EFFICIENT DICTIONARY LEARNING IMPLEMENTATION ON THE GPU USING OPENCL

Paul Irofti[1]

Abstract *The dictionary learning field offers a wide range of algorithms that are able to provide good sparse approximations and well trained dictionaries. These algorithms are very complex and this is reflected in the slow execution of their computationally intensive implementations. This article proposes efficient parallel implementations for the main algorithms in the field that significantly reduce the execution time.*

Keywords: sparse representation, dictionary design, parallel algorithm, GPU, OpenCL

1. Introduction

Sparse representation with overcomplete dictionaries has been an active research topic during the last decade in the signal processing community. The increased interest in the field is certainly due to the wide range of applications[1, 2] it provides such as noise removal[3], compression[5], classification and compressed sensing[4].

The field's core problem is: given a set of signals $Y \in \mathbb{R}^{p\times m}$, train a dictionary $D \in \mathbb{R}^{p\times n}$, whose columns are also called atoms, through which the signals from Y can be sparsely represented by using just a few atoms from D. This approximation is formalized as the following optimization problem:

$$\min_{D,X} \|Y - DX\|_F \tag{1}$$

where $X \in \mathbb{R}^{n\times m}$ are the sparse representations and the factorization DX is the Frobenius norm approximation of Y.

The dictionary learning (DL) problem from (1) is hard due to its bilinear nature of finding both the sparse representations and their associated well designed dictionary. This problem has many variables and thus require simple optimization methods which is why most algorithms approach DL through iterative methods that split the problem in two distinct stages: representation and dictionary update. These two stages are iterated a number of times until a local minima is reached.

[1]Eng, Department of Automatic Control and Computers University Politehnica of Bucharest 313 Spl. Independenţei, 060042 Bucharest, Romania, e-mail: paul@irofti.net.

1.1. Sparse Representations

When performing sparse representation the dictionary is fixed and the interest is 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:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \|x\|_0 \\ \text{subject to} \quad & y = Dx \end{aligned} \tag{2}$$

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 (2) by approximating y following a sparsity constraint s:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \|y - Dx\|_2^2 \\ \text{subject to} \quad & \|x\|_0 \leq s \end{aligned} \tag{3}$$

In the above formulations the pursuit of x can be split 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 $\mathcal{I}$, the representation can be computed as:

$$x_{\mathcal{I}} = (D_{\mathcal{I}}^T D_{\mathcal{I}})^{-1} D_{\mathcal{I}}^T y \tag{4}$$

where $x_{\mathcal{I}}$ are the coefficients corresponding to the current support and thus the other elements of x are zero.

Most DL algorithms use Orthogonal Matching Pursuit (OMP) [11] during the sparse representations stage. The main reason is that OMP is fast and 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, it is worth mentioning that there are other greedy algorithms like Orthogonal Least Squares (OLS) [12], Subspace Pursuit [13], Projection-Based OMP (POMP) or Look-Ahead OLS (LAOLS) [14], that, although more computationally intensive than OMP, are still fast enough for practical use and provide better representations. These algorithms follow the same steps described in the last paragraph with differences in the strategy of building the sparse support $\mathcal{I}$.

1.2. Dictionary Update

Moving on towards dictionary update methods the problem is changing as the representations are fixed and the dictionary goes through a refinement stage that sometimes affects not only its atoms but also their associated representations.

Among the dictionary update methods K-SVD [8] is by far the most popular approach. K-SVD starts by writing the DX factorization as:

$$DX = \sum_{\ell=1}^{n} d_\ell X_{\ell,\mathcal{I}_\ell} \tag{5}$$

where $\mathcal{I}_\ell$ are the indices of the nonzero elements of the ℓ-th row from X representing the signals that use atom d_ℓ in their representation. The representation error without atom j in the dictionary can then be written as the matrix:

$$E = Y - \sum_{\ell \neq j} d_\ell X_{\ell,\mathcal{I}_\ell} \tag{6}$$

And so the authors attack the dictionary refinement problem by posing the following optimization problem

$$\min_{d_j, X_{j,\mathcal{I}_j}} \left\| E - d_j X_{j,\mathcal{I}_j} \right\|_F^2 \tag{7}$$

where atom d_j and the affected representations $X_{j,\mathcal{I}_j}$ are updated while the rest of the atoms are fixed. The problem is treated as a rank-1 approximation of the error matrix whose solution is the singular vector of the largest singular value. AK-SVD[7] approaches the problem in an almost identical manner, except that it builds the singular vector through a single power method iteration.

A generalization to K-Means clustering of K-SVD led to SGK [9] that relaxes the update optimization problem (7) and updates just the atom without its associated representations:

$$\min_{d_j} \left\| \left(Y - \sum_{\ell \neq j} d_\ell X_{\ell,\mathcal{I}_\ell} \right) - d_j X_{j,\mathcal{I}_j} \right\|_F^2 \tag{8}$$

With this change the problem is simplified to least-squares. NSGK [10] builds on top of SGK by solving (8) and updating the dictionary and the representations as differences between their current and previous values:

$$\begin{cases} D = D^{(k-1)} + (D^{(k)} - D^{(k-1)}) \\ X = X^{(k-1)} + (X^{(k)} - X^{(k-1)}) \end{cases} \tag{9}$$

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.

This process can be simplified by changing the matrix Y from (8) with

$$Z = Y + D^{(k)} X^{(k-1)} - D^{(k)} X^{(k)} \tag{10}$$

For a more in-depth overview of the field the reader is invited to consult the works from [15] and [1].

1.3. Parallelism with OpenCL

Due to the complex problems that they attempt to solve, both the greedy representation algorithms and the atom update methods are computationally intensive tasks that take a long time to execute. The goal here is to provide fast and efficient solutions for these tasks by splitting them into smaller independent execution blocks that can be parallelized on multicore architectures.

OpenCL [6] is an open standard for portable parallelization created as a response to the growing number of applications that exploit the extensive set of features found on modern graphical proccessing units (GPU) in order to provide a general purpose programming environment. The main advantage of using OpenCL versus other existing vendor-specific frameworks (such as CUDA) is that the portable hardware abstraction imposed by the standard provides a single unified language for all types of GPUs, CPUs and FGPAs.

OpenCL abstracts the smallest execution unit available in hardware as proccessing elemnents (PE) that are organized in groups at which parallelism is guaranteed called compute units (CU) located on the OpenCL device. Compute units can also be executed in parallel and on the GPUs they are also called waves or wavefronts. The PEs execute identical small functions (also called kernels) and are organized in an n-dimensional space that is defined by the application. An occupied (busy) PE is also called a work-item. The set of active compute units are called work-groups. PEs share resources locally, within the compute unit, and globally, on the OpenCL device. For a bidimensional split of the PE set, the n-dimensional range definition is denoted as NDR($\langle x_g, y_g\rangle$, $\langle x_l, y_l\rangle$). There are $x_g \times y_g$ PEs, organized in work-groups of size $x_l \times y_l$, running the same kernel.

Even though the case-study of the experiments presented here use the GPU, note that this proposal can be applied on any multicore setup (including regular CPUs and FPGAs) that adheres to the OpenCL standard.

The manuscript is structured as follows: section 2 proposes and discusses an efficient parallel implementation for the sparse representation stage, section 3 discusses and analyzes the parallelization of the atom update stage for AK-SVD, SGK and NSGK and section 4 concludes the study.

2. Sparse Signal Representation

The dictionary learning process, as described in section 1, operates on large training signals sets that need to be sparsely represented. Following the equation from (3) the first stage of DL is naturally parallel, since the signal representations are completely independent. Given that this applies to all greedy algorithms, OMP is chosen as a case-study as it is the popular choice in the literature. The following subsections will present the details of the OMP algorithm and describe its parallel OpenCL implementation.

Algorithm 1: 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 $w \left\{Lw = G_{\mathcal{I},s}\right\}$
8          $L = \begin{pmatrix} L & 0 \\ w^T & \sqrt{1 - w^T w} \end{pmatrix}$
9      Compute new solution: $LL^T x_{\mathcal{I}} = \alpha^0_I$
10     Update: $\alpha = \alpha^0 - G_I x_{\mathcal{I}}$
```

2.1. Batch Orthogonal Matching Pursuit

The implementation followed the Batch OMP (BOMP) algorithm variant described in [7]. The steps from algorithm **??** present the operations necessary for performing sparse representation for a single signal y. The authors from [7] show that the best time performance is obtained when first precomputing the scalar products between the atoms and themselves and between the atoms and the signal vectors (step 1) and then proceed to compute the actual sparse representation of each signal. Since these are matrix multiplications of fairly large size, they can be easily parallelized. For each signal y, the atom selection is made by the standard matching pursuit criterion in step 4 and the sparse support is extended in step 5. Next, 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. Due to the precomputations from step 1, an explicit residual update is no longer necessary and can be replaced by the expression from step 10 (as explained in [7]) which has a lower computation complexity. Sparse representations are computed via BOMP in parallel for groups of $\tilde{m}$ signals.

2.2. 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. These operations are depicted in the first part of figure 1 where each grid represent one BLAS operation and each grid element represents a block matrix multiplication.

The Input and resulting matrix are kept in global memory. Each work-group performs the operations required for calculating one block from the result matrix. PEs are organized in a 2-dimensional space that is further split into 2-dimensional block-size dependent work-groups. Full resource occupancy of the

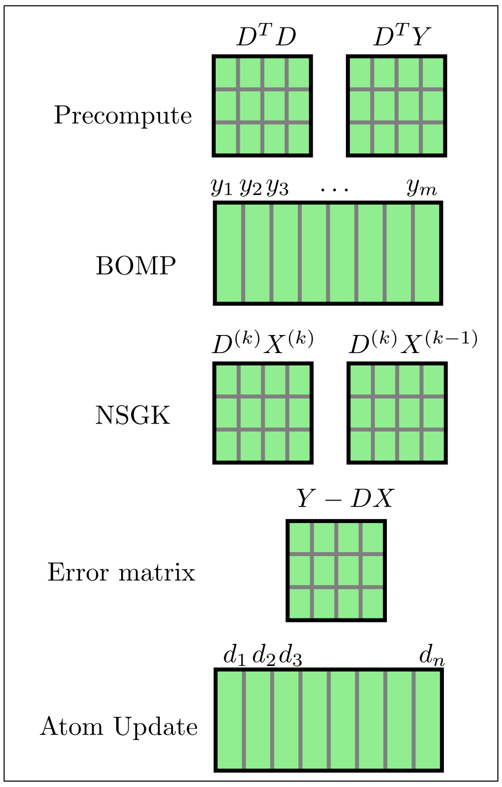

FIG. 1. Parallel DL: each row is executed simultaneously and each grid represents an OpenCL kernel whose elements are independent execution threads.

GPU (an indicator of maximum performance) was achieved when using work-groups of 16×16 PEs. Thus for a result matrix $A \in \mathbb{R}^{n \times m}$ the n-dimensional space was defined 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 the device available for this study, the fastest vectorized type was float4.

All the operations required for the sparse representation of a single signal with BOMP, were packed in and implemented by a single OpenCL kernel as shown in the second row from figure 1. 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 IO. In-lining proved to be a lot faster.

The main obstacles encountered during the implementation were memory bound. BOMP is a huge memory consumer and mostly due to auxiliary data.

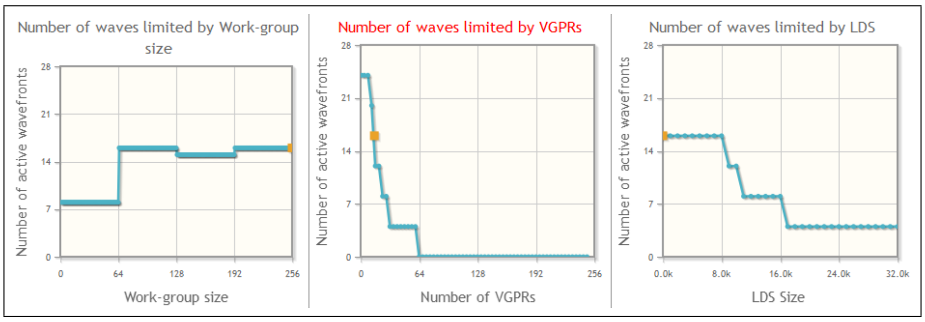

FIG. 2. BOMP representation kernel occupancy

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, it was 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 on this GPU. Experiments 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, there was no significant differences noted between values from 1024 to m.

Table 1 and figure 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 vector general purpose registers (VGPR) usage per work-item, the local data size (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 kernel is marked with a squared dot on the graphs from figure 2 where it can be seen how resources limit the number of active wavefronts. The limiting factor is the number of VGPRs used and changing the $\tilde{m}$ signal grouping brings no change in this value. More so, table 1 shows that varying $\tilde{m}$ indeed does not affect the kernel occupancy which is always at 67%.

2.3. Performance

This subsection presents the performance of the parallel GPU implementation of the BOMP algorithm and compares it to an almost identical CPU version. It was possible to keep an almost one-to-one instruction equivalence due to the fact that the OpenCL langauge is a custom subset of the C language. The execution times were measured when varying the number of signals and keeping a fixed dictionary dimension and vice-versa. In both scenarios $\tilde{m} = m$ was used for the OpenCL implementation.

TABLE 1. Kernel information and occupancy for BOMP

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

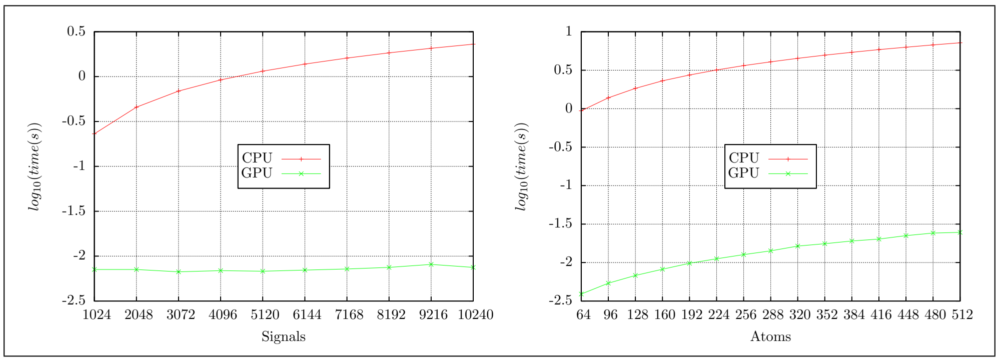

FIG. 3. BOMP performance with varied number of signals m on the left panel and varied dictionary sizes n on the right.

Figure 3 presents in the left panel the elapsed time, in logarithmic scale, 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 with a sparsity goal of $s = 8$. This experiment shows performance improvement of up to 312 times when using the OpenCL GPU version.

In the right panel of figure 3 increasing the dictionary size ($n = 64$ up to $n = 512$) has a visible performance effect on the GPU implementation as the total number of atoms has a direct impact on the number of instructions performed by each work-item. The tests 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.

3. Dictionary Update

This section proposes and studies the efficient OpenCL implementations for the dictionary update stage of the AK-SVD, SGK and NSGK methods. Even though the optimization problems from (7) and (8) impose a sequential atom-by-atom update it was shown in [16] that updating the atoms in parallel groups of $\tilde{n}$ provides similar approximation results, if not better. In order to completely separate the update instructions of each atom, the error matrix from (6) was precomputed but with all atoms included in dictionary D. The small self-exclusion task was left to each atom. And so, with minor adjustments, this lead to a completely parallel atom update stage preceded by the full error matrix computation.

3.1. OpenCL Implementation

The matrix multiplication needed for computing the current error is done in the same manner as the matrix precomputations for BOMP that were described in section 2. Note that for NSGK two extra BLAS operations are needed in order to compute the difference based matrix Z as described around equation (10). Matrix Z will replace Y when computing the error matrix E. That is why in figure 1 the NSGK row precedes the error calculation from the fourth row.

The actual atom update process was implemented by a single OpenCL kernel as depicted in the last row from figure 1. The error matrix E, the dictionary D and the representation matrix X are kept in global memory while the atom to be updated is transfered in private memory by each PE. This brings an increase in performance through lower latency during the update operations. The required extra storage is not an issue because the problem size is usually reasonable ($p \leq 64$).

There were difficulties with storing the list of indices $\mathcal{I}$ of the signals using the current atom in their representation. The list size varies a lot from one atom to another. Only 8000 indices fit in local memory on the GPU available for this study. This might be enough for some use-cases but not for all. If this bound is exceeded, $\mathcal{I}$ is stored in global memory which solves the problem at the cost of higher access times.

For all algorithms the PEs are partitioned as a 1-dimensional space of $\tilde{n}$ work-items and the work-group size is left up to the GPU scheduler by using $NDR(\langle n \rangle, \langle \text{any} \rangle)$.

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

The AK-SVD and the SGK kernels are the same until the point where the atom is updated. At that time SGK is done with the update stage while AK-SVD has to perform the extra instructions needed for updating the affected sparse representations. NSGK uses the same atom update kernel as SGK with the input error matrix calculated as $E = Z - DX$ instead of $E = Y - DX$ as explained earlier.

Table 2 analyzes the atom update kernel performance in terms of GPU occupancy. 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 focus here is on the main obstacle: the memory storage location of the indices set $\mathcal{I}$. Table 2 shows that moving the indices in local memory has a

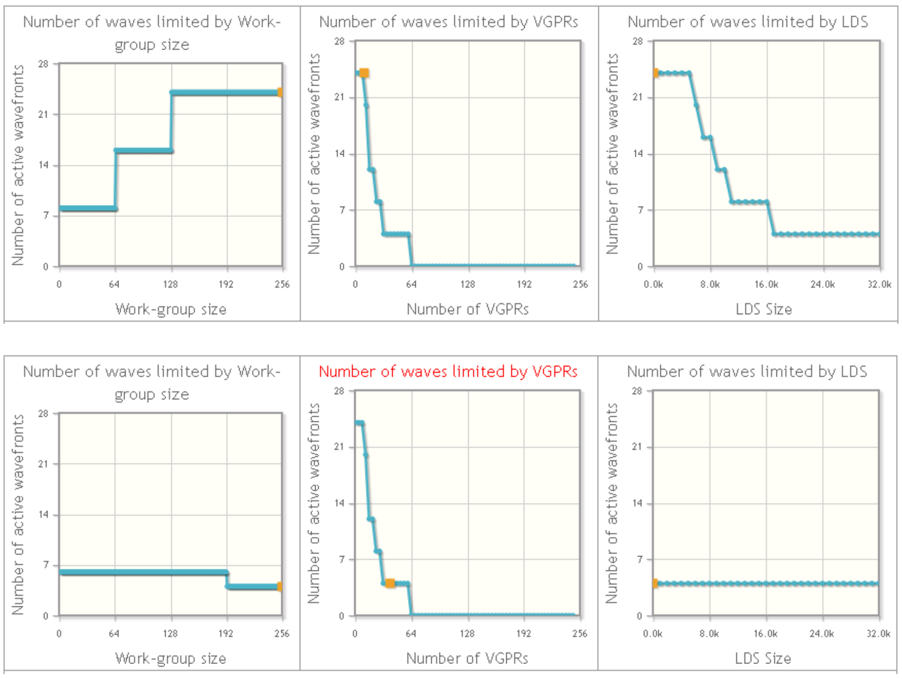

FIG. 4. AK-SVD dictionary update GPU occupancy. The top panel shows the benefits of keeping $\mathcal{I}$ in local memory while the bottom panel depicts what happens when it is moved it in global memory.

TABLE 2. Kernel information and occupancy for SGK

Kernel	SGK		AK-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

significant impact bumping occupancy from 16% to 100% due to lowering the number of used VGPRs by 32 and 30 in the SGK and, respectively, AK-SVD case. The difference can also be spotted in figure 4 by comparing the VGPR screens from the top and bottom panels. The rest of the occupancy factors are not affected by $\mathcal{I}$.

3.2. Performance

In figure 5 the parallel GPU versions were prefixed with P and compared their performance to the regular DL algorithms. Two experiments were performed that depict how execution time is affected by an increase in dictionary

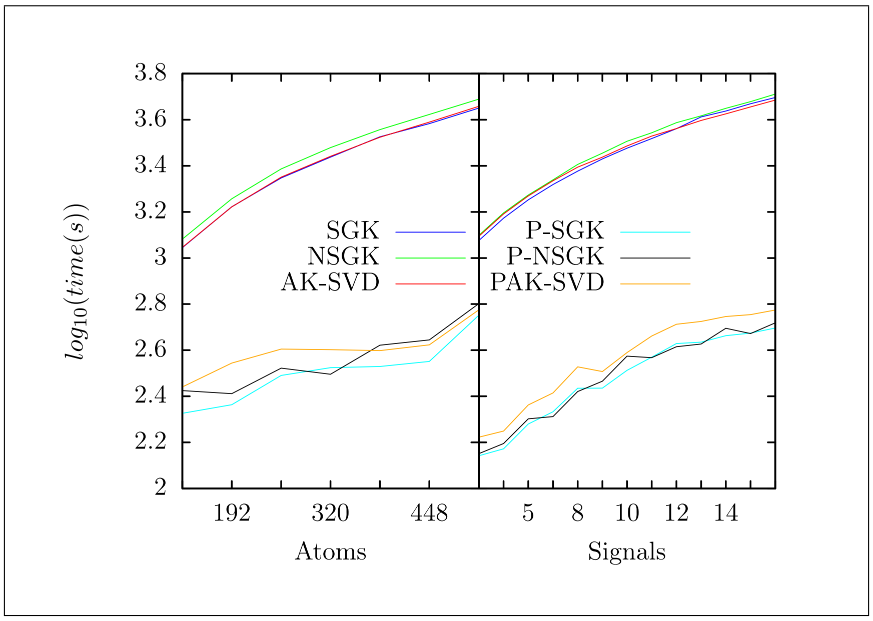

FIG. 5. DL execution times.

atoms and, respectively, in training signals. All methods were executed for $k = 200$ iterations with full atom parallelism $\tilde{n} = n$.

The dictionary experiment used a set of $m = 12288$ signals with a target sparsity of $s = 6$. The signal experiment used a dictionary of $n = 256$ atoms with sparsity $s = 10$. In the signals panel the x-axis represents thousands of signals. The results show an increase in speed of 10.6 times for NSGK, 10.8 times for SGK and 12 times for AK-SVD.

4. **Conclusions**

This paper studied and proposed efficient GPU implementations using OpenCL for the main algorithms in the dictionary learning field. A full description of the kernels was provided leading to complete parallel execution of the sparse representation and dictionary update stages. Also discussed was the n-dimensional topology of each kernel and the optimal storage location of the data structures in order to obtain the best GPU occupancy.

Acknowledgement

This work was supported by the Romanian National Authority for Scientific Research, CNCS - UEFISCDI, project number PN-II-ID-PCE-2011-3-0400 and by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/132395.

References

[1] *I. Tosic and P. Frossard*, "Dictionary Learning," IEEE Signal Proc. Mag., **28**(2011), no. 2, 27–38.

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

[3] *M. Elad and M. Aharon*, "Image denoising via sparse and redundant representations over learned dictionaries," Image Processing, IEEE Transactions on, **15**(2006), no. 12, 3736–3745.

[4] *D.L. Donoho*, "Compressed Sensing," Information Theory, IEEE Transactions on, **52**(2006), no. 4, 1289–1306.

[5] *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, **15**(2003), no. 2, 349–396.

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

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

[8] *M. Aharon, M. Elad, and A.M. Bruckstein*, "K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation," Signal Processing, IEEE Transactions on, **54**(2006), no. 11, 4311–4322.

[9] *S.K. Sahoo and A. Makur*, "Dictionary training for sparse representation as generalization of k-means clustering," Signal Processing Letters, IEEE, **20**(2013), no. 6, 587–590.

[10] *M. Sadeghi, M. Babaie-Zadeh, and C. Jutten*, "Dictionary Learning for Sparse Representation: a Novel Approach," *IEEE Signal Proc. Letter*, **20**(2013), no. 12, 1195–1198.

[11] *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, **1**(1993), 40–44.

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

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

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

[15] *R. Rubinstein, A.M. Bruckstein, and M. Elad*, "Dictionaries for Sparse Representations Modeling," Proc. IEEE, **98**(2010), no. 6, 1045–1057.

[16] *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.