TR-2015-12

ANALYSIS OF A SPLITTING APPROACH FOR THE PARALLEL SOLUTION OF LINEAR SYSTEMS ON GPU CARDS

Ang Li, Radu Serban, Dan Negrut

October 26, 2015

Contents

Abstract

We discuss an approach for solving sparse or dense banded linear systems $\mathbf{Ax} = \mathbf{b}$ on a Graphics Processing Unit (GPU) card. The matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ is possibly nonsymmetric and moderately large; i.e., $10\,000 \leq N \leq 500\,000$. The *split and parallelize* (`SaP`) approach seeks to partition the matrix $\mathbf{A}$ into diagonal sub-blocks $\mathbf{A}_i$, $i = 1, \ldots, P$, which are independently factored in parallel. The solution may choose to consider or to ignore the matrices that couple the diagonal sub-blocks $\mathbf{A}_i$. This approach, along with the Krylov subspace-based iterative method that it preconditions, are implemented in a solver called `SaP::GPU`, which is compared in terms of efficiency with three commonly used sparse direct solvers: `PARDISO`, `SuperLU`, and `MUMPS`. `SaP::GPU`, which runs entirely on the GPU except several stages involved in preliminary row-column permutations, is robust and compares well in terms of efficiency with the aforementioned direct solvers. In a comparison against Intel's `MKL`, `SaP::GPU` also fares well when used to solve dense banded systems that are close to being diagonally dominant. `SaP::GPU` is publicly available and distributed as open source under a permissive BSD3 license.

1 Introduction

Previously used in niche applications and by a small group of enthusiasts, general purpose computing on graphics processing unit (GPU) cards has gained widespread popularity after the release in 2007 of the CUDA programming environment [35]. Owing also to the release of the OpenCL specification [40] in 2008, GPU computing has been rapidly adopted by numerous groups with computing needs originating in a broad spectrum of application areas. In several of these areas though, when compared to the library ecosystem enabling sequential and/or parallel computing on x86 chips, GPU computing library support continues to be spotty. This observation motivated an effort whose outcomes are reported in this paper, which is concerned with solving sparse linear systems of equations on the GPU.

Developing an approach and implementing parallel code for solving sparse linear systems is not trivial. This, and the relative novelty of GPU comput-

[†]Electrical and Computer Engineering, University of Wisconsin–Madison, Madison, WI 53706

[‡]Mechanical Engineering, University of Wisconsin–Madison, Madison, WI 53706

ing explain the scarcity of solutions for solving $\mathbf{Ax} = \mathbf{b}$ on the GPU, when $\mathbf{A} \in \mathbb{R}^{N \times N}$ is possibly nonsymmetric, sparse, and moderately large; i.e., $10\,000 \leq N \leq 500\,000$. An inventory of software solutions as of 2015 produced a short list of codes that solved $\mathbf{Ax} = \mathbf{b}$ on the GPU: `cuSOLVER` [7], `Paralution` [1], and `SuperLU` [16], the latter focused on distributed memory architectures and leveraging GPU computing at the node level only. Several CPU multi-core approaches exist and are well established, see for instance [4, 43, 8, 16]. For a domain-specific application implemented on the GPU that calls for solving $\mathbf{Ax} = \mathbf{b}$, one alternative would be to fall back on one of these CPU-based solutions. This strategy usually impacts the overall performance of the algorithm due to the back-and-forth data movement across the PCI host–device interconnect, which in practice supports bandwidths of the order of 10 GB/s. Herein, the focus is not on this strategy. Instead, we are interested in carrying out the LU factorization on the GPU when the possibly nonsymmetric matrix $\mathbf{A}$ is sparse or dense banded with narrow bandwidth.

There are pros and cons to having a linear solver on the GPU. On the upside, since a parallel implementation of a LU factorization is memory bound, particularly for sparse systems, the GPU is attractive owing to its high bandwidths and relatively low latencies. At main-memory bandwidths of roughly 300 GB/s, the GPU is four to five times faster than a modern multicore CPU. On the downside, the irregular memory access patterns associated with sparse matrix factorization ablate this GPU-over-CPU advantage, which is further eroded by the intense logic and integer arithmetic requirements associated with existing algorithms. The approach discussed herein alleviates these two pitfalls by embracing a splitting strategy described for CPU-centric multicore and/or multi-node computing in [38]. Two successive row–column permutations attempt to increase the diagonal dominance of the matrix and reduce its bandwidth, respectively. Ideally, the reordered matrix would be (i) diagonal dominant, and (ii) dense banded. If (i) is accomplished, no LU factorization row/column pivoting is necessary, thus avoiding tasks at which the GPU does not shine: logic and arithmetic operations. Additionally, if (ii) holds, coalesced memory access patterns associated with dense matrix operations can capitalize on the GPU's high bandwidth.

The overall solution strategy adopted herein solves $\mathbf{Ax} = \mathbf{b}$ using a Krylov-subspace method and employs LU preconditioning with work-splitting and drop-off. Specifically, each outer Krylov-subspace iteration takes at least one preconditioner solve step that involves solving $\hat{\mathbf{A}}\mathbf{y} = \hat{\mathbf{b}}$ on the GPU,

where $\hat{\mathbf{A}} \in \mathbb{R}^{N \times N}$ is a *dense* banded matrix obtained from $\mathbf{A}$ after a sequence of possibly two reordering stages that can include element drop-off. Regardless of whether $\mathbf{A}$ is sparse or not, the salient attribute of the approach is the casting of the preconditioning step as a *dense* linear algebra problem. Thus, a reordering process is employed to obtain a narrow–band, dense $\hat{\mathbf{A}}$, which is subsequently LU–factored. For the reordering, a strategy that combines two stages, namely diagonal dominance boosting and bandwidth reduction, has yielded well balanced coefficient matrices that can be factored fast on the GPU leveraging a single instruction multiple data (SIMD)–friendly underlying data structure. The LU factorization relies on a splitting of the matrix $\hat{\mathbf{A}}$ in several diagonal blocks that are factored independently and a correction process to account for the inter-diagonal block coupling. The implementation takes advantage of the GPU's deep memory hierarchy, its multi-SM layout, and its predilection for SIMD computation.

This paper is organized as follows. Section 2 summarizes the solution algorithm. The discussion covers first the work-splitting-based LU factorization of dense banded matrices. Subsequently, the $\mathbf{Ax} = \mathbf{b}$ sparse case brings into focus strategies for matrix reordering. Section 3 summarizes aspects related to the GPU implementation of the solution approaches proposed. Results of a series of numerical experiments for both dense banded and sparse linear systems are reported in Section 4. Since reordering strategies play a pivotal role in the sparse linear system solution, we present benchmarking results in which we compared the reordering strategies adopted herein to established solutions/implementations. The paper concludes with a series of final remarks and a summary of lessons learned and directions of future work.

2 Description of the methodology

2.1 The dense banded linear system case

Assume that the banded dense matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ has half-bandwidth $K \ll N$. Following an approach discussed in [42, 38, 39], we partition the banded matrix $\mathbf{A}$ into a block tridiagonal form with P diagonal blocks $\mathbf{A}_i \in \mathbb{R}^{N_i \times N_i}$, where $\sum_i^P N_i = N$. For each partition i, let $\mathbf{B}_i$, $i = 1, \ldots, P-1$ and $\mathbf{C}_i$, $i = 2, \ldots, P$ be the super- and sub-diagonal coupling blocks, respectively – see Figure 1. Each coupling block has dimension $K \times K$ for banded matrices

with half-bandwidth $K = \max_{i,j,a_{ij}\neq 0} |i-j|$.

As illustrated in Fig. 1, the banded matrix $\mathbf{A}$ is expressed as the product of a block diagonal matrix $\mathbf{D}$ and a so-called *spike matrix* $\mathbf{S}$ [42]. The latter is made up of identity diagonal blocks of dimension N_i, and off-diagonal spike blocks, each having K columns. Specifically,

$$\mathbf{A} = \mathbf{D}\mathbf{S}\,, \tag{1}$$

where $\mathbf{D} = \mathrm{diag}(\mathbf{A}_1, \ldots, \mathbf{A}_P)$ and, assuming that $\mathbf{A}_i$ are non-singular, the so-called left and right spikes $\mathbf{W}_i$ and $\mathbf{V}_i$ associated with partition j, each of dimension $N_i \times K$, are given by

$$\mathbf{A}_1\mathbf{V}_1 = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{B}_1 \end{bmatrix} \tag{2a}$$

$$\mathbf{A}_i\left[\mathbf{W}_i \mid \mathbf{V}_i\right] = \begin{bmatrix} \mathbf{C}_i & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_i \end{bmatrix}\,, \quad i = 2, \ldots, P-1 \tag{2b}$$

$$\mathbf{A}_P\mathbf{W}_P = \begin{bmatrix} \mathbf{C}_P \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}\,. \tag{2c}$$

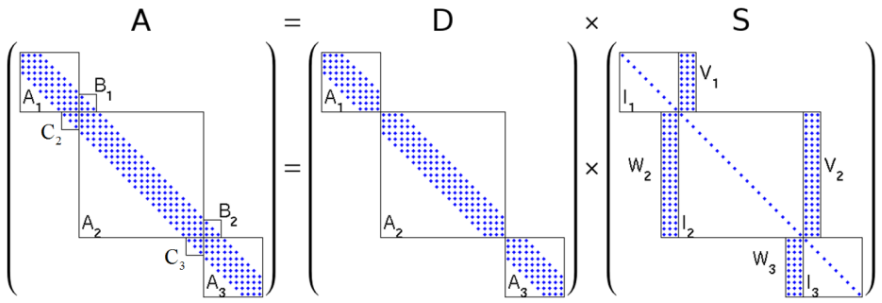

Figure 1: Factorization of the matrix $\mathbf{A}$ with $P = 3$.

Solving the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ is thus reduced to solving

$$\mathbf{D}\mathbf{g} = \mathbf{b} \tag{3}$$

$$\mathbf{S}\mathbf{x} = \mathbf{g} \tag{4}$$

Since $\mathbf{D}$ is block-diagonal, solving for the modified right-hand side $\mathbf{g}$ from (3) is trivially parallelizable, as the work is split across P processes, each charted to solve $\mathbf{A}_i\mathbf{g}_i = \mathbf{b}_i$, $i = 1, \ldots, P$. Note that the same decoupling is manifest in Eq. (2), and the work is spread over P processes.

The remaining question is how to solve quickly the linear system in (4). This problem can be reduced to one of smaller size, $\hat{\mathbf{S}}\hat{\mathbf{x}} = \hat{\mathbf{g}}$. To that end, the spikes $\mathbf{V}_i$ and $\mathbf{W}_i$, as well as the modified right-hand side $\mathbf{g}_i$ and the unknown vectors $\mathbf{x}_i$ in (4) are partitioned into their top K rows, the middle $N_i - 2K$ rows, and the bottom K rows:

$$\mathbf{V}_i = \begin{bmatrix} \mathbf{V}_i^{(t)} \\ \mathbf{V}_i' \\ \mathbf{V}_i^{(b)} \end{bmatrix}, \quad \mathbf{W}_i = \begin{bmatrix} \mathbf{W}_i^{(t)} \\ \mathbf{W}_i' \\ \mathbf{W}_i^{(b)} \end{bmatrix}, \tag{5a}$$

$$\mathbf{g}_i = \begin{bmatrix} \mathbf{g}_i^{(t)} \\ \mathbf{g}_i' \\ \mathbf{g}_i^{(b)} \end{bmatrix}, \qquad \mathbf{x}_i = \begin{bmatrix} \mathbf{x}_i^{(t)} \\ \mathbf{x}_i' \\ \mathbf{x}_i^{(b)} \end{bmatrix}. \tag{5b}$$

A block-tridiagonal reduced system is obtained by excluding the middle partitions of the spike matrices as:

$$\begin{bmatrix} \mathbf{R}_1 & \mathbf{M}_1 & & & \\ & \ddots & & & \\ & \mathbf{N}_i & \mathbf{R}_i & \mathbf{M}_i & \\ & & & \ddots & \\ & & & \mathbf{N}_{P-1} & \mathbf{R}_{P-1} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{x}}_1 \\ \vdots \\ \hat{\mathbf{x}}_i \\ \vdots \\ \hat{\mathbf{x}}_{P-1} \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{g}}_1 \\ \vdots \\ \hat{\mathbf{g}}_i \\ \vdots \\ \hat{\mathbf{g}}_{P-1} \end{bmatrix}, \tag{6}$$

where the linear system above, denoted $\hat{\mathbf{S}}\hat{\mathbf{x}} = \hat{\mathbf{g}}$, is of dimension $2K(P-1) \ll N$,

$$\mathbf{N}_i = \begin{bmatrix} \mathbf{W}_i^{(b)} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \qquad i = 2, \ldots, P-1 \tag{7a}$$

$$\mathbf{R}_i = \begin{bmatrix} \mathbf{I}_M & \mathbf{V}_i^{(b)} \\ \mathbf{W}_{i+1}^{(t)} & \mathbf{I}_M \end{bmatrix}, \quad i = 1, \ldots, P-1 \tag{7b}$$

$$\mathbf{M}_i = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_{k+1}^{(t)} \end{bmatrix}, \qquad i = 1, \ldots, P-2 \tag{7c}$$

and

$$\hat{\mathbf{x}}_i = \begin{bmatrix} \mathbf{x}_i^{(b)} \\ \mathbf{x}_{i+1}^{(t)} \end{bmatrix}, \; \hat{\mathbf{g}}_i = \begin{bmatrix} \mathbf{g}_i^{(b)} \\ \mathbf{g}_{i+1}^{(t)} \end{bmatrix}, \quad i = 1, \ldots, P-1 \; . \tag{8}$$

Two strategies are proposed in [38] to solve (6): (i) an exact reduction; and, (ii) an approximate reduction, which sets $\mathbf{N}_i \equiv \mathbf{0}$ and $\mathbf{M}_i \equiv \mathbf{0}$ and results in a block diagonal matrix $\hat{\mathbf{S}}$. The solution approach adopted herein is based on (ii) and therefore each sub-system $\mathbf{R}_i \hat{\mathbf{x}}_i = \hat{\mathbf{g}}_i$ is solved independently using the following steps:

$$\text{Form } \bar{\mathbf{R}}_i = \mathbf{I}_M - \mathbf{W}_{i+1}^{(t)} \mathbf{V}_i^{(b)} \tag{9a}$$

$$\text{Solve } \bar{\mathbf{R}}_i \tilde{\mathbf{x}}_{i+1}^{(t)} = \mathbf{g}_{i+1}^{(t)} - \mathbf{W}_{i+1}^{(t)} \mathbf{g}_i^{(b)} \tag{9b}$$

$$\text{Calculate } \tilde{\mathbf{x}}_i^{(b)} = \mathbf{g}_i^{(b)} - \mathbf{V}_i^{(b)} \tilde{\mathbf{x}}_{i+1}^{(t)} \tag{9c}$$

Note that a tilde was used to differentiate between the actual and approximate values $\tilde{\mathbf{x}}_i^{(t)}$ and $\tilde{\mathbf{x}}_i^{(b)}$ obtained upon dropping the $\mathbf{N}_i$ and $\mathbf{M}_i$ terms. An approximation of the solution of the original problem is finally obtained by solving independently and in parallel P systems using the available LU factorizations of the $\mathbf{A}_i$ matrices:

$$\mathbf{A}_1 \mathbf{x}_1 = \mathbf{b}_1 - \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{B}_1 \tilde{\mathbf{x}}_2^{(t)} \end{bmatrix} \tag{10a}$$

$$\mathbf{A}_i \mathbf{x}_i = \mathbf{b}_i - \begin{bmatrix} \mathbf{C}_i \tilde{\mathbf{x}}_{i-1}^{(b)} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} - \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{B}_i \tilde{\mathbf{x}}_{i+1}^{(t)} \end{bmatrix}, \quad i = 2, \ldots, P-1 \tag{10b}$$

$$\mathbf{A}_P \mathbf{x}_P = \mathbf{b}_P - \begin{bmatrix} \mathbf{C}_P \tilde{\mathbf{x}}_{P-1}^{(b)} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} . \tag{10c}$$

Computational savings can be made by noting that if an LU factorization of the diagonal blocks $\mathbf{A}_i$ is available, the bottom block of the right spike; i.e. $\mathbf{V}_i^{(b)}$, can be obtained from (2a) using only the bottom $K \times K$ blocks of L and U. However, obtaining the top block of the left spike requires calculating the entire spike $\mathbf{W}_i$. An effective alternative is to perform an additional UL factorization of $\mathbf{A}_i$, in which case $\mathbf{W}_i^{(t)}$ can be obtained using only the top $K \times K$ blocks of the new U and L.

Next, note that the decision to set $\mathbf{N}_i \equiv \mathbf{0}$ and $\mathbf{M}_i \equiv \mathbf{0}$ relegates the resulting algorithm to preconditioner status. Embracing this path is justified by the following observation that although the dimension of the reduced linear system in (6) is smaller that that of the original problem, its half-bandwidth is at least three times larger. The memory footprint of exactly solving (6) is large, thus limiting the size of problems that can be tackled on the GPU. Specifically, at each recursive step, additional memory that is required to store the new reduced matrix cannot be deallocated until the global solution is fully recovered.

Finally, it becomes apparent that the quality of the preconditioner is correlated to neglecting the $\mathbf{N}_i$ and $\mathbf{M}_i$ terms. For the sake of this discussion, assume that the matrix $\mathbf{A}$ is diagonally dominant with a degree of diagonal dominance $d \geq 1$; i.e.,

$$|a_{ii}| \geq d \sum_{j \neq i} |a_{ij}| \ , \forall i = 1, \ldots, N \ . \tag{11}$$

When $d > 1$, the elements of the left spikes $\mathbf{W}_i$ decay in magnitude from top to bottom, while those of the right spikes $\mathbf{V}_i$ decay from bottom to top [33]. This decay, which is more pronounced the larger the degree of diagonal dominance of $\mathbf{A}$, justifies the approximation $\mathbf{N}_i \equiv \mathbf{0}$ and $\mathbf{M}_i \equiv \mathbf{0}$. However, note that having $\mathbf{A}$ be diagonal dominant, although desirable, it is not a prerequisite as demonstrated by numerical experiments reported herein. Truncating when $d < 1$ will lead to a preconditioner of lesser quality.

2.1.1 Nomenclature, solution strategies

Targeted for execution on the GPU, the methodology outlined above becomes the foundation of a parallel implementation called herein "split and parallelize" (SaP). The matrix $\mathbf{A}$ is split into block diagonal matrices $\mathbf{A}_i$, which are processed in parallel. The code implementing this strategy is called `SaP::GPU`. Several flavors of `SaP::GPU` can be envisioned. At one end of the spectrum, one solution path would implement the exact reduction, a strategy that is not considered herein. At the other end of the spectrum, `SaP::GPU` solves the block-diagonal linear system in 3 and for preconditioning purposes uses the approximation $\mathbf{x} \approx \mathbf{g}$. In what follows, this will be called the decoupled approach, `SaP::GPU-D`. The middle ground is the approximate reduction, which sets $\mathbf{N}_i \equiv \mathbf{0}$ and $\mathbf{M}_i \equiv \mathbf{0}$. This will be called the

coupled approach, SaP::GPU-C, owing to the coupling that occurs through the truncated spikes; i.e., $\mathbf{V}_i^{(b)}$ and $\mathbf{W}_{i+1}^{(t)}$.

Neither the coupled nor the decoupled paths qualify as direct solvers and SaP::GPU employs an outer Krylov subspace scheme to solve $\mathbf{Ax} = \mathbf{b}$. The solver uses BiCGStab(ℓ) [46] and left-preconditioning, unless the matrix $\mathbf{A}$ is symmetric and positive definite, in which case the outer loop implements a conjugate gradient method [41]. SaP::GPU is open source and available at [2, 3].

2.2 The sparse linear system case

The discussion focuses next on solving $\mathbf{A}_s\mathbf{x} = \mathbf{b}$, where $\mathbf{A}_s \in \mathbb{R}^{N\times N}$ is assumed to be a sparse matrix. The salient attribute of the solution strategy is its fallback on the dense banded approach described in §2.1. Specifically, an aggressive row and column permutation process is employed to transform $\mathbf{A}_s$ into a matrix $\mathbf{A}$ that has a large d and small K. Although the reordered matrix will remain sparse within the band, it will be regarded to be dense banded and LU- and/or UL-factored accordingly. For matrices $\mathbf{A}_s$ that are either nonsymmetric or have low d, a first set of row permutations is applied as $\mathbf{QA}_s\mathbf{x} = \mathbf{Qb}$, to either maximize the number of nonzeros on the diagonal (maximum traversal search) [19], or maximize the product of the absolute values of the diagonal entries [20, 21]. Both reordering algorithms are implemented using a depth first search with a look-ahead technique similar to the one in the Harwell Software Library (HSL) [4].

While the purpose of the first reordering $\mathbf{QA}_s$ is to render the permuted matrix diagonally "heavy", a second reordering seeks to reduce K by using the traditional Cuthill-McKee CM algorithm [14]. Since the diagonal entries should not be relocated, the second permutation is applied to the symmetric matrix $\mathbf{QA}_s + \mathbf{A}_s^T\mathbf{Q}^T$. Following these two reorderings, the resulting matrix $\mathbf{A}$ is split to obtain $\mathbf{A}_1$ through $\mathbf{A}_P$. A third CM reordering is then applied to each $\mathbf{A}_i$ for further reduction of bandwidth. While straightforward to implement in SaP::GPU-D, this third stage reordering in SaP::GPU-C mandates computation of the entire spikes, an operation that can significantly increase the memory footprint and flop count of the numerical solution. Note that third stage reordering in SaP::GPU-C renders the UL factorization superfluous since computing only the top of a spike is insufficient.

If $\mathbf{A}_i$ is diagonally dominant, the LU and/or UL factorization can be safely carried out without pivoting [24]. Adopting the strategy used in PARDISO

[44], we always perform factorizations of the diagonal blocks $\mathbf{A}_i$ *without* pivoting but with *pivot boosting*. Specifically, if a pivot becomes smaller than a threshold value, it is boosted to a small, user controlled value ϵ. This yields a factorization of a slightly perturbed diagonal block, $\mathbf{L}_i\mathbf{U}_i = \mathbf{A}_i + \delta\mathbf{A}_i$, where $\|\delta\mathbf{A}_i\| = \mathcal{O}(u\|\mathbf{A}\|)$ and u is the unit roundoff [32].

2.2.1 Brief comments on the reordering algorithms

`SaP::GPU` employs two reordering strategies, namely Diagonal Boosting (`DB`) and Cuthill-McKee (`CM`), possibly multiple times, to reduce K and increase the degree of diagonal dominance. `DB` is applied first at the matrix $\mathbf{A}_s$ level, followed by `CM` applied at matrix level, and possibly followed by a set of P third-stage `CM` reorderings applied at the sub-matrix $\mathbf{A}_i$ level.

Diagonal Boosting. The `DB` algorithm seeks to improve diagonal dominance in $\mathbf{A}_s$ and draws on a minimum bipartite perfect matching [12, 28, 11, 13, 17, 26]. There are several variants of the algorithm aimed at different outcomes, e.g., maximizing the absolute value of bottleneck, the sum, the product or other metrics that factor in the diagonal entries. As a proxy for diagonal dominance, `SaP::GPU` maximizes the absolute value of the product of all diagonal entries.

The algorithm that seeks to leverage GPU computing is as follows. Given a matrix $\{a_{ij}\}_{n\times n}$, find a permutation σ that maximizes $\prod_{i=1}^{n} |a_{i\sigma_i}|$. Denoting $a_i = \max_j |a_{ij}|$ and noting that a_i is an invariant of σ, then we are to minimize

$$\log \prod_{i=1}^{n} \frac{a_i}{|a_{i\sigma_i}|} = \sum_{i=1}^{n} \log \frac{a_i}{|a_{i\sigma_i}|} = \sum_{i=1}^{n} (\log a_i - \log |a_{i\sigma_i}|) \,.$$

The reordering problem is reduced to minimum bipartite perfect matching in the following way: given a bipartite graph $G_C = (V_R, V_C, E)$, we define the weight c_{ij} of the edge between nodes $i \in V_R$ and $j \in V_C$ as

$$c_{ij} = \begin{cases} \log a_i - \log |a_{ij}| & (a_{ij} \neq 0) \\ \infty & (a_{ij} = 0) \end{cases} . \tag{12}$$

If we are able to find a minimum bipartite perfect matching σ such that $\sum c_{i\sigma_i}$ is minimized, according to the process of reduction above, then $\prod_{i=1}^{n} |a_{i\sigma_i}|$ is maximized.

Bandwidth reduction. Whether $\mathbf{QA}_s$ is sparse or not, there are $P-1$ pairs of always *dense* spikes, each of dimension $N_i \times K$. They need to

be stored unless one employs an LU and UL factorization of $\mathbf{A}_i$ to retain only the appropriate bottom and top components. Large K values pose memory challenges; i.e., storing and data movement, that limit the size of the problems that can be tackled. Moreover, the spikes need to be computed by solving multiple right-hand side linear systems with $\mathbf{A}_i$ coefficient matrices. There are $2K$ such systems for each of the $P-1$ pairs of spikes. Evidently, a low K is highly desirable. However, finding the lowest half-bandwidth K by symmetrically reordering a sparse matrix is NP-hard. The `CM` reordering provides simple and oftentimes effective heuristics to tackle this problem. Moreover, as the `CM` reordering yields symmetric permutations, it will not displace the "heavy" diagonal terms obtained during the `DB` step. However, to obtain a symmetric permutation, one has to start with a symmetric matrix. To this end, unless $\mathbf{A}$ is already symmetric and does not call for a `DB` step (which is the case, for instance, when $\mathbf{A}$ is symmetric positive definite), the matrix passed over for `CM` reordering is $(\mathbf{A}+\mathbf{A}^T)/2$. Given a symmetric $n \times n$ matrix with m non-zero entries `CM` works on its adjacency matrix. `CM` first picks a random node and adds the node to the work list. Then the algorithm repeats sorting all its neighboring nodes with non-descending vertex degree and adding them until all vertices have been added and removed once from the work list. In other words, `CM` is essentially a `BFS` where neighboring vertices are visited in order from lowest to highest vertex degree.

Third-stage reordering. The `DB–CM` reordering sequence yields diagonally-heavy matrices of smaller bandwidth. The band itself however can be very sparse. The purpose of the third-stage `CM` reordering is to further reduce the bandwidth within each $\mathbf{A}_i$ and reduce the sparsity within the band. Consider, for instance, the matrix ANCF88950 that comes from structural dynamics [45]. It has 513 900 nonzeros, $N = 88\,950$, and an average of 5.78 non-zero elements per row. After `DB–CM` reordering with no drop-off, the resulting banded matrix has a half-bandwidth $K = 205$. The band itself is very sparse with a fill-in of only 0.7% within the band. In its default solution, `SaP::GPU` constructs a block banded matrix where each diagonal block $\mathbf{A}_i$, obtained after the initial `DB–CM` reorderings, is allowed to have a different bandwidth. This is achieved using another `CM` pass, independently and in parallel for each $\mathbf{A}_i$. Applying this strategy to ANCF88950, using $P = 16$ partitions, the half bandwidth is reduced for all partitions to values no higher than $K = 141$, while the fill-in within the band becomes approximately 3%.

Note that this third-stage reordering does nothing to reduce the column-width of the spikes. However, it helps in two respects: a smaller memory

footprint for the LU/UL factors, and less factorization effort. These are important side effects, since the LU/UL GPU factorization is currently done in-core considering $\mathbf{A}_i$ to be *dense* within the band.

3 Brief implementation details

3.1 Dense banded matrix factorization details

This subsection provides implementation details regarding how the P partitions $\mathbf{A}_i$ are determined, how the banded matrix $\mathbf{A}$ is stored, and how the LU/UL steps are implemented on the GPU.

Number of partitions and partition size. The selection of P must strike a balance between two conflicting requirements. On the one hand, having a large P is attractive given that the LU/UL factorization of $\mathbf{A}_i$ for $i = 1, \ldots, P$ can be done independently and simultaneously. On the other hand, this negatively impacts the quality of the resulting preconditioner, due to the approximations in evaluating the spikes corresponding to the coupling of the diagonal blocks $\mathbf{A}_i$ and $\mathbf{A}_{i+1}$. Since this adversely impacts the quality of the resulting preconditioner, a high P could lead to poor preconditioning and an increase in the number of iterations to convergence. In the current implementation, no attempt is made to automate this selection and some experimentation is required.

Given a P value, the size of the diagonal blocks $\mathbf{A}_i$ is selected to achieve load balancing. The first P_r partitions are of size $\lfloor N/P \rfloor + 1$, while the remaining are of size $\lfloor N/P \rfloor$, where $N = P\lfloor N/P \rfloor + P_r$.

Matrix storage. For general dense banded matrices $\mathbf{A}_i$, we adopt a "tall and thin" storage in column-major order. All diagonal elements are stored in the K-th column. The rest of the elements are correspondingly distributed columnwise. This strategy, shown below for a matrix with $N = 8$ and $K = 2$, groups the operands of the LU/UL factorizations and allows coalesced

memory accesses that can fully leverage the GPU's bandwidth.

$$\begin{bmatrix} * & * & a_{11} & a_{21} & a_{31} \\ * & a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} & a_{53} \\ a_{24} & a_{34} & a_{44} & a_{54} & a_{64} \\ a_{35} & a_{45} & a_{55} & a_{65} & a_{75} \\ a_{46} & a_{56} & a_{66} & a_{76} & a_{86} \\ a_{57} & a_{67} & a_{77} & a_{87} & * \\ a_{68} & a_{78} & a_{88} & * & * \end{bmatrix}$$

LU/UL factorizations. The solution strategy pursued calls for an LU and an optional UL factorization of each dense banded diagonal block $\mathbf{A}_i$. The implementation requires a certain level of synchronization since for each $\mathbf{A}_i$, the factorization, forward elimination, and backward substitution phases each consist of $N_i - 1$ dependent steps that need to be choreographed. One aggravating factor is the GPU lack of native, low overhead, support for synchronization between threads running in different blocks. The established GPU strategy for inter-block synchronization is "exit and launch a new kernel". This guarantees synchronization at the GPU-grid level at the cost of non-negligible overhead. In a trade-off between minimizing the overhead of kernel launches and maximizing the occupancy of the GPU, we established two execution paths: one for $K < 64$, the second one for larger bandwidths. As a side note, the threshold value of 64 was selected through numerical experimentation over a variety of problems and is controlled by the number of threads that can be organized in a block in CUDA [35].

For $K < 64$, the code was designed to reduce the kernel launch count. Instead of having $N_i - 1$ kernel launches, each completing a step of the factorization of $\mathbf{A}_i = \mathbf{L}_i\mathbf{U}_i$ by updating entries in a $(K+1)\times(K+1)$ window of elements, a single kernel is launched to factor $\mathbf{A}_i$. It uses $\min(K^2, 1024)$ threads per block and relies on low-overhead stream-multiprocessor synchronization support *within* the block, without any need for global synchronization. In a so-called *window-sliding* method, at each step of the factorization; i.e., during the process of computing column entries in L and row entries of U, each thread updates a fixed number of $\mathbf{A}_i$ entries. On current GPU hardware, this fixed number is between 1 and 4. Once all threads in the block complete their work, they are synchronized and the $(K + 1) \times (K + 1)$ window slides down by one row and to the right by one column. The value 4 is explained

as follows. Assume that $K = 63$. Then, the sliding window has size 64×64. Since the two-dimensional GPU thread block size is $1024 = 32 \times 32$, each thread will handle four entries of the window of focus.

For $K \geq 64$, `SaP` uses multiple blocks of threads to update L and U entries. On the upside, there are more threads working on the window of focus. On the downside, there is overhead associated with leaving and reentering the kernel, a process that has the side effect of flushing the shared memory and registers. The window is larger than $K \times K$, and it slides at a stride of eight; i.e., moves down by eight rows and to the right by eight columns upon exiting and reentering the LU factorization kernel.

Use of registers and shared memory. If the user decides to employ a third-stage reordering, the coupling sub-blocks $\mathbf{B}_i$ and $\mathbf{C}_i$ are used to compute the entire spikes in a scheme that renders a UL factorization superfluous. Then, $\mathbf{B}_i$ and $\mathbf{C}_i$ are each first partitioned into sub-blocks of dimension $L \times K$ where L is at most 20. Each forward/backward sweep to get the spikes is unrolled, and in each iteration of the new loop, one entire sub-block, rather than a vector of length K, is calculated. To this end, the corresponding elements in the matrix $\mathbf{A}_i$ are pre-fetched into shared memory and the entries of the sub-block are preloaded into registers. This strategy, in which all operations to calculate the spikes draw on registers and shared memory, leads to 50% to 70% improvement in performance when compared with an alternative that calculates the spike elements in a loop without leveraging the low latency/high bandwidth of the GPU register file and shared memory.

Mixed Precision Strategy. The solution uses a mixed-precision implementation by falling back on single precision for the preconditioner and switching to double precision arithmetic in the outer `BiCGStab(2)` calculations. A battery of tests indicate that this strategy results in a 50% average reduction in time to solution when compared with an approach where all calculations are performed in double precision.

3.2 `DB` reordering implementation details

`SaP::GPU` organizes the `DB` algorithm into four stages, `DB-S1` through `DB-S4`. Due to differences in the nature and degree of parallelism of these stages, `DB` implements a hybrid strategy; namely, it relies on GPU computing for `DB-S1` and `DB-S4` and on CPU computing for `DB-S2` and `DB-S3`. A thorough discussion of the implementation is provided in [31]. Therein, a solution that kept the entire `DB` implementation on the GPU was discussed

and deemed decisively slower than the hybrid strategy adopted here.

`DB-S1`**: form bipartite graph.** This stage assembles a matrix that mirrors the structure of the original sparse matrix. The sparsity pattern of the input matrix is maintained and the values of its nonzero entries are modified according to Eq. (12). The stage is highly parallel and involves: (1) calculating for each row of the original matrix the max absolute value, and (2) updating each value to form the weighted bipartite graph.

`DB-S2`**: find initial partial match.** This stage is not mandatory but the availability of an initial partial match as a starting point for the next stage was found to considerably reduce the running time for the overall algorithm [31]. Like in [12], after setting $u_i = \min_j c_{ij}$ and $v_j = \min_i(c_{ij} - u_i)$, we try to match as many pairs of nodes as possible. The matched nodes (i, j) should satisfy $u_i + v_j = c_{ij}$. This yields augmenting paths of length one. This stage, which was implemented to execute in parallel, was compute intensive as it had to resolve scenarios where multiple column nodes would match the same row node. A CPU parallel implementation was found to be more suitable owing to intense integer arithmetic and control flow overhead.

`DB-S3`**: find perfect match.** Finding matches in a bipartite graph G_C is equivalent to finding the shortest paths in an associated reduced graph. Omitting some of the details, the shortest path problem is tackled using Dijkstra's algorithm [18], which is applied to all nodes i that are unmatched in the initial partial match obtained in `DB-S2`. This ensures that all row nodes, and therefore all column nodes, are eventually matched. The theoretical complexity of this stage is $O(n \cdot (m+n) \cdot \log n)$, where n and m are the dimension and number of nonzeros in the input matrix, respectively. However, thanks to the preprocessing `DB-S2`, actual run times for finding a perfect match are acceptable in all situations and this stage is the `DB` bottleneck only for about half of the matrices tested [31].

`DB-S4`**: extract permutation and scaling factors.** The matrix permutation can be obtained directly from the resulting perfect match: if the row node i was matched to the column node j then rows (or columns) i and j must be permuted. Optionally, scaling factors can be calculated and applied to rows and columns in order to bring the matrix to a so-called I-matrix form; i.e., a matrix with 1 or -1 on the diagonal and off-diagonal elements of absolute value less than 1, see [36]. This stage is highly parallelizable and amenable to GPU computing.

3.3 CM reordering implementation details

The unordered CM algorithm, which draws on an approach described in [27], is separated into three stages, CM−S1 through CM−S3. A high quality reordering calls for several BFS iterations, which are called herein "CM iterations". Just like the DB implementation, the CM solution (*i*) is hybrid – the overall algorithm leverages both CPU and GPU computing; and, (*ii*) it uses CPU–GPU unified memory, a recent CUDA feature [34], to provide for a simple and transparent memory management process. The latter feature allows the CUDA runtime to transparently manage the CPU–GPU data migration as the computation switches back and forth between the CPU and GPU. Since no explicit, programmer initiated, data transfer is required, the code is cleaner and more concise.

CM−S1**: pre-processing.** The first stage is implemented on the GPU to accomplish two objectives. First, it produces the data structure that is worked upon. As the input matrix $\mathbf{A}$ is not guaranteed to be symmetric, the sparse matrix structure for $(\mathbf{A} + \mathbf{A}^T)/2$ is produced in anticipation of the subsequent two stages of the algorithm. Second, in order to avoid repetitively sorting the neighbors of a given node, the nodes with the same row indices are pre-sorted by ascending vertex degree of column index.

CM−S2**: perform standard BFS.** After experimenting with the implementation, the strategy adopted started from several nodes and in parallel performed what would be a traditional CM−S2 & CM−S3 combo. The alternative of considering one node only, namely the node with the smallest vertex degree, yields a second level BFS tree with fewer nodes. Eventually, the resulting BFS tree will likely be "tall and thin". Starting from several nodes and completing the reordering process for each of them increases the likelihood of avoiding a "bad" initial node. In practical terms, owing to the use of parallel computing, this strategy yields smaller bandwidths at a modest increase in computational overhead.

For each starting node, a standard BFS pass yields the levels of all nodes in the BFS tree. Since the order of nodes at the same level is not critical in this stage, parallel computing can help by concurrently visiting the neighbors of all nodes at the previous level. We use an outer loop to iterate over the levels, and in each iteration, depending on the number of nodes n_p added in the previous iteration, we decide whether this iteration is executed on the GPU or CPU. The heuristics used are as follows: a kernel handles the iteration on the GPU only if $n_p \geq 10$. There are two notable implementation details.

First, the CM iterations are executed sequentially. After each iteration, we select the node at the previous level with the lowest vertex degree which has not yet been selected yet. If no such nodes exist; i.e., all nodes at the last level have been selected as starting nodes in previous iterations, a random node which has not been considered is selected. Second, the CM iterations terminate either when the height of the BFS tree does not increase, or when the maximum number of nodes over all levels does not decrease compared with the candidate optimal found so far. This strategy is proposed in [37] with the caveat that we only consider the leaf with the minimum degree. From practical experience, these heuristics lead to an algorithm that for most matrices terminates within three CM iterations.

CM-S3**: reorder nodes.** The previous stage determines the level of each node. Roughly speaking, nodes are ordered in ascending order, from level 0 up to the maximum level m_l and memory space can be pre-allocated for nodes at each level. Parallel computing is leveraged by observing that the order of nodes at level l depends only on the order of nodes at level $l-1$. To that end, a pair of read/write pointers is set for each level, and except for level 0, the read/write pointers of each level will point to the starting position of the level's pre-allocated space. We say a thread "works on" level l if it reads nodes at level l and writes their neighbors that are at level $l+1$. Thus the execution thread working on level l will read and modify the read pointer of level l and the write pointer of level $l+1$, and it will only read the write pointer of level l. Once the thread finishes reading all nodes at level l, it moves on to another level; otherwise it repeats checking whether or not the thread working on level $l-1$ has written nodes which it has not processed by checking if the read pointer at level l lags the write pointer at level l. If yes, the thread working on level l processes these nodes, i.e., writes their neighbors with level $l+1$, and goes back to checking again whether it has finished processing or not; otherwise, it spins and waits for the thread working on the previous level. Note that the parallelism in CM-S3 is rather coarse-grained and proved to be better suited for execution on the CPU.

3.4 SaP::GPU–components and computational flow

In the absence of column/row reordering before the LU factorization and pivoting during the factorization, the SaP::GPU dense banded linear system solver is straightforward to implement. Upon partitioning **A** into diagonal

blocks $\mathbf{A}_i$, each $\mathbf{A}_i$ is subject to an LU factorization that requires an amount of time T_{LU}. Next, in T_{BC} time, the coupling block matrices $\mathbf{B}_i$ and $\mathbf{C}_i$ are extracted on the GPU. The $\mathbf{V}_i$ and $\mathbf{W}_i$ spikes are subsequently computed in an operation that requires T_{SPK} time. Afterwards, in T_{LUrdcd} time, the spikes are truncated and the steps outlined in Eq. (9) are taken to produce the intermediary values $\tilde{\mathbf{x}}_i^{(t)}$ and $\tilde{\mathbf{x}}_i^{(b)}$. At this point, the pre-processing step is over and two sets of factorizations, for $\mathbf{A}_i$ and $\bar{\mathbf{R}}_i$, are available for preconditioning during the iterative phase of the solution. The amount of time spent iterating is T_{Kry}, the iterative methods considered being `BiCGStab(2)` and conjugate gradient.

The sparse linear system solution is slightly more convoluted at the front end. A sequence of two permutations, `DB` requiring T_{DB} and `CM` requiring T_{CM} time, are carried out to increase the size of the diagonal elements and reduce bandwidth, respectively. An additional amount of time T_{Drop} might be spent to drop off-diagonal elements in order to decrease the bandwidth of the reordered $\mathbf{A}$ matrix. Since the `DB` and `CM` reorderings are hybrid, $T_{Dtransf}$ is used to keep track of the overhead associated with moving data back and forth between the CPU and GPU during the reordering process. An amount of time T_{Asmbl} is spent on the GPU in book-keeping required to turn the reordered sparse matrix into a dense banded matrix.

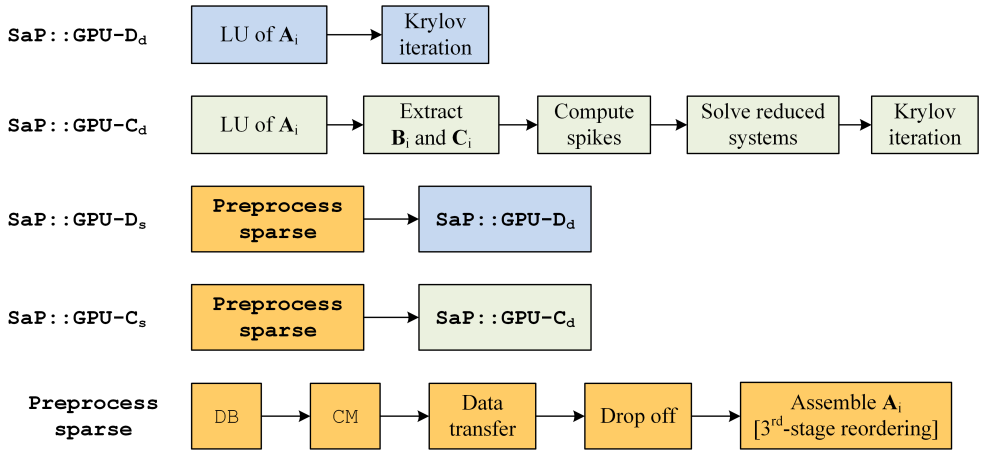

Figure 2: Computational flow for `SaP::GPU`.

The process described above is summarized in Fig. 2. The boxes in gray are associated with the solution of a dense banded linear system. For a sparse linear system solve that uses a coupled approach; i.e., `SaP::GPU-C`, the

total time is $T_{TotSparse} = T_{PrepSp} + T_{TotDense}$, where $T_{PrepSp} = T_{DB} + T_{CM} + T_{Dtransf} + T_{Drop} + T_{Asmbl}$ and $T_{TotDense} = T_{LU} + T_{BC} + T_{SPK} + T_{LUrdcd} + T_{Kry}$. For `SaP::GPU-D`, owing to the decoupled nature of the solution, $T_{TotDense} = T_{LU} + T_{Kry}$, where T_{LU} includes an `CM` process that reduces the bandwidth of each $\mathbf{A}_i$. The names introduced; i.e., T_{DB}, T_{CM}, T_{LUrdcd}, etc., are referenced in the profiling study discussed in §4.3.1 and used *ad verbum* on the `SaP::GPU` web-page [3] to report profiling results for approximately 120 linear systems.

4 Numerical Experiments

The next three subsections summarize results from three numerical experiments concerned, in this order, with the solution of dense banded linear systems, sparse matrix reordering, and the solution of sparse linear systems. The subsection order is meant to emphasize that dense banded linear system solution and matrix reordering are two prerequisites for an effective sparse linear system implementation in `SaP::GPU`. The hardware/software setup for these numerical experiments is as follows. The GPU used was Tesla K20X [6, 5]. `SaP::GPU` uses `CUDA 7.0` [35], `cusp` [9], and `Thrust` [25]. The CPU used was the 3GHz, 25 MB last level cache, Intel Xeon E5-2690v2. The node used hosted two such CPUs, which is the maximum possible for this type of chip, for a total of 20 cores executing up to 40 HTT threads. The two-CPU node was used to run Intel's MKL version 13.0.1, `PARDISO` [43], `MUMPS` [8], `SuperLU` [16], and Harwell's `MC60` and `MC64` [4]. Unless otherwise stated, all times reported are in seconds and were obtained on a dedicated machine. In an attempt to avoid warm up overhead, the results reported represent averages that drew on multiple successive identical runs.

When reporting below the results of several numerical experiments, one legitimate question is whether it makes sense to compare performance results obtained on one GPU with results obtained on two multicore CPUs. The multicore CPU is not the fastest, as Intel chips with more cores are presently available. Additionally, the Intel chip's microarchitecture is not Haswell, which is more recent than the Ivy Bridge microarchitecture of the Xeon E5-2690v2. Likewise, on the GPU side, one could have used a Tesla K80 card, which has roughly four times more memory than K20x and twice its memory bandwidth. Moreover, price-wise, the K80 would have been closer to the cost of two CPUs than K20x is. Finally, Kepler is not the latest

microarchitecture either, since Maxwell currently enjoys that status. We do not attempt to answer these questions and hope that the interested reader will modulate this study's conclusions by factoring in unavoidable CPU–GPU hardware differences. No claim is made herein of one architecture being superior since such a claim could be easily proved wrong by moving from algorithm to algorithm or from discipline to discipline. The sole and narrow purpose of this section is to report on how apt `SaP::GPU` is in tackling linear algebra tasks. To that end its performance is compared to that of established solutions running on CPUs and also of a recent GPU library.

4.1 Numerical experiments related to dense banded linear systems

The discussion in this subsection draws on a subset of results reported in [29] and presents results pertaining to the influence on `SaP`'s time to solution of the number of partitions P and of the diagonal dominance d of the coefficient matrix, as well as a comparison against Intel's MKL solver over a spectrum of problem dimensions N and half bandwidth values K.

4.1.1 Sensitivity with respect to P

The entire `SaP::GPU` solution for dense banded linear systems is implemented on the GPU. We first carried out a sensitivity analysis of the time to solution with respect to the number of partitions. The results are summarized in Fig. 3. This behavior; i.e., relatively small gains after a threshold value of P, is typical. As a rule of thumb, some experimentation is necessary to find an optimal P value. Otherwise, a conservatively large value should be picked in the neighborhood of 50 or above. For `SaP::GPU-D`, larger values of P help with load balancing, particularly for GPUs with many stream multiprocessors. The same argument can be made for `SaP::GPU-C`, with the caveat that the spike truncation factor comes into play in a fashion that is modulated by the value of d.

It is instructive to see how the solution time is spent by `SaP::GPU-C` and `SaP::GPU-D` and understand how changing P influences this distribution of the time to solution between the major implementation components. The results in Table 1 provide this information as they compare the coupled and decoupled strategies in regards to the factorization times, D_{pre} vs. C_{pre}; number of iterations in the Krylov solver, D_{it} vs. C_{it}; amount of time spent

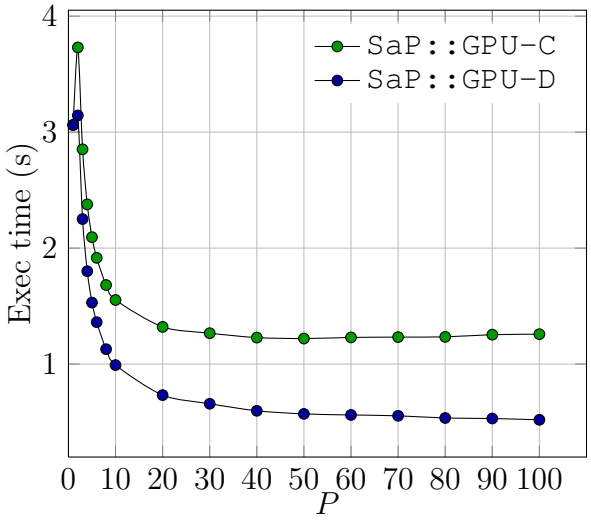

Figure 3: Time to solution as a function of the number of partitions P. Study carried out for a dense banded linear system with $N = 200\,000$, $K = 200$, and $d = 1$.

iterating to find the solution at a level of accuracy of at least 10^{-10}, D_{Kry} vs. C_{Kry}; and the total times, D_{Tot} vs. C_{Tot}. These times are defined as $D_{pre} = T_{LU}$, $C_{pre} = T_{LU} + T_{BC} + T_{SPK} + T_{LUrdcd}$, $D_{Tot} = D_{pre} + D_{Kry}$, and $C_{Tot} = C_{pre} + C_{Kry}$. Note that for `SaP::GPU`, quarters of number of iterations are reported. This is due to the fact that `BiCGStab(2)` contains three exits points during each iteration. Moving from one to the next roughly requires the same amount of effort, which justifies the adopted convention.

The number of iterations to convergence suggests that the quality of the coupled-version of the preconditioner is superior. Yet the price for getting this better preconditioner is higher and `SaP::GPU-D` ends up winning by taking as little as half the time required by `SaP::GPU-C`. When the same factorization is used multiple times, this conclusion could change since the metric that controls the performance would be D_{Kry} and C_{Kry}, or its number of iterations for convergence proxy. Also note that the return on increasing the number of partitions gradually fades away and for the coupled strategy there is no reason to go beyond $P = 50$.

P	D_{pre}	C_{pre}	D_{it}	C_{it}	D_{Kry}	C_{Kry}	D_{Tot}	C_{Tot}	SpdUp
2	1,016.8	1,987.6	1.75	0.75	2,127	1,742.4	3,143.8	3,730	0.84
3	803.7	1,672.5	1.75	0.75	1,446.4	1,179.2	2,250.1	2,851.7	0.79
4	694.7	1,480.7	1.75	0.75	1,105.9	896.3	1,800.6	2,377	0.76
5	630.1	1,371.5	1.75	0.75	900.1	722.7	1,530.2	2,094.2	0.73
6	595.1	1,304.4	1.75	0.75	766.1	611.3	1,361.2	1,915.7	0.71
8	535	1,210.5	1.75	0.75	593.2	471	1,128.3	1,681.5	0.67
10	500	1,166.7	1.75	0.75	491	385.6	991.1	1,552.4	0.64
20	442	1,099.9	1.75	0.75	290.2	220.4	732.1	1,320.3	0.55
30	432.7	1,098.5	1.75	0.75	225	167.7	657.8	1,266.2	0.52
40	410.2	1,087.2	1.75	0.75	186.9	141	597.1	1,228.2	0.49
50	403.5	1,094.8	1.75	0.75	166.6	125.1	570.2	1,219.9	0.47
60	408.4	1,115.9	1.75	0.75	152.7	113.7	561.1	1,229.6	0.46
70	405	1,126.7	1.75	0.75	148.8	105.7	553.8	1,232.4	0.45
80	397.3	1,132.9	1.75	0.75	137.7	101.7	535	1,234.6	0.43
90	397	1,151.4	1.75	0.75	133.5	101.9	530.5	1,253.3	0.42
100	387.8	1,155.9	1.75	0.75	131.6	101.8	519.4	1,257.6	0.41

Table 1: Performance comparison over a spectrum of number of partitions P for coupled (C) vs. decoupled (D) strategies in `SaP::GPU`. All timings are in milliseconds. Problem parameters: $N = 200\,000$, $d = 1$, $K = 200$. The symbols used are as follows: D_{pre}–amount of time spent in preprocessing by the decoupled strategy; D_{it}–number of Krylov iterations for convergence; D_{Tot}–amount of time to converge. Similar values are reported for the coupled scenario. SpdUp= D_{Tot}/C_{Tot}.

4.1.2 Sensitivity with respect to d

Next, we report on the performance of `SaP::GPU` for a dense banded linear system with $N = 200\,000$ and $K = 200$, for degrees of diagonal dominance in the range $0.06 \leq d \leq 1.2$, see Eq. (11). The entries in the matrix are randomly generated and $P = 50$. The findings are summarized in Fig. 4, where `SaP::GPU-C` and `SaP::GPU-D` are compared against the banded linear solver in `MKL`. When $d > 1$ the impact of the truncation becomes increasingly irrelevant, a situation that places the `SaP::GPU` at an advantage. As such, there is no reason to go beyond $d = 1.2$ since if anything, the results will get better. The more interesting range is $d < 1$, when the diagonal dominance requirement is violated. `SaP::GPU` solver demonstrates uniform performance over a wide range of degrees of diagonal dominance. For instance, `SaP::GPU-C` typically required less than one Krylov iteration

for all $d > 0.08$. As the degree of diagonal dominance decreases further, the number of iterations and hence the time to solution increase significantly as a consequence of truncating the spikes that now contain non-negligible values.

It is instructive to see how the solution time is spent by `SaP::GPU-C` and `SaP::GPU-D` and understand how changing d influences this distribution of the time to solution between the major implementation components. The results reported in Table 2 provide this information as they help answer the following question: can one still use a decoupled approach for matrices that are far from being diagonal dominant? The answer is yes, except in the most extreme case, when $d = 0.06$. Note that the number of iterations to convergence for the decoupled approach quickly recovers away from small values of d. In the end, the same $2\times$ speedup factor is obtained virtually over the entire spectrum of d values.

d	D_{pre}	C_{pre}	D_{it}	C_{it}	D_{Kry}	C_{Kry}	D_{Tot}	C_{Tot}	SpdUp
$6{\cdot}10^{-2}$	402.5	1,098.1	353.25	4.25	25,344.3	525.5	25,746.8	1,623.6	15.86
$8{\cdot}10^{-2}$	403.6	1,097.3	8.75	0.75	675.3	128	1,079	1,225.3	0.88
0.1	403.5	1,096.9	6.25	0.75	492.6	128.4	896.1	1,225.2	0.73
0.2	403.4	1,097.5	3.75	0.75	312.1	127.3	715.6	1,224.8	0.58
0.3	404.7	1,096.7	2.75	0.75	248.9	127.2	653.6	1,223.9	0.53
0.4	404	1,096.8	2.75	0.75	240.6	127.4	644.6	1,224.2	0.53
0.5	404.4	1,094.9	2.25	0.75	236.7	125.3	641	1,220.2	0.53
0.6	404	1,096.9	2.25	0.75	202.1	127.5	606.1	1,224.4	0.5
0.7	403.4	1,097.6	2.25	0.75	200.1	128.3	603.5	1,225.9	0.49
0.8	402.4	1,097.1	2.25	0.75	197.5	128.3	599.9	1,225.5	0.49
0.9	403.5	1,096.7	1.75	0.75	162.3	127.3	565.8	1,224	0.46
1	402.6	1,097.6	1.75	0.75	162.5	127.4	565.2	1,225	0.46
1.1	402.5	1,097.1	1.75	0.75	162.4	128.3	564.9	1,225.4	0.46
1.2	403.1	1,097.2	1.75	0.75	172	128	575.1	1,225.2	0.47

Table 2: Influence of d for coupled (C) vs. decoupled (D) strategies in `SaP::GPU` ($N = 200\,000$, $P = 50$, $K = 200$). All timings are in milliseconds. Symbols used are as specified for Table 1.

4.1.3 Comparison with Intel's MKL over a spectrum of N and K

This section summarizes results of a two-dimensional sweep over N and K. In this exercise, prompted by the results reported in Figs. 3 and 4, we fixed $P = 50$ and chose matrices for which $d = 1$. Each row in Table 3 lists

the value of N, which runs from 1000 to 1 000 000. Each column lists the dimension of half bandwidth K, which runs from 10 to 500. Each table row is split in three sub-rows: `SaP::GPU-D` results are reported in the first sub-row; `SaP::GPU-C` in the second sub-row; `MKL` in the third sub-row. All timings are in milliseconds. "OOM" stands for "out-of-memory" – a situation that arises when `SaP::GPU` exhausts during the solution of the linear system the GPU's 6 GB of global memory.

The results reported in Table 3 are statistically summarized in Fig. 5, which provides `SaP` over `MKL` speedup information. Assume that a test "α" successfully ran to completion in `SaP::GPU-D`, requiring $T_\alpha^{\texttt{SaP::GPU-D}}$, and/or in `SaP::GPU-C`, requiring $T_\alpha^{\texttt{SaP::GPU-C}}$. By convention, in case of failing to solve, a negative value; i.e. -1, is assigned to $T_\alpha^{\texttt{SaP::GPU-D}}$ or $T_\alpha^{\texttt{SaP::GPU-C}}$. If a test runs to completion both in `SaP` and `MKL`, the "α" speedup value used to generate the plot in Fig. 5 is computed as $s_{BD} \equiv T_\alpha^{\texttt{MKL}}/T_\alpha^{\texttt{SaP}}$, where $T_\alpha^{\texttt{MKL}}$ is `MKL`'s time to solution and $T_\alpha^{\texttt{SaP}} \equiv \min(\max(T_\alpha^{\texttt{SaP::GPU-D}}, 0), \max(T_\alpha^{\texttt{SaP::GPU-C}}, 0))$. Given that N assumes 10 values and K takes 6 values, "α" can be one of 60 tests. Since three (N, K) tests, namely $(1\,000\,000, 200)$, $(1\,000\,000, 500)$, and $(500\,000, 500)$, failed to solve in `SaP`, the sample population for the statistical study in Fig. 5 is 57. Out of 57 tests, $s_{BD} > 1$ in all but two cases: for $(1\,000\,000, 10)$ when $s_{BD} = 0.87825$, and for $(2000, 50)$ when $s_{BD} = 0.99706$. The highest speedup was $s_{BD} = 8.1255$, for $(2000, 200)$. The median is slightly higher than 2.0, which indicates that of the 57 tests, half were completed by `SaP` two times faster than by `MKL`. The figure also shows that about 25% of the tests run, roughly, between three and six times faster in `SaP`. The red crosses in the figure represent outliers.

4.2 Numerical experiments related to sparse matrix reorderings

When solving sparse linear systems, `SaP` reformulates the sparse problem as a dense banded linear system that is subsequently solved using `SaP::GPU-C` or `SaP::GPU-D`. Ideally, the "sparse–to–dense" transition yields a coefficient matrix that is diagonal heavy; i.e., has a large d, and has a small bandwidth K. Two matrix reorderings are applied in an attempt to meet these two objectives. The first one; i.e., the diagonal boosting reordering, is assessed in section §4.2.1. The second one; i.e., the bandwidth reduction reordering, is evaluated in §4.2.2.

4.2.1 Assessment of the diagonal boosting reordering solution

The first set of results, summarized in Fig. 6, correspond to an efficiency comparison between the hybrid CPU–GPU implementation of §3.2 and the Harwell Sparse Library (HSL) `MC64` algorithm [4]. The hybrid implementation outperformed `MC64` for 96 out of the 116 matrices selected from the Florida Sparse Matrix Collection [15]. The left pane in Fig. 6 presents results of a statistical analysis that used a median-quartile method to measure the spread of the `MC64` and `DB` times to solution. Assume that $T_\alpha^{\texttt{DB}}$ and $T_\alpha^{\texttt{MC64}}$ represent the times required by `DB` and `MC64`, respectively, to complete the diagonal boosting reordering in test α. A relative speedup is computed as

$$\mathcal{S}_\alpha^{\texttt{DB}-\texttt{MC64}} = \log_2 \frac{T_\alpha^{\texttt{MC64}}}{T_\alpha^{\texttt{DB}}} \,. \tag{13}$$

These $\mathcal{S}_\alpha^{\texttt{DB}-\texttt{MC64}}$ values, which can be either positive or negative, are collected in a set $\mathcal{S}^{\texttt{DB}-\texttt{MC64}}$ which is used to generate the left box plot in Fig. 12. The number of tests used to produce these statistical results was 116. Note that a positive value means that `DB` is faster than `MC64`, with the opposite outcome being the case for negative values of $\mathcal{S}_\alpha^{\texttt{DB}-\texttt{MC64}}$. The median values for $\mathcal{S}^{\texttt{DB}-\texttt{MC64}}$ was 1.2423, which indicates that half of the 116 tests ran more than 2.3 times faster using the `DB` implementation. On average, it turns out that the larger the matrix, the faster the `DB` solution becomes. Indeed, as a case study, we analyzed a subset of larger matrices. The "large" attribute was defined in two ways: first, by considering the matrix size, and second, by considering the number of nonzero elements. For the 116 matrices considered, we picked the largest 24 of them; i.e., approximately the largest 20%. To this end, in the first case, we selected all matrices whose dimension was higher than N =150 000. In the second case, we selected all matrices whose number of nonzero elements was larger than 4 350 000. For large N, the median was 1.6255, while for matrices with many nonzero elements, the median was 1.7276. In other words, half of the large tests ran more than three times faster in `DB`. Finally, the statistical results in Fig. 12 indicate that for large tests, with the exception of two outliers, there were no tests for which $\mathcal{S}_\alpha^{\texttt{DB}-\texttt{MC64}}$ was negative; i.e., with one exception, `DB` was faster. When all 116 tests were considered, `MC64` was faster in several cases, with an outlier for which `MC64` was four times faster than `DB`.

Two facts emerged at the end of this analysis. First, as discussed in [31], the bottleneck in the diagonal boosting reordering was either the `DB-S2`

stage; i.e., finding the initial match, or the DB-S3 stage; i.e., finding a perfect match, with an approximately equal split among them. Secondly, the quality of the reordering turned out to be identical – almost all matrices displayed the same grand product of the diagonal entries regardless of whether the reordering was carried out using MC64 or DB.

4.2.2 Assessment of the bandwidth reduction solution

The performance of the CM solution implemented in SaP was evaluated on a set of 125 sparse matrices from various applications. These matrices were the 116 used in the previous section plus several other matrices such as ANCF31770, ANCF88950, and NetANCF_40by40, etc., that arise in granular dynamics and the implicit integration of flexible multi-body dynamics [22, 23, 45]. Figure 7 presents results of a statistical analysis that used a median-quartile method to compare (i) the half bandwidths of the matrices obtained by Harwell's MC60 and SaP's CM; and, (ii) the time to solution; i.e., time to complete a band-reducing reordering. For (i), the quantity reported is the relative difference between the resulting bandwidths,

$$r_K \equiv 100 \times \frac{K_{\texttt{MC60}} - K_{\texttt{CM}}}{K_{\texttt{CM}}}\,,$$

where $K_{\texttt{MC60}}$ and $K_{\texttt{CM}}$ are, respectively, the half bandwidths K of the matrices produced by MC60 and CM. For (ii), the metric used was identical to the one introduced in Eq. (13). Note that CM is superior when r_K assumes large positive values, which are also desirable for the time-to-solution plot. As far as r_K is concerned, the median value is 0%; i.e., out of 125 matrices, about half are better off being reordered by Harwell's MC60 with the other half being better off reordered by SaP's CM. On a positive side, the number of outliers for CM is higher, indicating that there is a propensity for CM to "win big". In terms of times to solution, MC60 is marginally faster than CM's hybrid CPU/GPU solution. Indeed, the median value of the performance metric is -0.1057; i.e., it takes half of the tests run with CM at least 1.076 times longer to complete the bandwidth reduction task.

It is insightful to discuss what happens when this statistical analysis is controlled to only consider larger matrices. The results of this analysis are captured in Fig. 8. Just like in section §4.2.1, the focus is on the largest 20% matrices, where "large" is understood to mean large matrix dimension N, and then separately, large number of nonzeros nnz. Incidentally, the cut-off

value for the dimension was N =215 000, while for the number of nonzeros was nnz =7 800 000. When the statistical analysis included the 25 largest matrices based on size N, the median value for the half bandwidth metric r_K was yet again 0.0%. The median value for time to solution changed however, from -0.1057 to 0.6964 to indicate that for half of these large tests SaP ran more than 1.6 times faster than the Harwell solution. Qualitatively, the same conclusions were reached when the 25 large matrices were selected on the grounds on nnz count. The median for r_K was 0.4182%, which again suggested that the relative difference in the resulting bandwidth K yielded by CM and MC60 was practically negligible. The median time to solution was the same 0.6964. Note though that according to the results shown in Fig. 8, there is no large–nnz test for which the Harwell implementation is faster than the CM. In fact, 25% of the large tests; i.e., about five tests, run at least three times faster in CM.

Finally, it is worth pointing out the correlations between times to solutions and K values, on the one hand, and N and nnz, on the other hand. Herein, the correlation used is the Pearson product-moment correlation coefficient [10]. As a rule of thumb, a Pearson correlation coefficient of 0.01 to 0.19 suggests a negligible relationship, while a coefficient between 0.7 and 1.0 indicates a strong positive relationship. The correlation coefficient between the bandwidth and the dimension N of the matrix turns out to be small; i.e., 0.15 for MC60 and 0.16 for CM. Indeed, the fact that a matrix is large doesn't say much about what K value one can expect upon reordering this matrix. The correlation between the number of nonzeros and the amount of time to figure out the reordering is very high though. In other words, the larger the matrix size N, the longer the time to produce the reordering. For instance, the correlation coefficient was 0.91 for MC60 and 0.81 for CM. The same observation holds for the number of nonzeros entries: when there is a lot of them, the time to produce a reordering is large. The Pearson correlation coefficient is 0.71 for MC60 and 0.83 for CM. These correlation coefficients were obtained on a sample size of 125 matrices. Yet the same trends are manifest for the reduced set of 25 large matrices that we worked with. For instance, the correlation between dimension N and resulting K is very small at large N values: 0.04 for MC60 and 0.05 for CM. For the time to solution, the correlation coefficients with respect to N are 0.89 for MC60 and 0.76 for CM.

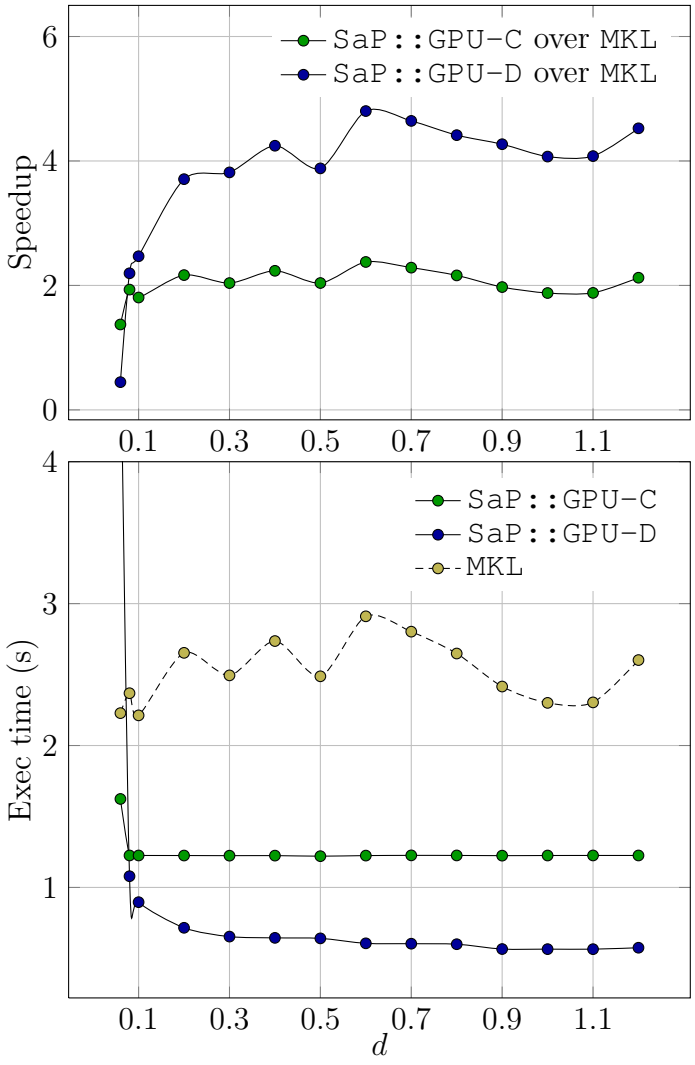

Figure 4: Influence of the diagonal dominance d, with $0.06 \leq d \leq 1.2$, for fixed values $N = 200\,000$, $K = 200$ and $P = 50$.

N	K					
	10	20	50	100	200	500
1000	2.433E1	1.755E1	1.816E1	2.067E1	2.755E1	2.952E1
	6.637	7.354	1.106E1	1.866E1	2.937E1	2.955E1
	1.145E1	1.080E1	1.281E1	2.208E1	2.145E2	2.208E2
2000	2.224E1	1.873E1	1.911E1	2.149E1	2.725E1	5.638E1
	6.158	8.514	1.328E1	2.464E1	3.569E1	9.514E1
	1.255E1	1.100E1	1.324E1	2.214E1	2.214E2	2.357E2
5000	2.517E1	2.062E1	2.101E1	2.327E1	3.259E1	8.002E1
	7.597	9.266	1.622E1	3.049E1	5.866E1	2.372E2
	1.307E1	1.233E1	2.145E1	3.827E1	2.531E2	2.944E2
10 000	2.823E1	2.758E1	2.385E1	2.686E1	4.509E1	1.183E2
	1.019E1	1.168E1	1.887E1	4.561E1	1.060E2	4.737E2
	1.560E1	1.509E1	2.959E1	5.881E1	3.009E2	3.928E2
20 000	3.393E1	3.235E1	3.302E1	4.198E1	5.991E1	2.016E2
	1.428E1	1.653E1	2.741E1	6.676E1	1.950E2	9.500E2
	2.087E1	2.323E1	4.879E1	1.117E2	3.373E2	5.947E2
50 000	6.433E1	5.825E1	5.869E1	9.085E1	1.466E2	4.361E2
	2.713E1	3.048E1	5.470E1	1.444E2	3.668E2	2.337E3
	3.263E1	4.107E1	1.030E2	2.597E2	7.151E2	1.107E3
100 000	9.838E1	8.703E1	1.112E2	1.527E2	2.917E2	9.571E2
	4.765E1	5.576E1	9.650E1	2.612E2	6.498E2	3.583E3
	5.392E1	6.966E1	1.910E2	4.956E2	1.275E3	2.277E3
200 000	1.808E2	1.590E2	1.877E2	3.285E2	5.679E2	2.003E3
	8.992E1	1.035E2	1.868E2	5.054E2	1.221E3	6.051E3
	9.509E1	1.259E2	3.676E2	9.831E2	2.386E3	4.211E3
500 000	3.720E2	3.651E2	4.425E2	7.240E2	1.411E3	OOM
	2.037E2	2.380E2	4.424E2	1.229E3	2.928E3	OOM
	2.135E2	2.924E2	8.969E2	2.539E3	6.231E3	1.071E4
1 000 000	7.242E2	7.092E2	9.788E2	1.442E3	OOM	OOM
	3.970E2	4.633E2	8.640E2	2.443E3	OOM	OOM
	3.486E2	5.692E2	1.778E3	4.712E3	1.137E4	2.159E4

Table 3: Performance comparison, two-dimensional sweep over N and K for $P = 50$ and $d = 1$. For each value N, the three rows correspond to the `SaP::GPU-D`, `SaP::GPU-C`, and `MKL` solvers, respectively.

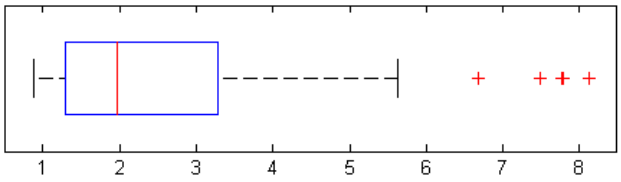

Figure 5: SaP speedup over Intel's MKL – statistical analysis based on values in Table 3.

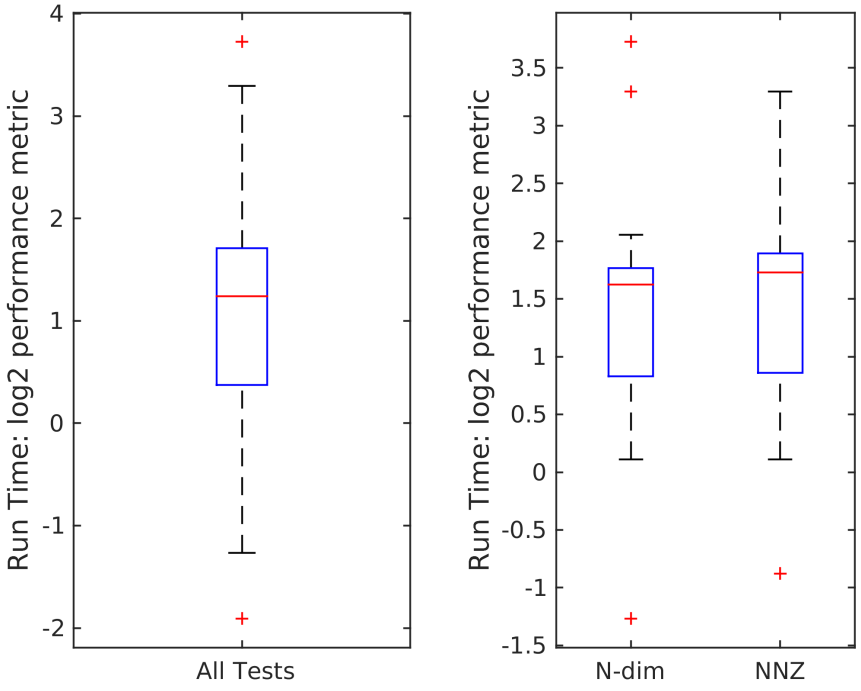

Figure 6: Results of a statistical analysis that uses a median-quartile method to measure the spread of the MC64 and DB times to solution. The speedup factor, or performance metric, is computed as in Eq. (13).

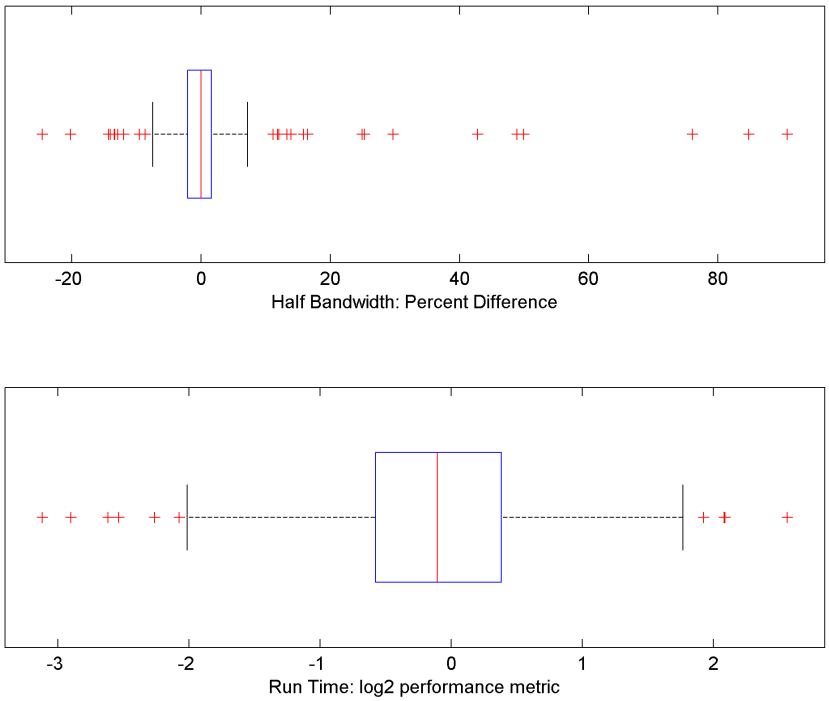

Figure 7: Comparison of the Harwell `MC60` and `SaP`'s `CM` implementations in terms of resulting half bandwidth K and time to solution.

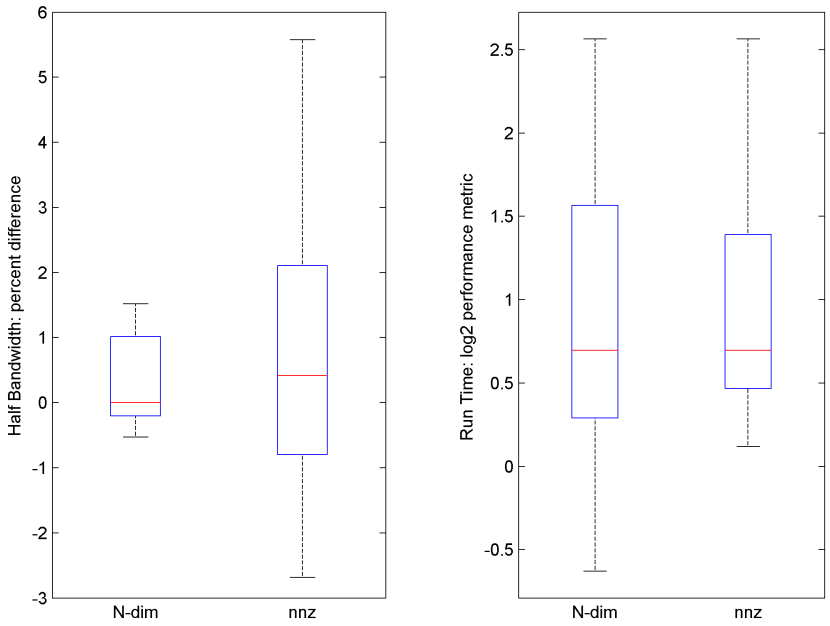

Figure 8: Comparison of the Harwell `MC60` and `SaP`'s `CM` implementations in terms of resulting half bandwidth K and time to solution. Statistical analysis of large matrices only.

4.3 Numerical experiments related to sparse linear systems

4.3.1 Profiling results

Figure 9 plots statistical results that summarize how the time to solution; i.e., finding $\mathbf{x}$ in $\mathbf{Ax} = \mathbf{b}$, is spent in `SaP::GPU`. The raw data used in this analysis is available on-line [3]; also, a discussion of exactly what it means to find the solution of the linear system is postponed for section §4.3.4. The labels used in the plot Fig. 9 are inspired by the notation used in section §3.4 and Fig. 2. Consider for instance the diagonal boosting reordering `DB` employed by `SaP`. In a statistical sense, the percent of time to solution spent in `DB` is represented using a median-quartile method to measure statistical spread. The raw data used to generate the `DB` box was obtained as follows. If a test "α" that runs to completion requires $T_\alpha^{DB} > 0$ for `DB` completion, then this test will generate one data entry in an array of data subsequently used to produce the statistical result. The actual entry that is used is $100 \times T_\alpha^{DB}/T_\alpha^{Tot}$, where T_α^{Tot} is the total amount of time that test "α" takes for completion. In other words, the entry is the percent of time spent when solving this particular linear system for performing the diagonal boosting reordering. The bars for the K-reducing reordering (`CM`), for multiple data transfers between CPU and GPU (`Dtrsf`), etc., are similarly obtained. Not all bars in Fig. 9 were generated using the same number of data entries; i.e., some tests contributed to some but not all bars. For instance, a symmetric positive definite linear system requires no `DB` step and such this test won't contribute an entry to the array of data used to determine the `DB` box in the figure. Of a batch of 85 tests that ran to completion with `SaP`, the sample population used to generate the bars is as follows: 85 data points for `CM`, `Dtrsf`, and `Kry`; 63 data points for `DB`; 60 for `LU`; 32 data points for `Drop`; and 9 data points for `BC`, `SPK`, and `LUrdcd`. These counts provide insights into the solution path adopted by `SaP` in solving the 85 linear systems. For instance, the coupled approach; i.e., the SPIKE method of [38] has been employed in the solution of nine of the 85 linear systems. The rest of them were used via `SaP::GPU-D`. Of 85 linear systems, 25 were most effectively solved by `SaP` resorting to diagonal preconditioning; i.e., after `DB` all the entries were dropped off except the heavy diagonal ones. Also, note that several of the linear systems considered were symmetric positive definite, from where the 60 points count for `DB`.

A statistical analysis of the time spent in the Krylov-subspace component of the solution reveals that the median time was 55.84%. The median times for the other components of the solution are listed in the first row of data in Table 4. The second row of data provides the median values when the Krylov-subspace component, which dwarfs most of the solution components is eliminated. In this case, the entry for DB, for instance, was obtained based on data points $100 \times T_\alpha^{DB}/T_\alpha^{Tot}$, where this time around T_α^{Tot} included everything except the time spent in the Krylov-subspace component of the solution. In other words, T_α^{Tot} is the time required to compute from scratch the preconditioner. The median values should be used in conjunction with the median-quartile boxplot of Fig. 9 for the first row of data, and Fig. 10 for the second row of data. Consider, for instance, the results associated with the drop-off operation. In the Krylov-inclusive measurement, Drop has a median of 4.1%; i.e., half of the 32 tests which employed drop-off spent more than amount in performing the drop-off, while half were quicker. The spread is rather large and there are several outliers that suggest that a handful of tests require a very large amount of time be spent in the drop-off part of the solution.

DB	CM	Dtransf	Drop	Asmbl	BC	LU	SPK	LUrdcd
3.4	1.4	1.9	4.1	0.7	1.4	24.8	23	4.1
11.4	3.7	4.1	25.5	2.7	2.3	73.4	41.8	6.4

Table 4: Median information for the SaP solution components as % of the time for solution. Two scenarios are considered: the first data row provides values when the total time; i.e., 100%, included the time spent by SaP in the Krylov-subspace component. The second row of data is obtained by considering 100% to be the time required to compute a factorization of the preconditioner. Note that values in each row of data does not add up to 100% for several reasons. First, these are statistical median values. Second, there are very many tests that do not include all the components of the solution. For instance, SPK is computed based on a set of nine points while Drop is computed using 32 data points, some of them not even obtained in conjunction with the same test.

The results in Fig. 9 and Table 4 suggest where the optimization efforts should concentrate in the future. For instance, the time required for the CPU↔GPU data transfer is, in the overall picture, rather insignificant and

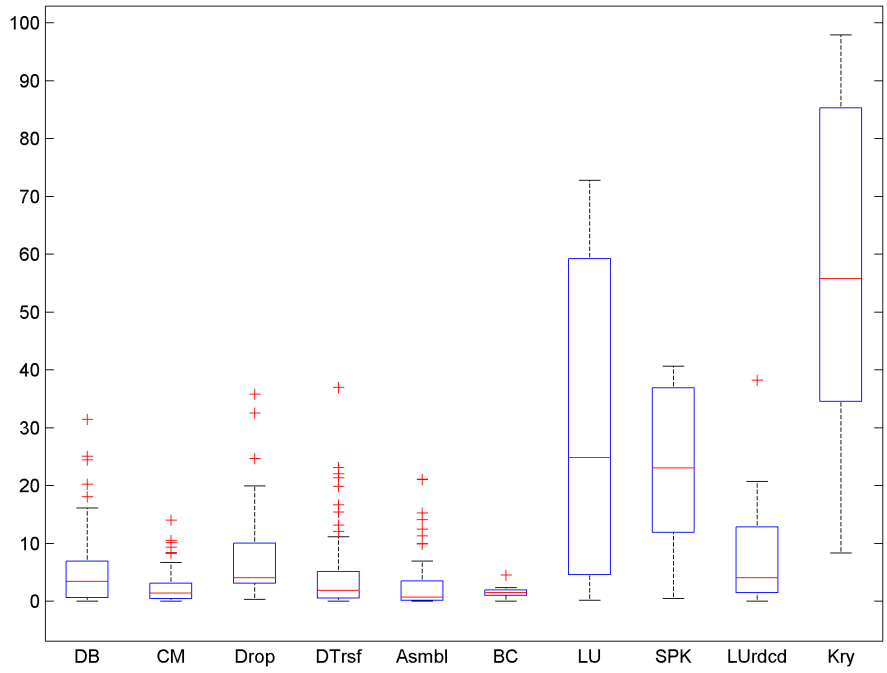

Figure 9: Profiling results obtained for a set of 85 linear systems that, out of a collection of 114, could be solved by `SaP::GPU`.

as such a matter of small concern. Somewhat surprising, the amount of time spent in drop-off came out higher than anticipated, at least in relative terms. One caveat is that no effort was made to optimize this component of the solution. Instead, the effort went into optimizing the `DB` and `CM` solution components. This paid off, as matrix reordering in `SaP`, particularly for large matrices, is fast when compared to Harwell and it reached the point where the drop-off became a more significant bottleneck. Another unexpected observation was the relative small number of scenarios in which `SaP::GPU-C` was preferred over `SaP::GPU-D`; i.e., in which the SPIKE strategy [38] was employed. This observation, however, should not be generalized as it might very well be specific to the `SaP` implementation. Indeed, it simply states that in the current implementation, a large number of iterations associated with a less sophisticated preconditioner is preferred to a smaller count of expensive iterations associated with `SaP::GPU-C`. Out of a sample population of 85 tests, when invoked, the median number of iterations to solution in `SaP::GPU-C` was 6.75. Conversely, when `SaP::GPU-D` was preferred,

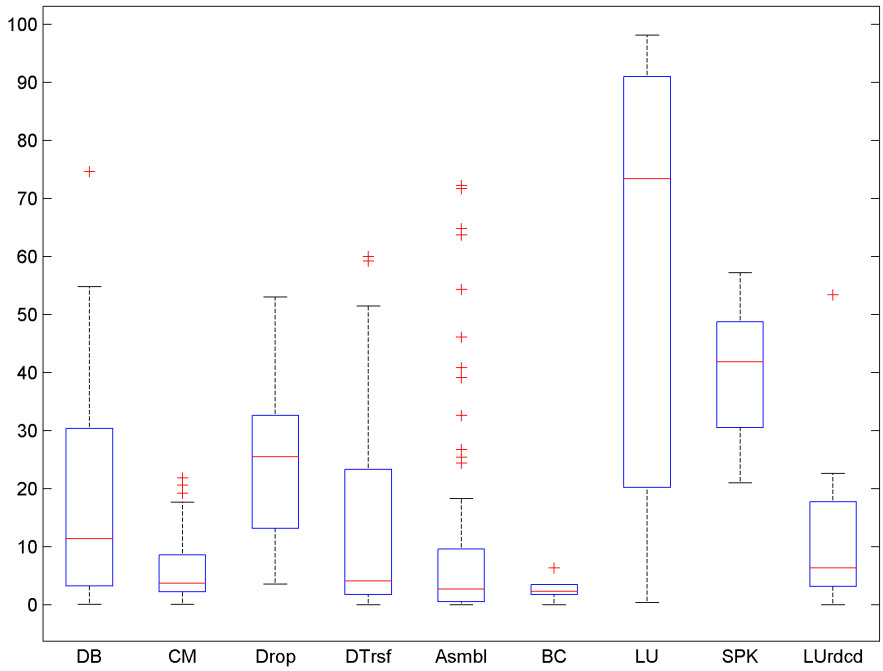

Figure 10: Profiling results obtained for a set of 85 linear systems that, out of a collection of 114, could be solved by `SaP::GPU`.

the median count was 29.375 [3].

4.3.2 The impact of the third stage reordering

It is almost always the case that upon carrying out a CM reordering of a sparse matrix, the resulting $\mathbf{A}$ matrix has a small number of entries in the first and last rows. Yet, as the row index j increases, the number of nonzero in row j increases up to approximately $j \approx N/2$. Thereafter, the nonzero count starts decreasing to reach small values towards $j \approx N$. Overall, $\mathbf{A}$ has its K value dictated by the worst offender. Therefore, a partitioning of $\mathbf{A}$ into $\mathbf{A}_i$, $i = 1, \ldots, P$ would conservatively require that, for instance, $\mathbf{A}_1$ and $\mathbf{A}_P$ work with a large K most likely dictated by a sub-matrix such as $\mathbf{A}_{P/2}$. Allowing each $\mathbf{A}_i$ to have its own K_i proved to lead to efficiency gains for two main reasons. First, in `SaP::GPU-C` it led to a reduction in the dimension of the spikes, since for each pair of coupling blocks $\mathbf{B}_i$ and $\mathbf{C}_i$, the number of columns in the ensuing spikes was determined as the larger of the values K_i

and K_{i+1}. Second, `SaP::GPU` capitalizes on the observation that, since $\mathbf{A}_i$ are independent and governed by their local K_i, there is nothing to prevent a third reordering, which attempts to further reduce the bandwidth of $\mathbf{A}_i$. As it comes on the heels of the `DB` and `CM` reorderings, this is called a "third stage reordering" and is applied independently and preferably concurrently to the P sub-matrices $\mathbf{A}_i$. As illustrated in Table 5, the decrease in local K_i can be significant and it can lead to non-negligible speedups, see Table 6.

Mat. Name	P	K_i before 3^{rd} SR	K_i after 3^{rd} SR
ANCF31770	20	123, 170, 204, 229, 247 247, 247, 247, 248, 242 213, 181, 134, 68, 106 129, 124, 124, 113, 82	89, 92, 79, 46, 45 48, 48, 59, 50, 58 72, 98, 64, 56, 42 36, 54, 49, 59, 82
ANCF88950	20	194, 274, 337, 387, 410 410, 410, 410, 410, 405 352, 296, 227, 116, 176 208, 204, 204, 191, 137	116, 74, 65, 109, 112 97, 100, 93, 97, 114 116, 56, 88, 75, 116 50, 96, 97, 118, 75
af23560	10	274, 317, 317, 317, 320 339, 334, 317, 314, 283	140, 71, 71, 102, 74 123, 127, 119, 114, 143
NetANCF40by40	16	256, 378, 458, 533 599, 634, 578, 517 436, 343, 215, 210 275, 295, 257, 178	125, 68, 122, 118 85, 93, 97, 91 57, 69, 112, 85 85, 73, 113, 101
bayer01	8	684, 1325, 1308, 1288 879, 501, 493, 508	532, 170, 122, 110 109, 110, 110, 121
ex19	8	139, 87, 87, 87 74, 46, 62, 40	136, 87, 87, 87 68, 46, 62, 40
finan512	16	1124, 1287, 1316, 1331 1331, 1331, 1331, 1331 1331, 1331, 1331, 1331 1331, 1331, 1331, 1015	587, 288, 288, 288 288, 288, 288, 288 288, 288, 288, 288 288, 288, 227, 211
gridgena	6	247, 405, 405 405, 405, 247	132, 81, 80 122, 72, 105
lhr10c	6	315, 348, 288 166, 156, 259	427, 247, 293 217, 226, 157
rma10	10	180, 281, 702, 678, 495 637, 560, 495, 478, 545	155, 241, 647, 540, 254 496, 422, 217, 349, 358

Table 5: Examples of matrices where the third stage reordering (3^{rd} SR) reduced more significantly the block bandwidth K_i for $\mathbf{A}_i$, $i = 1, \ldots, P$.

Mat. Name	w/o 3^{rd} SR		w/ 3^{rd} SR		SpdUp
	P	K_i	P	K_i	
ANCF31770	16	248	20	98	1.203
ANCF88950	32	410	20	118	1.537
af23560	10	339	10	143	1.238
NetANCF40by40	16	634	16	125	1.900
bayer01	8	1325	8	532	2.234
ex19	6	139	8	136	1.331
finan512	10	1331	16	587	1.804
gridgena	6	405	6	132	1.636
lhr10c	4	427	6	259	1.228
rma10	10	702	10	647	1.113

Table 6: Speed-up "SpdUp" values obtained upon embedding a third stage reordering step in the solution process, a decision that also changed the number of partitions P for best performance. When correlating the results reported to values provided in Table 5, this table lists for each matrix $\mathbf{A}$ the largest of its K_i values, $i = 1, \ldots, P$.

4.3.3 Comparison against state of the art

A set of 114 matrices, of which 105 are from the Florida matrix collection, is used herein to compare the robustness and time to solution of `SaP::GPU`, `PARDISO`, `SuperLU`, and `MUMPS`. This set of matrices was selected on the following basis: at least one of the four solvers can retrieve the solution $\mathbf{x}$ within 1% relative accuracy. For a sparse linear system $\mathbf{Ax} = \mathbf{b}$, this relative accuracy was measured as follows. An exact solution $\mathbf{x}^\star$ was first chosen and then the right-hand side was set to $\mathbf{b} = \mathbf{Ax}^\star$. Each sparse linear solver attempted to produce an approximation $\mathbf{x}$ of the solution $\mathbf{x}^\star$. If this approximation satisfied $||\mathbf{x} - \mathbf{x}^\star||_2/||\mathbf{x}^\star||_2 \leq 0.01$, then the solve was considered to have been successful. Given that `SaP::GPU` is an iterative solver, its initial guess is always $\mathbf{x}^{(0)} = \mathbf{0}_N$. Although in many instances the initial guess can be selected to be relatively close the actual solution, this situation is avoided here by choosing $\mathbf{x}^\star$ far from the aforementioned initial guess. Specifically, $\mathbf{x}^\star$ had its entries roughly distributed on a parabola starting from 1.0 as the first entry, approaching the value 400 at $N/2$, and decreasing to 1.0 for the N^{th} and last entry of $\mathbf{x}^\star$. The statistical results reported in this section draw on raw data provided in the Appendix in Table 8. Figure 11 employs a median-quartile method to measure the statistical spread of the 114 matrices used in this sparse solver comparison. In terms of

size, N is between 8192 and 4 690 002. In terms of nonzeros, nnz is between 41 746 and 46 522 475. The median for N is 71 328. The median for nnz is 1 167 967.

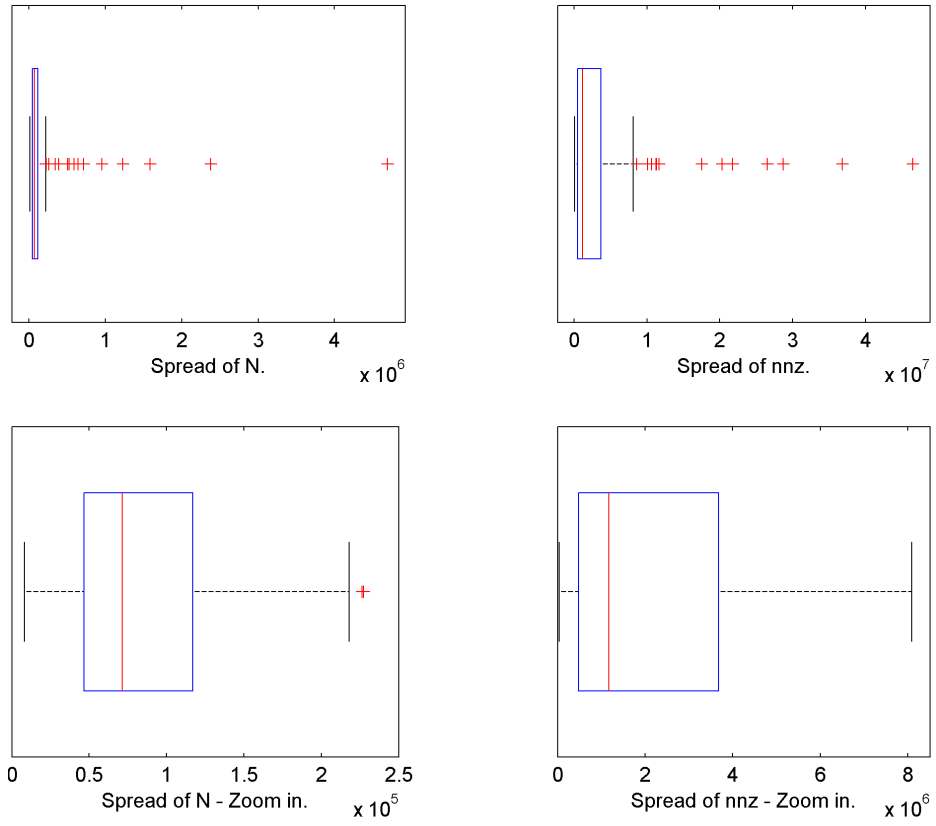

Figure 11: Statistical information regarding the dimension N and number of nonzeros nnz for the 114 coefficient matrices used to compare `SaP::GPU`, `PARDISO`, `SuperLU`, and `MUMPS`.

On the robustness side, `SaP::GPU` failed to solve 28 linear systems. In 23 cases, `SaP` ran out of GPU global memory. In the remaining five cases, `SaP::GPU` failed to converge. The rest of the solvers failed as follows: `PARDISO` 40 times, `SuperLU` 22 times, and `MUMPS` 35 times. These results should be qualified as follows. The GPU card had 6 GB of GDDR5-type memory. Given that in its current implementation `SaP::GPU` is an in-core solver, it does not swap data in and out of the GPU. Consequently, it ran 23 times against this memory-size hard constraint. This issue can be partially

alleviated by considering a better GPU card. Indeed, there are cards that have as much as 24 GB of global memory, which still comes short of the 64 GB of RAM that PARDISO, SuperLU, and MUMPS could tap into. Secondly, the PARDISO, SuperLU, and MUMPS solvers were used with default setting. Adjusting parameters that control these solvers' solution process would likely increase their success rate.

Interestingly, for the 114 linear systems considered there was a perfect negative correlation between speed and robustness. PARDISO was the fastest, followed by MUMPS, then SaP, and finally SuperLU. Of the 57 linear systems solved both by SaP and PARDISO, SaP was faster 20 times. Of the 71 linear systems solved both by SaP and SuperLU, SaP was faster 38 times. Of the 60 linear systems solved both by SaP and MUMPS, SaP was faster 27 times. Of the 60 linear systems solved both by PARDISO and SuperLU, PARDISO was faster 60 times. Of the 57 linear systems solved both by SaP and MUMPS, PARDISO was faster 57 times. And finally, of the 64 linear systems solved both by SuperLU and MUMPS, SuperLU was faster 24 times.

We compare next the four solvers using a median-quartile method to measure statistical spread. Assume that $T_\alpha^{\texttt{SaP}}$ and $T_\alpha^{\texttt{PARDISO}}$ represent the times required by SaP::GPU and PARDISO, respectively, to finish test α. A relative speedup is computed as

$$\mathcal{S}_\alpha^{\texttt{SaP}-\texttt{PARDISO}} = \log_2 \frac{T_\alpha^{\texttt{PARDISO}}}{T_\alpha^{\texttt{SaP}}}\,, \tag{14}$$

with $\mathcal{S}_\alpha^{\texttt{SaP}-\texttt{MUMPS}}$ and $\mathcal{S}_\alpha^{\texttt{SaP}-\texttt{SuperLU}}$ similarly computed. These $\mathcal{S}_\alpha^{\texttt{SaP}-\texttt{PARDISO}}$ values, which can be either positive or negative, are collected in a set $\mathcal{S}^{\texttt{SaP}-\texttt{PARDISO}}$ which is used to generate a box plot in Fig. 12. The figure also reports results on $\mathcal{S}^{\texttt{SaP}-\texttt{SuperLU}}$, and $\mathcal{S}^{\texttt{SaP}-\texttt{MUMPS}}$. Note that the number of tests used to produce these statistical measures is different for each comparison: 57 linear systems for $\mathcal{S}^{\texttt{SaP}-\texttt{PARDISO}}$, 71 for $\mathcal{S}^{\texttt{SaP}-\texttt{SuperLU}}$, and 60 for $\mathcal{S}^{\texttt{SaP}-\texttt{MUMPS}}$. The median values for $\mathcal{S}^{\texttt{SaP}-\texttt{PARDISO}}$, $\mathcal{S}^{\texttt{SaP}-\texttt{SuperLU}}$, and $\mathcal{S}^{\texttt{SaP}-\texttt{MUMPS}}$ are -1.4036, 0.0934, and -0.3242, respectively. These results suggest that when it finishes, PARDISO can be expected to be about two times faster than SaP. MUMPS is marginally faster than SaP, which on average can be expected to be only slightly faster than SuperLU.

Red crosses are used in Fig. 12 to show statistical outliers. Favorably, most of the SaP's outliers are large and positive. For instance, there are three linear systems for which compared to PARDISO, SaP finishes significantly faster, four linear systems for which it is significantly faster than SuperLU,

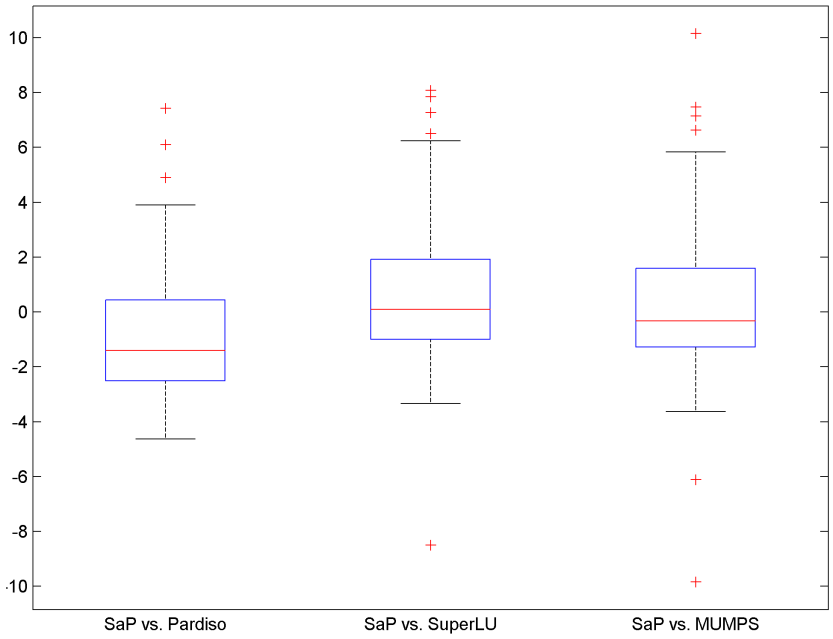

Figure 12: Statistical spread for `SaP::GPU`'s performance relative to that of `PARDISO`, `SuperLU`, and `MUMPS`. Referring to Eq. 14, the results were obtained using the data sets $\mathcal{S}^{\texttt{SaP}-\texttt{PARDISO}}$ (with 57 values), $\mathcal{S}^{\texttt{SaP}-\texttt{SuperLU}}$ (71 values), and $\mathcal{S}^{\texttt{SaP}-\texttt{MUMPS}}$ (60 values).

and four linear systems for which it is significantly faster than `MUMPS`. On the flip side, there are two tests where `SaP` runs slower than `MUMPS` and one test where it runs significantly slower then `SuperLU`. The results also suggest that about 50% of the linear systems run in `SaP` in the range between "as fast as `PARDISO` or two to three times slower", 50% of the linear systems run in `SaP` in the range "between four times faster to four times slower then `SuperLU`". Relative to `MUMPS`, the situation is just like for `SuperLU` if only slightly shifted towards negative territory: the second and third quartile suggest that 50% of the linear systems run in `SaP` in the range "between three times faster to three times slower then `MUMPS`". Again, favorably for `SaP`, the last quartile is long and reaches well into high positive values. In other words, when it beats the competition, it beats it by a large margin.

4.3.4 Comparison against another GPU solver

The same set of 114 matrices used in the comparison against PARDISO, SuperLU, and MUMPS was considered to compare SaP::GPU with the sparse direct QR solver in cuSOLVER library [7]. For cuSOLVER, the QR solver was run in two configurations: with or without the application of a reversed Cuthill–McKee (RCM) reordering before solving the system. RCM was optionally applied given that it can potentially reduce the QR factorization fill-in. cuSOLVER successfully solved 45 out of 114 systems when using either configuration. There are only three linear systems: ABACUS_shell_ud, ex11 and jan99jac120, which were successfully solved by cuSOLVER but not by SaP::GPU. Of the 42 systems solved both by SaP::GPU and cuSOLVER, cuSOLVER was faster than SaP::GPU in five cases. In all 69 systems cuSOLVER failed to solve, the implementation ran out of memory.

5 Conclusions and future work

This contribution discusses parallel strategies to (*i*) solve dense banded linear systems; (*ii*) solve sparse linear systems; and (*iii*) perform matrix reorderings for diagonal boosting and bandwidth reduction. The salient feature shared by these strategies is that they are designed to run in parallel on GPU cards. BSD3 open source implementations of all these strategies are available at [2, 3] as part of a software package called SaP. As far as the parallel solution of linear systems is concerned, the strategies discussed are in-core; i.e., there is no host-device, CPU-GPU, memory swapping, which somewhat limits the size of the problems that can be presently solved by SaP. Over a broad range of dense matrix sizes and bandwidths, SaP is likely to run two times faster than Intel's MKL. This conclusion should be modulated by hardware considerations and also the observation that the diagonal dominance of the dense banded matrix is a performance factor. On the sparse linear system side, the most surprising result was the robustness of SaP. Out of a set of 114 tests, most of them using matrices from the University of Florida sparse matrix collection, SaP failed only 28 times, of which 23 were "out-of-memory" failures owing to a 6 GB limit on the size of the GPU memory. In terms of performance, SaP was compared against PARDISO, MUMPS, and SuperLU. We noticed a perfect negative correlation between robustness and time to solution: the faster a solver, the less robust it was. In this context, PARDISO

was the fastest, followed by MUMPS, SaP, and SuperLU. Surprisingly, the straight split-and-parallelize strategy, without the coupling involved in the SPIKE-type strategy, emerged as the more often solution approach adopted by SaP.

The implementation of SaP is somewhat peculiar in that the sparse solver builds on top of the dense banded one. The sparse–to–dense transition occurs via two reorderings: one that boosts the diagonal entires and one that reduces the matrix bandwidth. Herein, they were implemented as CPU/GPU hybrid solutions which were compared against Harwell's implementations and found to be twice as fast for the diagonal boosting reordering, and of comparable speed for the bandwidth reduction.

Many issues remain to be investigated at this point. First, given that more than 50% of the time to solution is spent in the iterative solver, it is worth consider the techniques analyzed in [30], which sometimes double the flop rate in sparse matrix-vector multiplication operations upon changing the matrix storage scheme; i.e., moving from CSR to ELL or hybrid. Second, an out-of-core and/or multi-GPU implementation would enable SaP to handle larger problems while possibly reducing time to solution. Third, the CM bandwidth reduction strategy implemented is dated; spectral and/or hyper-graph partitioning for load balancing should lead to superior splitting of the coefficient matrix. Finally, as it stands, with the exception of parts of the matrix reordering, SaP is entirely a GPU solution. It would be worth investigating how the CPU can be involved in other phases of the implementation. Such an investigation would be well justified given the imminent tight integration of the CPU and GPU memories.

Acknowledgments

This work was funded through National Science Foundation grant SI2-SSE 1147337 and benefited from many discussions the authors had with Matt Knepley and Ahmed Sameh.

A Solver comparisons raw data

For completeness, we provide here the raw comparison data for the tested solvers which was used in generating the figures and plots in the paper. Table 7 gives the list of tested matrices, specifying their size N and number

of non-zero elements nnz. Table 8 reports the run times to solution (in ms) for the `SaP::GPU`, `PARDISO`, `SuperLU`, and `MUMPS` solvers. Table 9 reports the run times to solution comparison for `SaP::GPU` and `cuSOLVER`, the latter without or with Cuthill-McKee (`CM`) reordering.

Table 7: Dimension N and number of non-zero elements of tested matrices.

	Name	N	nnz
1	2cubes_sphere	101 492	1 647 264
2	2D_54019_highK	54 019	996 414
3	a2nnsnsl	80 016	347 222
4	a5esindl	60 008	255 004
5	ABACUS_shell_ud	23 412	218 484
6	af_5_k101	503 625	17 550 675
7	af23560	23 560	484 256
8	ANCF31770	31 770	183 540
9	ANCF88950	88 950	513 900
10	apache1	80 800	542 184
11	apache2	715 176	4 817 870
12	appu	14 000	1 853 104
13	ASIC_100k	99 340	954 163
14	ASIC_100ks	99 190	578 890
15	av41092	41 092	1 683 902
16	bayer01	57 735	277 774
17	bcircuit	68 902	375 558
18	bcsstk39	46 772	2 089 294
19	blockqp1	60 012	640 033
20	bmw3_2	227 362	11 288 630
21	bmwcra_1	148 770	10 644 002
22	boyd1	93 279	1 211 231
23	bratu3d	27 792	173 796
24	bundle1	10 581	770 901
25	c-59	41 282	480 536
26	c-61	43 618	310 016
27	c-62	41 731	559 343
28	cant	62 451	4 007 383
29	case39	40 216	1 042 160
30	case39_A_01	40 216	1 042 160
31	c-big	345 241	2 341 011
32	cfd1	70 656	1 828 364
33	cfd2	123 440	3 087 898
34	circuit_4	80 209	307 604
35	ckt11752_tr_0	49 702	333 029
36	cont-201	80 595	438 795
37	cont-300	180 895	988 195

Continued on next page

Table 7 – continued from previous page

	Name	N	nnz
38	copter2	55 476	759 952
39	CurlCurl_4	2 380 515	26 515 867
40	dawson5	51 537	1 010 777
41	dc1	116 835	766 396
42	dixmaanl	60 000	299 998
43	Dubcova2	65 025	1 030 225
44	dw8192	8192	41 746
45	ecl32	51 993	380 415
46	epb3	84 617	463 625
47	ex11	16 614	1 096 948
48	ex19	12 005	259 879
49	FEM_3D_thermal1	17 880	430 740
50	filter3D	106 437	2 707 179
51	finan512	74 752	596 992
52	G3_circuit	1 585 478	7 660 826
53	g7jac140	41 490	565 956
54	Ga3As3H12	61 349	5 970 947
55	GaAsH6	61 349	3 381 809
56	garon2	13 535	390 607
57	gas_sensor	66 917	1 703 365
58	gridgena	48 962	512 084
59	gsm_106857	589 446	21 758 924
60	H2O	67 024	2 216 736
61	hcircuit	105 676	513 072
62	HTC_336_4438	226 340	904 522
63	ibm_matrix_2	51 448	1 056 610
64	inline_1	503 712	36 816 342
65	jan99jac120	41 374	260 202
66	ldoor	952 203	46 522 475
67	lhr10c	10 672	232 633
68	Lin	256 000	1 766 400
69	lung2	109 460	492 564
70	mario002	389 874	2 101 242
71	mark3jac100	45 769	285 215
72	mark3jac140	64 089	399 735
73	matrix_9	103 430	2 121 550
74	minsurfo	40 806	203 622
75	msc23052	23 052	1 154 814
76	ncvxbqp1	50 000	349 968
77	nd24k	72 000	28 715 634
78	NetANCF40by40	63 603	569 262
79	offshore	259 789	4 242 673
80	oilpan	73 752	3 597 188

Continued on next page

Table 7 – continued from previous page

	Name	N	nnz
81	olesnik0	88 263	744 216
82	OPF_10000	43 887	467 711
83	parabolic_fem	525 825	3 674 625
84	pdb1HYS	36 417	4 344 765
85	poisson3Db	85 623	2 374 949
86	pwtk	217 918	11 634 424
87	qa8fk	66 127	1 660 579
88	qa8fm	66 127	1 660 579
89	raefsky4	19 779	1 328 611
90	rail_79841	79 841	553 921
91	rajat30	643 994	6 175 377
92	rajat31	4 690 002	20 316 253
93	rma10	46 835	2 374 001
94	s3dkq4m2	90 449	4 820 891
95	shallow_water1	81 920	327 680
96	shallow_water2	81 920	327 680
97	ship_003	121 728	8 086 034
98	shipsec1	140 874	7 813 404
99	shipsec5	179 860	10 113 096
100	Si34H36	97 569	5 156 379
101	SiO2	155 331	11 283 503
102	sparsine	50 000	1 548 988
103	stomach	213 360	3 021 648
104	t3dh	79 171	4 352 105
105	t3dh_a	79 171	4 352 105
106	thermal1	82 654	574 458
107	thermal2	1 228 045	8 580 313
108	torso3	259 156	4 429 042
109	TSOPF_FS_b162_c4	40 798	2 398 220
110	TSOPF_FS_b39_c19	76 216	1 977 600
111	vanbody	47 072	2 336 898
112	venkat25	62 424	1 717 792
113	xenon1	48 600	1 181 120
114	xenon2	157 464	3 866 688

Table 8: Run times to solution required by SaP::GPU, PARDISO, SuperLU, and MUMPS, reported in milliseconds. For PARDISO, SuperLU, and MUMPS, a "-" sign indicates an instance in which the solver failed to solve that particular linear system. When SaP::GPU fails, OOM stands for "out of memory" and NC for "no convergence".

	Name	Run times (ms)			
		SaP::GPU	PARDISO	SuperLU	MUMPS
1	2cubes_sphere	1.899E2	2.830E3	1.430E4	1.883E4
2	2D_54019_highK	3.805E3	-	-	-
3	a2nnsnsl	OOM	3.283E2	5.000E2	-
4	a5esindl	OOM	1.480E2	2.400E2	-
5	ABACUS_shell_ud	NC	-	2.300E2	2.196E2
6	af_5_k101	2.059E4	3.639E3	4.926E4	1.647E4
7	af23560	7.273E2	-	8.500E2	7.375E2
8	ANCF31770	4.132E2	2.054E2	3.700E2	-
9	ANCF88950	1.057E3	5.132E2	8.400E2	-
10	apache1	2.643E3	6.761E2	2.790E3	2.107E3
11	apache2	OOM	9.295E3	1.091E5	3.884E4
12	appu	3.387E2	5.816E4	9.169E4	-
13	ASIC_100k	6.880E2	6.283E2	-	3.920E4
14	ASIC_100ks	4.141E2	5.566E2	-	1.209E3
15	av41092	OOM	-	-	3.757E3
16	bayer01	3.415E3	-	8.600E2	-
17	bcircuit	4.259E3	3.747E2	1.250E3	7.078E2
18	bcsstk39	1.051E3	3.971E2	1.370E3	1.071E3
19	blockqp1	1.778E2	4.372E2	1.020E3	-
20	bmw3_2	OOM	2.179E3	-	8.652E3
21	bmwcra_1	1.941E4	3.344E3	1.421E4	1.346E4
22	boyd1	1.825E3	7.744E3	2.096E4	-
23	bratu3d	3.019E2	3.948E2	1.080E3	-
24	bundle1	1.803E2	9.836E1	1.700E2	-
25	c-59	OOM	5.325E2	8.970E3	4.750E3
26	c-61	OOM	2.765E2	1.240E3	8.445E2
27	c-62	OOM	7.228E2	1.591E4	-
28	cant	1.373E3	1.451E3	3.100E3	3.735E3
29	case39	OOM	-	1.090E3	-
30	case39_A_01	OOM	-	1.160E3	-
31	c-big	OOM	5.440E3	-	-
32	cfd1	6.850E3	1.292E3	3.410E3	3.761E3
33	cfd2	1.038E4	2.455E3	6.490E3	9.109E3
34	circuit_4	OOM	-	-	3.313E2
35	ckt11752_tr_0	2.122E5	-	5.900E2	2.311E2
36	cont-201	1.400E3	-	1.560E3	-

Continued on next page

Table 8 – continued from previous page

	Name	Run times (ms)			
		SaP::GPU	PARDISO	SuperLU	MUMPS
37	cont-300	7.081E3	-	2.580E4	-
38	copter2	1.583E4	7.445E2	4.040E3	2.816E3
39	CurlCurl_4	OOM	-	6.920E3	8.754E3
40	dawson5	4.839E3	4.552E2	1.630E3	7.542E2
41	dc1	1.450E3	-	-	-
42	dixmaanl	3.998E2	1.739E2	4.900E2	3.881E2
43	Dubcova2	5.102E2	5.035E2	8.900E2	7.417E2
44	dw8192	1.600E3	-	2.400E2	-
45	ecl32	1.306E3	-	3.270E3	4.059E3
46	epb3	1.358E3	-	1.630E3	5.592E2
47	ex11	NC	5.212E2	-	8.534E2
48	ex19	5.889E3	-	-	8.557E1
49	FEM_3D_thermal1	1.559E2	3.072E2	6.200E2	-
50	filter3D	3.914E4	1.581E3	4.870E3	4.343E3
51	finan512	9.366E1	4.604E2	1.540E3	5.857E2
52	G3_circuit	8.263E3	1.010E4	1.910E6	4.383E4
53	g7jac140	OOM	-	2.410E3	3.750E3
54	Ga3As3H12	3.780E5	3.528E4	1.838E5	4.751E5
55	GaAsH6	1.157E5	3.710E4	1.766E5	5.153E5
56	garon2	2.928E2	1.376E2	2.900E2	1.665E2
57	gas_sensor	4.365E3	1.306E3	5.430E3	6.522E3
58	gridgena	1.043E3	3.323E2	6.000E2	5.287E2
59	gsm_106857	OOM	7.766E3	-	2.395E4
60	H2O	1.093E3	3.275E4	1.682E5	-
61	hcircuit	5.423E3	-	5.700E2	-
62	HTC_336_4438	OOM	-	3.970E3	6.779E2
63	ibm_matrix_2	1.478E4	-	3.760E3	-
64	inline_1	OOM	9.869E3	7.389E4	3.626E4
65	jan99jac120	NC	-	1.300E3	1.147E3
66	ldoor	OOM	9.608E3	4.746E5	3.518E4
67	lhr10c	5.416E2	-	2.900E2	1.660E2
68	Lin	8.163E4	8.733E3	5.622E4	5.614E4
69	lung2	3.831E2	-	1.240E3	4.693E2
70	mario002	OOM	1.931E3	9.375E4	-
71	mark3jac100	1.008E4	-	1.440E3	4.155E3
72	mark3jac140	1.303E4	-	-	7.057E3
73	matrix_9	8.893E2	-	2.222E4	-
74	minsurfo	1.218E2	1.726E2	6.600E2	2.920E2
75	msc23052	2.988E3	1.363E2	-	-
76	ncvxbqp1	5.332E3	3.248E2	1.040E3	7.535E2
77	nd24k	4.576E3	6.232E4	4.168E5	8.154E5
78	NetANCF40by40	5.608E2	6.149E2	6.900E2	6.461E2

Continued on next page

Table 8 – continued from previous page

	Name	Run times (ms)			
		SaP::GPU	PARDISO	SuperLU	MUMPS
79	offshore	OOM	5.800E3	3.338E4	3.026E4
80	oilpan	3.740E3	1.084E3	1.250E3	1.763E3
81	olesnik0	7.074E3	-	1.590E3	-
82	OPF_10000	4.635E3	-	4.600E2	3.754E2
83	parabolic_fem	1.132E4	3.158E3	1.695E5	6.120E3
84	pdb1HYS	4.348E3	9.211E2	-	3.354E3
85	poisson3Db	1.361E3	-	8.610E3	1.009E4
86	pwtk	1.355E4	1.792E3	7.380E3	6.869E3
87	qa8fk	1.375E3	-	4.720E3	-
88	qa8fm	1.732E2	1.236E3	4.670E3	6.683E3
89	raefsky4	6.230E3	2.674E2	-	-
90	rail_79841	1.402E3	4.115E2	7.300E2	6.852E2
91	rajat30	6.414E3	-	-	-
92	rajat31	2.022E4	3.161E4	-	-
93	rma10	1.654E3	-	1.150E3	5.840E2
94	s3dkq4m2	2.884E3	1.385E3	3.710E3	3.852E3
95	shallow_water1	6.940E1	4.238E2	1.320E3	1.237E3
96	shallow_water2	9.859E1	3.862E2	1.300E3	8.518E2
97	ship_003	2.356E4	4.211E3	2.084E4	2.761E4
98	shipsec1	4.926E4	2.926E3	1.098E4	1.266E4
99	shipsec5	NC	3.807E3	1.859E4	1.937E4
100	Si34H36	OOM	1.118E5	-	1.620E6
101	SiO2	5.196E3	3.544E5	-	5.940E6
102	sparsine	NC	5.760E4	2.450E5	5.218E5
103	stomach	7.074E2	-	2.519E4	1.001E5
104	t3dh	1.459E4	-	1.539E4	-
105	t3dh_a	1.462E4	-	1.560E4	-
106	thermal1	1.477E3	4.087E2	7.700E2	8.733E2
107	thermal2	1.482E5	8.112E3	-	1.759E4
108	torso3	5.410E3	-	-	6.761E4
109	TSOPF_FS_b162_c4	OOM	-	4.830E3	-
110	TSOPF_FS_b39_c19	OOM	-	2.900E3	-
111	vanbody	5.213E3	3.543E2	-	8.035E2
112	venkat25	4.182E3	-	1.160E3	5.768E2
113	xenon1	4.086E3	1.006E3	2.240E3	2.560E3
114	xenon2	3.354E3	4.459E3	1.294E4	1.680E4

Table 9: Run times to solution required by SaP::GPU and cuSOLVER, reported in milliseconds. A "-" sign indicates a solver failure in solving a certain linear system.

	Name	SaP::GPU	cuSOLVER	
			w/o CM	**w/** CM
1	2cubes_sphere	1.899E2	-	-
2	2D_54019_highK	3.805E3	-	-
3	a2nnsnsl	-	-	-
4	a5esindl	-	-	-
5	ABACUS_shell_ud	-	1.371E3	-
6	af_5_k101	2.059E4	-	-
7	af23560	7.273E2	3.398E3	3.576E3
8	ANCF31770	4.132E2	1.120E3	1.128E5
9	ANCF88950	1.057E3	5.000E3	-
10	apache1	2.643E3	2.507E4	-
11	apache2	-	-	-
12	appu	3.387E2	-	-
13	ASIC_100k	6.880E2	-	-
14	ASIC_100ks	4.141E2	-	-
15	av41092	-	-	-
16	bayer01	3.415E3	-	-
17	bcircuit	4.259E3	5.951E3	-
18	bcsstk39	1.051E3	6.356E3	4.761E3
19	blockqp1	1.778E2	-	-
20	bmw3_2	-	-	-
21	bmwcra_1	1.941E4	-	-
22	boyd1	1.825E3	-	-
23	bratu3d	3.019E2	9.900E3	-
24	bundle1	1.803E2	-	-
25	c-59	-	-	-
26	c-61	-	-	-
27	c-62	-	-	-
28	cant	1.373E3	8.742E3	7.895E3
29	case39	-	-	-
30	case39_A_01	-	-	-
31	c-big	-	-	-
32	cfd1	6.850E3	-	-
33	cfd2	1.038E4	-	-
34	circuit_4	-	-	-
35	ckt11752_tr_0	2.122E5	2.123E5	6.945E4
36	cont-201	1.400E3	1.384E3	1.109E4
37	cont-300	7.081E3	-	-
38	copter2	1.583E4	-	-

Continued on next page

Table 9 – continued from previous page

	Name	SaP::GPU	cuSOLVER	
			w/o CM	**w/** CM
39	CurlCurl_4	-	-	-
40	dawson5	4.839E3	7.836E3	1.828E4
41	dc1	1.450E3	-	-
42	dixmaanl	3.998E2	8.235E2	-
43	Dubcova2	5.102E2	1.865E4	-
44	dw8192	1.600E3	2.162E3	3.249E2
45	ecl32	1.306E3	5.090E4	-
46	epb3	1.358E3	8.472E3	4.014E3
47	ex11	-	4.252E3	6.155E3
48	ex19	5.889E3	4.048E2	4.021E2
49	FEM_3D_thermal1	1.559E2	2.596E4	2.438E3
50	filter3D	3.914E4	-	-
51	finan512	9.366E1	4.192E3	-
52	G3_circuit	8.263E3	-	-
53	g7jac140	-	-	-
54	Ga3As3H12	3.780E5	-	-
55	GaAsH6	1.157E5	-	-
56	garon2	2.928E2	1.169E3	2.339E5
57	gas_sensor	4.365E3	-	-
58	gridgena	1.043E3	4.315E3	7.840E3
59	gsm_106857	-	-	-
60	H2O	1.093E3	-	-
61	hcircuit	5.423E3	3.989E4	-
62	HTC_336_4438	-	-	-
63	ibm_matrix_2	1.478E4	-	-
64	inline_1	-	-	-
65	jan99jac120	-	1.295E5	1.160E5
66	ldoor	-	-	-
67	lhr10c	5.416E2	4.549E4	2.646E4
68	Lin	8.163E4	-	-
69	lung2	3.831E2	2.628E3	2.658E5
70	mario002	-	-	-
71	mark3jac100	1.008E4	4.068E4	1.274E4
72	mark3jac140	1.303E4	5.925E4	1.804E4
73	matrix_9	8.893E2	-	-
74	minsurfo	1.218E2	2.394E3	3.478E3
75	msc23052	2.988E3	-	1.220E4
76	ncvxbqp1	5.332E3	3.580E4	-
77	nd24k	4.576E3	-	-
78	NetANCF40by40	5.608E2	1.219E4	-
79	offshore	-	-	-
80	oilpan	3.740E3	-	7.322E4

Continued on next page

Table 9 – continued from previous page

	Name	SaP::GPU	cuSOLVER	
			w/o CM	w/ CM
81	olesnik0	7.074E3	-	-
82	OPF_10000	4.635E3	1.250E3	1.075E4
83	parabolic_fem	1.132E4	-	-
84	pdb1HYS	4.348E3	4.569E4	-
85	poisson3Db	1.361E3	-	-
86	pwtk	1.355E4	-	-
87	qa8fk	1.375E3	-	-
88	qa8fm	1.732E2	-	5.096E4
89	raefsky4	6.230E3	-	-
90	rail_79841	1.402E3	9.077E3	-
91	rajat30	-	-	-
92	rajat31	2.022E4	-	-
93	rma10	1.654E3	4.818E3	-
94	s3dkq4m2	2.884E3	-	2.920E4
95	shallow_water1	6.940E1	1.118E4	7.923E4
96	shallow_water2	9.859E1	1.101E4	7.900E4
97	ship_003	2.356E4	-	-
98	shipsec1	4.926E4	-	-
99	shipsec5	-	-	-
100	Si34H36	-	-	-
101	SiO2	5.196E3	-	-
102	sparsine	-	-	-
103	stomach	7.074E2	-	-
104	t3dh	1.459E4	-	-
105	t3dh_a	1.462E4	-	-
106	thermal1	1.477E3	5.195E3	-
107	thermal2	1.482E5	-	-
108	torso3	5.410E3	-	-
109	TSOPF_FS_b162_c4	-	-	-
110	TSOPF_FS_b39_c19	-	-	-
111	vanbody	5.213E3	-	-
112	venkat25	4.182E3	3.138E4	2.141E5
113	xenon1	4.086E3	6.709E4	-
114	xenon2	3.354E3	-	-

References

[1] Paralution. http://www.paralution.com.

[2] SaP::GPU Github. https://github.com/spikegpu/SaPLibrary. Accessed: 2015-02-07.

[3] SaP::GPU Website. `http://sapgpu.sbel.org/`. Accessed: 2015-02-07.

[4] HSL: A collection of Fortran codes for large-scale scientific computation. http://www.cse.clrc.ac.uk/nag/hsl, 2011.

[5] NVIDIA TESLA KEPLER GPU accelerators, 2012.

[6] Tesla K20 GPU Accelerator, 2012.

[7] cuSOLVER. https://developer.nvidia.com/cusolver, 2015.

[8] MUMPS: a MUltifrontal Massively Parallel sparse direct Solver. `http://mumps.enseeiht.fr`, 2015.

[9] N. Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2012. Version 0.3.0.

[10] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. John Wiley & Sons, New York, 1978.

[11] R. Burkhard and U. Derigs. *Assignment and matching problems: Solution methods with FORTRAN-programs*. Springer-Verlag New York, Inc., 1980.

[12] G. Carpaneto and P. Toth. Algorithm 548: Solution of the assignment problem [h]. *ACM Transactions on Mathematical Software (TOMS)*, 6(1):104–111, 1980.

[13] P. Carraresi and C. Sodini. An efficient algorithm for the bipartite matching problem. *European journal of operational research*, 23(1):86–93, 1986.

[14] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th ACM Conference*, pages 157–172, New York, 1969.

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

[16] J. W. Demmel. SuperLU Users' Guide. *Lawrence Berkeley National Laboratory*, 2011.

[17] U. Derigs and A. Metz. An efficient labeling technique for solving sparse assignment problems. *Computing*, 36(4):301–311, 1986.

[18] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[19] I. Duff. Algorithm 575: Permutations for a zero-free diagonal [F1]. *ACM Transactions on Mathematical Software (TOMS)*, 7(3):387–390, 1981.

[20] I. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Analysis and Applications*, 20(4):889–901, 1999.

[21] I. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Analysis and Applications*, 22(4):973, 2001.

[22] L. Fang. *A Primal-Dual Interior Point Method for Solving Multibody Dynamics Problems with Frictional Contact*. M.S. thesis, Department of Mechanical Engineering, University of Wisconsin–Madison, `http://sbel.wisc.edu/documents/Luning_master_thesis.pdf`, 2015.

[23] L. Fang and D. Negrut. An Analysis of a Primal-Dual Interior Point Method for Computing Frictional Contact Forces in a Differential Inclusion-Based Approach for Multibody Dynamics. Technical Report TR-2014-13: `http://sbel.wisc.edu/documents/TR-2014-13.pdf`, Simulation-Based Engineering Laboratory, University of Wisconsin-Madison, 2014.

[24] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, Maryland, 1980.

[25] J. Hoberock and N. Bell. Thrust: A parallel template library, 2010. Version 1.7.0.

[26] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987.

[27] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzaran, and K. Pingali. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 921–932. IEEE, 2014.

[28] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[29] A. Li, O. Deshmukh, R. Serban, and D. Negrut. A Comparison of the Performance of SaP::GPU and Intel's Math Kernel Library for Solving Dense Banded Linear Systems. Technical Report TR-2012-07: `http://sbel.wisc.edu/documents/TR-2014-07.pdf`, Simulation-Based Engineering Laboratory, University of Wisconsin-Madison, 2014.

[30] A. Li, H. Mazhar, R. Serban, and D. Negrut. Comparison of SPMV performance on matrices with different matrix format using CUSP, cuSPARSE and ViennaCL. Technical Report TR-2015-02–`http://sbel.wisc.edu/documents/TR-2015-02.pdf`, SBEL, University of Wisconsin - Madison, 2015.

[31] A. Li, R. Serban, and D. Negrut. An implementation of a reordering approach for increasing the product of diagonal entries in a sparse matrix. Technical Report TR-2014-01: `http://sbel.wisc.edu/documents/TR-2014-01.pdf`, Simulation-Based Engineering Laboratory, University of Wisconsin-Madison, 2014.

[32] M. Manguoglu, A. Sameh, and O. Schenk. PSPIKE: A parallel hybrid sparse linear system solver. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 797–808, Delft, The Netherlands, 2009. Springer-Verlag.

[33] C. Mikkelsen and M. Manguoglu. Analysis of the truncated SPIKE algorithm. *SIAM J. Matrix Analysis Applications*, 30(4):1500–1519, 2008.

[34] D. Negrut, R. Serban, A. Li, and A. Seidl. Unified Memory in CUDA 6.0. A Brief Overview of Related Data Access and Transfer Issues. Technical Report TR-2014-09: `http://sbel.wisc.edu/documents/`

TR-2014-09.pdf, Simulation-Based Engineering Laboratory, University of Wisconsin-Madison, 2014.

[35] NVIDIA. CUDA Programming Guide. Available online at http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2015.

[36] M. Olschowka and A. Neumaier. A new pivoting strategy for Gaussian elimination. *Linear Algebra and its Applications*, 240:131–151, 1996.

[37] D. A. Padua Haiek. *Multiprocessors: Discussion of Some Theoretical and Practical Problems*. PhD thesis, Champaign, IL, USA, 1980. AAI8018194.

[38] E. Polizzi and A. Sameh. A parallel hybrid banded system solver: the SPIKE algorithm. *Parallel Computing*, 32(2):177–194, 2006.

[39] E. Polizzi and A. Sameh. SPIKE: A parallel environment for solving banded linear systems. *Computers & Fluids*, 36(1):113 – 120, 2007.

[40] Khronos OpenCL Working Group. The OpenCL specification, 2008.

[41] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2003.

[42] A. Sameh and D. Kuck. On stable parallel linear system solvers. *JACM*, 25(1):81–91, 1978.

[43] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with Pardiso. *Future Generation Computer Systems*, 20(3):475–487, 2004.

[44] O. Schenk, K. Gartner, W. Fichtner, and A. Stricker. PARDISO: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems*, 18(1):69–78, 2001.

[45] R. Serban, D. Melanz, A. Li, I. Stanciulescu, P. Jayakumar, and D. Negrut. A GPU-based preconditioned Newton-Krylov solver for flexible multibody dynamics. *International Journal for Numerical Methods in Engineering*, 102:1585–1604, 2015.

[46] G. Sleijpen and D. Fokkema. BiCGStab(l) for linear equations involving unsymmetric matrices with complex spectrum. *Electornic Transactions on Numerical Analysis*, 1:11–32, 1993.