Map-Scan Node Accelerator for Big-Data

Mihaela Maliţa
Computer Science Deptartment
Saint Anselm College
Manchester, NH, USA
mmalita@anselm.edu

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

***Abstract*—The current hybrid architectures, used to accelerate the nodes of the various distributed computing systems running Big Data applications, are mainly based on Nvidia's GPU or Intel's MIC accelerators. These accelerators are marked by limitations due to their too general and *ad hoc* structural and architectural features. In this paper, we propose a Map-Scan architecture, as a generalization of a Map-Reduce architecture, more appropriate for the parallel approach in defining the accelerator part of a hybrid system. The paper describes the organization and the architecture of a hybrid system based on our *Map-Scan Accelerator* (MSA). The degree of parallelism achieved by our proposal is compared with the current implementations. The energy consumption is estimated, by simulation, for the ASIC versions of MSA. We conclude that the Map-Scan approach in defining the accelerator of a hybrid system provides the appropriate solution for accelerating various Big Data applications and linear algebra based applications.**

***Keywords*-hybrid architecture; parallel accelerator; map-scan architecture; energy aware accelerator;**

I. Introduction

The three main types of platforms currently in use in Big Data – Symmetrical Multi-Processors, Clusters, or Grids – can benefit from a powerful improvement by using many-core accelerators in each of their nodes. The most used accelerators currently considered are GPUs, MICs[1] or FPGAs. Each of these solutions have their specific drawback. The first two have to face strong legacies, while the last one is hard to be efficiently used. Indeed, Nvidia's GPU emerges from a circuit for accelerating graphics applications, Intel's Xeon Phi MIC is an *ad-hoc* configured many-core based on x86 architecture, while for providing FPGA solutions, strong hardware design skills are requested. Graphics is too specific. Connecting circular x86 cores does not guarantee the requested features for solving efficiently parallel tasks. Using HDL offers high-performance circuits only in the hard-to-find hand of a highly-experienced digital designer. Not to mention the energy inefficiency associated with all the three solutions.

Let us see how perform the available many-core accelerators for Convolutional Neural Network (CNN), a very frequently used function in Big Data. We will learn that their huge computational power is too much under used. Indeed, while from Intel's i7 CPU, with 112 GFLOPs/sec, 32% is used for real time object detection, with Titan X GPU, for 40-90 fps, are used maximum 63 GFLOPs/sec from its peak performance of 6 TFLOPs/sec [9], or with Xeon Phi accelerator with 57 cores, having peak performance at 2 TFLOPs/sec, only 0.48 GFLOPs/sec is used from each core which is able to provide 35.2 GFLOPs/sec [8]. It is hard to explain why from 32% use of the peak performance for a CPU we go to less than 2% use for the many-core accelerators?

Another example is offered by the scan operation (a powerful parallel primitive operation with a broad range of Big Data applications). In [3] the authors evaluate the execution time for the prefix-sum scan operation on Nvidia platform, running at approximatively 1GHz, compared with a mono core Intel platform, running at approximatively 3GHz. The acceleration provided by the 575-core Nvidia GPU (NVIDIA GeForce 8800 GTX GPU) is less than 6× in the best case.

We must accept, on the basis of these two examples, that we are faced with a hidden architectural and organizational failure.

An appropriate solution must provide at least a good use of the peak performance, and a meaningful acceleration, at a reasonable energy use. In the same time, integrating the accelerator in the system must be done as easy as possible from the software point of view.

The map-scan architecture we propose as accelerator is an expansion of the map-reduce architecture, because the *reduction* function for the associative operator $\circ$ which computes:

$$x_1 \circ x_2 \circ \ldots \circ x_n$$

is only the last value computed by the *scan* function

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= x_1 \circ x_2 \\ &\ldots \\ y_n &= x_1 \circ x_2 \circ \ldots \circ x_n \end{aligned}$$

applied to the input vector $\langle x_1, x_2, \ldots, x_n \rangle$.

[1]Many Integrated Core

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

The structure of our Map-Scan Accelerator (MSA) consists of two main levels: the ***map level*** of a linear array of p cells able to execute predicated vector operations, and the ***scan level***, a log-depth circuit which computes the prefixes for few associative functions (add, max, ...). While the map level of the structure provides SIMD features, the scan, as a generalized reduction, adds the necessary features for an efficient general purpose functional accelerator.

The functionality accelerated on MSA can be best defined by *kernels* of various library of functions. These kernels are the implementations of the targeted libraries for limited size data structures. For example, the *Eigen* kernel on MSA contains all the Eigen functions defined on p-limited size vectors and matrices, so the *Eigen library* can be virtualized for data structure of any size.

Compared to the GPU and MIC based accelerators, we proposed an architecture derived from a functional computational model [5] capable of responding to the requirements of an accelerator for computationally intensive functions. Combining the SIMD abilities of the map level with the few most used reduction functions, generalized at the scan level, provides an architecture which exceeds the parallel features offered by the current accelerators. Obviously, we are able to do this because no legacy stops us to propose any kind of new architecture.

On the other hand, compared with the FPGA based solution, we can take advantage of the flexibility this technology offers by designing a parameterizable & configurable programmable structure. Our architecture can be designed in a maximal functionally version with all the sizes (word dimension, number of cells, the size of the memory in each cell, ...) parameterized. Once the program is written for the accelerator, the physical implementation is synthesized optimally *on* specific sizes and using *only* the requested features. Therefore, the FPGA technology could be both, a solution for small market products, or an intermediary step toward an ASIC solution for big consumer markets.

The next section describes the organization and the architecture of MSA. The third section evaluates the performance provided by out proposal for few specific functions frequently involved in the Big Data domain.

II. Map-Scan Organization and Architecture

In [11] and [7] we presented a Map-Reduce architecture and its first implementations. This architecture is based on the generic partial recursive rule of composition [5]. The generality of Map-Reduce recursive model is also presented in [1]. In this paper we introduce a generalization by substituting the **reduce** mechanism with its general implementation: **scan** mechanism. Reduce is only the last term of scan. We do this because the scan operation is a useful building block for many parallel algorithms, such as: radix-sort quick-sort, string comparison, lexical analysis, stream compaction, polynomial evaluation, solving recurrences, tree operations, histograms, etc.

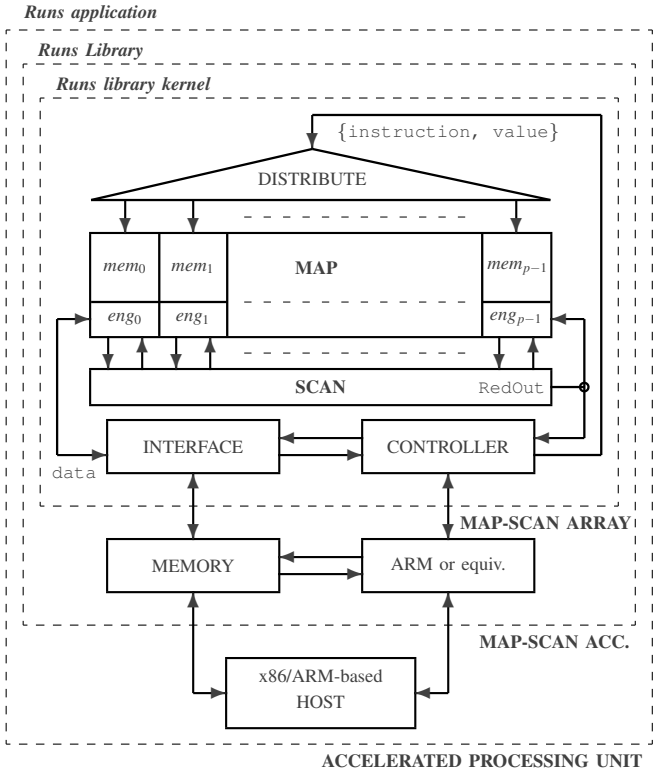

Figure 1. Hybrid system based on a Map-Scan Accelerator.

A. The Organization of Map-Scan Accelerator

The Map-Scan Accelerator (MSA) is conceived to work in conjunction with an x86/ARM-based HOST computer (see Figure 1). Any computational intensive task, performed by ACCELERATED PROCESSING UNIT (APU), is passed from HOST to MSA together with the associated data. Once the computation done, the result is send back to the host or it is used to accomplish another task. MSA consists of the following main parts:

- ARM or an equivalent processor
- local MEMORY for data and programs
- MAP-SCAN ARRAY the core of the accelerator containing the following blocks:
  - INTERFACE used to transfer data & programs between MEMORY and MAP-SCAN ARRAY
  - CONTROLLER which issues in each clock cycle a pair of instructions from its program memory, one for itself and another for the MAP array
  - DISTRIBUTE is a log-depth pipelined distribution tree for {`instruction, value`} sent by CONTROLLER to MAP array, where `value` is a scalar or an address

- MAP is a linear array of p cells each containing:
  - execution unit, eng_i, for $i = 0, 1, \ldots, p-1$, which receives in each clock cycle an instruction and executes it if the cell is in the active state
  - local data memory, mem_i, for $i = 0, 1, \ldots, p-1$, which stores m scalars.
- SCAN a *log*-depth pipelined network of cells used to perform prefix functions which take from MAP a p-component vector and sends back to MAP, in $O(\log p)$ time, a vector of the same size; a special, faster way, is used to send back to the MAP array and to CONTROLLER the result of the reduction operations, `RedOut`.

In each cell, eng_i is an integer execution unit with few specialized functions to accelerate the floating-point operations executed sequentially. The low weight of floating-point operations allows us to avoid the overload supposed by specific floating-point units.

Another way to keep the size of cells and the energy they use as small as possible is to decide the use in each eng_i an accumulator based execution unit. At this early stage of development, it is difficult to determine how many registers should be considered for a register file system, or how deep should be an execution stack. It turns out that, the simple, initial solution we considered is enough good for the applications we already investigated. If needed, the next step will be to substitute the accumulator (a one level stack) with a stack of few levels.

B. The Architecture of Map-Scan Accelerator

The data structure in the MAP array is the vector of scalars. The entire content of the local memories, mem_i, distributed along the array of cells, is represented by a $p \times m$ matrix M, where each line in M is a *horizontal vector*:

$$V_j = \langle s_{0j}, s_{1j}, \ldots, s_{(p-1)j} \rangle$$

distributed along the cells, for $j = 0, 1, \ldots, m-1$, and each column in M is a *vertical vector*:

$$W_i = \langle s_{i0}, s_{i1}, \ldots, s_{i(m-1)} \rangle$$

stored in the local memory of $cell_i$, for $i = 0, 1, \ldots, p-1$.

In addition, there are few specific horizontal vectors distributed along the array:

- $IX = \langle 0, 1, \ldots, p-1 \rangle$: the constant vector index, used to identify each cell
- $B = \langle b_0, b_1, \ldots, b_{p-1} \rangle$: a Boolean vector, used to activate the cells of the MAP array (the cell i is active only if $b_i = 1$, else the cell i ignores the instruction received in the current cycle from CONTROLLER through DISTRIBUTE)
- $ACC = \langle acc_0, acc_1, \ldots, acc_{p-1} \rangle$: accumulator vector, used as left operand and as destination for the result
- $CR = \langle cr_0, cr_1, \ldots, cr_{p-1} \rangle$: carry vector
- $ADDR = \langle addr_0, addr_1, \ldots, addr_{p-1} \rangle$: address vector, used to address in the local memories mem_i.

Correspondingly, in CONTROLLER there are the scalar resources: *acc*, *cr*, *addr*.

The instruction set architecture (ISA) of MRA is the Cartesian product of two ISAs:

$$ISA_{MRA} = cISA \times aISA$$

where *cISA* is executed by CONTROLLER, while *aISA* is executed in the array of cells.

The arithmetic and logic operations are the same in the two sets. In *cISA* these operations are defined on scalars, while in *aISA* are executed on vectors. The instructions look like:

$$acc <= acc\ OP\ operand$$

in CONTROL, and

$$acc_i <= b_i\ ?\ acc_i\ OP\ operand_i\ :\ acc_i$$

where *OP* represents an arithmetic or logic operation and *operand* and $operand_i$, the right operands, are selected in seven modes. For example, ADD operation in any eng_i is performed in the following modes:

```
VADD(val)  : acc  <= acc + val
ADD(val)   : acc  <= acc + mem[val]
RADD(val)  : acc  <= acc + mem[val+addr]
RIADD(val): acc  <= acc + mem[val+addr]
            addr <= val + addr
CADD       : acc  <= acc + coOperand
CAADD      : acc  <= acc + mem[coOperand]
CRADD      : acc  <= acc + mem[coOperand+addr]
```

where `val` is the immediate value, and `coOperand` is `acc`. For CONTROLLER the operations are performed similarly, but the `coOperand` is `RedOut`.

The main differences between *cISA* and *aISA* are in the control instructions subsets. The control instructions for CONTROLLER are the standard conditioned or unconditioned jumps and branches. In MAP array, *aISA* provides a ***spatial control*** using predicated operations. It is based on operations applied on the Boolean vector B. The main spatial control operations are:

- `activate` : $b_i <= 1$, for $i = 0, 1, \ldots, p-1$
- `where (cond)` : $b_i <= (b_i\ \&\ cond_i)\ ?\ 1 : 0$
- `endwhere` : restore B to the previous value

Example 2.1: In Figure 2 is shown a simple code which multiplies the index vector *IX* with the sum of its odd components. The line labeled with `LB(1)` is a wait loop for the latency introduced by the *log*-depth reduction network. In this example we consider $p = 512$, then between the cycle when the odd accumulators of MAP array are selected and the cycle when the reduction sum is loaded in CONTROL's accumulator we must allow a latency of 9 cycles.

The execution time of the program, for $p = 512$, is: $T(p) = 7 + log_2 p = 16$. In this 16 cycles are executed 3×515 load operations, 512 ANDs, 255 ADDs, 512 MULTs, and

1024 spatial selection operations, i.e., 340 operations per cycle which corresponds to a degree of parallelism of 66%.◇

```
/*******************************************************
Index vector is multiplied with the sum of its odd
components.

Number of cells: p = 512 => x = log_2 p
*******************************************************/
        cVLOAD(1);   IXLOAD;      // acc<=1; acc[i]<=i
        cNOP;        CAND;        // acc[i]<=acc[i] & acc
        cVLOAD(x+1);WHEREZERO;    // acc<=10; only even cells
LB(1);cBRNZDEC(1);IXLOAD;        // latency loop; load index
        cNOP;        ENDWHERE;    // reactivate all cells
        cCLOAD(0);   IXLOAD;      // acc<=redAdd; acc[i]<=i
        cNOP;        CMULT;       // acc[i]<=acc[i] * redAdd
```

Figure 2. Example of code executed by MRA. The left column contains instructions, prefixed with c, for CONTROLLER, while the right column contains instructions for the MAP array.

C. The Use of Map-Scan Accelerator

The most efficient way to software integrate a MSA is to run on it a *library* of computationally intense functions. Let us say the *Eigen* library. Then, at the MSA level, ARM loads into CONTROLLER the program for *EigenKernelLibrary* and uses the MAP-SCAN matrix as deployment environment to support the Eigen Library implementation for HOST.

The $EigenKernelLibrary(p,m,n)$ is a "bounded" library of functions defined on the limited size data structure according to the size of the MAP-SCAN ARRAY characterized by p cells, and m n-bit words local memories mem_i. Then at the level of MRA the *Eigen* library can be developed in a high level language as:

$$Eigen(EigenKernelLibrary(p,m,n))$$

Thus, $EigenKernelLibrary(p,m,n)$ is seen as the *firmware* level of our architecture.

III. Evaluation

For this preliminary presentation of our architecture we selected three examples of the frequently used tasks performed in Big Data applications: K-means clustering, scan, CNN.

Because it is hard to make fair comparisons when different technology are involved in actual implementations, we will consider the *architectural acceleration*, the acceleration due to the architectural decisions.

A. Architectural Acceleration

Let be two platforms, P_1 and P_2, each running on its clock frequency, $f_1 \neq f_2$. Running the same application, F, the execution time is $t_1(F)$ on P_1 and $t_2(F) < t_1(A)$ on P_2. The *actual acceleration* provided by P_2 is $t_1(F)/t_2(F)$. We define the *architectural acceleration*, the acceleration provided by P_2 running at the same frequency as P_1 which is

$$\frac{t_1(F)}{t_2(F)} \times \frac{f_1}{f_2}$$

B. K-Means Clustering

Given n d-dimension vectorial entities (points)

$$\langle \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n} \rangle$$

k-means clustering provides the partition into k sets. The generic algorithm consists of the following main steps:

1) set (randomly) k d-dimension centers and assign (randomly) each point to a center
2) compute for each d-dimension point the Euclidean distance to the k centers and assign each point to the "nearest" center
3) compare the new assignments to the old ones
   if no difference, **then** stop the process
   else continue
4) move the k centers to the means of created groups, and go to (2).

The degree of parallelism for the steps 2 and 3 on the previous loop is maximal. Only the step 4 is executed with a degree of parallelism p/k.

Let us consider initially a number of points equal with p. Then, each point is associated to a cell, which stores the d coordinates. The computation is not I/O bounded even for the smallest data bandwidth of 4GB/sec, if $30k > p$. In these easy to fulfil conditions, by simulation, the architectural acceleration of a pure sequential computation results, for $k > 10$:

$$A \simeq p \times \frac{\alpha}{1+\alpha}$$

where:

$$\alpha = \frac{execution\_time\_for\_steps\ 2+3}{execution\_time\_for\_step\ 4} \simeq \frac{16}{4+log_2 p}$$

For $p = 1024$, $A \simeq 546$.

The number of points can be easy expanded, maintaining the acceleration, to hundreds of thousands if the computation remains not I/O bounded. The data for each set of p points is stored in $d+1$ horizontal vectors.

C. Scan

In [6], the k-means clustering is accelerated using the scan operation of prefix-sum. Our architecture is featured with specific hardware for this operation. In contrast to the previous application, scan computation is IO bounded in our MSA, i.e., the weight of the time spent for the data transfer, when the operation is a "short-lived" process, dominates the computational time.

We consider first the case of a "short-lived" process.

The execution time on an NVIDIA GeForce 8800 GTX GPU for 2^{20} numbers is 1.11 ms, out of which 0.092 ms are due to the transfer (at 86GB/sec) [3].

The processing time on our MSA with $p = 1024$ for 2^{20} numbers is $0.033ms$ at 1 GHz. Considering the same bandwidth to the external memory, the total execution time becomes 0.133 ms. The architectural acceleration provided by our solution, compared to Nvidia GPU, is $11.18\times$ if the transfer time is considered.

But, if the transfer time does not count, because the function is executed as apart of an application which provides the data inside the MAP array and lets the result in the same place, the architectural acceleration is $38.86\times$.

The architectural efficiency of using a core in MSA is

$$38.86 \times \frac{575}{1024} \times = 21.82\times$$

compared with a core in the GPU we considered.

D. Deep Neural Networks

The Big-Data domain starts to be dominated by Machine Learning, as an AI technique based on deep CNN algorithms. The main computational pattern for CNN is the matrix-vector multiplication [4]. It is obvious for the fully connected layers, while for the convolutional layers we must consider: (1) each receptive fields in the input three-dimension volume (see [4]) a vector $\mathbf{V_i}$ of $F \times F \times D_1$ input components, and (2) the K filters as vectors of the same number of parameters (weights). Then, the convolution supposes to multiply the weights matrix $\mathbf{M}$ of $(F \times F \times D_1) \times K$ size with the input vector $\mathbf{V_i}$. Results a K-component vector. The nonlinear activation function f is applied to its components and results the K-component vector in the output volume. The matrix $\mathbf{M}$ is unique for a convolutional layer. Therefore, it is loaded only once for the computation of one convolutional layer.

In the training process floating point operations are usually requested, while in the running process almost all the time integer operations are used. Therefore, we provide both types of arithmetic operations.

1) Integer Matrix-Vector Multiplication: The algorithm for integer multiplication of $N \times M$ matrix ($N \leq min(m, p)$ lines, and $M \leq p$ columns) with a M-component vector consists of three main operations:

- control, performed by the CONTROLLER unit
- integer multiplication, performed in the MAP array
- integer addition, performed in the SCAN section

All these three operations are performed in parallel on distinct hardware resources. The main problem solved for optimizing the algorithm was to avoid the effect of the latency, of $O(log\ p)$, introduced by the SCAN section to its `RedOut` output (see Figure 1). An additional shift register introduced in the organization of the MAP array allows to insert back into the MAP array the output `RedOut` of SCAN, avoiding an explicit load in the CONTROLLER's accumulator. Thus, instead of providing each component of the resulting vector with a latency in $O(log\ p)$, only the final form of the resulting vector is provided with a $O(log\ p)$ latency. The program in assembly language is listed in Figure 3, where the instruction `IP(255)` multiplies the accumulator vector with the next line of the matrix and pushes the inner product provided by SCAN at `RedOut` in the above mentioned shift register. After N runs of the one step loop

```
LB('P);cBRNZDEC('P); IP(255);
```

the latency loop

```
LB('L); cBRNZDEC('L); NOP;
```

introduces a delay according to the size, p, of the MAP array.

```
/*****************************************************
FUNCTION NAME: Matrix-vector multiplication
The function multiplies a NxM matrix with a M-component
vector
Initial: addr[i] = I+1 : I is address of the last line
         acc[i] = V[i] : the vector
Final:   acc[i] = result
*****************************************************/
//Parameters:
 'define N     13        // number of lines
 'define S     (x-1)     // latency size because p = 2^x
//Labels:
 'define P     1         // main loop label
 'define L     2         // latency loop label

        cVLOAD('N);    NOP;        // acc <= N;
LB('P); cBRNZDEC('P); IP(255);    // loop control; IP
        cVLOAD('S);    NOP;        // init latency loop
LB('L); cBRNZDEC('L); NOP;        // latency loop
        cNOP;          SRLOAD;     // result in acc[i]
```

Figure 3. The program for matrix-vector multiplication. For big N the program is executed in $\sim N$ cycles.

The execution time for matrix-vector multiplication is

$$T(N) = N + 2 + log_2 p \in O(N)$$

Compared with a mono-core engine the acceleration is supra-linear, because besides the parallelism offered by the many-cell structure of the MAP array, we benefit by the parallelism in the SCAN section, and by the control running on a different physical resource, the CONTROLLER unit.

2) Floating Point Matrix-Vector Multiplication: The *log* latency introduced by the REDUCE unit can't be easy avoided like in the previous case. The price for avoiding latency when float reduction add is involved is to double the local memory used. A first loop computes all the N vector products and provides the vector of the maximum exponents in the serial register. Then, the second loop uses the exponents to compute the reduction add from the vector

products provided by the first loop. Thus the computing time for the float matrix-vector multiplication remains in $O(N)$.

E. Previous Work

The Map-Scan architecture is proposed based on previous work done in implementing a Map-Reduce architecture in few versions: CA4096, CA1024, BA1024 [11] [7]. The best performance, obtained in 65nm standard process technology, is $120\,GOPS/Watt$ and $6.25\,GOPS/mm^2$, where GOPS stands for **G**iga 16-bit integer **O**perations **P**er **S**econd. The application domain for these first versions was HDTV.

An FPGA implementation is done using the Zynq-7020 system-on-chip on the Zedboard development platform [2]. The ARM featured on Zynq-7020 provided a first version of MSA in the particular case of a Map-Reduce architecture.

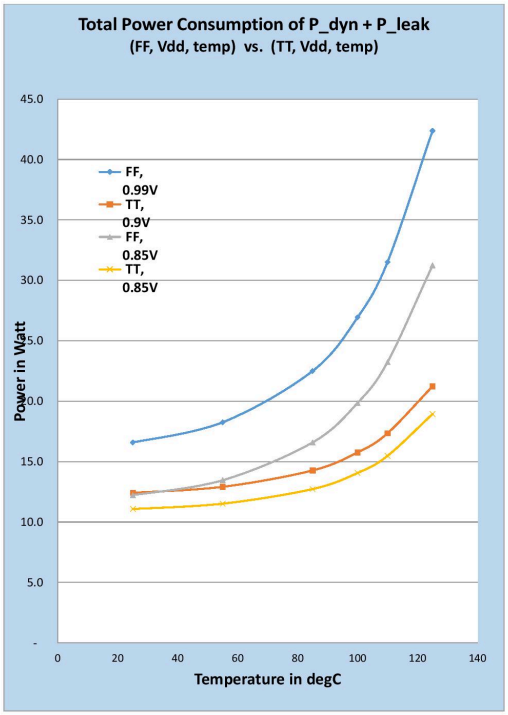

Figure 4. Power consumption evaluated for out MSA.

A 28nm simulation of a MAP-SCAN ARRAY running at $1GHz$ was used to evaluate the size and the power for a version having the word size $n = 32$, $p = 2048$ the number of cells $p = 2048$, and the memory size $m = 1024$. The resulting area of the chip is $9.2 \times 9.2\,mm^2 = 84.64mm^2$. The power consuming of the chip is shown in Figure 4.

IV. Conclusion

Our preliminary evaluations for frequently used tasks are very encouraging. For K-means cluster computation the acceleration is higher than the critical threshold of $O(p/\log p)$. For the scan function, the use of each cell in our proposal is $21.82\times$ higher than the use in Nvidia architecture. For CNN implementation a supra-linear acceleration is provided for matrix-vector multiplication.

The only drawback is given by the IO bounded computations. The "von Neumann bottleneck" persists.

References

[1] R. Andonie, M. Maliţa and G. M. Ştefan, "MapReduce: From Elementary Circuits to Cloud", in Kreinovich, Vladik (Ed.), *Uncertainty Modeling*, Springer, pp. 1-14, 2017.

[2] C. Bîra, R. Hobincu, L. Petrică, V. Codreanu, and S. Coţofană, "Energy-Efficient Computation of L1 and L2 Norms on a FPGA SIMD Accelerator, with Applications to Visual Search", in *Proc. of the 18th Intl. Conf. on Circuits, Systems, Com. and Computers (CSCC), Advances in Information Science and Applications - Volume II*, pp. 432-437, 2014.

[3] M. Harris, (2017, April) Parallel Prefix Sum (Scan) with CUDA. [Online]. Available: https://www.mimuw.edu.pl/∼ps 209291/kgkp/slides/scan.pdf

[4] A. Karpathy, "Cs231n: Convolutional neural networks for visual recognition". [Online]. Available: http://cs231n.github.io/

[5] S. Kleene, "General recursive functions of natural numbers", *Mathematische Annalen*, vol. 112, no. 1, pp. 727-742, 1936.

[6] K. J. Kohlhoff, V. S. Pande and R. B. Altman, "K-Means for Parallel Architectures Using All-Prefix-Sum Sorting and Updating Steps", *IEEE Transactions on Parallel and Distributed Systems*, Volume: 24, Issue: 8, Aug. 2013, pp. 1602 - 1612.

[7] M. Maliţa, G. Ştefan and D. Thiébaut, "Not Multi, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation", *ACM SIGARCH Computer Architecture News*, 35(5):32-38, December 2007.

[8] G. Raina (2016); Deep Convolutional Netvork evaluation on the Xeon Phi: Where Subword Parallelism meets Many-Core, *Eindhoven University of Technology*. [Online]. Available: http://repository.tue.nl/844256

[9] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You only look once: Unified, real-time object detection", *Cornell Univ. Library*, 2016.

[10] D. Singh and C. K. Reddy, "A survey on platforms for big data analytics", *Journal of Big Data* 2014 2:8. [Online]. Available: https://journalofbigdata.springeropen.com /articles/10.1186/s40537-014-0008-6

[11] G. M. Ştefan, A. Sheel, B. Mîţu, T. Thomson, and D. Tomescu, (2006, Aug.) "The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing" *Stanford University: Hot Chips: A Symposium on High Performance Chips*. [Online]. Available: https://youtu.be/HMLT4EpKBAw at 35:00.