MatRIS: Multilevel Math Library Abstraction for Heterogeneity and Performance Portability using IRIS Runtime

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

ABSTRACT

Vendor libraries are tuned for a specific architecture and are not portable to others. Moreover, they lack support for heterogeneity and multi-device orchestration, which is required for efficient use of contemporary HPC and cloud resources. To address these challenges, we introduce MatRIS—a multilevel math library abstraction for scalable and performance-portable sparse/dense BLAS/LAPACK operations using IRIS runtime. The MatRIS-IRIS co-design introduces three levels of abstraction to make the implementation completely architecture agnostic and provide highly productive programming. We demonstrate that MatRIS is portable without any change in source code and can fully utilize multi-device heterogeneous systems by achieving high performance and scalability on Summit, Frontier, and a CADES cloud node equipped with four NVIDIA A100 GPUs and four AMD MI100 GPUs. A detailed performance study is presented in which MatRIS demonstrates multi-device scalability. When compared, MatRIS provides competitive and even better performance than libraries from vendors and other third parties.

KEYWORDS

Performance portability; scalability; programming productivity; extreme heterogeneity; GPUs; BLAS; LAPACK; LU factorization

1 INTRODUCTION

Although heterogeneous architectures are now ubiquitous in contemporary HPC and cloud facilities, they introduce significant challenges in scalability, scheduling/management, portability, and programming productivity. In the extremely heterogeneous computing paradigm, very different architectures coexist in the same system. For example, a heterogeneous system may have GPUs from multiple vendors (or even FPGAs). Hence, utilizing different computing resources seamlessly and efficiently to extract meaningful performance benefits is necessary. A wide range of software stacks offered by vendors and open-source communities, including architecture-specific tuned libraries, adds to the complexity of harnessing performance. Moreover, making solutions that are portable to different heterogeneous systems adds another dimension to the overall challenges. Therefore, portable orchestration of the computation through management of different hardware architectures and their tuned libraries is becoming an active area of research.

In most cases, the current software and technology stacks lack the capability to ensure portability and support for extreme heterogeneity. Although some solutions target portability [23] or heterogeneity [14], finding a solution that targets both is difficult. And although math libraries for distributed systems is a well-researched area, a portable solution for diverse heterogeneity is uncommon. However, there are runtime systems [2, 4] and math libraries [14, 18] that provide some functionalities, but they lack support for diverse heterogeneity and often have architecture-dependent implementations that limit the development of math algorithms. All this brings us to our research question: Can an extensible abstraction layer be created so that the same algorithm—the same piece of code with no information about the hardware/software layers (i.e., architecture- and software-agnostic)—be effectively deployed (i.e., perform well and achieve scalability) on different multi-device and heterogeneous platforms (e.g., clusters and clouds)?

To address these challenges, we present MatRIS—a novel multilevel abstraction for *Mat*h library on top of the I*RIS* runtime [5, 17]. Being coupled with IRIS, the MatRIS software stack exposes different abstractions in each level. At the lowest-level, the *IRIS runtime layer* abstracts the vendor runtime and architecture. At the mid-level, the *kernel layer* provides an abstraction for different math libraries and their kernels. Finally, at the top-level, the *algorithm layer* provides an abstraction for architecture-agnostic BLAS/LAPACK algorithms. By doing so, MatRIS provides a means for the seamless inclusion of new algorithms, libraries, architectures, and devices without impacting the other layers.

MatRIS also enables the utilization of all devices for a single program and seamless portability to different heterogeneous systems. Moreover, MatRIS provides automatic and transparent tiling and orchestration capabilities. Overall, the MatRIS objectives are as follows: (i) provide performance-portable math codes (dense/sparse BLAS and LAPACK operations) on extremely heterogeneous systems, (ii) seamlessly include new libraries and architectures, (iii) exploit the computational capabilities of heterogeneous systems by using tiled algorithms, and (iv) use transparent, algorithm-specific performance models to guide scheduling in heterogeneous systems, thereby providing a highly productive programming solution for math libraries.

Using tiled sparse/dense LU factorization as test cases, the work described here demonstrates that MatRIS is portable to various heterogeneous systems in HPC and cloud environments and provides a highly productive programming solution that enables portability and scalability without any changes to the source code. Moreover, MatRIS can utilize all the processing components by orchestrating tuned kernel tasks from different vendor-provided/open-source math libraries.

This work extends a previously published paper [21] on the LaRIS library. The main contributions of the present work are the development of the following features of the MatRIS library:

- multilevel abstraction of heterogeneity for transparent, portable, and highly productive programming for sparse and dense BLAS/LAPACK operations;

- a transparent memory management strategy (for heterogeneous resources) that can identify and make decisions about different characteristics of the hardware and connectivity, thereby reducing the number of memory transfers and maximizing the use of high-bandwidth connections;
- complete separation of algorithm and architecture for improved programmability, which can hide all details related to data transformation during the decomposition/composition of the matrices;
- a detailed performance analysis using dense/sparse LU factorization as test cases on three heterogeneous systems with different hardware and connectivity configurations; and
- a comparison with vendor and other library solutions that shows MatRIS provides competitive and even better performance in some cases.

The rest of the paper is organized as follows: Section 2 describes the different optimizations, the algorithms used, and details about the implementation. The performance study is described in Section 3. Section 4 summarizes essential contributions from the literature. Finally, we conclude the paper with final remarks and future directions in Section 5.

2 MATRIS

This section discusses the MatRIS software stack and its three layers. It also describes the co-design effort with the IRIS runtime, which makes the multilevel abstraction possible.

2.1 Enabling Programming Productivity and Performance Portability

To make MatRIS portable, we separate the algorithm's design from the tuning. At the top layer, the algorithm design involves expressing the tiled algorithms by using tasks and dependencies. Tuning consists of choosing the target kernels and processors for executing each task and is facilitated by the kernel and runtime layers, which are completely transparent to the algorithm developers. The bottom two levels of abstractions are facilitated by the IRIS runtime [5, 17] and the unified kernel API, which enable the separation (Figure 1). Tiled algorithms are expressed by using type-less MatRIS APIs, which do not include any architecture- or vendor-library specific detail (Figure A1 in APPENDIX). Therefore, MatRIS codes are completely agnostic of the architecture and vendor library. This feature enables the implementation of different tiled algorithms for math codes without having to worry about vendor libraries or the underlying hardware.

Tuning occurs at run time. With the help of IRIS, MatRIS tasks are scheduled on different processors in a heterogeneous system on which vendor-specific kernels suitable for those processors are executed. The IRIS runtime's scheduler enables the full orchestrations, during which the unified kernel layer dynamically provides the vendor-specific tuned kernels. Thus, the tuning phase does not require code modifications at the algorithm level, but the scheduler conducts them internally and transparently for each task. Leveraging a dynamic scheduler, the set of tasks in a MatRIS algorithm attempts to obtain the maximum performance on the target architecture by maximizing the use of all the computational resources available in a heterogeneous system. This hierarchical design allows the addition of a new algorithm, math library, or vendor runtime without any interference from other layers.

Detailed descriptions of each layer and its essential components are provided in the following sections as a bottom-up approach to show how each layer provides a necessary abstraction to the upper layer.

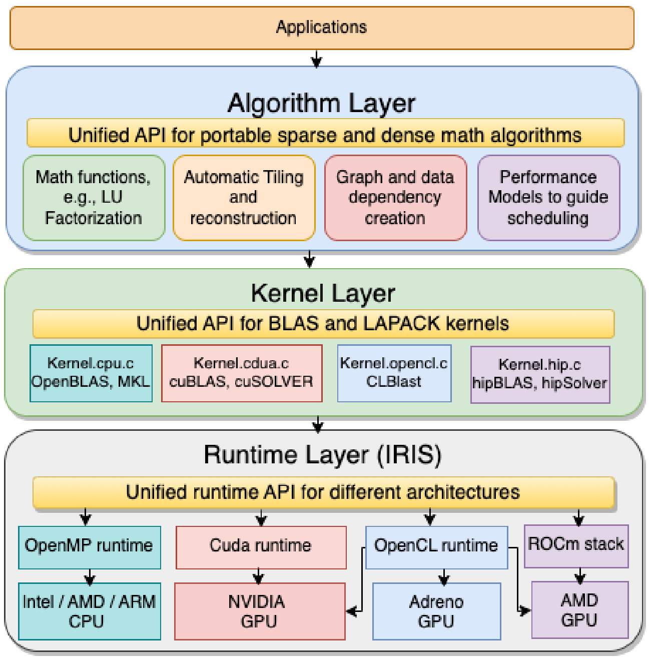
Applications
Algorithm Layer
Unified API for portable sparse and dense math algorithms
Math functions, e.g., LU Factorization
Automatic Tiling and reconstruction
Graph and data dependency creation
Performance Models to guide scheduling
Kernel Layer
Unified API for BLAS and LAPACK kernels
Kernel.cpu.c OpenBLAS, MKL
Kernel.cdua.c cuBLAS, cuSOLVER
Kernel.opencl.c CLBlast
Kernel.hip.c hipBLAS, hipSolver
Runtime Layer (IRIS)
Unified runtime API for different architectures
OpenMP runtime
Cuda runtime
OpenCL runtime
ROCm stack
Intel / AMD / ARM CPU
NVIDIA GPU
Adreno GPU
AMD GPU

Figure 1: MatRIS software stack and design.

2.2 Runtime Layer: IRIS Runtime

The bottom runtime layer of MatRIS (Figure 1) is facilitated by the IRIS [5, 17] runtime, which is a task-based programming model and runtime for heterogeneous computing systems that consist of multicore CPUs, GPUs (NVIDIA, AMD), DSPs (Qualcomm Hexagon), and/or FPGAs (Xilinx, Intel). IRIS accepts kernels written in OpenMP, OpenCL, CUDA, HIP, XilinxCL, IntelCL, Hexagon C++, and OpenACC programming languages. However, mapping the programming language to the compute device is constrained. For example, NVIDIA GPUs can be used only through kernels written in OpenCL and CUDA. The IRIS runtime has a task scheduler that maps the application tasks to a compute device, and the task kernels are executed using the compute device–specific runtime.

IRIS exposes a task-based programming model in which a task is a scheduling unit. This task runs on a single device but is portable across any processing element in a given system. A task can contain zero or more commands, and there are four types of commands: (1) Host-to-Device (H2D) memory copy, (2) Device-to-Host (D2H) memory copy, (3) kernel launch, and (4) host. Because a task can depend on other tasks, it cannot start until its prerequisite tasks are executed. Therefore, writing an IRIS application means building directed acyclic graphs (DAGs) of tasks. Each task has a target device selection policy when it is submitted. The application written using the IRIS task definition specifies the policy, which can be a device number, device type (e.g., CPU, GPU, FPGA, DSP), or a built-in policy provided by IRIS (e.g., greedy, random, locality-aware, profile).

IRIS provides shared virtual device memory across multiple, disjointed physical device memories to achieve application portability and flexible task scheduling with effective data orchestration. IRIS automatically transfers data across multiple devices to keep memory consistency across tasks. Therefore, all compute devices can share memory objects in the shared virtual device memory and see the same content in the memory objects. The IRIS scheduler and memory management are further enhanced to make MatRIS efficient. These enhancements are discussed below.

2.2.1 Scheduler. Effective workload distribution (tasks/tiles mapping) is vital to balancing computational effort between different devices and achieving good scalability by keeping all resources busy most of the time. To do so, we implemented an automatic task policy in the IRIS runtime based on the 2D block-cyclic algorithm (Figure 2). This algorithm is widely used to distribute the computational cost of linear algebra operations for MPI programs [13]. For example, it is used for the High Performance Linpack (HPL) benchmark.[1] We adopted such an algorithm to be used in multi-device and extremely heterogeneous systems. Moreover, MatRIS can also benefit from the existing scheduling policies of IRIS.

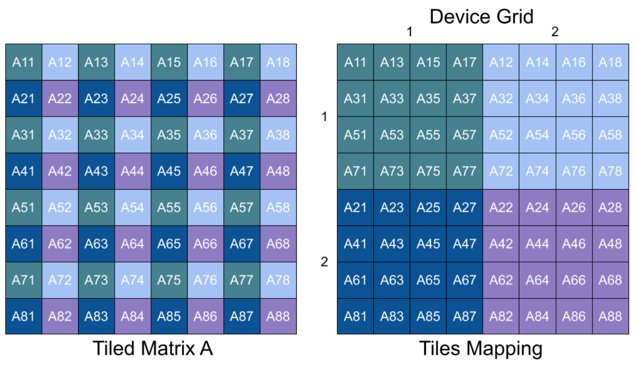
Device Grid
1
2
1
2
A11 A12 A13 A14 A15 A16 A17 A18
A21 A22 A23 A24 A25 A26 A27 A28
A31 A32 A33 A34 A35 A36 A37 A38
A41 A42 A43 A44 A45 A46 A47 A48
A51 A52 A53 A54 A55 A56 A57 A58
A61 A62 A63 A64 A65 A66 A67 A68
A71 A72 A73 A74 A75 A76 A77 A78
A81 A82 A83 A84 A85 A86 A87 A88
Tiled Matrix A
A11 A13 A15 A17 A12 A14 A16 A18
A31 A33 A35 A37 A32 A34 A36 A38
A51 A53 A55 A57 A52 A54 A56 A58
A71 A73 A75 A77 A72 A74 A76 A78
A21 A23 A25 A27 A22 A24 A26 A28
A41 A43 A45 A47 A42 A44 A46 A48
A61 A63 A65 A67 A62 A64 A66 A68
A81 A83 A85 A87 A82 A84 A86 A88
Tiles Mapping

Figure 2: Task mapping according to the 2D block cyclic algorithm on a 2 × 2 device grid.

2.2.2 Efficient Memory Management. Heterogeneous memory handling plays an important role in application performance on heterogeneous computing resources. Accelerators such as GPUs have self-controlled memories (e.g., DDR4, HBM2) to achieve higher-performance kernel executions. If the data movement between tasks and the reuse of data objects is not carefully orchestrated, then the data transfer costs can dominate the kernel acceleration performance on the devices. To make MatRIS efficient in heterogeneous systems, IRIS leverages Distributed Data Memory Management (DMEM), which enables transparent and efficient data communication (including Device-to-Device [D2D] communication) and management for heterogeneous systems.

DMEM is a logical memory handler that holds the addresses of host- and device-memory objects for an application's data object.

[1]https://netlib.org/benchmark/hpl/algorithm.html

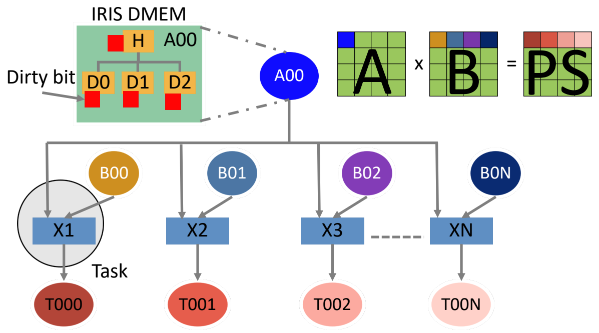

Figure 3: IRIS-DMEM data handler and usage. A and B are tiled matrices, and PS is a partial sum tiled matrix.

DMEM also maintains a *dirty flag* for each of these host- and device-memory objects (as shown in Figure 3) and tracks whether the host or device memory object has valid/most recent data. The DMEM controller logic sets the dirty flags based on the data transfers and task execution. It assumes that only one device executes a certain task and gains control of the DMEM object at any point in time. In the example shown in Figure 3, the *A00* tile (A tile of the *A* matrix) is an IRIS-DMEM memory object, which is shared to all tasks (*X1, X2, ..., XN tiled matrix multiplication kernels*). Programmers do not need to write H2D and D2H commands for the DMEM objects. The DMEM controller in IRIS can call the H2D, D2D, and D2H data transfers based on the workflow requirement at runtime.

The DMEM memory handler works as a write-back cache because it does not transfer the data to the host memory location by default. Hence, the programmer must explicitly write the new IRIS command *DMEM_FLUSH_OUT_CMD* (DMEM flush out command) after the execution of all tasks/task graphs to ensure the output is brought to the host memory object. The flush out command execution will check if the dirty host flag is empty; if it is dirty, then it will bring the data from the device with the valid data by using D2H data transfer.

Additionally, the host (CPU) data allocations are conducted using pinned memory by default in MatRIS. Pinned memory is used as a staging area for transfers from the device to the host, thereby avoiding the cost of the transfer between pageable and pinned host arrays.

In summary, the runtime layer abstracts the underlying runtimes and their efficient orchestration, and in doing so, it exposes a unified runtime environment to the upper layer.

```
int matris_task_dgemm( iris_task task0, IRISBlasType major,
  IRISBlasType a_trans, IRISBlasType b_trans,
  int M, int N, int K,
  float alpha, iris_mem IRIS_VAR(A), int lda,
               iris_mem IRIS_VAR(B), int ldb,
  float beta, iris_mem IRIS_VAR(C), int ldc ) {
    IRIS_TASK_NO_DT( task0, "iris_dgemm_kernel", 1,
      NULL_OFFSET, GWS(M), NULL_LWS,
      PARAM(IRISBlasType, major),
      PARAM(IRISBlasType, a_trans), ...
      IN_TASK(A, double*, double, A, sizeof(double)*M*N),
      PARAM(int, lda), ...
      IN_OUT_TASK(C, double*, double, C, sizeof(double)*M*K),
      PARAM(int, ldc) );
    return IRIS_SUCCESS;
}
```

Figure 4: DGEMM API from the kernel layer.

2.3 Kernel Layer: Unified API for BLAS and LAPACK Kernels

MatRIS's middle/kernel layer sits on top of the runtime layer to provide unified kernel APIs for BLAS and LAPACK kernels. The IRIS-BLAS [19] library has been modified and adopted in MatRIS to address the portability challenges of BLAS/LAPACK kernels for different heterogeneous architectures. The kernel layer links multiple vendor and open-source BLAS libraries (Figure 1) and supports OpenBLAS [37], Intel MKL [16], NVIDIA cuBLAS/cuSolver [22], and AMD hipBLAS/hipSolver [1]. In a heterogeneous system, the kernel layer provides the appropriate BLAS/LAPACK library kernels based on the task mapping decided by the runtime layer during execution. Thus, the kernel layer's API is portable across architectures and BLAS/LAPACK libraries, thereby facilitating the implementation of portable algorithms that use BLAS/LAPACK kernels. The effectiveness of the kernel layer has been demonstrated on different CPUs (e.g., Intel Xeon Skylake, AMD 249 EPYC 7763, Qualcomm Snapdragon ARM cores) and GPUs (e.g,. NVIDIA A100, AMD MI100, Qualcomm Snapdragon Adreno) [19].

This kernel layer also provides a simple and standard API for each BLAS/LAPACK operation (Figure 4). This API uses IRIS memory objects across different IRIS tasks to save the data transfers so the library can better interoperate with other IRIS tasks or applications/libraries. It also handles IRIS task kernel creation by setting the appropriate kernel parameters. By doing so, the kernel layer exposes task-level abstraction to the upper layer of MatRIS so that algorithms using those tasks can be developed. The upper layer provides IRIS memory objects (e.g., DMEM objects) in place of host pointers, and the kernel layer links those IRIS memory objects as kernel parameters. The runtime layer handles data transfers.

The kernel layer leverages IRIS macros, which are used to easily convert any function call to IRIS tasks. The macros (*IRIS_TASK _NO_DT* in Figure 4) are used when creating the IRIS tasks with host wrapper codes (i.e., boilerplate code required for calling the device-specific kernel functions). These wrapper codes are needed for the interoperability between the kernel layer API, kernel codes, vendor/open-source BLAS/LAPACK libraries, and the IRIS runtime. Other macros (e.g., *IN_TASK*, *OUT_TASK*, *IN_OUT_TASK*) are used to indicate whether the memory parameters are input, output, or both. Finally, the *PARAM* macro is used for scalar parameters.

In summary, the kernel layer is a unified MatRIS interface for optimized kernels from different libraries. It also creates/defines tasks and their parameters to be used by the runtime layer. By exposing this API, the kernel layer makes architecture-agnostic algorithm creation possible. While the runtime layer provides heterogeneous functionalities, the kernel layer enables seamless kernel portability.

2.4 Algorithm Layer: Unified API for Portable Dense and Sparse Algorithms

The top/algorithm layer of MatRIS provides the final level of abstraction by using the abstraction provided by the kernel and runtime layers. The algorithm layer provides an API for different tiled algorithms (e.g., GETRF, POTRF, GEMM, TRSM). At this level, a graph of tasks for an algorithm is specified by using the API from the kernel layer, thereby making MatRIS completely agnostic to vendor libraries and the underlying architecture. For this reason, any addition of a new architecture or library at the kernel or runtime layer does not require modification at the algorithm level. This agnosticism enables the abstraction for portability and heterogeneity. Moreover, the algorithm layer provides the option for including algorithm-aware device mapping derived from performance models and analysis. A seemingly serial implementation of an algorithm in the algorithm layer provides performance portability and heterogeneity with the option of introducing newer architectures or libraries at different layers of the software stack. Using these capabilities, MatRIS opens the door for future/diverse heterogeneous architectures.

2.4.1 Transparent Tiling at the Algorithm Layer. Compared with tiled algorithms for homogeneous compute units, the heterogeneous tiled algorithms require tiling specifications to manage heterogeneous device memories and task creation. The MatRIS algorithm layer provides a tiling mechanism that binds a runtime-specific heterogeneous device memory object handler to each tile [20]. The tiling feature provides iterators and index access support for writing tiled algorithms. It also alleviates the challenges and burdens associated with writing tiled algorithms by handling the heterogeneous tiled memory objects, which are then used to invoke the API from the kernel layer. The tiling feature supports 1D (traditional flat-to-tile or tile-to-flat) or 2D data copy during execution; therefore, an application only needs to provide the host pointer of the matrix, and MatRIS does the tiling and heterogeneous memory management (i.e., linking with DMEM objects) automatically. Using the MatRIS tiling feature for the tiled LU factorization algorithm is discussed below (also see Figure A1).

2.4.2 Example: Tiled Algorithms for LU Factorization. Although tiled algorithms are a popular strategy for enhancing the locality of data access to enable effective use of shared memory [35], MatRIS uses tiling to decompose the matrices to utilize all the processors in a heterogeneous system when processing large matrix sizes. Tiling occurs before task and dependency creation in the algorithm layer. While creating the graph, the algorithm layer associates memory chunks for different tiles to different tasks.

Using LU factorization, we demonstrate the capability of the MatRIS algorithm layer. LU factorization plays a key role in many computational science applications and is computationally expensive. Therefore, implementing an LU factorization by using MatRIS could facilitate performance portability on modern heterogeneous systems, and this is one of the goals of this research. Decomposing a matrix A into lower- and upper-triangular matrices (i.e., the LU factorization) is used to easily solve systems of linear equations:

$$Ax = LUx = B. \tag{1}$$

Decomposing a matrix into tiles is a common strategy to parallelize this operation. Defining kernels, memory tiles, and dependencies by using task-level programming makes such an implementation possible [11, 25, 26].

LU Factorization on Dense Matrices. The LU factorization on a tiled dense matrix (Figure 5) consists of (i) factorizing the first tile of the diagonal to obtain the L (dark-green) and U (light-green) matrices of the tile, (ii) computing several TRSMs (light-blue) by using the L matrix for the corresponding row and the U matrix for the corresponding column, and (iii) computing the so-called *update* step (dark-blue) by multiplying (i.e., GEMM) the result of the set of TRSMs and updating the tiles in the rest of the matrix. We compute the next tile of the diagonal and the next two steps until the entire matrix is computed. A MatRIS-created DAG for a 4×4 decomposition for tiled LU is shown in Figure 6.

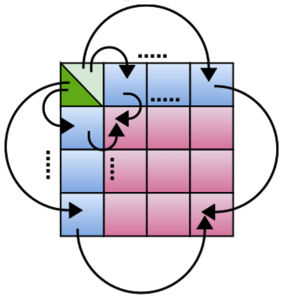

Figure 5: Tiled LU decomposition scheme [26].

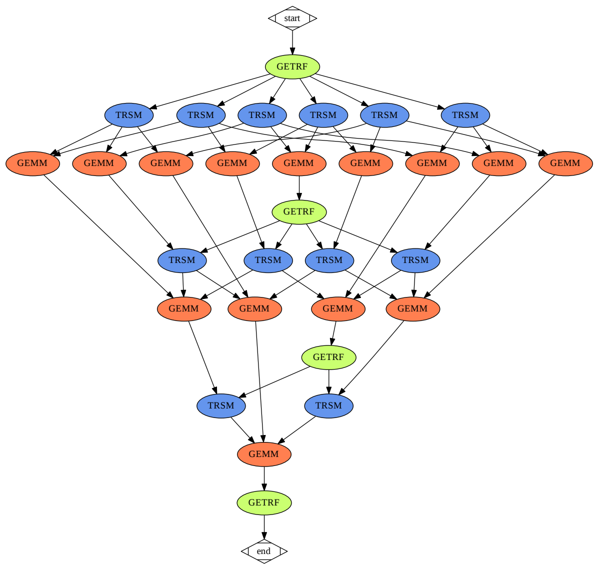

Figure 6: MatRIS-generated DAG for dense LU factorization when using 4×4 tile decomposition. Green ellipses are GETRF, blue ellipses are TRSM, and orange ellipses are GEMM.

Although the state-of-the-art routine for LU factorization involves pivoting, we considered a non-pivoting version for two reasons: (i) the pivoting is not necessary on well-conditioned matrices, and (ii) we want to analyze the performance of the proposed optimizations without the influence of pivoting on the performance analysis. Additionally, although using pivoting to solve systems of linear equations is commonly accepted, we found multiple problems in which the matrices were well conditioned, which made expensive operations such as pivoting unnecessary. For this reason, multiple implementations in reference libraries do not use such a technique. Examples include PLASMA (Parallel Linear Algebra Software for Multicore Architectures) [11], LASs [26], Intel's MKL, NVIDIA's cuSolver [30] and cuSparse [31], FISHPACK [33], and SuperLU [9].

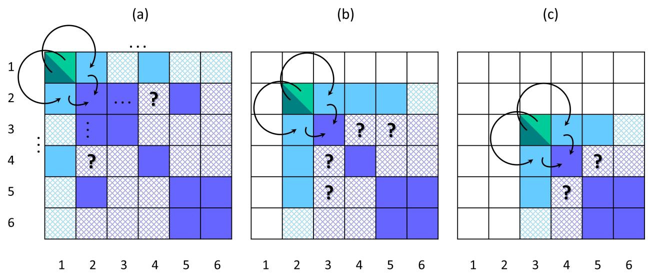

Figure 7: Example and schema of the LU algorithm for tiled sparse matrices [27].

LU Factorization on Sparse Matrices. The algorithm used in MatRIS to compute the LU factorization on general sparse matrices is based on the SparseLU library [27], a task-based library implemented in OpenMP for multicore CPUs. Like the algorithm for dense matrices, MatRIS divides the sparse matrices into tiles. This algorithm has proven effective for scalability and high performance at the cost of some zero-elements computation, and it provides faster computation overall than other state-of-the-art approaches [27]. Unlike other solutions [9], this algorithm does not need any preprocessing symbolic phase because memory management is handled concurrently with computation. Furthermore, the algorithm's use of tasks is completely transparent and determined by the runtime. This removes the burden of determining the target and size of the tasks. Another important benefit of this algorithm is that it only uses level-3 BLAS routines, which helps obtain higher performance on parallel processors.

One important difference between sparse and dense algorithms is how the memory is managed. When a tile contains only zeros, it is stored as NULL in our algorithm. These tiles are represented as the shaded squares in Figure 7 and are handled differently than the dense LU steps. For example, for LU's second step, in which TRSM is calculated on the corresponding row and column, no computation needs to be completed for null tiles, so they are left as-is. In LU's third step, it is possible for memory allocation to be required for a null tile before GEMM is completed using the corresponding TRSM set. For example, take tile (2,4) in Figure 7a. Here, this tile is null, but its corresponding row TRSM at (1,4) and column TRSM at (2,1) are not null. The multiplication (GEMM) of the corresponding row and column TRSM tiles produces new information. The result of this multiplication must be stored in tile (2,4) (Figure 7), so memory is allocated for that tile. Such dynamic allocation is commonly known as *fill-in*. This process repeats for each step of LU factorization along the diagonal. The tiles for which memory is allocated are marked with *?* in Figure 7. When one or both tiles in the TRSM set are null, the result of GEMM will produce no new information, so the tiles are left as-is. Using the aforementioned strategy, memory for the matrix data structure is only allocated as needed. Also, we can save computation time by avoiding any unnecessarily preprocessing [9].

3 PERFORMANCE ANALYSIS

For this work, we used three heterogeneous systems with CPUs and GPUs from different manufacturers located in different HPC and cloud computing resources at Oak Ridge National Laboratory (ORNL). Table 1 shows the hardware configurations, architectures,

Table 1: Heterogeneous systems used in this research.

System	Summit	Frontier	CADES cloud node
GPUs	Total 6 GPUs 6× NVIDIA V100	Total 4/8 GPUs/GCDs 4× AMD MI250X (each contains two GCDs)	Total 8 GPUs 4× NVIDIA A100 4× AMD MI100
CPU	POWER9, 42 cores	AMD EPYC 7A53, 64 cores	AMD EPYC 7763, 128 cores
Compiler	GNU-9.4.0	GNU-8.5.0	GNU-8.5.0
CUDA and ROCm versions	CUDA-11.7	CUDA-11.7 and ROCm-5.1.0	CUDA-11.7 and ROCm-5.1.2
Math libraries	cuBLAS and cuSOLVER	hipBLAS	cuBLAS, cuSOLVER, and hipBLAS

compilers, software stacks, and LAPACK/BLAS libraries that were used. For profiling and tracing, the native capabilities of the IRIS runtime were used. We demonstrate three cases in this section: (i) portability and multi-device heterogeneity for dense LU factorization with MatRIS, (ii) similar analysis for sparse LU factorization with MatRIS, and (iii) a comparison of MatRIS with vendor and open-source libraries. In these experiments, we demonstrate performance portability in the systems listed in Table 1. For both the HPC and cloud systems, MatRIS requires no changes in the software stack, thereby demonstrating portability and multi-device heterogeneity. Once built, the same MatRIS application can run in different hardware configurations available in a system by selecting an environment variable (cpu, `nvidia-gpu`, `cpu-nvidiaGPU`, or `nvidiaGPU-amdGPU`, `cpu-nvidiaGPU-amdGPU`, and other possible combinations). MatRIS executes the application graph and selects the appropriate kernel and their devices at runtime by respecting the scheduler.

3.1 Dense LU Factorization

Figure 8 illustrates the time consumed by our reference library (LaRIS [21]) and the MatRIS library. For Frontier and CADES, we used a 32,678 × 32,678 matrix and a 2,048 × 2,048 tile size, with a total of 1,496 tasks. For Summit, we used a 16,384 × 16,384 matrix and a 1,024 × 1,024 tile size, with a total of 1,496 tasks. We used a smaller matrix size on Summit because of its GPU memory limitation. All operations are computed in double precision. In Figure 8, the two dotted lines represent the execution time of MatRIS and LaRIS. The green-shaded area shows the speedup of MatRIS over LaRIS (LaRIS-time/MatRIS-time), and the red-shaded area shows the scalability of MatRIS (single-gpu-time-MatRIS/multiple-gpu-time-MatRIS). As shown, MatRIS can achieve significant acceleration for the LaRIS library by providing speedups close to 8× on Frontier, 7× on CADES, and 5× on Summit.

Different features of the MatRIS software stack have different impacts on performance depending on the system used. For example, using an optimized memory management strategy (Section 2) has a more substantial impact on Frontier, where it achieves a significant reduction in time and better scalability. However, on CADES, although the execution time is reduced considerably, we did not see significant improvement in terms of scalability in the beginning (not shown in the figure). However, after we applied our scheduler, we saw a more significant impact on CADES than on Frontier (Figure 8). Also, on Summit, the benefit of the optimizations (i.e., speedup) provided by MatRIS over LaRIS is lower than for the other two systems when increasing the number of GPUs. However, we see the opposite trend on Frontier. This is caused by the differences in the software stacks, hardware, and connectivity of each system.

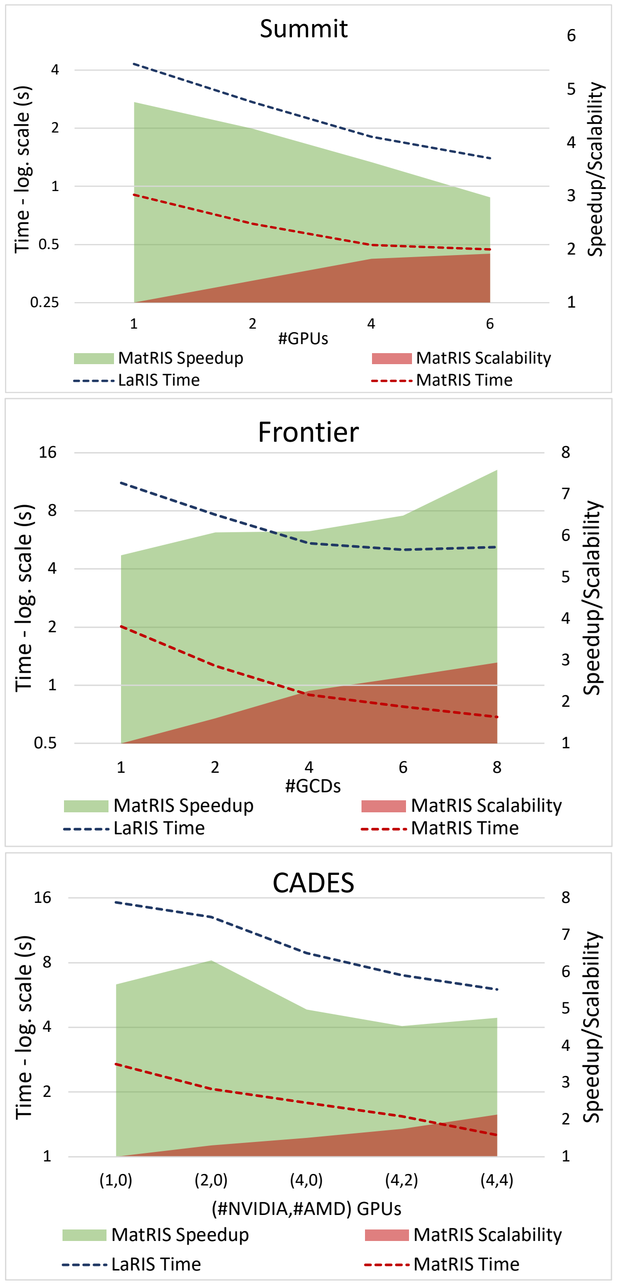

Figure 8: Strong scaling of LaRIS and MatRIS libraries for dense LU factorization on different heterogeneous systems.

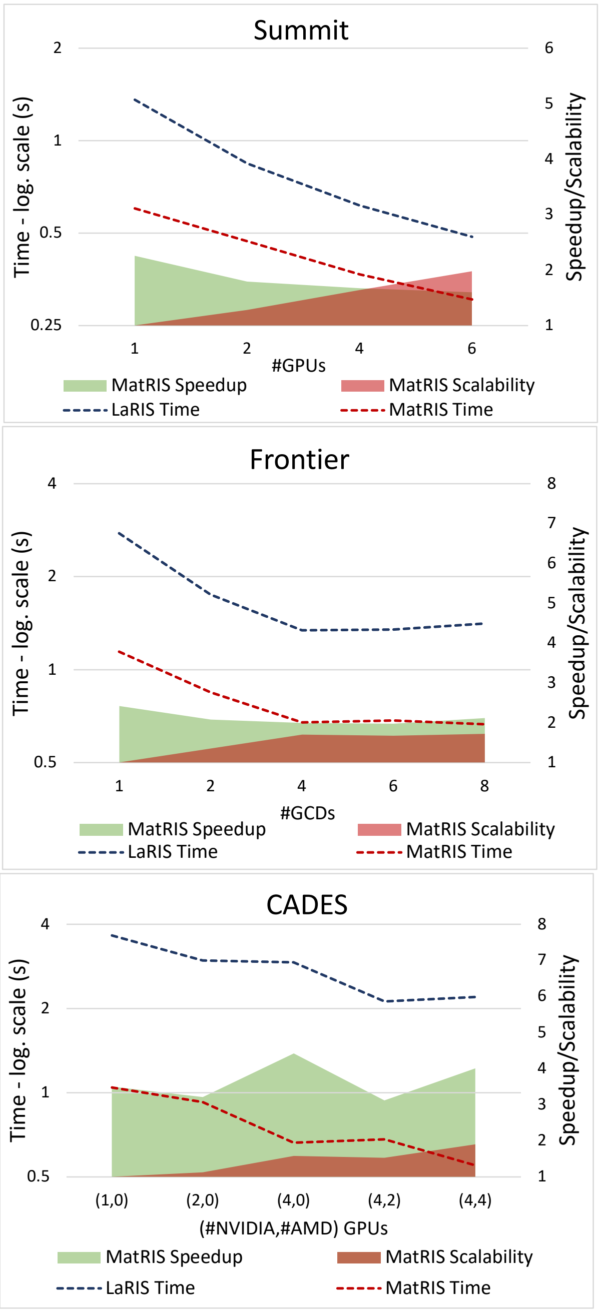

Figure 9: Strong scaling of LaRIS and MatRIS libraries for sparse LU factorization on different heterogeneous systems.

The main contributor of such a high acceleration is an important reduction in the number of memory transfers (Table 2) thanks to the efficient memory management strategy and scalable scheduler.

The numbers in Table 2 correspond to the tests using six GPUs on Summit and eight GCDs/GPUs on the other two systems, Frontier and CADES. The first row represents the memory transfer in LaRIS, which is the same for all the systems even though the matrix size for Summit is smaller. The same matrix decomposition is used for all the systems, and this created the same number of tasks. However, for MatRIS, an impressive reduction is observed. MatRIS not only reduces the number of memory transfers by more than 4×, it also uses faster connections (e.g., D2D communication) when possible. The bandwidth between GPUs of the same type is usually faster than the CPU-GPU bandwidth. For example, we found a bidirectional GPU-GPU bandwidth of 50GB/s + 50GB/s on Frontier, whereas the CPU-GPU bandwidth is 36GB/s + 36GB/s. This is different on Summit, where the CPU-GPU and GPU-GPU bandwidths are the same (50GB/s + 50GB/s). We observed ~25GB/s for CPU-GPU (for both NVIDIA and AMD GPUs) and ~36GB/s for GPU-GPU bandwidth for the CADES machine.

Notably, the number of D2D transfers is higher for CADES than for Frontier, although both systems contain the same number of accelerators. Once again this is due to the differences between the systems. In Frontier, all the GPUs/GCDs are connected to each other, whereas in CADES, we have two separate sets of GPUs (NVIDIA and AMD GPUs). Each NVIDIA GPU has a direct connection with the rest of the NVIDIA GPUs but is not directly connected to the AMD GPUs. The same relationship applies for AMD GPUs. So, for example, when one task executed on one NVIDIA GPU needs data or a tile located in the memory of an AMD GPU, this data or tile must be copied back to the host (CPU) memory and then sent to one of the NVIDIA GPUs. Therefore, fewer D2D communications and more H2D and D2H communications are observed on CADES than on Frontier. This difference in memory transfer has consequences for performance and/or scalability. As we can see in Figure 8, the scalability on Frontier is about 1.3× higher. Unlike CADES and Frontier, which both have eight accelerators each, Summit has only six GPUs per node, which means fewer memory transfers.

3.2 Sparse LU Factorization

We conducted our analysis on a synthetically generated sparse matrix (Figure 10). This approach enables us to perform fair, consistent, and comprehensive testing by replicating the conditions of real-world matrices. We used the SuiteSparse Matrix Collection to obtain the appropriate parameters for generating the matrix. The widely used SuiteSparse Matrix Collection contains over 2,800 sparse matrices collected from a variety of applications, including fluid dynamics, structural problems, and circuit simulation [8]. In our testing, we used a 32,678 × 32,678 matrix for Frontier and CADES and a 16,384 × 16,384 matrix for Summit. These are the same matrix sizes used for the dense LU factorization analysis, and they sufficiently represent real conditions without requiring unnecessarily long computation times for testing [27]. For sparsity, we tried

Table 2: Number and types of memory transfers for LaRIS and MatRIS on Summit (6× GPUs), Frontier (8× GCDs), and CADES (8× GPUs) when computing LU factorization on a 32,678 × 32,678 dense matrix for Frontier and CADES, and a 16,384 × 16,384 dense matrix for Summit.

System	Summit				Frontier				CADES			
Transfer	H2D	D2H	D2D	total	H2D	D2H	D2D	total	H2D	D2H	D2D	total
LaRIS	4,219	1,497	0	5,716	4,219	1,497	0	5,716	4,219	1,497	0	5,716
MatRIS	259	257	734	1,250	259	257	854	1,370	567	395	426	1,388

Table 3: Number and types of memory transfers for LaRIS and MatRIS on Summit (6× GPUs), Frontier (8× GCDs), and CADES (8× GPUs) when computing LU factorization on a 32,678 × 32,678 sparse matrix with a density factor of $\approx 3 \times 10^{-4}$ for Frontier and CADES, and a 16,384 × 16,384 matrix with the same density for Summit.

System	Summit				Frontier				CADES			
Transfer	H2D	D2H	D2D	total	H2D	D2H	D2D	total	H2D	D2H	D2D	total
LaRIS	1,055	397	0	1,452	1,064	402	0	1,466	1,064	402	0	1,466
MatRIS	154	121	173	448	201	127	240	568	292	155	149	596

to match the patterns found in matrices that are symmetric, square, non-binary, have entries along the matrix diagonal, and are either real-symmetric-assembled or integer-symmetric-assembled [8]. For density (calculated as $nnz/(M \times M)$, where nnz means the number of non-zeros), we used a relatively low density ($\approx 3\times10^{-4}$). Because most of the sparse matrices fall into this range, these parameters represent real-world conditions [8, 27]. We use the same tile size that we use in the dense LU factorization analysis. Given the dispersity of the matrix used, 401 tasks were created for Frontier and CADES, and 396 tasks were created for Summit.

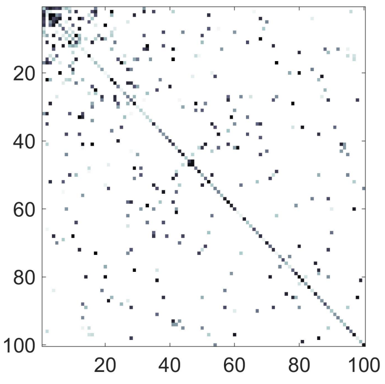

Figure 10: Example of the synthetically generated matrix.

Figure 9 illustrates the performance in terms of time (in seconds using a logarithmic scale) and speedup/scalability. Computing the sparse LU factorization is more challenging and complex in terms of scheduling and scalability. This is clear when looking at the scalability of both libraries. In this case, MatRIS achieves considerable acceleration over LaRIS. Also, MatRIS provides scalability by providing better performance on more GPUs. This is particularly important given the low density of *nnz* used in this analysis.

As expected, we found fewer memory transfers for the sparse LU factorization (Table 3), which minimizes the potential benefit of having a more efficient memory management strategy and a scalable scheduler. Also, the reduction in memory transfers achieved by the MatRIS library, although very important, is lower than the one achieved in the dense LU case (Table 2). The reduction of memory transfer is about 2.6× on Frontier and CADES and about 3.2× on Summit when all GPUs are used. All of this impacts performance, which means an acceleration of about 2× on Summit and Frontier and close to 5× on CADES. We observed a lower speedup compared to the dense cases in all the heterogeneous systems. Especially on Frontier and CADES, we see MatRIS's time reaching a seemingly flat line when using more than four GPUs. We correlate this behavior with less parallelism because the sparse version only creates ~400 tasks compared to the 1,496 tasks created for the dense case.

Table 4: Comparison with state of the art on a Summit node. LU factorization for a 16,384 × 16,384 matrix. The unit of measure is GFLOPS, and the best performance for each GPU configuration is in bold.

#GPUs	MatRIS	cuSolverMg	StarPU
1	3,240	3,714	**3,856**
2	4,580	**4,884**	4,067
4	**5,901**	5,101	2,874
6	**6,224**	4,526	X,XXX

3.3 Comparison

Table 4 shows a performance comparison between MatRIS, NVIDIA's cuSolverMg library, and the StarPU runtime. This comparison used a single Summit node for an LU factorization (non-pivoting). The best-performing library for each configuration is listed in bold text.

We ensured the matrix was diagonally dominant, so pivoting was not computed. We used the recently released cuSolverMg library,[2] which is a novel NVIDIA library with support for multi-GPU hardware. The algorithms implemented in this library use a 1D column block-cyclic layout for matrix decomposition, which is different from the one used in MatRIS. The code used for the performance analysis is publicly accessible.[3] We also used the non-pivoting LU factorization from StarPU, and the algorithm is similar to the one used in MatRIS.

For Summit's NVIDIA GPUs, MatRIS demonstrated competitive performance versus the NVIDIA cuSolverMg library on one GPU and two GPUs by reaching more than the 90% of the NVIDIA library's performance. Moreover, the NVIDIA library did not provide good scalability when using four GPUs, whereas MatRIS provided better performance and scalability. The better scalability from MatRIS is more evident when using more than two GPUs, a scenario in which the NVIDIA library provided flat scalability.

Unlike the cuSolverMg library, MatRIS continued reducing the execution time (hence better GFLOPS) as the the number of GPUs increased, until all available GPUs were used. Also, MatRIS provided better performance than the NVIDIA library when using four (speedup of 1.15×) and six (speedup close to 1.4×) GPUs. StarPU provided the best performance for a single GPU; however, performance declined as the number of GPUs increased. We could not generate the performance number with six GPUs using StarPU. We also investigated the MAGMA (Matrix Algebra on GPU and Multicore Architectures) library, which has separate implementations for CPU, GPU, and multi-GPU LU factorization. The multi-GPU version did not achieve great performance on Summit (possibly due to the differences in the algorithm); hence, we did not include MAGMA in the comparison.

[2]https://docs.nvidia.com/cuda/cusolver/index.html
[3]https://github.com/NVIDIA/CUDALibrarySamples/tree/master/cuSOLVER/MgGetrf

Also, MatRIS provided similar performance to the AMD library when using one AMD GPU on Frontier. For analysis of the AMD GPUs, we used the hipBLAS library.Notably, AMD BLAS/LAPACK libraries do not provide support for multi-GPU configurations.

Although the primary motivation of this work is not to provide better or competitive performance compared to existing libraries but rather to provide additional capabilities for mixed-GPU and multi-GPU computing, the efficiency of MatRIS *is remarkable* when compared with existing highly optimized vendor libraries.

4 RELATED WORK

Performance portability is becoming one of the most important challenges in the exascale and extreme heterogeneity era. The growing importance of C++ template metaprogramming libraries is one example [3, 23, 29]. These libraries can build different binaries from a single source code to target different architectures. Notably, however, they cannot use more than one architecture at a time.

Since OpenMP 4.0, it is possible to use GPU offloading in OpenMP codes. Valero-Lara et al. [28] used OpenMP 4.5 to implement a heterogeneous version of the TRSM level-3 BLAS routine and achieved good performance on one node of ORNL's Summit supercomputer, using one CPU and one GPU.

Most vendor or open-source math libraries are optimized for just one architecture. For example, PLASMA [12] is a dense linear algebra reference library based on OpenMP. PLASMA parallelizes BLAS and LAPACK level operations that target homogeneous multicore and multisocket CPU platforms. Like other libraries (e.g., Chameleon, which is based on the StarPU runtime, and LASs [7, 24–26], which is based on the OmpSs runtime), PLASMA uses tiled algorithms to distribute the workload among the cores in the platform by using task-based programming. Other relevant linear algebra libraries that implement dense, sparse, or both linear algebra operations include libFLAME [15], Intel MKL, and OpenBLAS [34].

Another example in linear algebra is ATLAS [36], which provides a complete collection of BLAS kernels and LAPACK operations and delivers high performance thanks to its autotuning approach. ATLAS exploits low-level features (e.g., the size of the different memory hierarchy levels) to customize required parameters (e.g., the block size) and consequently makes better use of the resources to improve performance on multicore CPU architectures. Other approaches try to adapt the number of computations to the resources available on the platform; however, this adaptation is not automatic and/or its implementation requires major changes in the code [6, 10, 32].

Another example is the MAGMA [14] open-source math library for BLAS and LAPACK operations on heterogeneous systems. It includes some heterogeneous implementations based on tiled algorithms that use NVIDIA's cuBLAS and AMD's hipBLAS math libraries. These implementations statically run the LAPACK operations of the tiled algorithms on the CPU, while most of the BLAS operations are run on one GPU.

All of the examples referenced above make improvements in specific scenarios. However, they also exhibit at least one of four drawbacks: (i) the support is very limited, only supporting one type of architecture or one CPU + GPU (or one vendor's GPU), (ii) the solution is not often portable; (iii) the solution is not automatically achieved (auto-tuning), or (iv) adding a new architecture requires considerable change. In contrast, the present work presents a multi-level abstraction for portability and multi-device heterogeneity that alleviates these challenges.

5 CONCLUSIONS

We presented MatRIS, an extensible multilevel abstraction for a productive and performance-portable programming solution that enables BLAS/LAPACK codes with transparent tiling and task/tile mapping and reaches good scalability on different multi-device and heterogeneous systems found in contemporary HPC and cloud computing environments. MatRIS accomplishes this as a fully architecture-agnostic library that decouples algorithm definitions from hardware-specific details, which makes MatRIS portable and capable of using all available compute devices in heterogeneous systems. MatRIS's performance reports significant acceleration over the LaRIS reference library thanks to an impressive reduction in memory transfers, effective memory management, and a scalable scheduler. MatRIS also reports better scalability and competitive or better performance in our comparison with vendor libraries.

ACKNOWLEDGMENTS

This research used resources of the Oak Ridge Leadership Computing Facility at ORNL, which is supported by the US Department of Energy's (DOE's) Office of Science under Contract No. DE-AC05-00OR22725.This work is funded, in part, by Bluestone, a X-Stack project in the DOE Advanced Scientific Computing Office with program manager Hal Finkel. This manuscript has been authored by UT-Battelle LLC under contract no. DE-AC05-00OR22725 with the US Department of Energy. The publisher, by accepting the article for publication, acknowledges that the US 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 US 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] AMD. 2022. hipBLAS, the Basic Linear Algebra Subroutine library. https://github.com/ROCmSoftwarePlatform/hipBLAS [Online; accessed 6-July-2022].

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

[3] David Beckingsale, Richard D. Hornung, Tom Scogland, and Arturo Vargas. 2019. 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*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 455–456. https://doi.org/10.1145/3293883.3302577

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

[5] Anthony M. Cabrera, Seth Hitefield, Jungwon Kim, Seyong Lee, Narasinga Rao Miniskar, and Jeffrey S. Vetter. 2021. 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, Waltham, MA, USA, September 20-24, 2021.* IEEE, 1–7. https://doi.org/10.1109/HPEC49654.2021.9622794

[6] Sandra Catalán, José R. Herrero, Enrique S. Quintana-Ortí, Rafael Rodríguez-Sánchez, and Robert A. van de Geijn. 2016. A Case for Malleable Thread-Level Linear Algebra Libraries: The LU Factorization with Partial Pivoting. *CoRR* abs/1611.06365 (2016). arXiv:1611.06365 http://arxiv.org/abs/1611.06365

[7] Sandra Catalán, Tetsuzo Usui, Leonel Toledo, Xavier Martorell, Jesús Labarta, and Pedro Valero-Lara. 2020. Towards an Auto-Tuned and Task-Based SpMV (LASs Library). In *OpenMP: Portable Multi-Level Parallelism on Modern Systems - 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22-24, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12295)*, Kent F. Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg (Eds.). Springer, 115–129. https://doi.org/10.1007/978-3-030-58144-2_8

[8] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1:1–1:25. https://doi.org/10.1145/2049662.2049663

[9] James Weldon Demmel, John R. Gilbert, and Xiaoye S. Li. 1999. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM J. Matrix Anal. Appl.* 20, 4 (1999), 915–952. https://doi.org/10.1137/S0895479897317685

[10] Jack Dongarra, Sven Hammarling, Nicholas J. Higham, Samuel D. Relton, Pedro Valero-Lara, and Mawussi Zounon. 2017. The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems. *Procedia Computer Science* 108 (2017), 495 – 504. https://doi.org/10.1016/j.procs.2017.05.138 International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[11] Jack J. Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim YarKhan, Maksims Abalenkovs, Negin Bagherpour, Sven Hammarling, Jakub Sístek, David Stevens, Mawussi Zounon, and Samuel D. Relton. 2019. PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP. *ACM Trans. Math. Softw.* 45, 2 (2019), 16:1–16:35. https://doi.org/10.1145/3264491

[12] Jack J. Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim YarKhan, Maksims Abalenkovs, Negin Bagherpour, Sven Hammarling, Jakub Sístek, David Stevens, Mawussi Zounon, and Samuel D. Relton. 2019. PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP. *ACM Trans. Math. Softw.* 45, 2 (2019), 16:1–16:35. https://doi.org/10.1145/3264491

[13] Jack J. Dongarra and Piotr Luszczek. 2011. ScaLAPACK. In *Encyclopedia of Parallel Computing*, David A. Padua (Ed.). Springer, 1773–1775. https://doi.org/10.1007/978-0-387-09766-4_151

[14] Mohammed A. Al Farhan, Ahmad Abdelfattah, Stanimire Tomov, Mark Gates, Dalal Sukkari, Azzam Haidar, Robert Rosenberg, and Jack J. Dongarra. 2020. MAGMA templates for scalable linear algebra on emerging architectures. *Int. J. High Perform. Comput. Appl.* 34, 6 (2020). https://doi.org/10.1177/1094342020938421

[15] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. 2001. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.* 27, 4 (Dec. 2001), 422–455. https://doi.org/10.1145/504210.504213

[16] Intel. 2022. The Intel Math Kernel Library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html?s=Newest [Online; accessed 6-July-2022].

[17] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S. Vetter. 2021. 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.* IEEE, 1–8. https://doi.org/10.1109/HPEC49654.2021.9622873

[18] Jannis Klinkenberg, Philipp Samfass, Michael Bader, C. Terboven, and Matthias S Müller. 2020. CHAMELEON: reactive load balancing for hybrid MPI+ OpenMP task-parallel applications. *J. Parallel and Distrib. Comput.* 138 (2020), 55–64.

[19] Narasinga Rao Miniskar, Alaul Haque Monil Mohammad, alero-Lara Pedro, Frank Liu, and Jeffrey S Vetter. 2022. 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.

[20] Narasinga Rao Miniskar, Mohammad Alaul Haque Monil, Pedro Valero-Lara, Frank Liu, and Jeffrey S Vetter. 2023. Tiling Framework for Heterogeneous Computing of Matrix-Based Tiled Algorithms. (2023).

[21] Mohammad Alaul Haque Monil, Narasinga Rao Miniskar, Frank Y. Liu, Jeffrey S. Vetter, and Pedro Valero-Lara. 2022. LaRIS: Targeting Portability and Productivity for LAPACK Codes on Extreme Heterogeneous Systems by Using IRIS. In *IEEE/ACM Redefining Scalability for Diversely Heterogeneous Architectures Workshop, RSDHA@SC 2022, Dallas, TX, USA, November 13-18, 2022.* IEEE, 12–21. https://doi.org/10.1109/RSDHA56811.2022.00007

[22] NVIDIA. 2022. cuBLAS, the CUDA Basic Linear Algebra Subroutine library. https://docs.nvidia.com/cuda/cublas/index.html [Online; accessed 6-July-2022].

[23] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Q. Dang, Nathan D. Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Trans. Parallel Distributed Syst.* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

[24] Pedro Valero-Lara, Diego Andrade, Raül Sirvent, Jesús Labarta, Basilio B. Fraguela, and Ramon Doallo. 2019. A Fast Solver for Large Tridiagonal Systems on Multi-Core Processors (Lass Library). *IEEE Access* 7 (2019), 23365–23378. https://doi.org/10.1109/ACCESS.2019.2900122

[25] Pedro Valero-Lara, Sandra Catalán, Xavier Martorell, and Jesús Labarta. 2019. 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.* IEEE, 25–32. https://doi.org/10.1109/EMPDP.2019.8671545

[26] Pedro Valero-Lara, Sandra Catalán, Xavier Martorell, Tetsuzo Usui, and Jesús Labarta. 2020. sLASs: A fully automatic auto-tuned linear algebra library based on OpenMP extensions implemented in OmpSs (LASs Library). *J. Parallel Distributed Comput.* 138 (2020), 153–171. https://doi.org/10.1016/j.jpdc.2019.12.002

[27] Pedro Valero-Lara, Cameron Greenwalt, and Jeffrey S. Vetter. 2022. 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.* IEEE, 25–31. https://doi.org/10.1109/IA356718.2022.00010

[28] Pedro Valero-Lara, Jungwon Kim, Oscar Hernandez, and Jeffrey S. Vetter. 2021. 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, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13098)*, Ricardo Chaves, Dora B. Heras, Aleksandar Ilic, Didem Unat, Rosa M. Badia, Andrea Bracciali, Patrick Diehl, Anshu Dubey, Oh Sangyoon, Stephen L. Scott, and Laura Ricci (Eds.). Springer, 445–455. https://doi.org/10.1007/978-3-031-06156-1_35

[29] Pedro Valero-Lara, Seyong Lee, Marc González Tallada, Joel E. Denny, and Jeffrey S. Vetter. 2022. KokkACC: Enhancing Kokkos with OpenACC. In *9th Workshop on Accelerator Programming Using Directives, WACCPD@SC 2022, Dallas, TX, USA, November 13-18, 2022.* IEEE, 32–42.

[30] Pedro Valero-Lara, Ivan Martínez-Perez, Raúl Sirvent, Xavier Martorell, and Antonio J. Peña. 2017. NVIDIA GPUs Scalability to Solve Multiple (Batch) Tridiagonal Systems Implementation of cuThomasBatch. In *Parallel Processing and Applied Mathematics - 12th International Conference, PPAM 2017, Lublin, Poland, September 10-13, 2017, Revised Selected Papers, Part I.* 243–253.

[31] Pedro Valero-Lara, Ivan Martínez-Pérez, Raúl Sirvent, Xavier Martorell, and Antonio J. Peña. 2018. cuThomasBatch and cuThomasVBatch, CUDA Routines to compute batch of tridiagonal systems on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience* 30, 24 (2018).

[32] P. Valero-Lara, I. Martínez-Pérez, S. Mateo, R. Sirvent, V. Beltran, X. Martorell, and J. Labarta. 2018. Variable Batched DGEMM. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP).* 363–367. https://doi.org/10.1109/PDP2018.2018.00065

[33] Pedro Valero-Lara, Alfredo Pinelli, and Manuel Prieto-Matias. 2014. Fast finite difference Poisson solvers on heterogeneous architectures. *Computer Physics Communications* 185, 4 (2014), 1265 – 1272. https://doi.org/10.1016/j.cpc.2013.12.026

[34] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, William Gropp and Satoshi Matsuoka (Eds.). ACM, 25:1–25:12.

[35] R. Clint Whaley. 2011. ATLAS (Automatically Tuned Linear Algebra Software). In *Encyclopedia of Parallel Computing*, David A. Padua (Ed.). Springer, 95–101.

[36] R. Clinton Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1998, November 7-13, 1998, Orlando, FL, USA.* IEEE Computer Society, 38.

[37] Martin Kroeker Zhang Xianyi. 2022. OpenBLAS. https://www.openblas.net/ [Online; accessed 6-July-2022].

A APPENDIX

The pseudocode for MatRIS's tiled LU factorization is provided below.

```
int matris_dgetrf( int target, double *A, int SIZE, int tile_size ) {
    Tiling2D<DTYPE> A_tiling( A, SIZE, SIZE, tile_size, tile_size );
    size_t n_row_tiles = A_tiling.row_tiles_count(), n_col_tiles = A_tiling.col_tiles_count();
    iris_task dgetrf_tasks[n_row_tiles];
    iris_task top_dtrsm_tasks[n_col_tiles];
    iris_task left_dtrsm_tasks[n_row_tiles];
    iris_task dgemm_tasks[n_row_tiles][n_col_tiles];

    size_t step = 0;
    for( auto & a_tile : A_tiling.items( TILE2D_RIGHT_DOWN_TREE_WISE ) ) {
        if ( a_tile.row_tile_index() == a_tile.col_tile_index() ) {
            // Do DGEMM
            step = a_tile.row_tile_index();
            if ( step != 0 ) {
                for( auto & gemm_tile : A_tiling.items( step, step )) {
                    size_t tile_jj = dgemm_tile.row_tile_index();
                    size_t tile_ii = dgemm_tile.col_tile_index();
                    Tile2D<DTYPE> & left_dtrsm_tile = A_tiling.getAt( step-1, tile_ii );
                    Tile2D<DTYPE> & top_dtrsm_tile = A_tiling.getAt( tile_jj, step-1 );
                    if (step-1 == 0) {
                        iris_task dgemm_depend_tasks[] =  { left_dtrsm_tasks[tile_ii], top_dtrsm_tasks[tile_jj] };
                        iris_task_depend( task, 2, dgemm_depend_tasks );
                    }
                    else {
                        iris_task dgemm_depend_tasks[] =  { left_dtrsm_tasks[tile_ii], top_dtrsm_tasks[tile_jj], dgemm_tasks[tile_jj][tile_ii] };
                        iris_task_depend( task, 3, dgemm_depend_tasks );
                    }
                    iris_task dgemm_task;
                    iris_task_create( &dgemm_task );
                    dgemm_tasks[tile_jj][tile_ii] = task;
                    matris_task_dgemm( graph, dgemm_task, target, MATRIS_NO_TRANS, MATRIS_NO_TRANS,
                                       -1.0, left_trsm_tile.IRISMem(), tile_size top_trsm_tile.IRISMem(), tile_size, 1.0, gemm_tile.IRISMem(), tile_size );
                }
            }
            // Do DGETRF
            iris_task_create( &dgetrf_tasks[step] );
            if (step != 0) {
                iris_task dgetrf_depend_tasks[] =  { dgemm_tasks[step][step] };
                iris_task_depend( dgetrf_tasks[step], 1, dgetrf_depend_tasks );
            }
            matris_task_dgetrf( graph, dgetrf_tasks[step], target,
                                tile_size,
                                a_tile.IRISMem(), tile_size );
        }
        else if ( a_tile.row_tile_index() == step ) {
            // Do LEFT DTRSM
            size_t tile_ii = a_tile.col_tile_index();
            Tile2D<DTYPE> & dgetrf_tile = A_tiling.getAt(step, step);
            iris_task_create( &left_dtrsm_tasks[tile_ii] );
            iris_task parent_dgemm_task=NULL;
            if (step != 0)
                parent_dgemm_task = dgemm_tasks[step][tile_ii];
            iris_task dtrsm_depend_tasks[] =  { dgetrf_tasks[step], parent_dgemm_task };
            iris_task_depend( left_dtrsm_tasks[tile_ii], 2, dtrsm_depend_tasks );
            matris_task_dtrsm( graph, left_dtrsm_tasks[tile_ii], target, MATRIS_RIGHT, MATRIS_UPPER,
                               MATRIS_NO_TRANS, MATRIS_NON_UNIT, tile_size, tile_size,
                               1.0, dgetrf_tile.IRISMem(), tile_size, a_tile.IRISMem(), tile_size );
        }
        else if ( a_tile.col_tile_index() == step ) {
            // Do TOP DTRSM
            size_t tile_jj = a_tile.row_tile_index();
            Tile2D<DTYPE> & dgetrf_tile = A_tiling.getAt( step, step );
            iris_task_create( &top_dtrsm_tasks[tile_jj] );
            iris_task parent_dgemm_task = NULL;
            if ( step != 0 )
                parent_dgemm_task = dgemm_tasks[tile_jj][step];
            iris_task trsm_depend_tasks[] =  { dgetrf_tasks[step], parent_dgemm_task };
            iris_task_depend( top_dtrsm_tasks[tile_jj], 2, dtrsm_depend_tasks );
            matris_task_dtrsm( graph, top_trsm_tasks[tile_jj], target, MATRIS_LEFT, MATRIS_LOWER,
                               MATRIS_NO_TRANS,  MATRIS_UNIT, tile_size, tile_size,
                               1.0, dgetrf_tile.IRISMem(), tile_size, a_tile.IRISMem(), tile_size );
        }
    }
    iris_graph_submit(graph);
}
```

Figure A1: LU factorization in MatRIS.

A.1 Artifact Identification

(1) **Contribution and role of the computational artifact(s):** This paper presents MatRIS, a scalable and portable math library abstraction for performance portability and heterogeneity. MatRIS is built on top of the IRIS runtime.

(2) **Description of the computational artifact(s):** The MatRIS software stack includes the IRIS runtime, which is used to implement the IRIS programming model and provide math library–specific functionalities. The MatRIS software stack has three layers of abstraction (see Figure 1 in the manuscript). The bottom layer is the IRIS runtime, which abstracts vendor runtime systems and their APIs. The middle layer (kernel layer) provides optimized kernels, including wrappers for different math libraries optimized for different architectures. This kernel layer also abstracts the calls for different vendor libraries and provides a unified API for BLAS and LAPACK kernels. The top layer of MatRIS provides architecture- and vendor library–agnostic APIs for tiled BLAS and LAPACK functionalities for dense and sparse linear algebra algorithms.

(3) **Contribution of the artifacts toward reproducibility:** All experiments and results were produced with the aforementioned artifacts (MatRIS) and by using the scripts that build IRIS and MatRIS, run MatRIS executables, and collect the results.

(4) **Systems used:** Three systems were used for the experiments: (1) the Summit supercomputer, (2) the Frontier supercomputer, and (3) a node from the CADES cloud environment. All three systems are housed at Oak Ridge National Laboratory. Table 1 in the manuscript provides more information about each system.

A.2 Reproducibility of Experiments

There are three kinds of results presented in the paper: (1) the scalability graphs for dense and sparse LU factorization in Figures 8 and 9, (2) a comparison with NVIDIA's cuSolverMG in Table 4, and (3) a memory transfer comparison in Tables 2 and 3. To better explain the process, please refer to our Google Drive folder, https://drive.google.com/drive/folders/1tXFBOBtyNsUZXq_O0Smj_TjDbm3gCEVA?usp=sharing, which contains the following:

- Snapshots of the MatRIS and IRIS repositories. The IRIS folder is inside the `1.matris` folder. This folder also contains source files and scripts for each system mentioned in Table 1.
- The `2.graphs_sc.xlsx` file contains all graph templates given in Figures 8 and 9. These graphs can be recreated by inserting the data generated from the script.
- The `cusolver_MgGetrf_example.cu` file generates NVIDIA's multiGPU result.

Initial Setup: For every system, there are source files in the `source.X folder` within the `1.matris` folder, where *X* stands in for Summit, Frontier, or CADES. First, the `source.modules.sh` file must be sourced for each machine. Doing so loads the appropriate library to build IRIS and MatRIS.

Initial Building: Before running the scripts, run the `build.X.sh` script located inside the `1.matris` folder (*X* stands in for Summit, Frontier, or CADES). This script will build, install, and source the IRIS and MatRIS library. Using `export IRIS_ARCHS=cuda` (for Summit), `export IRIS_ARCHS=hip` (for Frontier), or `export IRIS_ARCHS=cuda:hip` defines the IRIS architecture. Now, one can test by running the `./install/bin/dgetrf_dominant.x 32768 1 16` command from the `1.matris` folder. Note that each `build.X.sh` script has several options for both dense and sparse matrices. The first `cmake` command is for the LaRIS result, and the second `cmake` command is for the MatRIS result, both of which are reported in Figures 8 and 9 for dense and sparse, respectively.

Generating Figures 8 and 9: The generation process for Figures 8 and 9 is described below.

Workflow: After the initial set up and building described above, one can easily use the scripts located in the `script.X` folders within the `1.matris` folder, where *X* stands in for Summit, Frontier, or CADES. Each folder has two scripts: `dense.sh` and `sparse.sh`. These scripts build MatRIS for the LaRIS option (option 1) and MatRIS option (option 2) and run scalable experiments for multiGPU environments.

Estimation of the Execution Time: Execution time could vary because of the warm-up period for each processor. However, one script should not take more than 2.5 hours.

Expected Results: Running one file creates an output file named `out`, which is then grepped using `grep -i "MATRIX_SIZE" out`. The output of the grep command provides scalability results for execution time for different options (LaRIS and MatRIS) along with a different number of GPUs.

How Graphs are Prepared: The resulting file from the grep command is used to prepare the graphs provided in `2.graphs_sc.xlsx`. For that, use a pivot table in the given Excel file, where the rows are the options, columns are the number of GPUs, and values are the execution time averaged for multiple runs. The result can then be put in dense and sparse sheets in the Excel file. Notably, the option one result is for LaRIS, and option two is for MatRIS. Once the data is input, the graphs are immediately updated. This can be done for all graphs in Figures 8 and 9.

Generating Tables 2 and 3: Tables 2 and 3 are generated by the traces from the IRIS runtime, which provides total memory transfer. For each system, the `build.X.sh` file has two options: dense and sparse, which also provide LaRIS and MatRIS results. Building IRIS and MatRIS by using those scripts would allow the following command: `./install/bin/dgetrf_dominant.x 32768 1 16`. After running, the summary trace is shown at the end, where memory transfer for the total kernel is visible. We took the D2H, H2D, and D2D numbers from that summary and constructed both tables. Generating these should take 30 minutes at most.

Generating Table 4: To generate NVIDIA's multiGPU number reported in Table 4, one must download NVIDIA's example from https://github.com/NVIDIA/CUDALibrarySamples/tree/master/cuSOLVER/MgGetrf. To capture the timing information, please replace the `cusolver_MgGetrf_example.cu` file with the file provided in Google Drive. One must change the number of GPUs to generate the scalability result, which we did manually in the source file. Generating these results should not take more than 30 minutes. For StarPU, one must build StarPU with NVIDIA GPU support. The command `STARPU_NCPUS=0 STARPU_NCUDA=4 STARPU_SCHED=dmdas ./lu_example_double -size 16384 -nblocks 16` provides non-pivoting LU factorization for four GPUs.