On the Many-Processor Paradigm

Mihaela Malița[1], and **Gheorghe Ştefan**[2]
[1]Saint Anselm College, Manchester, NH, [2]BrightScale, Inc., Sunnyvale, CA,

Abstract – *We say* **multi***-processor if there is more than one processor, while* **many***-processor is used to designate a system with big-n processors. We claim that the theoretical foundation for the two kinds of parallel machines is different and is meaningful for understanding the evolution of computer science in the emerging parallel computing era. While a multi-processor is seen as a construct starting from Turing's model, a many-processor is better explained in a different conceptual environment. We propose Kleene's computational model as a theoretical framework for the many-processor paradigm. The many-processor approach is exemplified by the architecture of* ***Connex Core*** *part of the* ***BA1024*** *chip, a fully programmable SoC delivered by* BrightScale, Inc. *for the HDTV market.*

Key words: embedded computation, parallel architectures, many-processors, Kleene's computational model, HDTV.

1 Introduction

This paper provides a way to understand how we can conceive a computation involving n actors, with n **any big** number, that is a **many-processor** architecture. We exemplify the distinction between *multi* and *many* using two machines: *Intel Core 2*, and *Connex Core*.

Intel Core 2: a typical multi-processor In ***Intel Core 2*** each CPU independently implements various forms of *transparent parallelism*, such as: superscalar execution (data-parallelism), pipelining (time-parallelism), speculative executions (supporting time-parallelism). Both machines support independently multi-threaded executions. A system with 2 or more cores is able to effectively execute more threads concurrently. The multi-processor implemented as a multi-core machine controls the parallel execution mainly at the program level. Few additional physical resources are needed to support the multi-threaded parallelism actually performed on multi-processors. For this reason the computational model used to support mono-processors works very well for the multi-processor paradigm.

Connex Core: a typical many-processor Figure 1 represents the block diagram of the ***BA1024*** chip with emphasis on *Connex Core*, a many-processor engine.

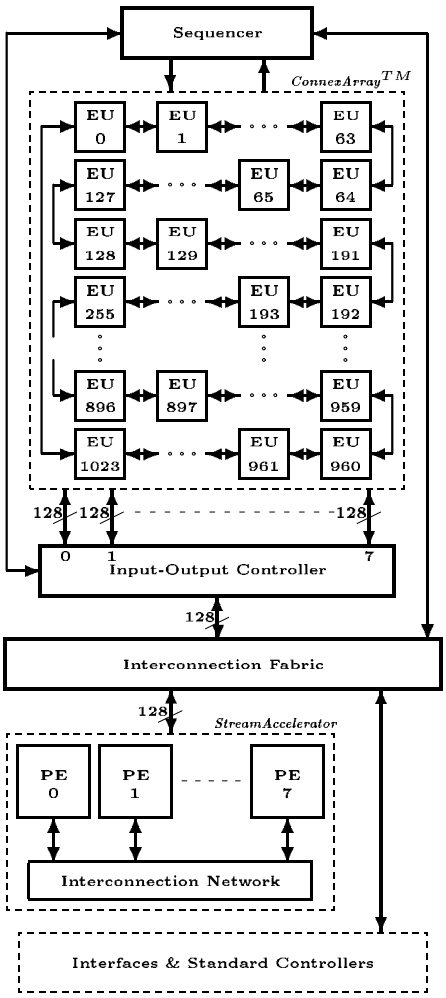

Figure 1: **The *BA1024* chip.** The many-processor aspects are: *ConnexArray*^{TM} and *StreamAccelerator*.

One thousand *execution units*, EU, execute conditionally (according to their internal states) the instruction issued in each clock cycle by the *Sequencer*. All data intensive computations are done by this linear array called *ConnexArray*^{TM} (CA). Eight *processing el-*

ements, PE, are devoted to accelerate pure sequential computations. The array, called *stream accelerator*, SA, is dynamically reconfigured by the interconnection network to solve problems like *arithmetic coding*. The *Interfaces & Standard Controllers* block contains general purpose interfaces (DDR, PCI, ...), specific interfaces (audio and video), and general purpose controllers (MIPS machines). (More details in section 4.)

The gap between "multi" and "many" The difference between "multi" and "many" is rather *qualitative* than *quantitative*. The multi-threaded execution takes few of almost *independent* processes, while the data-parallel or time-parallel computation refers to n *interdependent* ones.

There are well established techniques to deal with the multi-threaded approach, all developed for mono-cores, but equally useful for multi-cores.

Because we are in the infancy of the many-core approach, the programming techniques are far to be established. One main reason for this weakness is the theoretical framework which hosts the research in this too new domain.

Another difference evident in this early stage of the split between "multi" and "many" refers to ***intensity vs. complexity***. Multi-processors deal with complex computations, while many-processors are comfortable with the intense ones.

Thus, there are a lot of reasons to have a specific theoretical background for the many-processor paradigm. The envisaged computational model must have an explicit reference to n, the degree of parallelism involved in computation.

2 Two Computational Models

All computational models proposed starting from 1936 are equivalent. However, they are not equal in providing enough "expressive" backgrounds for different styles of computation. Regarding the "multi-many" distinction we mention here two models: Turing's and Kleene's.

Turing's model The architecture of the standard mono-processor computer derives almost directly from the structure associated with Turing's computational model [10]. The classic representation of the Turing Machine (TM) can be reformulated in terms of real circuits, where the infinite tape is an infinite random access memory, the access head is an up-down counter with a two-directional data connection, and the control section is a finite automaton (FA) (see Figure 2).

Optimizing the structure of the FA and the Up-Down Counter for an Universal TM (UTM) we obtain what we now call the *processor structure*, used to work on data stored in a memory, according to the program stored in the same memory. Thus, the sequential computing machines are supported directly by Turing's model. There is also a way to expand this model to the multi-processor paradigm using the multi-threaded programming style.

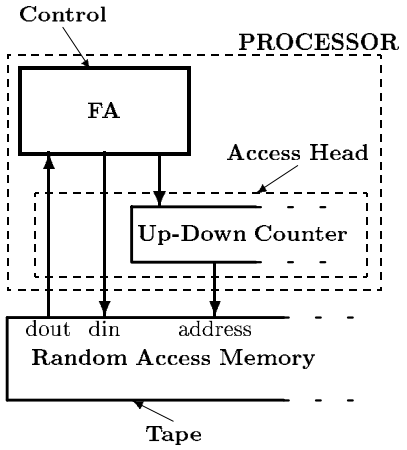

Figure 2: **Turing Machine with circuit components.** The infinite tape is implemented as an infinite memory, addressed by an infinite up-down counter, which performs in each cycle a *read-modify-write* operation under the control of a finite automaton.

Turing's model cannot be used "directly" to found many-processors. We prefer to start with a model which "naturally" fits the description of a machine with thousands components: Kleene's computational model [2].

Partial recursive functions In the year Turing published his paper, Kleene published the *partial recursive functions* model. He defines computation using basic functions (zero, increment, projection), and rules (composition, primitive recursiveness, and minimalization).

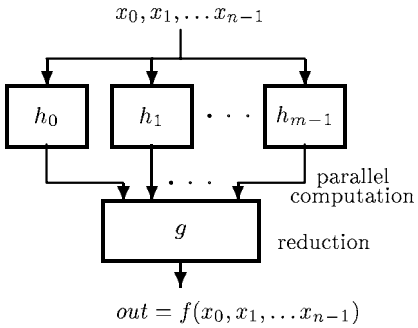

Figure 3: **The structure associated to composition.**

For implementing the basic functions there are small, simple and fast circuits. The *increment* circuit is an optimal one (polynomial size and poly-log time). The *multiplexor* used for the projection function is also optimal. The composition rule is actually the main rule. In Figure 3 is shown the structure associated with the composition rule: $f(x_0, \ldots x_{n-1}) = g(h_0(x_0, \ldots x_{n-1}), \ldots h_{m-1}(x_0, \ldots x_{n-1}))$.

Both, the first level of computing m functions $(h_0, \ldots h_{m-1})$, and the second of the reduction of m variables to one variable, are parallel processes. The only restriction in this process is that the reduction can't start before the end of computation on the first level. This is the *inherent sequentiality* of composition.

Two different starting points Turing's model is about one sequential process. The composition rule in Kleene's model is about m processes evolving in parallel with the reduction operation. In Kleene's model there are considered at least two kinds of parallel processes. The h_i functions are independent, and g can be independent if it works, in pipeline mode, on different input data $x_0, \ldots x_{n-1}$. An UTM can be seen as a sequential composition of TMs functions h_i and g. But the parallel aspects of computation are lost.

If an actual structure is directly associated with the composition rule, then results a **many-structure** as a consistent starting point. The elementary components of this structure are execution circuits or programmable machines. *We take off from the beginning in Kleene's model with a parallel configuration!* The number n, related with the degree of parallelism, is from start involved in the theoretical approach. It is about an n-guided development environment.

3 From Partial Recursiveness to Many-Processors

The general form of composition has particular, simplified forms able to express the other rules.

Data-parallel composition (Figure 4a) computes $n = m$ functions, each applied to a component of the vector $[x_0, \ldots x_{n-1}]$, and generates the vector $[h_0(x_0), \ldots h_{n-1}(x_{n-1})]$. Here g is the identity function.

There is a particular, real structure associated to the data-parallel composition going back to the organization of the BA1024 chip (see Figure 1). If $h_0 = h_1 = \ldots = h_{n-1} = h$, then CA from Connex Core is a system executing data-parallel composition for the same function h, whose execution is controlled by *Sequencer*.

Serial composition (see Figure 4b) is defined for multiple applications of the composition with $n = 1$. Results a pipe of p functions, $k_0, \ldots k_{p-1}$, that computes $f(x) = k_{p-1}(k_{p-2}(\ldots k_0(x)) \ldots)$. In each cycle a new value from the input stream, $< x_0, \ldots x_{s-1} >$, is inserted. After a **latency** of p cycles the result stream $< f(x_0), \ldots f(x_{s-1}) >$ is extracted. The circuit computes in $s+p$ cycles p values of the function f. This kind of parallel computation is very efficient when $s >> p$, where s is the length of the input stream and p the number of functions composed in order to compute $f(x)$.

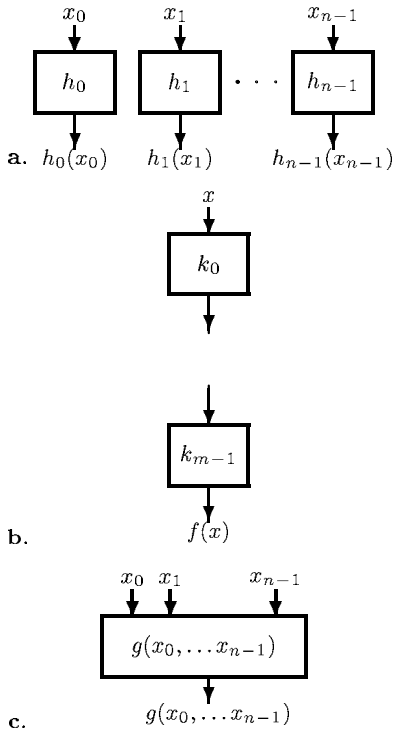

Figure 4: **The basic forms of composition. a.** Data parallel composition. **b.** Serial composition, working in parallel on successive input data. **d.** Reduction composition.

An example of serial composition occurs in Connex Core of the BA1024 chip. Connecting serially the 8 processors PE, from *StreamAccelerator*, results a pipe of $p = 8$ functions able to execute a pure sequential computation. Each processor has its own program defining one stage of the pipe.

Reduction composition (see Figure 4c) occurs when $h_i(x_i) = x_i$ for any i. The input vector $[x_0, \ldots x_{n-1}]$ is reduced to a scalar. This function is performed in poly-log time and the structure has a linear size. In CA of the BA1024 chip the data is extracted for *Sequencer* using a reduction tree.

3.1 Applying Primitive Recursion

The primitive recursion computes $f(x, y)$ using $f(x, y) = g(x, y, f(x, y-1))$, where $f(x, 0) = h(x)$. The

theoretical infinite circuit which computes f is in Figure 5. It has an infinite pipe of identical circuits (excepting the first), and a reduction network, distributed along the pipe, used to select the result.

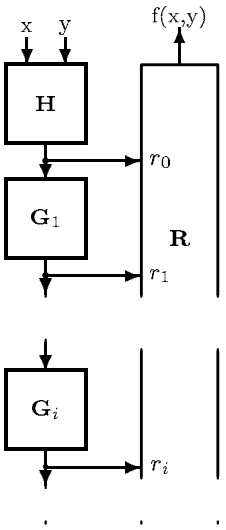

Figure 5: **The primitive recursive circuit.** An infinite pipe of machines, $H, G_1, G_2, \ldots, G_i, \ldots$ and an infinite reduction circuit, R, are used to apply the primitive recursive rule.

The function performed by the circuit H is $H(x,y) = \{x, y, f(x,0), (y == 0)\}$. Module H sends to the first input of the reduction network $r_0 = \{f(x,0), (y == 0)\}$, a pair $\{scalar, predicate\}$, and to the next level in pipe sends $\{x, y, f(x,0)\}$. The module G_i computes $G_i(x, y, f(x, i-1)) = \{x, y, f(x,i), (y == i)\}$. The module G_i sends $r_i = \{f(x,i), (y == i)\}$ to the corresponding reduction network input, and to the next stage $\{x, y, f(x,i)\}$.

The function of R is defined on the theoretically infinite vectors of pairs $\{scalar, predicate\}$, and returns **for sure** (the function $f(x,y)$ is total) the scalar which is paired by the predicate having the value 1.

3.2 Applying Minimalization

The minimalization rule computes the function $f(x)$ to the value of the minimal y for which $g(x,y) = 0$. See the associated circuit in Figure 6, where the function performed by each module G_i returns a pair $\{scalar, predicate\}$: $G_i(x) = \{i, (g(x,i) == 0)\} = r_i$. The reduction function R selects, from the input vector $\{r_0, \ldots r_i, \ldots\} = \{\{0, P(0)\}, \{1, P(1)\}, \ldots \{i, P(i)\}, \ldots\}$, the scalar from the first pair for which the predicate $P(i) = 1$, **if any** ($f(x)$ is a partial function) and shows it at the output accompanied by another predicate used to validate the result. The output is $\{scalar, predicate\}$, where the predicate is 1 if the output is valid. The predicate is needed because by minimalization *partial* functions are computed.

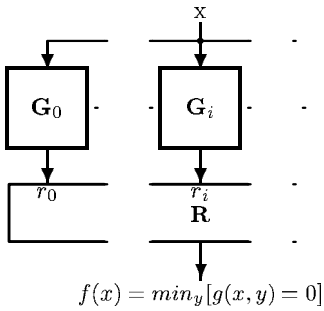

Figure 6: **Minimalization circuit.** Each circuit G_i computes the function $g(x,i)$, and the reduction circuit selects the minimal i for which $g(x,i) = 0$.

The data parallel composition performs a **speculative** computation. It computes g for x and "all" the values of y starting with 0. The reduction selects to the output, **if any**, the **first** result. A special feature is provided for the reduction network: the **function first**, $FRT(< B >) = < 0, \ldots 0, 1$ *or* $0, \ldots 0 >$, where $< B >$ is any n-bit binary stream. The function FRT indicates, by a single (if any) 1 in an n-bit binary stream, the position of the **first** 1, **if any**, in the input binary stream $< B >$.

In BA1024 chip the function FRT is used by CA, in conjunction with the reduction function, to select the scalar sent back to *Sequencer*.

3.3 Many-Processor Architecture

The model presented above can be translated into an actual **universal machine** in a few ways. The main problems to be considered are: (1) how to allow the execution of a sequence of many rules, (2) how to reuse as much as possible the hardware resources, i.e., how to make a programmable system, (3) what are the most efficient programming styles for the many-processor architecture. Because the domain of many-processor architecture is yet in its infancy, we are not ready to answer very well all the previous questions. Therefore, only some suggestions and a case study are provided.

For a many-processor architecture two data packages must be added to allow the description of the basic features. It is about **vectors**, $[X] = [x_0, x_1, \ldots x_{n-1}]$, and **streams**, $< X > = < x_0, x_1, \ldots x_{n-1} >$, where x_i are scalars or Booleans.

Data parallel composition means to receive vectors and to generate vectors. Serial composition supposes inserting streams and extracting streams. Reduction

composition receives vectors and outputs scalars or Booleans.

Data parallel composition can be used efficiently in vector operations and speculative computation. Serial composition is imposed by sequential algorithms. Sometimes, serial composition asks for speculative computation. Reduction composition is involved with both, data parallel and serial composition. Therefore, it seems to be useful to define a computing system having the possibility to combine in a very flexible way all kinds of compositions.

3.3.1 A Data Parallel Many-Processor

There are domains characterized by intensive data parallel computation and where the weight of the inherent serial computations is very small. Then, a minimal system is implemented by three parallel processing resources: (1) the data parallel array, (2) the loop closed over the data parallel array through a two-directional FRT network, and (3) the reduction tree. To each h_i an **execution unit** (EU) is associated (it contains only ALU & registers & data memory). The reduction composition is a tree structure which performs simple functions.

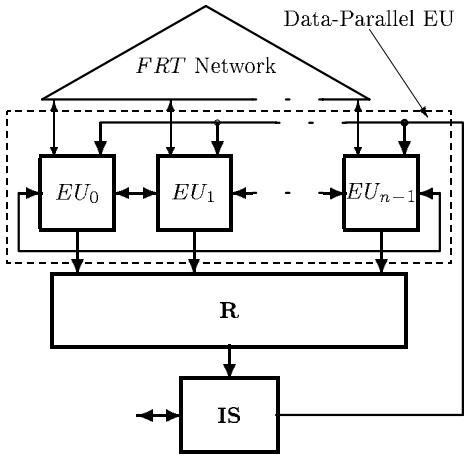

Figure 7: **Data parallel many-processor architecture.** Many execution units (EU) are interconnected in a two-direction ring. The reduction tree (R) closes the loop back to *Sequencer* (IS). The FRT network closes a Boolean loop over the Data-Parallel EU.

In order to link vector operations, in each EU_i is built a local scalar memory. All the components i of the vectors involved in computation are processed in EU_i. Another additional feature is the constant distance communication between EUs. The execution model is:

- in each clock cycle an **instruction sequencer** (IS) broadcasts one instruction to be executed by each EU (see Figure 7)
- each EU executes the instruction according to its internal state, for example:

```
where (bool_vect_q == 1)
    vector_n = f(vector_m, vector_p);
  elsewere
    vector_n = g(vector_m, vector_p);
```

- the instruction operates on data stored in each EU and, sometimes, on data stored in a small neighborhood (usually EU_{i-1} and EU_{i+1})
- the sequence of instructions evolves according to the IS internal state and according to the data provided by the reduction tree.

The minimal structure of a data parallel architecture is in Figure 7. A very important feature, imposed for solving the partial recursiveness, is the small and simple loop, closed over the entire n-EU array through the FRT network [5]. It classifies the EUs, depending on the actual values of a selected Boolean vector, as follows: (1) the *first EU* (FEU) with the selected Boolean on 1, (2) the EUs positioned before FEU, and (3) the EUs following FUE.

An example of Data Parallel Many-Processor is the module $ConnexArray^{TM}$ in BA1024 (Figure 1).

3.3.2 An Integral Many-Processor

There are applications where the data intensive and the intensive inherent sequential computations are balanced and interleaved. Then, both, data parallel composition and serial composition must be supported by the same hardware [4] [9] [3]. The resulting structure is topologically similar with the previous, with the important difference that the EUs are substituted by **processing units** (PU). An EU executes only the instruction received from IS, while PU executes also its own locally stored program.

If conditioned pipeline executions are needed, then the interconnection neighborhood must be expanded to allow speculative evaluations. If no more than m conditions are involved at any stage in the pipelined execution, then each PU_i must be able to select data from the previous 2^m PUs. Thus, an *Asymmetric Interconnection Fabric*, AIF, is added. Figure 8 shows the resulting *Integral-Parallel PU*, where each PU is allowed to select as left input data from the previous 8 PUs ($m = 3$). The AIF's outputs, connected to the left inputs of each PU are selected according to the following expression:

$$in_{i+1} = S(s_i, out_{i-0}, out_{i-1}, \ldots) = out_{i-s_i}$$

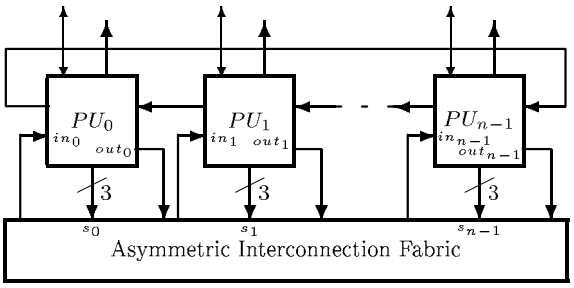

Figure 8: **Integral-Parallel PU.** The left connection of each PU is selected, by the 3-bit code s_i, one out of the 8 previous PUs.

The selection code, s_i is the condition code computed by each PU. The condition is used to select the result of the speculation performed by the previous up to 8 PUs.

Programming the serial composition part of this machine is done defining the **vector of functions**, $\mathbf{F} = [f_0, \ldots f_{p-1}]$, where f_i represents the local program executed by PE_i on data received from the previous PEs. For no conditional operation each PU_i receives its input data from PU_{i-1}. If the m-condition execution is performed, then PU_i selects its input from the output of $PU_{i-2^m}, \ldots PU_{i-1}$ ad-hoc involved in a speculative computation as a data parallel array of 2^m PUs.

3.3.3 Segregated Many-Processors

If the data intensive computations and the inherent sequential intensive computations are grouped in clearly distinct stages of the application, then they deserve specialized hardware units. Thus, advanced optimizations are allowed, because the number of EUs and of PUs are stated independently.

The structure consists in two subsystems: (1) a Data Parallel Many-Processor Architecture (see Figure 7), connected with (2) a Data-Parallel PU (see Figure 8) which has the inter-connectivity limited by the maximum value of m (the maximum number of conditions required by the programs f_i).

An actual implementation of a Segregated Many-Processor Parallel Architecture is done in BA1024 chip, where the tandem *ConnexArray*^{TM} *& StreamAccelerator* represents the Connex Core.

4 Case Study: *Connex Core*

This section provides an exemplification of what is a many-processor, using an actual segregated many-processor machine. The figures provided will allow us better understand the differences between multi- and many-processors. It is a sort of quantitative proof that starting from Turing's model or from Kleene's model makes a lot of difference.

The ***BA1024*** chip is a SoC designed by *BrightScale, Inc.* to implement a platform for the HDTV market. It is a Segregated Many-Processor. The chip, implemented in $130nm$ standard process, works at 200MHz. It contains the audio and video interfaces for two HD channels, the multi-threaded section of 4 MIPS processors, the DDR interface ($3.2GB/sec$), an 128-bit interconnection fabric, and the intensive parallel machine containing:

Sequencer: a 32-bit controller with stack architecture, used to issue in each clock cycle an instruction to be broadcasted into the data-parallel processing array

Input-Output Controller: another 32-bit controller with a stack architecture which communicates with the previous using interrupts; it is used to manage the transfers between the array and the rest of the system

ConnexArray^{TM} (CA) a *linear* data-parallel array of 1024 16-bit EUs, where each EU has: (1) 16-bit integer ALU (support functions for speeding the multiplication), (2) Boolean machine for 8 boolean variables, (3) a local data memory for 256 16-bit words. It receives in each cycle an instruction, issued by *Sequencer*, which is executed according to the value of the selected local Boolean

global loop: closed over CA, used mainly to find the first EU with the selected predicate 1

reduction: tree to extract data and some critical parameters from CA

I/OPlan: a *two-dimension* array which transfers data between CA and the rest of the chip under the control of *Input-Output Controller*; the I/O process is transparent to the processing performed by CA

StreamAccelerator: a serial composition engine of 8 16-bit PUs, each having its own program memory with $m = 4$.

One of the main design decisions was to keep the CA's interconnection network as simple as possible, while ***I/OPlan*** was designed to perform complex rearrangement of data during the transfer. This compromise works in the video processing domain.

Another important design choice was to perform in each EU only simple functions (no multiplications, no floats). Evaluating the frequency of the multiplication

operation we decided to add only a small set of simple support functions.

General performances of the Connex Core:

- 200 GOPS (OP means 16-bit simple operations, no multiplication, no FP)
- 3.2 GB/sec external bandwidth, 400 GB/sec internal bandwidth
- > 60 GOPS/W and > 2 GOPS/mm^2

Some video specific performances:

- DCT: 0.15 clockCycles/pixel (on 8×8 arrays)
- SAD: 0.0025 clockCycle/pixel
- decoding H.265 dual HD stream: 85% of the computational power.

The previous figures resulted running on ***BA1024*** programs written in *Connex Programming Language*, developed by BrightScale.

Because the HDTV domain requests data intensive computations, CA is used in its full power. The current implementation is I/O bounded. SA is needed to accelerate the pure sequential part of the codec (mainly for the H.264 standard).

5 Concluding Remarks

Partial recursive functions are computed using only various forms of composition.

Multi-processing for complex computation and many-processing for intense computation While the multi-processor solution remains to optimize complex applications, the many-processor solution applies where the intensity prevails complexity.

Complex versus intensive by numbers For multi-processors $(0.08\,GIPS + 0.08\,GFLOPS)/Watt$ and $(0.02\,GIPS + 0.02\,GFLOPS)/mm^2$ are usual performances. For the intensive many-processor machine ***BA1024*** $60\,GOPS/Watt$ or $(30\,GOPS + 0.6\,GFLOPS)/Watt$ and $2\,GOPS/mm^2$ are obtained. The two classes of architecture are perfectly differentiated by **two order of magnitude**. Turing's model manages *complexity*, while Kleene's model supports *intensity*.

Thus, segregating complex multi-processor architectures from intensive many-processor architectures is the best solution for optimizing both *price(area) versus performance* and *power versus performance*. Let be the slogan:

High Performance Architecture = mono/multi-processor + many-processor

Acknowledgments The authors got a lot of support from the main technical contributors to the development of the *ConnexArray*^{TM} technology, the *BA1024* chip, the associated language, and its first application: Emanuele Altieri, Frank Ho, Bogdan Mîţu, Marius Stoian, Dominique Thiebaut, Dan Tomescu, Tom Thomson.

References

[1] K. Asanovic, et. all.: *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report No. UCB/EECS-2006-183.

[2] S. Kleene: "General Recursive Functions of Natural Numbers", in *Math. Ann.*, 1936.

[3] M. Malita, G. Ştefan, D. Thiebaut: "Not Multi, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation" in *ACM SIGARCH Comp. Arch. News*, Vol. 35, 5, Dec. 2007.

[4] B. Mîţu: private communication, 2005.

[5] G. Ştefan, D. Thiebaut: "Memory Engine for the Inspection and Manipulation of Data", *United States Patent 6,760,821*, July 6, 2004; Filed: Aug. 10, 2001.

[6] G. Ştefan: "The CA1024: A Massively Parallel Processor for Cost-Effective HDTV", in *Spring Processor Forum: Power-Efficient Design*, May 15-17, San Jose, CA 2006.

[7] G. Ştefan, et all.: "The CA1024: A Fully Programable System-On-Chip for Cost-Effective HDTV Media Processing", in *Hot Chips: A Symposium on High Performance Chips*, Stanford Univ., August, 2006.

[8] G. Ştefan: "The CA1024: SoC with Integral Parallel Architecture for HDTV Processing", in *4th International System-on-Chip (SoC) Conference and Exhibit*, November, Newport Beach, CA, 2006.

[9] D. Thiebaut, M. Maliţa: "Pipelining the Connex Array," *BARC07*, Boston, Jan. 2007.

[10] A. Turing: "On computable Numbers with an Application to the Eintscheidungsproblem", in *Proc. London Mathematical Society*, 1936, 1937.