LaRIS: Targeting Portability and Productivity for LAPACK Codes on Extreme Heterogeneous Systems by Using IRIS

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

***Abstract*—In keeping with the trend of heterogeneity in high-performance computing, hardware manufacturers and vendors are developing new architectures and associated software stacks (e.g., libraries) to harness the best possible performance from commonly used kernels (e.g., linear algebra kernels). However, kernels tuned for one architecture are not portable to others. Moreover, the coexistence of different architectures in a single node makes orchestration difficult. To address these challenges, we introduce LaRIS, a portable framework for LAPACK functionalities. LaRIS ensures a separation between linear algebra algorithms and vendor-library kernels by using the IRIS run time and IRIS-BLAS library. Such abstraction at the algorithm level makes the implementation completely agnostic to the vendor library and architecture. LaRIS uses the IRIS run time to dynamically select the vendor-library kernel and suitable processor architecture at run time. Through LU factorization, we demonstrate that LaRIS can fully utilize different heterogeneous systems by launching and orchestrating different vendor-library kernels without any change in the source code.**

***Index Terms*—Portability; LAPACK; IRIS run time; LU factorization; Tasking; Tiling; CPU; GPU**

I. Introduction

Heterogeneous computing systems are the go-to solution for increasing computational capability for common high-performance computing (HPC) workloads. Although heterogeneous architectures are now ubiquitous in the pursuit of ever-increasing computational power, they introduce challenges in portability. In the heterogeneous computing paradigm, very different architectures (e.g., CPUs + GPUs) not only coexist in the same system but must also work together seamlessly and efficiently to extract meaningful performance benefits over a homogeneous (i.e., CPU-only) system. Moreover, the wide range of software stacks offered by processor manufacturers, including architecture-specific tuned libraries, adds to this complexity. Therefore, application and library developers must ensure the portability of their code on different architectures to extract the best possible performance of modern heterogeneous computing systems.

In most cases, the current software and technology stacks lack the capability to ensure portability and support extreme heterogeneity. Although we can find some solutions that target portability [17] or heterogeneity [11], it is difficult to find a solution that targets both. IRIS [4], [14] is one of the few solutions with such capacity. IRIS is a task-based run time designed to orchestrate complex directed acyclic graphs (DAGs) on heterogeneous systems. However, such a solution for LAPACK (Linear Algebra PACKage) functionality is uncommon. There are run time systems [2] that provide some functionalities but lack support for diverse heterogeneity for LAPACK codes and have architecture-dependent implementations that limit the development of tiled LAPACK algorithms.

To mitigate these challenges, we present our ongoing efforts to implement LaRIS, a novel *LA*PACK library on top of I*RIS*. LaRIS is a fully architecture-agnostic and portable LAPACK library for scientific applications and strives to utilize the full computational capacity of current and future heterogeneous systems through its automatic tiling and orchestration capabilities. Using tiled LU factorization as a test case, we demonstrate that LaRIS is portable to different heterogeneous systems without any changes in the source code. Moreover, it can utilize all the processing components by launching ≈90,000 tuned kernel tasks from different vendor-provided/open-source BLAS (Basic Linear Algebra Subprograms) libraries.

This research reports the following contributions:

- Improved portability and productivity for LAPACK codes by separating the algorithm description (application details) from the implementation, tasks mapping (hardware features), and vendor library kernels.
- Efficient utilization of different heterogeneous systems with a large number of computing components without changing one line of code.
- Performance study of the LaRIS implementation for LU factorization on three different heterogeneous systems: one NVIDIA DGX-1 system with $1\times$ Intel CPU and $4\times$ NVIDIA V100 GPUs; one representative node of the fastest TOP500[1] supercomputer today, Oak Ridge National Laboratory's (ORNL's) Frontier, with $1\times$ AMD CPU and $4\times$ AMD MI250X GPUs; and one extreme heterogeneous system with $1\times$ AMD CPU and $8\times$ GPUs ($4\times$ NVIDIA V100 GPUs + $4\times$ AMD MI100 GPUs).

The rest of the paper is organized as follows: Section II summarizes important contributions from the literature. Section III describes the different components used (IRIS and IRIS-BLAS) in this work and the algorithm used for the implementation. The details about the implementation are presented in Section V, and the performance study is described in Section VII. Finally, we conclude the paper with final remarks and future directions in Section VIII.

[1]https://www.top500.org/

II. Related Work

Recently, we have seen important progress toward performance portability. Some examples include the C++ template metaprogramming libraries Kokkos [17] and RAJA [3]. These libraries can build different binaries that target different architectures from one source code. However, they cannot use more than one architecture at a time.

Using CPUs and GPUs for HPC has been widely studied [21], [22], [27]. Since OpenMP 4.0, it is possible to use GPU offloading in OpenMP codes. Valero-Lara et al. [23] 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.

Most vendor or open-source math libraries are optimized for just one architecture. One example is PLASMA (Parallel Linear Algebra Software for Multicore Architectures) [9], which is a reference library for dense linear algebra. Based on OpenMP, PLASMA parallelizes BLAS[2] and LAPACK[3] level operations that target homogeneous multicore and multisocket CPU platforms. Like other libraries, such as Chameleon[4] and LASs [6], [18]–[20], 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 types of linear algebra operations include libFLAME [12], Intel MKL, and OpenBLAS [28]. Another example in the linear algebra field is ATLAS [29], which is a software package that provides a complete BLAS collection of kernels and a subset of LAPACK operations that deliver high performance thanks to its autotuning approach. ATLAS exploits low-level features such as 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, the adaptation is not done automatically, or its implementation requires major changes in the code [5], [8], [26]. In the first work, a batched GEMM (level-3 BLAS matrix-matrix multiplication) is proposed to ensure better resource utilization of the platform. The idea is that a GEMM is decomposed into batches that contain thousands of smaller, independent GEMMs to maximize the system's use. In the second work, a similar idea is proposed in which the authors vary the number of operations per batch to balance the workload among the batches. Finally, in the last work, the authors focus on malleability to assign the appropriate amount of resources in each stage of an LU decomposition. It also presents a strategy (early termination) that automatically tunes the block size used in the factorization when the workload between the panel factorization and the update tasks is unbalanced.

[2]http://www.netlib.org/blas
[3]http://www.netlib.org/lapack
[4]https://project.inria.fr/chameleon/

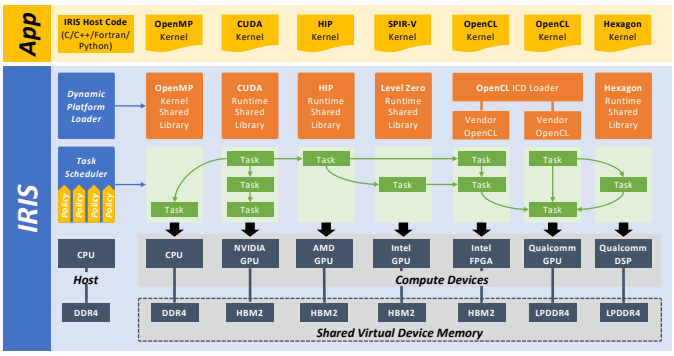


Figure 1: IRIS run-time system for heterogeneous architectures [4], [14].

Another example is MAGMA (Matrix Algebra on GPU and Multicore Architectures) [11], which is an 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. Although some multiGPU implementations can be found, MAGMA focuses on either a single GPU (BLAS) or one CPU and one GPU (LAPACK).

StarPU is a task-based framework for hybrid architectures. It offers a unified offloadable task abstraction named *codelets*, in which programmers can encapsulate existing functions or provide one function for each architecture. StarPU supports OpenMP, OpenCL, and CUDA programming languages for these codelets. HIP support was recently added, and its use is limited to increments of variable and reduction operations. StarPU and IRIS are both task programming models for hybrid architectures. Although StarPU is one of the pioneers of heterogeneous run times, IRIS supports a broader range of accelerators, programming models, and BLAS libraries (e.g., hipBLAS).

All of the examples referenced above make improvements in specific scenarios. However, they also exhibit at least one of three drawbacks: (1) the tuning is not applicable for all the input cases, (2) the solution is not easily implemented, and/or (3) portability is not automatically achieved. In contrast, this work presents an automatic, portable, and productive solution for LAPACK codes on extremely heterogeneous systems. Additionally, performance results show that adapting the behavior of the codes/algorithms at each hardware component (including a high number of GPU accelerators) to attain high performance is possible.

III. Background

A. IRIS Run Time

As a programming system for extremely heterogeneous architectures, IRIS [4], [14] enables application developers to write portable applications across diverse heterogeneous programming platforms, including CUDA, HIP, Level Zero, OpenCL, and OpenMP (Fig. 1). IRIS orchestrates multiple

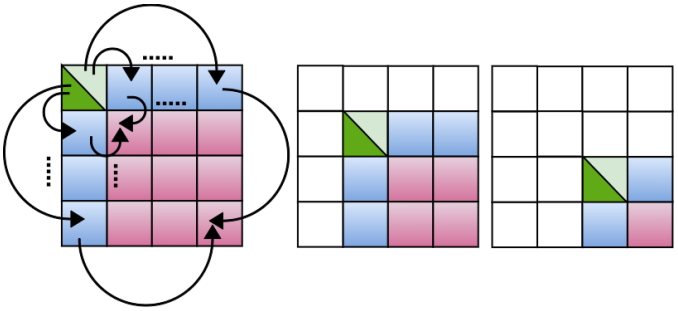


Figure 2: Tiled LU decomposition scheme [20].

programming platforms into a single execution/programming environment by providing portable tasks and shared virtual device memory.

IRIS provides 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 memory copy, (2) device-to-host 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 DAGs of tasks. Each task has a target device selection policy when it is submitted. The programmer specifies the policy, which can be a device number, device type (e.g., CPU, GPU, field-programmable array [FPGA], digital signal processor [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.

B. Tiled LU Factorization

Decomposing a matrix A into lower and upper triangular matrices (i.e., the LU factorization) is used to easily solve systems of linear equations:

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

LU factorization plays a key role in many computational science applications and is computationally expensive. Therefore, an LU factorization implementation on top of the IRIS programming model has the potential to facilitate performance portability on different modern heterogeneous systems, which is a goal of this research. Decomposing a matrix into tiles is a common strategy to parallelize this operation. Defining kernels, memory tiles, and dependencies by using task-level programming, such an implementation is possible [10], [19], [20].

The LU factorization on a tiled matrix (Fig. 2) consists of (1) factorizing the first tile of the diagonal to obtain the L (dark-green) and U (light-green) matrices of the tile; (2) computing several TRSMs (light-blue) by using the L matrix for the corresponding row and the U matrix for the corresponding column; and (3) 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.

Although the state-of-the-art routine for LU factorization involves pivoting, we considered a non-pivoting version for two reasons: (1) the pivoting is not necessary on well-conditioned matrices, and (2) we want to analyze the performance of the proposed optimizations without the influence of pivoting for 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 [10], LASs [20], Intel's MKL, NVIDIA's cuSolver [24] and cuSparse [25], FISHPACK [27], and SuperLU [7].

C. IRIS-BLAS

Still under development, IRIS-BLAS [15] is a novel, performance-portable BLAS library intended to address the portability challenges of BLAS for different heterogeneous architectures. IRIS-BLAS is built on top of the IRIS run time and multiple vendor and open-source BLAS libraries. IRIS-BLAS supports OpenBLAS [30], Intel MKL [13], NVIDIA cuBLAS [16], and AMD hipBLAS [1]. In a heterogeneous system, IRIS-BLAS offloads the appropriate BLAS library kernel based on the task mapping at run time. Thus, IRIS-BLAS is portable across a broad spectrum of architectures and BLAS libraries, thereby alleviating the worry of modifying the application source code. The effectiveness of IRIS-BLAS has been demonstrated on different CPUs (e.g., Intel Xeon Skylake, AMD 249 EPYC 7763, and Qualcomm Snapdragon ARM cores) and GPUs (e.g,. NVIDIA A100, AMD MI100, and Qualcomm Snapdragon Adreno). Although its objective is portability, IRIS-BLAS also provides competitive or even better performance compared with other state-of-the-art reference libraries [2]. By providing a vendor library kernel at run time, IRIS-BLAS provides an important building block for implementing complex linear algebra algorithms, and this is the main focus of this work, as discussed in the next section.

IV. LaRIS: Design and Objectives

LaRIS (an LAPACK library on the IRIS run time) is designed with three goals in mind: (1) simplify performance portability of LAPACK code on heterogeneous systems, (2) exploit the computational capabilities of heterogeneous systems to the fullest by using automatic tiled algorithms, and (3) facilitate the inclusion of algorithm-specific performance

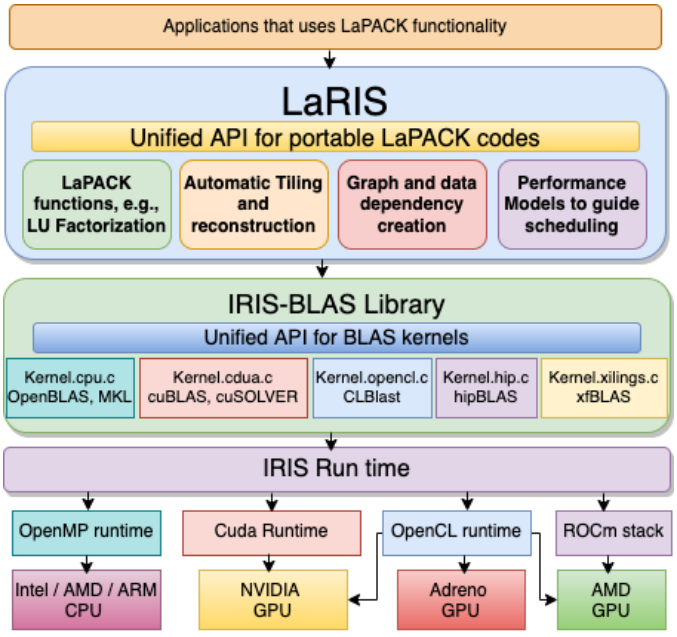


Figure 3: Software stack and design of LaRIS.

models to guide scheduling in heterogeneous systems. The LaRIS software stack is shown in Fig. 3.

A. Enabling Portability

To make LaRIS portable, we separated the algorithm design from the tuning. Algorithm design involves expressing the tiled LAPACK algorithms by using tasks and their dependencies, and tuning consists of choosing the target kernels and processors for executing each of the tasks. Two levels of abstractions facilitated by the IRIS run time and the IRIS-BLAS library enable such a separation (Fig. 3). Tiled LAPACK algorithms are expressed by using type-less LaRIS APIs along with the IRIS run time and the IRIS-BLAS APIs that do not include any architecture- or vendor-library specific detail (see Listings 2 and 3 in Appendix I). Therefore, LaRIS codes are completely agnostic of the architecture and vendor library at compile time. This feature facilitates the implementation of different tiled algorithms for LAPACK codes without having to worry about vendor libraries or the underlying hardware.

Tuning occurs at run time. By using dynamically linked libraries of IRIS and IRIS-BLAS at run time, LaRIS tasks are scheduled on different processors in a heterogeneous system on which vendor-specific kernels suitable for those processors are executed. IRIS's run time scheduler enables the full orchestrations during which IRIS-BLAS dynamically provides the vendor-specific tuned kernels. Thus, the tuning phase does not require code modifications at the algorithm level, but they are conducted internally and transparently by IRIS for each task. Leveraging IRIS's dynamic scheduler, the set of tasks in a LaRIS 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.

B. Tiled Execution

When processing larger matrix sizes, tiling is required to utilize all the processors in a heterogeneous system. LaRIS provides APIs for automatic tiling and reconstructing. Tiling occurs before task and dependency creation. While creating the graph, LaRIS associates memory chunks for different tiles to different tasks.

C. Current Status

LaRIS is under development along with IRIS-BLAS to support more LAPACK and BLAS-level functionalities. Although performance model–guided scheduling is in progress, the current implementation has enough support to make LAPACK codes portable for tiled execution. The following sections demonstrate one such capability with tiled LU factorization as a test case.

V. Tiled LU Implementation in LaRIS

This section discusses the LU factorization algorithm implemented in LaRIS. LaRIS decomposes a matrix into a set of square tiles (Fig. 2). Based on the provided tile configuration at the LaRIS level, the algorithm decomposes the problem by using different memory spaces, tasks, and dependencies. The pseudocode of the type-less LU factorization implementation in LaRIS can be found in Listing 2 in Appendix I.

A. Tasks

The implementation of LU factorization in LaRIS consists of four different tasks:

(1) GETRF, in which LaRIS computes a no-pivoting LU factorization on the diagonal tile of the matrix.
(2) TRSM-top, in which LaRIS computes the level-3 BLAS TRSM routine by using the lower side of the LU factorization computed in the previous task as the input matrix and a set of tiles located at the right of the LU matrix as the output matrices. One task per tile is created for computation.
(3) TRSM-left, in which LaRIS computes the same level-3 BLAS operation used in the previous tasks; however, it computes a different part of the matrix where the input is the upper side of the LU matrix processed by the first task (GETRF) and outputs a set of square tiles located under the lower side of the LU matrix.
(4) GEMM, in which LaRIS computes a set of matrix-matrix multiplications by using the output of the two previous tasks (TRSM-top and TRSM-left) as input and the tiles that correspond to the remaining matrix parts as output. LaRIS computes all the previous tasks until the entire matrix is computed (Fig. 2).

B. Automated DAG Creation and Scheduling

After tiling the memory, LaRIS creates a DAG that contains the number of tasks and the dependencies between these tasks. Because the execution environment is dynamic, incorrect dependencies lead to inaccurate results. In LU factorization, parallelization is possible between the tasks from different

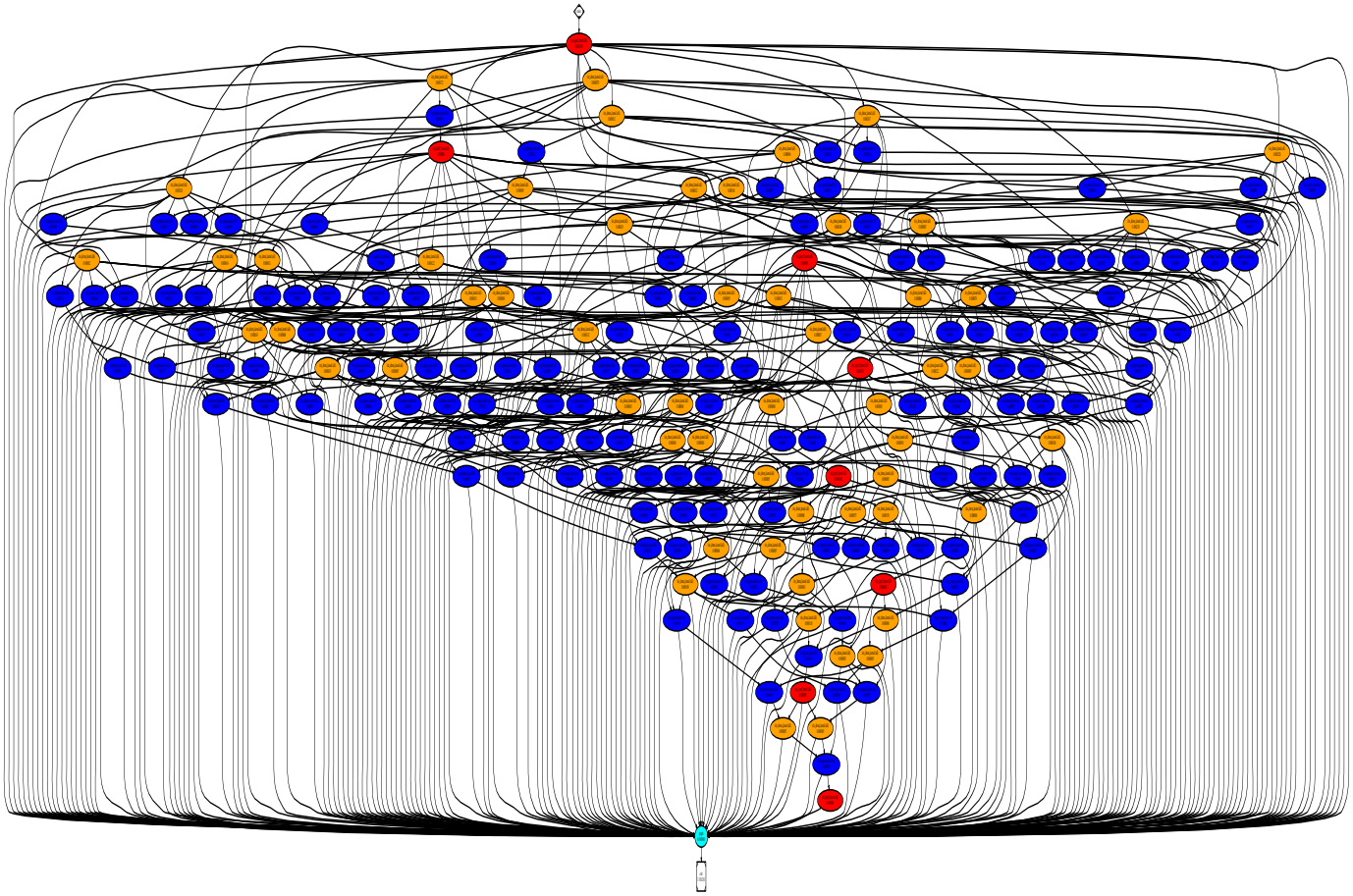


Figure 4: DAG of an 8×8 LU factorization that creates 204 tasks and their dependencies. Red ellipses are GETRF, orange ellipses are TRSM, and blue ellipses are GEMM.

iterations. Therefore, LaRIS follows data dependency instead of task dependency, where a dependency between two tasks is created if and only if the output memory of one task is an input memory of another.

Once the graph is created, LaRIS starts the execution by using `iris_graph_submit`. For the sake of completeness, Fig. 4 shows the DAG created by LaRIS LU code for a tile configuration (i.e., matrix decomposition) of 8×8 for a matrix of size $16,384 \times 16,384$. LaRIS is currently capable of computing the correct result of the LU factorization by decomposing a matrix with a tile configuration of up to 64×64, which creates $89,440$ kernel tasks. After the graph is submitted, the IRIS run time reads the task graph and submits them to the task queue. IRIS's dynamic task scheduling policy takes the tasks from the queue, chooses a target processor based on the policy, and submits the task to the device-specific task pool for execution.

C. Ensuring Utilization and Portability

As described in Section IV, one of LaRIS's main goals is to maximize the utilization of all available computational resources. To do that, apart from carrying out the matrix decomposition illustrated in Fig. 2, we use tuned multithreaded kernels (on CPU and GPUs) to exploit the parallelism at both algorithm and task levels. LaRIS leverages the dynamic scheduling policy from the IRIS run time to use all the available computational resources in a heterogeneous system. For example, the `iris_any` dynamic scheduler assigns each task to a CPU or GPU device depending on its availability at run time. Hence, the task execution is completely decoupled from the programming, and its IRIS run time's job is to care for it. The application's performance depends on the number of tasks (i.e., tiles), computational cost of each operation (GETRF/TRSM, GEMM), the type of underlying computing resources, and their count.

Portability is ensured by associating each task in the graph with one generic LAPACK or BLAS API from IRIS-BLAS. The parameters of these routines are similar to the specification of standard math libraries (i.e., LAPACK, BLAS). Hence, dynamic loading of the appropriate vendor-library kernel becomes possible. For example, LaRIS's double-precision GETRF routine is dynamically linked to the `LAPACKE_mkl_dgetrfnpi` kernel of the MKL library if a CPU is chosen by the IRIS dynamic scheduler to execute the task. If the IRIS scheduler chooses an AMD GPU, then `hipblasDgetrf` from hipBLAS is executed. Therefore, tasks of the DAG shown in Fig. 4 can execute on any available processor by launching the tuned kernel for that architecture. Relying on this dynamic selection capability, the same LaRIS code can be ported to a different set of computing resources without even changing a line in the code.

Table I: Heterogeneous systems used in this research

Nodes	NVIDIA DGX-1	Frontier-like node	CADES node
Facility	ExCL at ORNL	Crusher (early access) at ORNL	CADES at ORNL
GPUs	Total 4 GPUs 4× NVIDIA V100	Total 8 GPUs 4× AMD MI250X (each contains two)	Total 8 GPUs 4× NVIDIA A100 4× AMD MI100
CPU	Intel Xeon E5-2698, 20 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 for GPUs	cuBLAS and cuSOLVER	hipBLAS	cuBLAS, cuSOLVER and hipBLAS
Math libraries for CPU	OpenBLAS-0.3.20 and MKL-2022.1.0	OpenBLAS-0.3.17 and MKL-2020.4.304	OpenBLAS-0.3.20 and MKL-2020.4.304

VI. Experimental Setup

For this work, we used three heterogeneous systems with CPUs and GPUs from different manufacturers located in different computing facilities at ORNL. Table I shows the hardware configurations, architectures, compilers, software stacks, and BLAS libraries that were used. For profiling and tracing, the native capabilities of the IRIS run time were used. Execution times reported in Section VII are for the graph execution without the time spent for the initial memory allocation on the host side for matrix creation.

Experiments were conducted with tile configurations of [4×4, 8×8, 16×16, 32×32, and 64×64], which resulted in [30, 204, $1,496$, $11,440$, and $89,440$] tasks (kernels) to compute the final result. Double precision was used for all experiments. Experiments were run five times, and an average was reported.

VII. Experiments

We evaluated the efficacy of LaRIS's LU factorization in four steps. First, LaRIS's portability is explored. Then, strong scaling performance for an increasing number of GPUs is evaluated, followed by a discussion of the trade-off between parallelism and overhead. Finally, the optimization opportunities are identified.

A. Portability to Different Heterogeneous Devices

Two aspects are evaluated in this experiment: portability and system utilization. For this experiment, we used a $16,384\times 16,384$ matrix with a tile decomposition of 32×32, which created $11,440$ kernel tasks. Because one of the main objectives of LaRIS is to provide a portable solution for LAPACK codes, no architecture- or vendor-specific library changes were made in the LU factorization code while deploying and experimenting on the three heterogeneous systems mentioned in Table I. Once the graph was submitted to the IRIS run time, seamless execution was observed in these systems (Fig. 5). The IRIS run time was able to select appropriate vendor-specific BLAS kernels to schedule on the corresponding processors and orchestrate the correct result for the LU factorization. Because the `IRIS_any` scheduling policy was used, no binding was imposed between the kernel types, BLAS kernel, or target processors. Therefore, IRIS had complete freedom to run any kernel on any device to maximize the parallelization opportunities.

Traces for these executions are shown in Fig. 5. Figures 5a and 5b show the execution timeline on one CPU and on multiple GPUs from NVIDIA and AMD, respectively. Notably, the CPU bar represents all available cores. Figure 5c shows the most interesting case in which one AMD CPU, four NVIDIA A100 GPUs, and four AMD MI100 GPUs processed the entire graph. The complete separation of the LAPACK code from the BLAS libraries and the architecture made this seamless portability possible.

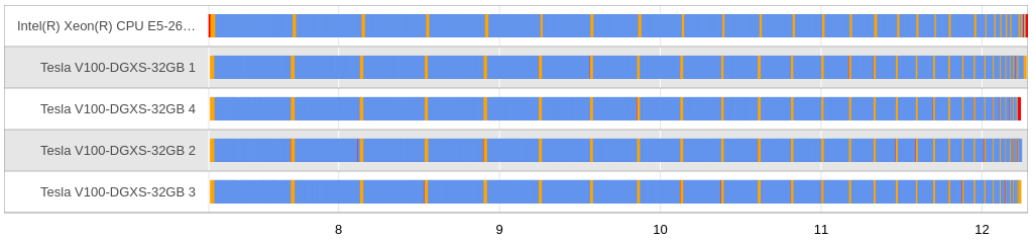
Three traces in Fig. 5 also demonstrate system utilization by depicting $11,440$ task kernels in different colors. Conceptually, these traces can be considered a counter-clockwise, 90° rotation of Fig. 4 when using multiple processors. White spaces indicate idle time and are barely noticeable in all these three traces, except at the end of the timeline. As shown in Fig. 4, parallelization reduces at the end of the LU factorization algorithm, which is why there are some white spaces at the very end. However, the main takeaway in terms of utilization is that when the algorithm exposed parallelism, LaRIS was able to utilize the parallelism by exploiting the computational power of all the available resources.

B. Strong Scaling in a MultiGPU Environment

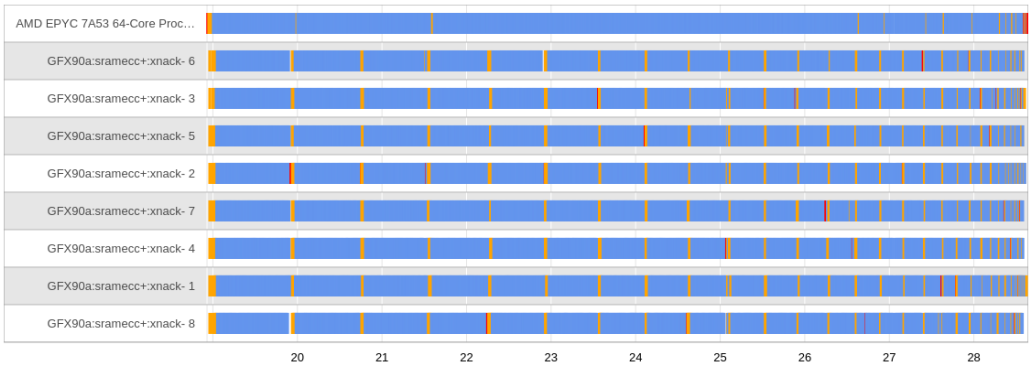
Strong scaling capability is evaluated in this experiment. The matrix size and tile configuration were kept the same as the number of GPUs used in the systems (Table I) was increased. A 16×16 tile configuration was used for all the experiments. A matrix size of $16,384\times 16,384$ was used for the DGX-1 machine, and $32,768\times 32,768$ was used for Crusher and the CADES node. The number of GPUs was increased to observe the change in execution time, which is presented in Fig. 6. The DGX-1 machine showed a performance improvement with each addition of an NVIDIA V100 GPU for processing LU factorization (Fig. 6b).

Scalability for the Crusher node is shown in Fig. 6c, and performance improvements were observed when adding up to four AMD MI250X GPUs. However, increasing the number of GPUs beyond four made the performance worse.

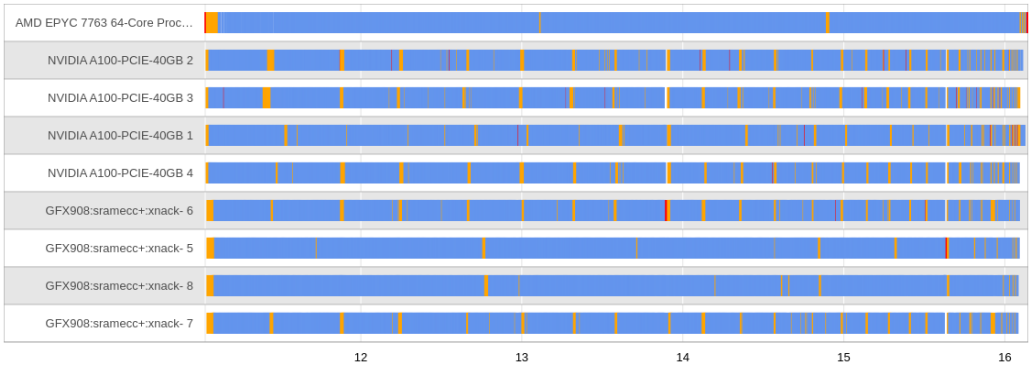
Scalability on the CADES machine showed a waterfall trend (Fig. 6a). When the number of NVIDIA A100 GPUs was increased, the performance improvement was saturated when using four GPUs. However, the performance improvement kept resurfacing when AMD MI100 GPUs were added and reached a saturation point when four NVIDIA and four AMD GPUs were used.



(a) Trace from NVIDIA DGX-1.



(b) Trace from Crusher node.



(c) Trace from CADES node.

Figure 5: Evaluating portability and exploiting parallelism in GPUs and CPUs from multiple vendors. Traces are generated for a $16,384 \times 16,384$ matrix with a tile decomposition of 32×32, which created $11,440$ tasks. Red is GETRF, orange is TRSM, and blue is GEMM. The horizontal axis shows the timeline only for graph execution in seconds, whereas the vertical axis shows different CPUs and GPUs.

The saturation shown in Figs. 6c and 6a can be correlated to the overhead of multidevice tasking, which induces an increased number of memory transfers. With an increasing number of GPUs, the probability that the data needed by the tasks is not located in the device increases; hence, it can cause extra memory transfers. We are working toward better and more efficient memory management for such scenarios.

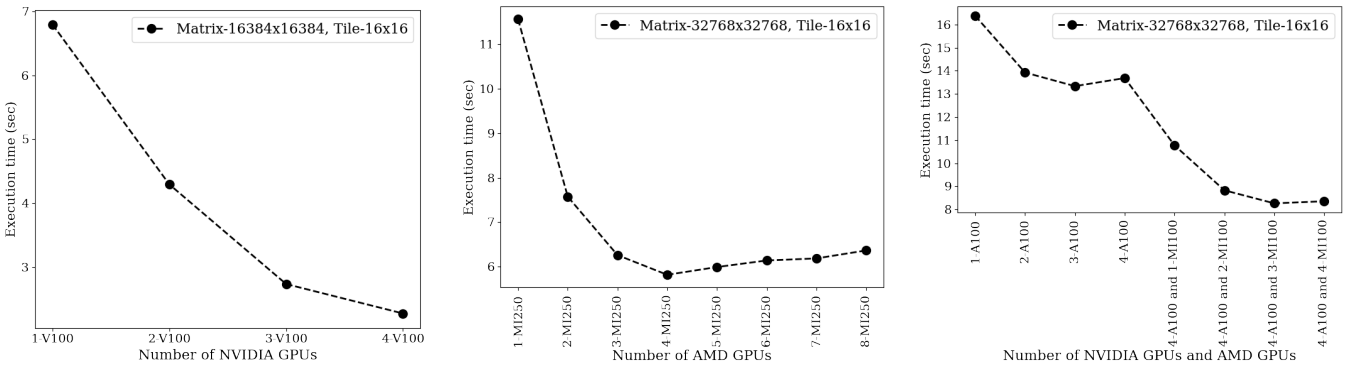
C. Best Tile Configuration: a Trade-off between Parallelism and Overhead

This experiment delves into finding the best tile configurations for a given matrix size in a node. Large matrices were processed by using different tile configuration launch-task kernels that ranged in number from 30 (for 4×4) to $89,440$ (for 64×64) kernel tasks (Fig. 7). For this experiment, all the processing units were used. Some matrix and tile configurations are not shown because the available memory in the existing devices did not allow all configurations.

Table II: Best tile configuration for different matrix sizes

Matrix	DGX-1	Crusher node	CADES node
$8,192 \times 8,192$	8×8	4×4	4×4
$16,384 \times 16,384$	8×8	8×8	8×8
$32,768 \times 32,768$	8×8	16×16	16×16
$65,536 \times 65,536$	NA	16×16	16×16

The main observation from Fig. 7 is that the best tile configuration for different matrices varies from node to node because of the underlying hardware capacity. Table II presents a summary of findings shown in Fig. 7 by showing the best tile configurations for different matrix sizes on different nodes. Only the $16,384 \times 16,384$ matrix has the same best tile configuration on all nodes. Moreover, the larger matrices prefer larger tiles on machines with more processing units. Such a preference is the byproduct of a node having more processing



(a) NVIDIA DGX-1 with four V100 GPUs. (b) Crusher node with eight MI250X GPUs. (c) CADES node with four NVIDIA A100 GPUs and four AMD MI100 GPUs.

Figure 6: Strong scaling on different machines.

elements that can leverage the extra parallelism generated by the large tile configurations. However, no matrix performed the best for the configurations 32×32 and 64×64. We correlate such outcomes with the inclusion of host-to-device and device-to-host memory transfer for each task. Thus, a larger number of tasks in a larger number of processors incur more memory transfer overhead. Adding memory transfers is not required for every task because the parent task might have already brought the data to the device. We plan to optimize such data transfers in the future.

D. Opportunities

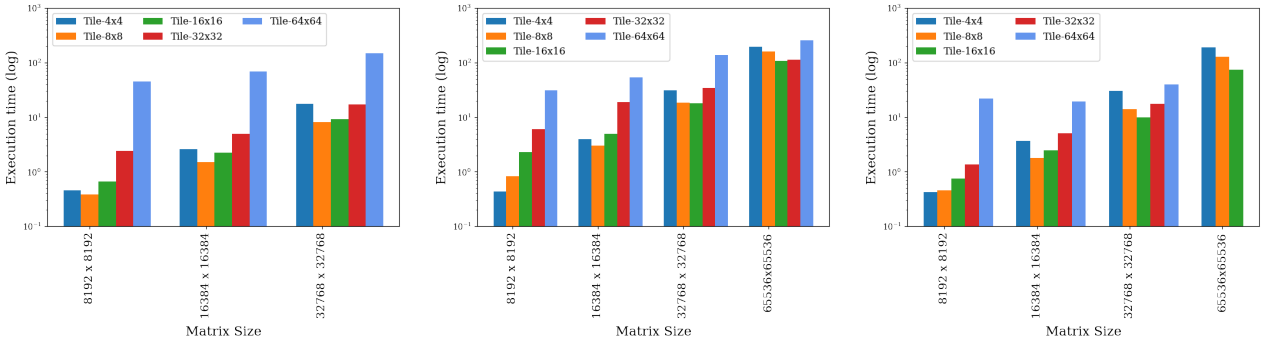
We have identified two opportunities from these experiments. First, we see that the best execution time does not depend on using the maximum number of available processors or the highest number of tiles. Every matrix configuration and node has a sweet spot for a particular tile and number of processors. Generating performance models to guide the scheduler to that desired tile and processor combination is a research opportunity. The second opportunity arises from the memory transfer inefficiency discussed in Section VII-C.

VIII. Conclusions and Future Work

In this paper, we presented LaRIS, a portable solution that enables LAPACK codes on all processing elements in a heterogeneous system by using tiled LAPACK algorithms. Using the IRIS run time and the IRIS-BLAS library, LaRIS decouples LAPACK algorithms from the architectures and architecture-specific tuned BLAS libraries. Moreover, LaRIS offers automatic tilling and can dynamically link to the corresponding vendor-provided BLAS kernel at run time, which enables LaRIS to be portable and capable of using all the processors in different heterogeneous systems. By launching $\approx 90,000$ BLAS kernels for LU factorization, LaRIS demonstrated its ability to be portable to different heterogeneous systems, including a representative node of the first exascale supercomputer, ORNL's Frontier, and showed its capability to utilize all the processors simultaneously to compute correct results. Some optimization opportunities (e.g., memory optimization) were also identified through experimentation. We plan to work on those optimizations and include performance model–guided scheduling in the future.

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 research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the DOE's Office of Science and the National Nuclear Security Administration. 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).



(a) NVIDIA DGX-1. (b) Crusher node. (c) CADES node.

Figure 7: Finding the best tile configuration for different matrix sizes when all the processing elements are used.

REFERENCES

[1] AMD. hipBLAS, the Basic Linear Algebra Subroutine library, 2022. [Online; accessed 6-July-2022].

[2] 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.

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

[4] A. M. Cabrera, S. Hitefield, J. Kim, S. Lee, N. R. Miniskar, and J. S. Vetter. Toward performance portable programming for heterogeneous systems on a chip: A case study with qualcomm snapdragon soc. In *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021*, pages 1–7. IEEE, 2021.

[5] S. Catalán, J. R. Herrero, E. S. Quintana-Ortí, R. Rodríguez-Sánchez, and R. A. van de Geijn. A case for malleable thread-level linear algebra libraries: The LU factorization with partial pivoting. *CoRR*, abs/1611.06365, 2016.

[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] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Anal. Appl.*, 20(4):915–952, 1999.

[8] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon. The design and performance of batched BLAS on modern high-performance computing systems. *Procedia Computer Science*, 108:495 – 504, 2017. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[9] 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.

[10] 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.

[11] M. A. A. Farhan, A. Abdelfattah, S. Tomov, M. Gates, D. Sukkari, A. Haidar, R. Rosenberg, and J. J. Dongarra. MAGMA templates for scalable linear algebra on emerging architectures. *Int. J. High Perform. Comput. Appl.*, 34(6), 2020.

[12] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4):422–455, Dec. 2001.

[13] Intel. The Intel Math Kernel Library, 2022. [Online; accessed 6-July-2022].

[14] 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.

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

[16] NVIDIA. cuBLAS, the CUDA Basic Linear Algebra Subroutine library, 2022. [Online; accessed 6-July-2022].

[17] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Q. Dang, N. D. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke. Kokkos 3: Programming model extensions for the exascale era. *IEEE Trans. Parallel Distributed Syst.*, 33(4):805–817, 2022.

[18] P. Valero-Lara, D. Andrade, R. Sirvent, J. Labarta, B. B. Fraguela, and R. Doallo. A fast solver for large tridiagonal systems on multi-core processors (lass library). *IEEE Access*, 7:23365–23378, 2019.

[19] 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.

[20] 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.

[21] P. Valero-Lara, F. D. Igual, M. Prieto-Matías, A. Pinelli, and J. Favier. Accelerating fluid-solid simulations (lattice-boltzmann & immersed-boundary) on heterogeneous architectures. *J. Comput. Sci.*, 10:249–261, 2015.

[22] P. Valero-Lara and J. Jansson. Heterogeneous CPU+GPU approaches for mesh refinement over lattice-boltzmann simulations. *Concurr. Comput. Pract. Exp.*, 29(7), 2017.

[23] P. Valero-Lara, J. Kim, O. Hernandez, and J. S. Vetter. Openmp target task: Tasking and target offloading on heterogeneous systems. In R. Chaves, D. B. Heras, A. Ilic, D. Unat, R. M. Badia, A. Bracciali, P. Diehl, A. Dubey, O. Sangyoon, S. L. Scott, and L. Ricci, editors, *Euro-Par 2021: Parallel Processing Workshops - Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers*, volume 13098 of *Lecture Notes in Computer Science*, pages 445–455. Springer, 2021.

[24] P. Valero-Lara, I. Martínez-Perez, R. Sirvent, X. Martorell, and A. J. Peña. 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*, pages 243–253, 2017.

[25] P. Valero-Lara, I. Martínez-Pérez, R. Sirvent, X. Martorell, and A. J. Peña. cuThomasBatch and cuThomasVBatch, CUDA routines to compute batch of tridiagonal systems on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience*, 30(24), 2018.

[26] P. Valero-Lara, I. Martínez-Pérez, S. Mateo, R. Sirvent, V. Beltran, X. Martorell, and J. Labarta. Variable batched DGEMM. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 363–367, March 2018.

[27] P. Valero-Lara, A. Pinelli, and M. Prieto-Matias. Fast finite difference Poisson solvers on heterogeneous architectures. *Computer Physics Communications*, 185(4):1265 – 1272, 2014.

[28] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. AUGEM: automatically generate high performance dense linear algebra kernels on x86 cpus. In W. Gropp and S. Matsuoka, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pages 25:1–25:12. ACM, 2013.

[29] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1998, November 7-13, 1998, Orlando, FL, USA*, page 38. IEEE Computer Society, 1998.

[30] M. K. Zhang Xianyi. OpenBLAS, 2022. [Online; accessed 6-July-2022].

APPENDIX I

This section provides pseudocodes for LaRIS's tiled LU factorization and GEMM operation.

```
// Creation of the IRIS graph
iris_graph graph; iris_graph_create(&graph);
iris_task getrf_tasks[TILE_NUM*TILE_NUM],
    trsm_tasks[TILE_NUM*TILE_NUM],
    gemm_tasks[TILE_NUM*TILE_NUM*TILE_NUM];
// Tiled LU factorization
for ( int step = 0; step < TILE_NUM; step++ ) {
  // GETRF tasks
  iris_task_create(
    &getrf_tasks[(step*TILE_NUM)+step]);
  // Dependencies for getrf tasks
  if ( step > 0 ){
```

```
  iris_task getrf_depend_tasks[] =
      { gemm_tasks[((step-1)*TILE_NUM*TILE_NUM)+
      ((step*TILE_NUM)+step)] };
  iris_task_depend( getrf_tasks[(step*TILE_NUM) +
      step], 1, getrf_depend_tasks );
}
// Encapsulate GETRF tasks into the graph
laris_getrf_graph( graph,
      getrf_tasks[(step*TILE_NUM)+step],
      step, step, step, TILE_SIZE, TILE_SIZE,
      IRIS_Ctile[step*TILE_NUM+step],
      A_tile[step][step], TILE_SIZE );
for ( int tile_jj = 1 + step; tile_jj < TILE_NUM;
      tile_jj++ ) {
  // trsm tasks
  iris_task_create( &trsm_tasks[(step*TILE_NUM)+
  tile_jj] );
  // Dependencies for trsm tasks
  if ( step == 0 ) {
    iris_task trsm_depend_tasks[] =
      { getrf_tasks[ ( step * TILE_NUM ) + step ]
  };
    iris_task_depend( trsm_tasks[(step*TILE_NUM) +
      tile_jj], 1, trsm_depend_tasks );
  } else {
      iris_task trsm_depend_tasks[] = {
          getrf_tasks[(step*TILE_NUM)+step],
          gemm_tasks[((step-1)*TILE_NUM*TILE_NUM)
          +(step*TILE_NUM)+tile_jj]};
      iris_task_depend( trsm_tasks[(step*TILE_NUM)
   +
            tile_jj], 2, trsm_depend_tasks );
  }
  // Encapsulate trsm tasks into the graph
  laris_trsm_top_graph( graph, trsm_tasks[(step*
  TILE_NUM)+
        tile_jj], step, step, tile_jj,
        TILE_SIZE, TILE_SIZE,
        IRIS_Ctile[step*TILE_NUM+step],
        A_tile[step][step], TILE_SIZE,
        IRIS_Ctile[step*TILE_NUM+tile_jj],
        A_tile[step][tile_jj], TILE_SIZE );
}
for ( int tile_ii = 1 + step; tile_ii < TILE_NUM;
      tile_ii++ ) {
  // trsm tasks
  iris_task_create(
      &trsm_tasks[(tile_ii*TILE_NUM)+step] );
  // Dependencies for trsm tasks
  if ( step == 0 ) {
    iris_task trsm_depend_tasks[] = {
      getrf_tasks[ ( step * TILE_NUM ) + step ]};
      iris_task_depend( trsm_tasks[ ( tile_ii *
      TILE_NUM ) + step ], 1, trsm_depend_tasks );
  } else {
    iris_task trsm_depend_tasks[] = {
      getrf_tasks[(step*TILE_NUM)+step],
      gemm_tasks[((step-1)*TILE_NUM*TILE_NUM)+
      (tile_ii*TILE_NUM)+step] };
    iris_task_depend(
      trsm_tasks[ ( tile_ii * TILE_NUM ) + step ],
      2, trsm_depend_tasks );
  }
  // Encapsulate trsm tasks into the graph
  laris_trsm_left_graph( graph,
      trsm_tasks[(tile_ii*TILE_NUM)+step],
      step, tile_ii, step, TILE_SIZE, TILE_SIZE,
      IRIS_Ctile[step * TILE_NUM + step],
      A_tile[step][step], TILE_SIZE,
      IRIS_Ctile[tile_ii * TILE_NUM + step],
      A_tile[tile_ii][step], TILE_SIZE );
  }
  for ( int tile_i = step + 1; tile_i < TILE_NUM;
        tile_i++ ) {
    for ( int tile_j = step + 1; tile_j < TILE_NUM
  ;
        tile_j++ ) {
      // gemm tasks
      iris_task_create( &gemm_tasks[(step*TILE_NUM
  *TILE_NUM) +
        ((tile_i*TILE_NUM)+tile_j)] );
      // Dependencies for gemm tasks
      if ( step == 0 ) {
        iris_task gemm_depend_tasks[] = {
          trsm_tasks[(tile_i*TILE_NUM)+step],
          trsm_tasks[(step*TILE_NUM)+tile_j] };
        iris_task_depend(
          gemm_tasks[(step*TILE_NUM*TILE_NUM)+((
  tile_i*
        TILE_NUM)+tile_j)], 2, gemm_depend_tasks
   );
      }
      else {
         iris_task gemm_depend_tasks[] = {
            trsm_tasks[(tile_i*TILE_NUM)+step],
  trsm_tasks[(step*TILE_NUM)+tile_j],
            gemm_tasks[((step-1)*TILE_NUM*
  TILE_NUM)+
            ((tile_i*TILE_NUM)+tile_j)] };
         iris_task_depend(
            gemm_tasks[(step*TILE_NUM*TILE_NUM)+
            ((tile_i*TILE_NUM)+tile_j)], 3,
            gemm_depend_tasks );
      }
      // Encapsulate gemm tasks into the graph
      laris_gemm_graph( graph,
          gemm_tasks[(step*TILE_NUM*TILE_NUM)+
          ((tile_i*TILE_NUM)+tile_j)], step,
  tile_i,
        tile_j, TILE_SIZE, TILE_SIZE, TILE_SIZE,
   -1.0,
        IRIS_Ctile[tile_i*TILE_NUM+step],
        A_tile[tile_i][step], TILE_SIZE,
        IRIS_Ctile[step*TILE_NUM+tile_j],
        A_tile[step][ tile_j], TILE_SIZE, 1.0,
        IRIS_Ctile[tile_i*TILE_NUM+tile_j],
        A_tile[tile_i][tile_j], TILE_SIZE );
    }
  }
}
iris_graph_submit(graph, iris_default, 1);
```

Listing 1: LaRIS's tiled LU factorization (GETRF) code.

```
int laris_gemm_graph(
    iris_graph graph,
    iris_task T,
    int step, int tile_i, int tile_j,
    int M, int N, int K,
    TYPE ALPHA, iris_mem d_A, TYPE *h_A, int LDA,
    iris_mem d_B, TYPE *h_B, int LDB,
    TYPE BETA, iris_mem d_C, TYPE *h_C, int LDC )
{
  // Memory communication -> input
  iris_task_h2d(T, d_A, 0, M*K*sizeof(TYPE), h_A);
  iris_task_h2d(T, d_B, 0, K*N*sizeof(TYPE), h_B);
  iris_task_h2d(T, d_C, 0, M*N*sizeof(TYPE), h_C);
  // IRIS-BLAS call
  iris_core_nodt_gemm( T, IRIS_BLAS_COL_MAJOR,
        IRIS_BLAS_NO_TRANS, IRIS_BLAS_NO_TRANS,
        M, N, K, ALPHA, d_A, LDA,
        d_B, LDB,BETA, d_C, LDC);
  // Memory communication -> output
  iris_task_d2h(T, d_C, 0, M*N*sizeof(TYPE), h_C);
  iris_graph_task( graph, T, iris_any, NULL );
  return 0;
}
```

Listing 3: LaRIS's GEMM code.

```
// Creation of the IRIS graph
iris_graph graph; iris_graph_create(&graph);
iris_task getrf_tasks[TILE_NUM*TILE_NUM], trsm_tasks[TILE_NUM*TILE_NUM], gemm_tasks[TILE_NUM*TILE_NUM*
    TILE_NUM];
// Tiled LU factorization
for ( int step = 0; step < TILE_NUM; step++ ) {
  // GETRF tasks
  iris_task_create(&getrf_tasks[(step*TILE_NUM)+step]);
  // Dependencies for getrf tasks
  iris_task getrf_depend_tasks[] = { gemm_tasks[((step-1)*TILE_NUM*TILE_NUM)+((step*TILE_NUM)+step)] };
  iris_task_depend( getrf_tasks[(step*TILE_NUM) + step], 1, getrf_depend_tasks );
  // Encapsulate GETRF tasks into the graph
  laris_getrf_graph( graph, getrf_tasks[(step*TILE_NUM)+step],
        step, step, step, TILE_SIZE, TILE_SIZE, IRIS_Ctile[step*TILE_NUM+step],
                                                A_tile[step][step], TILE_SIZE );
  for ( int tile_jj = 1 + step; tile_jj < TILE_NUM; tile_jj++ ) {
    // trsm tasks
    iris_task_create( &trsm_tasks[(step*TILE_NUM)+tile_jj] );
    // Dependencies for trsm tasks
    iris_task trsm_depend_tasks[] = { getrf_tasks[(step*TILE_NUM)+step],
            gemm_tasks[((step-1)*TILE_NUM*TILE_NUM)+(step*TILE_NUM)+tile_jj]};
    iris_task_depend( trsm_tasks[(step*TILE_NUM)+tile_jj], 2, trsm_depend_tasks );
    // Encapsulate trsm tasks into the graph
    laris_trsm_top_graph( graph, trsm_tasks[(step*TILE_NUM)+tile_jj], step, step, tile_jj,
          TILE_SIZE, TILE_SIZE, IRIS_Ctile[step*TILE_NUM+step], A_tile[step][step], TILE_SIZE,
                                IRIS_Ctile[step*TILE_NUM+tile_jj], A_tile[step][tile_jj], TILE_SIZE );
  }
  for ( int tile_ii = 1 + step; tile_ii < TILE_NUM;tile_ii++ ) {
    // trsm tasks
    iris_task_create( &trsm_tasks[(tile_ii*TILE_NUM)+step] );
    // Dependencies for trsm tasks
      iris_task trsm_depend_tasks[] = { getrf_tasks[(step*TILE_NUM)+step],
        gemm_tasks[((step-1)*TILE_NUM*TILE_NUM)+(tile_ii*TILE_NUM)+step] };
      iris_task_depend( trsm_tasks[ ( tile_ii * TILE_NUM ) + step ], 2, trsm_depend_tasks );
    // Encapsulate trsm tasks into the graph
    laris_trsm_left_graph( graph, trsm_tasks[(tile_ii*TILE_NUM)+step], step, tile_ii, step,
        TILE_SIZE, TILE_SIZE, IRIS_Ctile[step * TILE_NUM + step], A_tile[step][step], TILE_SIZE,
                              IRIS_Ctile[tile_ii * TILE_NUM + step], A_tile[tile_ii][step], TILE_SIZE );
    }
    for ( int tile_i = step + 1; tile_i < TILE_NUM; tile_i++ ) {
      for ( int tile_j = step + 1; tile_j < TILE_NUM; tile_j++ ) {
        // gemm tasks
        iris_task_create( &gemm_tasks[(step*TILE_NUM*TILE_NUM) + ((tile_i*TILE_NUM)+tile_j)] );
        // Dependencies for gemm tasks
        iris_task gemm_depend_tasks[] = { trsm_tasks[(tile_i*TILE_NUM)+step], trsm_tasks[(step*TILE_NUM)+
    tile_j],
                gemm_tasks[((step-1)*TILE_NUM*TILE_NUM)+((tile_i*TILE_NUM)+tile_j)] };
        iris_task_depend( gemm_tasks[(step*TILE_NUM*TILE_NUM)+((tile_i*TILE_NUM)+tile_j)], 3,
                gemm_depend_tasks );
        // Encapsulate gemm tasks into the graph
        laris_gemm_graph( graph, gemm_tasks[(step*TILE_NUM*TILE_NUM)+((tile_i*TILE_NUM)+tile_j)], step,
            tile_i, tile_j, TILE_SIZE, TILE_SIZE, TILE_SIZE,
            -1.0, IRIS_Ctile[tile_i*TILE_NUM+step], A_tile[tile_i][step],  TILE_SIZE,
                  IRIS_Ctile[step*TILE_NUM+tile_j], A_tile[step][ tile_j], TILE_SIZE,
            1.0, IRIS_Ctile[tile_i*TILE_NUM+tile_j], A_tile[tile_i][tile_j], TILE_SIZE );
    }
  }
}
iris_graph_submit(graph, iris_default, 1);
```

Listing 2: LaRIS's tiled LU factorization (GETRF) code.