Molecular Dynamics on FPGA Based Accelerated Processing Units

Mihaela Maliţa[1,3,⋆], *David* Mihăiţă[2,⋆⋆], and *Gheorghe M.* Ştefan[3,⋆⋆⋆]

[1] *Computer Science Dept., Anselm College, Manchester, NH*
[2] *Faculty of Electronics, Tc. & IT, Politehnica University of Bucharest, Bucharest, Romania*
[3] *Faculty of Electronics, Tc. & IT, Politehnica University of Bucharest, Bucharest, Romania*

Abstract. One of the main problems in providing the amount of computation requested by the Molecular Dynamic domain is to offer an appropriate architectural environment for solving ***all*** the aspects of the intense parts of the involved computation. Current solutions accelerate only partially the intense computation – forces computation & position and speed updates, which represents around 75% from the total computational effort – thus limiting the help provided by the parallel computing resources involved. The aim of this paper is to introduce a parallel accelerator featured with functions able to add to the accelerated functions the neighbourhood list building, which represent around 25% from the total computation. Thus, accelerations higher than the current $\sim 4\times$ are expected. Our proposal, the ***MapReduce Accelerator***, is evaluated using the Gromacs system. The Martini water example, running on a cycle accurate simulator, is used to evaluate the speed-up and the energy.

1 Introduction

The Molecular Dynamics computational domain belongs to the 4-th dwarf – *N-Body Methods* – emphasized in the Berkeley's View [2]. A short description of the problem considers N particles, p_i for $i = 1,\dots N$, each characterized by a small set of parameters, let say, for example, position, speed, charge and mass ($p_i : (x_i, y_i, z_i, v_x^i, v_y^i, v_z^i, q_i, m_i)$). Each simulation cycle, in the brut force approach, consists of the following two steps:

1. for each p_i is computed the force vector resulting from the forces exerted by all the other $N-1$ particles (in time belonging to $O(N^2)$)
2. the particles are let free to change their state (position and speed) for a very short time interval; the shorter the interval the more accurate the simulation is (in time belonging to $O(N)$).

While, for N bodies, the brute force approach leads to $O(N^2)$ computation for each simulation cycle, applying various strategies, the computational complexity are reducible to $O(NlogN)$ or even $O(N)$. For the particular case of Molecular Dynamics, the computational complexity is given by the huge number of simulation cycles requested for an accurate simulation of the molecular folding process. Thus, any optimization applied to the first step of the simulation cycle is welcome. There are various ways to reduce the number of forces considered as acting on each particle, such as *Barnes-Hut Algorithm*, *Fast Multipole Method* or *Particle Mesh Method.* We selected, for investigating how the Molecular Dynamics can be accelerated using hybrid computation, the Gromacs[1] system whose main idea is to limit the interactions of each particle p_i to a small neighbourhood of $M << N$ particles whose actions on the particle p_i worth taking into consideration. Thus, a three-step approach is currently practiced:

1. neighborhood search (in time belonging to $O(N^2)$)
2. forces computation (in time belonging to $O(N \times M)$)
3. up-date (in time belonging to $O(N)$)

There are many solutions for accelerating the Molecular Dynamics applications. The most radical one is to build ASICs based specialised engines, like the *Anton* machine build by D. E. Shaw Research [10]. At the other end, meaningful accelerations are also obtained by using the last Intel's families of multi cores processors supported by SSE[2] units.

A good compromise performance vs. price is provided by using hybrid computation solutions embodied in Accelerated Processing Units (APU). An APU integrate in the same system a PU with an accelerating parallel engine, such as a GPU[3], a MIC[4] or, increasingly more frequently, a FPGA. The small accelerations provided by the use of the current parallel accelerators are mainly due to the fact that only the last two of the previously listed steps are submitted to the parallel accelerator, while the first step – the neighbourhood search – which is theoretically the most

⋆e-mail: mmalita@anselm.edu
⋆⋆e-mail: David.Mihaita@infineon.com
⋆⋆⋆e-mail: gheorghe.stefan@upb.ro

[1]GROningen MAchine for Chemical Simulations, see http://www.gromacs.org/
[2]Streaming SIMD Extensions
[3]Graphic Processing Units such as Nvidia or ATI products.
[4]Many Integrated Core such as Intel's Xeon Phi family of processors

time consuming, is left to the PU. Usually, the weight of the first step is minimized by the compromising solution of applying it only once at 10-50 state up-dates.

Our proposal refers to an APU based on a parallel accelerator having a Map-Reduce architecture, implemented in FPGA, able to perform efficiently on *all* the previously emphasized three steps.

The next section discuss the state of the art. In the third section is a short description of the proposed *mapReduce* accelerator. The fourth section describe the use of the accelerator in running the Gromacs system, a widely used Molecular Dynamics environment. Final comments concludes the paper.

2 State of the Art

2.1 Multi-Core Approach

Current CPU are already parallel machines. Intel's processors, for example, are multi-core engines featured with Streaming SIMD Extensions (SSE) accelerators. We used an Intel i5 system for running Gromacs for Martini water [8]. Table 1 shows us the effects of this kind of two-level parallelism: multi cores, each accelerated by an SSE.

Table 1. The performance on i5 multi-core [8].

CPU type	**Performance** [μ/day]	Acceleration
1-Core, no SSE: 1 EU	5.84	1.00×
1-Core, with SSE: 1+4 EUs	9.78	1.67×
4-Core, no SSE: 4 EUs	18.94	3.24×
4-Core, with SSE: 4+16 EUs	31.48	5.39×

It looks like the SSE accelerator (with its 4 execution units, EU) is not of great use. The multi-core aspect is more useful. The overall acceleration, 5.39×, is higher than the acceleration provided by the hundred of EUs of a GPU or the 60 cores of a MIC, because all stages of the Gromacs computation are submitted to the acceleration. But, from architectural point of view, the acceleration provided by $p = 20$ EUs is too small, $0.27 \times p$.

2.2 Hybrid Solutions

In this paper we deal with systems, distributed or not, based on APU, but we consider only the task executed by one APU. We discuss and provide a hybrid solution for accelerating the Gromacs system.

The current hybrid solutions [1] [6], using GPUs or MICs to accelerate a CPU, develop an algorithm (see Figure 1) containing the following sections:

1. Build the neighbor lists; runs on CPU
2. Compute forces for the bonded interactions and update; runs on CPU
3. Compute forces for the non-bonded interactions and update; runs on GPU or MIC
4. Integrate the results; runs on CPU

The section 1 of the algorithm runs only once at 10-50 updates of the state. The Gromacs team decided experimentally that each update of the positions provides too small chances of positions to be considered in redefining the neighbourhood of each particle. Only the section 3 of the algorithm is sent to the accelerator. Because the force computation and update computation represent $\sim 75\%$ form the total computation [8], the acceleration provided by a hybrid system is theoretically limited by the Amdahl rule to $\sim 4\times$.

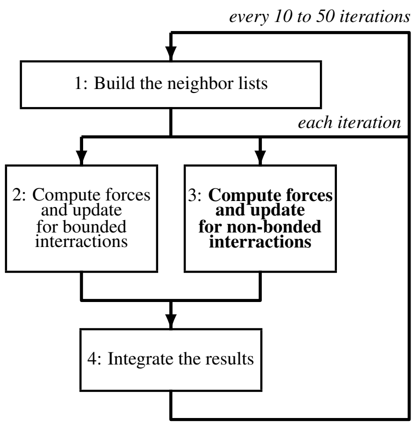
every 10 to 50 iterations
1: Build the neighbor lists
each iteration
2: Compute forces and update for bounded interractions
3: Compute forces and update for non-bonded interractions
4: Integrate the results

Figure 1. The hybrid algorithm. Only the section 3 is performed by the accelerator part of the hybrid system.

The current performances published for the hybrid systems are the following:

- using GPU as accelerator no more than 3× acceleration with GPUs equipped with hundred of cores [6] [4] [1]
- using the Intel Xeon Phi MIC as accelerator 1.8× acceleration is provided [9] [1]

The performances reported for the hybrid systems based on GPU and MIC are limited mainly due to the weak involvement of the accelerator part of the hybrid system in all the sections of the computation represented in Figure 1. We consider that there are more computation stages appropriated for parallel treatment. Therefore, we propose that, at least, **aspects related with the building of the neighbor list can be reconsidered in order to allow the accelerator to contribute more to the overall computation**. The architecture of the accelerator we propose is featured with mechanisms able to compute efficiently both, the neighbor list and the updates under the forces acting in the identified neighbourhood.

3 APU Based on the *mapReduce* Accelerator

3.1 The System

Figure 2 shows the block schematic of our proposal: an APU designed as a hybrid system where ACCELERATOR is based on the cellular system ***mapReduce*** [7] [11] [12] [13]. It consists (in the initial FPGA implementation) of:

- HOST: a general purpose computing system with its CPU, MEMORY, I/O system
- INTERFACE: a DMA (direct memory access) unit for data and program transfer between MEMORY and the internally distributed memories in ACCELERATOR
- ACCELERATOR: the ***mapReduce*** system with its:
  - MAP section: an array of p cells, C_i for $i = 1, 2, \ldots, p$, each containing:
    - *eng*: a 32-bit execution unit
    - *mem*: a 1KW of 32-bit words data memory
  - REDUCE: a log-depth circuit performing vector-to-scalar reduction operations (addition, maximum, ...)
  - CONTROL: a special cell containing
    - *eng*: a 32-bit execution unit (identic with those from the MAP section)
    - *mem*: a 1KW of 32-bit words data memory
    - *prog mem*: a 4KW of 32-bit words memory used to store the program executed by the accelerator

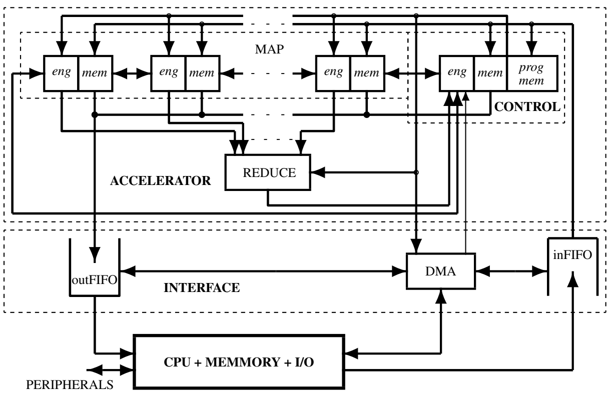

Figure 2. APU as a Hybrid Computing System based on the *mapReduce* Accelerator.

INTERFACE, connected between ACCELERATOR and the HOST is controlled by a DMA unit and the transfer is performed through the two FIFOs. DMA loads the program in CONTROL section and transfers data between ACCELERATOR's distributed memory along the cells and MEMORY. In each clock cycle the MAP section receives an instruction from CONTROL. There are two operating modes:

- slave mode: CPU loads the program(s) in *prog mem*, starts running the program(s) and controls the data transfer between ACCELERATOR and MEMORY
- autonomous mode: CPU loads the program(s) and starts running the program, but now the data transfer is completely under the control of ACCELERATOR.

3.2 Implementation

In this initial stage of the project, the ***mapReduce*** accelerator is implemented using the FPGA technology. For an advanced stage of the project, we are based on previous implementations of the mapReduce accelerator [11] and on evaluations for the 32*nm* technology. The FPGA technology evolved from a pure programmable structure to a mixed solution putting together frequently used ASIC blocks with the in field programmable blocks used mainly for interconnections. Thus, besides the usual Configurable Logic Blocks, the designers have access to memory blocks, DSP units, advanced micro-controllers (for example: Dual-ARM Cortex 9), and a lot of standard interfaces. Such FPGAs could be a wonderful support for a hybrid computing system centered on the micro-controller and based on the mapReduce accelerator developed on the programmable part.

3.3 The Architecture

Because each cell has its m-word memory, the data in the p-cell MAP section is represented as an array of m p-scalar full ***horizontal vectors*** as follows:

```
v[1] = <s[11], s[12], ..., s[1p]>
v[2] = <s[21], s[22], ..., s[2p]>
...
v[m] = <s[m1], s[m2], ..., s[mp]>
```

where: each “column":

```
w[j] = <s[1j], s[2j], ..., s[mj]>
```

is the full ***vertical vector*** of scalars associated to cell j, for $j = 1, 2, \ldots p$. We call *vector memory* this distributed memory in the MAP array. Distributed along the array of cells there are also the following vectors:

```
IX = <1,    2,    ..., p   >
B  = <b[1], b[2], ..., b[p]>
A  = <a[1], a[2], ..., a[p]>
```

Vector `IX` is the constant *index vector* (used to identify each cell), while `B` is a *Boolean vector*, used to enable or disable every cell ($b_i = 1$ the cell C_i executes the current instruction). Vector `A` is the *accumulator vector* distributed along the cells. CONTROL section has two special registers: `a`, the accumulator, and `pc`, the program counter.

Each cell, C_i, in MAP and CONTROL executes instructions from a similar ISA. The ISA is accumulator centered in this initial stage of the project. of The main difference is related with the control instructions. In the MAP section there are executed *spatial control* instructions (the cells are enabled or disabled acting on the content of the vector `B`, at the clock cycle level, according to locally tested conditions). In CONTROL the program counter `pc` ensures the *temporal control*. Thus, the ACCELERATOR's ISA is the Cartesian product of two ISAs:

$$ISA = (cISA \times aISA)$$

Besides the usual arithmetic & logic and control instructions, the controller's instruction set, *cISA*, contains specific instructions related with the REDUCE unit:

```
cCLOAD(0):
 a <= reduceSum = a[0]*b[0]+ ... +a[p]*b[p]
cCLOAD(1):
 a <= reduceMax = max(a[0]*b[0], ...,a[p]*b[p])
cCLOAD(2):
 a <= reduceBool = b[0] | ... | b[p]
cCADD(0):
 a <= a+reduceSum
```

where the output of the REDUCE section is the co-operand provided by the ARRAY.

Here are few samples from aISA, which are executed only in the cells where `b[i] = 1`:

```
ADD(val):  a[i] <= b[i] ? a[i]+mem[val] : a[i]
VADD(val): a[i] <= b[i] ? a[i]+val : a[i]
CADD:      a[i] <= b[i] ? a[i]+a : a[i]
MULT(val): a[i] <= b[i] ? a[i]*mem[val] : a[i]
WHERE(cd): b[i] <= cd ? 1 : 0
ENDWHERE:  b[i] <= 1
```

where, `a` is the co-operand, i.e., the accumulator of the CONTROL section.

The program is organized in pairs of instructions, one for CONTROL prefixed with `c`, and one for the cells in MAP. Here is an example of the assembler code which performs the scalar multiplication of the index vector with itself delivering the result in the controller's accumulator:

```
        cNOP;              ENDWHERE; // b[i] <= 1 overall
        cNOP;              IXLOAD;   // a[i] <= i
        cNOP;              IXMULT;
 // a[i] <= acc[i]*i
        cVLOAD(latency);   NOP;
 // a <= latency size
LB(1)   cBRNZDEC(1);       NOP;
 // wait for latency
        cCLOAD(0);         NOP;      // a <= reduceSum
```

The left column of instructions are executed by the CONTROL unit, while the right column instructions are executed in each active cell. The first line activates all the cells in the array, the second load the value of the index vector in accumulator, then the content of the accumulator is multiplied in the accumulator with the value of the index vector. The next two instructions delays the load of the accumulator `a` with the reduction sum provided by the REDUCE unit with a latency in $O(log\,p)$. The last line loads in the accumulator of the CONTROL unit the value of the reduction sum computed from the content of the accumulator vector `A`.

4 Gromacs on *mapReduce* based APU

The Gromacs simulations are defined for two cases: (1) in the simulated space, a cellular periodic structure of identic three-dimension boxes is defined, and (2) in the simulated space, the cellular approach considers a number of different boxes. In this paper the first case is considered.

The main difference from the actual algorithms consists in the fact that the parallel approach is considered for both, neighborhood selection and upgrade computation.

Each particle is represented by a *particle vector* of the form: `<x, y, z, vx, vy, vz, t, s>` where: `x, y, z` are spatial coordinates, `vx, vy, vz` are the three components of the velocity, `t` is a parameter and `s` is an integer. Initially, in MEMORY there is a string of unstructured particle vectors.

4.1 The Case of Periodic Cell Structure in the Simulation Space

In the Gromacs molecular dynamic simulation, periodic boxes of particles are considered for bulk glasses, liquids, crystals or mixtures. The computation considers only one box for which periodic boundary conditions are computed.

4.1.1 The Main Tasks

There are four tasks to be defined for describing the algorithm. For the sake of simplicity, we consider the number of particles equal with the number, p, of cells, C_i, in the MAP array.

LOAD task

It loads, from MEMORY, the vector memory of the accelerator with 8 horizontal vectors, each of $p/8$ particle vectors. Then, the $p/8$ 8×8 resulting matrices are transposed. Thus, each of the p cells, in the ***mapReduce*** accelerator, gets a particle vector as a partial 8-element vertical vector:

```
pw[j] = <x, y, z, vx, vy, vz, t, s>
```

for $j = 1,2,\ldots,p$. This task is performed in time $O(p)$.

SEARCH task

It builds in each cell C_i the particle list of neighbourhood, `pl[i]`, for the associated particle `pw[i]`, as follows:

```
/******************************************
task name: SEARCH

r[ij]: distance between particles i and j
rc: cutoff radius defining the neighbourhood
pl[i]: neighbourhood list of pw[i]
*******************************************/
    for (i=1; i<p+1; i=i+1) begin
        transfer pw[i] in CONTROL;
        doInParallel compute r[ij];
        where (r[ij] =< rc)
            add pw[i] to pl[i];
        endwhere
    end
```

At the end of this process, in each cell C_i, besides its particle vector `pw[i]`, there is stored a particle list `pl[i]` of various lengths, for example:

```
w[1]   = < pw[1], pl[1]> =
         < pw[1], <pw[6], pw[20], pw[131]>>
w[2]   = < pw[2], pl[2]> =
         < pw[2], <pw[3], pw[32],..., pw[50]>>
...
w[46]  = < pw[46], pl[46]> =
         < pw[46], <pw[4], pw[50], pw[100]>>
...
w[p] = < pw[p], pl[p]> =
       < pw[p], <pw[65],..., pw[315]>>
```

where, for example, the neighbourhood of particle 46 is: `pl[46] = <pw[4], pw[50], pw[100]>`.

The execution time of this task is in $O(p)$, because the search of cells where $r[ij] \leq rc$ is performed in $O(1)$ and the update of the neighbourhood list, `pl[i]`, in all the active cells is performed also in constant time. The number of particle vectors in `pl[i]` is no bigger than $q < p$.

UPDATE task

It consists, mainly, in two stages. In the first stage, the program computes the force exercised by the particles from `pl[i]` on the particle `pw[i]`. Then, in the second stage, for a very short period of time, the force is applied on the particle and its new particle vector `pw[i]` is computed. The execution time for this task is in $O(p \times q)$.

STORE task

It is performed on the result of the last update on 8 horizontal vectors containing p vertical 8-component partial vectors. First, the $p/8$ 8×8 matrices, contained in the 8 horizontal vectors, are transposed. Then the resulting 8 horizontal vectors, each of $p/8$ vectors, are stored in MEMORY.

4.1.2 The Algorithm

The algorithm which uses the previously defined tasks is:

```
LOAD
loop (how many times are needed)
    SEARCH
    loop (L times)
        UPDATE
    repeat
repeat
STORE
```

The main loop is repeated so many times how may times the application requires, while the inner loop runs a number of cycles, L, determined by the accuracy imposed to the simulation. The Gromacs team recommends $L \in [10, 50]$ updates of the positions at one neighbour search. The execution time of the algorithm is in $O(p \times q)$.

4.1.3 Evaluation

Using, in the Vivado environment, our *mapReduce* accelerator instantiated for $p = 400$, the Gromacs molecular dynamic simulation for the Martini water was done. The data, corresponding to 400 particles, are loaded and *all* the stages of simulation are computed on the parallel accelerator.

At the end of the SEARCH task, in each cell C_i, besides its particle vector `pw[i]`, there is stored a particle list `pl[i]` of various lengths, containing from 0 to 60 particle vectors, so as in each cell are stored no more than $61 \times 8 = 480$ scalars. The mean value is 41×8.

The degree of parallelism achieved for each main stage of the computation are presented in Table 2. The figures

Table 2. Degree of parallelism for running Gromacs for Martini water on the *mapReduce* accelerator [8].

Simulation part	Cycles	Percentage	Degree of Parallelism
Box periodicity	80	0.24%	66.8%
Neighbour search	26410	79.93%	79.2%
Force computation	6409	19.40%	60.4%
Thermostat	91	0.28%	51.0%
Update	50	0.14%	100%
Mean degree of parallelism ε	33040	100%	**75.6%**

are provided from the simulation, programmed in assembly language for FPGA implementation. The architecture performs pretty good for the neighbour search, the stage we added to be submitted to the parallel accelerator. Forces are computed in parallel with a smaller degree of parallelism because the mean length of the neighbourhood list is 40 while the maximal length is 60. the architecture are presented in Table 2 and Table 3. Because the neighbour search is performed once for 10 updates, its weight in the full simulation is diminished, and consequently, the overall degree of parallelism we measured in simulation is only $\varepsilon = 0.64$, instead of $\varepsilon = 0.756$.

Table 3 shows the performance of the two extreme solutions, CPU and specific circuit implemented as ASIC, in order to pe compared with our architectural solution in two implementations, FPGA and ASIC. The computation performance is expressed in microseconds of dynamic simulation performed per one day of computation, $\mu s/day$, while the energy use is expressed in Watts-hours per microseconds of dynamic simulation, $Wh/\mu s$.

Table 3. Comparing *mapReduce* based APU solution with PU and ASIC solutions [8].

Machine	Technology	Freq	$\mu s/day$	$Wh/\mu s$
Intel I5: one core no SSE	22nm	2.7 GHz	5.84	267.1
Intel I5: four cores & SSE	22nm	2.7 GHz	31.48	72.4
Anton	ASIC 65nm	0.4 GHz	572.32	3.1
mapReduce	FPGA 28nm	0.5 GHz	187.01	3.5
mapReduce	ASIC 22nm	2.7 GHz	1010.34	0.3

Theoretically, the acceleration for p execution cells is:

$$\alpha = p \times \frac{f_{CK\_Intel}}{f_{CK\_FPGA}} \times \varepsilon \times \frac{1}{CPI}$$

With of CPI[5] of 1.5, because the cells execute floating point operations in few clock cycles (to keep *eng small & simple*), we compute $\alpha = 31.6$. The acceleration measured in simulation for the FPGA solution, with 400 execution cells, is $\alpha = 32$ compared with one Intel core without SSE, i.e., one execution cell.

For the ASIC implementation of the *mapReduce* accelerator (with $f_{CK\_ASIC} = 2.7MHz$), the acceleration is

$$\alpha_A = 0.426p = 170.66$$

Let's call this acceleration, α_A, the *architectural acceleration*, i.e., acceleration which compares engines working

[5]CPI stands for Cycles Per Instruction.

at the same frequency, thus emphasizing the effects of the architectural design decisions. Indeed, for our case, the acceleration is limited because of $\varepsilon < 1$ (limited by the problem and by our ability to provide an efficient algorithm) and because we decided to execute floating point operations with a sequence of specific instructions.

Compared with a 4-core Intel PU with SSE, i.e., 4+16 execution units, the FPGA acceleration is 5.94×. In the section 2.2 we provided performances published for GPU and MIC based hybrid solutions. Our FPGA hybrid system performs at least 2× better than GPU, which provides 3×, or MIC which provides 1.8× acceleration. Do not mention the power of hundred of Watts consumed by GPU or MIC accelerators, compared with tenth of Watts of our FPGA solution.

The comparison with the *Anton* ASIC looks not too bad. Our FPGA implemented programmable solution has performances in the same range with this specialized circuit which is only 3× more efficient.

5 Final Remarks

The degree of parallelism achieved by the *mapReduce* accelerator in running Gromacs is 60% for force computation and 80% for neighbour search. The overall degree of parallelism is 64%. The current hybrid solutions accelerate only the force computation. Our solution brings a substantial improvement, because it involves the accelerator in the neighbour search also, for which its degree of parallelism is higher than for force computation.

We defined the *architectural acceleration* of an APU as the acceleration provided by an APU with both, one-core PU and the accelerator implemented in the same technology and running at the same frequency. Thus, the architectural acceleration of our proposal is $0.43p\times$. The acceleration is limited only by the degree of parallelism and by the weight of floating point operations, both specific to the application.

The acceleration provided by our *mapReduce* accelerator implemented in 28nm FPGA, related to the performance of an 22nm 4-core Intel CPU with SSE, is 6×. The simulation estimates, for an ASIC implementation of the ***mapReduce*** accelerator, 30× improvement.

The energy efficiency of the *mapReduce* accelerator is improved 20× for the FPGA version, and 300×, for ASIC version.

A promising compromise between FPGA and ASIC in implementing ***mapReduce*** accelerators for Gromacs system is the eASIC technology.

Acknowledgments

The authors got meaningful support from Nicolae Goga, in understanding important details about the Gromacs system, and form Lucian Petrică an exigent reader of the intermediary forms of the text.

References

[1] S. Alam, U. Varetto, *GROMACS on Hybrid CPU-GPU and CPU-MIC Clusters: Preliminary Porting Experiences, Results and Next Steps*, At: http://www.prace-ri.eu/IMG/pdf/wp120.pdf

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

[3] Călin Bîra, *Programming environment for parallel accelerators*, PhD Thesis, UPB, ETTI, Department of Electronic Devices, Circuits and Architectures (2013) http://arh.pub.ro/papers/LucrareDoctorat_v9d.pdf

[4] L. Jianguo, *Running GROMACS on GPUs: a Benchmark Study* (2014) At: https://www.acrc.a-star.edu.sg/docs/ASTAR

[5] S. C. Kleene, "General Recursive Functions of Natural Numbers", in *Math. Ann.*, 112, (1936)

[6] E. Lindahl, *Molecular Simulation with GROMACS on CUDA GPUs* (2013) At: http://on-demand.gputechconf.com/gtc/2013/webinar/gromacs-kepler-gpus-gtc-express-webinar.pdf

[7] M. Malita, G. Ştefan, D. Thiébaut, "Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation" *ACM SIGARCH Computer Architecture News*, Volume 35 , Issue 5, pp. 32-38 (2007)

[8] D. Mihăiţă, *N-Body Problem. Application, on a Map-Reduce Accelerator, to Molecular Dynamics*, Master Thesis, Politehnica University of Bucharest, (2016)

[9] M. Plotnikov, *GROMACS for Intel Xeon Phi^{T}M Coprocessor* At: https://software.intel.com/en-us/articles/gromacs-for-intel-xeon-phi-coprocessor

[10] D. E. Shaw, *et al.*, "Anton, A Special-Purpose Machine for Molecular Dynamics Simulation". *Communications of the ACM*. 51 (7): 91–97 (2008)

[11] G. Ştefan, *et al.*, "The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing", in *Hot Chips: A Symposium on High Performance Chips*, Memorial Auditorium, Stanford University (2006) At: https://youtu.be/HMLT4EpKBAw 35:00

[12] G. Ştefan, "One-chip TeraArchitecture", *Proceedings of the 8th Applications and Principles of Information Science Conference*. Okinawa (2009) http://arh.pub.ro/gstefan/teraArchitecture.pdf

[13] G. Ştefan, M. 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, 582-597 (2014)