Cuda implementation of K-LiMapS algorithm

Alice Sansoni, 970812

February 2022

1 Introduction

The problem of examining sparsity in signal processing and adjoining areas for large signal dimensions is non polynomial (NP-hard). Hence, the need arises here to find a fast algorithm which calculates a nearly-optimal sparse representation over redundant dictionaries with an arbitrary noisy input signal, which is restricted to be a linear combination of k atoms from a fixed dictionary. The method rests on non-linearity in order to promote sparsity followed by projections into linear spaces of fixed low dimensions with reduced reconstruction error. The main purpose of this interaction is that of reducing the gap existing between the two image spaces, where the first is a subspace containing sparse points (using the concept of l^0-norm which measures the total number of nonzero elements) while the second is a set containing infinite solutions for the problem. Fixed-points can be obtained as the limit of a fixed-point iteration scheme defined by repeated images under the mapping of an arbitrary starting point in the space. Randomly generated synthetic instances of a noisy linear system satisfying sparsity requirements are considered here.

There are some practical applications in features extraction, for example in the face recognition task with Single Sample Per Person (SSPP), where k-LiMapS is alternated with a sparse dictionary learning technique based on the method of optimal direction, to derive a succinct and consistent sub-dictionary for each subject. This is included as the last stage in the face recognition process [5]. It can also be applied to learn a dictionary for a class of ECG signals, where for many cases is of practical importance to fix a priory the desired level of sparsity in sparse recovery algorithms. Other applications include compression and regularization in inverse problems.

2 Sparse Representation

Let's consider the problem of finding the sparsest representation possible in an over-complete dictionary $\Phi = [\phi_1, ..., \phi_m]$ assumed to be a collection of $m > n$ atoms or vectors in R^n. A sparse representation for a given vector or signal $s = (s_1, ..., s_n)$ expressed as a linear combination of atoms, i.e., $s = \sum_i \alpha_i \phi_i$, is measured in terms of the so-called l^0-norm $\|\alpha\|_0$, simply representing the number of non-zero elements in α. Usually, the l^0-norm is defined as the cardinality of set $S(\alpha) = \{k : \alpha_k \neq 0\}$, called the support of α. More generally, it is not sensible to assume that $\|\alpha\|_0 = k << n$. A more plausible scenario assumes sparse approximate representation in which there is an ideal noiseless signal s (admitting a sparse representation) corrupted by noise, leading to the following model:

$$s = \Phi_\alpha + \epsilon \tag{1}$$

In which error or noise $\epsilon \in R^n$ gives rise, for instances, to measurements or estimates. Adopting this noisy setting, the general goal of finding the sparsest decomposition of the signal s can be rephrased as the constrained minimization problem

$$min_{\alpha \in R^m} \|s - \Phi\alpha\|^2 \text{ subject to } \|\alpha\|_0 \leq k \tag{P$_0$}$$

where $\|\cdot\|$ denotes the l^2-norm. However, this optimization problem is generally NP-hard.

LIMAPS is a new sparse approximation technique that consists of a fixed-point iteration schema based on a nonlinear mapping, aimed to uniformly enhance the sparseness level of each iterate. In fact, at each iteration step, every coefficient is either contracted (moved towards zero) or preserved (moved far from zero) to project the new coefficient vector onto the linear subspace spanned by the columns of the matrix, which is the affine space associated to the linear model (1). To deal with high dimensional data, the method provides a parametric family of nonlinear functions $F = \{f_\lambda : R^m \to R^m | \lambda \in R^+\}$ where a component is defined as:

$$f_\lambda(\alpha) = \alpha \odot (1 - e^{-\lambda|\alpha|}) \tag{2}$$

being $\odot$ the Hadamard (element-wise) product. When λ in $(1 - e^{-\lambda|\alpha|})$ goes to infinite the penalty vanishes becoming the identity function for all $\alpha_i \in R$. Combining nonlinear mappings belongings to the family (2) and orthogonal projections in the null space of matrix Φ, it can be showed that the sequence $(\alpha^{(t)})_{t \geq 0}$ always converges.

3 k-LiMapS algorithm

To solve the problem of sparsity analyses it has been decided to implement the k-LiMapS algorithm (k-Coefficients Lipschitzian Mapping for Sparsity) trying to parallelize some of the typical matrix operations that are required. The paper [1] has been followed as a guideline for the sequential part of this project.

The main drawback of LiMapS is represented by the need of providing the right sequence for the parameter λ indexing the function family in (2) to achieve a convergent sequence. The proposed method can adaptively find a suitable sequence $\{\lambda_t\}$ for approximately solving the problem P_0. This choice should also depend on imposed constraints of choosing k coefficients not null and discarding the remaining m-k.

3.1 Sequential version

The overall algorithm is sketched in Algorithm 1.

Algorithm 1 k-LiMapS

Require: - a dictionary $\Phi \in \mathbb{R}^{n \times m}$
- its pseudo-inverse $\Phi^{\dagger}$
- a signal $s \in \mathbb{R}^n$
- a sparsity level k

1: $\alpha \leftarrow \Phi^{\dagger} s$
2: **while** [cond] **do**
3: $\sigma \leftarrow$ sort $(|\alpha|)$ *<descending order coefficients>*
4: $\lambda \leftarrow 1/\sigma_k$ *<sparsity ratio update>*
5: $\beta \leftarrow f_\lambda(\alpha)$ *<increase sparsity>*
6: $\alpha \leftarrow \beta - \Phi^{\dagger}(\Phi\beta - s)$ *<orthogonal projection>*
7: **end while**
8: $\alpha_j \leftarrow 0 \quad \forall\, j$ s.t. $|\alpha_j| \leq \sigma_k$ *<thresholding>*

Ensure: an approx. solution of $s = \Phi\alpha$ s.t. $\| \alpha \|_0 \leq k$

It should be noted that the last step of the algorithm accomplishes thresholding of the final point carried out by the while loop because in some cases it can have some noise among the null coefficients. However, experimentally has been found that such coefficients, reach arbitrary close to zero values as the number of loops increases, making the threshold step not every useful.

K-LiMapS algorithm requires a dictionary $\Phi \in R^{nXm}$, its pseudo-inverse Φ^t, a signal $s \in R^n$ and a sparsity level k. So, first of all, it was necessary to create them. The user inputs are matrix dimensions m, n with $m > n$, the sparsity level $k < m$ and the number of maximum iterations. The algorithm is first split into two biggest functions:

1. Function that creates the dictionary, computes its pseudo-inverse and the required signal s

(a) Creation of an $m-$size alpha signal with k values different than 0, then randomly permuted

(b) Creation of a nXm random dictionary D, following the Gaussian normal distribution

(c) Normalization of the dictionary

(d) Computation of its pseudoinverse D_{inv}

(e) Computation of signal s multiplying the dictionary with the previously created alpha signal

2. Function that computes k-LiMapS with the resulting alpha, in which only k elements are different than 0.

   (a) Initialization of alpha multiplying the dictionary pseudo-inverse D_{inv} and the signal s

   (b) Sorting of alpha in descending order to choose λ

   (c) Loop to increase sparsity level

      i. Application of sparsity contraction mapping, see formula (2)

      ii. Application of the orthogonal projection

      iii. Update of the lambda coefficient, chosen from the new alpha sorted into descending order

      iv. Check of the stopping criteria, computing the norm of the difference of the previous and new alpha or a *nan* lambda.

   (d) Final refinements to remove noisy values and apply once again the sparsity contraction mapping on the non-zero elements

3.1.1 Complexity

The algorithm complexity is $O(nm)$ times the number of iterations in k-LiMapS.

3.2 Parallel version

The implementation of the parallel algorithm preserves the sequential skeleton, which performs sequential calls of different helper functions. Hence, the changes were mainly done in external kernels such as matrix Transpose, matrix Multiplication, and others. A few small functions are still performed in CPU, not worth the effort of moving data to the GPU. A time improvement was achieved by adding the shared memory for operations where data are accessed more than once, so matrix Transpose and Multiplication. However, the best results came with adding streams for all GPU kernels, because they are primarily a sequence of single-valued operations for each cell that can be easily split into different streams.

4 Implementation

As previously anticipated, the main idea for the parallel implementation of the k-LiMapS algorithm relies upon a faster execution of small kernels that are focused on array and matrices elaboration, which, for big dimensions, are more GPU oriented. To simplify certain operations, some libraries were used, such as Thrust [4], GSL (GNU scientific library) [6], cuRand [2] and cuSolver [3]. Thrust is a *C++* template library for *CUDA* that allows you to implement high-performance parallel applications with minimal programming effort through a high-level interface that is fully inter-operable with *CUDA C*. Thrust provides a rich collection of data-parallel primitives such as sort and max. These functions are used to sort the alpha array, find the maximum element, and compute the permutation of the elements in the array. The GNU Scientific Library (GSL) is a numerical library for *C++* and *C++* programmers. It is free software under the GNU General Public License. The library provides a wide range of mathematical routines such as random number generators, special functions, and least-squares fitting. There are over 1000 functions in total with an extensive test suite. It was used in the sequential version to obtain the SVD decomposition in order to compute the pseudo-inverse. The cuRAND library provides facilities that focus on the simple and efficient generation of high-quality pseudorandom and quasirandom numbers. A pseudorandom sequence of numbers satisfies most of the statistical properties of a truly random sequence but is generated by a deterministic algorithm. A quasirandom sequence of n-dimensional points is generated by a deterministic algorithm designed to fill an n-dimensional space evenly. cuRAND consists of two pieces: a library on the host (CPU) side and a device (GPU) header file. Random numbers can be generated on the device or the host CPU. For device generation, calls to the library happen on the host, but the actual work of random number generation occurs on the device. The resulting random numbers are stored in global memory on the device. Users can then call their kernels to use the random numbers, or they can copy the random numbers back to the host for further processing. The second piece of cuRAND is the device header file. This file defines device functions for setting up random number generator states and generating sequences of random numbers. This allows random numbers to be generated and immediately consumed by user kernels without requiring the random numbers to be written to and then read from global memory. The library was used to randomly generate the dictionary. The cuSolver library is a high-level package based on the cuBLAS and cuSPARSE libraries. It consists of two modules corresponding to two sets of API: the cuSolver API on a single GPU and the cuSolverMG API on a single node multi-GPU. Each of these can be used independently or in concert with other toolkit libraries. CuSolver intends to provide useful LAPACK-like features, such as common matrix factorization and triangular solve routines for dense matrices, a sparse least-squares solver, and an eigenvalue solver. In addition, cuSolver provides a new refactorization library useful for solving sequences of matrices with a shared sparsity pattern. In this project, it was used to compute matrix SVD in the accelerated version of the algorithm.

4.1 Data structures

All data are stored into floating points arrays or matrices. This code requires a lot of supporting temporary other data structures of the same size or lower, but always nXm as maximum. To create the dictionary, data are randomly sampled from a Gaussian standard distribution. The output of the process is a m-size array alpha with $k < m$ non-zero values. Unified memory with UVA was used, because some GPU operations needed to co-work with CPU processes, hence, its use simplified values addressing.

4.2 Steps

The implementation of the algorithm was divided into 2 main functions. One creates the dictionary and the input signal s while the other one takes care of computing the actual algorithm. Both functions, in turn, invoke other small kernels to do different elaboration stages. The biggest of them is the one in charge of calculating the pseudo-inverse, which is a more complex host function including different steps to obtain the desired result.

4.2.1 Pseudo-inverse

To compute the Moore-Penrose pseudo-inverse of a matrix, it's first necessary to perform its singular value decomposition (SVD) $A = U\Sigma V^T$. To do it we first need to check that the number of rows is bigger than the number of columns, otherwise we have to transpose the matrix and then transpose it back again once the resulting matrix is obtained. With the SVD, using the cuSolver library, U, s, V are obtained. The pseudo-inverse is $A^{-1} = V\Sigma^{-1}U^T$, where ^{T} indicates transpose and Σ^{-1} is obtained by taking the reciprocal of each nonzero element on the diagonal, setting all other values to zero.

4.3 Kernels

When invoking kernels, the size of the grid and block is redefined based on the size of the input, because the same kernel can be invoked at different points of code with different data. When managing matrices, a 2D grid and block ($32x32$) are used, while for managing arrays, a 1D grid and block (1024). When shared memory is used, the size of the shared block is predefined and can be 16 or 32.

1.
```
__global__ void generate_array( curandState*
↪  globalState, float * result, int count );
```

   It invokes the *setup_kernel (curandState * state, unsigned long seed, int n)* to initialize the thread status and generate a random float value to fill the alpha signal.

2.
```
__global__ void rand_gen_gpu(float *dict, curandState
↪  *states, int nRows, int nCols);
```

It initializes and generates a random float sampled from a normal distribution to fill the dictionary.

3.
```
__global__ void transpose(Smem)(float *in, float *out,
↪  int nrows, int ncols);
```

It computes the transpose of the matrix, and, in its advanced version, it uses shared memory to avoid non-coalescing accesses to global memory, reading and saving a line into the shared memory and then reading a column from it to write values back to global memory.

4.
```
__global__ void matrixMultStream(float* A, float* B,
↪  float* C, int row1, int col1, int col2, uint
↪  offset);
__global__ void matProdSMEMdynamic(float* A, float* B,
↪  float* C, int row1, int col1, int col2, const uint
↪  SMEMsize, uint offset);
```

It computes the matrix multiplication. In one advanced version, it uses shared memory to speed up the process and store the result of each block to later sum all of them up. In another version, it exploits different streams to split the work into blocks. There's also a solution where both are applied, but it still has coordination problems for big dimension matrices.

5.
```
__global__ void elemWise_mult(float *A, float *B, float
↪  *C, int numElements, uint offset);
```

It computes the element-wise matrix multiplication, where each thread multiplies the values of the same cell index in the two input matrices and saves it in the resulting one.

6.
```
__global__ void abs_array (float *arr, int dim, uint
↪  offset);
```

Each thread accesses a single cell and, if it's necessary, changes the sign of the value to make it positive.

7.
```
__global__ void copy_arr (float *src, float*dest ,int
↪  dim, uint offset);
```

It serves to speed up the copy of an array, operation needed in the process to store old values which will later be modified, but that are still needed to confront the previous configuration with the new one.

8.
```
__global__ void copy_matrix(float *src, float *dest, int
↪  nRows, int nCols, uint offset);
```

It serves to speed up the copy of an array, operation needed in the process to store old values which will later be modified, but that are still needed in their original configuration.

9. `__global__ void matrixDiff(float *A, float *B, float *C, int dim, uint offset)`

   It computes the matrix difference, where each thread subtracts the values of the same cell in the two input matrices and saves it in the resulting one.

10. `__global__ void arr_preProc(float *A, int dim, uint offset);`

    It serves to speed up an array pre-processing needed for the computation of the algorithm, where each cell is first changed in sign and then is elevated.

11. `__global__ void subMatrix(float *A, float*B, int *index, int nRows, int nCols);`

    It fills matrix A with cells of matrix B satisfying requirements expressed by the indices array. Each thread has to access these three cells.

12. `__global__ void array_initialize(float *tmp_lambaMat, float lambda, int dim, uint offset);`

    Each thread initializes a cell of the array with the same value.

Most of these kernels apply simple value elaborations in one cell, independently from the others. That is why splitting the work into 4, or 6, till 8 streams decreases the amount of computation time. Shared memory is only convenient for transpose and matrix multiplication, because they are the only kernels where values need to be accessed more than once, otherwise the time it takes to copy the values to the shared memory is not worth it.

5 Speedup

The Parallel k-LiMapS algorithm was tested via Google Colab Pro. The specs of this machine are: CPU Intel (R) Xeon (R) @ 2.20GHz and GPU Tesla P100-PCIE-16GB.

The sequential version of the algorithm, like the parallel one, was calculated by creating and invoking a Cuda event before and after each operation and then calculating their difference to get the number of milliseconds. Even if the algorithm includes a for loop with a number of maximum iterations, the shown results consider only one cycle, to be sure that all versions execute the same amount of operations.

Size	**CPU**	**Naive GPU**	**Speedup**
N = 128 M = 256 K = 20	63,654 ms	56,140 ms	1,13
N = 256 M = 512 K = 20	604,312 ms	450,776 ms	1,34
N = 600 M = 700 K = 20	2517,59 ms	1845,776 ms	1,36
N = 800 M = 1280 K = 32	23386,039 ms	702,169 ms	33,30

Table 1: Speedup of the execution w.r.t. CPU time of the entire algorithm and the naive Cuda version

As expected, especially for low-dimension matrices, the gain is very low.

Size	**CPU**	**SMEM Added**	**Speedup**
N = 128 M = 256 K = 20	63,654 ms	47,047 ms	1,35
N = 256 M = 512 K = 20	604,312 ms	428,790 ms	1,41
N = 600 M = 700 K = 20	2517,59 ms	1597,073 ms	1,58
N = 800 M = 1280 K = 32	23386,039 ms	550,184 ms	42,51

Table 2: Speedup of the execution w.r.t. CPU time of the entire algorithm and the addition of shared memory

Shared memory, in this project, turned out to be not that significant, leaving almost the same computation times compared to the naive version.

Size	**CPU**	**Streams Added**	**Speedup**
N = 128 M = 256 K = 20	63,654 ms	28,032 ms	2,27
N = 256 M = 512 K = 20	604,312 ms	350,611 ms	1,7
N = 600 M = 700 K = 20	2517,59 ms	387,187 ms	6.5
N = 800 M = 1280 K = 32	23386,039 ms	455,940 ms	51,29

Table 3: Speedup of the execution w.r.t. CPU time of the entire algorithm and the use of streams

Considering the inherent sequentiality of the algorithm, with a series of sequential steps which rely on numbers obtained in the previous step, a great speedup cannot be expected. What emerges from the last line is how fast cuSolver library manages to perform SVD decomposition wrt GSL library.

5.1 Improvements

In this algorithm, there are a couple of points where could be possible to split the computation into two CPU threads independent of each other, one that computes the pseudo-inverse of the matrix D or $D1$ and the other that proceeds with matrices and arrays processing until both branches are needed to move forward in the process.
Another version of the algorithm was implemented, that combines the use of streams with shared memory in the matrix multiplication, but it was not included because it only works with very low-dimensional inputs, raising many problems of memory management.
It could also be possible to create new kernels to implement some of the operations computed in CPU, but should be first evaluate the actual speedup and if it worth the time needed to transfer the data to GPU.

References

[1] Alessandro Adamo and Giuliano Grossi. "A fixed-point iterative schema for error minimization in k-sparse decomposition". In: *2011 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)* (2011), pp. 167–172.

[2] NVIDIA Corporation. *cuRAND - The API reference guide for cuRAND, the CUDA random number generation library.* URL: `https://docs.nvidia.com/cuda/curand/index.html`. (Last updated January 12, 2022).

[3] NVIDIA Corporation. *cuSOLVER - The API reference guide for cuSOLVER, a GPU accelerated library for decompositions and linear system solutions for both dense and sparse matrices.* URL: `https://docs.nvidia.com/cuda/cusolver/index.html`. (Last updated January 12, 2022).

[4] NVIDIA Corporation. *Thrust - The API reference guide for Thrust, the CUDA C++ template library.* URL: `https://docs.nvidia.com/cuda/thrust/index.html`. (Last updated November 23, 2021).

[5] Vittorio Cuculo et al. "Robust Single-Sample Face Recognition by Sparsity-Driven Sub-Dictionary Learning Using Deep Features". In: *Sensors* 19.1 (2019). ISSN: 1424-8220. DOI: `10.3390/s19010146`. URL: `https://www.mdpi.com/1424-8220/19/1/146`.

[6] Mark Galassi et al. *GNU Scientific Library - Reference Manual, Third Edition, for GSL Version 1.12 (3. ed.).* Jan. 2009. ISBN: 978-0-9546120-7-8.