A block-oriented, parallel and collective approach to sparse indefinite preconditioning on GPUs

Daniel Thuerck*
Graduate School CE
TU Darmstadt, Darmstadt, Germany

Maxim Naumov*
Facebook
Menlo Park, CA

Michael Garland
NVIDIA
Santa Clara, CA

Michael Goesele
TU Darmstadt, Darmstadt
Germany

***Abstract*—Large sparse symmetric indefinite matrices are notoriously hard to precondition. They often lack diagonal dominance and exhibit Schur-complements that render zero fill-in preconditioning ineffective. Pivoting, a necessity for stable $LDL^\top$ factorizations, complicates parallel approaches that can take advantage of the latest massively-parallel HPC hardware such as GPUs. We present an approach based on ad-hoc blocking and reordering strategies that allows local, independent collective-oriented processing of small dense blocks. A hybrid block-memory layout compensates for irregular memory access patterns found in sparse matrices. Our method allows restricted fill-in, supernodal pivoting and a dual threshold dropping strategy at little additional cost. It delivers robust preconditioners that in our experiments obtain an average speedup of $\sim 6\times$ even for tough matrices from optimization problems.**

I. Introduction

We consider the solution of linear systems $A\mathbf{x} = \mathbf{b}$ where the coefficient matrix $A \in \mathbb{R}^{n\times n}$ is sparse and symmetric indefinite with a solution vector $\mathbf{x} \in \mathbb{R}^n$ and right-hand-side vector $\mathbf{b} \in \mathbb{R}^n$. These linear systems appear frequently in scientific computing, most notably in Newton-based methods in numerical optimization.

The direct methods for solving symmetric indefinite systems often rely on the $LDL^\top$ factorization, with $L \in \mathbb{R}^{n\times n}$ being a lower-triangular matrix with unit diagonal and $D \in \mathbb{R}^{n\times n}$ a block-diagonal matrix with 1×1 and 2×2 blocks to avoid breakdown due to zeros on the diagonal. This factorization can be performed with different (symmetric) pivoting startegies (partial pivoting, Bunch and Kaufman [8] and full symmetric *Rook* pivoting). Further, in production-grade implementations, the coefficient matrices are preprocessed by permuting the largest elements close to the diagonal, equilibrating rows and columns by scaling, and finding a fill-reducing reordering, e.g.

$$(Q^T S P^T A P S Q)(Q^T S^{-1} P^T \mathbf{x}) = (Q^T S P^T \mathbf{b}) \qquad (1)$$

where $P \in \mathbb{R}^{n\times n}$ is a permutation, $Q \in \mathbb{R}^{n\times n}$ is the reordering and $S \in \mathbb{R}^{n\times n}$ is a diagonal scaling matrix.

Iterative methods can be an alternative to the direct approach. In particular, with the broad adoption of parallel computing platforms, the field of Krylov-based linear solvers has seen an explosion of research activities. Iterative methods feature sparse matrix-vector multiplications as their major computational primitive, attractive for SIMD-style execution. A drawback of these methods is the need of finding a good *preconditioner*, a matrix M such that $M^{-1} \sim A^{-1}$ and linear systems with M being relatively easy to solve. The best preconditioners are often problem-specific, but they require significant time and effort to be constructed. In the worst case, the construction itself is not amenable to parallelization. On the other hand, incomplete factorizations are common "black-box" solutions [11]. They combine the advantages (and disadvantages) of direct and iterative methods. While they benefit from techniques used by direct solvers, they need to minimize computation in order to leave enough time for multiple solves during steps of the iterative method.

In order to be efficient, modern incomplete factorization algorithms must make good use of available hardware with its complex memory hierarchy. By tuning several knobs (preprocessing mechanisms, reordering schemes, assembly of dense operations and the amount of allowed fill-in), the user can balance between parallelism in the construction and solve phases as well as quality of the preconditioner to minimize the overall time-to-solution.

We propose a compromise that combines techniques from full-blown $LDL^\top$ with dropping techniques for incomplete factorizations. We show that it is well-suited for parallel computing platforms by illustrating its performance on GPUs. Additionally, we investigate the effect of widely used preprocessing methods, cornerstones of modern indefinite solvers, with respect to their effect on parallelism in the factorization. Our contributions are:

- We develop block-oriented $iLDL^\top$, the first tile-based incomplete factorization that runs completely on the GPU and offers supernodal partial pivoting, fill-in and a dual threshold dropping strategy. Our method solves more systems than a state-of-the-art CPU-based preconditioner with full pivoting and achieves speed-ups of up to $\sim 22\times$ with an average speedup of $\sim 6\times$.
- We propose two blocking and reordering schemes which result in a tile-based matrix structure, that is ideally suited to exploit local operations on the GPU cores.
- We offer our implementation as an open-source package. Our code is templated to allow both single and double precision computations.

II. Background

In this section, we outline the basic concepts for (incomplete) factorizations and list notable related work. The vast

*(Parts of this) Work done while at NVIDIA.

majority of implementations employ a fixed preprocessing pipeline. A matching-based permutation first brings the largest entries close to the diagonal; a symmetrization of the optimal matching from MC64 (see [14], [18]) also delivers a scaling matrix as well as a static pivot order with 1×1 and 2×2 pivots. The largest elements in the resulting matrix have an absolute value of 1.0, with the largest elements mostly near the diagonal.

Before applying another reordering to minimize fill-in during a factorization, the matrix is compressed such that 2×2 pivots are kept. Such fill-in reducing reorderings are often based on either greedily permuting the elimination tree, see (SYM)AMD [2] or using nested dissection, see Karypis [22].

Therefore, the pipeline in (1) serves three purposes: Find an initial pivoting order, reduce the condition number of the matrix, and reduce the amount of fill-in, thereby decreasing time and memory consumption.

A. $LDL^\top$ factorization

Of the different formulations for $LDL^\top$ factorization, we use the right-looking, "multifrontal" variant derived from unrolling the following 3×3 block factorization from the top

$$\begin{pmatrix} A_{11} & A_{21}^\top & A_{31}^\top \\ A_{21} & A_{22} & A_{32}^\top \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} D_{11} & & \\ & D_{22} & \\ & & D_{33} \end{pmatrix} \begin{pmatrix} L_{11}^\top & L_{21}^\top & L_{31}^\top \\ & L_{22}^\top & L_{32}^\top \\ & & L_{33}^\top \end{pmatrix} \quad (2)$$

leading to the following steps, where $(A_{11}^\top \; A_{21}^\top \; A_{31}^\top)^\top$ denotes the previously processed columns and $(A_{21} \; A_{22}^\top \; A_{23}^\top)^\top$ the current (block) column, where A_{22} is a $k \times k, k > 0$ block:

$$L_{22} D_{22} L_{22}^\top = A_{22} - L_{21} D_{11} L_{21}^\top \quad (3)$$

$$L_{32} = (A_{32} - L_{31} D_{11} L_{32}^\top) L_{22}^{-\top} D_{22}^{-1} \quad (4)$$

$$L_{33} D_{33} L_{33}^\top = A_{33} - L_{31} D_{11} L_{31}^\top - L_{32} D_{22} L_{32}^\top \quad (5)$$

Each step is defined on the Schur complement, i.e., the result of previous rank-k updates (5). In a scalar $LDL^\top$ factorization with $k \leq 2$, this yields $L_{22} = I$ and thus 1×1 and 2×2 blocks on D's block-diagonal. For larger blocks, (3) requires a separate, often dense factorization of the updated A_{22}.

In full factorizations of sparse matrices, the final nonzero pattern is determined in a symbolic stage before the numeric factorization. Analysis of the Gaussian elimination steps performed during factorization leads to the elimination tree [25], compactly representing the final layout [12]. Permuting the matrix by a postordering of the elimination tree often leads to a packing of the nonzero elements, and thus better cache use.

Apart from computing the D_{11}-weighted outer product of L_{21} in (3) and simple (block-)diagonal scaling in (4), updating the A_{33} block in (5) is the most time-consuming computational primitive. For sparse matrices, this yields many indexing operations and irregular memory accesses. Many major performance improvements of direct methods in the past 25 years are due to organizing these updates in a way that allows efficient use of smaller, dense kernels. Starting from a postordered matrix, neighboring rows (resp. nodes in the elimination tree) with the same or similar nonzero pattern are *amalgamated*, resulting in "supernodes" that are represented during the factorizations as small, dense matrices [16]. Operations on these smaller matrices are handed off to highly-optimized BLAS libraries, which more than offset the costs of the additionally required scatter and gather operations.

For a reliable $LDL^\top$ factorization, partial pivoting is often mandated. In general, pivoting operations change the elimination tree, thus triggering an additional round of symbolic analysis. Highly optimized packages thus use methods such as deferred pivoting [20] or restrict pivots to supernodes [36].

B. Incomplete factorizations

An incomplete $LDL^\top$ factorization results in matrices $\tilde{L}, \tilde{D}$ such that $(\tilde{L}\tilde{D}\tilde{L}^\top)^{-1} \sim A^{-1}$. Incomplete factorizations drop elements according to a set of rules. Most implementations adhere to one or a combination of the following principles:

- *Level-of-fill*: Let the level of all nonzero elements of A be defined as 0. Then, other element's level is recursively set to $\text{lvl}(l_{ij}) = \max_{k<\min(i,j)}\{\text{lvl}(l_{ik}), \text{lvl}(l_{kj})\} + 1$. With the assumption that the magnitude of elements shrinks with higher levels, which holds for specific matrices (e.g., five-point matrices [34]) the permitted level-of-fill is bounded by the user [11], [27]. Such factorizations are often denoted by $LDL^\top(k)$ for level k.
- *Threshold dropping*: During factorization, all elements with magnitude smaller than a fixed threshold τ times the row/column 2-norm are dropped. Additionally, a specific capacity threshold c_i per i-th row/column can be set prior to starting the factorization so that only the largest c_i elements are kept per row [5], [7], [17], [32]. These threshold-based methods have been found to work better on general matrices, where the assumption regarding magnitude decay over levels of fill does not apply.

Due to dropping, near-zero pivots occur and can be dealt with by adding a perturbation onto the input matrix or postponing that pivot to a later step [6], [7].

C. Parallel approaches

There are mainly three sources of parallelism to be explored when solving linear systems: First, replacing dense linear algebra calls by parallel alternatives [5], [24], [30], [36]. Second, analyzing the dependecies in the nonzero sparsity pattern of the matrix. Then, the resulting dependency graph is either executed directly as in DAG-based methods [19] or level sets of rows/cols are discovered and explored [1], [28]. In particular, greedy multicolor (GMC) reorderings have been shown to shrink the number of level sets [26], [27], [35]. Third, a number of authors have proposed methods that compute incomplete factorizations for near diagonal-dominant matrices by iteratively solving a system of nonlinear equations by a fixed-point, asynchronous Jacobi method. This approach is applicable to both solution of triangular systems [3] and ILU(k)/IC(k) factorizations [10]. A recent extension enables a form of threshold dropping [4].

D. Recent trends in GPU architecture

One specific instance of massively parallel HPC hardware are NVIDIA's CUDA-enabled GPUs. Starting with the introduction of the Kepler architecture, the card's compute units are partitioned in more but smaller entities. I.e., the number of streaming multiprocessors (SMs) has grown faster than the number of CUDA cores (i.e. threads). Each SM schedules threads in groups of 32, so-called *warps*. These groups can be cheaply synchronized and share their registers using collective operations, such as warp-shuffles. Additionally, the size of the SMs' register files has grown and can now be used more effectively together with the warp operations. Finally, the Volta architecture has "Tensor Cores" for fast 4×4 matrix multiplication in half precision. The combination of these developments allow fast processing of local data in a cooperative manner in small groups of threads, e.g. warps.

III. Blocking, Reorderings and Parallelism

The effectiveness of a block-based factorization crucially depends on the partition of a sparse matrix into blocks. Finding and exploiting block-structures has been investigated by many authors (see, e.g., [5], [31], [33], [35]). In general, the objective of "blocking" the matrix A, i.e. permuting and determining the block starts, is to minimize the number of filled blocks while maximizing the number of nonzero entries in each block. Ideally, not just A's entries, but also future fill-ins during the factorization are considered. Since we use GPUs and heavily rely on warp-centric operations, most block-based operations take roughly the same amount of time as a 32×32 block. Consequently, we need to find much larger blocks than in previous work. If these blocks happen to be sparse, our hybrid block format still avoids wasted storage. As outlined in Sec. II, several reorderings have been used for varying purposes, e.g. reduce fill-in or the number of level sets. Since both reordering and blocking affect the number of level sets as well as the quality of the resulting preconditioner, we outline two strategies for computing blockings. The two strategies are then compared considering the level-of-fill in a factorization and their effect on parallelism.

A. Blocking strategies

We distinguish two different blocking strategies:

1) Strategy BR: In order to find blocks, we first group rows of the matrix with similar patterns and permute them next to each other. Applying that permutation to both sides of the matrix immediately yields a blocking structure. To detect rows with similar layouts, we use cosine-based blocking [33].

As Bollhöfer [5] notes, this often leads to small blocks far from 32×32. We thus adapt a multilevel strategy: since our hybrid block format can efficiently handle larger, sparse blocks to a certain degree, we apply the same cosine-based dropping to the coarse matrix, built from the blocks found in the earlier step. This method can be repeated recursively until the coarse matrix is sufficiently small. To facilitate merging on coarser levels we decrease the similarity threshold over the hierarchy; we start with a threshold of 0.8 and reduce it by a factor of 0.9 per iteration. Finally, on the last level, we apply the selected reordering (AMD, RCM or GMC).

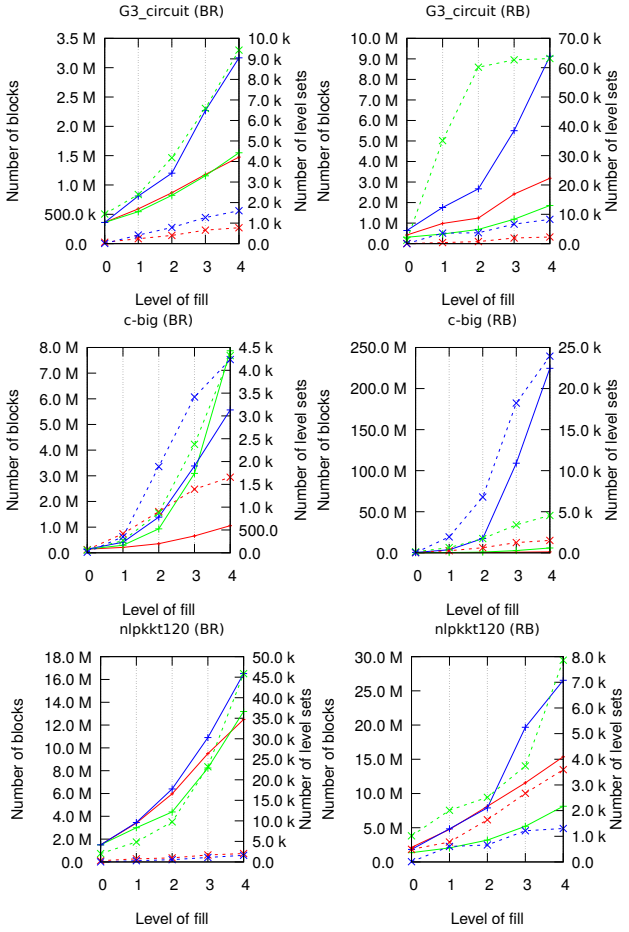


Figure 1: Statistics and patterns for BR and RB strategies, including the number of nonzero blocks (solid line) and level sets (dashed line) for different levels of fill with AMD (red), RCM (green) and GMC (blue) reorderings.

2) Strategy RB: We first apply a reordering to the coefficient matrix and then perform blocking. Again, we use cosine-based blocking; different than before, the blocking must preserve the ordering of rows and columns - thus, only neighboring rows can be clustered into the same block.

In the former strategy we are applying the reordering on a smaller coarse matrix C, while cosine-based blocking requires, in the worst case, to compute $C_k C_k^\top$ at each level k of the hierarchy. In the latter strategy we only need to compute a superdiagonal of $AA^\top$ once, but the effectivenes of the blocking depends highly on the reordering success in clustering rows with similar sparsity pattern.

B. Effectiveness and parallelism

We applied both strategies to our test set (see Sec. V). Fig 1 shows results for three matrices representing the trends found in the whole dataset. The matrices in the test set can effectively be separated into "deep" and "shallow" matrices with regard to their number of level sets. We omitted the option to not reorder

the matrix as these initial layouts often led to a massive fill-in, exceeding the host's memory capacity.

The first two rows in Fig. 1 show deep matrices. Here, fill-in reducing orderings succeed in keeping the number of fill-in blocks low over the levels of fill; GMC reordering, on the other hand, quickly adds a high amount of fill-in and looses its level-set reducing effect. While RCM sometimes creates less fill-in than AMD, especially for a small bandwidth matrix, packing elements close to the diagonal increases the number of level sets fast. Interestingly, in G3_circuit with RB ordering, the level sets created from RCM saturate fast; a sign that few levels of fill-in get close to the full factorization. This matrix is also one of the few examples where the RB strategy dominates BR; the initial layout of the matrix already exhibits similar row patterns close to the diagonal. In both shallow matrices, the GMC reordering distributes relatively dense submatrices over the independent sets, leading to fill-in later. nlpkkt120 is one of the few "shallow" matrices in our set. Even though GMC quickly adds more fill-in over the levels, it keeps the number of level sets small in both strategies. Shallow matrices often benefit from parallel processing for both incomplete and full factorizations.

These plots highlight two points: First, the effect of reorderings when adding fill-in is highly matrix-dependent, including the choice of strategy. For the majority of the matrices, BR was the better choice as the early row amalgamation changes rows' patterns, making them more similar. Contrary to zero-fill preconditioners, the GMC reordering delivers poor results on deep matrices. AMD and RCM, as fill-in reducing permutations, consider future fill-in when permuting the matrix, GMC does not. Second, the classical choice for full factorizations, AMD, offers a decent compromise. We did not find any example where AMD delivered the worst results of these three reorderings.

IV. A BLOCK-ORIENTED $iLDL^T$ WITH PIVOTING

We now present a block-oriented $iLDL^\top$ preconditioner that benefits from the recent trends in GPU architecture briefly mentions in Sec. II-D. We assume that the input matrix A has been appropriately permuted and scaled in the preprocessing phase. Additionally, we assume a partition of rows (and, due to symmetry, columns) has been computed, as shown in the previous section. We use the blocks resulting from the partition in two ways: first, they fit into an 32×32-sized tile of shared memory for on-chip processing; second, we allow arbitrary fill-in inside of all allocated blocks but none outside. The resulting preconditioner is thus a hybrid of a level-based $iLDL^\top(p)$ and a pure, thresholded $iLDL^\top(\tau)$ factorization. It is characterized by three parameters: The level-of-fill l_f that defines which of the blocks in the partitioned matrix may contain entries, a threshold τ for dropping of elements and a fill ratio r_f that determines how many elements each block may contain. All set-up is executed on the CPU; after upload, the complete factorization is performed by the GPU.

Algorithm 1 Block-oriented $iU^\top DU$ factorization (with deferred updates for one level $\mathcal{L}_i$)

```
for block row i ∈ L_l do
    [U_ii] ← apply_sym_updates(U_ii)
    [U_ii, D_ii, P_ii] ← factor_diagonal_block(U_ii)
    for block (i, j), j > i in block row i do
        [U_ij] ← apply_nonsym_updates(U_ij)
        [U_ij] = (P_ii U_ij) U_ii^{-1} D_ii^{-1}
    end for
    [r] = rowwise 2-norms of [U_ii ... U_in]
    for block (i, j), j > i in block row i do
        [U_ij] ← = block_row_dualdrop(U_ij)
        compress_store(U_ij)
    end for
end for
```

A. Preprocessing and Setup

Initially, the sparse input matrix A in CSR format is expanded into COO-format and all elements are mapped to their respective blocks. We then sort the resulting coordinate pairs and, after reduction, produce a COO description of the blocked ("coarse") matrix. We then generate CSR and CSC representations for the coarse matrix. We work on a transpose of L and thus actually compute an $iU^\top DU$ factorization.

We add blocks for fill-in according to l_f. Hysom's algorithm [21] allows computing such fill-in in parallel per row. With new blocks integrated into the coarse CSR representation, we then collect A's nonzero entries per block and determine the maximum number of elements per block as $\text{max\_nnz}(U_{ij}) = r_f \cdot \text{nnz}(U_{ij})$. For (empty) fill-in blocks, we base their allowed fill on the average number of nonzero elements in all occupied blocks in the same block row.

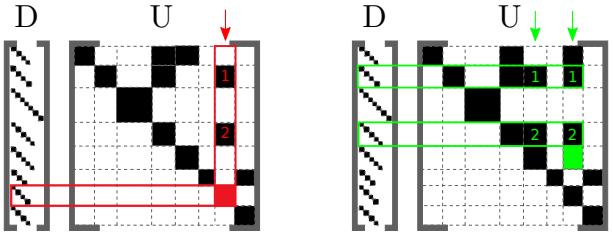
Our blocks are saved in a hybrid format: either sparse, with indices and values, or dense. We choose the format with smaller memory footprint given the maximum number of elements computed earlier. Blocks on the block-diagonal of U (dubbed "diagonal blocks") are, however, always set to dense - as well in the bottom 1% of blocks, where many updates occur. A compensation factor of 0.5 accounts for the additional indirect addressing when working with sparse blocks.

Finally, similar to Naumov [29], we compute level sets on the coarse graph which dictate the processing order for factorization and sort the blocks in memory after the number of their level set. We allocate an array of 32×32-sized, dense blocks to keep dense blocks processed in the curent level in memory ("in-flight" blocks).

B. Factorization

The high-level structure of our factorization algorithm is given in Alg. 1. In order to avoid repeated load and store operations, some functions are merged into one kernel. This section describes the resulting kernels.

A notable change to the derivation in (2) is the handling of updates: While (5) amounts to *pushing* the results of the outer products $L_{31}D_{11}L_{31}^\top, L_{32}D_{22}L_{32}^\top$ to the Schur complement, we use a *pull* principle: A block pulls in, i.e. applies, all updates once it is in the current level. This strategy avoids multiple applications of the dropping rules.



(a) Symmetric updates. (b) Nonsymmetric updates.

Figure 2: Block updates. (a) For diagonal block (i, i), all blocks (k, i) (column i marked with red arrow) for $k < i$ are required. Similarly, the dense factorization step modifies D's ith block. (b) For off-diagonal block (i, j), all blocks (k, i) and (k, j) (columns i, j marked with green arrows) for $k < i$ as well as the kth block of D are required.

Following the architectural changes in GPUs, we process blocks in shared memory whenever possible and use collective operations to avoid coarse-grained synchronization inside of blocks. To that end, we implemented most BLAS-like operations in a cooperative manner, where tasks are mapped onto groups that correspond to (parts of) a warp. Whenever a matrix product needs to be computed, we transpose one of the operands to avoid bank conflicts during the multiplication.

1) Diagonal blocks: Before factorizing a (dense) diagonal block U_{ii}, we pull in the updates $U_{ki}^\top D_{kk} U_{ki}$ for $k < i$, which are all found in the column i (see Fig. 2a), using the coarse CSC representation on the device. Each diagonal block is processed by one CUDA block. All updates, are also expanded into shared memory. The outer product is computed row-wise, each row assigned to one warp.

All blocks have at most size 32×32, enabling us to perform the subsequent, dense $U^\top DU$ factorization step on the result of the updates using a single warp. The dense factorization follows the same steps outlined so far, except for a *push* strategy for updates. The application of small, independent and dense block-factorizations allows efficient supernodal pivoting; we implemented three pivoting strategies using 1×1 and 2×2 pivots:

1) None/static pivoting with 1×1 or 2×2 pivots from MC64
2) Bunch-Kaufmann (partial) pivoting (scans 2 cols / step)
3) Rook (full) pivoting (possibly scans all columns)

With one warp computing the factorization, all comparisons can be handled efficiently with warp shuffle operations.

The resulting triangular matrix with a pivoting permutation, is stored as dense, 32×32 block in the in-flight storage (as B_{ii}). This kernel also contains a step that sums up the square of entries per row in the block for later use during dropping.

2) Off-diagonal blocks: Similar to the diagonal blocks, off-diagonal blocks U_{ij}, for $j > i$ first pull in their updates. In order to do that, the block columns i and j have to be traversed. The intersection of the columns' row indices yield the necessary updates (see Fig. 2b). Per common row, the pair of update blocks is loaded into shared memory and the outer product is computed (again in one CUDA block, as above). Once all updates are applied, we first permute rows according to the block rows' permutation vector from pivoting. Next, each column of the block is solved with U_{ii}, D_{ii}. The dense triangular solve is performed with one warp per column. Finally, the block is saved to in-flight storage (as B_{ij}). Using atomics, finalized blocks contribute the row-sums of the squared entries for the following threshold dropping.

Algorithm 2 Conservative protocol for distributing leftover memory to an oversubscribed in-flight block $B_{i,j}$

```
required = nnz(B_ij) − capacity(U_ij)
wasfree = atomic_subtract(leftover, required)
granted = max(min(required, wasfree), 0)
giveback = required − granted
if granted > 0 then
    atomic_add(leftover, giveback)
    return granted {extra units of memory for saving B_ij to block U_i,j}
end if
```

3) Dropping: If pivoting was used, an additional step becomes necessary before dropping: All superdiagonal blocks in the current block column need to be column-permuted according to the diagonal blocks' permutation vectors. Since permuting entries does not add additional elements, the operations may be done in-place in global memory (for sparse blocks, only the indices need to be changed; for dense blocks, warps reorder the columns).

We implement a dual pivoting strategy, following Saad [32]: First, we drop all elements $u_{kl} \leq \tau ||u_{k\cdot}||_2$ by setting them to 0. The number of remaining nonzeros is then counted and, if it exceeds max_nnz(U_{ij}), only the elements with largest magnitude are kept. For the latter, every thread in the block stores 4 matrix elements. A collective block-wide radix-sort yields the elements in the order of their magnitude. The first max_nnz(U_{ij}) can then directly be stored in the block storage in the case of a sparse block. For dense blocks, naturally, no dropping is performed.

By setting a maximum number of entries to a fixed number per block, we enable each block in the current level to be processed separately. However, this restricts the layout of the preconditioner and could worsen its quality. As a relief, we propose a protocol that roughly emulates an area-based heuristic by Li and Shao [24]. Memory that was reserved for blocks in previous levels but not used is redistributed. As a consequence, the fill-ratio r_f holds for the whole matrix, not for each block.

An outline of the protocol, applied to a single dense block B_{ij}, is given in Alg. 2. During the whole execution, we keep an atomic counter (leftover) for the space left. In our protocol, each thread first determines how much additional memory is needed and subtracts that amount atomically from the leftover counter, returning the old value. If the latter is positive, we compute the amount of extra space provided. Since we subtracted the complete amount of memory requested, we then add the difference to the amount of memory granted back ("giveback") to the leftover counter, atomically. This protocol, however, is not exact, but conservative: If a block asked for more memory than was left over, the leftover counter becomes

negative. Blocks that return their giveback after that increase the leftover counter, but as long as it is negative, no further memory can be distributed to other blocks. Only when the first block returns its giveback, the leftover counter is positive. This is due to the separation of the two atomic updates to leftover; the protocol thus may underestimate the amount of memory to distribute, but it never overallocates it.

In the setup phase, we order the blocks in memory by levels. Using an additional atomic counter for the total memory used, we can save the extended blocks in-place.

C. Blocked triangular solves and SQMR

We solve (1) using iterative methods, with the block-oriented $iLDL^\top$ preconditioner. In each step of the iterative methods, solve operations with L, D and $L^\top$ are executed. Since D is block-diagonal with 1×1 and 2×2 blocks, the operation is trivially parallelizable. For L and $L^\top$, we can (as in the factorization) exploit the discovered level sets, see [28], or use fixed-point methods, see [3], [9]. We reuse the level sets from the factorization – in the same order for L and inverted for $L^\top$. By replacing the scalars in a level-scheduled triangular solve algorithm, we arrive at a block-based algorithm. In our implementation, each CUDA block handles one matrix block; in the first kernel, each off-diagonal block is multiplied with the corresponding solution components by a single warp; lastly, a solve operation in a second kernel results in another part of the solution vector.

Since both A and $LDL^\top$ are indefinite, we use SQMR [15] as the method of choice. To allow for an indefinite preconditioner with split preconditioning, we use the D-symmetric version from [15].

V. Evaluation

We use a test set with 17 symmetric indefinite matrices also used in [17], [20]. All matrices are taken from [13] and preprocessed using symmetrized MC64 [18]. Also, we added KKT matrices resulting from an interior point method for linear programs. We created the test set by downloading mps files from MIPLIB [23], geometrically scaling the LPs, standardizing the LP format and then adding a diagonal in the (1,1)-block of the KKT matrix with random positive elements from $(0, 1)$ that span a logarithmic range of 10^{-6}. All right-hand sides in the experiments were created by multiplying the input matrices with an all-1 vector. We computed the blockings and permutations for both strategies offline. For fairness, the preprocessing time for all evaluated packages was excluded from the comparison.

We compare our preconditioner with SYM-iLDL [17] (Github commit d4b862b), using the CLI "ldl-driver" and varying levels of fill between 4.0 and 8.0, following the paper. When handling difficult, indefinite linear systems, pivoting is often required for good accuracy despite its negative effect on parallelism (pivoting and parallelism are often two conflicting design goals). Thus, it is valid to compare our approach to this sequential package. Note that while the solve phase using SQMR could be parallelized, we often received large numerical errors with cuSPARSE for the resulting triangular factors. We also compare to a fixed-point method, ParILU with levels-of-fill between 1 and 3 using the authors' implementation in their MAGMA library [37]. All solvers use the same input permutation for each matrix.

All experiments are performed on a system with an Intel i7 3930K processor, 64 GB RAM and Ubuntu 16.04. We benchmark on a NVIDIA K40 and a TITAN V GPU with CUDA 9.2 (driver 396.44). With the blocking strategy, reordering, level-of-fill, fill factor, dropping threshold, pivot strategy and precision (given in that order in Tab. I) we handle a huge parameter space. Since the influence of these parameters can be highly nonlinear (see below), we manually explored the space and only present the best result in terms of runtime that we found. These results are thus upper bounds on the best possible runtime. For a fair comparison, we did the same with our competitors: SYM-iLDL's fill factor was varied between 4.0 and 8.0, ParILU's level-of-fill between 1 and 3. We split all reported times in factorization and solve phase (and, in our case, an additional setup for the triangular block-solve).

Tab. I shows overall runtime. The experiments with MAGMA confirm that indefinite matrices are notoriously hard to precondition: It fails to converge after 1000 iterations with its ILU in all cases but one; it is thus mostly excluded from Table I. A second observation regarding convergence to the desired relative residual of 10^{-6} is shared between our approach and SYM-iLDL: a large fill factor of between 4.0 and 8.0 is necessary as well as a smaller dropping threshold than often used by default (10^{-4}). SYM-iLDL, being a scalar, sequential implementation, works best when the factors are relatively sparse (Si10H16) or our code suffers from tiny blocks (boyd1). The choice of blocking and permutation method is critical for our code's performance. Setup and overhead are comparable between the two implementations, as shown by the similar performance on smaller matrices where the overhead dominates factorization time.

Tab. II reports the block sizes and the amount of nonzero elements inside these blocks (dubbed "fill") for input and factorized matrices. Dense blocks are counted with fill = 1, which explains the differences of fill increase compared to the selected fill factor. These statistics correlate well with the matrices where we achieve the highest speed-ups compared to SYM-iLDL: most fill-in elements fall inside of our blocks, allowing effective pivoting and capturing the preconditioner better. For the large nlpkkt* matrices, this leads to an excellent preconditioner quality and runtime. For the matrix bley_xl1, no working preconditioner could be generated by any tested package. In such hard cases, a direct methods is the method of choice. Even in easier cases, double precision was necessary for a successful preconditioner generation and application. Notable exceptions are co-100 and nlpkkt80, where single precision and a moderate fill and dropping threshold were sufficient for the desired accuracy.

We also include results for both the K40 as well as the TITAN V. Our approach was specifically designed with the

Matrix	Package	Runtime (s)
qpband [17] $m = 20,000$ $nnz = 45,000$	SYM-ILDL (8.0)	0.01 + 0.04
	ours (BR, AMD, 0, 4.0, 10^{-4}, BK, D)	3.13 + 0.05 + 4.49, 1 it., K40
		0.01 + 0.00 + 0.01, 1 it., Titan V
mario001 [17] $m = 38,434$ $nnz = 204,912$	SYM-ILDL (8.0)	**0.14 + 0.01**
	ours (BR, AMD, 3, 8.0, 10^{-8}, Rook, D)	0.26 + 0.00 + 0.65, 30 it., K40
		0.12 + 0.00 + 0.29, 30 it., Titan V
bab5 [23] $m = 30,199$ $nnz = 343,545$	SYM-ILDL (8.0)	0.31 + 0.81
	ours (RB, AMD, 1, 8.0, 10^{-8}, Rook, D)	0.17 + 0.00 + 0.18, 8 it., K40
		0.10 + 0.00 + 0.05, 8 it., Titan V
c-72 [17] $m = 84,064$ $nnz = 707,546$	SYM-ILDL (4.0)	0.96 + 2.58
	ours (BR, AMD, 1, 8.0, 10^{-8}, BK, D)	2.25 + 0.01 + 1.87, 11 it., K40
		0.95 + 0.01 + 0.52, 11 it., Titan V
Si10H16 [20] $m = 17,077$ $nnz = 875,923$	MAGMA PBiCGStab + ILU(3)	3.16 + 0.54
	SYM-ILDL (8.0)	**0.32 + 0.37**
	ours (RB, AMD, 3, 4.0, 10^{-4}, Rook, D)	2.62 + 0.01 + 0.39, 8 it., K40
		2.62 + 0.01 + 0.37, 8 it., Titan V
boyd1 [17] $m = 93,279$ $nnz = 1,211,231$	SYM-ILDL (8.0)	**0.06 + 0.06**
	ours (RB, AMD, 0, 4.0, 10^{-4}, BK, D)	1.62 + 0.02 + 2.98, 1 it., K40
		0.98 + 0.01 + 0.1, 1 it., Titan V
Lin [20] $m = 256,000$ $nnz = 1,766,400$	SYM-ILDL (8.0)	1.7 + 8.47
	ours (RB, AMD, 1, 8.0, 10^{-8}, BK, D)	4.73 + 0.04 + 18.1, 168 it., K40
		1.4 + 0.04 + 5.96, 168 it., Titan V
map06 [23] $m = 703,690$ $nnz = 1,895,362$	SYM-ILDL (4.0, 8.0)	†
	ours (BR, AMD, 0, 4.0, 10^{-4}, BK, D)	3.44 + 0.03 + 2.5, 16 it., K40
		1.35 + 0.02 + 0.33, 16 it., Titan V
bley_xl1 [23] $m = 354,783$ $nnz = 2,264,609$	SYM-ILDL (4.0, 8.0)	†
	ours (all)	†
		†
c-big [17] $m = 345,241$ $nnz = 2,340,859$	SYM-ILDL (8.0)	†
	ours (BR, AMD, 1, 8.0, 10^{-8}, BK, D)	8.34 + 0.02 + 17.6, 56 it. K40
		3.53 + 0.02 + 3.38, 56 it., Titan V
co-100 [23] $m = 52,618$ $nnz = 4,046,093$	SYM-ILDL (8.0)	†
	ours (BR, AMD, 2, 4.0, 10^{-8}, Rook, S)	8.61 + 0.02 + 0.47, 4 it., K40
		4.61 + 0.02 + 0.11, 4 it., Titan V
bab3 [23] $m = 435,381$ $nnz = 7,053,012$	SYM-ILDL (8.0)	†
	ours (RB, AMD, 1, 8, 10^{-8}, Rook, D)	3.55 + 0.01 + 4.70, 36 it., K40
		1.74 + 0.01 + 1.26, 36 it., Titan V
G3_circuit [20] $m = 1,585,478$ $nnz = 7,660,826$	SYM-ILDL (8.0)	4.35 + 11.93
	ours (RB, AMD, 1, 8.0, 10^{-8}, BK, D)	7.93 + 0.12 + 21.6, 72 it., K40
		2.93 + 0.12 + 5.05, 72 it., Titan V
BenElechi1 [20] $m = 245,874$ $nnz = 13,150,496$	SYM-ILDL (8.0)	8.09 + 27.3
	ours (BR, AMD, 4, 8.0, 10^{-8}, BK, D)	2.96 + 0.01 + 7.02, 88 it., K40
		1.82 + 0.01 + 1.57, 88 it., Titan V
af_shell7 [20] $m = 504,855$ $nnz = 17,579,155$	SYM-ILDL (4.0, 8.0)	†
	ours (BR, AMD, 4, 8.0, 10^{-8}, Rook, D)	5.34 + 0.02 + 18.2, 64 it., K40
		3.15 + 0.02 + 10.1, 64 it., Titan V
nlpkkt80 [17] $m = 1,062,400$ $nnz = 28,192,672$	SYM-ILDL (4.0)	56.16 + 118.55
	ours (BR, AMD, 0, 8.0, 10^{-4}, BK, S)	1.87 + 0.04 + 7.58, 196 it., K40
		2.65 + 0.04 + 3.86, 196 it., Titan V
nlpkkt120 [17] $m = 3,542,400$ $nnz = 95,117,792$	SYM-ILDL (4.0, 8.0)	†
	ours (BR, AMD, 0, 8.0, 10^{-8}, Rook, D)	23.89 + 0.21 + 67.4, 96 it., K40
		11.04 + 0.19 + 19.4, 96 it., Titan V

Table I: The runtime of the software packages. It is reported as $x + [y] + z$, where x denotes preconditioner computation, y optional preconditioner analysis and z time taken by the iterative method. We also report number of iterations (it.) taken to convergence, while † denotes the lack of convergence.

latest advances in GPU architecture in mind – and that is visible in the results. Besides an 2x improvement across-the-board for the factorization phase, the solve phase benefited more: Our implementation relies on double precision atomics, which are implemented in hardware on the TITAN V but are emulated through integer atomics on the K40.

The effects of parameters on our preconditioners are often nonlinear and counter-intuitive; the characteristics vary between matrices. We try to capture some of that by presenting two extremes per parameter in Fig. 3. First, the level of fill (Fig. 3a): some matrices, here co-100, behave as expected with better preconditioners for higher levels that reduce the number of iterations but increase the number of level sets. For other matrices, such as Lin, a higher number of levels after a threshold results in an unstable preconditioner. Here, more aggressive dropping might help. Alone, the dropping threshold (Fig. 3b) often has small influence on the results, especially if the matrix has mostly dense blocks, where dropping is not applied. The fill factor often has a much larger effect (Fig. 3c). Naturally, the average block fill increases in conjunction with that parameter – but the number of SQMR iterations does not necessarily decrease accordingly; often, tiny elements are kept in denser blocks that do not effect the solution much. Lastly, the effect of supernodal pivoting (Fig. 3d) is split: For matrices such as map06 where MC64 was able to determine good pivots and the off-block-diagonal elements are small, there is not much difference to static pivoting. For other matrices, see e.g. bab3, pivoting is necessary for a working preconditioner. We observed multiple cases where pivoting allowed a preconditioner computed in single-precision to be effective. Combining a single-precision, pivoted preconditioner with a double-precision solve and SQMR could be advantageous.

Matrix	Mean block size	Mean fill Input	Mean fill Output
qpband	16	0.375	1
mario001	712.68	0.05	0.25
bab5	400.05	0.04	0.49
c-72	698.14	0.1	0.28
Si10H16	744.53	0.01	0.08
boyd1	15.02	0.58	0.99
Lin	708.24	0.01	0.19
map06	146.83	0.16	0.66
bley_xl1	136.87	0.12	0.42
c-big	787.99	0.07	0.25
co-100	299.99	0.09	0.47
bab3	649.77	0.04	0.41
G3_circuit	265.99	0.03	0.61
BenElechi1	590.9	0.08	0.99
af_shell7	467.17	0.1	0.98
nlpkkt80	989.39	0.03	0.31
nlpkkt120	1001.67	0.03	0.32

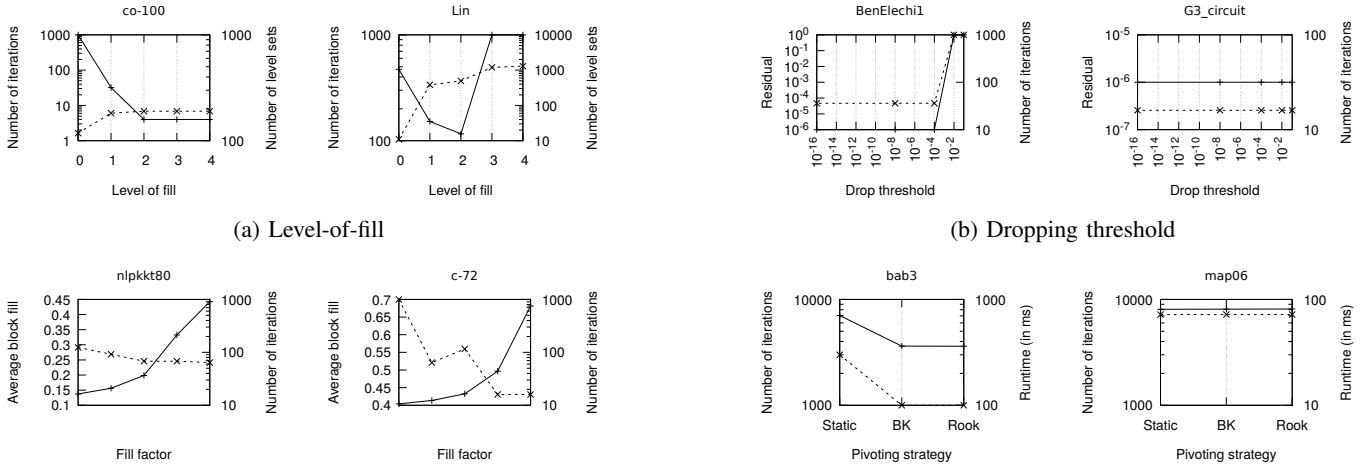
Table II: Fill statistics for matrix blocks.

VI. Conclusion and future work

Symmetric indefinite systems can be hard to precondition. Our approach combines techniques from full and incomplete factorizations and leverages recent changes in GPU architecture, yielding short runtimes and effective preconditioners for large, sparse matrices. The approach, rare for GPU-based implementations, allows restricted fill-in and pivoting. We plan to open-source our implementation to the community. In future work, we plan to investigate selection heuristics for our blocking strategy as well as parameter guidelines, possibly using machine learning, and to apply the same principles to other incomplete factorizations, such as iLU and iQR.

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 to NVIDIA for providing us with a Tesla K40 as part of their Hardware Grant Program.



(a) Level-of-fill

(b) Dropping threshold

(c) Fill factor

(d) Pivoting strategy.

Figure 3: Influence of factorization parameters. The solid line corresponds to the left axis, the dashed line to the right axis.

REFERENCES

[1] J. I. Aliaga, M. Bollhöfer, A. F. Martı, E. S. Quintana-Ortı, et al. Exploiting thread-level parallelism in the iterative solution of sparse linear systems. *Parallel Computing*, 37(3):183–202, 2011.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM TOMS*, 30(3):381–388, 2004.

[3] H. Anzt, E. Chow, and J. Dongarra. Iterative sparse triangular solves for preconditioning. In *Eur. Conf. on Parallel Proc.*, pages 650–661, 2015.

[4] H. Anzt, E. Chow, and J. Dongarra. ParILUT – a new parallel threshold ilu factorization. *SIAM J. Sci. Comput.*, 40(4):C503–C519, 2018.

[5] M. Bollhöfer. High performance block incomplete LU factorization. 2017.

[6] M. Bollhöfer and Y. Saad. Multilevel preconditioners constructed from inverse-based ILUs. *SIAM J. Sci. Comput.*, 27(5):1627–1650, 2006.

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

[8] J. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. 31:163–163, 01 1977.

[9] E. Chow and J. Scott. On the use of iterative methods and blocking for solving sparse triangular systems in incomplete factorization preconditioning. *Rutherford Appleton Lab., RAL-P-2016-006*, 2016.

[10] E. Chow and A. Patel. Fine-grained parallel incomplete LU factorization. *SIAM J. Sci. Comput.*, 37(2):C169–C193, 2015.

[11] E. Chow and Y. Saad. Experimental study of ILU preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics*, 86(2):387–414, 1997.

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

[13] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[14] 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*, 22(4):973–996, 2001.

[15] R. Freund and N. Nachtigal. Software for simplified Lanczos and QMR algorithms. *Appl. Numerical Mathematics*, 19(3):319–342, 1995.

[16] J. R. Gilbert and R. Schreiber. Highly parallel sparse Cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 13(5):1151–1172, 1992.

[17] C. Greif, S. He, and P. Liu. SYM-ILDL: Incomplete LDL^T factorization of symmetric indefinite and skew-symmetric matrices. *ACM TOMS*, 44(1):1, 2017.

[18] M. Hagemann and O. Schenk. Weighted matchings for preconditioning symmetric indefinite linear systems. *SIAM J. Sci. Comput.*, 28(2):403–420, 2006.

[19] J. Hogg, J. Reid, and J. Scott. Design of a multicore sparse Cholesky factorization using DAGs. *SIAM J. Sci. Comput.*, 32(6):3627–3649, 2010.

[20] J. D. Hogg, E. E. Ovtchinnikov, and J. A. Scott. A sparse symmetric indefinite direct solver for GPU architectures. *ACM TOMS*, 42(1):1–1, 2016.

[21] D. Hysom and A. Pothen. Level-based incomplete LU factorization: Graph model and algorithms. *Preprint UCRL-JC-150789, US Department of Energy, Nov*, 2002.

[22] G. Karypis and V. Kumar. METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[23] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, et al. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103, 2011.

[24] X. S. Li and M. Shao. A supernodal approach to incomplete LU factorization with partial pivoting. *ACM TOMS*, 37(4):43:1–43:20, February 2011.

[25] J. W. Liu. A compact row storage scheme for Cholesky factors using elimination trees. *ACM TOMS*, 12(2):127–148, 1986.

[26] D. Lukarski, H. Anzt, S. Tomov, and J. Dongarra. Multi-elimination ILU preconditioners on GPUs. In *Intl. Parallel & Dist. Proc. Symp.*, 2014.

[27] 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.

[28] M. Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. *NVIDIA Tech. Rep. NVR-2011*, 2011.

[29] M. Naumov. Parallel incomplete-LU and Cholesky factorization in the preconditioned iterative methods on the GPU. *NVIDA Tech. Rep.*, 2012.

[30] C. Nvidia. Cublas library. *NVIDIA, Santa Clara, CA*, 15(27):31, 2008.

[31] L. Polok, V. Ila, and P. Smrz. Cache efficient implementation for block matrix operations. In *Proceedings of the high performance computing symposium*, page 4, 2013.

[32] Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical linear algebra with applications*, 1(4):387–402, 1994.

[33] Y. Saad. Finding exact and approximate block structures for ilu preconditioning. *SIAM J. Sci. Comput.*, 24(4):1107–1123, 2003.

[34] Y. Saad. *Iterative methods for sparse linear systems*, volume 82. 2003.

[35] Y. Saad and J. Zhang. BILUM: block versions of multielimination and multilevel ILU preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 20(6):2103–2121, 1999.

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

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

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: "A BLOCK-ORIENTED, PARALLEL AND COLLECTIVE APPROACH TO SPARSE INDEFINITE PRECONDITIONING ON GPUS"

A. Abstract

This artifact includes software, data and instructions required to replicate our results from Sections III and V as well as perform further experiments on different data sets. A dockerized, scripted artifact execution process allows replication of our results and application to other datasets.

B. Description

1) Artifact meta information:

- **Algorithm:** block-$iLDL^\top$ using a predefined permutation and blocking
- **Program:** C++/CUDA programs; `culip-blocking-stats` for Section III and `culip-block-ildlt` for Section V
- **Compilation:** CUDA compiler `nvcc` as part of CUDA 9.2, used with gcc-7.3.0 as host compiler
- **Data set:** Matrices publicly available in the UFL collection as well as matrices constructed from IP problems in MIPLIB 2010
- **Hardware:** see hardware setup in Section V
- **Output:** Runtimes for factorization, analysis and solve phase resp. level-of-fill statistics
- **Experiment workflow:** Download artifact, execute build process, execute program per matrix, observe printed results
- **Experiment customization:** Yes, software accepts arbitrary symmetric indefinite matrices and blockings with block size ≤ 32
- **Publicly available?:** Yes

2) How software and data can be obtained: The artifact – including the automatic build process – can be obtained from our Github repository at https://github.com/dthuerck/ia3-18-artifact). This will automatically download the source code from https://github.com/dthuerck/culip. All contents of this artifact are licensed under a liberal 3BSD license.

3) Hardware dependencies: Our implementation requires a CUDA-compatible GPU with a compute capability (CC) of at least 3.5. To use fast atomics for double precision computations, we require at least compute capability 6.0.

4) Software dependencies: The artifact requires installation of the following packages: python3, nvidia-docker, docker-ce, CUDA 9.2 and CMake ≥ 3.2. The docker container uses an Ubuntu 18.04 base package. We tested the artifact exclusively under Ubuntu 18.04 with CUDA 9.2 installed.

5) Datasets: All matrices from UFL used in [17] and [20] have been preprocessed with MC64. Since MC64 requires a commercial license for some users, we also uploaded the processed matrices. The MIPLIB matrices were generated by removing symbolically dependent rows and 4 rounds of geometric scaling. The diagonals of their KKT form were filled with randomly generated numbers in $(0, 1)$ that logarithmically span 6 orders of magnitude to simulate a late IPM iteration. These matrices are downloaded as part of the build process.

C. Installation

After cd'ing into the cloned repository, execute the following command:

```
$ sudo python3 artifact.py setup
```

Note that using `sudo` is required for standard installations of docker and can be avoided by modifying the docker installation accordingly. The command sets up a docker container, clones the culip repository, builds the code and downloads our benchmark matrices.

D. Experiment workflow

We describe the necessary actions to replicate the experiments in Sections III and V as well as the application of `culip-block-ildlt` to other matrices.

1) Section III: This section reports block counts and level set statistics for different levels of (block-)fills given a reordering and blocking for a matrix. The following command computes statistics for block-fill levels 0 to 5:

```
$ sudo python3 artifact.py stats --matrix
    ↪ matrix_name --max_level 5
```

where `matrix_name` is taken from Tab. I.

2) Section V: This section reports timings for the solution of linear systems with block-$iLDL^\top$. The command line

```
$ sudo python3 artifact.py replicate --matrix
    ↪ matrix_name
```

where `matrix_name` is taken from Tab. I, solves a linear system with the selected matrix, using the parameters given in the same table.

3) Further experiments: To run `culip-block-ildlt` on other matrices, create a folder with the matrix' name under `shared/` and include the following files (for descriptions see `block-ildlt.cc`):

- the matrix as `matrix.mtx`
- the permutation vector as `perm.mtx`
- the vector of block starts `blks.mtx`
- the vector of pivot starts as `pivs.mtx`

and execute

```
$ sudo python3 artifact.py solve --matrix
    ↪ matrix_name
```

Optional arguments are: `--precision` (0 for Single, 1 for Double), `--pivot_method` (0 for static, 1 for BK, 2 for Rook) and `--fill_level`, `--fill_factor`, `--threshold`.

E. Evaluation and expected result

1) Section III: Given the input permutation and blocking, block-fill in for the levels 0 to 5 are generated. The application prints out the number of level sets as well as the number of blocks in total plus their sizes. Note that matrices can become so dense that they do not fit into the machine's memory.

2) Section V: `culip-block-ildlt` first computes a preconditioner for the given matrix A and then generates an approximate solution x to $Ax = A\mathbf{1}$. After convergence, the relative residual $\|A\mathbf{1} - Ax\|/\|A\mathbf{1}\|$ is reported. Similarly, the individual runtime for the preconditioner computation, (optional) analysis and the iterative solve phase are reported in milliseconds.

APPENDIX B
ARTIFACT EVALUATION APPENDIX: "A BLOCK-ORIENTED, PARALLEL AND COLLECTIVE APPROACH TO SPARSE INDEFINITE PRECONDITIONING ON GPUS"

A. Abstract

The evaluation of our paper "A block-oriented, parallel and collective approach to sparse indefinite preconditioning on GPUs" contains runtime results for culip-block-ildlt, SYM_ILDL and MAGMA on a number of symmetric indefinite matrices. This evaluation appendix describes the process of replicating these results and adds details on their evaluation.

B. Replication of the papers' results

1) culip-block-ildlt: We recommend to use our docker-based artifact script as in the artifact description, Sec. A-D, above. For any matrix `M` in the test set (see first column of Tab. I), a single call to

```
$ sudo python3 artifact.py replicate --matrix M
```

is sufficient to run the preconditioner and SQMR solver as described with the parameters used in our evaluation. To replicate all results in the paper, the command has to be executed for each of the included matrices.

After any changes to the Github repository or the uploaded data, a

```
$ sudo python3 artifact.py clean
```

can be used to remove the Docker container, enabling a rebuild from scratch.

2) SYM-ILDL: The authors of SYM-ILDL have released their code under the permissive MIT license. Their latest source can be found on Github (https://github.com/inutard/matrix-factor). For our evaluation, we used the latest commit #d4b862b from November 9, 2016. The package was then executed by

```
$ ./ldl_driver -filename=M.mtx
```

with the default options (according to the authors' paper) - only the fill factor (`-fill`) was set to 4.0 or 8.0. Using this call, a preconditioner is computed, a right hand side is generated and the resulting system is solved with the included implementation of SQMR. Preprocessing, factorization and solving time is reported. As mentioned in the paper, we excluded the preprocessing time for the purpose of our evaluation.

3) MAGMA: We obtained MAGMA 2.4.0 from https://icl.utk.edu/magma/software/index.html and built it on the same PC as used in the evaluation. Similar to the other two packages, we used a C++ program to read the input matrices and generate the real solution vector as all-1s.

By calling `magma_dsolverinfo_init` with the solver selection `Magma_BICGSTABMERGE2` and preconditioner selection `Magma_PARILU`, we initialize the library. For the preconditioner, we use 5 sweeps for computation and tried fill levels $1 - 3$.

C. Timing

For MAGMA and culip-block-ildlt, we used C++11's `chrono` (i.e. `std::chrono::system_clock::now()`) API to measure the passed wall-clock time. SYM-ILDL uses the POSIX `clock()` API to measure times.

D. Results analysis discussion

The runtime results for our Desktop PC (Intel i7 3930K, 64 GB RAM, Ubuntu 16.04, see Tab. II) were collected before the creation of our Docker image. Due to the way Docker works, we observe a small increase in processing times, especially for CPU-heavy parts.

Due to the use of CUDA atomics in culip-block-ildlt, runtimes between multiple runs of the application can slightly vary. In our experiments, the amount of variation was negligible – thus, we reported the worst (i.e. highest runtime) of 10 runs. Further sources for software-independent variations of the observed runtime are GPU overclocking, thermal boosting and PCI-E performance. Since large parts of the preprocessing is executed on the CPU and parallelized using NVIDIA's Thrust library, CPU performance can influence the runtime, especially for smaller matrices.

The included SQMR solver only reports its quasi-residual every fourth iteration in order to better overlap memory transfers with compute. To determine a precise timing, we tracked the real residual to determine the optimal number of iterations for the iterative method. For badly-conditioned matrices, there is often a large gap between quasi-residual and real residual.

E. Summary

In the hopes that our code can be of use to the community as a trustworthy piece of software, we release all code and data necessary for replicating the evaluation to the public. The provided Dockerfile and Python3 script allows an easy setup and replication; the evaluating party only needs a correctly set up (nvidia-) docker installation.