Alonso-Jordá, P.; Peinado Pinilla, J.; Ibáñez González, JJ.; Sastre, J.; Defez Candel, E. (2019). Computing Matrix Trigonometric Functions with GPUs through Matlab. The Journal of Supercomputing. 75(3):1227-1240. https://doi.org/10.1007/s11227-018-2354-1

Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Computing Matrix Trigonometric Functions with GPUs through Matlab

Pedro Alonso · Jesús Peinado · Javier Ibáñez · Jorge Sastre · Emilio Defez

Received: date / Accepted: date

Abstract This paper presents an implementation of one of the most up-to-day algorithms proposed to compute the matrix trigonometric functions sine and cosine. The method used is based on Taylor series approximations which intensively uses matrix multiplications. To accelerate matrix products our application can use from one to four NVIDIA GPUs by using the NVIDIA `cublas` and `cublasXt` libraries. The application, implemented in C++, can be used from the Matlab command line thanks to the `mex` files provided. We experimentally assess our implementation in modern and very high performance NVIDIA GPUs.

Keywords Matrix trigonometric functions, matrix cosine, matrix sine, GPU computing, MATLAB, `mex` MATLAB

1 Introduction

Many engineering processes which are described by differential equations can be solved through the computation of matrix trigonometric functions [1,2]. This functions can be computed using polynomial and rational approximations with scaling and recovering techniques [3,4,5,6,7,8]. The basic computational kernel on which these methods are based is matrix multiplication.

Pedro Alonso · Jesús Peinado
Depto. de Sistemas Informaticos y Computación
Pedro Alonso E-mail: palonso@upv.es
Jesús Peinado E-mail: jpeinado@dsic.upv.es
Javier Ibáñez
Instituto de Instrumentación para Imagen Molecular
E-mail: jjibanez@dsic.upv.es
Jorge Sastre
Instituto de Telecomunicaciones y Aplicaciones Multimedia (iTEAM)
E-mail: jsastrem@upv.es
Emilio Defez
Instituto de Matemática Multidisciplinar
E-mail: edefez@imm.upv.es
All authors belong to Universitat Politècnica de València

Matrix multiplication in a very costly operation that can be efficiently executed in NVIDIA GPUs through library `cublas` [9]. Moreover, matrix multiplication and the evaluation of matrix polynomials, in turn, have a high degree of parallelism that can be exploited in different parallel contexts. The work [10] studies the evaluation of matrix polynomials using one and two GPUs. Another work studies how to distribute the workload corresponding to the evaluation of a matrix polynomial among all over the GPUs and the CPU cores of a heterogeneous host [11]. We use the experiences of these last works to extend the implementation for one GPU presented in [12] to more GPUs. In general, performing a matrix multiplication into several GPUs is not complicated but there exist an overhead derived from data communications that can downgrade the application if it is not carefully tackled.

From our point of view the availability of a Matlab script capable of computing efficiently matrix trigonometric functions is a very desirable feature required by scientists. Thus, the challenge is twofold, exploiting several NVIDIA GPUs to improve computations and developing an easy-to-use interface that allows the user to execute our algorithm from Matlab. One of the main contributions is the design of a Matlab `mex` file that contains our implementation. We extend here the `mex` function proposed in [12] with more functionality. As a result, our `mex` function provides the user with the power of using one or two NVIDIA GPUs in an easy way. Furthermore, the `mex` function has also been adapted to use `cublasXt`, a new feature recently included in the lasts versions of `cublas` library that allows to use in the computation up to four GPUs and, in some situations, also the CPU cores.

This work also contributes with a study of computational aspects related to the algorithms proposed in [13], which is, to the best of our knowledge, the most recent contribution to the computation of matrix trigonometric functions that are based on Taylor series approximations. Since that work elaborates and proposes two algorithms for the efficient computation of these functions, we here study the real impact on performance of the two options when the matrix size is large and we use fast computing resources as GPUs.

The next section briefly describes the background of the maths that are behind the method we use to compute a matrix trigonometric function. A description of both our proposed "accelerated" implementation and the interface of the `mex` function is next. Section 4 shows the experimental results obtained with our software. And, finally, we summarize some conclusions in the last section.

2 Computing Matrix Trigonometric Functions

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

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

Let

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

be the Taylor series approximation of order $2m$ of $\cos(A)$, where $B = A^2$, then, once the algorithm computes (2) the approximation of $\cos(A)$ is recovered by

Algorithm 1 Given a matrix $A \in \mathbb{C}^{n\times n}$, this algorithm computes $C = \cos(A)$ by Taylor series.

1: Compute polynomial powers and selection of adequate values of s and m ▷ Phase I
2: $B^i \leftarrow 4^{-s}B^i$, $i = 1, \ldots, q$.
3: $C = P_{m_k}(B)$ ▷ Phase II: Compute Taylor series approximation
4: **for** $i = 1 : s$ **do** ▷ Phase III: Recovering $\cos(A)$
5: $\quad C \leftarrow 2C^2 - I$
6: **end for**

means of the double angle formula $\cos(2X) = 2\cos^2(X) - I$. The same algorithm can be used to compute de sine of a matrix given the fact that $\sin(A) = \cos(A - \frac{\pi}{2}I)$.

Algorithm 1 depicts the most important steps to compute the cosine of a matrix using our method based on Taylor series. In Phase I, the polynomial degree m (2) and a scale factor s are calculated so that the Taylor series approximation of the matrix can be computed in the most efficient way possible.

Phase II consists of computing the Taylor series approximation (2). The total number of matrix products to compute a matrix polynomial can be reduced using the Paterson-Stockmeyer method [14]. Using the same notation as in [6], we have that the Taylor matrix polynomial approximation (2), 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 using this method provided $m = m_k$, where m_k is one of the values shown in Table 1 (see [15] for a complete description). The algorithm computes firstly, in Phase I, the matrix powers $B^2, B^3, \ldots, B^q$, being q an integer divisor of m_k with value $q = \lceil\sqrt{m_k}\rceil$ or $q = \lfloor\sqrt{m_k}\rfloor$. As stated in [15] using these values for q results in the same cost when evaluating formula (2) using the Paterson-Stockmeyer's method. This method consists of computing the following [16],

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

Table 1 shows the values selected of $q = q_k$, which are in our case $q_k = \lceil\sqrt{m_k}\rceil$. Definitely, we should seek in Table 1 the proper pair of values m_k and q_k which gives as a result the minimum number of matrix products needed to evaluate the polynomial that is the best approximation to (2). As shown in [15, pp. 72–74] the cost of evaluating (2) by means of (3) in terms of matrix products is k.

Finally, Phase III is necessary to obtain the cosine of matrix A from $\cos(4^{-s}B)$ computed previously in Phase II.

The difficulty of the algorithms based on Taylor series is to find appropriate values m_k and s such that $\cos(A)$ is computed accurately taking into account computational costs and the truncation and rounding errors. In [13] we make a thorough analysis to determine the best values for the Taylor series approximation order m_k and the scaling factor s. One of the main conclusions is that, following [17], the final selection of m_k is the maximum order $m_k \in \{9, 12, 16\}$ giving

Table 1 Values of $q_k = \lceil \sqrt{m_k} \rceil$ for several values of m_k.

k	1	2	3	4	5	6	7	8	9	10	11	12
m_k	2	4	6	9	12	16	20	25	30	36	42	49
q_k	2	2	3	3	4	4	5	5	6	6	7	7

also the minimum cost. This selection provides the minimum scaling parameter s over all selections of m_k that provide the minimum cost. In this paper we use the two algorithms proposed in [13] to obtain the order m_k and the scaling factor s. The main difference between both algorithms is in the use of *estimation* of the norms of the matrix powers [18]. The algorithm that uses norm *estimation* can return lower values for m_k and s than if no norm estimation is used so resulting in a fewer number of matrix products. Norm *estimation*, however, is a costly process.

3 The "accelerated" version and the Matlab interface

We have implemented a version of Algorithm 1 that can use either one or two GPUs. In particular, we work with NVIDIA GPUs and the CUBLAS library to perform matrix multiplications, which is an implementation of library BLAS for NVIDIA devices. The accelerated version was developed with the aim of being not only efficient but also easy to use and easy to modify. Efficient means that fully exploits the capabilities of the existing GPUs in the system. We consider that the implementation is efficient if the performance achieved is equal or near to the performance obtained with the CUBLAS matrix multiplication, since the matrix multiplication is by far the most demanding computational kernel of Algorithm 1.

We implemented `mex` files in CUDA and C++ mainly of those parts of the original Matlab function that are good candidates to be accelerated. The Matlab interface of the `mex` function is the same independently of the number of GPUs. The strategy of implementation consisted of doing the matrix multiplication in the GPUs while leaving when possible the low cost operations to the host CPU. There are, however, some low cost operations implemented for the GPU with the aim at avoiding to download too many data from the GPU to the host. This is the case, for instance, of the operation to obtain the 1-norm of a matrix.

We decided to use only one `mex` function to implement all the different operations because some data must remain in the device memory between consecutive calls, and this is accomplished only if calling repeatedly to the same `mex` function. Thus, one `mex` function is called at different times in order to do different things. The Matlab `mex` function, called `call_gpu`, executes different operations depending on the arguments with which it is called. Data in both host and device memories are persistent all along the time the Matlab application is running. This feature is a key factor to allow versatility in our implementation, i.e. only those computationally expensive parts of the whole algorithm are carried out into the device, leaving the other parts to the Matlab script. Furthermore, this characteristic minimizes the overhead in which frequently incur these algorithms due to the amount of data transferred between the CPU and GPU subsystems.

The first argument of the `mex` function `call_gpu` is a string, named *command tag*, that labels the action to do. The rest of the arguments depend on the action to

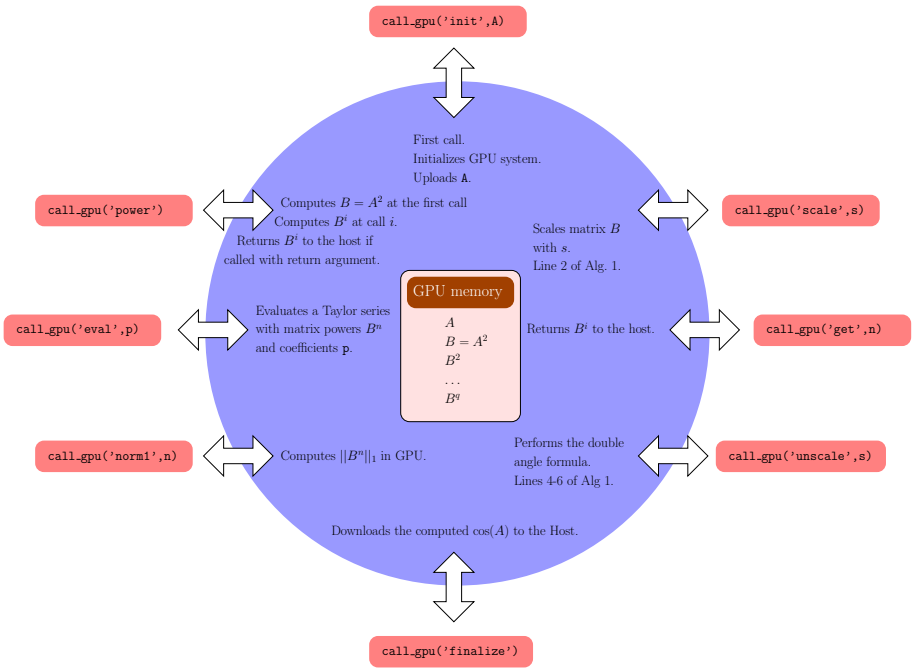

Fig. 1 Matlab interface of the `mex` function `call_gpu`.

be performed. For example, `call_gpu('init',A))`, which actually is the first call that must be written in the code, allocates memory into the device to host some matrices like, e.g. the input matrix A or the resulting matrix C, among others. Under this command, matrix A is also transferred from the Matlab working space in the host memory to the device memory. The rest of the actions that can be carried out only after initialization are (Fig. 1):

- `call_gpu('scale',s)`: Computes Phase I (line 2 of Algorithm 1).
- `call_gpu('unscale',s)`: Performs the loop specified in lines 4–6 of Algorithm 1 or Phase III.
- `call_gpu('finalize')`: Allocates memory in the Matlab working space and transfers the resulting matrix containing the cosine of matrix A from the device to the host. This must be the last call to the function.
- `call_gpu('power')`: Computes a matrix power. For the sake of simplicity, the power index is omitted in the routine arguments and is calculated according to the number of times the `mex` function has been called with this command tag. At the first call, computes $B = B_1 = A^2$; at the second one computes $B_2 = B_1^2$; and so on. This operation can optionally return the matrix power calculated if required, i.e. if the `mex` function has been called with a return argument, e.g. `M=call_gpu('power')`. This is useful in case the user wants norm *estimation* to obtain m_k and s. It must be note that, in this case, the matrix is explicitly downloaded from the GPU to the host.

Algorithm 2 Interface of class `cosmtay`.

```
class cosmtay {

  private:
    ... variables ...

  public:
    cosmtay( int n, double *A );
    ~cosmtay( );
    void power( );
    double norm1( int q ) const;
    void scale( int s );
    void evaluate( int m, double *p );
    void unscale( int s );
    void finalize( mxArray **plhs );
    void get( int i, double *A ) const;

};
```

- `call_gpu('eval',p)`: Evaluates the polynomial using the Paterson-Stockmeyer method explained through Eq. 3. This command implements entirely Phase II. Argument `p` represents the array with the m_k coefficients of the polynomial.
- `call_gpu('norm1',n)`: Computes the 1-norm of the polynomial matrix B_{n} inside the device, and returns the result to the host.

The software developed keeps track of actions carried out to avoid repeated actions of some commands, e.g. `'init'` can not be used twice or more times before calling the `mex` function with the command tag `'finalize'`.

This solution provides easiness for the user. To use the GPUs, the original Matlab code is modified with very few commands, i.e. only replacing few lines in the code with calls to the `mex` function with the appropriate tag command. This can be carry out by a non expert user on GPUs and/or on CUDA programming. The user who implements the Matlab code must be aware that once matrix A has been uploaded to the GPU, the derived matrices from A, i.e. the polynomial powers $B_1, B_2, \ldots$, will be stored only into the device. We have provided the `mex` function with another auxiliary command (`call_gpu('get',n)`) that returns matrix B_{n}. This operation, not necessary for the computation, is used for debugging purposes.

For the implementation of the `mex` function we used C++ language. We designed a C++ class (`cosmtay`) that implements all the tag commands through methods (Algorithm 2). All matrices allocated in GPU, among other objects, are members of the class. The object oriented programming style provided us with many facilities such as, for example, the reusability of code or the capacity of doing readily modifications. In particular, the version for 2 GPUs follows the same scheme: there exists one class that implements the basic structure, and another class, named `gpu`, that represents each one of the devices. The main class creates two objects of type `gpu` at the initialization step, and implements the same methods than the class of the version for 1 GPU (Algorithm 2). In this case, each method launches two threads using the OpenMP directive `sections` and each thread calls the same method over each instance of the class `gpu`, respectively. This way, both GPUs do the same work on their own partition of the data.

In the implementation of the algorithm for 2 GPUs, the matrix multiplications are split into two partitions. On each multiplication, one of the matrices is

partitioned into two equal pieces that are mapped onto each GPU in turn. The multiplication uses these matrices as one factor, while the other matrix is completely replicated on both GPUs. Some matrices, like e.g. matrix A, must be full stored on the two GPUs, but other matrices, like matrix powers B_1, B_2, ..., are half stored into each GPU. The benefit of the version for 2 GPUs is twofold: shorter execution time and the ability to solve problems of larger dimension. Notice that the amount of device memory is usually scarce, compared with the main memory of the current workstations. The downside of the version for 2 GPUs falls in the fact that there exists some data to be interchanged between the two devices. The GPUs used in our experiments are all attached to the host through a PCIe (Peripheral Component Interconnect Express) link. The PCIe bandwidth is lower than that between CPU processor and RAM memory in a host computer. Thus, the time consumed to move data from/to host to/from the device is not negligible and must be taken into account. In order to save time, we have exploited the possibility of doing this interchange through the PCIe directly between the two devices without host intervention, i.e. bypassing the host RAM memory, and exploiting the full duplex capability of the PCIe. In more detail, this has been carried out using CUDA Streams and the asynchronous copy routine `cudaMemcpyPeerAsync`.

Our `mex` function `call_gpu` can also execute on three and four NVIDIA GPUs. The `cublasXt` API of `cublas` exposes a multi-GPU capable computer that further accelerates BLAS Level-3 routines by dynamically routing BLAS calls to one or more NVIDIA GPUs. There exists the ability to use the CPUs as well for some routines. In order to exploit configurations with more than two GPUs we implemented a `cublasXt` version of our algorithm, keeping the Matlab interface designed for the former implementations. We note that when using this API the application only needs to allocate the required matrices on the host memory space. Thus, to implement this version we simply modified the version for 1 GPU replacing the corresponding `cublas` calls to the counterpart in `cublasXt` keeping all objects into the host memory. This option provides easiness and the capability of using very large matrix objects that do not fit into the GPUs memory.

4 Experimental Results

The experiments have been performed in two different computers. The first one is a host equipped with an Intel QuadCore i7-3820 (3.6Ghz) processor, to which there are two NVIDIA GPUs attached of type K20c (Kepler architecture). Each GPU has 13 multiprocessors with 192 cores each, resulting in a total of 2496 CUDA cores. Each device features 4800 MBytes of RAM memory. This equipment is representative of many workstations that, equipped with two GPUs devoted to scientific computing, provide a high performance capacity at a medium price. The second host, far more expensive and powerful, represents one of the most up-to-day high performance workstations that a researcher can use. This computer is equipped with two processors Intel Xeon CPU E5-2698 at 2.20 GHz featuring 20 cores each. Attached to the PCI of this board there are four NVIDIA Tesla P100 SMX2 (Pascal architecture) with 16 GB of memory each. One of these GPUs contains 56 multiprocessors with 64 CUDA cores each, resulting in a total of 3584 CUDA cores. The four GPUs are interconnected in turn through NVIDIA NVLink.

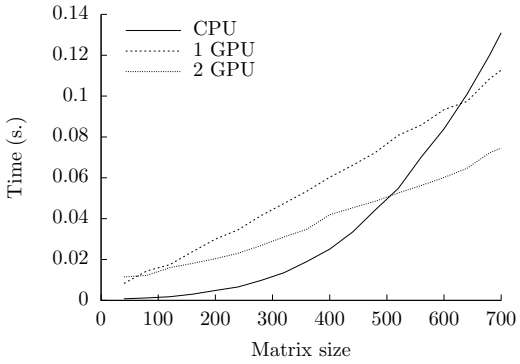

Fig. 2 Comparison of execution time among the CPU, the 1 GPU, and the 2 GPUs implementations, respectively.

The first experiment, carried out in the first host, is aimed at the selection of the best implementation according to its speed, i.e. the implementation that uses only the CPU (the Matlab script), 1 GPU or the 2 GPUs (Fig. 2). In general, it can be stated that the overhead incurred by data transference through the PCIe bus in implementations that use GPUs is too weighty in small problem sizes. The raw Matlab implementation for CPU is faster than the implementation for 1 GPU if the matrix size n is < 640, and faster than two GPUs if $n < 520$. The use of two GPUs is always better than using only one for matrices of size $n > 60$.

Our next analysis deals with the use or not of 1-norm estimation when using GPUs, i.e. the two algorithms of [13]. In our experiments we use Demmel [19], which are upper triangular Toeplitz matrices with just one eigenvalue (-1). Demmel matrices are generated using the Matlab script `demmel_demo.m` provided by the software EigTool [20]. With these matrices we can better observe the different behaviour when norm estimation is used or not since they are specially sensitive to this feature. To highlight the role of norm estimation with Demmel matrices, Table 2 shows the number of matrix products produced by the code for a given set of matrix sizes ranging from 3000 to 9000. According to the last two columns, clearly norm estimation produces fewer products.

In CPU, norm estimation always provides a performance improvement thanks to the reduction in the total number of matrix products. However, things are

Table 2 Polynomial degree (m), scale factor (s), and total of matrix products of Algorithm 1 for Demmel matrices.

	m		s		Total products	
n	No Est.	Est.	No Est.	Est.	No Est.	Est.
3000	16	16	10	7	17	14
4000	12	12	11	8	17	14
5000	12	16	11	8	17	15
6000	16	16	11	8	18	15
7000	12	16	12	8	18	15
8000	12	12	12	9	18	15
9000	12	16	12	9	18	16

Table 3 Comparison in execution time between using norm estimation (*Est.*) or not (*No Est.*), and the use of 1 or 2 GPUs of both architectures Kepler and Pascal.

		Phase I		Phase II		Phase III		Total	
	n	*No Est.*	*Est.*	*No Est.*	*Est.*	*No Est.*	*Est.*	*No Est.*	*Est.*
1 GPU Kepler	3000	0.62	0.92	0.18	0.19	0.57	0.41	**1.38**	1.51
	4000	1.06	1.48	0.29	0.29	1.46	1.07	**2.81**	2.84
	5000	1.70	2.23	0.55	0.81	2.82	2.07	**5.07**	5.11
	6000	2.55	3.66	1.35	1.35	4.74	3.47	8.63	**8.49**
	7000	3.72	5.24	1.44	2.13	8.20	5.51	13.36	**12.91**
	8000	5.16	6.88	2.10	2.10	12.06	9.09	19.33	**18.14**
	9000	7.05	9.47	2.98	4.43	17.19	12.95	27.22	**26.83**
2 GPUs Kepler	3000	0.34	0.58	0.11	0.11	0.44	0.31	**0.89**	1.01
	4000	0.58	0.92	0.17	0.17	1.01	0.74	**1.77**	1.83
	5000	0.93	1.33	0.32	0.45	1.85	1.36	**3.10**	3.13
	6000	1.39	2.31	0.74	0.74	2.98	2.18	**5.09**	5.28
	7000	2.02	3.28	0.80	1.15	5.02	3.37	7.84	**7.81**
	8000	2.79	4.19	1.16	1.16	7.27	5.47	11.23	**10.82**
	9000	3.78	5.79	1.63	2.36	10.16	7.65	**15.57**	15.79
1 GPU Pascal	3000	0.38	0.57	0.04	0.04	0.14	0.11	**0.57**	0.74
	4000	0.54	0.85	0.06	0.06	0.34	0.26	**0.95**	1.19
	5000	0.76	1.24	0.12	0.17	0.64	0.49	**1.53**	1.85
	6000	1.01	1.90	0.28	0.28	1.05	0.79	**2.35**	3.04
	7000	1.33	2.48	0.30	0.45	1.80	1.24	**3.44**	4.25
	8000	1.73	3.04	0.44	0.44	2.64	2.01	**4.80**	5.59
	9000	2.18	3.82	0.63	0.92	3.73	2.85	**6.56**	7.65
2 GPUs Pascal	3000	0.25	0.37	0.03	0.03	0.11	0.08	**0.39**	0.53
	4000	0.35	0.61	0.04	0.04	0.23	0.18	**0.64**	0.79
	5000	0.46	0.75	0.07	0.10	0.40	0.31	**0.93**	1.18
	6000	0.65	1.27	0.15	0.15	0.64	0.48	**1.42**	2.05
	7000	0.81	1.78	0.17	0.24	1.06	0.73	**2.06**	2.70
	8000	1.03	1.90	0.24	0.24	1.53	1.16	**2.79**	3.42
	9000	1.31	2.86	0.34	0.49	2.11	1.62	**3.73**	4.82

different when using the GPU implementations as it can be observed in Table 3. The last two columns show the execution time when no norm estimation is used (*No Est.*), and when it is used (*Est.*), respectively. There is not a direct correlation between the number of products and the execution time. To better understand the reason behind this behavior we have studied each part of the algorithm separately according to the three phases in which Algorithm 1 is partitioned.

Phase I involves matrix powers (B^2, B^3, ...) and, in case norm estimation is used, each computed power must be downloaded from the GPU to the CPU to figure out values m_k and s in this case into CPU. This transference time is not negligible. On the contrary, this transference is unnecessary in case no norm estimation is used. The figures representing the time used on Phase I in Table 3 show a significant difference. In the Kepler GPUs this overhead ranges from 30% to 50% for one GPU, and is always above 50% when using two GPUs. This overhead is more dramatic for the Pascal GPUs being close to 90% for some matrix sizes and one GPU, and reaching 120% with 2 Kepler GPUs. The good point here is that the performance obtained with two GPUs is very high. The efficiency is always above 90% for the Kepler GPUs if no norm estimation is used and around 80% if norm estimations is used. For the Pascal GPUs the efficiency is around 80% if no norm estimation is used and slightly lower when we compute norm estimation.

The computational cost of Phase II (for simplicity, Step 2 and 3 are both merged in this stage) is only influenced by the degree (m_k) of the polynomial to be evaluated. The polynomial degree is different whether norm estimation is used

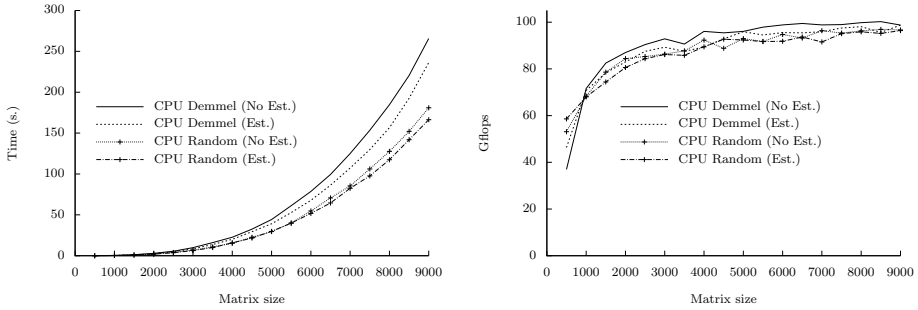

Fig. 3 Performance of the CPU implementation in time (left) and Gflops (right) for Demmel and random matrices with and without norm estimation (Intel QuadCore i7-3820).

or not (see Table 2), and so the time used to evaluate the polynomial. In general, this phase of the algorithm does not contribute too much to the total cost since, thanks to the use of the Paterson-Stockmeyer method and the particular value selected for q, the number of matrix products is as small as possible.

The last stage, Phase III, is the most critical because it accounts for the most part of the computational cost. The norm estimation highly contributes to reduce the number of matrix products taking place in this part due to the reduction of the scale factor (s) (See lines 4-6 of Algorithm 1). The scale factor when norm estimation is used is three (even four in one case) units smaller than when no norm estimation is used (see Table 2). This clearly explains the reduction in time of the two versions in this phase of the algorithm. The savings are around 35% for all sizes and 49% for $n = 7000$. These savings are a little lower for the Pascal GPUs. This phase of the algorithm is efficient ($\approx 80\%$ for $n \geq 6000$) when two GPUs are used, even the fact that, for the two devices can cooperate to recover the cosine at this stage of the algorithm, a data swapping between them is needed at each iteration of the loop. In the NVLink machine, this efficiency is around 3 points better, probably not too much as expected provided the high theoretical difference in performance between PCIe and NVLink.

We now introduce randomly generated matrices in our analysis. Figure 3 shows, on the left, the time spent in the host CPU with random and Demmel matrices and, on the right, the performance in Gflops, using norm estimation or not. Norm estimation reduces the execution time due to the fewer matrix products. Demmel matrices are a particular case in which norm estimation produces as a result a lower parameter s. However, this is not the case with randomly generated matrices, in which norm estimation does not produce a fewer number of products so, as conclusion, norm estimation should be use only in cases where it is known that this technique can effectively reduce the total number of matrix products.

Figure 4 shows the execution time for randomly generated and Demmel matrices with and without norm estimation in 1 and 2 Kepler GPUs. Norm estimation slightly reduces the execution time for Demmel matrices though, for of 2 GPUs it is impossible to appreciate any difference. Clearly, for randomly generated matrices it is better not to estimate. On the right, we show the performance in Gflops with 2 GPUs which actually achieves the theoretical maximum of 2 Tflops (the

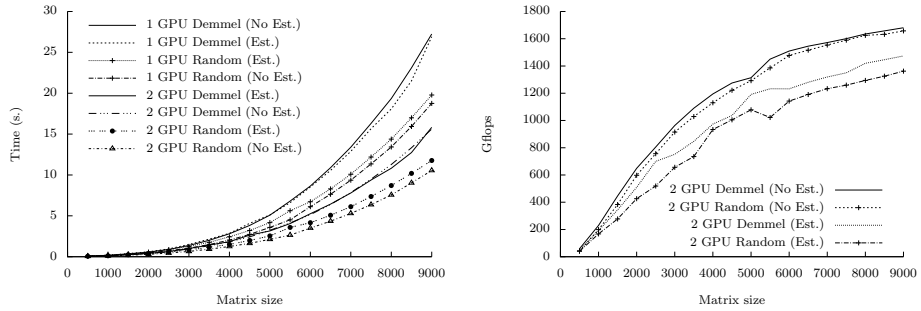

Fig. 4 Performance of the GPU implementation in the Kepler architecture. (Left) Time with 1 or 2 GPUs of Demmel and random matrices with and without norm estimation. (Right) Gflops achieved with 2 GPUs.

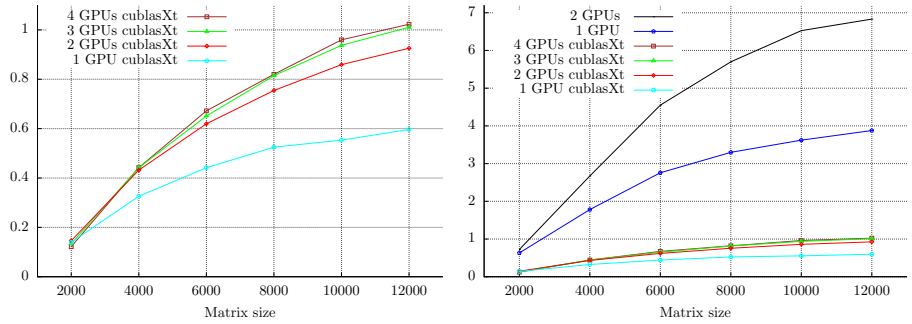

Fig. 5 Comparison in Tflops of using `cublasXt` with 1, 2, 3 or 4 GPUs (left), and plus `cublas` version with 1 or 2 GPUs (right), all in the Pascal GPUs host.

theoretical peak performance for 1 NVIDIA K20 GPU is 1 Tflop). We should notice here that, contrary to many other algorithms, the performance measured in flops does not perfectly relates with the best algorithm version, as it happens for Demmel matrices. It is also significant to observe that the performance without norm estimation outperforms the performance with norm estimation due to the smaller number of data transferences through the PCIe bus.

Finally, we show the results obtained with our `cublasXt` version in Fig. 5 on the Pascal GPUs. These tests have been taken with randomly generated matrices. There are two main observations. The first one is that there exists a certain degree of scalability that allows to use several GPUs without too much programming effort (Fig. 5, left). The second observation is easlily deduced from the figures on the right of Fig. 5. Clearly, in order to obtain a good performance of the algorithm the programmer must be strongly aware about where data are stored at each step to avoid as many transferences as possible. Downloading matrices to the host after each matrix multiplication results in a very poor performance, even in the case of compute bound applications like this one.

5 Conclusions

Algorithms based on Taylor series that compute the sine and cosine of a matrix use intensively matrix multiplication. This feature makes them good candidates to exploit one or more GPUs. The `cublasXt` API to `cublas` allows to use up to four GPUs but with less performance due to the large amount of data transferences. Our implementation for one or two GPUs is efficient since it is aware of where data is stored (host or device) at each step of the computation. We have shown that it is possible to readily access to several NVIDIA GPUs by means of a `mex` function, whose interface is independent of the underlying version to be used. Our application uses a very competitive and up-to-day algorithm to compute this matrix functions. The analysis regarding norm estimation allows to conclude that, as long as we do not know any particular feature of the matrix, the matrix is large and/or the computing device is more powerful, it is better not to estimate.

Acknowledgements

This work has been supported by Spanish Ministerio de Economía y Competitividad and the European Regional Development Fund (ERDF) grants TIN2014-59294-P and TEC2015-67387-C4-1-R.

References

1. S.M. Serbin. Rational approximations of trigonometric matrices with application to second-order systems of differential equations. *Appl. Math. Comput.*, 5(1):75–92, 1979.
2. Steven M. Serbin and Sybil A. Blalock. An algorithm for computing the matrix cosine. *SIAM J. Sci. Statist. Comput.*, 1(2):198–204, 1980.
3. G. I. Hargreaves and N. J. Higham. Efficient algorithms for the matrix cosine and sine. *Numer. Algorithms*, 40:383–400, 2005.
4. Awad H. Al-Mohy and Nicholas J. Higham. A new scaling and squaring algorithm for the matrix exponential. *SIAM J. Matrix Anal. Appl.*, 31(3):970–989, 2009.
5. E. Defez, J. Sastre, Javier J. Ibáñez, and Pedro A. Ruiz. Computing matrix functions arising in engineering models with orthogonal matrix polynomials. *Math. Comput. Model.*, pages 1738–1743, 2011.
6. J. Sastre, J. Ibáñez, P. Ruiz, and E. Defez. Efficient computation of the matrix cosine. *Appl. Math. Comput.*, 219:7575–7585, 2013.
7. Awad H. Al-Mohy, Nicholas J. Higham, and Samuel D. Relton. New algorithms for computing the matrix sine and cosine separately or simultaneously. *SIAM J. Sci. Comput.*, 37(1):A456–A487, 2015.
8. P. Alonso, J. Ibáñez, J. Sastre, J. Peinado, and E. Defez. Efficient and accurate algorithms for computing matrix trigonometric functions. *J. Comput. Appl. Math.*, 309(1):325–332, January 2017.
9. CUBLAS library, 2017. `http://docs.nvidia.com/cuda/cublas/index.html`, accessed May 2017.
10. P. Alonso Jordá, M. Boratto, J. Peinado Pinilla, JJ. Ibáñez González, and J. Sastre Martínez. On the evaluation of matrix polynomials using several GPGPUs. *Universitat Politècnica de València*, 2014. (published online). http://hdl.handle.net/10251/39615.
11. Murilo Boratto, Pedro Alonso, Domingo Giménez, and Alexey L. Lastovetsky. Automatic tuning to performance modelling of matrix polynomials on multicore and multi-gpu systems. *The Journal of Supercomputing*, 73(1):227–239, 2017.
12. P. Alonso, J. Peinado, J. Ibáñez, J. Sastre, and E. Defez. A fast implementation of matrix trigonometric functions Sine and Cosine. In *Proceedings of the 17th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE 2017)*, pages 51–55, Costa Ballena, Rota, Cadiz (Spain), July 4th-8th, 2017.

13. Jorge Sastre, Javier Ibáñez, Pedro Alonso, Jesús Peinado, and Emilio Defez. Two algorithms for computing the matrix cosine function. *Applied Mathematics and Computation*, 312:66–77, 2017.
14. Michael S. Paterson and Larry J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM J. Comput.*, 2(1):60–66, 1973.
15. Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. SIAM, Philadelphia, PA, USA, 2008.
16. J. Sastre, Javier J. Ibáñez, E. Defez, and Pedro A. Ruiz. Efficient orthogonal matrix polynomial based method for computing matrix exponential. *Appl. Math. Comput.*, 217:6451–6463, 2011.
17. J. Sastre, Javier J. Ibáñez, E. Defez, and Pedro A. Ruiz. Efficient scaling-squaring Taylor method for computing matrix exponential. *SIAM J. on Sci. Comp.*, 37(1):A439–455, 2015.
18. N. J. Higham and F. Tisseur. A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra. *SIAM J. Matrix Anal. Appl.*, 21:1185–1201, 2000.
19. J. W. Demmel. A counterexample for two conjectures about stability. *IEEE Trans. Auto. Control, AC-32*, pages 340–343, 1987.
20. Thomas G. Wright. EigTool library, 2002. `http://www.comlab.ox.ac.uk/pseudospectra/eigtool/`, accessed May 2017.