# Searching Beyond of the Turing Based Architectures to Surpass the Moore's Law Challenges

GHEORGHE M. ȘTEFAN

Politehnica University of Bucharest, gheorghe.stefan@upb.ro

**Abstract**. The parallel computing occurred and developed, in the context of the brute force structural possibilities offered by the action of Moors's Law, by putting together, in an *ad hoc* geometric configuration, a bunch of Turing-based machines without any specific idea about parallel computational aspects. The main result of this approach are engines whose floating-point peak performances are minimally used in real-world applications. The paper describes a research project whose main outcome is an abstract model for parallel computation. The proposal is based on actual implementations having as target the emerging domain of hybrid computing. In the same time, the proposed abstract model for computation is being established as an integrative mechanism for structuring, speeding and featuring for nano-era.

**Keywords**: language-oriented architecture, parallel computing, abstract model of parallel computing,

## Introduction

The functional aspects supported by micro- and nano-technologies in the last half of century results from a strong correlation between the structural and architectural aspects. The size of the structure integrated on a silicon die and its architectural complexity have an intricate and unquiet history. The theoretical approach is still disturbed by the confusion made between ***size*** and ***complexity*** in evaluating a digital system. One of the main consequence is related with how the architectural features are managed on the emergent domain of parallel computing systems. It is very easy to integrate many Turing-based cores on a single chip, but we are faced with big problems in putting them at work efficiently. In July 2010 David Patterson wrote that:

> *"... the semiconductor industry threw the equivalent of a Hail Mary pass when it switched from making microprocessors run faster to putting more of them on a chip doing so without any clear notion of how such devices would in general be programmed. The hope is that someone will be able to figure out how to do that, but at the moment, the ball is still in the air."* [20]

Only starting from a good fit between size and complexity and an appropriate architectural approach we have a chance *to put the ball successfully in the end zone*.

In this paper we will try to synthesize the work done, in the last few decades, in order to find a solution for the obvious discrepancies between how the structure and function of digital systems evolve under the pressure of the Moore's Law. In the next sections are revisited problems related with the distinction between size and complexity, the featuring mechanism in digital systems, the abstract model of parallel computation. All these subjects are strongly related with the topics of functional electronics: the electronics of information embedded in circuits. The function in the big sized digital systems results only from a tightly interleaving process of informational structures with circuits.

The function of a big and complex electronic system is based on computation. The correlation between what computation is and what circuits can do is very important for making electronic

systems working efficiently, i.e., fast and low powered. Therefore, understanding functional electronics starts with what computation means.

In 1936, triggered by Kurt Gödel's seminal paper, published in 1931 [9], four mathematical computational models are published. The authors, Alonzo Church [6], Stephen Kleene [11], Emil Post [21], and Alan Turing [41], worked completely independent providing few different, but equivalent, mathematical descriptions for what computation is. Out of these, Turing's model fit best with the technological limitations of the 1940s. (Post's model is very similar with Turing's but, it supposed a too sophisticated memory model.) Thus, the first abstract models for the computing machines, the Harvard model[1] and the von Neumann model [19], are Turing inspired.

The success of Turing's model determines our stubbornness to use it even when the nature of computation changes dramatically. It is the case of the emergence of

1. Lisp-based artificial intelligence applications
2. parallel computation.

As we know, the Lisp language is based on Church's model, and, as we will show in this paper, parallel computation must be founded on Kleene's model. Consequently, a Turing based architecture supports with major difficulties applications written in Lisp, while an *ad hoc* gathering of Turing-based engines cannot provide an efficient environment for parallel computation.

The fate of Lisp has been cruel: its impact became very low after successive AI winters which have influenced negatively the computer science in the last decades of the 20th century. Meantime the emergence of parallelism has been marked by chaotic developments that are imposed by the market rather than by the academia. Today, the Lisp language has only few niche applications, while the parallel computation is dominated by structures derived from specific machines (GPUs become GPGPUs) or by collections of standard cores gathered on a single chip without any computationally justified reason. The deep incongruities between the computational requests and the structural offering lead to very low efficiency in using the huge brute force of the current multi- and many-core machines.

The Moore's Law supported and still supports a spectacular structural development of the brute computational force. We are technologically able to integrate on a silicon die thousands of computational units without a correspondent increase of the actual performance. How else can be explained our inability to use the huge computational peak performance of the current many-core engines?

Our search outside the Turing-based way of thinking in computation originates in 1980s with the project of the Lisp machine DIALISP and continued in the CONNEX[2] project, based on a concept imposed in the effort of surpassing the main limitations imposed by the too restrictive distinction between storing and processing in the conventional architectures.

The second section (*The Lisp Machine and the Connex Memory*) describes the project of Lisp machine DIALISP and the main concept derived from the effort to solve specific list processing issues: the Connex Memory. The next section (*Initial Applications of the Concept of Connex Memory*) reviews few exotic applications of the Connex Memory concept developed before the actual technology was able to support actual implementations. The fourth section (*From Connex*

[1] The term originated from the Harvard Mark I relay-based computer.
[2] http://users.dcae.pub.ro/~gstefan/2ndLevel/connex.html

*Memory to Connex Machine*) describes how the concept of Connex Machine emerged from the concept of Connex Memory as a form of an In-Memory-Processing engine. The theoretical foundation of our approach to parallel computing is presented in the fifth section (*Map-Scan/Reduce Abstract Model for Parallelism*) in which Kleene's model is made responsible for what parallel computing is. The next section (*Map/Scan-Reduce Accelerator Based Hybrid Computation*) introduces one of the most promising application of parallel computation: the hybrid computation based on Map/Scan-Reduce parallel accelerators. The last section (*Structuring, Speeding and Featuring at Nano-Scale*) shows how in nano-era the size growing must be reconsidered and substituted by an integrative growing mechanism in the three-dimension space of structure-speed-feature.

## The Lisp Machine and the Connex Memory

The Lisp language, in contrast to the languages like Fortran and Cobol inspired by the structure of the Turing-based computer, is founded on the *lambda-calculus* proposed by Alonzo Church for its computational model [6]. As a result, in 1973 at *MIT*, in *Computer Science and Artificial Intelligence Laboratory* a project for designing a computer optimized for executing efficiently Lisp programs started. At the end of the decade two startups were founded, *LispMachines Inc*. (1979) and *Symbolics Inc*. (1980). In this context, the project of the Lisp machine DIALISP started in the Functional Electronics Laboratory of Faculty of Electronics and Telecommunications from Polytechnic Institute of Bucharest. The resulting machine [25] is a Lisp accelerator for the DIAGRAM[3] system.

The system DIALISP[4] has as central unit a ***two-threaded microprogrammed processor*** with interleaved execution of two programs. The first microprogrammed thread supports the execution of the EVAL function while the second microprogrammed thread supports the implementation, in parallel, of the STACK funciton used by the EVAL function. Thus, the DIALISP processor is the first computing engine which executes interleaved microprograms on the same mono-core structure [25][35].[5]

Working in the theoretical environment of lambda-calculus [6] with Turing-based technologies [41] appeared fundamental incongruences. Under the pressure of incompatibilities between a specific data structure (the *list*) and an unsupportive storage device (the *random-access memory*) emerged the concept of ***Connex Memory*** [26] [27] [34].

**Definition 1**: The Connex Memory is the hardware support for a string of symbols each having two states: marked and unmarked. The following set of functions can be applied on the sting:

- **`reset s`**: all the variables ***after the first marked*** variable take the value ***`s`***
- **`find s`**: all the variables that follow a variable having the value ***`s`*** switch to the marked state and the rest switch to the unmarked state
- **`cfind s`**: all the variables that follow a marked variable having the value ***`s`*** switch to the marked state and the rest switch to the unmarked state

[3] https://ro.wikipedia.org/wiki/Diagram
[4] https://ro.wikipedia.org/wiki/DIALISP
[5] The concept of multithreading is introduced theoretically by B. J. Smith [22] for multi-core machines, while a first embodiment is reported previously by J. E. Thornton [40] as the mono-core interleaved multi-threaded Input-Output processor designed for CDC 6600. In CDC 6600 an interleaved implementation was possible due to the very slow peripherals used by the system.

- **`insert s`**: the value ***s*** is inserted before the ***first marked*** variable
- **`readup | readdown | read`**: the output takes the value of the ***first marked*** variable and the marker moves one position to right or to left (down) or remains unchanged
- **`delete`**: the value stored in the first marked position is deleted, the position remains marked (in the current cycle the output takes the value of the deleted variable) and the symbols from the right are moved one position left.

■

The Connex Architecture, instantiated as Connex Memory and as Connex Engine, represented in Fig. 1, consists of:

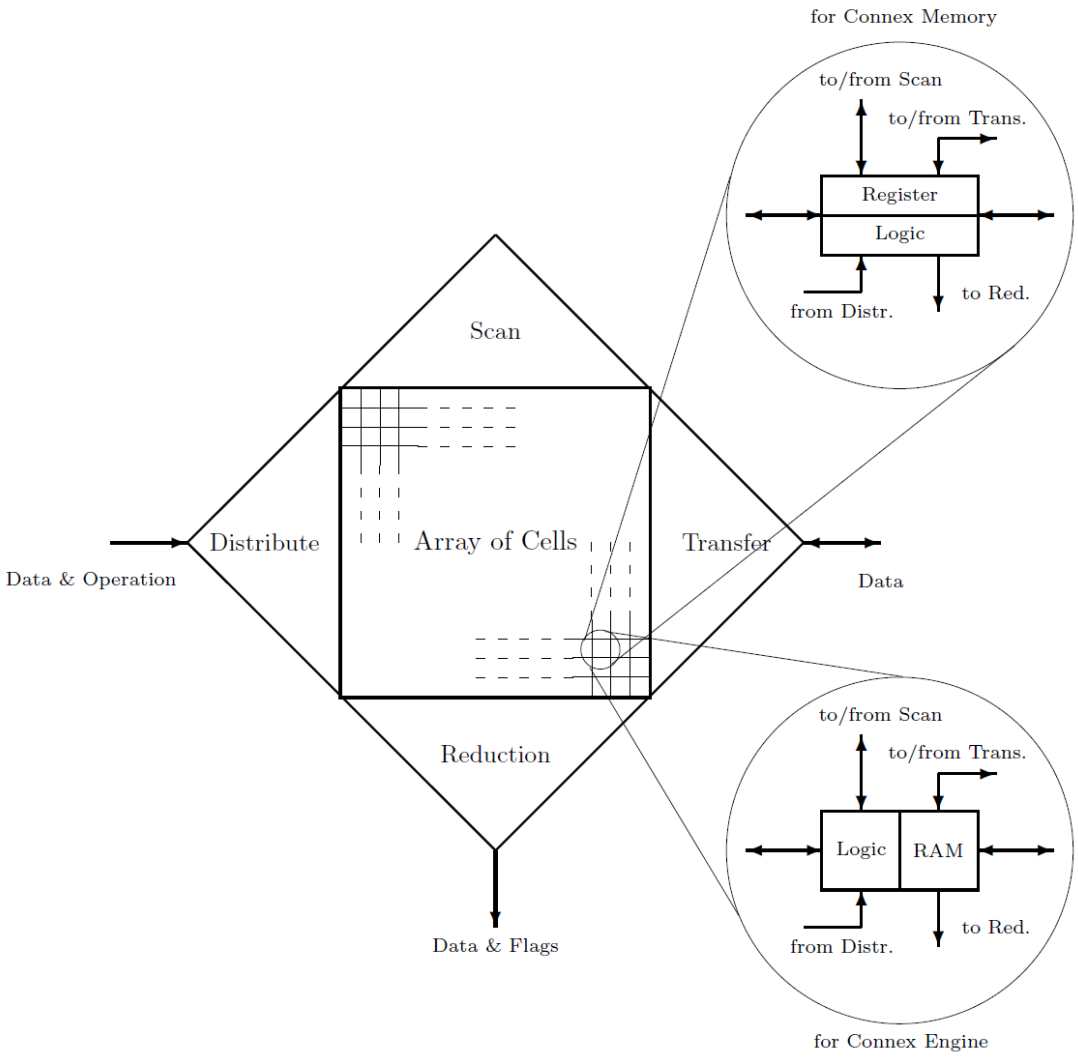


Fig. 1. ***Connex Architecture***. For the Connex Memory the storage resource per cell is a simple register, while for the Connex Engine version a RAM is used as storage support.

- a linearly organized Array of Cells, each having a register containing the pair

**{marker, value[n-1:0]}**

and the local logic block; each cell is two-direction connected to its left cell and right cell

- a *log*-depth Distribute network used to send a command (Data & Operation) in each clock cycle
- a *log*-depth Reduction network used to read the content of the value in the first marked cell or an information related to the markers
- a *log*-depth loop closed over the entire array performing Scan functions on the string of markers
- a two-direction Transfer unit used to exchange data serially.

The main operations applied on the list data structure are hardware supported by the Connex Memory. The main advantage is the possibility to avoid the garbage generated by the delete operation and consequently the garbage collector. The list remains "connected" avoiding the costly mechanism of pointers.

**Example 1**: Let be the list

…**(list1(a b (l a)))**…

positioned somewhere in the string of symbols stored in the Connex Memory. To access it and to substitute its first element with the one-component string **e,** the nest steps are followed:

```
find (
cfind l
cfind i
cfind s
cfind t
cfind 1
read
delete
insert e
```

The execution time is given by the length of the list's name and the strings involved. The content of the Connex Memory evolves, under the previous sequence of commands, as follows (the arrow under a symbol means that that symbol is marked):

```
…(list1(a b (l a)))…  // the initial state
  ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
…(list1(a b (l a)))…  // find (
   ↑      ↑     ↑
…(list1(a b (l a)))…  // cfind l
    ↑            ↑
…(list1(a b (l a)))…  // cfind i
     ↑
…(list1(a b (l a)))…  // cfind s
      ↑
…(list1(a b (l a)))…  // cfind t
       ↑
…(list1(a b (l a)))…  // cfind 1
        ↑
…(list1(a b (l a)))…  // read
```

```
          ↑
…(list1( b (l a)))…   // delete
          ↑
…(list1(e b (l a)))…  // insert e
          ↑
```

■

**Example 2**: In the first marked position a stack can be developed by assimilating the function **`insert`** with **`push`** and the function **`delete`** with **`pop`**.

■

The Connex Memory is a concept whose hardware implementation was far beyond the technological possibilities of the 1980s. Only at the beginning of the 2000's the silicon technology became able to support meaningful implementations of the concept.

The main features which differentiate the concept and the physical implementation of the Connex Memory are:

- the possibility to classify the elements of the string in
  - the first marked symbol
  - the symbols following the first marked symbols
- the possibility to access any point in the string using associative mechanisms in time independent of the length of the string
- the possibility to read, delete and insert in the position of the first marked symbol from the string

using structures with the size in *O(n)*, for *n*-cell Connex Memory.

The structure associated of this new kind of memory is a tightly interleaved net of storage cells with a net of simple processing structures. Thus, this approach is an important step on the way to avoid, or at least to minimize the effect of the "*von Neumann bottleneck*" which is characteristic for the Turing-based abstract models of computation (the Harvard and the von Neumann abstract models imposed form the 1940s). Church's approach of computation requests this fundamental reconsidering of the concept of memory. Random access memory is not a memory function, it is only a storage support. A memory function must have associated the access mechanism. Thus, a LIFO memory or a FIFO memory are true memories because include in their definition and implementation the accessing mechanism. Similarly, for the list data structure a specific physical support could be considered. The resulting concept and structure can be also used for other types of data structure to make efficient other application domains. The Connex Memory is an example which illustrates the emergence of a widely usable concept starting from a specific application.

While the concept of Lisp Machine promoted by the projects emanated from MIT laboratories (leading to the foundation of Lambda Machine Inc., Symbolics, Inc) kept the architecture of the hardware in the limit imposed by the Turing model (trying to cover the conceptual gap between Turing and Church within the space of the Turing-based abstract machines), the DISLISP project made two steps in departing from the conventional approach by:

- adding a new architectural layer by considering the engine as two-module structure – EVAL and STACK – tightly collaborating at the level of a two-thread processor

- proposing a new concept of memory – the Connex Memory – able to support at least two memory structures requested for the implementation of any Lisp machine: list and stack.

## Initial Applications of the Concept of Connex Memory

Prior to 2001, only theoretical investigation related to the Connex Memory concept were possible due to the technological imitation. Thus, few exotic applications were investigated.

### Artificial Life

In the 1990s, the concept of *Artificial Life* (AL) triggered a lot of theoretical researches aiming to improve the way computation is performed starting from the suggestions provided by the living systems. The concept of Connex Memory was used to support some implementations. In [28], starting from the concept of *eco-grammar systems* (EGS) introduced in the frame of AL, it is proposed a physical structure of "reasonable" complexity. The temptations to simplify the concept of EGS, in the aim of realizing it better, made our model more appropriate. Our simplifications permitted new mechanisms to be considered. One of them refers to the temporary evolution of the processes in an ecosystem, reconsidering parallelism at many levels. For the level for which parallelism is most critical, a specialized chip is defined: the *eco-chip*, starting from the concept of Connex Memory. The proposed architecture is formed by a network of processors working in the context of a network of eco-chips.

### Molecular Computing

Another AL domain *in vogue* at the beginning of the 1990s was molecular computing. Thus, the concept of Connex Memory has also been tested in this conceptual context [30] [31] [32]. We present in these papers the main ideas concerning the implementation in the solid-state circuits some of the *molecular mechanism*: the *splicing* operation and *insert/delete* operation. The physical support for these operations is based on the Connex Memory concept. We promote this solution because a pure biological process is very hard to be interfaced with machines implemented in nowadays technologies. In the same time, we believe that the mechanisms emphasized in the molecular process of computation are very good suggestions for the silicon-based machines requested to perform a fine grain parallelism. Using a Connex Memory, the splicing operation or the insert/delete operation is performed in linear time related to the length of the rules; the time does not depend on the length of the processed strings. In order to perform in parallel all possible applications of a rule in a set of strings, the function of the Connex Memory is extended over a two-dimension cellular automaton, thus defining the ***Eco-Chip***.

### Algorithmic Complexity

Algorithmic complexity (let us call it: ***Solomonoff-Kolmogorov-Chaitin complexity***) is a concept able to clarify, in the context of Moor's Law exponential growing mechanism, the necessary distinction between size and complexity of an entity, more specific an integrated electronic system. The confusion between size and complexity of digital systems can be removed using the concept of algorithmic complexity based on *Solomonoff-Kolmogorov-Chaitin complexity* [23][24][12][4][5].

**Definition 2**: The size of a digital system, *DS*, denoted by $S_{DS}$, is given by the number of maximum 2-input inverting gates used to implement it.

■

**Definition 3**: The complexity of a digital system, *DS*, denoted by $C_{DS}$, is proportional with the size of its minimal description.

■

**Definition 4**: A digital system *DS* is said complex if the complexity and the size are in the same range $C_{DS} \sim S_{DS}$.

■

**Definition 5**: A digital system *DS* is said simple if $C_{DS} << S_{DS}$.

■

A very big system cannot be very complex because its verification and testing cannot be done in a reasonable time with reasonable work power.

**Thesis**: While the size of digital system, $S_{DS}$, according to the Moore's Law increases exponentially in time, the complexity, $C_{DS}$, increases logarithmically in time. ■

In a fatal way, big digital systems must remain simple.

In this context, in [29] is described an implementation of Chaitin's ToyLisp on a Connex Memory Machine (CMM). The Connex Memory Machine has a smaller complexity than previous Universal Machines used to run Lisp programs, so the time and space used in running Lisp programs can be considerably decreased. A ToyLisp like language can be used with a CMM to construct a Lisp (Co)Processor for accelerating Lisp processing in conventional architectures. The current approach in Lisp implementation has imposed CAR and CDR as basic functions. Our different way of representing and processing the list in CM, as a "connex" string, may emphasize other basic functions such as: APPEND, MEMBER, LISTATOMS, INTERSECTION, UNION, DELETE, MATCH. Future work will be focused on optimizing the execution time and the size of memory, using graph reduction mechanisms.

**Theory of Computation**

One of the main mechanisms in symbol processing systems is based on *rewriting rules*. In such a system the dynamic data structure is modified by rules having the form: $\alpha \rightarrow \beta$ where $\alpha$ and $\beta$ are substrings over the system alphabet. The first (in Labeled Markov Algorithms) or all (in Lindenmeyer Grammars) the occurrences of $\alpha$ in the processed string are substituted with $\beta$. We choose two formalisms based on such mechanisms to justify the Connex Memory opportunity. The first is a sequential model and the second is a parallel one.

In 1954 Markov introduced a new computability model based on rewriting rules [18]. The main applications of this model are in symbolic computation. For example, the SNOBOL language is based on Labeled Markov Algorithms. We proved that a Connex Memory based architecture is a perfect match to implement it.

The Lindenmeyer Grammars [14] have spectacular applications in the Artificial Life domain. This formalism is based also on *rewriting rules* but implies applying the same rule in parallel in many places of the processed string. The initial CM version allows only a single access for INSERT or DELETE on the first marked place. The two-dimensioned structure of the cellular automaton

allows the "increase" or the "decrease" of the string simultaneously in many places. We called this extended Connex Memory version Eco-Chip, due to its Artificial Life applications.

The Markov Algorithms and the Lindenmeyer Grammars offer the theoretical support for the main Connex Memory applications: the ***rewriting systems***. We also introduce a parallel version (with multiple access) for the function of the Connex Memory. We called Eco-Chip a two-dimensioned cellular automaton used as support for a multiple accessed Connex Memory.

## From Connex Memory to Connex Machine

After 2000, when the size of the cache memories started to tend toward 1MB, the silicon technology reached the stage where the implementation of Connex Memory was possible at a size able to support real applications.

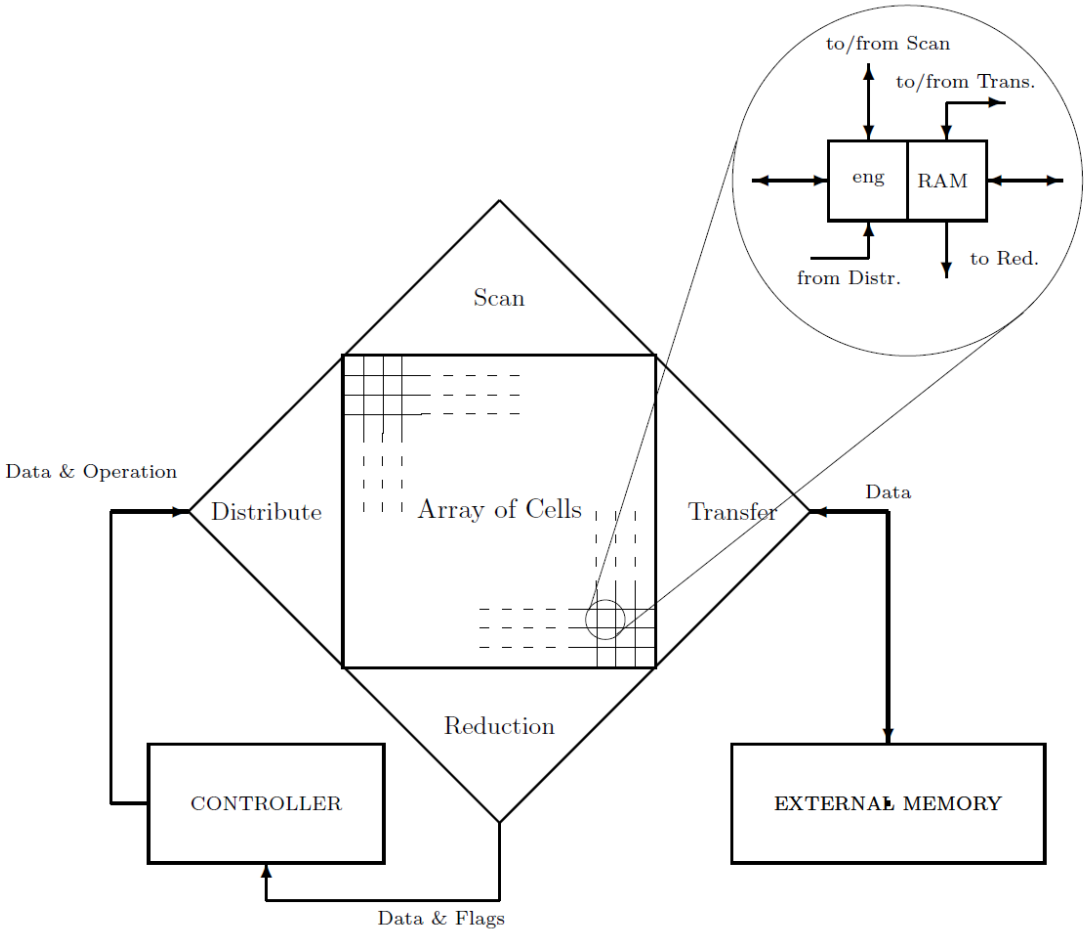


Fig. 2. Connex Machine.

In 2001 spring semester, during a half-year sabatical, the concept of Connex Memory is applied in bio-informatics [35] (the Human Genome Project has been completed and various searching mechanisms were investigated for real applications) and expanded to Connex Engine (see Fig. 1) enlarging the storage resources from one-word cell to many-word cells:

**{marker, localMemory[n-1:0][m-1:0]}**

Instead of a one-bit marker and a one-word storage, the Connex Engine contained a one-bit marker and a *m*-word *n*-bit random access memory[6].

Next step in the Connex evolution was made by upgrading the organization of the system at the level of Connex Machine by investigating the application in the domain of in-memory data-base.

**Example 3**: The code for Connex Machine (implemented in a version with accumulator-based execution units) which, starting from the index vector, compute the vector index multiplied by the sum of its even components, is:

```
-------------------------------------------------------------------------------------------------------------------------
      cSTART;      ACTIVATE; // start cycle counter; activate all cells
      cNOP;        IXLOAD;   // acc[i] <= i;
      cNOP;        VAND(1);  // acc[i] <= acc[i] & 00...01
      cVLOAD(x+1);WHEREZERO;// acc <= 1 + log_2 p; inactivate odd cells
LB(1);cBRNZDEC(1);IXLOAD;    // wait loop for latency; acc[i] <= i;
      cNOP;        ENDWHERE; // reactivate all cells
      cCLOAD(0);   IXLOAD;   // acc <= redAdd; acc[i] <= i
      cSTOP;       CMULT;    // stop cycle counter; acc[i]<=acc[i] * acc
      cHALT;       NOP;
-------------------------------------------------------------------------------------------------------------------------
```

The program has two columns, one for CONTROLLER (prefixed with **c**) and another for ARRAY of CELLS. The loop labeled with **LB(1)** is dimensioned according to the number of cells, $p = 2^x$.

The execution time for this program is:

$$T(p) = 9 + \log_2 p \in O(\log p).$$

The execution time is limited by the *log*-depth reduction network.

■

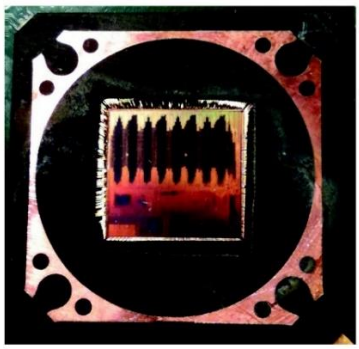

Fig. 3. The BA1024 chip.

[6] In the same time the patenting process started [43].

The first implementation for an industrial application was presented in Memorial Auditorium of Stanford University at *Hot Chips: A Symposium on High Performance Chips* [37]. In Febr. 2007 the BA1024 chip - a SoC for the HDTV market, containing a ConnexArray of 1024 engines, produced at TSMC (90 nm standard process) was the first one chip parallel machine (see Fig. 3) with the following performances:

- 200 GOPS (**G**iga 16-bit **OP**erations per **S**econd)
- >60 GOPS/Watt
- >2 GOPS/mm$^2$

The last silicon implementation was issued in March 2008: BA1024B - a SoC for the HDTV market, containing a ConnexArray of 1024 16-bit engines produced at TSMC (65 nm standard process) with the following performances:

- 400 GOPS (**G**iga 16-bit **OP**erations per **S**econd)
- >120 GOPS/Watt
- >6.25 GOPS/mm$^2$

The application of the BA1024B chip is real time frame rate conversion for HDTV.

## Map-Scan/Reduce Abstract Model for Parallelism

The emergence and the evolution of the Connex concept was somehow chaotic being driven by insights coming from various application domains. The efficiency of the successive physical implementations is hard to be justified only by good technical decisions made in solving some specific problems. There must be a theoretical motivation for this approach. Thus, starting from 2008, the Connex concept was investigated based on its deep mathematical foundations.

### The State of the Art

Why the Connex approach is considered efficient? Besides high energy efficiency and high area use, the main point is the high *effective performance vs. peak performance* ratio achieved by the Connex based systems.

How performs the "competition"? The main competitors are various embodiments of the x86 architecture (the SSE supported cores or Xeon Phi family) and Nvidia family of GPUs. Let us see how these commercially off-the-shelf solutions work.

The switch from the mono-core architecture to the many-core architecture is a pure quantitative, market driven process. Indeed, the main and almost only criteria for defining parallel architecture is to provide on a single chip simple *ad hoc* geometrically configurated networks of many processing elements. Let us show using few examples how the performance of the computing system improves by switching from mono- to many-core approach.

***The actual vs. peak performance of many-core accelerators***. Let us take a very frequently used computationally-intense operation: matrix-vector multiplication. While according to [15] a mon-core system uses 16-50% from its peak performance for matrix-vector multiplication, according to [42], the NVidia architecture activates only ~1% from its huge computational power for the same kind of computation, and in [8] for the same computation the same percentage from the peak performance is used by a Xeon Phi engine.

***Accelerating TensorFlow library in a deep learning task***. In [13] a deep learning task, such as training CNN on Cifar-10 dataset using tensorflow/models, is used to compare the execution speed on four systems:

- CPU 7th gen i7–7500U, 2.7 GHz (from my Ultrabook Samsung NP-900X5N), a 2-core multi-core CPU
- 2 x AMD Opteron 6168 1.9 GHz Processor (2x12 cores total) taken from PowerEdge R715 server
- GPU NVidia GeForce 940MX, 2GB (also from my Ultrabook Samsung NP-900X5N), a 384-core parallel machine
- GPU NVidia GeForce 1070, 8GB (ASUS DUAL-GTX1070-O8G) from my desktop, a 1920-core parallel machine

The acceleration provided the 24-core AMD engine was insignificant. The acceleration provided by the 384-core NVidia machine was ~3x, while the acceleration of the 1920-core NVidia was ~15x.

How can an 8.2 TFLOPS engine (GeForce 1070) accelerate ***only*** 15x a computation done by a 13.3 GFLOPS engine (i7–7500U)? The peak performance of the many-core machine is 616x higher than the peak performance of the 2-core CPU, but the acceleration is only 15x!

***Many-core vs. mono-core in encryption***. In [10] is compared a CPU having a single 2.66 GHz core with the 512-core GTX580 engine. The function used as benchmark is RSA encryption. The experiment shows 23x to 32x speedup.

All the three examples show that the many-core machines designed using *ad hoc* gatherings of cores, without any theoretically based computational reason, are unable to fructify in real applications the huge computational power resulting from summing the peak performance of each core. The computational power is not provided only by summing the power of components, it is also important how the components interact each other for providing an actual performance not too far from their overall peak performance. It looks like there are at least some architectural issues to be fixed, if not a deeper investigation must be done. There seems to be a lot to learn from the success story of mono-core computing.

**The Main Steps Made by the Mono-Core Computation**

The unsuccessful story of the decision problem - formulated by David Hilbert - which ended by the most important negative result in the history of mathematics, - the Gödel's theorem - started the most successful project in the history of technology: the information technology project. The main stages of the first period – the mono-core era – were [38]:

1. 1936 – computational models: four equivalent models are published [6] [11] [21] [41] (all reprinted in [7]), out of which the *Turing Machine* offered the most expressive and technologically appropriate suggestion for near future developments
2. 1944-45 – abstract machine models: MARK 1 computer, built by IBM for Harvard University, consecrated the term *Harvard abstract model*, while von Neumann's report [19] introduced what we call now the *von Neumann abstract model*; these two concepts backed the *RAM* (random access machine) abstract model used to evaluate algorithms for sequential machines
3. 1953 – manufacturing in quantity: IBM launched *IBM 701*, the first large-scale electronic computer

4. 1964 – computer architecture concept is introduced to allow independent evolution for the two different aspects of computer design, which have different rate of evolution: software and hardware; thus, there are now on the market few stable and successful architectures, such as x86, ARM, PowerPC.

What we learned from this evolution is that the stage 4 must be precede the stage 3, so as the normal flow of facts is:

1. Mathematical computational model
2. Abstract machine model
3. Architectural design
4. Manufacturing in quantity

Unfortunately, the history of many-core computation (parallel or distributed computation) did not follow the same way. It started with manufacturing in quantity (symmetrical MIMD engine introduced in 1962 on the computer market by Burroughs) and continued with confusing the mathematical computational model with the abstract machine model (see for details [38]).

**Kleene's Mathematical Model for Parallel Computation**

Although Kleene's model was published in the same year with of Turing, its technological impact was almost null. We reconsidered Kleene's partial recursive functions[7] by using it as a genuine mathematical computational model for parallel computation [16] [17]. Our approach is based on the following two theorems proved in [38].

**Theorem 1**: *The primitive recursive rule is reducible to repeated applications of specific compositions.*

**Theorem 2**: *The minimization (least-search) rule is reducible to repeated applications of specific compositions.*

**Corollary 1**: *Any computation can be done, according to Theorem 1 and Theorem 2, using the initial functions and the repeated application of the composition rule.*

Based on the previous theorems we define the concept of Kleene Machine. Informally, it is defined in Fig. 4, where the functional modules $h_i$ receives the input variable $X$ and the state of the left and right functional modules generating a value for the reduction function and a new state of the functional module. As with the Turing Machine which has the associated Universal Turing Machine, we must define for Kleene Machine an Universal Kleene Machine. In Fig. 5 the informal definition of the Universal Kleene Machine is defined. It consists of a "programmable" Kleene Machine loop connected with a Universal Turing Machine. Each cell, $c_i$, receives besides the data

[7] **Definition**: *Let be the positive integers x, y,* $i \in \mathbf{N}$ *and the sequence* $X = \langle x_0, x_1, \ldots, x_n \rangle \in \mathbf{N}^n$. *Any partial recursive function* $f : \mathbf{N}^n \rightarrow \mathbf{N}$ *can be computed using three initial functions:*

- $ZERO(x) = 0$ : *the variable x takes the value* zero
- $INC(x) = x+1$ : increments *the variable* $x \in \mathbf{N}$
- $SEL(I,X)=x_i$ : *i* selects *the value of* $x_i$ *from the sequence of positive integers X*

*and the application of the following three rules:*

- Composition: $f(X) = g(h_1(X),, \ldots, h_p(X))$, *where:* $f : \mathbf{N}^n \rightarrow \mathbf{N}$ *is a total function if* $g : \mathbf{N}^p \rightarrow \mathbf{N}$ *and* $h_i : \mathbf{N}^n \rightarrow \mathbf{N}$, *for* $i = 1, \ldots, p$, *are total functions*
- Primitive recursion: $f(X,y) = g(X, f(X, (y\text{-}1)))$, *with* $f(X,0) = h(X)$ *where:* $f : \mathbf{N}^{n+1} \rightarrow \mathbf{N}$ *is a total function if* $g : \mathbf{N}^{n+1} \rightarrow \mathbf{N}$ *and* $h : \mathbf{N}^n \rightarrow \mathbf{N}$ *are total functions.*
- Minimization: $f(x) = my[g(x,y) = 0]$, *which means: the rule computes the value of the function* $f : \mathbf{N} \rightarrow \mathbf{N}$ *as the smallest y, if any, for which another function* $g : \mathbf{N}^2 \rightarrow \mathbf{N}$ *takes the value* $g(x,y) = 0$.

to be processed, $X$, a code, $h_i$, used to select from a finite set of functions the function to be applied to $\{X, r_{i-1}, l_{i+1}\}$, while the reduction function of R is selected by the code $g$.

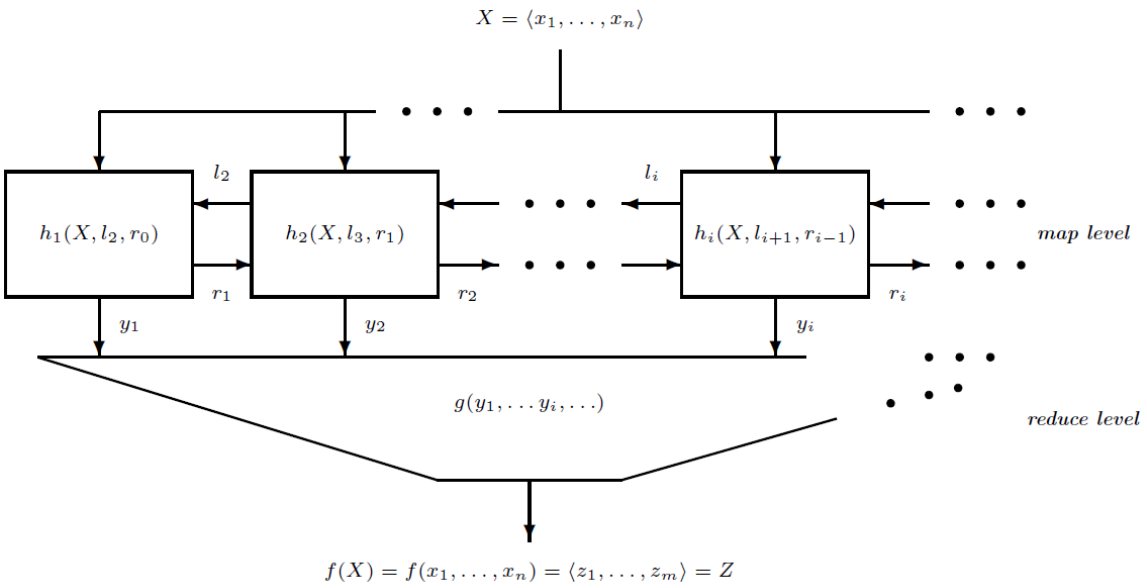


Fig. 4. Kleene Machine. The *synchronic parallelism* is performed on the map level and the *diachronic parallelism* works between the map level and the reduce level.

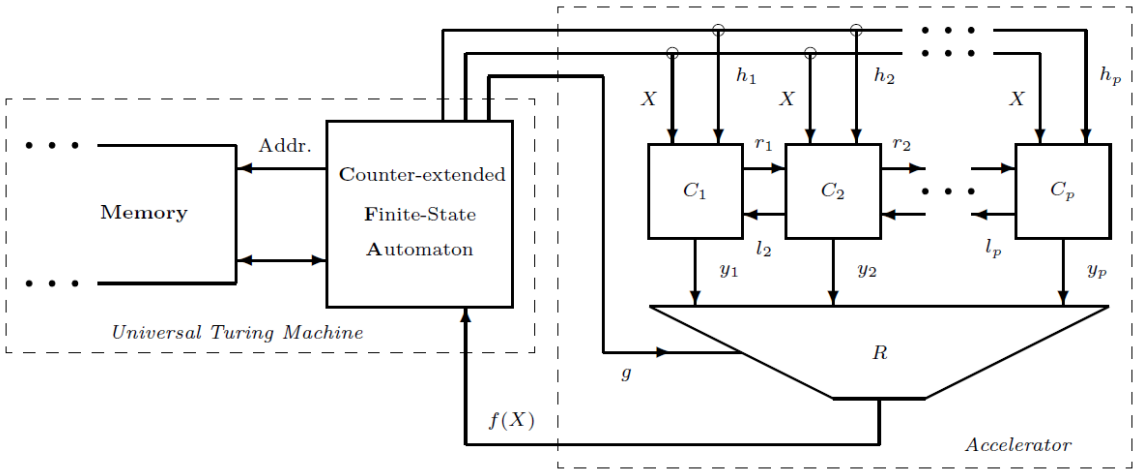


Fig. 5. Universal Kleene Machine: seen as an Accelerated Universal Turing Machine.

The programmable Kleene Machine works like an accelerator for the Universal Turing Machine seen as a host machine. The pair host-accelerator can be interpreted as the segregation of two engines, one dealing with the scalar operations and another involved with vector operations.

**Abstract model for parallel computation**

Starting from the formal structure of the Universal Kleene Machine, the *abstract model for parallel computation* is provided in Fig. 6, where a hierarchical recursive definition is established. At each level there is defined a cell as the pair **eng(i)+mem(i)**, where the engine, *eng*, starts, for *i=0*, from a simple execution/processing unit, while the memory, *mem*, starts from a small static RAM. For *i=1*, the REDUCE *log*-depth network is a circuit with few functions (*add, min, max,* …). For p cells, the size of the engine is in *O(p)*. There are two difficulties to be surpass:

1. The *log*-depth of the REDUCE network
2. The bottleneck between ***eng*** and ***mem***

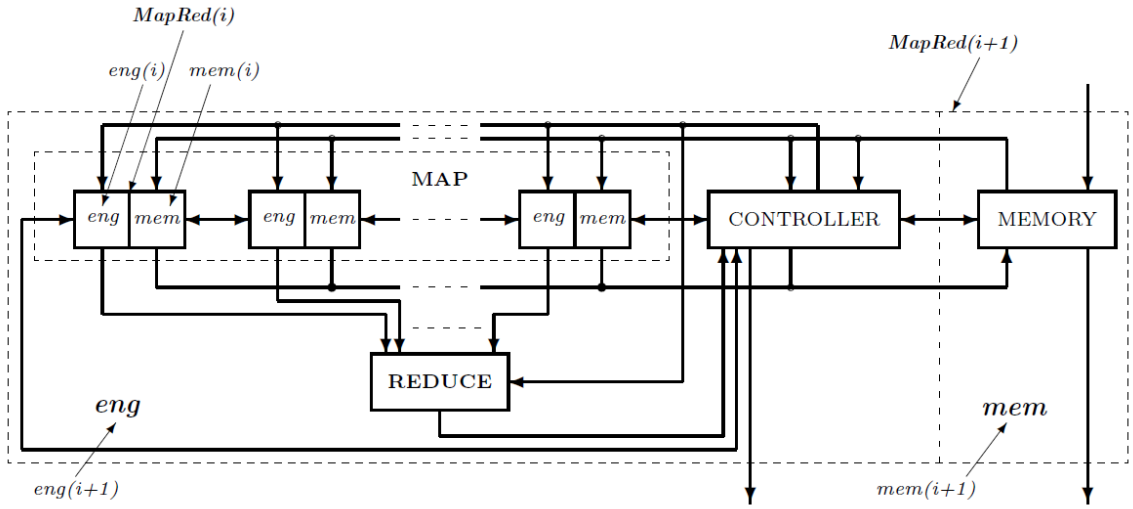


Fig. 6. *MapReduce recursive abstract model* for parallel computation.

The first one can sometimes be avoided at the algorithmic level.

**Example 4**: The matrix vector multiplication uses the REDUCTION section for performing the addition operation for the inner product between the vector and each line of the matrix. For an $N \times N$ matrix the latency of *O(log p)* should have been added *N* times. The left/right connections between the cells allow the implementation of a shift register along the MAP section. The output of the REDUCTION section can be pushed left or right into this serial register (cPUSHL/ cPUSHR), so as only at the end of the N multiplication $log_2 p$ clock cycles must be added to allow the push of the last inner product into the serial register. The code is:

```
-------------------------------------------------------------------------------------------------------------------
       cNOP;          STORE(W);    // mem[i][W] <= acc[i] = V[i]
       cVLOAD(N);     RLOAD(0);    // acc <= N; acc[i] <= firstMatrixLine
       cVSUB(1);      MULT(W);     // acc <= N-1; acc[i]<= acc[i]* mem[i][W]
LB(M);cCPUSHL(0);     RILOAD(255);// push redSum; acc[i] <= nextMatrixLine
       cBRNZDEC(M;    MULT(W);     // loop control; acc[i]<=acc[i]*mem[i][W]
       cVLOAD(S);     NOP;         // load for latency loop
LB(L);cBRNZDEC(L);NOP;            // latency loop (waits for the last IP)
        cNOP;         SRLOAD;       // load result in acc[i]
-------------------------------------------------------------------------------------------------------------------
```

The execution time for $N \leq p$ is:

$$T_{matrixVectMult}(N,p) = 2N + log_2 p + 5 \in O(N)$$

because the loop labeled with LB(M) runs N times, while the loop labeled with LB(L) runs $log_2 p$ times. The acceleration is supra-linear due to the following three processes running in parallel:

- the MAP computation (performing *N* multiplications in parallel)
- the REDUCTION computation (performing the *N-1* addition on $log_2 p$ pipeline levels)
- the control computation (running the two control loops)

■

Unfortunately, the bottleneck can not be avoided for I/O bounded applications. The only ways to attenuate the effect of the bottleneck are:

- to increase the size of the local memories, mem
- to execute in parallel the transfer and the computation.

For *eng(i)*, with $i > 1$, the REDUCTION network could be a *log*-depth tree-like network of processors/computers.

Because the reduction network returns in fact the last value computed by a scan network, a generalization of the Map-Reduce abstract machine is a Map-Scan abstract machine. A scan network is only $(2 - (log_2 n)/(n-1))$ times bigger than a reduction network defined for the same function. Thus, if the application domain requests scan functions, then substituting the reduction network with a scan network is not a big issue.

## Map-Scan/Reduce Accelerator Based Hybrid Computation

A real hard application is always a mixture of complex and intense computation. The complex computation is expressed by big sized code and small computation time, while the intense computation is expressed by a small code and big computation time. In order to implement this kind of applications efficiently, specific hardware must be used for the two types of computation. The emergent and almost imposed solution is the hybrid system of computation. A hybrid system consists of a host computer and an accelerator. The host computer runs the complex part of code, while the accelerator runs the intense part of code. The host computer is a Turing-based engine which controls the entire process. Currently, mono- and multi-core are used as hosts. For the accelerator part of the hybrid system a good solution, but not the only one, is a many-core parallel engine.

There are few solutions for the accelerator part of a hybrid computing system. The most used are:

- a parallel engine (the of-the-shelf solution is the Nvidia's GPU or Intel's Xeon Phi)
- an ASIC, when the accelerated part of the computation deserves such a big effort
- a FPGA implemented accelerator, whose structural flexibility is used to configure various circuits

About of-the-shelf solutions we discussed above we already claimed that their efficiency is very low due to the small value of the ratio (effective performance)/(peak performance). An ASIC is very rarely efficient because it is very costly. The FPGA solution request a very well designed

compiler (from a high level programming language, such as C++, to a HDL, such as Verilog) or very rarely accessible skilled digital designers.

Consequently, our proposal is a combination of the previous three solutions: we provide a ***parameterized and configurable*** Verilog design of a ***parallel engine*** to be implemented in FPGA which provide an almost ***circuit performance***. In Fig. 7 the proposed hybrid system is presented. The interface between HOST and MAP-SCAN ARRAY used as accelerator contains three asynchronous FIFOs, two for data transfer and one for receiving the stream of functions and parameters for accelerator. The use of the system is the following:

- the generic design of the accelerator, as a Verilog parametrized and configurable code, is available to the programmer
- the programmer writes the code for the main program, in a high-level programming language, and the code functions to run on the accelerator using the specific environment of the accelerator
- the Verilog design is actualized for the parameters requested by the resulting program
- the Verilog design is configured maintaining only the instructions and physical resources used by the resulting program
- the resulting design is loaded in FPGA
- the main program sends the code for the accelerated functions through *progFIFO* to the accelerator.

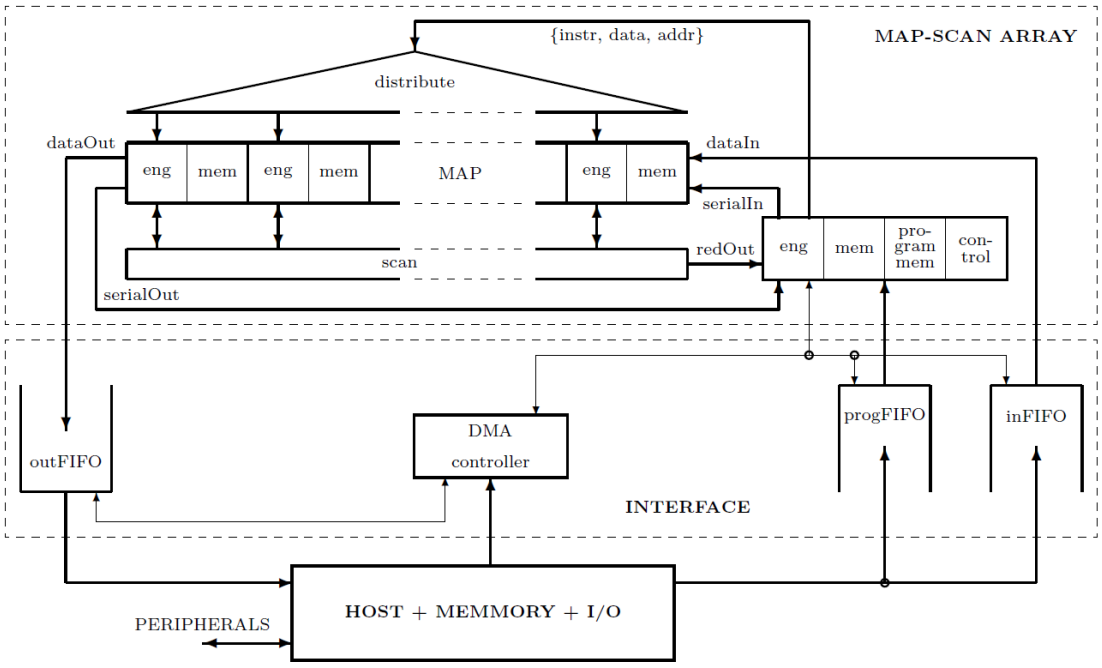


Fig. 7. *Hybrid Computing System* based on the MapReduce circuit family

This approach maintains the use of the system at the level of pure programming instead of an use which requests, besides programming abilities, advanced digital design skills performed by a human designer or by a compiler.

In this early stage of development, programming the Map-Scan accelerator means to develop the kernel of a function library which is expanded to the library by the programs running on the host. For example, implementing the *Eigen* library on the hybrid system is done by implementing a kernel defined on limited data structure to the size of the array of cells, *p*. Thus, the multiplication of an array of *4p×4p* components, for example, is done performing on accelerator 8 multiplications of sub-arrays of *p×p* components and then 4 additions of *p×p* matrices. At the level of *p×p* matrices the computation is (supra)accelerated *p* times (with time in $O(N^2)$), while at the level of multiplying matrices of *2×2* matrices of *p × p* matrices the computation remains in $O(N^3)$.

## Structuring, Speeding and Featuring at Nano-Scale

Moor's Law tells us how the size of the systems integrated on silicon grows. But, how the systems actually grow is a threefold process related to structuring, speeding and featuring. In [39], the structuring mechanism is a serial-parallel one, according to the composition rule introduced by Stephen Kleene in its definition of partial recursive functions (see Fig. 8). The map-composition is a parallel expanding mechanism, while the serial expanding is provided by the reduction-composition. Speeding is based on pipeline connection. The third mechanism of how new features are added in a digital system is based on adding loops which increase the autonomy of the system. In Fig. 9 the mechanism is exemplified.

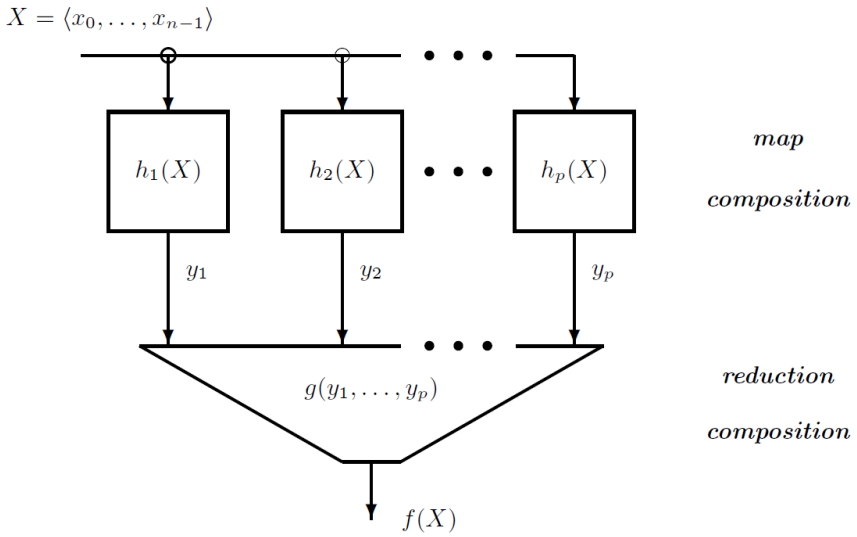


Fig. 8. Structuring by composing.

For the emerging nano-era, the threefold mechanism must become an integrated one. Structuring, speeding and featuring must be governed by one mechanism only: the recursive abstract model for parallel computation emphasized in Fig. 6. The map-reduce abstract model provides in the same time support for size, speed and complexity tightly interleaving the physical structure of circuits with the informational structure of programs running at different level of the system. The growing mechanism is governed by a good balance between size, speed and complexity provided by an appropriate step performed by adding a new level in the recursive hierarchy. The art of using the recursive evolving mechanism is to find the right "moment", in the three dimensioned space of size-speed-complexity, for adding a new recursive level in the hierarchy such that to obtain the maximum efficiency in the other three-dimension space of performance-energy-area.

Thus, in the nano-domain, the structure, the speed and the features involved in an application cannot be managed independently. The informational hierarchy of programs imposes mainly how the hardware hierarchy is organized.
Having at the top level the same system shape – a map-reduce engine connected with a memory – independent of the number of hierarchical levels, keeps the complexity of the entire construct simple.

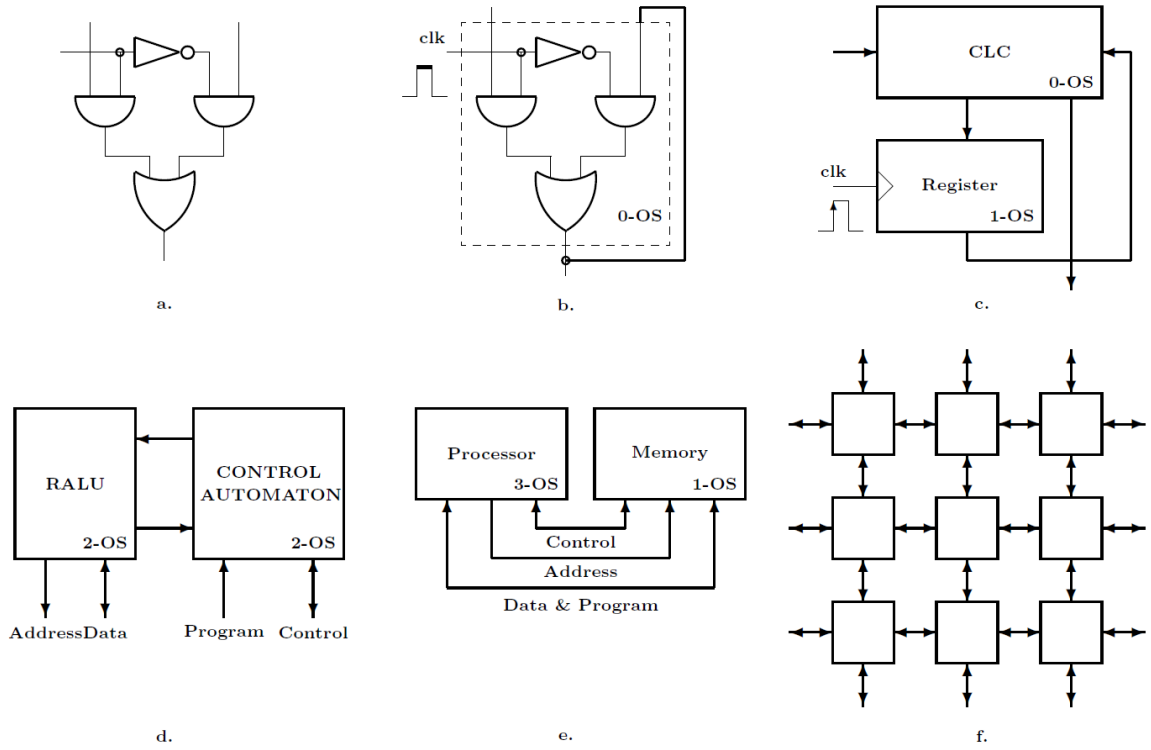


Fig. 9. The loop-based featuring mechanism. **a**. Zero order systems: no-loop circuits. **b**. First order systems: the 1-loop, memory circuits. **c**. Second order systems: the 2-loop, automata circuits. **d**. Third order systems: processors. **e**. Fourth order systems: computers. **f**. *N*-th order systems: the *O(n)*-loop cellular automaton.

The map-reduce shape turns out once more that is a foundational principle. From the level of elementary combinational circuits to the level of cloud computing the map-reduce mechanism is applicable [2]. The nano circuits must and maybe will enter in this implacable order using the map-reduce principle as a supportive principle.

## Concluding Remarks

In nano-era (under *32 nm* technologies[8]) the distinction between size and complexity become meaningful from architectural point of view. The growing process acquires multiple dimensions. The structural grow must be carefully corelated with speed and the functional specter. Thus, ***the growing process becomes an integral one*** evolving in the three-dimension space of size-speed-complexity.
Technological evolutions triggered a fundamental reconsidering of the theoretical environment for the architectural developments of digital systems. Turing-based architectures need to be gradually integrated with abstract models that better fit the large structures allowed by the unforgivable action

[8] Let us consider 32 nm geometrically five-step equally distant from both limits, *1 μm* and *1 nm*.

of Moore's Law. The Church's and Kleene's models offered new architectural perspectives for the huge structural possibilities of the nano-technological environment. Ignoring them already led to incongruent architectural developments.
The transition from the dominant Turing-based conceptual context, developed under technological restrictions long overdue to a new conceptual context fit to the nano-level technological opportunities, is an avatar illustrated in this paper by few of its aspects. A first step was the two-threaded functional approach used in designing the Church-based architecture of the Lisp machine DIALISP. Then, as a normal follow-up, the concept of Connex Memory emerged leading to the recursive Map-Reduce abstract model. And finally, the Map-Reduce abstract model is proposed as the integrative growing mechanism for nano-era: any growing process must take into account simultaneously all the three dimensions of the space size-speed-complexity.

**Acknowledgments**. The author got a lot of support from the main technical contributors to the development of the Lisp machine *DIALISP*, and *ConnexArray*$^{TM}$ technology, the *BA1024* chip, the associated language, and its first application: Emanuele Altieri, Andy Birnbaum, Virgil Bistriceanu, Călin Bîra, Frank Ho, Sanda Maican, Mihaela Malița, Bogdan Mîțu, Aurel Păun, Marius Stoian, Dominique Thiébaut, Tom Thomson, Dan Tomescu.

## References

[1] L. M. ADLEMAN, *Molecular Computation of Solutions to Combinatorial Problems*, Science, 226 (Nov. 1994), pp 1021 - 1024.
[2] R. ANDONIE, M. MALIȚA, G. M. ȘTEFAN: MapReduce*: From Elementary Circuits to Cloud*, in Kreinovich, Vladik (Ed.): *Uncertainty Modeling*, Springer, 2017, pp. 1-14.
[3] K. ASANOVIC, et al., The landscape of parallel computing research: A view from Berkeley, 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.
[4] G CHAITIN, *On the Length of Programs for Computing Binary Sequences*, J. of the ACM, Oct. 1966.
[5] G CHAITIN, Algorithmic Information Theory, Cambridge Univ. Press, 1987.
[6] A. CHURCH, *An unsolvable problem of elementary number theory*, The American Journal of Mathematics 58, 1936. 345-363. (Republished in [7])
[7] M. DAVIS, *The Undecidable. Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*. Dover Publications, Inc., Mineola, New-York, 2004.
[8] P. GEPNER et al., *Evaluation of DGEMM Implementation on Intel Xeon Phi Coprocessor*, Journal of Computers, Vol. 9, no. 7, July 2014.
[9] K. GÖDEL, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I*, Monatshefte für Mathematik und Physik **38**: 1931,173–98. (Republished in English version in [6])
[10] K. JANG, et al., *SSLShader: Cheap SSL Acceleration with Commodity Processors* [Online]. Available: https://pdfs.semanticscholar.org/c013/1d42fa76c8232bff6fb6cd2005a3f7d61596.pdf
[11] S. KLEENE, *General recursive functions of natural numbers*. Mathematische Annalen 112, 5, 1936. 727-742 (Republished in [7])
[12] A. A. KOLMOGOROV, *Three Approaches to the Definition of the Concept Quantity of Information*, Probl. Peredachi Inform., vol. 1, 3-11, 1965.
[13] A. LAZORENKO, *TensorFlow performance test: CPU VS GPU*. [Online]. Available: https://medium.com/@andriylazorenko/tensorflow-performance-test-cpu-vs-gpu-79fcd39170c
[14] A. LINDENMEYER, *Mathematical Models of Cellular Interactions in Development I, II*, Journal of Theor. Biology, 18, 1968.
[15] S. MACKE, *Optimizing Large Matrix Vector Multiplication*
[Online]. Available: https://simulationcorner.net/index.php?page=fastmatrixvector
[16] M. MALIȚA and G. M. ȘTEFAN *On the Many-Processor Paradigm*, Proceedings of the 2008 World Congress in Computer Science, Computer Engineering and Applied Computing, vol. PDPTA'08 (The 2008 International Conference on Parallel and Distributed Processing Techniques and Applications), 2008.
[17] M. MALIȚA, G. M. ȘTEFAN and 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, Special issue: *ALPS '07 - Advanced low power systems*; communication at *International Workshop on Advanced Low Power Systems* held in conjunction with 21st International Conference on Supercomputing June 17, 2007 Seattle, WA, USA.

[18] A. A. MARKOV, *The Theory of Algorithms*, *Trudy Matem. Instituta* in V. A. Steklova, vol. 42, 1954. (Translated from the Russian by J. J. Schorr-kon, U. S. Dept. of Commerce, Office of Technical Services, no. OTS 60-51085, 1954)
[19] J. VON NEUMANN, *First draft of a report on the EDVAC*. IEEE Annals of the History of Computing 5, 4, 1993.
[20] D. PATTERSON, *The trouble with multicore*. IEEE Spectrum, 47(7):28–32, July 2010.
[21] E. POST, *Finite combinatory processes. formulation I*. The Journal of Symbolic Logic 1, 1936. 103-105 (Republished in [7])
[22] B. J. SMITH: *A Pipelined, Shared Resource MIMD Computer*, Proc. of the 1978 Intl. Conf. on Parallel Processing, pp. 6-8, August 1978.
[23] R. J. SOLOMONOFF, *A Formal Theory of Inductive Inference Part I*, Information and Control. **7** (1): 1–22.
[24] R. J. SOLOMONOFF, *A Formal Theory of Inductive Inference Part II*. Information and Control. **7** (2): 224–254.
[25] G. M. ȘTEFAN, A. PĂUN, A. BIRNBAUM, and V. BISTRICEANU, *DIALISP - A LISP Machine*, The 1984 ACM Symposium on LISP and Functional Programming, Austin, Texas, USA, August 06 – 08, p. 123-128.
[26] G. M. ȘTEFAN, V. BISRICEANU, A. PĂUN, Către un mod natural de implementare a LISP-ului" (Toward a Natural Way to Implement LISP), *Sisteme cu inteligență artificială* (Systems with Artificial Intelligence), Ed. Academiei Române, București, 1991 (paper at *Al doilea simpozion național de inteligență artificială* (The Second National Symposium on Artificial Intelligence), Sept. 1985[9]). p. 218 - 224.
[27] G. M. ȘTEFAN, *Memorie conexă, CNETAC 1986* Vol. 2, IPB, București, 1986, p. 79 - 81. (In Romanian)
[28] G. M. ȘTEFAN and M. MALIȚA, The Eco-Chip: A Physical Support for Artificial Life Systems, in Gh. Paun (ed.) *Artificial Life. Grammatical Models*, BSU Press, 1995. p. 260 - 275.
[29] G. M. ȘTEFAN and M. MALIȚA, *Chaitin's Toylisp on a Connex Memory Machine*, Journal of Universal Computer Science, Vol. 2, No. 5, May 28, 1996, p.410 - 426. [Online]. Available: http://www.jucs.org/jucs_2_5/chaitin_s_toylisp_on/Stefan_G.pdf

[30] G. M. ȘTEFAN and M. MALIȚA, *DNA Computing with the Connex Memory*, RECOMB 97 First International Conference on Computational Molecular Biology, January 20 - 23, Santa Fe, New Mexico, 1997. p. 97-98.
[31] G. M. ȘTEFAN and M. MALIȚA, *The Splicing Mechanism and the Connex Memory*, Proceedings of the 1997 IEEE International Conference on Evolutionary Computation, Indianapolis, April 13 -16, 1997. p. 225-229.
[32] G. M. ȘTEFAN, *Silicon or Molecules? What's the Best for Splicing*, G. Paun (ed.) Computing with Bio-Molecules. Theory and Experiments, Springer, 1998. p. 158-181
[33] G. M. ȘTEFAN and R. BENEA, *Connex Memories & Rewriting Systems*, MELECON '98 9th Mediterranean Electrotechnical Conference, May 18-20, 1998, Tel-Aviv, Israel. p 1299-1303.
[34] G. M. ȘTEFAN, *The Connex Memory: A Physical Support for Tree / List Processing"*, The Roumanian Journal of Information Science and Technology, Vol.1, Number 1, 1998, p. 85 - 104.
[35] G. M. ȘTEFAN, *A Multi-thread Approach in order to Avoid Pipeline Penalties*, Proceedings of 12th International Conference on Control Systems and Computer Science, Vol. II, May 26-29, 1999, Bucharest, Romania. p. 157-162.
[35] G. M. ȘTEFAN and D. THIÉBAUT, *Hardware-Assisted String-Matching Algorithms*, WABI 2001, 1st Workshop on Algorithms in BioInformatics, BRICS, University of Aarhaus, Danemark, August 28-31, 2001.
[37] G. M. ȘTEFAN, A. SHEEL, B. MÎȚU, T. THOMSON and D. TOMESCU, *The CA1024: A Fully Programable 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://www.youtube.com/watch?v=HMLT4EpKBAw&feature=youtu.be at 35:00.
[38] G. M. ȘTEFAN and M. MALIȚA, *Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation*, 18th International Conference on Circuits, Systems, Communications and Computers, Santorini Island, Greece, pp. 582–597, 2014.
[39] G. M. ȘTEFAN, *Loops & Complexity in Digital Systems. Lecture Notes on Digital Design in the Giga-Gate/Chip Era* [Online]. Available: http://users.dcae.pub.ro/~gstefan/2ndLevel/teachingMaterials/0-BOOK.pdf
[40] J. E. THORNTON: *Parallel Operation in the Control Data 6600*, AFIPS Conference Proceedings FJCC, part 2, vol. 16, 1964.
[41] A. M. TURING, *On computable Numbers with an Application to the Eintscheidungsproblem*, Proc. London Mathematical Society, 42 (1936), 43 (1937). (Republished in [7])

[9] The publication of the proceedings of the conference was delayed due to political reaons. A 4th author - Andi Birnbaum - could not be listed as author due to problems related to its emigration.

[42] *** *Matrix-vector multiplication in CUDA: benchmarking & performance* [Online]. Available: http://stackoverflow.com/questions/26417475/matrix-vector-multiplication-in-cuda-benchmarking-performance
[43] Connex related patents [Online]. Available: http://users.dcae.pub.ro/~gstefan/publications.html

**Gheorghe M. Ștefan** teaches digital design in *Politehnica* University of Bucharest. Its scientific interests are focused on digital circuits, computer architecture and parallel computation. In the 1980s, he led a team which designed and implemented the Lisp machine DIALISP. In 2003-2009 he was, as co-founder and Chief Scientist, with *Brightscale*, a Silicon Valley start-up which developed the BA1024, a 1024-core chip for the HDTV market. More at http://users.dcae.pub.ro/~gstefan/.