On the evaluation of matrix polynomials using several GPGPUs

Pedro Alonso[1], Murilo Boratto[2], J. Peinado[3], J. Ibáñez[3], and Jorge Sastre[4]

[1]Dep. of Information Systems and Computation, Universitat Politècnica de València.
[2]Núcleo de Arquitetura de Computadores e Sistemas Operacionais, Universidade do Estado da Bahia.
[3]Instituto de Instrumentación para Imagen Molecular, Universitat Politècnica de València.
[4]Instituto de Telecomunicaciones y Aplicaciones Multimedia, Universitat Politècnica de València.

August 4, 2014

Abstract

Computing a matrix polynomial is the basic process in the calculation of functions of matrices by the Taylor method. One of the most efficient techniques for computing matrix polynomials is based on the Paterson–Stockmeyer method. Inspired by this method, we propose in this work a recursive algorithm and an efficient implementation that exploit the heterogeneous nature of current computers to evaluate large scale matrix polynomials is the shortest possible time. Heterogeneous computers are those which have any type of *hardware accelerator(s)*. For these type of computers, we propose a method to easily implement efficient algorithms that use several hardware accelerators in parallel. This methodology is built on the last versions of the OpenMP standard for implementing parallel algorithms on shared memory multiprocessors. In particular, we have used NVIDIA© cards, but the proposal can be readily generalized to other type of devices acting as coprocessors. In addition, we provide a high-level interface in Matlab© to be used by any researcher who is not aware of parallelism nor of other programming issues.

Keywords. matrix polynomial, function of matrices, GPGPU, CUDA, CUBLAS, Matlab.

1 Introduction

The calculus of matrix polynomials has received a large boost in the past since this is a core operation to compute matrix functions [1]. Many engineering and physics phenomena are governed by systems of linear first-order ordinary differential equations with constant coefficients, whose solution is given in terms of the matrix exponential. The matrix exponential plays an important role in many areas of science and technology, i.e. control theory, electrodynamic theory of stratified media, the theory of multimode electric power lines, etc. [2, 3, 4]. Other matrix functions like the matrix sign function and the matrix logarithm function appear, e.g., in control theory [1, pp 39]. Also, some other engineering processes are described by second order differential equations, whose exact solution is given in terms of the trigonometric matrix function sine and cosine.

There are different techniques for computing or approximating matrix functions. Some of them are very general but others are specialized to particular functions. Two techniques are widely used to approximate a matrix function, one is based on polynomial approximations and the other is based on rational approximations. Contrary to what was thought in the past, it is possible to obtain more accuracy with polynomial approximations than with rational approximations, even with similar or lower computational cost [5, 6, 7, 8]. This fact mainly motivates our study on how to efficiently compute matrix polynomials in this paper.

We define a matrix polynomial P of degree d as

$$P = \sum_{i=0}^{d} \alpha_{d-i} X^{d-i} = \alpha_d X^d + \alpha_{d-1} X^{d-1} + \cdots + \alpha_1 X + \alpha_0 I \ , \qquad (1)$$

where $X, I \in \mathcal{R}^{n \times n}$, being I the identity matrix. We also define array $\bar{\alpha}$ as $\bar{\alpha} = [\alpha_i]_{i=0,\ldots,d}$ for convenience in further descriptions. The main idea in this paper is to show how to compute matrix polynomials for large scale problems. We consider as large scale problems those in which the matrix size, the degree of the polynomial or both in turn are large.

We have designed and evaluated a set of algorithms to evaluate matrix polynomials concurrently using two GPUs. We propose a method to implement these algorithms based on Posix threads, one thread bound to a different GPU. The threads are spawned through easy of use OpenMP directives.

First, we have proposed and experimentally evaluated a fairly simple parallelization option with the aim at compare other more sophisticated options. Section 3 describes the evaluation of matrix polynomials in one and two GPUs using this simple parallelization, and Section 4 shows the experimental results. Then, we have studied and incorporated a method to reduce the number of needed operations for evaluating polynomials: the Paterson–Stockmeyer method [9]. We have tailored this method to the case of matrix polynomials giving as a result some simple algorithms that are shown in Section 5. The simplicity of these algorithms is due to their recursive nature, derived in turn from the Horner's rule,

a method to reduce the number of matrix multiplications in the evaluation of a polynomial [10]. In the same section we explain how to apply this method to the design of a parallel algorithm that uses two GPUs. The experimental results using this approach are explained in the following section. One of the highlights of our contribution is explained in Section 7, where we present and show some experimental results obtained using a Matlab interface designed by us to easily use the software developed in this work. Section 2 shows the environment (hardware and software) used for the experimental evaluation of the algorithms. This information is presented here since algorithms and experimental evaluation are interleaved along the rest of the paper. Finally, some conclusions are outlined in Section 8.

2 Environment: hardware and software used

In this section we describe the hardware and software used to develop all the serial and parallel programs and to obtain the experimental results. We used two NVIDIA devices model K20 ("Kepler" architecture) [11]. Both cards are connected to the PCI bus of a personal computer provided with an Intel processor Quadcore i7-3820 at 3.6 GHz and 16 GB of RAM. All the computations have been executed using double precision arithmetic. The CPU code has been compiled with the GNU's `gcc` compiler. We have used the Intel Math Kernel Library (MKL 11.0) implementation of BLAS and LAPACK [12] libraries for matrix computations. The most used routine in our case, i.e. routine `DGEMM` for matrix multiplication, is multithreaded in the MKL implementation to exploit as far as possible the use of a multicore CPU. CUDA language [13] was used to write the GPU code. These source files were compiled using the NVIDIA `nvcc` compiler. The library used for matrix computations in GPU was CUBLAS, a library that contains implementations for GPUs of many BLAS routines [14]. The compiler used to build the host part of the program implements the OpenMP API [15] (version 3.0).

Finally, we have developed a high level interface to use all the programs developed for both CPU and GPUs. Since the most likely users are mathematicians and engineers not specialized on computer systems, we provide the necessary tools to access our software in the friendliest way possible. That is why in this work we have developed commands that run in the execution environment of the Linux operating system, but we have also developed functions that allow to execute our routines from a Matlab interface. For instance, all the results shown in Section 7 have been taken using a Matlab interface [16]. The version of Matlab is modern (R2013b), but we checked that our codes work properly on earlier versions (from R2009 on).

Algorithm 1 Algorithm for the evaluation of a matrix polynomial.

```
1: function EVALUATE( n, X, d, ᾱ ) return P
2:     P ← α_0 I
3:     P ← P + α_1 X
4:     B ← X
5:     for i ← 2, d do
6:         A ← B
7:         B ← A · X
8:         P ← P + α_i B
9:     end for
10: end function
```

3 Simple algorithms to evaluate matrix polynomials in one or two GPUs

In this section we show a simple algorithm to evaluate a polynomial using only one GPU. We also study here a parallel algorithm that makes use of two GPUs to solve the same problem. This algorithm basically divides the work of powering matrices between two GPUs. Neither version make use of any existing technique for reducing the number of computations needed to evaluate polynomials.

The evaluation of the matrix polynomial (1) can be quite straightforward by using an algorithm like the one shown in Alg. 1.

An algorithm to evaluate (1) can be expressed in many ways. The chosen procedure EVALUATE (Alg. 1) helps the user to easily implement it on CPU using BLAS routines since workspaces A, B and P have been expressly introduced to this end. An implementation for one GPU of Alg. 1 is also very easy using CUBLAS routines and provided matrix X has been uploaded to the GPU.

The evaluation of matrix polynomial (1) in two GPUs is more complicated. Both GPU can work independently of each other so we need to reflect somehow this fact on our code to exploit this concurrency. The asynchronous nature of kernel execution with respect to the host allows implementing a solution based on, e.g. replicated calls, i.e. one call of each kernel/routine for each GPU. However, in this contribution we propose to use a different but easy model to program and manage this concurrency. We have chosen to use a SPMD (*Single Program, Multiple Data*) programming model to develop the GPU program. This model is easy to understand and very used in distributed memory parallel contexts where programmers use the Message Passing Programming model. Under this paradigm, both GPUs share the same code (only small parts of code are specific for each one of the devices). The strategy consists of spawning two Posix threads using OpenMP, in particular, by using an OpenMP `parallel for` loop. This way, each GPU executes one iteration of the loop over its own data. Since all the code might be very large, it is useful to use several parallel loops all along the source file. Yet creation and destruction of threads, also involves the creation and destruction of local variables to these threads, making it impossible

Algorithm 2 Algorithm for computing the matrix powers of X in two GPUs.

1: **function** COMPUTE_POWERS(d, X) **return** (A, m)
2: `#pragma omp parallel for`
3: **for** $g \leftarrow 0, 1$ **do**
4: $m = d/2 + g \cdot \mathrm{mod}(d, 2)$
5: $A(0) \leftarrow X$
6: $A(1) \leftarrow X \cdot A(0)$ ▷ $A(1) = X^2$
7: **if** $g = 0$ **then**
8: $A(0) \leftarrow A(1)$ ▷ $A(0) = X^2$ if $g = 0$
9: **end if**
10: $A(2) \leftarrow A(0) \cdot A(1)$ ▷ $A(2) = \begin{cases} X^4 & \text{if } g = 0 \\ X^3 & \text{if } g = 1 \end{cases}$
11: **for** $i \leftarrow 3, m$ **do**
12: $A(i) \leftarrow A(1) \cdot A(i-1)$ ▷ $A(i) = X^2 \cdot A(i-1)$
13: **end for**
14: **end for**
15: **end function**

to use this idea. For the solution of this problem, we used the `threadprivate` OpenMP feature (incorporated in the V3.0 specification) to qualify those variables that should be persistent throughout the entire process. Once created, these variables belong to a given thread and keep the value stored in it between the destruction and the creation of the thread, i.e., between consecutive parallel loops. This programming model is readily applicable to more than two GPUs, even to other different devices existing in a heterogeneous context.

In the case of two GPUs, we have divided the process into two stages: *a)* the computation of the powers of matrix X, and *b)* the evaluation of the polynomial. This method is more difficult to be programmed than the method described in Alg. 1 where computing the matrix powers and evaluating the polynomial all at the same time. Yet, separating the process in this way has the benefit of being able to evaluate more than one polynomial of the same order once the first stage has been carried out. On the contrary, the need of explicitly store all of the matrix powers poses a limitation to this method.

As described early, we manage both GPUs currently by spawning two CPU threads through an OpenMP `parallel for`, so each thread is bound to a GPU. Alg. 2 describes the first stage, i.e., the computation of the matrix powers, and uses the mentioned directive for the parallel loop. Both GPUs share the same code. This code is written inside a parallel OpenMP loop of two iterations, each GPU executes just one of these iterations according to its device number. Each GPU is in charge of computing a different set of matrix powers. We denote GPU0 the first GPU and by GPU1 the second one, so that GPU0 (with loop index $g = 0$) computes the even powers of X, and GPU1 (with loop index $g = 1$) computes the odd ones.

The powers of matrix X are stored in an array of matrices called A, so

$$A = [A(i)]_{i=0,\ldots,m},\ A(i) \in \mathcal{R}^{n \times n}\ ,$$

begin $m = \frac{d}{2} + g \cdot \mathrm{mod}(d, 2)$, where $g \in \{0, 1\}$ is the device number. The value of this array for each GPU after execution of Alg. 2 is

$$A = \begin{cases} \begin{pmatrix} X^2 & X^2 & X^4 & X^6 & X^8 & \ldots \end{pmatrix} & \text{if GPU0} \\ \begin{pmatrix} X & X^2 & X^3 & X^5 & X^7 & \ldots \end{pmatrix} & \text{if GPU1} \end{cases} . \tag{2}$$

Matrices $A(0)$ and $A(1)$ have been used in a special way in both devices so that matrix products can be carried out through the suitable routine of CUBLAS allowing to directly translate Alg. 2 into a CUDA implementation. This implies that GPU1 has to compute X^2 (line 5) although this is not an odd power, because this term is needed to compute the remaining powers. Also, GPU0 has to store X^2 in $A(0)$ (line 8) although this term is already in $A(1)$, because matrix arguments in the CUBLAS matrix multiplication routine can not reference the same memory space.

The second stage is the evaluation of the polynomial. This stage is described in Alg. 3. The implementation is based on the same idea used in Alg. 2 consisting of a parallel loop where each GPU executes one iteration. The GPUs use the array A computed in the first stage (Alg. 2). The algorithm receives a polynomial degree q as parameter that will be d as actual argument for the solution of (1). GPU0 computes the even terms of (1) in matrix B, whereas GPU1 computes the odd ones in its own matrix B. Matrix hB is an array of two matrices allocated in the host. Matrix $hB(0)$ is for GPU0 and $hB(1)$ is for GPU1. Each GPU uploads its own matrix B onto the corresponding component of matrix hB (line 13). The last step (line 14) is carried out by the Host to form the final result, i.e. the evaluation of the matrix polynomial P (1).

For simplicity in the description of the above algorithms we omitted references to transferences Host-GPU that are needed previously to the computation.

4 Results of the evaluation of the matrix polynomial

We firstly show in Fig. 1 time (left) and speedup (right) for the evaluation of a matrix polynomial like (1) with different degrees ranging from 4 to 20, where matrix X is of order $n = 4000$. Both plots show how the use of GPUs in our system clearly outperforms the computation on the CPU so we will omit the results on CPU in the rest of the analysis.

Fig. 2 shows the speedup achieved using 2 GPUs with regard to only one. The speedup grows with the degree of the polynomial and also with regard to the matrix size. Yet, in this last case, the increment in speedup is smaller (sometimes null) than the one obtained along the rising in the polynomial degree for this range of matrix sizes.

Algorithm 3 Algorithm for the evaluation of a matrix polynomial in two GPUs.

```
1: function EVALUATE2( n, q, ᾱ, A ) return P
2:     #pragma omp parallel for
3:     for g ← 0, 1 do
4:         m = q/2 + g · mod(q, 2)
5:         if g = 0 then
6:             B ← α_0 I
7:         else
8:             B ← α_1 A(0)
9:         end if
10:        for i ← 1 + g, m do
11:            B ← B + α_{2i−g} A(i)
12:        end for
13:        hB(g) ← B
14:    end for
15:    (Host) P ← hB(0) + hB(1)
16: end function
```

The sawtooth shape of the graph is due to the unbalanced workload for degrees of the polynomial which are odd, since in these cases one of the two GPUs performs one more matrix multiplication.

We also present the ratio between the time to compute the powers of matrices in the polynomial and the evaluation time (Fig. 3). The former is a cubical cost step while the last one is quadratic, it involves scalar-matrix products and matrix sums. The figure clearly shows how large the computation of matrix powers is regarding the evaluation of the polynomial, and how this ratio rises with the matrix size and the polynomial degree.

To finish the exposition of this solution, we stressed the system with the largest possible matrix size, which is $n = 11000$, and a polynomial degree of 6, obtaining the solution in 13.50 sec. with one GPU and in 8.43 sec. with the two GPUs.

5 An efficient parallel algorithm

There exists a sequential technique that allows to reduce the number of computations needed to evaluate a polynomial like (1). This technique is based on the Paterson–Stockmeyer method [9]. From now on, and for the sake of clarity and briefness, we will denote this method as *boxing*.

We show this technique through an example. Suppose the degree of the polynomial to evaluate is $d = 14$. Polynomial (1) can be expressed in the

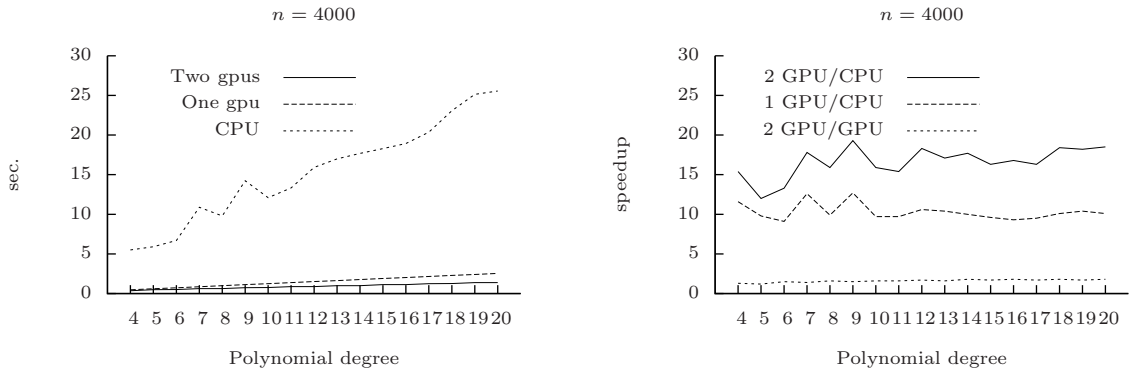

Figure 1: Time and speedup for the evaluation of a polynomial on matrices of size $n = 4000$ with regard to the polynomial degree.

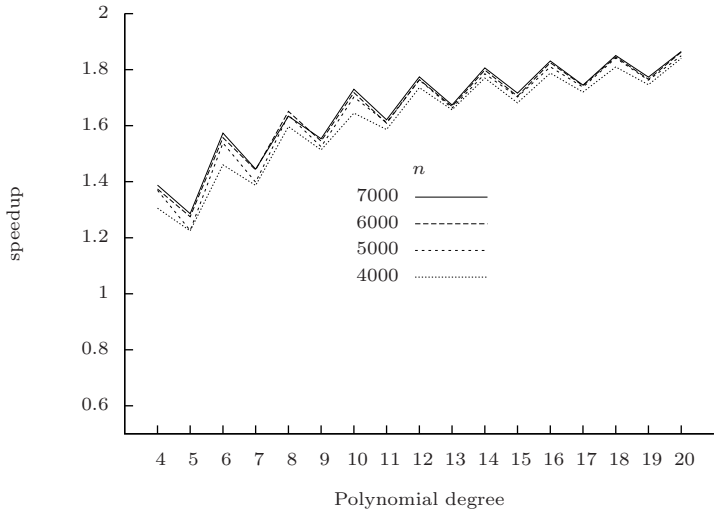

Figure 2: Speed up for the evaluation of a polynomial on 2 GPUs with respect to 1 GPU.

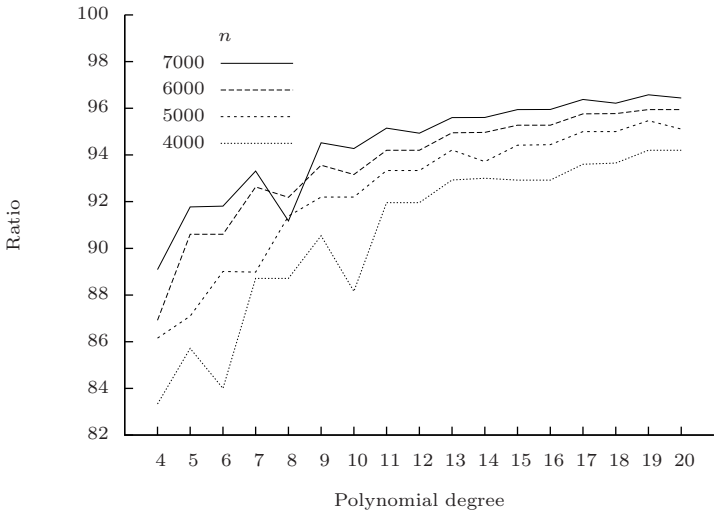

Figure 3: Ratio between computation of the matrix powers and evaluation of the polynomial (in percentage) in the two GPUs.

following way using the associative propriety:

$$\begin{aligned} P &= \sum_{i=0}^{14} \alpha_{14-i} X^{14-i} \qquad (3)\\ &= \alpha_{14}X^{14} + \alpha_{13}X^{13} + \alpha_{12}X^{12} + \alpha_{11}X^{11} + \alpha_{10}X^{10} + \alpha_9 X^9 + \alpha_8 X^8 \\ &+ \alpha_7 X^7 + \alpha_6 X^6 + \alpha_5 X^5 + \alpha_4 X^4 + \alpha_3 X^3 + \alpha_2 X^2 + \alpha_1 X + \alpha_0 I \\ &= X^{12}\left(\alpha_{14}X^2 + \alpha_{13}X + \alpha_{12}I\right) + X^8\left(\alpha_{11}X^3 + \alpha_{10}X^2 + \alpha_9 X + \alpha_8 I\right) \\ &+ X^4\left(\alpha_7 X^3 + \alpha_6 X^2 + \alpha_5 X^1 + \alpha_4 I\right) + \left(\alpha_3 X^3 + \alpha_2 X^2 + \alpha_1 X + \alpha_0 I\right) \ . \end{aligned}$$

Let $Q^q(\bar{\alpha}, X)$ be the polynomial of degree q in X with coefficients given by vector $\bar{\alpha} = \{\alpha_q, \alpha_{q-1}, \ldots, \alpha_1, \alpha_0\}$,

$$Q^q(\bar{\alpha}) = Q^q(\bar{\alpha}, X) = \alpha_q X^q + \alpha_{q-1} X^{q-1} + \ldots + \alpha_1 X + \alpha_0 I \ , \qquad (4)$$

then polynomial (3) can be written as

$$\begin{aligned} P &= X^{12}Q^2(\bar{\alpha}_{14:12}) + X^8 Q^3(\bar{\alpha}_{11:8}) + X^4 Q^3(\bar{\alpha}_{7:4}) + Q^3(\bar{\alpha}_{3:0}) \qquad (5)\\ &= X^4(X^4(X^4(Q^2(\bar{\alpha}_{14:12})) + Q^3(\bar{\alpha}_{11:8})) + Q^3(\bar{\alpha}_{7:4})) + Q^3(\bar{\alpha}_{3:0}) \ . \end{aligned}$$

The example uses a "boxing" size of $b = 3$, which means that the largest polynomial in expression (5) is 3. The boxing size b also means that power $b + 1$ of matrix X (X^4 in the example) is used as common factor to derive expression (5). It is easy to see that the number of operations in (5) is less than in (1).

Algorithm 4 Algorithm for the evaluation of a matrix polynomial using *boxing*.

1: **function** BOX_EVAL(n, X, d, $\bar{\alpha}$, b) **return** P
2: $A(0) = X$
3: **for** $i \leftarrow 1, b$ **do**
4: $A(i) \leftarrow A(i-1) \cdot X$
5: **end for**
6: $P \leftarrow$ BOXING(n, d, b, 0, $\bar{\alpha}$, A)
7: **end function**

Algorithm 5 Recursive algorithm to evaluate a matrix polynomial using *boxing*.

1: **function** BOXING(n, d, b, i, $\bar{\alpha}$, A) **return** P
2: $q \leftarrow d - i$
3: **if** $q \leq b$ **then**
4: $P \leftarrow$ EVAL(n, q, $\bar{\alpha}_{q+i:i}$, A)
5: **else**
6: $Q_1 \leftarrow$ BOXING(n, d, b, $b+i$, $\bar{\alpha}$, A)
7: $q \leftarrow b - 1$
8: $Q_2 \leftarrow$ EVAL(n, q, $\bar{\alpha}_{q+i:i}$, A)
9: $P \leftarrow A(q) \cdot Q_1 + Q_2$
10: **end if**
11: **end function**

Alg. 4 shows the process of evaluating a polynomial using boxing. Lines 2-5 fill array A of size $b+1$ with powers of X so that

$$A = [X^{i+1}]_{i=0,\ldots,b} = \left(\begin{array}{ccccccc} X & X^2 & X^3 & \ldots & X^{b-1} & X^b & X^{b+1} \end{array} \right) . \quad (6)$$

Array A will be used to evaluate polynomials of type $Q^q(\bar{\alpha}, X)$ in expression (4). This algorithm is equivalent to Alg. 1 since both solve the same problem. Once array A has been computed in Alg. 4, the algorithm calls routine BOXING, a name we have chosen to denote the evaluation of the matrix polynomial.

The BOXING routine is depicted in Alg. 5. Alg. 5 is a recursive algorithm that evaluates polynomials using the derivation shown in (5). Assuming a boxing factor of b, in the general case the algorithm computes a matrix P_i,

$$P_i \leftarrow X^{b+1} P_{b+i+1} + Q^b(\bar{\alpha}_{b+i:i}) = X^{b+1} Q_1 + Q_2 \, , \quad (7)$$

for $i = 0, b+1, 2(b+1), 3(b+1), \ldots$ The base case of this recursion is met when $d - i \leq b$, which means that there is no longer possibility of doing boxing. In this case, the algorithm evaluates the polynomial $Q^q(\bar{\alpha}_{q+i:i})$, being $q = d - i$. It can be easily shown that recursion (7) applied to example (5) gives

$$\begin{aligned} P = P_0 &= X^4 P_4 + Q^3(\bar{\alpha}_{3:0}) \\ P_4 &= X^4 P_8 + Q^3(\bar{\alpha}_{7:4}) \end{aligned}$$

Algorithm 6 Given the $b+1$ powers of a matrix X stored into array A as shown in (6), this algorithm evaluates a matrix polynomial.

```
1: function EVAL( q, ᾱ, A ) return R
2:     R ← α_0 I
3:     for i ← 0, q − 1 do
4:         R ← R + α_{i+1} A(i)
5:     end for
6: end function
```

$$\begin{aligned} P_8 &= X^4 P_{12} + Q^3(\bar{\alpha}_{11:8}) \\ P_{12} &= Q^2(\bar{\alpha}_{14:12}) \, . \end{aligned}$$

The recursive function BOXING makes use of another function called EVAL. Function EVAL(q, $\bar{\alpha}$, A), shown in Alg. 6, computes $Q^q(\bar{\alpha}_{q+i:i})$ provided $q \leq b$.

Now we elaborate on the theoretical cost of the algorithms in terms of matrix products since this is by far the most expensive operation to evaluate a polynomial. Let d be the degree of the polynomial to compute, then Alg. 1 performs $d-1$ matrix products to evaluate the polynomial. Suppose b is the boxing factor, i.e. the degree of the boxing polynomial, then the number of products to obtain the powers of X (6) is $b-1$, to which we add one more product to obtain X^{b+1}, giving a total of b matrix products (Alg. 4). For the sake of simplicity assume now that $d+1 = k \cdot (b+1)$, being k a positive integer, thus Alg. 5 performs $k-1$ matrix products in recursion (7). The total number of matrix products for Alg. 4 then is

$$b + k - 1 = b + \frac{d+1}{b+1} - 1 \, . \tag{8}$$

A minimum number of matrix products for the boxing method comes out from derivation of expresion (8) resulting in a value for b as the closest integer to $\sqrt{d+1} - 1$. Using this value, the speedup attainable can be calculated as

$$\text{speedup} = \frac{d-1}{2\left(\sqrt{d+1} - 1\right)} \, ,$$

and it is not difficult to show that $\frac{\sqrt{d+1}}{2}$ is a lower bound of the theoretical speedup, for $d \geq 3$.

The speedup of Alg. 2 when using 2 GPUs grows with the degree of the polynomial (Fig. 2). When boxing is used, the degree b of the boxing polynomial is likely to be small, thus limiting room for speeding up the application. Therefore, we propose a different approach to address the problem of improving the performance of polynomial evaluation on 2 GPUs. Our solution is based on matrix partitioning to allow both the 2 GPUs to participate in the multiplication of two matrices.

Consider the following partitions $X^{p-1} = \left[\begin{array}{cc} X_1 & X_2 \end{array} \right]$ and $X^p = \left[\begin{array}{cc} Y_1 & Y_2 \end{array} \right]$,

Algorithm 7 Algorithm for computing the $b+1$ matrix powers of X into two GPUs.

```
1: function COMPUTE_POWERS2( n, b, hX ) return (A, X, B)
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:         X ⇐ hX
6:         A(0) ← X(:, i_g : j_g)
7:         for k ← 1, b − 1 do
8:             A(k) ← X · A(k − 1)
9:         end for
10:        B(:, i_g : j_g) ← X · A(b − 1)
11:        B(:, i_ḡ : j_ḡ) ⇐ B̄(:, i_ḡ : j_ḡ)
12:    end for
13: end function
```

where $X_1, Y_1, X_2, Y_2 \in \mathcal{R}^{n\times(n/2)}$ (assume for simplicity that n is even), then

$$X^p = X \cdot X^{p-1} = X \cdot \left[\begin{array}{cc} X_1 & X_2 \end{array}\right] = \left[\begin{array}{cc} X \cdot X_1 & X \cdot X_2 \end{array}\right] = \left[\begin{array}{cc} Y_1 & Y_2 \end{array}\right] ,$$

proving that the computation of any power p of X can be carried out by performing two independent matrix products, provided we have the whole matrix X and the two halves of the partition of the power $p-1$ of X. Alg. 7 performs the computation of the powers of X using this partition of the matrices. At the first (line 5), both GPUs receive the whole matrix X from the host (denoted as hX). Symbol $\Leftarrow$ represents transferences Host-GPU or GPU-GPU. A half of X factor is copied onto the first matrix of the array, i.e. $A(0)$ (line 6). Next, by the loop at lines 7 to 9 the algorithm computes one half of each power of X. GPU0 computes the first $n/2$ columns of these factors and stores them in matrices $A(i) \in \mathcal{R}^{n\times n/2}$, $i = 0, \ldots, b-1$. Similarly, GPU1 stores the $n/2 + mod(n, 2)$ columns from $n/2$ to $n-1$ of the matrix powers in its corresponding matrices $A(i) \in \mathcal{R}^{n\times n/2+mod(n,2)}$. To evaluate boxing polynomials later of degree b, it is also needed that both GPUs have the whole factor X^{b+1}. Each GPU has computed only one half of this factor, that it has been stored in the corresponding half of B (line 10). We need now to interchange these parts of factor B so that finally the two GPUs have X^{b+1} stored into B. Int line 11, GPU g sends columns $i_g : j_g$ to the other GPU. The index of the other GPU is denoted by $\bar{g}$, where $\bar{g} = 1 - g$. Matrix $\bar{B}$ refers to matrix B on the other GPU. We used the CUDA routine `cudaMemcpyPeerAsync` to perform this peer-to-peer communication between the two GPUs. This function allows memory copies between the memories of two different devices bypassing the memory host. The routine does not start until all commands previously issued to either device have completed and is asynchronous with respect to the host.

The algorithm to evaluate polynomial (1) using bonxing in two GPUs is written in Alg. 8, provided Alg. 7 (COMPUTE_POWERS2) has already been computed

Algorithm 8 Recursive algorithm to evaluate a matrix polynomial using *boxing* in two GPUs.

```
1: function BOXING2( d, b, i, ᾱ, A, B ) return P
2:     #pragma omp parallel for
3:     for g ← 0, 1 do
4:         q ← d − i
5:         if q < b then
6:             P ← EVAL( q, ᾱ_{q+i:i}, A )                    ▷ Alg. 6
7:         else
8:             Q_1 ← BOXING2( d, b, b + i, ᾱ, A, B )
9:             q ← b − 1
10:            Q_2 ← EVAL( q, ᾱ_{q+i:i}, A )                  ▷ Alg. 6
11:            P ← B · Q_1 + Q_2
12:        end if
13:    end for
14: end function
```

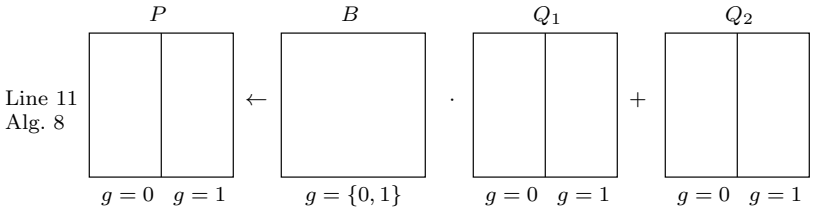

Figure 4: Matrix multiplication by 2 GPUs.

and the GPUs both have the powers 1 to $b+1$ of X computed and stored. Arguments A and B are actually factors previously stored into the device memory. Argument i of Alg. 8 is used for indexing array $\bar{\alpha}$ and also for controlling the recursion. The returning matrix P, which contains the polynomial solution, is a local matrix to each GPU of $n/2$ columns (or $n/2+1$ in the case of GPU1 if n is odd). Also, auxiliary matrices Q_1 and Q_2 share the size with P. They are used to perform the product in line 11 which corresponds to the operation shown in the recursion (7). Figure 4 graphically depicts the setup of this product between both GPUs. All the evaluation process is carried out in parallel between the two GPUs without communication. Only upon termination the CPU system receives factors P from both GPUs to build the final square matrix that is the solution to (1). This is carried out after the execution of Alg. 8.

6 Results of the evaluation of matrix polynomials using boxing

Next in this section we show different aspects of the behavior of the algorithm with boxing. Firstly, Fig. 5 plots the evolution of time regarding the boxing factor b. Based on the experiments our algorithm checks that value $b = \lfloor\sqrt{d+1}\rfloor - 1$ obtained through the theoretical derivation is the best to be chosen if we use one GPU. We get into the inside of the behaviour of the two GPUs to see if the computation of matrix products could yield a slightly different result. The time to obtain the resulting matrix P is split into two steps: the computation of the powers of matrix X (terms of polynomial $Q^b(\bar{\alpha})$ (4)) and the evaluation of the polynomial using the recursive algorithm BOXING2 Alg. 8. The computation of the powers of X increases linearly with the boxing factor b and is exactly the same for any polynomial degree since it only depends on b. The amount of work to be done to evaluate the polynomial through Alg. 8 decreases as long b increases since the number of terms P_i in recursion (7) decreases. The best boxing factor is around the crossing of these lines as Fig. 6 shows. Figure 7, which shows the total time for evaluating the polynomial on two GPUs, has the same shape as its counterpart Fig. 5 on one GPU.

The next experiment shows the speed up of the algorithm using *boxing*. Fig. 8 shows the increment in speed achieved by the use of two GPUs and how this improvement grows with the problem size thanks to the parallelization of the matrix multiplication depicted in Fig. 4. The degree of the polynomial does not involve big a difference in the speed up due to the small weight of communications (CPU-GPU) with regard to the weight of computations.

7 The Matlab environment

We have developed a Matlab interface to make easy access our routines written in CUDA. The algorithms implemented are only those using boxing. We provide a version written directly as a Matlab function (`polmat_ps`) that solves (1) using Alg. 8 and returns the solution matrix P. Sometimes, it is more efficient to use a program written in C that is called from the Matlab interface. This is because Matlab executes instructions provided by the user in the Matlab function as they are read runtime. In the case of many fast instructions, there exits an overhead associated to this interpretation. However, this is not the case due to our algorithm is rich high CPU demanding instructions like matrix multiplications. Matlab uses the Intel MKL library to compute the matrix multiplications associated to the evaluation of the polynomials.

The versions provided that use the GPUs have been implemented using MEX files [17], with C, OpenMP and CUDA for NVIDIA cards [18, 19]. MEX files are external interface functions that allow to use calls to custom C, C++ or Fortran routines directly from Matlab as if they were Matlab bulit-in functions. The complexity of the process lies on the programmer's side and the benefit is for the user of the Matlab routines. MathWorks provides other ways of programming

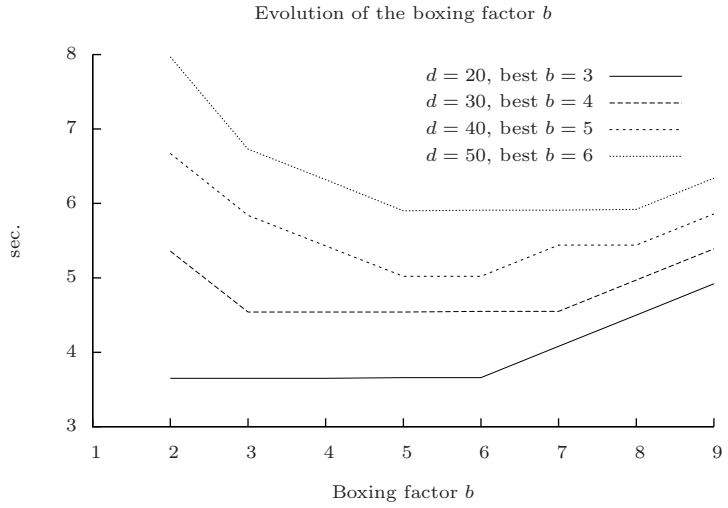

Figure 5: Time for computation the evaluation of a polynomial with matrix size $n = 6000$ varying the polynomial degree in function of the boxing factor on one GPU.

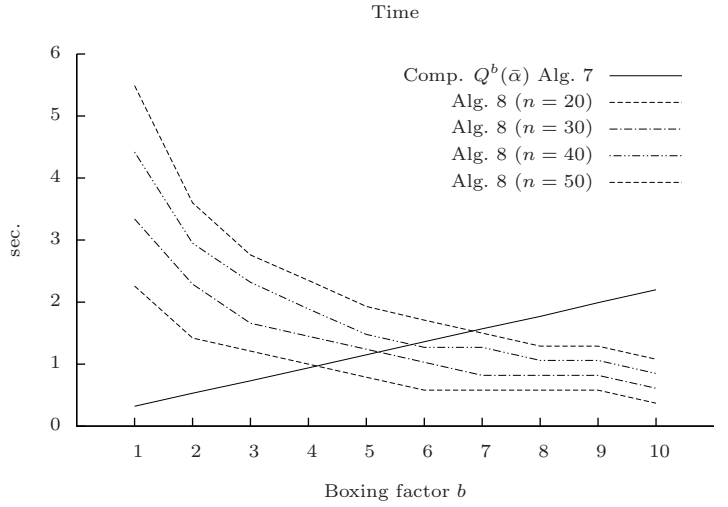

Figure 6: Time for computation of powers of X and evaluation of a polynomial with matrix size $n = 6000$ varying the polynomial degree in function of the boxing factor on two GPUs.

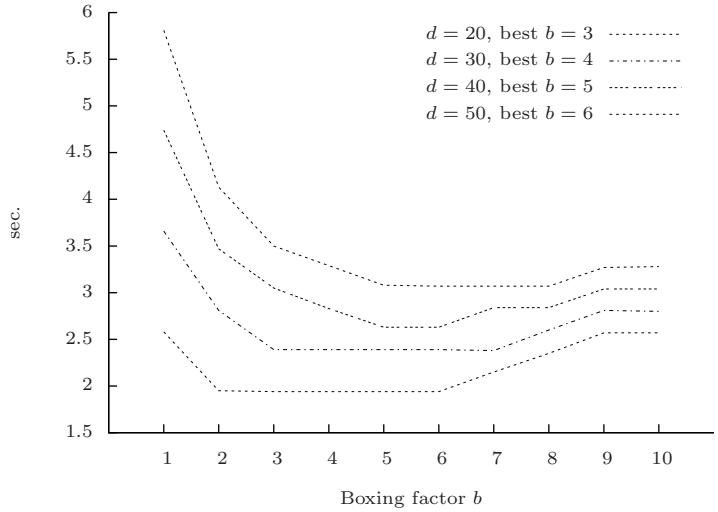

Figure 7: Total time for the evaluation of a polynomial with matrix size $n = 6000$ varying the polynomial degree in function of the boxing factor on two GPUs.

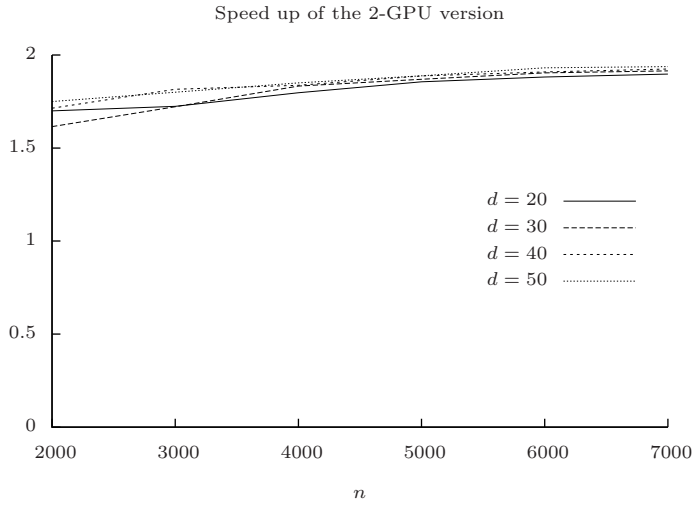

Figure 8: Speed up of the evaluation of polynomial (1) on two GPUs with regard to one GPU for different degrees and matrix sizes.

	Matrix size	2048		4096		6144	
		Ts	Gflops	Ts	Gflops	Ts	Gflops
$d = 12$ $b = 3$	CPU Quad i7	0.99	87	6.83	101	25.1	92
	1 GPU K 20	0.12	716	0.78	881	2.49	931
	2 GPU K 20	0.08	1074	0.46	1494	1.37	1693
$d = 20$ $b = 4$	CPU Quad i7	1.55	78	10.4	101	35.4	92
	1 GPU K 20	0.16	752	1.06	881	3.42	949
	2 GPU K 20	0.10	1203	0.60	1604	1.83	1774
$d = 30$ $b = 5$	CPU Quad i7	2.59	57	12.5	99	45.8	91
	1 GPU K 20	0.20	773	1.35	916	4.37	935
	2 GPU K 20	0.12	1289	0.75	1649	2.31	1807

Table 1: Performance in seconds (Ts) and Gigaflops (Gflops) for the evaluation of polynomials with different degrees (d), boxing factors (b), and matrix sizes.

from Matlab for GPUs [20], but programming in C lets also to have codes for linux commands. Implementing these codes is not trivial so we got some help from [21, 16] to build routine `cum_polmat_ps` that uses the GPUs. Prefix `cum` stands for CUDA multicard. The routine has an argument to specify the number of GPUs to use (although not yet implemented, the interface is ready for more than two devices).

As an example of the high computational capacity of the programs developed in this paper, we show some results in Table 1. One of the most significant aspects is that these results were obtained using the friendly interface Matlab. The speed up achieved for a polynomial degree $d = 12$ with two devices regarding only one is of 1.5, 1.7, and 1.82 for each one of the three matrix sizes, respectively. This speed up slightly increases with both the matrix size and the polynomial degree. The maximum speed up reached is of 1.89 for a polynomial degree of $d = 30$ and a matrix size $n = 6144$. Communications through the PCI bus are more and more hidden under the increasing computation demand when scaling the problem size. These figures rely on two important facts. The speed up increases more slightly with the matrix size than with the polynomial degree. Really, it is boxing factor b what affects this magnitude more than d. The other fact is that the matrix size influences more in the speed up, but also increases the communication time.

8 Conclusions

We have studied the problem of evaluating matrix polynomials in modern compute-intensive computers. More and more frequently, these computers count on some kind of accelerator device attached to them. Very commonly used accelerators are the NVIDIA cards which, although designed for graphics processing,

allow to increase the performance of many general purpose applications. Experimental results in this work have been achieved on this type of device, but the developed algorithms here can be easily migrated to other types of accelerators, like AMD GPUs, but also to the new Intel hardware proposals based on "knights" and "landing" corner architectures. The basic requirement is the existence of an efficient routine for matrix multiplication on these environments. The rest of the development, i.e., algorithms, Matlab MEX files, etc. would remain unchanged.

Our proposal consists of algorithms but also of a method to implement these algorithms that is quite effective. This method is based on writing the code for one GPU. This code will be executed by multiple GPUs. Two threads will be spawned through an OpenMP `parallel for`, each one bound to a different GPU. The compiler used must provide an implementation of OpenMP with a V3.0 specification at least to work properly.

We have used different software resources to implement our application: MKL and CUBLAS libraries, OpenMP, Matlab MEX files, CUDA language, so that all together provide the future user with an easy to use tool to solve more sophisticated problems. Furthermore, as shown by the results of the last section, it can be said that this tool uses the multiGPU environment efficiently.

References

[1] N. J. Higham, Functions of Matrices: Theory and Computation, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[2] M. Hochbruck, C. Lubich, H. Selhofer, Exponential integrators for large systems of differential equations, The SIAM Journal on Scientific Computing 19 (5) (1998) 1552–1574.

[3] S. M. Cox, P. C. Matthews, Exponential time differencing for stiff systems, Journal of Computational Physics, Elsevier 176 (2002) 430–455.

[4] A.-K. Kassam, L. N. Trefethen, Fourth-order time-stepping for stiff PDEs, The SIAM Journal on Scientific Computing 26 (4) (2005) 1214–1233.

[5] J. Sastre, J. J. Ibáñez, E. Defez, P. A. Ruiz, Efficient orthogonal matrix polynomial based method for computing matrix exponential, Applied Mathematics and Computation 217 (2011) 6451–6463.

[6] J. Sastre, J. J. Ibáñez, E. Defez, P. A. Ruiz, Accurate matrix exponential computation to solve coupled differential, Mathematical and Computer Modelling 54 (2011) 1835–1840.

[7] J. Sastre, J. J. Ibáñez, E. Defez, P. A. Ruiz, Efficient scaling-squaring Taylor method for computing matrix exponential, SIAM J. on Scientific Comput.(Accepted with modifications).

[8] E. Defez, J. Sastre, J. Ibáñez, P. Ruiz, Computing matrix functions arising in engineering models with orthogonal matrix polynomials, Mathematical and Computer Modelling 57 (2011) 1738–1743.

[9] M. S. Paterson, L. J. Stockmeyer, On the number of nonscalar multiplications necessary to evaluate polynomials, SIAM J. Comput. 2 (1) (1973) 60–66.

[10] E. Weisstein, CRC Concise Encyclopedia of Mathematics, Second Edition, Taylor & Francis, 2002.

[11] NVIDIA, Tesla GPU high performance computing for servers, cited on July 2014 (2013).
URL http://www.nvidia.com/object/tesla-servers.html

[12] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen, LAPACK Users' Guide, SIAM, 1992.

[13] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture, cited on July 2014 (2009).
URL https://developer.nvidia.com/about-cuda

[14] NVIDIA, CUDA. CUBLAS library, cited on July 2014 (2009).
URL http://docs.nvidia.com/cuda/cublas

[15] OpenMP Architecture Review Board, The OpenMP® API Specification for Parallel Programming (2014).
URL http://www.openmp.org

[16] J. Peinado, J. Ibáñez, V. Hernández, E. Arias, Speeding up solving of differential matrix Riccati equations using GPGPU computing and MATLAB, Concurrency and Computation: Practice and Experience 24 (2012), Issue 12 (2012) 1334–1348.

[17] MathWorks, MATLAB MEX Files.
URL http://www.mathworks.com/support/tech-notes/1600/1605.shtml#intro

[18] NVIDIA, NVIDIA MATLAB plugin.
URL http://developer.nvidia.com/object/matlab_cuda.html

[19] NVIDIA, Accelerating MATLAB with CUDA Using MEX Files, White Paper WP03495-001-V01 (2007).

[20] MathWorks, MATLAB, Parallel Computing 5 user's guide.

[21] B. Dushaw, Matlab and CUDA.
URL http://faculty.washington.edu/dushaw/epubs/Matlab_CUDA_Tutorial_2_10.pdf