A Simple Compressive Sensing Algorithm for Parallel Many-Core Architectures

Alexandre Borghi [1] Jérôme Darbon[2] Sylvain Peyronnet[1]
Tony F. Chan[2] Stanley Osher[2]

[1]Laboratoire de Recherche en Informatique (LRI), Université Paris Sud, France
[2]Department of Mathematics, UCLA, Los Angeles, California, USA

Original version: September 2008, revised version: August 2009

Abstract

In this paper we consider the l^1-compressive sensing problem. We propose an algorithm specifically designed to take advantage of shared memory, vectorized, parallel and many-core microprocessors such as the Cell processor, new generation Graphics Processing Units (GPUs) and standard vectorized multi-core processors (e.g. quad-core CPUs). Besides its implementation is easy. We also give evidence of the efficiency of our approach and compare the algorithm on the three platforms, thus exhibiting pros and cons for each of them.

1 Introduction

An algorithm that relies on standard optimization techniques is proposed for solving compressive sensing problems involving l^1 minimization. Its main characteristic is that it is designed for running efficiently on parallel many-core architectures and thus takes huge benefit from these technologies.

Let us briefly describe the compressive sensing problem. Given a sensing matrix $A \in \mathbb{R}^{m \times n}$ and some observed data $f \in \mathbb{R}^m$, one wishes to find an optimal solution $u^* \in \mathbb{R}^n$ of the following constraint minimization problem:

$$\begin{cases} \min_u \|u\|_1 \\ s.t.\ Au = f \end{cases} \tag{1}$$

where the number of constraints m is typically much lower than the size of the signal n. This problem has received a lot of attention since it has application for instance in signal/image processing, data compression and more recently as compressive sensing [8, 9, 10, 19, 50]. The latter corresponds to identify matrices that allow for exact recovery of the original signal which is assumed to be sparse, from the observations. Such matrices are for instance proposed in [8, 10, 16, 19, 3]. This approach has led to several applications such as machine learning [30, 37], dimension reduction [27], face recognition [54], image processing [11] and medical imaging [36]. There is a very large literature for algorithms to solve this problem [13, 18, 25, 28, 29, 32, 49, 51, 50, 55] for instance. We refer the reader to these for more detailed reviews of available algorithms. We note that many of them rely on iterative based thresholding/shrinkage approach that has been originally presented in [44] and [12] and with further developments and improvements such as [6, 7, 13, 22, 24, 15, 21]. This paper also falls into this category. However, contrary to these previous approaches, we focus on the practical considerations of such an approach (as opposed to pure theoretical ones) and our goal is to design an efficient algorithm that takes benefit from available computing technologies.

More precisely, we consider a standard Moreau-Yosida regularization as originally proposed in the seminal work of [39] and [56]. The latter yields an iterative algorithm that is also known as the proximal point iterations [13, 20, 31, 34] that requires the inversion of a non-linear operator. We show that the latter can be efficiently performed by a shrinkage approach by defining an appropriate Moreau-Yosida regularization. The convergence of the method is assured by the theoretical properties of the Moreau-Yosida regularization [31, 34]. This approach allows for an efficient implementation on parallel, shared memory and vectorized architectures. The latter are now standardly available and lead to tremendous speedups compared to unary non-vectorized processors. Eventually, we note the work of [38] that also performs proximal point iterations for solving the compressive sensing problem but on a dual formulation.

Note that in general, efficient implementations of a numerical algorithm on these parallel shared memory platforms are difficult to develop due to technical specificities of these architectures. However, our algorithm has been designed keeping in mind these technical issues so that it allows a short, easy and efficient implementation on these platforms. Experiments on three different parallel many-core architectures, namely vectorized multi-core processors, Graphic Processor Unit processors (GPUs) and the Cell Broadband Engine Architecture microprocessor (Cell), are presented and demonstrate the effectiveness of the approach.

The structure of the paper is as follows: section 2 describes main concepts of parallel computing and essential features of many-core architectures that must be taken into account while designing an algorithm intended to yield an easy and fast implementation to run on them. Our algorithm is described in section 3. Experiments along with comparisons with several different architectures are presented in section 4. Finally we draw some conclusions in section 5.

2 Parallel Architectures

We recall that our goal is to design an algorithm that benefits from current parallel architectures. In this section, we present general concepts about parallel computing along with their implementations in current parallel architectures we considered, before presenting the algorithm designed for them in section 3.

2.1 General parallel computing concepts

In order to design an efficient algorithm on current parallel architectures, we need to understand the main concepts they rely on. We now describe these concepts.

Coarse parallelism. Each platform features several computing units, thus tasks can be spread over them in order to improve performance. However, developers must take care of locks that are likely to arise in this context. A lock is used to prevent at least two computing units to update the same space of memory. It is well known that the use of locks to force unique access to data severely deteriorate performances. A good way to avoid the use of locks, and thus to enhance the performance, is to schedule operations so that they never interfere with each others. This general issue is known as the synchronisation problem and is one of the most important problem regarding performance.

Vectorization. Another level of parallelism, finer than the previous one, consists of processing the data as a vector. This technique receives the name of SIMD. Some architectures implement this technique and rely on the so-called vector units. Performances are greatly improved through the use of these vector units, provided the fact that data are well aligned (data must be accessed at specific adresses in memory). Note that not all algorithms can take benefit from this technique because they are not intrinsically built on vectorial primitives. Besides, even though the latter holds, the alignment property still needs to be fullfilled. In other words, it means that this requirement must be taken into account during the design phase of the algorithm. Our algorithm is taking advantage of this technique.

Memory considerations. Memory limitations are the main constraints of considered architectures, namely bandwidth, latency and cache memory issues. Recall that these platforms have two kinds of memory : cache memory, which is typically structured in different levels, and standard RAM. Cache memory has for goal to reduce the time needed to access data in RAM. Cache memory is an order of magnitude faster than RAM but also a lot smaller in terms of capacity storage. To perform computations on large inputs, data parts must be transfered back and forth between cache memory and RAM. This implies memory communications that can be a bottleneck of the method due to limited available bandwidth and latency. To efficiently achieve this memory management, implementations have to take into account the structure of the cache, i.e., *the cache hierarchy*. Indeed, an algorithm and its implementation should help a processor to maintain this structure and facilitate the prediction of data moves. Since several cores might require simultaneous memory accesses, performances can be limited by the available bandwidth. To cope with this problem, it is needed to satisfactorily schedule data transfers. If these memory issues are not taken into account then computing units may wait for the data to be processed. This is called computation starvation since a processor is not computing. To avoid this behavior, the frequency and the size of data transfers must be tuned to have no computing units without data.

2.2 Parallel architectures

We now explain how current parallel architectures implement concepts just described above. We consider three different architectures: standard vectorized multi-core CPU, modern GPU and the Cell Broadband Architecture (CBEA or Cell for short). These three platforms share common features. First, they are all based on a shared memory, i.e., a central memory accessible from all computing units. Second, they are parallel, i.e., composed of several computing units working together. However, these architectures do not implement parallelism in the same way. This leads to very different performances in both terms of computational power and memory transfer, which are two key aspects for global performance.

We now describe more precisely each of these architectures in view of the above considerations.

Multi-core CPU. A standard multi-core CPU is a processor made of several cores. Each core is superscalar, out-of-order and composed of vector computing units (such as Altivec [17] or Intel Streaming SIMD Extensions). From a parallel computing point of view, main differences between current commercial multi-core CPUs are core interconnections, cache hierarchy and how they access RAM.

The Intel Core 2 has cores clustered by 2. Each core has its own very fast level 1 cache (L1) and each cluster features a shared L2 cache. Clusters are linked together through a Front-Side Bus (FSB) with motherboard northbridge, which integrates the memory controller used to access RAM. The Core 2 Quad features 2 clusters of 2 cores. Because the 2 clusters are not connected through an internal crossbar nor a shared cache, cache coherency is maintained thanks to the motherboard northbridge. This implies slower communications between cores of separate clusters than cores of a unique cluster and more generally nefast repercussions on performance in a parallel context.

On the contrary, the Intel Core i7 has a unique cluster for all of its cores and a L3 cache shared over them in addition to their own L2. Thanks to this shared L3 cache, better cache use can be achieved when several threads work on the same data. The FSB is replaced by a QuickPath Interconnect (QPI), which is a point-to-point interconnection, and the memory controller is now on-die. That allows both reduced latency and higher bandwidth. More, the Core i7 has 2 important features from a parallel computing point of view : Turbo Boost (TB) and Hyper-Threading (HT). The first one consists of a dynamic increase of frequency for active cores when some cores are inactive. When enabled this feature must be taken into accoung as it acts as a bias in parallel results. The second one allows the OS to see 2 logical cores for each physical core, which can yield better uses of computational units : as 2 threads have independant instructions the processor pipeline can be filled more efficently and the out-of-order engine can achieve better performance. However, handling 2 threads per core can increase pressure on cache (i.e., more cache misses per core) and then on RAM.

Core 2 and Core i7 are shown in figure 1 in their 4 cores configuration. Current high-end quad-core processors achieve a theoretical peak performance of 96 GFLOPS in single precision at 3GHz.

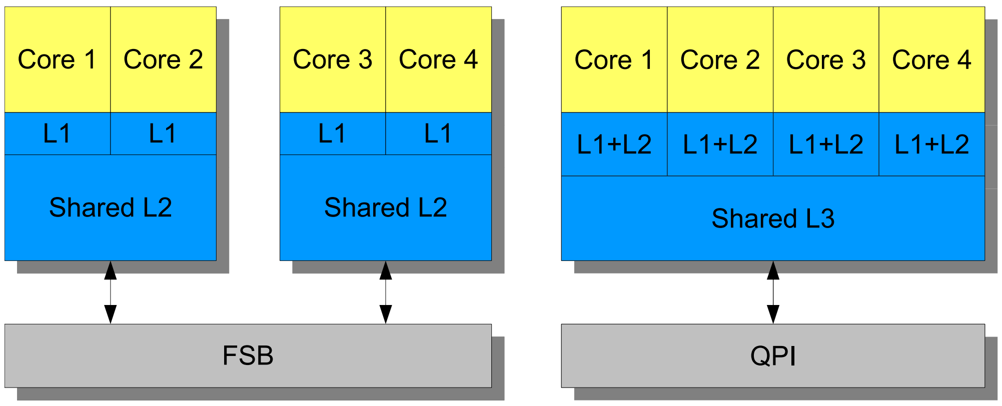


Figure 1: Architectural schemes of Intel multi-core CPUs (left : Core 2 Quad; right : Core i7). See text for details.

Graphics Processing Unit (GPU). A modern GPU (such as current NVIDIA GPUs) consists of a large set of the so-called stream processors that share a common memory [40]. These homogeneous processors have been originally designed for 3D graphics computations and thus had poor scientific computation units in terms of precision.

They now have the capabilities (in terms of precision) to achieve good performances for scientific computing. The latter is called General-Purpose GPU (GPGPU), see [46]. Note that a GPU is hosted by a standard CPU-based computer that embeds some RAM.

NVIDIA GPU architectures are depicted in figure 2. Contrary to CPU cores, GPU cores are scalar. They still use cache memory using several layers but in a very different way than CPUs. Cores are grouped into clusters with a shared L1 cache. Clusters are subdivided in groups of 8 cores which shares a local memory. These 8 cores execute the same instruction on multiple data. Therefore, vectorization is achieved using several scalar cores. The number of cores per cluster can variate according to models : clusters of 16 cores for G80 architecture or clusters of 24 cores for G200, which is an optimized version of G80.

Thanks to a dedicated bus, GPU cores have access to a very fast embedded memory of hundreds of MB. It allows an efficient use of the highly parallel and bandwidth-demanding units of GPUs. RAM can also be accessed through PCI Express. However, transfers between GPU embedded memory and RAM are very time consuming due to the PCI Express limited bandwidth. Therefore, they must be avoided as most as possible.

Thanks to their radically different architectures, GPUs have a theoretical peak performance an order of magnitude higher than current high-end CPUs. For instance a NVIDIA 8800 GTS (G80 architecture) with 96 stream processors at about 1.2GHz achieves a computational power of 345 GFLOPS in single precision and a NVIDIA GTX 275 (G200 architecture) with 240 stream processors at about 1.4GHz achieves 1010 GFLOPS, compared to the 96 GFLOPS of Intel quad-core processors at 3GHz.

Since GPUs were originally used for graphics purpose, they could only be accessed through dedicated graphic libraries. In order to implement numerical algorithms, NVIDIA has developped CUDA (Compute Unified Device Architecture), which is composed of a compiler and a set of tools organized as a library to help programmers. CUDA is currently the state-of-the-art for General-Purpose GPU. The last step for generalized General-Purpose GPU adoption is the release of GPU clusters specifically designed for scientific computing.

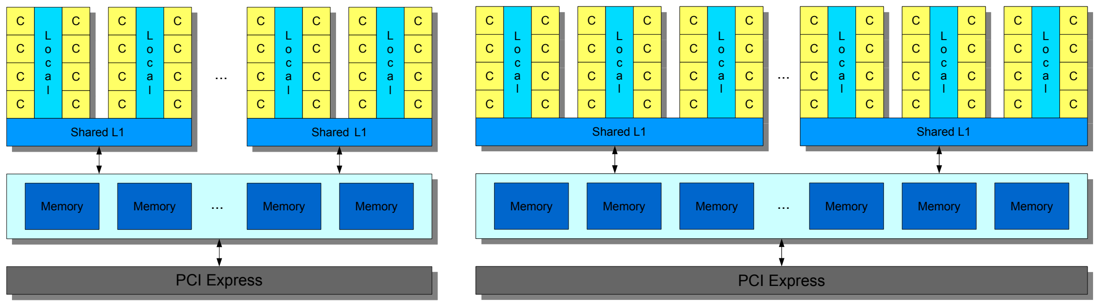


Figure 2: Architectural schemes of NVIDIA GPUs (left : G80; right : G200). See text for details.

Cell Broadband Engine Architecture (CBEA). The Cell Broadband Engine Architecture is a new architecture developed by Sony, Toshiba and IBM [47]. CBEA is designed for high computational throughput and bandwidth along with power efficiency. The general architecture is depicted in figure 3.

The Cell processor consists of an in-order 64-bit PowerPC core, also called Power Processing Element (PPE), that controls several vector computing cores called Synergistic Processing Element (SPE). Current implementations feature up to 8 SPEs. A SPE is an in-order core made of a 128-bit SIMD unit, i.e., a vector unit, and 256KB of local memory. These elements are linked together through a high bandwidth ring bus, denoted as EIB for Element Interconnect Bus.

Since only 256KB of local memory are available per SPE, it is unavoidable to prevent data parts transfer back and forth through the EIB. However, thanks to the high performance of this bus, some usually memory-bound operations, such as matrix/vector multiplication, do not suffer from this limitation on Cell architectures and therefore can be implemented efficiently in parallel [33]. Because no scalar unit is present in SPE cores, it is very important to fully use vectorization in order to achieve best performance.

The well known Sony Playstation 3 (PS3) features a 3.2GHz Cell with 7 SPEs and 256MB of RAM. However, only 6 SPEs and 192MB of RAM are available for scientific calculus. A single SPE at 3.2GHz has a peak performance of 25.6 GFLOPS in single precision. A PS3 Cell can therefore achieve a peak performance of about 150 GFLOPS thanks to its SPEs, i.e., about 150% of the theoretical peak performance of a high-end 3GHz quad-core from Intel. However, the performance increases at the cost of the ease of programming, as depicted by [23, 53]. Many efforts are currently underway to make it easier to exploit the Cell computing power, see [45] for instance.

After these technical considerations about theoretical performance of these architectures, we propose in the next section a simple algorithm that takes benefit from all these features.

3 A Proximal based Algorithm

In this section, we describe our algorithm that allows an efficient implementation on any vectorized parallel many-core architectures. Our approach consists of using a proximal operator on a penalized version of the original problem (1) such that the minimization scheme is reduced to a series of matrix/vector multiplications followed by separable minimizations (i.e., independent point-wise optimizations). This overall process is embedded into a dyadic continuation scheme that speeds-up the approach.

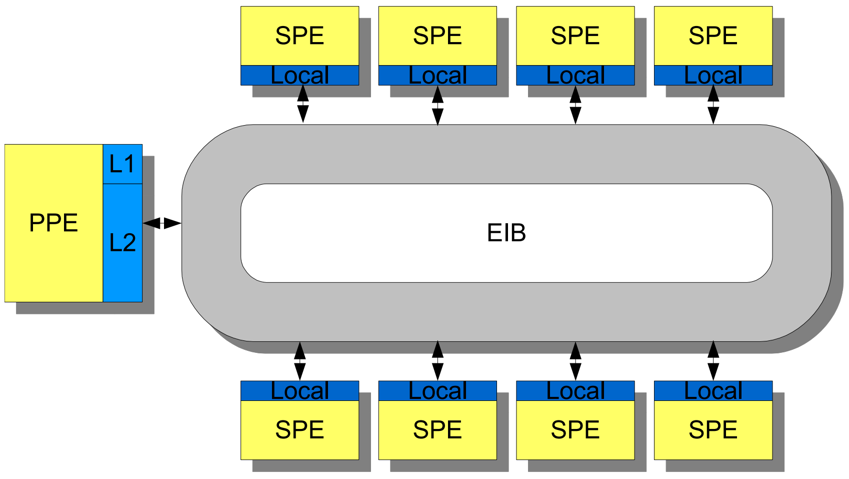


Figure 3: Architectural scheme of a Cell Broadband Engine Architecture. See text for details.

3.1 Moreau-Yosida Regularization

Instead of minimizing the original problem (1) we wish to minimize for $u \in \mathbb{R}^n$ the following energy:

$$E_\mu(u) = \|u\|_1 + \frac{\mu}{2}\|Au - f\|_2^2 \ , \tag{2}$$

where μ is a non-negative parameter whose role is to enforce the constraint $Au = f$. Note that the latter energy corresponds to a penalization method as described by [5]. It is worth to note that a solution of (2) is a solution of problem (1) provided $\mu = \infty$. However, for practical applications one only needs to compute a solution for a finite μ since the measures are corrupted by some noise, or because only a precise-enough solution is wished. Let us denote by $u^\star$ a minimizer of E_μ.

Let us first introduce some notations. Given some non-negative N, the usual Euclidean product in $\mathbb{R}^N$ is denoted by $\langle \cdot, \cdot \rangle$ and its associated norm by $\| \cdot \|_2$. Assume B is a symmetric positive definite linear operator. We set $\langle \cdot, \cdot \rangle_B = \langle B\cdot, \cdot \rangle$ and we note for any $x \in \mathbb{R}$, $\|x\|_B^2 = \langle x, x \rangle_B^2$. Let us introduce the Moreau-Yosida regularization F_μ of E_μ associated with the metric M as the following [31, 34]: at any point $u^{(k)} \in \mathbb{R}^n$,

$$F_\mu(u^{(k)}) = \inf_{u \in \mathbb{R}^n} \left\{ E_\mu(u) + \frac{1}{2}\|u - u^{(k)}\|_M^2 \right\} \ . \tag{3}$$

In other words, it corresponds to inf-convolve the function to minimize with the metric M. Note that F_μ is strictly convex, since it is the sum of a convex and a strictly convex function, and thus there is a unique point minimizing F_μ. Besides, it is not difficult to show that the infimum in (3) is reached and thus it can be replaced by a minimum. Following [39], this unique minimizer is also called the proximal point of u with respect to E_μ and M and is denoted by $p_\mu(u^{(k)})$.

Given a function $E : \mathbb{R}^n \to \mathbb{R}$, let us denote by ∂E the sub-differential of E defined by $w \in \partial E(u) \Leftrightarrow E(v) \geq E(u) + \langle w, v - u \rangle$ for every v (see [31, 31] for instance). This proximal point $p_\mu(u)$ is characterized by the following Euler-Lagrange equation:

$$0 \in \partial \left(E_\mu + \frac{1}{2}\| \cdot - u^{(k)}\|_M^2 \right) (p_\mu(u^{(k)})) \ .$$

The latter is equivalent to:

$$0 \in \partial E_\mu(p_\mu(u^{(k)})) + M(p_\mu(u^{(k)}) - u^{(k)}) \ .$$

And thus we have:

$$Mu^{(k)} \in (M + \partial E_\mu)\,(p_\mu(u^{(k)})) \ .$$

Since $p_\mu(u^{(k)})$ is uniquely defined, we get that:

$$p_\mu\left(u^{(k)}\right) = (M + \partial E_\mu)^{-1}\left(Mu^{(k)}\right) \ . \tag{4}$$

The idea of the Moreau-Yosida regularization approach to minimize E_μ is to iterate the update formula (4) until convergence. This is also known as the proximal point iterations.

The above approach yields the following generic algorithm for minimizing E_μ (for μ fixed).

Generic proximal point minimization algorithm

1. Set $k = 0$ and $u^{(0)} = 0$
2. Compute
$$u^{(k+1)} = p_\mu\left(u^{(k)}\right) = (M + \partial E_\mu)^{-1}\,(Mu^{(k)})$$
3. If "Not converged" then set $k \leftarrow k + 1$ and go to 2 else return $u^{(k+1)}$

The proof of convergence of this algorithm is gieven in Appendix A.

3.2 Proximal operator choices

So far only the generic version of the approach has been described. Indeed, as we have seen, the Moreau-Yosida regularization with any metric M yields an iterative algorithm that converges toward a solution of our problem. The versatility of the approach gives us some freedom on the choice of the metric M. One may wish to consider metrics that yield good speed of convergence, i.e., few iterations [31, 34]. However, recall that our goal is to design an approach that is easy to implement on parallel multi-core architectures. Thus, this goal should lead us to the design of M.

The most tedious part in the optimization problem (4) is that we need to invert $(M + \partial E_\mu)$ (for $Mu^{(k)}$). The optimality condition of the solution $u^{(k+1)}$ writes as follows:

$$s(u^{(k+1)}) + \mu A^t A(u^{(k+1)} - f) + M(u^{(k+1)} - u^{(k)}) = 0 \ ,$$

where $s(u^{(k+1)})$ is a sub-gradient of $\|\cdot\|_1$ at $u^{(k+1)}$. In a more concise form, from an optimization point of view, we have:

$$s(u^{(k+1)}) + (\mu A^t A + M)u^{(k+1)} = \mu A^t f + Mu^{(k)} \ .$$

This operation becomes algorithmically easy when it is separable, i.e., the optimization can be independently carried out dimension by dimension. For our problem, separability means that $(\mu A^t A + M)$ is a diagonal matrix. Besides, this separability property also leads to an implementation that enjoys the requirement properties of section 2. Indeed, since variables are decoupled, coarse parallelism is straightforward. The variables are easily arranged in an well-aligned array that allows vectorized computations. This approach is also wise for cache considerations since the prediction for the next data to process is straightforward.

In order to get a separable optimization problem, M should kill the off-diagonal elements of $\mu A^t A$. Besides, to ensure the global convergence of the approach, M should be positive semi-definite. Recall that good compressive sensing matrices corresponds to submatrices of matrices that satisfies the Restricted Isometry Property (all eigenvalues belong to $[1-\nu, 1+\nu]$ for $\nu > 0$ and small). In this paper, we assume that $\nu = 0$. Examples of such matrices are obtained by considering Discrete Fourier or Discrete Cosine Transform, or orthogonalized Gaussian matrices.

Thus, we have that the eigenvalues of A^tA are 0 or 1. And since A^tA is always a non-negative definite matrix we can define M as follows:

$$M = (1+\epsilon)\mu Id - \mu A^tA \ ,$$

where ϵ is a small positive real number to make M positive definite. Using this M we need to solve the following problem: Find $u^{(k+1)}$ that satisfies:

$$s(u^{(k+1)}) + (1+\epsilon)\mu u^{(k+1)} = \mu A^t f + Mu^{(k)} \ .$$

The latter is well known to be solved by using a shrinkage approach. It has been used and described in many papers such as in [12, 6, 7, 13, 22, 24, 15, 44, 21] for instance. We have that:

$$u^{(k+1)} = \frac{1}{(1+\epsilon)\mu} \begin{cases} \mu A^t f + Mu^{(k)} - sign\left(\mu A^t f + Mu^{(k)}\right) & \text{if } \left|\mu A^t f + Mu^{(k)}\right| > 1 \ , \\ 0 & \text{otherwise} \ , \end{cases} \quad (5)$$

where $sign(x) = \frac{x}{|x|}$ if $x \neq 0$ and 0 otherwise. As one can see, the update of the solution only involves matrix/vector multiplications and some standard scalar operations that can be implemented using vectorized instructions. Also note that one wishes to set ϵ as small as possible to have faster convergence. Setting ϵ to 0 empirically leads to convergence although the proof presented here does not hold for this case.

3.3 Our algorithm

We now fully describe our algorithm. So far, we have considered the optimization of E_μ when μ is set to some arbitrary positive value. Recall that one wishes to set μ to a large value in order to enforce the constraint $Au = f$. However, when μ is large then the procedure generates a series of signals $u^{(k+1)}$ that converges slowly to the solution. One standard way to speed the process is to use a continuation approach that consists of solving (approximately or not) for a series of increasing μ. This kind of approach has been successfully used for instance in [42, 41, 43, 21].

Recall that we choose the constant null signal as an initial guess. It turns out that if μ is too small E_μ does not decrease. We thus wish a μ large enough so that E_μ decreases. Besides, recall that for performance considerations, we would like the smallest one that generates a sequence. This can be achieved thanks a dichotomic-like approach. We assume that we have a lower bound on μ such that $\mu > \mu_{min} > 0$. The following procedure computes the smallest μ, up to a precision $e_{bitonic}$, such that the update gives an energy decrease:

Bitonic Search for initial μ
Input: f, A, μ_{min}

```
1. Set $\mu_{max} = 0$
2. While $\mu_{max} = 0$
    (a) Compute $\hat{u}$ using formula (5) for $\mu_{min}$ using the null signal as the previous solution
    (b) If $E_{\mu_{min}}(\hat{u}) < E_{\mu_{min}}(0)$ then set $\mu_{max} = \mu_{min}$ and $\mu_{min} = \frac{\mu_{min}}{2}$
    (c) Else set $\mu_{min} \leftarrow 2\mu_{min}$
3. While $|\mu_{max} - \mu_{min}| > e_{bitonic}$
    (a) Compute $\hat{u}$ using formula (5) for $\frac{\mu_{min}+\mu_{max}}{2}$ using the null signal as the previous solution
    (b) If $E_{\mu_{min}}(\hat{u}) < E_{\mu_{min}}(0)$ then set $\mu_{max} = \frac{\mu_{min}+\mu_{max}}{2}$
    (c) Else set $\mu_{min} = \frac{\mu_{min}+\mu_{max}}{2}$
4. Return $\mu_{max}$
```

This procedure first looks for μ_{max}, an upper bound on μ, during the iterations of $2-(a)$ to $2-(c)$. Then a standard bitonic search is performed through iterations $3-(a)$ to $3-(c)$.

Once a good initial μ is known, we embed the proximal iterations into a continuation process, i.e., μ will successively takes the values $\mu, 2\mu, \ldots, 2^{l_{max}}\mu$, where l_{max} is a strictly positive integer. Note that other coefficients than 2 could be used but experiments have shown that this value gives good results. We also need to give the stopping criteria of our approach. These criteria are used for any optimization of E_μ when μ is fixed. The first criteria concerns the closeness of the reconstructed solution to the observed values. The process is stopped when the reconstructed solution $u^{(k+1)}$ is below some prescribed tolerance e_{tol} measured as follows: $\frac{\|Au^{(k+1)}-f\|_2}{\|f\|_2}$. Since the proximal iterations converge toward the solution but not necessarily in finite time, we also stop the process when the energy decrease between two consecutive solutions is below some prescribed value denoted by e_{consec}, i.e., we stop as soon as $(E_\mu(u^{(k)}) - E_\mu(u^{(k+1)})) < e_{consec}$. When one of this stopping criteria is met, the value of μ is updated and we start a new descent minimization for E_μ using the current solution as initial guess.

Minimization Algorithm
Input: f, A and μ_{min}

```
1. Set $k = 0$ and $u^{(0)} = 0$
2. Compute the initial value of $\mu$ using the above bitonic approach
3. for $l = 1$ to $l_{max}$ (number of continuation values for $\mu$)
   (a) Do
       i. Set $k \leftarrow k+1$
       ii. Compute $u^{(k+1)}$ using formula (5)
   (b) While $\left(E_\mu(u^{(k)}) - E_\mu(u^{(k+1)}) > e_{consec}\right)$ and $\left(\frac{\|Au^{(k+1)}-f\|_2}{\|f\|_2} > e_{tol}\right)$
   (c) $\mu \leftarrow 2\mu$
4. Return $u^{(k+1)}$
```

4 Experimental results

In this section, we shortly present general implementation choices we have made and more specific ones that depend on the architectures we have considered. Then we give experimental results that assess the efficiency of our method running on these parallel many-core architectures.

4.1 Implementation details

The most time consuming operations in our method are those that involve the sampling matrix A. These operations can be performed in two different ways depending on the nature of A. Either A is described explicitly, i.e., all the coefficients of the matrix are stored in memory, and one needs to perform a standard matrix/vector multiplication; or A can be represented implicitly with the help of a transform. A typical example for the latter case is when A is a sub-matrix of the Discrete Fourier Fransform (FFT), or Discrete Cosine Transform (DCT). Both representations enable operations to done in parallel, i.e., they are very good candidates for both coarse and fine-grained parallelism implementations.

Allowing matrices to be stored explicitly allows for flexibility in the design of the matrix. Unfortunately, storing such a matrix is memory consuming and since a matrix/vector multiplication needs $O(n \times m)$ operations where $n \times m$ is the size of the matrix, it is also time consuming. On the contrary, the use of the implicit form is memory-wise. Indeed, it is then no longer needed to store A (and its transpose A^T). Besides, it generally allows for faster computations because of fast available transforms. According to the literature (see [4] for instance) the time complexity for computing an FFT is $O(n \log n)$. However, the price to pay is that it drastically reduces the set of possible sampling matrices.

In this paper, we consider these two kinds of matrices. For the explicit case, we consider orthogonalized Gaussian matrices where their elements are generated using i.i.d normal distributions $\mathcal{N}(0,1)$ and where their rows are orthogonalized. The second class of matrices we have used corresponds the partial DCTs. They are generated by randomly choosing, with uniform sampling, m rows from the full Discrete Cosine matrix. Please note that operations that involve orthogonalized Gaussian matrices or partial DCT comply with the requirements we present in section 2.

Due to the large amount of memory needed, only the CPU platform can run a code that uses large orthogonalized Gaussian matrices. On the contrary, partial DCTs can be used on any of the architectures we are interested in. To allow a fair comparison, only the latter was implemented on all three platforms. This partial DCT have been implemented through a complex-to-complex Fast Fourier Transform (FFT) with an additionnal $O(n)$ pre and postprocessing for converting the results to real numbers. Indeed, direct DCT implementations are currently subject to performance issues and cannot be used for our purpose. Using this complex-to-complex FFT means an overhead of 4 times the memory required by directly applying DCT but allows to keep good time performances.

All implementations use single precision floating point arithmetic because of GPU and Cell limitations. It is important to note that these limitations are currently partly addressed by developers of these platforms.

We now present architecture-dependent implementation details.

CPU implementation details. The code running on the CPU is parallelized using the OpenMP API (Open Multi-Processing Application Programming Interface). We refer the reader to [14] for further details. Our implementation is vectorized using SSE instructions and is also tuned to ensure the most efficient use of the cache. Recall that the thresholding approach (see section 3) has been chosen for its separability property (i.e., each element can be processed independently), therefore parallel and vector computing are profitable to obtain an efficient implementation. FFT computations are performed thanks to the FFTW 3.1 (Fastest Fourier Transform in the West). We refer the reader to [26] for further details. Last, the code is compiled using the Intel C compiler version 10.1.

Cell implementation details. Our Cell implementation uses the Cell SDK 3.0 (Software Development Kit) provided by IBM and the FFTW 3.2 alpha 3 library. To the best of our knowledge, FFTW is the current state-of-the-art FFT implementation in term of input size (for instance FFTC [2] is the fastest Fourier transform, but only for up to 16K complex input samples) and still have good time performances. This alpha version of the FFTW library requires exclusive access to the SPEs it uses (i.e., SPEs cannot run FFT and others processes together). Therefore, since FFT computations are the most time-consuming tasks of our method, we chose to assign all SPEs to FFT computations. FFT/DCT conversion, thresholding and energy computations are done on the PPE (which is not intended for performance). In order to balance the cost of using the PPE for these tasks, vectorized Altivec [17] code has been written to improve performances of the FFT/DCT conversions, which are the second most costly computations of our method. Nevertheless, our Cell implementation can still be optimized as a lot of computations could be done on SPEs instead of on PPE (SPEs are at least one order of magnitude faster than the PPE). However, this can only be done using a dynamic load balancing process, which will be time consuming and difficult to implement.

GPU implementation details. The GPU implementation is based on CUDA[1] (Compute Unified Device Architecture) 2.0 and the CUFFT library (part of the CUDA package). Since the PCI

[1]More informations can be found on the website : `http://www.nvidia.com/object/cuda_home.html`

Express bus suffers severe bandwidth limitations, we designed our implementation such that it uses very few data communications between central RAM and GPU embedded memory. Note that this is a very common problem in General-Purpose computing on GPUs. However, when absolutely needed and as explained in [35], transfer sizes must be carefully chosen. Modern GPUs feature highly hierarchical set of computational units, memories and caches. The number of threads must be maximized in order to achieve best performance, but attention must be paid for avoiding bank conflicts, i.e., multiple threads accessing the same shared memory bank at the same time. For example, [1] implements a memory manager to meet this goal.

4.2 Experiments

In this section, we first describe the experimental conditions we have considered for our experiments. Then, we present numerical results that show the effectiveness of our approach.

4.2.1 Experimental conditions

In order to prove the effectiveness of our approach and to compare the benefits and disadvantages of the different architectures, we use an experimental platform that consists of the following:

- Two standard Intel quad-core processors. The first one is an Intel Core 2 Quad Q6600 2.4GHz with 8MB of L2 cache and 4GB of RAM. Theoretically, this processor can achieve a peak performance of 38 GFLOPS. For our specific case where only single precision computations are used, this means that we can hope a peak performance of 76 GFLOPS in single precision. The second processor is an Intel Core i7 920 2.66GHz with 1MB of L2 cache, 8MB of L3 cache and 6GB of RAM. This processor can achieve a theoretical peak performance of 43 GFLOPS in double precision and 86 GFLOPS in single precision. The Turbo Boost feature has been systematically disabled to avoid bias in our results.

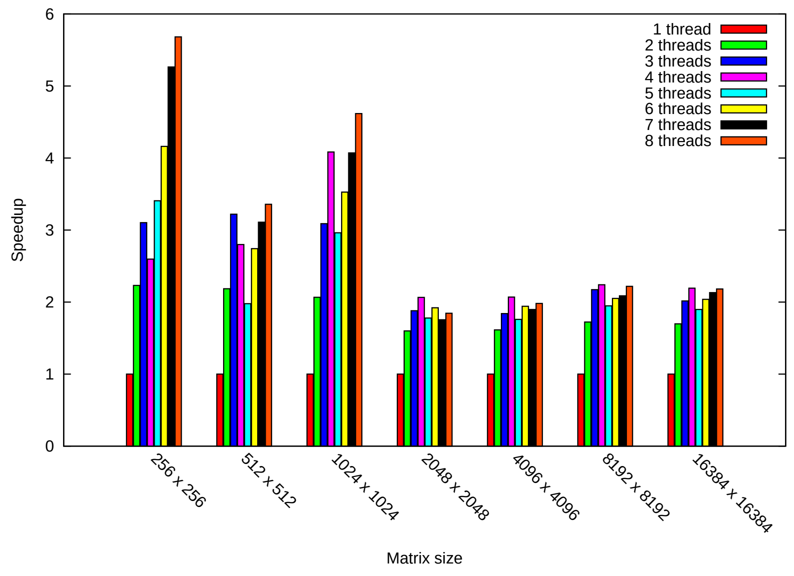
- A popular implementation of the Cell Broadband Engine Architecture that one can find in the Sony Playstation 3. The machine consists of a 3.2Ghz Cell processor with 6 usable SPEs and 192MB of RAM.

- Two commercial video cards produced by NVIDIA, namely the NVIDIA GEFORCE 8800 GTS and the NVIDIA GEFORCE GTX 275. The first one has 96 cores and embeds 640MB of memory while the second has 240 cores and 896MB of memory.

We now give the values of the parameters we have used for our experiments. We set $m = n/8$ (recall that m is assumed to be much smaller than n). The number of non-zero values in the original sparse signal is set to $k = m/10$. Recall that there are two stopping criteria: the tolerance error e_{tol} with associated stopping criteria $\frac{\|Au^{(k+1)}-f\|_2}{\|f\|_2} < e_{tol}$, and the consecutive variation tolerance e_{consec} associated with $(E(u^{(k)}) - E(u^{(k+1)})) < e_{consec}$. We set $e_{tol} = 10^{-5}$ and $e_{consec} = 10^{-3}$. Concerning the bitonic search, we set $\mu_{min} = 2$ and $e_{bitonic} = 0.5$. The number of continuation steps, l_{max}, performed after the bitonic step is set to $l_{max} = 10$.

4.2.2 Parallel speedup and caches issues

We now present preliminary experiments about parallel speedup in order to show particular issues arising with matrix/vector multiplication, which is a key operation in our method. Squared and rectangular single precision matrices are both used to run our benchmarks on the Intel Core i7 processor (with Turbo Boost disabled). The results are the average of 10 instances based on orthogonalized Gaussian matrices and randomly generated vectors. Note that results are not about absolute performance but parallel performance, therefore better results means better scaling but not necessarily better wall clock time.



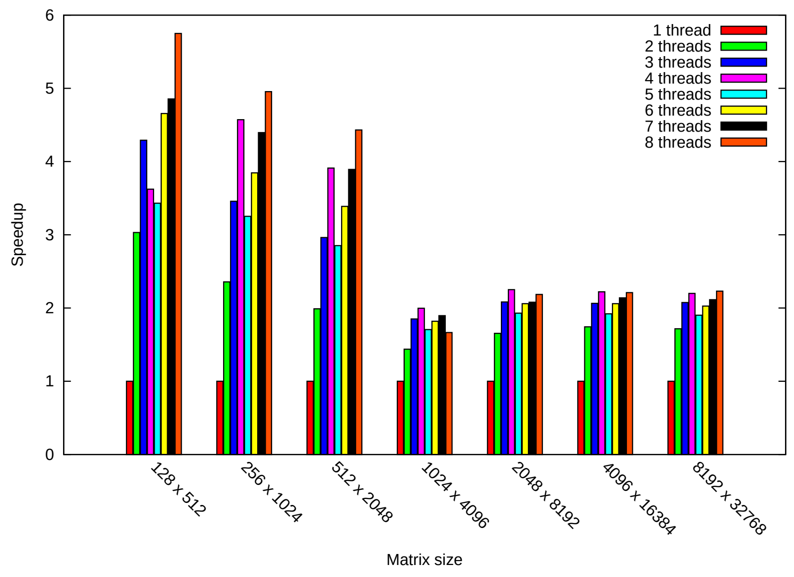


Figure 4: Parallel speedup with respect to the number of threads for matrix/vector multiplications (Intel Core i7 920).

A super linear speedup is achieved for matrice sizes up to 512×512 and 256×1024, i.e., 512KB matrices. The shared L3 cache is very efficiently used and each core can take full advantage of its own very fast 256KB L2 cache.

A cut-off effect appears for 2048×2048 and 1024×4096 matrices, i.e., when the matrices use

8MB of memory, which is the size of the L3 cache. The whole data used by the program do not hold anymore into the L3 memory, which implies that L3 reuse is compromised and then a huge performance penalty.

The use of Hyper-Threading can yield positive or negative effects, as explained in section 2. HT allows to more efficiently use computing units inside a core. It can yield super linear parallel speedup, which can be seen principally before reaching 8MB matrices. However, handling 2 threads per core can increase pressure on cache. This phenomenon can be seen after reaching 8MB matrices. The graphs show a balance between these two effects.

4.2.3 Results

We now present experimental results of our method. Figure 5 represents the variation of errors with respect to time. We chose a representative example to illustrate the different steps of the optimization process. We consider two error criteria : e_{tol} (defined above) and the relative error of the reconstructed signal $u^{(k+1)}$ to the ground truth defined as $relative = \frac{\|u^{(k+1)} - u^*\|_2}{\|u^*\|_2}$.

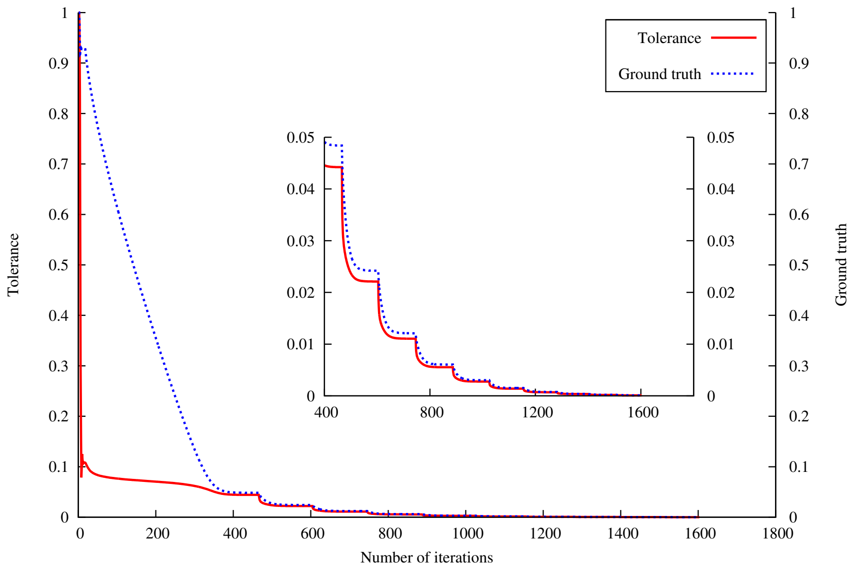


Figure 5: Errors with respect to time for a representative example of orthogonalized Gaussian matrix of size 2048×16384.

We can see at the very beginning of the process the bitonic search. When an appropriate initial μ is found, the continuation process is launched. Each step of the continuation is shown at the curve level as a huge decrease of both error criteria. We can also observe that the ground truth error decreases almost linearly until 400 iterations, from then the decrease is far slower. On the contrary, the relative error does not show this first fast decrease process. This graph depicts the fact that out method could be stopped earlier using tighter parameters. This is a very common characteristic for this kind of optimization methods, such as those previously presented in section 1. The downside is that it makes comparison between methods very difficult because of the multiple biased involved. In fact, each method has its own set of parameters that do not necessarily have equivalents. In particular, our method has the advantage to not rely on a *dt* parameter representing the optimization process step, unlike methods based on linearization.

Our other results are presented as tables where the final values of e_{tol} ($tolerance$) and the relative error of the reconstructed signal to the ground truth ($relative$) are given. The tables also present the number of iterations ($\#iter$) that have been performed to obtain the final solution and the wall clock time in seconds taken by the whole process.

Compressive sensing with orthogonalized Gaussian matrices. The first experiment (figure 1) uses orthogonalized Gaussian matrices and standard Intel quad-core processors. The tables give the time spent for the computation with respect to the number of threads used by the program.

in		out			time (s)		
m	n	tolerance	relative	#iter	1 thread	2 threads	4 threads
64	512	1.24e-03	1.40e-03	1163.2	0.076	0.044	0.029
128	1024	4.81e-04	5.22e-04	1155.2	0.238	0.128	0.072
256	2048	2.99e-04	3.26e-04	1326.4	1.144	0.562	0.293
512	4096	1.93e-04	2.15e-04	1461.7	5.659	3.584	3.386
1024	8192	1.29e-04	1.43e-04	1505.0	23.224	15.500	15.352
2048	16384	8.66e-05	9.62e-05	1611.1	97.123	64.527	64.778

Table 1: Results for orthogonalized Gaussian matrices on an Intel Core 2 Quad Q6600.

in		out			time (s)			
m	n	tolerance	relative	#iter	1 thread	2 threads	4 threads	8 threads
64	512	1.24e-03	1.39e-03	1162.0	0.042	0.025	0.017	0.252
128	1024	4.81e-04	5.22e-04	1155.5	0.115	0.068	0.046	0.350
256	2048	2.99e-04	3.26e-04	1321.5	0.423	0.227	0.132	0.375
512	4096	1.93e-04	2.15e-04	1465.8	2.362	1.495	1.130	1.359
1024	8192	1.29e-04	1.43e-04	1505.6	9.933	6.035	4.574	4.885
2048	16384	8.66e-05	9.62e-05	1604.9	41.842	25.284	19.576	19.997

Table 2: Results for orthogonalized Gaussian matrices on a Intel Core i7 920.

Table 1 presents results for an Intel Core 2 Quad and table 2 results for an Intel Core i7. The Core 2 Quad shows an interesting performance scaling until $n = 4096$, where the situation begins to change. When this value is reached, there is no more benefit to use 4 threads compared to 2. This is due to Core 2 Quad core interconnection using FSB and its cache hierarchy, which has huge repercutions on available bandwidth, as pointed by [52]. Note that the scaling ratio from 1 thread to 2 is only slightly affected as the 2 threads are executed on 2 cores of the same cluster.

Concerning the same experiment with the Core i7, let us recall that HT enables to handle up to 8 threads for a 4-core processor. For our particular experiment, the HT implies a loss in performance from 4 to 8 threads due to a too high cache pressure. The phenomenon beginning at $n = 4096$ with the Core 2 Quad does not happen with the Core i7 thanks to its new core interconnection and its optimized cache hierarchy. These results are coherent with our previous results of section 4.2.2. Above $n = 4096$ the scaling is not as good as before but remains constant and still yield a performance increase.

Compressing sensing with partial DCT. The second experiment (table 3 for an Intel Core 2 Quad and table 4 for an Intel Core i7) uses partial DCTs instead of orthogonalized Gaussian matrices. Since DCT and Inverse DCT are used instead of matrix/vector multiplications, memory can be saved and faster computations can be achieved. This means that n can take higher value in this experiment compared to the previous one.

One can see that performance scaling of multithreading for the Core 2 Quad is poor when n

is small. The benefit of using more cores only comes up when enough computations are done on each core. For $n = 2048$, the best wall clock time happens when using 2 threads. Inside a cluster of 2 cores, the shared L2 cache allows to scale for this problem size. However, the core interconnection and the cache hierarchy penalize the scaling over 2 cores. In our case, the benefit of using 4 cores happens when n reaches 4096. Larger n means better scaling ratio.

For the Core i7, there is no penalty using 2 threads instead of 1, nor 4 instead of 2 whatever the problem size. The bigger the problem is, the better the scaling is. Concerning HT, like for orthogonalized Gaussian matrices, it implies a loss in performance when using 8 threads instead of 4.

in		out			time (s)		
m	n	tolerance	relative	#iter	1 thread	2 threads	4 threads
64	512	1.02e-03	1.11e-03	1029.9	0.034	0.044	0.050
128	1024	4.74e-04	5.08e-04	1075.1	0.072	0.081	0.089
256	2048	2.81e-04	3.09e-04	1219.6	0.180	0.169	0.185
512	4096	1.92e-04	2.15e-04	1415.6	0.501	0.472	0.405
1024	8192	1.27e-04	1.40e-04	1456.6	1.110	0.884	0.818
2048	16384	8.68e-05	9.61e-05	1584.7	2.544	1.840	1.797
4096	32768	6.20e-05	6.91e-05	1780.5	6.366	4.647	4.424
8192	65536	4.33e-05	4.82e-05	2016.0	16.571	11.543	10.797
16384	131072	3.01e-05	3.35e-05	2256.5	46.512	32.627	27.568
32768	262144	2.15e-05	2.40e-05	2672.1	199.254	117.574	94.599
65536	524288	1.52e-05	1.70e-05	3261.0	508.388	298.467	204.657
131072	1048576	1.11e-05	1.25e-05	4067.5	1321.440	825.617	567.315

Table 3: Results for partial DCTs on a Intel Core 2 Quad Q6600.

in		out			time (s)			
m	n	tolerance	relative	#iter	1 thread	2 threads	4 threads	8 threads
64	512	1.02e-03	1.11e-03	1031.3	0.025	0.023	0.020	0.055
128	1024	4.74e-04	5.08e-04	1076.8	0.055	0.038	0.034	0.068
256	2048	2.81e-04	3.09e-04	1223.2	0.120	0.082	0.070	0.126
512	4096	1.92e-04	2.15e-04	1419.6	0.288	0.197	0.157	0.220
1024	8192	1.25e-04	1.38e-04	1438.5	0.598	0.413	0.316	0.391
2048	16384	8.76e-05	9.70e-05	1597.2	1.452	0.954	0.718	0.911
4096	32768	6.21e-05	6.91e-05	1783.2	3.494	2.112	1.641	1.887
8192	65536	4.33e-05	4.81e-05	2004.8	8.293	5.280	3.780	4.137
16384	131072	3.00e-05	3.35e-05	2253.2	20.026	11.961	8.755	11.017
32768	262144	2.13e-05	2.38e-05	2625.0	87.031	49.508	32.411	34.852
65536	524288	1.52e-05	1.70e-05	3262.0	225.341	129.467	82.432	91.006
131072	1048576	1.12e-05	1.26e-05	4074.6	628.962	337.811	222.477	239.236

Table 4: Results for partial DCTs on a Intel Core i7 920.

The next two experiments (figures 5 and 6/7) are specific implementations respectively for PS3 Cell and NVIDIA GPU platforms. Only partial DCT results are provided for these platforms as they have strong memory limitations. Note that, as explained in previous section and contrary to our CPU implementation, these specific implementations are yet not optimal and only reflect what kind of performances can be achieved with very little development effort.

Figure 6 gives a comparison between the three implementations on the five hardwares. GPUs are currently limited by the dimension size they can work on but our basic GPU implementation

m	n	tolerance	relative	#iter	time (s)
64	512	1.02e-03	1.12e-03	1033.3	0.071
128	1024	4.74e-04	5.08e-04	1077.7	0.153
256	2048	2.81e-04	3.09e-04	1231.1	0.359
512	4096	1.92e-04	2.15e-04	1420.6	0.913
1024	8192	1.27e-04	1.39e-04	1458.8	1.935
2048	16384	8.67e-05	9.61e-05	1581.1	4.340
4096	32768	6.21e-05	6.91e-05	1772.4	10.445
8192	65536	4.33e-05	4.82e-05	2006.8	29.523
16384	131072	3.00e-05	3.34e-05	2232.5	60.178
32768	262144	2.14e-05	2.38e-05	2663.0	149.679
65536	524288	1.51e-05	1.68e-05	3236.5	367.822
131072	1048576	1.09e-05	1.23e-05	4066.7	934.656

Table 5: Results for partial DCTs on a PS3 Cell.

m	n	tolerance	relative	#iter	time (s)
64	512	1.02e-03	1.12e-03	1014.6	0.309
128	1024	4.72e-04	5.03e-04	1034.1	0.344
256	2048	2.83e-04	3.12e-04	1213.1	0.461
512	4096	1.92e-04	2.14e-04	1411.3	0.623
1024	8192	1.26e-04	1.39e-04	1437.8	0.978
2048	16384	8.71e-05	9.64e-05	1573.0	1.985
4096	32768	6.25e-05	6.97e-05	1760.5	4.532
8192	65536	4.37e-05	4.88e-05	2017.2	11.036
16384	131072	3.09e-05	3.46e-05	2259.5	28.337

Table 6: Results for partial DCTs on a NVIDIA GPU 8800 GTS.

m	n	tolerance	relative	#iter	time (s)
64	512	1.02e-03	1.12e-03	1015.6	0.177
128	1024	4.72e-04	5.03e-04	1033.8	0.222
256	2048	2.83e-04	3.12e-04	1215.6	0.298
512	4096	1.92e-04	2.14e-04	1408.5	0.404
1024	8192	1.26e-04	1.39e-04	1428.1	0.500
2048	16384	8.71e-05	9.65e-05	1563.1	0.957
4096	32768	6.25e-05	6.97e-05	1757.2	2.011
8192	65536	4.37e-05	4.88e-05	2001.5	4.586
16384	131072	3.09e-05	3.46e-05	2251.2	11.274

Table 7: Results for partial DCTs on a NVIDIA GPU GTX 275.

is as fast as our optimized multi-core CPU one. More precisely, the NVIDIA 8800 GTS is head to head with the Core 2 Quad whereas the NVIDIA GTX 275 is head to head with the Core i7, the quad-core CPUs being always slightly faster that GPUs. Our Cell implementation running on the PS3 is slower than our multi-core CPU implementation but this is due to the fact that a lot of computations are done on the PPE, which is very slow (even if vectorized Altivec code reduces a little this overhead), although they could be fully done on SPEs to reach maximal performance. The reader should also note that the Cell is now quite old and here tested in a 6 SPEs configuration. Current GPU and Cell implementations give a lower bound on performances that can be achieved on these platforms and an idea about standard implementation performances. In particular, a better Cell implementation that fully takes advantage of the characteristics of this processor (i.e., use also the SPE not only for the FFT but also for the shrinkage process) is left as future work.

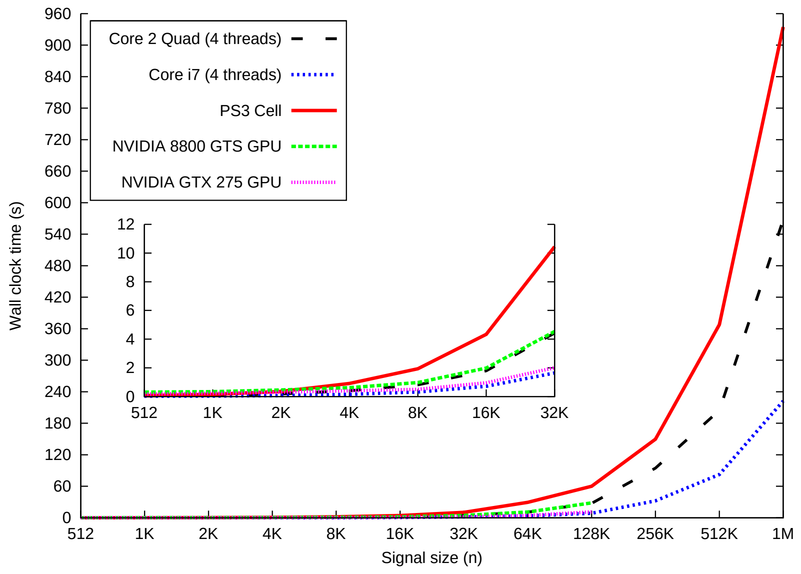


Figure 6: Results for partial DCTs on various platforms.

5 Conclusion

In this paper we presented a standard algorithm for solving compressive sensing problems involving l^1 minimization. This algorithm has been especially designed to take benefit of current parallel many-core architectures and achieves noticeable speedups. Besides, it is easy to implement on these architectures. To validate our approach, we proposed implementations on various current high-end platforms, such as vectorized multi-core CPU, GPU and Cell. Pros and cons of both platforms and implementations have been discussed. In particular, we have seen that multi-core CPUs can offer comparable performances with GPUs when their parallel features are used. The results are promising and allow to hope very fast implementations on new architectures such as the next generation many-core x86 architecture of the Intel Larrabee [48].

A Proof of Convergence

We briefly show the standard elements of the proof of the convergence of the proximal iterations. Recall that the approach is quite standard and some proof can be adapted for [31, 34] for instance.

First, let us note that the series $\{E_\mu\left(p(u^{(k)})\right)\}$ is clearly non-decreasing and bounbded by below, and thus converges toward some value referred to as η.

Then, let us recall a standard convex optimality result. Assume that $g : \mathbb{R}^N \to \mathbb{R}$ is convex and differentiable and $h : \mathbb{R}^N \to \mathbb{R}$ is convex, then $u^\star$ is a global minimizer of $(g + h)$ if and only if the following holds:

$$\forall u \in \mathrm{I\!R}^N \ \ \langle \nabla g(u^\star), u - u^\star \rangle + h(u) - h(u^\star) \geq 0 \ \ . \tag{6}$$

For our case, let us have $g(\cdot) = \frac{1}{2} \| \cdot - u^{(k)} \|_M^2$ and $h(\cdot) = E_\mu(\cdot)$ and recall that $p_\mu(u^{(k)})$ is the global minimum of the inf-convolution when it is fed with $u^{(k)}$:

$$\forall u \in \mathrm{I\!R}^N \ \ \left\langle p\left(u^{(k)}\right) - u^{(k)}, u - p\left(u^{(k)}\right) \right\rangle_M + E_\mu(u) - E_\mu\left(p\left((u^{(k)}\right)\right) \geq 0 \ \ . \tag{7}$$

Now, we consider this inequality for the two points $\hat{u}$ and $\bar{u}$ with associated proximal points $p(\hat{u})$ and $p(\bar{u})$, respectively. Some simple calculus lead to:

$$\langle p(\hat{u}) - \hat{u}, p(\bar{u}) - p(\hat{u}) \rangle_M + \langle p(\bar{u}) - \bar{u}, p(\hat{u}) - p(\bar{u}) \rangle_M \geq 0 \ \ ,$$

and thus we get:

$$\langle \bar{u} - \hat{u}, p(\bar{u}) - p(\hat{u}) \rangle_M \geq \| p(\hat{u}) - p(\bar{u}) \|_M^2 \ \ .$$

The latter is equivalent to:

$$\| \hat{u} - \bar{u} \|_M^2 - \| p(\hat{u}) - \hat{u} - p(\bar{u}) + \bar{u} \|_M^2 \geq \| p(\hat{u}) - p(\bar{u}) \|_M^2 \ \ .$$

Now, let us set $\bar{u}$ to a global minimizer of R_μ, i.e., $\bar{u} = u^\star$, and $\hat{u} = u^{(k)}$, in the previous inequality. Note that $p\left(u^\star\right) = u^\star$, and recall that $u^{(k+1)} = p\left(u^{(k)}\right)$. The following inequality holds:

$$\left\| u^{(k)} - u^\star \right\|_M^2 - \left\| u^{(k+1)} - u^{(k)} \right\|_M^2 \geq \left\| u^{(k+1)} - u^\star \right\|_M^2 \tag{8}$$

With this inequality we can conclude using the following points.

The series $\|u^{(k)} - u^\star\|_M^2$ is non-decreasing and bounded by below (by $E_\mu(u^\star)$)) and thus does converge. Thus, we deduce that $\lim_{k\to\infty} \left(\|u^{(k+1)} - u^\star\|_M^2 - \|u^{(k)} - u^\star\|_M^2 \right) = 0$. Using this result and (8) we get that $\lim_{k\to\infty} \|u^{(k+1)} - u^{(k)}\|_M^2 = 0$.

Note that by the convexity of the energy E_μ, we have:

$$E_\mu\left(u^\star\right) \geq E\left(u^{(k+1)}\right) + \left\langle \partial E_\mu, u^\star - u^{(k+1)} \right\rangle \tag{9}$$

Since $u^{(k+1)}$ is the global minimizer of $F(u^{(k)})$, we have that:

$$\left\langle \partial E_\mu, u^\star - u^{(k+1)} \right\rangle + \left\langle u^{(k+1)} - u^{(k)}, u^\star - u^{(k+1)} \right\rangle \geq 0 \ \ . \tag{10}$$

Recall that, as it has been shown above, $\lim_{k\to+\infty} \|u^{(k+1)} - u^{(k)}\|^2 = 0$, and since $\|u^{(k+1)} - u^\star\|^2$ is bounded we have that $\liminf_{k\to+\infty} \left\langle \partial E_\mu, u^\star - u^{(k+1)} \right\rangle \geq 0$. Injecting this information into (9), we obtain that $\lim_{k\to+\infty} E_\mu\left(u^{(k)}\right) \leq \eta$, and thus we conclude that $\lim_{k\to+\infty} E_\mu(u^{(k)}) = \eta$.

Acknowledgments

The research of A. Borghi on this work has been done while being at the mathematics department of UCLA and being supported by the ONR grant N000140710810. The research of J. Darbon and S. Osher was supported by ONR grant N000140710810. The research of T. Chan was supported by DMS-0610079 and ONR N00014-06-1-0345.

References

[1] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, M. Ripeanu, StoreGPU: exploiting graphics processing units to accelerate distributed storage systems, in: Proceedings of the 17th international symposium on High performance distributed computing (HPDC), 2008, pp. 165–174.

[2] D. A. Bader, V. Agarwal, FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine, in: S. Aluru, M. Parashar, R. Badrinath, V. K. Prasanna (eds.), HiPC, vol. 4873 of Lecture Notes in Computer Science, Springer, 2007, pp. 172–184.

[3] W. U. Bajwa, J. D. Haupt, G. M. Raz, S. J. Wright, R. D. Nowak, Toeplitz-structured compressed sensing matrices, in: Proceedings of the 14th IEEE/SP Workshop on Statistical Signal Processing (SSP), 2007, pp. 294–298.

[4] D. J. Bernstein, The tangent FFT, in: S. Boztas, H. feng Lu (eds.), Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, vol. 4851 of Lecture Notes in Computer Science, Springer, 2007, pp. 291–300.

[5] D. Bertsekas, Constrained Optimization and Lagrange Multiplier Methods, Athena Scientific, 1996.

[6] J. Bioucas-Dias, M. Figueiredo, A new TwIST: two-step iterative shrinkage/thresholding algorithms for image restoration, IEEE Trans. on Image Processing 16 (12) (2007) 2992–3004.

[7] K. Bredies, D. A. Lorenz, Iterated hard shrinkage for minimization problems with sparsity constraints, SIAM Journal on Scientific Computing 30 (2) (2008) 657–683.

[8] E. Candès, J. Romberg, Quantitative robust uncertainty principles and optimally sparse decompositions, Foundations of Computational Mathematics 6 (2006) 227–254.

[9] E. Candès, J. Romberg, T. Tao, Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information, IEEE Trans. on Information Theory 52 (2) (2006) 489–509.

[10] E. Candès, T. Tao, Near optimal signal recovery from random projections: Universal encoding strategies?, IEEE Trans. on Information Theory 52 (12) (2006) 5406–5426.

[11] V. Cevher, A. Sankaranarayanan, M. F. Duarte, D. Reddy, R. G. Baraniuk, R. Chellappa, Compressive sensing for background subtraction, in: Proceedings of the 10th European Conference on Computer Vision (ECCV), vol. 5303, 2008, pp. 155–168.

[12] A. Chambolle, R. A. DeVore, N.-Y. Lee, B. J. Lucier, Nonlinear wavelet image processing: variational problems, compression, and noise removal through wavelet shrinkage, IEEE Trans. on Image Processing 7 (1998) 319–335.

[13] P. Combettes, J.-C. Pesquet, Proximal thresholding algorithm for minimization over orthonormal bases, SIAM Journal on Optimization 18 (4) (2007) 1351–1376.

[14] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE Computational Science & Engineering 5 (1) (1998) 46–55.

[15] I. Daubechies, M. Defrise, C. D. Mol, An iterative thresholding algorithm for linear inverse problems with a sparsity constraint, Communications in Pure and Applied Mathematics 57 (11) (2004) 1413–1457.

[16] R. A. DeVore, Deterministic constructions of compressed sensing matrices, Journal of Complexity 4–6 (23) (2007) 918–925.

[17] K. Diefendorff, P. Dubey, R. Hochsprung, H. Scale, Altivec extension to PowerPC accelerates media processing, IEEE Micro 20 (2) (2000) 85–95.

[18] D. Donoho, Y. Tsaig, I. Drori, J.-L. Starck, Sparse solution of underdetermined linear equations by stagewise orthogonal matching pursuit, Tech. rep. (2006).

[19] D. L. Donoho, Compressed sensing, IEEE Trans. on Information Theory 52 (4) (2006) 1289–1306.

[20] F.-X. Dupe, J. Fadili, J.-L. Starck, A proximal iteration for deconvolving Poisson noisy images using sparse representations, IEEE Trans. on Image Processing 16 (12) (2008) 2992–3004.

[21] W. Y. E. T. Hale, Y. Zhang, A fixed-point continuation method for l1-regularized minimization with applications to compressed sensing, Tech. rep., Rice University (2007).

[22] M. Elad, Why simple shrinkage is still relevant for redundant representation?, IEEE Trans. on Information Theory 52 (2006) 5559–5569.

[23] G. D. Fabritiis, Performance of the Cell processor for biomolecular simulations, Computer Physics Communications 176 (11–12) (2007) 660–664.

[24] M. Figueiredo, R. Nowak, An EM algorithm for wavelet-based image restoration, IEEE Trans. on Image Processing 12 (8) (2003) 906–916.

[25] M. Figueiredo, R. Nowak, S. Wright, Gradient projection for sparse reconstruction: application to compressed sensing and other inverse problems, IEEE Journal of Selected Topics in Signal Processing 1 (3) (2007) 586–598.

[26] M. Frigo, S. Johnson, FFTW: an adaptive software architecture for the FFT, in: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, vol. 3, 1998, pp. 1381–1384.

[27] A. C. Gilbert, M. J. Strauss, J. A. Tropp, R. Vershynin, Algorithmic linear dimension reduction in the l_1 norm for sparse vectors, in: Proceedings of the 44th Annual Allerton Conference on Communication, Control and Computing, 2006, pp. 1411–1418.

[28] T. Goldstein, S. Osher, The split Bregman method for l1 regularized problems, Tech. Rep. CAM 08-29, UCLA (2008).

[29] R. Griesse, D. A. Lorenz, A semismooth Newton method for Tikhonov functionals with sparsity constraints, Inverse Problems 24 (3), 2008.

[30] C. Hegde, M. Wakin, R. Baraniuk, Random projections for manifold learning, in: Neural Information Processing Systems (NIPS), 2007.

[31] J.-B. Hiriart-Urruty, C. Lemaréchal, Convex Analysis and Minimization Algorithms, Springer Verlag, Heidelberg, 1996, two volumes - 2nd printing.

[32] S.-J. Kim, K. Koh, M. Lustig, S. Boyd, D. Gorinevsky, An interior-point method for large-scale l_1-regularized least squares, IEEE Journal of Selected Topics in Signal Processing 1 (4) (2007) 606–617.

[33] J. Kurzak, J. Dongarra, Implementation of mixed-precision in solving systems of linear equations on the CELL processor, Concurrency and Computation: Practice and Experience 19 (10) (2007) 1371–1385.

[34] C. Lemaréchal, C. Sagastizábal, Practical aspects of the Moreau-Yosida regularization: Theoretical preliminaries, SIAM Journal on Optimization 7 (2) (1997) 367–385.

[35] M. D. Lieberman, J. Sankaranarayanan, H. Samet, A fast similarity join algorithm using graphics processing units, in: Proceedings of the IEEE International Conference on Data Engineering (ICDE), 2008, pp. 1111–1120.

[36] M. Lustig, D. Donoho, J. M. Pauly, Sparse MRI: The application of compressed sensing for rapid MR imaging, Magnetic Resonance in Medicine 58 (6) (2007) 1182–1195.

[37] J. Mairal, F. Bach, J. Ponce, G. Sapiro, A. Zisserman, Discriminative learned dictionaries for local image analysis, in: IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), 2008, pp. 23–28.

[38] F. Malgouyres, T. Zeng, A predual proximal point algorithm solving a non negative basis pursuit denoising model, Tech. Rep. ccsd-00133050, Centre pour la communication scientifique directe (CCSD) (2007).

[39] J. Moreau, Proximité et dualité dans un espace hilbertien, Bulletin de la S.M.F. 93 (1965) 273–299.

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

[41] M. Nikolova, Markovian reconstruction using a GNC approach, IEEE Trans. on Image Processing 8 (9) (1999) pp. 1204–1220.

[42] M. Nikolova, J. Idier, A. Mohammad-Djafari, Inversion of large-support ill-posed linear operators using a piecewise Gaussian MRF, IEEE Trans. on Image Processing 8 (4) (1998) 571–585.

[43] M. Nikolova, M. K. Ng, S. Q. Zhang, W. K. Ching, Efficient reconstruction of piecewise constant images using nonsmooth nonconvex minimization, SIAM Journal on Imaging Sciences 1 (1) (2008) 2–25.

[44] R. Nowak, M. Figueiredo, Fast wavelet-based image deconvolution using the em algorithm, in: Proceedings of the 35th Asilomar Conference on Signals, Systems, and Computers, 2001, pp. 371–275.

[45] K. O'Brien, K. M. O'Brien, Z. Sura, T. Chen, T. Zhang, Supporting OpenMP on Cell, International Journal of Parallel Programming 36 (3) (2008) 289–311.

[46] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (1) (2007) 80–113.

[47] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, K. Yazawa, The design and implementation of a first-generation CELL processor, in: Proceedings of the Solid-State Circuits Conference, 2005, pp. 184–185.

[48] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan, Larrabee: a many-core x86 architecture for visual computing, in: ACM SIGGRAPH 2008 papers, ACM, New York, NY, USA, 2008, pp. 1–15.

[49] R. Tibshirani, Regression shrinkage and selection via the lasso, Journal of the Royal Statistical Society Series B 58 (2006) 267–288.

[50] J. Tropp, Just relax: Convex programming methods for identifying sparse signals, IEEE Trans. on Information Theory 51 (3) (2006) 1030–1051.

[51] J. A. Tropp, Greed is good: algorithmic results for sparse approximation, IEEE Trans. on Information Theory 50 (10) (2004) 2231–2242.

[52] S. Williams, J. Carter, L. Oliker, J. Shalf, K. Yelick, Lattice boltzmann simulation optimization on leading multicore platforms, IEEE International Parallel and Distributed Processing Symposium (2008) 1–14.

[53] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, K. Yelick, The potential of the Cell processor for scientific computing, in: Proceedings of the 3rd conference on Computing frontiers (CF), ACM, New York, NY, USA, 2006, pp. 9–20.

[54] J. Wright, A. Yang, A. Ganesh, S. Sastry, Y. Ma, Robust face recognition via sparse representation, IEEE Trans. on Pattern Analysis and Machine Intelligence 31 (2) (2009) 210–227.

[55] W. Yin, S. Osher, D. Goldfarb, J. Darbon, Bregman iterative algorithms for l1-minimization with applications to compressed sensing, SIAM Journal on Imaging Sciences 1 (1) (2008) 143–168.

[56] K. Yosida, Functional Analysis, Springer, 1965.