# Addressing Irregular Patterns of Matrix Computations on GPUs and Their Impact on Applications Powered by Sparse Direct Solvers

Ahmad Abdelfattah
*Innovative Computing Laboratory*
*University of Tennessee*
Knoxville, USA
ahmad@icl.utk.edu

Pieter Ghysels
*Scalable Solvers Group*
*Lawrence Berkeley National Laboratory*
Berkeley, USA
pghysels@lbl.gov

Wajih Boukaram
*Scalable Solvers Group*
*Lawrence Berkeley National Laboratory*
Berkeley, USA
wajih.boukaram@lbl.gov

Stanimire Tomov
*Innovative Computing Laboratory*
*University of Tennessee*
Knoxville, USA
tomov@icl.utk.edu

Xiaoye Sherry Li
*Scalable Solvers Group*
*Lawrence Berkeley National Laboratory*
Berkeley, USA
xsli@lbl.gov

Jack Dongarra
*Innovative Computing Laboratory*
*University of Tennessee*
Knoxville, USA
dongarra@icl.utk.edu

***Abstract*—Many scientific applications rely on sparse direct solvers for their numerical robustness. However, performance optimization for these solvers remains a challenging task, especially on GPUs. This is due to workloads of small dense matrices that are different in size. Matrix decompositions on such irregular workloads are rarely addressed on GPUs.**

**This paper addresses irregular workloads of matrix computations on GPUs, and their application to accelerate sparse direct solvers. We design an interface for the basic matrix operations supporting problems of different sizes. The interface enables us to develop *irrLU-GPU*, an LU decomposition on matrices of different sizes. We demonstrate the impact of irrLU-GPU on sparse direct LU solvers using NVIDIA and AMD GPUs. Experimental results are shown for a sparse direct solver based on a multifrontal sparse LU decomposition applied to linear systems arising from the simulation, using finite element discretization on unstructured meshes, of a high-frequency indefinite Maxwell problem.**

***Index Terms*—Irregular computational workloads, GPU Computing, LU factorization, multifrontal solvers, sparse direct solvers**

## I. Introduction and Motivation

Sparse direct solvers are widely used in both academia and industry for the solution of systems of linear equations. Direct methods, as based on lower-upper (LU) decomposition, i.e., Gaussian elimination, or QR decomposition, have the benefit that they are robust and require relatively little tuning by the user. This is in contrast to iterative solvers such as (preconditioned) Krylov methods, and multigrid and domain decomposition solvers and preconditioners, which require tuning of, e.g., the Krylov subspace size, the stopping criterion, the smoother, the restriction and prolongation operators, etc. Whereas iterative methods can have unpredictable convergence, direct methods solve linear systems with a predictable number of operations and memory usage. However, compared to most iterative methods, sparse direct solvers can require asymptotically more memory and floating point operations. State-of-the-art academic approaches for the solution of large scale linear systems often rely on a complex composition of methods, for instance using a direct solver as the sub-domain solver in a domain decomposition solver, or as coarse grid correction solver in multigrid. Another benefit of factorization based methods is that the factorization of the operator can be reused multiple times for the solution of different linear systems, with the same operator but different right hand sides, for instance from multiple source terms.

In this work, we focus on performance-critical components of a sparse direct solver that is used for the solution of high frequency wave equations, for which it is well-known that constructing efficient preconditioners is very challenging, and which are hence typically solved using direct methods. For instance the indefinite Maxwell equation with large wavenumbers results in highly indefinite linear systems, which are hard to precondition. The performance of direct solvers on the other hand is not so sensitive to the spectrum of the operator.

We solve the sparse linear systems resulting from the finite element discretization of the indefinite Maxwell problem using a multifrontal [1], [2] sparse LU solver, using a symmetrized sparsity pattern. In the multifrontal method, rows and columns with equivalent sparsity structure are grouped together in so-called supernodes. The solver then constructs a dense matrix – referred to as a frontal matrix or simply a front – for each supernode. The sizes of these fronts range from very small ($< 8 \times 8$) all the way up to the size of the largest separator of the graph corresponding to the sparsity pattern of the input matrix. By defining these dense blocks in the sparse triangular factors, the numerical factorization can be implemented using higher level BLAS routines, which greatly improves the performance compared to the scalar

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

implementations. For irregular problems, such as those from the unstructured discretization of partial differential equations using, for instance, finite elements or finite volumes, the distribution of the frontal matrix sizes depends on the initial sparsity structure of the matrix, and on the ordering of the matrix. A fill-reducing ordering is typically applied to reduce the total number of nonzero elements in the sparse triangular factors. This ordering is computed using graph heuristics, often relying on graph partitioning codes.

One of the performance bottlenecks in sparse direct solvers is the irregular patterns of relatively small dense matrices for which an LU or QR decomposition needs to be performed. This is often called *batch computation* in the literature. The importance and the growing community demand for high performance batch algorithms have driven many vendors of HPC architectures to provide dedicated kernels for these workloads in their numerical software. Libraries like Intel's Math Kernel Library (MKL) [3], NVIDIA's cuBLAS [4] and AMD's rocBLAS [5] all provide different types of batch subprograms. However, most of the efforts from the vendors and the research community focus on uniform (or semi-uniform) distributions of the matrix sizes within a batch. Both cuBLAS and rocBLAS provide interfaces that assume one batch of matrices having the same dimensions. The Intel MKL library provides a different interface that assumes groups (or sub-batches) of matrices, such that matrices belonging to the same group have the same dimensions. This is the only vendor-supplied interface that can be used to address irregular sizes. The currently available interfaces from GPU vendors do not meet the requirements of the sparse solver described above. The only resort would be to use concurrent kernel launches using parallel streams, which often performs very poorly, as we demonstrate later in the paper.

The pattern of sizes arising from sparse direct solvers depends on the sparsity pattern of the input matrix, so no pre-assumptions can be made. In addition, figuring out groups of uniform sizes, if any, can be challenging, especially if GPUs are considered. This is why we target a "flat interface", assuming one group of matrices of arbitrary sizes. For the LU decomposition, very few efforts addressed the case for very small matrices [6], for example up to $32 \times 32$. This is a relatively simple case, because all the matrices can be easily cached (and probably padded) in the shared memory, and a column-wise decomposition is used. Our application requires both small (e.g. $< 8 \times 8$) and large sizes, well beyond $1000 \times 1000$. To the best of our knowledge, no effort in the literature addresses the LU decomposition on a batch of matrices whose sizes are completely arbitrary.

The proposed solution (irrLU-GPU) works on any pattern of sizes. The sizes can be as small as $1 \times 1$, and as large as the GPU memory affords. This is particularly challenging to achieve on GPUs because of their programming model, which often favors uniform patterns of computation. One of the main contributions of irrLU-GPU is an interface concept that is able to efficiently address generic irregularity, meaning that there are no assumptions about the range or the distribution of the sizes. The interface is used to describe the LU decomposition with respect to the largest matrix in the batch. Since smaller matrices can be fully factorized at different stages, we develop a technique called *Dynamic Compute-workload Inference (DCWI)*, which uses the information passed through the interface to infer, on the matrix level, the exact amount of computation to be done. Both the interface and DCWI are applied to all the computational steps of the LU decomposition, such as matrix multiply on different sizes (irrGEMM) and triangular solve (irrTRSM). The latter is another contribution in the paper that outperforms a similar routine in the MAGMA library [7], both in performance and numerical accuracy. We believe that the proposed interface can be applied to a wide range of numerical algorithms that need to be simultaneously applied on matrices of different sizes.

### A. *Summary of Contributions*

1) We present a systematic way for designing numerical software addressing irregular matrix computations on GPUs. The two main design concepts are 1) an expanded interface for linear algebra kernels, and 2) the DCWI layer mentioned before.
2) The two design concepts are used to design irrLU-GPU, an LU decomposition that supports irregular workloads of independent matrices. Unlike previous solutions that presume a certain limited range, irrLU-GPU has no limitations on the range or the distribution of the sizes in a given workload.
3) We extend an existing multifrontal sparse direct solver to take advantage of irrLU-GPU, and show its performance impact on a representative application based on a high-frequency indefinite Maxwell problem.

## II. Related Work

There exists a number of sparse direct solvers, including SuperLU_Dist [8], SuiteSparse (UMFPACK [9] and Cholmod [10]), PaStiX [11], STRUMPACK [12], [13], MUMPS [14] and WSMP [15]. Out of these, SuperLU_Dist, Cholmod (only SPD), PaStiX and STRUMPACK have GPU support. SuperLU_Dist, PaStiX and Cholmod – supernodal but not multifrontal methods – have more complicated data dependencies than the multifrontal solver STRUMPACK. The multifrontal algorithm lends itself better to the use of batch routines on GPUs. However, STRUMPACK's current implementation uses a naive batch kernel which is restricted to matrix blocks smaller than $32 \times 32$, still leading to significant kernel launch overhead.

There is a wide range of applications that often require matrix computations on independent small problems, such as astrophysics [16], quantum chemistry [17], and applications benefiting from sparse direct solvers, which are the main motivation behind this work. Most of the research and development efforts have focused on batches of small matrices having the same size, since the then-existing interfaces and programming models can be extended to support uniform patterns relatively easily. Vendors like NVIDIA and AMD provide batch routines in their cuBLAS [4] and rocBLAS [5] libraries, respectively. They adopt a flat interface which assumes one batch of matrices having the same dimensions. Other GPU-centric libraries like MAGMA [7] adopt a similar interface. Another interface that gained attention is available in the MKL library [3], which adds one level of parallelism over the flat interface, as it assumes groups of uniform batches [18]. This interface can support completely irregular matrix patterns, if the extreme case of one matrix per group is used. Libraries such as Kokkos Kernels and MKL use interleaved data layouts for batch kernels on small matrices, which provides a performance advantage for SIMD arhchitectures [19]. All of these interfaces have a common issue, which appears when we consider an algorithm consisting of a sequence of calls to batch routines. Arithmetic operations on pointer arrays, especially on GPUs, can be daunting. In the case of irregular sizes, integer arithmetic on the sizes is also required.

Few efforts have discussed irregular batches using GPUs. To the best of our knowledge, no previous contributions have addressed a standard and systematic way of dealing with irregular matrices on GPUs, especially matrix decompositions. While some efforts have focused on fundamental operations like matrix multiply [20], addressing higher level algorithms using these building blocks is a different challenge. Previous contributions focused on a specific range of sizes, e.g. $32 \times 32$ [6], for which a simple design with implicit padding can be developed. In our design of irrLU-GPU, we have no limitations on the size distribution. A single batch can have any range of sizes as long as all matrices fit in the global memory of the GPU.

The approach of designing a single custom kernel that works for any size has very slim chances of achieving any high performance, since it will have to process large matrices with very little use of cache-blocking techniques on the GPU. We believe that the factorization must be broken down into its building blocks, like matrix multiply, triangular solve, and submatrix (panel) decomposition. These building blocks must be aware of the size irregularity at run time, and that some smaller matrices may have been fully decomposed while other matrices yet have some computational workloads. The design concepts behind irrLU-GPU address these challenges on GPUs, and we believe they are also applicable to other numerical algorithms for dense matrices.

## III. Algorithmic Background

### A. Multifrontal Sparse Direct Solvers

We consider the solution of a sparse linear system $Ax = b$ with $A \in \mathbb{C}^{N \times N}$. To achieve this, we compute a decomposition of $A$ as $P(D_r A D_c Q)P^T = LU$, where $P$ and $Q$ are permutation matrices, $D_r$ and $D_c$ are diagonal scaling matrices, and $L$ and $U$ are sparse lower and upper triangular factors. The permutation $P$, which aims to reduce the number of nonzeros in the triangular factors, is computed using the nested dissection algorithms from the METIS [21] library. The optional permutation $Q$, and the scaling factors $D_r$ and $D_c$ are computed using the MC64 [22] matching code. The goal of $Q$ is to maximize the product of the diagonal elements, which are then scaled by $D_r$ and $D_c$ such that all diagonal entries are 1 and all off-diagonal entries are less than 1 in absolute value.

One can typically distinguish three separate phases for the direct solution of sparse linear systems:

1) Reordering and symbolic analysis
2) Numerical factorization
3) Solve using forward and backward substitution

In phase 1, the permutation and scaling vectors are computed, the sparsity pattern is analyzed, and data-structures are initialized to guide the numerical factorization phase. After computation of the sparse factorization (computationally the most expensive phase), a linear system can be solved efficiently by applying the permutations and scalings, and performing the sparse triangular solves with the $L$ and $U$ factors.

To illustrate the numerical factorization phase, we consider a nested dissection permutation $P$ with only two levels, and a single vertex separator:

$$PAP^T = \begin{bmatrix} A_1 & & X_{1S} \\ & A_2 & X_{2S} \\ X_{S1} & X_{S2} & S \end{bmatrix}. \tag{1}$$

The lower-right sub-block $S$ corresponds to a separator in the graph of $A$, effectively splitting the problem in two unconnected components, represented by $A_1$ and $A_2$. We can now construct three frontal matrices

$$F_1 = \begin{bmatrix} A_1 & \hat{X}_{1S} \\ \tilde{X}_{S1} & \end{bmatrix},\ F_2 = \begin{bmatrix} A_2 & \hat{X}_{2S} \\ \tilde{X}_{S2} & \end{bmatrix},\ F_0 = S, \tag{2}$$

where $\hat{X}_{1S}/\tilde{X}_{S1}$ is the matrix consisting of only the columns/rows of $X_{1S}/X_{S1}$ which contain nonzero elements. These fronts are put in a binary tree with $F_0$ as the root and $F_1$ and $F_2$ as the children. The numerical phase of the multifrontal $LU$ factorization algorithm then traverses this binary tree from the leaves to the root. At each front $F_\tau = [F_{11}\, F_{12};\, F_{21}\, F_{22}]$ (except the root), the following steps are performed:

- $P_\tau L_\tau U_\tau \leftarrow LU(F_{11})$
- $F_{12} \leftarrow U_\tau^{-1} L_\tau^{-1} P_\tau^T F_{12}$
- $F_{22} \leftarrow F_{22} - F_{21} F_{12}$

At the root front, only the first of these steps needs to be performed. The first step computes a dense $LU$ factorization with partial pivoting. Note that the pivoting is restricted to the diagonal blocks, but for most problems, especially when combined with the permutation $Q$, this is sufficient to ensure numerical stability, and it greatly simplifies the implementation. After these operations are applied to front $F_\tau$, the Schur complement $F_{\tau;22}$ is added into the parent frontal matrix. However, since the parent front is typically larger than the Schur complement of it's child, this requires a scatter operation.

The described approach can be generalized by recursively applying the nested dissection heuristic to the subdomains $A_1$ and $A_2$, leading to a binary tree, referred to as the assembly tree, with $\mathcal{O}(\log N)$ levels. Going down the tree from the root to the leaves, subdomains and separators become smaller, leading to smaller frontal matrices, while the tree becomes wider. Note that all fronts in a given level can be handled concurrently. Our GPU implementation traverses the tree level-by-level, from leaves to root, using batch algorithms for the dense linear algebra operations (LU, triangular solve and matrix multiplication) for all fronts on a given level. If the entire assembly tree does not fit in the device memory, then the factorization is split in multiple traversals of subtrees that do fit on the device. Likewise, for the distributed memory parallel code, the assembly tree is split in multiple subtrees, each of which is assigned to a single MPI rank and corresponding GPU, while the top $\log P$ levels of the tree are distributed using a 2D block cyclic layout and then processed using either ScaLAPACK (CPU-only) or SLATE [23].

## *B. Lower-upper (LU) Matrix Decomposition*

We consider the dense LU decomposition with partial pivoting ($P \times L \times U = A$), which is the standard decomposition method for solving a linear system of equations. The $L/U$ matrices are called the triangular factors of $A$, while $P$ is a permutation matrix that reflects the necessary row interchanges to maintain numerical stability. We briefly describe the main computational steps of the algorithm, which are shown in Figure 1 for a single iteration. At each iteration, the algorithm begins with a "panel decomposition" (yellow submatrix), producing the $L/U$ factors of the panel. The computational steps of the panel decomposition are discussed in Section IV-E. The panel undergoes some row exchanges during its decomposition, which must propagate to the submatrices to the left ($A_l$) and right ($A_{r1}$ and $A_{r2}$) of the panel. The next step is to solve a triangular system of equations ($L_{tr} \times X = A_{r1}$), where $L_{tr}$ is the triangular part of $L$, and where the solution $X$ overwrites the $A_{r1}$ matrix. The final step in the iteration is a rank-k update of the form ($A_{r2} = A_{r2} - L_{rect} \times A_{r1}$), where $L_{rect}$ is the lower rectangular part of $L$. The next iteration starts at the $A_{r2}$ submatrix, repeating the same steps. The total operation count (or floating point operations; FLOPs) of an LU decomposition on an $M \times N$ matrix is $MN^2 - \frac{N^3}{3} - \frac{N^2}{2} + \frac{5N}{6}$. In our performance measurements, we do not drop the low order terms of the expression since we are dealing with relatively small matrices.

## *C. Batch Matrix Computations*

The development of high performance numerical software for parallel independent problems has been an active topic of research. The main motivation behind such efforts is two fold. First, scientific applications such as computer vision, high order finite element methods (FEM) [24], and sparse direct solvers [25] often produce computational workloads that involve independent small matrix computations. Second, modern hardware architectures are becoming so increasingly parallel that they can comfortably manage two levels of parallelism, within each problem (fine-grain) and across problems (coarse-grain). While it is possible for existing programming models to support these workloads, e.g. by utilizing OpenMP, OpenACC, and concurrent stream execution, dedicated optimizations often reveal that there is room for more performance on such workloads. This is apparent in many vendor-supplied numerical libraries that now provide batch algorithms. The research community also contributed numerous efforts, some of which are application-driven [26] [25] [27], and some others targeting generic use [28] and standard interfaces [18].

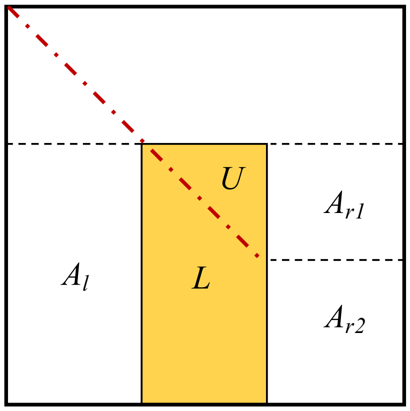


Fig. 1. The submatrices overwritten by one iteration of an LU decomposition of a dense matrix.

As mentioned in Section II, most of the research efforts on GPUs target matrices of the same size [29] [30], although some few exceptions exist [20]. Consider the cuBLAS interfaces shown in Figure 2 for the matrix multiply operation (regular and uniform-batch). By comparing the two interfaces, it is obvious that the matrix argument is promoted from a single pointer that is passed by value to an array of pointers that must exist in the GPU memory before the kernel is launched. This change has in fact noticeable consequences on developing algorithms that operate on submatrices. On a single matrix operation, the scalar pointer arithmetic can be easily inlined in the kernel launch itself. However, with a batch interface like Figure 2, the array of pointers must be updated either on the CPU and sent to the GPU, or on the GPU directly using a dedicated kernel. Both options have their respective overheads (CPU-GPU communication or launch overheads).

*1) Support for Nonuniform Batch Matrix Computations:* Multifrontal sparse direct solvers often encounter factorization workloads of matrices having different sizes. The range and distribution of such sizes depend on the size and the sparsity pattern of the input sparse matrix. Our target is to design a solution that supports these workloads without any limitations on the sizes of the multifrontal matrices. For arbitrary sizes, blocked decompositions are the only approach. The decomposition must be broken down into multiple computational stages (e.g. panel decompositions and updates) that operate on submatrices of different sizes. Since the CPU coordinates the kernel launches on the GPU, it needs to pass the necessary information about these submatrices (pointer, size, and leading dimension). A natural progression to the batch interface in Figure 2 would promote the sizes and the leading dimensions

```
/* regular gemm */
cublas<T>gemm(
        cublasHandle_t handle,
        cublasOperation_t transa, cublasOperation_t transb,
        int m, int n, int k,
        const T *alpha,
        const T *A, int lda,
        const T *B, int ldb,
        const T *beta,
        T       *C, int ldc);

/* batch gemm */
cublas<T>gemmBatched(
        cublasHandle_t handle,
        cublasOperation_t transa, cublasOperation_t transb,
        int m, int n, int k,
        const T *alpha,
        const T *Aarray[], int lda,
        const T *Barray[], int ldb,
        const T *beta,
        T       *Carray[], int ldc,
        int batch_size);
```

Fig. 2. Example interface of a uniform batch matrix multiply in cuBLAS. The type `T` represents the different data types supported by the library.

from scalar integers to arrays that must now reside on the GPU memory before launch. This extension further complicates submatrix computations in blocked algorithms, since both integer and pointer arithmetics are required. For example, consider that we would like to perform the LU decomposition on the second column on a group of matrices of different sizes. The pointer array must be updated to point to entry (1, 1) on every matrix (assuming c-style indexing). The dimension vectors must also decrease by one to reflect the size of the submatrix. Before every computational step (or kernel launch), the pointers and the sizes must be carefully updated. Such repetitive setup is undoubtedly daunting and costly. This is why we carefully study the interface of the building block operations in this paper. As an example, irrLU-GPU requires a matrix multiplication kernel that operates on a nonuniform batch of (sub)matrices. We use the codebase available in the open source MAGMA library [20] (version 2.6.1), but further improve the kernel interface to facilitate the development of a fully functional LU decomposition. The interface design is applicable to every computational step in the algorithm. It enables us to avoid numerous calls for auxiliary kernels performing pointer and integer arithmetics, which is costly in terms of development, launch overheads, and long-term software maintenance. The new interface, along with the proposed kernel semantics (i.e. the DCWI layer), also enables recursive algorithms on matrices of different sizes, which we use for the irrTRSM kernel contributed in this work.

## IV. Algorithmic Adaptation and Design

### A. New Interface and Semantics

One of the main contributions in this work is an expanded interface that enables the easy development of complex algorithms involving submatrix computations of different sizes. The interface completely avoids both pointer and integer arithmetic in the irrLU-GPU algorithm. Figure 3 shows an example interface of irrGEMM, one of the critical components in irrLU-GPU, to perform matrix multiply on a nonuniform set of matrices. The main idea is to embed, in a generic way, pointer/integer arithmetics in the interface itself. The obvious cost is the longer interface. However, we believe that the interface is easily understandable, and its benefit far outweighs the difficulties of using more BLAS-like interfaces like the one in [20].

The semantics of the interface and the operation itself are also different from the legacy BLAS API, and even those batch routines available in cuBLAS and rocBLAS. In such interfaces, the dimensions (e.g. `m`, `n`, `k`) represent both the sizes of the product being performed and the sizes of the matrices `A`, `B`, and `C`. Each matrix is represented by a scalar pointer and a leading dimension. The proposed interface has some differences in this regard. Below is some terminology that we use to describe the interface.

1) *Required Dimension(s)*: the interface uses a set of scalars (e.g. `m`, `n`, `k` in Figure 3) to describe the required operation to be performed on the batch. The required dimensions should be defined according to the largest matrix in the batch.
2) *Local Dimension(s)*: These are stored as integer arrays (`m_vec`, `n_vec`, and `k_vec`) whose individual entries define the dimensions of the matrices in the batch. These local dimensions are not changed throughout the computation.
3) *Pointer Offsets*: In addition to the local dimensions, each matrix is characterized by a pointer, a leading dimension, and a pair of *offset scalars* (e.g. $\mathtt{A_i}$ and $\mathtt{A_j}$) such that (`A[id] = Aarray[id] + Aj×lda_vec[id] + Ai, id∈[1:batch_size]`).
4) *Actual Workload*: Each developed kernel internally infers the actual sizes defining the local operation according to the required sizes, the local dimensions, and the pointer offsets. We call this step *Dynamic Compute-Workload Inference (DCWI)*, an integral part of the new interface.

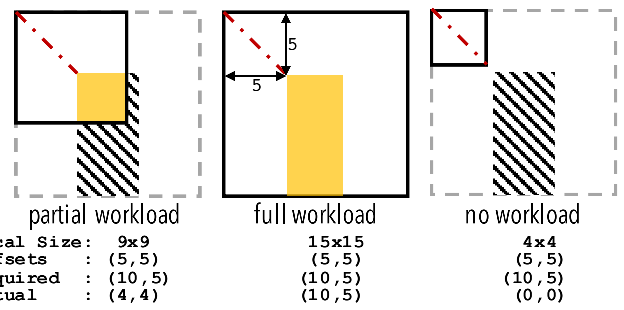
```
irr<T>gemm_GPU( /* handle or stream */
    trans_t transa, trans_t transb,
    int m, int n, int k, int* m_vec, int* n_vec, int* k_vec,
    const T alpha,
    const T *Aarray[], int Ai, int Aj, int* lda_vec,
    const T *Barray[], int Bi, int Bj, int* ldb_vec,
    const T *beta,
    T       *Carray[], int Ci, int Cj, int* ldc_vec,
    int batch_size);
```

Fig. 3. Example of the proposed interface for a nonuniform batch matrix multiply

Since we propose a new interface, an obvious consequence would be changing existing user codes in order to take advantage of this work. However, we emphasize that this is a relatively new area for GPUs. Apart from the MKL library, none of the vendors' numerical libraries (e.g. cuBLAS and rocBLAS) support non-uniform batch computations. Most of the existing codes either use custom kernels or use parallel streams.

## B. Dynamic Compute-Workload Inference (DCWI)

DCWI is a lightweight subkernel layer that significantly simplifies the description of matrix computations on independent problems of different sizes. In general, DCWI requires that an algorithm must be written according to the largest computational workload. For an LU decomposition with partial pivoting, this corresponds to $\max_{\texttt{id=1}}^{\texttt{batch\_size}} \min(\texttt{m\_vec[id]}, \texttt{n\_vec[id]})$. DCWI along with the new interface can then detect on-the-fly which matrices have been fully or partially decomposed. The actual workload, if any, is computed per matrix. If no workload is detected, the corresponding GPU threads perform no work at all. Consider the example shown in Figure 4, where three matrices of different sizes are factorized. For simplicity, we assume one thread block per matrix. We also assume a "blocked decomposition" that progresses by five columns at a time. In the given example, the algorithm would require three iterations in order to accomodate the largest $15 \times 15$ matrix. The figure shows the second iteration, where three different types of workloads are detected (full, partial, and none). Figure 5 shows the corresponding pseudo code for the DCWI layer. Note that without the proposed interface, DCWI would be replaced by multiple kernel calls performing pointer/integer arithmetic. It would also require additional workspaces to store new dimensions and pointers for submatrices. DCWI is adopted by all the computational steps in the LU decomposition.



|  | partial workload | full workload | no workload |
|---|---|---|---|
| Local Size: | 9x9 | 15x15 | 4x4 |
| Offsets : | (5,5) | (5,5) | (5,5) |
| Required : | (10,5) | (10,5) | (10,5) |
| Actual : | (4,4) | (10,5) | (0,0) |

Fig. 4. Dynamic compute-workload inference. The yellow areas represent the amount of workload per matrix in the second iteration of a blocked decomposition of three iterations.

The DCWI layer requires a careful implementation in order to guarantee correctness. Otherwise, memory faults occur and can be detected using the vendor's debugging tools. Inferring partial or no workload is kernel-specific and requires an understanding of the kernel semantics. As an example, for an operation like $C = A \times B$, the offsets ($\texttt{A}_\texttt{i}$, $\texttt{A}_\texttt{j}$) are compared against ($\texttt{m}$, $\texttt{k}$). However, if the operation is changed to $C = A^T \times B$, they should be compared against ($\texttt{k}$, $\texttt{m}$). A similar situation for triangular solves would be solving $AX = B$ versus $XA = B$.

## C. Irregular Matrix Multiply on Different Sizes (irrGEMM)

It is well-known that dense matrix decompositions can achieve relatively high performance if they could utilize dense matrix multiplication. The developed irrLU-GPU solution

```
/* input: ( m,  n) required panel size, passed by value */
/* input: (Ai, Aj) input offsets, passed by value */

/* read matrix-specific information based on id */
(local_m, local_n, ptrA, lda) =
    read_info( id, m_vec, n_vec, Aarray, lda_vec );

ptrA += Aj * lda + Ai // pointer arithmetic

local_m -= Ai; // maximum affordable m after offset
local_n -= Aj; // maximum affordable n after offset

// check for no workload (out of bound offsets)
if( local_m <= 0 || local_n <= 0 ) return;

// calculate actual workload
local_m = min(local_m, m);
local_n = min(local_n, n);

/* end of dcwi, proceed with computation */
```

Fig. 5. Pseudo code for DCWI

adopts a blocked decomposition for that purpose. We use the same codebase provided by the MAGMA library (version 2.6.1). While we do not claim any contributions in the design of this kernel, the code has been augmented with the interface in Section IV-A, and the DCWI layer in Section IV-B.

## D. Irregular Triangular Solves (irrTRSM)

Another key component in the LU algorithm is the triangular solve, which is required to update the upper factor. Similar to matrix multiplication, MAGMA-2.6.1 also provides such a functionality, but it suffers from a number of issues. First, the numerical behavior for solving $TX = B$ is different from a standard triangular solve. It computes the explicit inverse of the diagonal blocks in the triangular matrix $T$, so that the rest of the computation is done using matrix multiply. While this is a good design intuition, it generally produces larger backward error compared to a standard triangular solve. In addition, the solve is actually done out-of-place in a workspace, and then a memory copy is used to overwrite the right hand sides with the solution vectors. Recall that the standard LU decomposition requires an in-place solver during the update of the upper factor. It is clear that there is a noticeable overhead in performing a data copy for small sizes, plus the overhead of managing the extra workspace. For these reasons, we propose a more optimized and more accurate triangular solve routine (irrTRSM).

We use the same new interface, which enables us to develop a recursive triangular solve on matrices of different sizes. The recursive scheme is not new [31], but we claim that its adaptation to workloads of irregular sizes is novel. The host CPU coordinates the solve based on the largest dimensions, which are the order of the triangular linear system and the maximum number of right hand sides. The GPU kernels receive these information from the CPU, and the DCWI layer figures out the correct dimensions. Note that without the expanded interface, a recursive implementation is not possible without allocating workspaces for pointer/integer arithmetic at each level of the

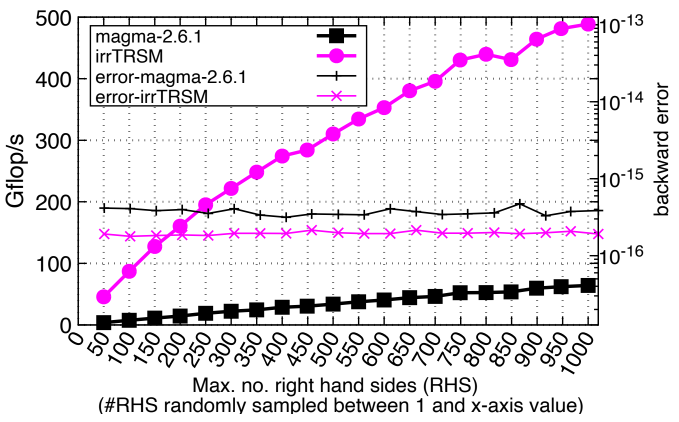


Fig. 6. Performance (left axis) and backward error (right axis) of irrTRSM (FP64) versus MAGMA-2.6.1 on the A100 GPU using CUDA-11.6. Each point represents 1000 triangular solves of different sizes. The orders of the triangular linear systems are randomly sampled between $1 \times 1$ and $32 \times 32$.

recursion. This would lead to considerable overhead, and also force a synchronous behavior for the triangular solve.

A comparison for the performance and backward error between irrTRSM and MAGMA is shown in Figure 6. The performance is shown in the FLOP rate, with the aggregate FLOP count estimated as $\sum_{i=1}^{batch\_size} n_i m_i^2$. The backward error is estimated as the maximum value of $\frac{||b-Tx||}{||T||.||x||}$ across all matrices. The comparison focuses on small triangular systems while varying the number of right hand sides, which is the typical use case in the LU decomposition. Through the output of the NVIDIA profiler, we observe that the MAGMA performance is severely impacted by the extra copy and workspace management. Figure 6 shows an asymptotic performance gain of 7.6×, while achieving a slightly better accuracy.

## *E. Block-column (Panel) Decomposition*

In order to utilize irrGEMM and irrTRSM, a decomposition of a block of columns must be performed efficiently on the GPU. This is often called the panel decomposition, which proceeds one column at a time. For each column, four steps are performed, (1) locating a *pivot* with the maximum magnitude (irrIAMAX), (2) if the pivot is an off-diagonal element, a row interchange is performed to bring the pivot on the diagonal (irrSWAP), (3) a vector scaling on the current column (irrSCAL), and (4) a rank-one update on the trailing subpanel (irrGER). Four GPU kernels are developed for these steps. However, it is sometimes possible to perform all these steps using a single GPU kernel (irrGETF2), which is the case when the largest panel can fit in the shared memory of the GPU. It is costly to calculate the exact size of the largest panel while the algorithm is progressing (for example, the tallest panel might not be the widest). We resort to a rough estimate, which is to assume that all the panels have the same width, which is a design parameter for irrLU-GPU (say typically $16-32$ columns per iteration). Then the largest panel size is $(16 \times (M_{max} - j))$, where $j$ is the global number of columns processed from the largest matrix. Depending on the shared memory capacity of the GPU, we either launch the irrGETF2 kernel, or the four separate kernels mentioned above. This is one example where an architectural feature of the GPU (the shared memory capacity) has a clear impact on the performance. For a GPU with a relatively small shared memory, the panel decomposition would switch from irrGETF2 to the slower column-wise approach earlier than on a GPU with a larger shared memory. The advantage of the irrGETF2 kernel is clear for relatively short panels, which is saving memory traffic. Figure 7 shows sample performance results for panels of different heights but of the same width. The performance is shown in Gflop/s, where the aggregate operation count is estimated as $\sum_{i=1}^{batch\_size} m_i n_i^2 - \frac{n_i^3}{3} - \frac{n_i^2}{2} + \frac{5n_i}{6}$.

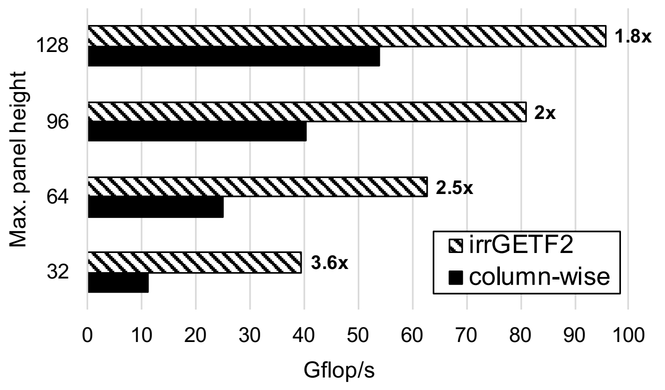


Fig. 7. Performance of irrGETF2 (FP64) versus column-wise block decomposition on the A100 GPU using CUDA-11.6. Each point represents 1000 panels whose widths are fixed at 32, while the heights are randomly sampled between 1 and the y-axis value.

## *F. Optimizing Row Interchanges*

One of the key challenges of the LU decomposition is the row interchanges on the entire width of the matrix following the panel decomposition step. Following our naming convention, this step is called (irrLASWP). Figure 8 shows that all rows adjacent to the upper square part of the panel must undergo a number of interchanges that reflect the pivoting steps in the panel decomposition. Note that for irrLASWP, the widths $w_l$ and $w_r$ are different for every matrix, and are calculated using the DCWI layer. A reference implementation for irrLASWP would be to call irrSWAP inside a loop, once for every row. However, in a column-major layout, a row access leads to poor utilization of the memory bandwidth. Since the pattern of row interchanges is data-dependent, non-contiguous data access is unavoidable, but there are means of mitigations that could be done.

We begin by "rehearsing" the row interchanges on auxiliary matrices containing exactly one column, and equal number of rows to the actual matrices. These matrices are initialized as $0, 1, 2, \cdots, M_{pi}$, where $M_{pi}$ is the height of each panel. We invoke the reference implementation kernel (looped irrSWAP) on these single-column matrices, which produces significantly less inefficient memory traffic. At the end of this step, the entries of each single-column matrix point to the final location of each row in the corresponding actual matrix. The data to be exchanged are split into small chunks as shown in Figure 8, so that the row interchanges can be done in shared memory, and then written back as one contiguous block.

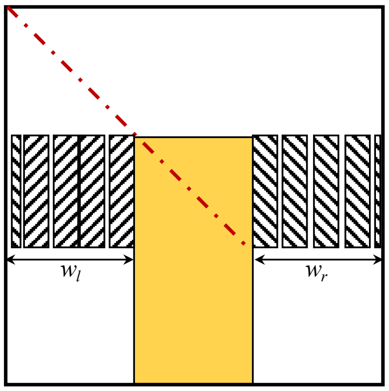


Fig. 8. Row interchanges in irrLU-GPU

While this optimization utilizes the bandwidth much more efficiently than a looped irrSWAP, it has some minor shortcomings. First, it needs a memory workspace for the auxiliary matrices, which breaks the asynchronicity of irrLU-GPU if done on the fly. This can be avoided if the interface of irrLU-GPU is modified to accept a workspace parameter so that the user controls where the allocation takes place. Second, corner cases where the pivots are always (or mostly) found on the diagonal might lead to worse performance than a looped irrSWAP. A looped irrSWAP will entirely skip a row interchange if the pivot is already on the diagonal. With the new technique, it is relatively difficult to isolate the rows that stayed in place from those that need to change location. In other words, the performance is irrelevant to the pivoting pattern. Since most realistic test cases require pivoting, the average performance is higher than a looped irrSWAP. Figure 9 (top) shows the impact of the two pivoting strategies on the performance of irrLU-GPU. Figure 9 (bottom) shows the corresponding time spent in both approaches for the performance test of irrLU-GPU. Note that the performance gain grows with the average width of the matrices, since a looped irrSWAP becomes more impacted by the non-contiguous memory accesses.

## V. Experimental Results

### A. Performance of irrLU-GPU

This section presents the final performance results of irrLU-GPU in FP64 arithmetic. Each figure shows two GPU solutions and one reference CPU solution. The CPU performance results are obtained by running the `getrf_batch` routine in MKL 2022.0.0 on a 36-core dual socket Intel CPU (Intel Xeon Gold 6140 CPU running @ 2.30GHz). For NVIDIA GPUs, irrLU-GPU is benchmarked using a Tesla A100-SXM4 GPU, clocked at 1.41 GHz, using CUDA-11.6. For AMD GPUs, benchmarking is performed using an AMD Instinct MI100

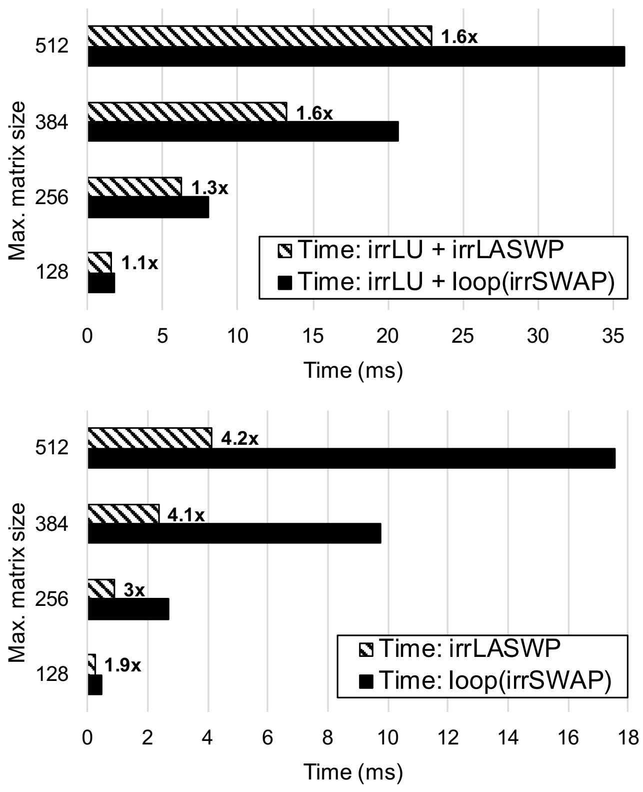


Fig. 9. Top: Timing of irrLU-GPU (FP64) using irrLASWP versus using looped irrSWAP on the A100 GPU using CUDA-11.6. Bottom: the corresponding elapsed time of both irrLASWP and looped irrSWAP. Each point represents 1000 square matrices of different sizes that are randomly sampled between 1 and the y-axis value.

GPU, clocked at 1.5 GHz, using ROCM-5.0. We also show reference GPU implementations using cuSOLVER/rocSOLVER called within 16 concurrent GPU streams. Each testing point represents one thousand square matrices, whose sizes are randomly sampled between 1 and the value shown on the x-axis. The FLOP rate is obtained by dividing the aggregate operation count by the execution time. The aggregate operation count is $\sum_{i=1}^{batch\_size} \frac{2n_i^3}{3} - \frac{n_i^2}{2} + \frac{5n_i}{6}$. The performance at each point is the arithmetic mean of the FLOP rate across 10 runs.

Figure 10 shows the performance for FP64 arithmetic. It is obvious that parallel calls to cuSOLVER/rocSOLVER do not yield any good performance. This is mainly due to the challenging distribution of sizes inside the batch, for which the overhead of the call becomes significant. Note that the performance numbers for cuSOLVER/rocSOLVER do not include creating/destroying streams or allocating/freeing workspaces. The performance of the CPU is quite competitive, especially against the MI100 GPU. We generally observe a performance gap between the A100 and the MI100 GPUs. Our speculation is that the HIP kernel language is not yet mature compared to CUDA. Another note is that the available shared memory on the MI100 (64 KB) is much smaller than the A100 GPU (192 KB). This generally limits the occupancy for kernels relying

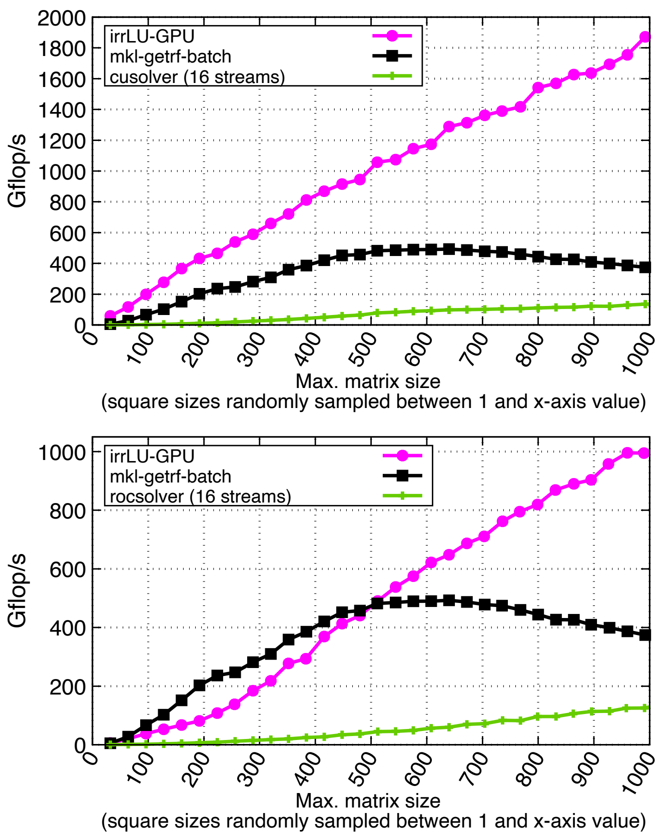


Fig. 10. Performance of irrLU-GPU (FP64) on the A100 GPU (top) and the MI100 GPU (bottom). Results are obtained using CUDA-11.6 and ROCM-5.0. Each point represents 1000 square matrices of different sizes that are randomly sampled between 1 and the x-axis value.

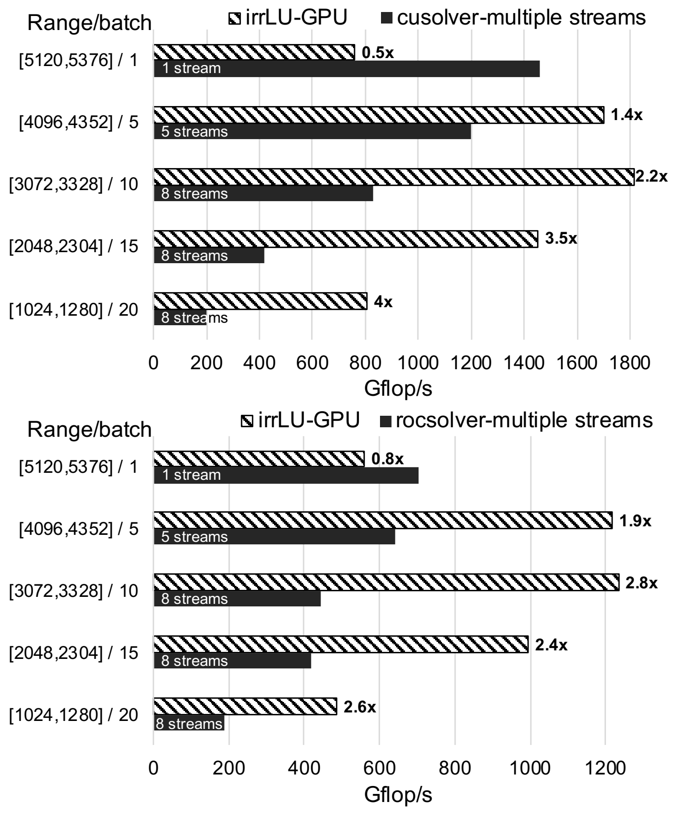


Fig. 11. Performance of irrLU-GPU (FP64) on the A100 GPU (top) and the MI100 GPU (bottom) against cuSOLVER/rocSOLVER with multiple streams. Results are shown for few, relatively large matrices. The labels of the y-axis can be read as [$N_{min}$:$N_{max}$] / batch_size.

on shared memory workspaces (especially in our case, the irrGETF2 kernel). IrrLU-GPU has a clear advantage on the A100 GPU, with an asymptotic performance gain of $4.5\times$ against the CPU. For the MI100 GPU, irrLU-GPU outperforms the CPU performance only for relatively larger workloads, scoring up to $2.7\times$ better performance. There is a potential room for improvement for the A100 GPU if the Tensor Cores are used for the irrGEMM kernel. A similar approach can be taken for AMD GPUs with Matrix Engine units for FP64 arithmetic (e.g. the MI250x GPU).

Figure 11 shows another performance comparison for a small number of matrices that are relatively large in size. This is a typical case in the sparse solver near the root of the assembly tree. We empirically tuned the number of streams for cuSOLVER/rocSOLVER at each test point. We observe a much smaller gap between irrLU-GPU and cuSOLVER/rocSOLVER, which even turns into the favor of the latter for matrices beyond $5k\times 5k$. This is an expected behavior, as there will be a crossover point where a design dedicated to batches of relatively small matrices is outperformed by parallel calls to a design that targets relatively large sizes.

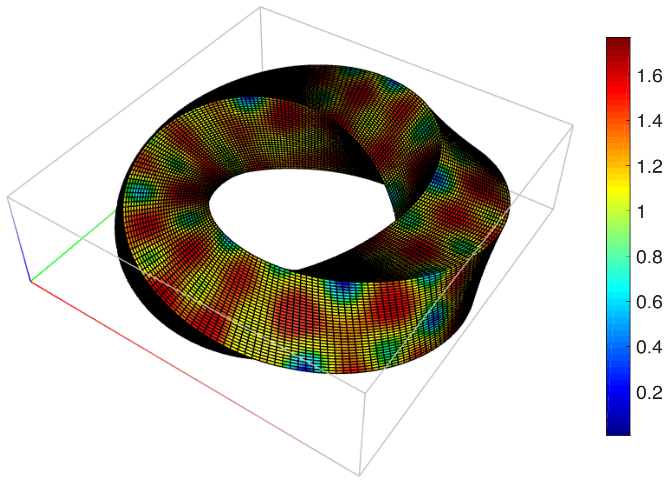


Fig. 12. Example solution of the second order Maxwell equation on a toroidal domain discretized using Nédélec finite elements. The mesh has 614,592 finite element unknowns with approximately 24 points per wavelength and the corresponding sparse matrix has 19,636,416 nonzeros.

### *B. Performance of the Sparse Direct Solver for Indefinite Maxwell Simulations*

We solve the electromagnetics problem corresponding to the second order Maxwell equation, $\nabla\times\nabla\times\mathbf{E}-\Omega^2\mathbf{E}=\mathbf{f}$, which is given in the weak formulation as $(\nabla \times \mathbf{E}, \nabla \times \mathbf{E}') - (\Omega^2 \mathbf{E}, \mathbf{E}') = (\mathbf{f}, \mathbf{E}')$ with a testing function $\mathbf{E}'$. A given tangential field $\mathbf{f}(\mathbf{x}) = (\kappa^2 - \Omega^2)(\sin(\kappa x_2), \sin(\kappa x_3), \sin(\kappa x_1))$ is used as the boundary condition for $\mathbf{E}$. For large wavenumber $\Omega$, the problem is highly indefinite and hard to precondition, so typically a direct solver is used. The weak form is discretized with first order Nédélec elements using the modular finite element library MFEM [32]. The results for $\Omega = 16$, $\kappa = \Omega/1.05$ and a toroidal geometry with hexahedral finite elements are shown in Figure 12. Results in this section all use FP64. Note that for these tests, the MC64 static pivoting is not required, and all sparse solves reach a backward error close to machine precision after a single step of iterative refinement. Computing the fill-reducing ordering with METIS takes about $10s$ and the symbolic analyis about $2s$. Note that the costs for both ordering and symbolic phase can be amortized when solving multiple consecutive linear systems with the same sparsity pattern. Figure 13 illustrates the distribution of the

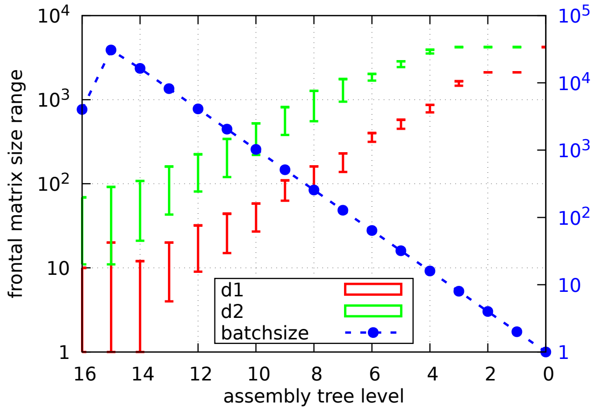


Fig. 13. Distribution of the matrix sizes within each batch (level of the assembly tree). $d1$ are the dimensions of the upper-left block of each front, i.e., the sizes for the irrLU operation. $d2$ are the dimensions for the lower-right part of each front. The irrTRSM operation is applied with $m = d1$ and $n = d2$, and the irrGEMM with $m = n = d2$ and $k = d1$. Level 15 has batchsize 30,727.

matrix sizes, as well as the batchsize, for each batch. As the assembly tree is traversed from the leaves to the root (level 0), the average matrix size increases, while the batchsize decreases.

*1) GPU Performance Breakdown:* Figure 14 shows the runtime, on the A100 GPU, for the different operations performed during the numerical factorization of the sparse matrix corresponding to the discretization of the problem illustrated in Figure 12. The batch operations are compared with a trivial implementation calling cuBLAS or cuSOLVER in a loop. cuBLAS outperforms irrGEMM for large matrix sizes and small batchcounts, hence we combine irrGEMM for matrix sizes $<= 256$ with cuBLAS GEMM in a loop for matrix sizes $> 256$. One of the reasons behind this behavior is that the current irrGEMM implementation does not take advantage of the Tensor Core units on the A100. Note that irrLU and irrTRSM outperform the corresponding routines GETRF and GETRS (2xTRSM + LASWP) for almost all matrix sizes. We observe a similar performance breakdown on the MI100 GPU, and so its respective results are not shown.

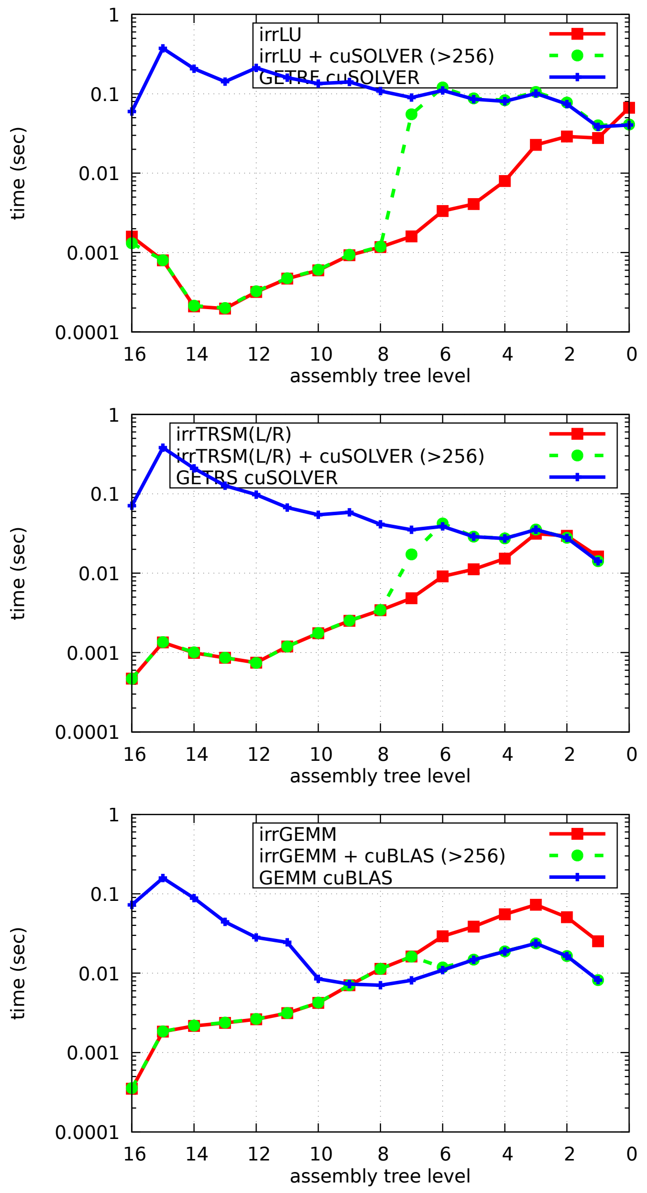


Fig. 14. Runtime on A100 for the different batches from the problem illustrated in Fig. 12. Top: LU decomposition time with irrLU vs. calling cuSOLVER's GETRF in a loop vs. irrLU for matrix sizes $<= 256$ combined with a cuSOLVER loop for matrices $> 256$. Middle: cuSOLVER GETRS vs. irrTRSM (lower and upper) and irrLASWP. Bottom: irrGEMM vs. cuBLAS GEMM in a loop.

*2) Solver Comparison:* Table I compares the total runtime for the numerical factorization on the A100 and MI100 with our optimized GPU implementation (relying on irrLU, irrTRSM and irrGEMM), with the naive cuBLAS/cuSOLVER loop implementation, with STRUMPACK [13] v6.3.1, and with SuperLU_Dist [8] v7.2.0. From SuperLU_Dist, the 3D factorization code pdgssvx3d was used, which offloads more operations to the GPU, and performs better, compared to

|  |  | A100 |  | MI100 |  |
|---|---|---|---|---|---|
|  |  | AMD 7763 (16c) |  | AMD 7662 (16c) |  |
|  | solver | time (s) | Gflop/s | time (s) | Gflop/s |
| GPU | using irrLU | 1.77 | 1394.7 | 1.97 | 1256.1 |
| GPU | cuBLAS/SOLVER | 5.44 | 453.5 | 19.37 | 127.4 |
| GPU | STRUMPACK | 1.93 | 1275.9 | 13.56 | 182.0 |
| GPU | SuperLU | 12.58 | 190.3 | - | - |
| CPU | STRUMPACK | 8.62 | 286.6 | 10.16 | 243.2 |
| CPU | SuperLU | 17.89 | 132.8 | 31.94 | 74.7 |

TABLE I
NUMERICAL FACTORIZATION PERFORMANCE ON BOTH NVIDIA A100 AND AMD MI100, AND COMPARISON WITH STRUMPACK [13] AND SUPERLU_DIST [8].

the earlier implementation in pdgssvx (the 2D algorithm). We used the default parameters for the STRUMPACK and SuperLU_Dist solvers. The CPU runs use 16 OpenMP threads and Cray LibSci for BLAS and LAPACK. On the A100 GPU, using the NVIDIA Nsight profiler reveals that STRUMPACK spends 9.1% (.91s) in cudaStreamSynchronize and 6.5% (.65s) in cudaLaunchKernel. In the optimized batched implementation both bottlenecks are reduced: .33s for cudaStreamSynchronize and .16s for cudaLaunchKernel.

There are few takeaways that we can extract from Table I. First, the proposed solution utilizing irrLU-GPU outperforms all the other solutions mentioned in Table I. Second, the closest competitor to the proposed solution is STRUMPACK on the A100 GPU, where a 9% advantage is observed for irrLU-GPU. Profiling STRUMPACK on the A100 GPU shows that special kernels are used for small size LU, TRSM and GEMM operations (see [13]). All blocks larger than $32{\times}32$ are handled by cuBLAS/cuSOLVER in STRUMPACK. The 9% improvement in favor of irrLU comes from supporting all sizes. On the MI100 GPU, the kernel launch overheads are more significant, which leads to bigger performance gains (13.56s for STRUMPACK vs 1.97s for irrLU). Third, despite the clear advantage of irrLU-GPU on the A100 GPU versus the MI100 GPU in Figure 10, the final solution times in Table I are close (1.77s for the A100 GPU versus 1.97s for the MI100 GPU). We observe that reducing the kernel launch overhead on the MI100 GPU has a bigger impact on performance compared to the A100 GPU. By removing the launch overheads for the large batches on the lower levels of the assembly tree, the total runtime of the solver is dominated by nodes at the higher levels of the assembly tree. More specifically, the runtime is most impacted by the GEMM performance on the largest nodes. Since we don't use the Tensor Cores on the A100 GPU, the theoretical peak performances for FP64 GEMMs are close (9.7 Tflop/s on the A100 GPU, versus 11.5 Tflop/s on the MI100 GPU).

### C. Software Availability

The irrLU, irrGEMM, and irrTRSM routines are available in the MAGMA library[1]. The performance of the sparse direct solver is based on an integration between the STRUMPACK library[2] and MAGMA, and is also planned for the next release of STRUMPACK.

## VI. CONCLUSION AND FUTURE WORK

This paper presents a systematic way of addressing matrix computations on GPUs for batches containing matrices of different sizes. While the proposed approach is generic, the paper present a case study for the LU decomposition on irregular set of dense matrices. The developed solution is called irrLU-GPU, and addresses a performance-critical component in a multifrontal sparse LU solver, that is used to simulate a high frequency indefinite Maxwell problem using finite element discretization on unstructured meshes. The experimental results show a significant impact on the performance of the sparse direct solver. The design concepts of irrLU-GPU, mainly the interface and the dynamic workload discovery layer, are applicable to other numerical algorithms operating on irregular sets of relatively small dense matrices.

There are several ways to extend this work going forward. We plan to seek functional and performance portability to Intel GPUs. The current implementation of irrGEMM does not take advantage of the hardware acceleration units for matrix multiply on NVIDIA or AMD GPUs, which would improve the performance of both irrGEMM and irrTRSM. This would require a redesign of the irrGEMM kernel, though, in order to take advantage of specific architectural features on different GPUs. There is also a chance of concurrent kernel execution which can be exploited in the case of performing the right and left swaps simultaneously. As mentioned in the paper, the proposed interface and the DCWI layer would work seamlessly for other decompositions, such as the QR factorization, which can be used in Sparse QR algorithms. Finally, it is challenging to find a robust way of auto-tuning the developed kernels. Most of the tuning techniques that we are aware of take the problem size as an input (along with other information). In the case of irrLU-GPU, irrGEMM, and irrTRSM, we have a mix of sizes that are known only at run time. It is certainly a research direction to find robust auto-tuning techniques based on the distributions of sizes in a single batch.

## ACKNOWLEDGMENT

This research is supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

## REFERENCES

[1] I. S. Duff and J. K. Reid, "The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations," *ACM Trans. Math. Softw.*, vol. 9, no. 3, pp. 302–325, September 1983. [Online]. Available: https://doi.org/10.1145/356044.356047

[2] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, "A survey of direct methods for sparse linear systems," *Acta Numer.*, vol. 25, pp. 383–âĂŞ566, 2016.

[3] "Intel Math Kernel Library," available at http://software.intel.com/intel-mkl/.

[4] "NVIDIA CUBLAS," available at https://developer.nvidia.com/cublas.

[5] "rocBLAS, Next Generation BLAS Implementation for ROCm Platform," available at https://github.com/ROCmSoftwarePlatform/rocBLAS.

[1] https://icl.utk.edu/magma/

[2] https://portal.nersc.gov/project/sparse/strumpack/

[6] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí, "Variable-size batched Gauss–Jordan elimination for block-Jacobi preconditioning on graphics processors," *Parallel Comput.*, vol. 81, pp. 131–146, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819117302107

[7] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *J. Phys.: Conf. Ser.*, vol. 180, p. 012037, July 2009. [Online]. Available: https://doi.org/10.1088/1742-6596/180/1/012037

[8] X. S. Li and J. W. Demmel, "SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems," *ACM Trans. Math. Softw.*, vol. 29, no. 2, pp. 110–âĂŞ140, June 2003. [Online]. Available: https://doi.org/10.1145/779359.779361

[9] T. A. Davis, "Algorithm 832: UMFPACK V4.3—an Unsymmetric-Pattern Multifrontal Method," *ACM Trans. Math. Softw.*, vol. 30, no. 2, pp. 196–âĂŞ199, June 2004. [Online]. Available: https://doi.org/10.1145/992200.992206

[10] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate," *ACM Trans. Math. Softw.*, vol. 35, no. 3, October 2008. [Online]. Available: https://doi.org/10.1145/1391989.1391995

[11] P. Hénon, P. Ramet, and J. Roman, "PaStiX: a high-performance parallel direct solver for sparse symmetric positive definite systems," *Parallel Comput.*, vol. 28, no. 2, pp. 301–321, 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819101001417

[12] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov, "An Efficient Multicore Implementation of a Novel HSS-Structured Multifrontal Solver Using Randomized Sampling," *SIAM J. Sci. Comput.*, vol. 38, no. 5, pp. S358–S384, 2016. [Online]. Available: https://doi.org/10.1137/15M1010117

[13] P. Ghysels and R. Synk, "High performance sparse multifrontal solvers on modern gpus," *Parallel Comput.*, vol. 110, p. 102897, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819122000059

[14] P. R. Amestoy and L. S. Duff, "Vectorization of a Multiprocessor Multifrontal Code," *Int. J. High Perform. Comput. Appl.*, vol. 3, no. 3, pp. 41–59, September 1989. [Online]. Available: https://doi.org/10.1177/109434208900300303

[15] A. Gupta, "WSMP: Watson Sparse Matrix Package Part II âĂŞ direct solution of general systems," IBM T. J. Watson Research Center, Tech. Rep., 2000, https://s3.us.cloud-object-storage.appdomain.cloud/res-files/1331-wsmp2.pdf.

[16] A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Automatic code generation for many-body electronic structure methods: the tensor contraction engine," *Mol. Phys.*, vol. 104, no. 2, pp. 211–228, 2006. [Online]. Available: https://doi.org/10.1080/00268970500275780

[17] O. E. B. Messer, J. A. Harris, S. Parete-Koon, and M. A. Chertkow, "Multicore and Accelerator Development for a Leadership-Class Stellar Astrophysics Code," in *Applied Parallel and Scientific Computing*, P. Manninen and P. Öster, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 92–106.

[18] A. Abdelfattah, T. Costa, J. Dongarra, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Kurzak, P. Luszczek, S. Tomov, and M. Zounon, "A Set of Batched Basic Linear Algebra Subprograms and LAPACK Routines," *ACM Trans. Math. Softw.*, vol. 47, no. 3, June 2021. [Online]. Available: https://doi.org/10.1145/3431921

[19] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam, "Designing Vector-friendly Compact BLAS and LAPACK Kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 55:1–55:12. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126941

[20] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, Design, and Autotuning of Batched GEMM for GPUs," in *High Performance Computing*, ser. ISC High Performance, J. M. Kunkel, P. Balaji, and J. Dongarra, Eds. Cham: Springer International Publishing, 2016, pp. 21–38. [Online]. Available: https://doi.org/10.1007/978-3-319-41321-1_2

[21] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998. [Online]. Available: https://doi.org/10.1137/S1064827595287997

[22] I. S. Duff and J. Koster, "The design and use of algorithms for permuting large entries to the diagonal of sparse matrices," *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 4, pp. 889–901, 1999. [Online]. Available: https://doi.org/10.1137/S0895479897317661

[23] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356223

[24] A. Abdelfattah, M. Baboulin, V. Dobrev, J. J. Dongarra, C. W. Earl, J. Falcou, A. Haidar, I. Karlin, T. V. Kolev, I. Masliah, and S. Tomov, "High-Performance Tensor Contractions for GPUs," in *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, 2016, pp. 108–118. [Online]. Available: https://doi.org/10.1016/j.procs.2016.05.302

[25] S. N. Yeralan, T. A. Davis, W. M. Sid-Lakhdar, and S. Ranka, "Algorithm 980: Sparse QR Factorization on the GPU," *ACM Trans. Math. Softw.*, vol. 44, no. 2, August 2017. [Online]. Available: https://doi.org/10.1145/3065870

[26] M. J. Anderson, D. Sheffield, and K. Keutzer, "A Predictive Model for Solving Small Linear Algebra Problems in GPU Registers," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 2–13. [Online]. Available: https://doi.org/10.1109/IPDPS.2012.11

[27] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra, "A Step towards Energy Efficient Computing: Redesigning a Hydrodynamic Application on CPU-GPU," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 972–981. [Online]. Available: https://doi.org/10.1109/IPDPS.2014.103

[28] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. J. Dongarra, "Batched matrix computations on hardware accelerators based on GPUs," *Int. J. High Perform. Comput. Appl.*, vol. 29, no. 2, pp. 193–208, 2015. [Online]. Available: https://doi.org/10.1177/1094342014567546

[29] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Batched one-sided factorizations of tiny matrices using GPUs: Challenges and countermeasures," *J. Comput. Sci.*, vol. 26, pp. 226–236, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750317311456

[30] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí, "Batched Gauss-Jordan Elimination for Block-Jacobi Preconditioner Generation on GPUs," in *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM'17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1âĂŞ–10. [Online]. Available: https://doi.org/10.1145/3026937.3026940

[31] A. Charara, H. Ltaief, and D. Keyes, "Redesigning Triangular Dense Matrix Computations on GPUs," in *Proceedings of the 22nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 477–489. [Online]. Available: https://doi.org/10.1007/978-3-319-43659-3_35

[32] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini, "MFEM: A modular finite element methods library," *Comput. Math. with Appl.*, vol. 81, pp. 42–74, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0898122120302583

# Appendix: Artifact Description/Artifact Evaluation

## SUMMARY OF THE EXPERIMENTS REPORTED

The paper provides two sets of experiments for the final results:

**FIRST: Experiments for testing the irrLU routine (Figures 10 and 11):**

The proposed solutions were developed inside a private fork of the MAGMA library (http://icl.utk.edu/magma/). The fork is available in a docker image that we provide below. The systems used for these results are:

**System 1**

(1) One DGX-A100 system equipped with eight A100-SXM4-80GB GPUs @ 1.41 GHz. Only one GPU is used.
(2) The CPU is AMD EPYC 7742 64-Core Processor @ 2.25 GHz, but it is not used in the experimental results.
(3) The MAGMA fork is built with CUDA-11.6 and uses MKL 2022.0.0 as a backend. MAGMA host code is compiled using GCC 7.3.0.

**System 2**

(1) The same MAGMA setup above is used on another system with a 36-core dual socket Intel CPU (Intel Xeon Gold 6140 CPU running @ 2.30GHz). This system is used only to test the MKL getrf_batch routine on the CPU.

**System 3**

(1) One node of the OLCF Spock system. The experiments use one AMD Instinct MI100 GPU, clocked at 1.5 GHz, using ROCM-5.0.
(2) OpenBLAS 0.3.17 is used to provide the CPU backend for MAGMA.

**System 4 (docker image)**

(1) The docker image provides the sources and the binaries of the MAGMA fork. It is compiled using GCC 4.8.5, CUDA-11.0.221, and uses OpenBLAS 0.3.17 as the CPU backend.

**SECOND: Experiments for testing the sparse direct solver (Figure 14 and Table 1)**

The sparse solver is based on a modified version of the STRUMPACK library that uses the MAGMA fork as a backend. The modified sparse solver is also available in a container image below. The driver application is a modified MFEM example, which is available in the docker image: /home/mfem/examples/ex3p_indef.cpp. The systems used for these experiments are:

**System 1**

(1) One node of NERSC's Perlmutter system. The experiments are conducted on one Ampere A100 GPU that is hosted by an AMD EPYC 7763 64-Core Processor. The following packages are used to run the application in Section V.B:
(2) GCC 11.2.0
(3) MFEM 4.4
(4) STRUMPACK batch_LU branch
(5) The MAGMA fork
(6) CUDA Toolkit 11.5
(7) BLAS/LAPACK/ScaLAPACK: cray-libsci/21.08.1.2
(8) SuperLU_Dist 7.2.0
(9) METIS 5.1.0
(10) Only needed as compilation dependencies: hypre 2.24.0, parmetis 4.0.3, cray-mpich/8.1.13

**System 2**

(1) One node of the OLCF Spock system. The packages used are the same as those on Perlmutter, except for ROCm 5.0.2

**System 3**

(1) Docker image to reproduce results from the application in Section V.B. The packages used in the docker image are the same as those on Perlmutter, except for:

(2) GCC 8.5.0
(3) CUDA Toolkit 11.4.0
(4) BLAS/LAPACK: OpenBLAS 0.3.17
(5) ScaLAPACK 2.2.0
(6) mpich 4.0.2

## AUTHOR-CREATED OR MODIFIED ARTIFACTS:

**Artifact 1**
Persistent ID: `https://bitbucket.org/abdelfattah83/magma-vlu`
Artifact name: Modified fork of MAGMA

**Artifact 2**
Persistent ID: `https://github.com/pghysels/STRUMPACK/tree/batch_LU`
Artifact name: Modified STRUMPACK to use the MAGMA fork

**Artifact 3**
Persistent ID: `https://hub.docker.com/r/abdelfattah83/irrlu_cuda`
Artifact name: Docker image of MAGMA fork

**Artifact 4**
Persistent ID: `https://hub.docker.com/r/pghysels/irrlu_cuda_11.4.0`
Artifact name: Docker image of MFEM example using STRUMPACK and MAGMA

*Reproduction of the artifact with container:* ==> FIRST: docker image for MAGMA
————————————————

This container image provides the sources and the binaries of the irrLU-GPU solution. The solution is developed inside a private fork of the MAGMA library. The image is built for Volta and Ampere architectures using cuda-11.0. However, reproducing the results in the paper (Figures 10 and 11) requires access to an A100-SXM4 GPU.

docker pull abdelfattah83/irrlu_cuda:latest

To run the image:

docker run -it –entrypoint=bash –privileged –userns=host abdelfattah83/irrlu_cuda

Then run the test script:
sh test_irrlu.sh

The script runs a test for a batch of 1000 square matrices, whose sizes are randomly sampled between 1 and 1024. The script can be edited inside the image using vi. You can change the testing parameters such as the range of sizes, the batch size, and the number of parallel streams for testing cusolver.

The top level routine for irrlu is: /home/irrlu/src/dgetrf_vbatched.cpp
The tester is: /home/irrlu/testing/testing_dgetrf_vbatched.cpp

==> SECOND: docker image for MFEM example using STRUMPACK and MAGMA
——————————————————————————————————
This container image provides the sources and the binaries for a sparse direct solver utilizing the solutions proposed in the paper (irrLU, irrGEMM, and irrTRSM). The sparse solver is based on the STRUMPACK library. The irrLU, irrGEMM, and irrTRSM routines are developed inside a private fork of the MAGMA library.

The image requires an Ampere A100 GPU, and is built using CUDA-11.4.0

The image must be run using nvidia-docker2 (https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html#docker)
sudo docker pull pghysels/irrlu_cuda_11.4.0:latest
sudo docker run -it –gpu all –rm pghysels/irrlu_cuda_11.4.0:latest

The image automatically runs the script that generates the results in Figure 14 and Table 1 (for the A100 GPU).
The test script is located at: /home/test_mfem.sh

All the source codes are available under the /home/ directory. The interfacing with MAGMA is located at:
/home/strumpack/src/sparse/fronts/FrontalMatrixGPU.cpp and
/home/strumpack/src/dense/MAGMAWrapper.hpp.