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
**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 × 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\textsubscript{max}) needed to have reliable results was PSNR\textsubscript{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 $2015 \times 2014 = 4.1e6$. 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.
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$_3$F) 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
$1025 \times 1024 = 1.0e6$ points, whatever the CPU used. The same problem was observed with GPU 32 bits version above a matrix size of $3005 \times 1024 = 3.1e6$ 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 mechanism¹. 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.

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

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.

<table>
<thead>
<tr>
<th>Graphics Processing Unit (GPU)</th>
<th>Type</th>
<th>Year</th>
<th>Fabrication (nm)</th>
<th>Number of cores</th>
<th>Bandwidth (GB/s)</th>
<th>Core frequency (MHz)</th>
<th>Memory frequency (MHz)</th>
<th>Memory size (MB)</th>
<th>Single precision float (GFLOPS)</th>
<th>Double precision float (GFLOPS)</th>
<th>CUDA compute capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nvidia Quadro FX 570 desktop</td>
<td>2007</td>
<td>80</td>
<td>16</td>
<td>12.8</td>
<td>460</td>
<td>400</td>
<td>256</td>
<td>29</td>
<td>#N/A</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>Nvidia GeForce 8400 GS desktop</td>
<td>2008</td>
<td>65</td>
<td>8</td>
<td>6.4</td>
<td>567</td>
<td>400</td>
<td>512</td>
<td>21</td>
<td>#N/A</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>Nvidia Quadro NVS 160M laptop</td>
<td>2008</td>
<td>65</td>
<td>8</td>
<td>11.2</td>
<td>580</td>
<td>700</td>
<td>256</td>
<td>23</td>
<td>#N/A</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>Nvidia Quadro FX 770M laptop</td>
<td>2008</td>
<td>65</td>
<td>32</td>
<td>25.6</td>
<td>500</td>
<td>800</td>
<td>512</td>
<td>79</td>
<td>#N/A</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>Nvidia GeForce GTX 260 desktop</td>
<td>2008</td>
<td>65</td>
<td>216</td>
<td>111.9</td>
<td>576</td>
<td>1000</td>
<td>896</td>
<td>533</td>
<td>67</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>Nvidia GeForce 820M laptop</td>
<td>2012</td>
<td>28</td>
<td>96</td>
<td>14.4</td>
<td>625</td>
<td>900</td>
<td>2048</td>
<td>315</td>
<td>31</td>
<td>2.1</td>
<td></td>
</tr>
<tr>
<td>Nvidia GeForce GTX 660 desktop</td>
<td>2012</td>
<td>28</td>
<td>960</td>
<td>144.2</td>
<td>1100</td>
<td>1500</td>
<td>2048</td>
<td>1707</td>
<td>88</td>
<td>3.0</td>
<td></td>
</tr>
</tbody>
</table>
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 $1537 \times 512 = 7.9e5$ vs. $1025 \times 1024 = 1.0e6$ points, and similarly for GPU at $3005 \times 1024 = 3.1e6$ vs. $2015 \times 2014 = 4.1e6$ 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 $4097 \times 4096 = 1.7e7$ 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 $3005 \times 1024 = 3.1e6$ 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 n² + 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 × 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 $1025 \times 1024 = 1.0e6$ 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 $1025 \times 1024 = 1.0e6$ points for Core 2 Quad Q8200, and up to $3073 \times 3072 = 9.4e6$ 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 $4097 \times 4096 = 1.7e7$ 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 $7169 \times 7168 = 5.1e7$ 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 $1025 \times 1024 = 1.0e6$ 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. 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 $1025 \times 1024 = 1.0e6$ 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 $1025 \times 1024 = 1.0e6$ 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 SSE23 to SSE4.14. 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 SSE35. 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 $1025 \times 1024 = 1.0e6$ 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 $1025 \times 1024 = 1.0e6$ 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 $1025 \times 1024 = 1.0e6$ 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 $1025 \times 1024 = 1.0e6$ 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**: $9217 \times 9216 = 8.5e7$ complex points in $150 + 32 = 182$ s.
- **Matlab R2018a + gpuArray + SP**: $12289 \times 12288 = 1.5e8$ complex points in $188 + 4 = 192$ s.
- **Python + CULA + SP**: $15219 \times 15218 = 2.3e8$ 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 $20000 \times 20000 = 4.0e8$ 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 recommend 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 cIMAGMA 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


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).


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


