Functional Virtual Prototyping Environment for a Family of Map-Reduce Embedded Accelerators

Călin Bîră, Mihaela Maliţa, and Gheorghe M. Ştefan

The emergence of the heterogenous computing is based mainly on various forms of parallel accelerators. We present a family of accelerators for embedded computation with a map-reduce architecture based on the partial recursive functions computation model introduced by Stephen Kleene. A three-level virtual prototyping environment is provided to support the development of embedded applications. The first level is written in a Lisp-like functional language. The second is a C-like environment which segregates the intense part of computation from the complex part of computation. The last one is a low level simulator able to provide support for advanced optimizations. The environment is designed for developing applications by tuning the architecture of a family of many-core machines which provide high performance per $Watt$ **and** cm^2**. The energy efficiency of processors backing our architectural approach is in the range of** $10\,pJ/flop$ **evaluated for the standard cell** $28\,nm$ **technology.**

I. Introduction

PARallel accelerators are increasingly utilized in embedded computation for accelerating applications having continuously increasing complexity and intensity. One solution for optimizing the computational resources is to make an appropriate segregation between the *complex* code and the *intense* code. Both, the static size (code size) of the complex code and the dynamic size (execution time) of the intense code are big. A heterogenous engine is the most efficient hardware to deal with both, complexity and intensity in the same time. But, the engines associated with the two kinds of computation are very different, while the program must be written in the same environment. On the other hand, the intense computational code has mainly functional aspects, while the complex computation has mainly control aspects.

It is about a sort of "co-design" which intends to put together the two very different aspects of computation: the intensity and the complexity. The prototyping environment must deal gradually with the segregation between complex and intense. Initially, we require a non-differentiated environment. Subsequently, the design environment must be able to deal differently with the two aspects of computation. Finally, the intense computation sections must be evaluated in detail with an accurate performance model. Thus, results the three levels of our approach.

Călin Bîră is with Dept. of Electronic Devices, Circuits and Architectures, Faculty of Electronics, Telecommunications and Information Technology, Politehnica University of Bucharest, 060042, Romania e-mail: calin.bira@upb.ro

Mihaela Maliţa is with Dept. of Computer Science, Saint Anselm College, Manchester, NH, 03102 USA e-mail: mmalita@anselm.edu

Gheorghe M. Ştefan is with Dept. of Electronic Devices, Circuits and Architectures, Faculty of Electronics, Telecommunications and Information Technology, Politehnica University of Bucharest, 060042, Romania e-mail: gheorghe.stefan@upb.ro

Manuscript received April 19, 2005; revised August 26, 2015.

Additionally, we lack an appropriate model for parallel and heterogenous computation. To overcome this impasse, we attempt a firm theoretical support for parallel computation [10]. Thus, Section II will introduce the Map-Reduce abstract model for parallel computation starting from what we call Kleene Machine. In the same section, based on [5] [7] [8] [9], a hardware solution for a family of heterogenous accelerators is proposed. Section III of the paper describes the three levels of the ***virtual prototyping environment***: the functional Lisp-like level, the segregational C-like level and the optimization level of Verilog-based simulation. Section IV shows a simple example demonstrating the use of our environment.

II. Map-Reduce Abstract Model

Since there is a mathematical and an abstract model for the mono-core computers, the same is mandatory for the multi/many-core parallel computers. What corresponds to the Turing Machine for the multi/many-core computation? Where is the equivalent of the von Neumann or Harvard abstract model for parallel computation? In their absence, *ad hoc* solutions dominate, imposed mainly by the corporate-driven approaches. In [10], starting from [4] and based on [2], a mathematical model is considered and an abstract model is proposed.

A. Kleene Machine

Out of the three rules of Kleene's model, the use of the composition rule is enough to define what we call ***Kleene Machine*** (KM). In [10] we proved that the primitive recursion rule and the minimalization rule are reducible to the application of specific compositions.

Definition 1: By definition, ***Kleene Machine*** (KM) consists of a two-layer construct, associated to the composition rule, see Figure 1, with:

1) ***map*** level: the independent functions h_i, for $i = 0, 1, \ldots$
2) ***reduction*** level: the function g

where h_i, for $i = 0, 1, \ldots$ and g are initial functions or compositions.

◇

Because Kleene's model is equivalent with the Turing Machine model the next corollary is true.

Corollary 1: ***Kleene Machine*** represents a mathematical model for parallel computation with two aspects: the *synchronic* parallelism on the map level and the *diachronic* (pipelined) parallelism between the two structural levels, the map level and the reduce level.

◇

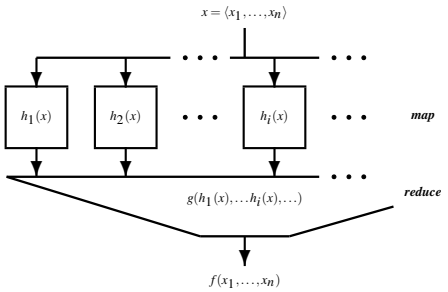

Fig. 1. **Kleene Machine**. *Synchronic* parallelism performed on the **map** level and *diachronic* parallelism between the **map** level and the **reduce** level.

Any function $f : \mathbb{N}^n \to \mathbb{N}$, where $\mathbb{N}$ is the set of positive integers including zero, is computable using the initial functions (*zero*, *inc*, *selection*) and applications of various forms of the composition rule

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

B. Map-Reduce Abstract Recursive Model

Starting from the mathematical model, a way to design actual processors is provided by the abstract model presented in Figure 2, where:

- MAP: is a linear array of cells ***MapRed(i)*** with two components:
  - *eng*: is an execution, processing or computing unit
  - *mem*: is the local memory associated to *eng*

  utilized to define recursively the parallel hardware
- REDUCE: is the reduction unit for functions defined on a set of vectors with values on a set of scalars
- CONTR: is a mono-core computing element utilized to sequence the computing process
- MEMORY: is the external memory of the Map-Reduce engine

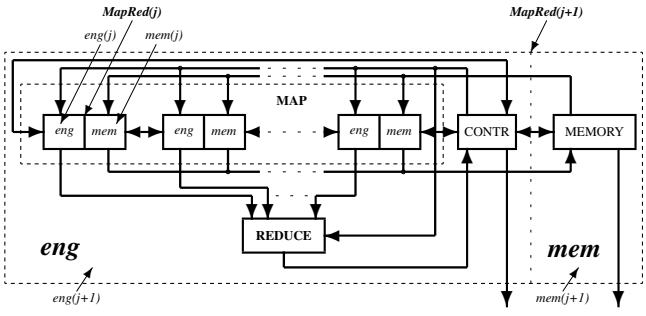

Fig. 2. ***MapReduce recursive abstract model*** for parallel computation.

The MapReduce abstract model allows to compose (to sequence) various KMs and basic functions in order to compute the desired function. The model is recursive with:

- *eng(0)*: a 16 or 32-bit mono-core execution or processing unit
- *mem(0)*: a static RAM of few Kwords
- REDUCE: a *log*-depth pipelined circuit performing functions as *add*, *max*, ...

In this paper we limit our approach to ***MapRed(1)***, an engine implemented as a one-chip solution for ***eng(1)*** and an external DRAM as ***mem(1)***. The first level of the simulation tool is implemented with a solution which allows an approach which consider also ***MapRed(i)*** for $i > 1$.

C. One-chip Implementation

The actual implementation takes into consideration a parameterized family of circuits with features selectable at synthesis. This approach allows us to work in a flexible parameterized and programmed environment.

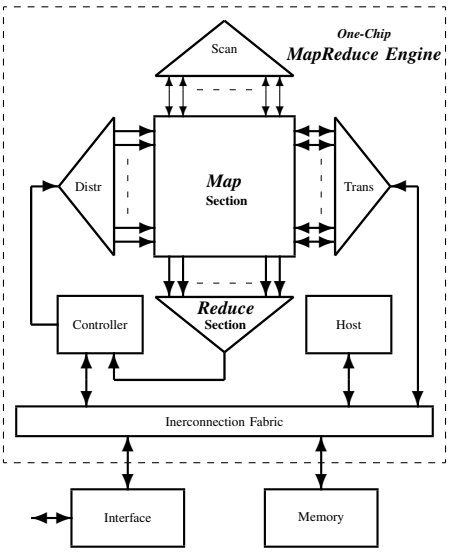

Fig. 3. One-chip implementation for a heterogenous embedded system with map-reduce accelerator. The *complex computation* is performed by Host, while the *intense computation* executes on the map-reduce engine under the control of Controller.

In Figure 3 the block schematic of the family of accelerators is composed out of:

- ***Map*** **Section**: a linear array of p cells, each containing $eng(0) + mem(0)$
- ***Reduce*** **Section**: a reduction network with latency in $O(\log p)$
- Distr: a *log*-depth tree distribution network which sends in each clock cycle from Controller to ***Map*** **Section** an instruction with associated scalar if required
- Trans: a two-dimension network utilized to insert/extract, in transparent mode related to the computational process, data in/from the array
- Scan: a loop closed over the linear array of cells which sends back to each cell global information related with the state of cells
- Controller: a mono-core computing engine which fetch in each clock cycle from its program memory a pair of instructions, one for its use and another to be broadcasted toward the many-core array through Distr

- Host: a mono- or multi-core processor which executes the complex part of the computation and controls the data transfers between the external memory, Memory, and the vector memory consisting of local memories, $mem(0)$, distributed along the cells.

III. Prototyping Environment

We view any new hardware design as having initially a liquid form. The more descriptive we are and the more time we spent on it, the more it solidifies, therefore the less changes it allows, but the closer it is to the true silicon implementation.

The prototyping environment we designed consists of three components listed below:

- High level **Map-Reduce** functional description environment utilized at high level of abstraction. It is able to provide a unified environment for the first level of implementation of the abstract model (***MapRed(1)***), as for any higher level of implementation in the future
- OPcode INjection for Connex-Arm Architecture (**OPINCAA**) programming environment generates executable code starting from a C-like code where the intense part and the complex part of the application are segregated but, in the same time, interleaved is an unified form
- **pRISC** family of parallel engines utilized as a generic parameterized environment actualized or improved according to the results provided by the previous two components.

A. *High Level* **Map-Reduce** *Functional Environment*

Dr.Racket was utilized to develop the highest level of our virtual prototyping environment: the Functional Level Verification. Its pure functional character, inspired from [2], allows a flexible description for both complex and intense computation and, in the same time, allows generating a functional hierarchy which supports our recursive structural definition for parallel computation (see Figure 2).

The user view of the Map-Reduce architecture consists of (1) the content of the external scalar memory, Memory, $S = \langle s_1, s_2, \ldots, s_M \rangle$ and (2) the content of the vector memory consisting of the memory elements, $mem(0)$ distributed in the Map section:

$$\begin{aligned} v_1 &= \langle s_{11}, s_{12}, \ldots, s_{1p} \rangle \\ v_2 &= \langle s_{21}, s_{22}, \ldots, s_{2p} \rangle \\ &\ldots \\ v_m &= \langle s_{m1}, s_{m2}, \ldots, s_{mp} \rangle \end{aligned}$$

We call $v_1, v_2, \ldots, v_p$ horizontal vectors. They are distributed along the cells of the MAP array.

The MAP section contains also two special vectors. The constant vector index: $ix = \langle 1, 2, \ldots, p \rangle$ utilized to identify each cell, and the p-component vector of integer scalars: $active = \langle a_1, a_2, \ldots, a_p \rangle$ utilized to select in each cycle the active cells, i.e., the cells which perform the instruction issued by the controller CONTR. If $a_i = 0$ then the cell i is active, else its state is not affected in the current cycle.

The GUI is organized to show the content of the scalar memory and of vector memory (see Figure 4).

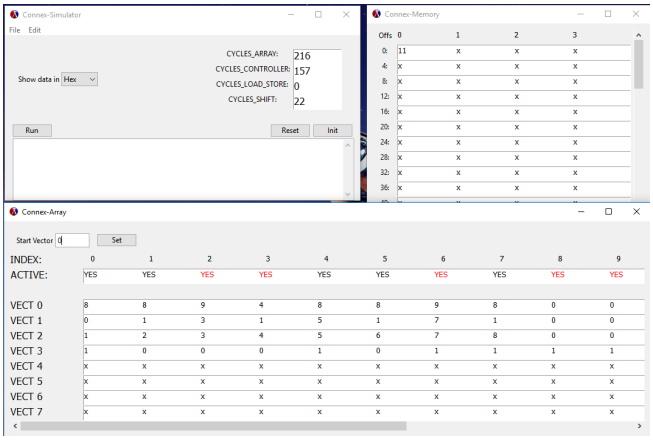

Fig. 4. The graphic interface for MapReduce functional environment.

The API offered to the programmer consists of initialization, transfer, map, reduce, spatial control functions and finally, global functions, as follows:

1) Initialization

`(InitSystem n p m)` : initialize a system with p cells, with n words in each local memory, and an external memory of m words.

`(SetAll aAddr vect)` : the vector `aAddr` takes the value `vect` in all active cells.

Example:

```
> (SetAll 0 #(7 6 5 4 3 2 1 0))
```

`(SetVector aAddr vect)` : the vector `aAddr` takes the value `vect` ***only in the active cells***.

`(SetStream mAddr stream)` : starting from the address `mAddr` in the external memory, Memory, is stored the stream `stream`.

Example:

```
> (SetStream 4 #(15 16 17 18 19 10 11))
```

2) Transfer

`(CopyVector aAddrD aAddrS)` : the vector destination, `aAddrD`, takes the value of the vector source, `aAddrS`

`(StoreVector aAddr mAddr)` : the vector `aAddr` is stored in the external memory starting from the address `mAddr`.

`(LoadVector aAddr mAddr)` : load `aAddr` starting from `mAddr` in Memory.

`(StoreVectorPerm aAddr mAddr indexVect)` : store vector `aAddr` in external memory starting from `mAddr` using the permute vector `indexVect`.

`(LoadVectorPerm aAddr mAddr indexList)` : load vector `aAddr` from `mAddr` using the permute vector `indexList`; see the result in Figure 5.

`(StoreVectorStrided aAddr mAddr burst stride)` : store vector `aAddr`, starting with the address `mAddr`, in bursts of `burst` words, with a stride of `stride` words.

`(LoadVectorStrided aAddr mAddr burst stride)` : load `aAddr`, from `mAddr`, in bursts of `burst`, with a stride of `stride`.

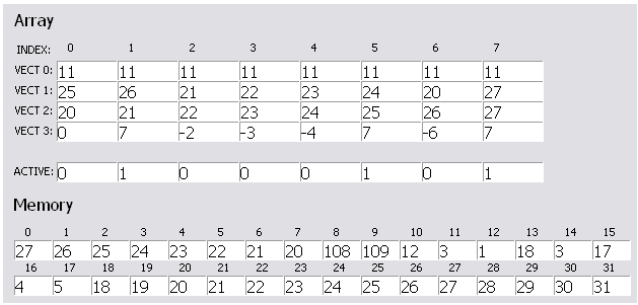

Fig. 5. The load with permute operation: `> (LoadVectorPerm 1 20 #(5 6 1 2 3 4 0 7))`.

`(StoreVectorScatter aAddr burst vAddr)` : store vector `aAddr`, in bursts of `burst` words at the locations specified by the sequence `vAddr`.

Example:

```
> (StoreVectorScatter 3 2 #(10 5 12 3))
```

`(LoadVectorGather aAddr burst vAddr)` : load `aAddr`, in bursts of `burst` words from the addresses `vAddr`.

3) Map Functions

a) Arithmetic & Logic Functions: The generic form of the arithmetic functions are: `(Func x y)`, or `(Func x)`, where `x` and `y` are scalars or vectors. The addition function is defined by: `(Add x y)`.

Example:

If the `ACTIVE = #(0 0 0 0 0 0 0 0)` then

```
> (Add 3 17)
20
> (SetAll 2 #(20 21 22 23 24 25 26 27)
#(20 21 22 23 24 25 26 27)
> (Add 10 (Vec 2))
#(30 31 32 33 34 35 36 37)
> (Add (Vec 2) 10)
#(30 31 32 33 34 35 36 37)
> (Add (Vec 2) #(0 1 2 3 4 5 6 7))
#(20 22 24 26 28 30 32 34)
> (Inc 2)
#(21 22 23 24 25 26 27 28)
```

b) Test Functions: The generic form of the test functions are: `(Cond x y)`, or `(Cond x)`, where `x` and `y` are scalars or vectors. These functions return Booleans or Boolean vectors. The set of test functions contains: `(Eq x y)`, `(Lt x y)`, `(Leq x y)`, ... `(Zero x)`.

4) Reduce Functions

`(RedAdd v)` : returns the sum of the components of the vector `v` from the active cells of the array.

`(RedMax v)` : returns the maximum value of the components of the vector `v` from the active cells of the array.

`(RedMin v)` : returns the minimum value of the components of the vector `v` from the active cells of the array.

5) Spatial Control Functions

`(ResetActive)` : `ACTIVE <= #(0 0 ... 0)`, activates all the cells in the array.

`(SetActive bVect)` : `ACTIVE <= vect`, the Boolean vector `bVect` is utilized to select the active cells in the array.

`(Where bVect)` : in all active cells where the Boolean vector `bVect` takes the value 1, the ACTIVE vector remains unchanged on 0; in all the other cells the ACTIVE vector is incremented.

`(ElseWhere)` : everywhere `ACTIVE < 2` 0 is substituted by 1 and *vice versa*

`(EndWhere)`: everywhere `ACTIVE > 0` is decremented

`(First)` : increments ACTIVE except the first occurrence of 0.

6) Global Functions

`(ShiftLeft many vect scal)` : shift left with `many` positions `vect` and insert at right end `scalar`.

Example:

```
> (SetAll 1 #(1 2 3 4 5 6 7 8))
#(1 2 3 4 5 6 7 8)
> (ShiftLeft 3 1 13)
#(4 5 6 7 8 13 13 13)
```

`(ShiftRight many vect scal)` :

`(RotateLeft many vect)` : rotate `vect many` positions left.

`(RotateRight many vect)` : rotate `vect many` positions right.

`(Permute v1 v2)` : permute `v1` according to `v2`.

`(FirstIndex)` : returns the index of the first active cell.

B. OPINCAA: *Segregating Complex from Intense*

The OPINCAA [3] (OPcode INjection for Connex-ARM Architecture) programming environment allows the programmer to write software for the Map Reduce accelerator, to run, to debug and to optimize it. Using a C-like syntax, the complex code is segregated from the intense code, but they are subsequently integrated in the same stream of code in order to generate the executable code. The complex part of code is executed on HOST while the intense part of code executes on the Map-Reduce section.

The resulting programming environment facilitates programming and debugging code for the accelerator (remote step-by-step debug using gdb), and simulation on better performing hosts machines. It also contains software for visualizing key components of the computation process (see Figure 6); this can be utilized to optimize the accelerated code.

The C part of the code is also utilized to control the data transfers between the external memory and the internal distributed vector memory. The setup utilized for experimenting consists of a Xilinx Zynq platform (dual-core ARM as HOST and 128-core vector machine defined in the associated FPGA)

The following code performs a data transfer test from HOST to the Map section, then executes a computation in Map section, to verify a basic functioning of the system. In the API utilized in the following example:

`writeDataToArray(src, vectors, offset)` : starts a HOST to Map data transfer from HOST's src memory address, size of vectors *vectors*, starting with vector $offset$.

`executeKernel(name)` : starts the computation called `name` on Map section.

`readReduction()` : starts the Reduce process and retrieves result.

Listing 1. OPINCAA example

```
static int testIowrite(ConnexMachine *m,
                       int KerNum,
                       int numVect,
                       int Param2)
{
  int index;
  for(int ct=0; ct<MACHINES*numVect; ct++)
    data[ct] = ct;
  m->writeDataToArray(data, numVect, Param2);
  for(index=0; index < numVect; index++)
   for (int ct = 0; ct < MACHINES; ct++)
    {
      _BEGIN_KERNEL(KerNum);
        EXECUTE_IN_ALL(R1 = INDEX;
                       R2 = ct;
                       R3 = 0;
                       R4 = (R1 == R2);
                       NOP;
                      )
        EXECUTE_WHERE_EQ
          (R3=LS[index+Param2];)
        EXECUTE_IN_ALL ( REDUCE(R3);)
      _END_KERNEL(KerNum);

      m->executeKernel(to_string(KerNum));
      int result = m->readReduction();

      if(data[index*MACHINES+ct) != result)
          return FAIL;
     }
      return PASS;
}
```

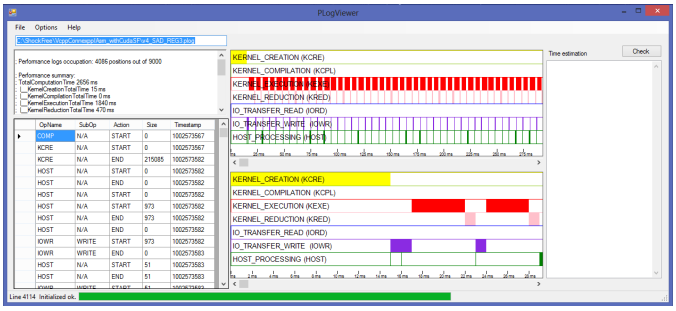

Fig. 6. The graphic interface for the performance analysis in OPINCAA functional environment. All computation and all data transfers are listed and graphically displayed separately to form a true and real perspective of the executed application

C. pRISC Family: *Estimation in Early Stage of Development*

The most accurate evaluation is provided by the third component: a parallel RISC family of processors (pRISC) described and simulated in Verilog using the VIVADO design suite. The pRISC family is defined using a parameterized behavioral description in Verilog with many optional features conditionally generated. The content of the program memory is provided by a two-pass assembler written in Verilog. The source code is written in an assembly language with a pair of instructions per line: one for Controller and another to be broadcasted to the Map section using Distr *log*-depth network (see Figure 3).

Around behavioral description we designed a test bench which includes the assembler code generator. A controlled cycle counter provides a cycle accurate measure of the time performance.

IV. Using the Prototyping Environment

The prototyping environment is utilized to investigate solutions at different levels and for different purposes. The first level supports mainly algorithmic developments, while the third level is involved in advanced optimizations. The OPINCAA environment is very efficient in partitioning and generating the two kinds of codes, segments of code executed on the cellular array and segments of code executed on the mono or multi-core controller. The data streams between the external scalar memory, Memory, and the internal vector memory distributed along the cells are efficiently designed in the OPINCAA environment.

Example 1: Let us take the simple example of the vector-matrix product. Only the first and the third level of virtual prototyping are involved (the second is not involved here because the application is purely intensive and the transfer issues are not addressed). The high level functional program is:

Listing 2. Lisp code

```
/*****************************************
Multiply _vector with _matrix [v1 v2 v3 v4]
Example: if
_vector:  (Vec aAddrV)    = (1 1 1 1)
_matrix: [(Vec aAddrM+0) = (1 1 1 1)
          (Vec aAddrM+1) = (2 2 2 2)
          (Vec aAddrM+2) = (3 3 3 3)
          (Vec aAddrM+3) = (4 4 4 4)]
_result:  (Vec aAddrRes) = (4 8 12 16)
*****************************************/
(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 assembly code associated to the previous program running on the pRISC family processors is:

```
/*****************************************
Assembly code for Matrix-Vector Multiplication

The 2-line loop (labeled 6) performs:
  - RILOAD(127): load line from local memories
  - MULT(0): multiplication, in map section
  - cCPUSHL(1): reduction add, with result
    in the global shift register
  - cBRNZDEC(6): test, decrement and branch
*****************************************/
```

```
        cSEND(6);       CADDRLD;
        cLOAD(0);       RLOAD(0);
        cVSUB(1);       MULT(0);
      // LOOP
LB(6);  cCPUSHL(1);     RILOAD(127);
        cBRNZDEC(6);    MULT(0);
      // END 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 $log\,P$ is due to the latency introduced by the distribution and reduction networks. For example, if $N = P = 1024$, the vector-matrix product is computed using 2.012 cycles per scalar component in the result vector.

◇

The code generated in the third level of simulation can be executed also in the OPINCAA simulator (in this regard the two simulators are interchangeable).

The current use of this environment is in the domains pointed by the Berkeley *dwarfs* [1], such as: *N*-body problem, with application in *molecular dynamics*, *convolutional neural networks* with applications in automotive, graph theory.

V. Conclusion

The virtual prototyping environment we built has three levels, each corresponding to a mandatory stage in designing and implementing (embedded) heterogenous accelerators: the algorithmic level with its dominating functional aspect, the segregated (between intense and complex) code generation, the optimization of the intense code.

On each level the appropriate type of language is utilized: *Lisp*-like for the functional level, *C*-like for providing the two type of code, and *Verilog HDL* for optimization of the accelerator part of the hybrheterogenous engine.

The three-level approach is imposed by the emergence of the absolute novelty of the many-core parallel accelerators in the context of the way of thinking too oppressively dominated by the mono- and multi-core computation. The "sequential" approach is not the starting point for parallel programs. Parallel programs result from new parallel algorithms, designed starting from functional definitions, not from parallelizing "sequential" programs. This is the main reason for the first level in our approach.

There is also another reason for the first, functional level, of our prototyping environment: the recursive definition of the abstract model ***MapRed(i)*** (see Figure 2) is supported for any $i > 1$ by our High Level **Map-Reduce** Functional Environment. The future work is related with the way our environment can be utilized for many-chip, many-board, ... many-cabinet approach of parallel accelerators.

Acknowledgment

The authors got a lot of support in different stages of the development of the heterogenous MapReduce architecture from: Frank Ho, Radu Hobincu, Bogdan Mîţu, Lucian Petrică, Marius Stoian, Dominique Thiebaut and Dan Tomescu.

References

[1] Krste Asanovic, *et al.*, The landscape of parallel computing research: A view from Berkeley, 2006. At: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.

[2] John Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM* 21, 8 (August) 1978. 613-641.

[3] Călin Bîră, R. Hobincu, Lucian Petrică, "OPINCAA: A Light-Weight and Flexible Programming Environment For Parallel SIMD Accelerators" *Romanian Journal of Information Science and Technology*, Volume 16, Numbers 4, 2013, 336-350

[4] Stephen Kleene, General recursive functions of natural numbers. *Mathematische Annalen* 112, 5, 1936. 727-742.

[5] Mihaela Maliţa, Gheorghe M. Ştefan, Dominique Thiébaut, Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation. *ACM SIGARCH Computer Architecture News*, Vol. 35, No. 5, December 2007. 32-39.

[6] Mihaela Maliţa, and Gheorghe M. Ştefan, Backus language for functional nano-devices. *CAS 2011, vol. 2*, 331-334.

[7] Gheorghe M. Ştefan, *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.

[8] Gheorghe M. Ştefan, One-chip TeraArchitecture. *Proceedings of the 8th Applications and Principles of Information Science Conference*. Okinawa, Japan, 2009.

[9] Gheorghe M. Ştefan, Integral parallel architecture in system-on-chip designs. *The 6th International Workshop on Unique Chips and Systems*, Atlanta, GA, USA, December 4, 2010, pp. 23-26.

[10] Gheorghe M. Ştefan, Mihaela Maliţa, 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, 582-597.

Călin Bîră teaches digital design applications in *Politehnica* University of Bucharest. His scientific interests are focused on embedded systems and parallel computation. More at `http://www.dcae.pub.ro/en/membri/4/bira_calin`.

Mihaela Maliţa teaches computer science at Saint Anselm College, US. Her interests are programming languages, computer graphics, and parallel algorithms. She wrote and tested different simulators for the Connex parallel chip. More at `http://www.anselm.edu/mmalita`.

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