Sum Reduction with OpenMP Offload on NVIDIA Grace-Hopper System

Zheming Jin
Oak Ridge National Laboratory
jinz@ornl.gov

***Abstract*— Sum reduction is a primitive operation in parallel computing. With OpenMP directives that enable data and computation offload to a graphics processing unit (GPU), the work annotates the serial sum reduction with the directives and evaluates the baseline and optimized reductions on an NVIDIA Grace-Hopper system. The study explores the impacts of the number of teams, the number of elements to sum per loop iteration, and simultaneous reduction on the central-processing unit (CPU) and GPU in the unified memory (UM) mode upon the reduction performance. The results show that the optimized reductions are 6.120X to 20.906X faster than the baselines on the GPU, and their efficiency ranges from 89% to 95% of the peak GPU memory bandwidth. Depending on where an input array is allocated in the program when co-running the reduction on the CPU and GPU in the UM mode, the average speedup over the GPU-only execution is approximately 2.484 or 1.067, and the speedup of the optimized reductions over the baseline reductions ranges from 0.996 to 10.654 or from 0.998 to 6.729.**

Keywords—Reduction, OpenMP offload, Unified memory, GPU

I. INTRODUCTION

Sum reduction is a primitive operation commonly used in scientific computing [1, 2, 3, 4]. There are different methods of parallel reduction over a large vector. For example, reductions were implemented and optimized using the CUDA programming model on NVIDIA graphics processing units (GPUs) [5, 6, 7]. OpenMP device offload may simplify the development of an application targeting a GPU by annotating a serial implementation with OpenMP directives. However, it might sacrifice performance for the ease of programming [8, 9, 10, 11].

The focus of this study is a parallel sum reduction with OpenMP device offload on a heterogeneous computing system with a central-processing unit (CPU) and a GPU connected with a hardware-based memory-coherent and high-bandwidth interconnect. The connection facilitates a unified memory (UM) space for a GPU programming language and faster data movement between a CPU and a GPU [12]. More specifically, this work annotates a serial loop of sum reduction with OpenMP directives and specifies a parameter space composed of the number of elements to accumulate in each loop iteration and the number of teams. Then, it sweeps over the space on the GPU in the system for performance exploration. With the performance optimization, it further explores simultaneous reduction on both devices in the UM mode. The experimental results show that the reduction performance can be improved by exploring the numbers of teams and the numbers of elements to accumulate in a loop iteration. When the OpenMP clause for the number of teams is not specified by a programmer, the heuristics may be further optimized in a compiler's implementation of the OpenMP reduction. Distributing the reduction across the CPU and the GPU in the UM mode could achieve higher performance than the CPU-only or GPU-only execution.

With the descriptions of the motivation and scope of the study, the rest of the paper is organized as follows. Section II introduces the OpenMP device offload support and the Grace-Hopper system. Section III describes the implementations and evaluations of the reductions with OpenMP offload on the GPU. Section IV describes the implementations and evaluations of simultaneous reductions on the CPU and the GPU. Section V summarizes related work, and Section VI concludes the paper.

II. BACKGROUND

A. Sum Reduction

The focus of the work is an unsegmented form of sum reduction. Listing 1 shows the sequential sum reduction as a reference. Taking a binary associative operator "+" and an array of "M" numbers as inputs, the reduction returns as output one value. To make the reduction more generic, the type of each input number is "T" and the result is of type "R". The data types are not necessarily the same. "M" is a 64-bit integer. From the perspective of device offload, the input numbers are copied from a host to a device, and then reduced in parallel, and finally the output of the reduction is copied back to a host.

```
T in[M];
R sum = 0;
for (i = 0; i < M; i++)
  sum += in[i];
```

Listing 1. The sequential sum reduction

A parallel implementation divides the reduction into independent partial sums, computes each partial sum in an arbitrary order, and produces a result by combining these partial sums. The idea can be generalized to reductions on vectors of arbitrary size.

B. OpenMP

OpenMP is an evolving standard that makes it easier to write portable, heterogeneous parallel codes. OpenMP-specific directives allow a user to parallelize C, C++ or Fortran applications with code annotation. Features have been added to the specification to keep up with heterogeneous computing [13]. In the specification, the OpenMP target directives specify that a region of code should be executed on a target device (e.g., GPU). An OpenMP application utilizing device offload is commonly labelled as an OpenMP target offload application. Multiple open-source and commercial compilers support execution of OpenMP code regions on a GPU device. The OpenMP 4.5 specification provides

significant support for device offload [14]. The latest OpenMP specifications add clarifications and feature enhancements to OpenMP device offload [15].

C. The NVIDIA Grace Hopper System

The system has distinct physical memory regions attached to the Grace CPU and the Hopper GPU. The ARM datacenter CPU is connected to the low-power double data rate 5X (LPDDR5X) memory subsystem [16] while the NVIDIA Hopper GPU is equipped with the third-generation high bandwidth memory (HBM3). The CPU and the GPU communicate via the high-bandwidth NVLink Chip-2-Chip interconnect [17]. The technology facilitates a unified single address space where the CPU and GPU can access memory without explicit data movement [18] and benefits most applications with minimal porting efforts [19]. Overall, the system provides a high-performance solution for scientific applications and benchmarks [20].

This work uses a Grace-Hopper (GH) system as a computing testbed. It consists of a 72-core ARM Neoverse V2 CPU, and an NVIDIA H100 GPU. The CPU has 480 GB LPDDR5X memory, and the GPU has 96 GB HBM3 memory. The operating system is Red Hat Enterprise Linux 9.3 with CUDA 12.4 and GPU driver 550.54.15. The peak GPU memory bandwidth is 4022.7 GB/s.

III. SUM REDUCTION WITH OPENMP OFFLOAD ON GPU

A. Parallel Reduction with OpenMP Offload

The OpenMP reduction clause specifies thread-private variables that are subject to a reduction operation in a parallel region. All supported reduction operators are described in the OpenMP specification. The OpenMP reduction clause specifies a reduction-identifier and one or more list items. In OpenMP C/C++, a reduction-identifier is an arithmetic, logical, or comparison operator. A list item is a variable that will combine private copies of the variable using the operator-associated combiner at the end of the parallel region. Annotating the sequential version of the reduction with OpenMP directives, a parallel version of the sum reduction is shown in Listing 2.

```
#pragma omp target teams distribute parallel for \
                                    reduction(+:sum)
for (i = 0; i < M; i++) {
  sum += in[i];
}
```

Listing 2. Sum reduction with the OpenMP offload directives

```
#pragma omp target teams distribute parallel for \
 num_teams(teams) thread_limit(threads) reduction(+:sum)
for (i = 0; i < M; i++) {
  sum += in[i];
}
```

Listing 3. Sum reduction with the specifications of team size and thread size in the OpenMP offload directives

The "target teams distribute parallel" worksharing-loop construct is semantically equivalent to explicitly specifying a "target" directive immediately followed by a "teams distribute parallel" worksharing-loop directive. The "target" construct maps variables to a device data environment and executes the construct on that device. The loop has a canonical loop form.

In addition, the OpenMP standard supports two clauses that allows a user to specify the number of teams and the number of threads in a team for performance tuning. The "num_teams" clause sets the bounds on the number of teams. The "thread_limit" clause specifies an upper bound to the number of threads that may participate in a contention group initiated by each team. The number of teams and the number of threads created by an OpenMP runtime are implementation defined. However, the runtime will process and check any values requested by a user through directives or environment variables. Listing 3 shows the two clauses.

Inspired by the vectorized memory accesses that can improve the reduction performance on accelerators [21, 22, 23], this work proposes to sum up "V" elements, where "V" is a power of two, in each loop iteration as shown in Listing 4. Compared to the loop construct in Listing 3, the number of teams are reduced by a factor of "V" and the loop counter is incremented by "V" after executing the loop body.

```
#pragma omp target teams distribute parallel for \
 num_teams(teams/V) thread_limit(threads) \
 reduction(+:sum)
for (i = 0; i < M; i = i + V) {
  sum += in[i] + in[i+1] + … + in[i+V-1];
}
```

Listing 4. Each loop iteration accumulates V elements.

Compiling the OpenMP program shows that the OpenMP compiler in the NVIDIA HPC Software Development Kit (NVHPC SDK) [24] may fail to build the program because the loop increment is not in a supported form. Hence, the loop is rewritten as shown in Listing 5 to address the issue.

```
#pragma omp target teams distribute parallel for \
 num_teams(teams/V) thread_limit(threads) \
 reduction(+:sum)
for (m = 0; m < M / V; m++) {
  i = V × m;
  sum += in[i] + in[i+1] + … + in[i+V-1];
}
```

Listing 5. Rewrite the loop in Listing 4

B. Evaluation Setup

This work chooses four use cases for evaluation. In the first case (C1), both input and output values are 32-bit signed integers ("T", "R" = int32). The number of 32-bit integers to reduce on the GPU is 1048576000, approximately 4 GB in memory size. In the second case (C2), each input number is an 8-bit signed integer ("T" = int8), and the output is a 64-bit

```
// Start timing
for (n = 0; n < N; n++) {
  sum = 0;
  #pragma omp target update to(sum)
  // Same as the code snippet in Listing 5
  #pragma omp target update from(sum)
}
// Stop timing

bandwidth = 1e-9 × M × sizeof(T) * N / elapsed_time
```

Listing 6. Performance measurement of the sum reduction

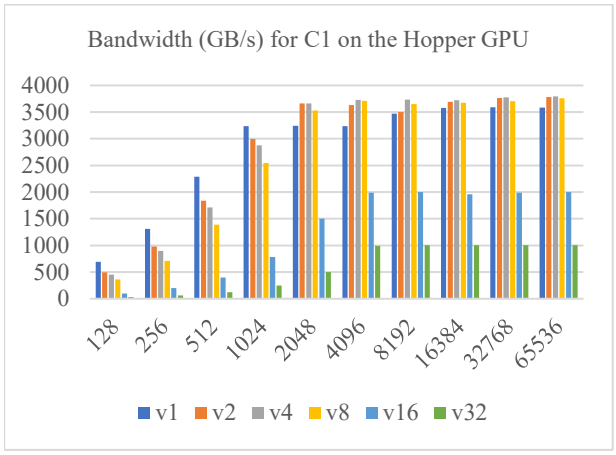

Fig. 1a. Reduction performance (y-axis) as a function of the number of teams (x-axis) and the number of elements to add per loop iteration (v) on the GPU. For C1, "T" and "R" are int32.

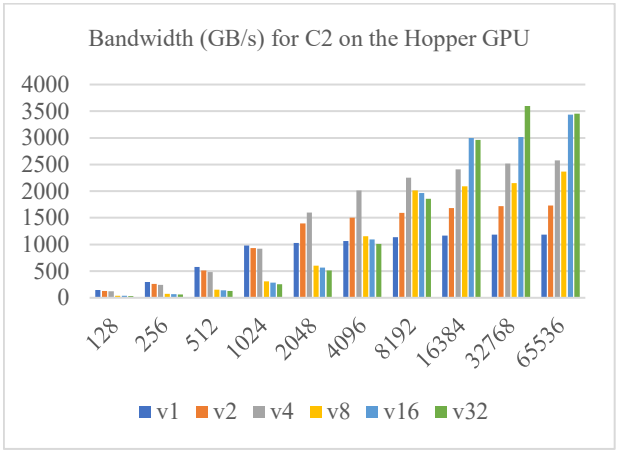

Fig. 1b. Reduction performance (y-axis) as a function of the number of teams (x-axis) and the number of elements to add per loop iteration (v) on the GPU. For C2, "T" is int8 and "R" is int64.

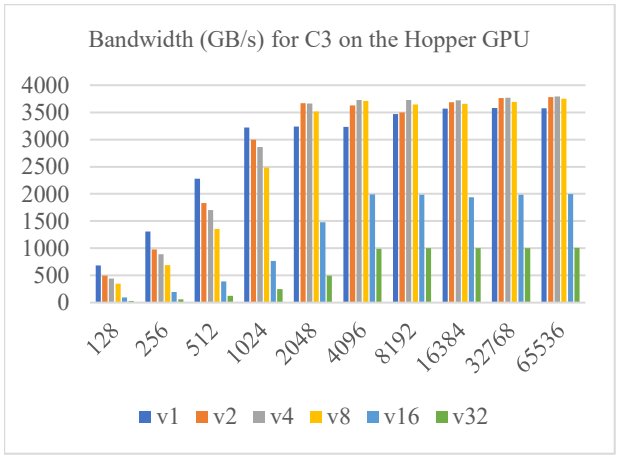

Fig. 1c. Reduction performance (y-axis) as a function of the number of teams (x-axis) and the number of elements to add per loop iteration (v) on the GPU. For C3, "T" and "R" are float32.

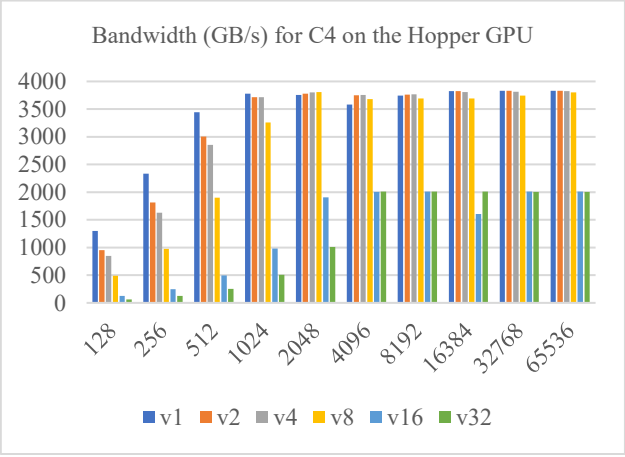

Fig. 1d. Reduction performance (y-axis) as a function of the number of teams (x-axis) and the number of elements to add per loop iteration (v) on the GPU. For C4, "T" and "R" are float64.

signed integer ("R" = int64). The number of 8-bit integers is four times the number of 32-bit integers in C1. In the third case (C3), both input and output values are single-precision floating-point numbers ("T", "R" = float32). In the last case (C4), both input and output values are double-precision floating-point numbers ("T", "R" = float64). Both C3 and C4 reduce 1048576000 floating-point numbers. Reduction over sufficiently large numbers may mitigate under-utilization of GPU sources such as compute units and memory bandwidth.

The OpenMP program is compiled with the NVHPC SDK, version 12.5. The compiler optimization option is "-O3". The total time of executing the reduction for N (N = 200) times is measured with the C++ standard chrono library. As shown in Listing 6, the elapsed time comprises the initialization of the "sum" variable for each kernel execution, kernel execution time on a device, and result copyback after the reduction is complete. Note the host-to-device transfer of input numbers is not included in the timing measurement. The performance of the reduction is reported with the bandwidth metric (GB/s). It measures the reduction performance when data resides on a GPU. The GPU results are verified using the CPU results.

C. Experimental Results

The sum reduction annotated with the OpenMP directives indicates that the parameter space for the reduction spans over three dimensions: the number of OpenMP teams, the OpenMP thread limits, and the number of elements to add in each loop iteration. The parameter search space may be reduced by setting the OpenMP thread limit to 256. The number of teams and the number of elements to accumulate in a loop are power-of-two numbers, ranging from 128 to 65536 and 1 to 32, respectively.

Figures 1 show the reduction performance as a function of the number of teams and the number of elements on the GPU for the four cases. The number of elements to sum per loop iteration is prefixed with the variable "v". For example, "v4" means that there are four elements to add in each loop

iteration. The team size for the "num_teams" clause (not shown in the figures) is the number of teams divided by the number of elements added per loop. Before a threshold is reached, increasing the team size could improve the reduction performance regardless of the number of elements to add per loop iteration. The increase turns a compute-bound kernel into a memory-bound kernel. For C1, C3, and C4, the performance becomes almost saturated when the number of teams is 4096. For C2, the performance becomes almost saturated when the number of teams is 32768. The highest bandwidths are 3795 GB/s, 3596 GB/s, 3790 GB/s, and 3833 GB/s for the four cases, respectively.

Profiling the OpenMP program reveals that the grid sizes of the GPU reduction kernels match the team sizes specified by the "num_teams" clause. When the clause is not specified, the OpenMP runtime selects a grid size that is equal to the number of input values divided by the number of threads in a team for C1, C3, and C4. The grid size is 16777215 (0xFFFFFF) for C2, which is less than the number of input values divided by the number of threads in a team. The number of threads in a team is 128 in any case.

Table 1 lists the performance of the baseline reduction shown in Listing 2, the highest performance of the optimized reduction, performance speedup, and efficiency for each case. The optimized reductions can improve the performance of the baseline reductions with a speedup ranging from 6.120 to 20.906. "Efficiency" is defined as the ratio of the OpenMP reduction performance in the experiment to the peak GPU memory bandwidth. The efficiency of the baseline reductions is capped at 15.4%. The optimized reductions achieve approximately 95% efficiency for C1, C3, and C4. For C2, the efficiency is approximately 89% though the speedup is the highest. The comparison shows that the heuristics may be further optimized in the vendor's implementation of the OpenMP reduction.

Table 1. Performance evaluation and comparison of the baseline and optimized sum reductions in OpenMP device offload on the GPU

Case	Base (GB/s)	Optimized (GB/s)	Speedup	Efficiency (%)
C1	620	3795	6.120	15.4 / 94.3
C2	172	3596	20.906	4.3 / 89.4
C3	271	3790	13.985	6.7 / 94.2
C4	526	3833	7.287	13.1 / 95.3

IV. EXECUTION OF SUM REDUCTION ON CPU AND GPU

As mentioned in Section III, the reduction performance is measured without considering the movement of an input array from the host to the device. In practice, the offload overhead may be considered as an integral part of a reduction application. While the reduction is offloaded to the GPU, the CPU is idle until the offloaded computation is complete. Using both the GPU and the CPU simultaneously [25] may improve the reduction performance. In addition, it will be interesting to evaluate the reduction performance with the UM technology available in the GH system.

```
LenH = M × CPU_part;
LenD = M − LenH;
sumD = sumH = 0;

#pragma omp parallel
{
  #pragma omp master
  {
    #pragma omp target teams distribute
     parallel for nowait map(to: inD[0:LenD])
    for (i = 0; i < LenD; i++)
      sumD += inD[i];
  }
  #pragma omp for simd
  for (i = 0; i < LenH; i++)
    sumH += inH[i];
}
sum = sumD + sumH;
```

Listing 7. The OpenMP implementation utilizes the CPU and GPU simultaneously. The lengths of the input array to reduce on the host and device are represented with `LenH` and `LenD`, respectively. `inD` and `inH` are pointers to the same input array at the proper places. The baseline reduction on the device is shown in the code snippets.

A. Parallel Reduction with OpenMP Offload

Listing 7 shows the implementation of the co-execution using OpenMP directives. The OpenMP "master" directive could exploit simultaneous CPU and GPU execution. The "master" directive specifies a structured block that will be executed by the master thread of the team. Other threads in the team will not execute the associated structured block. There is no implied barrier either on entry to or exit from the master construct [15]. In the structured block, part of the reduction work is offloaded to the GPU with the OpenMP target directive. The data "map" clause manages the data movement from the host to the device. The "nowait" clause avoids synchronization between the threads running on the CPU and GPU. The remaining work is parallelized on the CPU. The "simd" directive may instruct the compiler to make the loop vector-friendly by having multiple loop iterations executed simultaneously. The directive may provide tuning hints for CPU targets; the directive is ignored for GPU targets [24]. There is an implicit barrier at the end of the OpenMP

```
// A1: Allocate memory for input array
for (p = 0; p <= 1; p = p + 0.1) {
  // A2: Allocate memory for input array
  // Compute the array lengths for the CPU and GPU
  // Start timing
  for (n = 0; n < N; n++) {
    sumH = sumD = 0;
    #pragma omp parallel
    {
      // Simultaneous reduction on the CPU and GPU
    }
    sum = sumH + sumD;
  }
  // Stop timing
  bandwidth = 1e-9 × M × sizeof(T) * N / elapsed_time
}
```

Listing 8. Performance measurement of the sum reduction. The CPU part of the reduction ("p") ranges from 0 to 1. Memory allocation for the input array occurs at A1 or A2.

parallel region. Finally, the partial sums from the CPU and GPU are added to produce the final sum.

UM is supported on the GH system, and the feature is enabled with the option `-gpu=mem:unified` added to the command line of the OpenMP compiler. In the UM mode, the compiler assumes that system memory is accessible on the GPU. The whole program state is shared between the CPU and the GPU. The GPU does not operate on a copy of the data even if the program contains respective directives. The compiler may utilize CUDA managed memory for dynamic allocations by automatically replacing explicit memory allocation and deallocation (e.g., calls to malloc/free functions) with `cudaMallocManaged`- and `cudaFree`-style allocation and deallocation [26]. The "map" clause will not result in any device allocation or data transfer. The OpenMP runtime may leverage such a clause to communicate preferable data placement to the CUDA runtime by means of memory hint [24].

B. Evaluation

The total time of offloading the reduction for N (N = 200) times is measured with the C++ standard chrono library. As shown in Listing 8, the elapsed time comprises the initialization of partial sums, co-execution of the reduction on the CPU and GPU, and the sum of partial sums. The allocation of an input array can occur before (A1) or after (A2) the loop iteration over the CPU part ("p") of the reduction work. The impacts of the two locations upon the reduction performance will be assessed.

Figure 2a shows the performance of the baseline reduction with respect to the CPU part of the reduction in the UM mode. The allocation of the input array occurs at A1. When the CPU part is zero, the reduction is offloaded to the GPU entirely. When it is one, no reduction is offloaded to the GPU. The reduction performance for C1 and C3 are almost the same. The reduction performance for C2 and C4 are also close to each other. Distributing the reduction across both devices could achieve higher performance than the CPU-only or GPU-only execution. The highest speedups over the GPU-only execution are 2.732, 2.246, 2.692, and 2.297 for the four cases, respectively. The average speedup is approximately 2.492.

Figure 2b shows the performance of the optimized reduction with respect to the CPU part of the reduction in the UM mode. The allocation of the input array occurs at A1. Based on the reduction performance shown in Figures 1, the values of the parameters that result in saturated bandwidth on the GPU are selected. The number of teams is 65536. "V" equals 4 for C1, C3, and C4. For C2, "V" equals 32. Distributing the reduction across both devices could achieve higher performance than the CPU-only or GPU-only execution. The highest speedups over the GPU-only execution are 2.253, 3.385, 2.100, and 2.197 for the four cases, respectively. The average speedup is approximately 2.484.

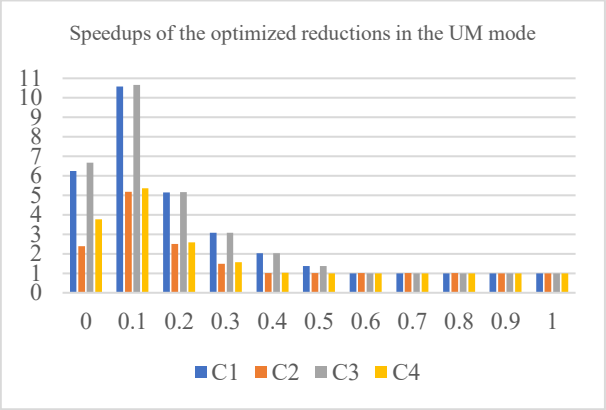

Fig. 3. Performance speedups of the optimized reductions over the baselines when co-running the reduction in the UM mode. The CPU part of the reduction ranges from 0 to 1. The input array is allocated at A1 for each case.

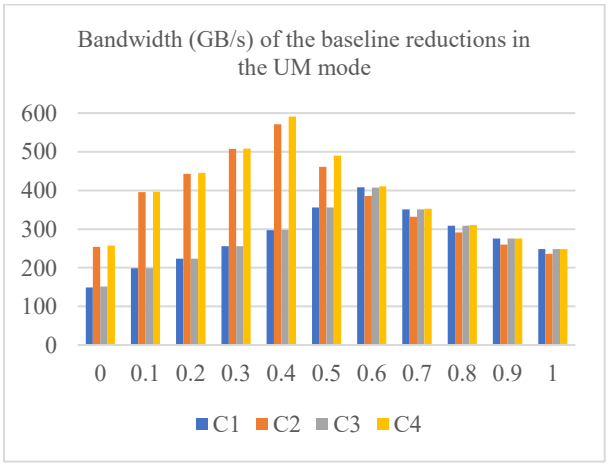

Fig. 2a. Performance of the baseline reductions in the UM mode for the four cases. The workload is distributed between the CPU and the GPU. The CPU part of the reduction ranges from 0 to 1. The input array is allocated at A1 for each case.

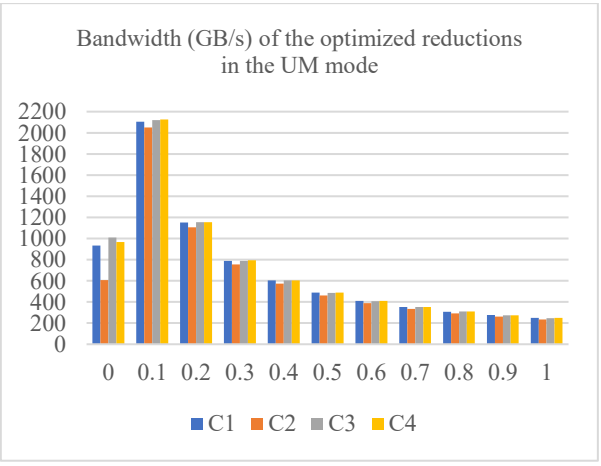

Fig. 2b. Performance of the optimized reductions in the UM mode for the four cases. The workload is distributed between the CPU and the GPU. The CPU part of the reduction ranges from 0 to 1. The input array is allocated at A1 for each case.

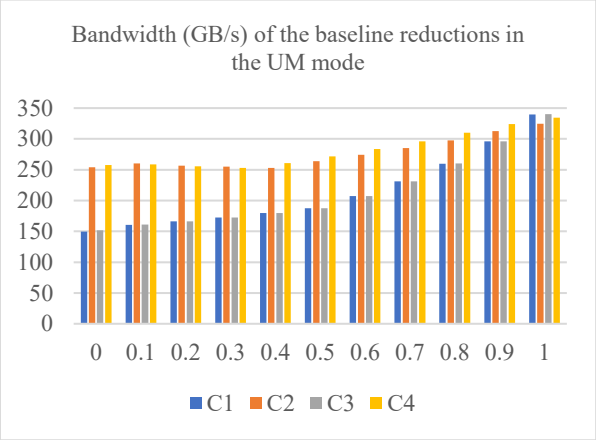

Fig. 4a. Performance of the baseline reductions in the UM mode for the four cases. The workload is distributed between the CPU and the GPU. The CPU part of the reduction ranges from 0 to 1. The input array is allocated at A2 for each case.

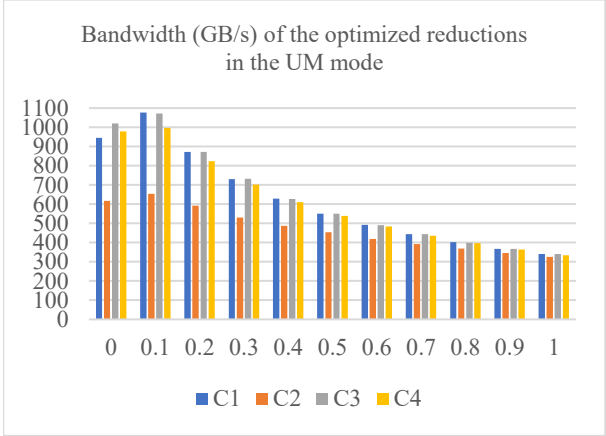

Fig. 4b. Performance of the optimized reductions in the UM mode for the four cases. The workload is distributed between the CPU and the GPU. The CPU part of the reduction ranges from 0 to 1. The input array is allocated at A2 for each case.

Figure 3 shows the performance speedups of the optimized reductions in Figure 2b over the baseline reductions in Figure 2a with respect to the CPU part of the reduction in the UM mode. The speedup ranges from 0.996 to 10.654. The speedups are significant when the GPU parts account for at least 50% of the total workloads.

Figure 4a shows the performance of the baseline reduction with respect to the CPU part of the reduction in the UM mode when the allocation of the input array occurs at A2. The performance trends indicate that distributing the reduction across both devices does not achieve higher performance than the CPU-only execution.

Figure 4b shows the performance of the optimized reduction with respect to the CPU part of the reduction in the UM mode when the allocation of the input array occurs at A2. Distributing the reduction across both devices could achieve higher performance than the CPU-only or GPU-only execution. The highest speedups over the GPU-only execution are 1.139, 1.062, 1.050, and 1.017 for the four cases, respectively. The average speedup is approximately 1.067.

Figure 5 shows the performance speedups of the optimized reductions in Figure 4b over the baseline reductions in Figure 4a with respect to the CPU part of the reduction in the UM mode. The speedup ranges from 0.998 to 6.729. The speedups are significant when the GPU parts account for at least 90% of the total workloads.

The performance of co-running the optimized reductions with A1 is on average 2.299X higher than that with A2. However, the performance of the CPU-only reduction with A1 is 1.367X lower than that with A2. The memory for the input array is allocated once in A1 whereas it is allocated per loop iteration in A2. The initialization of the input array is performed on the CPU, so memory pages are placed on the CPU during initialization. For the loop over the CPU part of the workload, the reduction is offloaded entirely to the GPU in the first loop iteration. Hence, the pages that will be accessed by the GPU are migrated from the CPU memory to

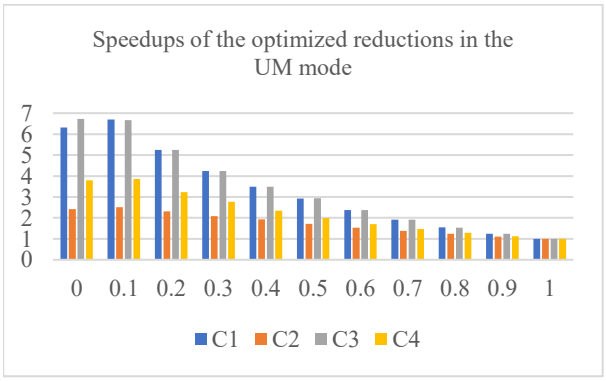

Fig. 5. Performance speedups of the optimized reductions over the baselines when co-running the reduction in the UM mode. The CPU part of the reduction ranges from 0 to 1. The input array is allocated at A2 for each case.

the GPU memory to improve the performance. For A1, these migrated pages will reside in the GPU memory, improving the reduction performance on the GPU as there is no migration overhead from the CPU to the GPU for subsequent loop iterations. For A2, allocating memory for each loop iteration will also enable automatic migration of memory regions between the GPU and CPU physical memory, but the migration overhead occurs per iteration. When the reduction runs entirely on the CPU in A2, there is no data migration from the GPU memory to the CPU memory or data read from the HBM memory by the CPU since memory pages are placed on the CPU memory during initialization.

V. Related Work

In [27], the performance of the integer sum reduction with OpenMP offload is evaluated on an AMD MI100 discrete GPU. This work evaluates the integer and floating-point reductions on an NVIDIA Grace-Hopper system. The experimental results show that the proposed reduction can

improve the baseline performance on the NVIDIA GPU. In [28], the performance of the baseline and optimized integer sum reductions written in HIP is evaluated on an AMD GPU. To improve the reduction performance over the baselines, the OpenMP offload implementations require less modifications to the existing code base while the HIP or CUDA implementations need more coding effort. In [29], the advantages and disadvantages of abstractions for reductions from different programming models are discussed. In this work, the OpenMP abstraction is the focus. It will be interesting to evaluate other abstractions in future studies. In [30], the authors find that it is beneficial to use additional private variables to reduce thread local contributions before reducing into OpenMP reduction variables. However, whether this technique can improve the performance of an application on an NVIDIA GPU depends on the implementation of an OpenMP compiler. In this work, adding multiple elements per loop iteration (i.e., thread local contribution) and exploring team sizes can improve the reduction performance with the vendor's compiler. In [14], the authors acknowledge that it is challenging for a compiler runtime to determine the best grid geometry for each target region. Hence, heuristics are often adopted by an implementation that yields high GPU occupancy. In this experiment, the performance of the baseline reductions indicates that the heuristics may be further optimized in the implementations of the OpenMP reduction. In [18], the authors introduce an instrumentation tool for automatic offload of a subset of the CPU linear algebra library calls to the GPU on the GH system without user code modifications or recompilation. The mechanism is commonly used in profilers. In our work, the offload of the reduction operations is implemented with OpenMP directives entirely.

In the co-execution experiments for the full airplane simulation on an IBM Power 9 cluster with NVIDIA V100 GPUs [31], the authors find that co-execution reduces the maximum elapsed time for the pure GPU execution by 23%. When benchmarking query performance of database engines on the NVIDIA Jetson AGX Xavier and AMD A10-7850K edge devices, the implementations of a co-running model obtain 2.95X bandwidth utilization over the GPU-only methods and 1.82X over the CPU-only methods [32]. On the Jetson platform, CUDA and OpenMP are the programming models for the GPU and CPU, respectively. After corunning the machine learning benchmarks on the AMD Ryzen 5 2400G architecture with an integrated GPU, the authors find that most applications gain performance benefits from CPU-GPU co-running, that the optimal work partitioning ratio varies with the characteristics of the workloads, and that the performance of co-running with zero-copy is always higher regardless of the ratio of task partitioning [33]. For six benchmarks that represent regular and irregular memory behaviors, the speedup of co-execution over the GPU execution approximately ranges from 1 to 2.5 with dynamic scheduling and a unified shared memory programming model on an Intel Core i5-7500 CPU with an integrated GPU [34]. In this work, co-running the reduction in the UM mode could achieve higher performance than the GPU-only execution. The performance speedups and optimal partition ratio depend on the optimization of the reduction kernels on the GPU, the data types of array elements, and where the array is allocated in the program.

VI. Conclusion

This work annotates a serial loop of sum reduction with OpenMP target directives for parallel execution on a target device. Then, it describes the proposed reduction in which the numbers of teams and the number of elements to sum per loop iteration are explicitly specified by a programmer. Comparing the baseline reductions with the proposed reductions on the NVIDIA Hopper GPU shows that the performance speedup of the optimized reductions over the baselines on the GPU ranges from 6.120 to 20.906, and the efficiency ranges from 89% to 95% of the theoretical memory bandwidth.

The study finds that simultaneous execution of the reduction on the CPU and GPU in the unified memory (UM) mode can improve the reduction performance compared to the CPU-only or GPU-only execution. However, where the input array is allocated in the program affects the performance of co-running the reduction. When the input array is allocated before the iteration of the CPU part of the workload, the average speedup over the GPU-only execution is approximately 2.484, the speedup of the optimized reductions over the baseline reductions ranges from 0.998 to 10.654, and the speedup is significant when the GPU parts account for at least 50% of the total work. When the input array is allocated for every iteration of the CPU part of the workload, the average speedup over the GPU-only execution is approximately 1.07, the speedup of the optimized reductions over the baseline reductions ranges from 0.998 to 6.729, and the speedup is significant when the GPU parts account for at least 90% of the total work. The data migration overhead from the CPU memory to the GPU memory contributes to the performance slowdown.

Future studies would evaluate other reduction abstractions and performance of applications that use sum reduction on the GH system or on other vendors' platforms where full UM capabilities are supported.

Acknowledgment

The author appreciates the anonymous reviewers for their comments and suggestions. The author is grateful for Joel Denny for the technical review. This research used resources at the University of Oregon Neuroinformatics Center.

References

[1] Viswanathan, G. and Larus, J.R., 1996. User-defined reductions for communication in data-parallel languages. University of Wisconsin-Madison Department of Computer Sciences.

[2] Deitz, S.J., Chamberlain, B.L. and Snyder, L., 2002. High-level language support for user-defined reductions. The Journal of Supercomputing, 23, pp.23-37.

[3] Kambadur, P., Gregor, D. and Lumsdaine, A., 2008. OpenMP extensions for generic libraries. In OpenMP in a New Era of Parallelism: 4th International Workshop, IWOMP 2008 West Lafayette, IN, USA, May 12-14, 2008 Proceedings 4 (pp. 123-133). Springer Berlin Heidelberg.

[4] Gan, G., Wang, X., Manzano, J. and Gao, G.R., 2009. Tile reduction: The first step towards tile aware parallelization in openmp. In Evolving

OpenMP in an Age of Extreme Parallelism: 5th International Workshop on OpenMP, IWOMP 2009 Dresden, Germany, June 3-5, 2009 Proceedings 5 (pp. 140-153). Springer Berlin Heidelberg.

[5] Dotsenko, Y., Govindaraju, N.K., Sloan, P.P., Boyd, C. and Manferdelli, J., 2008, June. Fast scan algorithms on graphics processors. In Proceedings of the 22nd annual international conference on Supercomputing (pp. 205-213).

[6] De Gonzalo, S.G., Huang, S., Gómez-Luna, J., Hammond, S., Mutlu, O. and Hwu, W.M., 2019, February. Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on GPUs. In 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (pp. 73-84). IEEE.

[7] https://nvidia.github.io/cccl/cub/. NVIDIA Corporation.

[8] Martineau, M., McIntosh-Smith, S. and Gaudin, W., 2016, May. Evaluating OpenMP 4.0's effectiveness as a heterogeneous parallel programming model. In 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 338-347). IEEE.

[9] Karlin, I., Scogland, T., Jacob, A.C., Antao, S.F., Bercea, G.T., Bertolli, C., de Supinski, B.R., Draeger, E.W., Eichenberger, A.E., Glosli, J. and Jones, H., 2016, October. Early experiences porting three applications to OpenMP 4.5. In International Workshop on OpenMP (pp. 281-292). Springer, Cham.

[10] Bercea, G.T., Bertolli, C., Antao, S.F., Jacob, A.C., Eichenberger, A.E., Chen, T., Sura, Z., Sung, H., Rokos, G., Appelhans, D. and O'Brien, K., 2015, November. Performance analysis of OpenMP on a GPU using a CORAL proxy application. In Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems (pp. 1-11).

[11] Diaz, J.M., Friedline, K., Pophale, S., Hernandez, O., Bernholdt, D.E. and Chandrasekaran, S., 2019. Analysis of OpenMP 4.5 offloading in implementations: correctness and overhead. Parallel Computing, 89, p.102546.

[12] NVIDIA GH200 Grace Hopper Superchip Datasheet. NVIDIA Corporation.

[13] Daley, C.S., Southwell, A., Gayatri, R., Biersdorfff, S., Toepfer, C., Özen, G. and Wright, N.J., 2021, November. Non-recurring engineering (NRE) best practices: a case study with the NERSC/NVIDIA OpenMP contract. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-14).

[14] Tiotto, E., Mahjour, B., Tsang, W., Xue, X., Islam, T. and Chen, W., 2019. OpenMP 4.5 compiler optimization for GPU offloading. IBM Journal of Research and Development, 64(3/4), pp.1-14.

[15] OpenMP Architecture Review Board. 2020. OpenMP Application Programming Interface. Version 5.1 November 2020. OpenMP Architecture ReviewBoard. Retrieved August 20, 2021 from https://www.openmp.org/wp-content/uploads/OpenMPAPI-Specification-5-1.pdf

[16] Evans, J., 2022, August. NVIDIA Grace. In 2022 IEEE Hot Chips 34 Symposium (HCS) (pp. 1-20). IEEE Computer Society.

[17] Wei, Y., Huang, Y.C., Tang, H., Sankaran, N., Chadha, I., Dai, D., Oluwole, O., Balan, V. and Lee, E., 2023, February. 9.3 NVLink-C2C: A coherent off package chip-to-chip interconnect with 40Gbps/pin single-ended signaling. In 2023 IEEE International Solid-State Circuits Conference (ISSCC) (pp. 160-162). IEEE.

[18] Li, J., Wang, Y., Liang, X. and Liu, H., 2024. Automatic BLAS Offloading on Unified Memory Architecture: A Study on NVIDIA Grace-Hopper. In Practice and Experience in Advanced Research Computing 2024: Human Powered Computing (pp. 1-5).

[19] Schieffer, G., Wahlgren, J., Ren, J., Faj, J. and Peng, I., 2024. Harnessing Integrated CPU-GPU System Memory for HPC: a first look into Grace Hopper. arXiv preprint arXiv:2407.07850.

[20] Simakov, N.A., Jones, M.D., Furlani, T.R., Siegmann, E. and Harrison, R.J., 2024, January. First Impressions of the NVIDIA Grace CPU Superchip and NVIDIA Grace Hopper Superchip for Scientific Workloads. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops (pp. 36-44).

[21] Jin, Z. and Finkel, H., 2018, May. Optimizing an atomics-based reduction kernel on OpenCL FPGA platform. In 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 532-539). IEEE.

[22] Jin, Z. and Finkel, H., 2019, May. Exploring Integer Sum Reduction using Atomics on Intel CPU. In Proceedings of the International Workshop on OpenCL (pp. 1-6).

[23] Jin, Z. and Vetter, J., 2021, August. Evaluating the Performance of Integer Sum Reduction in SYCL on GPUs. In 50th International Conference on Parallel Processing Workshop (pp. 1-8).

[24] NVIDIA HPC compiler User Guide. https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html

[25] Raju, K. and Chiplunkar, N.N., 2018. A survey on techniques for cooperative CPU-GPU computing. Sustainable Computing: Informatics and Systems, 19, pp.72-85.

[26] CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[27] Jin, Z. and Vetter, J.S., 2022, May. Integer Sum Reduction with OpenMP on an AMD MI100 GPU. In 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 496-499). IEEE.

[28] Jin, Z., Vetter, J. and Vetter, J., 2022, August. A Study on Atomics-based Integer Sum Reduction in HIP on AMD GPU. In Workshop Proceedings of the 51st International Conference on Parallel Processing (pp. 1-10).

[29] T. Deakin, S. McIntosh-Smith, S. J. Pennycook and J. Sewall, Analyzing Reduction Abstraction Capabilities, 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC), St. Louis, MO, USA, 2021, pp. 33-44

[30] Davis, J.H., Daley, C., Pophale, S., Huber, T., Chandrasekaran, S. and Wright, N.J., 2021. Performance assessment of OpenMP compilers targeting NVIDIA V100 GPUs. In Accelerator Programming Using Directives: 7th International Workshop, WACCPD 2020, Virtual Event, November 20, 2020, Proceedings 7 (pp. 25-44). Springer International Publishing.

[31] Borrell, R., Dosimont, D., Garcia-Gasulla, M., Houzeaux, G., Lehmkuhl, O., Mehta, V., Owen, H., Vázquez, M. and Oyarzun, G., 2020. Heterogeneous CPU/GPU co-execution of CFD simulations on the POWER9 architecture: Application to airplane aerodynamics. Future Generation Computer Systems, 107, pp.31-48.

[32] Liu, J., Zhang, F., Li, H., Wang, D., Wan, W., Fang, X., Zhai, J. and Du, X., 2022. Exploring query processing on CPU-GPU integrated edge device. IEEE Transactions on Parallel and Distributed Systems, 33(12), pp.4057-4070.

[33] Zhang, C., Zhang, F., Guo, X., He, B., Zhang, X. and Du, X., 2020. iMLBench: A machine learning benchmark suite for CPU-GPU integrated architectures. IEEE Transactions on Parallel and Distributed Systems, 32(7), pp.1740-1752.

[34] Nozal, R. and Bosque, J.L., 2021. Exploiting co-execution with oneAPI: heterogeneity from a modern perspective. In Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings 27 (pp. 501-516). Springer International Publishing.