A Portable and Heterogeneous LU Factorization on IRIS

Pedro Valero-Lara(✉), Jungwon Kim, and Jeffrey S. Vetter

Oak Ridge National Laboratory, Oak Ridge, USA
{valerolarap,vetter}@ornl.gov, kimj@ieee.org

Abstract. Here, the IRIS programming model is evaluated as a method to improve performance portability for heterogeneous systems that use LU matrix factorization. LU (lower-upper) factorization is considered one of the most important numerical linear algebra operations used in multiple high-performance computing and scientific applications. IRIS enables the separation of the algorithm's definition from the tuning by using tasks + dependencies. This considerably reduces the effort required to achieve performance portability on heterogeneous systems. One IRIS code can use different settings depending on the underlying hardware features. Different configurations are evaluated on two different heterogeneous systems to achieve important speedups for the reference code with minimal changes to the source code.

Keywords: IRIS · Tasking · Heterogeneity · Performance portability · CPU · GPU · LU factorization

1 Introduction

This paper describes performance portability on different heterogeneous systems using the IRIS programming model[1] for LU (lower-upper) matrix factorization. Iris is a task + dependency-based programming model in which each task can encapsulate almost any kind of current parallel code (e.g., OpenMP, CUDA, HIP, OpenACC) and targets almost any current parallel computer architecture (e.g., CPUs, graphics processing units [GPUs], digital signal processors [DSPs], field-programmable gate arrays [FPGAs]).

[1] https://iris-programming.github.io/.

J. Kim—Now at NVIDIA.

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

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
J. Singer et al. (Eds.): Euro-Par 2022 Workshops, LNCS 13835, pp. 17–31, 2023.
https://doi.org/10.1007/978-3-031-31209-0_2

We use LU factorization as a motivating case study given its importance in multiple high-performance computing (HPC) applications [1–3], but the ideas explored in this paper can also be effectively applied to other HPC applications. LU factorization is also one of the most important benchmarks [4,5] used to evaluate the performance of HPC systems.[2] Parallel LU factorization is composed of four major and completely different operations that must be computed on blocks of different shapes and sizes, and the size of these blocks are different along the computation. All these factors make LU factorization a challenging case study for performance portability on heterogeneous systems—for which the best target architecture of each application component is unclear.

To make performance portability on heterogeneous systems simpler, we separate the algorithm design from the tuning. While the algorithm is described by using tasks + dependencies on top of IRIS, the tuning consists of choosing the target/code for each of the tasks, which enables us to use one code for multiple platforms.

The rest of the paper is organized as follows: Sect. 2 presents the main characteristics of the IRIS programming model, Sect. 3 introduces the LU factorization case study, and Sect. 4 outlines the effort to implementation a portable and heterogeneous LU code using IRIS. The performance study is described in Sect. 5. Finally, related work is summarized in Sect. 6, and future directions and conclusions are presented in Sect. 7.

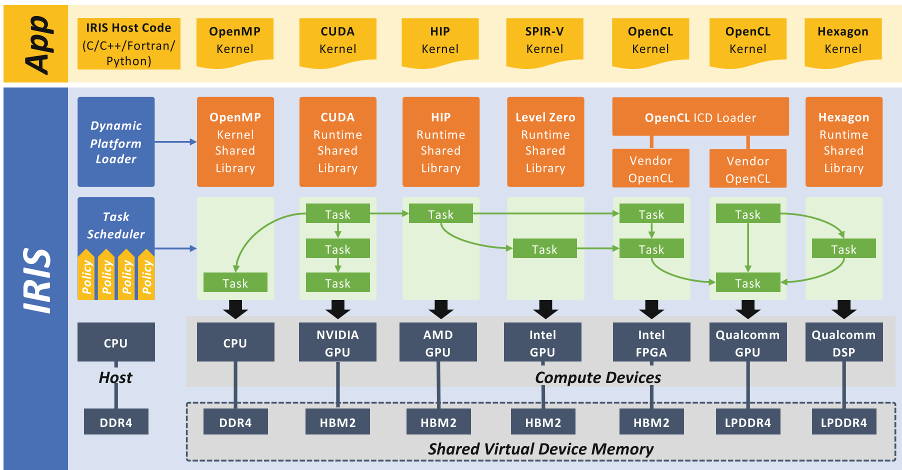

Fig. 1. The IRIS architecture.

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

2 IRIS Programming System

As a programming system for extremely heterogeneous architectures, IRIS [6] 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 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. A task runs on a single device but is portable across any compute device in a 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 have a dependency on other tasks, it cannot start until the prerequisite tasks complete. Therefore, writing an IRIS application means building directed acyclic graphs of tasks. Each task has a target device selection policy when it is submitted. This policy is specified by the programmer, and it 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).

To achieve application portability and flexible task scheduling with effective data orchestration, IRIS provides shared virtual device memory across multiple, disjointed physical device memories. 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 they can see the same content in the memory objects.

3 LU Factorization

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

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

LU factorization plays a key role in many computational science applications. However, it is also computationally expensive, which motivated us to develop a new LU factorization implementation on top of the IRIS programming model to provide performance portability on different modern heterogeneous systems.

One of the most common ways to parallelize this type of operation is to decompose the matrix into tiles by defining the dependencies between the tiles and the operations to be computed on each tile. This can be accomplished through tasking [7–9].

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., general

matrix multiply [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.

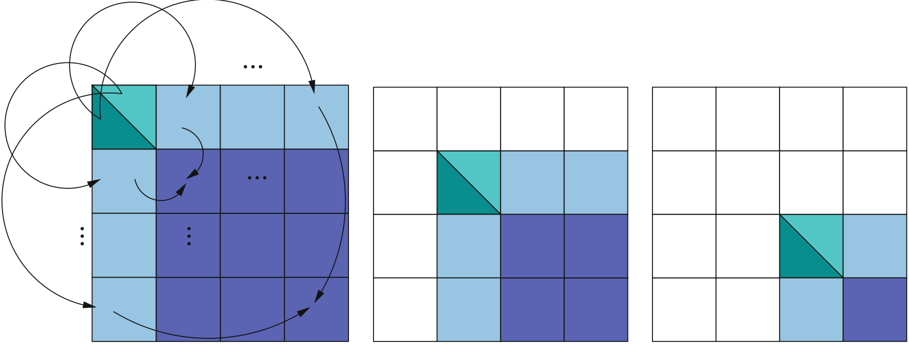

Fig. 2. LU decomposition.

Although the state-of-the-art routine for LU factorization involves pivoting, we developed 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 the sake of performance analysis. Additionally, although using pivoting to solve linear systems of 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 [9], LASs [7], Intel's MKL,[3] NVIDIA's cuSolver [10] and cuSparse [11], FISHPACK [12,13], and SuperLU [14].

4 Implementation

Figure 3 shows the pseudocode for the IRIS-implemented LU factorization. At this *algorithm level*, we declare the different memory spaces and tasks and dependencies among them and describe the algorithm to be computed. As shown, every task can be computed on a CPU, a GPU, or both depending on the optimizations and ideas we want to explore. These optimizations do not require code modifications at the algorithm level, but they are conducted internally in each of the tasks at the *implementation level*. Although the algorithm is described/implemented at the algorithm level in an architecture-agnostic way, the implementation level (set of tasks) attempts to obtain the maximum performance on the target architecture. One of the benefits of using IRIS is that one algorithm-level code can have multiple and different implementation levels with each optimized for a specific heterogeneous platform.

[3] https://software.intel.com/en-us/mkl-developer-reference-c-mkl-getrfnpi.

```
int SIZE= 16384; int TILE_SIZE = 512; int num_tiles = SIZE/TILE_SIZE
A = malloc(SIZE*SIZE);
//Creation of the IRIS graph
iris_graph graph;
iris_graph_create(&graph);
//Creation of the IRIS memory space
iris_mem A_iris, B_iris0, B_iris1, C_iris;
iris_mem_create(             TILE_SIZE * TILE_SIZE,  &A_iris);
iris_mem_create(      TILE_SIZE * (SIZE-TILE_SIZE), &B_iris0);
iris_mem_create(      TILE_SIZE * (SIZE-TILE_SIZE), &B_iris1);
iris_mem_create((SIZE-TILE_SIZE) * (SIZE-TILE_SIZE),  &C_iris);
//Creation of the IRIS tasks pointers
iris_task *getrf     = malloc(num_tiles*sizeof(iris_task));
iris_task *trsm_top  = malloc(num_tiles*sizeof(iris_task));
iris_task *trsm_left = malloc(num_tiles*sizeof(iris_task));
iris_task *gemm      = malloc(num_tiles*sizeof(iris_task));
//Creation of the IRIS tasks parameters
struct *getrf_params      = malloc( num_tiles * sizeof(getrf_params);
struct *trsm_top_params   = malloc( num_tiles * sizeof(getrf_params);
struct *trsm_left_params  = malloc( num_tiles * sizeof(getrf_params);
struct *gemm_params       = malloc( num_tiles * sizeof(getrf_params);
for ( d = 0; d < num_tiles; d++){
    //---GETRF TASK---
    //Creation of the getrf[d] task
    iris_task_create_perm(&getrf[d]);
    //Initialization of getrf task's parameters
    getrf_params[d].M      = TILE_SIZE;
    getrf_params[d].LDA    = TILE_SIZE;
    ...
    getrf_params[d].A_cpu = &A[( (d * TILE_SIZE) * LDA ) + ( d * TILE_SIZE )];
    getrf_params[d].A_gpu = A_iris;
    //Initialization of the task
    iris_task_host(getrf[d], getrf_task, &getrf_params[d]);
    //Queue task into the graph
    iris_graph_task(graph, getrf[d], iris_default, NULL);
    //---TRSM-TOP TASK---
    n = d + 1
    iris_task_create_perm(&trsm_top[d]);
    //Defining dependencies of the trsm-top tasks
    iris_task_depend( trsm_top[d], 1, &getrf[d]);
    trsm_top_params[d].M       = TILE_SIZE;
    trsm_top_params[d].LDA_cpu = SIZE;
    trsm_top_params[d].LDA_cpu = TILE_SIZE;
    ...
    trsm_top_params[d].A_cpu   = &A[( (d * TILE_SIZE) * LDA ) + ( d * TILE_SIZE )];
    trsm_top_params[d].A_gpu   = A_iris;
    trsm_top_params[d].B_cpu   = &B[( (n * TILE_SIZE) * LDA ) + ( d * TILE_SIZE )];
    trsm_top_params[d].B_gpu   = B_iris0;
    iris_task_host(trsm_top[d], trsm_task, &trsm_top_params[d]);
    iris_graph_task(graph, trsm_top[d], iris_default, NULL);
    //---TRSM-LEFT TASK---
    m = d + 1
    iris_task_create_perm(&trsm_left[d])
    iris_task_depend( trsm_left[d], 1, &getrf[d]);
    trsm_left_params[d].M       = TILE_SIZE;
    trsm_left_params[d].LDA_cpu = SIZE;
    trsm_left_params[d].LDA_cpu = TILE_SIZE;
    ...
    trsm_left_params[d].A_cpu   = &A[( (d * TILE_SIZE) * LDA ) + ( d * TILE_SIZE )];
    trsm_left_params[d].A_gpu   = A_iris;
    trsm_left_params[d].B_cpu   = &B[( (d * TILE_SIZE) * LDA ) + ( m * TILE_SIZE )];
    trsm_left_params[d].B_gpu   = B_iris1;
    iris_task_host(trsm_left[d], trsm_task, &trsm_left_params[d]);
    iris_graph_task(graph, trsm_left[d], iris_default, NULL);
    //---GEMM TASK---
    iris_task_create_perm(&gemm[d]);
    brisbane_task gemm_dep[] = { trsm_top[d], trsm_left[d] };
    iris_task_depend( gemm[d], 2, gemm_dep);
    gemm_params[d].M       = SIZE - ( m * TILE_SIZE);
    gemm_params[d].LDA_cpu = SIZE;
    ...
    gemm_params[d].A_cpu   = &A[( (d * TILE) * LDA ) + ( m * TILE )];
    gemm_params[d].B_cpu   = &A[( (n * TILE) * LDA ) + ( d * TILE )];
    gemm_params[d].C_cpu   = &A[( (n * TILE) * LDA ) + ( m * TILE )];
    gemm_params[d].A_gpu   = B_iris1;
    gemm_params[d].B_gpu   = B_iris0;
    gemm_params[d].C_gpu   =  C_iris;
    iris_task_host( gemm[d], gemm_task, &gemm_params[d] );
    iris_graph_task(graph, gemm[d], iris_default, NULL);
}
iris_graph_submit(graph, iris_default, 1);
```

Fig. 3. LU factorization code using IRIS.

The implementation of our LU factorization consists of four different tasks (Fig. 4): (1) GETRF, in which we compute a no-pivoting LU factorization on the top-left corner matrix TILE; (2) TRSM-top, in which we compute the level-3 BLAS TRSM routine by using the lower side of the LU factorization computed in the previous task as the input matrix (A in Fig. 4) and the rectangular tile located at the right of the LU matrix as the output matrix (B in Fig. 4); (3) TRSM-left, in which we compute the same level-3 BLAS operation used in the previous task but on a different part of the matrix by using the upper side of the LU matrix computed by the first task (GETRF) as input and a set of square tiles located under the lower side of the LU matrix as output (B_0, B_1, B_2, and B_3 in Fig. 4); and (4) GEMM, in which we compute a matrix-matrix multiplication by using the output of the two previous tasks as input and the remaining matrix parts as output. We compute all the previous tasks until the entire matrix is computed (Figs. 2 and 4 [left]).

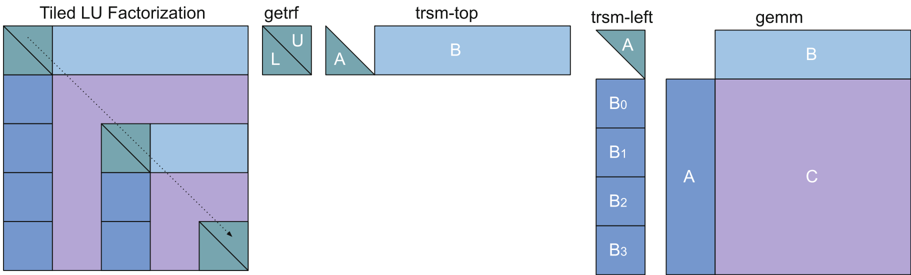

Fig. 4. Tasks of the LU decomposition implementation.

4.1 Memory Management

Our goal is to maximize the use of both the CPU and the GPU. In general, problems with larger tile sizes achieve relatively higher performance during computation. Knowing this, instead of using square tiles (Fig. 2), we decided to use rectangular tiles when possible. The only exception is the TRSM-left task/operation (Fig. 4), in which the rectangular tile is divided into a set of square tiles. This decomposition, which is carried out internally in the TRSM-left task at the implementation level, is necessary because this particular operation requires the vertical dimensions of both A (input) and B (output) matrices to be the same.[4]

In our code, we use one pointer (A in Fig. 3) to allocate the matrix to be factorized. Additionally, we create four different memory spaces that correspond to the memory computed (tiles) from each of the tasks. This way we can define the two-level memory space, one used at the algorithm (IRIS) level (A pointer) and one used at the implementation (task) level (A_iris, B_iris0/1, and C_iris).

[4] http://www.netlib.org/lapack/explore-html/db/def/group__complex__blas__level3_gaf33844c7fd27e5434496d2ce0c1fc9d4.html.

4.2 Tasking

As we described above, the idea is to maximize the use of both the CPU and the GPU. To do that, apart from carrying out the matrix decomposition illustrated in Fig. 4, we use multithreaded (CPU and GPU) computations to exploit the parallelism at both the algorithm and the implementation levels.

In our code, basically every task corresponds to one LAPACK or BLAS routine. As parameters, we need the same parameters that are described in the standard specification of these math libraries, so we can then see the tasks as a wrapper to a standard linear algebra library. For convenience, we implemented a different C structure data type per LAPACK or BLAS routine, which is then used to pass the arguments from the algorithm level to the implementation level.

Although it is well known that LU factorizations do not perform well for small matrices on GPUs, it is difficult to know which platform (i.e., CPU or GPU) is best suited for the rest of the tasks. This is particularly challenging when the workload and number of operations (i.e., size of the tiles) in each of the tasks change along the execution. Another important factor to consider is the differences of the components in our heterogeneous systems and the connections between them. Fortunately, in IRIS, one task can be run on either the CPU or the GPU; in other words, we can decide which architecture to use depending on the size of the tile or other factors. We implemented several approaches for each of the tasks using CPU-only, GPU-only, and CPU-GPU methods (Fig. 5).

Next, we explain the main characteristics of the different implementations of each task.

GETRF. The computation of the LU (no pivoting) factorization is carried out on the CPU. Although, we do not perform a GPU computation in this task, we can make some computationally expensive memory transfers between CPU and GPU, such as transferring the B matrices used by the TRSM tasks (TRSM-top and TRSM-left in Fig. 4) while LU factorization is being computed. So, we have two different implementations: (1) one in which we only compute the LU factorization on the CPU and (2) one in which we simultaneously compute the LU factorization, perform the CPU-to-GPU memory transfers for B matrices used by TRSM tasks, and perform the CPU-to-GPU transfer of the factorization output because this is also used by TRSM tasks.

TRSM-Top. Three different variants of TRSM-top were implemented: (1) a CPU version in which we make use of the TRSM routine within the CPU vendor libraries (e.g., IBM ESSL on Summit and Intel MKL on Oswald), (2) a GPU version in which we compute both a cuBLAS call for the TRSM computation and CPU-GPU memory copies for the input (from CPU to GPU) and output (from GPU to CPU), and (3) an optimization of the GPU version in which we compute a cuBLAS TRSM call and a GPU-CPU memory copy to transfer the result of the cuBLAS routine from GPU to CPU. In the last implementation, we do not carry out the CPU-to-GPU communication because this is performed in the GETRF task.

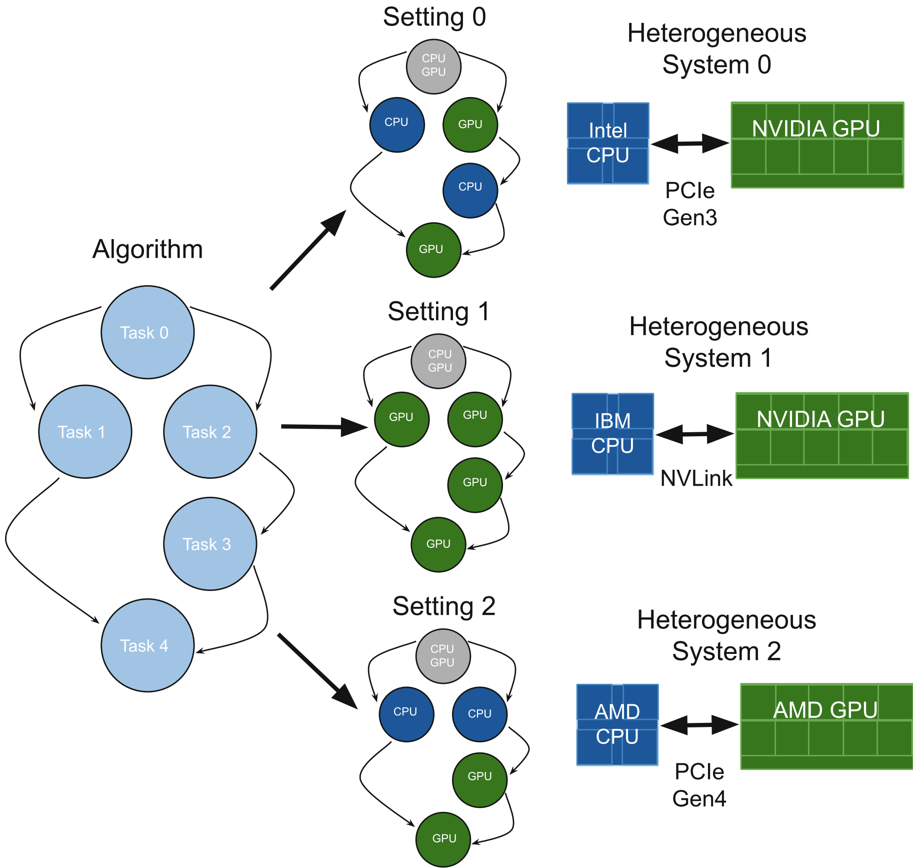

Fig. 5. The IRIS algorithm-implementation partition.

TRSM-Left. Two different versions of TRSM-left were implemented here: (1) a CPU implementation in which we compute the TRSM routine of the IBM ESSL library on Summit and the Intel MKL library on Oswald and (2) a GPU code in which we compute the CPU-GPU memory copies necessary to transfer the input to GPU memory and the output to CPU memory after computing the cuBLAS TRSM routine on the GPU.

4.3 GEMM

We implemented three different variants of GEMM: (1) a CPU code that uses CPU vendor libraries; (2) a GPU code in which the C matrix is transferred from CPU/GPU to GPU/CPU before/after the GPU computation of GEMM (cuBLAS), the A matrix is transferred from CPU to GPU before the computation, and the B matrix (output of TRSM-top) is already in GPU memory; and (3) an optimized GPU implementation in which only the A matrix is transferred from CPU to GPU before the computation of GEMM on the GPU, and the

Table 1. Summit and Oswald hardware specifications.

Name	Summit	Oswald
CPU Architecture	IBM Power 9	Intel Xeon E5-2683 v4
Frequency	3,800 MHz	2,100 MHz
Cores	22	32
Memory	512 GB	256 GB
Compiler	GCC 8.3.1	GCC 11.1.0
LAPACK/BLAS	ESSL	MKL
GPU Architecture	NVIDIA (Volta) V100	NVIDIA (Pascal) P100
Frequency	1,455 MHz	1,126 MHz
CUDA Cores	5,120	3,584
SM/CU Count	80	60
GPU-to-CPU Comm	NVLink 2.0 (50 GB/s)	PCIe Gen3 (16 GB/s)
Shared Memory	up to 96 KB per SM	64 KB per SM
L1	up to 96 KB per SM	64 KB per SM
L2	6,144 KB (unified)	4,096 KB (unified)
Memory	HBM2 16 GB	HBM2 12 GB
Bandwidth	900 GB/s	549 GB/s
Compiler	NVCC v11.0.221	NVCC v11.0.194
BLAS	cuBLAS	cuBLAS

top-left tile of the C matrix is transferred from GPU to CPU after the computation of GEMM. In the last implementation, we transfer the whole matrix to be factorized from CPU to GPU at the very beginning of the execution. Although this can be time consuming, it is only done once and has important implications for the overall performance (see Sect. 5).

5 Performance Analysis

This section describes the performance analysis of our code and the different variants/optimizations implemented. For a test case, we used a $16,384 \times 16,384$ matrix with a tile size of 512×512. We used two different heterogeneous systems for our analysis—Summit and Oswald (see Table 1 for the hardware features).

5.1 GETRF

For the GETRF task, we used Intel MKL's *LAPACKE_mkl_dgetrfnpi* routine on Oswald, whereas we used our own code on Summit because the IBM ESSL library does not have a routine for the non-pivoting LU factorization. In terms of performance, the optimized vendor library (i.e., Intel MKL on Oswald) achieved

much better performance (about 48 GFLOP/s) compared to our own implementation on Summit (15 GFLOP/s). However, as we describe below, this did not have a significant impact on the overall performance.

When overlapping communication with computation (i.e., to transfer the B matrix used by TRSM tasks and the output of the factorization from CPU to GPU), we see a fall in performance when compared to the CPU-only implementation, and we achieve an overall performance of 11 GFLOP/s on Summit and 42 GFLOP/s on Oswald.

5.2 TRSM-top

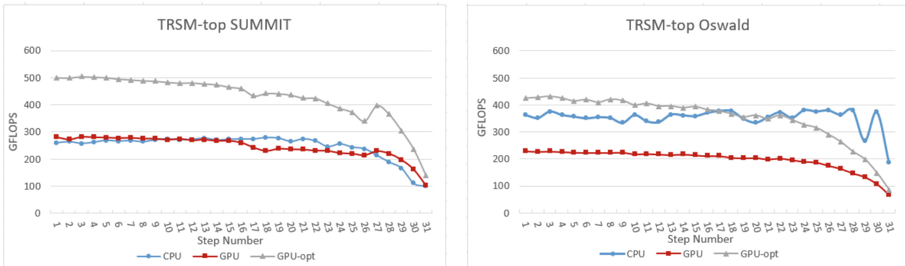

Fig. 6. TRSM-top performance.

Figure 6 illustrates the performance reached by the TRSM-top task on Summit and Oswald. As expected, the GPU-optimized implementation reaches the highest performance on both systems, at least in the first steps. On Summit, the performance of this implementation is considerably higher than the other two implementations—about 2× higher in some cases. In fact, this is the fastest implementation for all steps on Summit. On Oswald, the performance of the GPU-optimized implementation is higher than the CPU implementation (second fastest implementation) in the first steps; however, the CPU implementation is the faster one in the last steps, in which the computational cost and parallelism of computing TRSM is much lower. Although Summit has a much faster CPU-GPU connection (NVLink), a larger number of CPU cores and a more similar performance (GFLOP/s) between CPU and GPU makes the CPU implementation on Oswald faster than the GPU-optimized implementation for the last steps.

As shown, although the GPU-optimized implementation is the best choice for Summit, Oswald benefits from a heterogeneous approach (task) in which the GPU-optimized implementation is used during the first steps of the algorithm, and the CPU implementation is computed in the last steps.

5.3 TRSM-Left

Although we use the same level-3 BLAS operation from the previous task, we achieve very different performance results owing to the different matrix decomposition required by this operation. Again, as we can see in Fig. 7, the performance

varies significantly depending on the target platform. While the performance of the GPU implementation is lower than that of the CPU implementation on Oswald, we see the opposite scenario for Summit, where the GPU implementation is considerably faster than the CPU implementation.

As was the case for the TRSM-top task, we also need a different configuration here depending on the target platform.

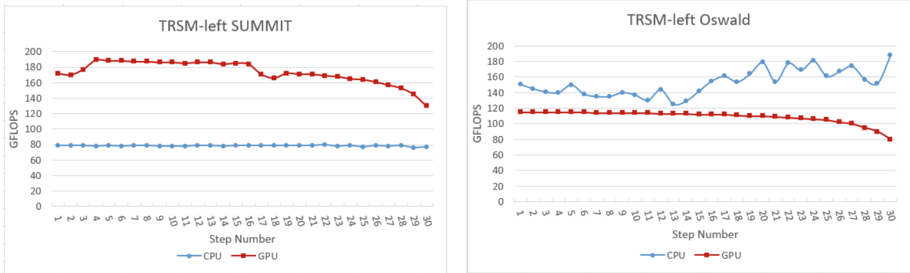

Fig. 7. TRSM-left performance.

5.4 Join TRSM-Top and TRSM-Left

For Oswald, we can see that although the GPU implementation is better for TRSM-top, the CPU implementation is the better choice for TRSM-left. Because both tasks are totally independent, this opens an opportunity for better performance on Oswald by computing both tasks in parallel using the more suitable implementation for each task. As shown Fig. 8, joining both tasks enables significant speedup.

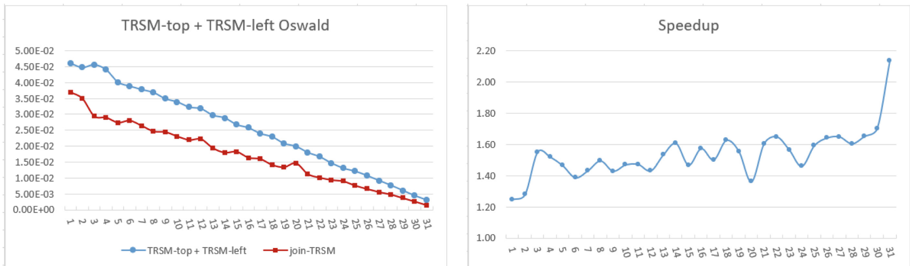

Fig. 8. Left: Time (s) of TRSM-top + TRSM-left and join-TRSM. Right: Join-TRSM speedup.

5.5 GEMM

Unlike the other two tasks, for GEMM we see the same behavior in both heterogeneous systems (Fig. 9). The GPU-optimized implementation has proven to be the fastest implementation in both systems and is 6×–8× faster than the second-fastest approach. As expected, the performance decreases along the execution; this can be seen in the other tasks too, in which the computational cost and the parallelism are much lower in the last steps than in the first steps of the algorithm.

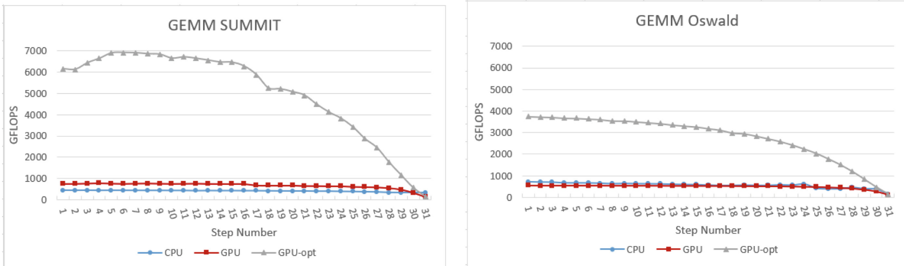

Fig. 9. GEMM performance.

5.6 Overall Performance

Here, we evaluate the overall performance for the different task implementations, and we start with the CPU-only implementations. On Oswald, we achieved an overall performance of 340 GFLOP/s, whereas on Summit we achieved 120 GFLOP/s. The relatively poor performance on Summit is from our implementation of the GETRF task.

Next, we evaluate using both the CPU and the GPU. For this, GETRF is computed on the CPU, and the rest of the tasks are computed on the GPU. We do not overlap computation with memory transfers at this level. On Oswald, we achieved 225 GFLOP/s, whereas on Summit, we achieved 511 GFLOP/s. Here, Summit's faster CPU-GPU connection results in better performance. The lower CPU-GPU bandwidth on Oswald has a negative impact on performance, and the overall performance is lower than using only the CPU.

Moving on, we focus on the overall performance impact of the different optimizations implemented in the tasks. We start with Oswald. By overlapping the LU factorization with CPU-GPU communication in the GETRF task, and by using the GPU to compute the TRSM-top task, we increased performance to 235 GFLOP/s. Also, by running TRSM-left on the CPU instead of the GPU, we achieved 275 GFLOP/s. As shown in Fig. 9, the most important optimization consists of moving the whole matrix from CPU to GPU at the very beginning. Using the GPU-optimized implementation in the GEMM task, we increased the

overall performance to 652 GFLOP/s. Finally, we conducted the last optimization, which consists of joining TRSM-left (computed on the CPU) and TRSM-top (computed on the GPU), which increased the overall performance to 700 GFLOP/s.

Next, we focus on Summit. By overlapping the LU factorization with the CPU-GPU communication in the GETRF task, we increased the overall performance to 546 GFLOP/s. As shown in Fig. 7, using the CPU is not faster than using the GPU for TRSM-left. With that in mind, the optimizations on Oswald for TRSM tasks are not beneficial on Summit, but computing both tasks on the GPU is better. Finally, using the GPU-optimized implementation of the GEMM task increases the overall performance considerably—achieving 1,972 GFLOP/s.

6 Related Works

Recently, we have seen important progress toward performance portability. Some examples are the C++ template metaprogramming libraries Kokkos [15] and RAJA [16]. These libraries can build different binaries that target different architectures from one source code. However, they cannot use more than a single architecture at a time.

Using CPUs and GPUs for HPC codes has been widely studied [13,17,18]. Since OpenMP 4.0, it is possible to use GPU offloading in OpenMP codes. Valero-Lara et al. [19] used OpenMP 4.5 to implement a heterogeneous version of the TRSM level-3 BLAS routine and achieved good performance on one node of Oak Ridge National Laboratory's Summit supercomputer. One important reference for heterogeneous linear algebra codes is the MAGMA [20] library. MAGMA, offers multiple heterogeneous implementations for several LAPACK routines. Unfortunately, there is not an implementation for our test case.

In contrast, our work focuses on the potential benefits of using IRIS for performance portability on heterogeneous HPC architectures. To the best of our knowledge, this is the first time that a portable and heterogeneous LU factorization code (i.e., IRIS) has been implemented and analyzed.

7 Final Remarks and Future Directions

The difference in current and upcoming heterogeneous systems hinders implementation of HPC codes. By using IRIS, we not only make this effort more affordable, but we can also implement portable and heterogeneous HPC codes by separating the algorithm design from the implementation. First, we described our algorithm using tasks + dependencies. After that, we had to decide which code to use in each of the tasks. A specific and different setting must be used depending on the target platform. In this paper, we were able to optimize one of the most important HPC algorithms, the LU factorization, on two different heterogeneous platforms with minimal modifications to the code.

However, a more thorough and computationally expensive study is required to evaluate which code/implementation should be used in each of the tasks. In

the future, we plan to implement alternatives that enable one to compute the setting in an automatic and computationally cheaper manner. We also want to extend this effort to other HPC applications and heterogeneous systems.

References

1. Bellavia, S., Morini, B., Porcelli, M.: New updates of incomplete LU factorizations and applications to large nonlinear systems. Optim. Methods Softw. **29**(2), 321–340 (2014). https://doi.org/10.1080/10556788.2012.762517
2. Eickhoff, K.M., Engl, W.L.: Levelized incomplete LU factorization and its application to large-scale circuit simulation. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **14**(6), 720–727 (1995). https://doi.org/10.1109/43.387732
3. Luciani, X., Albera, L.: Joint eigenvalue decomposition of non-defective matrices based on the LU factorization with application to ICA. IEEE Trans. Signal Process. **63**(17), 4594–4608 (2015). https://doi.org/10.1109/TSP.2015.2440219
4. Kudo, S., Nitadori, K., Ina, T., Imamura, T.: Implementation and numerical techniques for one eflop/s HPL-AI benchmark on fugaku. In: 11th IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA@SC 2020, Atlanta, GA, USA, 13 November 2020, pp. 69–76. IEEE (2020). https://doi.org/10.1109/ScalA51936.2020.00014
5. Gan, X., et al.: Customizing the HPL for china accelerator. Sci. China Inf. Sci. **61**(4), 042 102:1-042 102:11 (2018). https://doi.org/10.1007/s11432-017-9221-0
6. Kim, J., Lee, S., Johnston, B., Vetter, J.S.: IRIS: a portable runtime system exploiting multiple heterogeneous programming systems. In: Proceedings of the 25th IEEE High Performance Extreme Computing Conference, ser. HPEC 2021, pp. 1–8 (2021)
7. Valero-Lara, P., Catalán, S., Martorell, X., Usui, T., Labarta, J.: 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)
8. Valero-Lara, P., Catalán, S., Martorell, X., Labarta, J.: BLAS-3 optimized by ompss regions (lass library). In: 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, Pavia, Italy, 13–15 February 2019, pp. 25–32. IEEE (2019)
9. Dongarra, J.J., et al.: PLASMA: parallel linear algebra software for multicore using openmp. ACM Trans. Math. Softw. **45**(2), 16:1-16:35 (2019)
10. Valero-Lara, P., Martínez-Pérez, I., Sirvent, R., Martorell, X., Peña, A.J.: NVIDIA GPUs scalability to solve multiple (batch) tridiagonal systems implementation of cuThomasBatch. In: Wyrzykowski, R., Dongarra, J., Deelman, E., Karczewski, K. (eds.) PPAM 2017. LNCS, vol. 10777, pp. 243–253. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78024-5_22
11. Valero-Lara, P., Martínez-Pérez, I., Sirvent, R., Martorell, X., Peña, A.J.: cuThomasBatch and cuThomasVBatch, CUDA routines to compute batch of tridiagonal systems on NVIDIA GPUs. Concurr. Comput. Pract. Exp. **30**(24), e4909 (2018)
12. Valero-Lara, P., Pinelli, A., Favier, J., Matias, M.P.: Block tridiagonal solvers on heterogeneous architectures. In: IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, ser. ISPA 2012, pp. 609–616 (2012)
13. Valero-Lara, P., Pinelli, A., Prieto-Matias, M.: Fast finite difference Poisson solvers on heterogeneous architectures. Comput. Phys. Commun. **185**(4), 1265–1272 (2014)

14. Demmel, J.W., Gilbert, J.R., Li, X.S.: An asynchronous parallel supernodal algorithm for sparse gaussian elimination. SIAM J. Matrix Anal. Appl. **20**(4), 915–952 (1999)
15. Trott, C.R., et al.: Kokkos 3: programming model extensions for the exascale era. IEEE Trans. Parallel Distributed Syst. **33**(4), 805–817 (2022). https://doi.org/10.1109/TPDS.2021.3097283
16. Beckingsale, D., Hornung, R.D., Scogland, T., Vargas, A.: Performance portable C++ programming with RAJA. In: Hollingsworth, J.K., Keidar, I. (eds.) Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, 16–20 February 2019, pp. 455–456. ACM (2019)
17. Valero-Lara, P., Jansson, J.: Heterogeneous CPU+GPU approaches for mesh refinement over lattice-boltzmann simulations. Concurr. Comput. Pract. Exp. **29**(7), e3919 (2017)
18. Valero-Lara, P., Igual, F.D., Prieto-Matías, M., Pinelli, A., Favier, J.: Accelerating fluid-solid simulations (lattice-boltzmann & immersed-boundary) on heterogeneous architectures. J. Comput. Sci. **10**, 249–261 (2015)
19. Valero-Lara, P., Kim, J., Hernandez, O., Vetter, J.S.: Openmp target task: tasking and target offloading on heterogeneous systems. In: Chaves, R., et al. (eds.) Euro-Par 2021. LNCS, vol. 13098, pp. 445–455. Springer, Cham (2021). https://doi.org/10.1007/978-3-031-06156-1_35
20. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Technical report, 2008-01 (2008)