IRIS-MEMFLOW: Data Flow–enabled Portable Memory Orchestration in IRIS Runtime for Diverse Heterogeneity

Mohammad Alaul Haque Monil, Narasinga Rao Miniskar, Seyong Lee,
Beau Johnston, Pedro Valero-Lara, Aaron Young, Keita Teranishi and Jeffrey S. Vetter
Oak Ridge National Laboratory, Oak Ridge, TN, USA
{monilm, miniskarnr, lees2, johnstonbe, valerolarap, youngar, teranishik, vetter}@ornl.gov

***Abstract*—Task-based programming models and execution paradigms provide a means to decompose a computation by expressing it as a graph in which each node represents a specific computation operating on memory objects and the edges define the dependencies in the execution flow. In this execution model, independent nodes in the graph can be executed concurrently in different computing devices, making it suitable for heterogeneous systems in which computing devices with different architectures coexist. However, careful memory orchestration across heterogeneous devices is needed because copies of the same memory object may reside in multiple devices during execution. Manually ensuring such an orchestration is quite challenging. Not only must an application developer guard against race conditions, but they must also optimize data movement between the host and devices because unnecessary data movement significantly impacts performance. To mitigate these challenges, we enhance the IRIS heterogeneous runtime and introduce IRIS-MEMFLOW—a data flow–enabled portable memory abstraction for seamlessly orchestrating memory in diverse heterogeneous computing environments. By using data-flow analysis, IRIS-MEMFLOW guards against race conditions while multiple heterogeneous devices access memory objects. IRIS-MEMFLOW also optimizes data movement between the host and devices without manual intervention. As a result, IRIS provides improved programming productivity, performance, and portability for multidevice heterogeneous executions in high-performance computing and cloud systems that run diverse architectures from different vendors. The efficacy of IRIS-MEMFLOW is evaluated through experiments that show its capability in terms of programming productivity, multidevice heterogeneity, portability, and low overhead versus the state of the art.**

I. Introduction

Task-based programming models and runtimes expose task-level abstraction and are an active area of research owing to their ability to intelligently decompose the total computation into a graph of tasks. Through decomposition and scheduling, contemporary task-based runtimes (e.g., HPX [13], Charm++ [14]) can utilize a system in an efficient way as opposed to the de-facto bulk synchronous programming model (i.e., MPI). In addition to distributed high-performance computing environments, task-based abstractions are widely adopted for in-node heterogeneity (e.g., StarPU [1], OmpSs [9], OpenMP [26]). However, contemporary programming models and runtimes for heterogeneity face various challenges in programming productivity and performance optimization in terms of orchestrating the execution—especially memory orchestration in diverse heterogeneous architectures.

The in-node heterogeneity found in many modern computing systems exacerbates the portability, scalability, and utilization challenges, and the task-based computing paradigm is no different. Although the coexistence of different architectures from vendors/manufacturers opens the door for simultaneous task execution, orchestrating the total execution flow becomes challenging. The goal is to create a high-level abstraction that does not require architecture information while the runtime systems provide the necessary orchestration. One of the major challenges in such an orchestration is transparent memory management. Ideally, at a high level, an application developer would not need to worry about how the data moves across devices. Traditionally, data movement is specified using a directed acyclic graph (DAG) of tasks in which dependencies protect the execution against any race condition among tasks that are accessing the same memory.

Data flow–based task graph construction has been studied in the scope of programming models and runtimes (e.g., Nanos [2], SYCL [15], OpenMP [26], StarPU [1], Kaapi [11], TaskFlow [12]). However, the coexistence of heterogeneous processors complicates the scenario because multiple copies of the same data may exist in different devices managed by the runtime for a specific device [22]. Studies have shown some successes in supporting heterogeneity to some extent (XKaapi [11], StarPU [1]). However, data-flow analysis can facilitate guarding against race conditions and improve performance by pruning unnecessary data movement and providing computation-communication overlap. To address these challenges and leverage these opportunities, the research described here strives to answer the following question: *How can a data-flow enabled, high-level memory abstraction provide intelligent and efficient memory orchestration capability for diverse heterogeneity while providing an architecture-agnostic front end to ensure portability?*

To answer this question, we present IRIS-MEMFLOW, which enhances the memory abstraction of IRIS, a task-based runtime for diverse heterogeneity. IRIS provides architecture-agnostic APIs for task and memory object creations, and dependencies are introduced to create a DAG of tasks to

Notice: This manuscript has been authored by UT-Battelle LLC under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (https://www.energy.gov/doe-public-access-plan).

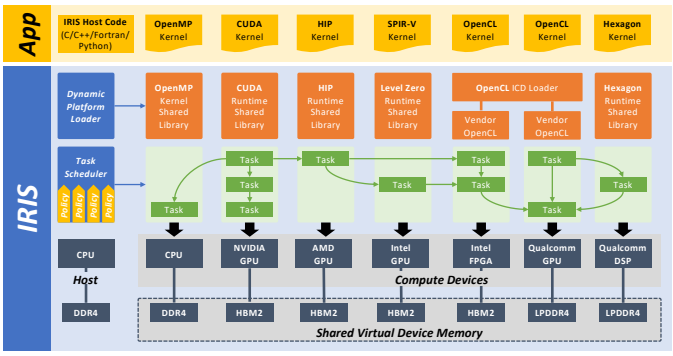


Fig. 1: IRIS runtime system for heterogeneous architectures [16].

intelligently schedule and orchestrate the tasks in diverse heterogeneous systems [16], [21], [24], [25]. The memory abstraction of the IRIS runtime (data memory [DMEM]) can orchestrate the memory copies that exist during execution [20]. IRIS-MEMFLOW further introduces data flow–based memory orchestrations to improve the existing memory abstractions (DMEM) and deliver the solution to the research question. Although data flow–based dependency analysis exists in other runtimes, IRIS-MEMFLOW provides not only a high-level portable abstraction for memory objects in the context of heterogeneity but also combines low-level memory orchestration in which multiple devices concurrently execute different tasks. IRIS-MEMFLOW enriches the IRIS runtime with two additional capabilities. (1) High-level portable abstraction: an architecture-agnostic memory object in IRIS uses data flow–enabled DAGs to automatically ensure race condition–free execution from a serial IRIS code and provides opportunities (e.g., DAG fusion) for increasing concurrency without changing the front-end code. (2) Low-level orchestration: data-flow enabled memory orchestration between host and devices ensures intelligent memory movement and compute-communication overlap while supporting automated heterogeneous execution and portability provided by the high-level abstraction.

The present work describes contributions toward an automated, data flow–enabled portable memory abstraction that can identify concurrency and guard against race conditions, data races, and deadlocks initiated by concurrent tasks executing in diverse heterogeneity. Additionally, we describe a solution for automatic and optimized data movement between host and devices during execution to ensure computation and communication overlap, reduced data movement, and DAG fusion while keeping the higher-level abstraction the same. Finally, we evaluate the proposed methods for various linear algebra algorithms by providing the overhead of the data-flow analysis and the improvement versus the state of the art.

II. Background

A. IRIS

Developed at Oak Ridge National Laboratory, 2024 R&D 100 award winner IRIS runtime [16] is an intelligent, task-based runtime system (Fig. 1) designed for diverse heterogeneous systems. IRIS provides a unique programming model and runtime environment that can simultaneously utilize devices and accelerators from different vendors. In addition to supporting mainstream CPUs and GPUs from Intel, NVIDIA, and AMD, IRIS also supports field-programmable gate arrays (FPGAs) and digital signal processors (DSPs) from different vendors. IRIS deviates from the traditional CPU and accelerator model and considers all accelerators and CPUs as IRIS devices. For this reason, unlike other state-of-the-art runtime systems, IRIS can simultaneously orchestrate computation among CPUs, multiple GPUs from the same or different vendors, single or multiple FPGAs, and DSPs. IRIS invokes APIs from OpenMP, OpenACC, HIP, CUDA, XilinxCL, OpenCL, Hexagon C++, and IntelCL to perform the total orchestration. IRIS provides two architecture-agnostic abstractions: (1) tasks for computation and (2) memory objects for data. Through these abstractions, a computation DAG is expressed and can be scheduled and executed by using different schedulers in IRIS.

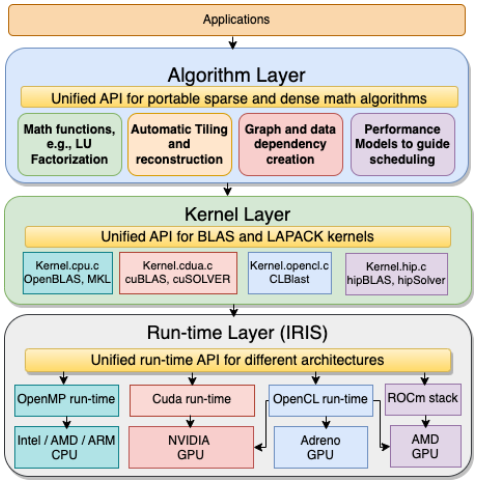


Fig. 2: MatRIS math library abstraction using IRIS.

B. MatRIS

The MatRIS (Fig. 2) multilevel math library abstraction [23], [25] uses the IRIS runtime and provides state-of-the-art tiled algorithms for different BLAS and LAPACK functionalities. Once compiled, MatRIS can utilize all the heterogeneous devices in a system by using the scheduling algorithm provided by the IRIS runtime. MatRIS supplies a fully portable and heterogeneous implementation of GEMM, GETRF (non-pivoting LU factorization), POTRF (Cholesky factorization), TRSM, POSV, GESV, and other tiled math algorithms. The present work shows the efficacy of using the tiled algorithms in MatRIS.

III. Motivation

As shown in Fig. 1, IRIS provides a shared virtual device memory space (includes CPU as well). Such abstraction is provided through the DMEM logical memory handler [20]. Each DMEM object in the IRIS runtime has copies in different

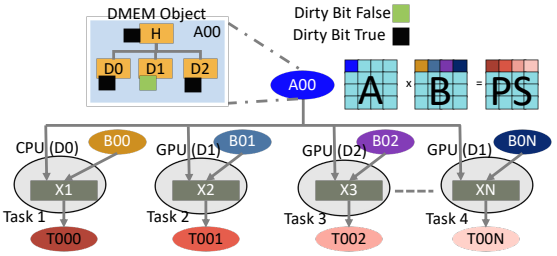


Fig. 3: IRIS managing multiple memory copies by using DMEM abstraction in a matrix multiplication [20].

devices (Fig. 3). Using dirty bits, IRIS's DMEM logic controller tracks which memory copies within a DMEM object have valid data. The diagram in Fig. 3 shows how IRIS keeps multiple copies for matrix multiplication for the data structure *A*. Because IRIS is informed about the status of each copy of the data, it enables automatic data transfer between devices when necessary. However, manual effort is required to ensure flush-out data movement and guard against race conditions.

Ideally, one programmer should be able to write a simple single-threaded, architecture-agnostic, and serial program using IRIS tasks and memory objects without specifying any data movement. The runtime should be able to perform the following ***four functionalities*** without manual intervention.

(1) Automatically find concurrency in the computation by constructing a task graph (DAG) to utilize the heterogeneous devices concurrently while guarding against race conditions, data races, and deadlocks.

(2) Ensure automatic data movement that includes copying input from host to device, data movement between devices, and sending the final results without introducing unnecessary/extra data movement.

(3) Ensure computation and communication overlap when possible in the existing synchronous and asynchronous execution in IRIS.

(4) Automatically fuse different DAGs of tasks when applicable to increase concurrency and efficiently utilize heterogeneous devices (e.g., multiple GPUs) while ensuring the three points mentioned above.

Currently, IRIS can accomplish all four functionalities through moderate-to-extensive manual efforts. However, some functionalities (two and three listed above) can be done automatically through the current memory abstractions, including data fetching and data movement between devices [20] and computation and communication overlap using streams. The present work extends current IRIS memory abstraction (DMEM) [20] to provide fully automated solutions to these problems by introducing data flow–based analysis.

IV. Data Flow–Enabled Memory Orchestration for Heterogeneity

This section describes the design of the data flow–enabled portable memory object orchestration found in the IRIS runtime. Data flow–based dependency creation has been investigated for both the compiler and the runtime for multicore

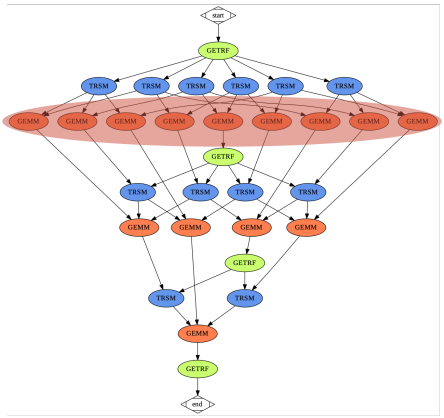


Fig. 4: DAG for dense LU factorization showing maximum concurrency using shaded area.

CPUs and for heterogeneity to some extent [1], [4]. However, our work uses the traditional concept of data flow and goes beyond previous efforts by establishing a connection between high-level memory abstraction and low-level memory orchestration for diverse heterogeneity and by implementing the four functionalities listed in Section III.

A. Concurrency and Accuracy

The traditional concept of data flow is applied to the sequential IRIS program (IRIS host code) to find concurrency by constructing a DAG of tasks. As shown in Fig. 3, IRIS memory objects are a bundle of their copies in the host and different heterogeneous devices (the high-level memory abstraction is shown in line 19 of Fig. A.1 in Appendix I). Data-flow analysis views these high-level abstract memory objects as single memory objects and considers the read-and-write properties of these memory objects in each task's definition (Lines 28–29 of Fig. A.1 in Appendix I). At a high level, dependency between tasks is created based on memory accesses by the kernels associated with the tasks. Because tasks are created in sequential code, the arrival of the task ensures proper ordering. If two tasks write to the same memory, then a dependency is created from the latter to the former. The same is maintained for reading and writing on the same memory object. Only the tasks reading the same memory object or with no common memory objects can proceed in parallel. So, starting from the first task creation, dependencies are created as the tasks are created and weave the full DAG and expose concurrency to the IRIS runtime (Fig. 4).

Initially, tasks are held in the IRIS scheduler's queue. Once a task no longer depends on any other tasks, it is moved to a worker queue attached to a device (e.g., CPU, GPU). During execution, at no point will two devices simultaneously try to update the device copy of the memory object. Because this serialization is ensured based on the arrival of the task from the sequential program, the sequential IRIS program and the IRIS DAG always provide the same result, thereby successfully guarding against race conditions.

In synchronous mode, any IRIS device works on only one task, and a memory object is updated by one device, thereby avoiding data races. However, IRIS's existing asynchronous execution capability, in which multiple streams can concurrently execute kernels and transfer data, can lead to data races between copies of data in a memory object. For example, for transferring data from one device to another through the host, one must issue two transfers: one from device-to-host and one from host-to-device. Both transfers can be scheduled concurrently by different streams in different devices. For such cases, the correct order is ensured by event-based synchronization between streams (e.g., CUDA and HIP events for streams). When a new task is created, then it creates a dependency only on the existing tasks, thereby eliminating the possibility of creating a cycle and ensuring a deadlock-free execution in IRIS.

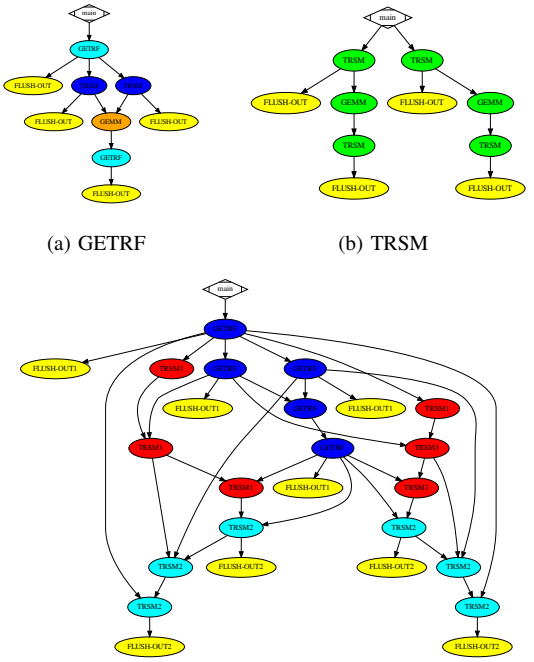
Applying traditional data-flow logic in the high-level memory object creates the DAG used to find concurrency and guards against race conditions, data races, and deadlock in the low-level orchestrations when multiple devices execute the same DAG (ensures the first functionality listed in Section III).

Before the introduction of IRIS-MEMFLOW, dependencies were manually created in IRIS, and this required significant effort and sometimes made it impossible to track all the dependencies. For example, in some cases, tasks and memory objects are created based on the property of the data at run time, making tracking a dependency quite challenging. IRIS-MEMFLOW eliminates these challenges to provide higher programming productivity.

B. Automatic Data Movement

IRIS provides a shared virtual device memory space. First, data must be moved from the host space to the device space for computation. After the computation finishes, data must then be brought back to the host space. This process contains three parts. (1) **Flush-in:** Data is brought to the device space for the first time, and that is when memory is allocated to a device. (2) **Between-devices:** When data is already brought to the device space, then IRIS fetches the updated device copy, either through device-to-device communication or device-to-host followed by a host-to-device communication. (3) **Flush-out:** Data is finally updated and sent back to the host space from the most up-to-date device.

During the execution of a task, IRIS knows what data is needed at that moment. The first two operations are handled on an on-demand basis. Therefore, the current memory abstraction in IRIS (DMEM [20]) performs the first two automatically because it keeps track of where the most up-to-date data lives. However, the third part depends on whether the executing task is the last to update that memory object, so manual effort is employed to ensure flush-out. However, if data is prematurely sent back to the host, then there is a risk of unnecessary data transfers. For this reason, IRIS-MEMFLOW ascertains when a memory object was last updated and inserts a flush-out command only at that point. The flush-out command uses the capability of the DMEM objects to find which device (or host) has the most up-to-date copy and issues a device-to-host transfer.



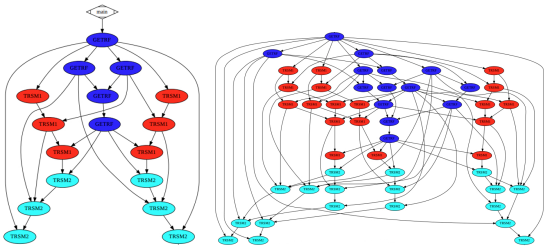
(a) GETRF (b) TRSM

(c) GESV

Fig. 5: Automated and non-blocking flush operation for LU factorization. GETRF, TRSM, and GESV with 2×2 decomposition. Yellow indicates a flush operation.

IRIS-MEMFLOW creates a floating task (flush-out task) that only contains the flush-out command when it discovers that a task's kernel will update a memory object. IRIS-MEMFLOW keeps moving the flush-out task whenever another task updates that memory object. In graph submission mode, a DAG is first created in IRIS before submission, and graphs can be retained to execute many times. For this reason, IRIS-MEMFLOW can ensure a maximum of one flush-out per memory object, eliminating the possibility of unnecessary flush-out data transfer.

The flush-in and between-devices data transfer must be completed before the kernel starts executing. However, the flush-out does not need to block the computation flow and is always kept as a leaf node in the DAG. The existing asynchronous execution in IRIS dedicates separate streams for computation and communication. Hence, the flush-out task in the leaf node increases the chances of computation and communication overlapping without blocking any computation. Automatic flush-out is shown in Figs. 5a and 5b, where a 2×2 decomposition is considered for LU factorization and TRSM. For GETRF, four memory objects are updated by five tasks. The GEMM and the last GETRF task update the same memory, and IRIS-MEMFLOW ensures there are four flush-out leaf tasks, avoiding unnecessary data movement. The same is demonstrated for TRSM. In this way, IRIS-MEMFLOW completes the automated data movement in IRIS



(a) A 2×2 decomposition for GESV.

(b) A 3×3 decomposition for GESV.

Fig. 6: Multiple DAG fusion. GESV is shown as an example. Blue tasks are from GETRF, red tasks are from the first TRSM, and cyan tasks are from the second TRSM.

while eliminating the possibility of unnecessary data transfer and increasing the possibility of computation and communication overlap (functionalities two and three mentioned in Section III).

C. DAG Fusion with Data Movement Optimization

Fusing multiple DAGs can increase concurrency, which can in turn increase performance by ensuring higher utilization of the computing devices. For example, the dense LU factorization in Fig. 4 shows a diamond-shaped DAG. If such a DAG is executed in a node with four GPUs, then some of the GPUs will remain underutilized during the beginning and the end of such a DAG. For this reason, more concurrency is desired. IRIS-MEMFLOW can fuse multiple DAGs by tracking the data flow of the memory objects. The DAG fusion of the GESV solver is shown in Fig. 6. The GESV solver combines three DAGs, one for GETRF and two for TRSM. Some computation of the first TRSM can be started before the full GETRF finishes its execution. For example, once GETRF is done with the first memory object, which is not updated by any other task in the GETRF DAG, the first task of TRSM can immediately start executing. Although manual dependency creation can be conducted in non-fused cases, the fused case makes it nearly impossible to track the memory usage for large DAGs. IRIS-MEMFLOW can automatically fuse DAGs by flowing the usage of memory objects, thereby providing higher concurrency.

IRIS-MEMFLOW provides optimized data transfers for fused cases as well, as shown in Fig. 5c. The DAGs in Figs. 5a and 5b show GETRF and TRSM with a 2×2 decomposition. Figure 5c shows the fusion of GESV (GETRF + TRSM + TRSM). In the fused cases, 17 tasks update 8 memory objects. IRIS-MEMFLOW moves the flush task as the new tasks are added, ensuring exactly 8 flush tasks. If DAGs were executed separately, then there would have been 12 flush tasks. Hence, fusion provided by IRIS-MEMFLOW automatically eliminates the extra flush-out data transfer, thereby providing optimized data movement (functionality four mentioned in Section III).

V. Experiments

This section describes the performance improvements and other benefits of using IRIS-MEMFLOW over previous approaches, including improved programming productivity, portability and accuracy, and performance improvements from automated flush-out. Performance improvement for DAG fusion is also presented, followed by a discussion of the overhead of IRIS-MEMFLOW. Finally, a comparison with the state of the art is presented.

TABLE I: A diverse heterogeneous system.

System	CADES cloud node
GPUs	Total 8 GPUs 4× NVIDIA A100 4× AMD MI100
CPU	AMD EPYC 7763, 128 cores
Compiler	GNU-8.5.0
CUDA and ROCm versions	CUDA-11.7 and ROCm-5.1.2
Math libraries	MKL, cuBLAS, cuSOLVER, hipBLAS, and hipSOLVER

As shown in Table I, a Compute and Data Environment for Science (CADES) cloud node at Oak Ridge National Laboratory was used for experimentation. The node is equipped with four NVIDIA A100 GPUs and four AMD MI100 GPUs. The four NVIDIA GPUs are connected by NVLINK as two pairs, which provides the opportunity for device-to-device transfer. Six dense linear algebra benchmarks from MatRIS were used: GEMM, TRSM, GETRF (non-pivoting), POTRF, GESV, and POSV, where GESV is a fusion of GETRF (non-pivoting) and two TRSMs, and POSV is a fusion of POTRF and two TRSMs. Non-pivoting GETRF is considered for all experiments. These tiled algorithms create complex graphs of computation when higher decomposition is used. This makes them suitable for verifying the efficacy of IRIS-MEMFLOW. Moreover, the numerical accuracy of these algorithms was checked by comparing the results of the tiled version of MatRIS versus a single kernel of the appropriate vendor libraries. Block-cyclic scheduling was used for each experiment to ensure all the devices were utilized concurrently. Inspired by state-of-the-art math libraries [3], [17], only the DAG execution time was considered during the performance assessment (i.e., the DAG creation time was not counted). NVIDIA's Nsight system was used to count the number of data transfer API calls.

A. Programming Productivity

The main objective of IRIS-MEMFLOW is to enhance the IRIS runtime by providing a portable abstraction for memory orchestration so that an application developer can express the computation by using architecture-agnostic IRIS APIs and IRIS to leverage all four functionalities mentioned in Section III. Providing all these functionalities requires writing more code and significant manual effort/analysis. IRIS-MEMFLOW eliminates the need for manual effort, thereby significantly boosting programming productivity, which can be difficult to quantify. However, the number of source code reductions can be quantified, and the GETRF implementation in MatRIS required 40% less code with the features provided by IRIS-MEMFLOW. For example, Fig. A.2 in Appendix I shows an example of POTRF, which comprises serial-for loops without any architecture description but becomes a fully portable Cholesky decomposition at run time in CPU and GPU environments. So, this unquantifiable boost in programming productivity is one of the significant benefits of

IRIS-MEMFLOW. The rest of this section discusses the more quantifiable performance results.

TABLE II: Portability and accuracy for different hardware configurations. Each of the six algorithms was run for two data sizes (32 × 32 and 1,024 × 1,024) with the decomposition of two tile numbers (2 × 2 and 16 × 16), where each combination is run five times, making 120 checks for each hardware combination.

Modes	**Hardware combination**
SYNC	ONLY CPU
SYNC	2 NVIDIA GPUs
SYNC	4 NVIDIA GPUs
SYNC	2 AMD GPUs
SYNC	4 AMD GPUs
SYNC	2 NVIDIA GPUs and 2 AMD GPUs
SYNC	4 NVIDIA GPUs and 4 AMD GPUs
SYNC	1 CPU, 4 NVIDIA GPUs, and 4 AMD GPUs
ASYNC	2 NVIDIA GPUs
ASYNC	4 NVIDIA GPUs
ASYNC	2 AMD GPUs
ASYNC	4 AMD GPUs
ASYNC	2 NVIDIA GPUs and 2 AMD GPUs
ASYNC	4 NVIDIA GPUs and 4 AMD GPUs

B. Portability and Accuracy

Once compiled, MatRIS algorithms are portable, and IRIS can use multiple heterogeneous devices concurrently at run time. IRIS supports asynchronous and synchronous execution. In asynchronous mode, IRIS uses multiple streams supported by CUDA and ROCm runtimes for data transfers and kernel execution, increasing the chance of computation and communication overlap. The same implementations of the MatRIS algorithms are tested on different hardware configurations (Table II), demonstrating the portability of IRIS's memory abstraction. As shown in Table II, both small and large DAGs created by IRIS-MEMFLOW for small and large data sizes are tested for different hardware combinations to check whether IRIS-MEMFLOW can provide the correct numerical results by guarding against deadlocks, race conditions, and data races. IRIS-MEMFLOW is evaluated for numerical accuracy by using MatRIS (up to the third digit of the floating points).

The execution provided correct numerical results for a total of 1,680 combinations of the six algorithms from MatRIS. This outcome shows that IRIS-MEMFLOW can provide executions free of deadlock, race conditions, and data races while ensuring the portability of MatRIS to all these hardware combinations, thereby providing the first functionally mentioned in Section III.

C. Performance Improvement for Automated Flush-Out

IRIS-MEMFLOW enhances the current capability of the IRIS runtime by introducing an automated and optimized data movement mechanism for the flush-out operations. Through data-flow analysis, IRIS-MEMFLOW ensures one flush-out data movement for each memory object during the execution of a DAG. Without such automation, there is a risk of unnecessary data movement introduced by manual intervention. A comparison between IRIS-MEMFLOW enabled flush-out and manual (inefficient) flush-out is presented in Fig. 7. For the manual (and inefficient) data movement cases, data is written back to the host when a memory object is updated, and the data movement happens right after kernel execution, thereby delaying other tasks/kernel execution. In contrast, IRIS-MEMFLOW only performs one flush-out for a memory object, and this is done asynchronously to the execution flow, enabling computation and communication overlap. In the asynchronous mode, IRIS uses different streams for executing kernel and data transfer, which increases the possibility of computation and communication overlap. For this reason, each benchmark running on the four NVIDIA A100 GPUs demonstrated 15%–28% performance improvement. Note that the matrix size (32,768 × 32,768), the tile size (4,096 × 4,096), the execution mode (asynchronous), the number of task/kernels, and the scheduling algorithm were kept the same for both cases, and Fig. 7 shows only the improvement achieved by the automated flush-out of IRIS-MEMFLOW. The number of data transfers (a sum of host-to-device, device-to-device, and device-to-host transfers) is shown in Table III. Notably, IRIS-MEMFLOW reduced data transfers by 59%. So, IRIS-MEMFLOW provides optimized data transfers while increasing the possibility of computation and communication overlap, thereby providing the second and third functionalities mentioned in Section III.

TABLE III: Number of data transfers in automated vs. manual flush-out.

Flush-out	GEMM	TRSM	GETRF	POTRF	GESV	POSV
Automated	388	263	260	134	533	420
Manual	800	487	400	218	1,185	1,016
Gain	52%	46%	35%	39%	55%	59%

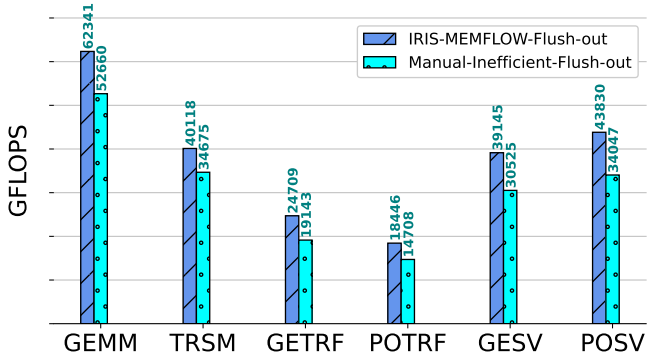
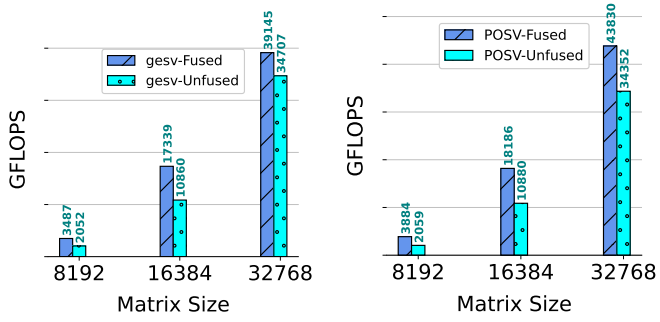


Fig. 7: Impact of data-flow enabled flush-out in IRIS-MEMFLOW.

D. Performance Improvement for DAG Fusion

Two benchmarks from the MatRIS algorithms, GESV and POSV, provide the opportunity to investigate DAG fusion. GESV combines GETRF with two TRSMs, and POSV combines POTRF with two TRSMs. IRIS-MEMFLOW automatically fuses these DAGs (Fig. 6). Through fusion, IRIS-MEMFLOW exposes more concurrency in the fused DAG and ensures fewer data movements. The performance comparison of fused versus unfused DAGs is shown in Fig. 8 for different matrix sizes, where the number of tiles is 8×8 and executed on four NVIDIA A100 GPUs. The fused execution of GESV and



(a) GESV (b) POSV

Fig. 8: Impact of DAG fusion in GESV and POSV.

POSV provides a 12%–89% improvement over the unfused and efficient versions of GETRF + TRSM + TRSM and POTRF + TRSM + TRSM. Smaller matrix sizes provide more performance improvement over larger matrix sizes because computation dominates the larger sizes. The improvement is from the increased concurrency in the DAG (Fig. 6) and fewer data transfers (Table IV). Through fusion, GESV and POSV reduce data transfer API calls for the CUDA runtime by up to 36%, thereby providing functionality four mentioned in Section III.

TABLE IV: Number of data transfers for fused vs. unfused DAGs.

	GESV	POSV
Fused	533	420
Unfused	786	660
Gain	32%	36%

E. Overhead of IRIS-MEMFLOW

DAGs in IRIS are created once but executed many times. The data-flow analysis is conducted during DAG creation; therefore, the overhead is added only once and does not impact the actual execution of the DAG. Additional decomposition of the matrix for the algorithms in MatRIS creates more tasks; therefore, more overhead is added for the data-flow analysis. The overhead of IRIS-MEMFLOW is shown in Table V in milliseconds. For the worst case, GESV and POSV with 16×16 decomposition take up to 47 milliseconds. For a 32,768 $\times$ 32,768 matrix, GESV and POSV take at least 2 seconds to execute on four NVIDIA A100 GPUs. Therefore, the one-time overhead is 2% of the execution time for the worst case shown in Table V. IRIS inspires coarse-grain task parallelism because each task executes a kernel. For this reason, most of the performance results reported in the paper use an 8×8 decomposition in which IRIS-MEMFLOW introduces negligible overhead during DAG creation.

F. Comparison with the State of the Art

Here, we compare the overall solution provided by IRIS-MEMFLOW for the algorithms in MatRIS against the DPLASMA math library [3] built on the PaRSEC runtime [4] and Chameleon [17] built on the StarPU runtime [1]. All the algorithms run on four NVIDIA A100 GPUs and use the same

TABLE V: Overhead of IRIS-MEMFLOW in milliseconds.

Name of the algorithms	Decomposition 8×8		Decomposition 16×16	
	No. of tasks	Overhead	No. of tasks	Overhead
GEMM	513	1.76 ms	4,097	15.01 ms
TRSM	289	2.00 ms	2,177	18.97 ms
GETRF	205	0.83 ms	1,497	4.61 ms
POTRF	121	0.74 ms	817	4.62 ms
GESV	781	5.76 ms	5,849	47.34 ms
POSV	697	4.71 ms	5,169	40.76 ms

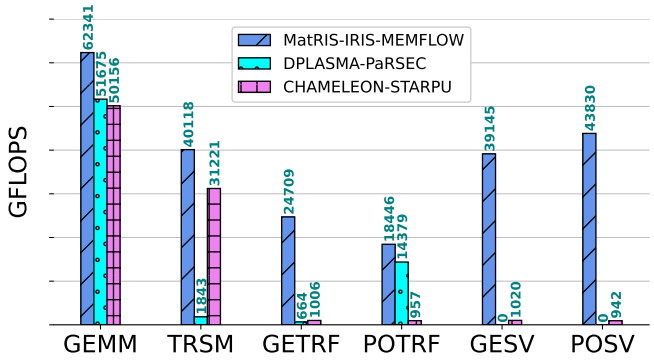


Fig. 9: Comparison of MatRIS using IRIS-MEMFLOW, DPLASMA using PaRSEC, and Chameleon using StarPU.

matrix size (32,768 $\times$ 32,768) and tile size (4,096 $\times$ 4,096). As shown in Fig. 9, MatRIS provides better performance in all cases. Chameleon uses the CPU for certain kernels in GETRF, POTRF, GESV, and POSV (the `getrf` and `potrf` kernels), which results in poor performance. However, Chameleon uses GPUs for GEMM and TRSM. Still, they introduce 72% and 67% more memory transfers, respectively, than MatRIS when measured with NVIDIA's Nsight, and these additional memory transfers can be correlated with lower performance versus MatRIS. DPLASMA did not have an implementation for non-pivoting GESV, and the POSV did not report the flops, so these results are not shown. DPLASMA does show comparable performance for GEMM and POTRF, but they introduce 39% and 50% more memory transfers, respectively, than when using MatRIS, which can be correlated with the lower performance. The very low performance of TRSM and GETRF when using DPLASMA is because some kernels execute on the CPU even though GPU-only execution is specified.

Empowered by the IRIS runtime and the features of IRIS-MEMFLOW, MatRIS provided better portability and performance than the state-of-the-art libraries.

VI. Related Work

A. Data Flow in Compilers and Runtimes

Many task-based programming systems (e.g., Cilk [27], OmpSs [2], OpenMP [26], Charm++ [14], X10 [7]) use their dedicated compilers for programming productivity and performance optimizations. For example, an OpenMP compiler automatically splits the input OpenMP codes into the host and device codes. The device code applies various optimizations and transformations and is converted into either low-level, device-specific code or a binary suitable for the target device. However, most task programming compilers still rely on

runtimes and/or user inputs for task graph constructions, task scheduling, and memory transfers due to their dynamic nature, which requires runtime information. The IRIS programming system also uses multiple front-end compilers (OpenARC [18] and charmSYCL [10]) to support high-level programming abstractions: (1) the OpenARC compiler takes OpenACC or OpenMP as input programs and performs source-to-source translation to generate output IRIS host code and device-specific kernels, and (2) the charmSYCL compiler internally uses the IRIS runtime as its SYCL back end to concurrently support multiple heterogeneous devices. Like other task programming systems, the IRIS compilers also rely on the IRIS runtime to create a DAG of tasks and orchestrate them in diverse heterogeneous systems. Therefore, the proposed IRIS-MEMFLOW can benefit both high-level IRIS compilers (OpenARC and charmSYCL) by enabling the compilers to exploit the automated, data flow–enabled portable memory abstraction offered by IRIS-MEMFLOW.

B. Data Flow in Math Libraries

Task-based (data flow) programming models have been used extensively in math libraries, specifically in level-3 BLAS and LAPACK routines [8], [28], [30]. Although most studies focus on dense linear algebra problems, some sparse linear algebra efforts [5], [6], [19], [31] do use data flow-based techniques and outperform highly optimized sparse linear algebra libraries by providing not only much better performance but also much simpler ways to implement relatively complex sparse algorithms in a portable way. In all these solutions, data flow is achieved by using task-based programming models.

This kind of problem is amenable to further optimization through hardware- and problem-specific tuning. Examples include weak-dependencies or hierarchical tasks, which help better match hardware and computation [29]. However, all these optimizations require programmers to change the code according to the characteristics of the problem and the hardware's features, and these optimizations are available only for multicore CPUs.

However, the optimizations described in the present work do not require any modification to the codes, thereby providing a better solution in terms of programming productivity and performance portability on multiGPU heterogeneous systems.

VII. Conclusions

This paper presents IRIS-MEMFLOW, a portable abstraction for memory orchestration in heterogeneous computing systems. IRIS-MEMFLOW enhances the capabilities of the IRIS task-based runtime by introducing data-flow analysis on high-level portable memory objects to ensure accurate execution when multiple heterogeneous devices are used concurrently. IRIS-MEMFLOW establishes a connection between high-level portable memory abstraction and low-level memory orchestration in heterogeneous devices, ensuring optimized data movement between host and devices. Moreover, IRIS-MEMFLOW increases the opportunity for computation and communication to overlap and fuse multiple DAGs. As a result, a programmer can write a serial, architecture-agnostic code without specifying any data movement. This code can then be ported to different heterogeneous systems without any change in the source code, ensuring significant programming productivity. Leveraging the optimization provided by IRIS-MEMFLOW, the MatRIS math library outperforms the state-of-the-art math libraries in portability and performance.

Acknowledgments

This work is funded in part by Bluestone, an X-Stack project in the US Department of Energy's Advanced Scientific Computing Office with program manager Hal Finkel and by the U.S. Department of Defense Advanced Research Projects Agency (DARPA), through the COSMIC project with the Microsystems Technology Office (MTO).

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[2] E. Ayguade, R. Badía, and J. Labarta. Ompss and the nanos++ runtime.

[3] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, et al. Distibuted dense numerical linear algebra algorithms on massively parallel architectures: Dplasma. 2010.

[4] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[5] S. Catalán, X. Martorell, J. Labarta, T. Usui, L. A. T. Díaz, and P. Valero-Lara. Accelerating conjugate gradient using ompss. In *20th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2019, Gold Coast, Australia, December 5-7, 2019*, pages 121–126. IEEE, 2019.

[6] S. Catalán, T. Usui, L. Toledo, X. Martorell, J. Labarta, and P. Valero-Lara. Towards an auto-tuned and task-based spmv (lass library). In K. F. Milfeld, B. R. de Supinski, L. Koesterke, and J. Klinkenberg, editors, *OpenMP: Portable Multi-Level Parallelism on Modern Systems - 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22-24, 2020, Proceedings*, volume 12295 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2020.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[8] J. J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, P. Wu, I. Yamazaki, A. YarKhan, M. Abalenkovs, N. Bagherpour, S. Hammarling, J. Šístek, D. Stevens, M. Zounon, and S. D. Relton. PLASMA: parallel linear algebra software for multicore using openmp. *ACM Trans. Math. Softw.*, 45(2):16:1–16:35, 2019.

[9] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[10] N. Fujita, B. Johnston, R. Kobayashi, K. Teranishi, S. Lee, T. Boku, and J. S. Vetter. Charm-sycl: New unified programming environment for multiple accelerator types. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 1651–1661, 2023.

[11] T. Gautier, F. Lementec, V. Faucher, and B. Raffin. X-kaapi: A multi paradigm runtime for multicore architectures. In *2013 42nd International Conference on Parallel Processing*, pages 728–735, 2013.

[12] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1303–1320, 2022.

[13] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 6. ACM, 2014.

[14] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[15] R. Keryell, R. Reyes, and L. Howes. Khronos sycl for opencl: a tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*, pages 1–1, 2015.

[16] J. Kim, S. Lee, B. Johnston, and J. S. Vetter. IRIS: A portable runtime system exploiting multiple heterogeneous programming systems. In *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021*, pages 1–8. IEEE, 2021.

[17] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, and M. S. Müller. Chameleon: reactive load balancing for hybrid mpi+ openmp task-parallel applications. *Journal of Parallel and Distributed Computing*, 138:55–64, 2020.

[18] S. Lee and J. S. Vetter. OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, Vancouver, 2014. ACM.

[19] A. Lisito, M. Faverge, G. Pichon, and P. Ramet. Enhancing sparse direct solver scalability through runtime system automatic data partition. In P. Diehl, J. Schuchart, P. Valero-Lara, and G. Bosilca, editors, *Asynchronous Many-Task Systems and Applications - Second International Workshop, WAMTA 2024, Knoxville, TN, USA, February 14-16, 2024, Proceedings*, volume 14626 of *Lecture Notes in Computer Science*, pages 105–110. Springer, 2024.

[20] N. R. Miniskar, M. A. Haque Monil, P. Valero-Lara, F. Y. Liu, and J. S. Vetter. Iris-dmem: Efficient memory management for heterogeneous computing. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2023.

[21] N. R. Miniskar, M. A. H. Monil, P. Valero-Lara, F. Liu, and J. S. Vetter. Iris-blas: Towards a performance portable and heterogeneous blas library. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 256–261. IEEE, 2022.

[22] N. R. Miniskar, M. A. H. Monil, P. Valero-Lara, F. Y. Liu, and J. S. Vetter. Iris-dmem: efficient memory management for heterogeneous computing. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2023.

[23] 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, Dallas, TX, USA, November 13–18, 2022*. IEEE.

[24] M. A. H. Monil, N. R. Miniskar, F. Y. Liu, J. S. Vetter, and P. Valero-Lara. Laris: Targeting portability and productivity for lapack codes on extreme heterogeneous systems by using iris. In *2022 IEEE/ACM Redefining Scalability for Diversely Heterogeneous Architectures Workshop (RSDHA)*, pages 12–21. IEEE, 2022.

[25] M. A. H. Monil, N. R. Miniskar, K. Teranishi, J. S. Vetter, and P. Valero-Lara. Matris: Multi-level math library abstraction for heterogeneity and performance portability using iris runtime. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 1081–1092, 2023.

[26] OpenMP. OpenMP reference, 1999.

[27] A. D. Robison. Composable parallel patterns with intel cilk plus. *Computing in Science & Engineering*, 15(2):66–71, 2013.

[28] D. Sukkari, H. Ltaief, M. Faverge, and D. E. Keyes. Asynchronous task-based polar decomposition on single node manycore architectures. *IEEE Trans. Parallel Distributed Syst.*, 29(2):312–323, 2018.

[29] P. Valero-Lara, S. Catalán, X. Martorell, and J. Labarta. BLAS-3 optimized by ompss regions (lass library). In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, Pavia, Italy, February 13-15, 2019*, pages 25–32. IEEE, 2019.

[30] P. Valero-Lara, S. Catalán, X. Martorell, T. Usui, and J. Labarta. slass: A fully automatic auto-tuned linear algebra library based on openmp extensions implemented in ompss (lass library). *J. Parallel Distributed Comput.*, 138:153–171, 2020.

[31] P. Valero-Lara, C. Greenwalt, and J. S. Vetter. Sparselu, A novel algorithm and math library for sparse LU factorization. In *12th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms, IA3@SC 2022, Dallas, TX, USA, November 13-18, 2022*, pages 25–31. IEEE, 2022.

APPENDIX I

Figure A.1 shows a simple directed acyclic graph (DAG) of two tasks using high level APIs of the IRIS runtime without IRIS-MEMFLOW. This example shows how IRIS memory objects are created and linked to the corresponding host addresses (lines 19–22). Then, tasks are created to invoke the `vecadd` kernel at run time to compute [C = A + B and B = C + B] and added to a graph object. The memory objects and access patterns are provided in lines 28–29 and 37–38. Flush tasks at lines 32 and 41 ensure the result is returned to the host from the most up-to-date device copies. Moreover, a dependency between tasks is specified in line 42 to enforce the execution order and avoid race conditions.

```
1  #include <iris/iris.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <malloc.h>
5
6  int main(int argc, char** argv) {
7    size_t SIZE;
8    int *A, *B, *C;
9    iris_init(&argc, &argv, true);
10   SIZE = argc > 1 ? atol(argv[1]) : 16;
11   size_t SIZE_bytes = SIZE * sizeof(int);
12   A = (int*) malloc(SIZE_bytes);
13   B = (int*) malloc(SIZE_bytes);
14   C = (int*) malloc(SIZE_bytes);
15
16   for (int i = 0; i < SIZE; i++)
17     {A[i] = I; B[i] = I; C[i] = 0;}
18
19   iris_mem mem_A, mem_B, mem_C;
20   iris_data_mem_create(&mem_A, A, SIZE_bytes);
21   iris_data_mem_create(&mem_B, B, SIZE_bytes);
22   iris_data_mem_create(&mem_C, C, SIZE_bytes);
23
24   iris_graph graph; iris_graph_create(&graph);
25
26   iris_task task0;
27   iris_task_create(&task0);
28   void* params0[3] = { &mem_A, &mem_B, &mem_C };
29   int pinfo0[3] = { iris_r, iris_r, iris_w };
30   iris_task_kernel(task0, "vecadd", 1, NULL,
31            &SIZE, NULL, 3, params0, pinfo0);
32   iris_task_dmem_flush_out(task0, mem_C);
33   iris_graph_task(graph, task0, iris_random, NULL);
34
35   iris_task task1;
36   iris_task_create(&task1);
37   void* params1[3] = { &mem_C, &mem_B, &mem_B };
38   int pinfo1[3] = { iris_r, iris_r, iris_w };
39   iris_task_kernel(task1, "vecadd", 1, NULL,
40            &SIZE, NULL, 3, params1, pinfo1);
41   iris_task_dmem_flush_out(task1, mem_B);
42   iris_task_depend(task1, 1, task0);
43   iris_graph_task(graph, task1, iris_random, NULL);
44
45   iris_graph_submit(graph, iris_random, 1);
46   iris_finalize();
47 }
```

Fig. A.1: A simple IRIS program without IRIS-MEMFLOW.

Using IRIS-MEMFLOW, the IRIS program can be expressed as a simple sequential code, and portability and compatibility with multiple heterogeneous devices is provided automatically. An example of Cholesky factorization in MatRIS is shown in Fig. A.2. In contrast to Fig. A.1, Fig. A.2 does not need manual specification of flush and dependencies; IRIS-MEMFLOW automatically performs those.

```
1  if (uplo == MATRIS_BLAS_LOWER) {
2    for (int k = 0; k < A_til.row_tiles_count(); k++) {
3       IRIS_FUNC(laris_graph_common_, POTRF_TYPE)
4            ( graph... A_til.GetAt(k,k).IRISMem()...);
5       for (int m = k+1; m < A_til.row_tiles_count(); m++){
6            IRIS_FUNC(laris_graph_common_, TRSM_TYPE)
7                 ( graph...A_til.GetAt(k,m).IRISMem()...);
8       }
9       for (int m = k+1; m < A_til.row_tiles_count(); m++){
10           IRIS_FUNC(laris_graph_common_, SYRK_TYPE)
11                ( graph...A_til.GetAt(m,m).IRISMem());
12           for (int n = k+1; n < m; n++) {
13                IRIS_FUNC(laris_graph_common_, GEMM_TYPE)
14                     ( graph...A_til.GetAt(n,m).IRISMem());
15           }
16      }
17   }
18 }
```

Fig. A.2: Sequential, architecture-agnostic code for Cholesky factorization in MatRIS using IRIS-MEMFLOW.