Journal of Computational and Applied Mathematics 309 (2017) 325–332

Contents lists available at ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

Efficient and accurate algorithms for computing matrix trigonometric functions☆

Pedro Alonso [a], Javier Ibáñez [b], Jorge Sastre [c], Jesús Peinado [b,*], Emilio Defez [d]

[a] Department of Information Systems and Computation, Universitat Politècnica de València, Camino de Vera s/n, 46022, Valencia, Spain

[b] Instituto de Instrumentación para Imagen Molecular, Universitat Politècnica de València, Camino de Vera s/n, 46022, Valencia, Spain

[c] Instituto de Telecomunicaciones y Aplicaciones Multimedia, Universitat Politècnica de València, Camino de Vera s/n, 46022, Valencia, Spain

[d] Instituto de Matemática Multidisciplinar, Universitat Politècnica de València, Camino de Vera s/n, 46022, Valencia, Spain

ARTICLE INFO

Article history:
Received 27 November 2015

Keywords:
Matrix cosine
Matrix sine
Scaling and squaring method
Taylor series
Backward error
Parallel implementation

ABSTRACT

Trigonometric matrix functions play a fundamental role in second order differential equations. This work presents an algorithm based on Taylor series for computing the matrix cosine. It uses a backward error analysis with improved bounds. Numerical experiments show that MATLAB implementations of this algorithm has higher accuracy than other MATLAB implementations of the state of the art in the majority of tests. Furthermore, we have implemented the designed algorithm in language C for general purpose processors, and in CUDA for one and two NVIDIA GPUs. We obtained a very good performance from these implementations thanks to the high computational power of these hardware accelerators and our effort driven to avoid as much communications as possible. All the implemented programs are accessible through the MATLAB environment.

1. Introduction

Many engineering processes are described by second order differential equations, whose exact solution is given in terms of trigonometric matrix functions sine and cosine. For example, the wave problem

$$v^2 \frac{\partial^2 \psi}{\partial x^2} = \frac{\partial^2 \psi}{\partial t^2}, \tag{1}$$

plays an important role in many areas of engineering and applied sciences. If the spatially semi-discretization method is used to solve (1), we obtain the matrix differential problem

$$X''(t) + AX(t) = 0, \qquad X(0) = X_0, \qquad X'(0) = X_1, \tag{2}$$

where A is a square matrix and X_0 and X_1 are vectors. The solution of (2) is

$$X(t) = \cos\left(\sqrt{A}t\right) X_0 + \left(\sqrt{A}\right)^{-1} \sin\left(\sqrt{A}t\right) X_1, \tag{3}$$

☆ This work has been supported by Spanish Ministerio de Economía y Competitividad and European Regional Development Fund (ERDF) grant TIN2014-59294-P.

* Corresponding author.
E-mail addresses: palonso@dsic.upv.es (P. Alonso), jjibanez@dsic.upv.es (J. Ibáñez), jorsasma@iteam.upv.es (J. Sastre), jpeinado@dsic.upv.es (J. Peinado), edefez@imm.upv.es (E. Defez).

http://dx.doi.org/10.1016/j.cam.2016.05.015
0377-0427/

where $\sqrt{A}$ denotes any square root of a non-singular matrix A [1, p. 36]. More general problems of type (2), with a forcing term $F(t)$ on the right-hand side arise from mechanical systems without damping, and their solutions can be expressed in terms of integrals involving the matrix sine and cosine [2].

Numerous methods have been proposed for computing $f(A)$, where $f(.)$ is a scalar function defined on the spectrum of the matrix $A \in \mathbb{C}^{n\times n}$. Many of them have a dubious numerical stability [3]. A complete theoretical study of matrix functions and their computational methods and algorithms can be found in [1], in particular the computation of matrix trigonometric functions. The main methods are based on matrix decompositions and on polynomial and rational approximations. Since polynomial and rational approximations are accurate only near the origin, scaling and recovering techniques [4–6] are usually used. Moreover, to reduce computational costs Paterson–Stockmeyer method [7] is used for evaluating the polynomials which appear in these approximations.

In this work we present sequential and parallel algorithms based on Taylor series that use Theorem 1 from [8] for computing matrix trigonometric functions.

Throughout this paper $\mathbb{C}^{n\times n}$ denotes the set of complex matrices of size $n \times n$, I the identity matrix for this set, $\rho(X)$ the spectral radius of matrix X, and $\mathbb{N}$ the set of positive integers. In this paper we use the 1-norm to compute the actual norms. Sections 2 and 3 present sequential and parallel Taylor algorithms for computing matrix trigonometric functions, respectively. Section 4 deals with numerical tests and finally in Section 5 the conclusions are presented.

2. Sequential algorithms for computing matrix cosine and sine

The matrix cosine can be defined for all $A \in \mathbb{C}^{n\times n}$ by

$$\cos(A) = \sum_{i=0}^{\infty} \frac{(-1)^i A^{2i}}{(2i)!},$$

and let

$$T_{2m}(A) = \sum_{i=0}^{m} \frac{(-1)^i B^i}{(2i)!} \equiv P_m(B), \tag{4}$$

be the Taylor approximation of order $2m$ of $\cos(A)$, where $B = A^2$. Since Taylor series are accurate only near the origin, in algorithms that use this approximation the norm of matrix B is reduced by scaling the matrix. Then, a Taylor or Padé approximation is computed, and finally the approximation of $\cos(A)$ is recovered by means of the double angle formula $\cos(2X) = 2\cos^2(X) - I$.

Using the same notation as in [5], we have that Taylor matrix polynomial approximation (4), expressed as $P_m(B) = \sum_{i=0}^{m} p_i B^i$, $B \in \mathbb{C}^{n\times n}$, can be computed with optimal cost by Paterson–Stockmeyer's method [7] choosing m from the set

$$\mathbb{M} = \{1, 2, 4, 6, 9, 12, 16, 20, 25, 30, 36, 42, \ldots\}, \tag{5}$$

where the elements of $\mathbb{M}$ are denoted as $m_1, m_2, m_3, \ldots$ (see [1, pp. 72–74] for a complete description). The algorithm computes firstly the matrix powers $B^2, B^3, \ldots, B^q$ being $q = \lceil\sqrt{m_k}\rceil$ or $q = \lfloor\sqrt{m_k}\rfloor$, and integer divisor of m_k. As stated in [1, p. 74] using those values for q results in the same cost. Thus, the evaluation formula (23) from [9, p. 6455] is computed as

$$\begin{aligned} P_{m_k}(B) = {} & (((p_{m_k}B^q + p_{m_k-1}B^{q-1} + p_{m_k-2}B^{q-2} + \cdots + p_{m_k-q+1}B + p_{m_k-q}I)\cdot B^q \\ & + p_{m_k-q-1}B^{q-1} + p_{m_k-q-2}B^{q-2} + \cdots + p_{m_k-2q+1}B + p_{m_k-2q}I)\cdot B^q \\ & + p_{m_k-2q-1}B^{q-1} + p_{m_k-2q-2}B^{q-2} + \cdots + p_{m_k-3q+1}B + p_{m_k-3q}I)\cdot B^q \\ & \ldots \\ & + p_{q-1}B^{q-1} + p_{q-2}B^{q-2} + \cdots + p_1 B + p_0 I. \end{aligned} \tag{6}$$

We define the boxing size as the largest polynomial degree which appear in (6), i.e. the value q. Table 1 shows the values of q for different values of m. Taking into account Table 4.1 from [1, p. 74], then the cost of evaluating (4) with (6) in terms of matrix products, denoted by Π_{m_k}, for $k = 1, 2, \ldots,$ is

$$\Pi_{m_k} = k. \tag{7}$$

The difficulty of the algorithms based on Taylor series is to find appropriate values m_k and the scaling factor s such that $\cos(A)$ is computed accurately and with minimal computational cost.

Next theorem will be used to bound the norm of the matrix Taylor series.

Theorem 1 ([5]). *Let $h_l(x) = \sum_{i=l}^{\infty} p_i x^i$ be a power series with radius of convergence w, $\tilde{h}_l(x) = \sum_{i=l}^{\infty} |p_i| x^i$, $B \in \mathbb{C}^{n\times n}$ with $\rho(B) < w$, $l \in \mathbb{N}$ and $t \in \mathbb{N}$ with $1 \leqslant t \leqslant l$. If t_0 is the multiple of t such that $l \leqslant t_0 \leqslant l + t - 1$ and*

$$\beta_t = \max\{b_j^{1/j} : j = t, l, l+1, \ldots, t_0 - 1, t_0 + 1, t_0 + 2, \ldots, l + t - 1\}, \tag{8}$$

Table 1
New values of Θ_{m_k} for [5, Table 2] and matrix powers $B^2, B^3, \ldots, B^{q_k}$ used to compute (4) by Paterson–Stockmeyer method.

k	m_k	q_k	Θ_{m_k}	k	m_k	q_k	Θ_{m_k}
1	1	1	6.661338018806219e−16	5	9	3	1.189983654063290
2	2	2	1.154075612730971e−07	6	12	4	4.924177884630485
3	4	2	2.491236564385514e−03	7	16	4	16.06054585896760
4	6	3	8.976968236812591e−02	8	20	5	35.62660483639449

where b_j is an upper bound for $\|B^j\|$, $\|B^j\| \leqslant b_j$, then

$$\|h_l(B)\| \leqslant \tilde{h}_l(\beta_t). \tag{9}$$

The error analysis for Taylor approximation of the matrix cosine, similar to that on Sec. 2.2 of [6] for Padé approximation, yields analogous results making it very restrictive. Instead of that we use an analysis based on the backward error of the matrix exponential computation by Taylor algorithm from [8], similar to that for Padé Sec. 2.3 of [6], overcoming the difficulty that the backward error for Taylor approximation is not odd as that for Padé.

If we denote $\bar{T}_{2m}(X) = \sum_{i=0}^{2m} \frac{X^i}{i!}$, $X \in \mathbb{C}^{n\times n}$, the truncated Taylor series of order $2m$ of e^X and $\bar{R}_{2m}(X) = e^x - \bar{T}_{2m}(X)$ its remainder, by [8, Sec. 2], if $\rho\left(e^{-X}\bar{T}_{2m}(X) - I\right) < 1$, then taking $s = 0$ in (3) from [8], by (4) and (7) also from [8], it follows that

$$\bar{T}_{2m}(X) = e^{X+\Delta X},$$

where the backward error of the matrix exponential Taylor approximation is

$$\Delta X = h_{2m+1}(X) = \log\left(e^{-X}\bar{T}_{2m}(X)\right) = \sum_{k\geq 2m+1} c_k^{(2m)} X^k, \tag{10}$$

where log is the principal logarithm and the coefficients $c_k^{(2m)}$ depend on the order $2m$. Since [1, Eq. 12.2]

$$\cos(X) = \frac{e^{iX} + e^{-iX}}{2}, \tag{11}$$

we can compute approximately $\cos(A)$ as

$$\begin{aligned} T_{2m}(A) &= (\bar{T}_{2m}(iA) + \bar{T}_{2m}(-iA))/2 = \left(e^{iA+h_{2m+1}(iA)} + e^{-iA+h_{2m+1}(-iA)}\right)/2 \\ &\neq \left(e^{i\left(A+h_{2m+1}(A)\right)} + e^{-i\left(A+h_{2m+1}(A)\right)}\right)/2, \end{aligned} \tag{12}$$

because calculating symbolically some terms of h_{2m+1} it follows that it is not odd for the different m in (5) and $h_{2m+1}(-iA) \neq -ih_{2m+1}(A)$. However, from (10) the bound

$$\|h_{2m+1}(\pm iA)\| \leq \sum_{k\geq 2m+1} |c_k^{(2m)}|\, \|A\|^k \tag{13}$$

holds for both $\|h_{2m+1}(iA)\|$ and $\|h_{2m+1}(-iA)\|$, and then, similarly to [8, Sec. 2], using Theorem 1 it follows that if $\beta_t \leq \Theta_m$, where Θ_m are given in Table 1 for some values of $m = m_k$ from (5), then

$$\frac{\|h_{2m+1}(\pm iA)\|}{\|\pm iA\|} \leq u.$$

These new values of parameter Θ_{m_k} substitute the values from [5, Table 2]. This does not guarantee that the backward error for the Taylor approximation of the matrix cosine is less than u. Note that we do not compute the matrix cosine by the computation of both Taylor polynomials in (12), and therefore there is no cancellation problems because of that. Hence, if the backward error of the two exponentials is small and then the approximation of both exponentials is accurate, the error for the matrix cosine computation will be also small, and experimental results support that (see Section 4).

Let m_k be the maximum order allowed. If $\|B\| \leqslant \Theta_{m_{k_0}}$ for some $m_{k_0} \leqslant m_k$, where B is the matrix from (4) and m_k are defined by Table 1, then the scaling of matrix B is not necessary and the order m_{k_0} is selected. Otherwise we scale the matrix B by selecting a positive integer s such that $4^{-s}\beta_{m_k} \leq \Theta_{m_k}$, where β_{m_k} can be obtained from Theorem 1 as

$$\beta_{m_k} = \max\left\{\left\|B^k\right\|^{\frac{1}{k}} :\ k \geq m_k\right\} = \max\left\{\left\|B^k\right\|^{\frac{1}{k}} :\ m_k \leq k \leq 2m_k - 1\right\}.$$

Analogously to [10, p. 374], we take for β_{m_k} the approximation

$$\bar{\beta}_{m_k} = \max\left\{\left\|B^{m_k}\right\|^{\frac{1}{m_k}}, \left\|B^{m_k+1}\right\|^{\frac{1}{m_k+1}}\right\}, \tag{14}$$

and then $s = \left\lceil \frac{1}{2}\log_2\left(\frac{\bar{\beta}_{m_k}}{\Theta_{m_k}}\right)\right\rceil$. Approximation (14) is justified since $\left\|B^k\right\|^{\frac{1}{k}} \to \rho(B)$ as $k \to \infty$, and then for the majority of matrices the values $\left\|B^k\right\|^{\frac{1}{k}}$ tend to be decreasing and tend to have less variations for higher matrix powers. Algorithm 1 computes the matrix cosine based on the ideas above with the objective of simplicity for a parallel implementation.

Algorithm 1 Given a matrix $A \in \mathbb{C}^{n\times n}$ and m_k, this algorithm computes $C = \cos(A)$ by scaling and using (4) with m lower than or equal to m_k.

1: $B_1 = A^2$
2: **if** $||B|| \leqslant \Theta_{m_k}$ **then** ▷ (see Table 1 for the values Θ_{m_k})
3: Compute the first positive integer m_{k_0} such that $\|B_1\| \leq \Theta_{m_{k_0}}$
4: Compute the matrices $B_2 = B_1^2, B_3 = B_2B_1, \ldots, B_{q_{k_0}} = B_{q_{k_0}-1}B_1$ (see Table 1 for the values q_k)
5: Compute $C = P_{m_{k_0}}(B)$ from (4) by using Paterson–Stockmeyer method (6) and the matrices B_i (see Step 4)
6: **else**
7: Compute the matrices $B_2 = B_1^2, B_3 = B_2B_1, \ldots, B_{q_k} = B_{q_k-1}B_1$
8: Compute $\bar{\beta}_{m_k}$ from (14)
9: $s = \left\lceil \frac{1}{2}\log_2\left(\frac{\bar{\beta}_{m_k}}{\Theta_{m_k}}\right)\right\rceil$
10: **for** $i = 1 : q$ **do**
11: $B_i = 4^{-is}B_i$
12: **end for**
13: Compute $C = P_{m_k}(B)$ from (4) by using Paterson–Stockmeyer method (6) and the scaled matrices B_i (see Steps 10–12)
14: **for** $i = 1 : s$ **do**
15: $C = 2C^2 - I$
16: **end for**
17: **end if**

By using the fact that $\sin(A) = \cos(A - \frac{\pi}{2}I)$, Algorithm 1 can be easily used to compute the matrix sine. The computational cost of Algorithm 1 is $2k_0n^3$ flops provided $\|B\| \leqslant \Theta_{m_k}$, or $2(k+s)n^3$ flops if $\|B\| > \Theta_{m_k}$. The storage cost is $(2+q_{k0})n^2$ if $\|B\| \leqslant \Theta_{m_k}$, and $(2+q_k)n^2$ otherwise.

3. The implementation of the parallel algorithms

Algorithm 1 has also been implemented in C language such that is called using the MATLAB interface through the corresponding "mex" file [11]. For the most demanding computational kernel, which is the matrix multiplication appeared in different places of the algorithm, we used the Intel Math Kernel Library (MKL) version 11.0.2.

Algorithm 1 was also implemented in CUDA [12] to exploit the high computational capabilities of NVIDIA GPUs (Graphics Processing Units). We implemented two versions, for one or two GPUs, since currently it is very common to find workstations with up two GPUs attached and available to accelerate computations. The computation of matrix powers in Steps 4 and 7, and the matrix multiplications in Steps 5, 13 and 15 on only one NVIDIA GPU is quite straightforward thanks to the CUBLAS library [13], a library that contains an implementation of the BLAS [14] routines for NVIDIA GPUs. In particular, the matrix multiplication is very optimized for this device, and it allows to improve the performance of Algorithm 1. However, the computations on several GPUs are usually penalized by the fact of being attached to the node through a PCIe connector. The need of transferring data through this bus adds an overhead that must be accounted for in order to obtain a good implementation. Given that, we implemented our algorithm so that once a matrix is transferred to the device or it has been computed there, it must be kept into the device memory as long as possible to avoid data thrashing between host and device memory.

The case with two GPUs is also conceptually easy to solve but difficult to implement. The matrix multiplications in two GPUs are basically carried out by splitting the matrices into two halves, each one stored into a different GPU device. Algorithm 2 describes the Steps 4 and 7 of Algorithm 1 in two GPUs. In fact, the input variable q can be q_{k_0} (Step 4) or q_k (Step 7) depending of whether the scaling is applied or not.

First of all, we notice that the algorithm is described as an OpenMP [15] "parallel loop". This loop has only two iterations. Since the loop has been parallelized through the suitable OpenMP directive, both iterations will be performed concurrently, i.e. iteration g will be executed by GPU g (GPUs are numbered with 0 and 1). This is an implementation detail that allows both, easiness in the actual CUDA implementation and clarity in the exposition. Thus, we write only once the code that will be executed by both devices, parametrized by the different values of g and its derived variables.

The algorithm receives hB as input, which is $hB = B_1 = A^2$ (Step 1 of Algorithm 1), where letter h preceding the matrix name denotes that is stored into the host memory. Each GPU hosts the following three objects upon return: the whole matrix B_1, an array $\mathbf{X}$ of matrices,

$$\mathbf{X} = [X^i]_{i=1,\ldots,q-1} = \begin{pmatrix} X^1 & X^2 & X^3 & \cdots & X^{q-2} & X^{q-1} \end{pmatrix},$$

Algorithm 2 Algorithm for computing the q matrix powers of B_1 with two GPUs.

```
1: function COMPUTE_POWERS2( n, q, hB ) return (B_1, X, B_q)
2:     i_0 = 0, j_0 = n/2 − 1, i_1 = n/2, j_1 = n − 1
3:     #pragma omp parallel for
4:     for g ← 0, 1 do
5:         B_1 ⇐ hB
6:         X(1) ← B_1(:, i_g : j_g)
7:         for k ← 2, q − 1 do
8:             X(k) ← B_1 · X(k − 1)
9:         end for
10:        B_q(:, i_g : j_g) ← B_1 · X(q − 1)
11:        B_q(:, i_ḡ : j_ḡ) ⇐ B̄_q(:, i_ḡ : j_ḡ)
12:    end for
13: end function
```

Algorithm 3 Recursive algorithm to evaluate a matrix polynomial using (6) in two GPUs.

```
1: function PS( d, q, i, p̄, X, B_q ) return P
2:     #pragma omp parallel for
3:     for g ← 0, 1 do
4:         if d = i then
5:             P ← EVAL( q, p̄_{d:d−q}, X )
6:         else
7:             Q_1 ← PS( d, q, i + q, p̄_{d:i}, X, B_q )
8:             Q_2 ← EVAL( q − 1, p̄_{i−1:i−q}, X )
9:             P ← Q_1 · B_q + Q_2
10:        end if
11:    end for
12: end function
```

and matrix B_q of order n that stores the whole factor B^q ($B_q = B^q$). Each component of the array of matrices $\mathbf{X}$ is a matrix of size $n \times n/2$. (For simplicity and without loss of generality, we assume in this discussion that n is even.) We use the superscript as the array index in the components of $\mathbf{X}$ to denote the correspondence between the component and the power of B_1, since each GPU stores the first and second half, respectively, of the n columns of each power of B_1. In other words, $B_1^i = \begin{bmatrix} X_0^i & X_1^i \end{bmatrix}$, for $i = 1, \ldots, q-1$, where the subscript denotes the GPU id (g). On one hand, it is clear that, in order to form the array $\mathbf{X}$ (Step 8), it is necessary that the two GPUs have the complete matrix B_1. This matrix is uploaded to the GPU in Step 5, and the operation is denoted with $\Leftarrow$. On the other hand, both GPUs need the whole matrix B^q so that the evaluation of the polynomial by means of the Paterson–Stockmeyer method (6) can be carried out concurrently by the two devices. Each GPU computes its correspondent half of B^q (Step 10) and stores it in submatrix $B_q(:, i_g : j_g)$, being index i_g and j_g the first and last columns, respectively, where the resulting matrix will be stored in GPU g. Finally, Step 11 denotes the data exchange necessary to complete the construction of matrix B^q in both GPUs, since each device calculates only one half of this factor. To understand this movement of data, it must be taken into account that $\bar{g}$ is 0 if $g = 1$, or 1 otherwise. In the same way, we used $\bar{B}_q$ to denote the half of factor B^q owned by the "other" device. The transference from one GPU to the other one is also denoted here by symbol $\Leftarrow$. We used specific CUDA routines which allow to interchange data between the two GPUs through the PCIe bus without host intervention, thus avoiding to temporarily store data into the host memory, and also exploiting the bidirectionality feature of the PCIe bus, i.e. both halves of factor B^q travel concurrently by the bus in opposite directions to their corresponding destination GPU.[1]

Algorithm 3 is used to evaluate polynomials $P_{m_{k_0}}$ (or P_{m_k}) in Step 5 (or 13) of Algorithm 1, that is, to evaluate (6) provided the powers $i = 2, \ldots, q$ of matrix B_1 have already been computed. It is not hard to see that this polynomial evaluation can be carried out in both GPUs at the same time, keeping powered matrices split into the two GPUs. The matrix objects: Q_1, Q_2, P, and the components of the matrix array $\mathbf{X}$ are all half-matrices, and factor B_q is n square. Obviously, the resulting matrix will be also split into the two devices and would be necessary to upload it to the host to form the whole resulting matrix C (Algorithm 1). The algorithm receives the degree of the polynomial as input (d), which will be m_{k_0} (or m_k) when called from Algorithm 1. Also, the algorithm receives q as defined in (6). The third argument, i, is an integer that controls the recursion. At the first call to function PS this parameter must be the same as q. The next argument is the array of coefficients used in (6), i.e. $\bar{p} = \begin{pmatrix} p_{m_k}, & \ldots, & p_0 \end{pmatrix}$. For simplicity, we have tailored the polynomial evaluation for the particular values of m_k and q_k used in Table 1, and it does not work for other different values. The generalization to the solution of any other matrix polynomial using the Paterson–Stockmeyer method is not hard but requires to rewrite Algorithm 3. Function EVAL in

[1] Peer-to-peer communication routine `cudaMemcpyPeerAsync`.

Step 8 represents the evaluation of a matrix polynomial of degree $q-1$ using the coefficients $\bar{p}_{i-1:i-q}$. We omit an explicitly exposition of this algorithm since it is quite straightforward. The evaluation in Step 5 is analogous, taking into account that the polynomial to evaluate has degree q and coefficients $\bar{p}_{d:d-q}$. For the evaluation of the largest coefficient in this last case, it is used the corresponding half of factor B_q.

Other operations like the scaling in Step 11 of Algorithm 1 are carried out inside the GPU to minimize data communications between host and GPU. Following the same strategy used in Algorithm 2, the loop in Steps 14–16 is also carried out by the GPUs.

4. Numerical experiments

4.1. Sequential tests

In this section we compare `costaym`, a MATLAB implementation of Algorithm 1, with `costay`, based on Taylor approximation [5] (http://personales.upv.es/~jorsasma/software/costay.m), and `cosm`, based on Padé approximants [6, Alg. 4.2] (http://github.com/sdrelton/cosm_sinm). In tests we used MATLAB(R2014b) running on an Intel Core 2 Duo processor at 3.00 GHz with 4 GB main memory and 105 matrices: 10 diagonalizable 128×128 matrices, with 1-norms increasing from 2.50 to 25.06. 10 non diagonalizable Jordan block 128×128 matrices with eigenvalues whose algebraic multiplicity vary between 1 and 128 and 1-norms varying from 5.27 to 21.97. Forty three 128×128 matrices from the function `matrix` of the Matrix Computation Toolbox [16]. 14 matrices with dimensions lower or equal to 128 from the Eigtool MATLAB package [17], and 28 matrices from the matrix function literature. The "exact" matrix cosine was computed exactly for the first two sets of matrices. In the other matrices we used MATLAB symbolic versions of a scaled Padé rational approximation from [6] and a scaled Taylor Paterson–Stockmeyer approximation (6) both with 4096 decimal digit arithmetic and several orders m and/or scaling parameters s higher than the ones used by `cosm` and `costaym`, respectively. The relative differences between both Padé and Taylor approximations for these matrices were between $1.36\ 10^{-23}$ and $5.52\ 10^{-25}$, except for two matrices where the relative differences were $7.29\ 10^{-10}$ and $3.08\ 10^{-07}$, but both `costaym` and `cosm` gave relative errors with respect to these "exact" values of orders between 10^{27} and 10^{43}, so there was no point in increasing the accuracy of these two "exact" values. The algorithm accuracy was tested by computing the relative error

$$\mathrm{E} = \frac{\|\cos(A) - \tilde{Y}\|_1}{\|\cos(A)\|_1},$$

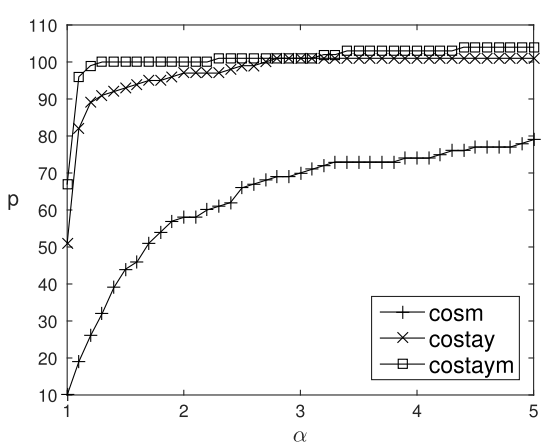
where $\tilde{Y}$ is the computed solution and $\cos(A)$ is the exact solution. To compare the relative errors of the functions we plotted the performance profile and the ratio of relative errors of `cosm`, `costay` and `costaym` (with $m_k = 16$). In the performance profile (Fig. 1(a)), the α coordinate varies between 1 and 5 in steps equal to 0.1, and the p coordinate is the probability that the considered algorithm has a relative error lower than or equal to α-times the smallest error over all the methods. Fig. 1(b) shows the ratio of relative errors of `cosm` and `costay` both with `costaym`, in decreasing order of the ratio with `cosm` (the same order was used in Fig. 2). The Matrices 1 to 5 do not appear in Fig. 1(b) since `costaym` error for them is 0 in double precision arithmetic. Fig. 1 shows that `costaym` was the most accurate function in tests, followed by `costay`. Figs. 2(a) and (b) show the relative number of flops and the relative execution times (mean of 100 executions). The best results in terms of computational cost was `costay` (flop ratio) and `costaym` (execution time ratio). This discrepancy may be due because `costaym` uses far fewer times one estimator based on [18] for computing the 1-norm of matrix powers (computational cost $O(n^2)$) than the one used by `costay` and the size of the matrices is not so large so that the matrix products ($O(n^3)$) be the main term in cost. We have found that for the Toolbox matrices of dimension 1000 the execution time of costaym is slightly greater than the one of costay.

4.2. Parallel tests

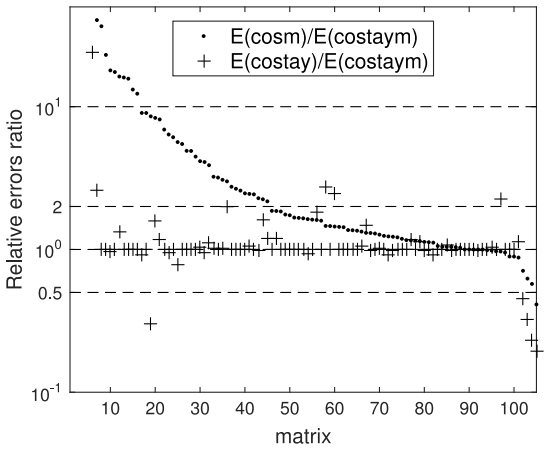
In the next experiments we used a host computer equipped with an Intel QuadCore i7-3820 (3.6 GHz) processor. Attached to this host there are two NVIDIA GPU devices K20 (Kepler) generation card. Each GPU has 13 multiprocessors with 192 cores each, resulting in a total of 2496 CUDA cores. Therefore, we have a total of 4992 CUDA cores for processing. The device memory of one GPU is 4800 MBytes. In order to evaluate the performance of the parallel implementations, we computed the matrix cosine for randomly generated matrices ranging from 1024 to 7168 in steps of 1024 (7168 is the maximum size allowed in one of our GPUs).

All the results shown in Table 2 where obtained using a MATLAB interface to the algorithms implemented in C or CUDA through the corresponding "mex" files. The CPU version uses Intel MKL and the four cores since this package contains BLAS routines threaded, so we can assume that this is the fastest version of the algorithm for the CPU. Clearly, the GPU versions outperform the CPU one as it can be seen in columns 5 and 6 of the table, where it is depicted the speed up achieved with 1 and 2 GPUs w.r.t. to the CPU.

Fig. 3 shows the performance comparison, measured in Gflops, using the CPU, 1 GPU, and 2 GPUs. The performance of the implementation using two GPUs is not completely efficient since some data must be transferred between both the GPUs through the PCIe communication channel. That explains that using 2 GPUs we do not achieve two times the performance

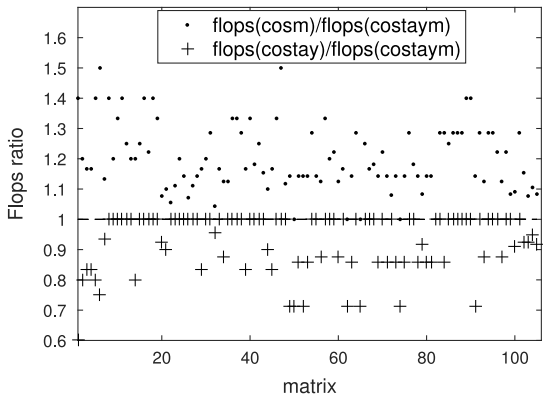


(a) Performance profile.

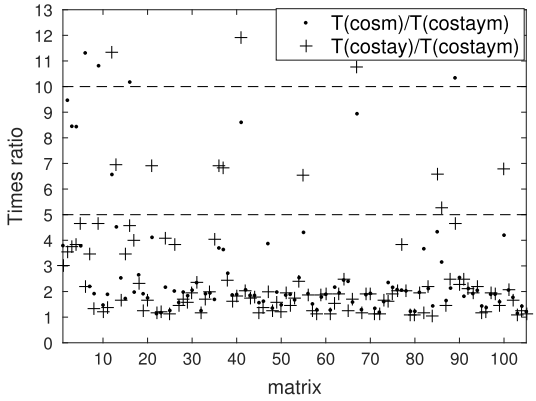


(b) Ratio of relative errors.

Fig. 1. Accuracy tests.



(a) Ratio of flops.



(b) Ratio of execution times.

Fig. 2. Computational cost tests.

Table 2
Table comparing the execution time (in seconds) and speed up of CPU, 1 GPU, and 2 GPUs.

Size	t. CPU	t. 1 GPU	t. 2 GPU	Sp. 1 GPU/CPU	Sp. 2 GPU/CPU
1024	0.79	0.05	0.07	13.66	11.08
2048	6.09	0.37	0.33	16.12	17.95
3072	21.04	1.25	1.08	16.76	19.44
4096	49.30	2.86	2.35	17.24	20.89
5120	95.57	5.50	4.39	17.35	21.75
6144	164.15	9.37	7.37	17.50	22.27
7168	164.70	14.76	11.48	11.15	14.34

with 1 GPU. The communication overhead incurred when using two GPUs with small sizes of the matrix (lower than $\approx$1600) is very large and completely masks the gain in the computation with two GPUs. Yet for larger sizes, we obtain the solution in less time with two GPUs than with only one thanks to the improvements incorporated into the implementation, and which allow taking advantage of the two-way capability of the PCIe to interchange data.

5. Conclusions

In this work accurate Taylor algorithms have been proposed to compute matrix cosine and matrix sine. The algorithms use the scaling technique based on the double angle formula of the cosine function, the Paterson–Stockmeyer's method for computing the Taylor approximation, and bounds obtained from the relative backward error of the matrix exponential Taylor approximation, which allow to calculate the optimal scaling factor. A MATLAB implementation of cosine algorithm

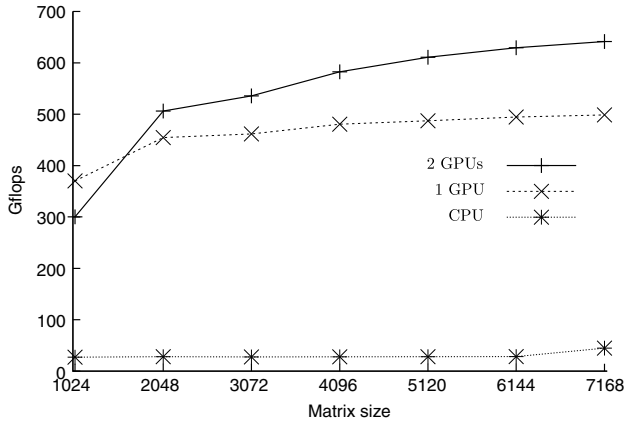


Fig. 3. Performance of Algorithm 1 measured in Gflops on the CPU, 1 GPU, and 2 GPUs.

(`costaym`) has been compared with other state-of-the-art MATLAB implementations. Numerical experiments show that in general the implementations developed have higher accuracy than the other functions in the majority of tests.

The most demanding computational kernel in the Taylor algorithms is by far the matrix multiplication. There exist high performance libraries that get the best of the most up to today computing resources, either general purpose processors or graphics accelerators. It is possible to access this high performance just using the regular matrix multiplication of MATLAB. Even using the MATLAB Parallel Computing Toolbox it is possible to exploit one GPU under the very friendly MATLAB interface. However, from the performance point of view, it is better to have the whole program that computes the cosine of a matrix in a mex file, implemented in C language to use the CPU processor or implemented in CUDA to use the GPU. Furthermore, the use of two GPUs to solve the problem in an efficient way (e.g. minimizing data transference through the PCI bus) must be done through a dedicated implementation in OpenMP+CUDA+mex for the case of the NVIDIA devices.

References

[1] N.J. Higham, Functions of Matrices: Theory and Computation, SIAM, Philadelphia, PA, USA, 2008.
[2] S. Serbin, Rational approximations of trigonometric matrices with application to second-order systems of differential equations, Appl. Math. Comput. 5 (1) (1979) 75–92.
[3] C.B. Moler, C.V. Loan, Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later*, SIAM Rev. 45 (2003) 3–49.
[4] S.M. Serbin, S.A. Blalock, An algorithm for computing the matrix cosine, IAM J. Sci. Statist. Comput. 1 (2) (1980) 198–204.
[5] J. Sastre, J. Ibáñez, P. Ruiz, E. Defez, Efficient computation of the matrix cosine, Appl. Math. Comput. 219 (2013) 7575–7585.
[6] A.H. Al-Mohy, N.J. Higham, S.D. Relton, New algorithms for computing the matrix sine and cosine separately or simultaneously, SIAM J. Sci. Comput. 37 (1) (2015) A456–A487.
[7] M.S. Paterson, L.J. Stockmeyer, On the number of nonscalar multiplications necessary to evaluate polynomials, SIAM J. Comput. 2 (1) (1973) 60–66.
[8] J. Sastre, J.J. Ibáñez, E. Defez, P.A. Ruiz, Accurate matrix exponential computation to solve coupled differential models in engineering, Math. Comput. Modelling 54 (2011) 1835–1840.
[9] J. Sastre, J.J. Ibáñez, E. Defez, P.A. Ruiz, Efficient orthogonal matrix polynomial based method for computing matrix exponential, Appl.Math. Comput. 217 (2011) 6451–6463.
[10] P. Ruiz, J. Sastre, J. Ibáñez, E. Defez, High perfomance computing of the matrix exponential, J. Comput. Appl. Math. 291 (2016) 370–379.
[11] Mathworks, MATLAB MEX Files. http://www.mathworks.com/support/tech-notes/1600/1605.shtml#intro.
[12] NVIDIA, NVIDIA CUDA compute unified device architecture, 2009.
[13] NVIDIA, CUDA. CUBLAS library, 2009.
[14] L.S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R.C. Whaley, An updated set of basic linear algebra subprograms (BLAS), ACM Trans. Math. Software 28 (2) (2002) 135–151.
[15] O.A.R. Board, OpenMP Application Program Interface Version 3.1, July 2011.
[16] N.J. Higham, The test matrix toolbox for MATLAB, Numerical Analysis Report No. 237, Manchester, England, 1993.
[17] T.G. Wright, Eigtool, version 2.1, 2009. URL: web.comlab.ox.ac.uk/pseudospectra/eigtool.
[18] N.J. Higham, Fortran codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation, ACM Trans.Math. Softw. 14 (4) (1988) 381–396.