Denoising applied to spectroscopies-part II: Decreasing computation time

Guillaume Laurent, Pierre-Aymeric Gilles, William Woelffel, Virgile Barret-Vivin, Emmanuelle Gouillart, Christian Bonhomme

Denoising applied to spectroscopies – part II: decreasing computation time

Guillaume LAURENT (1*), Pierre-Aymeric GILLES (1), William WOELFFEL (2), Virgile BARRET-VIVIN (1), Emmanuelle GOUILLART (2), Christian BONHOMME (1)

(1) Sorbonne Université, Collège de France, CNRS, Laboratoire de Chimie de la Matière Condensée de Paris, LCMCP, F-75005, Paris, France.

(2) Saint-Gobain, CNRS, Surface du Verre et Interfaces, SVI, F-93300, Aubervilliers, France.

(1) Case Courrier 174, 4 place Jussieu, 75252 Paris Cedex 05, France.

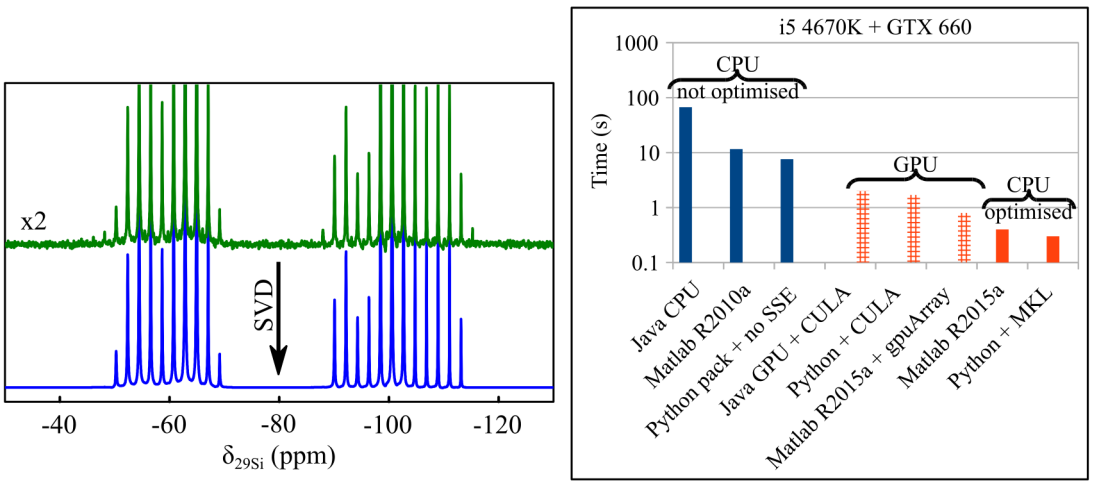
(2) 39 quai Lucien Lefranc, BP 135, 93303 Aubervilliers Cedex, France.

(*) Corresponding author

Denoising applied to spectroscopies – part II: decreasing computation time

Spectroscopies are of fundamental importance but can suffer from low sensitivity. Singular Value Decomposition (SVD) is a highly interesting mathematical tool, which can be conjugated with low-rank approximation to denoise spectra and increase sensitivity. SVD is also involved in data mining with Principal Component Analysis (PCA). In this paper, we focussed on the optimisation of SVD duration, which is a time-consuming computation. Both Intel processors (CPU) and Nvidia graphic cards (GPU) were benchmarked. A 100 times gain was achieved when combining divide and conquer algorithm, Intel Math Kernel Library (MKL), SSE3 (Streaming SIMD Extensions) hardware instructions and single precision. In such case, the CPU can outperform the GPU driven by CUDA technology. These results give a strong background to optimise SVD computation at the user scale.

Keywords: spectroscopy, signal processing, Cadzow denoising, Singular Value Decomposition (SVD), benchmarking



graphical abstract

Introduction

Spectrocopies are very efficient tools to help scientists in various domains: physics, chemistry, biology or medecine. However, the intrinsic sensitivity is highly dependant of the technique used. In particular, Nuclear Magnetic Resonance (NMR) (1) and Raman spectroscopy (2) are poorly sensitive but are very precise local probes commonly used to study materials. While NMR can detect only one nucleus over 10^5 in usual conditions (3), Raman spectroscopy can only detect one photon over 10^6 (4). Both NMR and Raman sensitivity were improved over the years, as was detailed in part (I) of this study (5), but still suffer from low signal-to-noise ratio (SNR), especially when studying amorphous or non-stable materials. Additionally, different physicochemical techniques can be hyphenated to obtain multiple signatures in a single experiment (6, 7).

This sensitivity gain entails an increasing amount of data to analyse, especially in the domain of metabolomics (8, 9). In such case, it is necessary to combine statistics and chemistry, what is called chemometrics (10, 11). A similar problem of data mining is present in social engineering (12) or with medical images dictionaries (13). Many tools are available to process these data: Principal Component Analysis (PCA) (14), Principal Component Regression (PCR) (15), Partial Least Squares (PLS) (16), Discriminant Analysis (PLS-DA) (17), Independent Component Analysis (ICA) (18), or Non-negative Matrix Factorisation (NMF) (19). The aim of these multivariate data analysis methods is to find relevant parameters in order to discriminate samples (ex wine from region A or B (20)). These tools are of paramount importance to apply data mining to spectroscopies (21). For instance, PCA was used recently with Gas Chromatography / Quadrupole Time-of-Flight (GC/Q-ToF) mass spectrometry (22), Mid-InfraRed (MIR) spectroscopy (23) and Inductively Coupled Plasma Optical

Emission Spectroscopy (ICP-OES) (24). An important step of PCA and relative techniques is Singular Value Decomposition (SVD) (25, 26).

Moreover, SVD was proposed as a method to denoise signals by Tufts *et al* (27), and was generalised by Cadzow in 1988 (28). In the context of low sensitivity and in the continuation of a previous communication (29), we thoroughly described SVD in part (I) of this work (5). We first gave theoretical background on SVD, low-rank approximation (30), Hankel or Toeplitz matrices (31) and SNR definitions. SVD was applied to Raman and NMR spectra. We highlighted that best results were obtained with square matrices and data in time domain rather than in frequency domain. Automatic thresholding was applied thanks to Malinowki's significant level indicator (32). $6 \times 7380 = 44280$ denoised spectra with known noise were compared to their non-noisy counterparts. It was evidenced that the minimum peak SNR measured on maximum of noise ($PSNR_{max}$) needed to have reliable results was $PSNR_{max} = 2$, leading to a gain on acquisition time of 2.3. Surprisingly, while Lorentzian peaks were correctly denoised, SVD transformed Gaussian peaks into intermediate Gaussian / Lorentzian ones, which overestimated their peak area by 20 %.

The main disadvantage of SVD is its long computation time, especially for big data sets. It is thus essential to optimise the computation procedure. Different approaches have been chosen in literature: specialised processors (33), wavelet transformation before performing the SVD (34), divide and conquer method (35) or sparse matrices (36, 37). Another approach is General Purpose computing on Graphics Processing Unit (GPGPU) (38). SVD has been applied multiple times using GPGPU (39–42). Two programming languages are available: CUDA (Compute Unified Device Architecture) for Nvidia graphic cards (43) and OpenCL (Open Computing Language) for all graphic cards (Graphic Processing Unit, GPU) and processors (Central

Processing Unit, CPU) (44). In addition, Dongarra *et al.* developed very efficient algorithms combining CPU and GPU (45, 46), what is called heterogeneous computing (47). Recently Man *et al* proposed a Java implementation of SVD for NMR (48) using CPU (49) or Nvidia GPU (50). Pending questions can be raised on these Java applications: (i) how powerful does the computer need to be? (ii) how long does computation take? (iii) how large can be the data set? (iv) is the used algorithm efficient? (v) will other programming languages give better results?

In this second part (II), after providing some experimental details in section "Materials and methods", we benchmarked SVD using Java, on various CPU and Nvidia GPU ranging over 10 and 6 years, respectively. We then focussed on algorithms and precision under Matlab. In a further step, we tried to decorrelate software libraries from hardware capabilities (Single Instruction Multiple Data, SIMD) (51). Finally, we compared Java, Matlab and Python to reach the fastest possible denoising computation.

Materials and methods

Solid-state NMR experiments

Two solid-state NMR spectra were used to benchmark SVD. The first one was a ^{29}Si spectrum with 4096 complex points, used for matrices up to 2015 × 2014 = 4.1e6. For matrices above this limit, a ^{87}Sr spectrum with 30504 complex points, was chosen. The noisy and denoised spectra for ^{29}Si and ^{87}Sr are presented on Figures 1a and 1b, respectively. SVD was applied on Free Induction Decay (FID, time domain) after removal of the first 68 points corresponding to oversampled digitalisation.

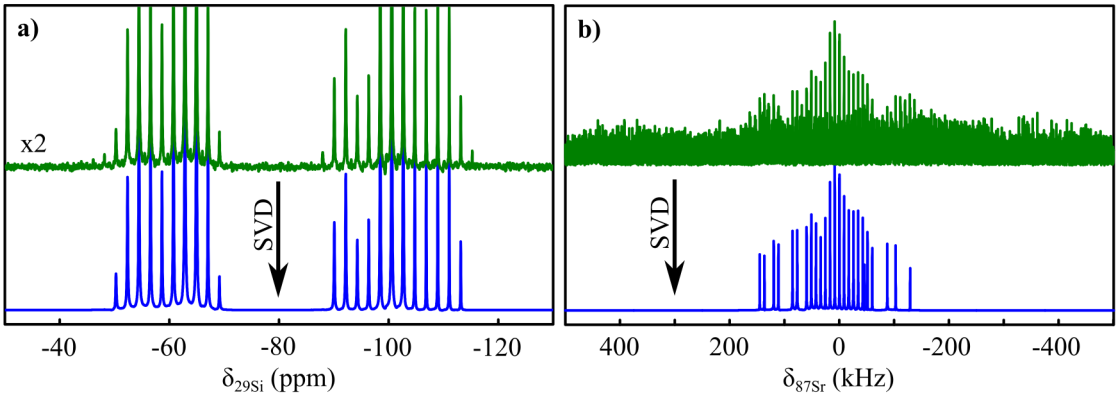


Figure 1: Spectra used to benchmark SVD; top: noisy spectra; bottom: denoised spectra. a) ^{29}Si CPMG MAS solid-state NMR spectrum of a 50:50 MTEOS:TEOS sample; processing: Java GPU application with 2015 × 2014 = 4.1e6 points, k = 25 singular values, computation time of 6 s on a GTX 660, cosine apodisation. b) ^{87}Sr DFS-WURST-CPMG solid-state static NMR spectrum of non-hydrated non-protonated $Sr(PO_3F)$; processing: Java GPU application with 4097 × 4096 = 1.7e7 points, k = 25 singular values, computation time of 31 s on a GTX 660, cosine apodisation, magnitude calculation.

The sample analysed by ^{29}Si solid state NMR and the experiments were presented in part (I) of this study (5). Briefly, it was representative of sol-gel chemistry, combining hydrophobicity and mechanical properties (52). A 50:50 mix of Methyltriethoxysilane (MTEOS) : tetraethylorthosilicate (TEOS) was prepared by spray drying giving spherical micrometer silica particles. Carr-Purcell-Meiboom-Gill (CPMG) Magic Angle Spinning (MAS) experiments (53) were performed in 40 minutes on a Bruker Avance III spectrometer operating at 300.29 MHz for ^{1}H and 59.65 MHz for ^{29}Si.

The sample analysed by ^{87}Sr solid state NMR was a model of biocompatible material in relation with bone substitutions (54). Non-hydrated non-protonated $Sr(PO_3F)$ was studied on a Bruker Avance III spectrometer operating at 699.98 MHz for ^{1}H and 30.34 MHz for ^{87}Sr in a 5 mm static probe. In order to enhance sensitivity, DFS (55), WURST (56) and CPMG were used with 58,000 transients and a relaxation delay of

300 ms, leading to a total acquisition time of 5.5 h. 260 echoes were acquired with a full echo delay of 0.12 ms.

Measurement of SVD computation times

Special care has been taken to validate measured times: the computer was checked to be idle, without any update running and without an active internet browser window, which would use the graphic card through Adobe Flash module. Computations were systematically repeated and results were highly reproducible. Nvidia drivers ranging from 331.113 to 352.30 were used, either under Linux Ubuntu 14.04, Centos 5, Fedora 21, 22 or Windows XP, 7 SP1 or 8.1. Under Windows it was necessary to modify the TdrLevel registry key to 0 in order to avoid graphics driver failure (57). Under Java (Oracle Corporation, Redwood Shores, CA, USA) and Matlab (The MathWorks, Inc., Natick, MA, USA), ^{29}Si FID was used up to 4028 points. Above this value, ^{87}Sr FID was used. The corresponding FID was truncated if needed to the desired data length. Under Python (58), data set was a simple list of increasing values with the corresponding length. Unless otherwise stated, $k = 25$ singular values were kept for low-rank approximation. This value corresponded to the major spikelets observed on ^{29}Si spectrum (Figure 1a) and was not changed for coherence along the series. The measured delays are the sum of decomposition and low-rank approximation steps, including all processor to graphic card latencies, if relevant.

Java

Two applications are available online: one for CPU (49) and the other one for Nvidia GPU (48, 50). While CPU version calls JAMPACK library (59), GPU version calls CULA R15 (60). The CPU 32 bits version failed above a matrix size of

1025 × 1024 = 1.0e6 points, whatever the CPU used. The same problem was observed with GPU 32 bits version above a matrix size of 3005 × 1024 = 3.1e6 points. The reason is that Java heap space is limited to around 1.5 GB for 32 bits applications (61), while it is 16 exabytes for 64 bits applications. This memory amount corresponds not only to the data, but also to the program and its libraries. 64 bits applications are not compatible with 32 bits Java runtime environment and 32 bits operating systems. As the source code was not accessible and no internal timer was implemented in these Java applications, computation times were measured with a handheld chronograph giving a time resolution of 1 s for both SVD step and low-rank approximation step. For a same GPU, no computation time difference was observed between Windows and Linux.

Matlab

Three versions were tested: R2010a, corresponding to the most recent compatible with CULA library free (in version R14) (62); R2014a, corresponding to the most recent for old graphic cards with Compute Capability (CC) less than 2.0, such as GTX 260; and R2015a, corresponding to a recent version. FID were imported thanks to matNMR (63). Under R2014a and R2015a, their respective Parallel Computing Toolbox was added to use gpuArray function. Computation times were measured with an internal timer. The source code is available in file Figure_II.4a_II.4b.m of (64).

Python

No significant time difference was observed between Python 2.7 and 3.5 versions, neither between 32 and 64 bits versions. The source code is compatible with all of these options and is available in file Figure_II.5.py of (64) (CPU and GPU). Different SVD implementations were tested with NumPy 1.10.1 (65), SciPy 0.16.1 (66) and Scikit-

Cuda 0.5.0 (67). The last one required PyCUDA 2015.1.3 (68) and CULA R18 free (60). Computation times were measured with an internal timer as well.

Under Linux, Automatically Tuned Linear Algebra Software (ATLAS) (69) and Open source Basic Linear Algebra Subprograms (OpenBlas) (70) development libraries, separately either one or the other, were installed as rpm packages. For each library, NumPy and SciPy were build with the pip mechanism[1]. Compiling these two latter packages with Intel Math Kernel Library (MKL) (71) was also tested. Under Windows, NumPy and SciPy superpack 32 bits with ATLAS library are available (72, 73), and also pre-compiled packages with MKL library (74).

Results and discussion

As stated in the introduction, the main disadvantage of SVD is its long computation time. Of course the hardware itself is important but multiple steps are present between the human level function call and the hardware level implementation, namely the algorithm, the libraries and the use of hardware instructions (Figure 2a). Even on hardware, we can choose to compute either on CPU or on GPU. In the following, we checked the respective benefits of all these parameters.

1 Forcing reinstallation of a specific python package can be done with 'pip install --ignore-installed numpy==1.10.1'.

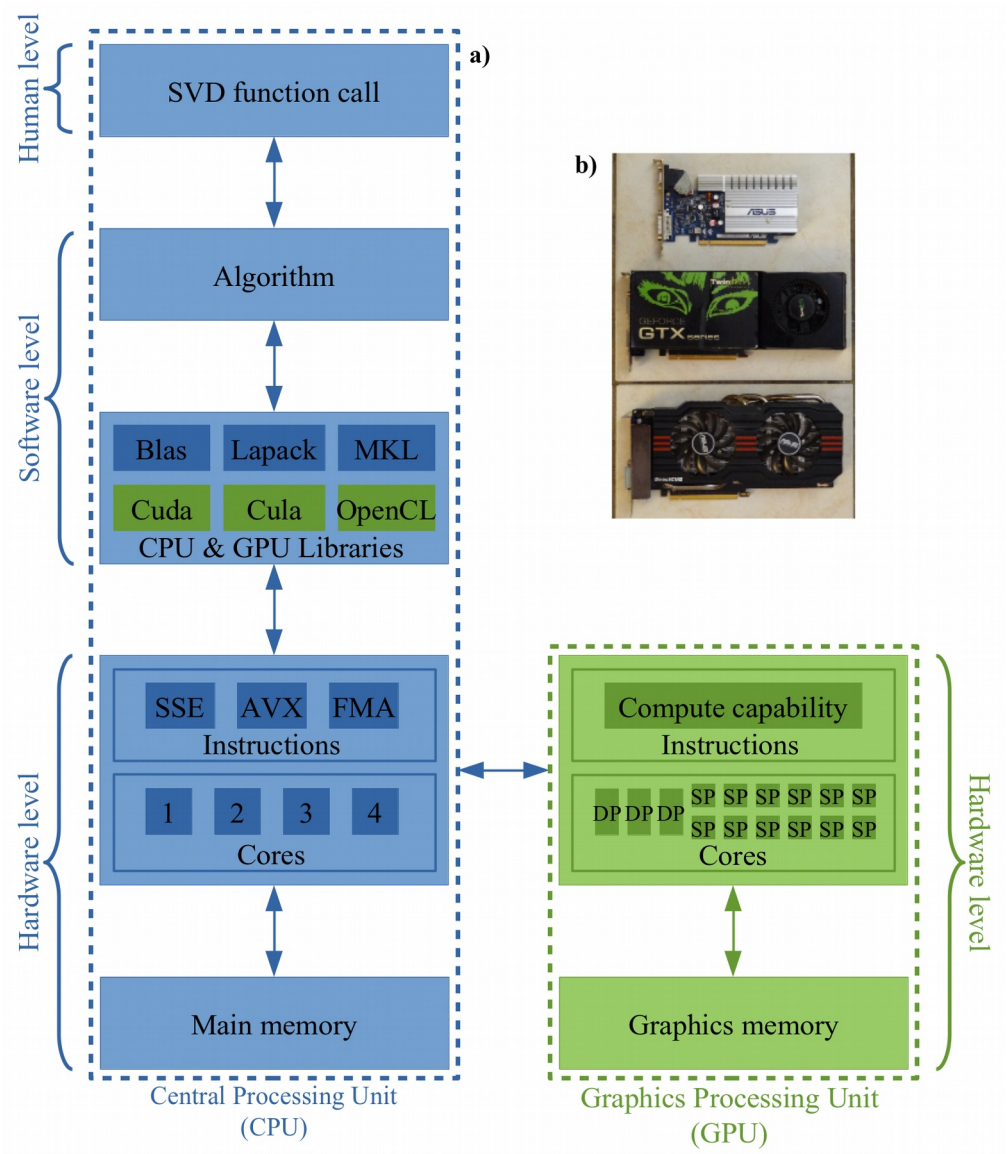


Figure 2: a) SVD function call diagram on CPU and GPU. MKL: Intel Math Kernel Library; SSE: Streaming SIMD Extensions; AVX: Advanced Vector Extensions; FMA: Fused Multiply Add; DP: double precision; SP: single precision. b) Some Nvidia GPU used. From top to bottom: 8400 GS, GTX 260, GTX 660.

Influence of hardware under Java CPU and GPU applications

Java CPU and GPU benchmarks

In an attempt to characterise the hardware needed to compute SVD, we used CPU ranging over 10 years and GPU ranging over 6 years, for both desktop and laptop computers (Tables 1 and 2 and Figure 2b). According to Figure 2a, we were changing

hardware level, supposing that everything was optimised at software level and measuring durations at human level. Study has been restricted to Intel CPU and Nvidia GPU because no AMD CPU was available in the laboratory and AMD GPU are not compatible with CUDA. Results are presented on Figure 3a.

We noticed first that computation were much faster for GPU than for CPU, with 2 to 23 s for GPU (dashed lines) and 67 to 500 s for CPU (top right end of plain lines), *i.e.* 22 to 34 times speed increase, for a quite small matrix of $1025 \times 1024 = 1.0e6$ points. This almost square shape was due to construction of the Hankel matrix with a dataset containing an even number of points. Indeed, in order to not overwrite the corner point, it is necessary to add one row over columns. No significant time difference was seen with a true square matrix and thus this small shape difference will be neglected in the following.

Surprisingly, even a low-end GPU of 2008 (8400 GS) was surpassing a middle-range CPU of 2013 (Core i5 4670K) (with 23 and 67 s, respectively). This behaviour was really intriguing, as we would expect at least similar computation performance (75). However, when checking CPU activity, we observed that, with the CPU application, only one core was busy, what is called mono-threading. On the contrary, with the GPU application, not only GPU was fully busy, but also all the available CPU cores were used, what is called multi-threading. This difference between mono- and multi-threading is explored in section“Influence of algorithm under Matlab”.

Table 1. Properties of **CPU used for SVD** under Java. Gray rows indicate hardware used for SVD under Matlab and Python. a: ark.intel.com; b: CPU-Z 1.72; c: Performance Test 8.0.

Central Processing Unit (CPU)	**Type**	**Year** [a]	**Fabri-cation (nm)** [b]	**Number of cores** [b]	**Cache (MB)** [a]	**Core frequency (MHz)** [b]	**Memory frequency (MHz)** [b]	**Memory size (MB)** [b]	**CPU Mark** [c]	**Mono-thread (Mops/s)** [c]	**Matrices (millions/s)** [c]
Intel Pentium M 745	laptop	2004	90	1	2	1800	133	1024	444.8	577	1.41
Intel Pentium 4 530	desktop	2004	90	2	1	3000	200	1024	335.2	726	0.29
Intel Core 2 Duo E6400	desktop	2006	65	2	2	2130	333	2048	1451	849	3.72
Intel Core 2 Quad Q8200	desktop	2008	45	4	4	2330	400	4096	2001	1004	5.8
Intel Core 2 Duo T9600	laptop	2008	45	2	6	2800	400	4096	2190	1161	4.67
Intel Core i3 4005U	laptop	2013	22	4	3	1700	800	4096	2551	1010	11.4
Intel Core i5 4670K	desktop	2013	22	4	6	4200	1000	8192	8824	2519	31.6

Table 2. Properties of **GPU used for SVD** under Java. Gray rows indicate hardware used for SVD under Matlab and Python. a: GPU-Z 0.8; b: Cuda-Z 0.9; #N/A: not available.

Graphics Processing Unit (GPU)	**Type**	**Year** [a]	**Fabri-cation (nm)** [a]	**Number of cores** [a]	**Band-width (GB/s)** [a]	**Core frequency (MHz)** [a]	**Memory frequency (MHz)** [a]	**Memory size (MB)** [a]	**Single precision float (GFLOPS)** [b]	**Double precision float (GFLOPS)** [b]	**CUDA compute capability** [b]
Nvidia Quadro FX 570	desktop	2007	80	16	12.8	460	400	256	29	#N/A	1.1
Nvidia GeForce 8400 GS	desktop	2008	65	8	6.4	567	400	512	21	#N/A	1.1
Nvidia Quadro NVS 160M	laptop	2008	65	8	11.2	580	700	256	23	#N/A	1.1
Nvidia Quadro FX 770M	laptop	2008	65	32	25.6	500	800	512	79	#N/A	1.1
Nvidia GeForce GTX 260	desktop	2008	65	216	111.9	576	1000	896	533	67	1.3
Nvidia GeForce 820M	laptop	2012	28	96	14.4	625	900	2048	315	31	2.1
Nvidia GeForce GTX 660	desktop	2012	28	960	144.2	1100	1500	2048	1707	88	3.0

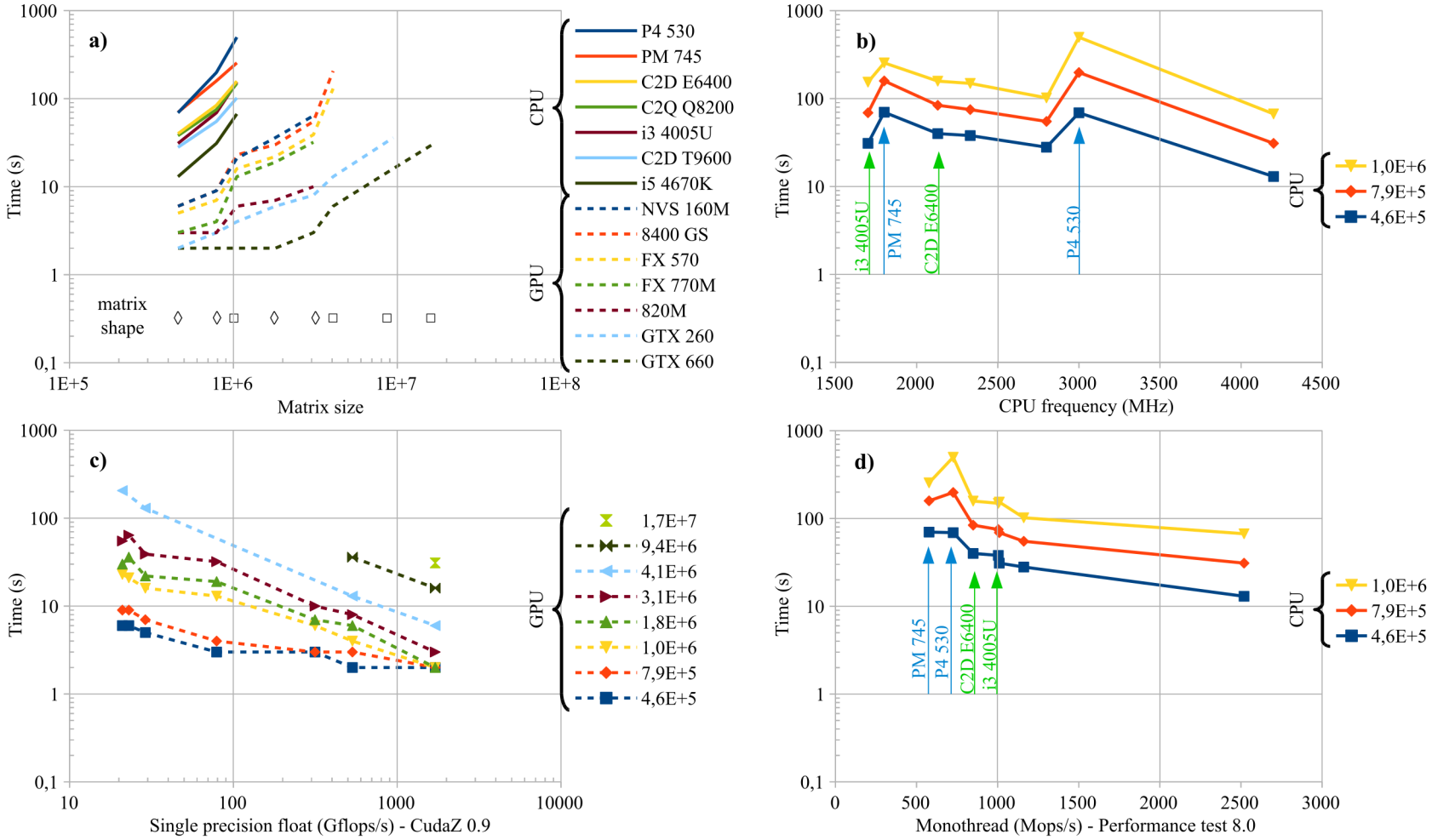


Figure 3: Computation times for **SVD under Java**. a) Comparison of CPU and GPU times against matrix size; ◊: rectangular matrix; □: square matrix. b) CPU times against CPU frequency. c) GPU times against single precision float performance. d) CPU times against monothread performance; light blue and light green arrows evidence CPU cache and CPU release year influence, respectively. CPU and GPU times are drawn with plain lines and dashed lines, respectively.

When increasing the square matrix size, a linear trend was visible in logarithmic scale, down-shifted for faster hardware (Figure 3a). However, computation time jumped when going from a rectangular matrix (diamond symbol) to a square one (square symbol), even if matrix size was not so different. This behaviour was observed for both CPU and GPU at 1537 × 512 = 7.9e5 *vs.* 1025 × 1024 = 1.0e6 points, and similarly for GPU at 3005 × 1024 = 3.1e6 *vs.* 2015 × 2014 = 4.1e6 points. This denoted that the number of mathematical operations dramatically increased for a square matrix. An explanation could be the use of reduced SVD for rectangular matrices, not computing the last rows and columns of U and V^T unitary matrices (see part (I) of this work (5)). This time jump is probably the reason why a rectangular matrix shape is chosen in most studies, despite a square matrix gives more precise singular values, *i.e.* more performant denoising.

Thanks to the speed up obtained on GPU, it was possible to compute a matrix of 4097 × 4096 = 1.7e7 points in 31 s on a mid-range GPU of 2012 (GTX 660). By extrapolating the curve in Figure 3a, it may take around 6000 s on a mid-range CPU of 2013 (Core i5 4670K). For square matrices, the slope seems similar between graphic cards. As underlined in section “Materials and methods”, only 64 bits Java versions can handle matrix sizes above 3005 × 1024 = 3.1e6 points. However, all the processors and some graphic cards (NVS 160M, FX 770M and 820M) were benchmarked using the 32 bits Java applications, which explains the truncated curves for this hardware.

Java CPU performance indicator

To better characterise the hardware needed for Java CPU SVD, we looked for a performance indicator. One would expect CPU frequency to be a good one but it was clearly not the case as shown on Figure 3b. However, mono-thread performance

evidenced a trend and was thus a usable parameter as shown on Figure 3d. This is coherent with our observation of only one active processor core under the Java CPU application.

Characteristic points are visible on these curves (Figures 3b and 3d). Light blue arrows highlight Pentium M 745 and Pentium 4 530, which were two processors of 2004. Their frequencies were very different (1800 and 3000 Mhz, respectively) but their mono-thread performance were similar. Additionally, with a bigger matrix size (yellow line), the Pentium M 745 was faster than the Pentium 4 530, even if the latter had a higher frequency. The main difference between them was their cache size, of 2 and 1 MB, respectively. The CPU cache is the amount of quick memory directly available inside the processor. On the contrary, memory plugged into motherboard is at least 10 times slower. As SVD request many matrix-vector multiplication, memory access is limiting.

Light green arrows evidence Core 2 Duo E6400 and Core i3 4005U, which were two processors with a similar frequency but released in 2006 and 2013, respectively. No performance increase was observed using Java SVD CPU application between these two processors. That was also questioning as we would expect that some hardware optimisations happened in 7 years. These observations highlight that the best CPU for SVD under Java will not necessarily be recent or have a high frequency, but rather have a high memory cache and a high monothread performance. In other words, it is better to use an old high- or middle-range CPU than a new low-range one. This explains why Core 2 Duo T9600 was faster than Core i3 4005U. The former processor is a good candidate to denoise all over the night a matrix of $8193 \times 8192 = 6.7e7$ points with the 64 bits CPU Java application.

Java GPU performance indicator

Graphic card computing power is characterized by core frequency and number of cores. Despite frequency was not so different along the series, number of cores was strongly increasing over the years and with them the Single Precision Floating Point performance (SP or FP32), expressed in Giga FLoating-point Operations Per Second (GFLOPS) (76). Only high-end professional cards have a high Double Precision Floating Point performance (DP or FP64). General public GPU are more commonly devoted to games and lack DP. The precision is the number of bits used to store numbers, 32 and 64 for SP and DP, respectively (77). The higher the precision used, the lower the computed error. However, the errors initially present in the matrix can be larger than rounding errors (78). Moreover, CULA free (60), the library implemented on Java GPU application could only use SP. It was thus useless to invest money in professional cards and we favoured general public GPU. For instance, a Nvidia Tesla P100 GPU costs around 8 k€. On Figure 3c, a time decreasing linear trend was obtained in logarithmic scale when increasing SP, which denoted a good indicator.

Another important parameter for SVD with Java GPU application, was the amount of memory available, both on GPU (device) and on motherboard (host). Plassman stated that SVD needed up to $8\ n^2 + 12\ n$ work storage (79), for a matrix with n columns. This value had to be multiplied by 4 bytes for both floating and integer numbers to be stored in memory. Additionally, there was a 1.5-3 times transient overhead during low-rank approximation. Following this rule, the largest tractable matrix was $6657 \times 6656 = 4.4e7$ points on a GPU with 2 GB of memory.

In this section we have seen influence of hardware on SVD computation time under Java. As stated above, the time difference between CPU and GPU Java application was intriguing, especially when comparing the low-end graphic card

8400 GS to the middle-range processor Core i5 4670K. Additionally, the CPU and GPU application were mono-threaded and multi-threaded, respectively. This was typically an algorithm problem and we explored it using Matlab.

Influence of algorithm under Matlab

Algorithms are the mathematical operations involved and their informatics implementation to obtain the relevant function. Plassman compared the available SVD algorithms and their impact on ill-conditioned matrices (79). The simplest computation method is to use eigendecomposition, but its lack of precision was demonstrated by Läuchli (80). The classic complete SVD uses a three steps process:

(1) reduction to bidiagonal form,
(2) computation of SVD on bidiagonal matrix,
(3) obtention of singular vectors.

Step (1) involves Householder reflections and step (2) can either use QR iteration in Golub-Kahan-Reinsch (GKR) algorithm (81, 82), divide-and-conquer method (35, 83) or Multiple Relatively Robust Representations (MRRR) (84). An alternative SVD algorithm, combining steps (1) and (2), is to use Jacobi rotations and convergence criteria (85).

To explore algorithm influence we focussed on two computers, one from 2008 with a Core 2 Quad Q8200 and a GTX 260 under Linux, and the other one from 2013 with a Core i5 4670K and a GTX 660 under Windows. Both were in the same price segment and reflected middle-range equipment available at those dates. The used Matlab versions were detailed in section “Materials and methods”. According to Figure 2a, we fixed hardware level and observed software level influence.

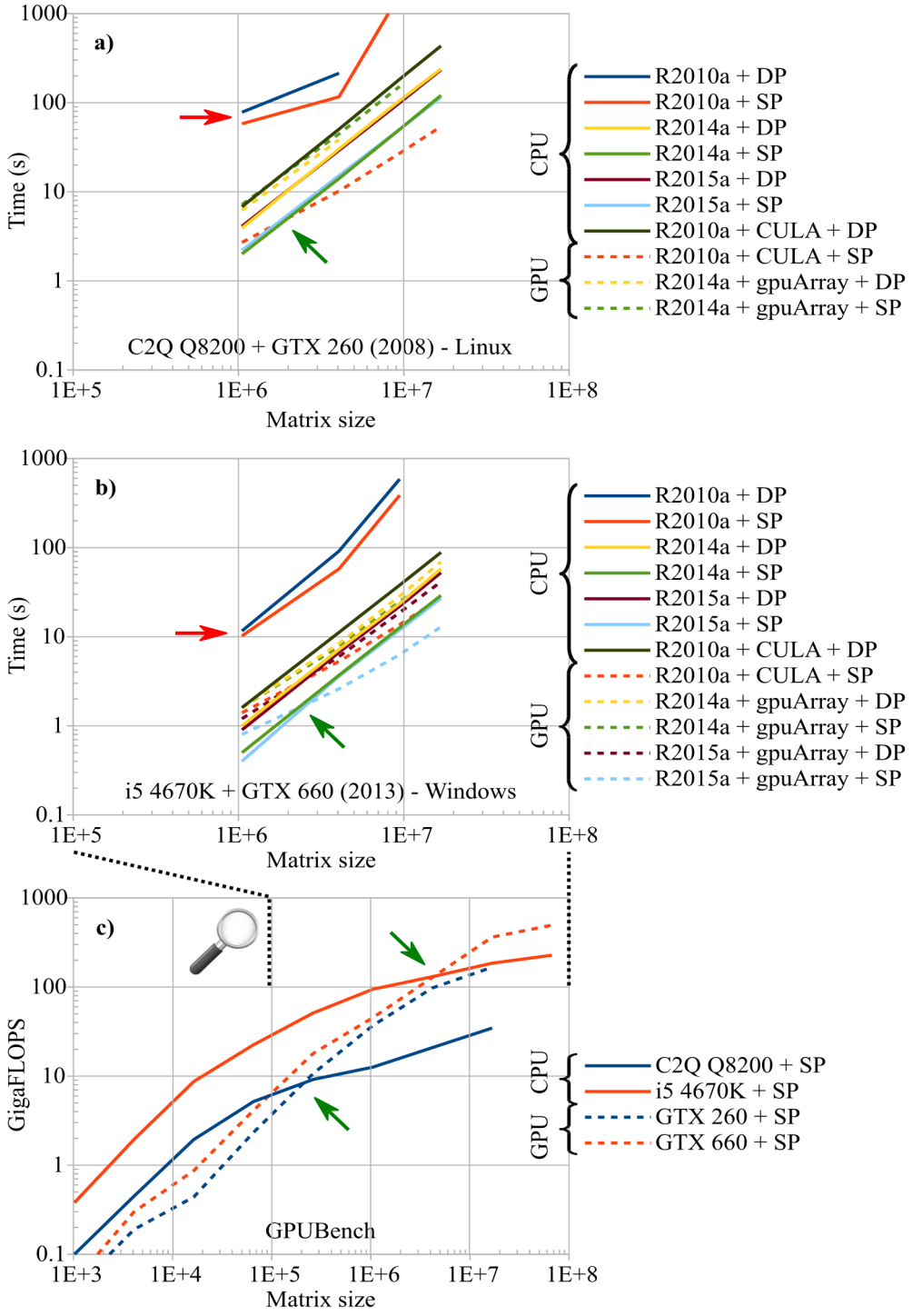


Figure 4: a) and b) Computation times for **SVD under Matlab**. Comparison of CPU and GPU times against matrix size for different Matlab versions. a) C2Q Q8200 + GTX 260 (2008). b) i5 4670K + GTX 660 (2013). c) SP performance against matrix size measured with GPUBench v1p7 under R2014a; the horizontal scale is larger than on a) and b). CPU and GPU times are drawn with plain and dashed lines, respectively. Red and green arrows indicate bad algorithm and CPU-GPU crossing, respectively.

Matlab R2010a

Under Matlab R2010a, DP computation times on CPU were already smaller than those with Java CPU application for a matrix of 1025 × 1024 = 1.0e6 points (plain dark blue line on Figures 4a and 4b): 78.2 s instead of 149 s for Core 2 Quad Q8200, and 11.6 s instead of 67 s for Core i5 4670K. It should be noted that computation times can be divided by two when using single precision (plain red line), and by two again with non-complex data (not shown). Java CPU application used complex numbers and accordingly to the above results, it presumably used DP. While Java used only mono-threading, Matlab computation started with a multi-threaded step and kept on with a mono-threaded one. This already denoted a different algorithm between the two programming languages.

Under this Matlab version it was also possible to use GPU with CULA, which was the library implemented under Java GPU application. Results were similar between Java GPU and Matlab R2010a + CULA + SP GPU applications (dashed red line). However, as we used the free version of CULA, DP computation was not allowed on GPU and R2010a + CULA + DP (plain black line) felt back on CPU with LAPACK library in multi-threading mode. As a consequence, strong improvement was observed against R2010a + DP (plain dark blue line). At this point, an order of magnitude on computation times has already been gained for CPU DP under Matlab.

Matlab R2014a

Further improvement on CPU was obtained with R2014a + DP (plain yellow line on Figures 4a and 4b) being almost two times faster than R2010a + CULA + DP (plain black line), and 7-33 times faster than R2010a + DP (plain dark blue line), depending on matrix size. This was explained by the divide and conquer approach preferred for SVD

starting from R2010b. Gu *et al* claimed that this algorithm was 9 times faster on bidiagonal matrices (86), in agreement with our observations. The extra gain is due to the four cores simultaneously used on the CPU. Again SP (plain green line) was two times faster than DP. Small matrices, up to 1025 × 1024 = 1.0e6 points for Core 2 Quad Q8200, and up to 3073 × 3072 = 9.4e6 points for Core i5 4670K, were even computed faster on CPU with R2014a + SP than on GPU with R2010a + CULA + SP, as indicated by the green arrow. Unfortunately, CULA free was not compatible with R2014a, but it was nevertheless possible to use GPU thanks to the gpuArray Matlab function. Surprisingly, worst results were obtained, with a GPU time longer than its corresponding CPU time. Moreover R2014a + gpuArray + DP times (dashed yellow line) were shorter or equal to R2014a + gpuArray + SP times (dashed green line), what is in contradiction with DP / SP ratio on GPU (1 / 8 and 1 / 24 for GTX 260 and GTX 660, respectively). This revealed that part of the computation was done in DP, despite SP was called. When checking CPU and GPU activity during SVD, it was observed that GPU was only used at the beginning and at the end of the processing. This denoted that SVD using gpuArray under Matlab R2014a was not an optimised algorithm and that this version should be avoided.

Matlab R2015a

In order to check if a new version of Matlab could further improve computation times, we used Matlab R2015a. A slight decrease was observed on CPU from 57.8 to 52.5 s and from 29.1 to 27.0 s for R2014a + DP (plain yellow line on Figures 4a and 4b), R2015a + DP (plain brown line), R2014a + SP (plain green line) and R2015a + SP (plain light blue line), respectively, for a matrix of 4097 × 4096 = 1.7e7 points. Matlab R2015a was not compatible with GTX 260 GPU, due to its compute capability of 1.3. A

much stronger improvement was obtained for the above matrix size with GTX 660 GPU: from 69.1 to 41.4 s and from 52.7 to 12.9 s for R2014a + gpuArray + DP (dashed yellow line), R2015a + gpuArray + DP (dashed brown line), R2014a + gpuArray + SP (dashed green line) and R2015a + gpuArray + SP (dashed light blue line), respectively. The latter configuration outperformed R2010a + CULA + SP (27.5 s, dashed red line) owing to a more pronounced GPU utilisation during SVD. Despite SP was faster than DP, the DP / SP = 1 / 24 ratio was not respected. However, SVD algorithm was strongly optimised in R2015a + gpuArray against R2014a + gpuArray. While Core i5 4670K CPU remained more efficient for matrices up to 1025 × 1024 = 1.0e6 points, GTX 660 GPU outperformed it in SP mode for larger matrices. The obtained computation times under Matlab R2015a are thus very good, both on CPU and GPU and were better than under Java.

Matlab GPUBench

The cross in computation time between CPU and GPU was further investigated with GPUBench v1p7 (87). This code compared CPU and GPU performance against matrix size for matrix-vector left division, which is a linear equations system solver. Such computation gives much less peak SP and DP float performance than reported in Table 2, and is rather compute-bond than memory-bound. This benchmark involves lots of matrix-vector operations as SVD does. For both 2008 and 2013 computers, a crossing was visible between CPU and GPU in SP mode (Figure 4c). This was explained by the time needed for data goings and comings between processor and graphic card and between graphic card core and its memory (Figure 2a). This is a hardware limitation. Interestingly, the cross appeared in the same matrix size range (1e5 to 1e7) than the one observed for SVD (Figures 4a and 4b). However, its position strongly depends on the

algorithm used and on the relative float performance of CPU and GPU. Low-end GPU are thus not recommended as better results are obtained with CPU. In DP mode, despite the half computing power of a Core i5 4670K against SP, even a GTX 660 never overpassed it (not shown).

Similarly to Java, matrix size under Matlab is limited by host and device memory. Nevertheless, memory consumption under Matlab is improved over Java as no overhead is present during low-rank approximation, pushing away maximum matrix size. Additionally, host memory amount is considerably reduced, down to be almost identical to device memory one. A GPU with 2 GB of memory is thus limited to matrices of 7169 × 7168 = 5.1e7 points.

In this section, we highlighted that the divide and conquer algorithm decrease SVD computation time by a factor of nine. SP gives an additional factor of two in computation time on CPU, being faster than GPU for matrices smaller than 1025 × 1024 = 1.0e6 points (Matlab R2014a *vs.* R2010a). Despite the strong improvement for SVD on CPU, middle-range GPU remains relevant in SP mode for matrices above this size, up to the GPU memory limit (see previous paragraph). For legacy hardware dating from 2008, the best compromise is to use Matlab R2010a and CULA free R14 with SP. For hardware dating from 2013, the best choice is to use the most recent Matlab version with SP and gpuArray function. CPU computation should especially be avoided on Matlab R2010a as evidenced by the red arrows on Figures 4a and 4b. Matlab R2014a is not recommended neither for GPU. Next step was to focus on the libraries used and their call to hardware instructions, which we explored under Python.

Influence of libraries and hardware instructions under Python

According to Figure 2a, software level is divided in algorithms and libraries. After changing the algorithms, *i.e.* the involved mathematical functions, we were interested in the underlying libraries, *i.e.* the link between software level and hardware level. A library is a collection of functions that consists of pre-written optimised code. A single library can be called by multiple software or by other libraries. Usually, SVD first calls LAPACK (Linear Algebra PACKage) (88) which itself calls BLAS (Basic Linear Algebra Subroutines) (89). While LAPACK is a high-level library, BLAS is a low-level one, optimised by CPU hardware specialists (90). On GPU, CULA (60) is a unified BLAS/LAPACK package based on nvidia CUDA technology (43).

ATLAS, OpenBLAS and MKL libraries

Two libraries are available for SVD on CPU under Python: NumPy and SciPy. Those packages provide algorithms which are linked to low-level libraries. Under Linux Fedora 22, ATLAS (69) was the default[2]. It was possible to replace it either with OpenBLAS (70) or with MKL (91). Results for a matrix size of $1025 \times 1024 = 1.0e6$ points are presented on Figure 5a for our reference computer with a Core 2 Quad Q8200 and a GTX 260 (2008). First, we noticed that decreased computation times were obtained when moving from ATLAS (left column) to OpenBLAS (middle column) and MKL (right column). While OpenBLAS improved only Scipy results, MKL was almost twice faster than OpenBLAS for both Numpy and Scipy. Secondly, with ATLAS (left column), SciPy computation times (yellow and green lines) were longer than NumPy ones (blue and red lines), both for DP and SP. This behaviour was surprising as SciPy was intended to do some scientific calculation. It may be improved in a newer ATLAS

2 NumPy library can be verified using 'numpy.show_config()'.

version. Third, for all libraries tested, no performance increase was visible with NumPy when changing from DP (blue line) to SP (red line), which may indicate a bug of NumPy. On the contrary, SciPy + SP computation times (green line) were almost half DP ones (yellow line), as expected from DP / SP computing power ratio. Finally, for this small matrix of 1025 × 1024 = 1.0e6 points, and no matter if ATLAS, OpenBLAS or MKL library was installed, CULA + SP (hatched red line) was slower on GPU than MKL + SciPy + SP on CPU.

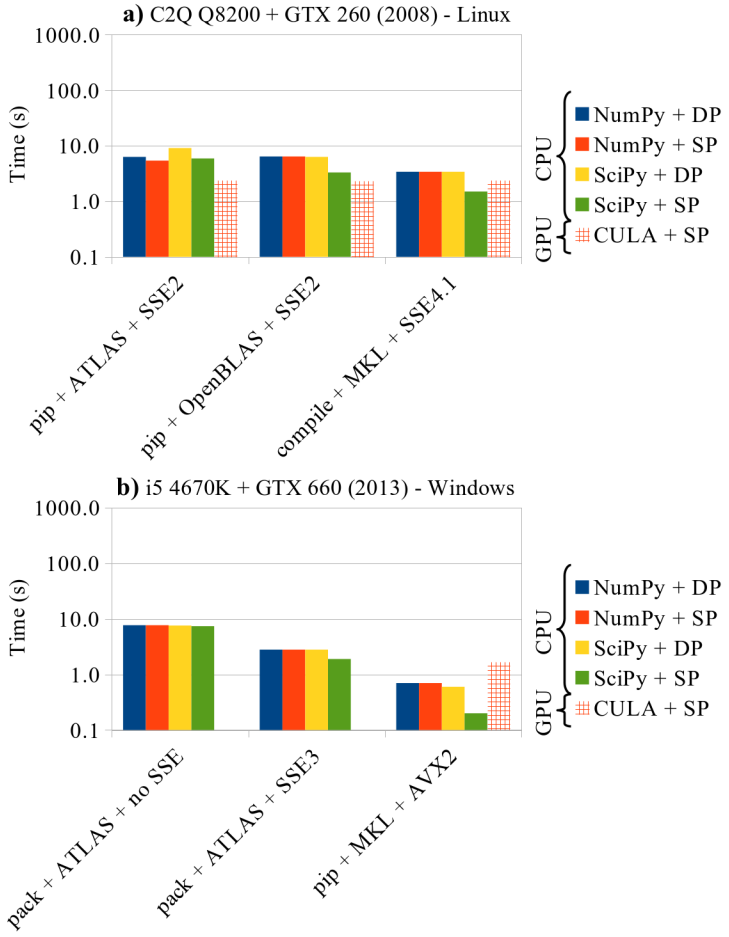


Figure 5: Computation times for **SVD under Python** with influence of libraries and hardware optimisations for a matrix size of 1025 × 1024 = 1.0e6 points. a) C2Q Q8200 + GTX 260 (2008) under Linux. b) i5 4670K + GTX 660 (2013) under Windows.

SSE and AVX hardware instructions

Despite MKL seemed very promising, it was not the only factor changing in the above experiment as the implemented hardware optimisations changed from SSE2[3] to SSE4.1[4]. SSE stands for Streaming SIMD Extensions and its number reflects the version used. SIMD are embedded capabilities on CPU. Since 2008, a new family of instructions is available, named Advanced Vector Extensions (AVX). Even if the processor support them, the library does not necessarily call them. A history of SIMD development is available in reference (51). Under windows, NumPy and SciPy superpack provided options to selectively use no SSE or SSE3[5]. Results are presented on Figure 5b for our reference computer with a Core i5 4670K and a GTX 660 (2013). When moving from no SSE (left column) to SSE3 (middle column), computation times were divided by three with an additional gain for SP. Moreover, when using both MKL and AVX2, a huge performance was obtained, outperforming GPU computation with CULA + SP (hatched red line).

As underlined here, the time needed to perform SVD was impressively reduced by a factor of 38 on the same CPU under Python, by optimising the used libraries and their hardware calls. Indeed, decomposition of a matrix of size 1025 × 1024 = 1.0e6 points was done with SciPy in 7.6 s without optimisations and in 0.2 s using MKL library and AVX2 instructions.

3 SSE2 instruction can be checked with 'objdump -d /lib64/python/site-packages/numpy/core/*.so | grep -i ADDPD'.

4 NumPy and SciPy are compiled with '-xHost' option enabling the highest SIMD. instruction set available, which is SSE4.1 on a Core 2 Quad Q8200.

5 No SSE option is 'numpy-1.10.1-win32-superpack-python2.7.exe /arch nosse'.

Comparison of Java, Matlab and Python

Computation times

In previous sections, Java, Matlab and Python software were used for their specific testing capabilities. But how do they compare to each other? In order to answer this question, Figure 6 shows SVD computation times for a matrix of 1025 × 1024 = 1.0e6 points, which is the maximum size for Java 32 bits CPU application. Similar conclusions were raised for our two reference computers (2008 and 2013). The measured computation times were grouped into three categories: unoptimised CPU (in plain blue), GPU (in hatched red) and optimised CPU (in plain red), from the slowest to the fastest. The first group consisted of Java CPU, Matlab R2010a and Python with default configuration. The second group contained Java GPU, Matlab R2010a with CULA or Matlab R2015a with gpuArray, depending on GPU generation, and of Python with CULA. The third group referred to a recent version of Matlab and to compiled Python, both with MKL library and all available SIMD instructions activated[6]. For this small matrix of 1025 × 1024 = 1.0e6 points, the CPU outperformed the GPU, due to data transfer delays limiting GPU efficiency. However, for larger matrices, computation was faster on GPU.

Comparing Java CPU and Python with MKL, there was a gain of 100 on the same CPU. This was explained as follows:

- a factor of 9 using the divide and conquer algorithm
- a factor of 3 using hardware instructions such as SSE3 or AVX2
- a factor of 2 using MKL library
- a factor of 2 using single precision instead of double precision

6 ‘version('-blas')’ under Matlab gives MKL 11.0.5 for R2014a and MKL 11.1.1 for R2015a.

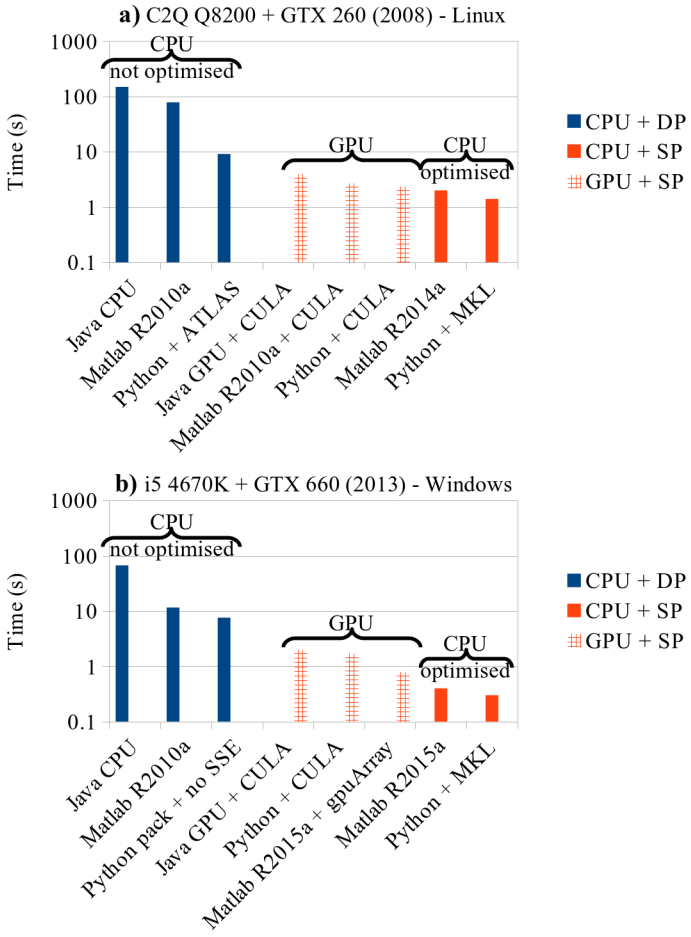


Figure 6: Comparison of **Java, Matlab and Python SVD** computation times for a matrix size of 1025 × 1024 = 1.0e6 points, in logarithmic scale. a) C2Q Q8200 + GTX 260 (2008) under Linux. b) i5 4670K + GTX 660 (2013) under Windows.

Maximum matrix size

This major time improvement raise the question of the absolute maximum matrix size that could be computed using SVD and low-rank approximation. As underlined in section "Java GPU performance indicator", the limiting parameter is memory, both on GPU device side and on CPU host side. The crucial point is to use 64 bits applications and a GPU with as much memory as possible. Nevertheless, this will depend on the way memory is allocated and released during SVD process. Our late investigations, on a GTX 1070 with 8 GB of memory and 7040 SP GFLOPS, gave the following maxima on

our ^{87}Sr FID. The computation times were the sum of SVD and low-rank steps. To maximize the latter, the operation was performed with all singular values, that is to say without any denoising.

- Java GPU: 9217 × 9216 = 8.5e7 complex points in 150 + 32 = 182 s.
- Matlab R2018a + gpuArray + SP: 12289 × 12288 = 1.5e8 complex points in 188 + 4 = 192 s.
- Python + CULA + SP: 15219 × 15218 = 2.3e8 complex points in 574 + 6 = 600 s.

Under Python, it was thus possible to apply SVD on the full ^{87}Sr FID, without any truncation. For comparison, a Nvidia P100 GPU, with 4670 DP GFLOPS and 16 GB of memory, completed the full SVD of a 20000 × 20000 = 4.0e8 real matrix in 90 s, with a highly optimised CPU-GPU algorithm (46). This result was really impressive as the authors obtained a faster computation on a much larger matrix with less computing power and double precision. There is thus plenty of place to improve SVD denoising.

Directly comparing Java, Matlab and Python was a difficult task as they were not optimised in the same way and it was hard to check what was hidden under the hood. However, Java GPU was less performant, both in speed and in matrix size. Better results may be obtained with optimised libraries. While Matlab was faster, the memory usage was limiting. Python computation was not as fast but could handle the biggest matrix. This time advantage for Matlab was explained by a better CPU usage during SVD on GPU. However memory was more finely managed under Python. Our results suggested that the key parameter was not the software and the programming language, but rather the used libraries and the calls to hardware instructions.

An additional advantage of Python is that it is free of charge and rather easy to program. In order to compute SVD in a minimum amount of time, we recommand to install a Python distribution with MKL library included, such as Anaconda (92), and to add the following libraries: SciPy (73), scikit-cuda (67), PyCUDA (68), and nmrglue (93) for NMR data. In addition, CULA (60) and CUDA toolkit (43) packages are necessary. We provide in file svd_auto.py of (64) an optimised SVD function using either the CPU or the GPU, depending on the matrix size. Our tests suggested a minimum value of 4096 columns or rows to switch from the CPU to the GPU. This default value will depend on the hardware used and can be checked by running directly the program. The code is designed to be as simple as possible, with only one necessary parameter, namely the matrix two-dimensional array. Automatic thresholding is applied using Malinowski's significant level indicator (32). This SVD function is also suitable to be used in PCA and related data mining techniques. In addition, we provide a second program (file denoise_nmr.py of (64)), in charge of importing and exporting Bruker NMR data and to prepare the matrix transferred to SVD program. Again, the only requested parameter is the data directory.

Conclusion

This article separated in two parts focussed on SVD, which is used both for spectra denoising and as part of PCA data mining. In the first part, we gave theoretical background and found the minimum experimental signal-to-noise ratio needed to have a correct denoised spectrum. We highlighted the overestimation of denoised Gaussian peaks. In this second part, we focussed on the computation time needed for SVD treatment. While our first attempts under Java CPU were extremely slow even with a recent processor, their counterparts with graphic cards were extraordinary fast. This

unexpected difference led us to check if different Matlab versions could improve this situation. The divide and conquer algorithm was very helpful. Additional tests were undergone under Python to check the influence of software libraries and of SIMD hardware instructions call. Combining these optimisations, computation times on processor were even better than on graphic cards, being 100 times faster than our first tests under Java CPU, for a matrix of $1025 \times 1024 = 1.0e6$ points. Despite this approach is generalisable to any intensive computation, specific time gain will depend on the involved mathematical operations. The take-home message is thus to update software and to use optimised libraries and especially Intel MKL if available. This choice should be preferred against hardware updates.

However, for matrix above $4097 \times 4096 = 1.7e7$ points and middle range hardware, GPU gave better results, up to GPU memory limit. We thus provided Python programs to apply SVD either on CPU or on GPU, and to denoise NMR FID. Further improvement could be obtained with mixed CPU/GPU optimised code, *i.e.* hybrid computing (94). However, such an approach is not suitable for non-computer-scientists people. Using clMAGMA library (95), combining divide and conquer on both CPU and GPU could be a good alternative (96). In this case, it would be possible not only to use Nvidia GPU with CUDA but also AMD GPU with OpenCL.

This study has given strong background and optimisations for experiments involving SVD, either for denoising or for PCA. It may thus help scientists who want to use efficiently this technique, which is expected to be widely used in the forthcoming years.

Supplementary material

Programs source codes are available online in (64).

Acknowledgements

The French Région Ile de France – SESAME program is acknowledged for financial support (700 MHz spectrometer). Sylvie Masse and Cedric Lorthioir are thanked for fruitful discussions.

References

1. Bonhomme, C., Gervais, C., and Laurencin, D. (2014) Recent NMR developments applied to organic–inorganic materials. *Prog. Nucl. Magn. Reson. Spectrosc.* 77: 1–48.

2. Das, R. S., and Agrawal, Y. K. (2011) Raman spectroscopy: Recent advancements, techniques and applications. *Vib. Spectrosc.* 57 (2): 163–176.

3. Levitt, M. H. (2008) *Spin dynamics: basics of nuclear magnetic resonance*. 2nd ed. John Wiley & Sons Ltd: Chichester, England.

4. Gautam, R., Samuel, A., Sil, S., Chaturvedi, D., Dutta, A., Ariese, F., and Umapathy, S. (2015) Raman and mid-infrared spectroscopic imaging: applications and advancements. *Curr. Sci.* 108 (3): 341–356.

5. Laurent, G., Woelffel, W., Barret-Vivin, V., Gouillart, E., and Bonhomme, C. Denoising applied to spectroscopies – part I: concept and limits. *Appl. Spectrosc. Rev.* accepted.

6. Tayler, M. C. D., Meerten, S. (Bas) G. J. van, Kentgens, A. P. M., and Bentum, P. J. M. van. (2015) Analysis of mass-limited mixtures using supercritical-fluid chromatography and microcoil NMR. *Analyst* 140 (18): 6217–6221.

7. Cardell, C., and Guerra, I. (2016) An overview of emerging hyphenated SEM-EDX and Raman spectroscopy systems: applications in life, environmental and materials sciences. *TrAC Trends Anal. Chem.* 77: 156–166.

8. Ali, M. R. K., Wu, Y., Han, T., Zang, X., Xiao, H., Tang, Y., Wu, R., Fernández, F. M., and El-Sayed, M. A. (2016) Simultaneous time-dependent surface-enhanced raman spectroscopy, metabolomics, and proteomics reveal cancer cell death mechanisms associated with gold nanorod photothermal therapy. *J. Am. Chem. Soc.* 138 (47): 15434–15442.

9. Marchand, J., Martineau, E., Guitton, Y., Dervilly-Pinel, G., and Giraudeau, P. (2017) Multidimensional NMR approaches towards highly resolved, sensitive and high-throughput quantitative metabolomics. *Curr. Opin. Biotechnol.* 43: 49–55.

10. Gemperline, P. (2006) *Practical guide to chemometrics*. 2nd ed. CRC Press, Taylor & Francis Group: Boca Raton, FL, USA.

11. Elmi Rayaleh, W. (2006) *Extraction des connaissances en imagerie microspectrométrique par analyse chimiométrique : application à la caractérisation des constituants d'un calcul urinaire* (PhD dissertation), Université des sciences et technologies de Lille 1, France.

12. Chen, X., Vorvoreanu, M., and Madhavan, K. (2014) Mining social media data for understanding students' learning experiences. *IEEE Trans. Learn. Technol.* 7 (3): 246–259.

13. Wang, L., Dong, X., Cheng, X., and Lin, S. (2018) An improved coupled dictionary and multi-norm constraint fusion method for CT/MR medical images. *Multimed. Tools Appl.*: 1–17.

14. Wold, S., Esbensen, K., and Geladi, P. (1987) Principal component analysis. *Chemom. Intell. Lab. Syst.* 2 (1–3): 37–52.

15. Wu, W., and Manne, R. (2000) Fast regression methods in a Lanczos (or PLS-1) basis. Theory and applications. *Chemom. Intell. Lab. Syst.* 51 (2): 145–161.

16. Geladi, P., and Kowalski, B. R. (1986) Partial least-squares regression: a tutorial. *Anal. Chim. Acta* 185: 1–17.

17. Gromski, P. S., Muhamadali, H., Ellis, D. I., Xu, Y., Correa, E., Turner, M. L., and Goodacre, R. (2015) A tutorial review: metabolomics and partial least squares-discriminant analysis – a marriage of convenience or a shotgun wedding. *Anal. Chim. Acta* 879: 10–23.

18. Comon, P. (1994) Independent component analysis, A new concept? *Signal Process.* 36 (3): 287–314.

19. Woelffel, W., Claireaux, C., Toplis, M. J., Burov, E., Barthel, É., Shukla, A., Biscaras, J., Chopinet, M.-H., and Gouillart, E. (2015) Analysis of soda-lime glasses using non-negative matrix factor deconvolution of Raman spectra. *J. Non-Cryst. Solids* 428: 121–131.

20. Monakhova, Y. B., Godelmann, R., Kuballa, T., Mushtakova, S. P., and Rutledge, D. N. (2015) Independent components analysis to increase efficiency of discriminant analysis methods (FDA and LDA): application to NMR fingerprinting of wine. *Talanta* 141: 60–65.

21. Maione, C., and Barbosa, R. M. (2018) Recent applications of multivariate data analysis methods in the authentication of rice and the most analyzed parameters: a review. *Crit. Rev. Food Sci. Nutr.*: 1–12.

22. Wang, M., Raman, V., Zhao, J., Avula, B., Wang, Y.-H., Wylie, P. L., and Khan, I. A. (2018) Application of GC/Q-TOF combined with advanced data mining and chemometric tools in the characterization and quality control of bay leaves. *Planta Med.*: online.

23. D. Vimalajeewa, D. Berry, E. Robson, and C. Kulatunga. (2017) Evaluation of non-linearity in MIR spectroscopic data for compressed learning. *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, November 18.

24. Othman, F., Chowdhury, M. S., Wan Jaafar, W. Z., Faresh, E. M. M., and Shirazi, S. M. (2018) Assessing risk and sources of heavy metals in a tropical river basin: a case study of the Selangor river, Malaysia. *Pol. J. Environ. Stud.* 27 (4): 1659–1671.

25. Orfanidis, S. J. (2002) *SVD, PCA, KLT, CCA, and all that*. Available at: http://www.ece.rutgers.edu/~orfanidi/ece525/svd.pdf (accessed 14 May 2018).

26. Shlens, J. (2014) *A tutorial on principal component analysis*. Available at: http://arxiv.org/abs/1404.1100 (accessed 14 May 2018).

27. Tufts, D. W., Kumaresan, R., and Kirsteins, I. (1982) Data adaptive signal estimation by singular value decomposition of a data matrix. *Proc. IEEE* 70 (6): 684–685.

28. Cadzow, J. A. (1988) Signal enhancement – a composite property mapping algorithm. *IEEE Trans. Acoust. Speech Signal Process.* 36 (1): 49–62.

29. Laurent, G. (2016) SVD Performances to denoise NMR and Raman spectra. Available at: https://hal.sorbonne-universite.fr/hal-01277387 (accessed 14 May 2018).

30. Tufts, D. W., and Shah, A. A. (1993) Estimation of a signal waveform from noisy data using low-rank approximation to a data matrix. *IEEE Trans. Signal Process.* 41 (4): 1716–1721.

31. Heinig, G., and Rost, K. (1984) Toeplitz and Hankel matrices. In *Algebraic methods for Toeplitz-like matrices and operators*, Birkhäuser Basel, pp 9–135.

32. Malinowski, E. R. (2002) *Factor analysis in chemistry*. 3rd ed. Wiley: Hoboken, NJ, USA.

33. Cavallaro, J. R., and Luk, F. T. (1988) CORDIC arithmetic for an SVD processor. *J. Parallel Distrib. Comput.* 5 (3): 271–290.

34. Vogt, F., and Tacke, M. (2001) Fast principal component analysis of large data sets. *Chemom. Intell. Lab. Syst.* 59 (1–2): 1–18.

35. Gu, M., and Eisenstat, S. (1995) A Divide-and-Conquer Algorithm for the Bidiagonal SVD. *SIAM J. Matrix Anal. Appl.* 16 (1): 79–92.

36. Halko, N., Martinsson, P., and Tropp, J. (2011) Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIAM Rev.* 53 (2): 217–288.

37. Condat, L., and Hirabayashi, A. (2015) Cadzow denoising upgraded: a new projection method for the recovery of Dirac pulses from noisy linear measurements. *Sampl. Theory Signal Image Process.* 14 (1): 17–47.

38. Bourgoin, M., Chailloux, E., and Lamotte, J.-L. (2017) High level data structures for GPGPU programming in a statically typed language. *Int. J. Parallel Program.* 45 (2): 242–261.

39. Andrecut, M. (2008) Parallel GPU implementation of iterative PCA algorithms. *J. Comput. Biol.* 16 (11): 1593–1599.

40. Lahabar, S., and Narayanan, P. J. (2009) Singular value decomposition on GPU using CUDA. *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, May.

41. Novakovic, V., and Singer, S. (2011) A GPU-based hyperbolic SVD algorithm. *BIT Numer. Math.* 51 (4): 1009–1030.

42. Irofti, P., and Dumitrescu, B. (2014) GPU parallel implementation of the approximate K-SVD algorithm using OpenCL. *2014 22nd European Signal Processing Conference (EUSIPCO)*, September.

43. Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008) Scalable parallel programming with CUDA. *Queue* 6 (2): 40–53.

44. Khronos group. (2009) OpenCL parallel computing for heterogeneous devices. *IEEE Hot Chips 21 Symposium (HCS)*, August.

45. Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Tomov, S., and Yamazaki, I. (2014) Accelerating numerical dense linear algebra calculations with GPUs. In *Numerical Computations with GPUs*, Kindratenko, V., Ed., Springer International Publishing, pp 3–28.

46. Gates, M., Tomov, S., and Dongarra, J. (2018) Accelerating the SVD two stage bidiagonal reduction and divide and conquer using GPUs. *Parallel Comput.* 74: 3–18.

47. Xie, G., and Zhang, Y. l. (2017) A Few of the Most Popular Models for Heterogeneous Parallel Programming. *2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES)*, October.

48. Man, P. P., Bonhomme, C., and Babonneau, F. (2014) Denoising NMR time-domain signal by singular-value decomposition accelerated by graphics processing units. *Solid State Nucl. Magn. Reson.* 61–62: 28–34.

49. Man, P. P. (2012) 2012 Java application for FID denoising with SVD. Available at: http://pascal-man.com/navigation/faq-java-browser/SVD-Java-application2012.shtml (accessed 14 May 2018).

50. Man, P. P. (2012) Graphic processing unit (GPU), 2012 SVD Java 7 application for FID denoising. Available at: http://www.pascal-man.com/navigation/faq-java-browser/SVD-Java-application-GPU.shtml (accessed 14 May 2018).

51. Hofmann, J., Treibig, J., Hager, G., and Wellein, G. (2014) Comparing the performance of different x86 SIMD instruction sets for a medical imaging

application on modern multi- and manycore chips. *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, ACM: New York, NY, USA.

52. Faustini, M., Nicole, L., Boissière, C., Innocenzi, P., Sanchez, C., and Grosso, D. (2010) Hydrophobic, antireflective, self-cleaning, and antifogging sol−gel coatings: an example of multifunctional nanostructured materials for photovoltaic cells. *Chem. Mater.* 22 (15): 4406–4413.

53. Malfait, W. J., and Halter, W. E. (2008) Increased ^{29}Si NMR sensitivity in glasses with a Carr–Purcell–Meiboom–Gill echotrain. *J. Non-Cryst. Solids* 354 (34): 4107–4114.

54. Laurencin, D., Ribot, F., Gervais, C., Wright, A. J., Baker, A. R., Campayo, L., Hanna, J. V., Iuga, D., Smith, M. E., Nedelec, J.-M., Renaudin, G., and Bonhomme, C. (2016) ^{87}Sr, ^{119}Sn, ^{127}I single and {^{1}H/^{19}F}-double resonance solid-state NMR experiments: application to inorganic materials and nanobuilding blocks. *ChemistrySelect* 1 (15): 4509–4519.

55. Iuga, D., Schäfer, H., Verhagen, R., and Kentgens, A. P. M. (2000) Population and coherence transfer induced by double frequency sweeps in half-integer quadrupolar spin systems. *J. Magn. Reson.* 147 (2): 192–209.

56. Schurko, R. W. (2013) Ultra-wideline solid-state NMR spectroscopy. *Acc. Chem. Res.* 46 (9): 1985–1995.

57. TDR registry keys. Available at: https://msdn.microsoft.com/en-us/Library/Windows/Hardware/ff569918(v=vs.85).aspx (accessed 14 May 2018).

58. van Rossum, G. (1995) *Python Tutorial*. (Technical report Report CS-R9526) Available at: https://gvanrossum.github.io//Publications.html (accessed 14 May 2018).

59. Stewart, G. W. Jampack: a Java matrix package. Available at: ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html (accessed 14 May 2018).

60. Humphrey, J. R., Price, D. K., Spagnoli, K. E., Paolini, A. L., and Kelmelis, E. J. (2010) CULA: hybrid GPU accelerated linear algebra routines. *Proc. SPIE 7705*, *Modeling and Simulation for Defense Systems and Applications V*, Orlando, FL, USA.

61. Javin, P. (2013) What is the maximum Heap Size of 32 bit or 64-bit JVM in Windows and Linux? Available at: http://javarevisited.blogspot.com/2013/04/what-is-maximum-heap-size-for-32-bit-64-JVM-Java-memory.html (accessed 14 May 2018).

62. Laurent, G. (2015) SVD under Matlab with CULA link. Available at: http://www.culatools.com/forums/viewtopic.php?p=2423#p2423 (accessed 14 May 2018).

63. van Beek, J. D. (2007) matNMR: A flexible toolbox for processing, analyzing and visualizing magnetic resonance data in Matlab®. *J. Magn. Reson.* 187 (1): 19–26.

64. Laurent, G., Gilles, P.-A., Woelffel, W., Barret-Vivin, V., Gouillart, E., and Bonhomme, C. (2018) Denoising applied to spectroscopies: parts I and II - programs and datas. Available at: http://doi.org/10.5281/zenodo.1406172 (accessed 30 August 2018).

65. Oliphant, T. E. (2006) *A guide to NumPy*. Trelgol Publishing USA.

66. Jones, E., Oliphant, T., and Peterson, P. (2001) SciPy: open source scientific tools for Python. Available at: http://www.scipy.org/ (accessed 14 May 2018).

67. Givon, L. E., Unterthiner, T., Erichson, N. B., Chiang, D. W., Larson, E., Pfister, L., Dieleman, S., Lee, G. R., Walt, S. van der, Menn, B., Moldovan, T. M., Bastien, F., Shi, X., Schlüter, J., Thomas, B., Capdevila, C., Rubinsteyn, A., Forbes, M. M., Frelinger, J., Klein, T., Merry, B., Pastewka, L., Taylor, S., Wang, F., and Zhou, Y. (2015) scikit-cuda 0.5.1: a Python interface to GPU-powered libraries. Available at: http://dx.doi.org/10.5281/zenodo.40565.

68. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., and Fasih, A. (2012) PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.* 38 (3): 157–174.

69. Whaley, R. C., and Dongarra, J. (1997) *Automatically Tuned Linear Algebra Software*. (Technical report UT-CS-97-366).

70. Xianyi, Z., Qian, W., and Yunquan, Z. (2012) Model-driven level 3 BLAS performance optimization on Loongson 3A processor. *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, December.

71. Gehrcke, J.-P. (2014) Building NumPy and SciPy with Intel compilers and Intel MKL on a 64 bit machine. Available at: https://gehrcke.de/2014/02/building-numpy-and-scipy-with-intel-compilers-and-intel-mkl-on-a-64-bit-machine/ (accessed 14 May 2018).

72. Numerical Python. (2015) Available at: https://sourceforge.net/projects/numpy/files/NumPy/1.10.1/ (accessed 14 May 2018).

73. SciPy: Scientific library for Python. (2015) Available at: https://sourceforge.net/projects/scipy/files/scipy/0.16.1/ (accessed 14 May 2018).

74. Gohlke, C. Python extension packages for windows. Available at: http://www.lfd.uci.edu/~gohlke/pythonlibs/ (accessed 14 May 2018).

75. Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. (2010) Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *Proceedings of the 37th Annual International*

Symposium on Computer Architecture, *ISCA '10*, ACM: Saint-Malo, France, June 19.

76. Dolbeau, R. (2018) Theoretical peak FLOPS per instruction set: a tutorial. *J. Supercomput.* 74 (3): 1341–1377.

77. Whitehead, N., and Fit-Florea, A. (2017) *Precision & performance: floating point and IEEE 754 compliance for NVIDIA GPUs*. (TB-06711-001_v8.0) Available at: http://docs.nvidia.com/pdf/Floating_Point_on_NVIDIA_GPU.pdf (accessed 14 May 2018).

78. Stewart, G. W. (1990) *Perturbation theory for the singular value decomposition*. (Technical Report CS-TR-2539) Available at: http://hdl.handle.net/1903/552 (accessed 15 May 2018).

79. Plassman, G. E. (2005) *A survey of singular value decomposition methods and performance comparison of some available serial codes*. Available at: https://ntrs.nasa.gov/search.jsp?R=20050192421 (accessed 14 May 2018).

80. Läuchli, P. (1961) Jordan-elimination und ausgleichung nach kleinsten quadraten. *Numer. Math.* 3 (1): 226–240.

81. Golub, G., and Kahan, W. (1965) Calculating the singular values and pseudo-inverse of a matrix. *J. Soc. Ind. Appl. Math. Ser. B Numer. Anal.* 2 (2): 205–224.

82. Golub, G. H., and Reinsch, C. (1970) Singular value decomposition and least squares solutions. *Numer. Math.* 14 (5): 403–420.

83. Cuppen, J. J. M. (1980) A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.* 36 (2): 177–195.

84. Willems, P. R., Lang, B., and Vömel, C. (2006) Computing the bidiagonal SVD using multiple relatively robust representations. *SIAM J. Matrix Anal. Appl.* 28 (4): 907–926.

85. Demmel, J., and Veselić, K. (1992) Jacobi's method is more accurate than QR. *SIAM J. Matrix Anal. Appl.* 13 (4): 1204–1245.

86. Gu, M., Demmel, J., and Dhillon, I. (1994) *Efficient computation of the singular value decomposition with applications to least squares problems*. (Technical report ut-cs-94-257-LAPACK Working Note 88) Available at: https://library.eecs.utk.edu/pub/463 (accessed 14 May 2018).

87. MathWorks parallel computing toolbox team. (2012) GPUBench. Available at: http://fr.mathworks.com/matlabcentral/fileexchange/34080-gpubench (accessed 14 May 2018).

88. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999) *LAPACK Users' Guide*. 3rd ed. Society for Industrial and Applied Mathematics: Philadelphia, PA, USA.

89. Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. (1990) A set of level 3 basic linear algebra subprograms. *ACM Trans Math Softw* 16 (1): 1–17.

90. Hogben, L. ed. (2006) *Handbook of Linear Algebra*. 1st ed. Chapman and Hall/CRC: Boca Raton.

91. Nguyen, V. (2014) Optimized R and Python: standard BLAS vs. ATLAS vs. OpenBLAS vs. MKL. Available at: http://blog.nguyenvq.com/blog/2014/11/10/optimized-r-and-python-standard-blas-vs-atlas-vs-openblas-vs-mkl/ (accessed 14 May 2018).

92. Anaconda. (2016) Anaconda software distribution. Available at: https://www.anaconda.com/ (accessed 14 May 2018).

93. Helmus, J. J., and Jaroniec, C. P. (2013) Nmrglue: an open source Python package for the analysis of multidimensional NMR data. *J. Biomol. NMR* 55 (4): 355–367.

94. Liu, D., Li, R., Lilja, D. J., and Xiao, W. (2013) A divide-and-conquer approach for solving singular value decomposition on a heterogeneous system. *Proceedings of the ACM International Conference on Computing Frontiers*, ACM: New York, NY, USA.

95. Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., and Dongarra, J. (2012) From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.* 38 (8): 391–407.

96. Haidar, A., Cao, C., Yarkhan, A., Luszczek, P., Tomov, S., Kabir, K., and Dongarra, J. (2014) Unified development for mixed multi-GPU and multi-coprocessor environments using a lightweight runtime environment. *Parallel and distributed processing symposium, 2014 IEEE 28th international*, Phoenix, AZ, USA, May 19.