Computing dense and sparse matrix functions using GPUs

A. Lamas Daviña, J. E. Roman

D. Sistemes Informàtics i Computació, Universitat Politècnica de València, Camí de Vera s/n, 46022 València, Spain

Abstract

Matrix functions are relevant in many scientific computing applications, and several numerical methods for computing them have been proposed in the last years. We focus on the case that a function of a dense matrix of moderate size must be computed explicitly, but also cover the case of iterative computation of the action of a matrix function on a vector with Krylov methods. Accelerating these computations by exploiting GPU hardware is interesting to reduce the response time of simulations in large-scale scientific applications. The most common functions encountered in applications are the matrix exponential and the matrix square root. We focus on these, and evaluate the gain obtained with different algorithms whose implementation is susceptible of achieving high arithmetic intensity on GPUs. We show results comparing these implementations with thread-parallel CPU implementations.

Keywords: GPU computing, matrix square root, matrix exponential, Krylov methods

---

*This work was partially supported by the Spanish Ministry of Economy and Competitiveness under grant TIN2016-75985-P (including European Commission FEDER funds). Alejandro Lamas was supported by the Spanish Ministry of Education, Culture and Sport through grant FPU13-06655.

*Corresponding author.

Email addresses: alejandro.lamas@dsic.upv.es (A. Lamas Daviña), jroman@dsic.upv.es (J. E. Roman)
1. Introduction

The area of matrix functions has developed significantly in the last years, both in formalizing the relevant theoretical concepts and in developing algorithms for their computation [1, 2]. Matrix functions can be found in many scientific computing applications, especially the exponential and the square root, but also other less frequent functions such as the logarithm, or trigonometric functions. We consider functions that map $\mathbb{C}^{n \times n}$ to $\mathbb{C}^{n \times n}$ and are defined in terms of a scalar function $f$. For instance, the matrix exponential $\exp(A)$ is defined in terms of the scalar exponential function $\exp(z)$ in the sense discussed below, not in other senses such as element-wise evaluation. There are several equivalent definitions for such matrix functions. One definition is based on the Jordan canonical form of the matrix, $A = Z J Z^{-1}$. Then

$$f(A) := Z \text{diag} \left( f(J_1(\lambda_{i_1})), \ldots, f(J_p(\lambda_{i_p})) \right) Z^{-1},$$

(1)

where $J_k(\lambda_{i_k})$ is the Jordan block corresponding to the eigenvalue $\lambda_{i_k}$, and $f(J_k(\lambda_{i_k}))$ can be defined in terms of the successive derivatives of $f$ evaluated on $\lambda_{i_k}$. Alternatively, matrix functions can be defined in terms of the polynomial that interpolates $f$ in the Hermite sense at the eigenvalues $\lambda_{i_k}$, and also by means of the Cauchy integral formula (assuming $f$ is analytic in the considered domain). All these definitions give rise to different computational methods.

As in other numerical linear algebra problems, such as linear systems or eigensystems, there are two broad classes of methods that we can generically refer to as dense and sparse. For dense matrices of relatively small size we can afford to explicitly compute the matrix $F = f(A)$ with dense methods. The computational cost of these methods depends cubically on $n$, the matrix size.

For sparse matrices, whose size is usually much larger, it is not possible to explicitly compute (or even store) the matrix $F = f(A)$, so the alternative is the implicit application of the matrix function to a given vector, $y = f(A)b$. The cost in this case is less than cubic, but this cost is replicated for every different vector $b$ to which the matrix function is applied.

This work focuses mainly on dense computations, but we will also discuss the sparse case, particularly Krylov methods for the iterative computation of $y = f(A)b$, where $A$ is typically large and sparse, or is available implicitly by means of a matrix-vector multiplication subroutine. Krylov methods basically amount to building an Arnoldi decomposition and evaluating the matrix function on the computed Hessenberg matrix. In an MPI parallel
implementation of this method, the former can be implemented efficiently, but the latter is a dense matrix computation, that must be done redundantly by all MPI processes. This redundant computation is in the critical path, so it may hinder performance if the size of the Hessenberg matrix is not very small, as it may happen when using restart techniques. In this scenario, having GPU implementations of the methods for dense matrix functions is very interesting as it may help improve the overall scalability of parallel Krylov solvers.

Hence, one of our motivations for providing GPU implementations of methods for dense matrix function computation is to alleviate the cost present in the critical path of parallel Krylov methods for the computation of \( f(A)b \) (assuming that every MPI process has access to a GPU). Another motivation is to explicitly compute \( f(A) \) in applications where the matrix \( A \) is not too large.

Previous works have compared different algorithms for matrix functions, for instance [3] analyzes Matlab implementations for the matrix exponential. Our contribution is not only to carry out a comparison, but also provide robust and efficient implementations for both CPU and GPU, that can be used from sequential or parallel codes written in C/C++ or Fortran. We have implemented different matrix function solvers in the SLEPc library [4]. They are included in the public release of SLEPc, and are already being used in different scientific computing applications. Our implementations can operate with either real or complex scalars, in single or double precision arithmetic (although we will show results for double precision only).

Other authors have considered the topic of GPU calculation of matrix functions, particularly the case of computing \( \exp(tA)b \) with iterative methods [5, 6]. Here, we cover both the exponential and the square root, and dense methods as well as sparse.

The main goals of this paper are: (1) to show which methods for the dense computation of matrix square roots and exponentials can be efficiently implemented on GPU; (2) to compare the performance of these methods; (3) to give the practitioners an idea of the speedup that GPU solvers can attain with respect to a thread-parallel CPU code; and (4) to advertise the availability of dense and sparse solvers in SLEPc.

The methods considered in this paper are not new, so our main contribution is to provide efficient implementations in SLEPc. These implementations are coded as a sequence of function calls to libraries such as the BLAS, relying on their high efficiency either on the CPU or the GPU.
The rest of the paper is organized as follows. Section 2 explains the usage of GPUs in SLEPc. Section 3 briefly describes a few motivating applications that we have used for evaluating the implementations. Section 4 contains a description of the considered methods for dense matrices, and provides computational results for the implemented solvers. Krylov methods are covered in section 5, including the evaluation of its performance. Some final remarks are given in section 6.

2. GPU implementation in SLEPc

Usage of graphics processing units in SLEPc relies on the design of the PETSc library [7], on which SLEPc is based. PETSc is an object oriented parallel toolkit meant to solve problems arising in PDE-based scientific computing applications. It uses MPI for the coarse grain parallelization, and also provides some support for GPU computing. PETSc represents vectors and matrices as a set of encapsulated data structures, and provides a uniform interface to various solvers for systems of linear and nonlinear equations, and related problems.

PETSc provides support for using GPUs transparently [8]. One of the different approaches that PETSc uses to do the computation on the GPU makes use of the libraries available in the NVIDIA Toolkit, particularly cuBLAS and cuSPARSE. PETSc’s vector and (sparse) matrix classes mirror the underlying data structures in the memory of the GPU, and transfer the computational effort to the graphics cards, all this transparently to the calling code. A coherence mechanism is responsible of the consistency of the data between CPU and GPU memory. When a node has more than one GPU available, different MPI processes can drive different GPUs, and they use their rank to establish the preference for a specific GPU.

Besides the transparent use of the GPU offered by PETSc objects, they also provide direct access to the allocated memory (being it on CPU or on GPU). This feature, together with the possibility of configuring PETSc to link with third party libraries, enables the use of other GPU-aware libraries such as MAGMA from SLEPc or from user-provided subroutines.

PETSc has support for dense matrices on the CPU by means of the MATDENSE matrix object type. Unfortunately, it does not provide an equivalent dense type for the GPU that synchronizes CPU and GPU memory and offers access to the device memory. In order to perform dense matrix computations on the GPU, a subroutine has to allocate space on the device, get access to
the CPU memory and copy the data. Once the subroutine has finished, the result has to be copied back to the CPU memory, and the device memory freed.

SLEPc builds its GPU support from this basis, and is able to enhance the computational intensity of its solvers by extending PETSc’s functionality by furnishing multi-vector operations on the GPU, on which BLAS-1 routines are replaced with BLAS-2 or even, in some cases, BLAS-3 routines.

A core subroutine in SLEPc is the construction of the Krylov subspace by means of the Arnoldi method, whose main operations are the sparse matrix-vector product and the orthogonalization of vectors. The involved data objects are (sparse) matrices and vectors. A data object in PETSc can be instantiated, programmatically and at run time, with one of multiple types which define the underlying storage representation used. For instance, the object type of a matrix can be set by means of the MatSetType function and by means of the -mat_type run time parameter. By using the aijcusparse storage of PETSc, the matrices will be of an object type (MATAIJCUSPARSE) that replicates its content in the GPU memory and do the computations on the GPU. Similarly, vectors created from such matrices will have a compatible type (VECCUDA) and hence GPU support. This allows SLEPc to perform the matrix-vector multiplication to expand the Krylov subspace, and the orthogonalization and normalization of the vectors, efficiently on the GPU.

Dense matrix functions can also be computed on the GPU, with special code that implements the methods described in section 4.

3. Sample applications

Before presenting the methods, we describe four applications with the aim of showing that different functions appear in different contexts. In most cases, it is sufficient to compute the action of the matrix function on a vector, so sparse methods would be enough. Still, we use these matrices also to test the dense algorithms, since they are more representative of real situations than, e.g., random matrices.

1. Data assimilation. Numerical weather prediction relies on modern data assimilation techniques for merging satellite observations (of order $10^5$ or more) with model forecasts in order to improve the initial conditions and hence obtain more accurate results. The EnSRF method [9] is a variant of
Ensemble Kalman Filter used with deterministic observations that includes a matrix square root to account for the uncertainty of the unperturbed ensemble observations.

For the ensemble mean \( x_m \) and ensemble perturbations \( X_A \), the square-root observation filter can be written as

\[
x_m^{(a)} = x_m^{(f)} + K(y - HX),
\]

\[
X_A^{(a)} = X_A^{(f)} + \tilde{K}(0 - HA),
\]

where the superscripts \( (a) \) and \( (f) \) denote the analysis and the previous forecast, respectively. Vector \( y \) contains the observations. The traditional Kalman gain is \( K = C_{x,y}D^{-1} \), with \( D = C_{y,y} + R \), where \( C_{x,y} \) and \( C_{y,y} \) are covariance matrices and \( R \) the observation error covariance. The correction from using unperturbed observations is \( \tilde{K} = C_{x,y}D^{-1/2}(\sqrt{D} + \sqrt{R})^{-1} \), which simplifies to \( \tilde{K} = C_{x,y}(D + \sqrt{D})^{-1} \). Further details can be found in [10].

2. Time evolution in quantum problems. The time-dependent Schrödinger equation \( i\frac{\partial}{\partial t}\Psi(t) = H(t)\Psi(t) \) can be solved numerically with a time-stepping scheme, where at time \( t_n \)

\[
\Psi(t_n) = e^{-iH(t_n)\Delta t} \Psi(t_{n-1})
\]

with \( \Delta t = t_n - t_{n-1} \). See [11] for an example that uses the matrix exponential in SLEPc for the simulation of quantum systems.

Here we will use a very simple problem where the Hamiltonian \( H \) is constant in all time steps, consisting in \( N \) spins in a uniform transverse field, with disordered potential. The dimension of the Hamiltonian matrix is \( 2^N \).

We emphasize that complex arithmetic is required to compute \( \exp(-iH\Delta t) \), even if \( H \) is real.

3. Migration modeling. In the last years, population genomic datasets have been collected, making it possible to use these data to infer demographic histories of populations, e.g., under models of migration and divergence. The method presented in [12] relies on a Markov chain representation, and uses the matrix exponential to obtain probability distributions at different times. More precisely, if \( M \) is the transition matrix, the vector \( \pi(t) \) of probabilities of being in each state of the Markov chain at time \( t \) is

\[
\pi(t) = \pi(0)e^{tM}.
\]
4. Stability of dynamical systems. In control theory and many other contexts, it is important to determine if a dynamical system is stable or not, and this question can often be formulated as a problem involving eigenvalues of matrices. In this context, a matrix $A$ is considered to be a stable matrix if all its eigenvalues lie in the open left half plane, $\text{Re}[\lambda_i] < 0$ for all $i$. As suggested in [1, §2.5], the matrix sign function can be used to count the number of eigenvalues of a matrix located in a particular region of the complex plane, and determine if a system is stable or unstable. The matrix sign function is related to the matrix square root, as discussed in section 4.2.

4. Dense methods for matrix functions

There are many different methods for the dense computation of $f(A)$. Here we point out three classes of methods that are relevant for our purposes:

- Similarity transformation. Computing $f(A)$ via (1) is not viable due to the difficulty of obtaining the Jordan form in a numerically stable way. An alternative is to compute the Schur form $A = QTQ^*$, with $Q$ orthogonal (or unitary in the complex case), and then evaluate the matrix function of the (quasi-)triangular matrix $T$. In the case that $A$ is symmetric (or Hermitian), it is even simpler because $T$ is then diagonal. The latter case will be referred to as diagonalization.

- Rational approximation, by means of Padé approximants.

- Matrix iterations, derived from Newton’s method (mainly for matrix roots).

Methods based on similarity transformation are appealing since they usually require less floating-point operations than methods of the other types. However, that same characteristic makes them less likely to take advantage of the GPU.

We have implemented four methods for the matrix square root and three for the matrix exponential. All of them have been implemented for both CPU and GPU, except the Schur method for the square root that is only available for CPU.
Algorithm 1: Blocked Schur method for the square root

Compute (real) Schur decomposition \( A = QTQ^* \);

for \( j = 1, 2, \ldots, n_{blk} \) do
  Evaluate \( X_{jj} = T_{jj}^{1/2} \);
  for \( i = j - 1, \ldots, 1 \) do
    Solve Sylvester eq. \( X_{ii}X_{ij} - X_{ij}X_{jj} = T_{ij} - \sum_{k=j+1}^{i-1} X_{ik}X_{kj} \)
  end
end

Backtransform \( F = QXQ^* \)

4.1. Square root

A square root of \( A \) is any matrix satisfying the matrix equation \( F^2 = A \). If \( A \) has no eigenvalues on \( \mathbb{R}^- \), the closed negative real axis, there is a unique principal square root, denoted as \( A^{1/2} \), whose eigenvalues have all positive real part. The methods discussed below compute the principal square root.

As mentioned above, an interesting strategy is to first reduce \( A \) to the Schur form, \( A = QTQ^* \). The Schur-Parlett method [2] uses a recurrence to evaluate \( f(T) \) exploiting the (quasi-)triangular structure. Implementing the Parlett recurrence in a numerically stable way is tricky. Fortunately, in the case of the matrix square root the method can be simplified, as the Parlett recurrence is not necessary, and hence it is simply called Schur method [13]. In this method the diagonal elements are determined as \( \sqrt{T_{ii}} \) and the off-diagonal ones can be obtained from the equation \( X^2 = T \). Once \( X \), the square root of \( T \), has been obtained, a final backtransform step is necessary to recover \( F \). For the CPU version, we have implemented a blocked variant (with block size of 64) as described in [14], see Algorithm 1. Although when solving in CPU, the \_gemm calls are the most time consuming operations, this method is not appropriate for implementation on GPU, due to the need of reduction to the Schur form and the steps after it, that in the real case, involve using level 2 BLAS to solve Sylvester equations.

The other methods that we consider are based on matrix iterations. Iterative methods are interesting to implement as they are easily built with common matrix operations, and are particularly rich in matrix-matrix products. Those characteristics make them a suitable choice for GPU computing.

The Newton method is the basis of many existing iterative methods.
When applied to the matrix equation $F^2 = A$, it gives the recurrence
\[
F_{k+1} = \frac{1}{2}(F_k + F_k^{-1}A), \quad F_0 = A. \tag{2}
\]

For $A$ having no eigenvalues on $\mathbb{R}^-$, the recurrence converges quadratically to the principal square root $A^{1/2}$ for $F_0$ sufficiently close to it. Nevertheless, this iteration is numerically unstable and is not useful for practical computation. It is necessary to rewrite the iteration in a different way to make it stable. Several variants of the Newton method that stabilize the iteration have been developed, and they often scale the $F_k$ terms to reduce the steps until the quadratic convergence starts.

One variant of the Newton method is the product form [15] of the Denman–Beavers iteration [16], with a scaling factor $\mu_k$:
\[
\begin{align*}
\mu_k &= |\det(M_k)|^{-1/(2n)}, \\
M_{k+1} &= \frac{1}{2} \left( I + \frac{\mu_k^2 M_k + \mu_k^{-2} M_k^{-1}}{2} \right), \quad M_0 = A, \\
F_{k+1} &= \frac{1}{2} \mu_k F_k (I + \mu_k^{-2} M_k^{-1}), \quad F_0 = A.
\end{align*} \tag{3}
\]

Contrary to the recurrence (2), that solves a multiple right-hand side linear system per iteration, (3) requires to compute a matrix inverse and, in addition, obtaining the scaling factor $\mu_k$ also requires computing the determinant of $M_k$. Following Higham’s reference implementation\(^1\), the scaling can be stopped when $\|F_k - F_{k-1}\|/\|F_k\| < 10^{-2}$. In our implementation, the determinant is computed from the LU factorization of $M_k$ (\texttt{getrf}). However, for the computational experiments in this section we have turned off the scaling in this method, as the determinant may suffer from overflow or underflow in finite precision arithmetic when the matrix is quite large. The scaling factor $\mu_k$ may accelerate the initial convergence, but the results obtained without scaling are equally accurate. Alternative scaling factors discussed in [1, Ch. 5] in the context of the matrix sign function are not suitable for the case of the square root.

Another method for computing the square root is Newton–Schulz [17], from the Padé family of iterations. It is an inverse-free iterative method.

\(^1\)The Matrix Function Toolbox, http://www.ma.man.ac.uk/~higham/mftoolbox/
It usually needs more iterations to converge, but it does not rely on having an efficient implementation of the matrix inverse, having the matrix-matrix multiplication as its main operation. It consists of two coupled recurrences,

\[
X_{k+1} = \frac{1}{2}X_k(3I - Z_kX_k), \quad X_0 = B, \\
Z_{k+1} = \frac{1}{2}(3I - Z_kX_k)Z_k, \quad Z_0 = I.
\] (4)

The initial guess \( B = A/\|I - A\|_p \) is a scaled version of \( A \), so scaling must be undone after convergence, \( F = \sqrt{\|I - A\|_p}X \). The condition \( \|I - A\|_p < 1 \) is sufficient for the method to converge for \( p = 1, 2 \) or \( \infty \). The scaling of the matrix may help the convergence of the method, although it does not guarantee it.

Recently, a cubically convergent iterative method has been developed by Sadeghi [18] that starts with \( B = A \) (or \( B = A/\|A\| \) in case \( \rho(A) > 1 \)), and builds two coupled recurrences

\[
X_{k+1} = X_k \cdot \left( \frac{5}{16}I_n + \frac{1}{16}M_k(15I_n - 5M_k + M_k^2) \right), \quad X_0 = I, \\
M_{k+1} = M_k \cdot \left( \frac{5}{16}I_n + \frac{1}{16}M_k(15I_n - 5M_k + M_k^2) \right)^{-2}, \quad M_0 = B,
\] (5)

being \( F = \sqrt{\|A\|}X \) when the initial guess has been scaled or \( F = X \) otherwise. This method, like (3), requires computing an inverse in each iteration. But as it converges faster, the number of iterations needed to compute the square root is expected to be smaller.

In our implementations, we use the Frobenius norm for scaling as well as for convergence tests.

4.2. Sign

Assuming that \( A \) is non-singular and has no eigenvalues on the imaginary axis, the matrix sign function \( \text{sign}(A) \) can be defined. Let

\[
A = Z \begin{bmatrix} J_n & \ast \\ \ast & J_p \end{bmatrix} Z^{-1}
\] (6)

be the Jordan canonical form of \( A \), where \( J_n \) and \( J_p \) are the Jordan blocks of eigenvalues with negative and positive real parts, respectively, being \( n \) and
Then the sign function of $A$ is defined as

$$\text{sign}(A) = Z \begin{bmatrix} -I_n & \ I_p \end{bmatrix} Z^{-1}. \quad (7)$$

A different representation of the matrix sign function, introduced in [19], shows the link with the matrix square root:

$$\text{sign}(A) = A(A^2)^{-1/2}. \quad (8)$$

Although there are specific Newton-type iterations for $\text{sign}(A)$ [1, Ch. 5], here we discuss its computation using square root solvers. Since $A$ is assumed to be non-singular, $(A^2)^{-1/2}$ can be obtained. Some of the methods for the square root mentioned above can be used to obtain the inverse square root. In particular, in Denman–Beavers (3) the sequence $F_k$ converges to $A^{-1/2}$ if the iteration starts with $F_0 = I$, and in Newton–Schulz (4) the sequence $Z_k$ converges to $B^{-1/2}$ so both the square root and its inverse are obtained simultaneously. If using another method to compute the square root, like Schur or Sadeghi, the inverse square root can be obtained by additionally solving a system of linear equations with multiple right-hand sides

$$AF = A^{1/2}, \quad (9)$$

where $F = A^{-1/2}$.

### 4.3. Exponential

For the matrix exponential, we use a rational function $r(A)$ to approximate $\exp(A)$. This rational function is chosen in a similar way as in scalar approximation theory, although in the case of matrix functions there is no guarantee that the approximation will be good (depending on the spectral properties of $A$) [2].

The first method that we have implemented is a rational approximation based on Padé approximants of order $[p/p]$ with $p = 6$, combined with scaling and squaring. The notation $[\cdot/\cdot]$ indicates the polynomial degree of the numerator and denominator, which are equal in this case (diagonal Padé approximant). The evaluation of the numerator and denominator is done following the Horner scheme, to reduce the number of required matrix-matrix multiplications. The last step for computing the rational function is done with a linear solve with multiple right-hand sides. The scaling and squaring
technique consists in determining the minimal integer $s \geq 0$ such that $\|A/2^s\|$ is smaller than a certain constant, then scale the matrix as $A/2^s$ prior to the computation of the rational matrix function. In that case, a post-processing is required to form $F^{2^s}$, which is done with $s$ additional matrix-matrix multiplications.

The above technique is very close to one of the methods used in Expokit [20]. A more recent work [21] suggests using higher degree Padé approximants (up to degree 13), but rearranging the computation in such a way that the number of required matrix products is much smaller. This approach (included in the latest versions of Matlab) is more accurate in some cases, and also avoids choosing a too large value of $s$ (overscaling). Our second implementation follows this approach, and will be referred to as Higham.

A third method, presented by Gütteil and Nakatsukasa in [22], also based on Padé approximations, tries to reduce its cost by using a subdiagonal Padé approximant of low degree (such as $[3/4]$), and also uses a small scaling and squaring factor to avoid potential instability caused by overscaling. Before starting, the method shifts the matrix

$$A_\sigma = A - \sigma I,$$  \hspace{1cm} (10)

being $\sigma$ the real part of the rightmost eigenvalue of $A$, so that all the eigenvalues of $A_\sigma$ are in the left half plane. It is supposed to perform well with an estimation of such eigenvalue, assuming that the rightmost eigenvalues do not have widely varying imaginary parts. If the matrix is stable, with no eigenvalues in the right half plane, the shift is not required. A drawback of this method is that it employs complex arithmetic even when $A$ is real, so its performance with real matrices is not expected to be good when compared with methods that do the computation with real scalars. This last implementation is referred to as Gütteil–Nakatsukasa.

4.4. Computational evaluation of dense solvers

We have analyzed the performance of the dense solvers implementations by conducting several tests on two platforms, with the following characteristics:

**Fermi** 2 Intel Xeon E5649 processors (6 cores per processor) at 2.53 GHz with 24 GB of main memory; 2 GPUs NVIDIA Tesla M2090, with 512 cores and 6 GB GDDR per GPU.
Kepler 2 Intel Xeon E5-2630 v3 processors (8 cores per processor) at 2.4 GHz with 128 GB of main memory; 2 NVIDIA K80 cards (2 GPUs each), with 2496 cores and 12 GB GDDR per GPU.

On both platforms, the codes are compiled with GCC 5.1.0, and linked against PETSc 3.8, SLEPc 3.8, CUDA 8.0, MAGMA 2.2.0 and MKL 11.3.2. The algorithm’s implementations are built using BLAS and LAPACK operations as main computational blocks. In general, the GPU implementations make use of cuBLAS to perform the BLAS operations and MAGMA for the LAPACK routines. Some auxiliary kernels are used for simple, non computationally intensive operations, like setting or modifying the diagonal elements of a matrix.

Running on two platforms with consecutive generations of GPUs, allows us not just to see the gain obtained with their use, but also the evolution of such cards compared with their coetaneous generations of CPUs.

We make use of PETSc’s logging functionality to measure the time and flops of the solvers. All the tables in this section show the total elapsed time needed to compute the respective (dense) matrix function and the achieved performance in gigaflops per second. The time includes, in all cases, the memory allocation (and deallocation) of the auxiliary variables needed, and in the case of the GPU implementations, also the copies to and from the device memory. The reported times are the minimum from three independent executions working in double precision arithmetic. When using sparse matrices during the experiments of this section, they are stored and treated as dense. All the experiments with dense solvers consist in a CPU execution, using as many threads as available cores (12 threads in Fermi and 16 threads in Kepler), and a GPU execution using only one of the available GPUs. The size of the matrices used in the tests is limited by the memory available in the GPU of the Fermi platform (6 GB).

Two matrices are considered for the matrix square root experiments. The first one is ensrf7864, a symmetric positive-definite matrix from Application 1 of section 3, with dimension 7864, for which the square root is computed. The second one is rdb5000, a sparse non-symmetric matrix belonging to the NEP collection [23], with dimension 5000, which is used to compute the matrix inverse square root, as part of the computation of the matrix sign function for Application 4.

Apart from the performance data, the tables for the square root tests (Tables 1 and 2) also show the relative error of the computed solution, $\|F^2 -$
Table 1: Results for computing the matrix square root of the ensrf7864 matrix. Time expressed in seconds. GF/s indicates gigaflops per second, Iter indicates iterations done, and Error is computed as $\|F^2 - A\|_F/\|A\|_F$.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Algorithm</th>
<th>CPU</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>GPU</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>GF/s</td>
<td>Iter.</td>
<td>Error</td>
<td>Time</td>
<td>GF/s</td>
<td>Iter.</td>
<td>Error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fermi</td>
<td>Diagonalization</td>
<td>196.9</td>
<td>27</td>
<td>-</td>
<td>1.5 · 10^{-14}</td>
<td>56.6</td>
<td>344</td>
<td>10</td>
<td>9.3 · 10^{-14}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Denman–Beavers</td>
<td>208.3</td>
<td>93</td>
<td>10</td>
<td>4.2 · 10^{-14}</td>
<td>127.0</td>
<td>391</td>
<td>17</td>
<td>3.0 · 10^{-14}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Newton–Schulz</td>
<td>489.4</td>
<td>101</td>
<td>17</td>
<td>7.4 · 10^{-15}</td>
<td>93.7</td>
<td>374</td>
<td>6</td>
<td>2.4 · 10^{-14}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sadeghi</td>
<td>346.5</td>
<td>101</td>
<td>6</td>
<td>3.9 · 10^{-15}</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kepler</td>
<td>Diagonalization</td>
<td>37.3</td>
<td>143</td>
<td>-</td>
<td>1.2 · 10^{-14}</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Denman–Beavers</td>
<td>74.4</td>
<td>262</td>
<td>10</td>
<td>4.4 · 10^{-14}</td>
<td>25.8</td>
<td>754</td>
<td>10</td>
<td>9.1 · 10^{-14}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Newton–Schulz</td>
<td>135.0</td>
<td>367</td>
<td>17</td>
<td>6.5 · 10^{-15}</td>
<td>49.6</td>
<td>1001</td>
<td>17</td>
<td>3.0 · 10^{-14}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sadeghi</td>
<td>88.2</td>
<td>397</td>
<td>6</td>
<td>3.9 · 10^{-15}</td>
<td>38.6</td>
<td>907</td>
<td>6</td>
<td>2.4 · 10^{-14}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Time results for computing the matrix square root (left) and the matrix inverse square root (right).
$A_F/\|A\|_F$ for the square root and $\|F^2A - I\|_F/\|A\|_F$ for the inverse square root. According to Higham [1, §6.1], the best relative error we can expect for the numerical computation of the matrix square root is of order $\alpha(F)u$, where $u$ is the unit roundoff and $\alpha(F) = \|F\|^2/\|A\|$. We have checked that in the considered matrices this quantity is of order $10^1$, and hence the maximum expected accuracy in double precision would be around $10^{-15}$. The tables also show the number of required iterations in the case of iterative methods. Figure 1 summarizes the square root and inverse square root executions, on which the fastest method is Denman–Beavers running on GPU.

Table 1 shows the results for the square root. Given that the matrix is symmetric, $f(A)$ can be computed as $Q\text{diag}(f(\lambda_i))Q^*$. This diagonalization method is the fastest one for the CPU runs, and the ranking between methods is the same on both platforms, but with notable differences with respect to their relative execution time.

The relative behaviour of the methods does not differ substantially between CPU and GPU executions. On the Fermi platform, the GPU runs obtain speedups ranging from 3.6 to 3.8 with respect to the CPU ones, and a speedup of 3.5 is obtained if comparing the fastest method on GPU (Denman–Beavers) with the fastest one on CPU (diagonalization). On the Kepler platform, the speedups achieved with the GPU runs are smaller, going from 2.3 to 2.9. And the comparison between the fastest methods on CPU and GPU gives a speedup of only 1.4.

Although Newton–Schulz is the method that attains the highest Gflop/s rate in almost all the cases due to the matrix-matrix operations, it is not competitive in terms of execution time because of the larger number of required iterations. Neither is Sadeghi despite its cubic convergence that allows it to terminate in only six iterations.

Table 2 shows the results of the inverse square root with the non-symmetric matrix. The Newton–Schulz method does not converge with this matrix, so no results are shown for it. The poor error obtained with Schur and Sadeghi comes from the final linear solve to obtain the inverse, not from computing the matrix square root. This additional step also implies increasing the time needed to obtain the inverse square root. Denman–Beavers is the fastest method on CPU and GPU on both platforms, achieving speedups of 3.3 on Fermi and 2.3 on Kepler.

Four matrices are considered for the matrix exponential tests (the first two belong to the Harwell-Boeing collection [24] and are commonly used
Table 2: Results for computing the matrix inverse square root of the rdb5000 matrix. Time expressed in seconds. GF/s indicates gigaflops per second, Iter indicates iterations done, and Error is computed as $\|F^2A - I\|_F/\|A\|_F$.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Algorithm</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>GF/s</td>
</tr>
<tr>
<td>Fermi</td>
<td>Schur</td>
<td>271.1</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Denman–Beavers</td>
<td>141.9</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>Sadeghi</td>
<td>393.8</td>
<td>101</td>
</tr>
<tr>
<td>Kepler</td>
<td>Schur</td>
<td>210.7</td>
<td>33.6</td>
</tr>
<tr>
<td></td>
<td>Denman–Beavers</td>
<td>49.8</td>
<td>241.0</td>
</tr>
<tr>
<td></td>
<td>Sadeghi</td>
<td>108.6</td>
<td>365.3</td>
</tr>
</tbody>
</table>

as benchmarks: orani678, nonsymmetric sparse matrix of order 2529, and bcspr10, symmetric sparse matrix of order 5300; te12, a complex matrix from Application 2 of section 3, with dimension 4096; and inclam55, a matrix from Application 3, sparse, nonsymmetric of dimension 6770.

Tables 3 and 4, and Figures 2 and 3 show the results for each of the three methods implemented to compute the matrix exponential function. Tables 3 and 4 also show the absolute error of the computed solution, $\|F - F_m\|_F$, taking as a reference the computation done in Matlab, $F_m$, that uses the algorithm described in [14].

When using Güttel–Nakatsukasa, computing the eigenvalues of $A$ to shift the matrix implies adding too much overhead to the method. For the executions, we have disabled that computation and eliminated the shift. Also, since most of the operations in this method are performed in complex arithmetic in the case of real matrices, the flop count discerns between real and complex operations.

The results of the executions on the Fermi platform are displayed in Table 3 and Figure 2. They show that Higham’s method (Padé up to degree 13) is faster than basic Padé and Güttel–Nakatsukasa, but it is slightly less efficient in terms of computational intensity. The high number of Gflop/s achieved by Güttel–Nakatsukasa with real matrices, comes from unconditionally using complex arithmetic. The speedup obtained with the GPU on this platform is smaller than in the square root case, ranging from 1.3 to 3.0 with the fastest method. Higham’s method always attains the smallest error on all
Table 3: Results for the matrix exponential running on the Fermi platform. Time expressed in seconds. GF/s indicates gigaflops per second.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Algorithm</th>
<th>CPU Time</th>
<th>CPU GF/s</th>
<th>CPU $|F - F_m|_F$</th>
<th>GPU Time</th>
<th>GPU GF/s</th>
<th>GPU $|F - F_m|_F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>orani678</td>
<td>Higham</td>
<td>3.1</td>
<td>67</td>
<td>$4.1 \cdot 10^{-12}$</td>
<td>2.5</td>
<td>83</td>
<td>$4.1 \cdot 10^{-12}$</td>
</tr>
<tr>
<td></td>
<td>Padé</td>
<td>5.0</td>
<td>80</td>
<td>$4.1 \cdot 10^{-12}$</td>
<td>2.3</td>
<td>219</td>
<td>$6.6 \cdot 10^{-12}$</td>
</tr>
<tr>
<td></td>
<td>Güttel–Nakatsukasa‡</td>
<td>15.5</td>
<td>97</td>
<td>$3.9 \cdot 10^{-12}$</td>
<td>5.7</td>
<td>260</td>
<td>$4.7 \cdot 10^{-12}$</td>
</tr>
<tr>
<td>bcsprw10</td>
<td>Higham</td>
<td>27.2</td>
<td>80</td>
<td>$2.7 \cdot 10^{-12}$</td>
<td>11.7</td>
<td>187</td>
<td>$3.7 \cdot 10^{-12}$</td>
</tr>
<tr>
<td></td>
<td>Padé</td>
<td>49.9</td>
<td>74</td>
<td>$4.7 \cdot 10^{-12}$</td>
<td>16.5</td>
<td>295</td>
<td>$1.8 \cdot 10^{-10}$</td>
</tr>
<tr>
<td></td>
<td>Güttel–Nakatsukasa‡</td>
<td>129.8</td>
<td>104</td>
<td>$8.4 \cdot 10^{-9}$</td>
<td>34.6</td>
<td>390</td>
<td>$8.4 \cdot 10^{-9}$</td>
</tr>
<tr>
<td>te12</td>
<td>Higham</td>
<td>55.3</td>
<td>83</td>
<td>$1.2 \cdot 10^{-13}$</td>
<td>18.5</td>
<td>248</td>
<td>$3.9 \cdot 10^{-13}$</td>
</tr>
<tr>
<td></td>
<td>Padé</td>
<td>71.9</td>
<td>94</td>
<td>$4.3 \cdot 10^{-13}$</td>
<td>22.5</td>
<td>399</td>
<td>$1.1 \cdot 10^{-10}$</td>
</tr>
<tr>
<td></td>
<td>Güttel–Nakatsukasa‡</td>
<td>77.6</td>
<td>102</td>
<td>$3.3 \cdot 10^{-9}$</td>
<td>22.4</td>
<td>359</td>
<td>$3.3 \cdot 10^{-9}$</td>
</tr>
<tr>
<td>imclam55</td>
<td>Higham</td>
<td>76.8</td>
<td>92</td>
<td>$5.0 \cdot 10^{-14}$</td>
<td>26.9</td>
<td>262</td>
<td>$5.1 \cdot 10^{-14}$</td>
</tr>
<tr>
<td></td>
<td>Padé</td>
<td>105.8</td>
<td>96</td>
<td>$1.6 \cdot 10^{-13}$</td>
<td>36.0</td>
<td>351</td>
<td>$1.7 \cdot 10^{-12}$</td>
</tr>
<tr>
<td></td>
<td>Güttel–Nakatsukasa‡</td>
<td>264.6</td>
<td>106</td>
<td>$1.6 \cdot 10^{-12}$</td>
<td>72.3</td>
<td>401</td>
<td>$1.4 \cdot 10^{-12}$</td>
</tr>
</tbody>
</table>

the executions, because Matlab’s reference solution implements the same algorithm.

Table 4 and Figure 3 contain the results for the tests on the Kepler platform. The results corresponding to the CPU runs maintain the same ranking as on Fermi, but the increase of computational intensity of Padé is remarkable. It is noticeable how the time ratio between Higham and Padé is reduced on this newer platform. Even working with complex arithmetic, where Güttel–Nakatsukasa can be competitive, it is the slowest method.

The GPU executions obtain the fastest times, but the speedups with respect to the CPU are smaller on this platform. The best speedup of 2.1 is obtained with Padé when working with the largest matrix. Contrary to the CPU results, Padé is always faster than Higham on GPU, benefiting from a higher computational intensity. The higher number of Gflop/s obtained by Padé comes from the smaller relative weight of the _gesv routine with respect to higher number of matrix-matrix multiplications.

‡Without computing the eigenvalues and shifting the matrix
Figure 2: Time results for computing the matrix exponential on Fermi.

Figure 3: Time results for computing the matrix exponential on Kepler.
Table 4: Results for the matrix exponential running on the Kepler platform. Time expressed in seconds. GF/s indicates gigaflops per second.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Algorithm</th>
<th>CPU</th>
<th></th>
<th></th>
<th>GPU</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>GF/s</td>
<td>|F - F_m|_F</td>
<td>Time</td>
<td>GF/s</td>
<td>|F - F_m|_F</td>
<td></td>
</tr>
<tr>
<td>orani678</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Higham</td>
<td>1.7</td>
<td>122</td>
<td>4.1 \times 10^{-12}</td>
<td>2.3</td>
<td>90</td>
<td>4.1 \times 10^{-12}</td>
<td></td>
</tr>
<tr>
<td>Padé</td>
<td>1.8</td>
<td>225</td>
<td>4.1 \times 10^{-12}</td>
<td>1.5</td>
<td>324</td>
<td>6.6 \times 10^{-12}</td>
<td></td>
</tr>
<tr>
<td>Güttel–Nakatsukasa†</td>
<td>4.9</td>
<td>310</td>
<td>3.9 \times 10^{-12}</td>
<td>2.9</td>
<td>505</td>
<td>4.7 \times 10^{-12}</td>
<td></td>
</tr>
<tr>
<td>bcsprw10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Higham</td>
<td>10.3</td>
<td>212</td>
<td>2.6 \times 10^{-12}</td>
<td>7.8</td>
<td>280</td>
<td>4.1 \times 10^{-12}</td>
<td></td>
</tr>
<tr>
<td>Padé</td>
<td>10.9</td>
<td>337</td>
<td>4.9 \times 10^{-12}</td>
<td>6.3</td>
<td>773</td>
<td>1.8 \times 10^{-10}</td>
<td></td>
</tr>
<tr>
<td>Güttel–Nakatsukasa†</td>
<td>30.1</td>
<td>449</td>
<td>8.4 \times 10^{-9}</td>
<td>17.9</td>
<td>753</td>
<td>8.4 \times 10^{-9}</td>
<td></td>
</tr>
<tr>
<td>te12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Higham</td>
<td>15.3</td>
<td>300</td>
<td>1.4 \times 10^{-13}</td>
<td>11.9</td>
<td>386</td>
<td>3.8 \times 10^{-13}</td>
<td></td>
</tr>
<tr>
<td>Padé</td>
<td>16.0</td>
<td>423</td>
<td>3.4 \times 10^{-13}</td>
<td>11.0</td>
<td>818</td>
<td>1.1 \times 10^{-10}</td>
<td></td>
</tr>
<tr>
<td>Güttel–Nakatsukasa†</td>
<td>19.2</td>
<td>410</td>
<td>3.3 \times 10^{-9}</td>
<td>13.7</td>
<td>588</td>
<td>3.3 \times 10^{-9}</td>
<td></td>
</tr>
<tr>
<td>imclam55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Higham</td>
<td>22.9</td>
<td>307</td>
<td>4.3 \times 10^{-14}</td>
<td>13.6</td>
<td>518</td>
<td>5.1 \times 10^{-14}</td>
<td></td>
</tr>
<tr>
<td>Padé</td>
<td>26.1</td>
<td>388</td>
<td>1.3 \times 10^{-13}</td>
<td>12.5</td>
<td>1012</td>
<td>1.7 \times 10^{-12}</td>
<td></td>
</tr>
<tr>
<td>Güttel–Nakatsukasa†</td>
<td>59.2</td>
<td>475</td>
<td>1.6 \times 10^{-12}</td>
<td>31.0</td>
<td>934</td>
<td>1.4 \times 10^{-12}</td>
<td></td>
</tr>
</tbody>
</table>

5. Krylov methods for matrix functions

The evaluation of a dense matrix function, as discussed in the previous section, is one of the steps required in the context of Krylov methods for computing \( y = f(A)b \). In this section, we discuss the specific method that we have considered for this problem, in the context of the SLEPc library.

5.1. Arnoldi with restarts

Krylov methods [1, Ch. 13] are appropriate for the case of large and sparse \( A \), although they are not the only alternative (see, e.g., [3]). They approximate the result vector \( y = f(A)b \) by an element of the Krylov subspace \( \mathcal{K}_n(A,b) = \text{span} \{ b, Ab, A^2b, \ldots, A^{n-1}b \} \), without explicitly building the matrix \( f(A) \). The computation comprises two parts. The first part is to build an orthonormal basis \( V_m \) of the Krylov subspace by means of the Arnoldi method. The computed quantities satisfy the relation \( AV_m = V_mH_m + h_{m+1,m}v_m e_m^T \), where \( H_m \) is an \( m \times m \) upper Hessenberg matrix. The second part is to compute the approximation of \( y \) as

\[
y^{(0)} = \beta V_m f(H_m) e_1,
\]
where $\beta = \|b\|_2$ and $e_1$ is the first coordinate vector. In this way, the problem of computing the function of a large matrix $A$ of order $n$ is reduced to computing the function of a small matrix $H_m$ of order $m$, with $m \ll n$. For the latter task, we can employ the algorithms of section 4.

In an MPI parallel implementation of this method, the main ingredients of the first part are the parallel sparse matrix-vector product and the orthogonalization of the $v_j$ vectors (the columns of $V_m$). These two operations can be implemented efficiently and in a scalable way, provided that $A$ has an appropriate sparse pattern. The second part, the evaluation of $f(H_m)$ as a dense matrix computation, must be done redundantly by all MPI processes. Although this computation is in the critical path, it is usually negligible whenever $m$, the size of the Hessenberg matrix $H_m$, is much smaller than the size of $A$. However, this is not always the case, as we now discuss.

The value of the $m$ parameter is difficult to choose. If $m$ is too small the Krylov subspace will not contain enough information to build an accurate approximation. And if $m$ is too large, the memory requirements for storing $V_m$ (as well as the associated computational cost) will be prohibitive. In a practical implementation, a restarted variant of the method must be used, where $m$ is prescribed to a fixed value and when the subspace reaches this size, a restart is carried out by keeping part of the data computed so far and discarding unnecessary information. We use the Eiermann-Ernst restart [25], in which only the last basis vector $v_{m+1}$ is kept (to continue the Arnoldi recurrence), along with the matrix $H_m$ that is glued together with the previous ones. More precisely, the approximation of $y$ is improved at each restart by an additive correction. After $k$ restarts, the approximation is updated as $y^{(k)} = y^{(k-1)} + c^{(k)}$, where the correction is

$$c^{(k)} = \beta V_m[0, I_m] f(H_{km}) e_1.$$  

(12)

The upper Hessenberg matrix $H_{km}$ is obtained by extending the one from previous restarts,

$$H_{km} = \begin{bmatrix} H_{(k-1)m} & 0 \\ h_{m+1,m}^{(k-1)} e_1 e_T^{(k-1)m} & H_m^{(k)} \end{bmatrix},$$  

(13)

where $H_m^{(k)}$ is the matrix computed by Arnoldi in the $k$th restart. The stopping criterion can be based on the norm of the correction: $||c^{(k)}|| < \beta \cdot \text{tol}$. We use a value $\text{tol} = 10^{-8}$ in the tests of section 5.2.
Note that the matrix of (13) has increasing size \( k \cdot m \), where \( k \) is the number of restarts. In practical applications, this size can often become as high as a few thousands. Note also that this matrix is not symmetric even if \( A \) is symmetric.

5.2. Computational evaluation of sparse solvers

In this section, we present performance results for the Krylov solver using two test cases. The first one, also used in [26], corresponds to the discretization of the advection diffusion equation

\[
\partial_t u = \varepsilon \Delta u + c \nabla u, \tag{14}
\]

on the domain \( \Omega = [0, 1]^2 \) with homogeneous Dirichlet boundary conditions. A standard 5-point finite-difference discretization with grid size \( h = \frac{1}{N+1} \) in both spatial directions results in a sparse matrix of order \( n = N^2 \) with at most five nonzero elements per row. The Pécelt number is defined as the advection diffusion ratio, scaled by \( h \), \( \text{Pe} = \frac{ch^2}{\varepsilon} \), allowing to control the normality of the matrix. In this case, we are interested in computing \( u_k = \exp(\Delta t A)u_{k-1} \) for a few values of \( k \) using a constant time step, \( \Delta t \). The computation can be done with the restarted Arnoldi method described above, together with the algorithms of section 4.3 for the explicit evaluation of \( \exp(H_k m) \) at each restart.

The other test case corresponds to Application 1 of section 3 (data assimilation). Details of the application and the solver can be found in [10], where only the CPU code was used. We now evaluate the GPU version as well. Note that in section 4 we used the same application, but a different matrix, to analyze the square root function, and now we deal with the complete function with a matrix of dimension 17733 and a sparsity of 40%, much denser than in the advection diffusion case. In this application, we are interested in solving linear systems of the form \( (D + \sqrt{D})x = b \), which is equivalent to computing \( x = f(D)b \), with

\[
f(D) = (D + \sqrt{D})^{-1}. \tag{15}
\]

SLEPc allows certain flexibility in the definition of functions, by combining two simpler functions. In our case, we define \( f(\cdot) \) as the reciprocal of another function, which in turn is defined as the sum of two functions (the identity and the square root). All these sub-functions can be evaluated easily, except for the matrix square root that needs the algorithms of section 4.1. These
evaluations will operate on the Hessenberg matrix in every restart of the Arnoldi method.

The sparse tests were conducted on two clusters composed by nodes with the same characteristics as the platforms of section 4.4, interconnected through an Infiniband network. In the experiments of this section, all the executions launch a single process per node, using 12 (Fermi) or 16 (Kepler) threads per process on the CPU runs, and a single GPU card per process on the GPU runs. The reported times are the minimum from three independent executions working in double precision arithmetic. The performance of the different steps of the computation may vary a lot depending on the problem solved and the parameters used. Here we break down the total computation time into the main computational units to analyze their behaviour. Figures 4 and 5, and Tables 5 and 6 contain, for CPU and GPU runs, the following times: the matrix-vector product used to expand the basis of the Krylov subspace, referred to as MatVec; the orthogonalization and normalization of the vectors, referred to as Orthog; and the computation of the projected dense problem, \( f(H_{km}) \).

Figure 4 and Table 5 show the results when computing \( u_k = \exp(\Delta t A)u_{k-1} \) for five repetitions with \( \Delta t = 10^{-4} \). The parameters that we have used to generate the advection diffusion discretization matrix are \( Pe = 0.5, \varepsilon = 1 \), and \( N = 1735 \) (hence the size of \( A \) is roughly 3 million). The initial vector used is the one given by the discretization of the initial state \( u_0(x, y) = x^2(1-x)^2y^2(1-y)^2 \). The Arnoldi restart parameter used in this case is \( m = 30 \), with which the solver needed 23 restarts on each time step.

On section 4.4 we saw that on the Kepler platform, none of the algorithms to compute the dense exponential was faster on both CPU and GPU, so with this problem we present results using Higham (faster on CPU) and Padé (faster on GPU). On Fermi, MatVec, Orthog and \( \exp(H_{km}) \) are able to achieve good speedups of 13.4, 3.5 and 9.5 respectively, with the single process execution on the GPU runs (comparing to CPU). The overall speedup obtained with a single process reaches only 4.3, as the dominant time consuming operation is the orthogonalization, that attains the smallest speedup. As the dense solver is not executed in parallel, its runtime should not vary in the multi-process execution, although on actual runs there can indeed be some variation due to idle times produced by process synchronization. On the multi-process execution, the speedup of MatVec is reduced to 3.8 and the speedup of Orthog improves up to 4.8. These speedups, together with the reduced relative weight of the orthogonalization when executing with mul-
Table 5: Results for the advection diffusion problem. Time expressed in seconds. The MatVec and Orthog columns show the time needed in the matrix-vector product, and in the orthogonalization and normalization of the basis vectors, respectively.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Processes</th>
<th>Algorithm</th>
<th>CPU Time</th>
<th>GPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>MatVec</td>
<td>Orthog</td>
</tr>
<tr>
<td>Fermi</td>
<td>1</td>
<td>Higham</td>
<td>136.9</td>
<td>554.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Padé</td>
<td>135.0</td>
<td>557.6</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>Higham</td>
<td>23.9</td>
<td>78.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Padé</td>
<td>19.3</td>
<td>71.9</td>
</tr>
<tr>
<td>Kepler</td>
<td>1</td>
<td>Higham</td>
<td>103.7</td>
<td>473.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Padé</td>
<td>112.8</td>
<td>473.1</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>Higham</td>
<td>8.7</td>
<td>19.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Padé</td>
<td>8.1</td>
<td>19.1</td>
</tr>
</tbody>
</table>

Multiple processes, increase the maximum speedup obtained with the GPU for the whole computation up to 5.8 when using 16 processes.

On Kepler, the single process execution on GPU obtains speedups of 13.9 and 6.1 for MatVec and Orthog, with respect to the CPU runs. The dense exponential performance is reduced to a speedup of 1.8 when running on GPU. The whole computation speedup for GPU runs compared with the CPU ones, achieve 6.6 with a single process, and a more limited speedup of 2.1 on the multi-process execution.

Figure 5 and Table 6 show the results obtained for computing $f(D)b$ for 30 different right-hand sides $b$ with the EnSRF matrix function of (15), and using Denman–Beavers (3) as the method to compute $\sqrt{H_{km}}$. In this experiment we have used an Arnoldi restart parameter of $m = 150$, with which the solver required only three restarts in all cases.

On Fermi, the single process execution running on GPU achieves speedups of 16.2 and 2.7 for MatVec and Orthog, with respect to the CPU, while the dense function obtains almost the same time on CPU and GPU. The overall obtained speedup of 12.7 when running on GPU stems from the MatVec operation. The multi-process execution reduces the speedup of MatVec to 9.2, and Orthog speedup increases to 6.8, reducing the total speedup down to 4.8 for the GPU with respect to the CPU run. The runs on GPU on the Kepler platform, with a single process, obtain better speedup in MatVec,
the dominant operation, reaching 18.1. This improvement of MatVec helps to accelerate the whole computation more than in the case of the Fermi platform, with a speedup of 15.2 when running a single process on GPU, with respect to CPU. Using several processes on Kepler also improves the gain obtained on Fermi, reaching a speedup of 5 on the whole computation with respect to the CPU runs.

6. Conclusions

We have studied the suitability of several dense methods to compute the matrix square root and the matrix exponential functions on GPU platforms. The simplicity of iterative methods entitle them to be implemented on GPU and allow the acceleration of its computation. In our tests, we have been able to obtain up to a gain factor of 3.8 when comparing GPU implementations against multi-threaded CPU implementations, for matrices with dimension of a few thousands.

The obtained acceleration can be useful to reduce the cost present in the critical path of parallel Krylov methods for the computation of \( f(A)b \), when executed on clusters of GPUs, provided that the size of the Hessenberg
matrix is not too small, that is, the computation associated with this matrix takes a non-negligible percentage of the overall computation time. The GPU implementations of the dense methods will also be helpful for explicitly computing $f(A)$ in some applications.

We have also analyzed the overall performance on GPU platforms of the restarted Arnoldi method for matrix functions, obtaining speedups of 5.8 in a multi-process execution.

In the future, we will continue adding more functionality to the matrix function solvers in SLEPc, both in the dense and sparse cases. In particular, alternative restart schemes have been proposed recently that may be more effective than the Eiermann-Ernst scheme presented here.

Acknowledgements

The executions corresponding to sections 4.4 and 5.2 were carried out on the supercomputer Minotauro, belonging to the Spanish Supercomputing Network (RES). The authors would like to thank Jeff Steward for his support in relation to test problems from application 1.

References

Figure 5: Time results for the EnSRF matrix function using Denman–Beavers to compute $\sqrt{\Pi_{km}}$.


