J Supercomput
DOI 10.1007/s11227-016-1694-y

# Automatic tuning to performance modelling of matrix polynomials on multicore and multi-GPU systems

**Murilo Boratto[1] · Pedro Alonso[2] · Domingo Giménez[3] · Alexey Lastovetsky[4]**



**Abstract** Automatic tuning methodologies have been used in the design of routines in recent years. The goal of these methodologies is to develop routines which automatically adapt to the conditions of the underlying computational system so that efficient executions are obtained independently of the end-user experience. This paper aims to explore programming routines that can automatically be adapted to the computational system conditions thanks to these automatic tuning methodologies. In particular, we have worked on the evaluation of matrix polynomials on multicore and multi-GPU systems as a target application. This application is very useful for the computation of matrix functions like the sine or cosine but, at the same time, the application is very time consuming since the basic computational kernel, which is the matrix multiplication, is carried out many times. The use of all available resources within a node in an easy and efficient way is crucial for the end user.

✉ Murilo Boratto
muriloboratto@gmail.com

Pedro Alonso
palonso@upv.es

Domingo Giménez
domingo@um.es

Alexey Lastovetsky
alexey.lastovetsky@ucd.ie

1 Núcleo de Arquitetura de Computadores e Sistemas Operacionais, Universidade do Estado da Bahia, Salvador, Bahia, Brazil

2 Departament de Sistemas Informàtics i Computació, Universitat Politècnica de València, Valencia, Spain

3 Departamento de Sistemas Informáticos, Universidad de Murcia, Murcia, Spain

4 Heterogeneous Computing Laboratory, School of Computer Science, University College Dublin, Dublin, Ireland

Published online: 28 March 2016

**Keywords** Automatic tuning · Matrix polynomials · Performance · Multicore · Multi-GPU

## 1 Introduction

It is up to software developers to create superscalar programs [7], since it is highly probable that a program written today will need to be totally modified tomorrow to adapt to machines with more and/or different processors. To meet this need, manipulation patterns were created which allow modifications in existing codes by incorporating directives. This means users do not need to radically update their applications to enjoy the benefits of multiprocessing environments.

There exist many approaches to *automatic tuning* focused on getting a model for the execution time of the routine to optimize. Once the model has been obtained, theoretically and/or experimentally, for a given problem size and execution environment, this model is used to obtain the values of some adjustable parameters with which minimize the execution time [9].

We use in this work a known technique to build the model consisting in the execution of small benchmarks of the routine. In this work, a technique for redesigning the model from regression has been included in our original methodology [1]. The basic idea is to start from the theoretical model using the information from basic routines to model the higher level ones without experimenting with them. However, if for a particular routine all this information is not useful enough, then it would be built again from the beginning using a series of experimental executions and regression applied appropriately.

The main contribution consists of a mathematical model to predict the execution time of high demanding applications on heterogeneous systems which consists of a general purpose multicore CPU with two equal or different GPUs attached. The model is simple, can be built in a reasonable time at the installation stage, and is accurate enough as to get the best workload distribution among the available computing resources. The rest of the paper is organized as follows: Sect. 2 presents the related work. Section 3 shows the fast parallel algorithm for solving matrix polynomials on multicore and multi-GPU. Section 4 explains the design of the automatic tuning methodology. Section 5 presents the experimental results. The conclusions section closes the paper.

## 2 Related work

There exist important automatic tuning systems that attempt to automatically adapt the software to tune to the conditions of the execution platform. These include, e.g. the FFTW package developed for the computation of discrete Fourier transform [8], ATLAS for BLAS [19], a library of linear algebra routines for sparse matrices [12], etc. The main goal of any automatic tuning system is to minimize the execution time of the routine to tune, keeping in turn the installation time below a reasonable threshold. In addition, the existence of automatically tuned software makes easy the efficient utilization of the library routines by non-expert users.

The approach chosen, e.g. by FAST [5], consists of a large benchmark followed by a polynomial regression to find the optimal parameters for different routines. Polynomial regression is used in [18] to decide which is the most appropriate version among the different variants of a routine. The authors of the same work also introduce a black-box running method to reduce the enormous implementation space. In the approach followed by FIBER [13] the execution time of a routine is approximated by fixing one parameter and varying the other one. In this case, a set of polynomial functions of grades 1–5 is generated and the best one of them all is selected. The values provided by these functions for different problem sizes are then used to generate another function where the second parameter is fixed now and the first one is varied. The work in [17] introduces a new method, named *Incremental Performance Parameter Estimation*. In this method, the estimation of the theoretical model by polynomial regression is started from the least sampling points and incremented dynamically to improve accuracy. Initially, they apply it on sequential platforms and with just one algorithmic parameter to seek. In [11] the number of sampling points is reduced by starting from a previous shape of the curve that represents the execution time.

The current hardware trends have inevitably brought the need for updates on existing legacy software packages, such as BLAS [4] and LAPACK [3]. This is reflected, for instance, in the *Parallel Linear Algebra Software for Multicore Architectures* (PLASMA) project [16], and the *Matrix Algebra on GPU and Multicore Architectures* (MAGMA) project [14], which is a recent effort on developing a LAPACK version for multicore and heterogeneous/hybrid architectures containing hardware accelerators like GPUs. The goal of this work falls within the context of all these above mentioned packages which try to build the best routine through the selection of some critical parameters at installation time, and is carried out on the evaluation of matrix polynomials.

## 3 Fast parallel algorithm for solving matrix polynomials on multicore and multi-GPU

The matrix polynomial is a simple algebraic structure that represents a real problem applied in the area of engineering and physics. We define a matrix polynomial $P$ of degree $d$ as

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

where $X, I \in \mathcal{R}^{n \times n}$, being $I$ the identity matrix [2]. We also define the array $\bar{\alpha}$ as $\bar{\alpha} = [\alpha_i]_{i=0,\ldots,d}$ for convenience in further descriptions.

There exists a sequential technique that allows to reduce the number of computations (number of matrix products) needed to evaluate a polynomial. This technique is based on the method that Paterson and Stockmeyer designed for scalar polynomials [15]. From now on, and for the sake of clarity, we will denote this method as *boxing*. The next example, where the degree of the polynomial to evaluate is $d = 14$, easily shows the idea behind this method.

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

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

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

and polynomial (1) can be written as

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

The example above uses a *boxing* size of $b = 3$, which means that the largest polynomial in the previous expression can not be larger than $b = 3$. The *boxing* size $b$ also means that the power $b+1$ of matrix $X$ ($X^4$ in the example) is used as the common factor. It is easy to see that the number of operations needed to evaluate polynomial (2) is lower than that needed for evaluating the original polynomial. Function EVALUATE of Algorithm 1 summarizes the overall process. In lines 2–5, an array of matrices $A$ is filled with the $b + 1$ matrix powers of $X$ needed for the evaluation of the polynomial so that

$$
A = [X^{i+1}]_{i=0,\ldots,b} = \left(X\ X^2\ \ldots\ X^b\ X^{b+1}\right). \quad (3)
$$

Array $A$ will be used to evaluate the "boxed" polynomials of type $Q^q(\bar{\alpha}, X)$. Once array $A$ has been computed, routine EVALUATE calls the routine BOXING, which is a recursive routine that allows to evaluate the polynomial using *boxing*.

Routine BOXING is based on the following recurrence

$$
P_i \leftarrow X^{b+1} P_{b+i+1} + Q^b(\bar{\alpha}_{b+i:i}) = X^{b+1} Q_1 + Q_2, \quad (4)
$$

for $i = 0, b+1, 2(b+1), 3(b+1), \ldots$, where matrix $P_i$ represents the general case. The base case of this recursion is met when $d - i \leq b$, which means that there is no longer the 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 the recursion (4) applied to the polynomial example in (2) results in

**Algorithm 1** Algorithm for the evaluation of the matrix polynomial using *boxing*

```
1: function EVALUATE( n, X, d, ᾱ, b ) return P
2:     A(0) = X
3:     for i ← 1, b do
4:         A(i) ← A(i − 1) · X
5:     end for
6:     P ← BOXING( n, d, b, 0, ᾱ, A )
7: end function
8: function BOXING( n, d, b, i, ᾱ, A ) return P
9:     q ← d − i
10:    if q ≤ b then
11:        P ← EVAL( n, q, ᾱ_{q+i:i}, A )
12:    else
13:        Q_1 ← BOXING( n, d, b, b + i, ᾱ, A )
14:        q ← b − 1
15:        Q_2 ← EVAL( n, q, ᾱ_{q+i:i}, A )
16:        P ← A(q) · Q_1 + Q_2
17:    end if
18: end function
```

**Algorithm 2** Recursive algorithm to evaluate a matrix polynomial using *boxing* on CPU cores and multi-GPU

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

$$
\begin{aligned}
P = P_0 &= X^4 P_4 + Q^3(\bar{\alpha}_{3:0}) \\
P_4 &= X^4 P_8 + Q^3(\bar{\alpha}_{7:4}) \\
P_8 &= X^4 P_{12} + Q^3(\bar{\alpha}_{11:8}) \\
P_{12} &= Q^2(\bar{\alpha}_{14:12}).
\end{aligned}
$$

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

The algorithm to evaluate a matrix polynomial using *boxing* on CPU cores and multi-GPU is written in Algorithm 2. The method proposed is based on OpenMP parallel loops [6]. Each iteration of the parallel loop is carried out by one thread which is, in turn, bound to a given GPU and/or CPU core, from 0 to $D$ devices. Before calling this routine, the powers of $X$ are assumed to be already computed and stored in array $A$ (3). All the components of this array (which are matrices) are partitioned in

blocks of columns and distributed among the GPUs and the CPU cores accordingly to their computing capability. This way, each thread executing the iteration of the loop performs the computation of a different set of data, those part of the matrix stored on this component in previous steps. The computation of the powers of $X$ is by far the most costly step and is distributed among the different devices of the computer. Only the first ($A(0)$) and the last ($A(b)$, represented by $B$ in the algorithm) components are needed to be fully stored in each device.

All the evaluation process is carried out in parallel between the devices without communication. Only upon termination the CPU system receives factors $P$ from both GPUs and the CPU cores to build the final square matrix.

## 4 Automatic tuning methodology

The automatic tuning methodology uses a theoretical model of the execution time of the routine which is used to select the suitable values of some parameters that will allow to get the result in the shortest possible time. We follow here the automatic tuning methodology presented in [1]. The model proposed must reflect, on the one hand, the computing and communication features of the algorithm, known as $AP$ (Algorithm Parameters), and, on the other hand, the features of the system under which the algorithm is executed, known as $SP$ (System Parameters). The mathematical model of the execution time ($t$) can be expressed as a function of the input size ($s$), which is in turn a function of the $AP$ and $SP$, i.e. $t(s) = f(s, AP, SP)$. The value of all these parameters should be selected to obtain a reduced execution time.

Typical $SP$ are: the cost of one arithmetic operation, communication start-up and word-sending times in communication operations. These parameters represent the characteristics of the computer and the communication system between CPU and GPU. Typical $AP$ are: the number of processors to use, the number of processes to be enabled along with their mapping in the physical system, or the size of communication blocks, or data partitioning and distribution among processes. To obtain a more realistic model, we can consider that the values for $SP$ are influenced by those for $AP$, i.e. $SP$ can be expressed as a function ($h$) of the input size ($s$) and the $AP$ so that $SP = h(s, AP)$.

The values for the $SP$ will be obtained at the moment of installing the routine in a new system. To this end, the routine designer should develop the runtime model, identify the $SP$ in the model, and design an installation strategy that includes, for each $SP$, the experiments to estimate its values, the $AP$ and the values that have to be experimented with. The values obtained for the $SP$ are included with the execution time model in the routine that is being optimized, which is, thus, installed with information of the system for which it is being optimized.

The $AP$ for the automatic tuning scheme presented in this paper are: the number of CPU cores ($c$), and the percentage of computation assigned to GPUs, defined as *workload* ($w$). These two parameters capture the key characteristics of the application performance and the machine that the application is running on. The CPU cores and workload parameters are used to show the scalability of the application. They reflect the reasons why the application performance becomes unscalable beyond a certain point. Thus, in our problem, the set of algorithmic parameters is

$AP = \{c, w\}$, so the total execution time can be written as $t(s) = f(s, AP, SP) = f(s, c, w, h(s, c, w))$.

The optimum number of CPU cores and workload are not constant but depend on the platform and on the problem size. Thus, a good selection of the values of the algorithm parameters is important, and the development of automatic tuning software makes the efficient utilization of the routines by non-expert users easy. The algorithm is studied theoretically and experimentally to determine the influence of different values for $SP$ on the $AP$. The most important part of the information system to be incorporated to the routine is the analytical execution model for the time as a function of the problem size ($s$), $SP$, and $AP$. We propose the following model for the execution time:

$$t(s, c, w) = t_{\text{rt}} + \text{CPU}_{\text{mt}} + \text{GPU}_{\text{mt}}, \tag{5}$$

where $t_{\text{rt}}$ is the routine execution time, and $\text{CPU}_{\text{mt}}$ and $\text{GPU}_{\text{mt}}$ are the CPU and GPU management times, respectively. This analytical model predicts the execution time as a function of the input data features, and requires direct information about the algorithm used and the underlying architecture.

The model for the execution time (Eq. 5) is detailed in the following one:

$$t(s, c, w) = \frac{k \cdot O(s)}{c + g \cdot S} + t_c \cdot c + t_g \cdot g. \tag{6}$$

Let $S$ be the relative speedup of a GPU versus a CPU core, then the model estimates the execution time starting from the sequential complexity cost $t_{\text{sec}}(s) = k \cdot O(s)$ divided by the number of the CPU cores ($c$) plus the number of GPUs ($g$) multiplied by $S$. Constant $k$ represents the cost of one arithmetic operation. To obtain this computation time it is necessary to select $w$ properly so that the workload is well balanced between the CPU and the GPUs. The model for the execution time depicted in Eq. 6 also involves the management time for both subsystems. These times depend on the initialization time of a CPU thread ($t_c$) and a GPU kernel ($t_g$), respectively, and are proportional to the number of cores ($c$) and the number of GPUs ($g$), respectively. The management of CPU cores are negligible in small systems (like those where the experiments have been carried out), but the management of GPU devices is significant since it includes the cost of data transference between CPU and GPU. Table 1 summarizes the meaning of each term used in our model (Eq. 6).

The performance parameters involved in the model are key factors that the automatic tuning methodology should consider for high performance applications. During the installation, we carry out experiments with different combinations of the problem size $s$, the number of cores $c$, and the workload $w$, seeking to minimize the time difference between the theoretical prediction and the experimental values obtained by using a least squares regression. By varying $s$ and $w$ we obtain a different optimum value for the number of cores $c$ in each execution environment. Constant $k$ is also estimated by using least squares. Once the execution time has been estimated the possible values of $c$ and $w$ are substituted into the formula and the routine is recompiled. The final values of the parameters are included in the routine together with a decision engine that gives the user the theoretical time to solve the problem. This methodology,

**Table 1** Description of the parameters in the execution time model

| Symbol | Description |
|---|---|
| $k$ | Computational parameter for each execution system |
| $d$ | Order of the polynomial. In the experiments it ranges from 4 to 20 |
| $s$ | The length of the matrix. In the experiments it ranges from 4000 to 8000 |
| $c$ | Number of CPU cores |
| $w$ | GPU workload |
| $g$ | Number of GPUs |
| $t_c$ | Cost of initialization of a thread in CPU |
| $t_g$ | Cost of initialization of a kernel in GPU |
| $S$ | Relative speedup of a GPU on one core in the CPU |

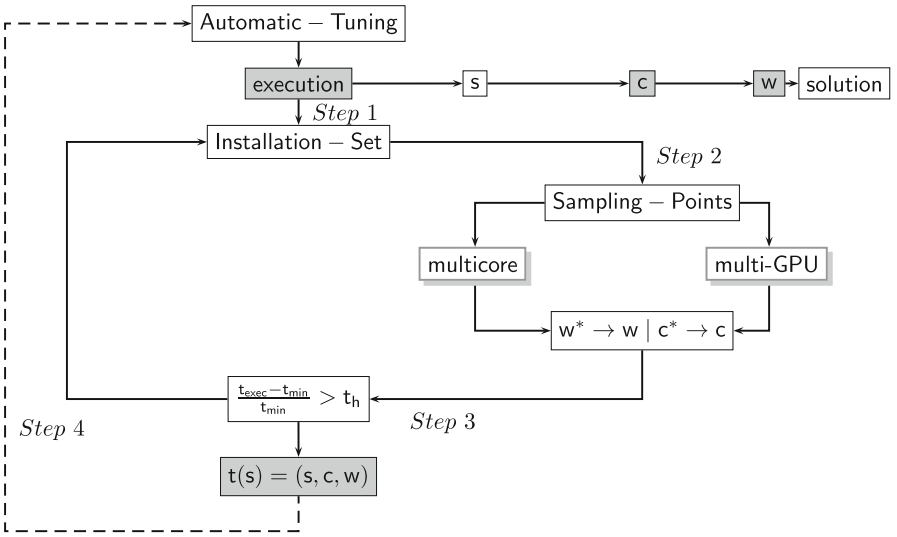


**Fig. 1** General scheme of automatic tuning methodology

which is essentially an iterative process, is likely to require many experiments to have a good parameter estimation. This is why we also pursue simplicity in our model by omitting some architectural aspects that lose impact in the execution time as the problem size increases. Either way, the model proves to be sufficient and useful for the target application.

The general scheme of the automatic tuning methodology is shown in Fig. 1. The installation of the *automatic routine tuning* in the system is done executing the routine for each matrix size specified in an *Installation-Set*, and varying the number of cores $c$ and the workload $w$ between the multicore and the multi-GPU subsystems. The performance parameter estimation the procedure is applied to get the optimal values:

**Step 1:** Choose range from values of the performance parameters in *Installation-Set*,

**Step** 2**:** Run the routine to obtain execution time at each sampling point,
**Step** 3**:** Search a minimum value point of the function which corresponds to the optimal value of the performance parameters,
**Step** 4**:** Define the execution time model and fit it to the executed values.

The *Sampling-Points* contains significant values, from small to large, so the installation gives satisfactory results for a wide range of problem sizes. Our model is based on the guided-search [10], where the search is made in many directions, and it finishes for each problem size when the execution time exceeds the minimum for that size by an amount greater than the threshold. The technique uses a good heuristic to direct the search in an enormous search space, since the number of solutions increases with the number of test variables. The optimal performance parameters are estimated using a previously variable number of sampling points. When the number of sampling points is large, the accuracy for estimation is high, but the computational efficiency is low, and when the number of fixed sampling points is small, vice versa.

## 5 Experimental results

This section presents the experiments with detailed explanation and useful insights. We used the following two execution environments:

[**System** 1] Execution environment with 2 **identical** GPUs. Comprises two Intel Xeon at 2.26 GHz and 24 GB DDR3 main memory. Each one is a quadcore processor with 12 MB of cache memory. It contains two GPUs NVIDIA Tesla C2050 with 14 stream multiprocessors (SM) and 32 stream processors (SP) each (448 cores in total).
[**System** 2] Execution Environment with 2 **different** GPUs. Comprises two Intel Xeon at 2.93 GHz and 86 GB DDR3 main memory. Each one is a quadcore processor with 12 MB of cache memory. It contains two GPUs: the first one is a NVIDIA Tesla K20 with 28 stream multiprocessors (SM) and 64 stream processors (SP) each (2496 cores in total); the second one is a NVIDIA Tesla C2050 with 14 stream multiprocessors (SM) and 32 stream processors (SP) each (448 cores in total).

In our experiments we use a parallel implemented version of Algorithm 2 using OpenMP and CUDA for the evaluation of matrix polynomials in heterogeneous environments. Many parameter values were used at installation time to estimate the best values for the $AP$. The available range for the CPU cores ($c$) is $1, 2, \ldots, 16$ in both systems (Intel Hyper-Threading is set on in both systems). Then, we checked for GPUs workloads from 10 to 45 %. The input sizes of the problem ($s$) for the experiments were $4, 5, \ldots, 20$. Table 2 shows the parameters used at installation time to estimate the values of $AP$ for the two environments (System 1 and System 2).

There are two important observations: (1) the $c$ values depend on the problem size in the system under test, and (2) for each problem size and for different values of $w$ we obtain a different optimum value for $c$ on each execution environment. Be aware of that this variability is essential to make good decisions in the later selection of the

**Table 2** Execution time (s) obtained at installation time with different values for the performance parameters

| System 1 | $w = 45, 45, 10$ | | $w = 40, 40, 20$ | | $w = 35, 35, 30$ | | $w = 30, 30, 40$ | |
|---|---|---|---|---|---|---|---|---|
| $s$ | $c$ | $t(s, c, w)$ | $c$ | $t(s, c, w)$ | $c$ | $t(s, c, w)$ | $c$ | $t(s, c, w)$ |
| 5000 | **16** | **5.93** | 14 | 7.56 | 12 | 13.47 | 12 | 22.82 |
| 6000 | **16** | **9.34** | 14 | 12.24 | 12 | 21.42 | 14 | 38.33 |
| 7000 | **16** | **14.62** | 16 | 19.54 | 14 | 32.94 | 16 | 60.03 |
| 8000 | **16** | **20.59** | 16 | 27.93 | 14 | 48.01 | 16 | 89.42 |
| System 2 | $w = 50, 30, 20$ | | $w = 45, 40, 15$ | | $w = 50, 35, 15$ | | $w = 55, 35, 10$ | |
| $s$ | $c$ | $t(s, c, w)$ | $c$ | $t(s, c, w)$ | $c$ | $t(s, c, w)$ | $c$ | $t(s, c, w)$ |
| 5000 | 12 | 5.13 | 16 | 4.90 | 14 | 4.85 | **16** | **4.44** |
| 6000 | 14 | 8.44 | 16 | 7.99 | 14 | 7.53 | **16** | **7.42** |
| 7000 | 16 | 14.04 | 16 | 12.61 | 14 | 11.75 | **16** | **11.37** |
| 8000 | 16 | 19.32 | 16 | 18.07 | 16 | 16.63 | **16** | **16.11** |

Best values marked in boldface

optimum $AP$ parameters. The $AP$ obtained after the automatic tuning process in the environments used are:

**System** 1(Using 2 identical GPUs + 2 Processors Quadcore)
Number of CPU cores ($c$) = 16
Workload ($w$) = (GPU, GPU, CPU) = (45 %, 45 %, 10 %)
**System** 2(Using 2 different GPUs + 2 Processors Quadcore)
Number of CPU cores ($c$) = 16
Workload ($w$) = (GPU, GPU, CPU) = (55 %, 35 %, 10 %)

The installation time spent on both platforms was the same (around 160 minutes). We did experiments with different combinations of $s$, $c$, and $w$, considering the small size problem to obtain the model on a given platform.

We show in both plots of Fig. 2 the time and the speedup, respectively, for the evaluation of matrix polynomials with different degrees ranging from 4 to 20 with a polynomial matrix of size 10,000 in System 1. The execution was carried out on each subsystem independently (CPU, 1 GPU, 2 GPUs) to have a measure for comparison purposes. The speedup has been obtained with regard to the use of the CPU subsystem only. Both plots show how the use of GPUs in our system clearly outperforms the computation on the CPU. The sawtooth shape of the graphs in Fig. 2 is due to the unbalanced workload for degrees of the polynomial which are odd.

The results shown in Fig. 3a, that were carried out on System 2, show the theoretically execution time according to the performance model (Eq. 6) and the experimental time for a matrix size of 10000 with regard to the polynomial degree. We consider that the difference between the two plots is low accounting for the simplicity of the model and the low installation time used to figure out the performance parameters. Figure 3b shows the speedup achieved when varying both the matrix size and the polynomial degree. These numbers are interesting since they allow to check that the

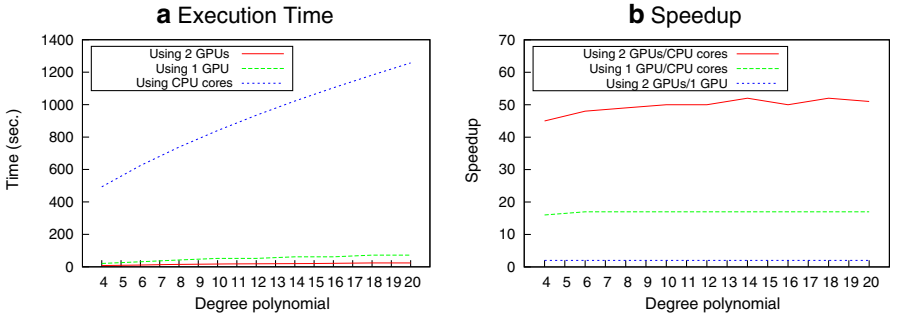


**Fig. 2** Evaluation of matrix polynomials with a matrix size of 10, 000 with regard to the polynomial degree on System 1. **a** Executing time. **b** Speedup

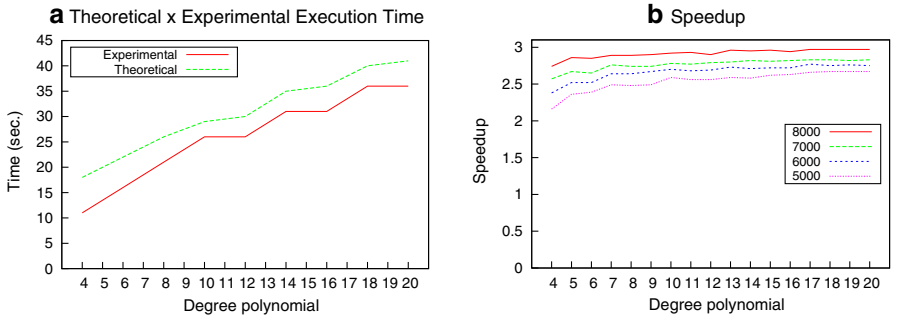


**Fig. 3** **a** Experimental versus theoretical execution time for evaluating matrix polynomials of size 10,000 with regard to the polynomial degree (System 2). **b** Speedup for the evaluation of matrix polynomials of sizes 5000, 6000, 7000, 8000 with regard to the polynomial degree (System 2)

speedup grows more with the matrix size than with the polynomial degree. This is because the matrices and, consequently, the multiplication of matrices have all been partitioned in pieces and have been scattered among the different subsystems of the heterogeneous environment. The smooth 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 show different aspects of the behavior of the algorithm with different workloads. First, Fig. 4a plots the evolution of time regarding identical GPUs. Based on the experiments it can be seen that the value $w$ obtained through the theoretical derivation is the best to be chosen if we use 2 GPUs and all the CPU cores. This value for the workload is $(w)$ = (GPU, GPU, CPU) = (45 %, 45 %, 10 %) for System 1. Second, we show in the next experiment the execution time of the algorithm using different GPUs. Figure 4b shows the reduction in time achieved by the use of 2 GPUs and the CPU cores, and how this improvement grows with the problem size thanks to the parallelization of the matrix multiplication. As expected, the degree of the polynomial does not involve a big difference for different workloads due to the small weight of communications (CPU–GPU) with regard to the weight of computations. The best behavior is around $(w)$ = (GPU, GPU, CPU) = (55 %, 35 %, 10 %).

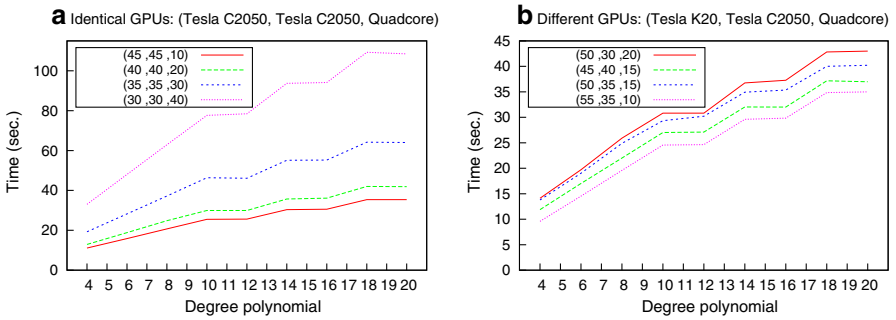


**Fig. 4** **a** Time for the evaluation of matrix polynomials of size 10000 with regard to the polynomial degree on identical GPUs (System 1). **b** Time for the evaluation of matrix polynomials of size 10000 with regard to the polynomial degree on different GPUs (System 2)

## 6 Conclusions

Inspired in the Paterson-Stockmeyer technique for computing polynomials, we proposed in this work a recursive algorithm and an efficient implementation for the evaluation of matrix polynomials in parallel. Also, we propose an automatic tuning methodology to easily adapt existing parallel algorithms that can execute efficiently on heterogeneous computers, i.e. computers featuring one or more *hardware accelerator(s)*.

Automatic tuning techniques must be established based on a good understanding of the target architecture, and an efficient algorithm implementation that exposes critical and relevant properties of a program performance for that architecture. Regarding the experiment data on the automatic tuning explains why the application achieves the best performance under a certain parameter setting, but not under other settings.

The experimental results indicate that our approach is efficient and scalable and the routines to solve this problem can incorporate an automatic tuning engine to obtain execution times close to the optimum without user intervention. The use of modelling techniques can contribute to improve the decisions taken to reduce the execution time of the routines. The modelling allows us to introduce information about the behavior of the routine in the automatic tuning process, guiding this process. It is necessary that the modelling time is small because at least part of this process could be carried out in each installation of the routines. Therefore, different ways of reducing it have been studied here, and the results have been satisfactory.

**Acknowledgments** This work has been partially supported by Generalitat Valenciana under Grant PROMETEOII/2014/003, and by the Spanish MINECO, as well as European Commission FEDER funds, under Grant TEC2015-67387-C4-1-R and TIN2015-66972-C5-3-R, and network CAPAP-H. Also, we have work in cooperation with the EU-COST Programme Action IC1305, "Network for Sustainable Ultrascale Computing (NESUS)".

## References

1. Alberti PV, Alonso P, Vidal AM, Cuenca J, Giménez D (2004) Designing polylibraries to speed up linear algebra computations. IJHPCN 1(1/2/3):75–84

2. Alonso P, Boratto M, Pinilla J, Ibañez J, Martinez J (2014) On the evaluation of matrix polynomials using several GPGPUs. Tech Rep Riunet/E10251/39615
3. Anderson E, Bai Z, Bischof C, Demmel J, Dongarra J, Croz JD, Greenbaum A, Hammarling S, McKenney A, Ostrouchov S, Sorensen D (2013) LAPACK users guide, 2nd edn. SIAM, Philadelphia
4. Blackford LS, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A, Pozo R, Remington K, Whaley RC (2001) An updated set of basic linear algebra subprograms (blas). ACM Trans Math Softw 28:135–151
5. Caron E, Uter F (2002) Parallel extension of a dynamic performance forecasting tool. Sci Ann Cuza Univ 11:80–93
6. Chandra R (2001) Parallel programming in OpenMP. Morgan Kaufmann, Burlington
7. Demmel J, Marques O, Parlett BN, Vömel C (2008) Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers. SIAM J.Sci Comput 30(3):1508–1526
8. Frigo M, Johnson S (1998) FFTW: an adaptive software architecture for the FFT. In: Proceedings of IEEE International Conference on Acoustics Speech and Signal Processing vol. 3, pp 1381–1384
9. García L, Cuenca J, Giménez D (2007) Including improvement of the execution time in a software architecture of libraries with self-optimisation. In: ICSOFT 2007, Proceedings of the Second International Conference on Software and Data Technologies, Volume SE, Barcelona, Spain, pp 156–161, 22–25 July
10. García LP, Cuenca J, Giménez D (2014) On optimization techniques for the matrix multiplication on hybrid cpu+gpu platforms. Ann Multicore GPU Program 1(1):10–18
11. Hasanov K, Quintin JN, Lastovetsky A (2014) Hierarchical approach to optimization of parallel matrix multiplication on large-scale platforms. J Supercomput 71(11):24–34
12. Katagiri T, Kise K, Honda H (2005) RAO-SS: a prototype of run-time auto-tuning facility for sparse direct solvers. Tech Rep 22(1):1–10
13. Katagiri T, Kise K, Honda H, Yuba T (2004) Effect of auto-tuning with user's knowledge for numerical software. Proceedings of the 1st conference on computing frontiers, Ischia, Italy. ACM, New York, NY, USA, pp 12–25
14. Nath R, Tomov S, Dongarra J (2010) An improved magma gemm for fermi graphics processing units. Int J High Perform Comput Appl 24(4):511–515
15. Paterson MS, Stockmeyer LJ (1973) On the number of nonscalar multiplications necessary to evaluate polynomials. SIAM J Comput 2(1):60–66
16. PLASMA (2015) Parallel linear algebra software for multicore architectures. Available in: http://www.netlib.org/plasma/. Accessed 1 June 2015
17. Tanaka T, Katagiri T, Yuba T (2007) D-spline based incremental parameter estimation in automatic performance tuning. In: International Conference on Applied Parallel Computing: State of the Art in Scientific Computing, PARA'06. Springer-Verlag, Berlin, Heidelberg, pp 986–995
18. Vuduc R, Demmel J, Bilmes J (2004) Statistical models for empirical search-based performance tuning. Int J High Perform Comput Appl 18:65–94
19. Whaley RC, Petitet A, Dongarra JJ (2001) Automated empirical optimizations of software and the ATLAS project. Parallel Comput 27:21–37