2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

ParILUT - A Parallel Threshold ILU for GPUs

Hartwig Anzt*†, Tobias Ribizel*, Goran Flegar‡, Edmond Chow§, Jack Dongarra†¶‖
*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany
†Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA
‡Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I Castellón, Spain
§School of Computational Science and Engineering, Georgia Institute of Technology, USA
¶University of Manchester, Manchester, UK
‖Oak Ridge National Lab (ORNL), Oak Ridge, USA
hartwig.anzt@kit.edu, tobias.ribizel@student.kit.edu, flegar@uji.es, echow@cc.gatech.edu, dongarra@icl.utk.edu

***Abstract*—In this paper, we present the first algorithm for computing threshold ILU factorizations on GPU architectures. The proposed ParILUT-GPU algorithm is based on interleaving parallel fixed-point iterations that approximate the incomplete factors for an existing nonzero pattern with a strategy that dynamically adapts the nonzero pattern to the problem characteristics. This requires the efficient selection of thresholds that separate the values to be dropped from the incomplete factors, and we design a novel selection algorithm tailored towards GPUs. All components of the ParILUT-GPU algorithm make heavy use of the features available in the latest NVIDIA GPU generations, and outperform existing multithreaded CPU implementations.**

***Index Terms*—ParILUT, parallel threshold ILU, incomplete factorization preconditioners, parallel selection, GPU**

I. Introduction

Preconditioners based on incomplete LU (ILU) factorizations [1] are popular components in solving large, sparse linear systems via iterative methods. The underlying principle is to approximate the LU decomposition of the system matrix with triangular factors that retain a high level of sparsity. To that end, the Gaussian elimination process is modified such that fill-in is reduced. One approach is to predefine a sparsity pattern on which nonzero elements are allowed (level-based ILU [1]). Alternatively, only an upper limit on the number of nonzero elements in the pattern can be imposed, and the pattern itself is then chosen during the factorization process to capture the elements with the largest magnitude (threshold-based ILU [1]). The quality of an incomplete factorization in terms of how well it works as a preconditioner depends on the problem (the matrix and its ordering), and the factorization's sparsity pattern. As threshold-based ILU factorizations not only take the structural properties of the system matrix into account but also the numerical values, they can reflect the problem's characteristics more effectively. As a result, for the same number of nonzero elements, threshold-based ILU preconditioners can be superior to level-based ILU preconditioners in terms of improving the convergence of the iterative solver. At the same time, thresholding techniques make the parallelization of the factorization process more challenging. In particular, since the sparsity pattern is not known beforehand, it is impossible to employ parallelization strategies such as level scheduling or multi-color ordering [2].

One strategy to parallelize threshold ILU factorizations is to use graph partitioning or domain decomposition [2], [3]. However, as a high number of subdomains usually degrades the preconditioner quality, domain decomposition can only provide coarse-grained parallelism. Furthermore, the factorization of the Schur complement corresponding to subdomain interfaces cannot be efficiently parallelized for dynamic thresholding.

More recently, a novel strategy for computing threshold-based ILU factorizations in a highly-parallel fashion was presented [4]. Its underlying idea is to interleave parallel fixed-point iterations that approximate the incomplete factors for an existing nonzero pattern with a strategy that dynamically adapts the nonzero pattern to the problem characteristics. The authors demonstrate that the ParILUT algorithm can efficiently exploit the compute power of multicore architectures featuring thread-independent execution paths and sophisticated cache hierarchies. In this paper, we (1) develop the first threshold-based ILU factorization for graphics processing units (GPUs); (2) design a novel selection algorithm that optimally utilizes the parallel processing power available on GPUs by combining techniques from the (super scalar) sample sort algorithm with the recursion tree pruning employed in quickselect; (3) use a performance assessment on a range of GPUs from different generations to demonstrate runtime advantages of the developed selection algorithm over state-of-the-art strategies; (4) modify the developed selection algorithm to relax accuracy in favor of reduced execution time; (5) show that the developed ParILUT-GPU algorithm outperforms its multicore counterparts on a range of architectures.

Section II provides background about threshold-based factorizations in general and the ParILUT algorithm in particular. Section III exclusively focuses on parallel selection on GPUs, a functionality critical for generating thresholds that separate the smaller values that can be dropped from the incomplete factors. Section IV provides details about the design and the implementation of the ParILUT-GPU algorithm architectures. Section V comprises a comprehensive experimental analysis of the developed selection and the complete ParILUT-GPU

This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers DE-SC0016513, DE-SC-0016564, and DE-SC-0010042. H. Anzt was supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association under grant VH-NG-1241.

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00033

algorithm. Aside from a runtime comparison against state-of-the-art algorithms, we also assess the performance portability across different GPU generations. Section VI concludes with a summary of the findings and lists related topics we plan to address in the near future.

II. Incomplete factorization preconditioners and the ParILUT algorithm

An incomplete factorization approximates the LU decomposition of a nonsingular sparse matrix A with a lower triangular matrix L and an upper triangular matrix U, i.e., $A \approx LU$, where much of the sparsity of the original system matrix is preserved.

The traditional way of generating incomplete factorizations is to modify the Gaussian elimination algorithm by truncating the fill-in that typically occurs during the factorization process. One possibility is to predefine a sparsity pattern S on which nonzero entries are allowed and to neglect any fill-in arising in the Gaussian elimination process outside this pattern (level-ILU). Alternatively, the sparsity pattern S can be determined dynamically during the factorization process (threshold-ILU [1]). In the latter case, the decision of whether an element is included in the incomplete factor is usually based on the element's significance, i.e., its magnitude in comparison to the other elements in the same row/column.

Independently of whether the sparsity pattern is predefined or generated dynamically, the Gaussian elimination process itself is inherently sequential. Natural parallelism only exists if it is possible to find multiple rows that only depend on rows that have already been eliminated. To increase the parallelism, strategies such as multicolor ordering or domain decomposition can be employed [2], [3], [5]–[9]. However, all these approaches often reduce the quality of the incomplete factorization [6], [8]. Furthermore, the scalability of the Gaussian elimination process enhanced with these strategies is still limited as they generally fail to leverage the fine-grained parallelism of current HPC architectures.

Obviously, employing a dynamic dropping strategy introduces additional synchronization points, virtually forbidding the parallelization of a Gaussian elimination process generating a threshold-ILU.

A fundamentally different strategy for generating incomplete factorizations is the ParILU algorithm that abandons the Gaussian elimination process [10]. Instead, it uses fixed-point iterations to approximate the incomplete factors on a predefined sparsity pattern. The idea is based on the observation that for a given ILU sparsity pattern S, the incomplete factorization is exact in the locations of S, that is [10]

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

where $(LU)_{ij}$ denotes the (i,j) entry of the product of the computed factors L and U, and a_{ij} is the corresponding entry in the matrix A.

A factorization fulfilling this property can be computed iteratively via a bilinear fixed-point iteration of the form $x = G(x)$ where x is the vector containing the unknown values

$$\begin{aligned} l_{ij}, &\quad i \geq j, \quad (i,j) \in S, \\ u_{ij}, &\quad i \leq j, \quad (i,j) \in S \end{aligned}$$

in the incomplete factors L and U. From (1), one can derive

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \quad i \geq j, \tag{2}$$

$$u_{ij} = \frac{1}{l_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right), \quad i < j, \tag{3}$$

$$u_{ij} = 1, \quad i = j. \tag{4}$$

Aside from the theoretical proof that the fixed-point iteration updating all values in the incomplete factors converges (for a suitable initial guess) in the asymptotic sense [10], experiments using the ParILU algorithm in highly parallel environments reveal that a few sweeps are often sufficient to generate preconditioners competitive to those generated via the (sequential) truncated Gaussian elimination process [10]–[12]. As a result, the ParILU algorithm outlined in Algorithm 1 has been established as an attractive alternative to the Gaussian elimination process for generating level-ILU preconditioners, and is today an integral part of many sparse linear algebra libraries designed for multi- and manycore architectures, such as ViennaCL [1] or MAGMA-sparse.[2] The attractiveness of the ParILU algorithm mostly stems from the kernel's simplicity (see Algorithm 1), and the potential to efficiently run on manycore architectures like GPUs [11], [13].

Algorithm 1 One sweep of the fixed-point ILU algorithm.

Input sparse matrix A, desired sparsity pattern S, and current L and U factors
for $(i,j) \in S$ **do**
  if $i > j$ **then**
    $l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right)$
  else
    $u_{ij} = \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right) / l_{ii}$
  end if
end for

The generation of threshold-based ILU factorizations requires a more sophisticated algorithm that can dynamically adapt the nonzero structure to the size of the fill-in elements.

The ParILUT algorithm [4] interleaves the fixed-point iterations (2), (3) and (4) approximating the values in the incomplete factors for a given sparsity pattern with a strategy that dynamically adapts the nonzero structure to the problem characteristics. To that end, ParILUT employs building blocks that identify structural fill-in locations, approximate values in the incomplete factors via fixed-point iterations, and iteratively add and remove nonzeros from the incomplete factors to

[1] http://viennacl.sourceforge.net/
[2] http://icl.cs.utk.edu/magma/

```
__global__ void parilu_kernel(
 const int num_rows, const int nnz,
 const int *rowidxA, const int *colidxA, const double *A,
 const int *rowptrL, const int *colidxL, double *L,
 const int *colptrU, const int *rowidxU, double *U) {

 int k = blockDim.x * blockIdx.x + threadIdx.x;
 int i, j, il, iu, jl, ju;
 double s, sp;

  // the sparsity pattern S are the nonzero locations in A
 if (k < nnz) { // sweep over all locations in A
  i = rowidxA[k]; // row of element in A
  j = colidxA[k]; // col of element in A
  s = A[k];     // start with value of A

  il = rowptrL[i];
  iu = colptrU[j];
  while (il < rowptrL[i+1] && iu < colptrU[j+1]) {
   sp = 0.0;
   jl = colidxL[il];
   ju = rowidxU[iu];
   sp = (jl == ju) ? L[il] * U[iu] : sp;
   s = (jl == ju) ? s-sp : s;
   il = (jl <= ju) ? il+1 : il;
   iu = (jl >= ju) ? iu+1 : iu;
  }
  s += sp;
  if (i > j)  // modify L-entry
   L[il-1] = s / U[colptrU[j+1]-1];
  else        // modify U-entry
   U[iu-1] = s;
 }
}
```

Fig. 1. CUDA kernel performing one ParILU sweep of Algorithm 1.

include the most significant elements while preserving the sparsity of the incomplete factors, see Figure 2.

The algorithm starts with some initial guess for the nonzero pattern and nonzero values in these locations. A natural starting point is to use the upper and lower triangular parts of the system matrix A as lower and upper incomplete factors [4]. Applying the ParILU algorithm would generate incomplete (level-)ILU factors with a zero ILU residual $R = A - L \cdot U$ in all locations included in the sparsity pattern S of the current incomplete factors L and U, as described by (1). At the same time, the ILU residual will not necessarily be zero in the locations outside S. This motivates us to consider the locations with a nonzero ILU residual as "candidate fill-in matrix" F:

$$F := R \backslash (L \cup U) = (A - L \cdot U) \backslash (L \cup U). \tag{5}$$

We note that computing F is equivalent to computing the level-1 fill of a level-ILU, considering the current incomplete factors as level 0 [4].

Once F is computed, the ParILUT algorithm adds the candidate locations to the incomplete factors, and uses a fixed-point sweep of the ParILU algorithm to adjust the values in the (extended) incomplete factors. The enlarged incomplete factors introduce additional nonzero locations to the new ILU residual. Obviously, recursively applying this strategy of adding nonzero locations to the sparsity pattern increases the nonzero count in the incomplete factors, and will ultimately result in a significant amount of fill-in. As a mitigation strategy, instead of adding additional locations, the ParILUT algorithm first selects a threshold separating the smallest values, and drops all elements smaller than this threshold from the incomplete factors. A second ParILU sweep is needed to adjust the values in the truncated factors. This way, the nonzero count of the original factors is preserved, and a new iteration can start with identifying potential fill-in candidates for the new factors. Iteratively applying the ParILUT cycle can result in incomplete factorizations that have a different sparsity pattern than the level-based ILU factorizations, and are superior in terms of reflecting the problem characteristics and improving the convergence of a top-level solver [4].

The ParILUT algorithm is the first parallel threshold ILU algorithm, and in [4] it is shown that it can be realized efficiently on multicore architectures. All of the building blocks forming the ParILU algorithm are amenable to parallelization and can efficiently exploit a sophisticated cache hierarchy due to a high data reuse rate. In particular, the threshold selection process traversing and rearranging the values in memory heavily benefits from data reuse.

Unfortunately, GPU architectures do not provide deep cache hierarchies, and data reuse across thread blocks is generally impossible. Hence, it is necessary to redesign the ParILUT algorithm and employ different strategies to parallelize the distinct building blocks. Most importantly, a fundamentally different strategy to derive the thresholds separating the smallest values is needed.

III. Parallel selection on GPUs

Identifying a threshold that separates the k smallest elements (in terms of magnitude) from a sequence is equivalent to finding its k-th smallest element, which is the typical setting for a selection algorithm.

A simple solution would be to sort the sequence, since the desired element would appear in position k. However, sorting the magnitudes in the complete sequence is computationally expensive, so most efficient selection algorithms are based on partitioning and sorting only partially. The elements are first partitioned among multiple buckets such that each bucket contains only the elements from a certain interval (and all the intervals are disjoint). Knowing the number of elements in each bucket, it is easy to determine which bucket contains the k-th smallest element. Then, the same partitioning procedure can be recursively applied only on this bucket, until the desired element is found.

An important ingredient of this approach is a procedure that efficiently determines good delimiters ("splitters") such that the buckets are (almost) balanced in size. The best delimiters for partitioning the sequence into b buckets are the k/b-quantiles for $k = 1, \ldots, b-1$. However, obtaining them can be computationally expensive. A more practical approach is to use the quantiles of a small random sample of the input sequence as an approximation. Together, these ideas lead to the Sampleselect algorithm which is outlined in Figure 3 and described in more detail in the following paragraphs.

Splitter selection. The first step of the algorithm includes the selection of "splitters" which will serve as the boundaries

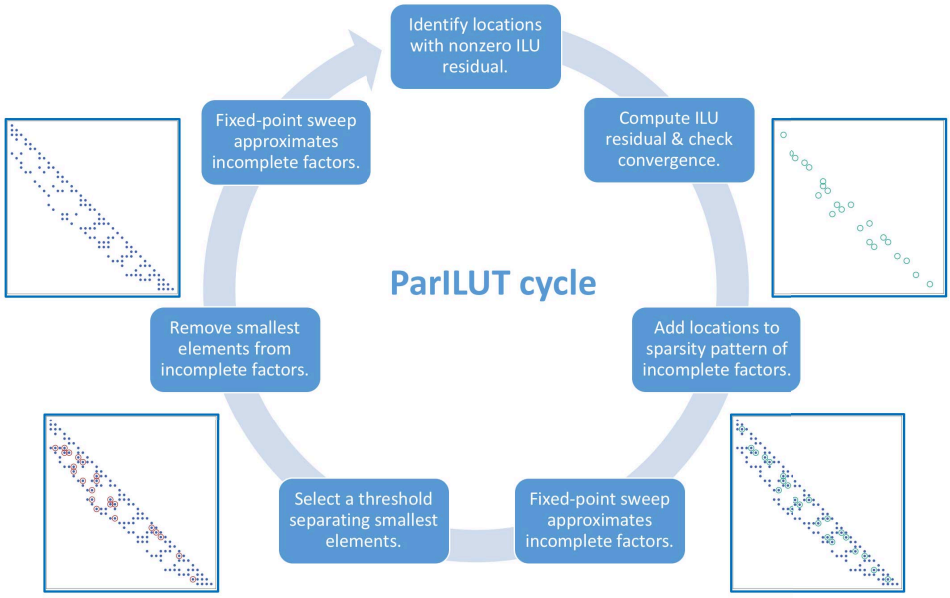

Fig. 2. The ParILUT algorithm for computing incomplete factorizations based on thresholding.

```
1  double select(data, rank) {
2      if (size(data) <= base_case_size) {
3          sort(data);
4          return data[rank];
5      }
6      // pick sample, select splitters from it
7      splitters = pick_splitters(data);
8      // compute bucket sizes
9      counts = count_buckets(data, splitters);
10     // compute bucket ranks
11     offsets = prefix_sum(counts);
12     // determine bucket containing rank
13     bucket = lower_bound(offsets, rank);
14     // recursive subcall
15     data = extract_bucket(data, bucket);
16     rank -= offsets[bucket];
17     return select(data, rank);
18 }
```

Fig. 3. High-level overview of the Sampleselect algorithm

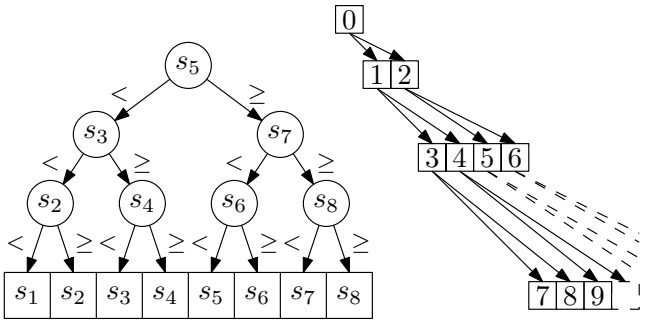

Fig. 4. Search tree based on bucket splitters $s_1, \ldots, s_8$ (left) and its implicit array storage order (right).

of the buckets. After picking a small random sample of the input data, we sort the elements to determine the sample quantiles. They are then organized into an search tree as shown on the left-hand side of Figure 4.

Element classification. The bucket of each element in the original sequence can be determined by descending from the root to a leaf of the splitter tree (see Figure 5). For each element, the path to take at a node depends on the comparison between that element and the node's value. If the value is smaller than the element, the next considered node is the left child. Otherwise, the right child is visited next. The leaf reached by the procedure is the largest splitter s_i smaller than the element, i.e., the lower delimiter of the corresponding bucket i.

As first discussed in the context of the super-scalar sample sorting algorithm [14], the index calculations in this procedure can be implemented efficiently if we store the search tree in level-order within an array (the root is assigned index 0 and the children of a node at index i are assigned indexes $2i+1$ and $2i+2$). This is visualized by the right-hand side of Figure 4. The ideal number of splitters is 2^h for some h, as the search tree then becomes a complete binary tree with h levels, where the i-th element in the last level corresponds to the i-th bucket.

Several important observations can be made about the procedure. First, the same result can be achieved by performing a simple binary search on the sorted array of splitters, without forming the search tree. However, this would require more complex index calculations. Second, the subsequent steps of the algorithm do not require the elements to be physically arranged into buckets, so just determining the size of each bucket is sufficient. However, bucket extraction on line 15 of Figure 3 needs to traverse the entire sequence once more.

```
double element = data[idx];
double tree[2 * tree_width - 1];
int i = 0;
for (int l = 0; l < tree_height; l++)
    i = 2 * i + (element < tree[i] ? 1 : 2);
int bucket = i - (tree_width - 1);
counts[bucket]++;
oracles[idx] = bucket;
```

Fig. 5. Loop for traversing the implicit search tree.

To reduce the memory footprint, the bucket index of each element (we use the term "oracle" according to [14]) is stored during the classification. Then, the second traversal can be made over oracles instead of the original sequence. If the number of buckets is small enough, the storage space required for the oracles can be significantly smaller than that of the whole sequence. For example, when using 256 buckets, each oracle can be stored within a single byte, reducing the memory transfers for the second traversal by up to 4 and 8 times (for single and double precision, respectively).

Selecting the k-th element. Once the size c_b of each bucket b is known, the bucket containing the k-th element can be determined by computing the lowest rank $r_b = \sum_{i=1}^{b-1} c_i$ of each bucket. The desired element is in bucket t for which $r_t < k \leq r_{t+1}$ holds. This operation can be performed using an exclusive scan over c_b to obtain r_b, followed by a search of the result to obtain t.

The desired element can be obtained by recursively applying Sampleselect on bucket t. Alternatively, if only an approximate threshold is required, the procedure can be stopped as soon as r_t is close enough to k. In that case, the splitter s_t of the bucket can be used as an approximate threshold.

Performance optimizations

Parallel counting and filtering. Global synchronization and communication operations can quickly become a bottleneck of parallel algorithms. In Sampleselect, the need for global communication appears in `count_buckets` and `bucket_extract`. `count_buckets` performs a histogram-like computation, which results in race conditions when updating the total counts. In `bucket_extract`, each element of the bucket has to be assigned an unoccupied index in the output array, demanding communication to avoid overwriting an existing element.

On modern GPUs, both issues can be solved via atomic operations. Atomic operations provide an efficient (but limited) form of communication between threads as they combine a data load, computation, and data store operation while eliminating the danger of race conditions. Atomic operations can be used in global and shared memory, but, due to the smaller access latency, shared memory atomics usually incur a significantly lower overhead. However,the scope of shared memory atomics is limited to the same thread block, thus their use requires a global reduction step.

Warp-aggregated atomics. Atomic operations on GPUs tend to suffer heavily from collisions, i.e., simultaneous atomic access to the same memory location from distinct threads. One popular mitigation strategy is the use warp-aggregated atomic operations [15]. As our search tree separates the input data into 256 buckets, the birthday paradox [16] suggests that even for well-distributed input data, a high chance of bucket index collisions within a warp can be expected. For general input data, this collision rate may be even higher. Combining the search tree traversal with intra-warp communication instructions, it is possible to simultaneously compute the bucket index as well as a bitmask indicating all thread lanes with the same index. This bitmask represents colliding atomic operations, which can be combined and executed by a single thread instead.

With the introduction of fast shared memory atomics in NVIDIA's Maxwell architecture [17], the technique becomes obsolete, and our experiments indicate that manual warp-aggregation slows down kernel execution — most likely because of the overhead of the additional intra-warp communication.

IV. ParILUT-GPU

Aside from the threshold selection, the ParILUT algorithm consists of the following building blocks: the candidate search identifying the locations with nonzero ILU residual; the computation of the ILU residual; the functionality to add/remove locations from the incomplete factors; and the fixed-point sweeps approximating the values of the incomplete factors.

Candidate search. As elaborated in Section II, the candidate search can be realized in terms of computing the fill-in matrix $F = (A - L \cdot U) \backslash (L \cup U)$. For this computation, we design a customized procedure similar to a general sparse matrix product (SpGEMM). In addition to computing the nonzero locations of the product $L \cdot U$, the procedure also includes the nonzero locations of the system matrix A, but filters out locations that are present in either L or U. As in most SpGEMM kernels, a two-pass approach is employed. The first row-parallel pass calculates the element count in the distinct rows. From the nonzero counts, the row pointers can be constructed using an exclusive scan. The exclusive scan also computes the total memory requirements of the column index and value arrays, enabling the memory allocation of those arrays. The second row-parallel pass inserts column indexes into the CSR structure for the elements identified in the first pass. No numeric values are assigned as the candidate search only identifies locations with a nonzero ILU residual. The actual values in these locations will later be computed in the "Residual" routine. The cost of the candidate search significantly depends on the nonzero structure of the matrices, in particular, the amount of fill-in elements.

Add/Remove elements. To adapt the sparsity pattern to the problem characteristics, the ParILUT features building blocks that either add candidate locations to the sparsity pattern, or remove locations if they are smaller than a certain threshold. Both functionalities employ a two-pass approach. In the first pass, the memory requirement of the modified incomplete factors is analyzed by a row-parallel procedure for computing the number of elements that are included in

```
__global__ void parilu_kernel_L(
 const int num_rows, const int nnz,
 const int *rowidxA, const int *colidxA, const double *A,
 const int *rowptrL, const int nnzL,
 const int *colidxL, const int *rowidxL, double *L,
 const int *colptrU, const int *rowidxU, const double *U)
     {

 int k = blockDim.x * blockIdx.x + threadIdx.x;
 int i, j, il, iu, jl, ju;
 double s=0.0, sp=0.0;

 // the sparsity pattern S are the nonzero locations in L
 if (k < nnzL) { // sweep over all locations in L
  i = rowidxL[k]; // row of element in L
  j = colidxL[k]; // col of element in L
  if (i == j) {  // L has a unit diagonal
   L[k] = 1.0;   // set value to 1.0
  } else {
   // check whether A contains an element in this location
   for (int z = rowidxA[i]; z<rowidxA[i+1]; z++) {
    if (colidxA[z] == j) {
     s = A[i];
     break;
    }
   }
   il = rowptrL[i];
   iu = colptrU[j];
   while (il < rowptrL[i+1] && iu < colptrU[j+1]) {
    sp = 0.0;
    jl = colidxL[il];
    ju = rowidxU[iu];
    sp = (jl == ju) ? L[il] * U[iu] : sp; // match
    s = (jl == ju) ? s-sp : s;
    il = (jl <= ju) ? il+1 : il; // increment row
    iu = (jl >= ju) ? iu+1 : iu; // increment column
   }
   s += sp;
   L[k] = s / U[colptrU[j+1]-1];
  }
 }
}
```

Fig. 6. CUDA kernel performing one fixed-point sweep on the lower triangular factor L.

```
__global__ void residual_kernel(
 const int num_rows, const int nnz,
 const int *rowidxA, const int *colidxA, const double *A,
 const int *rowptrL, const int *colidxL, double *L,
 const int *colptrU, const int *rowidxU, double *U,
 const int nnzF, const int *rowidxF, const int *colidxF,
 double *F) {

 int k = blockDim.x * blockIdx.x + threadIdx.x;
 int i, j, il, iu, jl, ju;
 double s, sp;
 // the sparsity pattern S are the nonzero locations in F
 if (k < nnzF) { // sweep over all locations in F
  i = rowidxF[k]; // row of element in F
  j = colidxF[k]; // col of element in F
  // check whether A contains an element in this location
  for (int z = rowidxA[i]; z<rowidxA[i+1]; z++) {
   if (colidxA[z] == j) {
    s = A[i];
    break;
   }
  }

  il = rowptrL[i];
  iu = colptrU[j];
  while (il < rowptrL[i+1] && iu < colptrU[j+1]) {
   sp = 0.0;
   jl = colidxL[il];
   ju = rowidxU[iu];
   sp = (jl == ju) ? L[il] * U[iu] : sp; // match
   s = (jl == ju) ? s-sp : s;
   il = (jl <= ju) ? il+1 : il;  // increment row
   iu = (jl >= ju) ? iu+1 : iu;  // increment column
  }
  s += sp;
  F[k] = s;
 }
}
```

Fig. 7. CUDA kernel computing the ILU residual values in the candidate locations.

the updated factors. In the routine adding new locations to the sparsity pattern, the number of nonzeros in a row comprises the nonzeros in the current incomplete factor and the number of candidates for this row. In the routine dropping locations from the incomplete factors, the values in each row are compared to the threshold, those larger than the threshold are counted, and those smaller than the threshold are marked for removal. A succeeding reduction calculates the total memory requirement of the updated sparse structures. Once the memory for the new incomplete factors is allocated, the second row-parallel pass fills the structures with the elements.

Fixed-point sweeps. The approximation of the values for a given sparsity pattern is realized via the fixed-point iterations given in (2), (3) and (4) forming the ParILU sweep. We employ the ParILU kernel designed in [11] which is parallelized across the nonzero elements in the sparsity pattern S (see Algorithm 1). However, the kernel has to be modified to take into account that, as a result of earlier ParILUT steps, the input sparsity pattern S of the incomplete factors has diverged from the sparsity pattern of the system matrix A. Hence, the input sparsity pattern S in Algorithm 1 is no longer the sparsity pattern of the system matrix, but the sparsity pattern of the incomplete factor(s). In Figure 6, we outline the CUDA kernel for updating the values in the lower incomplete factor L. A similar kernel can be derived for updating the values in the upper incomplete factor U. The sparsity-aware ParILU kernel can also be parallelized across the nonzero entries of the current sparsity pattern. However, explicitly retrieving potential nonzero values from the system matrix A introduces some overhead compared to Algorithm 1.

Computing the ILU residual. A central question when adding the candidates to the incomplete factors is how to choose the numerical values in these locations. Experiments in [4] indicate that the ILU residual values work well as an initial guess, and an approximation for these can be computed efficiently by modifying Algorithm 1 to take the candidate locations F as the sparsity pattern S and adapt the output values, see Figure 7. By complementing this residual sweep with a global reduction, we can compute an approximation for the ILU residual norm. We note that this approximation ignores the fact that some locations included in the sparsity pattern S may have a nonzero residual, as the fixed-point sweeps updating these values may not have converged yet. Nevertheless, the ILU residual norm approximation may be useful to detect convergence or breakdown of the ParILUT algorithm [4]. In terms of computational cost, the ILU residual routine thus combines a fixed-point sweep parallelized over the

TABLE I
KEY CHARACTERISTICS OF THE HIGH-END NVIDIA GPUS. THE SUSTAINED MEMORY BANDWIDTH IS MEASURED USING THE BANDWIDTH TEST SHIPPING WITH THE CUDA SDK.

	K40m	P100	V100
Architecture	Kepler (3.5)	Pascal (6.0)	Volta (7.0)
DP Performance	1.4 TFLOPs	5.3 TFLOPs	7 TFLOPs
SP Performance	4.3 TFLOPs	10.6 TFLOPs	14 TFLOPs
Operating Freq.	0.75 GHz	1.15 GHz	1.53 GHz
Mem. Capacity	6 GB	16 GB	16 GB
Mem. Bandwidth	288 GB/s	732 GB/s	900 GB/s
Sustained BW	193 GB/s	500 GB/s	742 GB/s
L2 Cache Size	1.5 MB	4 MB	6 MB
L1 Cache Size	64 KB	64 KB	128 KB

TABLE II
TEST MATRICES.

Matrix	Origin	Num. Rows	Nz
ANI5	2D anisotr. diff.	12,561	86,227
ANI6	2D anisotr. diff.	50,721	349,603
ANI7	2D anisotr. diff.	203,841	1,407,811
APACHE1	Suite Sparse [19]	80,800	542,184
APACHE2	Suite Sparse	715,176	4,817,870
CAGE10	Suite Sparse	11,397	150,645
CAGE11	Suite Sparse	39,082	559,722
JACOBIANMAT0	Fun3D fl. flow [20]	90,708	5,047,017
JACOBIANMAT9	Fun3D fl. flow	90,708	5,047,042
MAJORBASIS	Suite Sparse	160,000	1,750,416
TOPOPT010	Geometry opt. [21]	132,300	8,802,544
TOPOPT060	Geometry opt.	132,300	7,824,817
TOPOPT120	Geometry opt.	132,300	7,834,644
THERMAL1	Suite Sparse	82,654	574,458
THERMAL2	Suite Sparse	1,228,045	8,580,313
THERMOMECH_TC	Suite Sparse	102,158	711,558
THERMOMECH_DM	Suite Sparse	204,316	1,423,116
TMT_SYM	Suite Sparse	726,713	5,080,961
TORSO2	Suite Sparse	115,967	1,033,473
VENKAT01	Suite Sparse	62,424	1,717,792

candidate locations F with a global reduction to compute the approximate ILU residual norm.

V. EXPERIMENTS

A. Experiment setup

For the experimental evaluation, we use GPU architectures of different generations to reveal how architecture-specific features impact the performance of the ParILUT-GPU algorithm. The GPU architectures used in experiments and their key properties are listed in Table I. The kernels forming the ParILUT-GPU algorithm are implemented in CUDA (version 9.2) and use a default thread block size of 256. Only the selection algorithm employs a thread block size of 1024 (1024 is the maximum thread block size) to efficiently exploit shared memory atomics. We deploy the ParILUT-GPU algorithm in the MAGMA-sparse software library[3], and leverage all features provided by the MAGMA ecosystem. In particular, we use the Krylov solvers included in MAGMA-sparse [18] for experimentally assessing the preconditioner quality.

[3]http://icl.cs.utk.edu/magma/

We test the ParILUT algorithm for the same benchmark problems that were previously used in [4], which allows us to easily compare performance and preconditioner quality to the multicore implementation. For convenience, the test matrices are listed along with some key characteristics in Table II.

B. Selection algorithm on GPUs

First, we evaluate the GPU implementation of Sampleselect (`sselect`) and compare its performance to an implementation of the Quickselect (`qselect`) algorithm providing the same functionality. As the kernels are independent of the actual element values, we generate artificial datasets by randomly permuting sequences of various distributions of repeated elements. In Figure 8, the throughput of Sampleselect and of Quickselect are related to the size of the dataset. The results on the left-hand-side of Figure 8 are obtained with the older K40m GPU, and the results in the center of Figure 8 with the state-of-the-art Volta architecture. For both algorithms, two variants are examined. The variants labeled with `qselect-g` and `sselect-g` use global memory atomics, while `qselect-s` and `sselect-s` use shared memory atomics. The results reveal that on the older K40 architecture, the variants using global memory atomics outperform their shared memory counterparts. On the new V100 GPU, shared memory atomics are faster for large dataset sizes. On the right-hand-side in Figure 8, we evaluate the impact of atomic collisions on the older Kepler architecture as well as the effectiveness of warp-aggregation. We increase the collision rate by duplicating values in the dataset. While these collisions generally have a large performance impact, warp-aggregation proves to be an effective mitigation strategy. On the Volta architecture, the impact of the atomic collisions is much smaller, making warp-aggregation obsolete.

Overall, the Sampleselect algorithm consistently outperforms the Quickselect algorithm. The Sampleselect runtime can be further reduced by relaxing the algorithm's accuracy. To that end, we note that the thresholds needed in the ParILUT-GPU algorithm do not need to be exact. An approximate threshold of "good quality" will in the worst case result in small variations of the nonzero count of the incomplete factors [4].

In Figure 9, we visualize the relative runtime breakdown of a single recursion level. The results were obtained on the V100 GPU using shared memory atomics and an array of 2^{24} elements. While one level of Quickselect is faster than one level of Sampleselect, the Sampleselect algorithm requires far fewer levels. The approximate Sampleselect reduces the cost of every level by omitting the `extract_bucket` step and reducing the cost of the reduce operation, as no partial sums need to be stored. One level of the approximate Sampleselect is about two times faster than one level of Quickselect.

C. Approximate Sampleselect in ParILUT-GPU

Next, we assess the impact of using the approximate variant of Sampleselect inside the ParILUT-GPU algorithm. For this

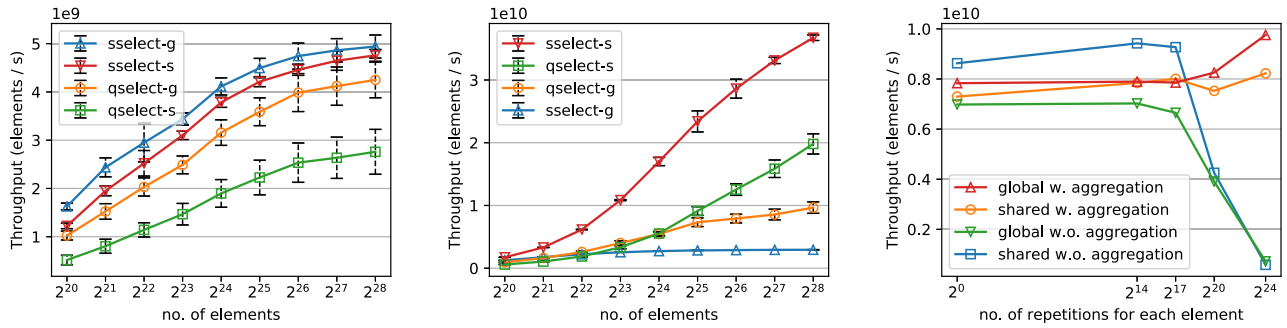

Fig. 8. Left and center: Throughput on the K40m and V100 GPU for different selection algorithms: `qselect-s` and `sselect-s` use shared memory atomics, `qselect-g` and `sselect-g` use global memory atomics; error bars indicate the variation. Right: Throughput on the K40m for a single recursion level using shared and global atomics with or without warp-aggregation, executed on datasets with different repetition rates for $n = 2^{24}$.

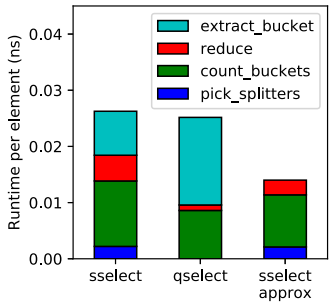

Fig. 9. Relative runtime breakdown on the V100 GPU for a single recursion level in the different selection algorithms using shared memory atomics for $n = 2^{24}$.

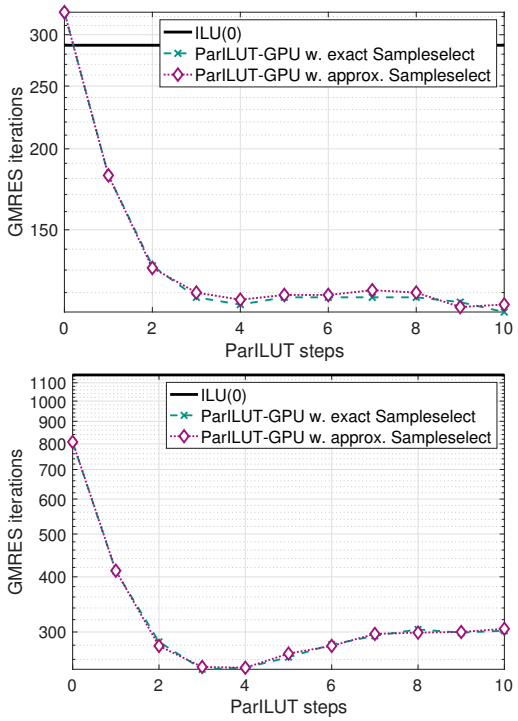

Fig. 10. Quality assessment of the preconditioner generated via the ParILUT algorithm using either the exact Sampleselect or the approximate Sampleselect. Test problems are ANI5 (top) and ANI6 (bottom).

purpose, we use the generated preconditioners inside a GMRES iterative solver and relate the iteration count necessary to reach a relative residual of 10^{-10} to the number of ParILUT-GPU steps. For comparison, Figure 10 also includes the iteration count required when employing a standard ILU(0) preconditioner. The results for "0 ParILUT steps" are obtained by taking the initial guess for the ParILUT algorithm (the lower and upper triangular factors of the system matrix) as a preconditioner. To accommodate minor differences in the iteration counts, we average the results for the approximate Sampleselect over 5 runs. The first observation is that both versions of the ParILUT-GPU preconditioner significantly decrease the GMRES iteration count compared to the ILU(0) preconditioner. Furthermore, only negligible quality differences can be observed between the version using exact Sampleselect and the one using approximate Sampleselect. Given the performance benefits, we choose to make the approximate Sampleselect the default choice inside the ParILUT-GPU algorithm.

In Figure 11 we investigate the corresponding performance benefits obtained from replacing the exact Sampleselect with the approximate Sampleselect. For each problem, we show the runtime breakdown of the ParILUT-GPU algorithm using either exact Sampleselect (left bar) or approximate Sampleselect (right bar) on the V100 GPU. As the approximate Sampleselect runs for some problems significantly faster, we choose to make it the the default choice inside the ParILUT algorithm.

D. Cross-platform portability of ParILUT-GPU

An important aspect of the ParILUT-GPU algorithm is to provide good performance portability across different GPU generations. To that end, we assess the existence of building blocks heavily optimized for one architecture, but achieving low performance on a different architecture. In the runtime breakdown of the ParILUT-GPU in Figure 12, we normalize

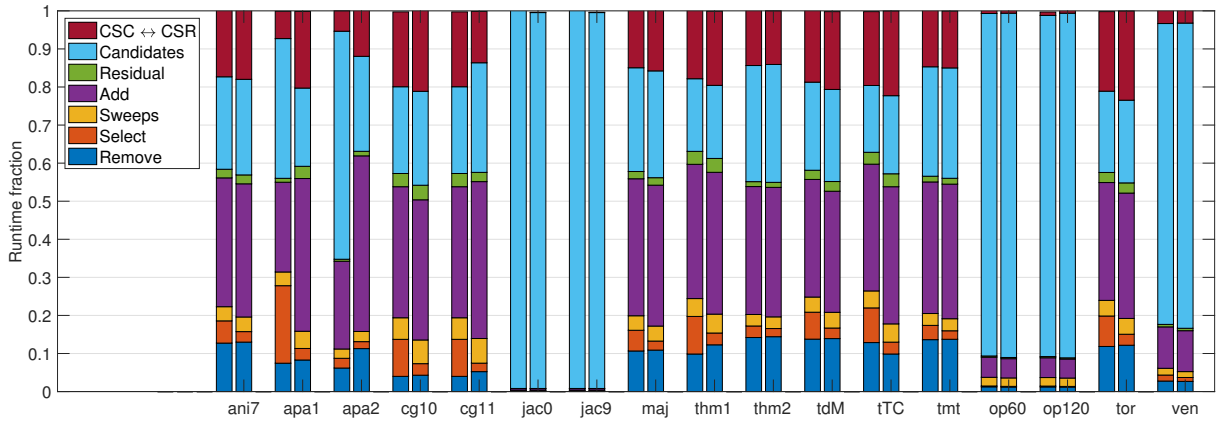

Fig. 11. Relative runtime of the distinct building blocks forming the ParILUT-GPU algorithm on the V100 GPU. The two bars for each problem reflect the breakdown of the ParILUT-GPU using exact Sampleselect (left) and approximate Sampleselect (right), respectively.

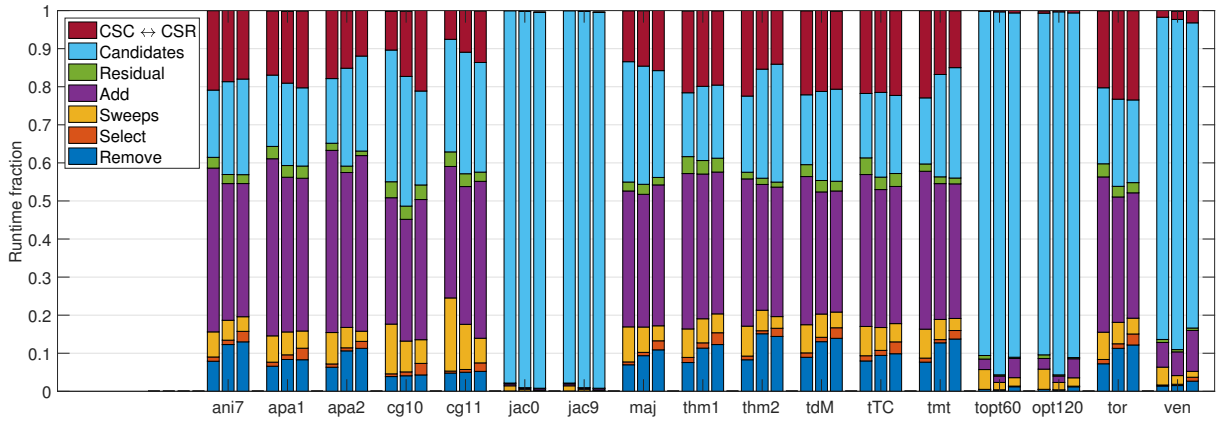

Fig. 12. Relative runtime of the distinct building blocks forming the ParILUT-GPU algorithm. The three bars for each problem reflect the breakdown on the three GPU architectures considered: K40 (left bar), P100 (center bar), and V100 (right bar).

the execution times for the specific problem and architecture configuration. The three bars for each problem correspond to the normalized execution times on the K40m, P100 and V100 GPUs. While this analysis provides information about the relative runtime of the building blocks for a specific architecture/problem setting, no information about the total execution time is given. The results indicate that the relative cost of the distinct building blocks heavily depends on the problem characteristics, but seems to be almost independent of the hardware. This indicates that the designed ParILUT-GPU algorithm consists of building blocks that all provide good performance portability. For the problems JAC0, JAC9, TOPOPT060, and TOPOPT120, the candidate search heavily dominates the ParILUT-GPU execution time. An explanation is the high nonzero-per-row ratio of these problems, see Table II, which results in a high amount of fill-in. Aside from the candidate search, the addition of nonzero locations to the sparsity structures also takes a significant portion of the runtime. A more detailed analysis reveals that the runtime contribution of the fixed-point iterations computing the values in the incomplete factors ("Sweeps") and the residuals ("Residual") decreases with newer hardware architectures.

E. Performance assessment of ParILUT-GPU

Finally, we compare the performance of the ParILUT-GPU algorithm we developed to its ParILUT counterpart designed for multicore architectures and parallelized using OpenMP. To that end, we take the performance results reported in [4] as reference point, and analyze the speedup of ParILUT running on diverse hardware architectures over the crout version of the threshold ILU (ILUT) taken from the SuperLU package[4] and running on an Intel Xeon Phi 7250 (KNL) processor.

The results in Figure 13 reveal that the ParILUT-GPU and the ParILUT(-OMP) outperform SuperLU's ILUT for all problems and all configurations. For the JAC0 and the JAC9 problems, the ParILUT-GPU is slower than the "ParILUT-OMP" running on the KNL. The reason is the sparsity pattern

[4]http://crd-legacy.lbl.gov/~xiaoye/SuperLU/

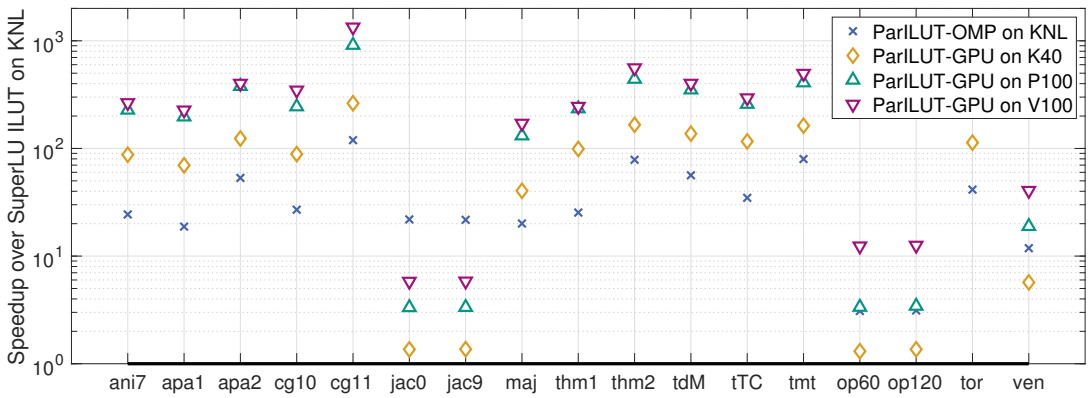

Fig. 13. Performance assessment of the ParILUT algorithm running on diverse architectures. The reference points for the speedup values are the runtimes of the SuperLU's ILUT running on an Intel KNL platform.

of these systems, which makes the candidate search extremely expensive, see Figure 12. This is a building block where the ParILUT-OMP can heavily benefit from reusing data present in cache. A similar effect can be observed for the TOPOPT060, TOPOPT120, and VEN problems, where the ParILUT-GPU outperforms the ParILUT-OMP only on the newer GPU architectures. Comparing the different GPU architectures, the ParILUT-GPU algorithm typically executes 2–3x faster when moving from the K40 to the P100 platform. On the V100 GPU, the ParILUT-GPU executes about 10%–30% faster than on the P100 GPU. Exceptions are the TOPOPT060, TOPOPT120, and VEN systems where the Volta architecture enables more substantial acceleration.

VI. Summary and Future Work

We have developed the first parallel threshold ILU algorithm for GPUs. The ParILUT-GPU algorithm interleaves fixed-point sweeps approximating values in the incomplete factors with a strategy that dynamically adapts the nonzero pattern to the problem characteristics. We compose the ParILUT-GPU out of heavily tuned GPU kernels. For threshold selection we designed the Sampleselect algorithm that outperforms other algorithms providing the same functionality. For a set of test matrices we show that the developed ParILUT-GPU algorithm executes faster than the counterpart designed to leverage the compute power of multicore processors. The performance portability analysis revealed that the search for potential fill-in candidates dominates the ParILUT-GPU runtime for many test problems and GPU architectures. Thus, future research will particularly focus on accelerating this step. Possible mitigation strategies include the development of randomized candidate search, and employing machine learning techniques for quickly generating ILUT sparsity patterns.

References

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems, 2nd Edition*. Philadelphia, PA, USA: SIAM, 2003.

[2] A. Basermann, "Parallel block ILUT/ILDLT preconditioning for sparse eigenproblems and sparse linear systems," *Numerical Linear Algebra with Applications*, vol. 7, no. 7-8, pp. 635–648, 2000.

[3] G. Karypis and V. Kumar, "Parallel threshold-based ILU factorization," in *1997 ACM/IEEE Conference on Supercomputing*, Nov 1997, pp. 1–24.

[4] H. Anzt, E. Chow, and J. Dongarra, "ParILUT—A New Parallel Threshold ILU Factorization," *SIAM Journal on Scientific Computing*, vol. 40, no. 4, pp. C503–C519, 2018. [Online]. Available: https://doi.org/10.1137/16M1079506

[5] M. Benzi, W. Joubert, and G. Mateescu, "Numerical experiments with parallel orderings for ILU preconditioners," *Electronic Transactions on Numerical Analysis*, vol. 8, pp. 88–114, 1999.

[6] S. Doi, "On parallelism and convergence of incomplete LU factorizations," *Applied Numerical Mathematics*, vol. 7, no. 5, pp. 417–436, 1991.

[7] D. Hysom and A. Pothen, "A scalable parallel algorithm for incomplete factor preconditioning," *SIAM Journal on Scientific Computing*, vol. 22, no. 6, pp. 2194–2215, 2001.

[8] D. Lukarski, "Parallel sparse linear algebra for multi-core and many-core platforms - parallel solvers and preconditioners," Ph.D. dissertation, Karlsruhe Institute of Technology (KIT), Germany, 2012.

[9] E. L. Poole and J. M. Ortega, "Multicolor ICCG methods for vector computers," *SIAM Journal on Numerical Analysis*, vol. 24, pp. 1394–1417, 1987.

[10] E. Chow and A. Patel, "Fine-grained parallel incomplete LU factorization," *SIAM Journal on Scientific Computing*, vol. 37, pp. C169–C193, 2015.

[11] E. Chow, H. Anzt, and J. Dongarra, "Asynchronous iterative algorithm for computing incomplete factorizations on GPUs," in *Proceedings of 30th International Conference, ISC High Performance 2015, Lecture Notes in Computer Science, Vol. 9137*, J. Kunkel and T. Ludwig, Eds., 2015, pp. 1–16.

[12] H. Anzt, E. Chow, J. Saak, and J. Dongarra, "Updating incomplete factorization preconditioners for model order reduction," *Numerical Algorithms*, vol. 73, pp. 611–630, 2016.

[13] H. Anzt, M. Baboulin, J. Dongarra, Y. Fournier, F. Hulsemann, A. Khabou, and Y. Wang, *Accelerating the Conjugate Gradient Algorithm with GPUs in CFD Simulations*. Cham: Springer International Publishing, 2017, pp. 35–43.

[14] P. Sanders and S. Winkel, "Super scalar sample sort," in *Algorithms – ESA 2004*, S. Albers and T. Radzik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 784–796.

[15] A. Adinets. Optimized filtering with warp-aggregated atomics. [Online]. Available: https://devblogs.nvidia.com/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

[16] F. H. Mathis, "A generalized birthday problem," *SIAM Rev.*, vol. 33, no. 2, pp. 265–270, May 1991. [Online]. Available: http://dx.doi.org/10.1137/1033051

[17] N. Sakharnykh. Fast histograms using shared atomics on Maxwell. [Online]. Available: https://devblogs.nvidia.com/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/

[18] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler, "Preconditioned Krylov solvers on GPUs," *Parallel Computing*, vol. 68,

pp. 32–44, 2017, applications for the Heterogeneous Computing Era. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819117300777

[19] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[20] NASA, "https://fun3d.larc.nasa.gov/."

[21] S. Wang, E. de Sturler, and G. H. Paulino, "Large-scale topology optimization using preconditioned Krylov subspace methods with recycling," *International Journal for Numerical Methods in Engineering*, vol. 69, no. 12, pp. 2441–2468, 2007.