**ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY**
Volume **19**, Numbers 1-2, 2016, 148–165

# FPGA-Based Programmable Accelerator for Hybrid Processing

Gheorghe M. ŞTEFAN[1], Călin BÎRA[1],
Radu HOBINCU[1], Mihaela MALIŢA[2]

[1] DCAE Department, Politehnica University of Bucharest, Romania
E-mail: {gheorghe.stefan,calin.bira,radu.hobincu}@upb.ro

[2] Computer Science Dept., Saint Anselm College, Manchester, NH, USA
E-mail: mmalita@anselm.edu

**Abstract.** The emergence of ***CPU/FPGA hybrid processors*** promotes the FPGA accelerators as circuits which perform the critical functions associated to an application. As circuits, they are designed and optimized by hardware designers or, with a debatable efficiency, by specialized compilers. We propose, as an intermediate solution, to use a predesigned parameterized family of programmable generic structures, ready to be synthesized in FPGA technology as highly parallel computing engines able to perform efficiently data intense computations. The solution loses, in the worst cases, tens of percent of the processing power, compared with optimized hardware design, but gains a lot in flexibility allowing the number of skilled users to increase with few orders of magnitude. The proposed solution supposes to select, from a family of predesigned structures, an appropriate RTL module, to implement it in FPGA, and to program the resulting parallel engine in a high-level programming language by a software designer. The theoretical foundation for the generic family of parallel engines is grounded on the synergy between Stephen Kleene's computational model of *Partial Recursive Functions* and the *Functional Programming Systems* proposed by John Backus.

## 1. Introduction

The research whose start is presented in this paper was triggered by a surprising and promising event: *Intel* announced in the summer of 2015 the acquisition of *Altera*. The biggest processor manufacturer couples its technology with one of the biggest FPGA producers. The marriage once consumed, the expected baby is a ***CPU/FPGA hybrid***

***processor***. Will it be a monster or a *Prince-Charming*? The talking of the town is encouraging.

> *"Large tech companies such as Microsoft (NASDAQ:MSFT) and Baidu (NASDAQ:BIDU) have already started experimenting with CPU/FPGA hybrid processors. Microsoft says that it used FPGAs in its data centers to speed up Bing Search. The company said that integrating FPGAs into its servers produced a 95% performance gain with a mere 10% increase in power consumption."*[1]
>
> *"The acquisition will couple Intel's leading-edge products and manufacturing process with Altera's leading field-programmable gate array (FPGA) technology. The combination is expected to enable new classes of products that meet customer needs in the data center and Internet of Things (IoT) market segments."*[2]

Putting together a processor and a fully reconfigurable circuit provides a big paradigm shift. The expectations are very high. Remember the saying: *90% of time the engine runs 10% of code*. In many applications most of the time the computation runs a small part of code that calls only a few *intense computational functions*. Well, for these intense computational functions the FPGA, part of the CPU/FPGA hybrid processors, could be the most appropriate ***computational accelerator***.

We are already accustomed with the distinction between the complex computational part and the intense computational part of an application, i.e., with the segregation of a small part of the code which runs most of the time. In order to minimize the running time we deal usually with a small fragment of code. The current radical solution for this (data) intense part of computation is to add an FPGA-based specific accelerator.

The accelerator is an FPGA centered system, featured usually with its own local memory, which is connected as tightly as possible with the host computer. The most common interfaces are FSB, PCIe or PCI.

There are two main distinct ways to actualize the function for an FPGA accelerator:

- generating the RTL code of a circuit programmed in FPGA (see Figure 1a)
- writing in OpenCL, or another programming language, the program which describes the function; for example, the Altera SDK for OpenCL provides a compiler to compile specific kernel functions into the FPGA [25] (see Figure 1b).

We propose a third, intermediary, way (see Figure 1c): to select from a family of generic highly parallel computational structures a component implementable in FPGA for which the application is written in a high level programming language. The solution we propose has advantages over both, the RTL description and the family of OpenCL kernels. On the one hand, an efficient RTL description can be provided only by a well trained digital designer, and on the other hand it is hard to meet the performance requirements and to find an easy way to use the hardware support for a family of OpenCL kernels. The proposed solution (see Figure 1c) consists of:

[1]`http://amigobulls.com/articles/intel-altera-merger-given-the-nod-by-eu-regulators`

[2]`http://www.intc.com/releasedetail.cfm?ReleaseID=915707`

- the system, containing the HOST & MAIN MEMORY connected to the FPGA & LOCAL MEMORY
- a family of Generic RTL parameterized codes which describe massive parallel computing engines
- Compilers & debugging environment for our generic family of parallel engines
- Libraries of functions written in high level languages for various application domains and optimized for our generic family of parallel engines
- Applications written in high level languages using the optimized library of functions.

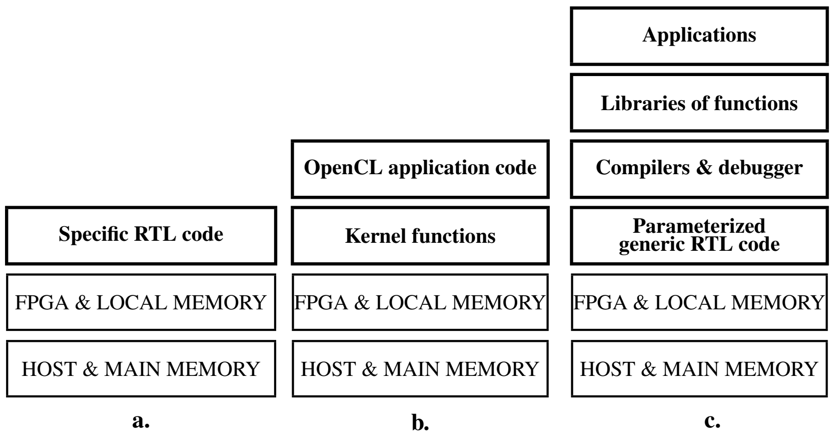

**Fig. 1. FPGA-based accelerators**: on top of the hardware (Host & FPGA with their associated memory) the specific code consists of: (a) *specific RTL code* in Verilog, VHDL, . . ., (b) *OpenCL code* which uses kernel functions translated in specific RTL code, or ***our solution***: (c) *applications code* in C, R, . . . OpenCL complied to run on a parameterized generic highly parallel programmable engine.

The advantages of our proposal are:

- the application is a program instead of a hardware design or a program running a predefined library of functions implemented in RTL
- the reusability of FPGA resources for a big functional spectrum, because the targeted programable engine is a general purpose programmable one
- time to market is very short because the hardware design step is avoided
- the user friendly proposed system will enlarge the number of users because no specialised hardware designers are involved in the process.

The main application domains are: image processing, financial applications (in R language), big data mining, oil search, molecular dynamics, .... Some of them are already partially investigated (see, for example, [3] [5] [17] [18] [19] [20] [4]). There are also investigations related to how the proposed architecture can be programmed (see [13] [4]).

The paper describes, in the next section, the high-level architecture of the generic massive parallel engine which supports our solution. The third section presents the structure of the generic family of engines. In the last section few applications are sketched and evaluated.

## 2. The architecture of the generic parallel engine

Because our generic parallel engine is conceived to substitute efficiently a circuit which performs data intensive functions, the Kleene's model of computation looks the most promising, because of its function/circuit-based form.

### 2.1. Kleene's parallel computational model

The first basic rule in the partial recursive function model of computation proposed by Stephen Kleene [9], the composition rule:

$$f(x_1,\dots,x_n) = g(h_1(x_1,\dots,x_n),\dots,h_p(x_1,\dots,x_n))$$

is the only one to be considered in our approach, because we already proved, in [11] and [23], that the other two, primitive recursion and minimalization, can be expressed as special compositions. The circuit form of the composition rule is in Figure 2.

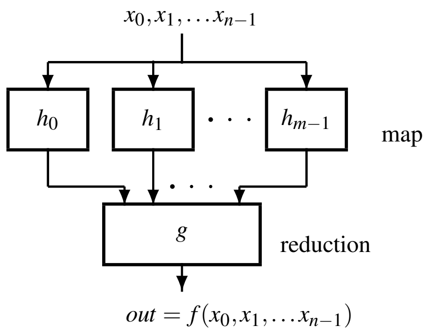

**Fig. 2. The structure associated to the composition rule.** The composition of function $g$ with the functions $h_0, \dots, h_{m-1}$ is performed by a ***two level circuit***. On the first level **many** functions are ***mapped*** together, while on the second a ***reduction*** function is performed.

Five particular forms of composition are used to define the main abstract forms of parallel engine, as follows.

**Data-parallel:** if we consider $h_i(x_1,\ldots,x_p) = h(x_i)$ and $g(y_1,\ldots,y_p) = \{y_1,\ldots,y_p\}$, then,

$$f(x_1,\ldots,x_p) = \{h(x_1),\ldots,h(x_p)\}$$

where $x_i = \{x_{i1},\ldots,x_{im}\}$ are sequences of data. A little more complex form of data-parallel operation is the predicated execution:

$$f(\{x_1,\ldots,x_p\},\{b_1,\ldots,b_p\}) = \{(b_1\ ?\ h_T(x_1) : h_F(x_1)),\ldots,(b_p\ ?\ h_T(x_p) : h_F(x_p))\}$$

where: $b_i$ are Boolean variables.

**Reduction-parallel:** if $h_i(x_i) = x_i$, then the general form becomes:

$$f(x_1,\ldots,x_p) = g(x_1,\ldots,x_p)$$

which operates a reduction from vector(s) to scalar(s).

**Speculative-parallel:** if the functionally different cells – $h_i$ – receive the same input variable, while the reduction performs identity function ($g(y_1,\ldots,y_p) = \{y_1,\ldots,y_p\}$), then,

$$f(x) = \{h_1(x),\ldots,h_p(x)\}$$

where: $x$ is a sequence of data. The *speculative-parallelism* returns a sequence of sequences. There are two ways to differentiate the functions $h_i(x)$:

1. $h_i(x)$: represents a specific sequence of operation for each $i$.
2. $h_i(x) = g(i,x)$: the function has a parameterized variable.

**Time-parallel:** for the special case $p = 1$, $f(x) = g(h(x))$. Then, here is no synchronous parallelism. Only the pipelined parallelism is possible if in each "cycle" a new value is applied to the input. Thus, in each "cycle" the function $h$ is applied to $x(t)$ (which is $x$ at the "moment" $t$) and $g$ is applied to $h(x(t-1))$ (where $x(t-1)$ is the value applied to the input at the "moment" $t-1$).

Many applications of $f(x) = g(h(x))$ results in the $m$-level *"pipe"* of functions:

$$f(x) = f_m(f_{m-1}(\ldots f_1(x)\ldots))$$

where: $x$ is an element in a stream of data. The resulting structure is a *time-parallel* one if in each "cycle" a new value for $x$ is inserted in the pipe, *i.e.*, it is applied to $f_1$.

**Thread-parallel:** if $h_i(x_1,\ldots,x_n) = h_i(x_i)$ and $g(h_1,\ldots,h_p) = \{h_1,\ldots,h_p\}$, then the general form of composition is reduced to:

$$f(x_1,\ldots,x_p) = \{h_1(x_1),\ldots,h_p(x_p)\}$$

where: $x_i$ is an sequence of data. Each $h_i(x_i)$ represents a distinct and independent *thread.*

### 2.2. Integral parallel architecture & Backus's FP systems

The previous five types of parallelism define [22] an Integral Parallel Architecture (IPA). Backus's concept of *Functional Programming Systems* (FPS) [2] can be seen also as an *architectural description* for IPA. In the following we use a FPS-like form to sketch the architecture for the abstract model defined in the previous subsection. Thus, we provide the *virtual machine* description of a parallel engine able to compete with circuits in performing data intense computations. The description contains functions which map objects into objects, where an object could be:

- atom, $x$; special atoms are: $T$ (true), $F$ (false), $\phi$ (empty sequence)
- sequence of objects, $\langle x_1, \ldots, x_p \rangle$, where $x_i$ are atoms or sequences
- $\perp$: undefined object

**Primitive functions** are informally and partially described in the following.

**Atom** : $atom : x \equiv (x\ is\ an\ atom) \rightarrow T;F$
`atom` applied to `x` means: if the argument is an atom, then T is returned, else F is returned (it is the formalism used in [2]).

**Null** : $null : x \equiv (x = \phi) \rightarrow T;F$

**Equals** : $eq : x \equiv ((x = \langle y,z \rangle)\ \&\ (y = z)) \rightarrow T;F$

**Binary operations** : $op2 : x \equiv ((x = \langle y,z \rangle)\ \&\ (y,z\ atoms)) \rightarrow y\,op2\,z$
where: $op2 \in \{add, sub, mult, eq, lt, gt, leq, and, or, ...\}$

**Unary operations** : $op1 : x \equiv ((x = y)\ \&\ (y\ atom)) \rightarrow op1\,y$
where: $op1 \in \{inc, dec, zero, not\}$.

**Selector** : $i : x \equiv ((x = \langle x_1, \ldots, x_p \rangle)\ \&\ (i \leq p)) \rightarrow x_i$

**Rotate** : $rot : x \equiv (x = \langle x_1, \ldots, x_p \rangle) \rightarrow \langle x_2, \ldots, x_p, x_1 \rangle$

**Transpose** : $trans : x \equiv (x = \langle \langle x_{11}, \ldots, x_{1m} \rangle, \ldots, \langle x_{n1}, \ldots, x_{nm} \rangle \rangle) \rightarrow$
$\langle \langle x_{11}, \ldots, x_{n1} \rangle, \ldots, \langle x_{1m}, \ldots, x_{nm} \rangle \rangle$

**Distribute** : $distr : x \equiv (x = \langle y, \langle x_1, \ldots, x_p \rangle \rangle) \rightarrow \langle \langle y, x_1 \rangle, \ldots, \langle y, x_p \rangle \rangle$

The above described functions and similar ones are used to write functional forms used to define programs.

**Functional forms** are used to define complex functions starting from the set of primitive functions.

**Apply to all** : $\alpha f : x \equiv (x = \langle x_1, \ldots, x_p \rangle) \rightarrow \langle f : x_1, \ldots, f : x_p \rangle$
For example:
$\alpha\, add : \langle \langle x_1, y_1 \rangle, \ldots, \langle x_p, y_p \rangle \rangle \rightarrow \langle add : \langle x_1, y_1 \rangle, \ldots, add : \langle x_p, y_p \rangle \rangle$
expands the function *add*, defined on atoms, to sequences, $\langle \langle x_1, \ldots, x_p \rangle \langle y_1, \ldots, y_p \rangle \rangle$, transposed in a sequence of pairs $\langle x_i, y_i \rangle$.

`Apply to all` represents the ***data-parallel*** computation.

**Insert** : $/f : x \equiv ((x = \langle x_1, \ldots, x_p \rangle)) \rightarrow f : \langle x_1, /f : \langle x_2, \ldots, x_p \rangle \rangle$
The function $f$ has as argument a sequence of objects and returns an object.

`Insert` represents the ***reduction-parallel*** computation.

**Construction** : $[f_1, \ldots, f_n] : x \equiv \langle f_1 : x, \ldots, f_n : x \rangle$

`Construction` represents the ***speculative-parallel*** computation.

**Composition** : $(f_q \circ f_{q-1} \circ \ldots \circ f_1) : x \equiv f_q : (f_{q-1} : (\ldots : (f_1 : x) \ldots)))$

`Composition` represents ***time-parallel*** computation if the computation is applied to a stream of objects, $x = |x_n, \ldots, x_1|$.

**Threaded construction** :
$\theta[f_1, \ldots, f_p] : x \equiv (x = \langle x_1, \ldots, x_p \rangle) \rightarrow \langle g_1 : x_1, \ldots, g_p : x_p \rangle$
is a special case of construction for: $f_i = g_i \circ i$, where: $g_1 : x_1$ represents an independent thread.

`Threaded construction` represents the ***thread-parallel*** computation.

**Condition** : $(p \rightarrow f; g) : x \equiv ((p : x) = T) \rightarrow f : x; ((p : x) = F) \rightarrow g : x$
represents a conditioned execution.

**Binary to unary** : $(bu\, f\, x) : y \equiv f : \langle x, y \rangle$
It is used to express any function as an unary function. This function allows the algebraic manipulation of programs.

**Definitions** are used to write programs conceived as functional forms, as follows:

$$\textbf{Def}\ new\_function\_symbol \equiv functional\_form$$

**Example** : Let be the following definitions used to compute the *sum of absolute difference* (SAD) of two sequence of numbers:

**Def** $SAD \equiv (/+) \circ (\alpha ABS) \circ trans$
**Def** $ABS \equiv lt \rightarrow (sub \circ REV); sub$
**Def** $REV \equiv (bu\ perm \langle \bar{2}, \bar{1} \rangle)$

**Kleene – Backus synergy** occurs as the relation between the five abstract parallel engines resulting from Kleene's model and the FPSs proposed by Backus as following:

| **Kleene's parallelism** | ↔ | **Backus's functional forms** |
|---|---|---|
| *data-parallel* | ↔ | `apply to all` |
| *reduction-parallel* | ↔ | `insert` |
| *speculative-parallel* | ↔ | `construction` |
| *time-parallel* | ↔ | `composition` |
| *thread-parallel* | ↔ | `threaded construction` |

Thus, the consistency between the Kleene's model, and the FPS proposed by Backus represent a solid foundation for parallel computation as a good competitor for circuits in computing data intense functions.

### 2.3. The map-reduce structure of the generic parallel engine

The generic parallel engine is designed as a map-reduce structure having as theoretical prototype the circuit associated to the composition rule in Kleene's model of partial recursive functions (see Figure 2). It is about a linear array of cells whose output is a sequence of atoms which is reduced to an atom using a reduction network[3]. The map-reduce structure is represented in Figure 3, where:

**Linear Array** is a network of linearly connected cells, $[cel_1, cel_2, \ldots cel_p]$, it is used mainly to map a function over a sequence/sequence_of_sequences (data parallelism) or a sequence/atom over a sequence of functions (speculative parallelism); current FPGAs support projects up to 4096 32-bit cells, each containing:

- an execution unit which receives in each cycle an instruction executed according to the state of each cell
- a cell control unit when cells are used as processing units
- cell's memory used to store data (sequences and atoms) or instructions when the cells are used as processing units
- a left/right connection unit used to connect each cell with its left and right neighbour

**Controller** issues in each clock cycle an instruction to be broadcasted over **Linear Array**

**Broadcast** is a *log*-depth network used to distribute the instruction issued by **Controller**

**Reduction** is a *log*-depth circuit used to perform the reduction functions (addition, logic OR, max, ...)

[3]The first embodiment of this kind of structure is the many-cell engine ***ConnexArray***$^{TM}$, the core of the *BA*1024 SoC implemented by *Brightscale* for the HDTV market (see [17] [19] [21] [22]).

**Scan** is a *log*-depth loop closed over **Linear Array** (it receives usually a Boolean sequence and sends back the result of a prefix function)

**Trans** is a two-direction connection used to exchange data between **Linear Array** and **Local Memory**

**FSB/PCIe** is the host connection.

The user's view of this map-reduce engine is given by the two memory domains:

$V$ **domain** : defined by the sum of the cell's memory organised as a two-dimension array

$$V = \{v_1, v_2, \ldots, v_m\}$$

containing $m$ $p$-scalar *horizontal* vectors:

$$\begin{gathered} v_1 = \langle x_{11}, \ldots, x_{1p} \rangle \\ v_2 = \langle x_{21}, \ldots, x_{2p} \rangle \\ \ldots \\ v_m = \langle x_{m1}, \ldots, x_{mp} \rangle \end{gathered}$$

which can by seen also as an array of $p$ $m$-scalar *vertical* vectors of form

$$w_i = \langle x_{1i}, \ldots, x_{mi} \rangle$$

for $i = 1, \ldots, p$, each of them stored in a cell memory

$S$ **domain** : $S = \{s_0, s_1, \ldots, s_{n-1}\}$ of the local memory.

## 3. The Generic Map-Reduce High Level Architecture

The Generic Map-Reduce High Level Architecture (MRA) hides to the user the vector register level and the scalar register level operations. Three independent sub-architectures are defined in MRA:

- *data processing* sub-architecture: $f_{map} : (V \times V|S) \rightarrow V$ or $f_{reduce} : V \rightarrow S$
- *data transfer* sub-architecture: $f_{load} : S^* \rightarrow V$ or $f_{store} : V \rightarrow S^*$, functions used to transfer streams of data (from $S^*$) between $V$ and $S$
- *inter-cell communication* sub-architecture: $f_{com} : V \rightarrow V$, functions used to perform specific data transfers in $V$.

The functions of the three sub-architectures are defined in SCHEME.

### 3.1. Data processing sub-architecture

The data processing sub-architecture is partially described by the following functions:

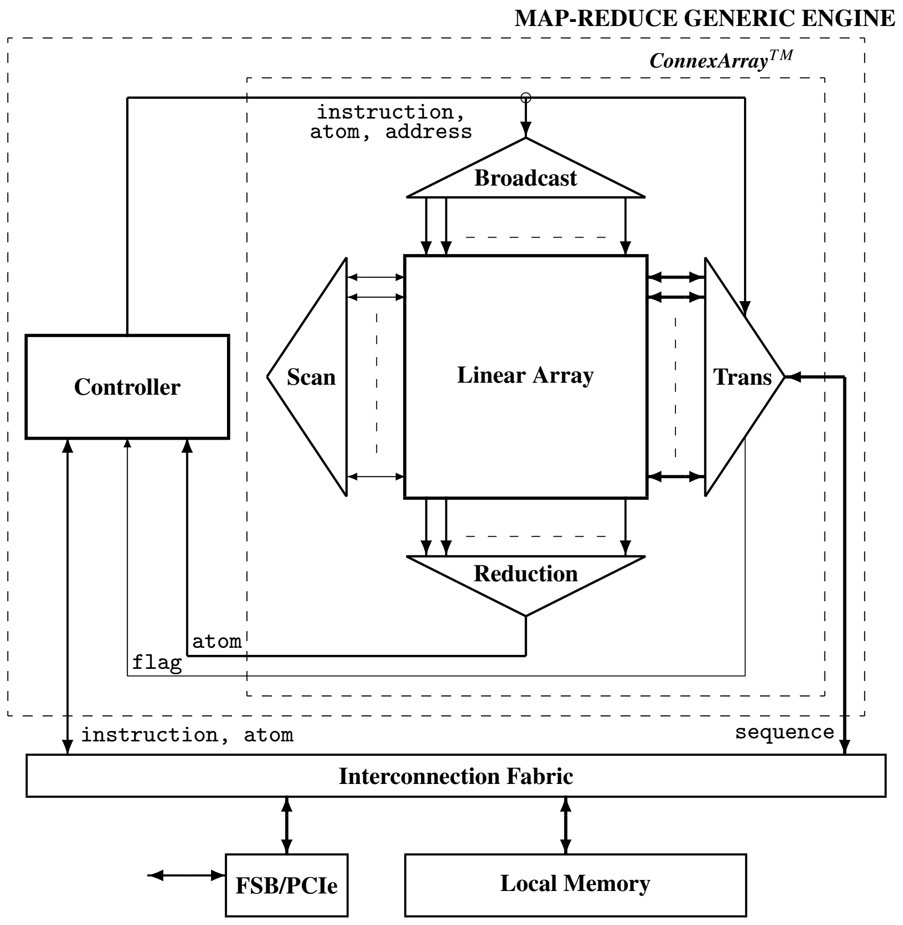

**Fig. 3. The Map-Reduce generic parallel engine.** It is centered on ***ConnexArray***$^{TM}$, a linear array of $p$ execution units, each with its own local memory, connected to the external memory through the **Trans** network. The array has two loops, an external one through **Reduction** net, **Controller** and **Broadcast** net, and an internal one through the **Scan** net.

- `(ResetActive)`: activates all the cells for the execution
- `(Where x)`: keep active the cells where the Boolean vector `x` is 1
- `(ElseWhere)`: keep active only the cells where the Boolean vector `x` is 0 in the previous `Where`
- `(EndWhere)`: restore the configuration of active cells before the previous `Where`
- `(Test x y)`: where `Test` $\in$ {`Lt, Gt, ...`}, and $x, y \in (V|S)$; returns a Boolean vector
- `(SetVector x y)`: where `x` is the vector's address in $V$, and `y` is the vector's content
- `(SetStream x y)`: where `x` is the start address in $S$ and `y` is the stream's content
- `(UnaryOp x)`: where `UnaryOp` $\in$ {`Inc, Dec, Abs, ...`}, and $x \in (V|S)$
- `(BinaryOp x y)`: where `BinaryOp` $\in$ {`Add, Sub, Mult, ...`}, and $x, y \in (V|S)$
- `(RedOp x)`: where `RedOp` $\in$ {`RedAdd, RedMax, RedOr`} are the reduction operations, $x \in V$; returns a scalar
- ...

## 3.2. Data transfer sub-architecture

The data transfer sub-architecture contains functions performed in parallel with the functions of the two other sub-architectures. The competition to the locally distributed memory is dynamically prioritized according to the applications. Few functions are exemplified in the following:

- `(StoreVector many vAddr sAddr)`: stores `many` vectors from $V$, starting at the address `vAddr`, in $S$ starting with the address `sAddr`
- `(LoadVector many vAddr sAddr)`: loads `many` vectors in $V$, starting at the address `vAddr`, from $S$ starting with the address `sAddr`
- `(StoreVectorStrided many vAddr sAddr burst stride)`: performs `StoreVector` in bursts of size `burst` strided in $S$ with stride `stride`
- `(LoadVectorStrided many vAddr sAddr burst stride)`: performs `LoadVector` in bursts of size `burst` strided in $S$ with stride `stride`
- ...

### 3.3. Inter-cell communication sub-architecture

The inter-cell communication sub-architecture contains functions performed in parallel with the functions of the two other sub-architectures. The competition on the locally distributed memory, if it is the case, is always solved favourably for functions belonging to this sub-architecture. Few functions belonging to this sub-architecture are exemplified in the following:

- `(ShiftOp x y)`: with `ShiftOp` $\in$ {`ShiftLeft, ShiftRight, ...`}, `x` is the vector address in $V$, and `y` is the size of the shift operation; the operation is executed in time belonging to $O(y)$
- `(Transpose x y)`: transpose the square matrices of $y \times y$ elements from $V$, stored in $V$ starting at the address `x`, in `y` vectors
- `(Permute x y)`: permute `x` according to the vector of indexes `y`
- ...

### 3.4. Parallel execution in the three sub-architectures

The functions belonging to the three sub-architectures are executed in parallel. The parallel execution implies the competition on the vector memory distributed in the array. For example, the data transfer functions are performed sharing with the other two sub-architectures the internal vector memory $V$. These functions *compete very little* with the main user of the shared resource: the processing sub-architecture. Indeed, the function `(LoadVector vAddr sAddr)`, for example, reads in one clock cycle an entire vector from $V$ and requires $t_{trans} \in O(p)$ cycles for the whole transfer. The inter-cell communication functions use more intense the shared resource $V$. A shift operation, for example, accesses two times the vector memory $V$. For small shifts the share of the vector memory use could be important. In order to be able to optimize the execution, the access to the vector memory $V$ could be dynamically prioritized.

The previously defined sub-architectures request a special function which allow the parallel execution of up to three threads of program: *data processing*, *data transfer*, and *inter-cell communication* thread. The function `ParallelEval` has two or three arguments, as follows:

```
(ParallelEval Prog1 Prog2)
(ParallelEval Prog1 Prog2 Prog3)
```

where, for example, `Prog1` is a data processing program and `Prog2` is a data transfer program, or an inter-cell communication program. The two or the three program threads interact by flags. The following functions are defined to use the flags:

```
    (define (Set f)(define f true))
    (define (Reset f)(define f false))
    (define (Wait cond)(do()(cond)))
```

## 4. Applications

We claim that for the most of functions to be accelerated the above described array of cells behaves almost like a circuit. Let us take the functions for which the Altera SDK for OpenCLTM provides a design environment to implement Open Computing Language applications with FPGA-based accelerators [26]. On the list we find, for example, functions like: vector addition, matrix multiplication, time-domain FIR filter, 1D and 2D FFT, 3D finite difference computation. All these functions, and the similar ones, are suitable for circuit implementation, but in the same time for implementation on our generic map-reduce platform.

The vector addition operation, for example, is the system function `(Add x y)`, belonging to the data processing sub-architecture, where `x` and `y` are vectors. The matrix multiplication is based on the inner product (IP) function, a typical map-reduce operation, defined s follows:

```
(define (IP x y)
        (RedAdd (Mult x y))
)
```

There is no difference between the efficiency in implementing IP on a circuit or in our map-reduce engine. For finite difference computation, the vectorisation of the algorithms is the solution to use efficiently the map-reduce platform. In [3], are presented efficient implementation for the AES or the Batcher's sorting algorithm.

One of the most used function is the vector-matrix multiplication. On our architecture the program is:

```
(define (vectMatrixMult aAddrRes aAddrV aAddrM many )
 (do (( i 0 (+ i 1)))((= i many))
  (SetVector aAddrRes
   (ShiftLeftVal 1 (Vec aAddrRes)
    (RedAdd (Mult (Vec aAddrV) (Vec (+ aAddrM i)))
)))))
```

The resulting two-column executable code, the first column for Controller and the second for Linear Array (see Figure 3), is:

```
         cSEND(6)     ;   CADDRLD     ;
         cLOAD(0)     ;   RLOAD(0)    ;
         cVSUB(1)     ;   MULT(0)     ;
LB(6);   cCPUSHL(1)   ;   RILOAD(127);  // first step of loop
         cBRNZDEC(6);     MULT(0)     ; // second step of loop
         cNOP         ;   NOP         ; // latency step
         ...
         cNOP         ;   NOP         ; // latency step
         cLOAD(9)     ;   SRLOAD      ;
```

The execution time for a $N \times N$ matrix in a system with $P$ cells, where $N \leq P$, is: $T_{vm}(N) = 2N + 4 + log P \in O(N)$ where $log P$ occurs due to the distribution and reduction networks latency. For big $N$ the time for vector-matrix product converges to $2N$ (for $N = 1024$, $T_{vm}(N) = 2.012$).

### 4.1. A case study: FFT computation

FFT computation demands a more elaborated approach because of the "butterfly communication" scheme it supposes. In [3] is presented a solution which avoid the troubles generated by the "butterfly communication" scheme: the linear streams of samples are organized as a two-dimension arrays and loaded in the cell memories. For example, the computation of 1024-sample FFTs is performed organizing each stream in a $32 \times 32$ array. If the number of cells is $p = 256$, then 8 1024-sample FFTs are computed in parallel in the map-reduce engine. Each cell is loaded with a 32-sample column. Thus, the initial data is loaded in 32 256-component horizontal vectors. The algorithm is [8]:

1. load 8 $(32 \times 32)$ arrays in 32 horizontal vectors using `(LoadVectorStrided many vAddr sAddr burst stride)`
2. perform in parallel 256 32-sample FFTs, one in each cell; results $(8 \times 2)$ $(32 \times 32)$ arrays, 8 for the real part and 8 for the imaginary part, stored in 64 256-component horizontal vectors
3. transpose the $(8 \times 2)$ $(32 \times 32)$ arrays, 8 at a time in parallel
4. perform in parallel 256 32-sample FFTs, one in each cell
5. store $(8 \times 2)$ $(32 \times 32)$ arrays (64 256-component vectors) in the local memory using `(StoreVectorStrided many vAddr sAddr burst stride)`

There are two FFT stages, (2) and (4), in one the constants used are identical for each cell, while in the another the constants are pre-stored as horizontal vectors in cell memories. There are two transfer stages, one for load, (1), and another for store, (5). In the middle there is a transpose stage, (3).

The stages (2) and (4) are the computational stages performed with a maximal efficiency, because the degree of parallelism 100%. For this part the programmable system works like a circuit. In order to minimize the overall execution time the contribution of the other three stages, (1), (3) and (5), must be reduced, possibly eliminated. Our threefold architecture allow us to hide the effect of data transfer (load and store) and data move (transpose). The transpose and transfer operation may be executed transparently to the computation due to the parallel execution in the three sub-architectures.

In order to make transparent the transpose operation, a double buffer approach is used. The engine is loaded with samples for $(2 \times 8)$ FFTs. After the stage (2) on the first 8 FFTs, the stage (2) is applied to the next 8 FFTs and in parallel the stage (3) is applied to the first 8 FFTs; then the stage (4) is applied to the first 8 FFTs, while the stage (3) is applied to the next 8 FFTs, and so on. Thus, continuously the data sub-architecture is involved in the stage (2) or (4), while the inter-cell communication sub-architecture is involved in the stage (3).

To make transparent the stages (1) and (5) we must double again the number of FFTs. Then, there are $(4 \times 8)$ FFTs loaded in the cell memories of the map-reduce engine. For $(2 \times 8)$ FFTs are performed the transfer operations, other 8 FFTs are submitted to the stage (2) or (4), and the last 8 FFTs are transposed. Thus, three processes are performed in parallel – transfer, computation, and transpose – competing with great chances with a pure circuit approach.

### 4.2. Implementation

A version of the map-reduce generic structure was synthesised for XILINX vc6vlx240t-1ff784. The design has 256 16-bit cells, each containing 2KB of cell memory. The execution unit is centered on the DSP48E1 module. The reduction functions are: `Add, Max, Or`. The resulting device utilization summary is:

```
Slice LUTs: 59716 out of 150720 39%
Block RAMs:   133 out of    416 31%
DSP48E1s:     257 out of    768 33%
```

Most of the resources are used for ***ConnexArray***$^{TM}$ only (see Figure 3):

```
Slice LUTs: 58710 out of 150720 38%
Block RAMs:   128 out of    416 30%
DSP48E1s:     256 out of    768 33%
```

We conclude that the biggest part of the structure ($\sim 97\%$) represents a simple circuit whose functionality is embedded using the program issued by the **Controller**.

### 4.3. The family of map-reduced generic engine

The family of map-reduced generic engine expands in two ways:

- at the level of ***ConnexArray***$^{TM}$ specifying
  - various size of cells: 8-bit, 16-bit or 32-bit cells, for example

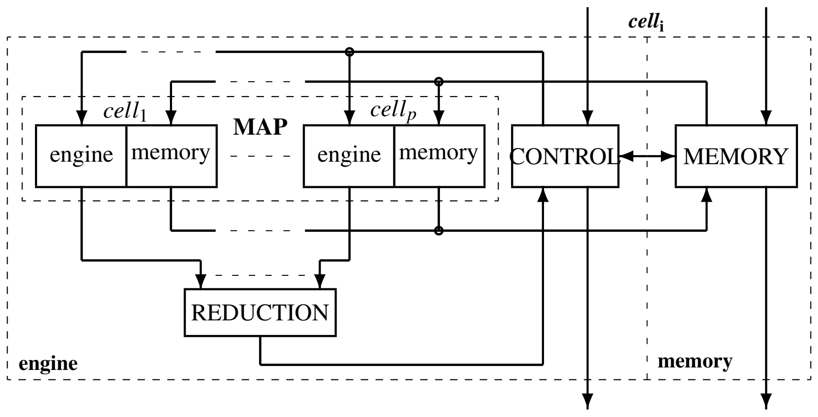

**Fig. 4. The recursive definition of the hierarchy of generic map-reduction engines.** At each level in hierarchy, the MAP stage is a linear array of cells, while the REDUCTION stage is a *log*-depth circuit.

  - the types of arithmetic operations: integer or floating point arithmetic

- as a hierarchy of map-reduce organizations recursively defined in Figure 4, where, at the lowest level:
  - "engine" is an execution/processing unit of 8 to 64 bits
  - "memory" is a few KB of static RAM
  - "MEMORY" is a few GB of dynamic RAM
  - "CONTROL" is a 32-bit sequential processor.

The development of the accelerator is possible in one FPGA or in a network of FPGAs, depending on the actual parameter of the design. The actual trend in FPGA market offers a synergetic support for our approach, because increasingly more standard circuits are implemented directly in silicon, while the programmable part of the FPGA remains to be used mainly for the interconnections between the ASIC style implemented circuits such as memory blocks, DSP modules, ....

## 5. Conclusions

A generic parallel map-reduce architecture is a good solution for competing with circuits in performing data intense computations.

The Kleene-Backus synergy validates theoretically the proposed map-reduce architecture as a generic parallel engine.

The *circuit* aspect of Kleene's composition rule and the *functional* programming style proposed by Backus support our presupposition that an accelerator implemented in FPGA and programmed using a functional programming language must be based on a family of parallel map-reduce engines.

Because more than 97% of the structure in an uniform one, dominated by the local interconnections, the circuit implementation of a map-reduce engine fits with the main restrictions imposed for an FPGA design.

For the emerging ***CPU/FPGA hybrid processor*** our map-reduce generic environment is a promising solution. The maturation of the proposed architecture is an underway process guided by the computational motifs proposed in [1].

**Acknowledgments.** The authors would like to thank to 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 Thiébaut, Tom Thomson, Dan Tomescu.

## References

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

[2] BACKUS J., *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, Communications of the ACM, August 1978, 613–641.

[3] BÎRA C., GUGU L., MALIŢA M., ŞTEFAN GH. M., *Maximizing the SIMD Behavior in SPMD Engines*, Proceedings of the World Congress on Engineering and Computer Science 2013, Vol. **I** WCECS 2013, 23–25 October, 2013, San Francisco, USA, pp. 156–161.

[4] BIRA C., PETRICA L., HOBINCU R., *OPINCAA: A Light-Weight and Flexible Programming Environment For Parallel SIMD Accelerators*, Romanian Journal of Information Science and Technology, Volume **16**, Issue 4, pp. 336–350, 2013.

[5] BIRA C., HOBINCU R., PETRICA L., CODREANU V., COTOFANA S., *Energy-Efficient Computation of L1 and L2 Norms on a FPGA SIMD Accelerator, with Applications to Visual Search*, 18th International Conference on Circuits, Systems, Communications and Computers (CSCC 2014), Volume **2**, pp. 432–436, 2014.

[6] CHU E., GEORGE A., *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. CRC Press, 2000.

[7] COOLEY J.W., TUKEY J.W., *An algorithm for the machine calculation of complex fourier series*, Mathematics of Computation, **19**(90):297–301, 1965.

[8] LORENTZ I., MALIŢA M., ANDONIE R., *Fitting FFT onto an Energy Efficient Massively Parallel Architecture*, The Second International Forum on Next Generation Multicore/ Manycore Technologies, June, 2010.

[9] KLEENE S., *General recursive functions of natural numbers*, Mathematische Annalen, **112**, 5, 1936, pp. 727–742.

[10] MALIŢA M., ŞTEFAN GH., THIEBAUT D., *Not Multi, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation*, ACM SIGARCH Comp. Arch. News, Vol. **35**, 5, Dec. 2007.

[11] MALIŢA M., ŞTEFAN GH., *On the Many-Processor Paradigm*, Proceedings of the 2008 World Congress in Computer Science, Computer Engineering and Applied Computing, vol. **PDPTA'08**, 2008.

[12] MALIŢA M., ŞTEFAN GH., *Parallel RISC Architecture. A Functional Approach Based on Backus's FP language*, Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, 2011, pp. 492–498.

[13] MALIŢA M., GUGU L., BÎRA C., *Connex Simulator*, posted at: `http://www.anselm.edu/internet/compsci/Faculty_Staff/mmalita/HOMEPAGE/research.html`

[14] MOCANU D., GHEOLBANOIU A., HOBINCU R., PETRICA L., *Global Feedback Self Programmable Cellular Automaton Random Number Generator*, Rev. Tc. Ing. Univ. Zulia., Vol. **39**, No. 1, pp. 1–9, 2016.

[15] ŞTEFAN GH., THIÉBAUT D., *Memory Engine for the Inspection and Manipulation of Data*, United States Patent 6,760,821, July 6, 2004; Filed: Aug. 10, 2001.

[16] ŞTEFAN GH., MALIŢA M., *Granularity and Complexity in Parallel Systems*, Proceedings of the 15 IASTED International Conf., 2004, Marina Del Rey, CA, pp. 442–447.

[17] ŞTEFAN GH., *The CA1024: A Massively Parallel Processor for Cost-Effective HDTV*, Spring Processor Forum: Power-Efficient Design, May 15–17, San Jose, CA 2006.

[18] ŞTEFAN GH., SHEEL A., MÎŢU B., THOMSON T., TOMESCU D., *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.

[19] ŞTEFAN GH. *et al.*, *The CA1024: A Fully Programable System-On-Chip for Cost-Effective HDTV Media Processing*, Hot Chips: A Symposium on High Performance Chips, Stanford Univ., August, 2006.

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

[21] ŞTEFAN GH, *One-Chip TeraArchitecture*, Proceedings of the 8th Applications and Principles of Information Science Conference, Okinawa, Japan on 11–12 January 2009. Posted at: `http://arh.pub.ro/gstefan/teraArchitecture.pdf`

[22] ŞTEFAN GH, *Integral Parallel Architecture in System-on-Chip Designs*, The 6th International Workshop on Unique Chips and Systems, Atlanta, GA, USA, December 4, 2010, pp. 3–26.

[23] ŞTEFAN GH, 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 Ciruits, Systems, Communications and Computers, Santorini, July 17–21, 2014, pp. 582–597.

[24] ŞTEFAN GH: The *Connex* Project, posted at: `http://arh.pub.ro/gstefan/Connex.html`.

[25] `http://www.altera.com/products/software/opencl/opencl-index.html#sdk`

[26] `http://www.altera.com/support/examples/opencl/opencl.html`