IRIS-DMEM: Efficient Memory Management for Heterogeneous Computing

Narasinga Rao Miniskar, Mohammad Alaul Haque Monil, Pedro Valero-Lara, Frank Y. Liu, Jeffrey S. Vetter
Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA
{miniskarnr, monilm, valerolarap, liufy, vetter}@ornl.gov

***Abstract*—This paper proposes an efficient data memory management approach for the Intelligent RuntIme System (IRIS) heterogeneous computing framework along with new data transfer policies. IRIS provides a task-based programming model for extreme heterogeneous computing (e.g., CPU, GPU, DSP, FPGA) with support for today's most important programming languages (e.g., OpenMP, OpenCL, CUDA, HIP, OpenACC). However, the IRIS framework either forces the programmer to introduce data transfer commands for each task or relies on suboptimal memory management for automatic and transparent data transfers. The work described here extends IRIS with novel heterogeneous memory handling and introduces novel data transfer policies by employing the *Distributed data MEMory handler* (DMEM) for efficient and optimal movement of data among the various computing resources. The proposed approach achieves performance gains of up to 7× for tiled LU factorization and tiled DGEMM (i.e., matrix multiplication) benchmarks. Moreover, this approach also reduces data transfers by up to 71% when compared to previous IRIS heterogeneous memory management handlers. This work compares the performance results of the IRIS framework's novel DMEM with the StarPU runtime and MAGMA math library for GPUs. Experiments show a performance gain of up to 1.95× over StarPU and 2.1× over MAGMA.**

***Index Terms*—Heterogeneous computing, IRIS, memory, data transfer**

I. Introduction

This paper describes efforts to implement the *Distributed data MEMory handler* (DMEM), which is a novel and disruptive method for managing data transfers on extremely heterogeneous memory hierarchies. As part of the Intelligent RuntIme System (IRIS) [1], [2], this work proposes the following advances: (1) superior performance by exploiting the high-speed networks that connect the myriad existing devices (e.g., hardware accelerators), (2) a reduction in data transfers by developing more efficient memory management policies, and (3) minimization of the many programmability constraints for implementing scientific and high-performance computing (HPC) codes on extremely heterogeneous architectures. These improvements provide a highly productive programming environment enabled by IRIS's high-level and intelligent heterogeneous memory model: IRIS-DMEM.

Although deploying multiple GPUs is becoming the norm for many HPC hardware configurations, an important gap exists in current programming solutions in the form of efficient, productive, and transparent support for multi-GPU implementations. We can divide high-level programming models into three categories: (1) those based on pragmas, such as OpenMP and OpenACC; (2) those based on C++ abstraction libraries, such as Kokkos [3] and RAJA [4]; and (3) those based on task-based programming models with runtimes, such as StarPU [5] and IRIS [1].

OpenMP recently incorporated a new list of pragmas for GPUs into its specification [6]–[8]. These new pragmas allow for offloading the execution onto one GPU by using the OpenMP `target` clause. However, no OpenMP construct currently targets multiple GPUs. Historically, using multiple GPUs with OpenACC [9] has required MPI [10]–[12]. Matsumura et al. [13], [14] developed and proposed an extension to the OpenACC specification to handle multi-GPU systems, and they argued that their extension was competitive with some MPI + OpenACC codes on NVIDIA multi-GPU systems. Unfortunately, this extension was not included in the OpenACC standard. Like OpenACC, Kokkos requires MPI to exploit the use of multiple GPUs [15]. Recently, Valero et al. [16] proposed an extension of the Kokkos front end to support multi-GPU programming and reported better performance and scalability than achievable by using MPI + Kokkos. IRIS and StarPU are task-based programming models with runtimes for heterogeneous targets. The IRIS runtime is more modular in code structure and is written in C++ object-oriented programming, whereas StarPU is a C library. Compared with StarPU, IRIS supports more heterogeneous programming languages, including OpenMP, CUDA, HIP, OpenCL, Xilinx CL (for Xilinx FPGAs), Intel CL (for Intel FPGAs), and Hexagon DSP, whereas StarPU supports mainly OpenMP, OpenCL, and CUDA and has limited support for AMD HIP.

The rest of the paper is organized as follows: Section II describes the different components used in this work, such as IRIS, IRIS-BLAS, LaRIS-LU factorization, and previous IRIS memory handlers. The novel IRIS-DMEM heterogeneous memory handler is proposed in Section III. The performance evaluation of the IRIS-DMEM memory handler and state-of-the-art approaches are presented in Section IV. Finally, we conclude the paper and describe future work in Section V.

II. Background

A. IRIS

IRIS [1] is a task-based programming model for extremely heterogeneous architectures. It enables application developers to write portable applications across diverse heterogeneous programming platforms, including CUDA, HIP, Level Zero, OpenCL, and OpenMP. IRIS orchestrates multiple programming platforms and consolidates them into a single execu-

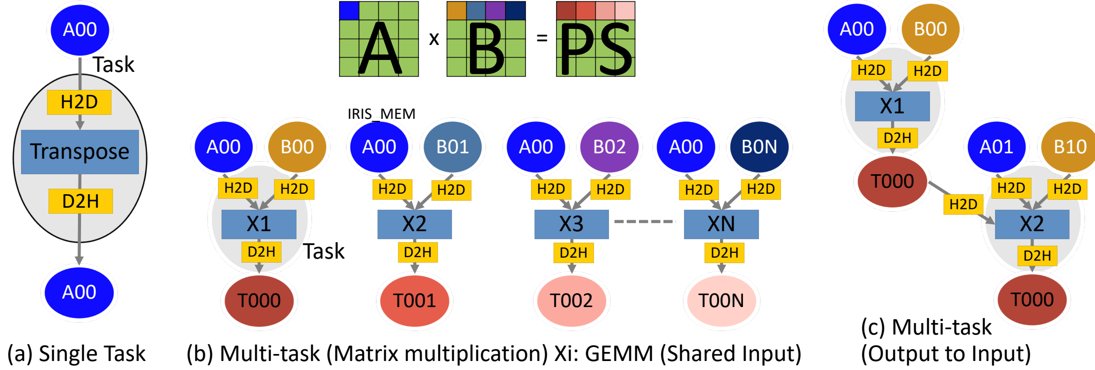

Fig. 1: IRIS task graph examples. A and B are input tiled matrices, and PS is a partial sum tiled matrix.

tion/programming environment by providing portable tasks and shared virtual device memory.

B. IRIS-BLAS and LaRIS

IRIS-BLAS [17] is a novel heterogeneous and performance-portable basic linear algebra subprograms (BLAS) library that addresses the portability challenge of BLAS library use for different heterogeneous architectures. IRIS-BLAS is built on top of the IRIS runtime and offers multiple vendor and open-source BLAS libraries. LaRIS [18] is a performance-portable LAPACK library built on top of IRIS-BLAS and the IRIS runtime. These three components (IRIS, IRIS-BLAS, and LaRIS) support abstracting OpenBLAS, Intel MKL, NVIDIA cuBLAS, and AMD hipBLAS kernels and can transparently use all the devices available in a heterogeneous system.

C. Challenges of IRIS-MEM Manual Memory Handling

IRIS provides the data structure *iris_mem* (IRIS-MEM) for handling device memories. It also provides H2D (Host to Device) and D2H (Device to Host) data transfer APIs with host and IRIS memory objects. The IRIS-MEM object is not associated with the host memory object during creation. Although the H2D and D2H commands do not specify the target, they take the host memory object and its size as arguments. When the task is mapped to the device by the IRIS runtime, it executes the H2D command, which transfers the data from the host to the target device, executes the kernel (matrix transpose), and calls the D2H command. The D2H command then transfers the data from target device to host memory.

Data transfers can be avoided in the following scenarios:

1) *Shared Input:* If two tasks depend on the same data, then transfer of the input data can be avoided given that the two tasks are mapped to the same device and input data is already available in-device. Even if the data is not available on the device, then we can still avoid one data transfer. For example, using IRIS-MEM allows for input data *A00* to be shared across tasks X1, X2, ..., XN, as shown in Figure 1b. However, when conducting manual memory handling, the programmer must add H2D and D2H commands in all tasks because the location of the tasks' device mapping is unknown during the development phase and is a pure runtime decision. Currently, no programming model can efficiently represent heterogeneous memory handling in the state of the art.
2) *Dependent Data:* If a task's output data is the input data of another task, then the data transfer can be avoided if both tasks are mapped to the same device. An example using IRIS-MEM for dependent data *T000* between tasks *X1* and *X2* is shown in Figure 1c. The data transfers of *T000* (both D2H and H2D) can be avoided if both tasks run on the same device, but this is known only at runtime.

In the above scenarios, the manual mode of handling data transfers (called IRIS-MANUAL) incurs unnecessary $N-1$ data transfers for *A00* (Figure 1b) and two unnecessary data transfers for *T000* (Figure 1c), irrespective of where the tasks are mapped, whether on the same device or on different devices. In this way, current heterogeneous programming models (e.g., IRIS, StarPU [5]) lack the ability to represent the heterogeneous memory objects during programming but provide complete freedom to decide or introduce the H2D and D2H commands at runtime. Therefore, the programmer should not introduce H2D and D2H commands in task specification; rather, this must be decided by the runtime system (IRIS).

Currently, IRIS addresses the problem of unnecessary memory transfers with an automatic movement of data with a memory management model. The IRIS-MEM object maintains a device memory for each device and employs a centralized heterogeneous memory coherency management scheme that utilizes a shared locking mechanism to synchronize the heterogeneous devices' memory objects. The IRIS-MEM object maintains the *Owners* set. When the tasks are mapped to the device, the device object of the IRIS runtime checks whether the device owns the IRIS-MEM object using the *Owners* set. To maintain the consistency, the sets are locked (i.e., mutex) before updating the ownership information in the sets. This is one of the drawbacks of this approach—the principal among them is that only one device can access the *Owners* set due to locks, and this limitation can create additional overhead. This approach creates new tasks with data transfer commands and submits them to the runtime scheduler, thereby adding an additional performance overhead.

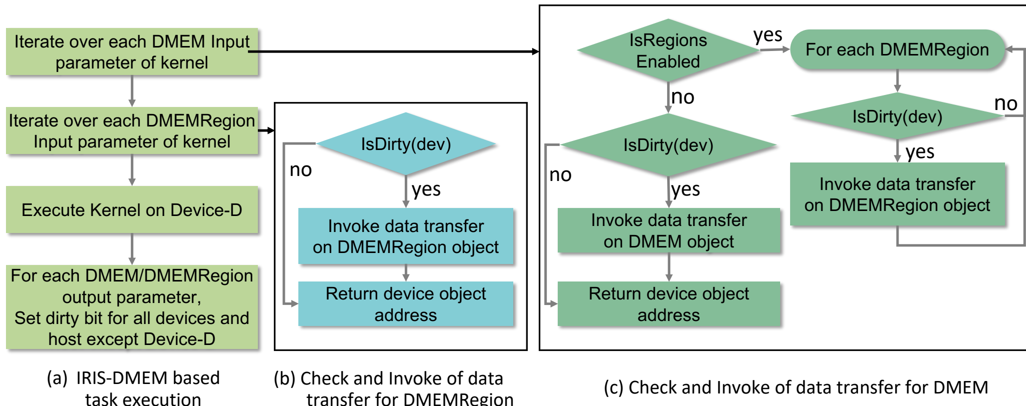

Fig. 2: Check DMEM/DMEM-Region data transfer requirement.

III. Proposed Heterogeneous Memory Handler

Heterogeneous memory handling plays an important role in application performance on heterogeneous computing resources. If the data movement between tasks and the reuse of data objects are not carefully orchestrated, then the data transfer costs can dominate the kernel's performance on the device.

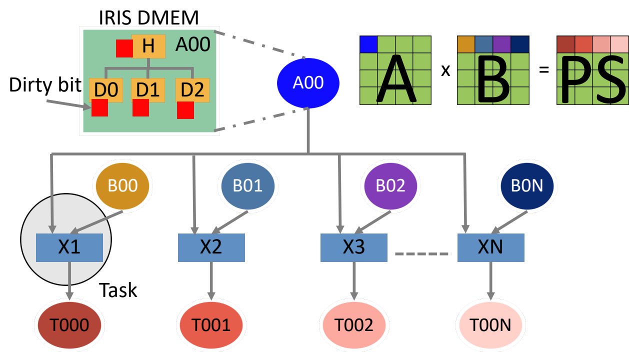

Fig. 3: IRIS-DMEM data handler and use.

We propose IRIS-DMEM as a means of distributed control over heterogeneous memory coherency similar to the multicore cache coherency protocol. Unlike hardware caches, heterogeneous memory objects are controlled by the IRIS runtime. IRIS-DMEM is tightly associated with the host memory data object during its initialization, unlike IRIS-MEM. However, it maintains a corresponding device memory for each host memory object, the same as with IRIS-MEM. Moreover, DMEM maintains a *dirty* flag for each host and device memory object (Figure 3) to determine whether the host or device memory object possesses valid and recent data. The DMEM controller logic sets the dirty flags based on the data transfers and task execution. This follows the assumption that only one device executes the task and gains the control of the DMEM object to write at any point in time. In the example shown in Figure 3, *A00* is an IRIS-DMEM memory object and is shared to all tasks (i.e., X1, X2, ..., XN kernels). Programmers need not write H2D and D2H commands for the DMEM objects; the DMEM controller in IRIS can call the H2D and D2H data transfers based on the workflow requirements at run time.

DMEM also provides an extension to split the larger chunk of memory into continuous address regions for parallel execution of kernels on these independent regions of heterogeneous compute units and applies a reduction operation at the end. A use case for DMEM regions is the tiled matrix multiplication algorithm, in which all row tiles in a multiplicand matrix (*A*) can be multiplied with all column tiles of a multiplier matrix (*B*) in parallel, and the partial sums outcome of these multiplications must be combined with a sum reduction operation that adds all partial sum matrices and produces a tile of a *C* matrix. Figure 4a illustrates the tile matrix multiplication. Figure 4b shows the traditional approach, in which the programmer writes a column sum kernel with the number of inputs equal to the number of rows of *A*. However, it is not generic to write the column sum kernels for all possible sets of inputs because the number of rows is a runtime parameter. The approach in Figure 4c can use the DMEM for *T000*, *T001*, ..., *T00N* partial sum objects with a single input column sum kernel (*Colsum*), which expects a continuous address for all partial sum tiles. However, the programmer must write D2H commands for all partial sum DMEM objects to evict the data to host memory and use an H2D command to feed the data to the device. The column sum kernel is then executed on the device. This approach requires unnecessary data transfers of D2H and H2D, irrespective of where the kernels are mapped for execution.

In the proposed DMEM-Regions (*R0, R1, ..., R3*) approach shown in Figure 5, DMEM provides APIs to split its device and host memory objects into regions, and each region (DMEM-Region) object is given as an output parameter for tile multiplication (*X1, X2, ..., X4*) kernels. The regions are created as a cross-cut across all device objects that correspond to memory chunks. It also maintains a dirty flag for all device chunks in the DMEM-Region object to maintain the consistency of valid data. Once the data is produced in its corresponding chunks of devices, in which the tile multiplication kernels are executed, the entire DMEM object is given as a single input to the column sum kernel, which can then determine which regions are dirty and which are not. The regions with a dirty flag will pull the data from other devices by using either D2D data transfer or D2H followed by H2D data transfer. For the example, as shown for the device mapping (*D0, D1, D2*) in Figure 5, this approach requires only two D2D data transfers and can boost the performance and increase the parallel computing capability.

Figure 2 shows the DMEM/DMEM-Region controller al-

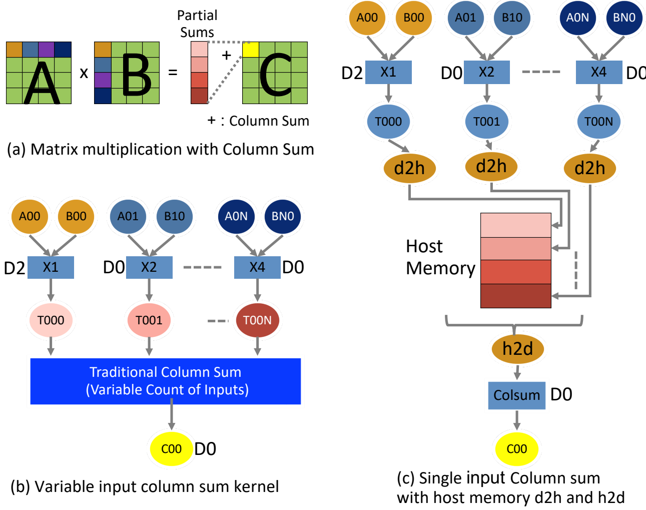

Fig. 4: Traditional tiled matrix multiplication column sum reduction without regions.

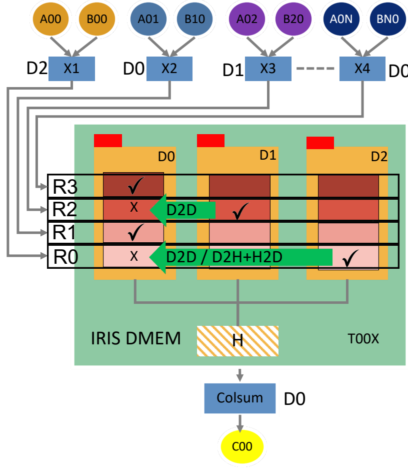

Fig. 5: IRIS-DMEM with regions (R0, R1, ..., R3) based tiled matrix multiplication column sum reduction.

gorithm with task execution and data transfers. IRIS invokes DMEM/DMEM-Region data transfers based on the extendable heterogeneous data transfer policies shown in Figure 6. Presently, five policies distinguish the data transfers into five categories. Ultimately, the device running the task fetches the data from either the source-adjacent device or from host memory. By following the correct policies, DMEM ensures optimal data transfer between two compute resources. Notably, DMEM ensures optimal data transfer for a given scheduling decision. Although ensuring optimal scheduling is an interesting research problem, it is beyond the scope of this work. The policies in order of priority to provide optimal data movement are given below:

1) *D2D for homogeneous devices*: For example, NVIDIA GPUs with NVLink connections provide better and optimal transfer rates when compared with PCIe data transfers. IRIS-MEM lacks support for D2D data transfers.
2) *Fetch from CPU device for the OpenMP-accelerated CPU device memory to the current task device memory*: IRIS considers OpenMP multicore CPUs as devices with their own memory. Thus, if the DMEM finds valid data in its CPU device memory, then it is the next-best performance efficient data transfer method for bringing the data to the current device. IRIS-MEM introduces D2H and then H2D data transfers to fetch the data from the CPU device for this scenario.
3) *Fetch from adjacent device to current OpenMP device*: If the current device is a CPU OpenMP device, then it is best to fetch data from the adjacent device with valid data by using D2H data transfer within the context of the adjacent device.
4) *Fetch from host*: If DMEM has valid data in host memory, then it is the next-best data transfer method for bringing the data to the current device.
5) *Heterogeneous device data transfer (D2H_H2D)*: If none of the above scenarios is satisfied, then it means the data is available in the adjacent heterogeneous device (e.g., AMD device to NVIDIA device). For this scenario, a D2H data transfer is first initiated to bring the data from the source device to host memory. Then, the H2D data transfer is performed to bring the data from the host memory to the current device. If a future interface (e.g.,

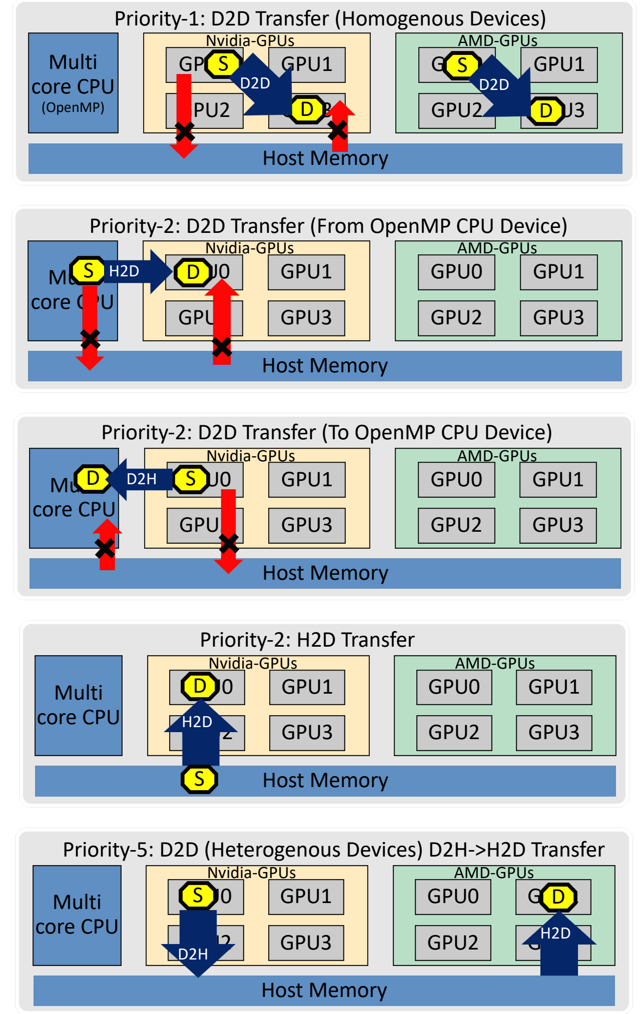

Fig. 6: IRIS-DMEM data transfer policy and priorities.

CXL) enables a direct data transfer among heterogeneous devices, then we can extend the DMEM data transfer policy with a new category with priority for the CXL data transfer.

We set the same priority for any CPU/host memory–related data transfers because the data transfer bandwidth would be the same for these three categories, and their order can be changed.

The DMEM memory handler works as a write-back cache because it does not transfer the data to the host memory location by default. Therefore, the programmer must explicitly write the new IRIS command *DMEM_ FLUSH_OUT_CMD* (i.e., the DMEM flush out command) after execution of all tasks/task graph to ensure that the output is provided to the host memory object. The flush out command execution will check whether the host's dirty flag is true or false. If host dirty flag is true, then DMEM will source valid data from the device where the dirty flag is false, by using the D2H data transfer APIs.

IV. Performance Evaluation

A. Experimental Setup

For this work, we used a heterogeneous HPE-Cray system with four NVIDIA A100 GPUs and four AMD MI100 GPUs. Each set of GPUs is connected by NVLink (for NVIDIA GPUs) or Infinity Fabric (for AMD GPUs) high-bandwidth connections. All GPUs are connected to the CPU via PCIe. Although this configuration is not very common by today's standards, it is a representative configuration for future and upcoming extremely heterogeneous systems, in which different architectures, networks, and memories coexist in the same node.

We considered two benchmark applications: tiled DGEMM from IRIS-BLAS [17] and LU factorization from LaRIS [18]. The effectiveness of the proposed IRIS-DMEM heterogeneous memory handler is shown in terms of performance gain/speedup when compared with the traditional IRIS-MANUAL memory handler and non-optimal IRIS-MEM centralized memory handler. We also compared the performance results of the IRIS-DMEM memory handler with the StarPU [5] framework and with MAGMA [19]. Moreover, we considered different variants of IRIS-DMEM with D2D transfers and without D2D data transfers (IRIS-DMEM-NoD2D). IRIS-MEM and IRIS-MANUAL require a flattening step to flatten the 2D tile into a continuous-memory 1D array. IRIS-DMEM has the ability and support to transfer the 2D tiles by using vendor-specific 2D memory copy APIs.

B. Tiled DGEMM Benchmark

The results shown in Figure 7 from the tiled DGEMM benchmark demonstrate the effectiveness of DMEM-Region. The size of the input (square) matrix is varied exponentially from 1,024 to 16,384. The tiled algorithm uses massively parallel tiled matrix multiplication with a column sum reduction operation. The tiled algorithm is configured to generate the task graph for the input matrices split into 4×4 tiles. This algorithm requires an intermediate buffer size of N^3, where N is the size of the square matrix. Thus, we limited our experiments to a maximum matrix size of 16,384. The performance results for the tiled DGEMM benchmark are similar to the LU factorization benchmark results. The tiled DGEMM algorithm has three inputs, and the reuse of input tiles depends on the mapping of tiled multiplication to compute units. The reuse factor is higher in tiled multiplication versus in LU factorization. Therefore, direct 2D data transfers with a greater reuse factor can provide better performance in this benchmark. The performance gain of IRIS-DMEM over IRIS-MEM is $4.1\times$ on average.

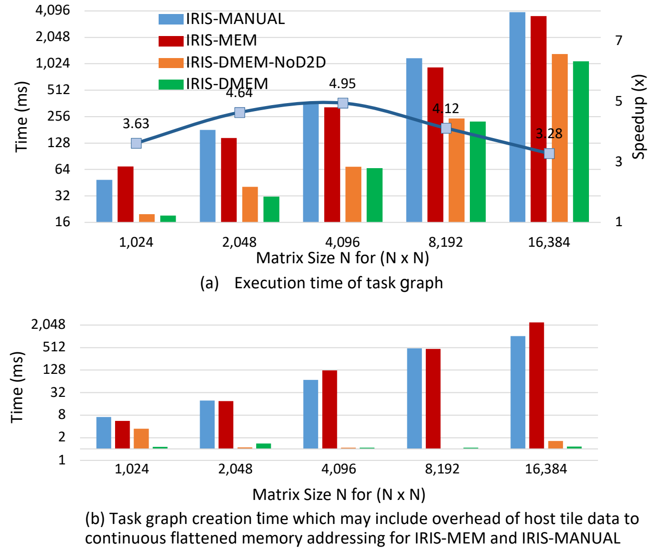

Fig. 7: Matrix multiplication results for four NVIDIA A100 CUDA GPUs and four AMD MI100 GPUs. Y-axis is on log scale.

The effectiveness of IRIS-DMEM is also evident in Figure 8, which shows a 26% reduction in data transfers and a 36% reduction in the size of data transfers versus IRIS-MEM and IRIS-MANUAL. Both IRIS-MEM and IRIS-MANUAL have a similar number of data transfers owing to the requirement of data transfers for the inputs of column sum operations and the outputs of tiled DGEMM operations. IRIS-DMEM optimizes these data transfers with the help of DMEM-Regions.

C. LU Factorization Benchmark

The comparison of IRIS-DMEM with state-of-the-art IRIS memory handlers for the tiled LU factorization benchmark is shown in Figure 9. The LU factorization matrix size ($N \times N$) is varied from 1,024 to 32,768 on the x-axis, and the execution time of the LU factorization is measured in milliseconds on the y-axis. The tiled algorithm is configured to generate the task graph for the input matrix split into 16×16 tiles.

According to the results, the speedup of IRIS-DMEM is $7.2\times$ over IRIS-MEM on average. For LU factorization, only one input matrix exists, and multiple accesses of each input tile could occur. IRIS-DMEM can intelligently explore the data locality of input tiles with different data transfer policies and priorities to achieve a speedup of $7.2\times$. Interestingly, IRIS-MANUAL outperforms IRIS-MEM. This could be due to overhead in IRIS-MEM, which uses a centralized memory coherency protocol for synchronizing the data transfers. The effectiveness of the D2D data transfers is demonstrated by com-

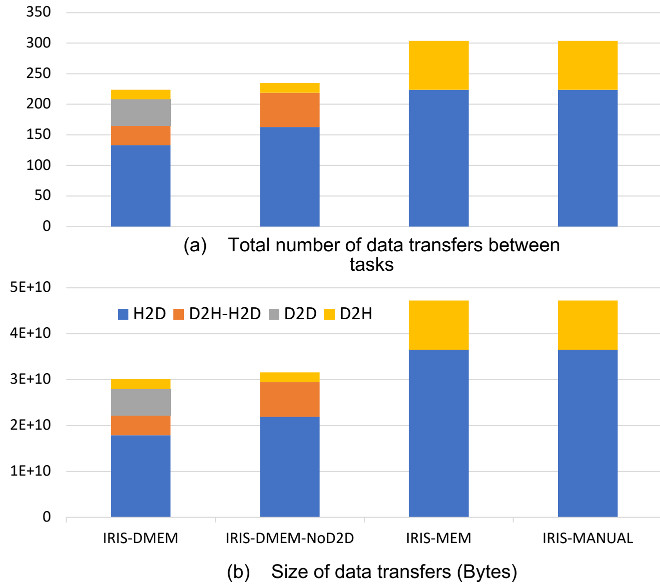

Fig. 8: Comparison of data transfers for the matrix multiplication (reduction) benchmark.

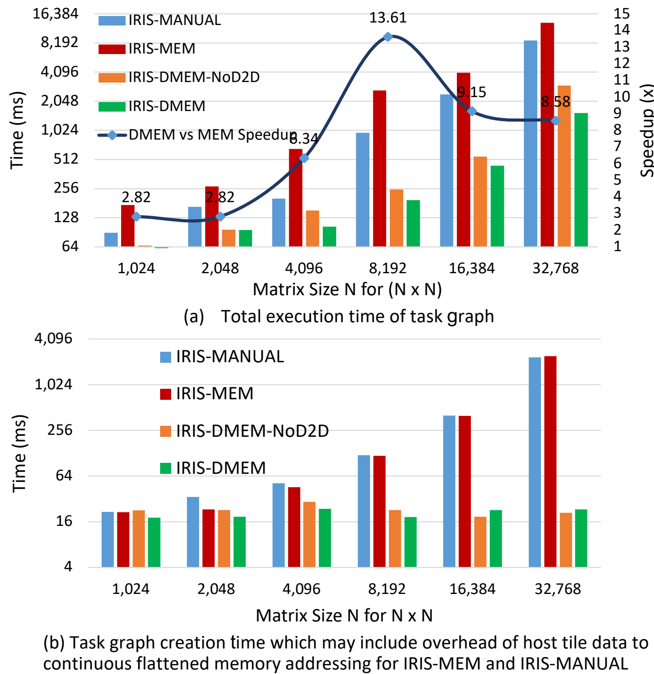

Fig. 9: LU factorization results for four NVIDIA A100 CUDA GPUs and four AMD MI100 GPUs. Y-axis is on log scale.

paring the results of IRIS-DMEM-NoD2D and IRIS-DMEM. It proves the need for D2D data transfers in the IRIS runtime.

The effectiveness of data transfer policies for LU factorization is shown in Figure 10. The exploration of data transfers with priorities in IRIS-DMEM results in fewer data transfers versus IRIS-MANUAL (Figure 10b). IRIS-DMEM results in 20% fewer data transfers versus IRIS-MEM and and 71% fewer data transfers versus IRIS-MANUAL. Moreover, IRIS-DMEM also reduced the total overall data transfer size by 42% versus IRIS-MEM and 60% versus IRIS-MANUAL (Figure 10a). IRIS-DMEM is more efficient in terms of the reduction in data transfers and size of data transfers when compared to IRIS-MEM and IRIS-MANUAL. IRIS-DMEM-NoD2D has a similar number of data transfers and data transfer sizes as IRIS-DMEM, but it has more H2D data transfers, which impacts the task graph execution time.

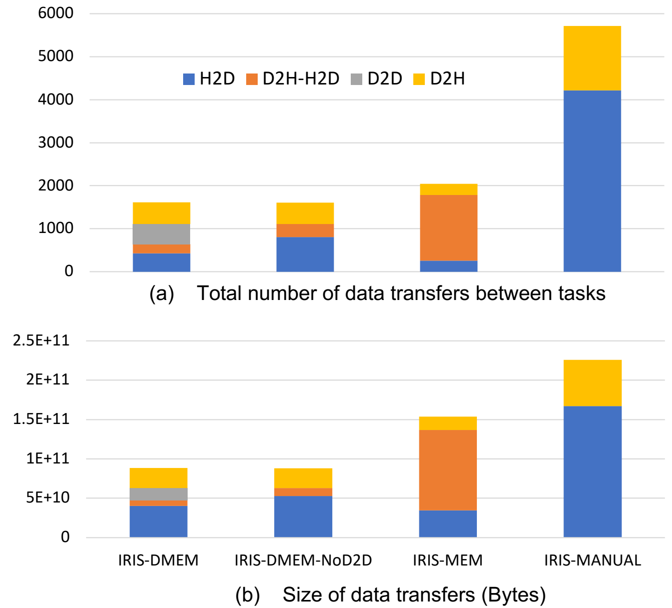

Fig. 10: Comparison of data transfers for the LU Factorization benchmark for all variants of IRIS-DMEM, IRIS-MEM, and IRIS-MANUAL.

D. Comparison with StarPU and MAGMA

In terms of alternatives to IRIS (LaRIS), StarPU for heterogeneous computing with a task-based programming model and the MAGMA linear algebra library offer interesting opportunities for comparison. MAGMA [19], [20] offers math libraries for CPUs, NVIDIA GPUs (CUDA), and AMD GPUs (HIP), but it is essentially a linear algebra library rather than a true task-based programming runtime system. StarPU [5], similar to the IRIS runtime, supports heterogeneous computing and is an ideal candidate to compare against. The StarPU runtime supports asynchronous execution of tasks, whereas IRIS does not yet offer this support. Moreover, IRIS lacks support for a static task scheduling policy like StarPU has with DMDAS.

A comparison of IRIS-DMEM and StarPU is shown in Table I. We also compared IRIS-DMEM results with those of MAGMA. Although the recent versions of StarPU and MAGMA support AMD GPUs, neither supports using both NVIDIA GPUs and AMD GPUs concurrently. Thus, for a fair comparison, we used the results for the multi-GPU NVIDIA/CUDA platform, which has four NVIDIA A100 GPUs, and the tiled DGEMM benchmark. The performance reported in the table is in GFLOP/s. Using random and HPL (High Performance Linpack) policies, IRIS-DMEM performs better versus StarPU's dynamic scheduling and static scheduling policies. The HPL policy [21] used by IRIS-DMEM distributes the computational cost of linear algebra operations.We fine-tuned this algorithm for IRIS-DMEM to distribute the tasks across

multiple NVIDIA/CUDA GPUs. Because of task creation overhead, the MAGMA framework is better than IRIS-DMEM only for smaller matrix sizes, such as 1,024 and 2,048.

TABLE I: Comparison of DMEM with StarPU and MAGMA. Performance in GFLOP/s. Benchmark: Tiled matrix multiplication. Platform: Four NVIDIA A100 GPUs.

Size (N×N)	StarPU Random	StarPU DMDAS	DMEM Random	DMEM HPL	MAGMA
1,024	195	228	142	463	**2,683**
2,048	585	973	582	1,856	**3,845**
4,096	2,148	3,265	2,648	**6,396**	4,319
8,192	5,196	7,048	5,385	**10,458**	5,152
16,384	12,718	16,333	13,929	**18,120**	6,509

V. Conclusion and Future Work

This paper presents the IRIS-DMEM novel heterogeneous memory handler for the IRIS runtime and introduces novel data transfer policies for efficient and optimal data movement between compute units. The proposed approach was evaluated with tiled LU factorization and tiled DGEMM benchmarks. We achieved up to 7× performance gains and a 75% reduction in data transfer sizes. This work also compares IRIS with the proposed heterogeneous memory handler with the state-of-the-art StarPU runtime and MAGMA frameworks. Results have shown up to 1.95× performance uplift over StarPU and 2.1× uplift over MAGMA. Future work includes exploration of IRIS-DMEM for different static and dynamic task scheduling policies for heterogeneous computing.

Acknowledgment

Notice: This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for U.S. Government purposes. The DOE will provide public access to these results in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

References

[1] J. Kim, S. Lee, B. Johnston, and J. S. Vetter, "IRIS: A Portable Runtime System Exploiting Multiple Heterogeneous Programming Systems," in *Proceedings of the 25th IEEE High Performance Extreme Computing Conference*, ser. HPEC '21, 2021, pp. 1–8.

[2] A. Cabrera, S. Hitefield, J. Kim, S. Lee, N. R. Miniskar, and J. S. Vetter, "Toward performance portable programming for heterogeneous systems on a chip: A case study with qualcomm snapdragon soc," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021.

[3] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[4] D. Beckingsale, R. D. Hornung, T. Scogland, and A. Vargas, "Performance portable C++ programming with RAJA," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 455–456.

[5] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. Pract. Exp.*, vol. 23, no. 2, pp. 187–198, 2011. [Online]. Available: https://doi.org/10.1002/cpe.1631

[6] R. D. Budiardja and C. Y. Cardall, "Targeting gpus with openmp directives on summit: A simple and effective fortran experience," *Parallel Comput.*, vol. 88, 2019.

[7] T. Cramer, M. Römmer, B. Kosmynin, E. Focht, and M. S. Müller, "Openmp target device offloading for the sx-aurora TSUBASA vector engine," in *Parallel Processing and Applied Mathematics - 13th International Conference, PPAM 2019, Bialystok, Poland, September 8-11, 2019, Revised Selected Papers, Part I*, ser. Lecture Notes in Computer Science, vol. 12043. Springer, 2019, pp. 237–249.

[8] P. Valero-Lara, J. Kim, O. Hernandez, and J. S. Vetter, "Openmp target task: Tasking and target offloading on heterogeneous systems," in *Euro-Par 2021: Parallel Processing Workshops - Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021*, ser. Lecture Notes in Computer Science, vol. 13098. Springer, 2021, pp. 445–455.

[9] L. Toledo, P. Valero-Lara, J. S. Vetter, and A. J. Peña, "Static graphs for coding productivity in openacc," in *28th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2021, Bengaluru, India, December 17-20, 2021*. IEEE, 2021, pp. 364–369. [Online]. Available: https://doi.org/10.1109/HiPC53243.2021.00050

[10] M. Wolfe, "Scaling openacc applications accross multiple gpus," 2014, GPU Technology Conference (GTC). [Online]. Available: https://on-demand.gputechconf.com/gtc/2014/presentations/S4474-scaling-openacc-across-multiple-gpus.pdf

[11] J. Larkin, "Multi-GPU Programming with OpenACC," 2017, GPU Technology Conference (GTC). [Online]. Available: https://on-demand.gputechconf.com/gtc/2017/presentation/S7546-jeff-larkin-multi-gpu-programming-with-openacc.pdf

[12] J. Kraus, "Multi gpu programming with mpi and openacc," 2015, GPU Technology Conference (GTC). [Online]. Available: https://on-demand.gputechconf.com/gtc/2015/presentation/S5711-Jiri-Kraus.pdf

[13] K. Matsumura, M. Sato, T. Boku, A. Podobas, and S. Matsuoka, "MACC: an openacc transpiler for automatic multi-gpu use," in *Supercomputing Frontiers - 4th Asian Conference, SCFA 2018, Singapore, March 26-29, 2018, Proceedings*, ser. Lecture Notes in Computer Science, R. Yokota and W. Wu, Eds., vol. 10776. Springer, 2018, pp. 109–127.

[14] K. Matsumura, S. G. de Gonzalo, and A. J. Peña, "JACC: an openacc runtime framework with kernel-level and multi-gpu parallelization," in *28th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2021, Bengaluru, India, December 17-20, 2021*. IEEE, 2021, pp. 182–191.

[15] S. Khuvis, K. Tomko, J. M. Hashmi, and D. K. Panda, "Exploring hybrid mpi+kokkos tasks programming model," in *3rd IEEE/ACM Annual Parallel Applications Workshop: Alternatives To MPI+X, PAW-ATM@SC 2020, Atlanta, GA, USA, November 12, 2020*. IEEE, 2020, pp. 66–73. [Online]. Available: https://doi.org/10.1109/PAWATM51920.2020.00011

[16] P. Valero-Lara and J. S. Vetter, "A performance-portable and productivity solution based on kokkos for multigpu programming," in *International Workshop on Performance, Portability and Productivity in HPC, P3HPC@SC 2022, Dallas, TX, USA, November, 2022*. IEEE, 2021.

[17] N. R. Miniskar, A. H. M. Mohammad, V.-L. Pedro, F. Liu, and J. S. Vetter, "Iris-blas: Towards a performance portable and heterogeneous blas library," in *29th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2022, Bengaluru, India, December 18-21, 2022*. IEEE, 2022, pp. 1–10.

[18] M. A. H. Monil, N. R. Miniskar, F. Liu, J. S. Vetter, and P. Valero-Lara, "LaRIS: Targeting Portability and Productivity for LaPACK Codes on Extreme Heterogeneous Systems using IRIS," in *IEEE/ACM Redefining Scalability for Diversely Heterogeneous Architectures Workshop, RSDHA@SC 2022, Dallas, TX, USA, November 13–18, 2022*. IEEE.

[19] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, Jun. 2010.

[20] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra, "Batched matrix computations on hardware accelerators based on gpus," *The International Journal of High Performance Computing Applications*, vol. 29, no. 2, pp. 193–208, 2015. [Online]. Available: https://doi.org/10.1177/1094342014567546

[21] J. J. Dongarra and P. Luszczek, "Scalapack," in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed. Springer, 2011, pp. 1773–1775. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_151