SPIKE::GPU
A SPIKE-based preconditioned GPU Solver for Sparse Linear Systems

Ang Li Andrew Seidl Radu Serban Dan Negrut

January 13, 2014

Abstract

This contribution outlines an approach that draws on general purpose graphics processing unit (GPGPU) computing to solve large linear systems. To methodology proposed relies on a SPIKE-based preconditioner with a Krylov-subspace method and has the following three stages: (*i*) row/column reordering for boosting diagonal dominance and reducing bandwidth; (*ii*) applying single precision truncated SPIKE on a dense banded matrix obtained after dropping small elements that fall outside a carefully selected bandwidth; and (*iii*) preconditioning within the BiCGStab(2) framework. The reordering strategy adopted aims at generating a narrow bandwidth dense matrix that is diagonally heavy. The results of several numerical experiments indicate that when used in conjunction with large dense banded matrices, the proposed approach is two to three times faster than the latest version of the `MKL` dense solver as soon as $d > 0.23$. When handling sparse systems, synthetic results obtained for large random matrices suggest that the proposed approach is three to five times faster than the `PARDISO` solver as long as the reordered matrices are close to being diagonally dominant. For a random set of smaller dimension application matrices, some out of the University of Florida Sparse Matrix Collection, the proposed approach is shown to be better or comparable in performance to `PARDISO`.

1 Introduction and Motivation

Recently, GPGPU computing has been adopted at an increasingly faster pace in a wide spectrum of scientific and engineering data analysis, simulation, and visualization tasks. In fields as diverse as genomics, medical physics, astronomy, nuclear engineering, quantum chemistry, finance, oil and gas exploration, etc., GPU computing is attractive owing to its capacity to deliver speed-ups that in some cases can be as high as one order of magnitude compared to the execution on traditional multi-core CPUs. This performance boost is typically associated with computational tasks where the application data is amenable to single instruction multiple data (SIMD) processing. The modern GPUs have a deep memory hierarchy that at the low-end displays bandwidths in excess of 300 GB/s, while at the high-end, for scratch pad memory and registers, displays low latency and bandwidths that approach 1 TB/s. From a hardware architectural perspective, one of the salient features of GPUs is their ability to pack a very large number of rather unsophisticated (from a logical point of view) scalar processors. These are organized in streaming multiprocessors (SMs), each GPU being the sum of several of these SMs that work independently by typically sharing less than 8 GB of main memory and 1 GB of L2 cache.

Although employed on a small scale as early as the 1990s, in relative historical terms, leveraging GPGPU computing in scientific and engineering applications has only recently gained widespread acceptance. As such, this is a time marked by a perceived lack of productivity tools to support domain-specific applications. This paper is motivated by this observation and addresses this aspect by proposing a methodology for the numerical solution of linear systems of equations. Specifically, we focus on a GPU solution of the problem

$$\mathbf{A}\mathbf{x} = \mathbf{b}\,, \tag{1}$$

where $\mathbf{A} \in \mathbb{R}^{N\times N}$ is a sparse nonsymmetric nonsingular matrix with $10^4 \leq N \leq 10^6$. For the parallel solution of this problem, CPU multi-core approaches exist and are well established, see for instance [1, 19]. Thus, for a domain-specific application implemented on the GPU that calls for solving $\mathbf{A}\mathbf{x} = \mathbf{b}$, one alternative is to fall back on Pardiso and CPU computing. This approach has at least two disadvantages: it is cumbersome to implement owing to its half-GPU half-CPU nature, and it degrades the overall performance of the algorithm due to the back-and-forth data movement across the PCI host/device interconnect, which typically supports bandwidths that come short of 10 GB/s.

The solution adopted herein for the problem $\mathbf{A}\mathbf{x} = \mathbf{b}$ draws on Krylov-subspace approaches and calls for preconditioning that relies on a divide-and-conquer approach with drop-off. The overall methodology is mapped for

execution onto the GPU and as such it is aware of the deep memory hierarchy, the multi-SM organization of the device, and its SIMD bias. The most salient feature of the solution methodology is the choice to always cast the preconditioning step as a *dense* linear algebra problem. Specifically, each outer Krylov-subspace iteration calls for at least one preconditioning step that is implemented on the GPU by solving $\hat{\mathbf{A}}\mathbf{y} = \hat{\mathbf{b}}$, where $\hat{\mathbf{A}} \in \mathbb{R}^{N\times N}$ is a dense banded matrix conveniently obtained from $\mathbf{A}$ after a sequence of reordering strategies and element drop-off. It becomes apparent that good performance of the proposed approach hinges upon (i) a fast banded solver, and (ii) an effective reordering strategy. In terms of (i), we show that the implemented GPU banded solver is faster than the corresponding MKL solver [3]; for (ii), a strategy that combines a bandwidth reduction RCM reordering [7] with a diagonal dominance boosting reordering has yielded well balanced coefficient matrices that can be factored fast on the GPU owing to an SIMD-friendly underlying data structure.

The remainder of this paper is structured as follows. After an overview of the SPIKE algorithm in §2, we discuss the use of the truncated SPIKE variant as a preconditioner in §3. We present details of the GPU implementation in §4, after which we describe the experiments performed and present numerical results in §5. We conclude and provide directions of future work in §6.

2 SPIKE algorithm overview

Assume that the matrix $\mathbf{A}$ is (or can be brought to be) banded with half-bandwidth K much less than the matrix size N. For some partition $\{N_i, i = 1, \ldots, P \mid \sum_i N_i = N\}$, any banded matrix can be partitioned into a block tridiagonal form with diagonal blocks $\mathbf{A}_i \in \mathbb{R}^{N_i\times N_i}$. 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 $M \times M$ (with $M \ll N_i$). It is trivial to observe that for banded matrices with half-bandwidth K, $M = K$ always holds.

The SPIKE algorithm [18, 15, 16], a divide-and-conquer technique, is based on a factorization of the partitioned banded matrix into the product of a block diagonal matrix $\mathbf{D}$ and a so-called *spike matrix* $\mathbf{S}$ with identity diagonal blocks and spike blocks, each having M columns, in the adjacent off-diagonal blocks (see Figure 1):

$$\mathbf{A} = \mathbf{D}\mathbf{S}\,, \tag{2}$$

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

$$\mathbf{A}_1\mathbf{V}_1 = \begin{bmatrix}\mathbf{0}\\ \mathbf{0}\\ \mathbf{B}_1\end{bmatrix} \tag{3a}$$

$$\mathbf{A}_i\left[\mathbf{W}_i \mid \mathbf{V}_i\right] = \begin{bmatrix}\mathbf{C}_i & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{B}_i\end{bmatrix}\,, \quad i = 2, \ldots, P-1 \tag{3b}$$

$$\mathbf{A}_P\mathbf{W}_P = \begin{bmatrix}\mathbf{C}_P\\ \mathbf{0}\\ \mathbf{0}\end{bmatrix}. \tag{3c}$$

Assuming that each $\mathbf{A}_i$ is non-singular, the spikes associated with each partition can be obtained by solving the independent linear systems (3). Solving the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ is thus reduced to solving

$$\mathbf{D}\mathbf{g} = \mathbf{b} \tag{4}$$

$$\mathbf{S}\mathbf{x} = \mathbf{g} \tag{5}$$

Since $\mathbf{D}$ is block-diagonal, solving for the modified right-hand side $\mathbf{g}$ from (4) is a trivial task which can be performed with perfect parallelism (assuming a fine enough partitioning), simultaneously with the generation of the spikes in (3).

The crux of the SPIKE algorithm, solving (5), relies on the observation that this problem can be further reduced to one of much smaller size, $\hat{\mathbf{S}}\hat{\mathbf{x}} = \hat{\mathbf{g}}$. In [15], Polizzi and Sameh introduced two main versions: (i) an exact reduction, in which $\hat{\mathbf{S}}$ is a block tridiagonal matrix and (ii) an approximate reduction resulting in a block diagonal matrix $\hat{\mathbf{S}}$. These two approaches are briefly described next.

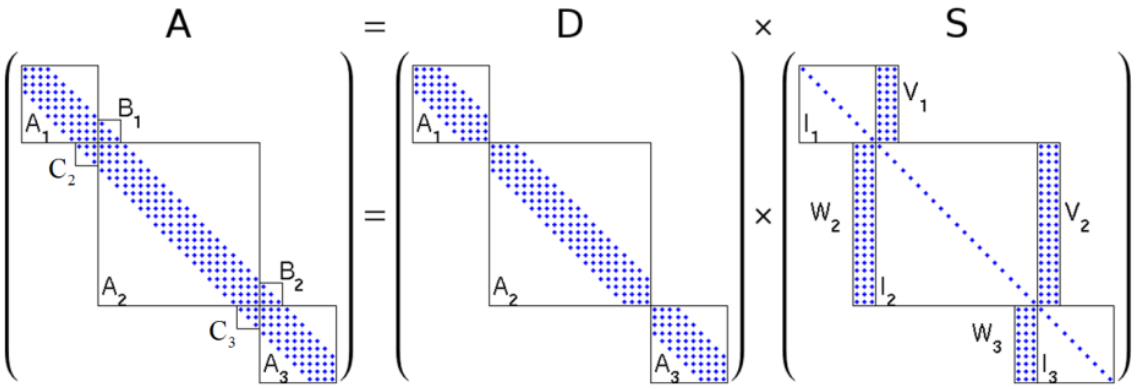


Figure 1: Factorization of the matrix $\mathbf{A}$ with $P = 3$.

2.1 Basic SPIKE algorithm

The spikes $\mathbf{V}_i$ and $\mathbf{W}_i$, as well as the modified right-hand side and unknown vector partitions $\mathbf{g}_i$ and $\mathbf{x}_i$ in (5) can be further partitioned into their top and bottom M rows and the middle $N_i - 2M$ rows:

$$\mathbf{V}_i = \begin{bmatrix} \mathbf{V}_i^{(t)} \\ \mathbf{V}_i' \\ \mathbf{V}_i^{(b)} \end{bmatrix}, \quad \mathbf{W}_i = \begin{bmatrix} \mathbf{W}_i^{(t)} \\ \mathbf{W}_i' \\ \mathbf{W}_i^{(b)} \end{bmatrix}, \tag{6a}$$

$$\mathbf{g}_i = \begin{bmatrix} \mathbf{g}_i^{(t)} \\ \mathbf{g}_i' \\ \mathbf{g}_i^{(b)} \end{bmatrix}, \qquad \mathbf{x}_i = \begin{bmatrix} \mathbf{x}_i^{(t)} \\ \mathbf{x}_i' \\ \mathbf{x}_i^{(b)} \end{bmatrix}. \tag{6b}$$

With this, a block tridiagonal reduced system, $\hat{\mathbf{S}}\hat{\mathbf{x}} = \hat{\mathbf{g}}$ of dimension $2M(P-1) \ll N$, is obtained by excluding the middle partitions of the spike matrices as:

$$\begin{bmatrix} \mathbf{R}_1 & \mathbf{M}_1 & & & \\ & \ddots & & & \\ & \mathbf{N}_i & \mathbf{R}_i & \mathbf{M}_i & \\ & & & \ddots & \\ & & & \mathbf{N}_{P-1} & \mathbf{R}_{P-1} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{x}}_1 \\ \vdots \\ \hat{\mathbf{x}}_i \\ \vdots \\ \hat{\mathbf{x}}_{P-1} \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{g}}_1 \\ \vdots \\ \hat{\mathbf{g}}_i \\ \vdots \\ \hat{\mathbf{g}}_{P-1} \end{bmatrix}, \tag{7}$$

where

$$\mathbf{N}_i = \begin{bmatrix} \mathbf{W}_i^{(b)} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \qquad i = 2, \ldots, P-1 \tag{8a}$$

$$\mathbf{R}_i = \begin{bmatrix} I_M & \mathbf{V}_i^{(b)} \\ \mathbf{W}_{i+1}^{(t)} & I_M \end{bmatrix}, \quad i = 1, \ldots, P-1 \tag{8b}$$

$$\mathbf{M}_i = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_{k+1}^{(t)} \end{bmatrix}, \qquad i = 1, \ldots, P-2 \tag{8c}$$

and

$$\hat{\mathbf{x}}_i = \begin{bmatrix} \mathbf{x}_i^{(b)} \\ \mathbf{x}_{i+1}^{(t)} \end{bmatrix}, \hat{\mathbf{g}}_i = \begin{bmatrix} \mathbf{g}_i^{(b)} \\ \mathbf{g}_{i+1}^{(t)} \end{bmatrix}, \quad i = 1, \ldots, P-1 \tag{9}$$

The global solution $\mathbf{x}$ of the original system can then be reconstructed from $\hat{\mathbf{x}}$ using:

$$\mathbf{x}_1' = \mathbf{g}_1' - \mathbf{V}_1' \mathbf{x}_2^{(t)} \tag{10a}$$

$$\mathbf{x}_i' = \mathbf{g}_i' - \mathbf{V}_i' \mathbf{x}_{i+1}^{(t)} - \mathbf{W}_i' \mathbf{x}_{i-1}^{(b)}, \quad i = 2, \ldots, P-1 \tag{10b}$$

$$\mathbf{x}_P' = \mathbf{g}_P' - \mathbf{W}_P' \mathbf{x}_{P-1}^{(b)} \tag{10c}$$

Various options for the efficient implementation of the above algorithm as a direct linear solver are presented in [15], including a recursive form of the SPIKE algorithm for the solution of the reduced system (7) [1].

While this direct approach to the solution of the reduced system is attractive as it makes no assumptions on the matrix $\mathbf{A}$ (it is general enough to handle non-diagonally dominant matrices), we do not consider it here for our parallel GPU implementation. This decision is justified by the following two observations. First, even though the reduced system has (much) smaller dimension than the original system, its half-bandwidth is larger resulting in diminishing returns in terms of efficiency. Second, at each recursive step, additional memory is required for storing the new reduced matrix which cannot be deallocated until the global solution is fully recovered, thus limiting the size of problems that can be tackled. As such, we adopt an approach based on the so-called *truncated* SPIKE approximation which we describe next.

2.2 Truncated SPIKE algorithm

If the matrix $\mathbf{A}$ is diagonally dominant[2], it can be shown that 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 [14]. This decay, which is more pronounced the larger the degree of diagonal dominance of $\mathbf{A}$, allows the approximation of the reduced matrix $\hat{\mathbf{S}}$ in (7) by its diagonal blocks only; i.e., setting the top and bottom parts of the right and left spikes, respectively, to zero. This results in the so-called *truncated* version of the SPIKE algorithm [15].

Each diagonal block $\mathbf{R}_i\hat{\mathbf{x}}_i = \hat{\mathbf{g}}_i$ of the resulting truncated reduced system can be solved independently using the following elimination steps:

$$\text{Form } \bar{\mathbf{R}}_i = I_M - \mathbf{W}_{i+1}^{(t)}\mathbf{V}_i^{(b)} \tag{11a}$$

$$\text{Solve } \bar{\mathbf{R}}_i\mathbf{x}_{i+1}^{(t)} = \mathbf{g}_{i+1}^{(t)} - \mathbf{W}_{i+1}^{(t)}\mathbf{g}_i^{(b)} \tag{11b}$$

$$\text{Calculate } \mathbf{x}_i^{(b)} = \mathbf{g}_i^{(b)} - \mathbf{V}_i^{(b)}\mathbf{x}_{i+1}^{(t)} \tag{11c}$$

Assuming that 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 (3a) using only the bottom $M \times M$ 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 a second UL factorization of $\mathbf{A}_i$, in which case $\mathbf{W}_i^{(t)}$ can be obtained using only the top $M \times M$ blocks of the new U and L.

Since the spikes now contain parts that are never calculated explicitly, the global solution $\mathbf{x}$ cannot any longer be retrieved from the solution of the reduced system as in (10). Instead, the right-hand side $\mathbf{b}$ is first *purified* from the influences of the coupling blocks $\mathbf{B}_i$ and $\mathbf{C}_i$:

$$\mathbf{A}_1\mathbf{x}_1 + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{B}_1\mathbf{x}_2^{(t)} \end{bmatrix} = \mathbf{b}_1 \tag{12a}$$

$$\begin{bmatrix} \mathbf{C}_i\mathbf{x}_{i-1}^{(b)} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} + \mathbf{A}_i\mathbf{x}_i + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{B}_i\mathbf{x}_{i+1}^{(t)} \end{bmatrix} = \mathbf{b}_i\,, \quad i = 2, \ldots, P-1 \tag{12b}$$

$$\begin{bmatrix} \mathbf{C}_P\mathbf{x}_{P-1}^{(b)} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} + \mathbf{A}_P\mathbf{x}_P = \mathbf{b}_P \tag{12c}$$

after which the global solution is obtained by solving the resulting block diagonal system, using the available LU (or UL) factorizations.

3 Truncated SPIKE as a preconditioner

Due to the approximation introduced by the spike truncation described in §2.2, an iterative solution refinement for achieving sufficient accuracy is necessary, thus making the overall SPIKE algorithm a preconditioner. In this case, the diagonal dominance restriction can be relaxed and, using various reordering algorithms, a solver for general sparse

[1] This represents a direct solver for the original problem (1), only as long as (4) is solved exactly. Otherwise, an outer iterative scheme is required to ensure sufficient accuracy.

[2] Let $d \geq 0$ be such that $|a_{ii}| \geq d \sum_{j \neq i} |a_{ij}|$, $\forall i = 1, \ldots, N$. The matrix $\mathbf{A}$ is diagonally dominant if $d \geq 1$. We call d the degree of diagonal dominance.

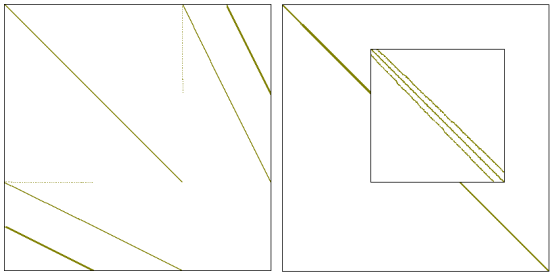


Figure 2: Sparsity pattern for the original (left) and reordered (right) ANCF matrix of dimension $N = 88,950$. The sparsity pattern for a 2000×2000 diagonal block of the reordered matrix is displayed in the inset.

linear systems can be constructed. Moreover, additional approximations (of the matrix and in the solution scheme) can be introduced to further improve efficiency, as described below.

The approach embraced by SPIKE is predicated on the matrix $\mathbf{A}$ being banded and diagonally dominant. As this is most often not the case, for the solution of general sparse linear systems, an aggressive preprocessing stage is employed to reorder the columns and rows of $\mathbf{A}$. Since $\mathbf{A}$ is not assumed symmetric, no attempt is made to preserve symmetry, which adds an extra degree of freedom when a first set of row permutations is applied as $\mathbf{QAx} = \mathbf{Qb}$, to either maximize the number of nonzeros on the diagonal of $\mathbf{A}$ (maximum traversal search) [9], or maximize the product of the absolute values of the diagonal entries [10, 11]. Both reordering algorithms are implemented using simple depth first search with a look-ahead technique as available in the Harwell Software Library (HSL) [1].

While this first reordering renders $\mathbf{QA}$ to be diagonally "heavy", a second reordering, using the traditional Reverse Cuthill-McKee (RCM) algorithm [8], is applied in order to reduce its bandwidth. Since the diagonal entries should not be relocated, the second permutation is applied to the symmetric matrix $\mathbf{QA} + \mathbf{A}^T\mathbf{Q}^T$.

Following the reordering, in an attempt to produce an even narrower bandwidth, we drop off elements of the reordered matrix $\mathbf{A}$ that are small in magnitude and far away from the main diagonal. More precisely, we select the smallest half-bandwidth K such that

$$\sum_{|j-i|\leq K} |a_{ij}|^2 \geq \delta \cdot \sum_{i,j} |a_{ij}|^2 \,, \tag{13}$$

where $\delta \in (0, 1)$ represents the fraction of the Frobenius norm of $\mathbf{A}$ that is retained.

Following the (optional) drop-off, we apply a second-stage RCM reordering to each main diagonal block for further reduction of bandwidth. Even though the combination MC64 – RCM typically leads to significant bandwidth reduction, it is very common to obtain banded matrices with the band itself being very sparse. Consider an example ANCFBeams matrix $88950 - lhs.mtx$ (with dimension $88,950$ and number of non-zeros $513,900$) from multibody dynamics simulation in Fig. 2. The matrix has on average only 5.78 non-zero elements per row. After reordering with no drop-off, the resulting banded matrix has a half-bandwidth $K = 205$. As shown on the right in Fig. 2, the band itself is very sparse with a fill-in of only 0.7%. Based on this observation, the default preconditioner in `SPIKE::GPU` constructs a block banded matrix where each diagonal block (corresponding to one of the P partitions of the banded matrix obtained after the first stage MC64 – RCM reordering) is allowed to have a different bandwidth. This is achieved using a second RCM pass, this time on each diagonal block separately. Applying this strategy to the matrix in Fig. 2, using $P = 16$ partitions, the equivalent halfbandwidth is reduced to $K = 141$, while the fill-in of the band becomes 2.8%.

When the matrix $\mathbf{A}$ is diagonally dominant, the LU (and UL) factorization of each diagonal block $\mathbf{A}_i$ can be done without pivoting. However, if the matrix is not diagonally dominant, one cannot guarantee that the diagonal blocks $\mathbf{A}_i$ are nonsingular. Adopting the strategy used in `PARDISO` [20], we continue to use LU factorizations of the diagonal blocks without pivoting but with *diagonal boosting* (perturbation). In this case, we obtain 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 [13].

With the above, an approximate solver is obtained which can be used as a preconditioner for an outer Krylov subspace scheme. In this work, we use BiCGStab(2)) [21] and left-preconditioning, although other choices are possible.

4 GPU implementation details

Number of partitions and partition size. Selection of the number of partitions P must strike a balance between two conflicting requirements: workload on each thread block and accuracy of the truncated approximation. In the

current implementation, no attempt is made to automate this selection and some experimentation is required.

For a given number of partitions P, load balancing is acheieved in a straight-forward manner, by setting the partition sizes of the first P_r partitions to be $\lfloor N/P \rfloor + 1$ with the remaining partitions being of size $\lfloor N/P \rfloor$, where $N = P\lfloor N/P \rfloor + P_r$.

Matrix Storage. In the context of using the truncated SPIKE algorithm as a preconditioner for a Krylov subspace method, the sparse matrix $\mathbf{A}$ is used during the pre-processing stage for reordering and element-wise sorting and during the solution phase, in the form of matrix-vector products. In the current implementation, the pre-processing phase is performed on the CPU and, for convenience, we adopted the simplest format, namely DOK (Dictionary of Keys). On the other hand, for the SpMV operations required by the BiCGStab(2) algorithm, coalesced memory access on the GPU is best achieved using a CSR (Compressed Sparse Row) format which ensures that non-zero elements in the same row are stored contiguously.

For the dense banded diagonal matrices $\mathbf{A}_i$, we adopt a "tall and thin" storage in column-major order (as shown below for a matrix with $N = 8$ and $K = 2$).

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

With this, all diagonal elements are stored in the K-th column while all other elements are distributed columnwise accordingly. This manner of storage groups the operands of the LU/UL factorizations and allows coalesced memory access.

LU/UL Implementations. As described in §2.2, we adopt a strategy in which we perform both an LU and a UL factorization of each (dense banded) diagonal block $\mathbf{A}_i$. With a proper partitioning, each of these diagonal blocks is handled separately by one thread block, or several thread blocks, depending on the half-bandwidth K. Within each block synchronization is unavoidable since both the factorization and the (forward) elimination/(backward) substitution consist of $(N_i - 1)$ dependent steps. To maximize performance (in a trade-off between minimizing the overhead of kernel launches and maximizing occupancy), we distinguish between two situations, based on the value K of the half-bandwidth. The threshold value $K = 64$ below was selected through numerical experimentation over a variety of problems and is appropriate for the current hardware capabilities in terms of the maximum allowable threads in a block.

For $K \leq 64$, we attempt to reduce the overhead of kernel launches. Thus, instead of repeating $(N_i - 1)$ kernel launches, each completing a single step of the factorization, we launch a single kernel using $\min(K^2, 1024)$ threads per block and synchronize explicitly by calling the routine `__syncthreads()`, with no need for global synchronization. In this case, we employ a so-called *window-sliding* method which is illustrated in Figure 3. At each step of the factorization (one iteration of the outer loop in LU), each thread updates a fixed number of entries in the matrix $\mathbf{A}_i$ (anywhere between 1 and 4 with the current parameters). Once all threads in the block complete their work, they are synchronized and the window of current entries slides down and to the right by one unit. The UL factorization is performed in a similar way with the difference that the active window starts at the bottom-right and moves up and to the left at each step. Since the UL factorization is only used in calculating $\mathbf{W}_i^{(t)}$, we can restrict the factorization to the top-left $l \times l$ block of each $\mathbf{A}_i$ (with $l > K$). In our implementation, we select $l = 2K$ and the execution time of the UL factorization is negligible compared to that required for the LU factorization (since $l \ll N_i$).

For $K > 64$, on current GPU hardware, the maximum allowable threads in a block is significantly smaller than K^2. As a consequence, the overhead of launching multiple kernels is overshadowed by the work required to update all values within a $K \times K$ block. In this case, we implemented the blocked version of LU, which makes trade-off between the total amount of work in each kernel launch and the total overhead of kernel launches and the penalty of flushing the cache. In each kernel launch, a fixed number of rows in $\mathbf{U}$ matrix as well as the same number of columns in $\mathbf{L}$ matrix are calculated, and a kernel with multiple blocks of threads is launched for updating rest elements. These processes are repeated until we complate calculating all elements in $\mathbf{L}$ and $\mathbf{U}$.

Use of shared memory. In our current implementation, the LU/UL factorization and elimination/substitution kernels do not use shared memory. A possible approach is as follows: (*i*) allocate space for $(K + 1) \times (K + 1)$ elements in a circular buffer to hold all values inside the window at the current step; (*ii*) prior to the first step, set the window's position to the top-left corner and load the values in the window to shared memory; (*iii*) before sliding the window in preparation for the next step, write all values exiting the window back to global memory and read

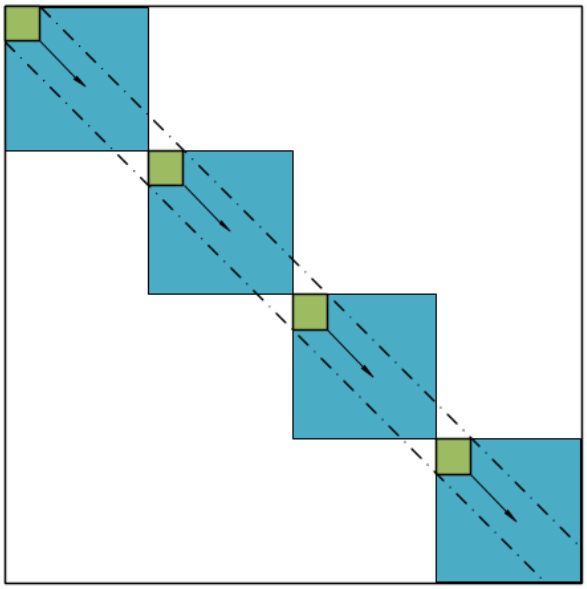


Figure 3: Window-sliding method

in new values entering the window (unless the partition boundary was reached). This however, exhibits memory divergence in step (*iii*) while accessing elements in the north-bound and south-bound directions. A solution to this problem (which will be adopted in the next implementation) is to pre-fetch elements for several steps when loading from global memory and delay global memory writes (while buffering elements) for several steps.

Mixed Precision Strategy. Microprocessor performance characteristics for 32-bit floating point arithmetic (single precision) is significantly higher than 64-bit floating point arithmetic (double precision). While in newer generation GPU accelerators, such as the NVIDIA Tesla K20 used for this work, this performance gap has been tremendously reduced [2], strategic use of precision in GPU calculations can have a substantial effect on overall performance.

In our current implementation, we use a simple mixed-precision strategy by performing all SPIKE-related operations in single precision throughout the preconditoner stage and switching to double precision arithmetic in the outer BiCGStab(2) calculations. Numerical experiments indicate that this strategy results in an average of 50% improvement in performance when compared with an approach where all calculations are performed in double precision.

Second-stage reordering. (TODO: Add details of second-stage reordering.)

5 Numerical experiments

In this section, we present results obtained with the proposed approach on a variety of test problems. Following a brief description of the hardware and software environment used for these experiments, we describe the three groups of problems investigated therein and then discuss the performance of the proposed approach and implementation (called in the sequel `SPIKE::GPU`).

5.1 Test Environment

All numerical experiments presented here were performed on a machine with an Intel Nehalem Xeon E5520 2.26 GHz processor and an NVIDIA Tesla K20 GPU accelerator [4]. The solution times reported are inclusive time that account for the amount of time to move data to/from the device. Also, where applicable, the solution times include times required to do the necessary matrix reorderings.

The GPU code was built with the `nvcc` compiler in CUDA version 5.0, while all CPU codes were built with Intel's compiler ICPC 13.0. In all cases, level 2 optimization was used.

The proposed solver was compared against the dense banded Lapack solver (`dgbtrf`/`dgbtrs`) provided with Intel's Math Kernel Library (`MKL`) version 11.0 [3] and the sparse solver `PARDISO` version 4.1.2 (April 2011) distributed by ICI USI Lugano [20].

5.2 Test Problems

In order to assess its overall performance, as well as that of its various components, we exercised the `SPIKE::GPU` code on three distinct types of problems, as described next. In all tests, we used the *method of manufactured solutions* to specify the linear system's right-hand side vector (i.e., we set $\mathbf{b} = \mathbf{A}\tilde{\mathbf{x}}$ for a randomly generated solution vector $\tilde{\mathbf{x}}$).

Synthetic banded matrices. In order to assess the effectivness of the truncated SPIKE approach as a preconditioner, excluding the impact of the various reordering strategies, we use `SPIKE::GPU` on a series of banded matrices of varying size N, half-bandwidth K and degree of diagonal dominance d. Matrices in this group were generated programatically, using random entries for a given set of parameters N, K, followed by a rescaling of the main diagonal to impose a specified degree of diagonal dominance. We compare the performance of the `SPIKE::GPU` solver against the `MKL` dense banded solver.

Synthetic sparse matrices. The next group of tests aimed at evaluating the ability of the proposed reordering scheme to recover a known band structure and degree of diagonal dominance. To this end, sparse matrices (of random sparsity pattern) were programatically generated by applying random independent row and column permutations to banded matrices generated as described above [3]. The performance of `SPIKE::GPU` on matrices from this group was compared against the sparse solver `PARDISO`.

Real application matrices. To evaluate how `SPIKE::GPU` performs, a group of in total 86 real application matrices were selected. Most of these matrices were randomly picked from Florida Sparse Matrix Collection. Among these matrices, three were ANCFBeams matrices with different problem sizes. For performance comparison, we compared the performance of solving real application matrices with `SPIKE::GPU` against that of the sparse solver `PARDISO`.

5.3 Test Results

Here, we present the parametric studies and performance comparisons for problems in the three groups described above.

Synthetic banded matrices. In a first set of experiments, we investigate the efficacity of the truncated SPIKE algorithm as a general preconditioner, even when used on matrices that are not diagonally dominant. Figure 4 displays the number of iterations in the outer BiCGStab(2) solver for a banded matrix of dimension $N = 100,000$ and half-bandwidth $K = 500$ for degrees of diagonal dominance in the range $0.2 \leq d \leq 1.2$, using both the truncated SPIKE and a banded-block-diagonal (BBD) preconditioner (equivalent to setting the spikes to zero). Results are presented for $P = 40$ partitions and indicate that the ability of the truncated SPIKE algorithm to capture some of the influence of the off-diagonal blocks $\mathbf{B}_i$ and $\mathbf{C}_i$ always leads to fewer BiCGStab(2) iterations than when these coupling blocks are ignored. Similar results were obtained for different number of partitions.

Next, we compare the performance of the `SPIKE::GPU` solver to the banded linear solver in `MKL`, for a variety of problem dimensions, half-bandwidths, and degree of diagonal dominance. Our solver outperforms `MKL` in all situations, as long as the problem size is sufficiently large ($N > 10,000$); see Figure (Figure 5(a)). On large, well-conditioned problems (Figure 5(b)), `SPIKE::GPU` is 2 to 3 times faster than `MKL`. As expected, the `MKL` banded solver is insensitive to changes in d. Somewhat surprising is the fact that the `SPIKE::GPU` solver demonstrated uniform performance over a wide range of degrees of diagonal dominance on the (relatively small-sized) matrices in Figure 5(c); `SPIKE::GPU` required only 1 or 2 outer loop iterations for all $d > 0.25$. As the degree of diagonal dominance decreases further, the number of iterations and hence the time to solution increase significantly, since the assumptions supporting the truncated SPIKE approximation are strongly violated.

Synthetic sparse matrices. Bringing into picture the reordering schemes described in §3, we performed similar parametric studies for sparse matrices obtained through random permutations of synthetically-generated banded matrices. The performance of the `SPIKE::GPU` solver on such matrices was now compared against `PARDISO`.

As shown in Figure 6, in all these experiments `SPIKE::GPU` outperforms `PARDISO` for large enough matrices ($N \geq 5,000$); moreover, `SPIKE::GPU` exhibits better parallel scaling. Note that the results presented here were obtained with no additional post-reordering element drop-off; judicious selection of the parameter δ in Equation (13), would further improve the observed performance of the `SPIKE::GPU` solver. Note also that, for the synthetic sparse matrices considered in this group, a study of the influence of the degree of diagonal dominance is more difficult to set up (due to the difficulty of taking into account the effect of the reordering algorithm) and is thus not provided.

Real application matrices. We run the tests of all selected real application matrices nightly [5]. On each result page, we report information of three types. First we report the information of each test matrix (matrix name, its size and whether it is symmetric positive definite). Secondly we report the options we solve the system. This includes how we precondition the problem, the number of partitions, and what Krylov method we use. At last, we also report all statistics. This mainly includes the half-bandwidths of original matrix, after MC64 reordering, after all reordering,

[3] In this case, we call the half-bandwidth of the banded seed matrix, the *pseudo half-bandwidth*.

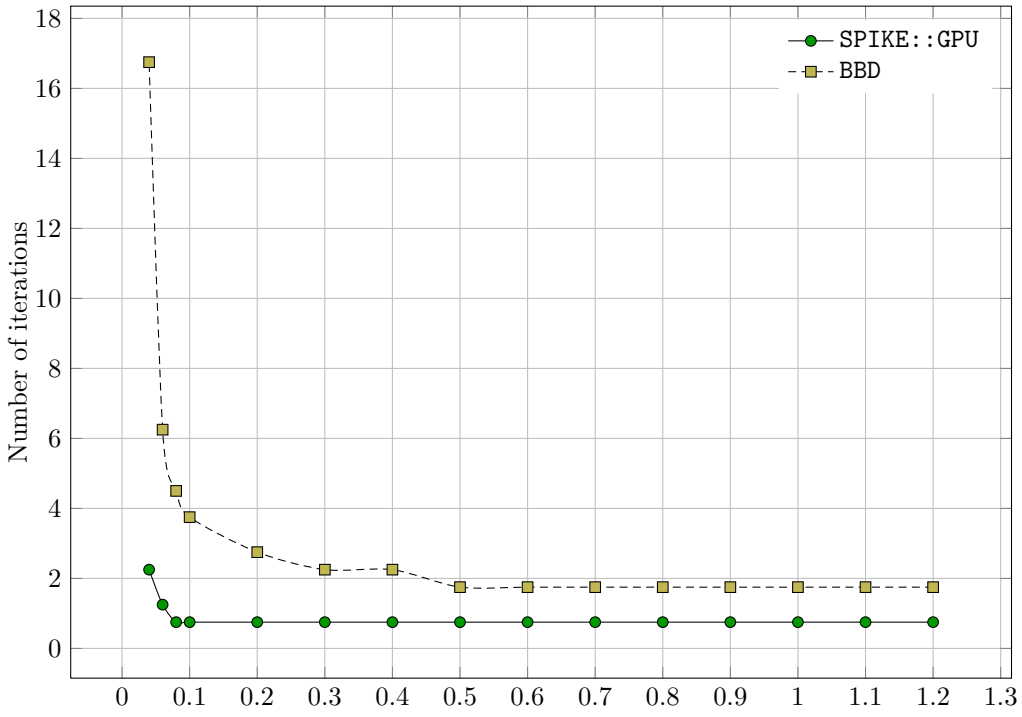


Figure 4: Number of BiCGStab(2) iterations using truncated SPIKE and a BBD preconditioner for banded matrices with $N = 100,000$ and $K = 500$, as function of the degree of diagonal dominance d.

and after drop-off. The statistics also includes all timing breakdowns and how we compare with the commercial solver `PARDISO` and how our own solver behaves with our historical best performance.

6 Conclusions and future work

This paper discusses the algorithm and GPGPU implementation details of `SPIKE::GPU`, a general sparse linear solver which relies on a SPIKE-based preconditioner within the framework of a Krylov-subspace method (in this present work, BiCGStab(2)). Numerical results demonstrate that `SPIKE::GPU` compares favorably in terms of performance to the `MKL` banded solver, when applied to large banded systems, and to `PARDISO`, when solving large sparse problems.

Technical effort will focus next on performance improvements:

- Investigate more sophisticated reordering techniques to achieve narrower bandwidth and boost diagonal dominance. In particular, we will consider various spectral methods, based on the Fiedler vector [6, 12], which hold the promise of significantly superior profile reduction.
- Extend the current implementation to support multiple GPUs. Global memory available on a single GPU; e.g., 5 GB on Tesla K20, is a limiting factor in terms of the problem size and/or post-reordering bandwidth that `SPIKE::GPU` can currently handle.
- Integrate `SPIKE::GPU` within the PETSc framework [17].

7 Acknowledgments

This work has been supported by National Science Foundation grant SI2-SSE 1147337.

References

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

[2] NVIDIA TESLA KEPLER GPU computing accelerators. http://www.nvidia.com/content/tesla, 2012.

[3] Intel Math Kernel Library (Intel MKL) 11.0. http://software.intel.com/en-us/intel-mkl, 2012.

[4] Tesla K20 GPU Accelerator. http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf, 2012.

[5] Nightly Execution Results. http://spikegpu.sbel.org/nightly_build, 2013.

[6] S. Barnard, A. Pothen, and H. Simon. A spectral algorithm for envelope reduction of sparse matrices. *Numerical Linear Algebra with Applications*, 2(4):317–334, 1995.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] Y. Hu and J. Scott. Multilevel algorithms for wavefront reduction. *SIAM Journal of Scientific Computing*, 23(4):1352–1375, 2001.

[13] 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.

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

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

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

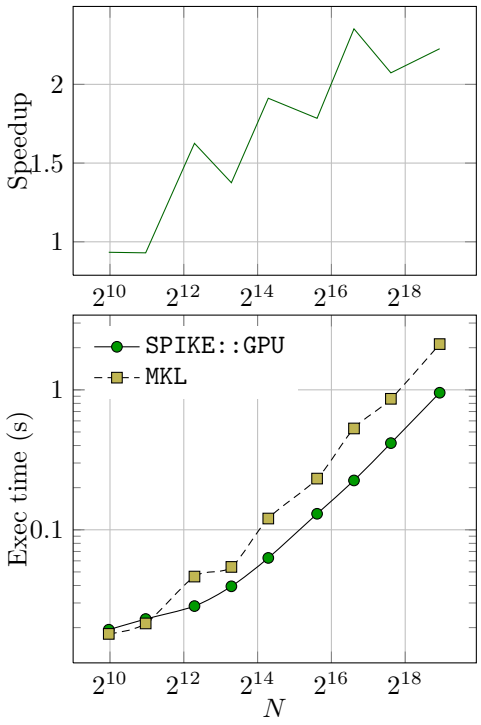
[17] B. S., B. J., B. K., E. V., G. W. D., K. D., K. M. G., M. L. C., S. B. F., and Z. H. PETSc Users Manual. Technical Report ANL-95/11, Argonne National Laboratory, 2011.

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

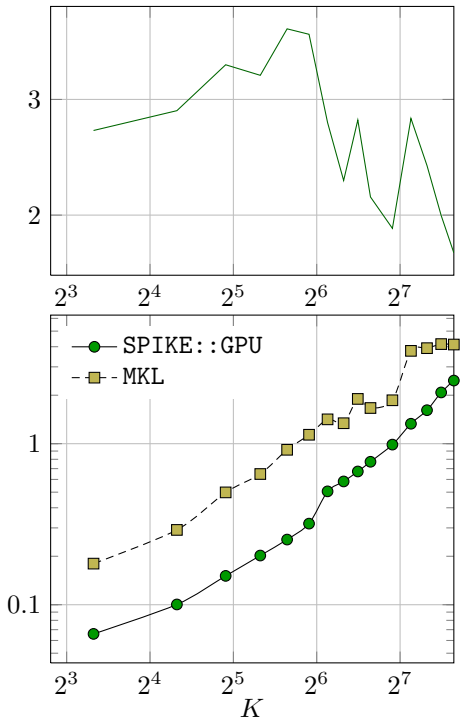
[19] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with pardiso. *Future Generation Computer Systems*, 20(3):475–487, 2004.

[20] 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.

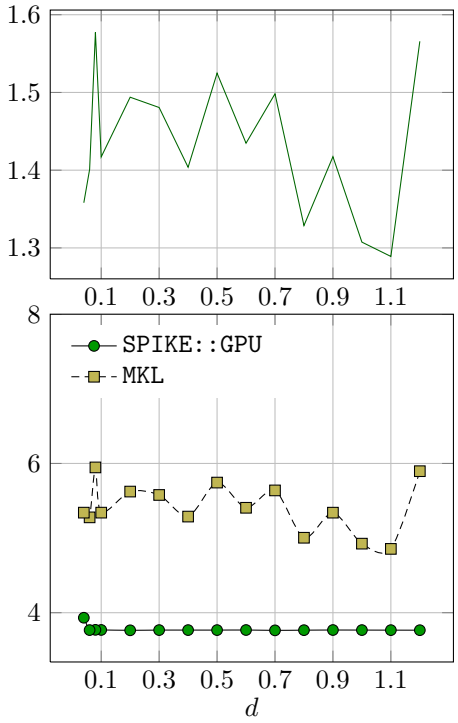
[21] G. Sleijpen and D. Fokkema. BiCGStab(l) for linear equations involving unsymmetric matrices with complex spectrum. *Electornic Transactions on Numerical Analysis*, 1:11–32, 1993.



(a) Influence of the problem dimension N for fixed values $K = 100$ and $d = 1$.

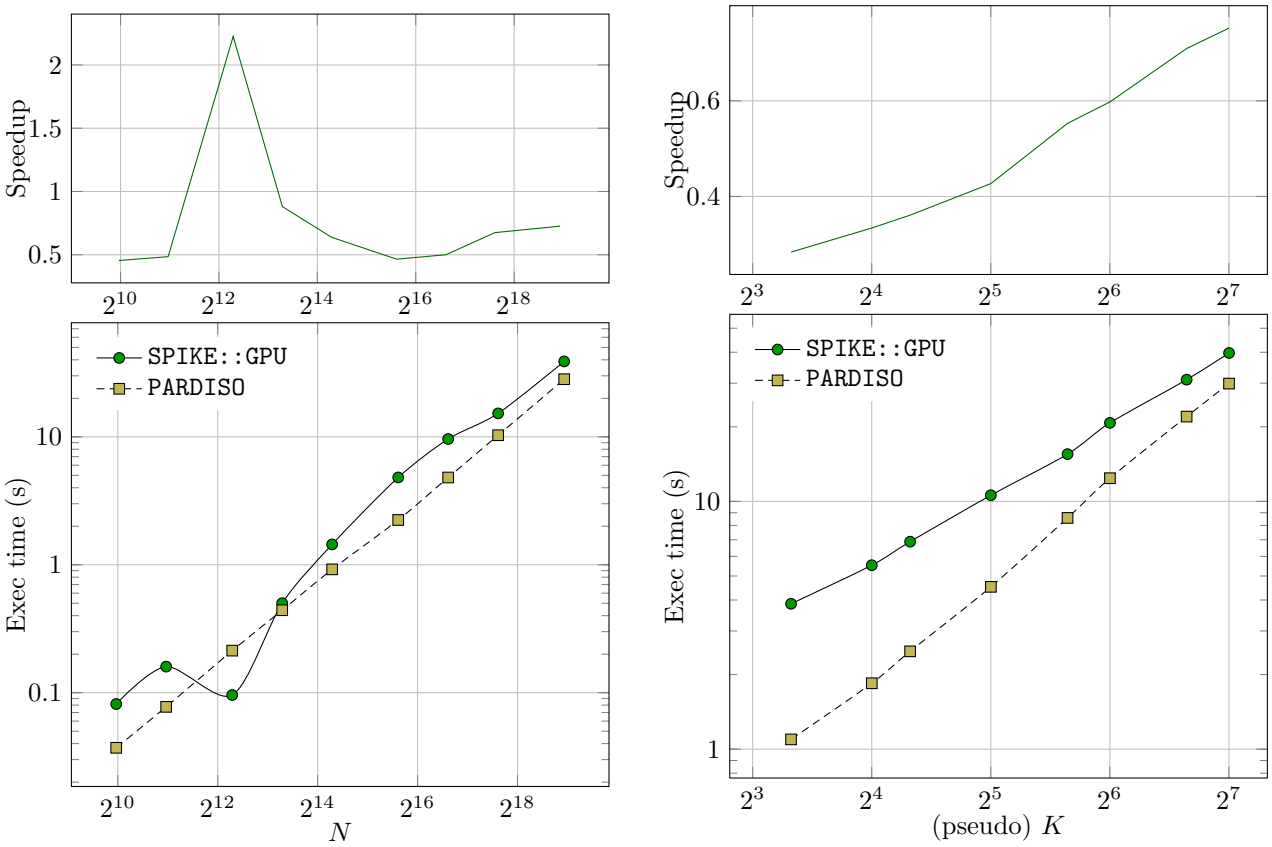


(b) Influence of the half-bandwidth K for fixed values $N = 1,000,000$ and $d = 1$.



(c) Influence of the degree of diagonal dominance d for fixed values $N = 100,000$ and $K = 500$.

Figure 5: Parametric studies on synthetic banded matrices. Timing results for `SPIKE::GPU` and `MKL` are provided when one of the parameters N (problem dimension), K (half-bandwidth), and d (degree of diagonal dominance) is varied, while the other two are kept constant.



(a) Influence of the problem dimension N for fixed values $K = 100$ and $d = 1$.

(b) Influence of the half-bandwidth K for fixed values $N = 400,000$ and $d = 1$.

Figure 6: Parametric studies on synthetic sparse matrices. Timing results for `SPIKE::GPU` and `PARDISO` are provided when varying N (problem dimension) or K (pseudo half-bandwidth), while keeping the other one fixed.