Stretching Jacobi: A Two-Stage Pivoting Approach for Block-Based Factorization

Daniel Thuerck
AIML Lab and Graduate School CE
TU Darmstadt, Darmstadt, Germany

***Abstract*—Given enough parallel compute units, the processing time for sparse (in-)complete factorization is determined by the number of level sets found in the pattern of the matrix. For positive definite or diagonal dominant matrices, the iterative Jacobi-method presents itself as a scalable alternative on massively-parallel systems. In this paper, we present modifications to the Jacobi algorithm, extending its applicability to a wider class of matrices. Our approach cuts the matrix into blocks to allow register-based pivoting in batched CUDA factorization kernels and, for the first time on GPUs, also flexible permutations on the block level in an *a posteriori* threshold pivoting scheme. Experiments show our batched kernels to be on par with partially pivoted approaches and that the resulting sparse factorization compares favorably in quality and speed with their traditionally scheduled counterparts.**

I. Introduction

We consider the problem of solving real linear systems of equations of the form $Ax = b, A \in \mathbb{R}^{n \times n}$, where A is sparse. We consider heterogeneous computer systems which we define as compute systems with a CPU (denoted as *host*) and at least one massively-parallel accelerator card (*device*). While our approach is of general nature, we only consider NVIDIA's CUDA-capable GPUs here.

Especially in parallel computing, iterative (Krylov) solvers such as GMRES [1] are considered a viable alternative to direct methods. Their main computational primitive is the sparse matrix-dense vector multiplication (SpMV), often responsible for over 99 % of the execution time. Practical experience shows that the distribution of eigenvalues often governs the convergence; ideally, the eigenvalues are clustered in a small interval; otherwise, these matrices suffer from stagnation of the residual after a few iterations. In order to solve these systems, we employ a *preconditioner* M such that AM^{-1} resp. $M^{-1}A$ has a more amenable eigenvalue distribution and linear systems with M are computationally efficient to solve.

While there are many types of preconditioners available, the lack of problem-specific preconditioners naturally leads to 'black box'-types of preconditioners such as *incomplete factorizations*. As their name suggests, *incomplete* factorizations of sparse matrices are factorizations (e.g. LU - type) where some numerical nonzero entries have been removed or *dropped*. These can be generated by either traditional factorization packages and applying filters to the generated entries; alternatively, Chow and Patel [2] have presented a massively-parallel scheme that offers tremendous speedups by solving a fixed-point equation by an iterative Jacobi method; however it is only applicable to matrices that are close to diagonal dominant. For tough matrices, they usually fail.

When dealing with such matrices, even direct methods require *pivoting*, i.e. changing the matrix' structure to minimize numerical errors. As pivoting changes the order of computations on-the-fly, this especially hurts parallel approaches in two ways: (a) after a pivoting operation, the symbolic analysis needs to be re-evaluated, leading to some sequential, symbolic operations and a stall in numerical computation; (b) even inside dense blocks, this leads to irregular memory accesses – in the sense that these accesses cannot be planned ahead, preventing the use of highly-tuned, static kernels. For these reasons, pivoting has rarely been considered for preconditioners on parallel systems. Yet, pivoting can be crucial for the success of a factorization.

In this paper, we propose to use a-priori blocking and present a data structure and algorithms to enable global pivoting on matrices in parallel, enabling the use of such Jacobi methods even on tough matrices. Our contributions are as follows:

1) We present a blocking-based data structure that allows parallel matrix permutations on the block level, enabling pivoting over the whole matrix.
2) To maintain accuracy inside these blocks, we present CUDA-kernels that, for the first time, allow irregular *full* and *symmetric* pivoting with regular, static memory accesses.
3) We present a modification for the Jacobi factorization scheme that enables convergence even for indefinite, non-diagonal dominant matrices.

II. Background

This section gives a short introduction into the computation of numerical factorizations and lists notable related works.

A. Parallel (In)complete factorizations

Numerical factorizations decompose a sparse, real input matrix A into $A = LDU$ with diagonal D in the unsymmetric case; $A = LL^\top$ in the symmetric positiove definite case; or $A = LDL^\top$ with a 1×1 and 2×2-block diagonal D in the symmetric indefinite case (this work only considers the latter). In order to improve the condition number of the input matrix, minimize the fill-in produced during the factorization and select 2×2-pivots ahead of time, a preprocessing pipeline to scale, permute and match [3] A is usually employed; the

factorization then works on the matrix $P_R S_R A S_C P_C$ with permutation matrices P and diagonal scaling matrices S. In the case of an incomplete factorization, dropping entries by their location or magnitude [4], [5] introduces an error matrix $E \neq 0$ such that $A = LDL^\top + E$. Ideally, the norm of E should be kept small. More details on the effects of dropping are discussed in Saad [6].

Departing from 'classic' methods that process the matrix along its nonzero structure, Chow and Patel [2] proposed to use a fixed-point iteration scheme to approximate an incomplete LU-factorization. Their main insight is that for an incomplete preconditioner $A \approx LU$ and a predefined sparsity pattern S, it holds that

$$A_{ij} = (LU)_{ij}, (i,j) \in S \tag{1}$$

which leads to the following equations:

$$L_{ij} = U_{jj}^{-1}(A_{ij} - \sum_{k=1}^{j-1} L_{ik}U_{kj}), U_{ij} = (A_{ij} - \sum_{k=1}^{i-1} L_{ik}U_{kj}). \tag{2}$$

These Equations can then be solved in a fixed-point manner, always taking the initial matrix A on the right hand side as input and consecutively updating L and U - each iteration is called a *sweep*.

B. Related work

As discussed, allowing pivoting causes sequential symbolic operations and irregular memory accesses. Reconciliating pivoting and parallel computations has often been discussed in the literature concerning direct methods (i.e. *full* factorizations) , but only rarely in conjunction with incomplete factorizations.

Pivoting: We find that the scope and amount of pivoting allowed in a package can be represented on a quasi-continuous scale: The simplest approaches processes the factorization in the order as supplied by the input matrix [7]; whenever a small pivot is encountered, they usually add a small numerical shift to it and rely on iterative refinement in the Krylov solver later on. For these cases, the mentioned preprocessing and scaling of the input matrix is crucial. A step more towards global pivoting is given in out previous work [8]: When the matrix is cut into dense blocks a priori, then pivoting operations inside these blocks come without the burden of requiring a data structure modification. Increasing the size of these blocks more and more approximates full pivoting; frontal matrices as in PARDISO [9], [10] can reach sizes up to 1024×1024. Lastly, the extreme case allows unlimited, arbitrary pivoting as in Greif et al. [11]. With this work, we add another option that is located somewhere around the center of this axis: pivoting inside modestly-sized blocks, but we also permit the permutation of the blocks themselves.

Blocking: Any successful operation on such blocks requires heuristics to initially find dense blocks in the sparse matrix without wasting too much storage saving numerical zeros. Mostly, we distinguish between two methods: a priori [8], [12] and dictated by amalgamating nodes of the elimination tree [13]. While left-looking blocked approaches permit to initially set up these block structures, multifrontal packages with a right-looking computation order often pack/unpack scalar rows and columns of the matrix into so-called *frontal* matrices, which are, by design, almost fully dense. Apart from minimizing the symbolic computations, using blocks also permits us to exploit dense BLAS and LAPACK kernels on the blocks. On the CPU, such cache-friendly kernels even offset the additional work of increasing the size of blocks progressively during the computation as done in Bollhoefer [7]. Other than these traditional methods, Götz and Anzt [14] have presented the first deep learning based approach to detect dense blocks near a matrix' diagonal.

Parallelization: Sparse linear algebra, due to its content of symbolic operations and irregular memory accesses, presents a challenge for the effective use of SIMD-based, throughput-oriented accelerator cards. In the case of factorizations, parallel performance rests on two pillars: (1) using dense, parallel kernels on blocks and (2) detecting independent operations due to the dill structure of the matrix. The latter leads to multilevel-types of factorizations [15] that process rows resp. columns of matrix that are independent, at the same time. Those sets are discovered by a breadth-first search on the matrix' adjacency graph. Since the latter changes when permuting the matrix during preprocessing, Naumov investigated [16] graph coloring as a means to expose more parallelism; we [8] studied the effects of such reordering. In case of direct factorizations or significant expected fill-in, the computation is guided by the matrix' elimination tree [17], [18]. Recently, Hogg has also explored the use of DAGs [19] for scheduling. Lastly, we note the efforts made to include more and more special abilities and hardware of GPUs, such as NVIDIA's Tensor Cores for fast processing of half precision; mixed-precision approaches [20], [21] could help to save computation where lower precision is sufficient.

Implementing the fixed point equations 2 from above leads to an embarassingly parallel method that is ideally suited for the GPU. This method, often referred as 'Jacobi style' (not to be confused with diagonal preconditioning!) is used in subsequent works as a building block for e.g. triangular solves. ParILUT, an iterative threshold-dropping based Jacobi method [22], [23], enables the generation of fill-in by alternating between a step of matrix-matrix multiplication to determine new candidate locations for explicit zero elements and Jacobi iterations to determine their value. All these methods' drawback is their applicability, a high degree of diagonal dominance seems to be a condition for convergence. The methods we present in this paper are an attempt at creating a remedy to this problem: (a) Using blocks naturally opens up many different locations for fill-in and (b) full pivoting improves the dominance of diagonal entries during computation.

III. Two-Level Pivoting

Our factorization code works as follows: First, we overlay the input matrix A with a regular grid of a user-defined size and extract all nonzero elements which are then combined in dense blocks. We then set up data structures for organizing

these blocks (see Sec. V). This block structure is fixed for the remainder of the factorization, i.e. blocks are neither created nor deleted; however, larger blocks are notably better at capturing possible fill-in on the sparse level. Thus, the ordering of the matrix together with the size of the regular grid leads to a trade-off between more flexible fill-in and inefficiencies caused by empty blocks.

While factorizing the matrix, we perform all operations on these blocks, including dense factorization of blocks on the block-diagonal. These batched factorizations use special kernels (see Sec. IV). Furthermore, we also permit to postpone whole blocks to the end of the matrix and factor them later. In sum, we offer two levels of pivoting: *inner* pivoting inside of dense blocks and *outer* pivoting.

Our code splits the calling-the-actual-kernels from the analysis and planning phase. We refer to the symbolic part as the *frontend* on the host while the set of kernels makes up the *backend* on the device.

IV. Backend

We start by discussing *inner* pivoting in the backend. This mainly covers the kernels performing a dense numerical factorization of a block. We have presented [8] GPU kernels for $LDL^\top$ factorizations, which similarly process one block per warp and heavily make use of collective operations. Our previous code, however, stores the block in shared memory during factorization, which enables dynamic accesses and therefore allows straightforward pivoting. Recent GPU generations, especially of the Volta and Turing generation, offer an register file of 256 kB per streaming multiprocessor; shared memory is limited to 96 kB for the same hardware unit. Registers greatly exceed shared memory in bandwidth; the downside, however, is that dynamic accesses to registers get translated to local memory accesses – which are just cached, slow global memory accesses. A task like pivoting with its irregular (i.e. dynamic at runtime) access pattern is not a natural fit to register-based kernels.

Full pivoting in registers: Previously, Anzt et al. [24] reported a way to support partial pivoting in register-based kernels: they assign one row each to a thread in the warp; with shuffle-instructions, these threads then swap values with destinations determined at runtime. In the LDU-case, that constitutes partial (row-) pivoting.

Limiting a kernel to either row interchanges *or* column interchanges is straightforward. The situation, however, is more complicated when handling both. A possible alternative traverses the whole matrix after each factorization step and applies masks during the rank-1 downdate. While this eliminates all dynamic accesses, the size of the generated code for $k \geq 16$ negatively affects the instruction cache and lets performance drop below a baseline shared memory version. As a remedy, we propose to allow one dynamic access via binary search per factorization step and amortize this through two measures: (1) explicit permutation to the current column avoids repeated iteration over all columns and (2) usage of masks to fuse rank-1 update, row scaling and pivot update into one step. Essentially, we fuse the last two steps of a classical right-looking factorization into one *fused Schur-complement*. The resulting code is presented in Alg. 1. In the interest of notational brevity, we use 'telling' pseudo-function names, e.g. `swap_with_ix`, for collective primitives that are straightforward to implement with shuffle instructions. Similar to a CUDA kernel, the code snippet is written for one thread with id `t_ix` in its local (sub-)warp. It accesses the `j`-th element of its row by `reg[j]` – iff the index `j` is static. Otherwise, we mark it as a costly dynamic access – requiring a binary search on a runtime index to access the register. An important part of Anzt' kernel is handling the row interchanges *implicitly* - rows are never actually interchanged, each thread only stores the *position* of its row in the matrix. The actual permutation is postponed to when the factorized block is written back to global memory, where dynamic accesses are cheaper.

For full LDU pivoting, we always select the entry of maximum magnitude in the Schur complement. Initially, each thread iterates over its row to determine the maximum (line 2); a warp reduction then determines the first pivot element – (piv_ix_r, piv_ix_c) (line 6). During the whole factorization, thread i keeps a record of which of the original rows resp. columns are permuted to position i in `(t_pivot_r, t_pivot_c)`; these values are exchanged accordingly (lines 9, 10). Since we handle column permutations explicitly, the only necessary dynamic access occurs in line 14: each thread copies the value from the column selected as the next pivot column. These values are scaled and written into the current column at step `s` (line 20). Doing the same with the pivot column would warrant another dynamic access; instead, we defer this part of the permutation to line 34 in the rank-1 downdate: This expression combines copying entries from column `s` to the pivot column and executing the rank-1 downdate for other columns into one expression via operand semantics. We replace all conditions by explicit multiplication with masks to avoid warp divergence. While traversing the columns in the downdate, we also select the maximum magnitude entry per row; by putting all these steps in one loop, we incentivize the compiler to make use of ILP inside the loop. Since registers cannot be manually addressed in CUDA code, we use Cog [25] as code generator to explicitly unroll all loops and avoid any dynamic arrays accesses.

The symmetric ($LL^\top$) case generally mirrors the method in Alg. 1; row and column permutation is then the same. The $LDL^\top$ case with its 2×2-pivots is more complex and generally requires 2 dynamic accesses, but otherwise follows the same ideas. For details, we refer to our code.

We evaluate the resulting kernels in Fig. 1 on a NVIDIA Titan RTX with 24 GB memory and CUDA 10.1 on Ubuntu 18.04 with driver 418.39. We compare the register-based (`GPU−Reg`) with the shared memory version from [8] (`GPU−SMem`) as baseline. Both cuBLAS (`cublasSgetrfBatched`) and MAGMA [26] (`magma_sgetrf_batched_smallsq_shfl`) also offer a batched LU-kernel with partial pivoting; the latter being the

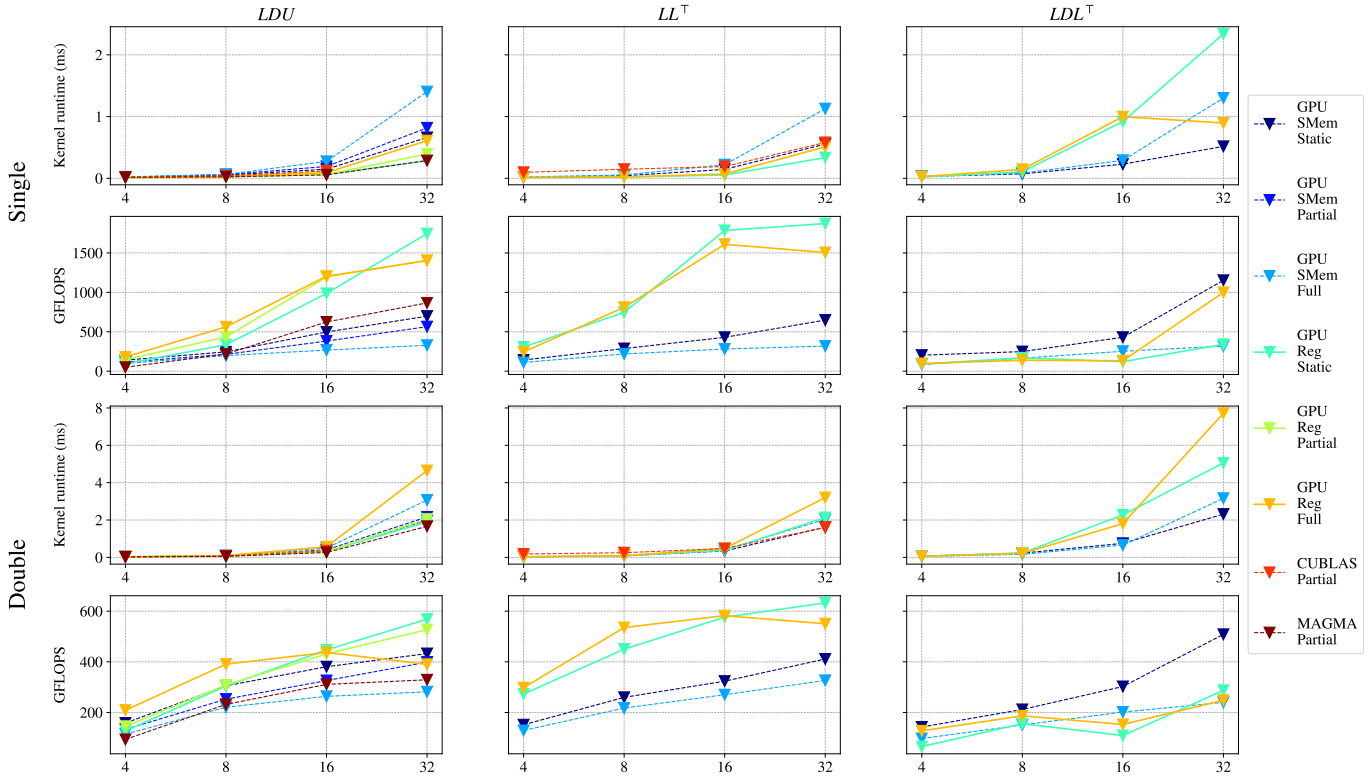

Figure 1: Runtimes and GFLOPs for pivoting kernels of different factorization methods. All runs use a batch size of $10,000$ matrices that are either unsymmetric (LU), symmetric positive definite ($LL^\top$) or symmetric indefinite ($LDL^\top$). Since the double-precision kernels also contain single-precision operations, we count 1 DP-Flop as 2 SP-Flops.

implementation of Anzt et al. [24]. While our code uses CUDA's unified memory interface, we manually prefetched all data and used the profiler to make sure no page faults affect the benchmark results. Our METAPACK backends offer static and full pivoting for all three factorizations as well as a partial LDU-kernel for the sake of comparison. We execute all kernels on random block matrices of sizes $4\times4, 8\times8, 16\times16$ and 32×32 with a batch size of $10,000$. Additionally, we use nvprof to get the FLOP-count of each kernel. While the results for our kernels and MAGMA are on par with [24], we exclude cuBLAS from the GFLOPS plot as the profiler returned results that were beyond the theoretical maximum of the TITAN RTX (16.31 TFLOPS FP32, 0.509 TFLOPS FP64). All timings include load and store operations from and to global memory as well as fetching the frontends' job descriptions. Our LDU-kernels achieve $1,745/1,407/1,402$ GFLOPS for static/partial/full pivoting and single precision using the register-based method from above. MAGMA's partial LDU kernel achieves 898 GFLOPS. Runtimes for the batch were $0.28/0.39/0.60$ ms for ours and 1.41 ms for MAGMA. The full pivoting kernel, despite the much higher FLOP count and symbolic intensity, is only 50% slower than the partial pivoting case; this confirms that Alg. 1 allows the GPU to leverage mainly ILP to compensate for many of the additional FLOPS. In general, the register-based backend beats the shared memory-based kernels; full pivoting in registers is often as fast as static pivoting in shared memory.

A notable exception is the case of $LDL^\top$: To maintain numerical accuracy, static pivoting selects 2×2 pivots in roughly 40% of steps; since we use Givens transformations to solve the resulting 2×2 systems, this especially affects larger block sizes. Hence, different block sizes, operating on different random matrices, cannot be compared - their ratio of 2×2 pivoting is just too different. The full pivoting variant only requires 2×2 pivots 10% of the time on average; thus, it is faster than the static case. here, shared memory variants are generally faster since the high register pressure in the register-based variants cause many accesses to go to local memory.

For double precision, the register-based kernels suffer from high register pressure as well and are also restricted by the FP64-hardware on the TITAN RTX. As a consequence, shared memory kernels are preferable in this situation. Note that, since the double precision kernels also include single precision operations, we regard 1 FP64 operation as 2 FP32 operations in Fig. 1. Notably, despite the slower runtime, the register-based kernels still manage to come close to approx. 60% of the theoretical FP64 performance.

V. Frontend

For very tough matrices where inner pivoting still leads to small pivots, we propose to allow *outer* pivoting, i.e. change the structure of the matrix by pushing blocks from the block

Algorithm 1 Full pivoting kernel for an $n \times n$ - sized LDU factorization (see IV for definitions of variables and symbols used). This code snippet assumes that `t_ix` is the current thread's index into the (sub-) warp.

```
processed_r = False
[t_piv_val, t_piv_ix_c] = max(abs(reg[0:(n − 1)]))
[t_pivot_r, t_pivot_c, t_piv_ix_r] = [t_ix, t_ix,
    t_ix]
for s = 0 : (n − 1)
  % find maximum value in Schur complement as next
      pivot
  [piv_val, piv_ix_r, piv_ix_c] = reduce_with_ix(
      t_piv_val, t_piv_ix_r, t_piv_ix_c)

  % record row and column permutation
  swap_threads(t_pivot_r, s, piv_ix_r)
  swap_threads(t_pivot_c, s, piv_ix_c)

  % fetch values in columns s and the pivot
  cur_col = reg[s]
  piv_col = dynamic(reg, s)

  if(t_ix == piv_ix__r)
    processed_r = True

  % explicitly permute pivot column to position s
  reg[s] = piv_col / (processed_r ? 1.0 : piv_val)

  % fused rank−1 downdate: includes row scale,
      column swap and pivot search
  t_piv_val = 0.0
  t_piv_ix = s

  % mult0: scales the pivot row
  mult0 = 1.0 / ((t_ix == piv_ix_r) ? piv_val :
      1.0)
  for j in (s + 1) : (n − 1)
    src = (j == piv_ix_c) ? cur_col : reg[j]
    o_val = shfl(src, piv_ix_r) / piv_val
    update = o_val * piv_val * reg[s]

    % swtch between update and relacement
    reg[j] = mult0 * src − (!processed_r ? update
        : 0.0)

    % prepare next pivot selection
    if abs(reg[s]) > abs(t_piv_val)
      t_piv_val = reg[s]
      t_piv_ix = j
```

diagonal and their respective rows and columns to the end of the matrix. The subsequent symbolic operations are, depending on the matrix' nonzero structure, expensive. In this section, we propose a simple and effective semi-implicit scheme for block-pivoting in the frontend. We first give details on the data structure involved and then outline the operations that are required to permute a block and its row resp. column to the end of the matrix.

Frontend data structure: After initially locating the nonzero elements in the input matrix A, we create the following management data structure: Similar to the CSR and CSC formats for conventional sparse matrices, we use a *blocked* dual CSR/CSC format. For each block row, we list all blocks as array R_i with index pairs $(i, \texttt{ptr})$ containing the original block column as well as a pointer to the dense block in memory. The same is kept for all block columns as lists C_j; in case of symmetric matrices, we only keep the row-based structure. An example, containing only the row index pairs, is given in Fig. 2a (the green arrows represent C_2, C_3 and the black arrows R_2, R_3). In addition, we save the index of the block on the diagonal in R_i/C_i in arrays $\hat{R}_i, \hat{C}_i$ for convenient access to the lower and upper triangular part of the block matrix. Assuming we chose a large enough block size $k \times k$ and a suitable reordering during preprocessing, the nonzero structure (i.e. the location of nonzero blocks inside the matrix), is significantly smaller than that of the input matrix A.

A. Pivoting

With this structure in mind, we turn to the implementation of *outer* pivoting. In order to create an efficient, parallel pivoting approach, we restrict ourselves to 'push-back' types of permutations, i.e. each operation can pivot a set of rows and columns to the end of the matrix; yet the relative order among them stays constant. Secondly, we only perform symmetric permutations which do not affect the blocks on the block diagonal. This scheme leads to a preconditioner approach that is often also classified as *multilevel* [27], [28] (not to be confused with the level scheduling for parallelization), an *a posteriori* pivoting scheme. During the factorization, blocks with small pivots are recorded and marked for push back; this, however, requires additional storage as such blocks must be rolled back before the pivoting operation. Note that this does not, in any way, result in a loss of generality - we can still generate arbitrary (symmetric) permutations, just at the price of more steps. In our implementation, we maintain 3 arrays:

- P_i which saves the current diagonal block on position i, i.e. the first row and column of the matrix,
- its inverse P_i^{-1}, pointing to the current position of (initial) diagonal block i and
- X_i which store a diagonal block's level.

These levels in X_i are an upper bound to levels resulting from level scheduling based preconditioners. All pivots pushed back in one level will stay together in that level, so no additional dependency analysis is necessary.

A 'push-back' operation executes the following steps:

1) Increment the X entries for diagonal blocks to be permuted,
2) execute a stable sort on the diagonal block ids by their respective level; this results in the updated array for P.
3) Invert P into P^{-1}.
4) Sort all index pairs $(i, \texttt{ptr})$ in R, C by comparing diagonal blocks by their entries in P^{-1}.
5) Lastly, update arrays $\hat{R}, \hat{C}$.

An example of a 5×5 matrix pushing diagonal blocks 2 and 3 is given in Fig. 2. Following the steps above, we ask the reader to draw his attention to the following points: (a)

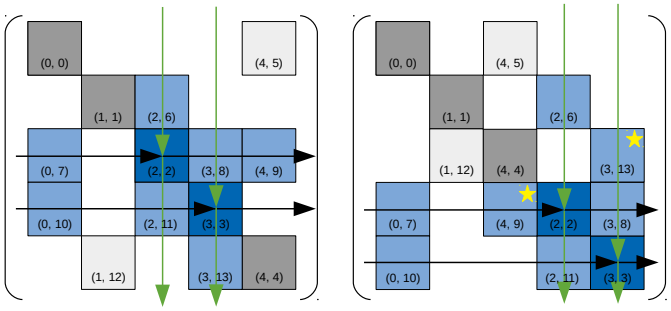

Figure 2: Example of a 'push-back' type pivoting operation: we permute rows and columns 2, 3 symmetrically to the rear of the matrix and sort the block references according to their new value in `piv_position` (see Sec. V). Note: only the index pairs for R. are shown.

the most expensive step (4.) operates on all diagonal blocks independently, parallelized over diagonal blocks; (b) no actual permutation of diagonal blocks in memory is performed, we solely operate on indexes. After a permutation, the index pairs do not represent valid column resp. row coordinates any more. Instead, the frontend always needs to map those via P^{-1}; in this sense, the index pairs are always sorted. Although indirecting through these indices seems like a hassle, it can be a benefit for the symmetric case: Consider the two blocks marked with yellow stars in Fig. 2b: these have switched their sides of the block-diagonal, i.e. a block from $L^\top$ moved into L and vice versa. Since we do not store the blocks of L, $L^\top$ twice in memory, such blocks need to transparently transposed in the backend. Due to our pivoting scheme, we already imply this marker: if we encounter an index pair in L whose first component is larger that the *initial* column of this diagonal blocks' diagonal block, the block should be transposed (and vice versa for $L^\top$).

After such pivoting operations, the job schedules for subsequent levels need to be regenerated. To avoid additional latency, scheduling higher levels can be executed in a separate host thread while the backend starts working on the first, newly scheduled levels.

VI. A modified Jacobi-$LDL^\top$ algorithm

The fixed point equations of Eq. 2 are well suited for (massively-) parallel systems: each equation is indepedent from others, enabling the exploitation of coarse grained parallelism in the hardware. A sufficient condition for convergence of the fixed point iteration is diagonal dominance of the matrix A; for symmetric indefinite matrices, that is rarely the case. Pivoting operations during numerical factorizations are designed to move large elements to the matrix' diagonal. Therefore, we hypothesize that embedding pivoting in *Jacobi* (fixed-point) factorization should help with convergence. To that end, we adapt Eq. 2 for the $LDL^\top$ case and track the local permutations from inner pivoting in Alg. 2. Two points

Algorithm 2 Embedding pivoting and blocking into the Jacobi $LDL^\top$ algorithm. Returns L'', D'' that minimizes $||L''D''L''^\top - A||_\infty / ||A||_\infty$ for a symmetric, indefinite matrix A. For convenience, we refer to block (i, j) in the blocked version of A as $A(i, j)$. p are permutation vectors for inner pivoting.

$L'' = L' = L \leftarrow \text{tril}(A), D'' = D' = D \leftarrow I, p'' = p' = p = I$
for $sweep = 1, \ldots$ **do**
  $L' \leftarrow L'', D' \leftarrow D'', p' \leftarrow p''$
  for $i = 1 \ldots m$ **do**
    $B \leftarrow L(i,i)(p'(i), p'(i)) - \sum_{k<i} L'(i,k)D'(k)L'(i,k)^\top$
    $[L''(i,i), D''(i), q] \leftarrow \text{LDL}(B, \epsilon||A||_1)$
    $p''(i) \leftarrow p'(i)(q)$
  end for
  for (i, j) where $j < i$ **do**
    $C \leftarrow L(i,j)(p'(i), p'(j)) - \sum_{k<i} L'(i,k)D'(k)L'(j,k)^\top$
    $L''(i,j) \leftarrow C(p''(i), p''(j))L''(j,j)^{-T}D''(j)^{-1}$
  end for
  $\epsilon \leftarrow \epsilon\delta$
end for

are the deciding factors for convergence even in the symmetric indefinite case: (1) local permutations must be updated in sync with L', D', i.e. whenever accessing L', D', we also have to access p' to adapt the initial guess to the current iterate's permutation; (2) due to the lack of diagonal dominance, we frequently encountered small pivots, leading to large values in the off-diagonal blocks and, in consequence, divergence. As a remedy, we found aggressive pivot perturbations especially helpful: we set the ϵ in Alg. 2 to large values such as 0.1 or 0.01. This usually stabilizes the iterates, such that it can be relaxed after each sweep by multiplication with $\delta < 1$. In out experiments, we use $\epsilon = 0.1, \delta = 0.95$. Going one level further, outer pivoting can be included as well: after the first sweep, we detect and push problematic diagonal blocks back. Then, all block accesses in Alg. 2 need to go through P^{-1}. Lastly, we remark the following: When processing positive definite, but not diagonal dominant matrices, some diagonal blocks turn indefinite during the factorization; thus, the $LDL^\top$-variant also covers that case.

VII. Evaluation

We implemented the modified Jacobi algorithm and the pivoting data structures as part of our experimental linear algebra code METAPACK. METAPACK is focused on researching new algorithms and parallelization concepts, stressing adaptivity and modularity over performance. Since this paper is mostly concerned with enabling the use of Jacobi-factorizations (in comparison with the conventional level scheduling parallelization), we leave a performant implementation to future work. Despite that fact, Tab. I gives some timings for level scheduling factorizations. All experiments

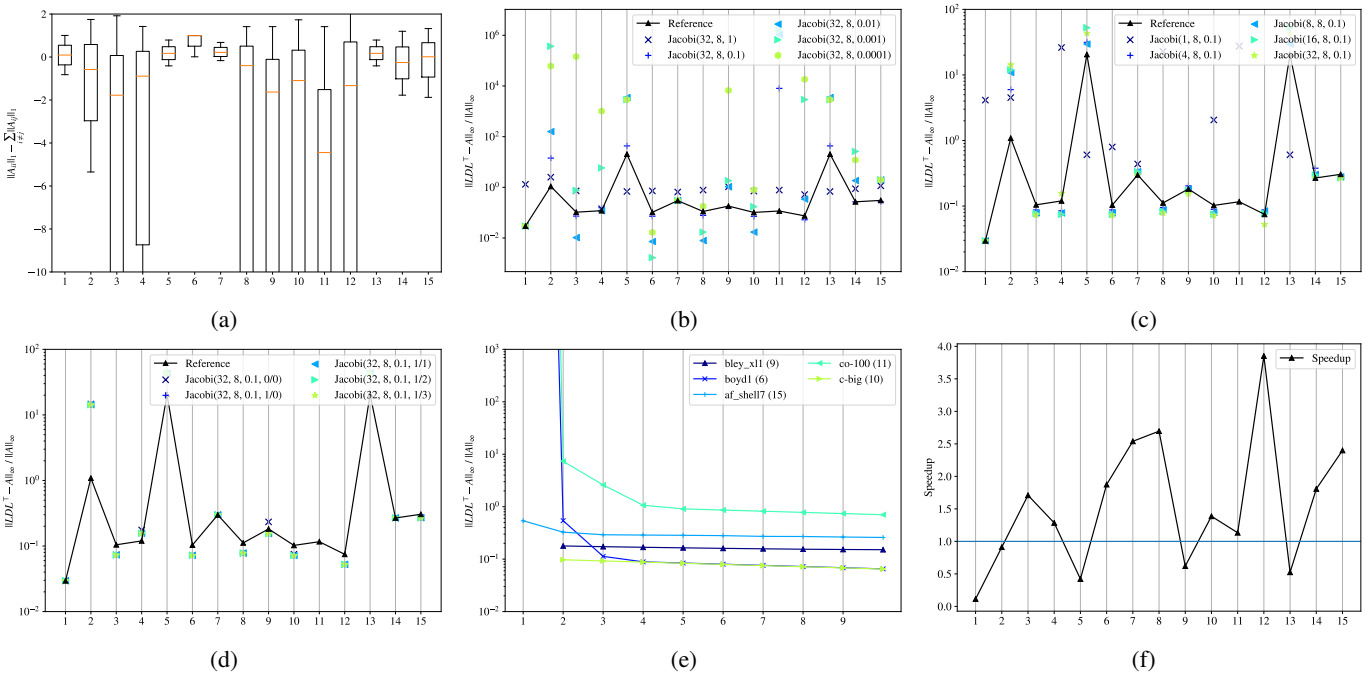

Figure 3: Plots for ablation experiments.

	Jacobi(32, 0.1)		Multilevel(32, 0.1)		
Matrix	**B**	**R**	**B**	**R**	**t**
qpband	-6	-8	-6	-8	1.2 s
mario001	-1	-1	-1	-1	0.4 s
bab5	-4	-1	-4	-1	0.6 s
c-72	-3	-1	-3	-1	0.9 s
Si10H16	-4	-3	-4	-3	1.2 s
boyd1	-5	-4	-5	-4	0.9 s
Lin	-4	-3	-4	-3	0.3 s
map06	-4	-2	-4	-2	1.9 s
bley_xl1	-3	-2	-3	-2	4.3 s
c-big	-4	-2	-4	-2	2.3 s
co-100	-3	-2	-3	-2	2.4 s
bab3	-3	-2	-4	-2	8.3 s
G3_circuit	-4	-3	-4	-3	0.9 s
BenElechi1	-5	-5	-5	-6	0.9 s
af_shell7	-5	-4	-5	-4	1.2 s

Table I: Using the computed preconditioner to solve matrices using 4 runs of GMRES(25). **B** refers to the backward error $\log(||b - Ax||_\infty/(||A||_\infty||x||_\infty + ||b||_\infty))$; **R** is the unpreconditioned relative residual $\log(||b - Ax||_2/||b||_2)$ - **t** marks the computation time for the multilevel preconditioner.

were executed on the same machine as those in Sec. IV with 8 Jacobi sweeps.

We evaluate our method on a set of 15 matrices from our earlier work [8] in that paper's order; all matrices are symmetric indefinite and sparse. Every metric is permuted and scaled with MC64 and then overlapped with a regular grid. Since the original Jacobi method is sensitive to diagonal dominance, Fig. 3a lists the minima, average and maxima of the quantity $||A_{ii}||_1 - \sum_{i \neq j} ||A_{ij}||_1$, a measure of the degree of diagonal dominance, over block rows. We notice a clear separation: while matrices 1, 5, 6, 7, 13, 14, and 15 are close to diagonal dominance, the other matrices are far from it. Applying a regular Jacobi method to the latter leads to strong divergence. No fill-in is added nto the initial block structure on input matrices.

In Fig. 3b - 3d, we present ablation studies for (b) the threshold ϵ, (c) the block size k as well as the the effectiveness of both inner and outer pivoting in (d). In all cases, we use the reconstruction error $||LDL^\top - A||_\infty/||A||_\infty$; in our experience, this error behaves similar to the backward error when solving a linear system with the computed factorization as preconditioner. Results for the backwards error and relative residual in that case are given in Tab. I; he Jacobi results were generated with the best setting extracted from Fig. 3b per matrix. In all experiments, our reference is a level scheduling factorization (same backend, but same frontend) with otherwise the same parameters.

Additionally, we also used Magma's ParILU and ParILUT [26] and applied it to all systems as a reference. However, we did not get a single success - no system ever came beyond a residual of 1e-1; most did not finish the computation of the preconditioner. This lines up with our earlier, independently produced results [8]. In the interest of a fair comparison, we implemented a scalar variant of Jacobi-$LDL^\top$, see Fig. 3c with $k = 1$.

In Fig. 3b, we notice the importance of a strong ϵ for non-diagonal dominant matrices: small thresholds such as 0.001 or 0.0001 often lead to reconstruction errors magnitudes over the reference. On the contrary. a large threshold seems to contain

numeric artifacts and help with convergence. For almost all matrices, we can find a parameter setting such that the Jacobi preconditioner matches or outperforms the level schedules one in its error. Furthermore, such a high ϵ also dampens the sensitivity with regard to block size k: As Fig. 3c reports, the results of most block sizes over $k = 4$ show only minor differences. The scalar Jacobi algorithm, however, diverges strongly from that: its error is mostly more than two orders of magnitude higher, since it is restricted to its initial nonzero structure. Especially when dealing with indefinite matrices, capturing relevant fill-in is crucial.

During most of the Jacobi sweeps, the high ϵ keeps all pivots in such a range that inner pivoting and, surprisingly, outer pivoting, do not play a striking role in the final error of the computation (see Fig. 3d). Beyond the Jacobi method, inner pivoting can still make or break a factorization in traditional schedules.

Lastly, we investigated the optimal number of sweeps and estimated the performance benefits of using the Jacobi method. As visible in Fig. 3f, the factorization phase can be accelerated in this way; integrated implementations of frontend and backend could offer even more speedup. Fig. 3e resembles the curves in Chow and Patel's [2] original paper: saturation often accurs after the fifth sweep – for tough matrices.

VIII. Conclusion and Future Work

We presented blocking and a two-level pivoting scheme as an approach to use massively-parallel Jacobi-schemes for preconditioning even for numerically tough matrices. We found that aggressive pivot thresholding presents a useful remedy to a lack of diagonal dominance. Our results show that paired with the right ingredients, these methods can close the gap to traditional preconditioners. In future work, we would like to try the same methods on the triangular solves that are critical for accelerating iterative solvers and study the effects of preprocessings such as reorderings and scaling on the method. Furthermore, we would like to experiment with the extension to exact factorizations. Our code will be released on the author's Github[1] page.

Acknowledgements

D. Thuerck is supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt. We are grateful for the anonymous reviewers' comments on the initial submission.

References

[1] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.

[2] E. Chow and A. Patel, "Fine-grained parallel incomplete LU factorization," *SIAM J. Sci. Comput.*, vol. 37, no. 2, pp. C169–C193, 2015.

[3] I. S. Duff and J. Koster, "On algorithms for permuting large entries to the diagonal of a sparse matrix," *SIAM Journal on Matrix Analysis and Applications*, vol. 22, no. 4, pp. 973–996, 2001.

[4] Y. Saad, "ILUT: A dual threshold incomplete LU factorization," *Numerical linear algebra with applications*, vol. 1, no. 4, pp. 387–402, 1994.

[5] E. Chow and Y. Saad, "Experimental study of ILU preconditioners for indefinite matrices," *Journal of Computational and Applied Mathematics*, vol. 86, no. 2, pp. 387–414, 1997.

[6] Y. Saad, *Iterative methods for sparse linear systems*, 2003, vol. 82.

[7] M. Bollhöfer, "High performance block incomplete LU factorization," 2017.

[8] D. Thuerck, M. Naumov, M. Garland, and M. Goesele, "A block-oriented, parallel and collective approach to sparse indefinite preconditioning on gpus," in *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2018, pp. 1–10.

[9] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker, "Pardiso: a high-performance serial and parallel sparse linear solver in semiconductor device simulation," *Future Generation Computer Systems*, vol. 18, no. 1, pp. 69–78, 2001.

[10] O. Schenk and K. Gärtner, "On fast factorization pivoting methods for symmetric indefinite systems," *Electronic Transactions on Numerical Analysis*, vol. 23, pp. 158–179, 01 2006.

[11] C. Greif, S. He, and P. Liu, "SYM-ILDL: Incomplete LDL^T factorization of symmetric indefinite and skew-symmetric matrices," *ACM TOMS*, vol. 44, no. 1, p. 1, 2017.

[12] Y. Saad, "Finding exact and approximate block structures for ilu preconditioning," *SIAM J. Sci. Comput.*, vol. 24, no. 4, pp. 1107–1123, 2003.

[13] Y. Saad and J. Zhang, "BILUM: block versions of multielimination and multilevel ILU preconditioner for general sparse linear systems," *SIAM J. Sci. Comput.*, vol. 20, no. 6, pp. 2103–2121, 1999.

[14] M. Götz and H. Anzt, "Machine learning-aided numerical linear algebra: Convolutional neural networks for the efficient preconditioner generation," in *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*. IEEE, 2018, pp. 49–56.

[15] M. Naumov, "Incomplete-LU and Cholesky preconditioned iterative methods using cusparse and cublas," *Nvidia white paper*, 2011.

[16] M. Naumov, P. Castonguay, and J. Cohen, "Parallel graph coloring with applications to the incomplete-lu factorization on the GPU," *Nvidia Tech. Rep. NVR-2015-001*, 2015.

[17] J. W. Liu, "A compact row storage scheme for Cholesky factors using elimination trees," *ACM TOMS*, vol. 12, no. 2, pp. 127–148, 1986.

[18] T. A. Davis, *Direct methods for sparse linear systems*, 2006, vol. 2.

[19] J. Hogg, J. Reid, and J. Scott, "Design of a multicore sparse Cholesky factorization using DAGs," *SIAM J. Sci. Comput.*, vol. 32, no. 6, pp. 3627–3649, 2010.

[20] H. Anzt, J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana-Ortí, "Adaptive precision in block-jacobi preconditioning for iterative sparse linear system solvers," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 6, p. e4460, 2019.

[21] T. Grützmacher, T. Cojean, G. Flegar, F. Göbel, and H. Anzt, "A customized precision format based on mantissa segmentation for accelerating sparse linear algebra," *Concurrency and Computation: Practice and Experience*, p. e5418, 2019.

[22] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning," in *Eur. Conf. on Parallel Proc.*, 2015, pp. 650–661.

[23] ——, "ParILUT – a new parallel threshold ilu factorization," *SIAM J. Sci. Comput.*, vol. 40, no. 4, pp. C503–C519, 2018.

[24] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí, "Variable-size batched LU for small matrices and its integration into block-jacobi preconditioning," in *Parallel Processing (ICPP), 2017 46th International Conference on*. IEEE, 2017, pp. 91–100.

[25] N. Batchelder. (2004) cog - code generator. [Online]. Available: https://nedbatchelder.com/code/cog/

[26] A. Tomov, R. Nath, P. Du, and J. Dongarra, "MAGMA: Matrix algebra on GPU and multicore architectures," 2012.

[27] M. Bollhöfer and Y. Saad, "Multilevel preconditioners constructed from inverse-based ILUs," *SIAM J. Sci. Comput.*, vol. 27, no. 5, pp. 1627–1650, 2006.

[28] M. Bollhöfer, Y. Saad, and O. Schenk, "ILUPACK-preconditioning software package," *http://ilupack.tu-bs.de/, Rel. 2, 2016.*

[1] https://github.com/dthuerck