RaiderSTREAM to run across multiple nodes using both conventional distributed memory and partitioned global address space (PGAS) programming models, respectively. Notably, our modified kernels do not utilize any inter-process communication (IPC) routines such as `MPI_SEND` or `SHMEM_PUT` during execution because they have a much higher and disproportionate latency. Doing so results in the benchmark execution time being dominated by inter-process communication, dwarfing the significance of the system's memory bandwidth capabilities in the benchmark's reported output. Instead, as shown in Fig. 2, we use

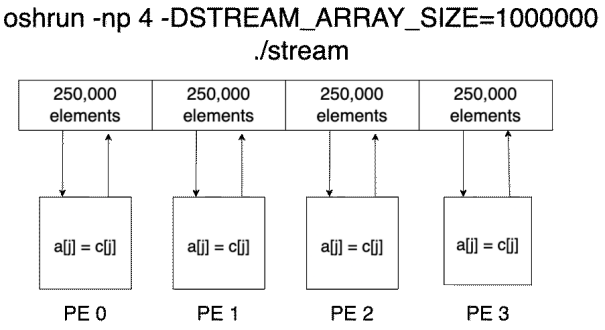

Fig. 2: Illustration of Multi-Node Support

MPI and OpenSHMEM to distribute segments of the arrays across a user-specified number of processing elements (PEs), effectively leveraging these programming models as a resource allocator. During benchmark execution, each PE utilizes its own array segment for kernel computation and writes its result back to the same array segments. Users can utilize a job scheduler such as Slurm [13] or compiler flags to customize with a fine degree of granularity how the benchmark is run. For example, PEs can be assigned as individual nodes, CPU sockets within a node, cores within a socket, etc.

C. Benchmark Kernels

Table I shows all twelve kernels included in our new benchmark. The first four entries represent the original STREAM kernels. In these benchmark kernels, consecutive memory blocks are loaded in a sequential manner. This behavior results in frequent cache hits with the only cache misses being compulsory misses when a new memory block is referenced and capacity misses when the problem size exceeds cache capacity. The following eight table entries represent our newly introduced *gather* and *scatter* kernels that exhibit irregular memory access patterns. These new kernels are variations of the original STREAM kernels wherein the only significant differences are the order in which elements of the arrays are accessed with respect to the memory load and store instructions. As shown in Fig. 1, the irregular kernels access memory locations in a random, or non-sequential manner, causing the CPU to reference memory addresses that are far apart from one another, often outside of cache. This characteristic results in much more frequent cache misses and lower memory bandwidth, but provides a more accurate portrayal of real-world performance.

TABLE I: Modified STREAM Benchmark Kernels

Name	Kernel	Bytes/Iter	FLOPS/Iter
COPY:	a[i] = b[i]	16	0
SCALE:	a[i] = q*b[i]	16	1
SUM:	a[i] = b[i] + c[i]	24	1
TRIAD:	a[i] = b[i] + q*c[i]	24	2
GATHER COPY:	a[i] = b[IDX[i]]	16	0
GATHER SCALE:	a[i] = q*b[IDX[i]]	16	1
GATHER SUM:	a[i] = b[i] + c[IDX[i]]	24	1
GATHER TRIAD:	a[i] = b[i] + q*c[IDX[i]]	24	2
SCATTER COPY:	a[IDX[i]] = b[i]	16	0
SCATTER SCALE:	a[IDX[i]] = q*b[i]	16	1
SCATTER SUM:	a[IDX[i]] = b[i] + c[i]	24	1
SCATTER TRIAD:	a[IDX[i]] = b[i] + q*c[i]	24	2

D. Counting Bytes

STREAM records the number of bytes read from, and written to, memory during kernel execution in order to calculate effective memory bandwidth. However, the number of memory accesses necessary across kernels varies depending on the kernel in question. The effect of ALU operations, which occur three of the four kernels, are not represented in this metric but instead are characterized the by different behavior of each kernel operation As an example, while the *copy* operation does no arithmetic, its bytes are counted those of other operations. Table I shows how many bytes and FLOPS are counted for each iteration of the STREAM loops by kernel, assuming that the STREAM_TYPE environment variable is set to the default data type of *double*. Equation 1 shows the formula used for calculating bandwidth (MB/s), where α is the number of memory accesses per iteration of the main STREAM loops, γ is the size in bytes of the `STREAM_TYPE`, and λ is the problem size, or `STREAM_ARRAY_SIZE`.

$$Bandwidth(MB/s) = \frac{(1.0E6 \times \alpha \times \gamma \times \lambda)/1e^9}{mintime} \quad (1)$$

As shown in Table I, there is some variation in the number of bytes counted for each iteration of the main loops within the benchmark. This is because STREAM take into account the number of memory accesses that occur each time the kernels are executed to ensure that the results are normalized across all kernels. For example, each variation of the *sum* kernel counts twenty-four bytes per iteration because the kernel accesses three eight byte memory locations.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

E. Validation

The purpose of validation within the benchmark is to reproduce the kernel operations outside of the timer, and then compare the reproduced (expected) results with the actual values of the arrays. Doing so ensures that results from the parallelized kernels produce the same array values as when the kernels are executed sequentially. Since our additional *gather* and *scatter* kernels use the same arrays and array values as the original/sequential kernels, the validation process in our new benchmark remains, for the most part, unchanged from the original implementation. The only difference is that the additional eight kernel operations are included when calculating the expected values of the elements within the arrays. If any of the expected values do not match the observed values of the arrays, the benchmark run will be considered "invalid".

F. Run Rules

STREAM is intended to measure the bandwidth from main memory. However, it can be used to measure cache bandwidth as well by the adjusting the environment variable `STREAM_ARRAY_SIZE` such that the memory needed to allocate the arrays can fit in the cache level of interest. The general rule for `STREAM_ARRAY_SIZE` is that each array must be at least *4x* the size of the sum of all the last-level caches, or 1 million elements – whichever is larger [16]. The `NTIMES` environment variable can be used to specify the number of times the benchmark is run, and the "best" bandwidth in the unit of MB/s is recorded across all `NTIMES` runs. The `STREAM_TYPE` environment variable can be used to change the data type of the elements within the STREAM arrays used throughout the benchmark kernels. Different data types will have different data sizes, but the change in data size should not affect the benchmark results as this is accounted for in the bandwidth formula, as shown in Equation 1.

IV. Evaluation

In order to assess the efficacy of our RaiderSTREAM, we ran extensive tests across two different HPC clusters: Texas Tech University's Nocona cluster and a cluster hosted by the HPC Advisory Council called Thor. A detailed description of nodes within these two systems is shown in Table II. It is worth noting the significant differences of the CPU and cache configurations between these two systems. The Nocona system utilizes dual AMD Epyc 7702 processors with 64 cores per socket, 1 thread per core, and a base clock rate of 2.0 GHz while the the Thor system employs dual Intel Xeon E5-2697A v4 processors with only 16 cores per socket, 1 thread per core, and a base clock rate of 2.6 GHz. Furthermore, the Nocona system has a 32 KiB L1 d/i cache, a 512 KiB L2 cache, and a 16 MiB L3 cache, giving the system a total cache size of approximately 17,367 KiB, or 17,367,040 bytes.

Following the run rules discussed in Section III-F, we can find the proper problem size, or `STREAM_ARRAY_SIZE` by simply multiplying the total cache size by 4 and dividing the quotient by the size in bytes of `STREAM_TYPE`. Let `sizeof(STREAM_TYPE)` $= 8$ bytes, that gives us an approximate `STREAM_ARRAY_SIZE` of $4 \times 17,367,040/8 = 8,683,520$, or 8.7 million for the sake of simplicity. The Thor system has a 32 KiB L1 d/i cache, a 256 KiB L2 cache, and a 40 MiB L3 cache giving it a much larger total cache size of approximately 42,271 KiB, or 42,270,720 bytes. The `STREAM_ARRAY_SIZE` for an official STREAM run on Thor can be calculated as $4 \times 42,270,720/8 = 21,135,360$, or 22 million for the sake of simplicity.

TABLE II: System Information

	Nocona	Thor
ISA	x86_64	x86_64
CPU	2x AMD Epyc 7702 64 cores/socket 1 thread/core	2x Intel Xeon E5-2697A v4 16 cores/socket 1 thread/core
Main Memory	512 GiB DDR4, 3200 MHz	256 GiB DDR4, 2400 MHz
Cache Configuration	L1 d/i: 32 KiB, L2: 512 KiB, L3: 16 MiB	L1 d/i: 32 KiB, L2: 256 KiB, L3: 40 MiB
Interconnect	Infiniband HDR 200G	Infiniband HDR 200G
Operating System	CentOS 8.1	Rocky Linux 8.6
Compiler	GCC 10.1.0	GCC 8.5.0
MPI	OpenMPI 4.0.4	OpenMPI 4.1.0
OpenSHMEM	OpenSHMEM 4.0.4	OpenSHMEM 4.1.0

Next, we will discuss our performance evaluations on a single-core basis to demonstrate continuity and correctness as well as multi-node performance to demonstrate scalability. The *double* datatype, which has a size of 8 bytes, is used throughout all of the tests done in this section. Each of the results shown in our evaluation figures represent the average results over five runs.

A. Single Core Performance

In order to demonstrate the performance divergence between sequential and irregular memory access patterns, we used our OpenSHMEM implementation of STREAM and all twelve of its benchmark kernels to produce Fig. 3 and Fig. 4. As expected, the sequential kernels outperformed the *gather* and *scatter* kernels by a great deal due to the *gather* and *scatter* kernels' frequent cache misses and references to DRAM. Markedly, on both systems, *copy* and *scale* marginally outperform *add* and *triad* in the *gather* variation of the kernels while the inverse is true for the *scatter* variation. This observation tells us that both of these systems have a slightly higher capacity to complete memory accesses *to* sparse arrays when doing arithmetic than *from*. In the context of high-performance computing, this could be useful information when writing scientific applications that make optimal use of the memory system. Although this performance difference in Fig. 3 and Fig. 4 is relatively small, with much larger real-world problem sizes and a higher number of memory accesses, the performance gains from writing applications with this characteristic in mind can be consequential.

In order to demonstrate correctness and continuity across each benchmark implementation within RaiderSTREAM, we

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

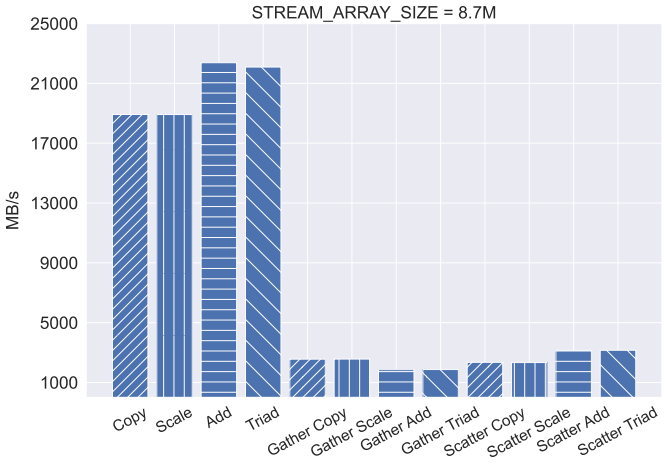

Fig. 3: Single Core Performance (All Kernels) - Nocona

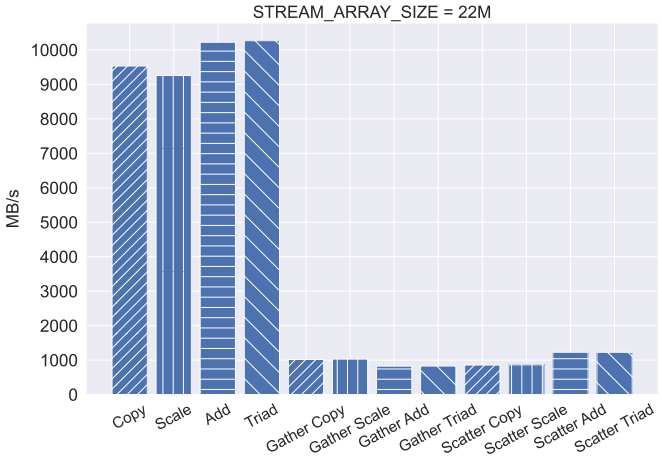

Fig. 4: Single Core Performance (All Kernels) - Thor

have done a series of performance tests with different problem sizes using a single CPU core on each of the systems outlined in Table II. In this series of tests, as shown in Fig. 5 and Fig. 6, we started with a `STREAM_ARRAY_SIZE` of 2 million elements, and the `STREAM_ARRAY_SIZE` was halved during each subsequent trial.

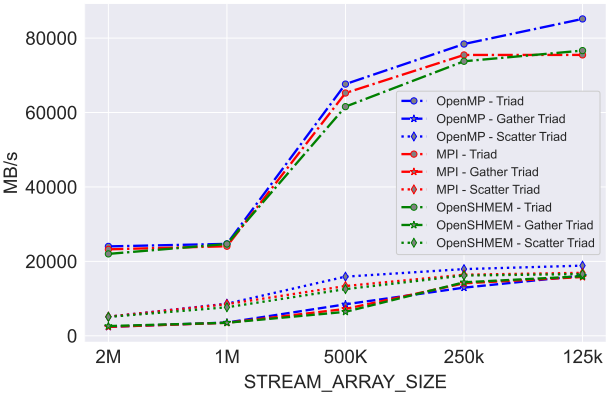

Fig. 5: Single Core Performance by Problem Size - Nocona

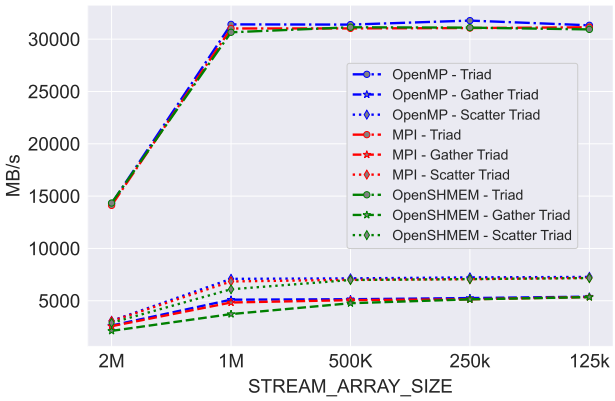

Fig. 6: Single Core Performance by Problem Size - Thor

The single-core cache analysis reflected in Fig. 5 and Fig. 6 includes our OpenMP, MPI, and OpenSHMEM implementations of each variation of the *triad* kernel. On both systems, these tests show a significant performance increase as we decrease the problem size. This is because as the problem size is reduced, more of it fits in the CPU cache, which has much higher memory bandwidth than DRAM. On the Nocona system, we observe performance increases at each increment on the x-axis with a significant jump when decreasing the problem size from one million to five-hundred thousand array elements. In contrast, Thor hits its performance peak for most kernels when the problem size is reduced to 1 million array elements. This is because the Thor system has a much larger cache that can fit larger problem sizes than the cache of Nocona. As expected, particularly on a single-core basis due to the fact that there is no inter-process communication in the MPI and OpenSHMEM benchmark implementations, the performance of each variation of the *triad* kernels is consistent across all three programming models. Notably, the sequential kernels receive a far more substantial performance increase than their irregular counterparts as the problem size is reduced. This can be explained by the behavior of the memory access patterns exhibited by the kernels. Since the *gather* and *scatter* kernels are referencing sparse memory locations resulting in a greater number of cache misses, the memory accesses still take a considerable amount longer than sequential memory access patterns, giving them less of a performance benefit from problem size reduction.

B. Multi-Node Performance

Since one of the contributions of RaiderSTREAM is the addition of multi-node support to STREAM, we have included a multi-node analysis to demonstrate memory bandwidth scalability as the number of nodes used at execution time is increased. The results are reported in Fig. 7 and Fig. 8. For each multi-node test, we ran our hybrid OpenSHMEM + OpenMP implementation using two, four, six, eight, and ten nodes, wherein each processing element (PE) is assigned to a distinct single node, and an individual OpenMP thread is used for each individual CPU core. The same problem sizes of 8.7 million and 22 million for Nocona and Thor, respectively, were borrowed from our first set of tests. Although these problem sizes

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

are meant for single CPUs on these systems under the guidance of section III-F, we use the same STREAM_ARRAY_SIZEs throughout these tests to demonstrate how sustained memory bandwidth scales as we apply an increasing amount of nodes and CPUs to the same problem size, with segments of the problem size distributed evenly across processing elements. As expected, the sustainable memory bandwidth is somewhat linearly scalable for all kernels as the number of nodes is increased with strong scaling, with the exception being a slight decrease in scalability as we increase the number of nodes from eight to ten.

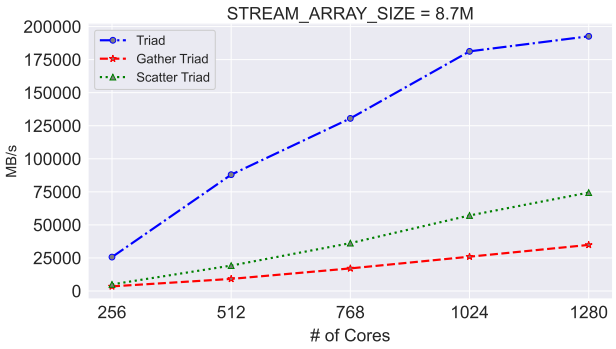

Fig. 7: Multi-Node Performance (Strong Scaling) - Nocona

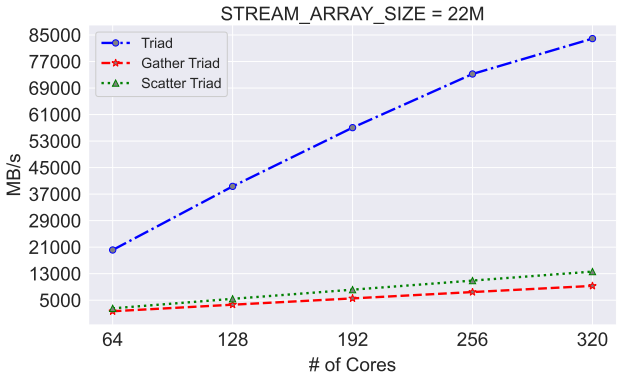

Fig. 8: Multi-Node Performance (Strong Scaling) - Thor

V. Related Work

There have been similar efforts to improve on the STREAM benchmark such as BabelSTREAM [4], [9], which measures memory transfer rates to and from global device memory on GPUs with the use of several different programming models including OpenCL [22], CUDA [18], HIP [5], OpenACC [1], and more as well as support for a variety of programming languages. However, BabelSTREAM only includes the sequential STREAM kernels, and thus does not measure real-world performance. Spatter [3], [15] is another memory bandwidth benchmark that looks specifically at *gather* and *scatter* memory access pattern performance on CPUs and GPUs, also with support for a variety of programming models. Both BabelSTREAM and Spatter have made significant contributions with respect to memory bandwidth evaluation on heterogeneous devices across multiple uses cases. However, both of these benchmark tools lack multi-node support, which makes it impossible to measure or verify scalability across nodes. One of the limitations of RaiderSTREAM is the lack of inter-process communications in our benchmark kernels, which are nearly ubiquitous in the execution of scientific applications in a high-performance computing environment. An existing benchmark that does this is the Ohio State University Network-Based Computing Laboratory's micro-benchmarks for inter-process communications [6] with both MPI and OpenSHMEM.

VI. Conclusion and Future Work

In this paper, we have introduced RaiderSTREAM, our novel HPC-optimized versions of the STREAM benchmark built on the OpenMP, MPI, and OpenSHMEM programming models with supplementary *gather* and *scatter* kernels for measuring memory bandwidth when implementing irregular access patterns. Even with the contributions laid out in this work, there is still much to be done to improve the applicability of our new benchmark implementations. We will in part rely on community utilization and feedback to identify areas for future improvement. However, we have identified some future directions for this research. One limitation of RaiderSTREAM is the lack of customizability with respect to memory access patterns. In RaiderSTREAM, we simulate sparsity and irregularity in our *gather* and *scatter* kernels by way of randomness. This is done in the source code by populating IDX arrays with randomly generated non-repeating integer values ranging from 0 to STREAM_ARRAY_SIZE. The issue with this approach is that users do not have the ability to specify how the IDX arrays are populated without making changes to the source code. In future work, we plan to add functionality to read in memory access patterns via a user-specified input file, allowing users to tailor the benchmark to match their specific use case.

Another future direction for RaiderSTREAM is the incorporation of inter-process communication routines within the benchmark kernels. Our MPI and OpenSHMEM benchmark implementations can partition the arrays across processing elements. However, measuring inter-process communication coupled with irregular memory access would paint a more full picture of system-wide performance and scalability. Another area in which we can improve the benchmark is adding functionality that will simulate hot-spot memory behavior as opposed to total randomness. This would provide insight into memory bandwidth capabilities for applications that frequently access the same memory locations, and can simply be enabled by way of a compilation flag. Furthermore, we can continue improving the versatility of RaiderSTREAM by adding additional memory access patterns such as the *stride-1*, *stride-N*, *pointer-chase* and *central* patterns found in the CircusTent benchmark suite [2], [23], [24].

Acknowledgements

The research reported in this paper was supported by a Los Alamos National Laboratory membership contribution to the U.S. National Science Foundation Industry-University Cooperative Research Center on Cloud and Autonomic Computing

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

(CNS-1939140), and authorized for release under LA-UR-22-29091. This research is also supported in part by the National Science Foundation under grant OAC-1835892. The authors would also like to thank colleagues at the HPC Advisory Council for allowing us to use their Thor system as well colleagues at the Texas Tech University High-Performance Computing Center for allowing us to use their Nocona system.

REFERENCES

[1] "OpenACC," 2011. [Online]. Available: https://www.openacc.org/specification

[2] "CircusTent Benchmark Suite Repository," https://github.com/tactcomplabs/circustent, 2019.

[3] "Spatter Repository," https://github.com/hpcgarage/spatter, 2021.

[4] "BabelSTREAM Repository," https://github.com/UoB-HPC/BabelStream, 2022.

[5] "HIP Repository," https://github.com/ROCm-Developer-Tools/HIP, 2022.

[6] "OSU Micro-Benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks/, 2022.

[7] "STREAM Code Repository," https://github.com/michael-beebe/stream.git, 2022.

[8] A. Mallon, Damian and Taboada, Guillermo and Koesterke, Lars, "MPI and UPC broadcast, scatter and gather algorithms in Xeon Phi," *Concurrency and Computation: Practice and Experience*, vol. 28, 05 2015.

[9] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via babelstream," *International Journal of Computational Science and Engineering*, vol. 17, p. 247, 01 2018.

[10] J. Dongarra, P. Luszczek, and A. Petite, "The linpack benchmark: Past, present and future," *Concurrency and Computation: Practice Experience*, vol. 15, no. 9, pp. 803–820, Aug. 2003.

[11] M. P. I. Forum, "MPI: A Message-Passing Interface Standard Version 4.0," 09 2012, chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[23] B. Williams, J. Leidel, X. Wang, D. Donofrio, and Y. Chen, "Circustent: A benchmark suite for atomic memory operations," in *The International Symposium on Memory Systems*, ser. MEMSYS 2020. New York,

[12] J. Hennessy and D. Patterson, *Computer Architecture - A Quantitative Approach*, 01 2007.

[13] M. A. Jette, A. B. Yoo, and M. Grondona, "Slurm: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.

[14] J. Kim and Y. Kim, "Hbm: Memory solution for bandwidth-hungry processors," in *2014 IEEE Hot Chips 26 Symposium (HCS)*, 2014, pp. 1–24.

[15] P. Lavin, J. Young, R. Vuduc, J. Riedy, A. Vose, and D. Ernst, "Evaluating gather and scatter performance on cpus and gpus," 09 2020, pp. 209–222.

[16] J. D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, a continually updated technical report. http://www.cs.virginia.edu/stream/. [Online]. Available: http://www.cs.virginia.edu/stream/

[17] McCalpin, J. D., "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec 1995.

[18] NVIDIA, P. Vingelmann, and F. H. Fitzek, "CUDA," 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[19] OpenMP Architecture Review Board, "OpenMP application program interface version 5.1," 2020. [Online]. Available: https://www.openmp.org/specifications/

[20] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, 2011, pp. 1–24.

[21] S. W. Poole, O. Hernandez, J. A. Kuehn, G. M. Shipman, A. Curtis, and K. Feind, *OpenSHMEM - Toward a Unified RMA Model*. Boston, MA: Springer US, 2011, pp. 1379–1391.

[22] Stone, John E. and Gohara, David and Shi, Guochun, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.
NY, USA: Association for Computing Machinery, 2020, p. 144–157. [Online]. Available: https://doi.org/10.1145/3422575.3422789

[24] B. Williams, J. D. Leidel, X. Wang, D. Donofrio, and Y. Chen, "CircusTent: A Tool for Measuring the Performance of Atomic Memory Operations on Emerging Architectures," in *Workshop on OpenSHMEM and Related Technologies*. Springer, 2021.

[25] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious. sigarch comput. archit. news 23, 1 (1995), 20–24," 1995.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE