# Heterogenous Computing for Markov Models in Big Data


Mihaela Maliţa
*Computer Science Dept.*
*Saint Anselm College*
Manchester, NH
mmalita@anselm.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, Circ. and Arch. Dept.*
*Politehnica University of Bucharest*
Bucharest, Romania
gheorghe.stefan@upb.ro



***Abstract*—Many Big Data problems, Markov Model related included, are solved using heterogenous systems: host + parallel programmable accelerator. The current solutions for the accelerator part – for example, GPU used as GPGPU – provide limited accelerations due to some architectural constraints. The paper introduces the use of a programmable parallel accelerator able to perform efficient vector and matrix operations avoiding the limitations of the current systems designed using "of-the-shelf" solutions. Our main result is an architecture whose actual performance is a much higher percentage from its peak performance than those of the consecrated accelerators. The performance improvements we offer come from the following two features: the addition of a reduction network at the output of a linear array of cells and an appropriate use of a serial register distributed along the same linear array of cells. Thus, instead of a solution with the size in $O(n^2)$ and an acceleration in $O(n^2/log n)$, we offer an accelerator with the size in $O(n)$ and the acceleration in $O(n)$.**



***Index Terms*—Big Data, Markov Models, parallel architecture, accelerators, heterogenous computing,**


## I. Introduction

The use of Markov Models (MM) in predictive analytics is a common practice in Big Data. The amount of computation requested for learning or for decoding increases continuously as the complexity of applications grow.

The amount of computation depends on the MM's number of states, $n$, and on the observation time steps $T$. The computational complexity evaluated for a mono-core computing systems is, for all MM algorithms involved, in $O(Tn^2)$. Ideally, for a heterogenous computing system, having as accelerator a many-core computing system with at least $n^2$ cores, the computation complexity goes down theoretically to $O(T\,log\,n)$. The theoretical acceleration provided by the parallel approach is thus limited to $O(n^2/log\,n)$ by the sum involved in the dot product operations requested in some of the algorithmic stages.

When $n > 100$, which is the case for an increased number of applications, it is not possible to accommodate on a silicon die $n^2$ execution units. In [12] are considered applications with $n > 1024$. Therefore, we are far from providing solutions with millions of execution cells.

Let us consider accelerators with the number of processing cores limited to $n$. Then, the theoretical acceleration belongs to $O(n/log\,n)$ for an accelerator having the size in $O(n)$. The currently used technological nodes – 5-10 nm – support accelerators with thousands of cells. Then, the theoretical problem is to remove the "log" part from the acceleration. The architecture we propose provides, for the accelerator part of the heterogenous system, an acceleration in $O(n)$, maintaining the accelerator as a parallel engine with the size in $O(n)$. We avoid the $log\,n$ divisor using a specific architectural feature. Thus, instead of an $O(n^2/log\,n)$ acceleration for an accelerator of size in $O(n^2)$, we provide an architecture with the acceleration in $O(n)$ for an accelerator of size in $O(n)$.

The next section reviews the main computational aspects related with MM and Hidden Markov Models (HMM). The third section presents the state of the art in the domain. Follows a section describing our proposal for a heterogenous computing system. The fifth section evaluates how the proposed system performs in the applications involving MM and HMM. Concluding remarks end our contribution.

## II. Computational Challenges for Markov Models

MM are stochastic models used to model systems that evolve randomly with the next state depending solely on the present state. MM use probabilities to find what the next state will be. Mathematically, the distribution of the state $s$ at given time $t$ depends on the previous state at the time $t-1$.

HMM are stochastic process where the states cannot be observed, and there is uncertainty about the current state of the system. They are process that have two levels. The upper level has the unobservable states; it is the MM level. The lower level is what actually the states of MM emit.

### A. Markov Model's Computational Aspects

The simplest MM are the first order MM defined for internal states depending only on the previous state.

*Definition 1:* A first-order MM is a $n$-node graph where each node is a state $s_i \in S = \{s_1, \ldots, s_n\}$ and each edge $a_{ij}$ represents the probability of going from the state $s_i$ to the state $s_j$. It is completely described by the pair:

$$\mathscr{M} = (\mathbf{A}, \Pi(0))$$

where:

$$\mathbf{A} = \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \ldots & a_{nn} \end{bmatrix}$$

is the $n \times n$ ***state transition matrix*** with $a_{ij} = P(s_j|s_i)$, and

$$\Pi(0) = \begin{bmatrix} p_1 & p_2 & \dots & p_n \end{bmatrix}$$

is the initial ***state distribution vector*** with $p_i = P(s_i)$. $\diamond$

The behavior of $\mathscr{M}$ is given by the sequence of states

$$\vec{\sigma} = \{\sigma_1, \sigma_2, \dots, \sigma_t, \dots\}$$

where $\sigma_t$ is the state of $\mathscr{M}$ at the moment $t$.

Being given $\mathscr{M}$, it is used to predict the state of the system modeled after $T$ steps as follows:

$$\Pi(t) = ((((\Pi(0)\mathbf{A})\mathbf{A})\dots)\mathbf{A}) = \Pi(0)\mathbf{A}^T$$

Thus, the use of $\mathscr{M}$ involves only vector-matrix multiplication. For a mono-core computational engine the amount of computation is in $O(n^2T)$.

### *B. Hidden Markov Model's Computational Aspects*

The HMM mechanism could be associated to a finite automaton build over the MM half-automaton.

*Definition 2:* A HMM is defined by the following pair:

$$\mathscr{H} = (\mathscr{M}, \mathbf{B})$$

where: $\mathscr{M}$ is a MM (see Definition 1), and $\mathbf{B}$ describes the observable behavior of $\mathscr{M}$ defined by elements of the ***output alphabet*** $O = \{o_1, \dots, o_m\}$

$$\mathbf{B} = \begin{bmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{bmatrix}$$

where: $b_{kj} = P(\omega_t = o_k|\sigma_t = s_j)$, with $\omega_t \in O$ the value at any moment $t$ in the observed series

$$\vec{\omega} = \{\omega_1, \omega_2, \dots, \omega_T\}$$

$\diamond$

There are three fundamental problems that characterize the HMM:

- ***Evaluation***: since the state of MM is hidden, we cannot actually be sure that it had generated the outcome. Having $\mathscr{H}$ and a sequence of observations, $\vec{\omega}$, what is the probability that what we can see is really an emission product? To compute this probability, we use the *Forward Algorithm* and the *Backward Algorithm*.
- ***Learning***: given the set of possible states and the output sequence $\vec{\omega}$, A and B with the highest probability of generating the observed outcome must be provided. The *Baum-Welch Algorithm* is used for solving the learning problem.
- ***Decoding***: having $\mathscr{H}$ and the $\vec{\omega}$ sequence, the most likely hidden state sequence must be provided. To decode we use the *Viterbi Algorithm*.

### *C. Notations*

The computational support for $\mathscr{M}$ and $\mathscr{H}$ will be described using the following notations:

- Vector-Matrix Multiplication: $W_{1\times n} = V_{1\times n}\mathbf{M}_{n\times n}$
- Vector-Transp-Matrix Multiplication: $W_{1\times n} = V_{1\times n}\mathbf{M}'_{n\times n}$
- Hadamard (entrywise) product: $\mathbf{C} = \mathbf{A} \circ \mathbf{B} = \mathbf{B} \circ \mathbf{A}$ is defined for three cases, as follows:
  1) if $\mathbf{A}_{n\times m}$ and $\mathbf{B}_{n\times m}$ are $n \times m$ matrices, then $\mathbf{C}_{n\times m}$ is a $n \times m$ matrix with $c_{ij} = a_{ij}b_{ij}$ for $i = 1 \dots n$ and $j = 1 \dots m$
  2) if $\mathbf{A}_{1\times m}$ and $\mathbf{B}_{n\times m}$, then $\mathbf{C}_{n\times m}$ is a matrix with $c_{ij} = a_j b_{ij}$ for $i = 1 \dots n$ and $j = 1 \dots m$
  3) if $\mathbf{A}_{n\times 1}$ and $\mathbf{B}_{n\times m}$, then $\mathbf{C}_{n\times m}$ is a matrix with $c_{ij} = a_i b_{ij}$ for $i = 1 \dots n$ and $j = 1 \dots m$
- Hadamard (entrywise) division: $\mathbf{C} = \mathbf{A} \oslash \mathbf{B}$ is defined similarly with the Hadamard product, but the multiplication, $c_{ij} = a_{ij}b_{ij}$, is substituted with division, $c_{ij} = a_{ij}/b_{ij}$
- Matrix Pseudo-Multiplication: $\mathbf{C} = \mathbf{A} \bullet \mathbf{B}$ applies for the matrices $\mathbf{A}_{n\times m}$ and $\mathbf{B}_{m\times q}$, and is performed by substituting, in the matrix multiplication algorithm, the dot product of the m-component vectors (a line and a column), with a Hadamard product followed by the selection of the component with the highest value.
- Reduction Sum: $redSum[v_1 \dots v_n] = \Sigma_1^n v_i$
- Reduction Maximum: $redMax[v_1 \dots v_n] = Max_1^n v_i$
- Matrix Sum: $\mathbf{C} = \mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$ if $\mathbf{A}_{n\times m}$ and $\mathbf{B}_{n\times m}$ are $n \times m$ matrices, then $\mathbf{C}_{n\times m}$ is a $n \times m$ matrix with $c_{ij} = a_{ij} + b_{ij}$ for $i = 1 \dots n$ and $j = 1 \dots m$
- Row Sum:

$$\mathscr{C}(\mathbf{A}_{n\times m}) = \begin{bmatrix} \Sigma_j a_{1j} \\ \vdots \\ \Sigma_j a_{nj} \end{bmatrix}$$

- Column Sum:

$$\mathscr{R}(\mathbf{A}_{n\times m}) = \begin{bmatrix} \Sigma_i a_{i1} & \dots & \Sigma_i a_{im} \end{bmatrix}$$

### *D. Preliminary definitions*

The following definitions allow to address the previous three questions referring to $\mathscr{H}$.

*Definition 3:* Let us define the vector

$$\alpha_t = [\alpha_t(1)\alpha_t(2)\dots\alpha_t(n)]$$

where: $\alpha_t(i) = P(\omega_1\omega_2\dots\omega_t, \sigma_t = s_i)$ is the joint probability of the partial observation sequence $\{\omega_1, \dots, \omega_t\}$ and that of the state at time $t$ is $s_i$. $\diamond$

*Definition 4:* Let us define the vector

$$\beta_t = [\beta_t(1)\beta_t(2)\dots\beta_t(n)]$$

where: $\beta_t(i) = P(\omega_{t+1}\omega_{t+2}\dots\omega_T)|\sigma_t = s_i)$ is the joint probability of the partial observation sequence $\{\omega_{t+1}, \dots, \omega_T\}$ given that the state at time t is $s_i$. $\diamond$

*Definition 5:* Let us define the matrix:

$$\mathbf{G_t} = \begin{bmatrix} g_t(1,1) & \dots & g_t(1,n) \\ \vdots & \ddots & \vdots \\ g_t(n,1) & \dots & g_t(n,n) \end{bmatrix}$$

with $g_t(i,j) = \alpha_t(i) a_{ij} b_{\omega_t j} \beta_{t+1}(j)$ the probability of the transition between state $s_i$ and $s_j$ at time $t$ given the observations $\{\omega_1, \omega_2, \ldots, \omega_t\}$. $\diamond$

### E. Algorithms

Using the previous notations and definitions the four algorithms associated to $\mathscr{H}$ will be described.

*1) Forward evaluation algorithm:* let us consider

$$\mathscr{H} = (\mathbf{A}, \mathbf{B}, \Pi(0))$$

and the observed outputs $\vec{\omega} = \{\omega_1, \omega_2, \ldots, \omega_T\}$. The probability that $\mathscr{H}$ generates $\vec{\omega}$ starting from $\Pi(0)$ is computed as follows:

Initialization:

$$\alpha_1 = \Pi(0) \circ \mathbf{B}(\omega_1, -)$$

where $\mathbf{B}(\omega_1, -)$ is the line in $\mathbf{B}$ selected by $\omega_1$

Forward recursion:

$$\alpha_{t+1} = (\alpha_t \mathbf{A}) \circ \mathbf{B}(\omega_{t+1}, -)$$

for $t = 1 \ldots (T-1)$

Termination:

$$P(\vec{\omega}) = redSum(\alpha_T)$$

The execution time for a mono-core is $T_{frwEval} \in O(n^2 T)$.

*2) Backward evaluation algorithm:* let us consider

$$\mathscr{H} = (\mathbf{A}, \mathbf{B}, \Pi(0))$$

and the observed outputs $\vec{\omega} = \{\omega_1, \omega_2, \ldots, \omega_T\}$. The probability that $\mathscr{H}$ generates $\vec{\omega}$ starting from $\Pi(0)$ is computed as follows:

Initialization:

$$\beta_T = \left[\beta_T(1)\beta_T(2)\ldots\beta_T(n)\right] = \left[11\ldots1\right]$$

Backward recursion:

$$\beta_t = (\beta_{t+1} \circ \mathbf{B}(\omega_{t+1}, -))\mathbf{A}'$$

for $t = (T-1)\ldots 1$ where $\mathbf{A}'$ is the matrix $\mathbf{A}$ transposed

Termination:

$$P(\vec{\omega}) = redSum(\beta_1 \circ \mathbf{B}(\omega_1, -) \circ \Pi(0))$$

.

The execution time for a mono-core is $T_{backEval} \in O(n^2 T)$.

*3) Baum-Welch learning algorithm:* Learning means to establish the content of matrices $\mathbf{A}$ and $\mathbf{B}$ using the observation sequence $\vec{\omega} = \{\omega_1, \ldots, \omega_T\}$ running the following algorithm:

Initialization:

$$\mathscr{H} = (\mathbf{A}, \mathbf{B}, \Pi(0))$$

is instantiated using randomly distributed probabilities.

Repeat until convergence:

$$\mathbf{G}_t = \alpha_t' \circ \mathbf{A} \circ (\mathbf{B}(\omega_t, -) \circ \beta_{t+1})$$

for $t = 1 \ldots T$.

*Re-estimate* the probability matrices $\mathbf{A}$ and $\mathbf{B}$ as follows:

$$\mathbf{A} = (\Sigma_T \mathbf{G}_t) \oslash \mathscr{C}(\Sigma_T \mathbf{G}_t)$$

$$\mathbf{B} = \Gamma \oslash \mathscr{R}(\Sigma_T \mathbf{G}_t)$$

where:

$$\Gamma = \begin{bmatrix} \mathscr{R}\mathbf{G}^1 \\ \mathscr{R}\mathbf{G}^2 \\ \vdots \\ \mathscr{R}\mathbf{G}^m \end{bmatrix}$$

with: $\mathbf{G}^k = \Sigma_T \mathbf{G}_t(\omega_t = o_k)$ for $k = 1 \ldots m$

The execution time for a mono-core is $T_{BaumWelch} \in O(n^2 T)$, but with a very high and unpredictable constant.

*4) Viterbi decoding algorithm:* Given $\mathscr{H} = (\mathbf{A}, \mathbf{B}, \Pi(0))$ and the observation sequence $\vec{\omega} = \{\omega_1, \omega_2, \ldots, \omega_T\}$, we must calculate the most likely hidden states sequence $\vec{\sigma}$ that produced $\vec{\omega}$.

Initialization:

$$\alpha(1) = \Pi(0) \circ \mathbf{B}(\omega_1, -)$$

Iteration:

$$\alpha(t) = (\alpha(t-1) \bullet \mathbf{A}) \circ \mathbf{B}(\omega_t, -)$$

$$\vec{\sigma}(t) \Leftarrow \{\vec{\sigma}(t-1), s_{indexMax(\alpha(t+1))}\}$$

where: $indexMax(\alpha(i))$ is the index of the maximum value in $\alpha(i)$.

Termination:

$$\vec{\sigma}(T)$$

The execution time for a mono-core is $T_{Viterbi} \in O(n^2 T)$.

## III. State of the Art

Due to the high parallelism of the algorithms involved in HMM, the most obvious choice in terms of hardware execution platforms were the GPUs. In literature, we can find several tests performed on both CPUs and GPUs in order to evaluate the performance of each hardware when dealing with HMM tasks.

In [4], the Forward Algorithm was evaluated for NVIDIA GeForce 9800 GTX, a GPU with 128 CUDA cores and Intel Pentium E5200, a CPU with 2 cores. Their testing setup

consisted of several HMM analyzed in parallel, thus trying to use the GPU resources as efficiently as possible. Each model had 8 states, 8 symbols and 200 observations. The evaluation revealed a speedup of maximum 25×, but for more than 60 models analyzed in parallel.

The performance is pretty good because for small $n$ and $T >> n$ many (independent) $\mathscr{H}$s runs in parallel.

In [3], an evaluation in terms of speed and power was performed for wireless applications. Only HMM with large number of states (hundreds, thousands) can take advantage of the GPUs capabilities. Comparing an Intel Core 2 Duo CPU and a 72 CUDA cores NVIDIA GeForceGT 335M, the maximum speed increase is for a 4000 hidden states HMM. In these conditions, the GPU is 181× faster for Forward Algorithm, 4.6× faster for Viterbi Algorithm and 65x faster for Baum-Welch Algorithm. From the energy consumption evaluation, we can observe that, for Forward and Baum-Welch Algorithms, the power consumed by the GPU is almost constant over the number of hidden states variation. Thus, for small number of states, the CPU is more energy efficient.

It is hard to accept that the Forward Algorithm runs $181/4.6 = 39.34$ times faster then Viterbi Algorithm, because Viterbi Algorithm is just like the Forward Algorithm "except that instead of tracking the total probability of generating the observations seen so far, we need only track the maximum probability and record its corresponding state sequence" [9].

In [12], HMM Algorithms were evaluated for speech recognition, considering the end-to-end performance of the algorithms. Two machines were evaluated: Intel Core i7-920, a 4 cores CPU and NVIDIA GeForce GTX 680, a GPU with 1536 CUDA cores. For Forward Algorithm, the GPU implementation was 5.6× times faster when the number of hidden states was 4096. For Backward Algorithm, the best performance that GPU implementation achieved was 5.4× faster than the optimized CPU implementation.

This last example looks like the most realistic approach using a many-core accelerator. Comparing the GPU – a 1536-cell engine – with Intel Core i7-920 – a 32-cell engine due to its 128-bit SSE on each core – we expect an acceleration less than $(1536/16)/log_2 4096 = 8$. Indeed, the "log" seems to be the main responsible for the low performance of GPUs.

In the same time we must take into account the weight of the data transfer time. In [12], the analysis has shown that the amount of time spent with data transfer increases with the number of states, reaching about 30% for 4096 states.

## IV. Our Heterogenous System

The heterogenous system consists in a mono- or multi-core HOST processor and a many-core programmable ACCELERATOR (see Figure 1).

The programmable ACCELERATOR consists of:

- a linear array of $p$ cells each containing a $m$-word data memory, $mem_i$, and an $n$-bit execution unit, $eu_i$
- a $log$-depth reduction network, R, a circuit which applies *reduction add* and r*eduction maximum* to the vector provided by the active cells of the array
- a sequencer designed as a system with a processing unit, $pu$, and a data & program memory ($prog\,\&\,data$); it issues in each cycle, through a $log$-depth distribution network, an instruction executed in each active cell of the array.

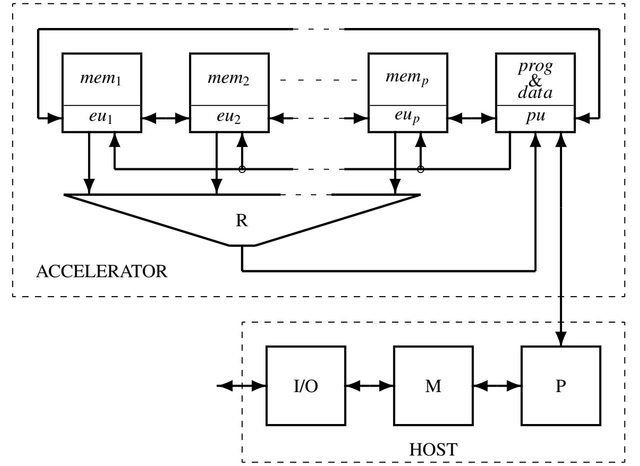


Fig. 1. Heterogenous computing system.

The size of the accelerator is in $O(p)$. For $p = 2048$, $n = 32$, $m = 1024$ in 28$nm$ tech node, the accelerator is deployed on $9.2 \times 9.2\,mm^2$ and is powered with 12 $Watt$ at $80^oC$ running at 1 $GHz$ (figures provided by simulation). Few versions were implemented in silicon [6] [11]. The last ($p = 1024$, $n = 16$, $m = 256$ in 65$nm$, running at 0.25 $GHz$) provided $200GOPS/sec$ (GOPS/sec stands for *Giga 16-bit OPerationS per second*), $> 60GOPS/sec/Watt$, $> 2GOPS/sec/mm^2$. Various application domains were investigated, such as video processing [10], or graph computation [1], [2].

### A. Accelerator's Architecture

The architecture of the system is described as the library of functions used to implement the MMl applications.

The data memory resources of the accelerator are:

- the distributed memory along the array's cells seen as the $m \times p$ matrix

$$\mathbf{S} = \begin{bmatrix} s_{11} & \dots & s_{1p} \\ \vdots & \ddots & \vdots \\ s_{m1} & \dots & s_{mp} \end{bmatrix}$$

  where: $s_{ij}$, for $i = 1 \dots m$ and $j = 1 \dots p$, are scalars; $\mathbf{S}(i,-) = V(i)$, for $i = 1 \dots m$, is the $i$-th *horizontal vector* distributed along the cells of the array, while $\mathbf{S}(-,j) = W(j)$ is the content of $mem_j$ for $j = 1 \dots p$, i.e., the *vertical vector* stored in the cell $i$.
- the Boolean vector used to perform predicated executions in each cell

$$B = \begin{bmatrix} b_1 & \dots & b_p \end{bmatrix}$$

  If $b_i = 1$ then the cell $i$ is active, else the cell does't execute the current instruction

- the index vector used to identify the cells

$$X = \begin{bmatrix} 1 & \dots & p \end{bmatrix}$$

- the serial register

$$S = \begin{bmatrix} s_1 & \dots & s_p \end{bmatrix}$$

which can interact in parallel with **S** distributed in the array, and serially with the sequencer
- the data memory, *data*, of the sequencer

$$D = \begin{bmatrix} d_1 & d_2 & \dots \end{bmatrix}$$

The host computer sees the accelerator as a hardware implemented kernel library of functions. It is about a kernel library of functions, because the fix size of the array defined by the product $m \times p$. A library of functions can be developed using programs written in a high level language on the host engine. In the following we will present a Markov-oriented kernel of functions required for accelerating MM and HMM computations.

*1) Vector-Matrix Multiplication:* multiplies in $V(x)$ a horizontal vector $V(y)$ with the $n$-line matrix stored in **S** starting with the vector $V(z)$.

```
function vmMult(x,y,z,n);
 for(i = z; i < z+n+1; i = i+1)
  S <= [redSum(V(y)*V(i)) s1 s2 ... s(p-1)];
 V(x) <= S;
endfunction
```

The execution time $T_{vmMult} = n + 2 + log_2 p \in O(n)$ for a $n \times m$ matrix if $n \leq m$ and $m \leq p$. The operation

```
S <= [redSum(V(y)*V(i)) s1 s2 ... s(p-1)];
```

pushes in the serial register $S$, with a latency in $O(log\, p)$, the output of the reduction network. Hence, the term $log_2 p$ is added instead of multiplied in $T_{vmMult}$. Thus, the serial register $S$ provides the means of avoiding $T_{vmMult} \in O(n \times log\, p)$. The acceleration, in $O(p)$, is supra-linear because multiplication, summation and control are executed concurrently in the array of cells, in the reduction network and in sequencer.

*2) Vector-Transposed Matrix Multiplication:* multiplies in $V(x)$ a horizontal vector $V(y)$ with the transposed $n$-line matrix stored in **S** starting with the vector $V(z)$.

```
function vtmMult(x,y,z,n);
    S <= V(y);
    V(x) <= [0 0 ... 0];
    for(i = z; i < z+n+1; i = i+1)
        V(i) <= V(i) * s1;
        S <= [s2 ... s(p) 0];
    for(i=z; i<z+n+1; i=i+1)
        V(x) <= V(x) + V(i);
endfunction
```

The execution time $T_{vtmMult} = 2n + 4 + log_2 p \in O(n)$ if $n \leq m$. The acceleration, in $O(p)$, is supra-linear.

*3) Hadamard Multiplication (Division, Sum):* is performed in its three forms as follows:

*a) matrix-matrix Hadamard multiplication (division, sum):* takes two equally sized $n$-line matrices, starting in **S** from $V(y)$ and $V(z)$, and provides the result in a matrix of the same size starting with the vector $V(x)$.

```
function mmHmult(x,y,z,n);
    for(i = 0; i < n; i = i+1)
        V(x+i) <= V(y+i)) * V(z+i);
endfunction
```

*b) column vector-matrix Hadamard multiplication (division, sum):* considers the horizontal vector $V(y)$ as a column vector and performs Hadamard multiplication with a $n$-line matrix stored starting with the horizontal vector $V(z)$ and stores the result starting with $V(x)$

```
function cmHmult(x,y,z,n);
    S <= V(y);
    for(i = 0; i < n; i = i+1)
        V(x+i) <= V(z+i) * s1;
        S <= [s2 ... s(p) 0];
endfunction
```

*c) row - matrix Hadamard multiplication (division, sum):* considers the horizontal vector $V(y)$ as a row vector and performs Hadamard multiplication with a $n$-line matrix stored starting with the horizontal vector $V(z)$ and stores the result starting with $V(x)$

```
function rmHmult(x,y,z,s);
    for(i = 0; i < n; i = i+1)
        V(x+i) <= V(z+i) * V(y);
endfunction
```

All three forms of the Hadamard multiplications (division) are accelerated a number of times in $O(p)$.

*4) Reductions:* are hardware functions performed, with a latency of $\lambda = log_2 p$, by the reduction network $R$ (see Figure 1) for addition and maximum.

*5) Row Sum:* is performed on a $n$-line matrix stored starting with the vector $V(y)$ with the result in $V(x)$.

```
function rowSum(x,y,n);
 for(i = 0; i < n; i = i+1)
  S <= [redSum(V(y+i)), s1, s2, ... s(p-1)];
 V(x) <= S;
endfunction
```

The acceleration is in $O(p)$.

*6) Column Sum:* is performed on a $n$-line matrix stored starting with the vector $V(y)$ with the result in $V(x)$. The acceleration is in $O(p)$.

```
function colSum(x,y,n);
    V(x) <= [0 0 ... 0];
    for(i = 0; i < n; i = i+1)
        V(x) <= V(x) + V(y+i);
endfunction
```

*7) Transfer Operations:* are performed on vectors between **S** distributer in array and the memory M (see Figure 1):

*a) Load Matrix:* starting with the vector $V(vAddr)$ of size *nCol* are loaded from M, starting from the address *sAddr*, *nLine* vectors, $loadM(vAddr, sAddr, nLine, nCol)$

*b) Store Matrix:* $storeM(vAddr, sAddr, nLine, nCol)$

*c) Load Vector:* a vector of length *size* is loaded, from the address *sAddr* in M, into the sequencer's data memory starting with the address *xAddr*, $loadV(xAddr, sAddr, size)$

*d) Store Vector:* $storeV(xAddr, sAddr, size)$

Unfortunately, the transfer time is proportional with the amount of data transferred generating sometimes I/O-bottlenecks, i.e., I/O-limited performance in executing programs where the data use is low.

### B. Performances

Excepting the transfer operations, all the above described operations are accelerated $\delta$ times, with $\delta \in O(p)$. The bad news is: the transfer operations pay a toll to the *"von Neumann Bottleneck"*. But, the good news is: data once in **S** can be used for a big number of operations in most of cases.

## V. Evaluation

The program for Forward Evaluation Algorithm is:

```
program FEA(n,   // number of states
            m,   // number of outputs
            t,   // number of observations
            a,   // address of A in M
            b,   // address of B in M
            p,   // address of Pi(0) in M
            o,   // address of O in M
            r);   // address of result in M
 loadM(1,a,n,n);        // A starts with V(1)
 loadM(1+n,b,n,m);      // B starts with V(1+x)
 loadM(1+n+m,p,1,n);  // Pi is at V(1+x+y)
 loadV(1,o,t);          // O starts at addr 1
 mmHmult(0,1+n+m,1+n+o1,1);  //Pi(0)*B(o1,-)
 for(i = 1; i =< (t+1);   i = i+1)
     vmMult(0,0,1,n);
     mmHmult(0,0,1+n+o(i),1);
 d0 <= redSum(V(0));
 storeV(r,0,1);          // store result in M
endprogram
```

If the number of cells and the amount of memory per cell are enough big, the execution time of the computation is in $T_{fwdEv} \in O(max(n^2, nm, nt))$ depending on the relations between the size of the parameters $n$, $m$, and $t$. Because, usually $t >> n$, $t >> m$, $T_{fwdEv} \in O(nt)$ for $n \leq p$. Then the acceleration is in $O(p)$. Current technology nodes allow us easy to deploy on single silicon die thousands of 32-bit cells each with its own memory up to 8-16-Kword memory. Thus, $\mathscr{H}$ applications with $n$ and $m$ in the range of thousands, as in [12], can be solved without embarrassing additional data transfers.

## VI. Concluding Remarks

There are two main problems in the current acceleration process for $\mathscr{H}$ applications:

- diminishing of the acceleration $log$ times due to the reduction add and reduction max operations
- I/O bounded executions for big $n$ and $m$.

The first issue is solved by our architecture providing for an accelerator with size in $O(p)$ an acceleration in $O(p)$ by properly using a reduction network which captures the output of an array of cells and the serial register.

The second issue is only partially solved because it depends on the possibility to accommodate on the same silicon die enough memory distributed along the cells. For the current size of the problems considered by the users, the 7 nm technological node works well enough.

## References


[1] V. Dragomir, "Graph Traversal on One-Chip MapReduce Architecture", *18th Int. Conference on Circuits, Systems, Communications and Computers*, Santorini Island, Greece, July 17-21, 2014, pp. 559-563.

[2] V. Dragomir, "Breadth-First Search on a MapReduce One-Chip System", *International Journal on Recent and Innovation Trends in Computing and Communication (IJRITCC)*, vol. 4, issue 4, April 2016, pp 76-81.

[3] S. R. Hymel, "Massively parallel hidden markov models for wireless applications", Ph.D. dissertation, Virginia Polytechnic Institute and State University, 2011.

[4] J. Li, S. Chen, Y. Li, "The fast evaluation of hidden Markov models on GPU", *2009 IEEE Int. Conference on Intelligent Computing and Intelligent Systems*, Shanghai, 2009, pp. 426-430.

[5] Q. Lv, Y. Qiao, N. Ansari, J. Liu, J. Yang, "Big Data Driven Hidden Markov Model Based Individual Mobility Prediction at Points of Interest", IEEE Transactions on Vehicular Technology PP(99):1-1, 2016.

[6] 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.

[7] G. Manogaran, V. Vijayakumar, R. Varatharajan, P. M. Kumar, R. Sundarasekar, C.-H. Hsu, "Machine Learning Based Big Data Processing Framework for Cancer Diagnosis Using Hidden Markov Model and GM Clustering", Springer Science+Business Media, LLC, part of Springer Nature 2017.

[8] D. Marten, A. Heuer, "Machine Learning on Large Databases: Transforming Hidden Markov Models to SQL Statements", Open Journal of Databases (OJDB), Volume 4, Issue 1, 2017.

[9] Daniel Ramage: *Hidden Markov Models Fundamentals*, 2007. [Online]. Available: http://cs229.stanford.edu/section/cs229-hmm.pdf

[10] 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

[11] 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.

[12] L. Yu, Y. Ukidave, D. Kaeli, "GPU-Accelerated HMM for Speech Recognition", *43rd International Conference on Parallel Processing Workshops*, Minneapolis, MN, 2014, pp. 395-402.