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

ANG LI†, RADU SERBAN‡, AND DAN NEGRUT†‡

Abstract. We discuss an approach for solving sparse or dense banded linear systems \(Ax = b\) on a graphics processing unit (GPU) card. The matrix \(A \in \mathbb{R}^{N \times N}\) 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 \(A\) into diagonal subblocks \(A_i\), \(i = 1, \ldots, P\), which are independently factored in parallel. The solution may choose to consider or to ignore the matrices that couple the diagonal subblocks \(A_i\). This approach, along with the Krylov-subspace-based iterative method that it preconditioned, 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 for several stages involved in preliminary row and 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 fared 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 BSD-3 license.

Key words. sparse linear system solution, parallel computing, GPU computing, Krylov-subspace method, preconditioning, work splitting, matrix reordering

AMS subject classifications. 65F10, 65F50, 65Y05, 65Y10

DOI. 10.1137/15M1039523

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 [22]. Owing also to the release of the OpenCL specification [26] 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 the 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 computing explain the scarcity of solutions for solving \(Ax = b\) on the GPU, when \(A \in \mathbb{R}^{N \times N}\) 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 for solving linear systems on the GPU: cuSOLVER [23], Paralution [18], and SuperLU [7], the latter focused on distributed memory architectures and leveraging GPU computing at the node level only. Several CPU multicore approaches exist and are well established; see, for instance, [12, 32, 3, 7]. For a domain-specific application implemented on the GPU that calls for solving \(Ax = b\), 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 presently 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 solution, including factorization, on the GPU when the possibly nonsymmetric matrix $A$ 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 an LU factorization is memory bound, particularly for sparse systems, the GPU is attractive due to its high bandwidths and relatively low latencies. At roughly 300 GB/s, the GPU has four to five times higher main-memory bandwidth when compared to 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 multinode computing in [24]. 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 pivoting is necessary in the LU factorization, thus avoiding logic and branching tasks at which the GPU does not shine. 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 $Ax = b$ 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 $\hat{A}y = \hat{b}$ on the GPU, where $\hat{A} \in \mathbb{R}^{N \times N}$ is a dense banded matrix obtained from $A$ after a sequence of possibly two reordering stages that can include element drop-off. Regardless of whether $A$ is sparse, 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 $\hat{A}$, 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 $\hat{A}$ in several diagonal blocks that are factored independently and a correction process to account for the interdiagonal block coupling. The implementation takes advantage of the GPU’s deep memory hierarchy, its multi-stream multiprocessor (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 dense banded matrix case. Subsequently, the sparse case brings into focus strategies for matrix reordering. Section 3 summarizes aspects related to the GPU implementation of the proposed solution methods. Results from numerical experiments for both dense banded and sparse linear systems are reported in section 4. The paper concludes with 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 $A \in \mathbb{R}^{N \times N}$ has half-bandwidth $K \ll N$. Following an approach discussed
in [28, 24, 25], we partition the banded matrix $A$ into a block-tridiagonal form with $P$ diagonal blocks $A_i \in \mathbb{R}^{N_i \times N_i}$, where $\sum_i N_i = N$. For each partition $i$, let $B_i$, $i = 1, \ldots, P-1$, and $C_i$, $i = 2, \ldots, P$, be the super- and subdiagonal coupling blocks, respectively—see Figure 1. Each coupling block has dimension $K \times K$ for banded matrices with half-bandwidth $K = \max_{i,j, a_i \neq 0} |i-j|$. As illustrated in Figure 1, the banded matrix $A$ is expressed as the product of a block-diagonal matrix $D$ and a so-called spike matrix $S$ [28]. The latter is made up of identity diagonal blocks of dimension $N_i$ and off-diagonal spike blocks, each having $K$ columns. Specifically,

$A = DS$,

where $D = \text{diag}(A_1, \ldots, A_P)$. Assuming that $A_i$ are nonsingular, the so-called left and right spikes $W_i$ and $V_i$ associated with partition $j$, each of dimension $N_i \times K$, are given by

\begin{align*}
A_1 V_1 &= \begin{bmatrix} 0 \\ 0 \\ B_1 \end{bmatrix}, \\
A_i [W_i | V_i] &= \begin{bmatrix} C_i & 0 \\ 0 & 0 \\ 0 & B_i \end{bmatrix}, \quad i = 2, \ldots, P-1, \\
A_P W_P &= \begin{bmatrix} C_P \\ 0 \end{bmatrix}.
\end{align*}

Solving the linear system $Ax = b$ is thus reduced to solving

\begin{align*}
Dg &= b, \\
Sx &= g.
\end{align*}

Since $D$ is block-diagonal, solving for the modified right-hand-side $g$ from (2.3) is trivially parallelizable, as the work is split across $P$ processes, each charted to solve $A_i g_i = b_i$, $i = 1, \ldots, P$. Note that the same decoupling is manifest in (2.2) and the work is also spread over $P$ processes.

The remaining question is how to efficiently solve the linear system in (2.4). This problem can be reduced to one of smaller size, $\hat{S}\hat{x} = \hat{g}$. To that end, the spikes $V_i$
and $\mathbf{W}_i$, as well as the modified right-hand-side $\mathbf{g}_i$ and the unknown vectors $\mathbf{x}_i$ in (2.4) are partitioned into their top $K$ rows, the middle $N_i - 2K$ rows, and the bottom $K$ rows as

\[
\begin{align*}
\mathbf{V}_i &= \begin{bmatrix} \mathbf{V}_i^{(t)} \\ \mathbf{V}_i^{(b)} \end{bmatrix}, \\
\mathbf{W}_i &= \begin{bmatrix} \mathbf{W}_i^{(t)} \\ \mathbf{W}_i^{(b)} \end{bmatrix}, \\
\mathbf{g}_i &= \begin{bmatrix} \mathbf{g}_i^{(t)} \\ \mathbf{g}_i^{(b)} \end{bmatrix}, \\
\mathbf{x}_i &= \begin{bmatrix} \mathbf{x}_i^{(t)} \\ \mathbf{x}_i^{(b)} \end{bmatrix}.
\end{align*}
\]

A block-tridiagonal reduced system is obtained by eliminating the middle partitions of the spike matrices, resulting in the linear system

\[
\begin{bmatrix}
\mathbf{R}_1 & \mathbf{M}_1 & \\
\vdots & \ddots & \\
\mathbf{N}_i & \mathbf{R}_i & \mathbf{M}_i & \\
\vdots & \ddots & & \\
\mathbf{N}_{P-1} & \mathbf{R}_{P-1}
\end{bmatrix}
\begin{bmatrix}
\mathbf{x}_1 \\ \vdots \\ \mathbf{x}_i \\ \vdots \\ \mathbf{x}_{P-1}
\end{bmatrix}
= \begin{bmatrix}
\mathbf{g}_1 \\ \vdots \\ \mathbf{g}_i \\ \vdots \\ \mathbf{g}_{P-1}
\end{bmatrix},
\]

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

\[
\begin{align*}
\mathbf{N}_i &= \begin{bmatrix} \mathbf{W}_i^{(b)} & 0 \\ 0 & 0 \end{bmatrix}, \\
\mathbf{R}_i &= \begin{bmatrix} \mathbf{I}_M & \mathbf{V}_i^{(b)} \\ \mathbf{W}_i^{(t)} & \mathbf{I}_M \end{bmatrix}, \\
\mathbf{M}_i &= \begin{bmatrix} 0 & 0 \\ 0 & \mathbf{V}_i^{(t)} \end{bmatrix},
\end{align*}
\]

and

\[
\mathbf{\hat{x}}_i = \begin{bmatrix} \mathbf{x}_i^{(b)} \\ \mathbf{x}_i^{(t)} \end{bmatrix}, \\
\mathbf{\hat{g}}_i = \begin{bmatrix} \mathbf{g}_i^{(b)} \\ \mathbf{g}_i^{(t)} \end{bmatrix},
\]

Two strategies are proposed in [24] to solve (2.6): (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 $\mathbf{\hat{S}}$. The solution approach adopted herein is based on (ii) and therefore each subsystem $\mathbf{R}_i\mathbf{\hat{x}}_i = \mathbf{\hat{g}}_i$ is solved independently using the following steps:

\[
\begin{align*}
\text{(2.9a)} & \quad \text{Form } \mathbf{\hat{R}}_i = \mathbf{I}_M - \mathbf{W}_i^{(t)} \mathbf{V}_i^{(b)} \\
\text{(2.9b)} & \quad \text{Solve } \mathbf{\hat{R}}_i \mathbf{\hat{x}}_i^{(t)} = \mathbf{g}_i^{(t)} - \mathbf{W}_i^{(t)} \mathbf{g}_i^{(b)} \\
\text{(2.9c)} & \quad \text{Calculate } \mathbf{\hat{x}}_i^{(b)} = \mathbf{g}_i^{(b)} - \mathbf{V}_i^{(b)} \mathbf{\hat{x}}_i^{(t)}.
\end{align*}
\]

Note that a tilde was used to denote the approximate values $\mathbf{\hat{x}}_i^{(t)}$ and $\mathbf{\hat{x}}_i^{(b)}$ 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 $P$ systems independently and in parallel using the available LU factorizations of the $\mathbf{A}_i$ matrices:
ANALYSIS OF SaP SOLUTION OF LINEAR SYSTEMS ON GPU

(2.10a) \[ A_1 x_1 = b_1 - \begin{bmatrix} 0 \\ 0 \\ B_1 \tilde{x}^{(t)}_2 \end{bmatrix}, \]

(2.10b) \[ A_i x_i = b_i - \begin{bmatrix} C_i \tilde{x}^{(b)}_{i-1} \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ B_i \tilde{x}^{(t)}_{i+1} \end{bmatrix}, \quad i = 2, \ldots, P - 1, \]

(2.10c) \[ A_P x_P = b_P - \begin{bmatrix} C_P \tilde{x}^{(b)}_{P-1} \\ 0 \\ 0 \end{bmatrix}. \]

Three remarks are in order. First, note that if an LU factorization of the diagonal blocks \( A_i \) is available, the bottom block of the right spike, i.e., \( V_i^{(b)} \), can be obtained from (2.2a) 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 \( W_i \). An effective alternative is to perform an additional UL factorization of \( A_i \), in which case \( W_i^{(t)} \) can be obtained using only the top \( K \times K \) blocks of the new U and L.

Next, note that the decision to set \( N_i \equiv 0 \) and \( M_i \equiv 0 \), i.e., using the approximate reduction rather than the exact reduction, relegates the algorithm to preconditioner status. This decision is justified as follows: although the dimension of the reduced linear system in (2.6) is smaller than that of the original problem, its half-bandwidth is at least three times larger. The memory footprint of exactly solving (2.6), i.e., employing the exact reduction, is large, thus limiting the size of problems that can be tackled on the GPU due to the relatively small amount of memory available on the device. 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 is manifest that the quality of the preconditioner is dictated by the entries in \( N_i \) and \( M_i \). For the sake of this discussion, assume that the matrix \( A \) is diagonally dominant with a degree of diagonal dominance \( d \geq 1 \), i.e.,

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

When \( d > 1 \), the elements of the left spikes \( W_i \) decay in magnitude from top to bottom; those of the right spikes \( V_i \) decay from bottom to top [21]. This decay, which is more pronounced the larger the degree of diagonal dominance, justifies the approximation \( N_i \equiv 0 \) and \( M_i \equiv 0 \). However, having \( A \) be diagonal dominant, although desirable, is not mandatory, as demonstrated by numerical experiments reported herein. Truncating when \( d < 1 \) will lead to a preconditioner of lesser quality.

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 \( A \) is split into block-diagonal matrices \( A_i \), 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, the solution path would implement the exact reduction; as justified above, this strategy is not pursued herein. At the other end of the spectrum, SaP : : GPU solves the block-diagonal linear system in 2.3 and for preconditioning purposes uses the approximation \( x \approx g \). In what follows, this will be called the decoupled approach, SaP : : GPU-D. The middle ground is the approximate reduction, which sets \( N_i \equiv 0 \) and \( M_i \equiv 0 \). This will be called the coupled approach,
SaP::GPU-C, owing to the coupling that occurs through the truncated spikes, i.e., $V_i^{(b)}$ and $W_{i+1}^{(t)}$.

Neither the coupled nor the decoupled approaches qualify as direct solvers. Consequently, SaP::GPU employs an outer Krylov subspace scheme to solve $Ax = b$. The solver uses BiCGStab($\ell$) [34] and left-preconditioning, unless the matrix $A$ is symmetric and positive definite, in which case the outer loop implements a conjugate gradient method [27]. SaP::GPU is open source and available at [30, 31].

2.2. The sparse linear system case. The discussion focuses next on solving $A_sx = b$, where $A_s \in \mathbb{R}^{N \times N}$ is assumed to be sparse. The salient attribute of the solution strategy is its fallback on the dense banded approach of section 2.1. Specifically, several permutation steps are employed to transform $A_s$ into a matrix $A$ that has a large $d$ and small $K$. Although the reordered matrix will remain sparse within the band, it will be regarded as dense banded and LU- and/or UL-factored accordingly.

When $A_s$ is nonsymmetric or has low $d$, a first reordering, called diagonal boosting (DB), is applied as $QA_sx = Qb$ to maximize the product of the absolute values of the diagonal entries [9, 10]. This is done via a depth first search with a look-ahead technique similar to the one in the Harwell Software Library (HSL) [12]. A detailed description of the DB implementation in SaP is available in [16]. Therein, we provide a performance comparison against HSL on a set of more than 100 matrices.

While the purpose of the first reordering $QA_s$ is to yield a diagonally “heavy” matrix, a second reordering seeks to reduce $K$ by using the traditional Cuthill–McKee (CM) algorithm [6]. Since the diagonal entries should not be relocated, the second permutation is applied to the symmetric matrix $QA_s + A_s^TQ^T$. A detailed description of the CM implementation in SaP is available in [15]. Therein, we provide a performance comparison against HSL on a set of more than 130 matrices.

Following these two reorderings, the resulting matrix $A$ is split to obtain $A_1$ through $A_P$. A third CM reordering is then applied to each $A_i$ for further bandwidth reduction. 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 $A_i$ is diagonally dominant, the LU and/or UL factorization can be safely carried out without pivoting [11]. We always perform factorizations of the diagonal blocks $A_i$ 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 $\epsilon$. This yields a factorization of a slightly perturbed diagonal block, $L_iU_i = A_i + \delta A_i$, where $\|\delta A_i\| = O(u\|A\|)$ and $u$ is the unit roundoff [19]. The semantics of diagonal boosting and pivot boosting are different. Herein, the former refers to a matrix reordering; the latter concerns a modification of a pivot during the LU process.

3. Brief implementation details. This section discusses SaP::GPU implementation details for solving both dense banded and sparse linear systems. A discussion of the DB and CM reordering algorithms and their GPU implementation falls outside the scope of this contribution. The interested reader is referred to [17].

3.1. Dense banded matrix factorization details. This subsection provides implementation details regarding how the $P$ partitions $A_i$ are determined, how the banded matrix $A$ 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 \( A_i \) for \( i = 1, \ldots, P \) can be done independently and simultaneously. On the other hand, a large \( P \) negatively impacts the quality of the resulting preconditioner due to the increase in the number of instances in which the coupling of the diagonal blocks \( A_i \) and \( A_{i+1} \) is approximated. In the current implementation, no attempt is made to optimally select \( P \) and some experimentation is required. Given a \( P \) value, the size of the diagonal blocks \( A_i \) 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. Dense banded matrices \( A_i \) are stored in a “tall and thin” 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 \( A_i \). As \( i = 1, \ldots, P \), these \( P \) LU/UL factorizations can proceed independently and in parallel. One important question is whether \( A_i \) can be factored using one block of threads. The answer is relevant given the GPU’s lack of native, low overhead, support for synchronization between threads running in different blocks. The established GPU strategy for interblock synchronization is “exit and launch a new kernel.” This guarantees synchronization at the GPU-grid level at the cost of nonnegligible overhead. In a trade-off between minimizing the overhead of kernel launches and maximizing the occupancy of the GPU, we adopted two, mutually exclusive, execution paths: one for \( K < 64 \) and one for larger bandwidths. As a side note, the threshold value of 64 was selected through numerical experimentation over a variety of problems and was dictated by the number of threads that can be organized in a block in CUDA [22].

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 \( A_i = L_i U_i \) by updating entries in a \((K + 1) \times (K + 1)\) window of elements, a single kernel is launched to factor \( A_i \). It uses \( \min(K^2, 1024) \) threads per block and relies on low-overhead SM 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 \( A_i \) 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 \( 64 \times 64 \). 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 subblocks $B_i$ and $C_i$ are used to compute the entire spikes in a scheme that renders a UL factorization superfluous. In this case, $B_i$ and $C_i$ are each first partitioned into subblocks 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 subblock, rather than a vector of length $K$, is calculated. To this end, the corresponding elements in the matrix $A_i$ are prefetched into shared memory and the entries of the subblock 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.** SaP::GPU uses a mixed-precision implementation. Single precision is used in obtaining and applying the preconditioner. Double precision is used in the sparse matrix vector multiplications embedded in the Krylov iterative component of the solution. A battery of tests indicates 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. SaP::GPU components and computational flow.

The SaP::GPU dense banded linear system solver is relatively straightforward to implement. Upon partitioning $A$ into diagonal blocks $A_i$, each $A_i$ is subject to an LU factorization that requires an amount of time $T_{LU}$. Next, in $T_{BC}$ time, the coupling block matrices $B_i$ and $C_i$ are extracted on the GPU. The $V_i$ and $W_i$ spikes are subsequently computed in an operation that requires $T_{SPK}$ time. Afterward, in $T_{LUrdcd}$ time, the spikes are truncated and the steps outlined in (2.9) are taken to produce the intermediary values $\tilde{x}_i^{(t)}$ and $\tilde{x}_i^{(b)}$. At this point, the preprocessing step is over and two sets of factorizations, for $A_i$ and $R_i$, 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 SaP::GPU 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, is carried out to increase the size of the diagonal elements and reduce bandwidth. An additional amount of time $T_{Drop}$ might be spent to drop off-diagonal elements in order to decrease the bandwidth of the reordered $A$ 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 bookkeeping required to turn the reordered sparse matrix into a dense banded matrix.

The definition of the times to solution is tied to the computational flow in Figure 2, where subscripts $d$ and $s$ are used to differentiate between the dense and sparse paths, respectively. For a sparse linear system solve that uses the coupled approach, i.e.,
**ANALYSIS OF SaP SOLUTION OF LINEAR SYSTEMS ON GPU**

**Fig. 2.** Computational flow for SaP::GPU. SaP::GPU-D is the decoupled dense solver; SaP::GPU-C is the coupled dense solver. The analogue sparse solvers have an s subscript. The third-stage reordering in square brackets is optional but typically leads to substantial reductions in time to solution.

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 a CM process that reduces the bandwidth of each $A_i$. The names introduced, i.e., $T_{DB}$, $T_{CM}$, $T_{LUrdcd}$, etc., are referenced in the profiling study discussed in section 4.2.1 and used *ad verbum* on the SaP::GPU web page [31] to report profiling results for approximately 120 linear systems.

### 3.3. Solver parameter selection aspects.

No analytically backed strategy is available in SaP::GPU to select (i) the bandwidth $K$ that a sparse linear system should be solved with, (ii) the number of partitions $P$, or (iii) the SaP::GPU-C or SaP::GPU-D flavor. These choices are both problem and GPU architecture specific. Insofar as the architecture is concerned, GPUs with a large number of SM will work best with large values of $P$, which will ensure that each SM will be assigned a task and avoid hardware underutilization. For sparse linear systems, the user has to specify the preferred $K$ value; if none is specified, the default value implies no drop-off, which can lead to large bandwidths. Note that $K$ selection is relevant only in the context of solving sparse linear systems; for dense banded, $K$ is a given. We rely on a SaP utility that for a given linear system, via a parametric study, helps the user identify the best $P$, $K$ (for sparse problems) and SaP::GPU-C versus SaP::GPU-D choice. Obviously, if one is interested in solving a linear system once, this utility is of no use—the solver will fall back on default choices, which can be controlled through the SaP interface. However, when SaP is used in the context of GPU applications where numerous and qualitatively similar matrices are handled, the upfront effort of identifying a suitable parameter choice pays off. For instance, SaP::GPU is well positioned to tackle nonlinear computational dynamics applications where one would repeatedly solve a nonlinear algebraic system using a Newton method [33]. The latter calls for the solution of a sparse linear system several times per time step. Then, presuming that the dimension of the matrix does not change over time and that the sparsity pattern stays roughly the same, a set of $P$, $K$ and solver flavor determined at the beginning of the analysis is used throughout the simulation.
4. Numerical experiments. The next two subsections summarize results from two numerical experiments concerned with the solution of dense banded and sparse linear systems, respectively. The hardware/software setup for these numerical experiments is as follows. The GPU used was Tesla K20X [2, 1]. SaP::GPU uses CUDA 7.0 [22], cusp [4], and Thrust [5]. The CPU used was the 3 GHz, 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 amount of memory per node was 64 GB. When invoking the multithreaded libraries Intel MKL v-13.0.1, PARDISO [32], MUMPS [3], or SuperLU [7], they used both CPUs on the node. SaP always ran on one GPU card.

When reporting below the results of several numerical experiments, one legitimate question is whether it is justifiable 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 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 was. Finally, Kepler is not the latest microarchitecture either, as Nvidia released the Maxwell architecture in early 2014 and Pascal in 2016. 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 purpose of this section is to gauge the performance of SaP::GPU relative to that of existing solvers, in other words, to understand where this GPU solver fits in the big picture and what levels of expectations one should have in relation to its use. In the interest of full disclosure, we point out that being an iterative solver, some parameters need to be set for SaP, i.e., $P$, $K$, and strategy, i.e., SaP::GPU-C versus SaP::GPU-D—see the discussion in section 3.3. Herein, we rely on a SaP utility to select these parameters, which are both problem and hardware dependent. This might give SaP an unfair advantage comparing against PARDISO, MUMPS, and SuperLU as no attempt has been made to optimize the behavior of these direct solvers. A GPU-to-GPU comparison was also carried out and the results are reported in section 4.2.3. We found the performance of the GPU alternative, i.e., cuSOLVER, lacking to the point where it could not provide that “placing SaP in the big picture” aspect that was met by using the CPU and direct solvers mentioned above.

Unless otherwise stated, all times reported below are in seconds and were obtained on a dedicated machine. In an attempt to avoid warm-up overhead, the results represent averages that drew on multiple successive identical runs. SaP stops its iterative solution as soon as the condition

\[
\|A\tilde{x} - b\|_2 \leq atol + rtol \cdot \|b\|_2
\]

is satisfied; here $\tilde{x}$ is a solution candidate produced during the iterative process. Unless specified by the user, SaP uses the default values $atol = 0$ and $rtol = 10^{-10}$. These default values were used for all numerical experiments reported herein.

4.1. Numerical experiments related to dense banded linear systems. SaP::GPU-D and SaP::GPU-C are entirely implemented on the GPU. The
subscript $d$ will be dropped for convenience. This discussion draws on ample numerical experiments reported in [13].

4.1.1. Sensitivity with respect to $P$. The results in Figure 3 summarize the sensitivity of the time to solution with respect to the number of partitions $P$. This behavior, namely, relatively small gains after a threshold value of $P$, is typical. It is instructive to see how the solution time is spent by $\text{SaP::GPU-C}$ and $\text{SaP::GPU-D}$ 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, $D_{\text{pre}}$ versus $C_{\text{pre}}$: number of iterations in the Krylov solver, $D_{\text{it}}$ versus $C_{\text{it}}$: amount of time spent iterating to find the solution, $D_{\text{Kry}}$ versus $C_{\text{Kry}}$; and the total times, $D_{\text{Tot}}$ versus $C_{\text{Tot}}$. These times are defined as $D_{\text{pre}} = T_{LU}$, $C_{\text{pre}} = T_{LU} + T_{BC} + T_{SPK} + T_{LUrdcd}$, $D_{\text{Tot}} = D_{\text{pre}} + D_{\text{Kry}}$, and $C_{\text{Tot}} = C_{\text{pre}} + C_{\text{Kry}}$. Note that for $\text{SaP::GPU}$, the number of iterations is reported in increments of 0.25, indicating convergence at one of four possible points during a $\text{BiCGStab(2)}$ iteration.

The number of iterations to convergence suggests that the quality of the coupled version of the preconditioner is superior. Yet the cost for getting this better preconditioner is higher and $\text{SaP::GPU-D}$ ends up winning by taking as little as half the time required by $\text{SaP::GPU-C}$. When the same factorization is used multiple times, this conclusion could change since the metric that controls the performance would be $D_{\text{Kry}}$ and $C_{\text{Kry}}$, 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 $P = 50$.

4.1.2. Sensitivity with respect to $d$. Next, we report on the performance of $\text{SaP::GPU}$ 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 (2.11). The entries in the matrix are randomly generated and $P = 50$. The findings are summarized in Figure 4, where $\text{SaP::GPU-C}$ and $\text{SaP::GPU-D}$ are compared against the banded
linear solver in MKL. When $d > 1$ the impact of the truncation becomes increasingly irrelevant—\(d\) a situation that places 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. The 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 nonnegligible 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 the split of the time to solution between the major components of the implementation. 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 SaP::GPU-D over SaP::GPU-C 2× speedup factor is obtained virtually over the entire spectrum of $d$ values.

### 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 Figures 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 into three subrows: SaP::GPU-D results are reported in the first subrow, SaP::GPU-C in the second subrow, and MKL in the third subrow. 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.
Influence of $d$ for coupled (C) versus 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.

<table>
<thead>
<tr>
<th>$d$</th>
<th>$D_{pre}$</th>
<th>$C_{pre}$</th>
<th>$D_{st}$</th>
<th>$C_{st}$</th>
<th>$D_{Kry}$</th>
<th>$C_{Kry}$</th>
<th>$D_{tot}$</th>
<th>$C_{tot}$</th>
<th>SpuUp</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6\times10^{-2}$</td>
<td>0.26</td>
<td>0.19</td>
<td>0.31</td>
<td>0.14</td>
<td>0.25</td>
<td>0.14</td>
<td>0.44</td>
<td>0.27</td>
<td>0.6</td>
</tr>
<tr>
<td>$8\times10^{-2}$</td>
<td>0.28</td>
<td>0.21</td>
<td>0.34</td>
<td>0.16</td>
<td>0.27</td>
<td>0.16</td>
<td>0.48</td>
<td>0.29</td>
<td>0.62</td>
</tr>
<tr>
<td>$0.1$</td>
<td>0.32</td>
<td>0.23</td>
<td>0.38</td>
<td>0.18</td>
<td>0.29</td>
<td>0.18</td>
<td>0.52</td>
<td>0.30</td>
<td>0.64</td>
</tr>
<tr>
<td>$0.2$</td>
<td>0.42</td>
<td>0.32</td>
<td>0.48</td>
<td>0.24</td>
<td>0.34</td>
<td>0.24</td>
<td>0.56</td>
<td>0.31</td>
<td>0.68</td>
</tr>
<tr>
<td>$0.3$</td>
<td>0.50</td>
<td>0.40</td>
<td>0.58</td>
<td>0.30</td>
<td>0.40</td>
<td>0.30</td>
<td>0.60</td>
<td>0.34</td>
<td>0.72</td>
</tr>
<tr>
<td>$0.4$</td>
<td>0.60</td>
<td>0.50</td>
<td>0.68</td>
<td>0.36</td>
<td>0.45</td>
<td>0.36</td>
<td>0.64</td>
<td>0.38</td>
<td>0.76</td>
</tr>
<tr>
<td>$0.5$</td>
<td>0.70</td>
<td>0.60</td>
<td>0.78</td>
<td>0.42</td>
<td>0.50</td>
<td>0.42</td>
<td>0.68</td>
<td>0.44</td>
<td>0.80</td>
</tr>
<tr>
<td>$0.6$</td>
<td>0.80</td>
<td>0.70</td>
<td>0.88</td>
<td>0.48</td>
<td>0.55</td>
<td>0.48</td>
<td>0.72</td>
<td>0.48</td>
<td>0.84</td>
</tr>
<tr>
<td>$0.7$</td>
<td>0.90</td>
<td>0.80</td>
<td>0.98</td>
<td>0.54</td>
<td>0.60</td>
<td>0.54</td>
<td>0.76</td>
<td>0.54</td>
<td>0.88</td>
</tr>
<tr>
<td>$0.8$</td>
<td>1.00</td>
<td>0.90</td>
<td>1.08</td>
<td>0.60</td>
<td>0.65</td>
<td>0.60</td>
<td>0.80</td>
<td>0.60</td>
<td>0.92</td>
</tr>
<tr>
<td>$0.9$</td>
<td>1.10</td>
<td>1.00</td>
<td>1.18</td>
<td>0.66</td>
<td>0.70</td>
<td>0.66</td>
<td>0.84</td>
<td>0.66</td>
<td>0.96</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>$N$</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2.43E1</td>
<td>7.1E1</td>
<td>1.1E1</td>
<td>7.1E1</td>
<td>1.1E1</td>
<td>7.1E1</td>
</tr>
<tr>
<td>2000</td>
<td>2.24E1</td>
<td>8.5E1</td>
<td>1.3E1</td>
<td>8.5E1</td>
<td>1.3E1</td>
<td>8.5E1</td>
</tr>
<tr>
<td>5000</td>
<td>7.99E1</td>
<td>2.7E1</td>
<td>4.1E1</td>
<td>2.7E1</td>
<td>4.1E1</td>
<td>2.7E1</td>
</tr>
<tr>
<td>10000</td>
<td>1.05E1</td>
<td>5.2E1</td>
<td>8.6E1</td>
<td>5.2E1</td>
<td>8.6E1</td>
<td>5.2E1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$K$</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2.51E1</td>
<td>7.5E1</td>
<td>1.2E1</td>
<td>7.5E1</td>
<td>1.2E1</td>
<td>7.5E1</td>
</tr>
<tr>
<td>200</td>
<td>2.64E1</td>
<td>8.1E1</td>
<td>1.3E1</td>
<td>8.1E1</td>
<td>1.3E1</td>
<td>8.1E1</td>
</tr>
<tr>
<td>500</td>
<td>8.6E1</td>
<td>2.7E1</td>
<td>4.1E1</td>
<td>2.7E1</td>
<td>4.1E1</td>
<td>2.7E1</td>
</tr>
<tr>
<td>1000</td>
<td>1.10E1</td>
<td>5.2E1</td>
<td>8.6E1</td>
<td>5.2E1</td>
<td>8.6E1</td>
<td>5.2E1</td>
</tr>
</tbody>
</table>
The results reported in Table 3 are statistically summarized in Figure 5, which provides SaP over MKL speedup information. Assume that a test “α” successfully ran to completion in SaP:GPU-D, requiring $T_{\alpha}^{\text{SaP:GPU-D}}$, and/or in SaP:GPU-C, requiring $T_{\alpha}^{\text{SaP:GPU-C}}$. By convention, in case of failing to solve, a negative value ($-1$) is assigned to $T_{\alpha}^{\text{SaP:GPU-D}}$ or $T_{\alpha}^{\text{SaP:GPU-C}}$. If a test “α” runs to completion in both SaP and MKL, the speedup value used to generate Figure 5 is $s_{BD}^{\alpha} = T_{\alpha}^{\text{MKL}} / T_{\alpha}^{\text{SaP}}$, where $T_{\alpha}^{\text{MKL}}$ is MKL’s time to solution and $T_{\alpha}^{\text{SaP}} = \min(\max(T_{\alpha}^{\text{SaP:GPU-D}}, 0), \max(T_{\alpha}^{\text{SaP:GPU-C}}, 0))$. 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 Figure 5 is 57. 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 approximately three to six times faster in SaP. The crosses in the figure represent outliers.
4.2. Numerical experiments related to sparse linear systems.

4.2.1. Profiling results. Figure 6 plots statistical results that summarize how the time to solution, i.e., finding $x$ in $Ax = b$, is spent in SaP::GPU. The raw data used in this analysis is available online [31]. A discussion of exactly what “finding the solution of the linear system” means is postponed to section 4.2.3. The labels used in Figure 6 are inspired by the notation used in section 3.2 and Figure 2. Consider, for instance, the diagonal boosting reordering DB employed by SaP. 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 “$\alpha$” that runs to completion requires $T_{DB}^{DB} > 0$ 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 $100 \times T_{DB}^{DB} / T_{Tot}^{DB}$, where $T_{Tot}^{DB}$ is the total amount of time that test “$\alpha$” 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 Figure 6 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 so this test won’t contribute an entry to the array of data used to determine the DB box. 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 Lurcd. These counts provide insight into the solution path adopted by SaP in solving the 85 linear systems. For instance, the coupled approach; i.e., the SPIKE method of [24], has been employed in the solution of 9 of the 85 linear systems. The rest were solved 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 $100 \times T_{DB}^{DB} / T_{Tot}^{DB}$, where this time around $T_{Tot}^{Tot}$ included everything except the time spent in the Krylov-subspace component of the solution. In other words, $T_{Tot}^{Tot}$ is the time required to compute from scratch the preconditioner. The median values should be used in conjunction with the median-quartile boxplot of Figure 6 for the first row of data and Figure 7 for the second row of data. Consider, for instance, the results associated with the drop-off
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 do 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 9 points, while Drop is computed using 32 data points.

<table>
<thead>
<tr>
<th>DB</th>
<th>CM</th>
<th>Dtransf</th>
<th>Drop</th>
<th>Asmbl</th>
<th>BC</th>
<th>LU</th>
<th>SPK</th>
<th>LUrddc</th>
<th>Kry</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>1.4</td>
<td>1.9</td>
<td>4.1</td>
<td>0.7</td>
<td>1.4</td>
<td>24.8</td>
<td>23</td>
<td>4.1</td>
<td></td>
</tr>
<tr>
<td>11.4</td>
<td>3.7</td>
<td>4.1</td>
<td>25.5</td>
<td>2.7</td>
<td>2.3</td>
<td>73.4</td>
<td>41.8</td>
<td>6.4</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 6. Profiling results obtained for a set of 85 linear systems that, of a collection of 114, could be solved by SaP::GPU.

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 that 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.

The results in Figure 6 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. Somewhat unexpected, 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 relatively small number of cases in which SaP::GPU-C was preferred over SaP::GPU-D, i.e., in which the SPIKE strategy [24] was employed. This observation, however, should not be generalized as it might 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. Of a sample population of 85 tests, when SaP::GPU-C was used, the median number of iterations to solution was 6.75. Conversely, when SaP::GPU-D was preferred, the median count was 29.375 [31].

4.2.2. Comparison against state of the art. A set of 114 matrices [29], of which 105 are from the Florida matrix collection, is used herein to compare the time to solution and robustness of SaP, PARDISO, SuperLU, and MUMPS. In this context, robustness is regarded as a qualitative measure of the ability of a solver to find the solution of linear systems whose coefficient matrices (i) have widely different dimensions \( N \) and (ii) are associated with practical problems that originate in vastly different application areas. The 114 matrices were selected on the following basis: at least one of the four solvers can retrieve the solution \( x \) within 1\% relative accuracy. For a sparse linear system \( Ax = b \), this relative accuracy was measured using the method of manufactured solutions: an exact solution \( x^\ast \) was first chosen and then the right-hand side was set to \( b = Ax^\ast \). Each sparse linear solver \( s \) attempted to produce an approximation \( \tilde{x}_s \) of the solution \( x^\ast \). If this approximation satisfied \( ||\tilde{x}_s - x^\ast||_2/||x^\ast||_2 \leq 0.01 \), then the solve was considered to have been successful. For all tests, \( x^\ast \) 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 \( x^\ast \). Figure 8 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.

All solvers have been used in all tests "out of the box," i.e., given a linear system, no attempt was made to change parameters that might optimize the process of finding a solution—the solvers were used with default values. For SaP, this meant \( atol = 0 \) and \( rtol = 10^{-10} \)—see (4.1). Given that SaP::GPU is an iterative solver, in many
instances the initial guess can be selected to be relatively close to the actual solution. This is avoided here by choosing as initial guess $\tilde{x}(0) = 0_N$, i.e., far from $x^\star$. Against this backdrop, is the quality of the solution produced by solver $s_1$ with its default settings comparable with what $s_2$ produces with its very own default settings? This question is answered in Figure 9, which can be interpreted as a cumulative probability distribution. For a specified value $\epsilon$, the plot shows for SaP, MUMPS, and SuperLU what fraction of the 114 tests were solved by a solver $s$ at a value of relative error tighter than $\epsilon$, i.e., so that $||\tilde{x}_s - x^\star||_2/||x^\star||_2 < \epsilon$. For instance, using default settings, about 45%, 55%, and 70% of the tests were solved with a relative error smaller than or equal to $\epsilon = 10^{-8}$ by SaP, MUMPS, and SuperLU, respectively. One thing that is not apparent from this plot is the fact that when one solver produced a solution with a larger relative error, i.e., lower quality approximation, more often than not the other solvers produced low quality approximations as well.

In terms of robustness, SaP::GPU failed on 28 linear systems. In 23 cases, SaP ran out of GPU 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, in 23 instances it ran 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 memory, which yet comes short of the 64 GB that PARDISO, SuperLU, and MUMPS could tap into. There were three main failures modes for PARDISO, SuperLU, and MUMPS, i.e., the direct solvers. They produced a solution that was not within 1% accuracy, they seg faulted, or they started solving the problem but hung.
In terms of speed, PARDISO was the fastest, followed by MUMPS, then SaP, and finally SuperLU. Of the 57 linear systems solved by both SaP and PARDISO, SaP was faster 20 times. Of the 71 linear systems solved by both SaP and SuperLU, SaP was faster 38 times. Of the 60 linear systems solved by both SaP and MUMPS, SaP was faster 27 times. Of the 60 linear systems solved by both PARDISO and SuperLU, PARDISO was faster 60 times. Of the 57 linear systems solved by both 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. This relative speed issue is revisited in section 4.2.4.

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

$$S_{\alpha}^{\text{SaP-PARDISO}} = \log_2 \frac{T_{\alpha}^{\text{PARDISO}}}{T_{\alpha}^{\text{SaP}}}$$

with $S_{\alpha}^{\text{SaP-MUMPS}}$ and $S_{\alpha}^{\text{SaP-SuperLU}}$ similarly computed. These $S_{\alpha}^{\text{SaP-PARDISO}}$ values, which can be either positive or negative, are collected in a set $S_{\alpha}^{\text{SaP-PARDISO}}$ which is used to generate a box plot in Figure 10. The figure also reports results on $S_{\alpha}^{\text{SaP-SuperLU}}$ and $S_{\alpha}^{\text{SaP-MUMPS}}$. Note that the number of tests used to produce these statistical measures is different for each comparison: 57 linear systems for $S_{\alpha}^{\text{SaP-PARDISO}}$, 71 for $S_{\alpha}^{\text{SaP-SuperLU}}$, and 60 for $S_{\alpha}^{\text{SaP-MUMPS}}$. The median values for $S_{\alpha}^{\text{SaP-PARDISO}}$, $S_{\alpha}^{\text{SaP-SuperLU}}$, and $S_{\alpha}^{\text{SaP-MUMPS}}$ 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.

Crosses are used in Figure 10 to show statistical outliers. Favorably, most of the SaP's outliers are large and positive. For instance, there are three linear systems...
Statistical spread for SaP::GPU’s performance relative to that of PARDISO, SuperLU, and MUMPS. Referring to (4.2), the results were obtained using the data sets $S_{\text{SaP}}^{\text{PARDISO}}$ (with 57 values), $S_{\text{SaP}}^{\text{SuperLU}}$ (71 values), and $S_{\text{SaP}}^{\text{MUMPS}}$ (60 values). For which, when compared to PARDISO, SaP finishes significantly faster, four linear systems for which it is significantly faster than SuperLU, 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,” and 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 toward negative territory: the second and third quartiles 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 other solvers, it does so by a large margin.

4.2.3. 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 the cuSOLVER library [23]. 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 of 114 systems. There were 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 by both 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.

4.2.4. Performance evaluation. We provide a final sparse linear solver performance comparison using performance profiles [8]. Consider the collection $\mathcal{P}$ of $n_P = 114$ test problems used herein [29], along with the set $\mathcal{S}$ of $n_S = 5$ sparse linear
Fig. 11. Performance profiles in two different ranges: \( \tau \in [1, 10] \) on the left and \( \tau \in [1, 100] \) on the right.

solvers: PARDISO, SuperLU, MUMPS, cuSOLVER, and SaP::GPU. Following the notation in [8], let \( t_{p,s} \) be the computing time required to solve problem \( p \) using solver \( s \). The performance ratio

\[
    r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} : s \in S\}}
\]

compares the performance of solver \( s \) on problem \( p \) with the best performance of any solver on that problem. If solver \( s \) does not solve problem \( p \), we set the corresponding performance ratio to a value \( r_M > r_{p,s} \forall p, s \). It can be shown that such an \( r_M \) does not affect the results of the analysis [8]. Here, we use \( r_M = 10^5 \).

A global estimate of the performance of a solver \( s \) can then be obtained by defining the probability that the performance ratio \( r_{p,s} \) is within a factor \( \tau \geq 1 \) of the best possible performance ratio:

\[
    \rho_s(\tau) = \frac{1}{|P|} \sum_{p \in P} \mathbf{1}_{\{r_{p,s} \leq \tau\}}.
\]

As the cumulative distribution function for the performance ratio, \( \rho_s \) is monotonically increasing and \( \rho_s(\tau) \leq 1 \). Termined here performance profiles, distribution functions of a performance metric have been shown to be robust benchmarking indicators. A plot of the performance profile unveils several main performance characteristics. The interested reader is directed to [8] for more details. Here we only mention that, assuming a set of problems \( P \) that is sufficiently large and representative of problems of interest, solvers with a large probability \( \rho_s(\tau) \) are to be preferred. As a particular case, \( \rho_s(1) \) represents the probability that solver \( s \) will have the best performance over all other solvers.

Figure 11 shows the performance profiles for the five solvers considered herein on the set of 114 matrices used in the benchmark [29]. The plot on the left, for small values of \( \tau \), reveals the solvers with the most wins (or close to the winner). PARDISO has the most wins, i.e., has the highest probability of being the best solver—this probability being almost 0.5. SaP::GPU comes in second, with a value \( \rho_s(1) \approx 0.3 \). cuSOLVER, i.e., the other GPU solver considered in this benchmarking, has effectively 0 probability of ever being the optimal solver on a given problem. The plot on the right, showing the trends of the performance profiles for larger values of \( \tau \), allows for selection of the solver able to solve most problems. Indeed, since \( 1 - \rho_s(\tau) \) represents
the fraction of problems in the set \( P \) that solver \( s \) cannot solve within a factor \( \tau \) of the best solver, including the cases for which that solver fails, for large values of \( \tau \) this quantity is an indication of the solver robustness (defined here as the probability to solve more problems, regardless of the time to solution). The plot on the right in Figure 11 shows \( \text{SaP::GPU} \) as being able to solve nearly 80% of the problem set.

5. Conclusions and future work. This contribution discusses parallel strategies to solve dense banded and sparse linear systems on GPU cards. BSD-3 open source implementations of all these strategies are available at [30, 31] as part of a software package called \( \text{SaP} \). At the time of this study and over a broad range of dense matrix sizes and bandwidths, \( \text{SaP} \) was likely to run two times faster than Intel’s MKL. This conclusion should be modulated by hardware considerations and also by 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 \( \text{SaP} \). Of a set of 114 tests, most of them using matrices from the University of Florida sparse matrix collection, \( \text{SaP} \) 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, \( \text{SaP} \) was compared against PARsISO, MUMPS, and SuperLU. The straight split-and-parallelize strategy, without the coupling involved in the SPIKE-type strategy, emerged as the more often solution approach adopted by \( \text{SaP} \).

\( \text{SaP} \) has been successfully used for the implicit numerical integration of flexible multibody dynamics, in a Newton–Krylov context [33], and current efforts target its use in conjunction with interior point methods for large-scale granular dynamics simulations [20].

Several issues remain to be investigated. First, since more than 50% of the time to solution is spent in the iterative solver, it is worth considering the techniques analyzed in [14], 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 might enable \( \text{SaP} \) to handle larger problems while possibly reducing time to solution. Third, the CM bandwidth reduction strategy implemented is dated; spectral and/or hypergraph partitioning for load balancing could lead to superior matrix splitting. Finally, as it stands, with the exception of parts of the matrix reordering, \( \text{SaP} \) is entirely a GPU solution. It would be worth investigating how the CPU can be involved in other phases of the solution. Such an investigation would be well justified given the imminent tight integration of the CPU and GPU memories.

Acknowledgments. This work benefited from discussions with Matt Knepley and Ahmed Sameh.

REFERENCES