Politehnica University of Bucharest
Faculty of Automatic Control and Computers
Department of Automatic Control and Systems Engineering

PHD THESIS

Parallel Dictionary Learning Algorithms for Sparse Representations

Paul Irofti

Advisor
Prof. dr. ing. Bogdan Dumitrescu

Bucharest, 2015

Contents

List of Figures

List of Tables

Nomenclature

$\lfloor a \rfloor$	real number a rounded towards $-\infty$
$\mu(A)$	mutual-coherence of matrix A
$\mathbb{R}$	real numbers set
$\|\|a\|\|_0$	l_0 pseudo-norm
$\|\|A\|\|_F$	Frobenius norm of A
A^T, a^T	transpose of matrix A, transpose of vector a
A_i, a_i	column i of matrix A
$a_{i,j}, A_{i,j}$	element at line i and column j of matrix A
E_x	energy of signal x
AK-SVD	approximate K-SVD algorithm
atoms	the columns of a given dictionary D
BP	basis pursuit
BPDN	basis pursuit denoising
CPU	central processing units
CU	compute unit
DL	dictionary learning
FOCUSS	focal underdetermined system solver
GPGPU	general-purpose programming on GPU
GPU	graphical processing unit
GWS	global work size
JAU	Jacobi atom updates
LAOLS	look-ahead OLS
LDS	local data store memory, local memory
LP	linear programming
LS	least-squares
LWS	local work size
MOD	method of optimal directions
NDR	n-dimensional range definition
NSGK	new SGK algorithm
OLS	orthogonal least squares
OMP	orthogonal matching pursuit
OpenCL	open computing language
PE	processing element
POLS	projection-based OLS

POMP	projection-based OMP	SBO	single block orthogonal algorithm
RAM	random access memory	SGK	sequential generalization of K-means algorithm
RIP	restricted isometry property	UONB	union of orthonormal basis algorithm
RMSE	room mean square error	VGPR	vector general purpose register

Chapter 1

Prologue

Sparse representations are intensively used in signal processing applications, like image coding, denoising, echo channels modeling, compression and many others. Recent research has shown encouraging results when the sparse signals are created through the use of a learned dictionary. This raised the following optimization problem

$$\begin{aligned} \underset{D,X}{\text{minimize}} \quad & \|Y - DX\|_F^2 \\ \text{subject to} \quad & \|x\|_0 \leq s, \ \forall x \in X, \end{aligned}$$

that attempts to find the best dictionary D generating the best sparse representations X when modeling a given set of signals Y with a target sparsity of s imposed on each sparse signal x from X. This is a hard open problem for which the signal processing field has been able, so far, to only offer local minima solutions that are usually very involved and take a long time to process.

The current study focuses on finding new methods and algorithms, that have a parallel form where possible, for obtaining sparse representations of signals with improved dictionaries that lead to better performance in both representation error and execution time.

In Part I we start by describing in Chapter 2 the tools for obtaining the sparsest solutions and the guarantees they provide along with the currently available dictionary design strategies. We continue with Chapter 3 where we present the OpenCL parallelism framework along with the analysis and efficiency indicators for optimal implementations.

In Part II we attack the general dictionary learning problem by first investigating and proposing new solutions for the sparse representation stage in Chapter 4 and then moving on to the dictionary update stage in Chapter 5 where we propose a new parallel update strategy and describe its effect on existing algorithms. Lastly, we study in Chapter 6 how mixing different representation algorithms with different dictionary update methods affects the quality of the final dictionary.

Part III focuses on dictionary learning solutions where the dictionary has a specific form. In Chapter 7 we propose a new parallel algorithm for dictionaries structured as a union of orthonormal bases. Next, in Chapter 8, we study the cosparse view on dictionary learning and propose new algorithms for creating cosparse orthogonal dictionaries. Finally, in Chapter 9 we analyse denosing through dictionary learning and propose new methods based on composite dictionaries.

We conclude this thesis in Part IV where we list and describe our contributions and present future research directions.

Part I

Field Overview

Chapter 2

Dictionary Learning

We start this chapter by presenting the sparse representation field [1–3] and its guarantees in providing the sparsest solution to a linear system of equations. In the second part we shift focus from the solution towards its representation matrix and the different design strategies used for promoting sparsity.

2.1 The l_0 Pseudo-Norm

We attempt to provide sparse solutions for the underdetermined linear systems of equations of the form

$$Ax = b, \tag{2.1}$$

where $A \in \mathbb{R}^{n \times m}$ is full rank, with $n < m$.

Our interest is providing the sparsest solutions possible and so we will focus almost exclusively on the l_0 pseudo-norm which acts as a non-zero indicator on the support of a given vector x:

$$\|x\|_0 = |\{i : x_i \neq 0\}|, \tag{2.2}$$

where $|.|$ denotes the cardinality of the set.

It is called pseudo-norm because it does not satisfy all the properties of a proper norm.

Definition 2.1 *A proper norm follows:*

- *zero vector:* $\|v\| = 0$ *if and only if* $v = 0$
- *absolute homogeneity:* $\|tv\| = |t|\|v\|$, $\forall t \neq 0$
- *triangle inequality:* $\|u + v\| \leq \|u\| + \|v\|$

While l_0 respects the zero vector and the triangle inequality properties, the non-zero element counter behaviour breaks absolute homogeneity.

The l_0 optimization problem for solving (2.1) is

$$\min_x \|x\|_0 \quad \text{subject to} \quad Ax = b. \tag{2.3}$$

New and existing solutions, algorithms and implementations for this optimization problem are discussed and analyzed in detail in Chapter 4.

2.2 When is a solution the sparsest solution?

Knowing that l_0 provides us with the most sparse solution possible we present here some of its known properties such as uniqueness, stability and performance in order to motivate its core position in our study.

2.2.1 Uniqueness Constructs

Given a sparse solution x of the linear system $Ax = b$, can we tell if it is the sparsest one? In this subsection we present the tools available for answering this question.

The first result is based on the property of the matrix A called the *spark* [4,5].

Definition 2.2 *The spark represents the smallest number of linearly-dependent columns of a given matrix.*

The spark is used in the following theorem:

Theorem 2.1 *If a given sparse solution x of the linear system $Ax = b$ has $\|x\|_0 < spark(A)/2$ then it is also the sparsest.*

Finding the spark is not an easy task because it involves sweeping through all possible column subsets and perhaps that is why weaker but faster bounds such as the mutual-coherence [6–8] bound are preferred.

Definition 2.3 *The mutual-coherence is the absolute largest normalized inner-product of the columns of a given matrix.*

For a matrix A, this can be formally written as

$$\mu(A) = \max_{i \neq j} \frac{|a_i^T a_j|}{\|a_i\|_2 \|a_j\|_2}. \tag{2.4}$$

Mutual-coherence lower-bounds the spark, as proven in [2], according to the following relationship:

$$\operatorname{spark}(A) \geq 1 + \frac{1}{\mu(A)} \tag{2.5}$$

and can guarantee the sparsest solution was reached as shown in [8]:

Theorem 2.2 *If a given sparse solution x of the linear system $Ax = b$ has $\|x\|_0 < \frac{1}{2}\left(1 + \frac{1}{\mu(A)}\right)$ then it is also the sparsest.*

2.2.2 Stability for Sparse Approximations

When performing sparse approximations, instead of exact solutions, the uniqueness of a given solution, using tools like the ones presented in the last subsection, can no longer be claimed.

Approximations can present feasible solutions that are susceptible to scaling and, worse, they can have different supports while maintaining the same l_0 cardinality that still make them the sparsest possible. That is why we are interested in the stability [9, 10] of these solutions: if a solution is found that is also the sparsest we want to be sure that it is also the best within a given margin ϵ. Put differently, we want to be sure that the others are also nearby.

A mutual-coherence based result from [9] states that:

Theorem 2.3 *Given a sparse solution x_0 that satisfies the mutual-coherence criteria from Theorem 2.2 and that provides an approximation within a given error ϵ ($\|b - Ax_0\| < \epsilon$), then the distance to every other feasible sparsest solution x is bounded by*

$$\|x - x_0\|_2^2 \leq \frac{4\epsilon^2}{1 - \mu(A)(2\|x_0\|_0 - 1)}. \tag{2.6}$$

Another powerful tool that can also be used for stability analysis is the restricted isometry property [11, 12] (RIP):

Definition 2.4 *Given a matrix A with normalized columns and an integer s such that A_s represents the matrix A restricted to only s columns, suppose there exists a quantity δ_s, that is also the smallest, such that for all possible submatrices A_s the following holds true:*

$$(1 - \delta_s)\|c\|_2^2 \leq \|A_s c\|_2^2 \leq (1 + \delta_s)\|c\|_2^2 \quad \forall c \in \mathbb{R}^s. \tag{2.7}$$

Then A is said to have an s-RIP with a δ_s constant.

Using this definition it has been shown in [2] that we can get to the same stability claims from Theorem 2.3.

2.2.3 Performance

Sweeping across all the column subsets of A to find the sparsest solution or approximation x to the linear system $Ax = b$ is clearly a hard and time consuming task. Even so, empirical evidence of various greedy pursuit algorithms has shown good results which triggered researchers to look at the average performance instead of focusing on the worst-case scenario.

Probabilistic average performance analysis started with the work from [13] on fixed and structured matrices A. Others shortly followed [14–17] and showed that even though the worst-case scenario is grim we can still hope for good enough results when employing pursuit algorithms. Soon after, a breakthrough from [18], continued in [19], has shown that the above results also hold for unstructured random matrices.

These results motivate our focus on greedy pursuit algorithms that work in tandem with both structured and random matrices A.

2.3 Dictionary Types

So far we looked at how we can find the sparsest solution or approximation x of the underdetermined linear system $Ax = b$ where both A and b are fixed. In this section we turn our focus towards the properties of A and its sparsifying effect on the final solution x. If we turn this into a representation problem, then x is the expression of b through the matrix A. A sparse representation of x would then seek the sparsest solution which is the solution that uses the least amount of columns from A in its representation. From this perspective the literature often refers to matrix A as a representation dictionary D whose columns are termed atoms.

The crude dictionary form is built from popular matrix transforms such as Fourier, discrete cosine or wavelets [20] whose components are vectorized as dictionary atoms. The created dictionary remains fixed and is used universally for the representation of all signal types. Picking a dictionary and using it ubiquitously will be very efficient for certain classes of signals, but also quite disappointing for others.

A different approach uses pre-designed, but still fixed, dictionaries that are used only for specific classes of signals. Such tuned dictionaries can be built using something like wavelet packets [21], bandlets [22], curvelets [23] or contourlet [24]. While this loses the generality of the former universal approach it provides substantially better representations. Even so, when we are provided with such a dictionary, we can still obtain mediocre results on our signal sets due to D being too specialized or too general.

Algorithm 1: Dictionary learning – general structure

```
1 Arguments: signal matrix $Y$, target sparsity $s$
2 Initialize: dictionary $D$ (with normalized atoms)
3 for $k = 1, 2, \dots$ do
4     With fixed $D$, compute sparse representations $X$
5     With fixed $X$, update atoms $d_j$, $j = 1 : n$
```

2.4 Learning

The shortcomings of the above methods have paved the way for a third versatile approach called dictionary learning (DL) [25,26]. The central idea in DL is to first train a specific dictionary on a relevant training set until a well-suited specialized final dictionary is obtained that can be used in fixed form to perform efficient representations of other signal sets of the same class. This strategy has been shown [27] to be very well suited for sparse representations and our study will gravitate around this approach when investigating and proposing new dictionary designs methods.

Formally, the DL problem is posed as follows. Given a data set $Y \in \mathbb{R}^{p \times m}$, made of m vectors (signals or data items) of size p, and a sparsity level s, the aim is to solve the optimization problem

$$\begin{aligned} \underset{D,X}{\text{minimize}} \quad & \|Y - DX\|_F^2 \\ \text{subject to} \quad & \|x_i\|_0 \le s, \ 1 \le i \le m, \end{aligned} \tag{2.8}$$

where the variables are the atoms of dictionary $D \in \mathbb{R}^{p \times n}$, and the sparse representations matrix $X \in \mathbb{R}^{n \times m}$, whose columns have at most s nonzero elements. By x_i we denote the i-th column of the matrix X and by $\|\cdot\|_F$ the Frobenius norm of a matrix. In practice, the dictionary D is initialized either randomly or by a random selection of signal vectors. The norm of the atoms is forced to 1 at all stages, in order to eliminate the multiplicative indeterminacy in the product DX.

Algorithm 1 presents the general structure of most DL algorithms. First, a sparse representation algorithm (see Chapter 4) is used to compute the sparse representation matrix X. Note that the problem can be decoupled, since each column of X can be computed separately by attempting to find a sparse solution through least-squares (LS) $Dx_i = y_i$, for $i = 1 : m$. Then, the atoms are updated using different methods (see Chapter 5), all aiming to reduce the objective of (2.8).

Among the existing dictionary design algorithms that attempt to solve problem (2.8) we mention method of optimal directions (MOD) [28], K-SVD [29], approximate K-SVD (AK-SVD) [30], sequential generalization of K-means (SGK) [31], new SGK (NSGK) [32], union of orthonormal basis(UONB) [33], and single block orthogonal algorithm (SBO) [34] which we discuss in Chapters 5 and 7. Our research proposes parallel alternatives for most of the above algorithms and provides a new parallel DL framework called Jacobi Atom Updates (JAU).

2.5 Experiments

We tested the performance of the DL methods for two standard problems: recovery of a given dictionary and dictionary training for sparse image representation.

2.5.1 Dictionary Recovery

Following the numerical experiments from [31] and [32], when performing dictionary recovery tests we generate a random dictionary with $n = 50$ atoms of size $p = 20$ each, and a signal set Y of $m = 1500$ data vectors, each vector being generated as a linear combination of $s \in \{3, 4, 5\}$ different atoms. We then add white gaussian noise with SNR levels of 10, 20, 30 and ∞ dB to the signal set. We apply $9s^2$ dictionary learning iterations (step 3 in Algorithm 1) on this signal set for each algorithm and compare the resulting dictionaries with the original. Following the method from [29], if the scalar product between two atoms, one from the resulting dictionary and one from the original dictionary, is larger than 0.99 in absolute value (with 1 being the maximum value because the atoms are normalized), then the atom is considered successfully recovered. This verification process can be easily implemented by processing the correlation matrix between the original and the trained dictionary. The dictionary is initialized with a random selection of data vectors. The algorithms are given the fixed sparsity target s that was used to generate the original signal set. Unless otherwise stated, our results present percentages of recovered atoms averaged over 50 runs.

2.5.2 Dictionary Training

When generating training signal sets we used colored and gray scale bitmap images taken from the USC-SIPI [35] image database (e.g. Barbara, Lena, boat etc.). The images were normalized and split into random 8×8 blocks representing the patches. The initial dictionary was built similarly or by generating random atoms; when comparing different algorithms, the initialization was always the same. We compare the resulting sparse image representations with the original signals by computing the root mean square error (RMSE)

$$\mathrm{RMSE} = \frac{\|Y - DX\|_F}{\sqrt{pm}}. \tag{2.9}$$

Chapter 3

Parallelism with OpenCL

Signal processing has been a field of active development for GPU algorithms and, in the last few years, for OpenCL [36] implementations. The problems solved in this framework are typical for image or video processing, as naturally fit for GPUs. There is work on segmentation [37], feature matching [38], motion estimation [39] and real time particle filtering [40], among others. Also more intensive computation tasks have been tackled, like the solution of optimization problems (rank minimization) [41]. Closer to our interest, we see algorithms for computing sparse representations [42].

In this chapter we focus on the details regarding the OpenCL standard and the hardware using it underneath. This will create the necessary context in which future chapters describe and analyze various parallel dictionary learning implementations.

OpenCL is an open standard allowing portable parallel programming, aimed especially at graphics processing units (GPU) but not restrained to them. Despite its recent proposal, OpenCL has gained support from the industry and its implementation is supported by the major GPU manufacturers. Although some implementations miss certain features and there are difficulties in portability [43], there are much more incentives for using OpenCL than languages specialized to a single type of GPUs.

Despite the fact that its original motivation can be clearly tracked back to the increasing need for general-purpose programming on graphical processing units (GPGPU), the standard defines an abstract device type that can be modeled by other types of hardware like central processing units (CPUs), digital signal processors (DSPs) and field-programmable gate arrays (FGPAs).

3.1 Hardware Abstraction

3.1.1 Execution

OpenCL abstracts the smallest execution unit available in hardware as processing elements (PE), that are organized in equally sized groups at which parallelism is guaranteed, called compute units (CU), located on the OpenCL device. Compute units can also be executed in parallel.

Each PE has its own private memory, with no visibility on the outside, which is the fastest type of memory on the OpenCL device but also the smallest. PEs share resources locally, within the compute unit, and globally, on the OpenCL device. The local memory of a CU is visible only to its PEs and represents the middle-ground between size and latency. All PEs from all CUs can read from and write to global memory. This is the largest type of memory but also the slowest. Although optional, on some devices a fraction of it can be reserved for read-only memory that can be used as a caching mechanism. The global memory is the only type of memory accessible from the outside by the host on which the OpenCL device is installed.

In Figure 3.1 we present a theoretical OpenCL device that is composed of two compute units with four processing elements each. Also visible in this example are the private memory associated with each PE along with the local memory of each CU and the device global memory at the bottom.

3.1.2 Scheduling Work

OpenCL does not permit individual PE or CU control. We can not execute different tasks on different PEs like we do on regular CPUs. Instead, work is scheduled simultaneous for all PEs across the entire device. Each PE executes the same task and its work is differentiated in software through indexing as described in the following subsection. Once the current task is executed by all PEs a new one can take its place.

If a task consumes too much local or private memory the OpenCL scheduler can disable a PE subset from executing in order to accommodate the resource requirements: if 10 PEs require 10KB of memory and the physical limit is 5KB, the scheduler could decide to keep 5 PEs active and 5 idle in order to fit all the data in memory.

The standard allows the task to request a number of individual processing items called work-items that are all guaranteed to be executed on the device's PEs, but the parallelism is left up to the scheduler. The task can group work-items into work-groups. The work-items within one work-group are executed by the PEs from a single compute unit.

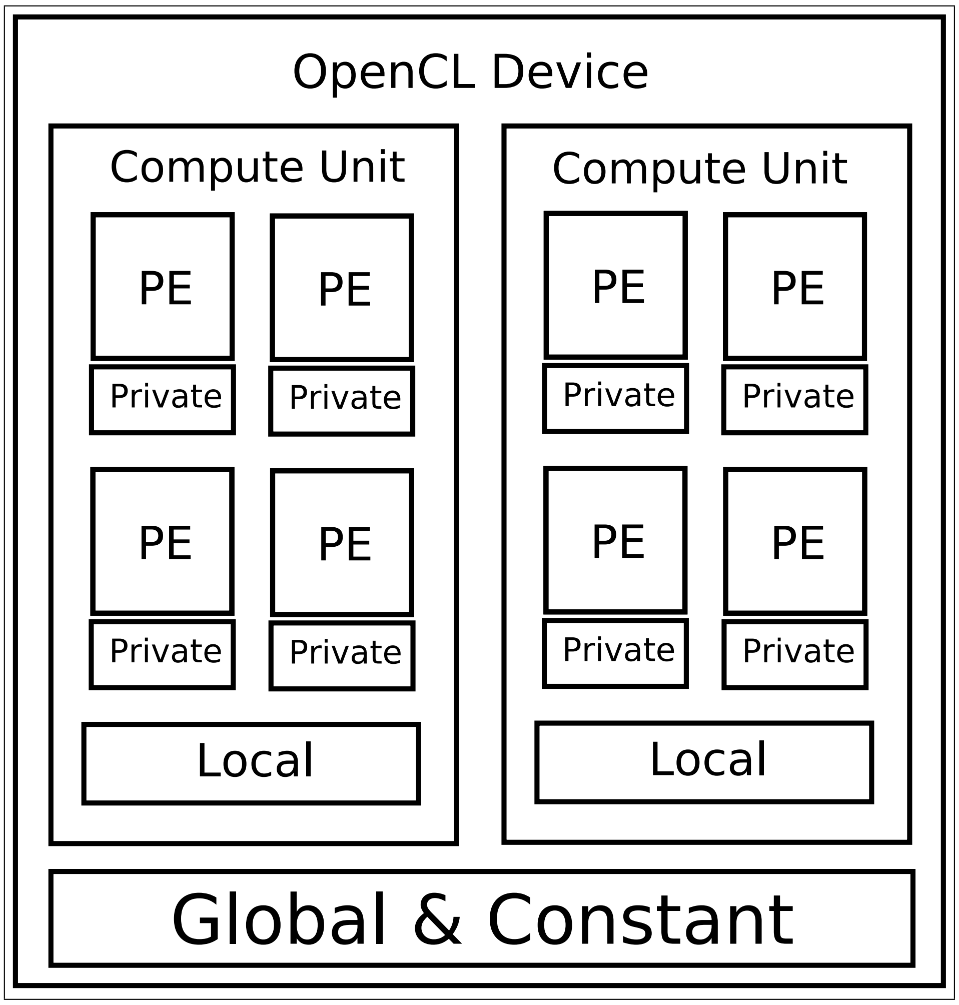

Figure 3.1: OpenCL hardware abstraction.

The way a task splits work has a direct impact on the scheduler and its ability to execute work-items and work-groups in parallel.

3.1.3 Topology

Work-items consist of identical small functions (also called kernels) and are organized in an n-dimensional space that is defined in software by the application. This logical split allows to differentiate work among work-items through indexing: each work-item has a global unique ID and a local ID unique within the compute unit. The chosen dimension creates a tuple of cartesian coordinates that define each of these global and local IDs. OpenCL currently allows no more than 3 dimensional spaces to be defined.

For a bidimensional split of the PE set, we can denote the n-dimensional range definition as NDR($\langle G_x, G_y\rangle$, $\langle L_x, L_y\rangle$). There are $G_x \times G_y$ PEs, organized in work-groups of size $L_x \times L_y$, running the same kernel.

No matter of the defined NDR, all work-groups have an identical number of work-items and the local work-group dimensions have to fit perfectly within the global dimensions. If we note with W_x and W_y the number of work-groups in each dimension from our 2D example, then the following relationship holds:

$$\begin{aligned} L_x &= G_x/W_x \\ L_y &= G_y/W_y. \end{aligned} \tag{3.1}$$

Through the relation between these dimensions we can also find out the global ID of an work-item from its local ID:

$$\begin{aligned} g_x &= w_x * L_x + l_x \\ g_y &= w_y * L_y + l_y, \end{aligned} \tag{3.2}$$

where $g_{x,y}$ is the work-item global ID, $w_{x,y}$ is the work-group ID and $l_{x,y}$ is the work-item local ID. And vice-versa, we can find out the work-group ID and the local work-item ID just from its global ID:

$$\begin{aligned} w_x &= g_x/L_x \\ w_y &= g_y/L_y \\ l_x &= g_x\%L_x \\ l_y &= g_y\%L_y. \end{aligned} \tag{3.3}$$

Figure 3.2 presents a 2D NDR example that helps visualizing the dimensions and indexing process described above. There, we organized a total of 8 work-groups, each with 16 work-items, in 2 rows and 4 columns: NDR($\langle 16, 8\rangle$, $\langle 4, 4\rangle$).

In this thesis we focus on GPU implementations of our methods, but we note that our solutions can be applied on any multicore setup (including regular CPUs and FPGAs) that adheres to the OpenCL standard.

3.2 GPU Particularities

While GPUs are able to successfully model the abstract device from Figure 3.1 the hardware underneath is usually quite different and knowing a bit about its particularities helps us design efficient kernels.

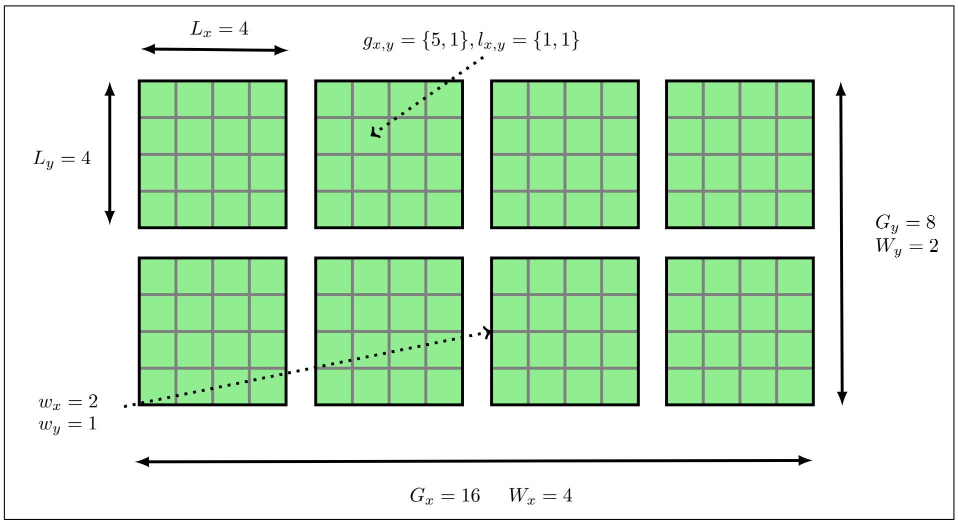

Figure 3.2: 2-dimensional split example where indexing starts from 0

Memory. GPUs have a much smaller memory when compared to CPUs and their processing elements operate at lower frequencies. Private memory is represented by register memory on CPUs, but on GPUs it can be incorporated in a dedicated memory bank on each CU or it might also be absent. GPU register memory is built from the set of vector general purpose registers (VGPRs). OpenCL local memory is implemented in some GPUs [44] by on-chip local data store (LDS) and on CPUs through cache or a dedicated memory range from random-access memory (RAM). Global memory is built from RAM chips on both GPUs and CPUs.

Work. A compute unit on the GPU is composed of one or more wavefronts [44]. Wavefront is the smallest grouping at which parallelism takes place. The processing elements from a wavefront execute each instruction from an OpenCL kernel at once by locking on to the instruction pointer: a kernel instruction is processed by all elements and when the last PE finishes execution the entire wavefront moves on to the next instruction. The size of a wavefront is hardware specific. Wavefronts are sometimes also referred to as waves.

Synchronization. Work-groups are further split into work-item groups the size of a wavefront. Work-items from different wavefronts can be synchronized through the use of memory barriers. Barriers ensure memory consistency by stopping execution of the work-items reaching it until all of them do and waiting for all local or global memory writes to settle. Within a wavefront, divergent paths (such as *if-else* branches) are synchronized by concatenating all possible paths and forcing every work-item to run every instruction from this reunion. This ensures lock-step instruction execution within the wavefront and the hardware takes care of the side-effects generated by this approach. This divergence cost can be minimized through special OpenCL instructions such as *select* that can replace *if-else* blocks.

3.3 Performance and Occupancy

The main goal when designing an OpenCL kernel is performance through parallelism. When the target is a GPU device, this often implies ensuring full resource occupancy. In other words we need to make sure that all the PEs are actively processing at any given time during the execution of our kernel.

There are of course corner cases where the best performance does not imply full occupancy like when the time cost of transferring data from the host to the OpenCL device or from global to local memory is higher than the actual execution time.

A strategy for minimizing the memory input-output (I/O) impact on performance is hiding the I/O costs by scheduling more work-groups than the available CUs in order to increase the chances of replacing one work-group waiting on I/O with another that is ready for execution. Thus the n-dimensional split that we choose for our kernels is tightly coupled with the I/O we need to do. The bigger the global work size (GWS) is compared to the local work size (LWS) the more work-groups we have.

The NDR needs to also take into account the wavefront size when it defines the local work size because, as described in the last section, wavefronts split work-groups for parallel execution and if the work-group size (or LWS) is not an integer multiple of the wavefront size, some PEs might be forced to remain idle for padding during execution. In this scenario device occupancy would be suboptimal.

Even with a properly defined NDR, high memory requirements might force the GPU to schedule less work (see Section 3.1.2). Having too many data-structures shared among PEs in local memory (or LDS) will lead to fewer active wavefronts within a CU. Abuse of auxiliary variables or improper floating-point vectorization can lead to VGPR starvation which, in turn, limits the total number of active PEs due to private memory depletion.

Figure 3.3 presents the occupancy analysis of our matrix multiplication kernel generated by CodeXL [1]. In the left panel we can see how work-group

[1] We used CodeXL version 1.7 available at `http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/`

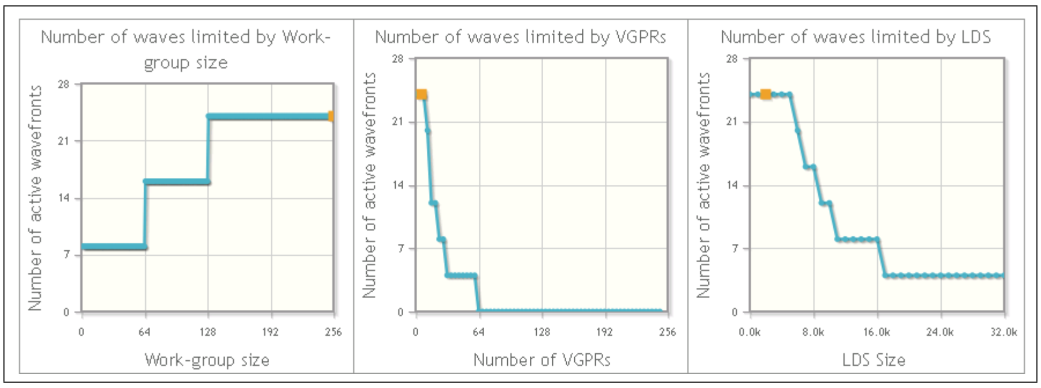

Figure 3.3: Matrix multiplication kernel occupancy

size influences the total number of active wavefronts. The orange dot indicates our current setting. The next two panels focus on the usage of private and, respectively, local memory of our kernel. We can see that the number of used VGPRs (middle) has a more drastic effect on the number of active waves than the amount of used LDS (right).

3.4 Experiments

We tested our OpenCL implementations on an ATI FirePro V8800 (FireGL V) card from AMD, running at a maximum clock frequency of 825MHz, having 1600 streaming processors, 2GB global memory and 32KB local memory. Also, the CPU tests for our C implementations were made on an Intel i7-3930K CPU running at a maximum clock frequency of 3.2GHz.

As a rule, we chose data dimensions as powers of two because this way the data objects and the work-loads are easier divided and mapped across the NDRs without the need for padding.

3.5 Conclusions

In this chapter we briefly described the OpenCL device as abstracted by the standard and how execution units can be organized as an n-dimensional space. We also detailed how GPUs model this abstract device and the hardware used to build the different types of memory. Lastly, we presented the main factors that influence the design and efficiency of OpenCL kernels and their execution. In the following chapters we will use these concepts to motivate and describe our dictionary learning implementations.

Part II

General Dictionary Learning

Chapter 4

Sparse Representations

4.1 Introduction

When looking at the sparse representation problem we are interested in finding the representation with the largest number of zeros in its support that uses a known fixed dictionary to represent a given full signal. This can be formalized as the following optimization problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \|x\|_0 \\ \text{subject to} \quad & y = Dx, \end{aligned} \tag{4.1}$$

where y is the signal, D the dictionary, and x the resulting sparse representation. This is a hard problem and most of the existing methods propose an alternative to (4.1) by approximating y following a sparsity constraint s:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \|y - Dx\|_2^2 \\ \text{subject to} \quad & \|x\|_0 \leq s \end{aligned} \tag{4.2}$$

or an error threshold ϵ:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \|x\|_0 \\ \text{subject to} \quad & \|y - Dx\|_2^2 \leq \epsilon. \end{aligned} \tag{4.3}$$

In all of the above formulations we can split the pursuit of x in two parts: finding the best few columns from D, to be used as the support of x, and then filling its non-zero entries with the coefficients found through least-squares (LS). Denoting $\mathcal{I}$ as the support set and $D_{\mathcal{I}}$ the restriction of D to the columns belonging to $\mathcal{I}$, we can compute the representation as

$$x_{\mathcal{I}} = (D_{\mathcal{I}}^T D_{\mathcal{I}})^{-1} D_{\mathcal{I}}^T y, \tag{4.4}$$

where $x_{\mathcal{I}}$ are the coefficients corresponding to the current support and thus the other elements of x are zero. We can reason that because the support $\mathcal{I}$ is limited to a few entries, the algorithms that construct it will have a reasonable execution time which probably explains why most of the proposed solutions from the field approach the problem through greedy methods.

The popular dictionary design algorithms that we will present in Chapter 5 (MOD [28], K-SVD [29], AK-SVD [30], SGK [31], NSGK [32]) use Orthogonal Matching Pursuit (OMP) [45] in the first stage to compute the sparse representations. The main reason is that OMP is fast and also that it is used in applications together with the optimized dictionary; it makes sense to appeal to the same representation algorithm in training the dictionary as well as in using it. However, there are other greedy algorithms, like Orthogonal Least Squares (OLS) [46], Subspace Pursuit [47], Projection-Based OMP (POMP) or Look-Ahead OLS (LAOLS) [48], that are still fast enough for wide practical use, but achieving typically better representations than OMP. OLS can be implemented efficiently as an orthogonal triangularization with pivoting, while POMP and LAOLS are able to trade off complexity and representation error. Besides the greedy category, the class of Basis Pursuit algorithms [1] offers a number of algorithms with very good representation error, but much slower than OMP and its improved versions.

Our first contribution here is an empirical investigation of the impact that the sparse representation algorithm has in DL. The spark to start this study was the somewhat disconcerting fact that, in the dictionary recovery test problem (see Section 2.5.1), often used in the DL community, several algorithms gave quite similar result. The immediate but apparently ignored question is if OMP is the bottleneck of DL algorithms and thus progress in dictionary design might be masked by it. Would better representation algorithms allow to discern which atom update method is in fact superior ? Would better representation algorithms cause a significant decrease of the overall error in the DL problem (2.8) ? Although we cannot provide definitive answers, we try at least to gain more insight into the DL process and assess DL algorithms on a steadier ground.

The second contribution is orientated towards the technical details of efficiently implementing the representation algorithms for heterogeneous parallel systems. Here we present in detail a case-study of the OpenCL implementation of OMP and then proceed to show and discuss the obtained numerical results and their associated running times when executing on a multiprocessor graphical device unit.

The chapter is structured as follows. In Section 4.2 we review several greedy representation algorithms that are good candidates to replace OMP. Among them, there is a new proposal, Projection-Based OLS (POLS), naturally derived from POMP. Section 4.3 focuses on parallelizing the algorithms and presents an in-depth analysis of the OMP implementation for the GPU. We close up by presenting the slower but better performing alternatives to the greedy approach in Section 4.4.

4.2 Greedy Minimizations

For the sake of completeness, we review here the considered sparse representation algorithms, although without giving all the implementation details. We will assume that a generic function is available for finding the least-squares solution (4.4) to the system $A_\mathcal{I} x_\mathcal{I} = b$ with known support $\mathcal{I}$ and denote its use by

$$x = \mathrm{LS}(A, b, \mathcal{I}). \tag{4.5}$$

Since in greedy algorithms the support usually grows on the current one, this function could be implemented efficiently by using an orthogonal partial triangularization of A or a partial Cholesky factorization of the associated normal matrix $A^T A$. However, we will leave these details out of the presentation and give only the main ideas of the methods.

Orthogonal Matching Pursuit (OMP) [45], presented in Algorithm 2, grows the support by looking at the correlations of the matrix columns (atoms) with the current residual and adding the index of the largest correlation to the support; this is traditionally called matching pursuit criterion. Then, it computes the LS solution for the current support and updates the corresponding residual, thus preparing the next iteration. The residual is orthogonal on the selected columns, hence a column cannot be twice selected. (This is also the reason for the word "orthogonal" in OMP.) The interpretation of the selection criterion is simple: the new column is the one that decreases the most the residual norm, keeping fixed the values of the current solution $x_\mathcal{I}$.

Orthogonal Least Squares (OLS) [46] takes a slightly different approach. The next column is the one that, together with the previous ones, gives a solution that minimizes the residual norm; hence, the elements of $x_\mathcal{I}$ can change during the selection process. Algorithm 3 describes OLS. It can be efficiently implemented as an orthogonal triangularization with pivoting, which permanently orthogonalizes the not selected columns on the selected ones. This allows the computation of the selection criterion as a simple matrix-vector multiplication. In average, OLS is slightly better than OMP.

A refinement of OMP was proposed in [48], in the form of Projection-Based OMP (POMP), presented in Algorithm 4. Unlike OMP, a number of L candidate columns are selected via the matching pursuit criterion, where

Algorithm 2: Orthogonal Matching Pursuit (OMP)

```
1  Arguments: $A$, $b$, $s$
2  Initialize: $r = b$, $\mathcal{I} = \emptyset$
3  for $k = 1 : s$ do
4      Compute correlations with residual: $z = A^T r$
5      Select new column: $i = \arg\max_j |z_j|$
6      Increase support: $\mathcal{I} \leftarrow \mathcal{I} \cup \{i\}$
7      Compute new solution: $x = \text{LS}(A, b, \mathcal{I})$
8      Update residual: $r = b - A_{\mathcal{I}} x_{\mathcal{I}}$
```

Algorithm 3: Orthogonal Least Squares (OLS)

```
1  Arguments: $A$, $b$, $s$
2  Initialize: $\mathcal{I} = \emptyset$
3  for $k = 1 : s$ do
4      for $j \notin \mathcal{I}$ do
5          Build new support: $\mathcal{J} = \mathcal{I} \cup \{j\}$
6          Try solution: $x = \text{LS}(A, b, \mathcal{J})$
7          Residual norm: $\rho_j = \|b - A_{\mathcal{J}} x_{\mathcal{J}}\|^2$
8      Select new column: $i = \arg\min_j \rho_j$
9      Increase support: $\mathcal{I} \leftarrow \mathcal{I} \cup \{i\}$
10     Compute new solution: $x = \text{LS}(A, b, \mathcal{I})$
```

L is an argument of the algorithm. Then, an LS solution is computed for the support extended with all these columns. The winner is the column with the largest element of the solution. This approach is partly inspired from Subspace Pursuit [47], where selection is made by attempting to find LS solutions with larger support and looking at the magnitude of the solution elements: a large magnitude means a higher likelihood that the position belongs to the true support. It is clear that POMP with $L = 1$ is identical to OMP. Also, it is not necessarily true that POMP with $L > 1$ gives a better result than OMP, although this is usually the case.

An immediate extension, not investigated until now, is the Projection-Based OLS (POLS), presented in Algorithm 5, which is the application of the POMP selection idea in the context of OLS. Of course, for $L = 1$ POLS is identical to OLS. The performance of POLS with respect to POMP should be similar to that of OLS with respect to OMP, but we will be able to say more in the numerical experiments section.

Algorithm 4: Projection-Based Orthogonal Matching Pursuit (POMP)

```
1  Arguments: $A$, $b$, $s$, $L$
2  Initialize: $r = b$, $\mathcal{I} = \emptyset$
3  for $k = 1 : s$ do
4      Compute correlations with residual: $z = A^T r$
5      Select indices $\mathcal{J}$ of $L$ largest $|z_j|$
6      Compute potential solution: $x = \text{LS}(A, b, \mathcal{I} \cup \mathcal{J})$
7      Select largest element index: $i = \arg\max_{j \in \mathcal{J}} |x_j|$
8      Increase support: $\mathcal{I} \leftarrow \mathcal{I} \cup \{i\}$
9      Compute new solution: $x = \text{LS}(A, b, \mathcal{I})$
10     Update residual: $r = b - A_{\mathcal{I}} x_{\mathcal{I}}$
```

Algorithm 5: Projection-Based Orthogonal Least Squares (POLS)

```
1  Arguments: $A$, $b$, $s$, $L$
2  Initialize: $\mathcal{I} = \emptyset$
3  for $k = 1 : s$ do
4      Compute values $\rho_j$ like in steps 4–7 of OLS
5      Select indices $\mathcal{J}$ of $L$ largest $\rho_j$
6      Apply steps 6–9 of POMP
```

The last sparse representation algorithm that we use is Look-Ahead OLS (LAOLS), given in Algorithm 6. Like in POMP, L indices are selected at each iteration via the matching pursuit criterion. After appending each of these indices to the current support, OMP is run starting from this support to the completion of an s-sparse solution. The newly selected index is that giving the lowest residual for the final OMP solution. So, a look-ahead search is performed for each index selection. We note that in fact the proper name of this algorithm would be LAOMP, since OMP is run in step 7 in the selection process. An OLS version is easy to derive by replacing step 4 with steps 4–7 of OLS and using OLS instead of OMP in step 7. However, we did not pursue the investigation of such an algorithm due to the higher complexity of LAOLS with respect to the other presented algorithms.

Although we have mentioned Subspace Pursuit [47], we skip its presentation due to the poor results obtained in dictionary learning and we will not report any experiments with it.

Algorithm complexity. The significant instructions from an OMP iteration

Algorithm 6: Look-Ahead Orthogonal Least Squares (LAOLS)

```
1  Arguments: $A$, $b$, $s$, $L$
2  Initialize: $r = b$, $\mathcal{I} = \emptyset$
3  for $k = 1 : s$ do
4      Compute correlations with residual: $z = A^T r$
5      Select indices $\mathcal{J}$ of $L$ largest $|z_j|$
6      for $j \in \mathcal{J}$ do
7          Run OMP starting from $\mathcal{I} \cup \{j\}$, obtaining $x$
8          Compute residual norm $\rho_j = \|b - Ax\|^2$
9      Select new column: $i = \arg\min_j \rho_j$
10     Increase support: $\mathcal{I} \leftarrow \mathcal{I} \cup \{i\}$
11     Compute new solution: $x = \text{LS}(A, b, \mathcal{I})$
```

and their complexities are: the correlations in step 4, $\mathcal{O}(pn)$, the least squares computation from step 7 (which we assume to be incrementally computed for each new column), $\mathcal{O}(sp)$, and the residual update, $\mathcal{O}(sp)$. For s iterations, this amounts to a total complexity of $\mathcal{O}(spn+s^2p)$. POMP performs an extra LS operation in step 6 on the support and its L-sized extension $\mathcal{J}$. This amounts to L incremental LS calculations at each iteration, which results in a total $\mathcal{O}(sL(s+L)p)$ extra cost compared to plain OMP. If $L = \mathcal{O}(s)$, then the extra cost is $\mathcal{O}(s^3p)$.

An efficient implementation of OLS consists of two intensive tasks: selection via correlations (similar to OMP), $\mathcal{O}(spn)$ and performing the transformations needed by the partial orthogonal triangularization, $\mathcal{O}(spn)$. Even though OLS has a similar theoretical complexity as OMP, it is slower by a constant factor, due to a larger constant multiplying the complexity term spn. POLS adds a single computationally significant instruction to OLS which is, again, the LS on the extended support, that was shown earlier to total to an extra cost of $\mathcal{O}(sL(s+L)p)$.

The increase in complexity for POMP and POLS with respect to OMP and OLS is negligible for small s, but becomes significant when $s^2 > n$. (We assume that $L \leq s$, which is a good practical choice.)

While the fastest option for representation remains OMP, POLS (and the other alternatives) are still a good candidate for DL, where execution time is not critical and representation error is important. For a more in-depth comparison and analysis of the algorithmic complexity we refer the reader to Table I and Section V from [48].

Algorithm 7: Batch OMP

```
1  Arguments: $\alpha^0 = D^T y$, $G = D^T D$, sparse goal $s$
2  Initilize: $\alpha = \alpha^0$, $\mathcal{I} = \emptyset$, $L = 1$
3  for $k = 1 : s$ do
4      Select new column: $i = \arg\max_j |\alpha_j|$
5      Increase support: $\mathcal{I} \leftarrow \mathcal{I} \cup \{i\}$
6      if $k > 1$ then
7          Solve for $w \{Lw = G_{\mathcal{I},s}\}$
8          $L = \begin{pmatrix} L & 0 \\ w^T & \sqrt{1 - w^T w} \end{pmatrix}$
9      Compute new $x_I$ solution: $LL^T x_{\mathcal{I}} = \alpha^0_I$
10     Update: $\alpha = \alpha^0 - G_I x_{\mathcal{I}}$
```

4.3 Parallelism

The dictionary learning process, as described in Chapter 2, operates on large training signals sets that need to be sparsely represented. And so, following the equations from either (4.2) or (4.3), the first stage of the dictionary learning process is naturally parallel, since the signal representations are completely independent. Given that this applies to all algorithms described in Section 4.2, we picked OMP as a case-study as it is the popular choice in the literature. In the following subsections we will present the details of the OMP algorithm and describe its parallel OpenCL implementation.

4.3.1 Orthogonal Matching Pursuit

In our implementation we followed the Batch OMP (BOMP) algorithm variant described in [30]. The steps from Algorithm 7 present the operations necessary for performing sparse representation for one signal y. The authors of [30] show that the best time performance is obtained if we first precompute the scalar products between the atoms and themselves and between the atoms and the signal vectors (step 1). Since these are matrix multiplications of fairly large size, they can be easily parallelized. The selection of columns is made by the standard matching pursuit criterion in step 4. Then, the algorithm builds the Cholesky decomposition of the matrix of the normal system associated with the sparse least-squares problem (steps 6–8) and computes the new representation by solving it in step 9. The residual update is no longer necessary and can be replaced by the expression from step 10 (as ex-

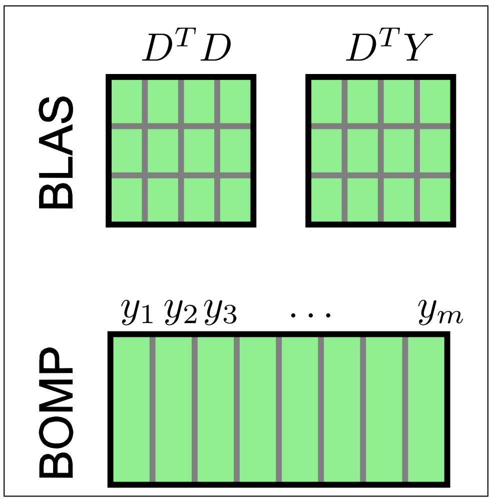

Figure 4.1: BOMP

plained in [30]) which has a lower computation complexity. We compute the sparse representations in parallel for groups of $\tilde{m}$ signals.

4.3.2 OpenCL Implementation

The matrix precomputations needed by BOMP were performed by a dedicated BLAS kernel that implements block matrix multiplication. Since the two multiplications are independent of each other, they can be also performed in parallel. We depict these operations in the first part of Figure 4.1 where each grid represent one BLAS operation and each grid element represents a block matrix multiplication.

We keep the input and resulting matrix in global memory. Each work-group performs the operations required for calculating one block from the result matrix. We organized the PEs in a 2-dimensional space that is further split into 2-dimensional block-size dependent work-groups. Full resource occupancy of our GPU (an indicator of maximum performance) was achieved when using work-groups of 16×16 PEs. Thus for a result matrix $A \in \mathbb{R}^{n \times m}$ we defined our n-dimensional space as: NDR($\langle n, m \rangle$, $\langle 16, 16 \rangle$). Before doing the actual multiplication, each work-item within a work-group copies a few elements from the input block sub-matrices into vectorized variables in local memory. On our device, the fastest vectorized type was float4.

All the operations required for the sparse representation of a single signal, were packed in and implemented by a single OpenCL kernel.

The input matrices as well as the resulting sparse signal are kept in global memory. The BLAS operations required for performing the Cholesky update and for recalculating the residual are done sequentially inside the BOMP kernel, not through a separate call to the BLAS kernel. Due to the rather small size of the matrices involved in these operations, measurements showed that using a dedicated kernel (as for precomputing the matrices from step 1) does not even begin to pay for the required GPU I/O. In-lining proved to be a lot faster.

The main obstacles we encountered during the implementation were memory bound. BOMP is a huge memory consumer and mostly due to auxiliary data. The necessary memory is of size $O(ns)$. Keeping all the auxiliary data in local memory would permit only the processing of one signal per compute-unit, corresponding to an $NDR(\langle \tilde{m} \rangle, \langle 1 \rangle)$ splitting. This would be wasteful as it would not reach full GPU occupancy and thus it would not cover the global memory latency costs.

After trying several work-group sizes, like 64, 128 and 256, we decided to leave the decision to the GPU scheduler, by using $NDR(\langle \tilde{m} \rangle, \langle \text{any} \rangle)$. This solution appears the best in our case. We took $\tilde{m} = m$.

This is a compromise between leaving full decision to the GPU scheduler (when $\tilde{m} = m$) and a tight control of the parallelism (when $\tilde{m}$ is small, for example equal to the number of compute units). However, we did not notice significant differences between values from 1024 to m.

Table 4.1 and Figure 4.2 provide a more in-depth analysis of this fact. The table is split in two parts with each column representing the results for different $\tilde{m}$ values starting from 1024 all the way to $\tilde{m} = m$. The first part shows the VGPR usage per work-item, the LDS usage per work-group, the flattened work-group size, the flattened global work size, and the number of waves per work-group, respectively for each kernel. The second part shows how resource utilization limits the number active waves; the resulting device occupancy is shown last. Our kernel is marked with a squared dot on the graphs from figure 4.2 where we can see how resources limit the number of active wavefronts. We can see that the limiting factor is the number of VGPRs used and that changing the $\tilde{m}$ signal grouping brings no change in this value. More so, Table 4.1 shows that varying $\tilde{m}$ indeed does not affect the kernel occupancy which is always at 67%.

4.3.3 Performance

In this subsection we present the performance of the parallel GPU implementation of the BOMP algorithm and compare it to an almost identical CPU version. We were able to keep an almost one-to-one instruction equivalence due to the fact that the OpenCL langauge is a custom subset of the C lan-

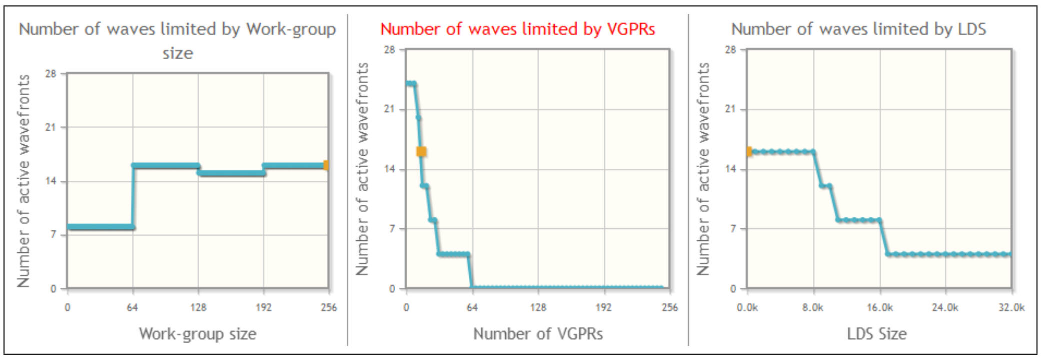

Figure 4.2: BOMP representation kernel occupancy

Table 4.1: Kernel information and occupancy for $m = 8192$

Kernel	1024	2048	4096	8192	Limits
VGPRs	15	15	15	15	248
LDS	0	0	0	0	32768
LWS	256	256	256	256	256
GWS	1024	2048	4096	8192	16777216
Waves	4	4	4	4	4
VGPRs	16	16	16	16	24
LDS	24	24	24	24	24
LWS	24	24	24	24	24
Occ.(%)	67	67	67	67	100

guage. We measured the execution times when varying the number of signals and keeping a fixed dictionary dimension and vice-versa. In both scenarios we used $\tilde{m} = m$ in the OpenCL implementation.

Figure 4.3 presents the elapsed time when representing a signals set with a varied size between $m = 1024$ to $m = 10240$ with a fixed dictionary of $n = 128$ atoms of $p = 64$ size each and a sparsity goal of $s = 8$. This experiment shows a performance improvement of up to 312 times when using the OpenCL GPU version.

In Figure 4.4 the performance improvement of the GPU implementation is clearly visible as increasing the dictionary size ($n = 64$ up to $n = 512$) has a direct impact on the number of instructions performed by each work-item. In our tests we used a fixed number of $m = 8192$ signals of dimension $p = 64$ and a target sparsity of $s = 8$. Here, the GPU version is up to 250 times faster.

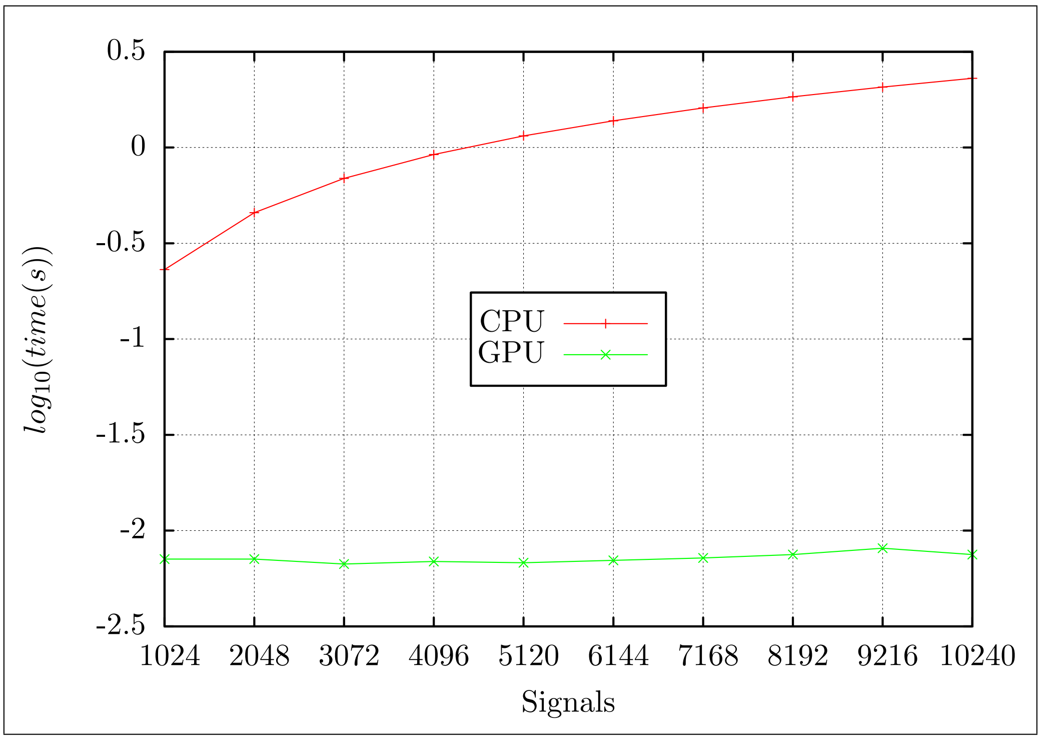

Figure 4.3: BOMP performance with varied number of signals m

4.4 Relaxation Techniques

In this section we present a few sparsifying methods that take a step back from the l_0 norm and relax the constraint by replacing it with continuous norms. While the worst-case scenario shows the relaxation techniques to have a higher succes rate at recovering the sparsest representation when compared to the greedy methods from Section 4.2, empirical evidence and average performance analysis show that the two are in fact similar. The performance guarantee comes with a complexity cost given by the sophistication required for solving these relaxed optimization problems which makes the greedy choice more attractive for certain practical applications.

4.4.1 Basis Pursuit

Chen, Donoho and Saunders [49] approach the sparse representation problem by finding a basis from the dictionary columns that provides the system solution with the smallest l_1 norm. They term this strategy Basis Pursuit (BP).

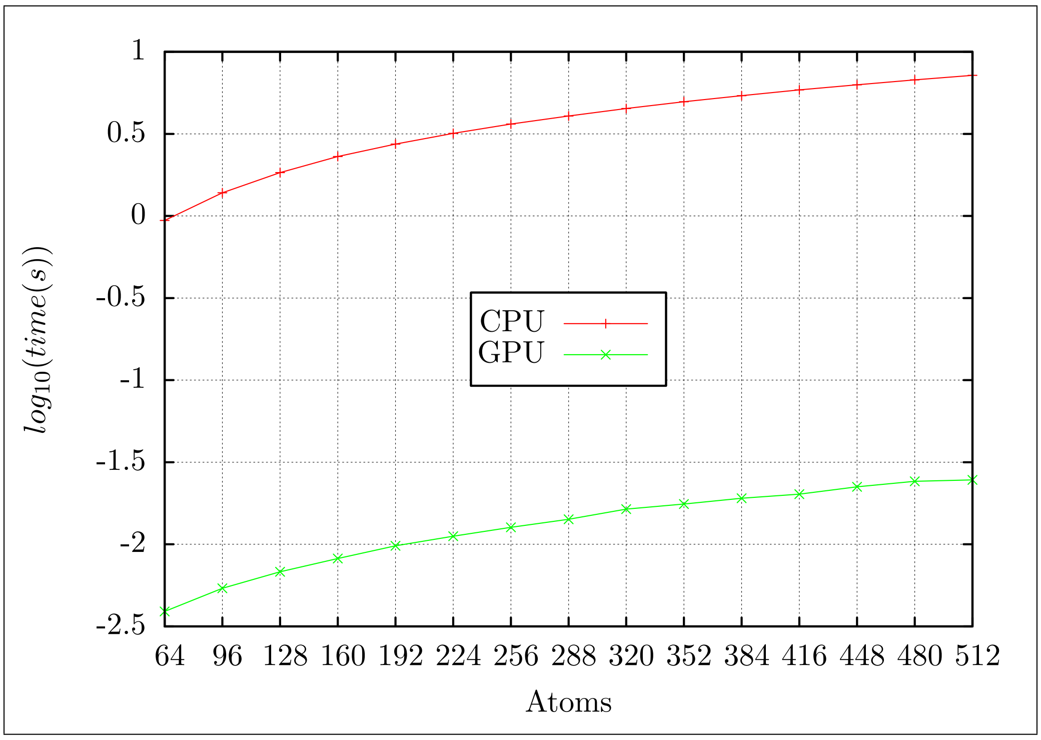

Figure 4.4: BOMP performance with varied number of atoms n

Formally this is expressed as:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \|x\|_1 \\ \text{subject to} \quad & Ax = b, \end{aligned} \tag{4.6}$$

which is shown to be equivalent to the standard form of the constrained optimization problem

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax = b, \quad x \geq 0, \end{aligned} \tag{4.7}$$

that can be solved as a linear programming (LP) problem.

A simplex approach for this LP problem would start with a basis formed from a linearly independent subset of columns from A which is then iteratively improved by performing one column swaps following the anticycling rules which guarantee convergence. The algorithm stops when no noticeable error improvement is observed.

Interior-point methods solve this problem by iteratively modifying the coefficients from x, while maintaining the constraints, and promoting sparsity.

The iterations are repeated until a subset of the x's elements stand-out and are obviously the ones corresponding to the final sparse solution.

A noise removal variant of the method exists which is termed Basis Pursuit Denoising (BPDN). If the known signal b is perturbed by a standard white Gaussian noise z, with noise level σ, and b_0 is the unknown clean signal

$$b = b_0 + \sigma z, \tag{4.8}$$

then an exact formulation such as (4.6) is replaced by an approximation

$$b = Ax + r, \tag{4.9}$$

where the residual r matches the noise σz:

$$b_0 \approx Ax, r \approx \sigma z. \tag{4.10}$$

BPDN can then be expressed as

$$\underset{x}{\text{minimize}} \quad \frac{1}{2}\|b - Ax\|_2^2 + \lambda\|x\|_1, \tag{4.11}$$

where the residual size is given by the penalty λ. This problem has been shown [50] to be equivalent to the following perturbed linear system:

$$\begin{array}{ll} \text{minimize} & c^T x + \frac{1}{2}\|p\|^2 \\ \text{subject to} & Ax + \delta p = b, \quad x \geq 0, \quad \delta = 1, \end{array} \tag{4.12}$$

which, again, can be solved by simplex or interior-point LP methods. The recommended [49] penalty choice is

$$\lambda = \sigma\sqrt{2\log(n)}. \tag{4.13}$$

4.4.2 Focal Underdetermined System Solver

Gorodnitsky and Rao present in their paper [5] an iterative reweighted least squares based algorithm termed FOCUSS. Their method relaxes the l_0 norm and replaces it with a nicer and continuous weighted l_2 approximation.

Given a general underdetrmined linear system of equations

$$Ax = b, \tag{4.14}$$

FOCUSS iteratively approximates x by using the technique of Affine Scaling Transformations (AST) [51]. Applying AST at iteration k, we express the current approximation x_k as a weighted factorization of the solutions from past iterations

$$x_k = X_{k-1}q, \tag{4.15}$$

where X_{k-1} is a diagonal matrix built from the elements of the solution given by the previous iteration. This transformation leads to an l_2 optimization problem in $q = X_{k-1}^{+}x_k$:

$$\begin{aligned} \underset{x_k}{\text{minimize}} \quad & \|X_{k-1}^{+}x_k\|_2^2 \\ \text{subject to} \quad & Ax_k = b. \end{aligned} \tag{4.16}$$

Sparsity is promoted at each iteration due to the ratio

$$\|q\|_2^2 = \|X_{k-1}^{+}x_k\|_2^2 = \sum_{i=1}^{p} \frac{x_k(i)}{x_{k-1}(i)}, \tag{4.17}$$

which encourages larger values for the entries in x_k corresponding to the columns in A that fit b best and at the same time reduces the others until they asymptotically reach 0.

The minimization in (4.16) can be solved through Lagrange multipliers as follows:

$$\begin{aligned} \mathcal{L}(x) &= \|X_{k-1}^{+}x_k\|_2^2 + \lambda^T(b - Ax) \\ \Rightarrow \frac{\partial \mathcal{L}(x)}{\partial x} &= 2(X_{k-1}^{+})^2 x_k - A^T\lambda \\ \frac{\partial \mathcal{L}(x)}{\partial x} &= 0 \Leftrightarrow x_k = \frac{1}{2}X_{k-1}^2 A^T\lambda. \end{aligned} \tag{4.18}$$

The penalty is obtained by plugging x_k back into (4.14):

$$\begin{aligned} A\left(\frac{1}{2}X_{k-1}^2 A^T\lambda\right) &= b \\ \Rightarrow \lambda &= 2(AX_{k-1}^2A^T)^{+}b. \end{aligned} \tag{4.19}$$

Finally, x_k can be formulated as:

$$x_k = X_{k-1}^2 A^T(AX_{k-1}^2A^T)^{+}b. \tag{4.20}$$

Algorithm 8 puts these results together and presents the final form of the FOCUSS technique. Given a dictionary A with n atoms, the signal b and an approximation threshold ϵ (step 1), the algorithm initializes the weights with the unit matrix in step 2 and then proceeds with the iterations in step 3. Because the method iteratively zeros the non-essential elements of x, we note that it is important to start with all the entries set to non-zero values

Algorithm 8: FOCUSS

```
1 Arguments: $A \in \mathbb{R}^{p \times n}$, $b \in \mathbb{R}^n$, approximation error $\epsilon$
2 Initilize: $X_0 = I_n$
3 for $k = 1 : \infty$ do
4     Compute new solution: $x_k = X_{k-1}^2 A^T (A X_{k-1}^2 A^T)^+ b$
5     Update: $X_k = diag(x_k)$
6     Stop if: $\|x_k - x_{k-1}\|_2 < \epsilon$
```

so that the zeroing is left entirely to the algorithm. That is why we choose the unit matrix as the initial value of X_0 (in which case the solution is the same as with LS), but in fact any full-rank diagonal matrix would have been good. A FOCUSS iteration starts with solving Equation (4.20) from step 4, updating the weights with the new solution x_k (step 5) and checking to see if the improvement is sufficient to justify a new iteration (step 6).

The stopping criterion is sufficient because the solutions sequence $\{x_k\}_{k=0}^{\infty}$ converges [5] to a fixed-point no matter of the initialization choice x_0. More so, the convergence follows the descent function

$$L(x) = \Pi_{i=1}^{p} |x_k(i)|. \tag{4.21}$$

We note that even though convergence is guaranteed, the fixed point that will be reached is not guaranteed to be the best in approximating the global minimum.

4.5 Conclusions

In this chapter we presented multiple l_0 representation methods and proposed a new one, POLS, as a viable candidate for substituting the ubiquitous OMP. To that end we also provided an in-depth case study of the parallel implementation of the OMP algorithm and its efficiency when executing on a multiprocessor GPU. Lastly, we provided an overview of the existing alternatives to the fast greedy approach which relax the l_0 norm in favour of the convex and continuous l_1 or even of the l_2 norm. While these offer tempting worst-case scenario performance guarantees, average performance analysis and experiments have shown that the average result of the two approaches are about the same. In Chapter 6 we will see the effects of combining several sparse representation and atom update methods for solving the problem of overcomplete dictionary design and we will show the beneficial effects of mixing POLS with OMP in the tightly coupled process of learning a dictionary and then using it for representing new signals.

Chapter 5

Dictionary Design

Given a set of signals $Y \in \mathbb{R}^{p \times m}$ and a sparsity level s, the goal is to find a dictionary $D \in \mathbb{R}^{p \times n}$ that minimizes the Frobenius norm of the approximation error

$$E = Y - DX, \tag{5.1}$$

where $X \in \mathbb{R}^{n \times m}$ is the associated s-sparse representations matrix, with at most s nonzero elements on each column. This is the dictionary learning problem from Equation (2.8) with known sparse representations.

5.1 Atom update methods

The general structure of most DL algorithms was presented and described around Algorithm 1 from Chapter 2. The iterative process is composed of the two main stages depicted by steps 4 and 5 in the algorithm. In this section we will present several techniques for updating atom d_j in step 5. We denote $\mathcal{I}_j$ the set of the indices of nonzero elements on the j-th row of X. Otherwise said, these are the indices of the signals whose representation involves the atom d_j.

K-SVD [29] solves the optimization problem

$$\min_{d_j, X_{j,\mathcal{I}_j}} \left\| \left(Y_{\mathcal{J}} - \sum_{\ell \neq j} d_\ell X_{\ell,\mathcal{I}_\ell} \right) - d_j X_{j,\mathcal{I}_j} \right\|_F^2, \tag{5.2}$$

where all atoms excepting d_j are fixed. Note that the matrix within parenthesis is the representation error of the dictionary without the atom d_j. To minimize the error, the problem (5.2) is seen as a rank-1 approximation of this modified error matrix. The solution is given by the singular vectors corresponding to the largest singular value. The less complex AK-SVD [30] runs a single iteration of the power method to compute the two vectors. Note that (5.2) allows the representations to be changed also in this stage.

SGK [31] considers the same optimization problem, but only with d_j as variable:

$$\min_{d_j} \left\| \left(Y_{\mathcal{J}} - \sum_{\ell \neq j} d_\ell X_{\ell,\mathcal{I}_\ell} \right) - d_j X_{j,\mathcal{I}_j} \right\|_F^2 . \tag{5.3}$$

This is a simple least-squares problem, for which an explicit solution can be easily computed. NSGK [32] operates similarly, but using differences with the previous values of the dictionary and representations.

5.2 Jacobi Atom Updates Stategy

All the algorithms from Section 5.1 update the atoms sequentially, using thus their most recent values when updating another atom. This is the typical Gauss-Seidel approach. We investigate here the Jacobi version, in which the atoms are updated independently, meaning that problems like (5.2) or (5.3) are solved simultaneously for $j = 1 : n$. Jacobi atom updates (JAU) can be applied to all presented dictionary update algorithms. We append the initial P (from parallel) to their name to denote the Jacobi versions.

The general form of the proposed dictionary learning method with Jacobi atom updates is presented in Algorithm 9. At iteration k of the DL method, the two usual stages are performed. In step 1, the current dictionary $D^{(k)}$ and the signals Y are used to find the sparse representation matrix $X^{(k)}$ with s nonzero elements on each column; we used OMP, as widely done in the literature.

The atom update stage takes place in groups of $\tilde{n}$ atoms. We assume that $\tilde{n}$ divides n only for the simplicity of description, but this is not a mandatory condition. Steps 2 and 3 of Algorithm 9 perform a full sweep of the atoms. All the $\tilde{n}$ atoms from the same group are updated independently (step 4), using one of the various available rules; some of them will be discussed in the next section. Once a group is processed, its updated atoms are used for updating the other atoms; so, atom $d_j^{(k+1)}$ (column j of $D^{(k+1)}$) is computed in step 4 using $d_i^{(k+1)}$ if

$$\lfloor (i-1)/\tilde{n} \rfloor < \lfloor (j-1)/\tilde{n} \rfloor, \tag{5.4}$$

i.e. $i < j$ and d_i not in the same group as d_j, and $d_i^{(k)}$ otherwise.

Putting $\tilde{n} = 1$ gives the usual sequential Gauss-Seidel form. Taking $\tilde{n} = n$ leads to a fully parallel update, i.e. the form that is typically labeled with Jacobi's name.

Algorithm 9: General structure of a DL-JAU iteration

```
Data:   current dictionary $D^{(k)} \in \mathbb{R}^{p \times n}$
        signals set $Y \in \mathbb{R}^{p \times m}$
        number of parallel atoms $\tilde{n}$
Result: next dictionary $D^{(k+1)}$
1 Compute $s$-sparse representations $X^{(k)} \in \mathbb{R}^{n \times m}$ such that
  $Y \approx D^{(k)} X^{(k)}$
2 for $\ell = 1$ to $n/\tilde{n}$ do
3     for $j = (\ell - 1)\tilde{n} + 1$ to $\ell\tilde{n}$, in parallel do
4         Compute $d_j^{(k+1)}$
5         Update $d_j^{(k+1)} \leftarrow \alpha d_j^{(k+1)} + (1 - \alpha) d_j^{(k)}$
6         Normalize: $d_j^{(k+1)} \leftarrow d_j^{(k+1)} / \|d_j^{(k+1)}\|$
```

We also propose (step 5) to update an atom via a convex combination of its new and former value, with a weight $\alpha \in (0, 1]$. The choice $\alpha = 1$ was always used until now, which is meaningful for the sequential approach, where we seek the optimal value of an atom, given all the others. In a parallel context, where a group of atoms are optimized simultaneously, it may be wise to temper their progress, since their independent evolution may overreach the target. Finally, step 6 is the usual normalization constraint on the dictionary.

The proposed form has obvious potential for a smaller execution time on a parallel architecture. We touch this issue in Section 5.3 where we present a case-study of a GPU implementation of the AK-SVD algorithm (also published in [52]). Here, we will focus solely on the quality of the designed dictionary.

5.2.1 Particular Forms of the Algorithm

Typically, the atom update problem is posed as follows. We have the dictionary, denoted generically D, and the associated representations matrix X and we want to optimize atom d_j. In the DL context, at iteration k of the learning process, the dictionary is made of atoms from $D^{(k)}$ and $D^{(k+1)}$, as explained by the phrase around Equation (5.4). We denote $\mathcal{I}_j$ the (column) indices of the signals that use d_j in their representation, i.e. the indices of the nonzero elements on the j-th row of X. Excluding atom d_j, the representation error matrix (5.1), reduced to the relevant columns, becomes

$$F = E_{\mathcal{I}_j} + d_j x_{j,\mathcal{I}_j}. \qquad (5.5)$$

The updated atom d_j is the solution of the optimization problem

$$\min_{t \in \mathbb{R}^p} \quad \|F - t\, x_{j,\mathcal{I}_j}\|_F^2, \tag{5.6}$$

which is a rewrite of Equation (5.3) in terms of matrix F. The norm constraint $\|t\|_2 = 1$ is usually imposed after solving the optimization problem.

AK-SVD. The K-SVD algorithm and its approximate version AK-SVD [30] treat (5.6) by considering that $x_{j,\mathcal{I}_j}$ is also a variable (as described around Equation (5.2)). Problem (5.6) becomes a rank-1 approximation problem that is solved by AK-SVD with a single iteration of the power method (to avoid ambiguity, we add superscripts representing the iteration number):

$$\begin{aligned} d_j^{(k+1)} &= F(x_{j,\mathcal{I}_j}^{(k)})^T / \|F(x_{j,\mathcal{I}_j}^{(k)})^T\|_2 \\ x_{j,\mathcal{I}_j}^{(k+1)} &= F^T d_j^{(k+1)}. \end{aligned} \tag{5.7}$$

Note that the representations are also changed in the atom update stage, which is the specific of this approach.

SGK. Dictionary learning for sparse representations as a generalization of K-Means clustering (SGK) [31] solves directly problem (5.6). This is a least squares problem whose solution is

$$d_j^{(k+1)} = F x_{j,\mathcal{I}_j}^T / (x_{j,\mathcal{I}_j} x_{j,\mathcal{I}_j}^T). \tag{5.8}$$

The atom updates part of the general JAU scheme from Algorithm 9 has the form described by Algorithm 10, named P-SGK (with P from Parallel). The error E is recomputed in step 2 before each group of atom updates, thus taking into account the updated values of the previous groups. Depending on the value of $\tilde{n}$, the error can be computed more efficiently via updates to its previous value instead of a full recomputation. Steps 4 and 5 implement relations (5.5) and (5.8), respectively. Steps 6 and 7, the weighted combination and the normalization, are identical with those from the general scheme.

To obtain the JAU version of AK-SVD (named PAK-SVD), we replace step 5 by the operations from (5.7). Note that, for full parallelism ($\tilde{n} = n$) and no weighting ($\alpha = 1$), P-SGK and PAK-SVD are identical, since the atoms produced by (5.7) and (5.8) have the same direction. For full parallelism the representations updated by PAK-SVD are not used, while if $\tilde{n} < n$, some updated representations affect the error matrix from step 2. If $\alpha < 1$, the weighted combination gives different results, because the updated atom is also normalized before the combination in PAK-SVD, while in P-SGK only after the combination.

Algorithm 10: P-SGK Atom Updates

```
Data:   current dictionary $D \in \mathbb{R}^{p\times n}$
        signals set $Y \in \mathbb{R}^{p\times m}$
        sparse representations $X \in \mathbb{R}^{n\times m}$
        number of parallel atoms $\tilde{n}$
Result: next dictionary $D$
1 for $\ell = 1$ to $n/\tilde{n}$ do
2     $E = Y - DX$
3     for $j = (\ell - 1)\tilde{n} + 1$ to $\ell\tilde{n}$, in parallel do
4         $F = E_{\mathcal{I}_j} + d_j x_{j,\mathcal{I}_j}$
5         $t = F x_{j,\mathcal{I}_j}^T / (x_{j,\mathcal{I}_j} x_{j,\mathcal{I}_j}^T)$
6         $d_j \leftarrow \alpha t + (1-\alpha) d_j$
7         $d_j \leftarrow d_j / \|d_j\|_2$
```

NSGK. The update problem (5.6) is treated in [32] in terms of differences with respect to the current dictionary and representations, instead of working directly with D and X:

$$\begin{cases} D = D^{(k-1)} + (D^{(k)} - D^{(k-1)}) \\ X = X^{(k-1)} + (X^{(k)} - X^{(k-1)}), \end{cases} \tag{5.9}$$

where $X^{(k-1)}$ is the sparse representation matrix at the beginning of the k-iteration of the DL algorithm, while $X^{(k)}$ is the matrix computed in the k-th iteration (e.g. in step 1 of Algorithm 9).

Applying this idea to SGK, the optimization problem is similar, but with the signal matrix Y replaced by

$$Z = Y + D^{(k)}X^{(k-1)} - D^{(k)}X^{(k)}. \tag{5.10}$$

The P-NSGK algorithm (NSGK stands for New SGK, the name used in [32]) is thus identical with P-SGK, with step 2 modified according to (5.10). Also, in (5.8), the representations $x_{j,\mathcal{I}_j}$ are taken from $X^{(k-1)}$, not from $X^{(k)}$ as for the other methods.

5.2.2 Numerical Results

We give here numerical evidence supporting the advantages of the JAU scheme, for dictionary recovery and sparse image representation. We compare the JAU algorithms PAK-SVD, P-SGK and P-NSGK with their sequential counterparts. A notation like P-SGK(α) shows the weight α of the convex combination from step 5 of Algorithm 9. We report results obtained with the same input data for all the algorithms; in particular, the initial dictionary is the same. The sparse representations were computed via OMP[1].

Dictionary Recovery

We generate the expermintal data as described in Section 2.5.1 and use full parallelism for the JAU methods ($\tilde{n} = n$). The percentages of recovered atoms, averaged over 50 runs, are presented in Table 5.1. (PAK-SVD is not reported, since it gives the same results as P-SGK.)

We note that, although the JAU algorithms give the best result in 9 out of the 12 considered problems, the results are rather similar for all algorithms. (We may infer that, in this problem, the sparse representation stage is the bottleneck, not the atom update stage.) We can at least conclude that, for dictionary recovery, the JAU scheme is not worse than the sequential ones. A weight $0.9 \leq \alpha < 1$ appears to be slightly beneficial.

Dictionary Learning

We generated the training data and the dictionary as described in Section 2.5.2. In a first experiment, we used $m = 32768$ signals of dimension $p = 64$ to train dictionaries with $n = 512$ atoms, with a target sparsity of $s = 8$. In Figure 5.1 we can see the evolution of the representation RMSE, averaged over 10 runs, for the JAU and sequential algorithms. JAU algorithms have full parallelism ($\tilde{n} = n$) and weight $\alpha = 1$. (Note that the PAK-SVD and P-SGK curves are slightly different, due to the computation of the errors at the end of a DL iteration, where PAK-SVD has different representations; otherwise, the dictionaries are identical.) Although the sequential algorithms have smoother convergence, the proposed parallel versions obtain clearly better results. Among the sequential algorithms, NSGK is the best, confirming the findings from [32]. However, all parallel algorithms are better than NSGK.

The same conclusion is supported by a second experiment, where the conditions are similar but, for faster execution, only $m = 16384$ training signals were used. Table 5.2 shows the lowest RMSE after $i = 200$ iterations, averaged over 10 runs, for three values of the dictionary size n. Weights $\alpha < 1$ were also considered. In all cases, the JAU algorithms are clearly the best. For P-NSGK, the best weight value appears to be between 0.9 and 0.95, as these values are better than no weighting ($\alpha = 1$). However, P-SGK

[1]We used OMP-Box version 10 available at `http://www.cs.technion.ac.il/~ronrubin/software.html`

Table 5.1: Percentage of recovered atoms

s	Method	SNR			
		10	20	30	∞
3	NSGK	87.16	**90.16**	89.32	89.56
	P-NSGK(0.8)	84.56	89.84	88.08	89.36
	P-NSGK(0.9)	85.00	90.12	89.04	**90.56**
	P-NSGK(0.95)	88.04	90.00	89.12	89.84
	P-NSGK	**88.36**	89.64	89.92	89.76
	SGK	87.44	89.40	88.80	90.12
	P-SGK(0.8)	85.92	89.68	89.68	89.44
	P-SGK(0.9)	86.04	90.12	89.96	89.36
	P-SGK(0.95)	87.72	89.24	89.28	89.52
	P-SGK	86.48	89.84	89.00	88.24
	AK-SVD	86.20	90.00	**90.00**	88.52
4	NSGK	**70.68**	91.88	92.16	93.16
	P-NSGK(0.8)	62.40	92.28	91.72	92.52
	P-NSGK(0.9)	63.92	92.16	92.08	93.16
	P-NSGK(0.95)	65.48	92.88	91.60	93.20
	P-NSGK	68.08	92.48	**92.88**	**93.28**
	SGK	67.28	92.16	91.68	91.92
	P-SGK(0.8)	67.44	91.56	91.24	92.36
	P-SGK(0.9)	65.24	92.68	91.60	92.40
	P-SGK(0.95)	65.00	93.04	92.04	91.88
	P-SGK	68.28	**93.48**	91.88	92.56
	AK-SVD	70.08	92.76	92.16	92.32
5	NSGK	10.24	92.36	92.72	94.40
	P-NSGK(0.8)	4.88	92.28	91.48	93.48
	P-NSGK(0.9)	7.80	93.08	92.40	93.36
	P-NSGK(0.95)	7.68	**93.56**	93.56	93.56
	P-NSGK	10.60	93.08	93.32	94.72
	SGK	11.68	93.28	92.60	93.92
	P-SGK(0.8)	6.08	92.88	92.44	93.60
	P-SGK(0.9)	8.08	92.56	**93.68**	93.56
	P-SGK(0.95)	6.88	93.20	92.40	**95.00**
	P-SGK	**12.04**	93.00	92.92	93.92
	AK-SVD	11.64	92.72	92.88	94.96

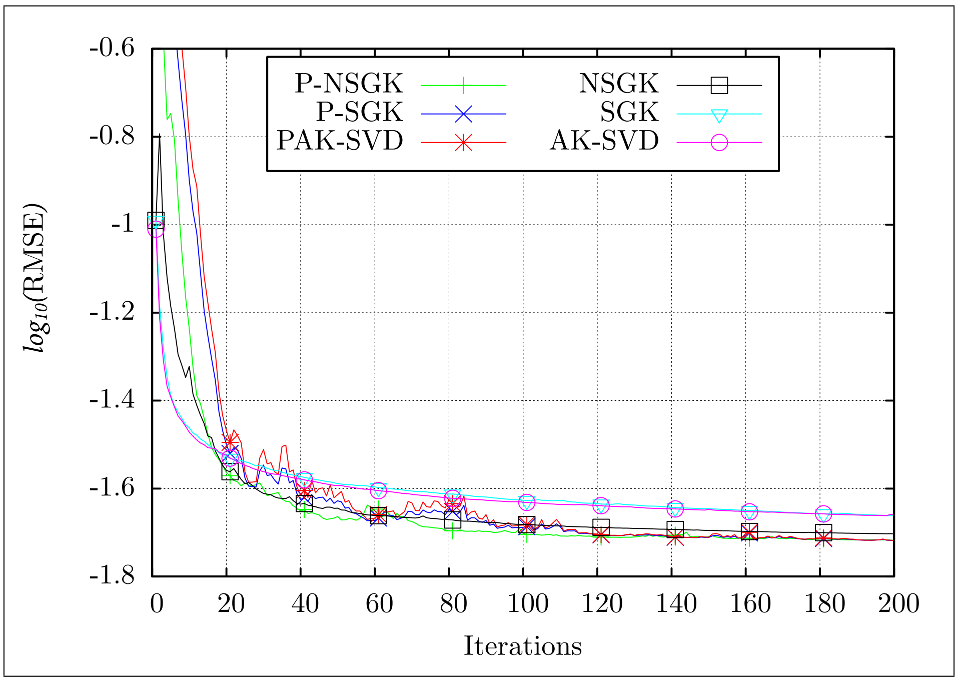

Figure 5.1: Error evolution for parallel and sequential algorithms.

seems to have less benefits from weighting and taking $\alpha = 1$ appears the best option.

To show the influence of the two parameters of the JAU algorithms, the weight α and the group size $\tilde{n}$, we present the evolution of the error, averaged over 10 runs, in figures 5.2 and 5.3. We note that the curves for $\alpha = 0.9$ and $\alpha = 1$ almost coincide after a sufficiently large number of iterations. Expectedly, a smaller α offers lower performance.

The effect of group size is less intuitive. Full parallelism ($\tilde{n} = n$) is the winner, although some smaller values of $\tilde{n}$ are good competitors, almost all being better than the sequential version ($\tilde{n} = 1$). Even though in this example $\tilde{n} = 256$ is worse than some smaller values, the error usually decreases as $\tilde{n}$ grows, the best value being $\tilde{n} = n$ in all our tests, for all parallel methods. A possible explanation is that the JAU strategy, due to the independent atom updates, is less prone to get trapped in local minima. Modifying atoms one by one, although locally optimal, may imply only small modifications of the atoms; in contrast, JAU appears to be able of larger updates that make convergence more erratic, but can reach a better dictionary.

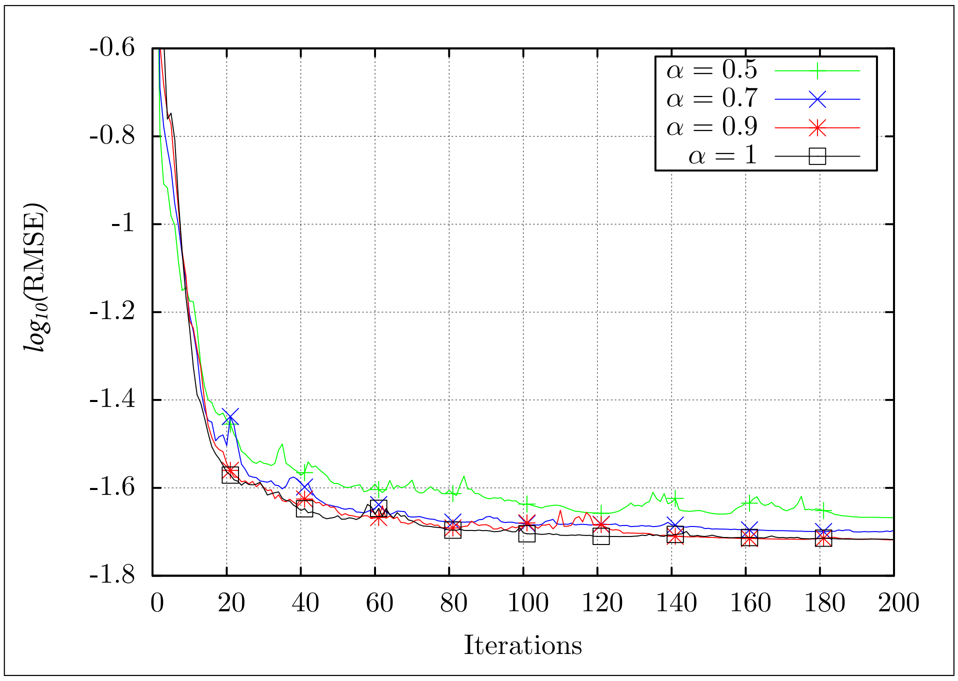

Figure 5.2: P-NSGK error evolution for weigthed atom updates.

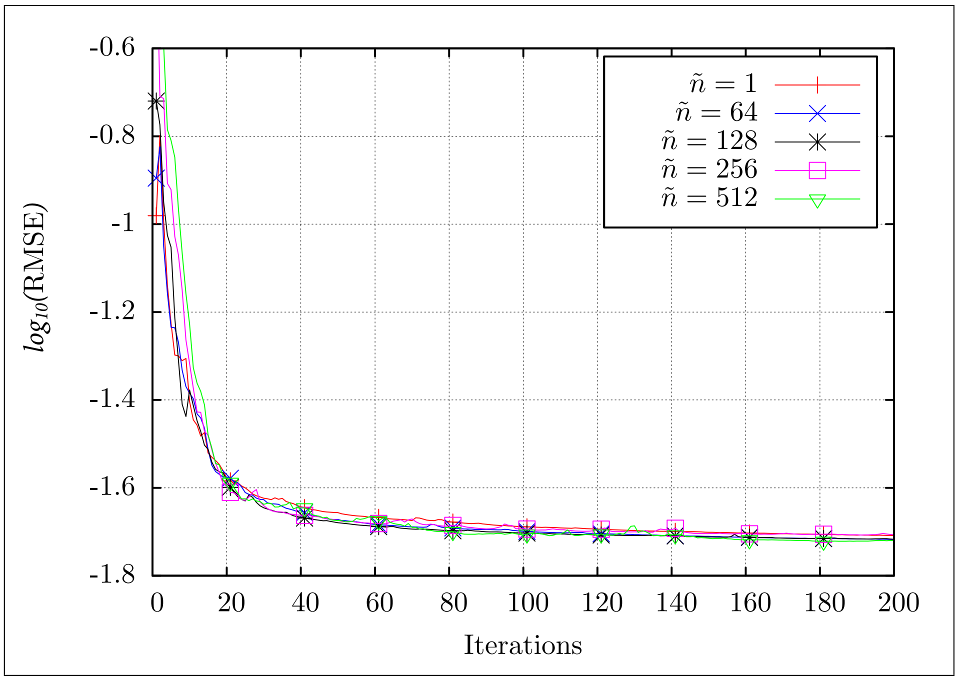

Figure 5.3: P-NSGK error evolution for various group sizes.

Table 5.2: Best RMSE values after 200 iterations

	$n = 128$	$n = 256$	$n = 512$
NSGK	0.0185	0.0168	0.0154
P-NSGK(0.8)	0.0167	0.0154	0.0140
P-NSGK(0.9)	0.0166	0.0153	0.0138
P-NSGK(0.95)	0.0166	0.0153	0.0138
P-NSGK	0.0167	0.0154	0.0139
SGK	0.0201	0.0185	0.0166
P-SGK(0.8)	0.0168	0.0156	0.0145
P-SGK(0.9)	0.0166	0.0153	0.0142
P-SGK(0.95)	0.0166	0.0153	0.0139
P-SGK	0.0165	0.0153	0.0138
AK-SVD	0.0201	0.0184	0.0163

JAU versus MOD

We now compare the performance in representation error of the JAU algorithms with the intrinsically parallel algorithm named method of optimal directions (MOD) [28]. MOD uses OMP for representation and updates the dictionary D with the least-squares solution of the linear system $DX = Y$. For completeness we also include the sequential versions on which JAU algorithms are built.

In figures 5.4–5.7 we depict the JAU algorithms with green, the sequential versions with red and MOD with black. All algorithms performed DL for $k = 200$ iterations. Each data point from these figures represents an average of 10 runs of the same algorithm with the same parametrization and data dimensions but with training sets composed of different image patches.

To see how sparsity influences the end result, Figure 5.4 presents the final errors for sparsity levels starting from $s = 4$ up to $s = 12$ when performing DL for dictionaries of $n = 128$ atoms on training sets of size $m = 8192$. We notice that for all three algorithms (NSGK, SGK and AK-SVD) the JAU methods perform similar to MOD at lower sparsity constraints, but as we pass $s = 8$ our proposed parallel strategy is clearly better. The sequential versions always come in last, except perhaps for NSGK that comes close to MOD past $s = 10$.

Figure 5.5 presents the final errors for DL on training sets of $m = 12288$ signals, with a sparsity constraint of $s = 12$, when varying the total number of atoms in the dictionary from $n = 128$ to $n = 512$ in increments of 64. Again, the JAU versions are the winners for all three algorithms. Out of the

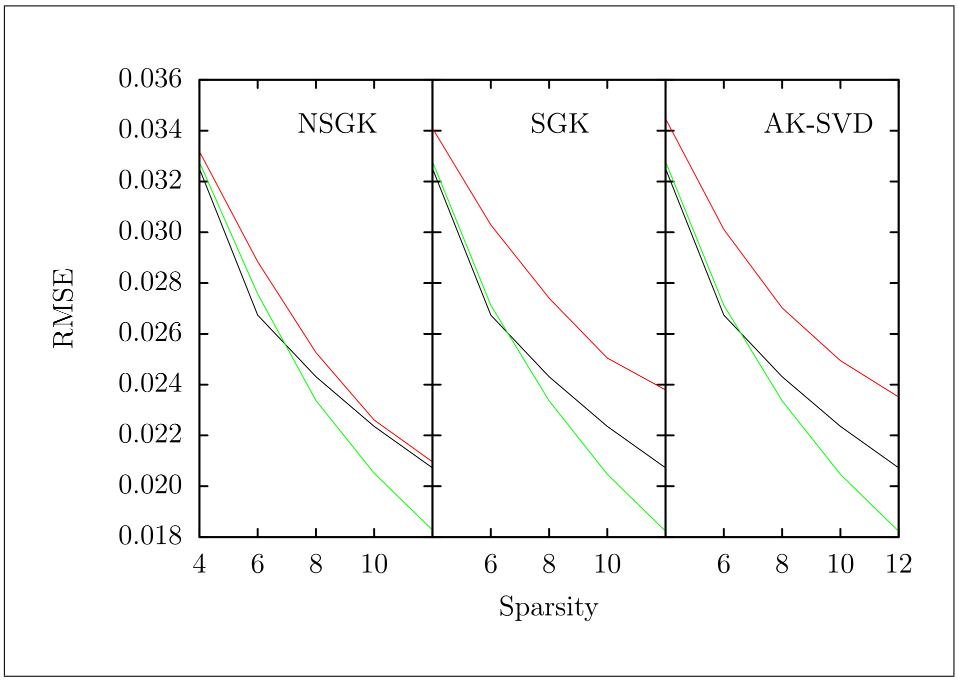

Figure 5.4: Final errors for different sparsity constraints. The sequential versions are red, JAU algorithms are green and MOD is black.

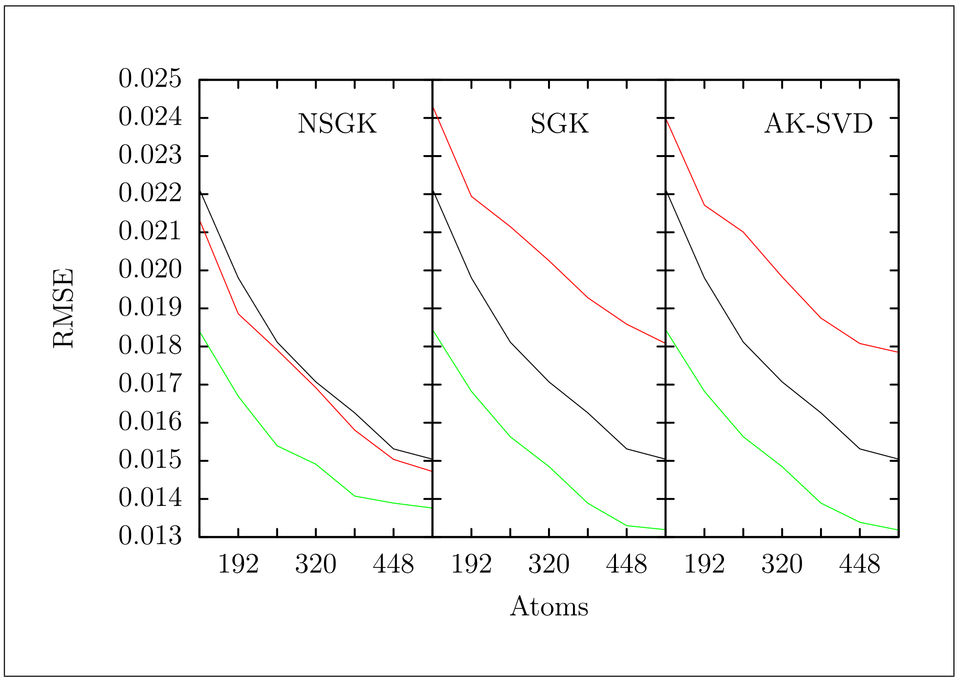

Figure 5.5: Final errors for varied dictionary sizes. The sequential versions are red, JAU algorithms are green and MOD is black.

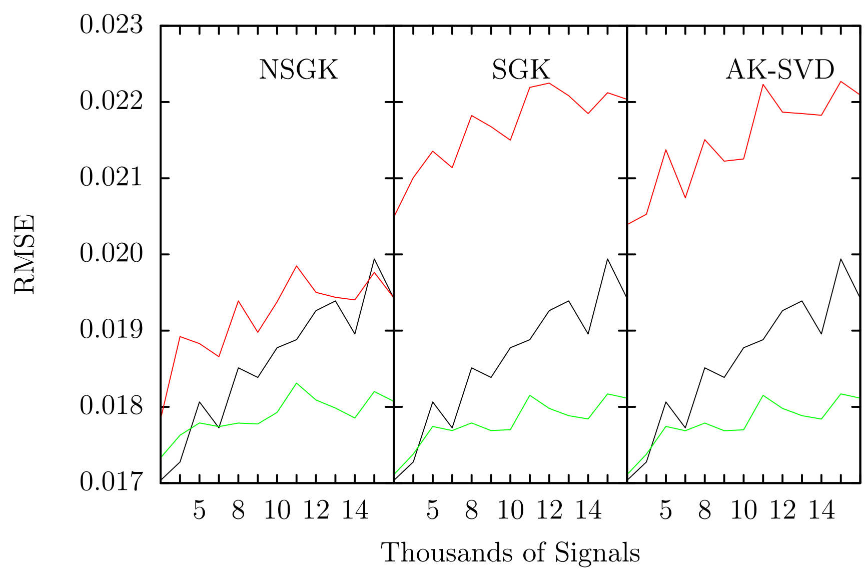

Figure 5.6: Final errors for varied training set sizes. The sequential versions are red, JAU algorithms green and MOD is black.

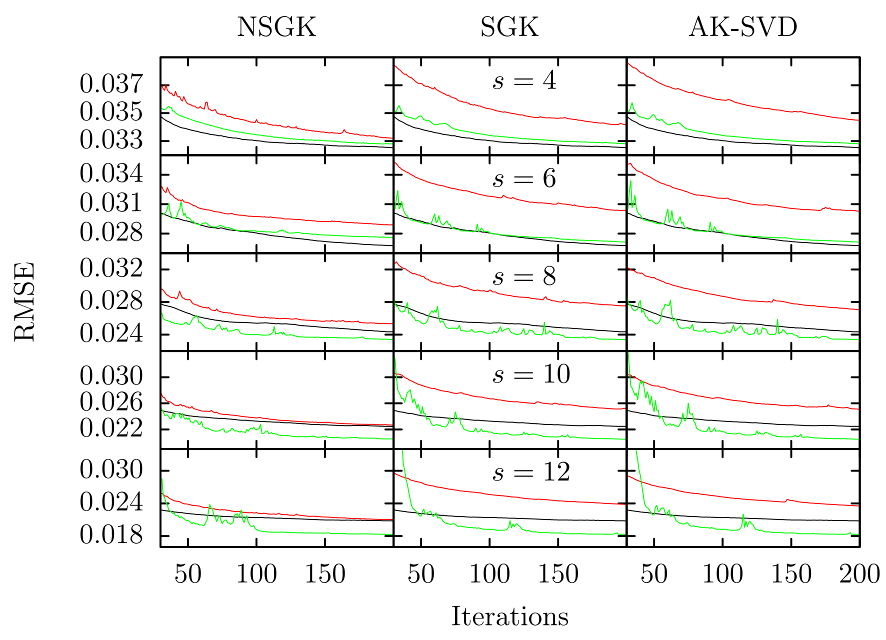

Figure 5.7: Error evolution at different sparsity constraints. The sequential versions are red, JAU algorithms green and MOD is black.

sequential algorithms, NSGK is the only one that manages to outperform MOD, while the others lag behind.

The next experiment investigates the influence of the signal set size on the final errors. In Figure 5.6 we kept a fixed dictionary size of $n = 256$ and a sparsity of $s = 10$ and performed DL starting with training sets of $m = 4096$ signals that we increased in increments of 1024 up to $m = 16384$. JAU stays ahead of MOD almost everywhere, except for small signal sets with $m < 5000$ where the results are similar. The sequential versions are once again the poorest performers.

Finally, we present in Figure 5.7 the error improvement at each iteration for all algorithms, for several sparsity levels. In this experiment we used a dictionary of $n = 128$ atoms and a training set of $m = 8192$ signals. We can see that the JAU versions can jump back and forwards, specially during the first iterations. We think that this is due to the parallel update of the dictionary atoms which leads to jumps from one local minima to another until a stable point is reached. This is, perhaps, the reason why in the end it manages to provide a lower representation error. Even though the JAU convergence is not as smooth as MOD or the sequential versions, it has a consistent descendent trend.

5.3 Parallel Dictionary Learning with OpenCL

In this section we focus on the study of efficient OpenCL implementations for the dictionary update stages of the AK-SVD, SGK and NSGK methods. This, together with the parallel results for the sparse representation stage described in Chapter 4, allows us to perform the entire dictionary learning process in parallel on the GPU (see Figure 5.8).

The PAK-SVD and the P-SGK kernels are the same until the point where the atom is updated. At that time P-SGK is done with the update stage while PAK-SVD has to perform the extra instructions needed for updating the affected sparse representations as per Equation (5.7). P-NSGK uses the same atom update kernel as P-SGK with the input error matrix calculated as $E = Z - DX$ instead of $E = Y - DX$ as explained around Equation (5.10). The operations described in the following paragraphs are assumed to be common to all three methods unless explicitly stated otherwise.

5.3.1 OpenCL Implementation Details

Before refining the dictionary we need to compute the error matrix described in Equation (5.1) and portrayed in step 2 of Algorithm 10. The matrix

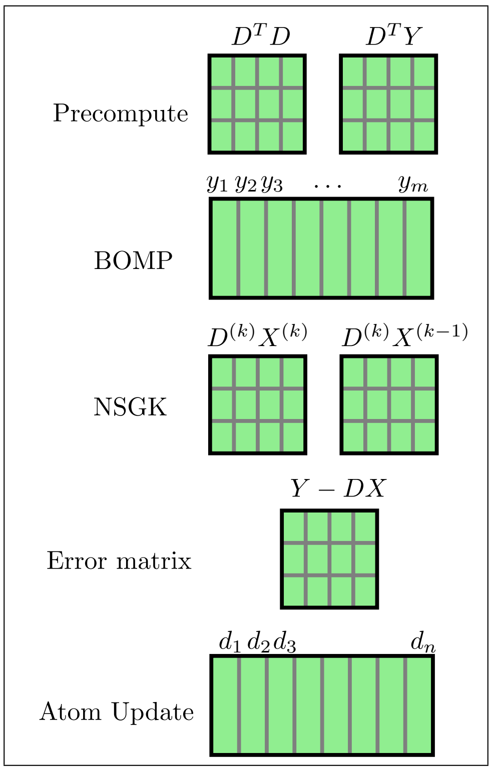

Figure 5.8: Parallel DL: each row is executed simultaneously and each grid represents an OpenCL kernel whose elements are independent execution threads.

multiplication needed for the current error is done in the same manner as the matrix precomputations for BOMP that were described in Section 4.3. We note that for NSGK two extra BLAS operations are needed in order to compute the difference based matrix Z as described around equation (5.10). Matrix Z will replace Y when computing the error matrix E. That is why in Figure 5.8 the NSGK row precedes the error calculation from the fourth row.

The atom updating process was implemented through a dedicated OpenCL kernel. The input matrices E and X are kept in global memory as well as the existing dictionary D. However, the atom D_j is transferred to private memory, where it is kept during the update operations. This poses no problem due to the reasonable problem size (we used $p \leq 64$).

On the other hand, storing a list of indices $\mathcal{I}$ (step 11) turns out to be difficult, since the number of indices varies a lot, depending on how much an atom is used in sparse signal representations. The GPU we have used

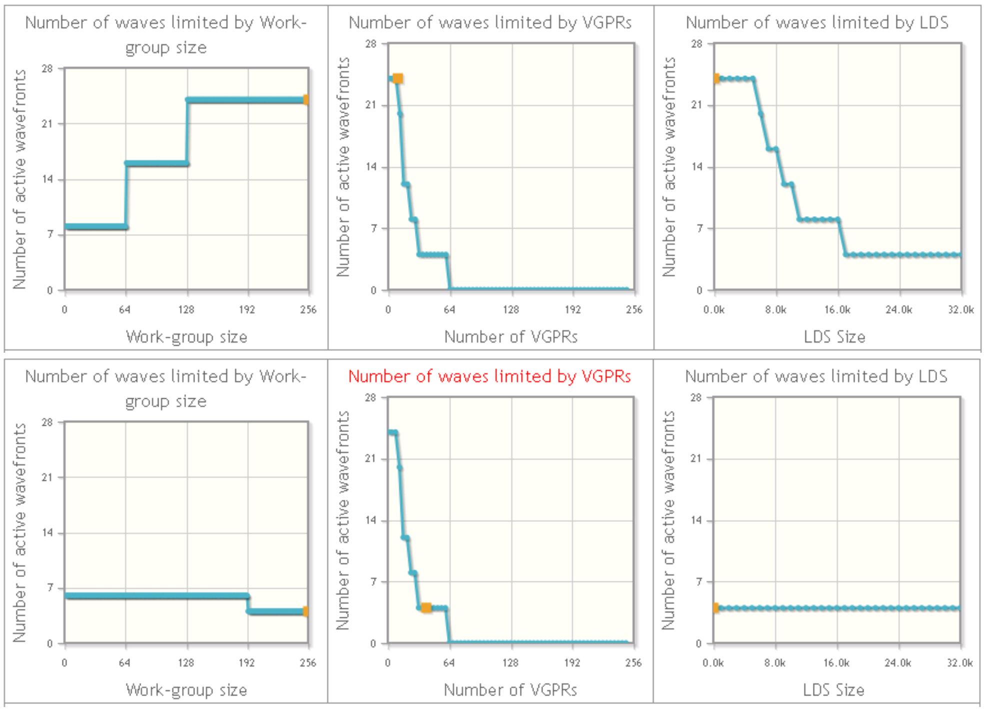

Figure 5.9: PAK-SVD dictionary update GPU occupancy. The top panel shows the benefits of keeping $\mathcal{I}$ in local memory while the bottom panel depicts what happens when we move it in global memory.

has 32k local memory, which allows storage of about 8000 indices. If the number of signals is smaller, we use local memory, which is the ideal solution. Otherwise, the set $\mathcal{I}$ is stored as a global variable and hence global memory access latency diminishes the performance of the update stage.

In both cases we define a 1-dimensional space of $\tilde{n}$ global PEs. We leave defining the work-group size to the GPU scheduler by using $NDR(\langle\tilde{n}\rangle, \langle\text{any}\rangle)$.

The BLAS operations required for performing the power method were all done inside the update kernel in a sequential fashion for the same reasons enumerated when describing BOMP.

BLAS issues. For further optimization we tried using the BLAS library for OpenCL from AMD. While probably good for one-time use scenarios, it did not give good performance in our case, which needed multiple calls for mostly quickly changing, small sized, data. The loss in transfer times between host and OpenCL memory was not compensated at all by the parallel computations. We were hence obliged to implement our own versions of BLAS operations, described in Section 4.3, for both big and small data sets.

Occupancy. In Table 5.3 we analyze the atom update kernel performance

Table 5.3: Kernel information and occupancy for P-SGK and PAK-SVD

Kernel	P-SGK		PAK-SVD		
$\mathcal{I}$	local	global	local	global	Limits
VGPRs	6	38	10	40	248
LDS	0	0	0	0	32768
LWS	256	256	256	256	256
GWS	512	512	512	512	16777216
Waves	4	4	4	4	4
VGPRs	24	4	24	4	24
LDS	24	24	24	24	24
LWS	24	24	24	24	24
Occ.(%)	100	16	100	16	100

in terms of GPU occupancy. Our experiments showed that varying the number of atoms in the dictionary from $n = 64$ up to $n = 512$ had no effect on occupancy. That is why we focus here on the main obstacle: the memory storage location of the indices set $\mathcal{I}$. Table 5.3 shows that moving the indices in local memory has a significant impact bumping occupancy from 16% to 100% due to lowering the number of used VGPRs by 32 and 30 in the P-SGK and, respectively, PAK-SVD case. The difference can also be spotted in Figure 5.9 by comparing the VGPR screens from the top and bottom panels. The rest of the occupancy factors are not affected by $\mathcal{I}$.

5.3.2 Results and Performance

We generated experimental data using the same method as described in Section 2.5.2 and executed our tests as specified in Section 3.4.

We picked $p = 64$ and $n \in \{64, 128, 256, 512\}$, while we ran through a wide range of signal set dimensions (1024–131072). While we did most of the profiling around $s = 8$, we consistently investigated $s \in \{4, 6, 8, 10, 12\}$ when it came to minimizing the error. For parallelism, we used almost all the time $\tilde{m} = m$ while we walked $\tilde{n}$ from 1, in increments of powers of 2, up to n.

PAK-SVD Case-Study

We first take PAK-SVD as a case-study for the improvements in execution time, and start by looking at the influence of $\tilde{n}$. As expected, we can see in Figure 5.10, where $m = 16384$, that larger $\tilde{n}$ gives better results, producing a speed-up of at least 10 when going from $\tilde{n} = 1$ to $\tilde{n} = n$.

Algorithm 11: PAK-SVD

```
Data:   initial dictionary $D \in \mathbb{R}^{p \times n}$
        signals set $Y \in \mathbb{R}^{p \times m}$
        number of parallel atoms $\tilde{n}$
        number of update iterations $u$
Result: trained dictionary $D$
        sparse representations $X \in \mathbb{R}^{n \times m}$
1 for $i \leftarrow 1$ to $u$ do
2     for $\ell \leftarrow 1$ to $n/\tilde{n}$ do
3         $E = Y - DX$
4         for $j \leftarrow (\ell - 1)\tilde{n} + 1$ to $\ell\tilde{n}$, in parallel do
5             $F = E_{\mathcal{I}} + D_j X_{j,\mathcal{I}}$
6             $D_j = F X_{j,\mathcal{I}}^T / \|F X_{j,\mathcal{I}}^T\|_2$
7             $X_{j,\mathcal{I}} = F^T D_j$
```

A similar behavior is visible in Figure 5.11, where we kept a fixed dictionary size of $n = 512$ atoms and increased the number of patches in the signal set. Naturally, the speed-up with respect to the case $\tilde{n} = 1$ grows as the size of the problem increases.

To put things in perspective, we implemented AK-SVD in C for comparing times spent on the CPU versus times spent on the GPU. We tried to keep the instructions between the two versions the same wherever possible (the update stage obviously had to vary). As it can be seen in figures 5.10 and 5.11, PAK-SVD is around 10 times faster when $\tilde{n} = n$. It outperforms the CPU version except for the case where $\tilde{n} = 1$, which is to be expected due to the error calculation and compensation for each atom update and also due to the low GPU utilization. For the largest problem, with $n = 512$, $m = 65536$, the speed-up is 11.98.

We also compared the performance of KSVD-Box to our OpenCL implementation of PAK-SVD. The results varied based on the hardware underneath, but on new desktops with multicore processor KSVD-Box has comparable or even smaller execution times than PAK-SVD with $\tilde{n} = n$. This is not surprising, due to the heavily-optimized full and sparse matrix routines available in Matlab and its multi-threading capabilities. We expect that further development of GPU software for numerical computations will increase the performance of our implementation.

Our measurements showed that a single dictionary update (steps 2–7 of PAK-SVD) can be 2 to 3 times faster than the sparse representation stage.

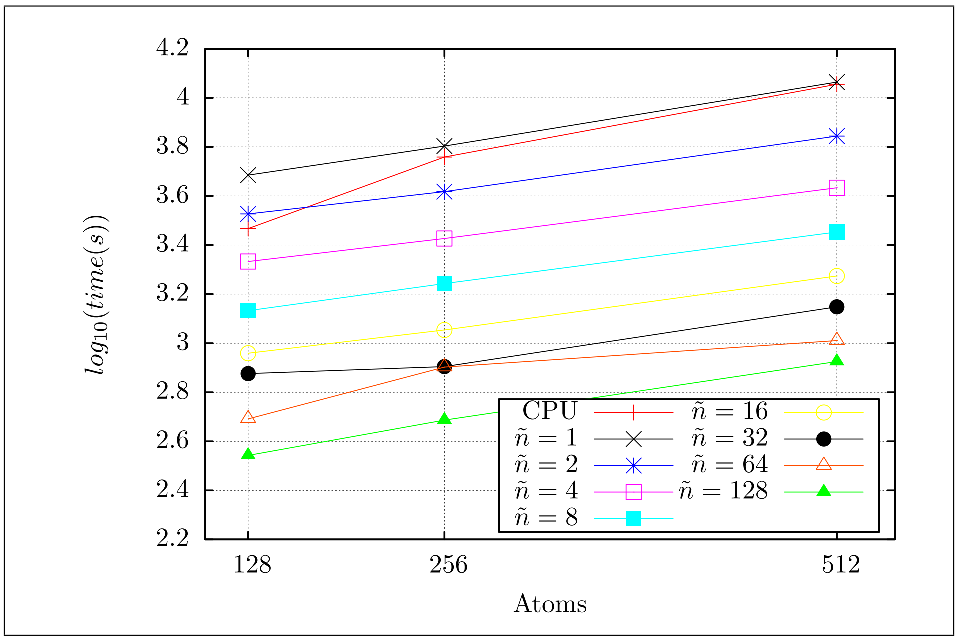

Figure 5.10: Execution times for $m = 16384$, $s = 10$, $K = 200$.

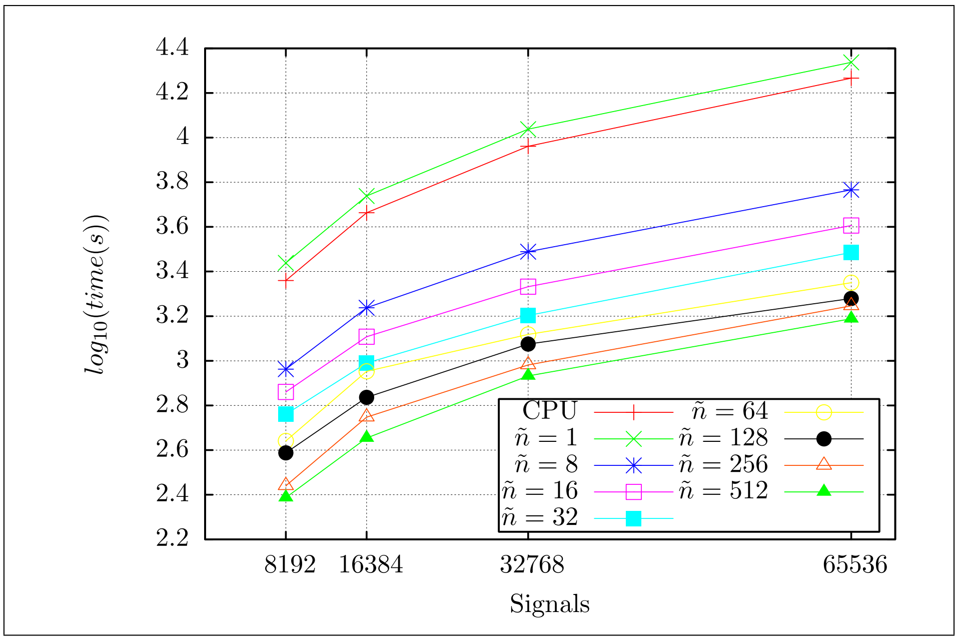

Figure 5.11: Execution times for $n = 512$, $s = 8$, $K = 100$.

This observation lead to modifying the original K-SVD algorithm so that multiple update rounds can be performed during the same SVD iteration. Thus our experiments included varying the parameter u (step 1). Taking as a reference the case where $u = 1$, we found that bumping the number of rounds to $u = 2$ or $u = 3$ does not significantly increase the execution time. So incrementing u allows us to either reach the same approximation error faster or to obtain a better approximation in a similar time as $u = 1$. We obtained the best results by keeping $\tilde{n} \leq 64$ and by using $u = 1$ for the first 50 iterations, and only afterwards increasing the number of dictionary updates per SVD iteration ($u \geq 2$). We noticed that when $u \geq 2$ from the start with large dictionaries ($n = 512$) and high parallelism ($\tilde{n} \geq 128$), the error actually increases at the second or third consecutive update.

Sparsity, Atoms and Training Set Influence

Here we analyze the execution time improvements brought by our JAU strategy when applied on AK-SVD, SGK and NSGK by varying one dictionary design parameter (such as the sparsity constraint) and keep the others fixed (the number of dictionary atoms and training signals).

We present 3 experiments in Figure 5.12 where we vary the sparsity constraint, the atoms in the dictionary and the number of signals in the training set. We depict the JAU versions with green and the sequential versions with red. Because of the significant difference in execution time, we use a logarithmic scale. Again, we used $k = 200$ iterations for all methods.

For the sparsity experiment we used a dictionary of $n = 128$ and a training set of $m = 8192$ and we increased the sparsity from $s = 4$ to $s = 12$ in increments of 2. When studying the dictionary impact on the execution performance we kept a fixed training set of $m = 12288$ and a sparsity of $s = 6$ and varied the atoms from $n = 128$ in increments of 64 up to $n = 512$. Finally, we increased the signal set in increments of 1024 starting from $m = 4096$ until $m = 16384$ with a fixed dictionary of $n = 256$ and a sparsity of $s = 10$. In the panel the abscissa tics represent thousands of signals.

In all of our experiments the JAU versions showed important improvements in execution time, the speed-up reaching values as high as 10.6 times for NSGK, 10.8 times for SGK and 12 times for AK-SVD. This was to be expected, since JAU algorithms are naturally parallel in the atom update stage.

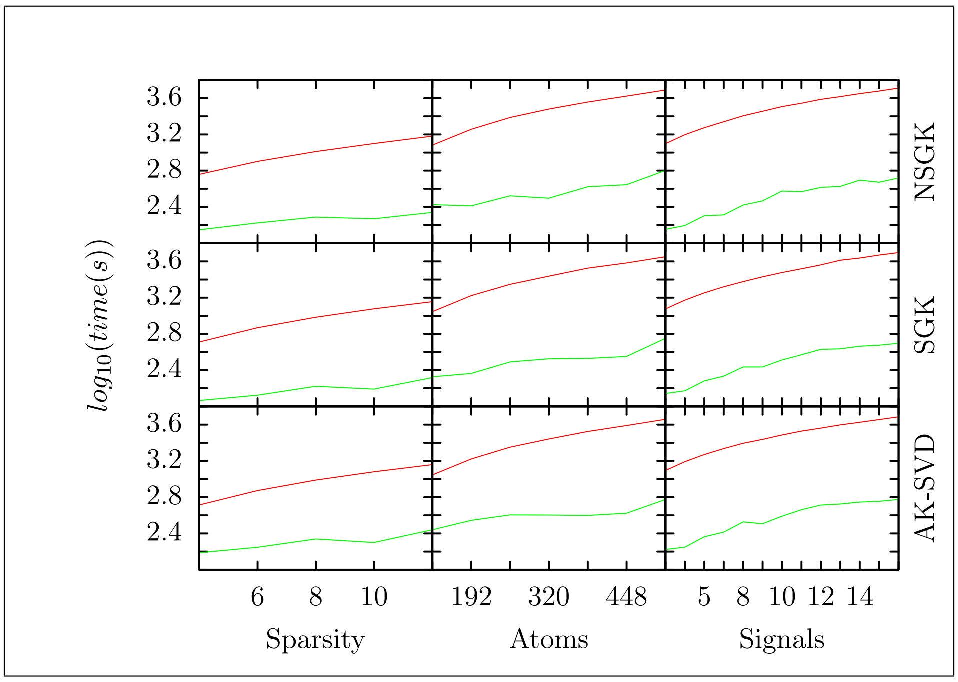

Figure 5.12: Execution times. The sequential versions are red and the JAU algorithms are green.

5.4 Conclusions

We have shown that several dictionary learning algorithms, like AK-SVD [30], SGK [31] and NSGK [32], benefit from adopting Jacobi (parallel) atom updates instead of the usual Gauss-Seidel (sequential) ones. In the mostly academic dictionary recovery problem, the parallel and sequential versions have similar performance. However, in the more practical problem of dictionary learning for sparse image representation, the proposed parallel algorithms have a clearly better behavior.

In the second part of this chapter we studied and proposed efficient GPU implementations using OpenCL for the main algorithms from the dictionary learning field. We provided a full description of the kernels leading to complete parallel execution of the sparse representation and dictionary update stages. We also discussed the n-dimensional topology of each kernel and the optimal storage location of the data structures in order to obtain the best GPU occupancy.

Chapter 6

Mixing Representation and Design Methods

In this chapter we provide an empirical investigation of the impact that the sparse representation algorithm has in DL. The spark to start this study was the somewhat disconcerting fact that, in a test problem often used in the DL community, several algorithms gave quite similar result. As explained in Chapter 4 popular DL methods use OMP in their representation stage. The immediate but apparently ignored question is if OMP is the bottleneck of DL algorithms and thus progress in the atom update stage might be masked by it. Would better representation algorithms allow to discern which atom update method is in fact superior? Would better representation algorithms cause a significant decrease of the overall error in the DL problem (2.8)? Although we cannot provide definitive answers, we try at least to gain more insight into the DL process and assess DL algorithms on a steadier ground.

We give here numerical results for the two DL problems described in Chapter 2 that are used very often as benchmark: dictionary recovery and DL for sparse image representation. In the general structure of Algorithm 1, we use all the combinations of 5 representation methods (OMP, OLS, POMP, POLS, LAOLS) and 6 atom update methods (SGK, P-SGK, NSGK, P-NSGK, AK-SVD, PAK-SVD). All DL algorithms are run with the same data, in particular with the same initial dictionary. For POMP, POLS and LAOLS we took $L = s$.

6.1 Dictionary recovery

We generated the training set and the dictionaries as described in Section 2.5.1. Tables 6.1, 6.2, and 6.3 show the percentages of recovered atoms for all combinations of methods, averaging over 50 tests.

Here are some conclusions that can be drawn from the results.

1. As expected, the more complex representation methods bring indeed an increase in performance: the recovery percentage obtained with POMP, POLS and LAOLS (especially the first two) is clearly better than with OMP, for all atom update methods. OLS and OMP are almost at the same level, with a marginal advantage for OLS.

2. For the same representation method, there is little difference between the atom update methods. The conclusion is that the recovery test (with these commonly used data) is not relevant for comparing atom update methods and that the sparse representation is actually the bottleneck here. A reason may be the small size of the problem or the constant used to decide similarity between the recovered and original atoms. In any case, it appears that the progress in atom update methods can no longer be assessed with this test.

6.2 Dictionary learning

We built the training signals Y from $m = 8192$ random patches that we generated as described in Section 2.5.2. With these signals, we trained dictionaries of size $n = 256$ over 200 iterations, with target sparsity $s = 8$, using again all combinations of methods. The final errors (2.9) are shown in Table 6.4 while the evolution of the representation error over the number of iterations is depicted in figures 6.1–6.11.

Regarding the final error, the conclusions are mixed. With a single exception, all the other representation methods are better than OMP, often much better; in this sense, the results are meeting normal expectations. The best results for an atom update method (in bold) are similar and OLS, LAOLS and POLS share the winners, while POMP is rather disappointing. OLS is clearly better for the parallel methods, a feature shared by OMP and POMP. On the contrary, LAOLS and POLS are systematically better for the sequential methods. We do not have an explanation for this feature of the results.

Figures 6.1–6.6 present the error evolution for each atom update method combined with the diverse representation methods. The error does not decrease uniformly, especially in the first iterations. The parallel methods suffer more from sudden increases in error, however having a decreasing trend.

In Figure 6.1, depicting the results for the SGK algorithm, we can see that OMP is the worst performer with POLS and LAOLS being the clear winners. The parallel version (Figure 6.2) keeps OMP in last place, with OLS taking the lead. Here the error curves are not as smooth and the differences

Table 6.1: Percentage of recovered atoms for SGK and P-SGK

s	Method	SNR			
		10	20	30	∞
3	SGK(OMP)	86.84	90.32	90.20	88.72
	SGK(OLS)	89.08	89.04	89.48	89.00
	SGK(POMP)	91.48	92.72	93.48	93.48
	SGK(POLS)	91.80	92.88	93.48	92.44
	SGK(LAOLS)	89.24	92.24	92.52	**92.88**
	P-SGK(OMP)	87.76	90.44	90.12	89.16
	P-SGK(OLS)	86.52	90.60	90.76	89.40
	P-SGK(POMP)	**92.04**	**94.56**	93.48	92.52
	P-SGK(POLS)	91.48	92.64	**93.64**	92.48
	P-SGK(LAOLS)	88.92	92.92	92.24	92.40
4	SGK(OMP)	71.52	92.16	92.32	91.72
	SGK(OLS)	73.80	93.20	93.36	93.20
	SGK(POMP)	88.04	96.24	95.48	96.40
	SGK(POLS)	87.60	96.28	95.76	**96.72**
	SGK(LAOLS)	85.84	95.52	93.80	94.96
	P-SGK(OMP)	65.64	92.68	92.04	92.48
	P-SGK(OLS)	73.36	92.64	91.68	93.40
	P-SGK(POMP)	87.16	**96.84**	95.76	96.44
	P-SGK(POLS)	**89.96**	96.36	**96.04**	96.04
	P-SGK(LAOLS)	84.64	95.32	95.24	94.88
5	SGK(OMP)	10.56	93.32	93.64	93.64
	SGK(OLS)	12.28	94.96	94.84	94.56
	SGK(POMP)	53.72	98.12	97.52	**98.84**
	SGK(POLS)	64.72	**98.52**	97.48	97.84
	SGK(LAOLS)	58.44	96.76	97.32	97.24
	P-SGK(OMP)	10.44	93.24	93.80	93.96
	P-SGK(OLS)	12.12	94.16	94.12	94.72
	P-SGK(POMP)	56.32	98.28	**98.12**	98.20
	P-SGK(POLS)	**69.28**	98.36	**98.12**	97.60
	P-SGK(LAOLS)	57.76	96.56	97.48	97.36

Table 6.2: Percentage of recovered atoms for NSGK and P-NSGK

s	Method	SNR			
		10	20	30	∞
3	NSGK(OMP)	87.12	90.52	90.32	89.92
	NSGK(OLS)	87.44	91.52	90.28	90.52
	NSGK(POMP)	91.40	93.16	92.48	**93.96**
	NSGK(POLS)	90.96	93.32	**93.28**	93.40
	NSGK(LAOLS)	88.76	92.52	92.20	92.44
	P-NSGK(OMP)	86.12	90.52	91.36	89.84
	P-NSGK(OLS)	87.60	92.08	90.40	90.04
	P-NSGK(POMP)	90.36	93.48	**93.28**	91.88
	P-NSGK(POLS)	**91.48**	**93.76**	92.96	93.32
	P-NSGK(LAOLS)	88.44	92.52	92.60	93.36
4	NSGK(OMP)	68.16	92.88	92.76	92.84
	NSGK(OLS)	70.76	94.04	93.28	93.04
	NSGK(POMP)	85.40	**96.60**	95.96	**96.68**
	NSGK(POLS)	86.32	96.36	95.80	95.80
	NSGK(LAOLS)	83.44	96.08	95.60	95.48
	P-NSGK(OMP)	69.28	93.64	92.92	93.88
	P-NSGK(OLS)	70.24	93.48	93.44	93.96
	P-NSGK(POMP)	87.44	96.44	**96.20**	96.04
	P-NSGK(POLS)	**88.76**	96.52	96.12	96.44
	P-NSGK(LAOLS)	83.64	95.00	95.20	95.48
5	NSGK(OMP)	9.60	94.12	93.88	94.16
	NSGK(OLS)	8.08	94.96	94.72	95.04
	NSGK(POMP)	53.48	**98.52**	97.80	97.72
	NSGK(POLS)	**67.96**	97.48	97.48	97.76
	NSGK(LAOLS)	56.96	97.08	97.08	97.72
	P-NSGK(OMP)	11.68	94.32	93.60	93.52
	P-NSGK(OLS)	8.96	94.84	95.12	94.40
	P-NSGK(POMP)	51.72	98.32	**98.56**	**98.32**
	P-NSGK(POLS)	65.52	98.20	98.20	97.72
	P-NSGK(LAOLS)	51.96	97.64	97.56	97.36

Table 6.3: Percentage of recovered atoms for AK-SVD and PAK-SVD

s	Method	SNR			
		10	20	30	∞
3	AK-SVD(OMP)	88.00	89.16	89.96	88.20
	AK-SVD(OLS)	88.72	90.52	89.68	89.84
	AK-SVD(POMP)	91.24	94.00	92.92	92.32
	AK-SVD(POLS)	91.48	93.40	92.88	**92.56**
	AK-SVD(LAOLS)	88.40	92.08	92.76	92.28
	PAK-SVD(OMP)	87.76	90.44	90.12	89.16
	PAK-SVD(OLS)	86.52	90.60	90.76	89.40
	PAK-SVD(POMP)	**92.04**	**94.56**	93.48	92.52
	PAK-SVD(POLS)	91.52	92.64	**93.64**	92.48
	PAK-SVD(LAOLS)	88.92	92.64	92.28	92.44
4	AK-SVD(OMP)	71.80	92.44	92.76	92.68
	AK-SVD(OLS)	73.12	93.44	92.96	92.96
	AK-SVD(POMP)	88.04	**96.84**	95.84	**96.76**
	AK-SVD(POLS)	88.24	96.04	**96.36**	95.60
	AK-SVD(LAOLS)	82.92	94.92	96.24	95.68
	PAK-SVD(OMP)	65.64	92.68	92.04	92.48
	PAK-SVD(OLS)	73.36	92.64	91.68	93.40
	PAK-SVD(POMP)	87.16	**96.84**	95.76	96.44
	PAK-SVD(POLS)	**90.08**	96.36	96.04	96.04
	PAK-SVD(LAOLS)	84.60	95.32	95.24	94.92
5	AK-SVD(OMP)	10.48	93.44	93.20	93.00
	AK-SVD(OLS)	12.24	93.64	94.60	95.20
	AK-SVD(POMP)	54.04	**98.44**	97.64	98.12
	AK-SVD(POLS)	62.56	98.12	**98.28**	**98.96**
	AK-SVD(LAOLS)	48.16	96.96	96.72	96.44
	PAK-SVD(OMP)	10.44	93.24	93.80	93.96
	PAK-SVD(OLS)	12.12	94.16	94.12	94.72
	PAK-SVD(POMP)	56.32	98.28	98.12	98.20
	PAK-SVD(POLS)	**69.16**	98.36	98.12	97.52
	PAK-SVD(LAOLS)	59.60	97.56	97.00	97.36

between the sparse representations methods are not as pronounced.

Again, for AK-SVD (Figure 6.3) OMP provides the worst approximation but now POLS comes first with LAOLS in second place. The differences between the representation algorithms are also clearer in this graph. PAK-SVD (Figure 6.4) has a relatively similar behaviour as PSGK in that the error differences are smaller than in the sequential version. OLS is first and OMP is last.

Figure 6.5 shows NSGK whose error curves are not smooth, unlike its sequential competitors, with OMP (which comes in last) having the most accentuated variations. POLS and LAOLS distance themselves to the top again. It is a bit of a surprise to see the parallel version (Figure 6.6) with smooth curves for all algorithms besides OMP (which again places last). For the third time, OLS is the clear winner.

Figures 6.7–6.11 present the error evolution from the viewpoint of the sparse representation methods. We note that POMP, POLS and LAOLS have a smoothing effect on the curves for all atom update methods, the decrease being more or less uniform after the first iterations.

We can see in Figure 6.7 that the dictionary update algorithms that make the most out of OMP are the Jacobi versions that give almost identical results. P-NSGK stands out for its irregular error descend. SGK and AK-SVD are clearly the poorest performers.

Figure 6.8 shows that OLS has a profound smoothing effect on P-NSGK placing it first with P-SGK and PAK-SVD coming in second with curves similar to the OMP case. SGK and AK-SVD are again last.

POMP (Figure 6.9) bumps P-SGK and PAK-SVD first and SGK last. We notice a general smoothing effect on all the parallel update methods.

POLS (Figure 6.10) provides better approximations for all 6 atom algorithms and maintains the smoothing effect. It is interesting to see that AK-SVD is the best choice for POLS while, for the first time a parallel algorithm, P-NSGK, comes in last.

In terms of smoothness, LAOLS (Figure 6.11) seems to be the best choice. Even though it does not provide the smallest errors for all methods, its results are similar to POLS. At the top we find SGK and NSGK, sequential wins again, with P-NSGK in last place, again.

Finally, one may wonder if the designed dictionaries could be used with OMP as representation method, although another method has been employed in learning. The reason is that one may need the dictionaries in practical applications where speed is of the essence, hence one may want to use the fastest reliable representation method, i.e. OMP, which is at least a few times faster than the other methods discussed in this study. On the contrary, in learning, which is a one-time operation, we often have the luxury to use

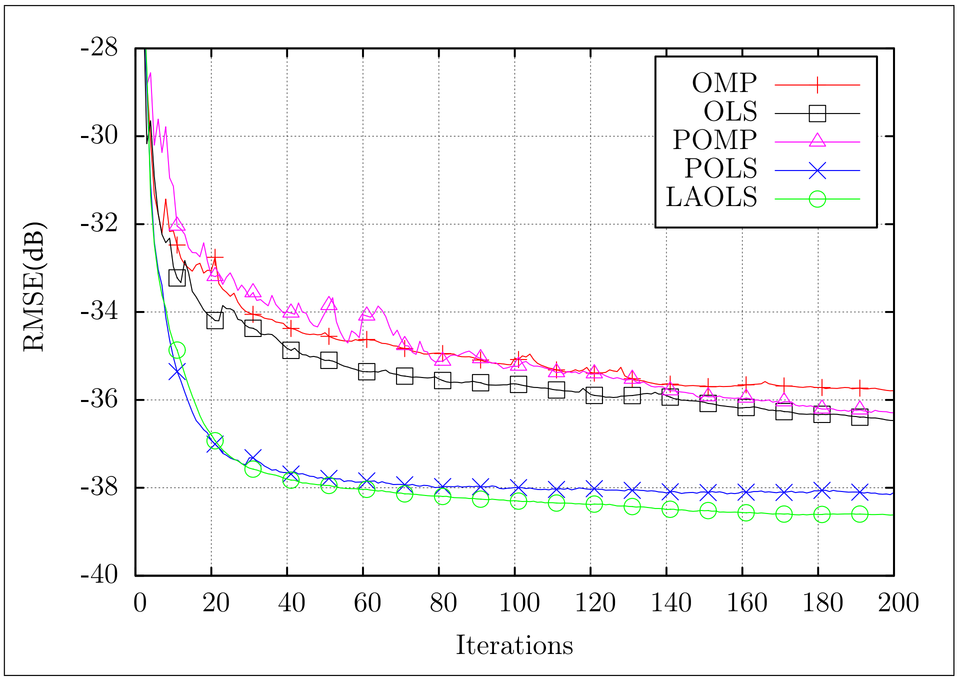

Figure 6.1: Error evolution for SGK with different representation methods.

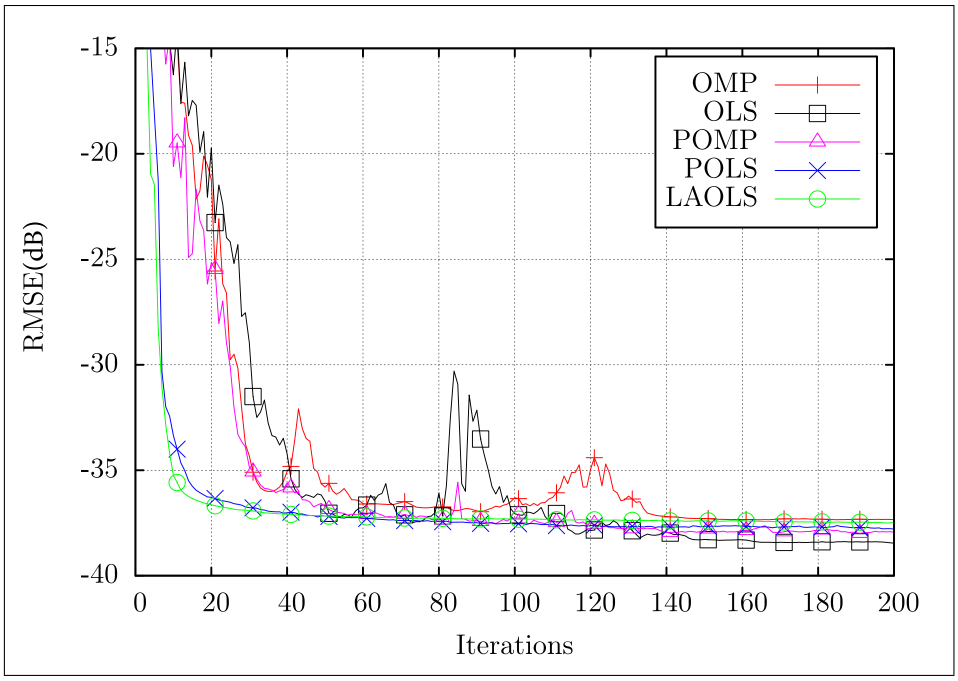

Figure 6.2: Error evolution for P-SGK with different representation methods.

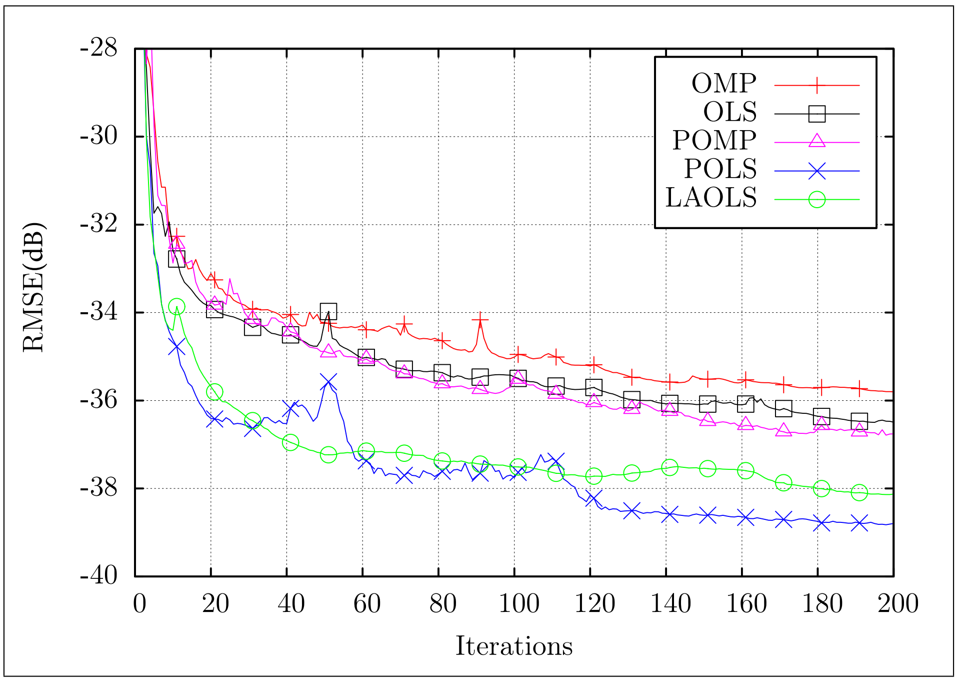

Figure 6.3: Error evolution for AK-SVD with different representation methods.

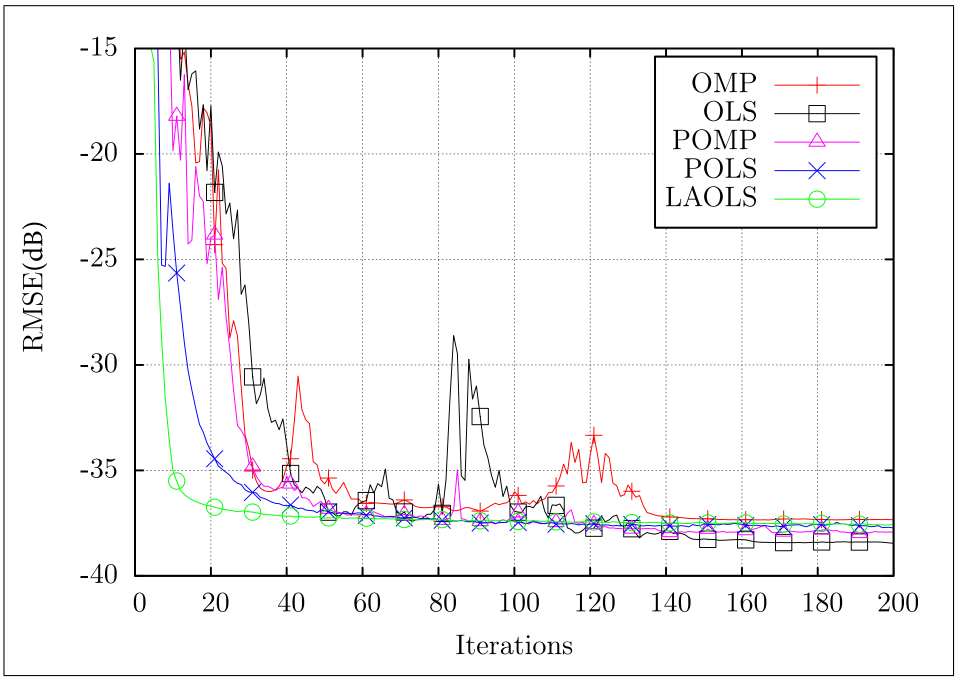

Figure 6.4: Error evolution for PAK-SVD with different representation methods.

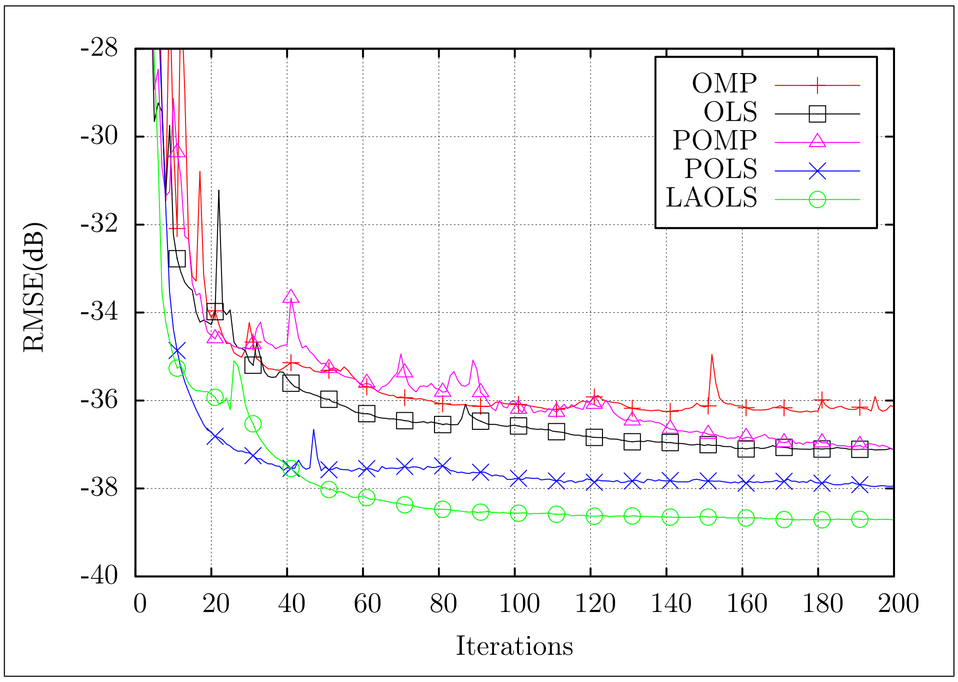

Figure 6.5: Error evolution for NSGK with different representation methods.

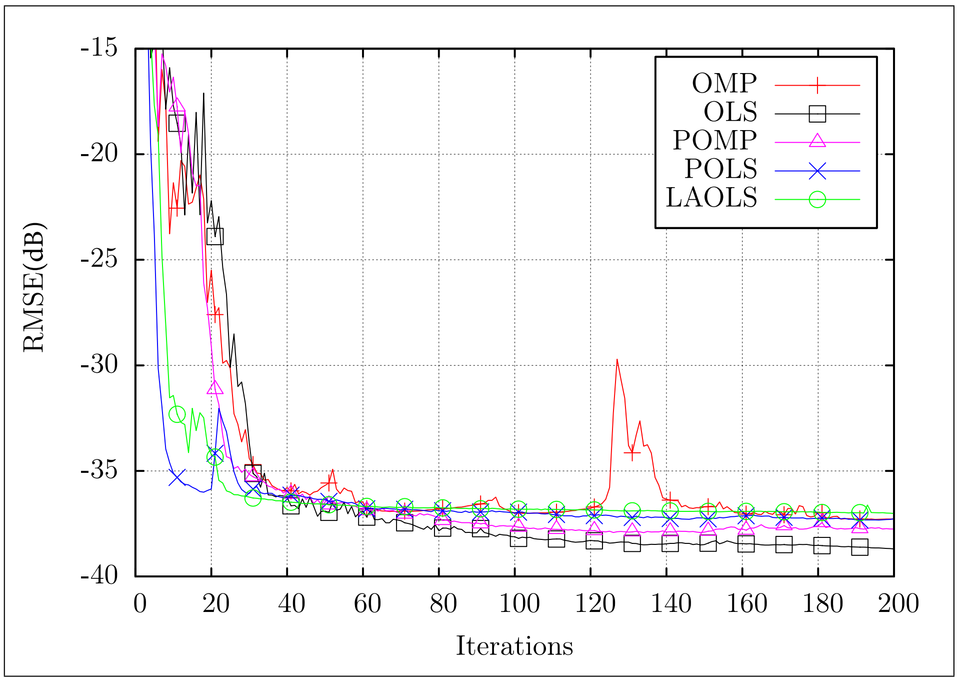

Figure 6.6: Error evolution for P-NSGK with different representation methods.

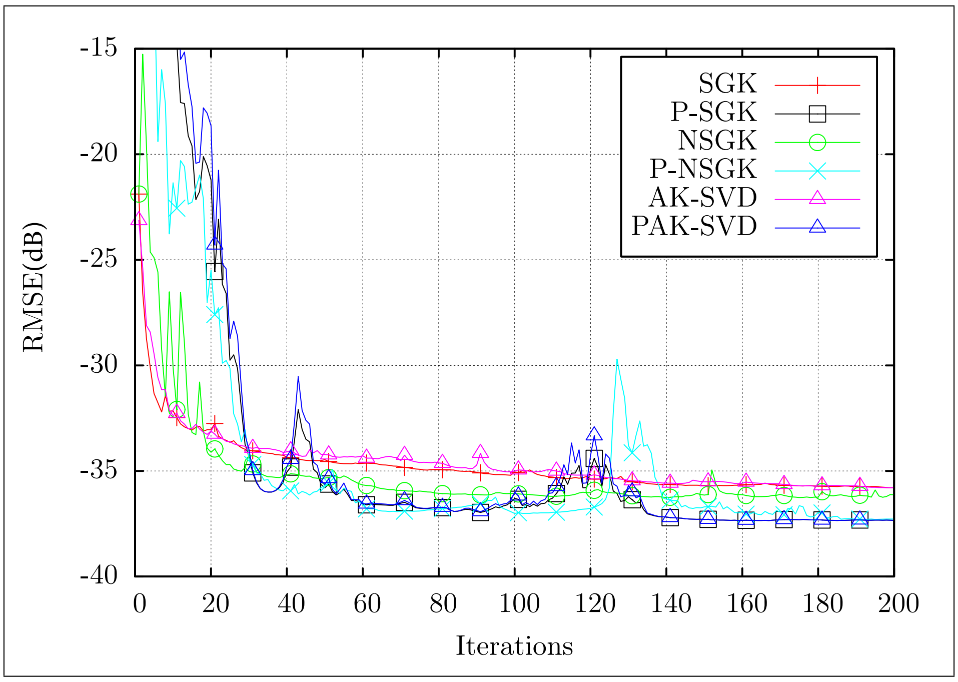

Figure 6.7: Error evolution for OMP with different dictionary update methods.

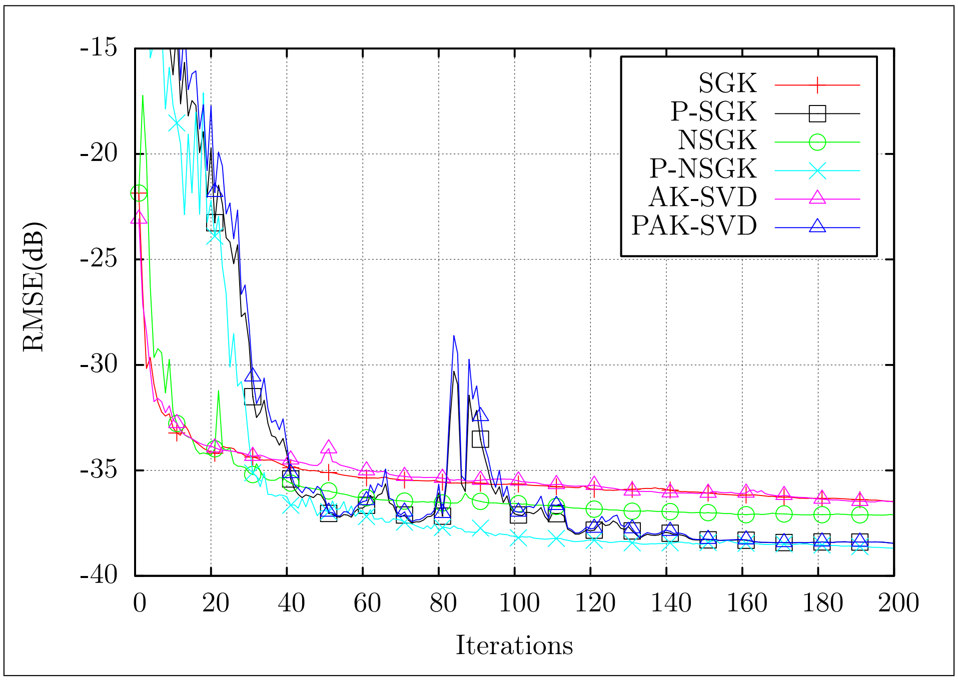

Figure 6.8: Error evolution for OLS with different dictionary update methods.

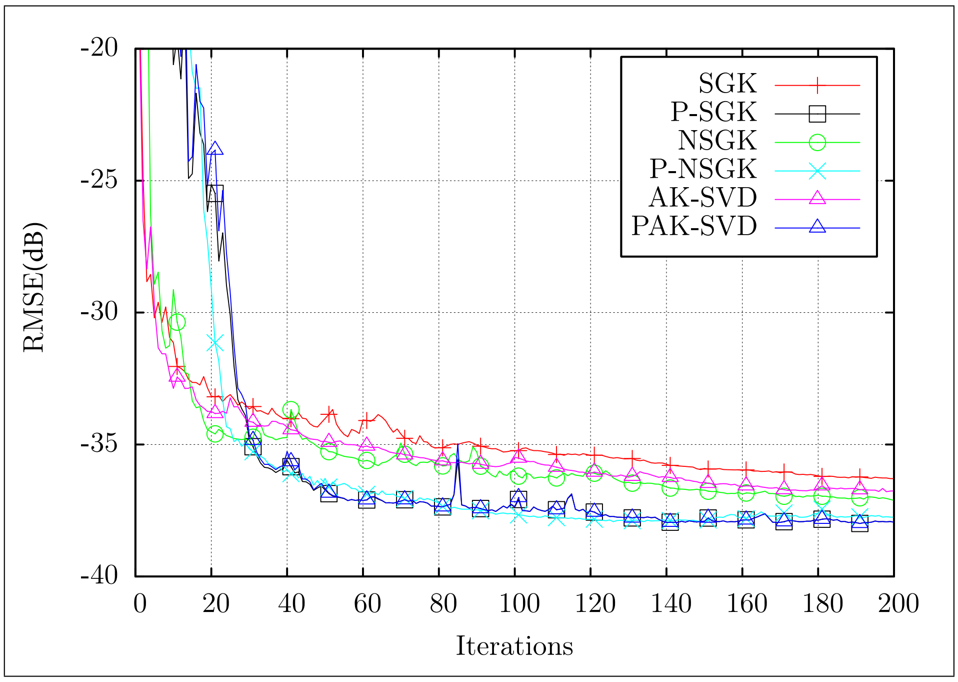

Figure 6.9: Error evolution for POMP with different dictionary update methods.

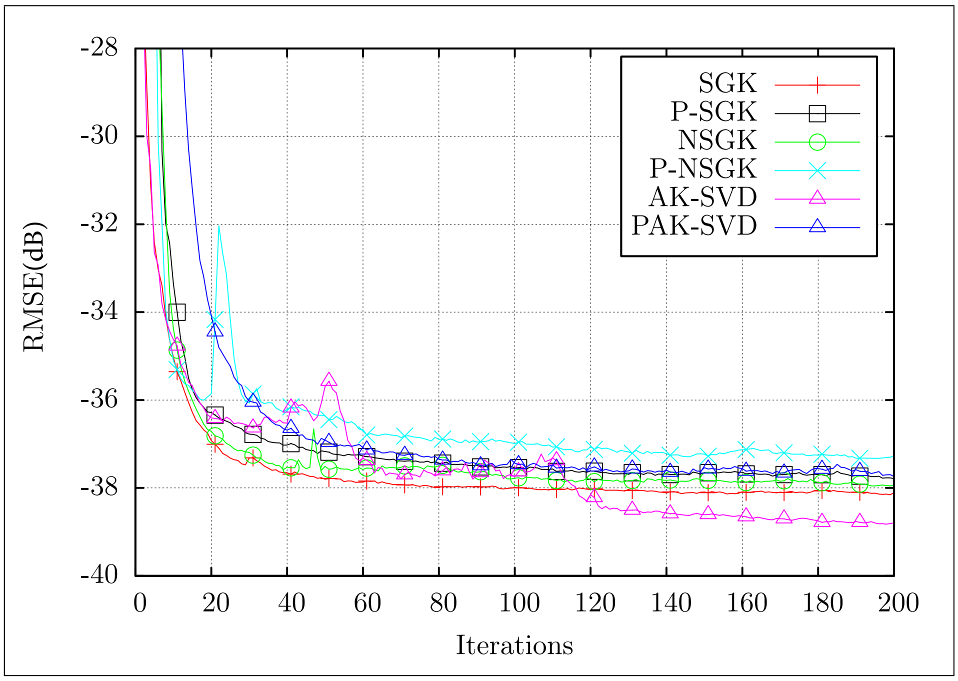

Figure 6.10: Error evolution for POLS with different dictionary update methods.

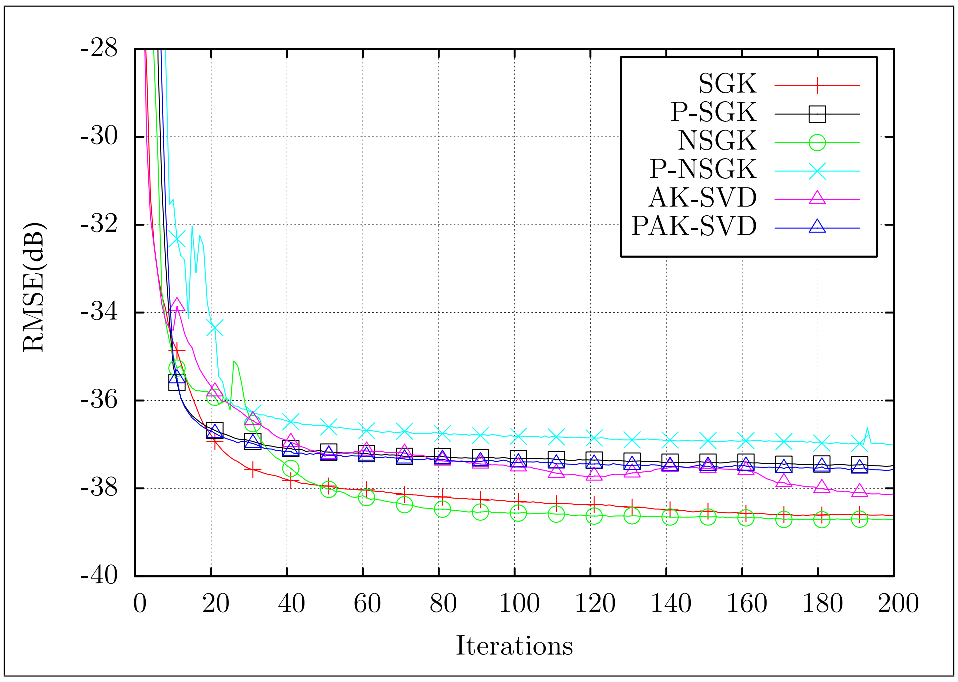

Figure 6.11: Error evolution for LAOLS with different dictionary update methods.

higher complexity methods for obtaining a very good dictionary. So, we computed with OMP the representations X associated the dictionaries D designed by the various methods. The errors are given in Table 6.5, whose structure is similar to that of Table 6.4. The first column is identical, since OMP is used both for learning and representation. The other columns are different, since the method used for learning is not OMP. As expected, the results in Table 6.5 are generally worse than those in Table 6.4. However, we are interested in the results that are better than in the first column; these are shown with bold digits. Remarkably, POLS gives consistently better results, which means that we could use any atom update method coupled with POLS in the DL process, then use the designed dictionary with OMP and thus get better representations than using OMP in training. We can say that this is a new design method that is better than the current methods in exactly the same conditions.

Table 6.4: DL final errors

	OMP	OLS	POMP	POLS	LAOLS
SGK	0.0162	0.0150	0.0153	0.0124	**0.0117**
P-SGK	0.0136	**0.0119**	0.0126	0.0129	0.0133
NSGK	0.0154	0.0139	0.0139	0.0126	**0.0116**
P-NSGK	0.0136	**0.0116**	0.0127	0.0136	0.0141
AK-SVD	0.0162	0.0150	0.0148	**0.0114**	0.0124
PAK-SVD	0.0136	**0.0119**	0.0126	0.0130	0.0132

Table 6.5: Representation errors using designed dictionaries and OMP

	OMP	OLS	POMP	POLS	LAOLS
SGK	0.0162	0.0332	0.0170	**0.0119**	**0.0127**
P-SGK	0.0136	0.0156	0.0157	**0.0124**	0.0136
NSGK	0.0154	0.0252	**0.0149**	**0.0123**	**0.0124**
P-NSGK	0.0136	**0.0133**	0.0159	**0.0128**	0.0144
AK-SVD	0.0162	0.0251	0.0165	**0.0119**	**0.0136**
PAK-SVD	0.0136	0.0156	0.0157	**0.0124**	0.0136

6.3 Conclusions

We have studied the effects of combining several sparse representation and atom update methods for solving the problem of overcomplete dictionary design. As expected, replacing OMP with more sophisticated methods like OLS, POMP, POLS and LAOLS leads to better results and smoother convergence. In image representation applications, we also conclude that combining OLS with parallel atom update methods gives systematically good results. Finally, combining POLS and any considered atom update method produces dictionaries that give small representation errors also with OMP, thus allowing the fastest implementation for the use of the designed dictionary and obtaining better representations than all existing dictionary design methods that employ OMP in the learning process.

Part III

Particular Dictionary Forms

Chapter 7

Structured Dictionaries

7.1 Introduction

While the generic dictionary learning problem does not impose any specific structure on the dictionary D, some methods [33,34] build the dictionary as a union of smaller blocks consisting of ortonormal bases (ONBs) that transform the optimization problem into:

$$\begin{aligned} \underset{D,X}{\text{minimize}} \quad & \|Y - [Q_1 Q_2 \dots Q_K]X\|_F^2 \\ \text{subject to} \quad & \|x_i\|_0 \leq s, \ \forall i \\ & Q_j^T Q_j = I_p, \ 1 \leq j \leq K, \end{aligned} \tag{7.1}$$

where the union of K ONBs denoted $Q_j \in \mathbb{R}^{p \times p}$, with $j = 1 \dots K$, represents the dictionary D.

The union of orthonormal basis algorithm (UONB) [33] and the single block orthogonal (SBO) [34] algorithm enforce this structure on the dictionary by using singular value decomposition (SVD) to create each orthonormal block. The difference between the two is that for representing a single data item the former uses atoms selected via OMP from all bases, while the later uses atoms from a single orthoblock. Because of its representation strategy, SBO uses more dictionary blocks than UONB but also executes faster while maintaining the same representation error.

We are interested in parallelizing SBO because it brings data-decoupling through its single block representation system and also because it does not depend on OMP which raised hard full GPU occupancy problems, even when applying the partitioned global address space approach, due to its high memory footprint [52].

This chapter presents in Section 7.2 an improved parallel algorithm called P-SBO, followed by details of its OpenCL implementation in Section 7.3, and

Algorithm 12: SELECT

```
Data:   unsorted list $x \in \mathbb{R}^n$, partial selection $s$
Result: partially sorted list $x$
1 for $i \leftarrow 1$ to $s$ do
2     $maxidx = i$
3     $maxval = x(i)$
4     for $j \leftarrow i + 1$ to $n$ do
5         if $|x(j)| > |maxval|$ then
6             $maxidx = j$
7             $maxval = x(j)$
8     $x(i) \leftrightarrow x(maxidx)$
```

the numerical results supporting its representation error and execution time improvements in Section 7.4.

7.2 The P-SBO Algorithm

We term our algorithm P-SBO which is short for parallel single block orthogonal algorithm. P-SBO builds the dictionary as a union of orthoblocks. Each data-item from Y is constrained to use a single block Q for its sparse representation x such that:

$$y \approx Qx. \tag{7.2}$$

The representation of x results from computing the product $x = Q^T y$ and then hard-thresholding the s highest absolute value entries. This is performed through partial selection as described in algorithm 12.

Given a list x the algorithm proceeds to do in-place sorting by finding the absolute maximum value (steps 4–7) and placing it at the top of the list (step 8). This is repeated s times (step 1) by performing the search on the remaining entries from x (steps 2–3). Even though this has an $O(sn)$ complexity, which is asymptotically inefficient, in our case s is small enough that it makes our choice sufficiently efficient and trivial to implement. We note that the fastest (but more involved) partial sorting algorithm [53] has $O(n)$ complexity.

The best orthonormal base j to represent a given signal y is picked by computing the energy of the resulting representation coefficients from x and

Algorithm 13: 1ONB

```
Data:   signals set $Y$, initial dictionary $Q_0$,
        target sparsity $s$, number of rounds $R$
Result: trained dictionary $Q$, sparse coding $X$
1 $Q = Q_0$
2 for $r \leftarrow 1$ to $R$ do
3     $X = Q^T Y$
4     $X_j = \text{SELECT}(X_j, s), \forall j$
5     Apply Procrustes orthogonalization (7.6) on $Y$ and $X$ to
      approximate $Q$
```

selecting the orthobase where the energy is highest. Let

$$E_x = \sum_{n=1}^{p} |x_n|^2 \tag{7.3}$$

denote the energy of a given signal x and

$$x^i = \text{SELECT}(Q_i^T y, s) \tag{7.4}$$

denote the representation of signal y with base i, then picking the best orthonormal base can be expressed as

$$j = \underset{i=1\ldots K}{\text{argmax}}(E_{x^i}). \tag{7.5}$$

It is enough to compute the energy of the representations because the norm is preserved by the unitary dictionary blocks. Following this method, each data-item from Y is represented by a single orthobase in a process that we will call representation.

The alternative optimization iterations for performing dictionary learning on a single orthonormal base is presented in Algorithm 13.

By keeping a fixed dictionary Q, step 3 computes the new representations X and step 4 performs hard-thresholding through partial sorting to select the largest s values on each column. Using the new matrix X, the dictionary is refined (step 5) by using the product of the resulting orthonormal matrices from the SVD computation of YX^T. This orthogonal approximation of X and Y is also called Procrustes orthogonalization [54] in the literature:

$$\begin{aligned} P &= YX^T \\ U\Sigma V^T &= \text{SVD}(P) \\ Q &= UV^T. \end{aligned} \tag{7.6}$$

Algorithm 14: P-SBO

```
Initialization
1 Iteratively train $K_0$ orthonormal blocks by randomly selecting $P_0$
  signals from $Y$ and applying 1ONB $R$ times: $D = [\ Q_1 \ \ \dots \ \ Q_{K_0}\ ]$
2 Represent each data-item with only one of the previously computed
  ONBs following (7.5)
Iterations
3 Construct the set of the worst $W$ represented data items and train $\tilde{K}$
  new orthobases with this set. Add the new bases to the existing union
  of ONBs.
4 Represent each data item with one ONB
5 Train each orthobase over its new data set
6 Check stopping criterion
```

The results from [33] show that, with a good initialization (step 1), good results can be reached by just a few iterations ($R < 5$ in step 2). Also, a good starting point when creating a new orthoblock is to use the left-hand side orthonormal matrix of the SVD decomposition of the given data set

$$Y = U\Sigma V \rightarrow Q_0 = U. \tag{7.7}$$

Based on the above, P-SBO is described in Algorithm 14. The method is split in two parts: the initialization phase and the dictionary learning iterations.

The initialization phase builds a small start-up dictionary consisting of K_0 orthobases each trained with P_0 sized signal chunks that are used by 1ONB to initialize and train a new orthobase (step 1). The resulting dictionary is used by step 2 to perform data item representation which leads to an initial sparse representation set.

The training iterations start by building $\tilde{K}$ new orthobases for the worst W represented signals using Algorithm 13 and expanding the dictionary to include the new ONBs (step 3). Training $\tilde{K} > 1$ orthobases per iteration improves the SBO algorithm proposed in [34] by providing a better representation error and at the same time reducing the execution time as can be seen in the numerical experiments from Section 7.4.

Given that the dictionary has changed, a new data-item representation is needed and with that step 4 computes a new set of sparse representations. Step 5 refines the dictionary D by applying 1ONB on each orthobase over its newly associated data set. The learning process is stopped by either reaching a given target error or the permitted maximum number of orthonormals.

7.3 Parallel SBO with OpenCL

In this section we will go through the main points behind our parallel version, then give some details on the OpenCL specifics.

7.3.1 Parallel Representations

The sparse representations are completely independent and so their computation is done in parallel by applying (7.5) on each data-item. More specific, for each signal from Y we compute the representations with every available orthoblock and pick the one that has the highest energy.

This task fits naturally on the map-reduce model (Figure 7.1). We map the data in signal-orthobase pairs that produce the energy of the resulting sparse representation. Each pair computes the representation with the current dictionary block j ($x = Q_j^T y$), does a hard-threshold on the largest s items in absolute value, and outputs the energy E of the resulting sparse coding. Parallelization is done in bulk by performing the above for all ONBs at once in groups of $\tilde{m}$ signals. The result is that each data item has an associated energy list of its representation with each block from the dictionary. We reduce the list, for each signal in Y, to the element with the largest energy leading to the choice of a single representation block.

7.3.2 Parallel Dictionary Training

Dictionary learning is performed by the operations of 1ONB described in Algorithm 13. P-SBO makes use of 1ONB in three different contexts: once during the initialization phase (step 1), and twice during the training iterations while learning a new dictionary for the W worst represented signals (step 3) and while training the existing dictionary over its new data set (step 5).

Due to the decoupled nature of the data, we add parallelism at the dictionary level (each orthoblock is initialized and trained in parallel) and we also further parallelize the steps of each orthoblock training instance (see Figure 7.2). This approach allows us to execute the sequential operations inside 1ONB (mainly the SVD routines) in parallel for each dictionary block.

The following provides the parallel implementation details of each step from Algorithm 13. If an initial orthonormal basis is not supplied, we gen-

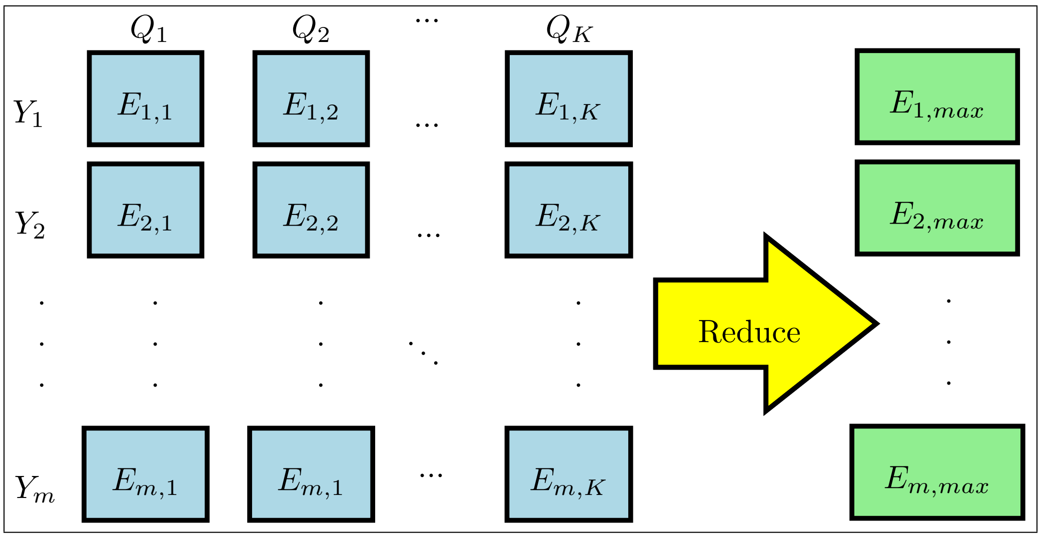

Figure 7.1: MapReduce for $\tilde{m} = 1$ and K orthobases

erate a new basis by using the singular value decomposition as described in (7.7). This, along with the other SVD operation from step 5 are executed in parallel for each dictionary block. The alternative optimization iterations (steps 3–5) train the orthonormal dictionary Q such that $\|Y - QX\|_F$ is minimized or reduced. First, keeping a fixed dictionary, the sparse representations are computed in step 3. Since this is done via matrix multiplication of large dimensions it can be easily parallelized through the classic concurrent sub-block multiplication routines. The target sparsity is obtained by hard-thresholding the largest s absolute value entries (step 4). We compute the thresholding in parallel for groups of $\tilde{m}$ signals by evenly partitioning the global address space for each thread of execution. Second, using the new matrix X, we update the dictionary in step 5 via the SVD decompositon of YX^T (see Equation 7.6) by using the resulting orthonormal matrices U and V. We perform the YX^T matrix multiplication and the decomposition in parallel just as we did before. $Q = UV$ represents a matrix multiplication of relatively small dimensions $(p \times p)$ for which analysis showed that it is better to employ a partitioned global address space strategy so that each thread performs a few corresponding vector-matrix operations resulting in a simultaneous update of all orthobases.

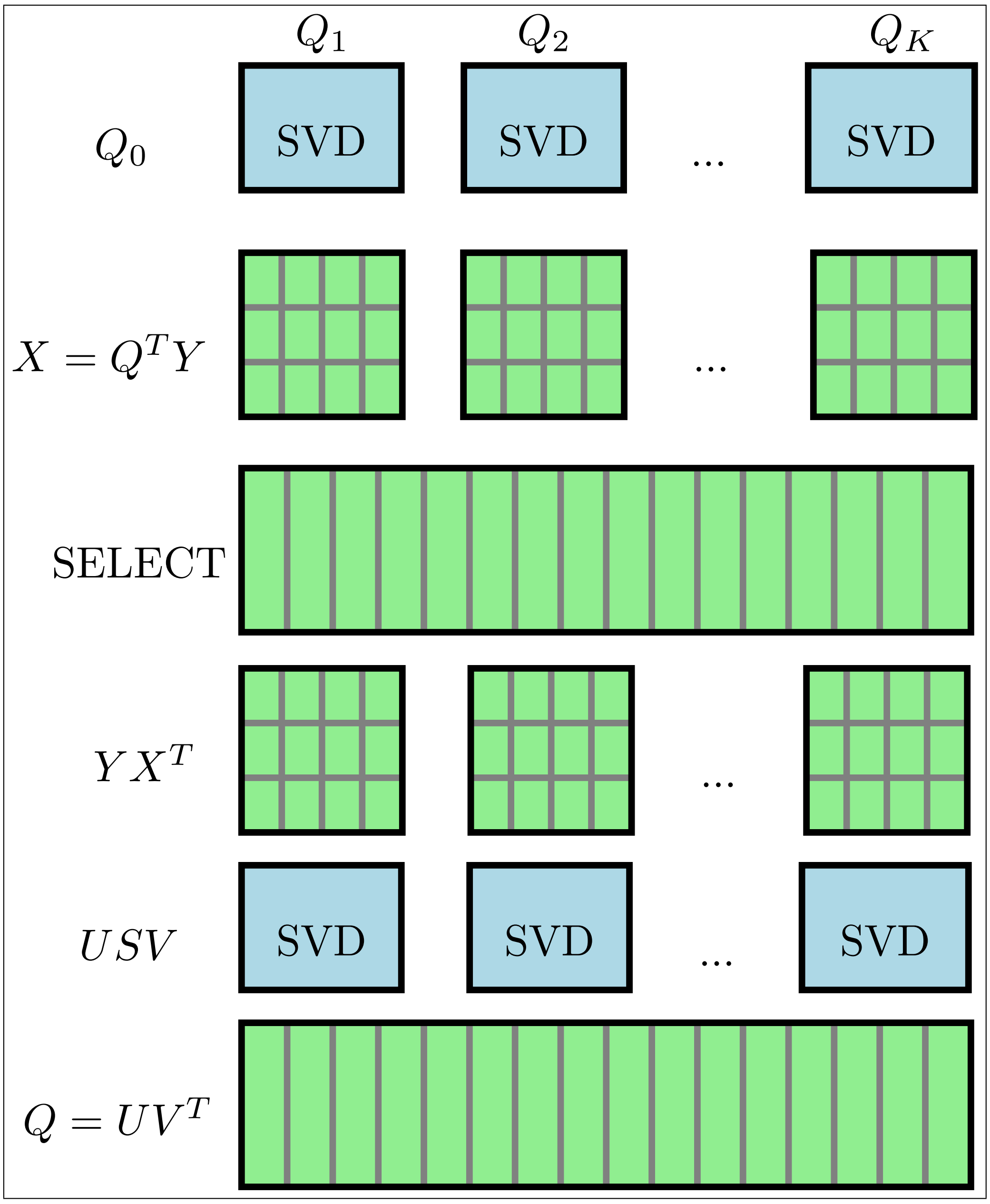

Figure 7.2: The parallel execution of 1ONB for $R = 1$ rounds and K orthobases. Each block represents a task and each sub-block depicts a thread of execution within that task.

7.3.3 OpenCL Implementation Details

We described the OpenCL standard and its execution model along with measuring kernel efficiency in Chapter 3. Here, we are going to use these notions to look at the SBO and 1ONB implementations in OpenCL.

Matrix Multiplication

Steps 3 and 5 from the 1ONB algorithm were implemented using the BLAS library for OpenCL from AMD. The AMD kernels follow the classic GEMM BLAS model. Input matrices and the result are stored in global memory. The operation first creates matrix sub-groups and then does block-based full-matrix multiplication on them. While the AMD implementation does not take full advantage of the hardware underneath, it is fast enough for our use-case. We compensate its poor occupancy of the GPU resources (profiling our simulations with AMD's CodeXL showed 33.3% for the sub-grouping and 25% for the block multiplication) by scheduling as many simultaneous GEMM operations as there are orthobasis (P-SBO step 1 and step 5).

Representation

Given k orthoblocks, all the operations required for finding the best dictionary block for the sparse representation of each data item from the signal set, P-SBO step 2 and 4, were packed and implemented by a single OpenCL kernel following the optimization problem (7.5).

The input matrices as well as the resulting orthobase representation index of each signal and its energy are kept in global memory. We can keep the actual sparse representations in private memory because only the energy and base representation indices are needed by P-SBO. During representation, the sparse signal storage is accessed multiple times for each orthobase in order to compute $x = Q^T y$. Keeping the memory private gains us low latency times at the expense of an increased number of vector general purpose registers used which, in turn, leads to a lower occupancy level. Our numeric experiments showed that lower latency outbids by far a partitioned global memory, full-occupancy version of the kernel.

We designed the representation kernel following the map-reduce paradigm. We map each work-item to a signal-orthoblock couple. Each processing element is in charge of sparse coding and computing the resulting energy of a few $\tilde{m}$ signals using a single orthobase. The energy is saved in a matrix in local memory at the signal-orthobase coordinates corresponding to the work-item's position in the work-group. We keep 2-dimensional work-groups with orthobases in the first dimension and signals on the second as depicted

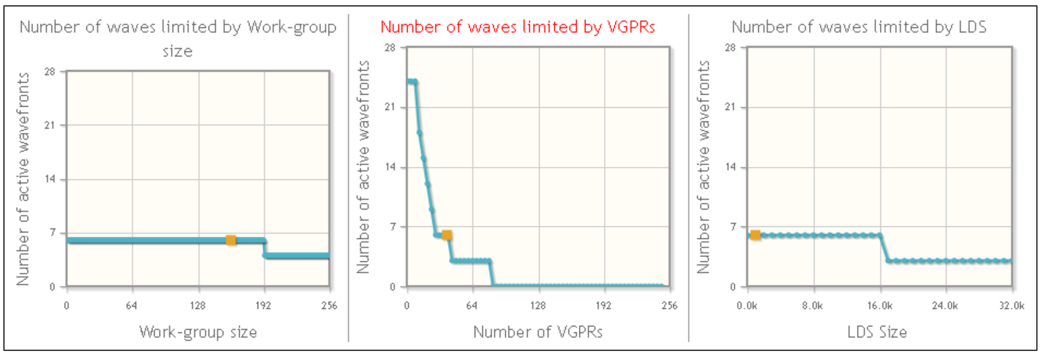

Figure 7.3: Representation kernel occupancy for $K = 24$ orthobases

on the left side of Figure 7.1. And so we split the signal set in $\tilde{m}$ sized chunks representing the number of work-groups scheduled for processing on the compute-units, corresponding to an $NDR(\langle k, m\rangle, \langle k, \tilde{m}\rangle)$ splitting. The reduction on the columns of the energy matrix is performed by each work-item with ID 0 in the orthobase dimension (see the right-side of Figure 7.1). Even though this approach leaves most of the work-items idling when reducing, the overhead of doing map-reduce in the same kernel (opposed to doing it in two separate ones) is insignificant in this case.

This design choice and the way it affects occupancy can be observed in Figure 7.3 that shows how resource utilization affects the number of simultaneous active wavefronts for the representation kernel. As expected, keeping the sparse representations in private memory increased the number of VGPRs used that in turn limited the number of active wavefronts to 6 as depicted in the center graph. The left and right panes show that increasing the work-group size to more than 192 work-items or expanding the LDS past 16KB would decrease the device occupancy even further.

Dictionary Training

The dictionary update process, P-SBO step 1 and step 5, was split into parts and implemented by multiple OpenCL kernels. We keep the input matrices for the dictionary and the signal set in global memory as well as the resulting sparse representations. The dictionary bases are modified in-place.

Before starting the dictionary training phase in P-SBO's step 5, we group the signals in blocks based on the dictionary-base used for their representations. This speeds-up the training process by using coalesced memory in P-SBO's parallel implementation. We first build a list of signals for each base Q and then we walk it contiguously copying the signals in base order and

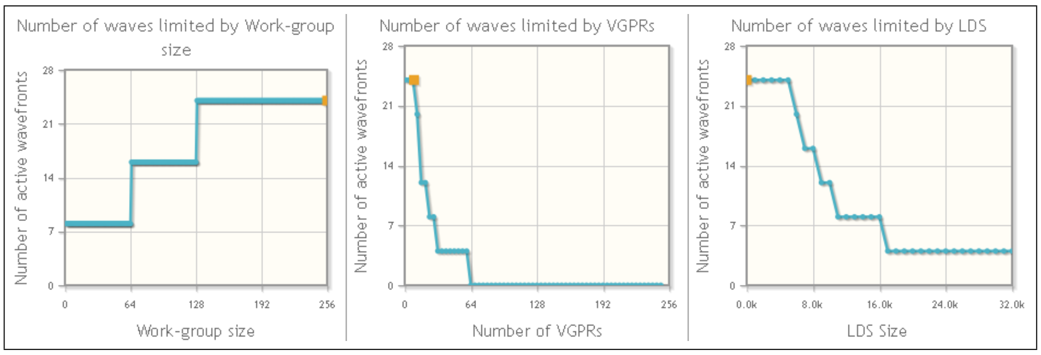

Figure 7.4: Partial selection kernel occupancy for $m = 16384$ signals

overwriting the matrix Y. This is a cheap operation that brings a big performance boost by helping data access times of the execution threads. Copying proved to be up to $1000\times$ more effective by mapping the signal matrix in host memory and using *memcpy* than plainly using *clEnqueueCopyBuffer*.

For the implementation of Algorithm 13 we decided to use a Numerical Recipes based implementation of the SVD algorithm. We execute it in parallel through an OpenCL kernel for each orthoblock on the GPU with an $NDR(\langle k\rangle, \langle 1\rangle)$ splitting. The matrix multiplications (steps 3 and 5), as discussed earlier, are processed by the BLAS kernels from AMD.

The operations for partial selection (step 4 in 1ONB) were packed and implemented as a separate OpenCL kernel. The sparse signal set is kept in global memory and each work-item is in charge of doing SELECT on $\tilde{m}$ signals. Numerical experiments on our hardware pointed out that a splitting of $NDR(\langle m\rangle, \langle \tilde{m} = 256\rangle)$ gives the best performance results while keeping full GPU occupancy.

Figure 7.4 shows how resources limit the number of active wavefronts for the partial selection kernel. We can see that using a work-group size within 128 and 256 work-items, up to about 10 VGPRs and an LDS size of less than 10KB would permit the partial selection kernel to reach full utilization of the GPU. Our kernel is marked with a squared dot on the graphs from figure 7.4 and it is clearly within these limits.

Due to the small dimensions p of the block dictionaries, using the BLAS library from AMD for processing $Q = UV$ from step 5 of 1ONB for each orthobase did not cover the I/O costs. For that, we implemented a custom matrix multiplication kernel that performs the operation in parallel for the entire dictionary. And so, each work-group is in charge of computing the updated orthobase corresponding to its group-id, resulting in an $NDR(\langle k \times \tilde{m}\rangle, \langle \tilde{m}\rangle)$ splitting. Work-items within a work-group are performing vectorized vector-matrix multiplication that calculate the lines of the new orthobase corre-

Table 7.1: Kernel information and occupancy for $m = 16384$, $K = 16$ and $s = 4$

Kernel	Rep.	Select	GEMV	Energy	Limits
VGPRs	35	9	8	4	248
LDS	1024	0	0	0	32768
LWS	80	256	256	192	256
GWS	81920	16384	1280	3072	16777216
Waves	2	4	4	3	4
VGPRs	6	24	28	24	24
LDS	16	24	24	24	24
LWS	16	24	24	24	24
Occ.(%)	25	100	100	100	100

sponding to their local-id. Given that $Q \in \mathbb{R}^{p \times p}$, the rows each work-item has to compute is given by the ratio of $p/\tilde{m}$. For p dimensioned k orthobases we found that a subunitary ratio of the form $NDR(\langle k \times \tilde{m}\rangle, \langle \tilde{m} = p \times 8\rangle)$ gives full occupancy on our GPU.

Updating the energy of the newly created sparse representations (needed in step 3 of the next P-SBO iteration for building the worst represented signals set W) is implemented by partitioning the global address space with another OpenCL kernel. The representation matrix and the associated energy set are kept in global memory. Each work-item independently computes the energy for $m/\tilde{m}$ signals with no work-group cooperation resulting in an $NDR(\langle \tilde{m}\rangle, \langle any\rangle)$ split. We found that full-occupancy is reached on our hardware by using the $NDR(\langle K \times l\rangle, \langle l = 192\rangle)$ partitioning, where K is the maximum allowed number of orthobases.

Table 7.1 provides an overview of the kernels n-dimensional topology and resource utilization while performing dictionary learning with a training signals set of $m = 16384$ of size $p = 32$ each with a target sparsity $s = 4$ and $K = 16$ orthoblocks.

Each column but the last represents a kernel (representation, partial selection, custom vector-matrix multiplication and energy update, respectively). The last column shows the device limits.

The table is split in two parts. The first part shows the vector GPR usage per work-item, the LDS usage per work-group, the flattened work-group size, the flattened global work size, and the number of waves per work-group, respectively for each kernel. We can see that the representation kernel uses a lot of VGPRs in comparison with the other kernels resulting in a reduced number of waves. It is also visible that it uses the highest number of work-

Table 7.2: PAK-SVD performance for $m = 32768$, $p = 64$, $s = 8$ with $\tilde{n} = n$ and $K = 100$.

n	64	96	128	160	256
$t_{learn}(s)$	366.8	396.7	416.5	438.4	642.4
$t_{rep}(s)$	0.3467	0.3753	0.8207	0.5889	2.2436
RMSE	0.0271	0.0246	0.0242	0.0230	0.0216

Table 7.3: Parallel SBO performance for $m = 32768$, $p = 64$, $s = 8$ with $K_0 = 5$ and $R = 6$

K	8	16	24	32	64
$t_{learn}(s)$	1.8	6.7	12.3	20.9	85.4
$t_{rep}(s)$	0.0020	0.0021	0.0022	0.0021	0.0021
RMSE	0.0268	0.0245	0.0240	0.0238	0.0235

items due to the mapping strategy described earlier. The rest of the kernels require similar resources for execution, maximizing the number of waves per work-group and thus leading to full GPU occupancy.

In the lower part of the table we can see the constraint imposed on the total number of active waves by each resource utilization: VGPRs, local memory and local work size, respectively. The last entry shows the resulting percentage of GPU occupancy. While local memory and work-group size would allow for the simultaneous execution of 16 wavefronts, the VGPRs permit only 6 out of 24 thus resulting in a 25% occupancy for the representation kernel. For the rest of the kernels the constraints permit the maximum number of waves to be executed. More so, in the case of the vector-matrix kernel the reduced number of used VGPRs would allow more active waves than the device's limit.

7.4 Results and Performance

We generated our experimental data as described in Section 2.5.2 and executed our tests as specified in Section 3.4.

7.4.1 Execution Improvements

Tables 7.2 and 7.3 depict the differences in final representation error, the total time spent on dictionary learning (t_{learn}) and the time it takes to represent

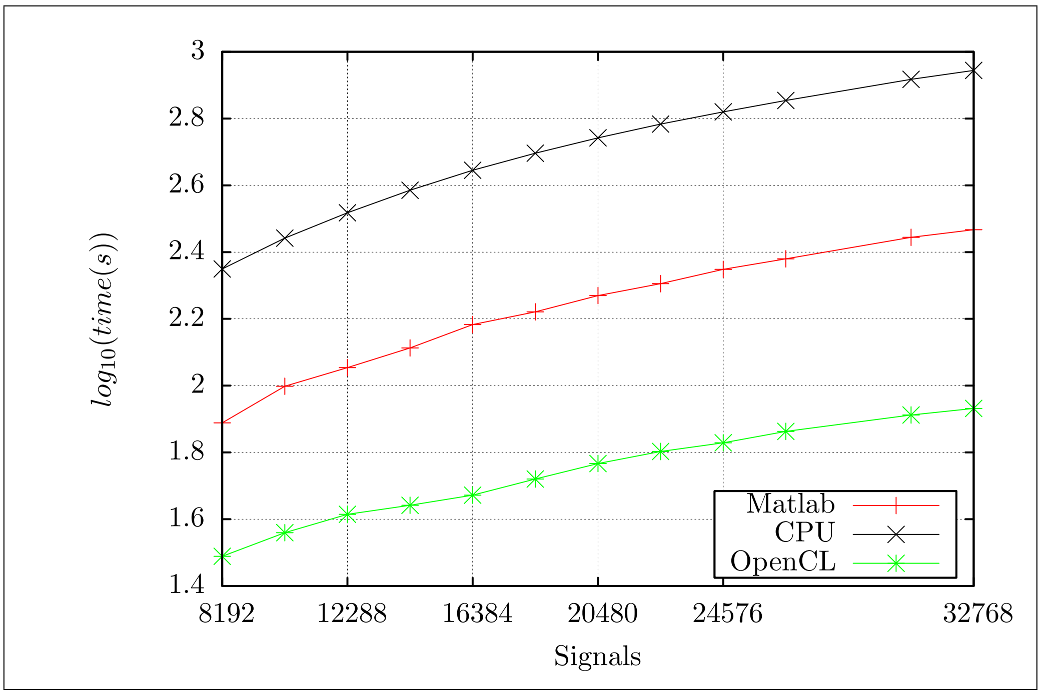

Figure 7.5: Execution times for $K = 64$, $s = 8$, $p = 64$.

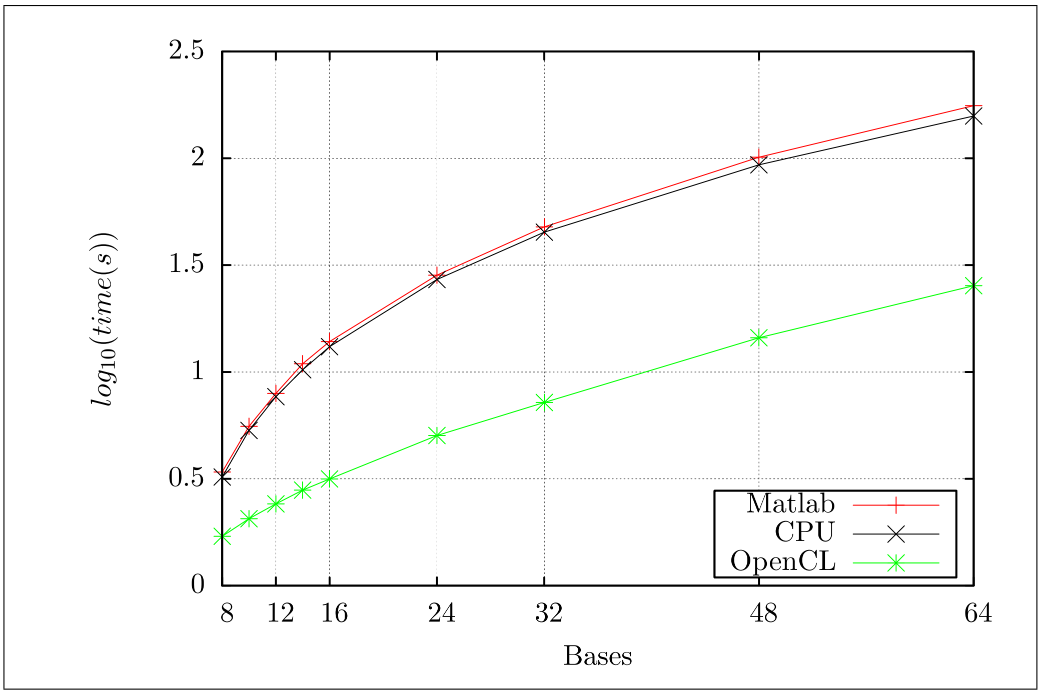

Figure 7.6: Execution times for $m = 24576$, $s = 4$, $p = 32$.

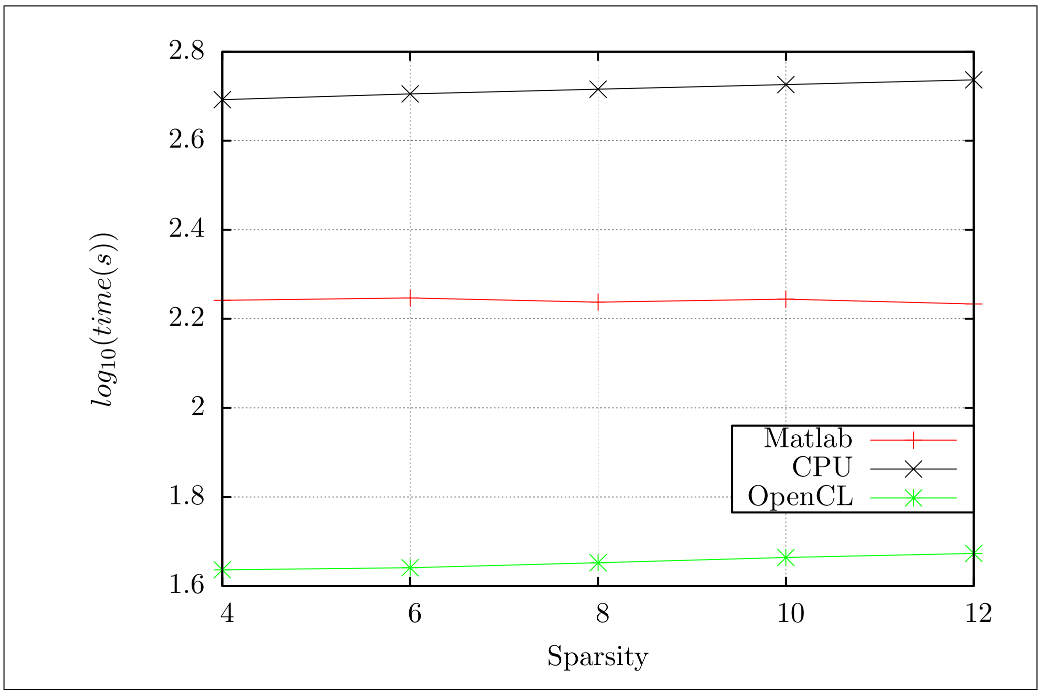

Figure 7.7: Execution times for $m = 32768$, $K = 48$, $p = 64$.

the data set with the final dictionary (t_{rep}). We vary the total number of P-SBO orthoblocks $K = \{8, 16, 32, 64\}$ and compare with PAK-SVD instances running with a dictionary of $n = \{64, 96, 128, 256\}$ atoms and $K = 100$ iterations using full parallelization during the atoms update phase ($n = \tilde{n}$) for which the numerical simulations in [52] gave the best representation error and the fastest execution times. For PAK-SVD we used the OpenCL implementation described in Section 5.3. We compare the resulting approximations by looking at the root mean square error (2.9) which we express graphically in decibels.

While PAK-SVD can produce a slightly better error than P-SBO, the time difference is significant with P-SBO being up to 203.8 times faster than PAK-SVD at dictionary learning and 1068.4 times faster at producing sparse representations. Even though P-SBO's dictionary size is larger, the total memory footprint is smaller than PAK-SVD because of OMP's high memory requirements. This issue was detailed in Section 4.3.1, where we discussed the Batch OMP parallel OpenCL implementation.

Turning our focus towards different P-SBO implementations, we see in Figure 7.5 that the OpenCL implementation gives better results than the Matlab and C counterparts. Keeping a fixed number of orthonormal bases

Table 7.4: P-SBO performance for $m = 32768$, $p = 64$, $s = 8$ with $K_0 = 5$, $R = 6$

$\tilde{K}$	W					
	8192		4096		2048	
	t(s)	RMSE	t(s)	RMSE	t(s)	RMSE
1	379	0.0222	399	0.0224	361	0.0226
2	192	0.0212	192	0.0216	187	0.0221
4	101	0.0208	98	0.0213	96	0.0217
8	57	0.0207	57	0.0213	56	0.0218
16	32	0.0206	32	0.0213	30	0.0218
32	18	0.0209	19	0.0214	18	0.0219
64	12	0.0215	12	0.0218	-	-

$K = 64$ and representing signal sets from as low as $m = 8192$ up to $m = 32768$, the parallel version performs 3.4 times faster than the Matlab implementation and 10.3 times faster than the single CPU C implementation.

Figure 7.6 describes the performance results with a fixed signal set of $m = 24576$ and a variable dictionary size starting from $K = 8$ orthoblocks up to $K = 64$. Again we can see that the OpenCL version performs a lot better than the other implementations, giving speed-ups up to 7 times.

Looking at Figure 7.7 we see that the target sparsity s does not really affect running times. We kept a fixed signal set $m = 32768$ and a fixed dictionary of $K = 48$, and we varied the sparsity from $s = 4$ to $s = 12$ on a fixed signal dimension of $p = 64$.

7.4.2 Training Multiple $\tilde{K}$ Bases

The representation error and execution improvement of P-SBO over SBO is depicted in Table 7.4 where we varied the value of $\tilde{K}$ in the Matlab implementation of P-SBO starting from $\tilde{K} = 1$ to $\tilde{K} = 64$. We used an identical training signals set of $m = 32768$ items of size $p = 64$ each with a sparsity constraint of $s = 8$ and $R = 6$ 1ONB training rounds. The representation error is improving as $\tilde{K}$ grows until it reaches a point where the training set is too small for properly training an orthobase and so the error starts to slightly depreciate. The result is consistent with different sizes of the worst reconstruction set W.

Figure 7.8 shows the error evolution of the P-SBO algorithm as new bases are trained and added to the union of ONBs for different values of $\tilde{K}$. We can see that the representation error improves and drops a lot faster as we increase the number of orthobases trained at step 3 in algorithm 14.

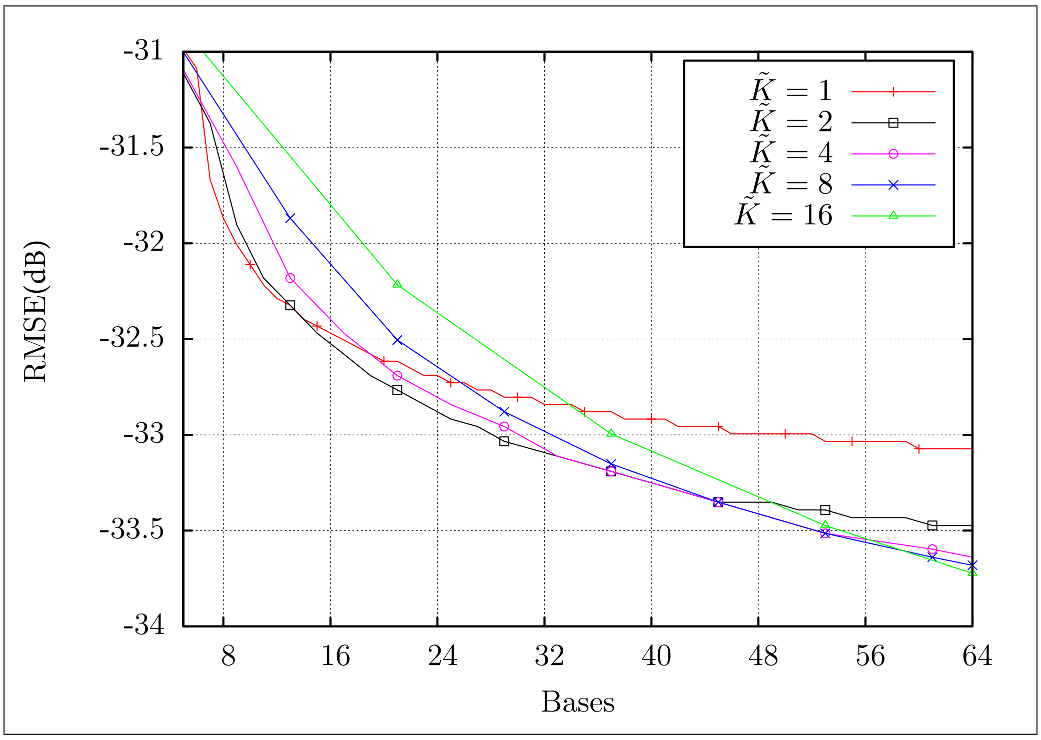

Figure 7.8: P-SBO error evolution for $m = 32768$, $p = 64$, $W = 8192$, $s = 8$, $K_0 = 5$, $R = 6$.

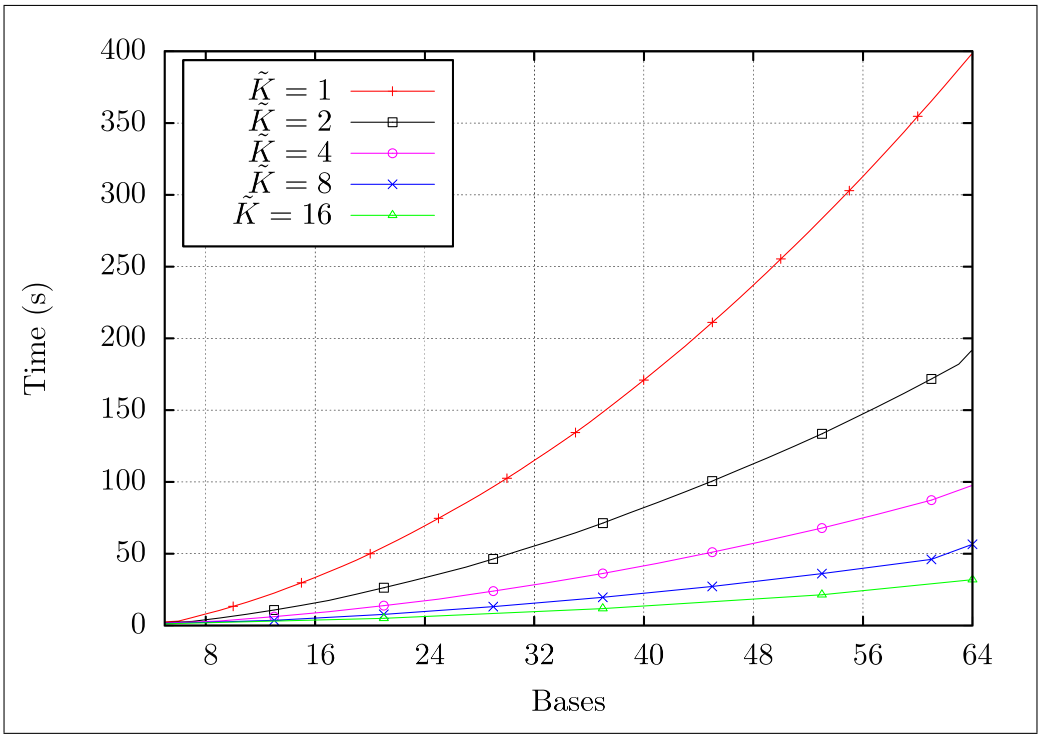

Figure 7.9: P-SBO execution times for $m = 32768$, $p = 64$, $W = 4096$, $s = 8$, $K_0 = 5$, $R = 6$.

As the number of orthobases trained for the worst-reconstructed signals set W increases the number of training iterations (P-SBO steps 3–6) shrinks resulting in faster execution times. This improvement is depicted in Figure 7.9 where we show the elapsed time after each training stage when running P-SBO with the same inputs and the same constraints but with different values of $\tilde{K}$.

7.5 Conclusions

We provided an improved algorithm that reduces the representation error and cuts the execution time and we also proposed an efficient parallel implementation of the P-SBO algorithm. Dictionary updates are performed by refining each of the orthonormal bases concurrently. Also, we completely parallelized, in a map-reduce manner, the pursuit of finding the single best orthobase for representing a given signal. Our implementation was done in OpenCL and tested on the GPU.

Our parallel version achieves a good trade-off between algorithm complexity and data-set approximations compared to PAK-SVD due to the different representation approach and the low-memory footprint of P-SBO's representation strategy leading to better GPU occupancy confirmed in our numerical results that show a speed-up of about 200 times for dictionary learning while providing an improved representation quality. Despite its much larger dictionary size, P-SBO has a significantly lower representation time (simulations show about 1000 times speed improvement), which makes it appealing for real time applications. Also, simulations showed that P-SBO can perform about 33 times faster on the same data than SBO while also providing an improved dictionary resulting in better sparse representations.

Chapter 8

Cosparse Learning

8.1 Introduction

So far in this thesis we have focused on the dictionary learning process of Equation (2.8), where we are interested in the few non-zero entries of the approximations. This process is also called in the literature the synthesis-based sparse representation model [27]. Recent years have shown approximation improvements when instead we analyze the set of atoms that do not participate in representing a signal. This process is also called the cosparse analysis model and is described in detail in [55].

In the cosparse or analysis model, the overcomplete dictionary is $\Omega \in \mathbb{R}^{n \times p}$, with $n > p$, and the atoms are the rows of the dictionary. For a given signal y, the representation, now denoted $z \in \mathbb{R}^p$, is orthogonal on a set $\mathcal{I}$ of $n-s$ atoms (named cosupport), so again it lies in an s-dimensional subspace, and the representation problem is

$$\begin{aligned} \underset{z,\mathcal{I}}{\text{minimize}} \quad & \|y - z\|_2^2 \\ \text{subject to} \quad & \Omega_{\mathcal{I}} z = 0 \\ & \text{rank}(\Omega_{\mathcal{I}}) = n - s, \end{aligned} \tag{8.1}$$

where $\Omega_{\mathcal{I}}$ contains the rows of Ω with indices in $\mathcal{I}$.

Here, we propose a new dictionary training algorithm for the orthogonal case (described in Chapter 7), inspired from the cosparse DL method from [56]. The combination of techniques from both the sparse and cosparse approaches is the key to better representations and is possible due to the special characteristics induced by orthogonality.

The chapter is structured as follows: Section 8.2 describes our new cosparse orthonormal block training algorithm and its relation to the synthesis ver-

Algorithm 15: 1ONB-COSP

```
Data:   signals set $Y$ and target sparsity $s$
Result: dictionary $Q$ and sparse representations $X$
1 Initialization: Let $Q = U$ where $U\Sigma V^T = \text{SVD}(Y)$
2 Compute $X = Q^T Y$ and select the largest $s$ entries of each column
3 foreach atom $i$ in dictionary $Q$ do
4     Extract: $\mathcal{I} = \{j | X_{i,j} = 0\}$
5     Refine: solve (8.2) to get $q_i$
6     Update: $X = Q^T Y$ and select the largest $s$ entries of each column
7     Restructure: apply Procrustes approximation (7.6) on $Y$ and $X$
      to orthogonalize $Q$
```

sion, followed by numerical results supporting its dictionary recovery and representation error improvements in Section 8.3.

8.2 Cosparse Orthonormal Block Training

We start with the simple remark that the sparse (4.1) and cosparse (8.1) representation problems have the same optimal error if the dictionary is orthogonal. Indeed, given D orthogonal, the problem (4.1) is solved by computing $x = D^T y$ and keeping only the largest (in absolute value) s elements, the others being forced to zero. This holds because the objective of (4.1) is equal to $\|D^T y - x\|_2^2$. For the cosparse problem (8.1), the atoms are now rows instead of columns, so the dictionary is $\Omega = D^T$. The two problems are connected via the relation $D^T z = x$. The $n - s$ atoms that are orthogonal on z are those corresponding to the positions of zeros in x. Otherwise said, the problem (8.1) is solved by computing $D^T y$ and setting to zero the $n - s$ smallest elements (in absolute value). We work on complementary subspaces, but the final result is the same.

8.2.1 Building Cosparse Orthonormal Blocks

UONB [33] and especially SBO [34] use 1ONB (Algorithm 13 from Chapter 7) to build one orthonormal block. Using the idea behind 1ONB and concepts inspired from cosparse DL, we propose a new method for training an orthogonal block, described in Algorithm 15. Since the sparse and cosparse models are interchangeable in the orthogonal case, as explained above, we adopt an idea used for atom update in the cosparse K-SVD algorithm [56].

Algorithm 16: 1ONB-COSP+

```
Data:   signals set $Y$, sparsity $s$, rounds $R$
Result: dictionary $Q$ and sparse representations $X$
1 $\{Q, X\} = $ 1ONB-COSP$(Y, s)$
2 for $r \leftarrow 1$ to $R$ do
3     Update: $X = Q^T Y$ and select the largest $s$ entries of each column
4     Approximation: apply Procrustes orthogonalization (7.6) on $Y$
      and $X$ to approximate $Q$
```

Denoting Q the orthogonal dictionary, an atom q_i is optimized by solving the problem

$$\begin{aligned} \underset{q_i}{\text{minimize}} \quad & \|q_i^T Y_{\mathcal{I}}\|_2^2 \\ \text{subject to} \quad & \|q_i\|_2 = 1, \end{aligned} \tag{8.2}$$

where $\mathcal{I}$ is the set of signals that do not use the atom q_i in their representation (or, taking the cosparse view, on which q_i should ideally be orthogonal). The solution of (8.2) is the singular vector corresponding to the smallest singular value of $Y_{\mathcal{I}}$. (Note the duality with sparse K-SVD, where the singular vector of the largest singular value was involved.)

We give now a step-by-step description of Algorithm 15, named 1ONB-COSP in the sequel. The initialization of the orthoblock Q in step 1 and the computation of the sparse representations X in step 2 are done the same way as described in Algorithm 13. Following the general approach for atom optimization in DL, we sequentially update each atom q_i from Q in the loop from step 3, using the atom refinement solution described in (8.2): first we extract the signals that are not using the current atom i in their representation (step 4) and then we proceed to refine q_i in step 5 by solving (8.2). We found that updating the representations immediately after the change in atom q_i significantly improves the final representation error. So, in step 6 we create new representations with the updated dictionary Q the same way we did in step 2. Note that, with the replacement of q_i, the dictionary Q is no longer orthogonal and it is important to use it unstructured when updating the representations in step 6. We restructure Q as an orthogonal matrix right before proceeding to the next atom update by applying (7.6) in step 7. Numerical simulations showed that it is better to use the old representations built in step 6 in the next atom update iteration instead of computing new ones with the restructured dictionary from step 7, although this may be counterintuitive.

Updating each individual atom at a time (step 3) shows an increased complexity of 1ONB-COSP when compared to 1ONB that updates the entire dictionary at once. This might partly explain why 1ONB needs a few refinement rounds (e.g. $R = 5$ or $R = 6$ in step 2) until error improvement stalls [34], whereas for 1ONB-COSP our simulations showed that a single refinement of each atom is enough and repeating the dictionary training steps 3–7 does not improve the final quality of the orthoblock Q.

We also found that further error improvement appears if, as described in Algorithm 16, 1ONB-COSP (step 1) is followed by R 1ONB training rounds (steps 2–4). We denote 1ONB-COSP+ this succession of algorithms. On the contrary, running 1ONB first and then performing 1ONB-COSP did not show any improvement in the end result.

8.3 Results and Performance

We present numerical results indicating the quality improvements when using the cosparse approach. First we show the benefits of 1ONB-COSP and 1ONB-COSP+ for the dictionary recovery and the sparse image representation problems when using a single orthogonal block. Then, we present its impact on the algorithms that make use of it for DL, when the dictionary is a union of orthogonal blocks. We always used identical input and parametrization (where applicable) when comparing methods.

8.3.1 Dictionary Recovery

The dictionary recovery experiment (see Section 2.5.1) had to be adapted to the dimensions of the orthogonal dictionary. We started with a random square matrix of dimension $p = 20$ on which we ran the SVD decomposition and used the left orthogonal transformation matrix as the original dictionary. We then generated a data set Y of $m = 600$ columns, each obtained as a linear combination of $s \in \{3, 4, 5\}$ randomly chosen atoms. We ran 1ONB and 1ONB-COSP+ with $R = 5$ rounds (enough to converge as described in [34]) on the new signal set and compared the original dictionary with the learned dictionary. The algorithms were provided with the original sparsity level s that was used in generating the clean data set Y. Table 8.1 shows a big improvement in the percentages of recovered atoms, averaged over 50 runs. 1ONB-COSP is vastly superior to 1ONB. 1ONB-COSP+ improves the results where there is room for improvement, especially for larger s.

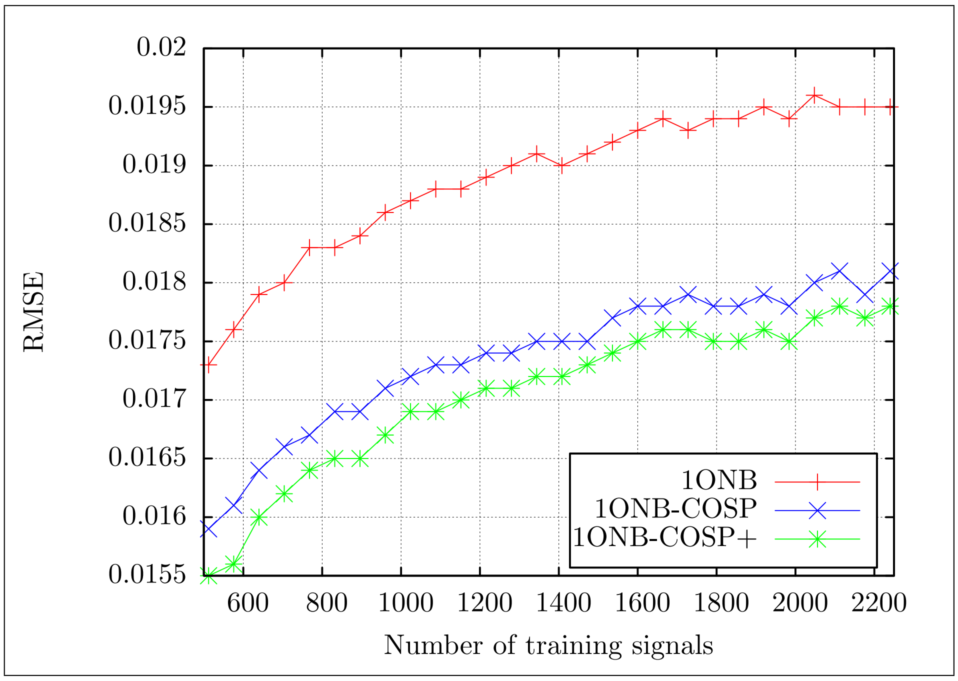

Figure 8.1: Error evolution for sparse and cosparse algorithms.

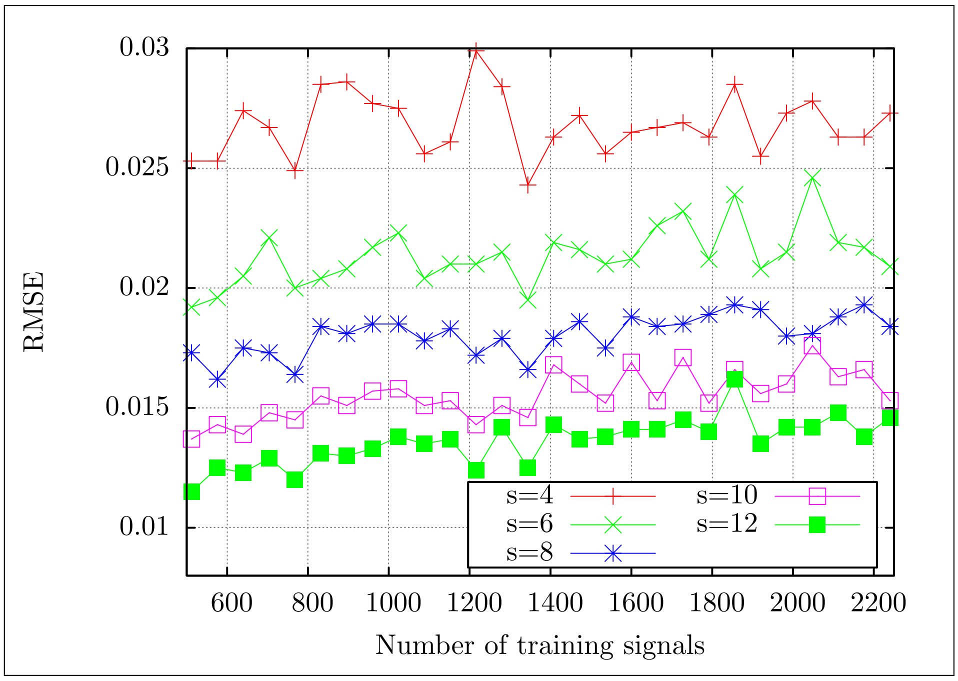

Figure 8.2: Error evolution for different sparsity constraints.

Table 8.1: Percentage of recovered atoms

s	Method	SNR			
		10	20	30	∞
3	1ONB	46.7	53.5	57.4	53.8
	1ONB-COSP	99.9	100.0	100.0	91.9
	1ONB-COSP+	100.0	100.0	100.0	99.4
4	1ONB	15.5	30.9	28.9	28.8
	1ONB-COSP	96.7	99.1	98.8	89.8
	1ONB-COSP+	98.2	99.8	99.4	97.8
5	1ONB	2.3	9.1	12.6	11.1
	1ONB-COSP	85.4	91.5	95.8	90.2
	1ONB-COSP+	91.5	95.2	98.0	95.8

8.3.2 Dictionary Learning

We generated our experimental data as described in Section 2.5.2.

In Figure 8.1 we present the average representation error over 100 runs for varying signals set sizes when using orthogonal blocks of dimension $p = 64$ ($n = 64$ atoms) with a sparsity constraint of $s = 8$. We used $R = 5$ rounds for 1ONB and 1ONB-COSP+. Both cosparse methods are consistent in providing a better dictionary than plain 1ONB. More so, at the cost of an increase in execution time, 1ONB-COSP+ performs better than 1ONB-COSP.

Table 8.2 shows the final errors after running a single round of tests on $p = 64$ sized dictionary blocks with varied sparsity constraints $s \in \{4, 6, 8, 10, 12\}$ and different training set sizes ($m \in \{512, 1024, 1536, 2048\}$) for each sparsity level. Except for two results ($s = 12$), 1ONB-COSP presents an improvement in approximation error over 1ONB, while 1ONB-COSP+ always outperforms both methods.

Figure 8.2 shows the sparsity impact on the representations obtained with Algorithm 15. Using the same dimensions for the input data as the ones used in Figure 8.1 we changed the sparsity constraint from $s = 4$ up to $s = 12$ for 1ONB-COSP while keeping the same training signal set. We can see a natural increase in error as the number of signals in the data set grows. Also visible is the clear difference in representation quality as the sparsity constraint is loosened.

Table 8.2: Final errors for sparse and cosparse algorithms

s	m	1ONB	COSP	COSP+
4	512	0.0288	0.0280	0.0268
	1024	0.0292	0.0283	0.0275
	1536	0.0293	0.0278	0.0276
	2048	0.0304	0.0295	0.0293
6	512	0.0239	0.0236	0.0217
	1024	0.0243	0.0231	0.0230
	1536	0.0245	0.0243	0.0230
	2048	0.0257	0.0244	0.0242
8	512	0.0204	0.0185	0.0183
	1024	0.0211	0.0198	0.0189
	1536	0.0209	0.0196	0.0192
	2048	0.0224	0.0223	0.0205
10	512	0.0176	0.0158	0.0155
	1024	0.0183	0.0171	0.0167
	1536	0.0183	0.0171	0.0168
	2048	0.0195	0.0191	0.0184
12	512	0.0152	0.0151	0.0139
	1024	0.0159	0.0173	0.0154
	1536	0.0160	0.0163	0.0153
	2048	0.0170	0.0165	0.0160

8.3.3 Unions of Orthonormal Bases with Cosparse Training

Algorithms that train overcomplete dictionaries as a union of orthonormal bases (such as UONB [33] and SBO [34]) use 1ONB to build one orthonormal block. This type of methods were discussed in Chapter 7.

To show how our cosparse approach behaves we substitute 1ONB with 1ONB-COSP or 1ONB-COSP+ in the dictionary initialization and update stage of SBO and UONB, without any other algorithmic modifications. Our goal here is to improve the performance of SBO and UONB, for a comparison with generic DL methods such as AK-SVD we direct the reader to the numeric simulations from [34].

Following the comparison tests from [34] we used $M = 3$ orthobases for UONB and $M = 16$ for SBO (this makes representation speed similar for the two methods). The signals have size $p = 64$, being generated from images

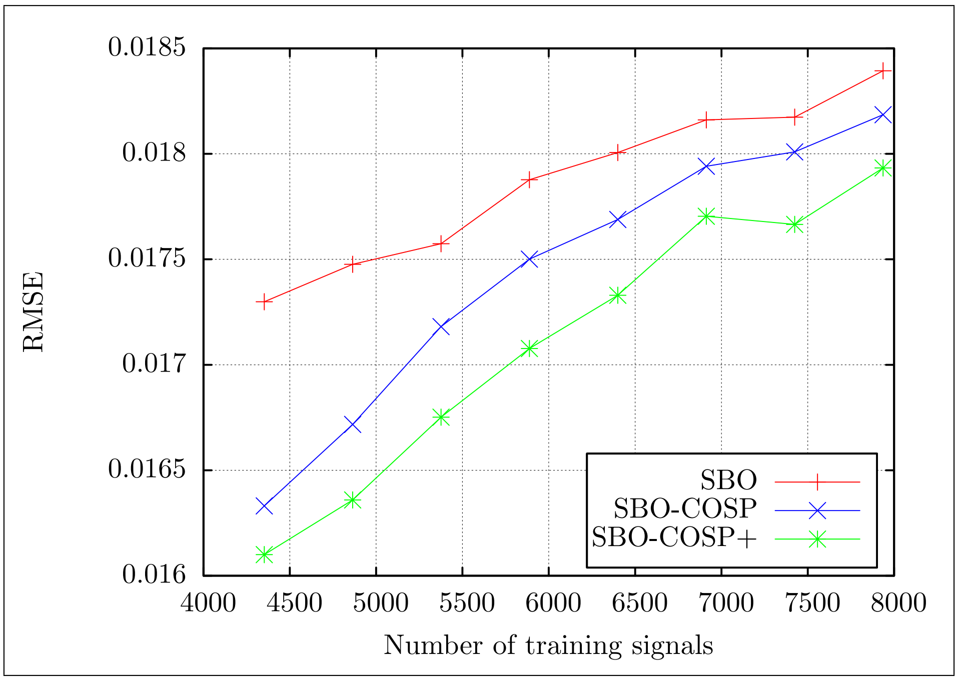

Figure 8.3: Representation error comparison of SBO variants.

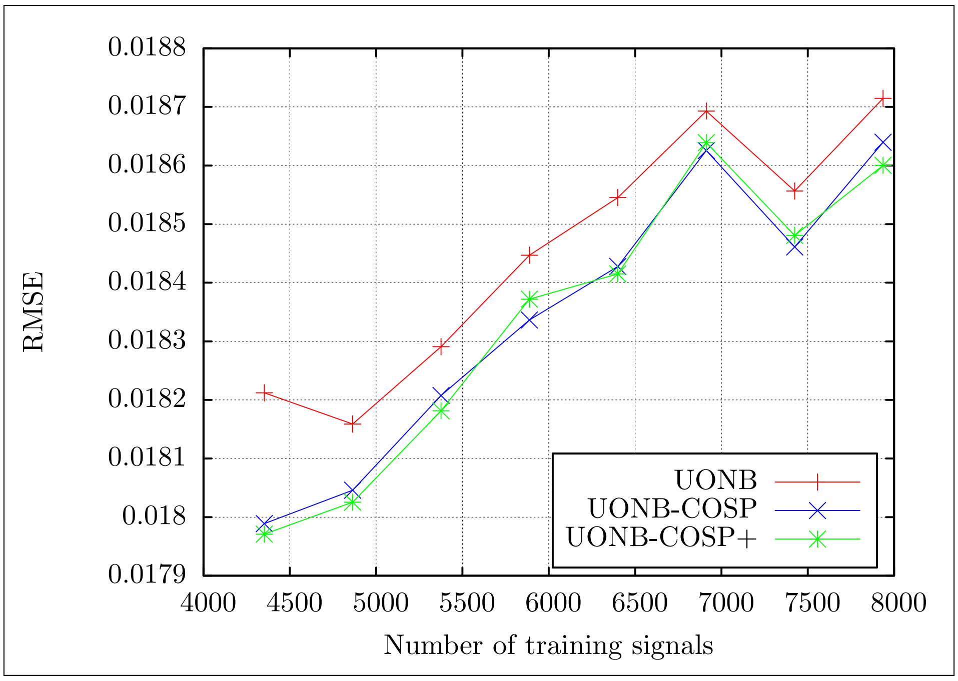

Figure 8.4: Representation error comparison of UONB variants.

as described in Section 2.5.2. We impose a sparsity constraint $s = 10$ and a number of $R = 5$ training rounds for 1ONB and 1ONB-COSP+. We ran multiple tests on varying signals set sizes from $m = 4096$ to $m = 8192$. We plot the results in figures 8.3 and 8.4.

Figure 8.3 shows the average representation error over 10 runs when performing SBO with 1ONB, 1ONB-COSP and 1ONB-COSP+. Because SBO makes use of 1ONB training during initialization and also during the main iterations, the approximation improvement is consistent with the results seen in Figure 8.1.

Using the same average as described in Figure 8.3, we show the performance of UONB with all three 1ONB variants in Figure 8.4. Because 1ONB is only used at initialization the cosparse variants have less impact on the overall performance of UONB.

8.4 Conclusions

In this chapter we have presented a new algorithm for learning orthogonal dictionary blocks in a cosparse fashion. The new algorithm shows significant improvements at recovering dictionary atoms and provides a smaller representation error when tested on synthetic and empirical data. We also show that the improvement in representation holds when applying the cosparse algorithms within existing methods that create the dictionary as a union orthonormal bases.

Chapter 9

Composite Dictionaries

9.1 Introduction

This chapter studies dictionary learning for signal denoising when using a special class of dictionaries called composite dictionaries. Given a set of signals Y that has been perturbed by a standard white gaussian noise Z, with noise level σ, and denoting with Y^c the unknown clean signals set that we want to recover, we have

$$Y = Y^c + \sigma Z. \tag{9.1}$$

If we rewrite the approximation from (5.1) as

$$Y = DX + R, \tag{9.2}$$

where R is the residual denoted as the error matrix E in (5.1), the goal of a DL denoising algorithm is to model Y^c such that the residual R from (9.2) matches the added noise:

$$Y^c \approx DX, R \approx \sigma Z. \tag{9.3}$$

There are two approaches when choosing the training set for denoising with DL. The first one uses an external signals set [57] that results in a versatile set of representation atoms making the dictionary suited for a larger class of signals with the risk of running in to cases where none of the atoms are able to properly fit a specific model. The second approach focusses on training the dictionary with the same internal set [58] (or similar sets) of signals as the ones we want to denoise. This has the advantage of giving sharper results as the atoms are more specialized with the downside of a rather poor performance at high noise levels where the noise can become a participating part of the modeling atoms [59].

A recent trend in the DL community [60] [61] is taking advantage of both worlds by creating a dictionary following each training approach and merging the two into a larger dictionary with which the actual denoising is performed. First a dictionary is obtained from an external training set, using an algorithm such as K-SVD [29], and then for each new signal set an extra dedicated dictionary is trained and composed with the former [61]. Then, the new signal set is sparsely represented by selecting atoms from both dictionaries through algorithms such as orthogonal matching pursuit (OMP) [62]. These composite dictionaries have been shown to outperform the vanilla approach [61]. The disadvantage with this way of doing things is the loss of generality of the DL algorithms due to the fact that the composite representations are now tightly coupled to the second specialized dictionary. This is not a problem for applications such as image denoising but would be, for example, suboptimal for compression.

Here, we study the composite approach when applied to the class of dictionaries structured as a union of orthonormal basis (as described in Chapter 7) and provide new algorithms based on SBO [34] that improve the quality of the denoised signals while also providing smaller execution times.

The chapter is structured as follows: in Section 9.2 we present the strategy of applying dictionary composition to structured dictionaries and explain the resulting algorithms, while in Section 9.3 we provide numeric simulations to support our results, with conclusions in Section 9.4.

9.2 Composite Structured Dictionaries

As described in Chapter 7, SBO performs dictionary refinement independently on each block through 1ONB [33]. Even though SBO has larger memory requirements due to an increased dictionary size, its advantage over AK-SVD is the training and represntation speed while maintaining a competitive approximation quality. To give a general impression of the differences between the two we refer the reader to Table 7.2 and Table 7.3.

9.2.1 Composing SBO with 1ONB

Our first proposal for denoising with composite structured dictionaries is to mix an SBO trained external dictionary with a specific orthoblock trained with 1ONB on the noisy set. We term this method SBO-C1 and describe it in Algorithm 17.

We first use SBO to train the external dictionary E on the training set Y^t and 1ONB to train the internal dictionary F on the noisy set Y (steps 1 and

Algorithm 17: SBO-C1

```
Data:   training signals set $Y^t$, noisy signals $Y$
Result: denoised signals $Y^c$
1 Train generic dictionary $E = \text{SBO}(Y^t)$
2 Train orthoblock with noisy set: $F = \text{1ONB}(Y)$
3 Assign each column $k$ from $Y$ to one orthobase $Q_E^{(k)}$ from $E$ following
  (7.5)
4 foreach signal k from Y do
5     Compose: $D^c = [Q_E^{(k)} F]$
6     Represent: $X_k = \text{OMP}(D^c, Y_k)$
7     Denoise: $Y_k^c = D^c X_k$
```

2). Next we allocate each signal from Y to a block from E (step 3). We denote a column k from matrix Y with Y_k and the SBO base it has been assigned to with $Q_E^{(k)}$. The focal point in the algorithm is composing a dedicated dictionary D^c for each signal made out of its assigned block $Q_E^{(k)}$ and the internal orthoblock F (step 5). With the composite dictionary we proceed to compute the sparse representations X_k using the OMP algorithm in step 6. The factorization of the composite dictionary and the representations from step 7 provides us with the denoised signal Y_k^c as described around Equation (9.3).

The simplistic 1ONB training of the internal dictionary leads to mediocre denoising results (as described in Section 9.3), but it is worth mentioning that it manages to perform better than the composite AK-SVD algorithm when the sparsity constraint is loosened such that $s > \sqrt{p}$, where p is the signal size. In terms of speed, it offers the fastest dictionary training phase (steps 1 and 2), but representation and denoising (steps 6 and 7) take just as long as it would with an identically sized dictionary trained with plain or composite AK-SVD.

9.2.2 Composite SBO

A natural step towards improving the representation quality of SBO-C1 is to expand the internal dictionary to more than one orthoblock. This can be achieved by performing another SBO session, this time on the noisy set, in order to create an extended internal dictionary that is still smaller than the external one. We keep the internal dictionary small in order to avoid modelling the noise as described in the introduction. Our experiments have

Algorithm 18: SBO-C

```
Data:   training signals set $Y^t$, noisy signals $Y$
Result: denoised signals $Y^c$
1 Train external dictionary $E = \mathrm{SBO}(Y^t)$
2 Train internal dictionary $F = \mathrm{SBO}(Y)$
3 Assign each column $k$ from $Y$ to one orthobase $Q_E^{(k)}$ from $E$ and one
  orthobase $Q_F^{(k)}$ from $F$ following (7.5)
4 foreach signal k from Y do
5     Compose: $D^c = [Q_E^{(k)} Q_F^{(k)}]$
6     Represent: $X_k = \mathrm{OMP}(D^c, Y_k)$
7     Denoise: $Y_k^c = D^c X_k$
```

shown good results with keeping a size of half the number of bases from the external dictionary. This algorithm is a direct corespondent of the composite AK-SVD method. We call it SBO-C and describe it in Algorithm 18.

In the new algorithm, step 2 is modified in order to train an internal SBO dictionary instead of a single 1ONB base. This also affects step 3 where an extra assignment operation needs to be performed for the internal dictionary. We denote with $Q_F^{(k)}$ the base from the internal dictionary assigned to signal k from the noisy set Y. The composed dictionary (step 5) maintains its size but its SBO internal dictionary component leads to a more specialized orthobase $Q_F^{(k)}$ and provides sharper results. Representation and denoising (steps 6–7) do not take advantage of the composite structure of the dictionary and so they perform the same as they would with plain or composite AK-SVD.

Even though the training stage (steps 1–3) has an increased complexity due to the second SBO training round, it is still a lot faster than plain and composite AK-SVD which suffer from large execution times as shown in Table 7.2.

9.2.3 Hybrids

In our pursuit of improving denoising performance, we also studied the case of hybrid dictionary compositions between AK-SVD and SBO.

One option is to use SBO as the external dictionary and train an AK-SVD block instead of the 1ONB base in Algorithm 17. The rest of SBO-C1 remains the same with the observation that the composite dictionary is now done with F learned from AK-SVD in step 5. We found that preserving the block size for the AK-SVD dictionary is enough to outperform SBO-C1.

Algorithm 19: Composite AK-SVD with SBO

```
   Data:   training signals set $Y^t$, noisy signals $Y$
   Result: denoised signals $Y^c$
1  Train external dictionary $E = \text{AK-SVD}(Y^t)$
2  Train internal dictionary $F = \text{SBO}(Y)$
3  Assign each column $k$ from $Y$ to one orthobase $Q_F^{(k)}$ from $F$ following
   (7.5)
4  foreach signal $k$ from $Y$ do
5      Compose: $D^c = [E\; Q_F^{(k)}]$
6      Represent: $X_k = \text{OMP}(D^c, Y_k)$
7      Denoise: $Y_k^c = D^c X_k$
```

This also helps prevent noise modelling and keeps a minimal execution time. The effects of increasing the dictionary size can be observed in the first line of Table 7.2.

Reversing the roles, we can train AK-SVD on the external data set and then use SBO on the internal noisy sets as described in Algorithm 19.

This way the expensive AK-SVD training is performed on the large training set (step 1) and SBO on the small noisy set (step 2). The orthobase assignment needs to be performed only on the internal dictionary in step 3 resulting in a specialized orthobase $Q_F^{(k)}$ as described in Section 9.2.2. The composed dictionary is then made of the AK-SVD external dictionary and the assigned internal orthobase (step 5). Representation (step 6) and denoising (step 7) perform the same steps as former algorithms and have an identical execution cost as plain or composite AK-SVD.

Algorithm 19 would outperform a composite AK-SVD solution in an online denoising scenario. External training, representation and denoising would be identical, but the internal training for the hybrid would perform much faster in production, with an efficient SBO implementation handling incoming noisy data instead of another AK-SVD instance.

9.3 Results and Performance

Our denoising experiments were performed on images from the USC-SIPI [35] database. For the training set we used multiple grayscale images normalized and organized into 8×8 random patches. The noisy set was built in the same way from a different image to which we added white gaussian noise in

Table 9.1: Denoising with $n = 64 + 64$ and $s = 4$

N	SBO	SBO-C1	SBO-C	AK	AK-C	S-AK	AK-S
10	0.0514	0.0498	0.0481	0.0473	**0.0462**	0.0465	0.0467
20	0.0490	0.0473	0.0439	0.0452	0.0441	0.0435	**0.0427**
30	0.0488	0.0468	0.0434	0.0449	0.0439	0.0434	**0.0421**
40	0.0487	0.0468	0.0434	0.0450	0.0439	0.0433	**0.0423**
50	0.0491	0.0468	0.0435	0.0450	0.0438	0.0433	**0.0422**

Table 9.2: Denoising with $n = 64 + 64$ and $s = 8$

N	SBO	SBO-C1	SBO-C	AK	AK-C	S-AK	AK-S
10	0.0441	0.0421	0.0412	0.0401	**0.0395**	0.0403	0.0402
20	0.0392	0.0363	0.0339	0.0357	0.0355	0.0341	**0.0329**
30	0.0389	0.0358	0.0333	0.0352	0.0350	0.0335	**0.0322**
40	0.0388	0.0356	0.0330	0.0350	0.0350	0.0334	**0.0322**
50	0.0386	0.0355	0.0330	0.0353	0.0351	0.0335	**0.0321**

order to obtain the desired singnal to noise ratio (SNR). Denoising performance is expressed as the RMSE (2.9) between the clean image Y^c and the DX factorization result of steps 7 from the algorithms in Section 9.2 (see description around Equation (9.3)).

We shorten AK-SVD with AK, composite AK-SVD with AK-C and the hybrid variants with S-AK where we used SBO as the external dictionary and AK-SVD as the internal one and vice-versa as AK-S.

9.3.1 Composite Dictionaries with $n = 64 + 64$

The first series of experiments constraints each composite dictionary to a fixed size of $n = 128$ with 64 atoms from the external dictionary and another 64 atoms from the internal dictionary.

In Tables 9.1–9.3 we present the results of denoising at $s = \{4, 8, 12\}$ sparsity constraints. We used a set of $m = 8192$ signals randomly picked from 9 images for training the external dictionaries. The internal set was built from a single noisy image at different $N = \{10, 20, 30, 40, 50\}$ dB SNR levels. SBO was used with $K = 16$ bases when training external dictionaries and $K = 8$ when training internal dictionaries. 1ONB was always ran for $R = 5$ rounds as that is sufficient according to [34]. AK-SVD trained external

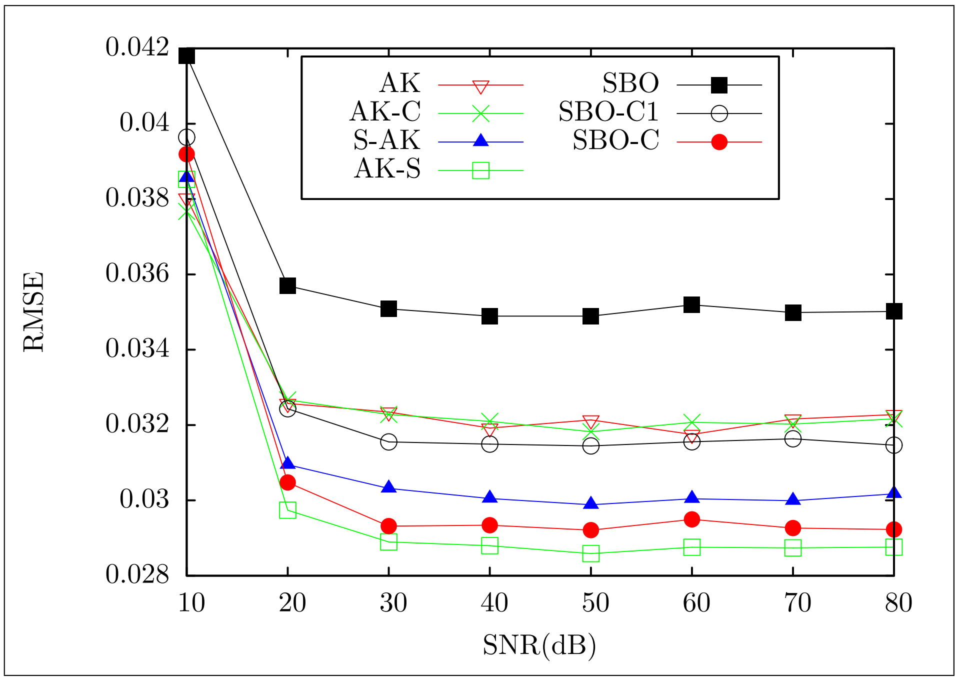

Figure 9.1: Denoising with $n = 64 + 64$ and $s = 10$.

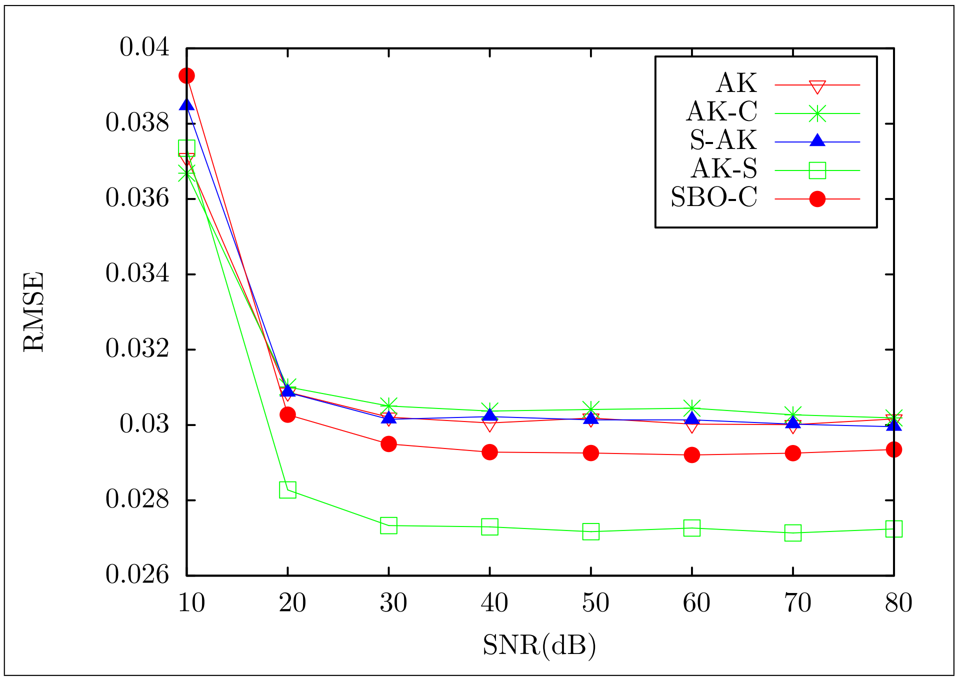

Figure 9.2: Denoising with $n = 128 + 64$ and $s = 10$.

Table 9.3: Denoising with $n = 64 + 64$ and $s = 12$

N	SBO	SBO-C1	SBO-C	AK	AK-C	S-AK	AK-S
10	0.0400	0.0380	0.0377	0.0364	**0.0361**	0.0372	0.0371
20	0.0328	0.0291	0.0275	0.0303	0.0303	0.0284	**0.0272**
30	0.0319	0.0281	0.0262	0.0293	0.0297	0.0275	**0.0258**
40	0.0318	0.0279	0.0261	0.0293	0.0295	0.0273	**0.0258**
50	0.0317	0.0280	0.0260	0.0294	0.0297	0.0271	**0.0259**

and internal dictionaries with $n = 64$ atoms and $n = 128$ for the classic non-composite version. We followed the dictionary sizes used to compare the two methods in [34].

As expected, when the sparsity constraint is loosened the denoising performance improves. The tables also show that the methods have a consistent performance across sparsity constraints and noise levels. The hybrid version AK-S is the winner in most cases except for the very noisy one where composite AK-SVD takes the lead. Even then we can see that the difference is not significant and both hybrid versions come in close in second place. If, on the other hand, speed requirements might justify a small performance compromise SBO-C is the best choice due to its small execution time at less than 3% performance loss when compared to AK-S. SBO-C is also more than 60x faster than AK-S as can be seen from tables 7.2 and 7.3. At higher sparsity targets, such as the ones from Tables 9.2 and 9.3, even SBO-C1 might provide a good option being the fastest method with a 10% performance penalty. Another interesting observation is that, except for the 10dB case, SBO-C is outperforming composite AK-SVD.

In Figure 9.1 we show the denoising performance evolution of each method as the SNR drops. For this experiment we used the same data as that from the tables above and changed the sparsity to be $s = 10$. AK-S is a clear winner while plain SBO is the poorest denoiser. AK-SVD, AK-C and SBO-C1 are somewhere close in the middle presenting similar performances. Naturally, SBO-C outperforms the hybrid S-AK due to a larger internal dictionary.

9.3.2 Composite Dictionaries with $n = 128 + 64$

We present here a second experiment where we increased the size of the external dictionaries to $n = 128$ and maintained the internals at $n = 64$ leading to a fixed composite dictionary size of $n = 192$. This experiment was aimed at comparing the plain and composite AK-SVD variants with our

Table 9.4: Denoising with $n = 128 + 64$ dictionaries

s	N	SBO-C	AK	AK-C	S-AK	AK-S
4	10	0.0480	0.0462	**0.0454**	0.0466	0.0455
	20	0.0440	0.0439	0.0431	0.0436	**0.0413**
	30	0.0435	0.0437	0.0429	0.0433	**0.0410**
8	10	0.0412	0.0388	**0.0385**	0.0403	0.0391
	20	0.0339	0.0343	0.0341	0.0342	**0.0316**
	30	0.0331	0.0337	0.0336	0.0335	**0.0309**
12	10	0.0378	0.0355	**0.0354**	0.0371	0.0360
	20	0.0273	0.0282	0.0286	0.0282	**0.0257**
	30	0.0262	0.0273	0.0279	0.0273	**0.0244**

hybrid proposals. We used the same data as in our first experiment, the only change here is the dictionary size. For reference we also ran SBO-C on the same input data even though its composite dictionary size is limited to $n = 128$.

Table 9.4 is structured the same way as Tables 9.1–9.3 with a reduced number of noise levels $N = \{10, 20, 30\}$dB because higher SNRs showed identical results to $N = 30$dB. As we can see AK-C is still the best performer in the worst case scenario, but AK-S is coming close in second place. The rest of the cases present AK-S as the clear winner at all sparsity levels. It is interesting to see that SBO-C is the runner-up for larger sparsity values ($s = \{8, 12\}$) with AK-C taking second place at $s = 4$.

Figure 9.2 shows the denoising performance as the SNR improves. We can clearly see here that AK-S is maintaining its first place position with SBO-C coming in second and AK, AK-C and AK-S fighting for third place. We can also see the plateau past the 30dB mark that allowed us to resume Table 9.4 to the three SNR levels.

9.4 Conclusions

In this chapter we studied and proposed 4 new algorithms for denoising using composite dictionaries structured as a union of orthonormal bases. SBO-C1 is the fastest algorithm providing a composition between SBO dictionaries and 1ONB blocks. SBO-C is an extended version that instead composes two SBO dictionaries offering improved denoising results while being more than 60x faster than the best performing algorithm with a penalty on the denoising quality of less than 3%. We also investigated the hybrid case where we mix a generic dictionary built with AK-SVD with a structured SBO dictionary. The hybrid case provided the best results in our experiments performed on grayscale images with a significant impact on the execution time due to the use of AK-SVD.

Part IV

Epilogue

Chapter 10

Our Research

This chapter lists the articles we wrote on the subject presented in this thesis, followed by a brief description of individual contributions and the novel end results of our activity. At the end we conclude with future research.

10.1 Publications

Journal articles:

1. P. Irofti, Efficient Parallel Implementation for Single Block Orthogonal Dictionary Learning, to appear in Journal of Control Engineering and Applied Informatics, 2015, 1–8 (ISI journal with 2015 impact factor 0.537)

2. P. Irofti, Efficient Dictionary Learning Implementation on the GPU using OpenCL, to appear in U.P.B. Scientific Bulletin, series C, 2015, 1–12

Conference articles:

1. P. Irofti and B. Dumitrescu, GPU parallel implementation of the approximate K-SVD algorithm using OpenCL, in 22nd European Signal Processing Conference, 2014, 271–275 (indexed in ISI Proceedings)

2. P. Irofti and B. Dumitrescu, Overcomplete Dictionary Design: the Impact of the Sparse Representation Algorithm, in The 20th International Conference on Control Systems and Computer Science, 2015, 1–8 (indexed in ISI Proceedings)

3. P. Irofti and B. Dumitrescu, Cosparse Dictionary Learning for the Orthogonal Case, 19th International Conference on System Theory, Control and Computing, 2015, 343–347 (indexed in ISI Proceedings)

4. P. Irofti, Sparse Denoising with Learned Composite Structured Dictionaries, 19th International Conference on System Theory, Control and Computing, 2015, 331–336 (indexed in ISI Proceedings)

Under review:

1. P. Irofti and B. Dumitrescu, Overcomplete Dictionary Learning with Jacobi Atom Updates, 2015, `http://arxiv.org/abs/1509.05054`

10.2 Detailed Contributions

All of my research activity was guided by professor Bogdan Dumitrescu (B.D.). Following, I will list the contributions brought on by me in each article.

Efficient Parallel Implementation for Single Block Orthogonal Dictionary Learning At the suggestion of B.D. I implemented the SBO algorithm in OpenCL. The idea to add more than one orthonormal dictionary blocks at each iteration was mine. I performed the experiments and wrote [63].

Overcomplete Dictionary Learning with Jacobi Atom Updates After our results with K-SVD from [52], I had the idea to generalize the atom update strategy and apply it to other DL algorithms. I implemented the algorithms and performed the experiments; the article [64] was written by B.D. and I.

Efficient Dictionary Learning Implementation on the GPU using OpenCL The OpenCL DL framework was designed and implemented by me; [65] was written by me.

GPU parallel implementation of the approximate K-SVD algorithm using OpenCL The idea to parallelize K-SVD was given by B.D. We both worked on getting a feasible parallel atom update stage. The implementation and experiments were done by me and [52] was written by B.D. and I.

Overcomplete Dictionary Design: the Impact of the Sparse Representation Algorithm B.D. had idea to compare sparse representations algorithms and their impact on the dictionary update stage. I came up with the new POLS algorithm. The implementations were done by me and B.D. and the experiments were performed by me; [66] was written by B.D. and me.

Cosparse Dictionary Learning for the Orthogonal Case The idea of applying cosparse DL for orthogonal dictionaries was mine. B.D. helped with ideas and suggestions about refining the 1ONB-COSP algorithm. The implementation and experiments were done by me. Both I and B.D. wrote the article [67].

Sparse Denoising with Learned Composite Structured Dictionaries The idea of performing denoising with composite dictionaries was mine. I implemented the algorithms, performed the experiments, and wrote [68].

10.3 Original End Results

In this section we list the finite novel products of our research.

- Theory:
  - Jacobi Atom Update framework for dictionary design that reduces representation error, improves execution times and permits full parallelism (Chapter 5)
  - progress in the atom update stage is masked when using the same sparse representation algorithm (Chapter 6)
  - using a more involved sparse representation algorithm (such as POLS) when performing DL and then switching to a faster method (like OMP) when doing representation leads to improved error minimization and execution performance (Chapter 6)
- Algorithms:
  - new sparse representation algorithm POLS that leads to better results and smoother convergence (Chapters 4 and 6)
  - new parallel dictionary update algorithms dervied from the JAU framework: PAK-SVD, P-SGK and P-NSGK (Chapters 5 and 6)
  - new parallel P-SBO algorithm for dictionaries structured as a union of orthonormal basis that improves the final error and the execution times (Chapter 7)

  - new cosparse algorithms 1ONB-COSP and 1ONB-COSP+ that significantly improve dictionary recovery and reduce representation when used as stand-alone and also within SBO and UONB (Chapter 8)
  - new denoising algorithms using composite dictionaries SBO-C1, SBO-C, S-AK and AK-S that provide sharper results with faster execution times (Chapter 9)

- Software:
  - Parallel dictionary learning library for GPUs using OpenCL that includes all the parallel algorithms for generic and structured dictionaries listed above (implementation detailed in the chapters with the corresponding algorithms)
  - Dictionary Learning library for the popular algorithms from the field implemented in C for CPUs
  - Matlab software for image denoising using composite dictionaries

10.4 Future Research

We are currently interested in finding new methods that take advantage of the orthogonal block cosparse training when learning multiple orthoblock dictionaries, researching ways in which we can adapt the representation stage to profit from the structure of composite dictionaries, and refining our parallel implementations as new OpenCL numerical libraries are made available.

The sparse representation field is relatively new with lots of hidden treasures still waiting to be found by current and future researchers. In the future we plan to take part and make the best of this quest.

Bibliography

[1] A.M. Bruckstein, D.L. Donoho, and M. Elad, "From sparse solutions of systems of equations to sparse modeling of signals and images," *SIAM Rev.*, vol. 51, no. 1, pp. 34–81, 2009.

[2] M. Elad, *Sparse and Redundant Representations: from Theory to Applications in Signal Processing*, Springer, 2010.

[3] D.L. Donoho, "Compressed sensing," *Information Theory, IEEE Transactions on*, vol. 52, no. 4, pp. 1289–1306, 2006.

[4] D.L. Donoho and M. Elad, "Optimally sparse representation in general (nonorthogonal) dictionaries via l_1 minimization," *Proceedings of the National Academy of Sciences*, vol. 100, no. 5, pp. 2197–2202, 2003.

[5] I.F. Gorodnitsky and B.D. Rao, "Sparse signal reconstruction from limited data using FOCUSS: a re-weighted minimum norm algorithm," *Signal Processing, IEEE Transactions on*, vol. 45, no. 3, pp. 600–616, Mar 1997.

[6] D.L. Donoho and P.B. Stark, "Uncertainty principles and signal recovery," *SIAM Journal on Applied Mathematics*, vol. 49, no. 3, pp. 906–931, 1989.

[7] D.L. Donoho and X. Huo, "Uncertainty principles and ideal atomic decomposition," *Information Theory, IEEE Transactions on*, vol. 47, no. 7, pp. 2845–2862, 2001.

[8] M. Elad and A.M. Bruckstein, "A generalized uncertainty principle and sparse representation in pairs of bases," *Information Theory, IEEE Transactions on*, vol. 48, no. 9, pp. 2558–2567, 2002.

[9] D.L. Donoho, M. Elad, and V. Temlyakov, "Stable recovery of sparse overcomplete representations in the presence of noise," *Information Theory, IEEE Transactions on*, vol. 52, no. 1, pp. 6–18, 2006.

[10] D.L. Donoho and M. Elad, "On the stability of the basis pursuit in the presence of noise," *Signal Processing*, vol. 86, no. 3, pp. 511–532, 2006.

[11] E.J. Candes and T. Tao, "Decoding by Linear Programming," *IEEE Trans. Info. Theory*, vol. 51, no. 12, pp. 4203–4215, Dec. 2005.

[12] J.D. Blanchard, C. Cartis, and J. Tanner, "Compressed Sensing: How Sharp is the Restricted Isometry Property?," *SIAM Rev.*, vol. 53, no. 1, pp. 105–125, 2009.

[13] E.J. Candes and J.K. Romberg, "Signal recovery from random projections," in *Electronic Imaging 2005*. International Society for Optics and Photonics, 2005, pp. 76–86.

[14] E.J. Candes and T. Tao, "Near-optimal signal recovery from random projections: Universal encoding strategies?," *Information Theory, IEEE Transactions on*, vol. 52, no. 12, pp. 5406–5425, 2006.

[15] E.J. Candes, J. Romberg, and T. Tao, "Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information," *Information Theory, IEEE Transactions on*, vol. 52, no. 2, pp. 489–509, 2006.

[16] E.J. Candes and J. Romberg, "Quantitative robust uncertainty principles and optimally sparse decompositions," *Foundations of Computational Mathematics*, vol. 6, no. 2, pp. 227–254, 2006.

[17] E.J. Candes, J.K. Romberg, and T. Tao, "Stable signal recovery from incomplete and inaccurate measurements," *Communications on pure and applied mathematics*, vol. 59, no. 8, pp. 1207–1223, 2006.

[18] D.L. Donoho, "For most large underdetermined systems of linear equations the minimal l_1-norm solution is also the sparsest solution," *Communications on pure and applied mathematics*, vol. 59, no. 6, pp. 797–829, 2006.

[19] D.L. Donoho, "For most large underdetermined systems of equations, the minimal l_1-norm near-solution approximates the sparsest near-solution," *Communications on pure and applied mathematics*, vol. 59, no. 7, pp. 907–934, 2006.

[20] S. Mallat, *A wavelet tour of signal processing*, Academic press, 1999.

[21] R.R. Coifman and M.V. Wickerhauser, “Adapted waveform analysis as a tool for modeling, feature extraction, and denoising,” *Optical Engineering*, vol. 33, no. 7, pp. 2170–2174, 1994.

[22] E. Le Pennec and S. Mallat, “Bandelet image approximation and compression,” *Multiscale Modeling & Simulation*, vol. 4, no. 3, pp. 992–1039, 2005.

[23] E.J. Candes and D.L. Donoho, *Curvelets: A surprisingly effective nonadaptive representation for objects with edges*, DTIC Document, 1999.

[24] M.N. Do and M. Vetterli, “The contourlet transform: an efficient directional multiresolution image representation,” *Image Processing, IEEE Transactions on*, vol. 14, no. 12, pp. 2091–2106, 2005.

[25] I. Tosic and P. Frossard, “Dictionary Learning,” *IEEE Signal Proc. Mag.*, vol. 28, no. 2, pp. 27–38, Mar. 2011.

[26] K. Kreutz-Delgado, J.F. Murray, B.D. Rao, K. Engan, T.W. Lee, and T.J. Sejnowski, “Dictionary learning algorithms for sparse representation,” *Neural computation*, vol. 15, no. 2, pp. 349–396, 2003.

[27] R. Rubinstein, A.M. Bruckstein, and M. Elad, “Dictionaries for Sparse Representations Modeling,” *Proc. IEEE*, vol. 98, no. 6, pp. 1045–1057, June 2010.

[28] K. Engan, S.O. Aase, and J.H. Husoy, “Method of optimal directions for frame design,” in *IEEE Int. Conf. Acoustics Speech Signal Proc.*, 1999, vol. 5, pp. 2443–2446.

[29] M. Aharon, M. Elad, and A.M. Bruckstein, “K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation,” *Signal Processing, IEEE Transactions on*, vol. 54, no. 11, pp. 4311–4322, 2006.

[30] R. Rubinstein, M. Zibulevsky, and M. Elad, “Efficient Implementation of the K-SVD Algorithm using Batch Orthogonal Matching Pursuit,” *Technical Report - CS Technion*, 2008.

[31] S.K. Sahoo and A. Makur, “Dictionary training for sparse representation as generalization of k-means clustering,” *Signal Processing Letters, IEEE*, vol. 20, no. 6, pp. 587–590, 2013.

[32] M. Sadeghi, M. Babaie-Zadeh, and C. Jutten, “Dictionary Learning for Sparse Representation: a Novel Approach,” *IEEE Signal Proc. Letter*, vol. 20, no. 12, pp. 1195–1198, Dec. 2013.

[33] S. Lesage, R. Gribonval, F. Bimbot, and L. Benaroya, “Learning unions of orthonormal bases with thresholded singular value decomposition,” in *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP ’05). IEEE International Conference on*, March 2005, vol. 5, pp. v/293–v/296 Vol. 5.

[34] C. Rusu and B. Dumitrescu, “Block orthonormal overcomplete dictionary learning,” in *21st European Signal Processing Conference*, 2013, pp. 1–5.

[35] A.G. Weber, “The USC-SIPI Image Database,” 1997.

[36] Khronos OpenCL Working Group, *The OpenCL Specification, Version 1.2, Revision 19*, Khronos Group, 2012.

[37] M. Martins, G. Falcao, and I.M. Figueiredo, “Fast Aberrant Crypt Foci Segmentation on the GPU,” in *IEEE Int. Conf. Acoustics Speech Signal Proc.*, 2013, pp. 1113–1117.

[38] G. Condello, P. Pasteris, D. Pau, and M. Sami, “An OpenCL-based feature matcher,” *Signal Proc. Image Comm.*, vol. 28, pp. 345–350, 2013.

[39] C. Garcia, G. Botella, F. Ayuso, M. Prieto, and F. Tirado, “Multi-GPU based on multicriteria optimization for motion estimation system,” *EURASIP J. Adv. Signal Proc.*, vol. 23, 2013.

[40] K. Hwang and W. Sung, “Load Balanced Resampling for Real-Time Particle Filtering on Graphics Processing Units,” *IEEE Trans. Signal Proc.*, vol. 61, no. 2, pp. 411–419, Jan. 2013.

[41] K. Konishi, “Parallel GPU Implementation of Null Space Based Alternating Optimization Algorithm for Large-Scale Matrix Rank Minimization,” in *IEEE Int. Conf. Acoustics Speech Signal Proc.*, 2012, pp. 3698–3692.

[42] P. Nagesh, R. Gowda, and B. Li, “Fast GPU Implementation of Large Scale Dictionary and Sparse Representation Based Vision Problems,” in *IEEE Int. Conf. Acoustics Speech Signal Proc.*, 2010, pp. 1570–1573.

[43] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming,” *Parallel Computing*, vol. 38, pp. 391–407, 2014.

[44] Advanced Micro Devices, *AMD Accelerated Parallel Processing OpenCL Programming Guide*, Advanced Micro Devices, Inc., 2013.

[45] Y.C. Pati, R. Rezaiifar, and P.S. Krishnaprasad, "Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition," in *27th Asilomar Conf. Signals Systems Computers*, Nov. 1993, vol. 1, pp. 40–44.

[46] S. Chen, S.A. Billings, and W. Luo, "Orthogonal Least Squares Methods and Their Application to Non-Linear System Identification," *Int. J. Control*, vol. 50, no. 5, pp. 1873–1896, 1989.

[47] W. Dai and O. Milenkovic, "Subspace pursuit for compressive sensing signal reconstruction," *Information Theory, IEEE Transactions on*, vol. 55, no. 5, pp. 2230–2249, 2009.

[48] S. Chatterjee, D. Sundman, M. Vehkapera, and M. Skoglund, "Projection-based and look-ahead strategies for atom selection," *Signal Processing, IEEE Transactions on*, vol. 60, no. 2, pp. 634–647, 2012.

[49] S. S. Chen, D. L. Donoho, and M. A. Saunders, "Atomic decomposition by basis pursuit," *SIAM Review*, vol. 43, no. 1, pp. 129–159, 2001.

[50] S.S. Chen, *Basis pursuit*, Ph.D. thesis, Dept. of Statistics, Stanford University, Standford, CA, 1995.

[51] B.D. Rao and I.F. Gorodnitsky, "Affine scaling transformation based methods for computing low complexity sparse solutions," in *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, May 1996, vol. 3, pp. 1783–1786 vol. 3.

[52] P. Irofti and B. Dumitrescu, "GPU parallel implementation of the approximate K-SVD algorithm using OpenCL," in *22nd European Signal Processing Conference*, 2014, pp. 271–275.

[53] J.M. Chambers, "Algorithm 410: partial sorting," *Communications of the ACM*, vol. 14, no. 5, pp. 357–358, 1971.

[54] J.C. Gower and G.B. Dijksterhuis, *Procrustes problems*, vol. 3, Oxford University Press Oxford, 2004.

[55] S. Nam, M.E. Davies, M. Elad, and R. Gribonval, "The cosparse analysis model and algorithms," *Applied and Computational Harmonic Analysis*, vol. 34, no. 1, pp. 30–56, 2013.

[56] R. Rubinstein, T. Peleg, and M. Elad, “Analysis K-SVD: A dictionary-learning algorithm for the analysis sparse model,” *Signal Processing, IEEE Transactions on*, vol. 61, no. 3, pp. 661–677, 2013.

[57] M. Elad and M. Aharon, “Image denoising via sparse and redundant representations over learned dictionaries,” *Image Processing, IEEE Transactions on*, vol. 15, no. 12, pp. 3736–3745, 2006.

[58] K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian, “Image denoising by sparse 3-d transform-domain collaborative filtering,” *Image Processing, IEEE Transactions on*, vol. 16, no. 8, pp. 2080–2095, 2007.

[59] C.Y. Yang, J.B. Huang, and M.H. Yang, “Exploiting self-similarities for single frame super-resolution,” in *Computer Vision–ACCV 2010*, pp. 497–510. Springer, 2011.

[60] L. Le Magoarou and R. Gribonval, “Chasing butterflies: In search of efficient dictionaries,” in *Acoustics, Speech, and Signal Processing, 2015. Proceedings. (ICASSP ’15). IEEE International Conference on*, 2015, pp. 1–5.

[61] Z. Wang, Y. Yang, J. Yang, and T.S. Huang, “Designing a composite dictionary adaptively from joint examples,” *CoRR*, vol. abs/1503.03621, 2015.

[62] Y. C. Pati, R. Rezaiifar, and P. S. Krishnaprasad, “Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition,” in *Conference Record of The Twenty-Seventh Asilomar Conference on Signals, Systems and Computers*, 1993, pp. 1–3.

[63] P. Irofti, “Efficient parallel implementation for single block orthogonal dictionary learning,” *Journal of Control Engineering and Applied Informatics*, vol. 17, no. 3, pp. 1–8, 2015.

[64] P. Irofti and B. Dumitrescu, “Overcomplete Dictionary Learning with Jacobi Atom Updates,” *ArXiv e-prints*, vol. abs/1509.05054, Sept. 2015.

[65] P. Irofti, “Efficient dictionary learning implementation on the GPU using OpenCL,” *U.P.B. Scientific Bulletin, Series C*, vol. 77, no. 3, pp. 1–12, 2015.

[66] P. Irofti and B. Dumitrescu, “Overcomplete dictionary design: the impact of the sparse representation algorithm,” in *The 20th International Conference on Control Systems and Computer Science*, 2015, pp. 901–908.

[67] P. Irofti and B. Dumitrescu, "Cosparse dictionary learning for the orthogonal case," in *19th International Conference on System Theory, Control and Computing*, 2015, pp. 343–347.

[68] P. Irofti, "Sparse denoising with learned composite structured dictionaries," in *19th International Conference on System Theory, Control and Computing*, 2015, pp. 331–336.