Highly Parallel Linear Forest Extraction from a Weighted Graph on GPUs

Christoph Klein
Robert Strzodka
christoph.klein@ziti.uni-heidelberg.de
robert.strzodka@ziti.uni-heidelberg.de
Institute of Computer Engineering (ZITI)
Heidelberg, Germany

ABSTRACT

For graph matching, each vertex is allowed to match with exactly one other vertex, such that the spanning subgraph of the matching has a maximum degree of one, i.e., the subgraph is a [0,1]-factor. In this work, we provide a highly parallel algorithm to extract a spanning subgraph with a maximum degree of n (the subgraph is a [0,n]-factor) and demonstrate the efficiency of our GPU implementation for n=1,2,3,4 by expressing the algorithm in terms of generalized sparse matrix-vector products. Moreover, from the [0,2]-factor, we compute a maximum linear forest (union of disjoint paths) by breaking up cycles and permuting the subgraph with respect to the vertex order within the paths. Both tasks execute efficiently on the GPU because of our novel parallel scan implementation, which does not require a random access iterator. As an application of linear forests, we demonstrate the algebraic creation of enhanced tridiagonal preconditioners for various large matrices from the Sparse Matrix Collection and report runtimes in the order of milliseconds for graphs with millions of edges and vertices on an RTX 2080 Ti.

CCS CONCEPTS

• **Mathematics of computing** → **Matchings and factors**; **Graph enumeration**; **Graph algorithms**; **Mathematical software performance**; *Solvers*.

KEYWORDS

CUDA, bidirectional scan, SpMV, iterative solver, tridiagonal, preconditioner

ACM Reference Format:
Christoph Klein and Robert Strzodka. 2022. Highly Parallel Linear Forest Extraction from a Weighted Graph on GPUs. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545035

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08.
https://doi.org/10.1145/3545008.3545035

1 INTRODUCTION

We recall some graph terminology [10]. An undirected graph of order N is a pair $G := (V, E)$ with a set of vertices $V \subset \mathbb{N}_0$ ($N := |V|$) and a set of edges $E := \{\{v, w\} \mid v, w \in V\}$. A *weighted* graph has in addition a function $\omega : V^2 \to \mathbb{R}$, which returns a weight $\omega(e) \neq 0$ for each edge $e \in E$ and zero otherwise. A *path* in G is a non-empty subgraph $P := (V_P, E_P) \subseteq G$ with distinct vertices v_i in $V_P = \{v_0, v_1, \ldots, v_{k-1}\}$ and edges $E_P = \{\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-2}, v_{k-1}\}\}$. Vertices v_0 and v_{k-1} are called the *ends* of P. If we connect the ends with the additional edge $\{v_{k-1}, v_0\}$ then this is a *cycle* in G. A *spanning* subgraph of G is a graph with the same vertices as G but only a subset of its edges E.

In this paper we are interested in the fast extraction of spanning subgraphs with certain properties. A $[0, n]$-*factor* [31] is a spanning subgraph of G in which each vertex has a degree of at most n. A $[0, 1]$-factor does not contain cycles, but all $[0, n]$-factors with $n \geq 2$ can. Removing cycles in general is hard. For the $[0, 2]$-factor we will present the fast removal of cycles to obtain an acyclic $[0, 2]$-factor. It is a union of disjoint paths, also called a *linear forest*. Quickly reordering the vertices in the linear forest with respect to their order in the paths is a challenge, and we will discuss fast parallel algorithms for that.

Why are graph factors of interest? Classical graph matchings [6] compute $[0, 1]$-factors, which are used for optimizing the power consumption of wireless networks [38], preconditioning sparse linear systems [18], solving the data path allocation problem [4], and for the reordering and scaling of sparse matrices [20]. Computing maximum linear forests is the edge analog of the maximal path set problem [5], which is solved to approximate the shortest superstring problem occuring during DNA sequencing [29]. Linear forests, which contain many strong edges, are also used for directional coarsening in algebraic multigrid [24], for adaptive algebraic smoothers [30], and for the setup of tridiagonal preconditioners, which we will discuss here as an application.

Tridiagonal systems can be inverted at the bandwidth limit of the GPU [21], so they lend themselves as fast preconditioners for a linear system of equations $Ax = d$. However, simply extracting the tridiagonal part of the system matrix A does not consider the strength of the included coefficients in the tridiagonal preconditioner and therefore gives suboptimal convergence rates. Instead, we want a tridiagonal preconditioner that contains many strong coefficients from A. We obtain it by extracting a linear forest from A and a permutation under which the adjacency matrix of the linear forest has a tridiagonal form.

This paper consists of two major parts: first, formulation of the $[0, n]$-factor extraction (Section 3.2) in terms of generalized sparse matrix-vector products (Section 4.1), and second, the computation of a linear forest from a $[0, 2]$-factor by breaking up cycles and determining the path ID and position (Section 3.3). The two tasks in the second part run efficiently in parallel because of our novel parallel scan implementation, which does not require a random access iterator (Section 4.2). Section 5 presents benchmarks and Section 6 the application of enhanced tridiagonal preconditioners.

2 RELATED WORK

$[0, 1]$-*Factor Computations:* Hagemann et al. [18] use weighted matchings to precondition symmetric indefinite linear systems, also similarly performed by Naim et al. [27] on GPUs. Cohen [6] and Naumov et al. [28] describe how graph matching and coloring is implemented efficiently on GPUs. Auer and Bisseling [16] developed a greedy graph matching on GPUs with an MD5 coloring technique to coarsen a graph in the context of graph partitioning. The graph matching problem is most related to our algorithms and was well studied on GPUs [6, 16, 27, 28], but only extracts matched vertex pairs instead of long paths from a graph.

Linear Forest Computations: Uehara and Chen [36] formulate three parallel algorithms for the calculation of maximal linear forests. The work of Shoudai and Miyano [35] shows that finding a maximal vertex-induced subgraph with a maximum degree of n is an NC^2 problem. These papers are theoretical, they do not provide actual parallel implementations.

GraphBLAS:. the GraphBLAS standard [23] expresses graph problems in terms of linear algebra operations, e.g., a shortest-path calculation expressed as sparse matrix-vector multiplication on the semiring $\{\min, +, \mathbb{R} \cup \{+\infty\}, +\infty\}$. However, a memory efficient $[0, n]$-factor computation requires different types for the input and output vector, the sparse matrix, and the accumulator; this flexibility is supported by our generalized sparse matrix-vector multiplication.

Minimum Spanning Trees: Minimum spanning tree (MST) algorithms [22, 32, 37] compute an acyclic $[0, n']$-factor for an unconstrained n', so that n' potentially equals the maximum degree in the graph. An unconstrained n' allows non-mutual, parallel edge confirmations, which is not possible for a constrained n in our $[0, n]$-factors. We resolve the conflict of more than n propositions to a single vertex by requiring mutual propositions. However, the main difference is that MST algorithms keep track of connected components to avoid cycles during construction, which requires irregular data structures and limits parallelism to the number of currently connected components. Instead, we utilize highly parallel and more regular algorithmic building blocks of SpMV and bidirectional scan to get around these limitations.

Other Relations: Our graph algorithms are also related to linear sum assignment, and matrix transversal problems. The linear sum assignment problem [2, 3] is solved by a column permutation, which minimizes the sum of the diagonal entries of the column-permuted matrix, which generally represents a weighted bipartite graph, and was implemented on a GPU by Date and Nagi [7]. Maximum matrix transversals aim to provide a permutation, which maximizes the sum, product, or amount of non-zero entries of the diagonal elements of the permuted matrix, and was intensively studied by Duff et al. [11–15]. Both related problems can also be used to design algorithms which extract one-dimensional subgraphs, but although they could provide long paths, they rely on parallel breadth-first search algorithms [7] or consider dense matrices only [7, 17, 33].

3 ALGORITHMS

We divide the algorithmic contributions of this paper into two major parts: first, the computation of the $[0, n]$-factor π, and second, the extraction of a linear forest from a $[0, 2]$-factor.

3.1 Factor Notation

We generalize the notation from [16] and choose the following functional representation of the $[0, n]$-factor π:

$$\pi : V \to \Phi := \{\phi \in \mathcal{P}(V) \mid |\phi| \leq n\}, \quad \pi(v) \subseteq V_v, \tag{1}$$

$$\text{with } V_v := \{w \in V \mid \exists e \in E : \{v, w\} = e\} \setminus \{v\}, \tag{2}$$

and $\mathcal{P}(V)$ representing the power set of V. Thus, $\pi(v)$ returns either the empty set, one vertex, ..., or n vertices from the *neighborhood* V_v of v. Function π is subject to the following conditions:

(1) For all $v \in V$ there exist at most n different vertices $w_i \in V \setminus \{v\}$ such that $v \in \pi(w_i)$ with $i = 0, \ldots, n-1$, i.e., we allow a vertex to have at most n neighboring vertices in the $[0, n]$-factor.
(2) For all $v, w \in V$, $v \neq w$, if $v \in \pi(w)$, then $w \in \pi(v)$, $w \in V_v$, and $v \in V_w$, i.e., we only include existing edges $\{v, w\} \in E$ in the $[0, n]$-factor.

Algorithm 1: Sequential greedy $[0, n]$-factor computation on a weighted graph $G = (V, E)$.

```
for v ∈ V do
    π(v) ← ∅
end
for (v, w) ∈ E in order of decreasing |ω({v, w})| do
    if |π(v)| < n, |π(w)| < n, and v ≠ w then
        π(v) ← π(v) ∪ {w}
        π(w) ← π(w) ∪ {v}
    end
end
```

If it is not possible to increase the size of $\pi(V)$ further without breaking the conditions, the $[0, n]$-factor is *maximal*.

The *weight* of a $[0, n]$-factor ω_π is defined as

$$\omega_\pi := \sum_{e \in E_\pi} |\omega(e)|, \quad E_\pi := \{\{v, w\} \in E \mid v \in \pi(w)\}, \tag{3}$$

and the *relative weight coverage* c_π as

$$c_\pi := \frac{\omega_\pi}{\omega_G}, \quad \omega_G := \sum_{\{v,w\} \in E, v \neq w} |\omega(\{v, w\})|. \tag{4}$$

For comparisons with the original vertex ordering we also define

$$c_{\text{id}} := \left(\sum_{i=0}^{N-1} |\omega(\{i, i-1\})| + |\omega(\{i, i+1\})| \right) / \omega_G \tag{5}$$

Algorithm 2: Parallel $[0, n]$-factor algorithm. Function `charge` returns positve(+) or negative(-).

```
1  Function max_target(v, Θ)
2      return arg max_{w∈Θ} (|ω({v, w})|)
3  end
   Data: Weighted graph G = (V, E), ω, and integers m, k_m
   Result: Number of iterations M_max and [0, n]-factor π
4  for v ∈ V do in parallel
5      π(v) ← ∅
6  end
7  for k = 0, ..., M − 1 do
8      if k mod m ≠ k_m then
9          for v ∈ V do in parallel // kernel launch
10             q(v) ←charge(v,k)
11         end
12     end
13     π′ ← π // copy current π
14     for v ∈ V do in parallel // kernel launch
15         W ← V_v \ {w ∈ V | |π′(w)| = n}
16         if k mod m ≠ k_m then
17             W ← W \ {w ∈ V | q(w) = q(v)}
18         end
           // Propose edges to neighbors
19         while |π(v)| < n and |Θ ← W \ π(v)| > 0 do
20             π(v) ← π(v) ∪ {max_target(v, Θ)}
21         end
22     end
       // [0, n]-factor is maximal?
23     if |π(V)| = |π′(V)| and k mod m = k_m then
24         return k + 1 // return M_max
25     end
       // Remove non-mutual propositions
26     for v ∈ V do in parallel // kernel launch
27         π(v) ← {w ∈ π(v) | v ∈ π(w)}
28     end
29 end
30 return M
```

Algorithm 3: Parallel bidirectional scan to compute path IDs and positions for a linear forest.

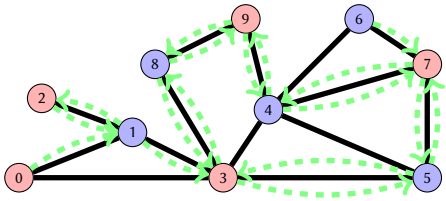
```
   Data: acyclic [0, 2]-factor π
   Result: path IDs l, and positions p
1  for v ∈ V do in parallel // kernel launch
2      r(v) ← (1, 1)
3      q(v) ←make_tuple(π(v))
4  end
5  while
       ∃ w_i ∈ q(v) for any v ∈ V such that not is_path_end(w_i) do
6      q′ ← q// copy
7      r′ ← r// copy
8      for v ∈ V do in parallel // kernel launch
9          (w_0, w_1) ← q′(v)
10         (r_0, r_1) ← r′(v)
11         for i = 0, 1 do
12             if not is_path_end(w_i) then
13                 (v_0, v_1) ← q′(w_i)
14                 (t_0, t_1) ← r′(w_i)
15                 for j = 0, 1 do
16                     if v_j ≠ v then
17                         r_i ← r_i + t_j
18                         w_i ← v_j
19                     end
20                 end
21             end
22         end
           // write updated values
23         q(v) ← (w_0, w_1)
24         r(v) ← (r_0, r_1)
25     end
26 end
   // chose one path end as path ID
27 for v ∈ V do in parallel // kernel launch
28     (w_0, w_1) ← q(v)
29     (r_0, r_1) ← r(v)
30     i ← arg min_{j∈{0,1}}(w_j)
31     l(v) ← w_i
32     p(v) ← r_i
33 end
```

3.2 Parallel $[0, n]$-Factor Algorithm

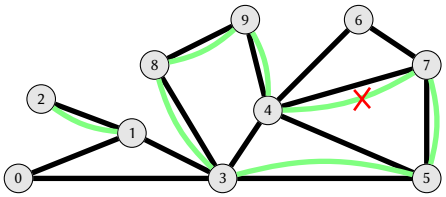
To evaluate the quality of our parallel $[0, n]$-factor algorithm, we use the sequential greedy $[0, n]$-factor Algorithm 1, which sorts the edges with respect to their weight in decreasing order and adds them to π if possible. Note that for $n = 1$, this algorithm computes a matching with at least half of the maximum weight considering all possible matchings [16].

Our parallel $[0, n]$-factor Algorithm 2 is structured similarly to the graph matching techniques of Auer and Bisseling [16], but contrary to classical graph matching, a vertex is allowed to have n other neighboring vertices in the $[0, n]$-factor. For each vertex at most n outgoing edges with the largest absolute weights are *proposed* in parallel in Algorithm 2 Line 19 and non-mutually proposed edges are removed in Line 27. The remaining edges are the *confirmed* edges and part of the $[0, n]$-factor. The neighboring set for potential propositions Θ in Line 19 excludes neighbors which already have n confirmed edges (Line 15), the same charge if $k \mod m \neq k_m$ (Line 17), and neighbors with existing confirmed edges (Line 19). If a vertex is charged prior to the edge proposition, it is positive(+) or negative(-) with a probability of p and $(1 - p)$, respectively, and is only allowed to propose to vertices with a different charge. The randomness of charge assignments enables larger $[0, n]$-factors for graphs with unfavorable structural edge weight properties, e.g., strict monotonically increasing edge weights in a specific direction. The charge of a vertex depends on its ID and the iteration index k.

The propositions and confirmations of edges are done iteratively with loop index k, to include more edges in the $[0, n]$-factor. The parameters m and k_m control when vertex charging is enabled, M represents an upper limit for the number of propositions, and the algorithm returns the number of actual propositions $M_{\max}$ for a maximal $[0, n]$-factor in Line 24.



(a) Parallel proposition of edges to vertices of different charge (+,-). Each vertex proposes at most two edges to its neighbors (two strongest edges).



(b) Confirmation of mutually proposed edges. For the linear forest extraction, the match between vertex 4 and 7 is removed to break up the cycle.

Figure 1: Edge proposition and confirmation for a $[0, 2]$-factor (Algorithm 2, with $n = 2$, $k = 0$, $k_m = 0$) executed on a small graph.

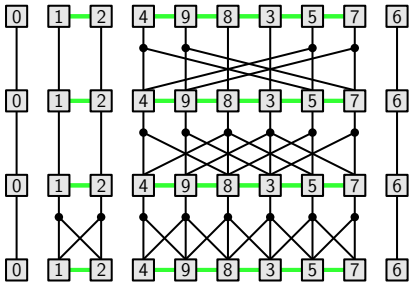


Figure 2: Three steps (bottom to top) of a parallel bidirectional scan for the graph of Figure 1 with $N = 10$ vertices and 4 paths. Vertices of the same path are connected with green horizontal lines. Each step represents one kernel launch.

$((A')_{4,j},\ j)$	(0.2, 3)	(0.3, 5)	(0.9, 6)	(0.4, 7)	(0.5, 9)
accumulator without charging	(0.2, 3) (0.0, _)	(0.3, 5) (0.2, 3)	(0.9, 6) (0.3, 5)	(0.9, 6) (0.4, 7)	(0.9, 6) (0.5, 9)
charge	+	-	-	+	+
accumulator with charging	(0.2, 3) (0.0, _)	(0.2, 3) (0.0, _)	(0.2, 3) (0.0, _)	(0.4, 7) (0.2, 3)	(0.5, 9) (0.4, 7)

Table 1: Edge proposition for vertex 4 (-) of Figure 1 expressed as reduction along matrix row $(A')_{4,j}$ from left to right. The accumulator consists of two ($n = 2$) sorted pairs $((A')_{i,j}, j)$.

Figure 1 shows the edge proposition and confirmation for charged vertices for $n = 2$. For each iteration step k, the charging of the vertices is disabled if $k \mod m = k_m$, such that each vertex is allowed to propose to every neighbor which did not reach the maximum number of confirmed edges yet. This has two advantages: first, for some graphs, the unrestricted edge proposition creates large $[0, n]$-factors after the first proposition step. If vertex charging had been used for these graphs, more iterations would have been required to obtain the same $[0, n]$-factor size. Second, if nothing is proposed without vertex charging enabled, the $[0, n]$-factor is maximal and we may stop the iterations (Alg. 2 Line 23).

3.3 From a $[0, 2]$-Factor to a Linear Forest

The $[0, 2]$-factor represented by π contains only connectivity information. Only the two or fewer neighbors are known for each vertex, whereas it is unknown in which path or cycle a vertex is located (there is no path ID), nor at which position within the path or cycle a vertex resides. We first break up the cycles. For that purpose, the weakest edge of each cycle is removed to keep the weight ω_π (Eq. 3) of the linear forest large. Afterward, we compute the path ID and position within the path for each vertex. A permutation of the adjacency matrix of the linear forest, which makes it tridiagonal, is obtained by sorting the vertex IDs with respect to their key composed of path ID and position.

We arrange the linear forest extraction algorithm from a $[0, 2]$-factor π into four *steps*:

(1) **Identify cycles:** identify cycles and break them up by removing their weakest edge.
(2) **Identify paths:** obtain the path ID and position within the path for all vertices.
(3) **Compute permutation:** sort vertex IDs with respect to their path ID and position to get the permutation in which the adjacency matrix of the linear forest is tridiagonal.
(4) **Extract weight coefficients:** with the permuattion, extract coefficients from the adjacency matrix of G.

For steps (1) and (2), we use a bidirectional scan to design parallel algorithms. A *bidirectional* scan executes two scans in two opposite directions simultaneously. In this way, we can compute the result of two different opposed scans or find and broadcast a specific value in parallel. The access pattern of our bidirectional scan on multiple paths is shown in Figure 2. Considering a single path, this pattern appears in the Parallel Cyclic Reduction Algorithm [9]. Such an access pattern allows, for example, to broadcast a value to all threads participating in the scan, although a single thread does not visit every neighbor explicitly. The scan must be bidirectional because π is structured like a double-linked list but with unknown orientation about which neighbor is forward and which backward, e.g., within the $[0, 2]$-factor, the forward neighbor of vertex 8 in the right part of Figure 1 might be vertex 9, but vertex 4 might be the backward neighbor of vertex 9.

In the bidirectional scan for step (2), the path-ends and the path-positions for all vertices are determined, which is shown in Algorithm 3. For the latter, each vertex initializes its forward and backward oriented position with a '1' in Line 2 and the bidirectional scan with an addition operator, which is applied in Line 17,

computes the path position in both directions. The initialization of the stride-q neighbors in Line 3, sets $q(v)$ to the $[0,2]$-factor neighbors $\pi(v)$, and fills up the tuple with the vertex ID v, but marked as a path end. First, q saves the vertices, which are visited in the next scan iteration step but also contains both path ends after execution, which is assigned to the result in Line 31. We define the path ID as the minimum ID of the vertices at the path ends, and this defines also the orientation: the vertex at the path end with the smaller ID is at position 1, its neighbor at position 2, etc.

Scan algorithms are often parameterized on the operation (e.g., `thrust::inclusive_scan` [19]) so that prefix sums and other properties like minima can be computed by changing the operator to `min`. In the same fashion, we use a bidirectional scan for step (1) to determine the weakest edge within a cycle. The weakest edge is uniquely identified by the weight and the IDs of the incident vertices. The algorithm for step (1) is constructed analogously to step (2) but only identifies cycles and their weakest edge.

Theoretically, steps (1) and (2) can be merged by searching for the weakest edge and the distance to it, but in practice this incurs more data movement and longer running times.

4 IMPLEMENTATION

All implementations use CUDA for a parallel execution on NVIDIA GPUs. In the following, we assume that the graph is saved as an adjacency matrix A. Thus, each matrix entry a_{ij} corresponds to the weight of the edge between vertex i and vertex j. To avoid additional branching in the kernels the diagonal of A is deducted and the coefficients are set to their absolute values with $A' := |A| - \texttt{diag}(|A|)$ before the $[0,n]$-factor computation. The implementation of Algorithm 2 also supports directed input graphs for the calculation of π, which works well for the test cases presented in this paper. However, constructing π from an underlying undirected graph and extracting the coefficients from the original graph is a better alternative for general graphs.

4.1 Parallel $[0,n]$-Factor Computation for $n \leq 4$

The edge proposition (Alg. 2, Line 14) was implemented by leveraging a *generalized* sparse matrix-vector product on the GPU, where the multiplication is replaced by a different binary operation ($\otimes$) and the summation is replaced by a different reduction operation ($\oplus$). For the first edge proposition ($k = 0$), a reduction-by-key algorithm finds the n maximum edge weights in each row i of matrix A', which is shown in Table 1 for vertex 4, $n = 2$ and A' in CSR format. The corresponding accumulator type saves n sorted pairs of a value and its corresponding column index j. In the beginning of the example the first value-column pair $(0.2, 3)$ is the initial value of the accumulator, and pairs with larger coefficients are inserted into the accumulator in subsequent steps to the right. With vertex charging enabled, the coefficients of the same charge as the current matrix row are ignored, which results in a proposition of vertex 4 to vertices 9 and 7 because these edges are of maximum weight and different charge. Without charging, vertex 4 proposes to vertices 6 and 9.

In subsequent edge propositions ($k > 0$), a vertex is may only propose edges to vertices, which do not have n confirmed edges yet. That indirect lookup is expressed in sparse matrix-vector products $A'x$ by vector x, which saves the $n \cdot N$ confirmed edges. If a vertex j has already n confirmed edges, the result of the abstract multiplication operator $(A')_{i,j} \otimes x_j$ evaluates to zero, and the edge is ignored during edge proposition. When vertex charging is enabled, another indirect lookup in the abstract multiplication operator ensures that zero is returned if the vertices have the same charge. The charges are calculated before each edge proposition in Line 10 with a part of the MD5 algorithm, which was also used by Auer and Bisseling [16]. We summarize the memory requirements for the edge proposition in Table 2. Note that the edge weights of A' are also written if $n = 2$, such that the minimum edge weight of cycles can be identified in the subsequent Algorithm 3.

After the edge proposition, the parallel loop in Algorithm 2 Line 26 is executed by another kernel to remove non-mutually proposed edges.

4.2 From a $[0,2]$-Factor to a Linear Forest

We implemented a step-efficient bidirectional scan to obtain the path IDs and position of the vertices with $\log_2(N)$ kernel launches and a butterfly access pattern, which is visualized in Figure 2 for a linear forest. Instead of explicitly checking the condition in Algorithm 3 Line 5, the kernel is launched $\log_2(N)$ times, so even if all vertices reside in one path we obtain the correct result. Overall work is $N \log_2(N)$, whereas a work-efficient scan is $O(N)$.

If we reach the path's end, during the stride-q neighbor computation, we mark this by setting the stride-q neighbor to the negative 1-based index of the path's end ID. Let $q_{\max}$ be the last and largest stride of the algorithm. A positive stride-$q_{\max}$ neighbor after $\log_2(N)$ scan steps indicates that the vertex is part of a cycle because had it reached a path's end, it would be negative.

The first bidirectional scan of step (1) in Section 3.3 requires read/write buffers for the stride-q neighbors, the weakest edge weights, and the vertex IDs incident to the weakest edge. The second bidirectional scan for step (2) in Section 3.3 requires only read/write buffers for the stride-q neighbors and the path positions, thus requiring fewer buffers during execution.

Each buffer mentioned above is allocated twice as an input and output buffer and used in a ping-pong fashion. Otherwise, other threads might read a value of a neighboring vertex during the scan execution after the update for that vertex has already overwritten the original input value in memory.

The above algorithms could not have been implemented with other scan operators of GPU libraries like Thrust [19] or CUB [26] as these are restricted to random access iterators. Even on CPUs, all parallel implementations of reduction or scan always assume random access iterators, cf. parallel STL in C++17. Our novel parallel scan implementation does not have this restriction; bidirectional connectivity suffices. This is even weaker than the concept of a bidirectional iterator, which includes global orientation information of what is forward and backward. We only have bidirectional connectivity, not knowing which neighbor is forward and which is backward along the path and still compute the scan in parallel.

4.3 Linear Forest Extraction

To obtain a permutation Q of A such that the edge weights, which are part of a linear forest of A are located in the tridiagonal part of

	READ			WRITTEN		
	label	length	type	label	length	type
for $k = 0$	CSR values	nnz	value	proposed edges	nN	index
	CSR col indices	nnz	index	proposed edge weights	nN	value
	CSR row ptrs	$N + 1$	index			
	vertex charges	N	bool			
additional data for $k > 0$	confirmed edges	nN	index			

Table 2: Read and written buffers in global GPU memory for the implementation of the edge proposition, which is expressed as generalized sparse matrix-vector product.

MATRIX	symmetric	N	nnz	$\overline{\Delta(G)}$
AF_SHELL8	y	504 855	17 588 875	34.84
ANISO1	y	6 250 000	56 220 004	9.00
ANISO2	y	6 250 000	56 220 004	9.00
ANISO3	y	6 250 000	56 220 004	9.00
ATMOSMODD	n	1 270 432	8 814 880	6.94
ATMOSMODJ	n	1 270 432	8 814 880	6.94
ATMOSMODL	n	1 489 752	10 319 760	6.93
ATMOSMODM	n	1 489 752	10 319 760	6.93
BUMP_2911	y	2 911 419	127 729 899	43.87
CUBE_COUP_DT0	y	2 164 760	127 206 144	58.76
CURLCURL_3	y	1 219 574	13 544 618	11.11
CURLCURL_4	y	2 380 515	26 515 867	11.14
ECOLOGY1	y	1 000 000	4 996 000	5.00
ECOLOGY2	y	999 999	4 995 991	5.00
G3_CIRCUIT	y	1 585 478	7 660 826	4.83
GEO_1438	y	1 437 960	63 156 690	43.92
HOOK_1498	y	1 498 023	60 917 445	40.67
LONG_COUP_DT0	y	1 470 152	87 088 992	59.24
ML_GEER	n	1 504 002	110 879 972	73.72
STOCF-1465	y	1 465 137	21 005 389	14.34
THERMAL2	y	1 228 045	8 580 313	6.99
TRANSPORT	n	1 602 111	23 500 731	14.67

Table 3: Pattern symmetric test matrices which are from the Sparse Matrix Collection [8] or from [21] (ANISO{1,2,3}). $\overline{\Delta(G)}$ denotes the mean degree of the graph G.

$Q^T AQ$, the vertex IDs are sorted with a radix sort from CUB [25] with respect to their key composed of path ID and position. The coefficients of the tridiagonal system are taken from the original input matrix A by converting A into a COO format and assigning one GPU thread to one coefficient of A. With the vector of the confirmed edges, each thread checks if the edge is part of the linear forest and scatters its value with the permutation into the tridiagonal system, which is saved in three buffers of length N.

5 RESULTS

For the results presented in this paper, we use a machine with CentOS 7, CUDA Toolkit 11.4.48, CUDA driver 470.74, GCC 10.2.0, a GeForce 2080 Ti and an Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz. If not mentioned explicitly, the experiments were done in single-precision as the RTX 2080 Ti only has a few double-precision units. For double-precision performance, professional accelerator cards (V100, A100) can be used. Additionally to the test matrices from the Sparse Matrix Collection [8], which are listed in Table 3 we use three 2D anisotropic problems (ANISO1,2,3) from [21], which represent an equidistant grid with the stencils:

$$\underset{\text{ANISO1}}{\begin{pmatrix} -0.2 & -0.1 & -0.2 \\ -1.0 & 3.0 & -1.0 \\ -0.2 & -0.1 & -0.2 \end{pmatrix}}, \quad \underset{\text{ANISO2}}{\begin{pmatrix} -0.1 & -0.2 & -1.0 \\ -0.2 & 3.0 & -0.2 \\ -1.0 & -0.2 & -0.1 \end{pmatrix}},$$

and matrix ANISO3 is obtained by permuting ANISO2, such that the coefficients with value -1.0 are located on the sub- and superdiagonal of A.

5.1 Weight Coverage Results

Table 4 shows the weight coverage results for the parallel $[0, 2]$-factor computation with Algorithm 2 in comparison to the greedy sequential Algorithm 1. When A' is not symmetric, the $[0, n]$-factor computations use $A' + A'^T$, but the weight coverage results are calculated with respect to the original matrix A. For the parallel $[0, 2]$-factor computation, we use three different *configurations*:

(1) $m = 1,\ k_m = 0$: no vertex charging enabled $\forall k$.
(2) $m = 5,\ k_m = 0$: no charging on iterations $k = 0, 5, 10, \ldots$.
(3) $m = 5,\ k_m = 1$: no charging on iterations $k = 1, 6, 11, \ldots$.

All configurations use $p = 0.5$, which is the rounded optimal value for graph matching determined by Auer and Bisseling [16]. For configuration (1), the weight coverage of the maximal $[0, 2]$-factor exceeds the results of the sequential algorithm for matrices ECOLOGY1,2, ATMOSMODD, and ATMOSMODJ (Table 4). However, the number of iterations to reach a maximal $[0, 2]$-factor is often high and the weight coverage increase per iteration is very little for the same matrices, which is indicated by low values of the weight coverage after five iterations $c_\pi(5)$. Configurations (2) and (3) perform much better on these matrices. The comparison between them points out the possible limitations of c_π if vertex charging is applied in the first iteration, e.g., for matrices STOCF-1465, G3_CIRCUIT, and LONG_COUP_DT0.The same limiting effect is also observable for $n = 1, 3, 4$. Therefore, for all following results, we utilize configuration (2) with $M = 5$ as the default configuration because it results in the same weight coverage as the sequential $[0, 2]$-factor algorithm in most cases.

In Table 5 the weight coverages for $n = 1, 2, 3, 4$ are shown with the previously chosen default parameters in comparison to the sequential results. With the maximum difference between the parallel and sequential result of 0.04 for $n = 1$ on matrix ATMOSMODM, the parallel algorithm reaches almost the same weight coverage as the sequential algorithm. Additionally, the table contains the weight coverage of the sub- and superdiagonal c_{id} given by Equation 5.

	PARALLEL ALG. 2									SEQUENTIAL ALG. 1
	no charging $\forall k$			no charging on $k = 0, 5, 10, \ldots$			no charging on $k = 1, 6, 11, \ldots$			
MATRIX	$c_\pi(5)$	$c_\pi(M_{\max})$	$M_{\max}$	$c_\pi(5)$	$c_\pi(M_{\max})$	$M_{\max}$	$c_\pi(5)$	$c_\pi(M_{\max})$	$M_{\max}$	c_π
AF_SHELL8	0.20	0.24	195	0.23	0.23	16	0.22	0.22	17	0.23
ANISO1	0.67	0.67	1252	0.67	0.67	11	0.54	0.54	17	0.67
ANISO2	0.67	0.67	1251	0.67	0.67	11	0.57	0.57	12	0.67
ANISO3	0.67	0.67	55	0.67	0.67	11	0.56	0.56	17	0.67
ATMOSMODD	0.02	0.47	164	0.41	0.42	16	0.42	0.42	17	0.44
ATMOSMODJ	0.02	0.47	164	0.41	0.42	16	0.42	0.42	17	0.44
ATMOSMODL	0.48	0.49	297	0.49	0.49	16	0.43	0.43	12	0.49
ATMOSMODM	0.95	0.95	297	0.95	0.95	16	0.74	0.74	12	0.95
BUMP_2911	0.81	0.82	31	0.81	0.82	26	0.64	0.64	27	0.82
CUBE_COUP_DT0	0.26	0.26	102	0.26	0.26	21	0.22	0.22	22	0.26
CURLCURL_3	0.34	0.34	47	0.34	0.34	16	0.36	0.36	12	0.34
CURLCURL_4	0.33	0.34	47	0.33	0.33	16	0.35	0.35	12	0.34
ECOLOGY1	0.00	0.50	1037	0.46	0.47	16	0.46	0.47	17	0.47
ECOLOGY2	0.00	0.50	1038	0.46	0.47	16	0.46	0.47	17	0.47
G3_CIRCUIT	0.56	0.71	159	0.70	0.70	16	0.59	0.59	17	0.70
GEO_1438	0.28	0.28	18	0.28	0.28	16	0.25	0.25	17	0.28
HOOK_1498	0.22	0.22	11	0.22	0.22	16	0.20	0.20	17	0.22
LONG_COUP_DT0	0.70	0.70	110	0.69	0.69	31	0.55	0.55	27	0.70
ML_GEER	0.20	0.20	383	0.20	0.20	11	0.17	0.17	17	0.20
STOCF-1465	1.00	1.00	11	1.00	1.00	16	0.78	0.78	17	1.00
THERMAL2	0.47	0.47	7	0.47	0.47	16	0.44	0.44	12	0.47
TRANSPORT	0.24	0.49	290	0.45	0.45	16	0.44	0.44	17	0.47

Table 4: $[0, 2]$-factor computation on the undirected graph of the given matrix and their relative weight coverage (Eq. 4) after five iterations $M = 5$, $c_\pi(5)$, and the maximal $[0, 2]$-factor $c_\pi(M_{\max})$, which is reached after $M_{\max}$ iterations.

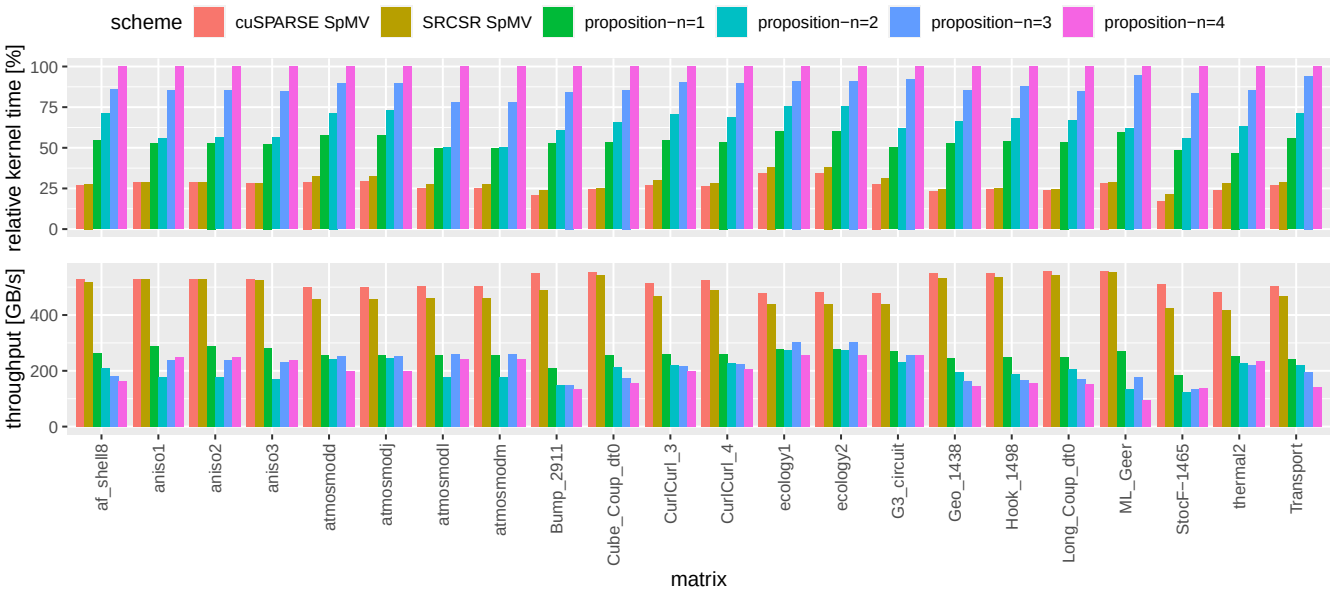


Figure 3: Performance results for one kernel execution of edge proposition according to Algorithm 2, Lines 14-22 with $k > 0$, $m = 1$, $k_m = 0$ and different n in comparison to SpMV algorithms. Except cuSPARSE SpMV, all schemes use our generalized sparse matrix-vector implementation.

Comparing c_{id} with $c_\pi(5)$ for $n = 2$ allows an estimation of the weight of an algebraically extracted tridiagonal system with the tridiagonal part of A in the original vertex order, e.g., the tridiagonal part of ATMOSMODD already contains strong coefficients, whereas the algebraically extracted $[0, 2]$-factor of ATMOSMODM

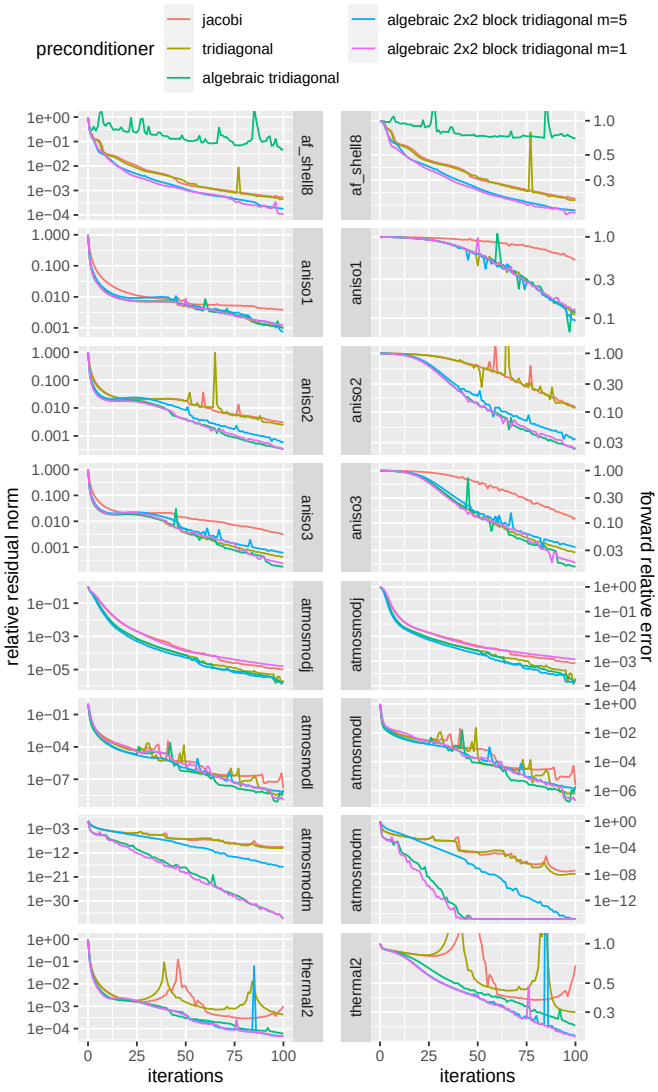


Figure 4: Double-precision BiCGStab convergence results with our algebraically constructed scalar and 2x2 block tridiagonal preconditioner in comparison to a Jacobi and tridiagonal preconditioner based on the original vertex ordering.

holds a much larger weight than just the tridiagonal part of the matrix in the original order.

5.2 Performance Results

5.2.1 Edge Proposition of Parallel $[0, n]$*-Factor Computation.* We use our generalized sparse matrix-vector implementation for the parallel edge proposition of Algorithm 2, Line 14 and compare the performance with the normal cuSPARSE SpMV and our segmented reduction (SRCSR) SpMV implementation, which both calculate $d = Ax + d$ for a CSR matrix. The implementation of the parallel edge proposition and the SRCSR SpMV schemes only differ by data types and functors, and use the same generic sparse matrix-vector API. The time of the SpMV setup kernels for cuSPARSE and our generalized sparse matrix-vector implementation is not included in the time measurement. Although the setup of cuSPARSE's SpMV is not explicitly exposed, a binary search kernel is executed prior to the actual sparse-matrix vector calculating kernel, which is visible with the profiler.

The average runtimes and throughputs were measured with NVIDIA Nsight Compute and are shown in Figure 3. Due to significantly different matrix sizes, the upper plot shows the times relative to the longest kernel. For the normal sparse matrix-vector product ($d = Ax + d$), our general SRCSR code has similar performance to the specialized cuSPARSE assembly optimized code. Therefore, SRCSR has an efficient implementation, despite its generality which cuSPARSE does not have. The kernel (Algorithm 2, Line 14) is executed by SRCSR with appropriate lambda and type parameterization, but the lambdas contain a lot of complex code (70 lines). So the resulting SRCSR code after lambda inlining by the compiler

- does much more work than a normal SpMV,
- has more instruction flow divergence (if-statements) than a normal SpMV,
- uses more registers than a normal SpMV,
- uses more shared memory than a normal SpMV,
- uses more input and output vectors (more DRAM traffic) than a normal SpMV (see Table 2).

Therefore, we cannot expect Algorithm 2 to run at the speed of a normal SpMV. The performance of the normal SpMV serves as roofline in the comparison because it solves a simpler problem but on the same matrix structure. Achieving 30-50% of this roofline with a more demanding and irregular algorithm (Alg. 2), proves high efficiency of the implementation.

We evaluated alternative implementations for $[0,n]$-factor computation in which the SRCSR kernel contains less work and has a similar runtime to normal SpMV but this requires more substeps and other additional kernels. Although the additional kernels run at full bandwidth the approach presented here, which is limited by the reduction operation along the rows, has the shortest overall runtime.

Other implementations to find the columns of the n maximal values within each matrix row with CUB's [25] segmented reduction or segmented sort are approximately one order of magnitude slower for $2 \leq n \leq 4$.

5.2.2 Bidirectional Scan to Extract the Linear Forest. The linear forest extraction consists of the identification of the cycles, the path identification, and the extraction of the coefficients (see Section 3.3). The first two steps are implemented with our bidirectional scan that is executed $\log_2(N)$ times for each step. Top of Figure 5 shows the throughput statistics of the bidirectional scans as a boxplot, which reveals worse throughput for some kernel executions due to the expected irregular global memory accesses when visiting the stride-q vertex neighbors. However, in most cases, the median of the throughputs is close to the performance of a simple copy kernel. The comparison of the averaged total running times of the sequential CPU version with the parallel GPU version in the lower part of Figure 5 reports speedups from factor 4x to 24x. Contrary to the parallel GPU algorithm, the sequential version performs far

	c_{id}	$c_\pi(5)$								2x2 block tridiagonal	
		$n=1$		$n=2$		$n=3$		$n=4$			
MATRIX		PAR	SEQ	PAR	SEQ	PAR	SEQ	PAR	SEQ	$m=1$	$m=5$
AF_SHELL8	0.01	0.14	0.14	0.23	0.23	0.34	0.34	0.40	0.40	0.38	0.43
ANISO1	0.68	0.27	0.29	0.67	0.67	0.72	0.73	0.79	0.79	0.68	0.64
ANISO2	0.13	0.27	0.29	0.67	0.67	0.72	0.73	0.79	0.79	0.68	0.64
ANISO3	0.68	0.27	0.29	0.67	0.67	0.72	0.73	0.79	0.79	0.68	0.64
ATMOSMODD	0.46	0.19	0.21	0.41	0.44	0.65	0.67	0.93	0.93	0.02	0.50
ATMOSMODJ	0.46	0.19	0.21	0.41	0.44	0.65	0.67	0.93	0.93	0.02	0.50
ATMOSMODL	0.25	0.21	0.22	0.49	0.49	0.60	0.61	0.73	0.73	0.41	0.45
ATMOSMODM	0.03	0.38	0.42	0.95	0.95	0.96	0.96	0.97	0.97	0.94	0.86
BUMP_2911	0.01	0.46	0.49	0.81	0.82	0.84	0.84	0.86	0.86	0.84	0.83
CUBE_COUP_DT0	0.06	0.11	0.13	0.26	0.26	0.33	0.34	0.38	0.38	0.29	0.29
CURLCURL_3	0.15	0.17	0.17	0.34	0.34	0.54	0.55	0.76	0.76	0.44	0.54
CURLCURL_4	0.15	0.17	0.17	0.33	0.34	0.53	0.54	0.74	0.74	0.40	0.53
ECOLOGY1	0.50	0.21	0.23	0.46	0.47	0.71	0.71	1.00	1.00	0.00	0.55
ECOLOGY2	0.50	0.21	0.23	0.46	0.47	0.71	0.71	1.00	1.00	0.00	0.55
G3_CIRCUIT	0.29	0.50	0.51	0.70	0.70	0.83	0.84	1.00	1.00	0.61	0.73
GEO_1438	0.04	0.13	0.14	0.28	0.28	0.36	0.37	0.44	0.44	0.33	0.33
HOOK_1498	0.04	0.11	0.11	0.22	0.22	0.28	0.28	0.33	0.33	0.25	0.25
LONG_COUP_DT0	0.10	0.49	0.50	0.69	0.70	0.79	0.79	0.87	0.87	0.84	0.83
ML_GEER	0.05	0.09	0.09	0.20	0.20	0.25	0.26	0.32	0.32	0.23	0.26
STOCF-1465	0.23	0.92	0.93	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
THERMAL2	0.10	0.23	0.24	0.47	0.47	0.68	0.68	0.84	0.84	0.58	0.58
TRANSPORT	0.49	0.20	0.22	0.45	0.47	0.68	0.70	0.98	0.98	0.25	0.53

Table 5: $[0,n]$-factors of underlying undirected graphs and their relative weight coverages (Eq. 4) after five iterations $M = 5$, $c_\pi(5)$, $k_m = 0$, $m = 5$ of the parallel (PAR) Algorithm 2, in comparison to the results of the sequential (SEQ) Algorithm 1. The coverage of the sub- and superdiagonal in the original ordering is shown by c_{id} (Eq. 5). The weight coverage of the algebraically constructed 2x2 block tridiagonal preconditioner is shown in the right part of the table (see Section 6).

less work: it creates the permutation while the vertices are visited without an explicit sorting.

We cannot compare against well-known parallel scan implementations from Thrust [19] or CUB [26] because of their random access iterator requirement, the end of Section 4.2 has more details.

6 APPLICATION

As an exemplary application, we show the setup of an algebraic scalar (AlgTriScalPrecond) and 2x2 block tridiagonal preconditioner (AlgTriBlockPrecond). The $[0,2]$-factor computation and the extraction of a linear forest are used to setup AlgTriScalPrecond, which is shown in Figure 6 by the time breakdown of the complete setup. Algorithm 2 for the $[0,2]$-factor computation and the bidirectional scans (including Algorithm 3) consumes most of the time, whereas the actual coefficient extraction only requires at most 10% of the setup time.

AlgTriBlockPrecond is constructed by a $[0,1]$-factor and a subsequent $[0,2]$-factor computation. With the $[0,1]$-factor, the graph is coarsened, such that the matched pairs represent a single vertex in the coarser graph. On that coarse graph, the $[0,2]$-factor is computed, resulting in a 2x2 block tridiagonal system on the fine graph. For vertices without a match in the $[0,1]$-factor, we add an uncoupled ghost equation by setting the diagonal and right-hand side value in the corresponding additional row to one. Otherwise an irregular 2x2/1x1 block tridiagonal system would have to be processed. The weight coverage of AlgTriBlockPrecond is shown in the right part of Table 5 for $m = 1, 5$, which is used for both factor computations. For matrices ANISO1,2,3 and ATMOSMODM, $m = 1$ (no vertex charging $\forall k$) results in a higher weight coverage, whereas for matrices AF_SHELL8 and ECOLOGY1,2, $m = 5$ (no vertex charging on $k = 0, 5, 10, \dots$) is better. Hence we conclude that the parameters for a $[0,n]$-factor computation and for recursive $[0,n]$-factor computations on the coarser graphs, must be chosen differently to maximize the weight coverage but automatic parameter control in nested factor computations is beyond the scope of this paper.

To evaluate the convergence of our new preconditioners, we compare it with a Jacobi and a tridiagonal preconditioner (TriScalPrecond) which are constructed based on the original vertex ordering and use a BiCGStab as the outer Krylov solver [34]. Obviously, there are many more choices of preconditioners. The compared preconditioners here all have a tridiagonal matrix structure and take a similar time to solution. Comparing against preconditioners of completely different complexity (e.g. ILU) would require many plots on the tradeoff between numerical vs. computational efficiency and a detailed discussion of suitability with respect to matrix types.

The implementation of the outer solver and the Jacobi preconditioner is taken from the MAGMA [1] library. The right-hand side is constructed from a generated solution with $x_t[i] := \sin(16\pi i/N)$. With the true solution being known, we calculate the forward relative error as $\text{FRE} := |x - x_t|_2/|x_t|_2$, where x is the computed solution. The convergence results with the relative residual norm and the forward relative error are shown in Figure 4. Here, all results are in double-precision as we want to highlight the convergence improvement of the new preconditioners. In single-precision the

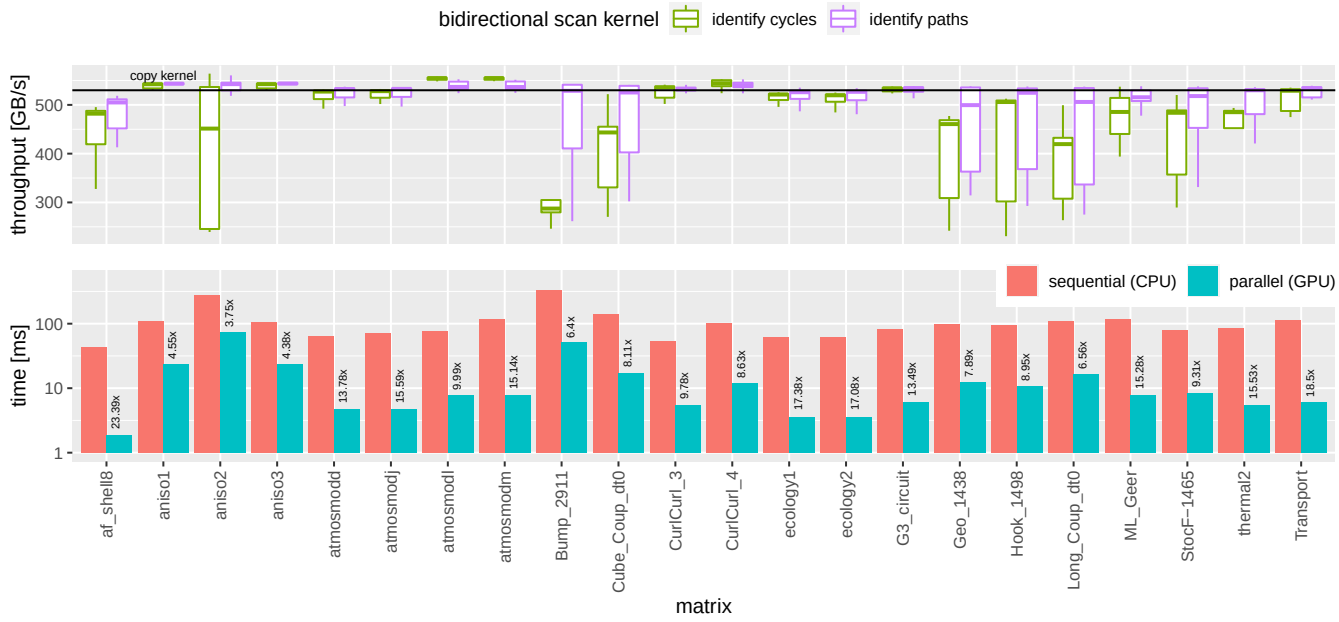


Figure 5: Memory throughput statistics (top) for both bidirectional scan kernels and total runtimes (bottom) including steps (1), (2), (3) from Section 3.3. The speedup of the parallel version is written above the bars.

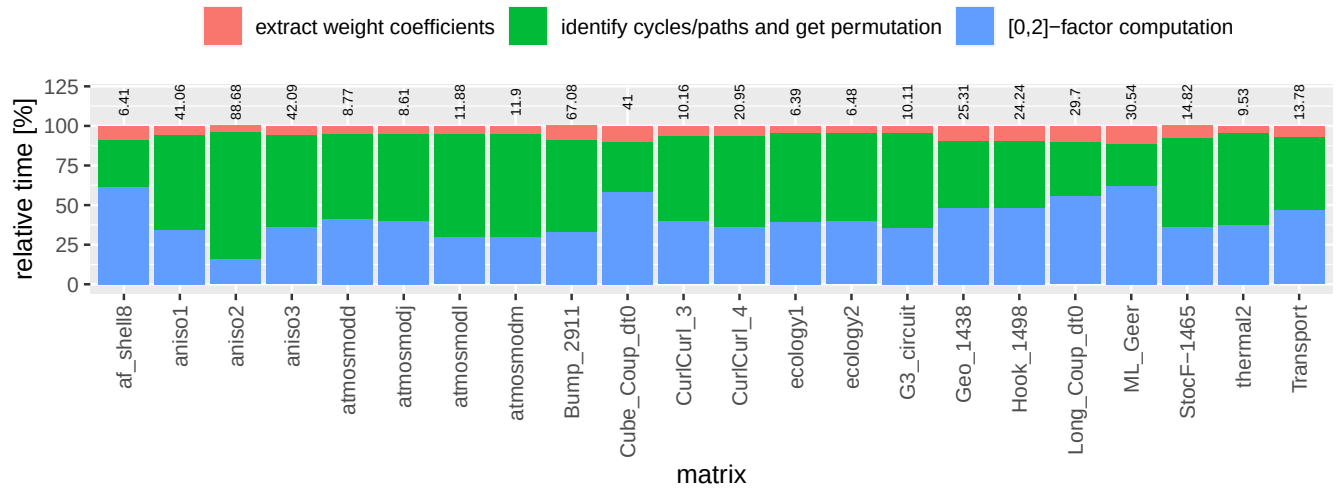


Figure 6: Time breakdown for $[0,2]$-factor computation (Algorithm 2 with $M = 5, m = 5, k_m = 0, n = 2$), and the extraction of the linear forest (Section 3.3) to algebraically construct a tridiagonal preconditioner. The total absolute running time in milliseconds is written above the bars.

tridiagonal solves execute at the bandwidth limit of the GPU [21], for double-precision performance professional accelerator cards (V100, A100) would be required. The improvement of the algebraic tridiagonal preconditioners for matrix ANISO2 is expected, as they include the strong coefficients from the diagonal of the stencil, which were permuted manually to the sub- and superdiagonal in ANISO3. For matrices ATMOSMODJ, ATMOSMODL, and ATMOSMODM, the $[0,2]$-factor contains increasingly more weight relative to the original vertex ordering c_{id}, which is visible in Table 5. The convergence improvement for matrix ATMOSMODM is strongest, as the algebraic preconditioners have a weight coverage of up to 95%, whereas TriScalPrecond has a weight coverage of only 3%. This example also shows the coupling between convergence rate and weight coverage of AlgTriBlockPrecond, which has either a weight

coverage of 94% for $m = 1$ and performs as well as the AlgTriScalPrecond ($c_\pi(5) = 0.95$), or a weight coverage of 86% for $m = 5$ and performs worse. For matrix AF_SHELL8, the TriScalPrecond with a weight coverage of 1% includes approximately the same coefficients as the Jacobi preconditioner. The AlgTriScalPrecond with $c_\pi(5) = 0.23$ includes not enough off-diagonal coefficients to obtain a stable convergence behaviour, which is achieved by AlgTriBlockPrecond with a weight coverage of 38% or 43%. In summary, in most cases, we see significant benefits of algebraically creating the tridiagonal system (AlgTriScalPrecond) rather than relying on the tridiagonal part of the matrix in the original ordering. The block version (AlgTriBlockPrecond) performs consistently even better.

7 CONCLUSION

We have shown how to compute [0,n]-factors efficiently in parallel with our generalized sparse matrix-vector product and how our new bidirectional scan, which does not require a random access iterator, identifies cycles, paths and positions in a $[0, 2]$-factor to create a linear forest. Benchmarks on large graphs demonstrate our parallel algorithms' high weight coverage at high speed. In the application to linear equation systems the high coverage results in superior convergence of the algebraically constructed scalar and 2x2 block tridiagonal preconditioners.

REFERENCES

[1] Hartwig Anzt, William Sawyer, Stanimire Tomov, Piotr Luszczek, Ichitaro Yamazaki, and Jack Dongarra. 2014. Optimizing Krylov Subspace Solvers on Graphics Processing Units. In *Fourth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), IPDPS 2014*. IEEE, IEEE, Phoenix, AZ.

[2] Sébastien Bougleux and Luc Brun. 2016. Linear Sum Assignment with Edition. 1 (2016), 1–27. arXiv:1603.04380 http://arxiv.org/abs/1603.04380

[3] R.E. Burkard. 1980. The Linear Sum Assignment Problem. (1980).

[4] Chu-Yi Huang Yen-Shen Chen and Youn-Long Lin Yu-Chin Hsu. 1990. Data Path Allocation Based on Bipartite Weighted Matching'. (1990).

[5] Zhi Zhong Chen. 1996. Fundamental Study Parallel constructions of maximal path sets and applications to short superstrings. *Theoretical Computer Science* 161, 1-2 (1996), 1–21. https://doi.org/10.1016/0304-3975(95)00110-7

[6] Jonathan Cohen and Patrice Castonguay. 2012. Efficient graph matching and coloring on the gpu. In *GPU Technology Conference*. 1–10.

[7] Ketan Date and Rakesh Nagi. 2016. GPU-accelerated Hungarian algorithms for the Linear Assignment Problem. *Parallel Comput.* 57 (2016), 52–72. https://doi.org/10.1016/j.parco.2016.05.012

[8] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1 (2011). https://doi.org/10.1145/2049662.2049663

[9] Adrian Perez Dieguez, Margarita Amor, and Ramon Doallo. 2016. New Tridiagonal Systems Solvers on GPU Architectures. *Proceedings - 22nd IEEE International Conference on High Performance Computing, HiPC 2015* (2016), 85–93. https://doi.org/10.1109/HiPC.2015.17

[10] Reinhard Diestel. 2017. *Graph Theory* (5th ed.). Springer-Verlag, Heidelberg.

[11] I. S. Duff. 1981. Algorithm 575: Permutations for a Zero-Free Diagonal [F1]. *ACM Transactions on Mathematical Software (TOMS)* 7, 3 (1981), 387–390. https://doi.org/10.1145/355958.355968

[12] I. S. Duff. 1981. On Algorithms for Obtaining a Maximum Transversal. *ACM Transactions on Mathematical Software (TOMS)* 7, 3 (1981), 315–330. https://doi.org/10.1145/355958.355963

[13] Iain S. Duff, Kamer Kaya, and Bora Uçar. 2011. Design, implementation, and analysis of maximum transversal algorithms. *ACM Trans. Math. Software* 38, 2 (2011). https://doi.org/10.1145/2049673.2049677

[14] Iain S. Duff and Jacko Koster. 1999. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.* 20, 4 (1999), 889–901. https://doi.org/10.1137/S0895479897317661

[15] I. S. Duff and J. Koster. 2001. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.* 22, 4 (2001), 973–996. https://doi.org/10.1137/S0895479899358443

[16] Bas O. Fagginger Auer and Rob H. Bisseling. 2012. A GPU Algorithm for Greedy Graph Matching. (2012), 108–119. https://doi.org/10.1007/978-3-642-30397-5_10

[17] Stefan Guthe and Daniel Thuerck. 2021. Algorithm 1015 A Fast Scalable Solver for the Dense Linear (Sum) Assignment Problem. *ACM Trans. Math. Software* 47, 2 (2021). https://doi.org/10.1145/3442348

[18] Michael Hagemann and Olaf Schenk. 2006. Weighted matchings for preconditioning symmetric indefinite linear systems. 28, 2 (2006), 403–420.

[19] Jared Hoberock and Nathan Bell. 2010. Thrust: A Parallel Template Library. http://thrust.github.io/

[20] Jonathan Hogg and Jennifer Scott. 2015. On the use of suboptimal matchings for scaling and ordering sparse symmetric matrices. *Numerical Linear Algebra with Applications* 22, 4 (aug 2015), 648–663. https://doi.org/10.1002/nla.1978 arXiv:arXiv:1112.5346v3

[21] Christoph Klein and Robert Strzodka. 2021. Tridiagonal GPU Solver with Scaled Partial Pivoting at Maximum Bandwidth. In *50th International Conference on Parallel Processing*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3472456.3472484

[22] Artur Mariano, Alberto Proenca, and Cristiano Da Silva Sousa. 2015. A generic and highly efficient parallel variant of boruvka's algorithm. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 610–617.

[23] Timothy G. Mattson, Carl Yang, Scott McMillan, Aydin Buluc, and Jose E. Moreira. 2017. GraphBLAS C API: Ideas for future versions of the specification. *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017* (2017), 1–6. https://doi.org/10.1109/HPEC.2017.8091095

[24] D.J. Mavriplis. 1998. Multigrid Strategies for Viscous Flow Solvers on Anisotropic Unstructured Meshes. *J. Comput. Phys.* 145, 1 (sep 1998), 141–165. https://doi.org/10.1006/jcph.1998.6036

[25] Duane Merrill. 2021. CUB. https://nvlabs.github.io/cub

[26] Duane Merrill and Michael Garland. 2016. Single-pass Parallel Prefix Scan with Decoupled Look-back. *NVIDIA Technical Report NVR-2016-002* (2016), 1–9. http://research.nvidia.com/sites/default/files/publications/nvr-2016-002.pdf

[27] Md Naim, Fredrik Manne, Mahantesh Halappanavar, Antonino Tumeo, and Johannes Langguth. 2016. Optimizing Approximate Weighted Matching on NVIDIA Kepler K40. *Proceedings - 22nd IEEE International Conference on High Performance Computing, HiPC 2015* (2016), 105–114. https://doi.org/10.1109/HiPC.2015.15

[28] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka. 2015. AMGX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing* 37, 5 (jan 2015), S602–S626. https://doi.org/10.1137/140980260

[29] Hannu Peltola, Hans Söderlund, Jorma Tarhio, and Esko Ukkonen. 1983. Algorithms for Some String Matching Problems Arising in Molecular Genetics.. In *IFIP Congress Series*, Vol. 9. 59–64.

[30] Bobby Philip and Timothy P. Chartier. 2012. Adaptive algebraic smoothers. *J. Comput. Appl. Math.* 236, 9 (2012), 2277–2297. https://doi.org/10.1016/j.cam.2011.11.016

[31] Michael D. Plummer. 2007. Graph factors and factorization: 1985-2003: A survey. *Discrete Mathematics* 307, 7-8 (2007), 791–821. https://doi.org/10.1016/j.disc.2005.11.059

[32] Scott Rostrup, Shweta Srivastava, and Kishore Singhal. 2011. Fast and memory efficient minimum spanning tree on the GPU. In *Proceedings of the 2nd Intl. Workshop on GPUs and Scientific Applications (GPUScA, 2011). Held in conjunction with PACT*. 3–13.

[33] Roberto Roverso, Amgad Naiem, Mohammed El-Beltagy, Sameh El-Ansary, and Seif Haridi. 2010. A GPU-enabled solver for time-constrained linear sum assignment problems. *INFOS2010 - 2010 7th International Conference on Informatics and Systems* (2010).

[34] Yousef Saad. 2003. Iterative Methods for Sparse Linear Systems. *Iterative Methods for Sparse Linear Systems* (2003). https://doi.org/10.1137/1.9780898718003

[35] Takayoshi Shoudai and Satoru Miyano. 1995. Using maximal independent sets to solve problems in parallel. *Theoretical Computer Science* 148, 1 (aug 1995), 57–65. https://doi.org/10.1016/0304-3975(94)00221-4

[36] Ryuhei Uehara and Zhi-Zhong Chen. 1997. Parallel Algorithms for Maximal Linear Forests.

[37] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and PJ Narayanan. 2009. Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics 2009*. 167–171.

[38] Guoliang Xing, Chenyang Lu, Ying Zhang, Qingfeng Huang, and Robert Pless. 2007. Minimum power configuration for wireless communication in sensor networks. *ACM Transactions on Sensor Networks* 3, 2 (2007), 1–33. https://doi.org/10.1145/1240226.1240231