**ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY**
Volume 4, Number 4, 2021, 366-383

# Pseudo-Reconfigurable Computing

Gheorghe M. ŞTEFAN

Politehnica University of Bucharest

Email: gheorghe.stefan@upb.ro

**Abstract.** The pseudo-reconfigurable computing we propose is an implementation of heterogeneous computing in the form of a compromise between the realization of accelerators as circuits through reconfiguration techniques using FPGAs and the realization of accelerators as parallel computing structures made in ASIC technology. The proposed solution is implementable in FPGA in the form of a programmable structure that is parameterizable and configurable. The programmable accelerator is a cellular parallel engine with a structure and architecture that efficiently covers most of the parallel computing patterns. The structural and architectural aspects of the proposed system are introduced based on Stephen Kleene's Partial Recursive Function Model, and evaluated using: (1) Functional Forms introduced by John Backus, (2) 'dwarfs" listed in the Berkeley's View of Parallel Landscape, and last but not least (3) patterns already imposed in the practice of parallel computing.

## 1. Introduction

Reconfigurable computing (RC) is a new solution for optimizing time and energy which combines the flexibility of software with the high performance of hardware by supporting the processing with very flexible high speed computing fabrics like field-programmable gate arrays (FPGAs). The main difference when compared to using ordinary microprocessors is the ability to make substantial changes to the data-path which is deflected, when its flow is very high, toward an accelerator. On the other hand, the main difference from custom hardware, i.e. application-specific integrated circuits (ASICs) is the possibility to adapt the hardware of the accelerator during runtime by "loading" a new circuit on the reconfigurable fabric.

The main challenge of RC is the translation of portions of code written in a high-level language (C, C ++, Java, Python, ...) that must be accelerated in a hardware description language (HDL) such as Verilog , VHDL, …. This translation can be done in two ways. By rewriting the code in a HDL by a digital system designer or automatically by compilation. Unfortunately, the number of high-performance designers in the digital field is very small compared to that of programmers, and compilers have (at least for now) performance dependent on the target function, but usually low, sometimes very low.

In order to avoid translating by rewriting or compiling the code submitted to acceleration, a solution would be to use, for the sections of the program to be accelerated, a predefined kernel

library executed in the accelerator part of the heterogenous system by a tunable & programmable parallel engine. The programmer, or a compiler, could use a kernel of functions specially tuned (in the future maybe *autotuned*) for the field in which he writes his program or even for the current program. This kernel would be accelerated by the FPGA and it will be *loaded* when the program to be accelerated is launched. Thus, loading will not involve instantiating some circuits, but updating a programmable structure together with the associated programs that implement the functions necessary to accelerate the program to be run.

Therefore, instead of RC, we will define what we will call from now on Pseudo-Reconfigurable Commputing (pRC) which will use as accelerator a configurable and parameterizable programmable structure, instantiated only once at the start of program run, instead of circuits that are instantiated during running the program whenever needed.

This paper is only a theoretical introduction to the concept of pRC. The proposed structure and architecture are based on a mathematical model and justified starting from theoretical approaches that emerge from the practice of programming and development of applications made on mono- and multi-core systems.

The next section introduces the concept of pRC. Third section provides the theoretical foundation for the parallel version of an accelerator whose abstract model is defined starting from the mathematical model of Stephen Kleene. Fourth section shows how Functional Forms introduced by John Backus fit with a Map-Scan-Reduce architecture proposed in the previous section. Next section investigate how the proposed accelerator performs related with main parallel programming patterns highlighted in [19]. Sixth section looks over how the proposed organization and architecture of the parallel proposed accelerator support application domains (“dwarfs”) emphasized in *A View from Berkeley*, the seminal Berkeley’s report. The seventh section concludes our paper. We added an appendix containing some technicalities.

## 2. pRC vs. RC

The increasingly clear shaping of the difference between complex and intense computation has led to different forms of ***heterogeneous computation***, which segregates into a computing system two subsystems: the HOST, responsible for the complex part of computation, and an ACCELERATOR, to which tasks related to intense computing are transferred. One of the most promoted form is that of RC systems based on the spectacular development of FPGA technology. The ACCELERATOR is implemented in an FPGA so that it can be configured, statically or dynamically, as a circuit that accelerates a specific calculation.

As it is known, any computable function can be calculated with a digital circuit. Moreover, for almost any function the circuit has better performance – in terms of execution time and energy consumed – compared to a solution offered by a programmable system. Recent research and commercial High Level Synthesis (HLS) tools accept synthesizable subsets of high level languages such as C/C++/SystemC/MATLAB. The code written in one of these languages is analyzed, architecturally constrained, and scheduled to *trans-compile* into a RTL design in a HDL, such as Verilog or VHDL, which is in turn synthesized to the gate level using logic synthesis tools.

The trans-compilation process is part of the compilation of the program that will run on the heterogeneous system. In the process of compiling a program whose execution must be accelerated, it is generated, besides the code executed by HOST, the bit level image of the circuit or circuits that will be used in the acceleration of the program. The circuits will be ”loaded” into

the FPGA depending on when the HOST program calls them. There are two problems that can affect the efficiency of RC:

1. the overhead introduced by the circuit update in the FPGA during the program execution
2. the efficiency with which the trans-compilation process takes place.

Both issues are in the attention of RC system developers, and we will certainly see significant improvements in the more or less distant future. However, alternative solutions, such as pRC we propose, may be of interest.

Indeed, FPGA designers offer increasingly efficient solutions for loading bitstream files. The partial modification of the implemented circuit is more and more supported by the FPGA manufacturers.

The second problem, that of trans-compilation is a more difficult one. It offers solutions with an efficiency that still depends very much on the problem. Too often a digital circuit designer offers a much better solution. But, unfortunately, competent digital circuit designers are very few compared to programmers.

**Example 1.** *Using the Vivado HLS environment offered by Xilinx, two applications were solved. The first calculates the product of two* $5 \times 5$ *matrices, and the second implements the calculation made by a FIR filter. The C-language cross-compilation in Verilog provided for the first problem a solution that used* $20\times$ *more resources than the one obtained by the direct description in Verilog, while for the second problem it offered a solution almost identical to the written one directly in Verilog.*

*It is clear that there is a problem dependence on the effectiveness of HLS use.* $\diamond$

For both problems we propose the following solution: an ACCELERATOR implemented as a ***configurable and parameterizable parallel programmable computation engine***. Instead of synthesizing in FPGA each time a different circuit, as in a true reconfigurable approach, we will customize for each program/application a programmable structure, belonging to a generic class, and a set of programs associated with the functions that we have to accelerate. That is, instead of resorting to the *natural parallelism* of circuits, we consider the *artificial parallelism* of many-cellular structures. Designing a digital circuit will be substituted by the following mechanism of instantiating the ACCELERATOR part of a pRC:

1. from a kernel of already programmed functions, the functions we want to use are selected and parameterized
2. if the desired functions do not exist we will define and implement them by adding new programs to the kernel
3. the generic structure of the parallel accelerator is instantiated according to the resources requested by the previously considered programs
4. the hardware structure and the associated programs are loaded in ACCELERATOR implemented in FPGA only once at the beginning of the program execution

The physical structure of the accelerator is designed only once by a specialist in the field of digital circuits. Programs are made, as the generic accelerator is used, by developers working in low-level languages (often assembler), as is the case with the development of efficient function

libraries. The good news is: we no longer need high-performance hardware designers and time to market is reduced. The bad news is: the performance of pRC systems is sometimes lower than those obtained in the RC approach.

At an advanced stage, the ACCELERATOR will be able to be used as a ***hardware kernel of a function library*** on the basis of which it can be developed different function libraries to be seen by users as a hardware library of functions much more efficient than current function libraries developed purely software (even if in some cases the code is written in assembler).

There are still fundamental problems to be solved. What does this programmable parallel accelerator look like? What are the parameters and characteristics that can be specified at the time of synthesis? The next section will show how the organization and architecture of this system can be obtained in a well-founded theoretical way. In the sixth section we will discus about the second question.

# 3. MapScanReduce Accelerator

The year 1936 was a very rich one in theoretical achievements in the field of computer science. It is not easy to answer the question: how did four fundamental works for computer science appear independently in the same year? Martin Davis republished them [7] along with the work that seems to have triggered them: the seminal text of Kurt Gödel [11]. These are the works published by Alonzo Church, Stephen Kleene, Emil Post and Alan Turing in which mathematical models of computation are presented. We focus in the following on Kleene's work [15] on the definition of ***partial recursive functions***.

Only one of the rules that allows the calculation of a partially recursive function is independent (see Appendix). It is about the composition rule:

$$f(X) = g(h_0(X), h_1(X), \ldots, h_{p-1}(X))$$

Because the $p$ functions $h_i(X)$ are independent they can be calculated in parallel by distinct physical structures, and the function $g$ can be calculated only after all the $p$ functions $h_i(X)$ have been calculated, we are dealing with two parallel processes : (1) the *synchronous parallelism* that allows the calculation of the $h_i(X)$ functions and (2) the *diachronic parallelism* (pipeline parallelism) that allows the parallel calculation, for two successive values of $X$, carried out on the two levels, that of the synchronous parallelism and that of the reduction type $g$ function (see Fig. 3 in Appendix).

Starting from the previous observation and from the need to use finite versions associated to the theoretical constructs from Fig. 5 and Fig. 6 (see Appendix), we can propose the real physically achievable structure from Fig. 1 as an ***elementary abstract model for parallel computing***. The loop closed through CONTROLLER and the memories, *mem*, distributed along the $p$ cells in the MAP array allow the reduction from a theoretically infinite structure to an actual finite $p$-cell organization.

The infinity of the theoretical structures represented in Fig. 5 and Fig. 6 refers to the fact that the index $i$ must be able to take a sufficiently high value for the real problems to be solved. If we want to design an actual structure, then we have to consider a technologically acceptable maximum value for index $i$, be it $p-1$. For problems involving a maximum index greater than $p-1$ we can use:

- additional storage items for data associated with the index greater than $p$ (see the local memory *mem* in each cell)

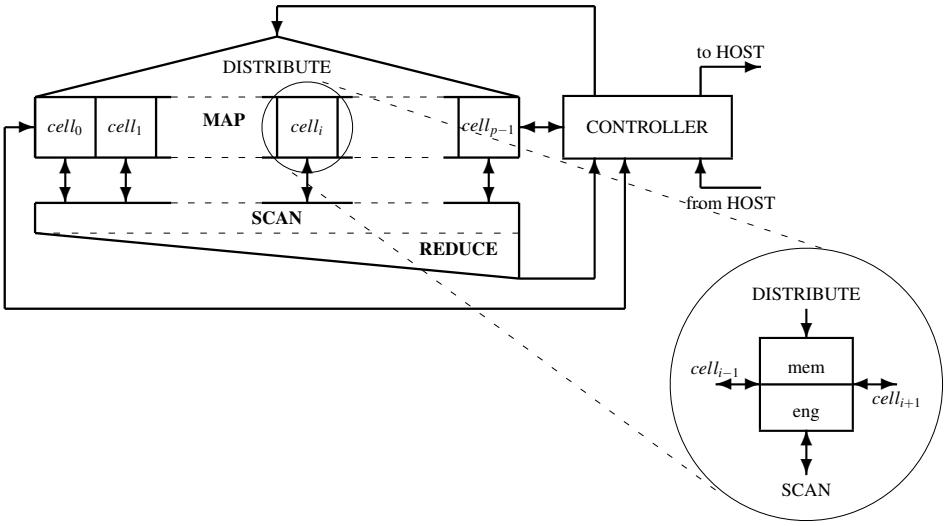


**Fig. 1. Elementary abstract model for parallel computing.** Each cell contains a local memory, *mem*, and an execution unit, *eng*. DISTRIBUTE is a *log*-depth network used to propagate at the level of cells the command issued in each cycle by CONTROLLER. The *log*-depth network SCAN performs vector-to-vector functions, while the *log*-depth network REDUCE, part of SCAN, performs vector-to-scalar functions.

- a sequencing mechanism that controls the extended calculation for indices that exceed the value of $p$ several times (see the block CONTROLLER, able to issue a sequence of operations to be performed in the MAP array).

Thus, a problem, defined for data distributed in $m \times p$ $P_i$ type cells (see Fig. 5 and Fig. 6) working without the supervision of a controller, is solved with a structure having only $p$ cells, endowed with sufficient local memory, operating under the control of a sequencer.

If each cell-memory *mem* stores $m$ scalars, then the entire MAP section can be seen as storing a matrix of $m$ lines and $p$ columns, $M_{mp}$, organized in horizontal vectors – $V_i = [s_{i0}s_{i1}\ldots s_{i(p-1)}]$ for $i = 0,\ldots,(m-1)$ – and vertical vectors – $W_j = [s_{0j}s_{1j}\ldots s_{(m-1)j}]$ for $j = 0,\ldots,(p-1)$. Each component of a horizontal vector has two state: active and inactive. The function issued by CONTROLLER in each cycle affects only the active components of the vector involved in computation. The elementary abstract model for parallel computation supports the following types of operations:

**spatial selection functions** used to control the state of each cell which can be *activated* or *inactivated* according to conditions fulfilled by selected components of its vertical vector

**map functions** defined on the active components of horizontal vectors with value a horizontal vector whose components depend on scalars in the same cell or in strictly neighboring cells

**scan functions** defined on the active components of a horizontal vector with value a horizontal vector whose components may depend on one or more components of the argument vector

**reduction functions** defined on the active components of a horizontal vector with value a scalar sent back to CONTROLLER

**scalar functions** performed in CONTROLLER

**transfer functions** used to exchange data with the HOSTs memory in a transparent mode related to the computation in order to reduce the effect of the von Neumann Bottleneck.

Parallel computing remains an eminently sequential process, with the only difference that it operates directly on larger and more complex data structures. Instead of scalars, a parallel machine operates with tensors of different orders.

I named the abstract model just described as *elementary*, because a recursive hierarchical definition for parallelism is possible, but this topic is not the subject of this paper[1].

The abstract model we have defined can be used to design the structure of a programmable parallel accelerator. We will instantiate the *eng* block as a logical-arithmetic unit configured based on the DSP module, and the *mem* memory based on the BRAM modules, both available in current FPGAs. We will call this accelerator MapScanReduce Accelerator (MSRA).

In the following we will try to validate the model and its first implementation as MSRA (1) from the programmer's perspective, (2) in relation to the patterns that have already been imposed in parallel computing and (3) from the point of view of the one who develops applications.

# 4. Functional Forms on MSRA

John Backus's approach [5] for programming a computer is a functional one based on building up a program by combining hierarchical functions using an algebra of programs based on the principle of compositionality. For the intense computation, on many-core engines, we can benefit from the programming style Backus proposed: *Functional Programming Systems*. In its seminal paper, the actual structure of the computing engine is not mentioned, but his proposal is an efficient way to conceive the parallel programming on a our MSRA.

The functional forms proposed by Backus fit perfectly with the way the proposed MSRA works, thus offering an additional validation for the MSRA, besides to the purely theoretical one, this time from the point of view of programmability. Let's consider in turn the functional forms proposed by Backus:

***Apply to all*** : are executed in the MAP array, the same function is applied in each cell to the locally stored data.

***Construction*** : are executed in the MAP array, the same data is submitted in each cell to different functions.

***Insert*** : are reduction functions.

***Composition*** functional forms are performed using global shift and map functions.

***Condition*** functional forms correspond to the predicated execution based on spatial selection functions.

[1] Just as this text was being finalized, an article appeared in the latest issue of *Communications of the ACM* discussing the concept of ***Accelerator -Level Parallelism*** [14], a concept that is perfectly integrated into the hierarchical recursive model proposed in [22].

Thus, as the functions in a mono-core processor were defined for precisely dimensioned data structures (16-bits or 32-bit integers, or 32-bit or 64-bit floats, ...), for our accelerator, functions can be defined on precisely dimensioned data sequences: $p$-component vectors of $n$-bit scalars.

**Functional Forms in John Backus's FP System**

The functional forms help for expanding the logic and arithmetic functions to sequences (the first four forms), and for spatial control in the MAP array (the last one).

**Apply to all** implemented as data-parallel execution

$$\alpha f : \langle x_1, \ldots, x_i, \ldots, x_n \rangle) \rightarrow \langle f : x_1, \ldots f : x_i, \ldots, f : x_n \rangle$$

**Construction** implemented as speculative-parallel execution

$$[f_1, \ldots, f_i, \ldots, f_n] : x \rightarrow \langle f_1 : x, \ldots, f_i : x, \ldots, f_n : x \rangle$$

The *eng* part of the cells are in general processors, but in many cases the function is the same and depends on the parameter $i$.

**Insert** implemented as a reduction-parallel execution

$$/f : \langle x_1, \ldots, x_n \rangle \rightarrow f : \langle x_1, /f : \langle x_2, \ldots, x_n \rangle \rangle$$

Formally, this recursive definition looks like a sequential process, but it translates into a $log$-deep reduction-parallel operation.

**Composition** implemented as time-parallel (pipelined) execution

$$(f_q \circ f_{q-1} \circ \ldots \circ f_1) : x \rightarrow f_q : (f_{q-1} : (f_{q-2} : (\ldots : (f_1 : x) \ldots)))$$

The ***stream*** of objects, $|x_1, \ldots, x_n|$, is considered to be inserted, starting with $x_1$, in the left most cell of the MAP section.

**Condition** represents the predicated execution
$(\pi \rightarrow f; g) : \langle \langle c_1, \ldots, c_n \rangle, \langle x_1, \ldots, x_n \rangle \rangle \rightarrow$
$\rightarrow \langle (\pi : c_1)?(f : x_1) : (g : x_1), \ldots, (\pi : c_n)?(f : x_n) : (g : x_n) \rangle$
where $c_i$ and $x_i$ are objects, for $i = 1, \ldots n$. This functional form provides the ***spatial control*** in the MAP section using the predicate $\pi$.

The beauty of Backus's approach is that it does not refer to a particular organization (mono- multi- or many-core). Only the efficiency with which the functional forms are implemented is the one that differentiates the various solutions imposed by the technological limitations that have existed over time. Analyzing the functional forms we are tempted to consider parallel computing as the generic form of computing. Mono-core computing has been imposed by the technological constraints of the first decades of computer science, and multi-core computing seems to be a compromise that we hope will be overcome with widespread acceptance of the distinction between complex and intense computing.

# 5. Parallel Programming Patterns on MRSA

Since the 1960s, parallel computing has been practiced unsystematically and only vaguely theoretically substantiated [2]. However, its very intense practice manages to impose in the 2010s significant models for an efficient parallel computation [19]. We cannot neglect the efforts of a huge number of researchers who have investigated the possibilities of multi- and many-core computing. A number of patterns have been crystallized that can be used to assess the appropriateness of the MSRA approach we propose.

In [19] (see pag. 21) the authors make an overview of parallel patterns (reproduced as such in Fig. 2). The patterns they present "encode practices and distill experience" in a way that we can use them to assess the extent to which our MSRA model satisfies the conditions of a generic parallel structure. Some of the highlighted patterns are directly implementable on our model while others are implemented as a sequence of operations, as follows:

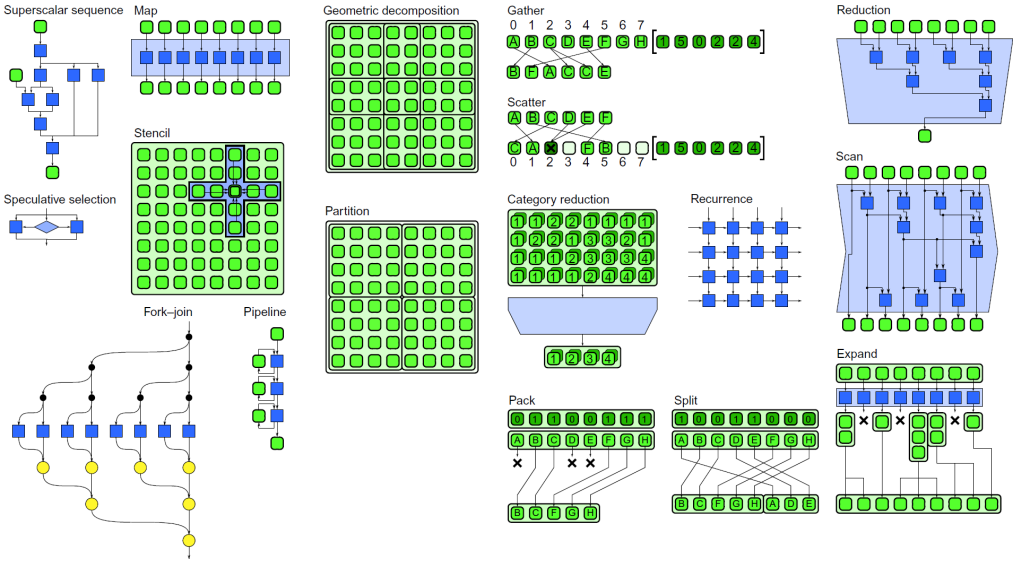


**Fig. 2. Overview of parallel patterns** [19].

**Map** pattern is directly implemented in the MAP array having also the advantage of the predicated execution due to the spatial selection

**Reduce** pattern is directly implemented in the REDUCE network with a *log*-cycle latency whose effect can sometimes be hidden by skilled programming

**Scan** pattern, exemplified in [19] with the prefix function, is directly implemented in the SCAN network which is designed on the frame of a permute network [1]

**Pipeline** pattern is implemented based on the local connections between the cells from MAP

[2] In the field of parallel computing, the steps started too early with the production of multi-processors, which was only then followed by the raising of architectural problems and much later research was initiated from the point of view of abstract models that were unfortunately confused with mathematical models (see details in [22]).

**Speculative selection** pattern is applicable at the level of each cell in the MAP array using the spatial selection function

**Fork-join** pattern uses, to fork the problem, the MAP array and, to join the result, the REDUCE network

**Stencil** pattern is executed on data loaded in the matrix $M_{mp}$; using the left-right connection in the MAP array, the content of each element of the matrix can be recomputed in parallel by a function whose arguments are the element and a finite neighborhood of it

**Geometric decomposition & Partition** is performed loading data as a matrix and then, using spatial selection and line addressing, these two patterns are applied

**Category reduction** can be implemented using an appropriate sequence of spatial selection, reduction and scan functions

**Gather & Scatter** are implementable using the permutation function offered by the SCAN module using an additional vector to specify the source for gather or the destination for scatter

**Pack & Split** are implementable using the permutation function offered by the SCAN module in conjunction with the spatial selection function

**Recurrence** is efficiently implemented using spatial selections and functions applied sequentially on horizontal vectors of the matrix $M_{mp}$

**Superscalar sequence** and **Expand** patterns obviously describe complex processes, which is why we are not worried that an MSRA solution will not prove to be effective in implementing them. As mentioned, in heterogeneous computing complex procedures are the responsibility of the HOST, the ACCELERATOR being involved only in performing intense computing.

# 6. Berkeley's *Dwarfs* on MSRA

In [4] the parallel landscape is described in term of computational patterns, initially called *dwarfs* in [3]. Each dwarf represents a domain defined by specific mathematical mechanisms used. In the following we will see how the 12 dwarfs (one is omitted in [4]) can be accelerated using the proposed MSRA.

**Dense Linear Algebra** dwarf operates on matrices loaded in $M_{mp}$, for $m \geq 3p$, involving mainly map and reduce operations; the effect of the latency introduced by $log$-depth REDUCE network is eliminated through a special stack mechanism distributed along the cells of the MAP linear array (see [17]); the acceleration provided belongs to $O(p)$.

**Sparse Linear Algebra** dwarf is based mainly on the spatial selection functions (to match the rows and columns at the intersection of which non-zeros appear), but involve also map and reduce functions [9].

**Spectral Methods** dwarf is supported mainly by an appropriate use of predicated execution in map and permute functions, for example in accelerating FFT computation; MSRA provides accelerations in $O(p)$ [2].

**N-Body Methods** dwarf refers to algorithm with execution time in $O(N^2)$ for which MSRA provides accelerations in $O(p)$; for example, molecular dynamics applications are executed with the power efficiency which exceeds 16× the x86 based solution while for the ASIC implementation of MSRA exceeds 15.5× the ASIC circuit available on the market [10].

**Structured Grids** dwarf is supported by the possibility to represent the structured grid as part of the matrix $M_{mp}$ to which the stencil pattern is applied; for example in video decoding accelerator [21].

**Unstructured Grids** dwarf is supported by the possibility to represent the unstructured grids as sparse matrices like in FIDAP simulations.

**Combinational Logic** dwarf has various application in various codings, such as AES, DES, SHA (e.g SHA256 is performed for mining cryptocurrencies with 3.7× less energy compared with GPUs), or in convolutional computation which is accelerated $O(p)$ times [17].

**Graph Traversal** dwarf benefits of the dwarf Dense Linear Algebra by using matrix representation for graphs; a 128-cell MRA (no scan operations are involved, thus in a pseudo-reconfigurable approach the SCAN section is not synthesized) accelerates 118× at 20× less energy a mono-core, and 3× at 26× less energy a 128-core GPU [8].

**Dynamic Programming** dwarf is best exemplified by Viterbi algorithm whose implementation benefits mainly from Dense Linear Algebra dwarf.

**Back-track and Branch+Bound** dwarf is typically represented by SAT problem which can be solved dividing hierarchically the problem in $p$ subproblems at each level pruning the ineffective branches using the SCAN and REDUCE network to select the promising branches; the solution is embarrassingly parallel, thus MRSA provides an acceleration in $O(p)$.

**Graphical Models** dwarf is typically represented by problems related to Hidden Markov Models; the acceleration provided by our accelerator is in $O(p)$, with the amendment that for big $n$ the size of local memory, *mem*, in each cell must be big [18].

**Finite State Machine** dwarf is involved, among others, in solving video compression applications (e.g H264 advance video codec [16]).

The acceleration offered by our solution is usually in $O(p)$, but always at least in $O(p/log\ p)$. Note that in the case of the most widely used dwarf, Dense Linear Algebra (see [4] Fig. 3), the acceleration is in $O(p)$.

None of the dwarfs listed above make full use of the structural and architectural facilities of the MSRA. Our pRC approach allows that when compiling the program, once the portions of code to be accelerated have been identified, the list of structural and architectural needs can be established. This list is used to establish the configuration and parameters of the accelerator that will be synthesized and implemented in the FPGA for each program running on the proposed pRC system. For example:

- $p$, the number of cells in the MAP array, is dimensioned according to the size of the data structure used when it is not limited to the size of the FPGA which we have
- $m$, the size of the local memory in each cell, is dimensioned according to the size of the data structure used and to the functions to be accelerated

- $n$, the size of the scalars, is set according to the application domain; from AI applications to scientific computing the range spans from 8 to 64 bits
- the type of arithmetic can be adapted to integer, fix point or floating point operations
- the type of spatial selection in the MAP array which can be with a single selection level or with a maximum of $p$ levels managed as a stack of selections
- the global functions – such as search, insert, delete, shifts, rotates – applied on horizontal vectors
- the depth of the execution stack in the *eng* part of cells according to the actual accelerated computation; the depth of the execution stack can be extended from one level (the most common case of accumulator-based processing) to dozens of levels for exotic applications
- SCAN functions, – such as permute, prefix, split, ... – are used only in a small number of applications, which is why they will not be synthesized unless absolutely necessary (for example for FFT, pooling, ...), especially because the size of the SCAN section is in $O(p \times log\ p)$
- REDUCE functions are frequently used – primarily for summing the active elements of a horizontal vector – although not entirely (maximum or minimum functions have a lower frequency, which is why they are sometimes not selected for synthesis)

The parameters & features listed above, and not only, can be used to tune the structure & architecture of the heterogeneous system. Moreover, it allows the development of an *autotuning* environment in our parallel setting.

# 7. Concluding Remarks

Two things were pursued in this paper:

1. defining pRC as an efficient particular form of heterogeneous computing
2. defining an elementary abstract model for the accelerator used in pRC.

pRC computing is an intermediate form that compromises between RC and hybrid systems that use programmable ASICs. RC systems require high-performance circuit designers or efficient compilers from high-level programming languages to hardware description languages. We lack both resources. Insufficient quantity, in the first case, and of an acceptable quality in the second case. At the other end, programmable ASICs used as accelerators are parallel machines whose efficient use is hindered by structural and/or architectural mismatches. Indeed, the parallel mono-chip structures we have are either *ad hoc* aggregates of several mono-core processors, or many-core structures dedicated strictly to a particular domain.

The flexibility offered by FPGA technology was used to define the accelerator as a configurable & parameterizable programmable parallel cellular structure. Instead of an *ad hoc* configuration or a specialized structure, we started from a mathematical computational model – that of Stephen Kleene –, and the resulting *elementary abstract model* was subjected to three tests: the functional forms introduced by John Backus, the patterns identified for parallel computation and *dwarfs* investigated at the University of Berkeley.

Regarding the programming of a heterogeneous system, the experience of the last two decades directs the attention of developers to function libraries as the main architectural tools. In this sense we quote:

> *The software span connecting applications to hardware relies more on parallel software architectures than on parallel programming languages. Instead of traditional optimizing compilers, we depend on autotuners, using a combination of empirical search and performance modeling to create highly optimized libraries tailored to specific machines.* [4]

A hardware-accelerated and well-tuned, why not autotuned, application oriented kernel-library seems to be the most realistic solution for an MSRA accelerator as part of a pseudo-reconfigurable computing system.

**Acknowledgements** - The author received technical support, in the first attempt to impose a pRC product, from the following collaborators who contributed to the development of the *ConnexArray*$^{TM}$ technology [23]: Emanuele Altieri, Frank Ho, Mihaela Maliţa, Bogdan Mîţu, Marius Stoian, Dominique Thiébaut, Tom Thomson, Dan Tomescu.

# References

[1] ANTONESCU M., ŞTEFAN M. G., *Multi-Function Scan Circuit*, Proceedings of the 43nd International Semiconductor Conference CAS 2020, October 2020, Sinaia, Romania, pp. 123–126.

[2] ANTONESCU M., MALIŢA M., ŞTEFAN M. G., *FFT on a Heterogeneous System with a Map-Scan Accelerator*, Available at: `http://users.dcae.pub.ro/~gstefan/2ndLevel/technicalTexts/2021_FFT.pdf`

[3] ASANOVIC K., BODIK R. *et al.*, *The landscape of parallel computing research: A view from Berkeley*, Technical Report EECS-2006-183, Berkeley University, December 2006. Available at: `https://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf`

[4] ASANOVIC K., BODIK R. *et al.*, *A View of the Parallel Computing Landscape*, Communications of the ACM, **52**(10):56–67, 2009.

[5] BACKUS J., *Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*, Communications of the ACM, **21**(8):613–641, 1978.

[6] CHURCH A., *An unsolvable problem of elementary number theory*, American Journal of Mathematics **58**(2):345–363, 1936.

[7] DAVIS M., *The Undecidable. Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, Dover Publications, Inc., Mineola, New-York, 2004.

[8] DRAGOMIR V., ŞTEFAN G., *All-Pair Shortest Path on a Hybrid Map-Rreduce Based Architecture*, Proceeding of the Romanian Academy, Series A, **20**(4):411–417, 2019

[9] DRAGOMIR V., ŞTEFAN G., *Sparse Matrix-Vector Multiplication on a Map-Reduce Many-Core Accelerator*, ROMJIST, **23**(3):262–273, 2020.

[10] GOGA N., MALIŢA M., MIHĂIŢĂ D., ŞTEFAN M. G., *FPGA Based Accelerator for Molecular Dynamics*, Proceedings of ACSE 2016, Rome, 27-29 July

[11] GÖDEL K., *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I* (On Formally Undecidable Propositions of Principia Mathematica and Related Systems), Monatshefte für Mathematik und Physik, **38**:173–198, 1931. Republished in [7].

[12] HILBERT D., *Mathematical Problems. Lecture Delivered Before the International Congress of Mathematicians at Paris in 1900*. Available at: https://www.ams.org/journals/bull/1902-08-10/S0002-9904-1902-00923-3/S0002-9904-1902-00923-3.pdf

[13] HILBERT D., and ACKERMAN W., *Grundzüge der theoretischen Logik* (Principles of Mathematical Logic). Springer-Verlag 1928.

[14] HILL M. D., REDDY V. J., *Accelerator-level Parallelism*, Com. of the ACM, **64**(12)36:38, 2021.

[15] KLEENE S., *General recursive functions of natural numbers*, Math. Annalen, **112**(5):727–742, 1936.

[16] MALIŢA M., ŞTEFAN G., THIÉBAUT D., *Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation*, ACM SIGARCH Computer Architecture News, **35**(5):32–38, 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.

[17] MALIŢA M., POPESCU G. V., ŞTEFAN G., *Heterogenous Computing System for Deep Learning*, in Witold Pedrycz, Shyi-Chen (Eds.): *Deep Learning: Concepts and Architectures*, Springer International Publishing, pp. 287–319, 2019.

[18] MALIŢA M., POPESCU G. V., ŞTEFAN G., *Heterogenous Computing for Markov Models in Big Data*, 2019 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA pp. 1500–1505

[19] McCOOL M., ROBISON A. D., REINDERS J., *Structured Parallel Programming. Patterns for Efficient Computation*, Morgan Kaufman, 2012.

[20] POST E., *Finite Combinatory Processes – Formulation 1*, J. of Symbolic Logic, **1**(3):103–105, 1936.

[21] ŞTEFAN G., *et al.*, *The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing*, Hot Chips: A Symposium on High Performance Chips, Memorial Auditorium, Stanford University, August 20 to 22, 2006. Available at: https://old.hotchips.org/archives/2000s/hc18/ in HC18-S5: Parallel Processingat at 34'40".

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

[23] ŞTEFAN G., *The Connex Project*, Available at: http://users.dcae.pub.ro/~gstefan/2ndLevel/connex.html

[24] TURING A., *On computable numbers with an application to the Eintscheidungsproblem*, Proceedings of the London Mathematical Society, **42**(1):230–256, 1936 and a corection in **43**(6):544–546, 1937.

## APPENDIX

# Composition is the only independent rule in Kleene's model of partial recursive functions

When, around 600 BC, the semi-mythical seer Epimenides of Cnossos, so an inhabitant of Crete, reportedly stated that "All Cretans are liars", in the Occidental culture all the logical minds get in trouble for more than two and half millenia, until, after a rigorous reformulation of the problem by David Hilbert (1862 – 1943) [12] [13], the logician Kurt Friedrich Gödel (1906 – 1978 ) proved that the truth of such a sentence is not decidable [11]. Then, after only five years, independently, but almost "synchronously", four mathematicians

– Alonzo Church (1903 – 1995) [6], Stephen Cole Kleene (1909 – 1994) [15], Emil Leon Post (1897 – 1954) [20], and Alan Mathison Turing (1912 – 1954) [24] – published their mathematical version for Gödel's incompleteness theorem. All the four versions proved to be equivalent, but Turing's solution was first considered as the source of an engineering solution for computation. Thus, the most important negative result in the history of logic – the Gödel's incompletness theorem – triggered the start of the computing era. Now, after more than half a century, it is time for another approach to take the lead. To substantiate the parallel calculation, Kleene's solution seems to fit better.

## Kleene's Model of Partial Recursive Functions

**Definition 1.** *Let be the positive integers $x, y, i \in \mathbb{N}$ and the sequence $X = \langle x_0, x_1, \ldots, x_{n-1} \rangle \in \mathbb{N}^n$. Any partial recursive function $f : \mathbb{N}^n \to \mathbb{N}$ can be computed using three **initial functions**:*

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

*and the application of the following three **rules**:*

- **Composition:** *$f(X) = g(h_0(X), \ldots, h_{p-1}(X))$, where: $f : \mathbb{N}^n \to \mathbb{N}$ is a total function if $g : \mathbb{N}^p \to \mathbb{N}$ and $h_i : \mathbb{N}^n \to \mathbb{N}$, for $i = 0, 1, \ldots p-1$, are total functions*
- **Primitive recursion:** *$f(X, y) = g(X, f(X, (y-1)))$, with $f(X, 0) = h(X)$ where: $f : \mathbb{N}^{n+1} \to \mathbb{N}$ is a total function if $g : \mathbb{N}^{n+1} \to \mathbb{N}$ and $h : \mathbb{N}^n \to \mathbb{N}$ are total functions.*
- **Minimization:** *$f(x) = \mu_y[g(x, y) = 0]$, which means: the value of the function $f : \mathbb{N} \to \mathbb{N}$ is the smallest $y$,* ***if any****, for which the function $g : \mathbb{N}^2 \to \mathbb{N}$ takes the value $g(x, y) = 0$.*

◇

Kleene's model looks like a good candidate for a mathematical model for parallel computing as the Turing's model was for the mono-core computation. In this respect, the composition seems to be a natural embodiment of a many-core abstract model for the parallel computing engine. The following conjecture has a big chance to become a theorem.

**Conjecture 1.** *The composition rule, implemented as a two level structure (see Figure 3):*

- *the* **map** *level: a linear array of circuits, one for each $h_i(X)$ function*
- *the* **reduce** *level: a log-depth tree-like network of circuits for $g(h_0(X), \ldots, h_{p-1}(X))$*

*where the functions $h_i(X)$ and the function $g(h_0(X), \ldots, h_{p-1}(X))$ are initial functions or hierarchic compositions of initial functions, computes any functions $f : \mathbb{N}^n \to \mathbb{N}$.*

◇

Two kinds of parallelism are emphasized by the composition rule:

- a $n$-degree of synchronic parallelism between the computation performed in the circuits of the map level
- a 2-degree of diachronic (pipeline) parallelism between the computation on the map level and the computation of the reduction level

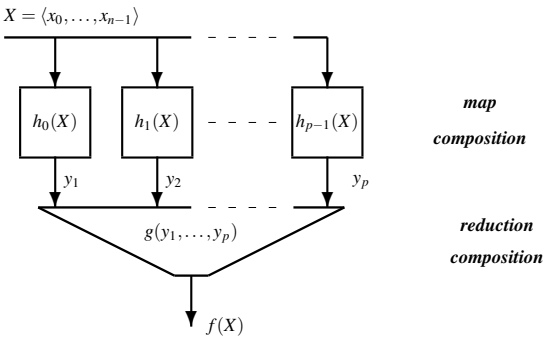


**Fig. 3. The circuit version of composition**. It is a two-layer construct: the parallel expanded *map* layer serially connected with the *reduction* layer.

If the reduce circuit is detailed so as its function is performed as a composition, as follows:

$$g(y_0, y_1 \ldots, y_p) = g(g(y_0, y_1), g(y_2, y_3), \ldots)$$

where $g$ is an associative and commutative function, then, occurs a new map level with a $n/2$-degree of synchronic parallelism followed by a reduce level with $p/2$ inputs. The process continues until a the binary form of the function $g$ is reached. Therefore, the overall degree of parallelism is theoretically: $\delta = (2p - 1)/(1 + log_2 p)$.

In the next sections will be proved that, for the other two rules, specific compositions can be used, so as we are in the position to conclude that the computation model proposed by Kleene leads to implementations involving only compositions for which the previous theorem applies.

## Preliminary Definitions

**Definition 2.** *The reduction-less composition or* map composition*, MC, is the particular composition* $f : \mathbb{N}^n \to \mathbb{N}^p$ *where:*

$$f(X) = f(x_0, \ldots x_{n-1}) = \langle h_0(X), \ldots h_{p-1}(X) \rangle = \langle y_0, \ldots y_{p-1} \rangle$$

$h_i : \mathbb{N}^n \to \mathbb{N}$*, and* $g(y_0, \ldots, y_{p-1}) = \langle y_0, \ldots, y_{p-1} \rangle$ *is the identity function, for* $i = 0, \ldots, p-1$.

◇

**Definition 3.** *The map-less composition or* reduction composition*, RC, is the particular composition* $f : \mathbb{N}^n \to \mathbb{N}$ *where:*

$$f(X) = f(x_0, \ldots x_{n-1}) = g(x_0, \ldots x_{p-1})$$

*with* $y_i = h_i(X) = SEL(i, X) = x_i$*, for* $i = 0, \ldots, p-1$ *and* $n = p$.

◇

According to the previous two definitions, the composition rule can be considered as having a ***map-reduce*** structure (Figure 3), where a MC is serially connected with a RC. The two functional level can have associated the physical implementation with the $h_i$ functions and the $g$ function embodied in various forms, starting from combinational circuits and reaching the complexity and competence of a processor, even a computer.

**Definition 4.** *The RC function* $redOR : \mathbb{N}^n \to \mathbb{N}$ *is*

$$redOR(X) = x_0 | x_1 | \ldots | x_{n-1}$$

*where:* $X = \langle x_0, x_1, \ldots, x_{n-1} \rangle$*, and* $|$ *denote the bitwise OR logical function.*

◇

**Definition 5.** *Map composing two MC*

$$MAC_H(X) = \langle h_0(SEL(0,X)), h_1(SEL(1,X)), \ldots \rangle = \langle y_0, y_1, \ldots \rangle$$

$$MAC_G(X) = \langle g_0(SEL(0,X)), g_1(SEL(1,X)), \ldots \rangle$$

*means:* $MAP_G(MAP_H(X)) = \langle g_0(y_0), , g_1(y_1), \ldots \rangle$, *where:* $X = \langle x_0, x_1, \ldots \rangle$.

◇

**Definition 6.** *Shift map composing* $MAC_H(X)$ *and* $MAC_G(X)$ *(defined in the previous definition) means:*

$$shiftMAP_G(\langle x, MAP_H(X) \rangle) = shiftMAP_G(\langle x, y_0, y_1, \ldots \rangle) = \langle g_0(x), g_1(y_0), g_2(y_1), \ldots \rangle$$

◇

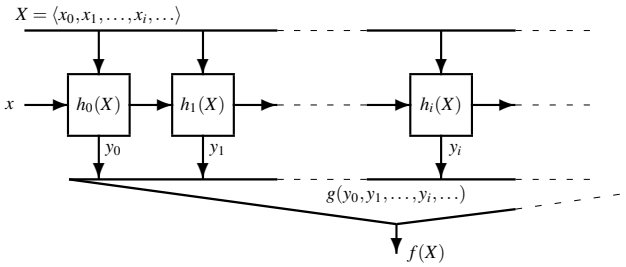


**Fig. 4. The circuit associated to the composition rule expanded with the serial connection.**

Thus, the shift map composition could be associated to serial connections between the cells performing the functions $g_i$, while the leftmost cell receives $x$. Results the circuit version in Figure 4.

**Definition 7.** *A SCAN circuit is a* $(-1+2log_2 n)$*-layer network of MC each defined by:*

$$scanLAYER(X) = \langle h_0(SEL(left_0, X), SEL(right_0, X)), \ldots, h_{n-1}(SEL(left_{n-1}, X), SEL(right_{n-1}, X)) \rangle$$

*On each layer the indexes* $left_i$ *and* $right_i$ *are specified according to the connections defined in the Beneš permute nework, and the functions* $h_i$ *are defined according to the function of the SCAN network: permute, prefix, split, ... defined in [1].*

◇

**Definition 8.** *We define prefixSCAN as a custom SCAN circuit for the OR prefix function, so that receiving a binary input vector* $B = \langle b_0, b_1, \ldots, b_{n-1} \rangle$ *returns* $prefixB = \langle b_0, b_0 | b_1, \ldots, OR_0^i b_j, \ldots, OR_0^{n-1} b_j \rangle$.

◇

**Definition 9.** *The MC function scanFIRST* : $\{0,1\}^n \to \{0,1\}^n$ *is:*

$$scanFIRST(B) = prefixSCAN(B) \& \ (shiftMAP(prefixSCAN(B)))$$

*where:* $B = \langle b_0, b_1, \ldots, b_{n-1} \rangle$ *is a Boolean sequence.*

◇

The *scanFIRST(B)* function identifies the first occurrence of 1 in a the Boolean sequence $B$.

## Primitive Recursion Computed as a Sequence of Compositions

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

*Proof.* The primitive recursion rule could be applied using its iteratively expanded form:

$$f(x,y) = g(x, f(x,y-1)) = \ldots = \underbrace{g(x, g(x, g(x, \ldots g}_{(y-1)\ times}(x, f(x,1))\ldots))) =$$

$$= \underbrace{g(x, g(x, g(x, \ldots g}_{y\ times}(x, f(x,0))\ldots))) = \underbrace{g(x, g(x, g(x, \ldots g}_{y\ times}(x, h(x))\ldots)))$$

Let be, in Figure 5, the specific instantiation of the *shiftMAP* function (see Definition 5). It computes iteratively, starting in the first stage, $P_0$, with the function $f(x,0) = h(x)$, the values $f(x,i)$ for $i = 0,1,....$ In each stage the predicate $(y = i)$ is computed. The functions $P_i$, for $i = 1,2,\ldots$, takes form $P_{i-1}$ the value of $f(X,i-1)$ and computes $f(X,i)$. The *redOR* function takes from the *shiftMAP* function its arguments as $(y = i)\ ?f(X,i) : 0$ for $i = 0,1,2,\ldots$. Because for only one $i$ the predicate $y = i$ takes the value 1, the function *redOR* returns the value of $f(X,y)$.

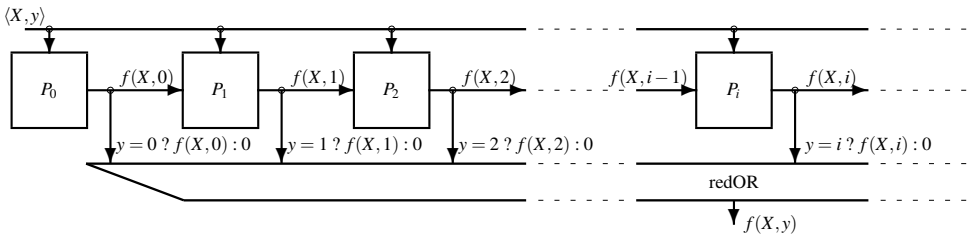


**Fig. 5. The** *shiftMAP* & *redOR* **circuit version for the partial recursive rule**.

Thus, for primitive recursion we need to compose two compositions, *shiftMAP* and *redOR*. □

Figure 5 presents the circuit version of the function obtained by composing a specific *shiftMAP* function with the *redOR* function. The two stage computation just described, as a structure indefinitely extensible to the right, is a theoretical model because the index $i$ takes values no matter how large, similar with the "infinite" tape of Turing Machine. But, it is very important that the algorithmic complexity of the description is in $O(1)$, because the functions $P_i$, *shiftMAP* and *redOR* have constant size descriptions.

## Minimization Computed as a Sequence of Compositions

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

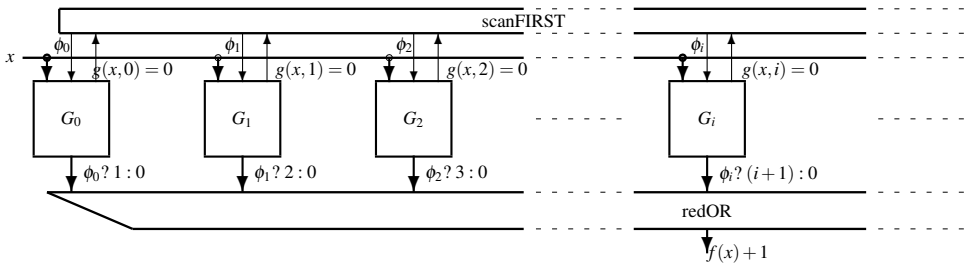


**Fig. 6. The circuit structure for the minimization rule**.

*Proof.* The minimization (least-search) rule computes the value of $f(x)$ as the smallest $y$, **if any**, for which $g(x,y)=0$.

Let be, a specific structure from Figure 6. Each cell on the map level computes the pair $\langle predicate, value\rangle$: $G_i(x,\phi_i)=\langle(g(x,i)=0),(\phi_i\ ?\ (i+1):0)\rangle$. The predicate is sent to the scan circuit, while the value to the reduction circuit. The *scanFIRST* loop points to the first cell, if any, which provided the predicate $(g(x,i)=0)=1$. The reduction level computes *redOR* selecting to the output the value $i+1$ for $\phi_i=1$, if any. If for $x=a$ the output of the reduction is 0, then the function is not defined for $x=a$, else the output takes the value $f(x)+1$, because the value 0 is reserved to indicate that the function is not defined for the value $x$ applied on the input. □

The computation just described is only a theoretical model, because the index $i$ has an indefinitely large value. But, the size of the algorithmic description remains $O(1)$.

## Partial Recursion Means Composition Only

Kleene's approach defines, besides the composition rule, the other two rules, ordinary (primitive) recursion and minimization (least-search), only for providing the means for classifying the recursive functions (to emphasize in the class of recursive functions the partial recursive functions and primitive recursive functions). Therefor, to define the computation the next corollary makes the necessary and sufficient delimitation.

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

◇

The primitive recursion is differentiated from the partial recursion by the main fact that it does not require the scan loop. It is only a straight forward sequence of compositions. For the partial recursive rule, the additional loop interferes with the sequence of compositions in order to validate intermediate results. Thus, an actual abstract model based on the Kleene's computational model must provide also a sequencing mechanism. This means at least to provide programmability at the cells level and an external control.

Corollary 1 can be used to define an abstract model for parallel computation. The resulting structure is able to support the hybrid solution for advanced parallel computation. A Church inspired lambda architecture [6] can be defined for the complex control, while a Kleene inspired architecture could be used for solving problems raised by the intense computation. The operating systems problems can remain to be handled by Turing/Post [24] [20] inspired engines.