Pseudo-Reconfigurable Heterogeneous Solution for Accelerating Spectral Clustering

Mihaela Maliţa
Computer Science Dept.
Smith College
Northampton, MA 01063, US
mmalita@smith.edu

George Vlăduţ Popescu
Electronic Devices, Circ. and Arch. Dept.
Politehnica University of Bucharest
Bucharest, Romania
georgevlad.popescu@yahoo.com

Gheorghe M. Ştefan
Electronic Devices, Circuits and Arch. Dept.
Politehnica University of Bucharest
Bucharest, Romania
gheorghe.stefan@upb.ro

***Abstract*—Spectral clustering is a Machine Learning technique intensively used in Big Data applications. It makes extensive use of linear algebra. This article introduces the concept of MapReduce Accelerator (MRA) as the reconfigurable part of a heterogeneous computing system. Although the accelerator we propose is a general purpose one, it has some specific features related to the targeted application. This is possible due to the pseudo-reconfigurable environment which deploys in FPGA a parameterizable programmable accelerator. The main specific characteristics of the accelerator are proposed as a result of the analysis performed on the spectral clustering algorithms. The architecture is described and the spectral clustering algorithms are evaluated. The proposed solution is compared, in terms of computing performance and energy consumption, with other solutions published in the literature. The increase in computing performance is accompanied by a 3-5 times reduction in energy consumed. The accelerator is a linear array of cells controlled by a sequencer loop closed through a reduction network. Each cell is a simple, accumulator-based execution unit with a big two-port register file. The reduction network is a *log*-depth pipelined circuit performing few reduction functions such as add, min, max. The experimental system is a PYNQ-Z2 board equipped with Zinq 7020 SoC; it is used to implement and evaluate the acceleration provided by an 128-cell MRA.**

***Index Terms*—Spectral clustering, parallel algorithm, parallel computing, accelerator, heterogeneous computing, pseudo-reconfigurable computing.**

I. Introduction

Clustering is an unsupervised Machine Learning technique used in finding a structure in a collection of objects. Thus, a ***cluster*** is a group of objects *similar* between them and *dissimilar* to the rest of the objects which belong to other clusters. There are two main types of clustering algorithms: (1) *Euclidean distance-based* clustering: the objects belong to the same cluster if they are *close* according to a given distance, and (2) *Hamming distance-based* clustering: objects are grouped according to the match of their attributes usually expressed by the Hamming distance. In both cases a similarity $n \times n$ matrix S for a number of n objects is defined. Then, S is submitted to various computational procedures to make the requested partitions. One of these algorithms is the spectral clustering algorithm.

Spectral clustering is a popular unsupervised Machine Learning algorithm which often outperforms other approaches by its accuracy. In addition, spectral clustering is very simple to implement and can be solved efficiently by standard linear algebra methods. In spectral clustering, the affinity, and not the absolute location (i.e. k-means), determines what points fall under which cluster. It is particularly useful in tackling problems where the data, the group of objects, forms complicated configurations.

Despite its good performance, the spectral clustering algorithm is difficult to implement for large set of objects due to its high computational complexity and energy demand. Indeed, for a data set consisting of n data points, we need to compute the $n \times n$ adjacency matrix in time $\in O(n^2)$, and calculate the eigen-decomposition of the resulting Laplacian in time $\in O(n^3)$ (due to the inverse matrix computation involved). Both of these two steps are computational expensive, in both time and energy, and thus unbearable for large-scale applications.

In the last decades, the researchers have been focused to accelerate the algorithm. One option was to find methods for reducing the computational cost of the eigen-decomposition of the graph Laplacian for efficiently computing an *approximate* solution of the eigenproblem [4]. Another option is the reduction of the data size beforehand by replacing the original data set with a smaller number of data points, conducting to a much smaller number of operations performed on the resulting adjacency matrix [14]. These approaches, and other similar [11] [17] [10], lead to an elegant balance between running time and accuracy.

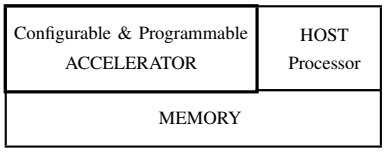

Fig. 1. Pseudo-reconfigurable heterogenous computing system.

In this paper we do not contribute to the improvement of the spectral clustering algorithm. Instead, we offer a solution to speed up the calculation, without any loss of accuracy, by using an accelerator in a pseudo-reconfigurable heterogeneous

computing system (see Figure 1). The speed up will be achieved in an energy aware structure.

The computational stages involved in this application are not only very intense computationally, but some of them involve an unpredictable, usually big, number of iterations. Thus, in comparing various accelerators used to implement the algorithm, it is hard to use information published in the literature due to the fact that figures depend on the input data which determines the number of iterations. Therefore, in this paper we are focused in providing an efficient accelerating method working on an energy aware hardware with an appropriate instruction set architecture.

In the following, we consider the possibility to use a heterogenous computing system having a p-core accelerator implementable in two possible versions:

1) as an IP with ten thousands of cells integrated on the same silicon die with the host engine using it
2) as an FPGA implemented parameterized accelerator of up to thousands of cells integrated in a pseudo-reconfigurable computing system ("pseudo", because it is a programmable system which is instantiated once, with appropriate parameters, in FPGA and called by the host computer for functions implemented by the programs it contains).

Why another type of accelerating system for spectral clustering? First of all, the off-the-shelf solutions are optimized for small matrix operations (involved in graphics) and their energy consumption is very high. The second reason is given by the opportunity to use FPGAs or ASICs to adopt an appropriate architectural and structural approach, flexible for the FPGA version, and highly optimized as a general purpose accelerator for the ASIC version.

The next section emphasizes the main functional features involved in computing the spectral clustering for a set of n objects. The third section describes the structure of the MapReduce Accelerator (MRA) and the associated instruction set architecture. The fourth section is about how the main functions involved in spectral clustering algorithms are accelerated using the architecture we proposed. The fifth section elaborates on the use of the accelerator for big data input. The sixth and seventh sections describe and comment the implementation of the heterogenous system based on our MRA in two versions: FPGA-based and ASIC-based. In the eight section, the state of the art is shortly investigated and our solution is compared with those designed using off-the-shelf devices.

II. Functional requirements for spectral clustering

The bi-partitioning spectral clustering algorithm has the following main steps:

1) define the similarity measure between the elements of the n-object set
2) compute the $n \times n$, distance-based or attribute-based, symmetric similarity matrix $\mathbf{S}$,
3) compute the eigenvector corresponding to the second smallest eigenvalue, called the Fiedler vector, which defines the spectral graph bi-partitioning by the sign of the scalar it contains.

We describe the spectral clustering algorithm, in the first subsection, in order to emphasize, in the second subsection, the functions to be accelerated by the accelerator we propose.

A. Algorithm

The bi-partitioning clustering algorithm is applied to the similarity matrix $\mathbf{S}$:

$$\mathbf{S} = \begin{bmatrix} s_{11} & \dots & s_{1n} \\ \vdots & \ddots & \vdots \\ s_{n1} & \dots & s_{nn} \end{bmatrix}$$

considered as the representation of a graph having in its vertexes elements of the set of objects

$$O = \{o_1, \dots, o_n\}$$

Each edge of the graph is marked by the distance, s_{ij}, between the elements it connects. The matrix is symmetric and the main diagonal contains only zeroes. Starting from this matrix, its Laplacian defines a bi-partition by computing the Fiedler vector. The bi-partition is defined by the signs of the scalars of the Fiedler vector. The algorithm is well known (see [2] [3]).

ALG. 1: Spectral bi-partitioning algorithm

1) build the Laplacian matrix of $\mathbf{S}$,
$$\mathbf{L} = \mathbf{D} - \mathbf{S}$$
where $\mathbf{D}$ is the *degree matrix* of $\mathbf{S}$ defined as the diagonal matrix with
$$d_{ii} = \sum_{j=1}^{n} s_{ij}$$
2) compute the Fiedler vector: the eigenvector corresponding to the second smallest eigenvalue obtained by solving the equation
$$\mathbf{L}\mathbf{v} = \lambda \mathbf{v}$$
where $\mathbf{L}$ is an $n \times n$ matrix, $\mathbf{v}$ is a non-zero n-component vector and λ is a scalar
3) use the resulting Fiedler vector to make the bi-partition upon the sign of its scalar components.

For an efficient and accurate method to compute the Fiedler vector, based on the Householder deflation and the inverse power iteration, we must take into account the following assumptions:

1) The power method is a direct iteration method for obtaining the dominant eigenvalue (i.e. the largest in magnitude) for a given matrix $\mathbf{A}$ and also the corresponding eigenvector.

2) If the matrix $\mathbf{A}$ has an eigenvalue λ and the associated eigenvector X, then the matrix $\mathbf{A}^{-1}$ has an eigenvalue $1/\lambda$ with the associated eigenvector X.
3) If $\mathbf{A}$ has the dominant eigenvalue λ then its inverse $\mathbf{A}^{-1}$ has an eigenvalue $1/\lambda$, which will be the smallest magnitude eigenvalue of $\mathbf{A}^{-1}$
4) $\mathbf{L}$ is symmetric and positively semi-defined
5) The Laplacian matrix $\mathbf{L}$ has n positive real eigenvalues, $0 = \lambda_1 < \lambda_2 < \ldots < \lambda_n$
6) The sum of lines of $\mathbf{L}$ is 0 which implies

$$(det(\mathbf{L}) = 0) \Rightarrow \nexists\, \mathbf{L}^{-1}$$

7) The eigenvectors of a matrix $\mathbf{A}$ are identical with the eigenvectors of its inverse, $\mathbf{A}^{-1}$.

The algorithm we use for computing the Fiedler vector, following [6] [19], is:

ALG. 2: Fiedler vector for a $n \times n$ Laplacian L

1) compute the scalar $\alpha = n + n^{1/2}$ and build the vector
$$u = [1 + n^{1/2}, \underbrace{1, 1, \ldots, 1}_{n-1}]^T$$
2) compute the vector: $v = h - \gamma u/2$, where:
$$h = \mathbf{L}u/\alpha$$
$$\gamma = u^T h/\alpha$$
3) extract the vectors: $r = u[2:n]$ and $s = v[2:n]$ and build the matrix:
$$\mathbf{L}_2 = \mathbf{L}_{2:n,2:n} - rs^T - sr^T$$
having the eigenvalues of $\mathbf{L}$ except the smallest one, which is 0
4) compute $\mathbf{L}_2^{-1}$
5) apply the inverse power method to $\mathbf{L}_2^{-1}$ to obtain the eigenvector corresponding to the smallest eigenvalue, following the next steps:
   a) instantiate the first iteration for the $n-1$-component vector X^0, randomly or so: $\underbrace{1, 1, \ldots, 1}_{n-1}$
   b) $X^0_{normed} = X^0/norm$
   c) **until** $\Delta X_{normed} = X^{i+1}_{normed} - X^i_{normed} < error$ **do**:
      i) $X^{i+1} = \mathbf{L}_2^{-1} X^i$
      ii) $X^{i+1}_{normed} = X^{i+1}/norm$
   d) return X^{i+1}_{normed}
6) compute the Fiedler vector:
$$X = [0, X^{i+1}_{normed}]$$
$$F = X - (u^T X u)/\alpha$$

B. Functional Requirements

The previously described algorithm supposes the following parallel computational patterns [12] in order to be implemented efficiently using linear algebra operations:

- **Map Pattern**: a given function is **map**ped or is applied-to-all elements of vectors of data, returning vectors (for example, in computing the degree matrix, Gauss-Jordan elimination, matrix-vector multiplication)
- **Predicated Map Pattern**: the mapped function is applied only to the vector components selected according to the binary values distributed along a Boolean vector whose value is computed using the map pattern (for example, searching for all the occurrences of a given value in a vector, searching for the first occurrence of a value in a vector, selecting a column in a matrix as in the Gauss-Jordan elimination method).
- **Reduction Pattern**: a given associative and commutative operation (add, min, max, for example) uses as argument all, or selected, elements of a vector, returning a scalar (for example, in matrix-vector multiplication, in computing the sum of the selected components of a vector, or in searching the maximum value in a sequence).

It seems that a kind of Map-Reduce structure is outlined.

III. MapReduce accelerator's structure

A. Structure

The previously emphasized functional requirements leads to an accelerator (see Figure 2) with a linear array of p cells connected in a loop with a controller through a reduction network (various versions of this kind of structure are already implemented in silicon [15] [9], while the theoretical foundation for this proposal is presented in [16]). The main components of the accelerator are:

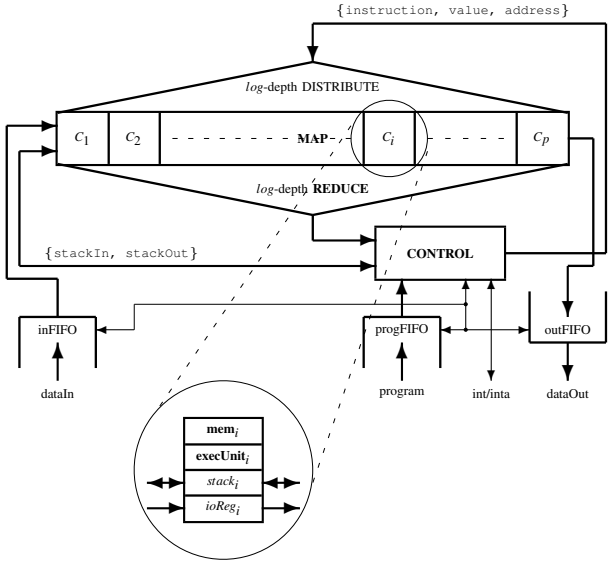

Fig. 2. Programmable accelerator as a linear array of cells connected in a loop with a controller through a reduction unit.

- MAP: a linear array of cells which receives from the CONTROL unit in each clock cycle, through a log-depths DISTRIBUTE network, an instruction accompanied by a value and an address
- CONTROL: is a mono-core processing unit with data memory and program memory from which in each cycle is fetched a pair of instructions, one for CONTROL and another for the MAP array where it is executed in each active cell
- REDUCE: is a log-depth pipelined circuit performing few reduction functions; it provides, with a log-latency the sum of the scalars provided by the active cells of the MAP array, or the max/min value selected from the same source
- three FIFOs, two – inFIFO and outFIFO – for data exchange with the system memory and one – progFIFO – to receive programs and commands from the host computer

Each cell in the MAP array contains:

- $execUnit_i$: a w-bit accumulator-based execution unit
- mem_i: a two-port big register file of 1 to 4 K of w-bit words
- $stack_i$: the component of a stack distributed along the cells; the stack is accessed at the left end and in parallel for read and write from/to the accumulator of the execution units in each cell
- $ioReg_i$: is the component of the serial register used to get and send vectors from/to the external memory through inFIFO and outFIFO.

The main issue raised by the structure is the log-cycle latency introduced by the REDUCE net. It is addressed almost any times by an appropriate use of the stack distributed along the MAP array.

B. Variables

The computational storage resources distributed in the MAP array are the following:

- the matrix
$$\mathbf{M} = \begin{bmatrix} v_{11} & \dots & v_{1p} \\ \vdots & \ddots & \vdots \\ v_{m1} & \dots & v_{mp} \end{bmatrix}$$
representing the data distributed in the local data memories, mem_i, of the MAP array, organized as:
  - horizontal vectors, distributed along the cells, represented as:
$$V_i = \begin{bmatrix} v_{i1} & \dots & v_{ip} \end{bmatrix}$$
for $i = 1, \dots, m$
  - vertical vectors in each cell:
$$W_i = \begin{bmatrix} v_{1j} & \dots & v_{mj} \end{bmatrix}$$
for $j = 1, \dots, p$
- $A = [a_1 \dots a_p]$: the address vector used for relative addressing in each local memory mem_i
- $B = [b_1 \dots b_p]$: the Boolean vector used to select the active cells in the MAP array (for $b_i = 1$ the i-th cell is active)
- $IX = [1 \dots p]$ the index vector, whose components are wired in each cell, used to identify the cell
- $STACK = [stack_1 \dots stack_p]$: the stack vector used to implement two functions:
  - a stack accessed at its left end
  - a shift register distributed along the cells
- $IOREG = [ioReg_1 \dots ioReg_p]$: the input-output register connected to the two FIFOs involved in the data transfer.

C. Instruction Set Architecture

The parallel operations performed in the MAP array and the REDUCE network, defined on horizontal vectors V_i, for $i = 1, 2, \dots, m$, are mainly the following:

- ***predicated binary or unary map operations***:
$$V_i \Leftarrow BOP(V_k, V_q) ::= v_{ij} \Leftarrow b_j \; ? \; BOP(v_{kj}, v_{qj}) \; : \; v_{ij}$$
$$V_i \Leftarrow UOP(V_k) ::= v_{ij} \Leftarrow b_j \; ? \; UOP(v_{kj}) \; : \; v_{ij}$$
where BOP is a binary operation (ADD, SUB, MULT, AND, OR, XOR, ...) and UOP is a unary operation (INC, NOT, SHIFT, ...)
- ***spatial selections***, acting on the content of the Boolean vector B; the simplest set of spatial selection functions is:
$$WHERE(cond) ::= b_i \Leftarrow cond_i \; ? \; 1 \; : \; 0$$
$$ENDWHERE ::= b_i \Leftarrow 1$$
where: $cond = \{CARRY, ZERO, SGN, FIRST, \dots\}$

for $j = 1, \dots, p$, performed in time $\in O(1)$.

- ***reduction operations***, defined on the active components of vectors from the MAP array, with values in a scalar stored in the location i of the CONTROL data memory:
$$acc \Leftarrow redOP(V_i) ::= acc \Leftarrow OP_{j=1}^{p} b_j \; ? \; s_{ij} \; : \; NE$$
where: NE is the neutral element of the operation OP and acc the accumulator of CONTROL; performed with a latency $\in O(log\ p)$.
- ***data transfer operations***: between $\mathbf{M}$, the horizontal vectors distributed along the MAP array in mem_i, for $i = 1, \dots, p$, and the system memory through the two data FIFOs:
$$LOAD(i,k) ::= [v_{i1} \dots v_{ik}] \; is \; popped \; from \; inFIFO$$
$$STORE(i,k) ::= [v_{i1} \dots v_{ik}] \; is \; pushed \; in \; outFIFO$$
for $k \leq p$, performed in time $\in O(k)$.

IV. A Kernel Library of Functions for Spectral Clustering

The MRA is part of a pseudo-reconfigurable computing system working under the control of a host computer. It is "pseudo", because it is configured only once at the beginning of the program to be accelerated. Configuring means two steps:

- load the parameterized design: parameters, such as w, m, or p, and features, such as specific BOP, UOP, or OP functions, are set, at the synthesis moment, depending on the accelerated program
- load the programs: in the CONTROL's program memory are loaded the programs which implement the kernel function library used to implement on the HOST (see Figure 3) the library of functions requested by the current application.

In a pure reconfigurable computing system, during the program execution various circuits are instantiated in the FPGA associated to the host computer. In our pseudo-reconfigurable approach, the accelerator is a parameterized programmable parallel engine. It is instantiated in FPGA with the programs performing the functions to be accelerated.

In this section the main functions of a ***kernel library*** used to accelerate the spectral clustering are defined, described and evaluated. These kernel functions are used to develop, on the host computer, the ***library*** of functions for accelerating the spectral clustering. The kernel library, developed in assembly language, works with data structures with the size limited by the hardware implementation of the accelerator, while the library is developed, in a high level language, to support data structure with any size.

A. The Control Path

The accelerator is controlled using the 32-bit `program` input and the one-bit interrupt signal. On the program path the host computer sends two types of commands:

- `INIT`: used to load the initial programs in the memory of CONTROL as a self-delimiting string of 32-bit words
- `FUNCTION_NAME`(`PARAM_1`,...,`PARAM_q`), with $q = 0, 1, \ldots$, used to run one of the programs loaded by `INIT`; it is a string of $q+1$ 32-bit words.

After `LOAD` or after the run of a program triggered by a function `FUNCTION_NAME`(...) MRA gets in the halt state waiting for the occurrence of a new function at the progFIFO output (see Figure 2).

B. Functions

The main functions implemented for our MRA to support spectral clustering are described in the following. The accelerations provided by MRA are evaluated for $p \geq n$, i.e., for the cases when the data structures, matrices and vectors, can be loaded entirely in the memory distributed along the cells in the MAP array.

In the following a matrix of $p \times p$ elements is specified by the index, belonging to $[1,m]$, of its first horizontal vector in the $m \times p$ matrix $\mathbf{M}$.

- `SIM`(`dest`,`source`): in the matrix `dest` in $\mathbf{M}$, is computed the similarity matrix of the objects contained in the vector `source`; `source` $\in [1,\mathtt{m}]$. The acceleration $\in O(p)$.
- `LAPLACE`(`name`): replace of the matrix `name` with its Laplacian. The acceleration $\in O(p)$.
- `INV`(`dest`,`source`): computes in the matrix `dest` the inverse of the matrix `source`. The acceleration $\in O(p)$ using Gauss-Jordan elimination method.
- `MVMULT`(`dest`,`left`,`right`): computes in the horizontal vector `dest` the product of the matrix `left` with the horizontal vector `right` $\in [1,m]$. The acceleration $\in O(p)$.
- `SQMULT`(`dest`,`left`,`right`): computes in the matrix `dest` the product of the matrices `left` and `right`. The acceleration $\in O(p)$.
- `SQMACCM`(`dest`,`left`,`right`): add to the matrix `dest` the product of the matrices `left` and `right`. The acceleration $\in O(p)$.
- `SQSUB`(`dest`,`left`,`right`): computes in the matrix `dest` the difference of the matrices `left` and `right`. The acceleration $\in O(p)$.
- `EIGENVEC`(`dest`,`source`): computes, using the power method, in vector `dest` the biggest eigenvector of the matrix `source`. The acceleration $\in O(p)$; the actual time is given by the number of iterations determined by the initial values of the matrix.
- `GET`(`dest`,`size`): gets from inFIFO in $\mathbf{M}$, starting with the horizontal vector `dest`, a `size` $\times$ `p` matrix. The execution time $\in O(size)$.
- `SEND`(`source`,`size`): send to outFIFO a `size` $\times$ `p` matrix stored in $\mathbf{M}$ starting with horizontal vector `source`. The execution time $\in O(size)$.

Unfortunately, the last two functions, `GET` and `SEND` are not accelerated. The curse of the *"von Neumann Bottleneck"* follows us relentlessly.

The accelerations claimed for the previously listed functions are obvious, except for those involving in their execution the matrix-vector multiplication: `MVMULT`, `SQMULT`, `SQMACCM`, `EIGENVEC`. Indeed, if vector multiplication can be distributed in the MAP array and executed in constant time, the sum, supposed for the inner product, contributes with a latency in $O(log\ p)$, thus limiting the acceleration to $O(n/log\ p)$. To avoid this latency, the STACK distributed along the MAP array is used. With this feature, the matrix-vector multiplication is performed using the following program:

```
/*****************************************
MATRIX-VECTOR MULTIPLICATION
   Vi,...,Vj: lines of the matrix
   W: multiplier vector
   Y: result vector
   matrix size: (j-i+1) x p
*****************************************/
(1) for (k=i; k<(i+j+1); k=k+1)
      PUSH(redAdd(W x Vk)) // push in STACK
(2) wait (log p) cycles // due to latency
(3) Y <= STACK
```

Step (1) is performed in a pipeline manner. The multiplication is executed in the MAP section, then the resulting vector enters in the first pipeline level of the REDUCE network while the next multiplication is executed, and so on. After $log_2 p$ clock cycles, the output of the REDUCE network sends the first sum to the input of STACK. After the last multiplication we must wait $log_2 p$ clock cycles for the inner product to propagate through the REDUCE network. Thus, the latency of the REDUCE network bothers us ***only once*** when waiting for the result of the last multiplication. The execution time is

$$T_{matrixVectorMult} = 2(j-i+1) + log_2 p \in O(j-i+1)$$

and the overall acceleration for `MVMULT`, `SQMULT`, `SQMACCM`, `EIGENVEC` $\in O(p)$.

The total execution time is given by the transfer time plus the processing time. The transfer time (inputing the vector of objects, O, and outputing the Fiedler vector, F, both having n components) is in $O(n)$ with the associated constant smaller than 1. Theoretically, the computing time is in $O(n^2)$, for $n \leq p$, due to the inverse matrix operation (see ALG. 2, step 4). But the actual time could be given by the number of iterations executed for EIGENVEC function. Therefore, experimentally, the acceleration depends on the actual content of the **S** matrix, because the convergent process in applying the inverse power method (see ALG. 2, step 5) is performed in a number of iterations which varies in a very large domain. The only provable performance is the **acceleration** $\in O(p)$ provided for $n \leq p$.

V. Implementing Spectral Clustering for $n > p$

When $n > p$, the transfer time between the HOST's MEMORY and the distributed memory in ACCELERATOR (see Figure 1), represented as the matrix **M**, makes his mark on the overall performance. The matrices **S**, **D** and **L** are bigger than the space defined by the matrix **M**, consequently these matrices must be kept in the system's memory, MEMORY, and loaded *tile by tile* in MRA to make the computation. Therefore, the overall acceleration is limited by the fact that the data transfer time between the system's memory and the internal memory of MRA is accelerated only by $O(1)$ times.

In the application we consider, the spectral clustering, there are two cases when a tile of size $s \times s$ is loaded in **M** from MEMORY:

1) the computation on the tile(s) is performed in $O(s^2)$ accelerating an $O(s^3)$ computation $O(s)$ times
2) the computation on the tile(s) is performed in $O(s)$ accelerating an $O(s^2)$ computation $O(1)$ times, because the transfer time remains in $O(s^2)$.

The spectral clustering computation is dominated by two stages in ALG. 2: stage 4 (compute $\mathbf{L}_2^{-1}$) and stage 5 (apply the inverse power method to $\mathbf{L}_2^{-1}$). The stage 4 corresponds to the case 1, while the stage 5 to the second case.

The overall acceleration is approximated, considering only these dominant stages, to:

$$\alpha \simeq \frac{t_{inv} + t_{pow}}{t_{inv}/\alpha_{inv} + t_{pow}/\alpha_{pow}} \tag{1}$$

where: t_{inv} and t_{pow} are the execution times for stage 4 and stage 5 on a mono core computing engine, while α_{inv} and α_{pow} are the corresponding accelerations provided by MRA.

Because $\alpha_{inv} \in O(p)$, while $\alpha_{pow} \in O(1)$ we can approximate α by:

$$\alpha \simeq (\frac{t_{inv}}{t_{pow}} + 1)\alpha_{pow} \tag{2}$$

The acceleration for stage 5 is:

$$\alpha_{pow} = \frac{t_{trans} + t_{comp}}{t_{trans}/\alpha_{trans} + t_{comp}/\alpha_{comp}} \tag{3}$$

where: t_{trans} and t_{comp} are the execution times for data transfer and computation on a mono core computing engine, while α_{trans} and α_{comp} are the corresponding accelerations provided by MRA.

Because $\alpha_{comp} \in O(p)$, while $\alpha_{trans} \in O(1)$ we can approximate

$$t_{trans}/\alpha_{trans} >> t_{comp}/\alpha_{comp} \tag{4}$$

and α becomes:

$$\alpha \simeq (\frac{t_{inv}}{t_{pow}} + 1)(1 + \frac{t_{comp}}{t_{trans}})\alpha_{trans} \tag{5}$$

Because the overall acceleration is limited by the constant value of α_{trans}, for a p bigger than a maximal value the performance of the system does not improve. This value p is computed starting from the inequality 4:

$$p_{max} \simeq 20\frac{t_{comp}}{t_{trans}}\alpha_{trans} \tag{6}$$

For $p > p_{max}$ the acceleration grows insignificantly for spectral clustering if $p < n$.

VI. FPGA Experimental Setting

The accelerator we propose is seen as a part of a heterogenous pseudo-reconfigurable computing system where the host machine accelerates computation using a hardware implemented standard library. The system is pseudo-reconfigurable because it is configured only once at the start of the program. Instead of a sequence of circuits instantiated during the execution, only one programmable circuit, our accelerator, is instantiated initially and is accessed during the execution to run the programs loaded initially together with the design of the circuit. Because the design of the accelerator is parameterized, at each instantiation it is tuned to the application it supports. The number, p, of cells, the size of the word, the type of arithmetic (integer or various forms of floating point arithmetic), the type of reduction functions are decided at the synthesis time of the circuit. Even, the instruction set architecture is adapted, reconfigured for the specific application.

The experimental environment is provided by a PYNQ-Z2 board equipped with a Zynq 7020 SoC in which we deployed a MRA with 128 32-bit cells (see Figure 3). Our project used 60% of the FPGA's programmable resources, all block RAMs and 129 out of the 220 DSPs.

For $n > p$ the acceleration provided by this implementation of MRA is, according to Equation 5:

$$\alpha \simeq 32$$

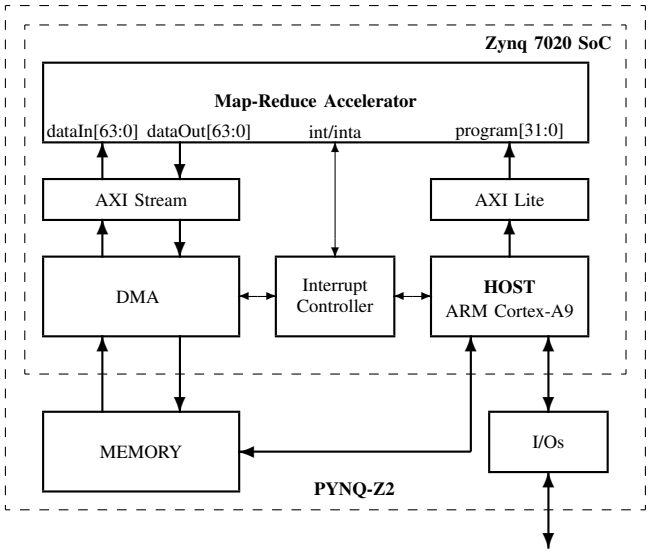

Fig. 3. The pseudo-reconfigurable system.

because $t_{comp}/t_{trans} = 1$, $\alpha_{trans} = 8$ for our system in matrix-vector multiplication, and we consider $t_{inv}/t_{pow} = 1$. If the weight of the inverse matrix computation is bigger or lower, then α is modified accordingly. Because the ratio t_{inv}/t_{pow} depends on the initial data (the actual content of vector O) the real performance cannot be stated absolutely.

The value of p is 128, which is in the proximity of $p_{max} = 160$ (according to relation 6).

VII. ASIC Version Evaluation

If an ASIC solution is envisaged, then the approach could be based on few versions of the proposed architecture already implemented in silicon [15] [9]. The last version was implemented in $65nm$ and it provided: $> 120GOPS/Watt$ and $> 6.25GOPS/mm^2$ (where GOPS stands for 16-bit Giga Integer Operations per Second).

The last evaluation for $7nm$ technology provided, for 128 32-bit cells, running at $f_{clock} = 1GHz$, with 4096 KB of memory each, implemented on $0.4mm^2$ powered at 64 mW [8]. The computational performance is 2 TOPS/W (where OPS stands for 32-bit integer operations per second) or 833 GFLOPS/W for 32-bit floating point applications.

The area and energy used in our approach are reduced because:

- the floating-point operations are implemented as a sequence of integer operations (on an average of 8 cycles per float operation)
- the predictability of the data flow in intense computation is very high, instead of a cache-based memory hierarchy we adopted a buffer-based memory hierarchy.

In this initial stage of the project, the accelerated functions of the kernel library, used to support the library developed at the host level, are written in assembler.

VIII. State of the art

The core of the spectral clustering algorithm is dominated by the matrix inverse operation. In [18], the LU factorization, the main mechanism in computing the inverse of a matrix, is investigated by programs written for two GPUs and compared with a program running on a CPU. There are two cases to be compared.

First case is for n in the range of thousands. The acceleration obtained by the two GPUs is insignificant, while for our solution the acceleration $\in O(p)$.

The second case is for $n > p$. The accelerations provided by the two GPUs are $\alpha_1 = 4.4\times$ and $\alpha_2 = 2.7\times$, while their bandwidth are $BW_1 = 142GB/s$ and $BW_2 = 86GB/s$. Obviously, the computation in the two GPUs is I/O bounded, because, according to Equation 5, the acceleration they provide is proportional with their bandwidth to the external memory: $\alpha_1/\alpha_2 = 1.62$ while $BW_1/BW_2 = 1.65$.

In the second case, the difference is given only by energy consumption. NVidia's Kepler architecture implemented in $28nm$ provides around 10 GFLOPS/Watt [13].

To compare "apples with apples" we evaluated our architecture for $28nm$ technology node. For 2048 32-bit cells, running at $f_{clock} = 1GHz$, with 4096 KB of memory each, implemented on $9.2 \times 9.2mm^2$, using standard cell $28nm$ library, 2 TOPS or 0.83 TFLOPS powered at $\sim 15Watt$. These figures translate in more than 50 GFLOPS/Watt.

To conclude the comparison:

- for $n \leq 4096$ our solution provides an acceleration in $O(p)$, while off-the-shelf solutions provide almost no acceleration
- for $n > 4096$ our solution provides similar acceleration to off-the-shelf solutions, with several times lower power consumption.

The number $n = 4096$ is, in this stage of silicon technology, the approximative threshold between what can be implemented in FPGA and what must be implemented as ASIC.

IX. Conclusion

The proposed accelerator is evaluated in two cases. In the first case, it works CPU-bound ($n \leq p$: the number of execution units in MRA allow to keep inside de accelerator the matrices involved in computation). In the second case, it works I/O-bound ($n > p$: the matrices involved in computation are stored in the external memory and are processed tile by tile).

In the CPU-bound regime the acceleration provided for all stages of the spectral clustering is in $O(p)$. This acceleration is supported by a reduction network whose log-depth latency is avoided by an appropriate use of a distributed stack along the cells in MAP array.

In I/O-bound regime the acceleration depends on the bandwidth with the external memory.

In both cases, the accelerations are provided by an energy saving hardware: 3 to 5 times more GFLOPS/Watt than off-the-shelf solutions.

Acknowledgements

The authors got support from the technical contributors to the development of the ***ConnexArray***^{TM} technology: Emanuele Altieri, Frank Ho, Bogdan Mîţu, Marius Stoian, Dominique Thiébaut, Tom Thomson, Dan Tomescu.

REFERENCES

[1] I. Fischer, J. Poland: *New Methods for Spectral Clustering*, Technical Report IDSIA-12-04, 2004.

[2] M. Fiedler: "Algebraic connectivity of Graphs", *Czechoslovak Mathematical Journal* 23(98) (1973), pp.298–305.

[3] M. Fiedler: "Laplacian of graphs and algebraic connectivity", *Combinatorics and Graph Theory* (Warsaw, 1987), Banach Center Publications 25(1) (1989), pp. 57–70.

[4] C. Fowlkes, S. Belongie, F. Chung, J. Malik: "Spectral grouping using the nystrom method", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 2, pp. 214–225, 2004.

[5] D. Hamad, P. Biela: *Introduction to spectral clustering*, 2008. [Online]. Available: http://lagis-vi.univ-lille1.fr/~lm/classpec/reunion_28_02_08/Introduction_to_spectral_clustering.pdf

[6] W. Jian-ping, S. Jun-qiang, Z. Wei-min, "An efficient and accurate method to compute the Fiedler vector based on Householder deflation and inverse power iteration", Journal of Computational and Applied Mathematics, Volume 269, 15 October 2014, pp. 101-108.

[7] Y. Jin, J. F. JaJa, "A High Performance Implementation of Spectral Clustering on CPU-GPU Platforms", *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*,

[8] S. Lupu, "Implementarea unui accelerator Map-Reduce sub forma de ASIC (The ASIC Implementation of a Map-Reduce Accelerator)", Master Thesis (in Romanian), Politehnica University of Bucharest, 2020.

[9] M. Maliţa, G. Ştefan, D. Thiébaut: "Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation", *ACM SIGARCH Computer Architecture News*, Volume 35 , Issue 5, Dec. 2007, pp. 32-38.

[10] L. Martin, A. Loukas, P. Vandergheynst: "Fast Approximate Spectral Clustering for Dynamic Networks", *Proceedings of the 35th International Conference on Machine Learning*, Stockholm, Sweden, 2018

[11] D. Mavroeidis: "Accelerating spectral clustering with partial supervision", *Data Min Knowl Disc* **21**, 241–258 (2010).

[12] M. McCool, A. D. Robison, J. Reinders: *Structured Parallel Programming. Patterns for Efficient Computation*, Elsevier & Morgan Kaufmann, 2012.

[13] H. Mujtaba, *Nvidia Pascal GP100 GPU Expected to Feature 12 TFLOPs of Single Precision Compute, 4 TFLOPs of Double Precision Compute Performance*, 2016. [Online]. Available: https://wccftech.com/nvidia-pascal-gp100-gpu-compute-performance/

[14] H. Shinnou, M. Sasaki: "Spectral clustering for a large data set by reducing the similarity matrix size" *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, pp. 201–204, 2008

[15] G. Ştefan, A. Sheel, B. Mîţu, T. Thomson, D. Tomescu: "The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing", *Hot Chips: A Symposium on High Performance Chips*, Memorial Auditorium, Stanford University, August 20 to 22, 2006. [Online]. Available: https://youtu.be/HMLT4EpKBAw at 35:00.

[16] G. Ştefan, M. Maliţa: "Can one-chip parallel computing be liberated from ad hoc solutions? A computation model based approach and its implementation", *18th Inter. Conf. on Circuits, Systems, Communications and Computers*, 2014, pp. 582–597.

[17] N. Tremblay, G. Puy, P. Borgnat, R. Gribonval, P. Vandergheynst: "Accelerated Spectral Clustering Using Graph Filtering Of Random Signals", *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4094–4098, 2016.

[18] V. Volkov, J. W. Demmel: "Benchmarking GPUs to Tune Dense Linear Algebra", *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[19] *** *Power Method for Approximating Eigenvalues*, [Online]. Available: https://ergodic.ugr.es/cphys/lecciones/fortran/power_method.pdf