Parallel Dedicated Hardware Devices for Heterogeneous Computations

Alessandro Marongiu
CASPUR
Piazzale Aldo Moro 5
00185 Rome (Italy)
+39 06 3048 4004
marongiu@die.ing.uniroma1.it

Paolo Palazzari
ENEA – HPCN Project
Casaccia Research Centre
P.O. Box 2400 00100 Rome (Italy)
+39 06 3048 3167
palazzari@casaccia.enea.it

Vittorio Rosato[§]
ENEA – HPCN Project
Casaccia Research Centre
P.O. Box 2400 00100 Rome (Italy)
+39 06 3048 4825
rosato@casaccia.enea.it

ABSTRACT

We describe a design methodology which allows a fast design and prototyping of dedicated hardware devices to be used in *heterogeneous* computations. The platforms used in heterogeneous computations consist of a general-purpose COTS architecture which hosts a dedicated hardware device; parts of the computation are mapped onto the former, parts onto the latter, in a way to improve the overall computation efficiency. We report the design and the prototyping of a FPGA-based hardware board to be used in the search of low-autocorrelation binary sequences. The circuit has been designed by using a recently developed Parallel Hardware Generator (PHG) package which produces a synthesizable VHDL code starting from the specific algorithm expressed as a System of Affine Recurrence Equations (SARE). The performance of the realized devices has been compared to those obtained on the same numerical application on several computational platforms.

General Terms

Algorithms, Performance, Design, Theory.

Keywords

Dedicated Hardware Device, Systems of Affine Recurrence Equations, Low-Autocorrelation Binary Sequences.

1. INTRODUCTION

Processors used in general purpose platforms are suitably designed to efficiently execute 32/64 bit floating/fixed point numerical operations and contain complex memory hierarchies to hide the large memory latency. General purpose commodity-off-the-shelf (COTS) architectures are often deployed with a very low efficiency when the computation does not involve standard numerical operations, as in the case of 1-bit computations or string processing. Functional units and data bus bandwidth are not fully exploited due to the execution of simple operations involving not standard numerical (or character) data represented with few bits. The partial inadequacy of COTS platforms to sustain high computational efficiency on specific class of applications has fostered a renewed interest in the design of dedicated hardware devices, purposely realized to cope with the algorithmic complexity inherent to the cited classes of numerical processing [1]. A pioneering example of this approach was given by the Connection Machine CM-1[2], a MPP SIMD system based on 1-bit processors, which had very high efficiency on image processing and, conversely, very poor performance on numerical processing. In order to combine efficiency on both the numerical and not-numerical applications, some platforms use COTS processors boosted by custom chips. The same architectural approach, at the chip level, has been followed by the Cray SV1 designers which embedded, within the microprocessor core, both a vector floating point unit and a Bit Matrix Multiply (BMM) unit devoted to perform some basic bit level matrix operations [3].

We propose a more general approach where we attempt to increase the computational efficiency of the architectures in the tasks where COTS components fail to achieve acceptable sustained performances. We have thus introduced a novel methodology [4] for rapidly designing and prototyping dedicated hardware devices which realize specific computational tasks and can be easily linked to COTS platforms. The result of such assembly is an *heterogeneous architecture* composed by a COTS platform linked to one or more dedicated hardware devices. The computational codes running on heterogeneous platform are thus composed of an usual software part and one (or more) sections which are directly mapped on the dedicated devices. In order to produce, at reasonable costs, a custom dedicated hardware, an interesting trade-off between (expensive) custom silicon and (cheap) integration of COTS components is given by the FPGA (Field Programmable Gate Array) technology. This technology, to date, allows to implement complex circuits (up to 2 million logic gates) on one chip. These circuits can be designed, simulated and prototyped, by using powerful and versatile development tools (like, e.g., the Synopsys FPGA Express, the Xilinx Foundation Express, the Altera MaxPlusII). They can be hosted, as dedicated computational booster, by a COTS processor. A further property of the FPGA technology is its re-programmability, i.e. the possibility of reusing the same hardware support to implement a different circuit which performs a different computation. The re-

[§] also Istituto Nazionale di Fisica della Materia (INFM), Unità di Ricerca Roma I.

SC2001 November 2001, Denver © 2001 ACM 1-58113-293-X/01/0011 $5.00

configuration operation can be performed in a very short time (of the order of tens of milliseconds), thus leading this technology suited to host, in the same hardware support, different types of specialized calculations [5-6].

In order to give a short overview about projects on configurable computing, we recall 1) the Cameron Project (see http://www.cs.colostate.edu/cameron) in which a framework to automatically compile C programs onto programmable devices was developed, 2) the NAPA architecture [7] with the NAPA C compiler [8] which allows the partitioning of applications between fixed instruction processors and configurable coprocessors, 3) the DEFACTO project (see http://www.isi.edu/asd/defacto/) in which, starting from C or MATLAB specifics and using the SUIF compiler (http://suif.stanford.edu/) to extract the parallelism from the applications, the application is mapped onto a target Hardware Description Language which will be compiled onto a programmable device. A detailed review on configurable computing can be found in [9].

To date, there is still a relevant gap between the attainable performances of COTS and FPGA devices. FPGA's, in fact, are still constrained to a clock speed ranging from a few tens up to 100 MHz. This limitation, however, can be overridden by a wise design of the circuit, by exploiting as much as possible the parallelism inherent to the specific computation to be mapped. The use of dedicated hardware devices is the key ingredient to realize heterogeneous architectures for heterogeneous computing [10]. In heterogeneous computing, the algorithm is partitioned: a part is implemented on a COTS-based architecture with the usual high level programming languages, a part allotted to the dedicated hardware device(s) performing the part(s) of the algorithm which does not fit the COTS architecture. In most cases, the exploitation of the algorithm parallelism into the circuit design allows to attain remarkable performances.

In this work we stress the potentiality of our methodology to design components for heterogeneous platforms. As a test-case, we present the design of a dedicated hardware device, based on the FPGA technology, designed to be used in the optimization problem of the search of low-autocorrelation binary sequences (LABS) [11-13]. This problem, of relevance in communication technology, involves a computationally heavy bit-level processing and, as we will show in detail, it is not suited to be efficiently mapped onto COTS processors. From the computational point of view, this task requires the use of an optimization technique where the target function is represented by the sum of the square of all the string autocorrelation values. The design of the presented device has been automatically generated by using the automatic Parallel Hardware Generator (PHG) package [4,14] which is able to produce a synthesizable VHDL from the C-code specifying the computation. The algorithm to be implemented on the special purpose parallel architecture is firstly described in an high level language by means of a set of recurrence equations. Then, the PHG, based on the high level synthesis methodology developed by two of the authors and extensively described in [4,14], produces the VHDL code which is subsequently processed with usual electronic CAD tools and implemented on the FPGA support. The layout of the automatic PHG is briefly recalled in Section 2. Section 3 describes the LABS problem. Sections 4 gives the rationale to use FPGA to implement specialized devices in specific computational tasks, Sections 5 and 6 describe the hardware design. Section 7 reports the resulting performance and a comparison with those arising from the use of state-of-the-art COTS processors.

2. The PHG package

The automatic Parallel Hardware Generator (PHG) package has been developed in ENEA (Italian Agency for New Technology, Energy and Environmental studies) by two of the authors in the framework of the HADES project (HArdware DEsign for Scientific applications, see, e.g., www.enea.it/hpcn/moshpce/hlsynth1e.html). The PHG theoretical framework is described in [9] while the detailed theory is reported in [4].

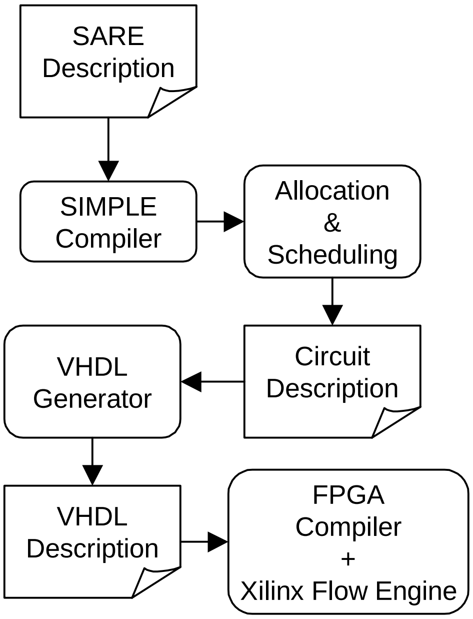

Figure 1. Layout of the design flow. The PHG package contains the SIMPLE compiler, the Allocation and Scheduling Module, the VHDL Generator.

PHG produces a synthesizable VHDL [15] starting from high level specifications given by means of a System of Affine Recurrence Equations (SARE) [16-18]. The SARE is specified through the SIMPLE (Sare IMPLEmentation) language. Details on the SIMPLE language can be found in [4,19].

In order to achieve the final circuit description, PHG performs the following steps (figure 1):

- parsing of the SARE describing the algorithm to be implemented through the SIMPLE compiler and generation of the intermediate format;
- automatic extraction of parallelism by allocating and scheduling the computations through a processor-time mapping [4]. The result of the mapping process is represented by an integer unimodular matrix derived through an optimization process [20]. This step produces the architecture of the system expressed as a set of interconnected functional units (data path) managed by a control Finite State Machine (FSM) (data path controller)

which enforces the scheduling;

- generation of the synthesizable VHDL representing the architecture determined in the previous step.

The VHDL code is then synthesized through the standard Electronic Design Automation (EDA) tools. We used the Synopsys FPGA compiler II to produce the optimized netlist and the Xilinx Foundation Express 3.1 to place and route it into the target FPGA).

3. The Low Auto-Correlation Binary String Problem and the Parallel Tempering Optimization Strategy.

Let us refer to binary strings $\boldsymbol{s}$ of length N, namely $\boldsymbol{s} = \{s_i\}$ ($i = 1,2,\ldots,N$) defined over the binary alphabet {-1,+1}. The Low Auto-Correlation Binary String problem concerns with the search of strings characterized by the lowest possible autocorrelation H_k

$$H_k(\boldsymbol{s}) = \sum_{i=1}^{N-k} s_i s_{i+k} \qquad (1)$$

for all lags k [11-13].

The LABS problem arises from the area of digital communication. In fact, LABS's can be used to generate efficient codes for error correction and robust procedures for communication synchronization. The problem of finding LABS's is usually tackled by using an efficient optimization strategy which minimizes the cost function obtained by cumulating the square of all the autocorrelation lags $H_k(\boldsymbol{s})$:

$$H(\boldsymbol{s}) = \sum_{k=1}^{N}\left(\sum_{i=1}^{N-k} s_i s_{i+k}\right)^2 = \sum_{k=1}^{N} H_k^2 \qquad (2)$$

Due to its complexity, optimization heuristics like Genetic Algorithms (GA) [21], Simulated Annealing (SA) [22,23] or others are used to minimize H($\boldsymbol{s}$).

We have used, in this work, that proposed by Marinari and Parisi [24] (known as "Parallel Tempering", PT hereafter) as it has been shown to give better results, in this specific problem, than the other heuristics previously cited. Let $H(\boldsymbol{s})$ indicate the energy of the system in the state configuration $\boldsymbol{s}$. In the SA case, the algorithm generates a series of configurations s_i with a Boltzmann probability distribution i.e.

$$p(\boldsymbol{s}) = e^{-\beta H(s)} \qquad (3)$$

where β is the usual Boltzman factor $1/kT$. Each time that the T value is changed the system is driven out of equilibrium. With a usual Monte Carlo (MC) scheme, the system is brought all along the configuration space; high T moves allow the system to take over energy barriers. When the system is brought back to $T = 0$, the system should, in principle, visit the local minimum configuration. The repeated application of the Monte Carlo sampling will allow the system to visit several local minima. From the algorithmic point of view, the MC search is performed by generating a new point $x' \in \mathbf{X}^n$ ($\mathbf{X}^n$ being the search space) via the perturbing function GN(x) which receives a point $x \in \mathbf{X}^n$ and returns the new point *close* (according to some predefined metric) to x. The pseudo-code for the Monte Carlo algorithm is shown in figure 2, where *NS* - the number of transitions attempted - is a linearly increasing function of the cardinality of the searching space.

```
/* Monte Carlo cycle at temperature T, MC(T) */
for i=1 to NS
  x' = GN(x)
  accept = false
  if f(x') < f(x) then
    accept = true
  else if exp((f(x)-f(x'))/T) > random(0,1) then
    accept = true
  end if
  if accept then x = x'
end for
/*end of Monte Carlo cycle at temp. T, MC(T)*/
```

Figure 2: Schematic layout of the code describing a Monte Carlo Sampling.

```
input
  M replicas of MC
  temperature values $T_i$ for each MC
output
  x* | f(x*)<= f(x) ∀x generated
begin
  for i=1 to M x[i] = random
  while not stop_condition
    for i=1 to M execute MC(Ti)
    for i=1 to M-1 do
      swap = false;
      if f(x[i+1])<f(x[i]) then
        swap = true
      else if exp((1/$T_{i+1}$-1/$T_i$)*(f(x[i+1])-f(x[i]))
              <random(0,1) then
        swap = true
      end if
      if swap then swap(x[i+1],x[i])
      /* x[i] becomes the solution for MC
         at temperature $T_{i+1}$ and x[i+1] becomes
         the solution for MA at temperature $T_i$*/
    endfor
  endwhile
end
```

Figure 3: Pseudo-code of the PT algorithm.

PT algorithm is based on a set of M replicas of MC cycles at different temperatures. The number of temperatures and their values T_i are chosen, as described in [11], so that:

- all the solutions x have a significant probability to be generated at least in one MC replica, i.e. $\forall\ x \in \mathbf{X}^n\ \exists\ T_i \mid P(\mathrm{f}(x),T_i) > \mathrm{p}_T$, being $P(\mathrm{f}(x),T_i)$ the Boltzman distribution at temperature T_i and p_T a threshold probability. The previous condition implies that the temperature ranges from $T = 0$ to $T = \infty$;

- given two successive temperatures T_i and T_{i+1}, there is a significant number of solutions for which the above expression holds for both the temperatures; this implies that the replica temperatures should display significant superposition in the probability distribution of the corresponding states.

The pseudo-code of the PT algorithm is shown in figure 3.

4. Rationale to develop specialized HW to solve the LABS problem

Performances of general purpose processors are usually limited by the sustainable memory bandwidth as processors are getting faster more quickly than the access rate to memory [25]. This produces a reduction of the available theoretical CPU performances being the memory bandwidth of the system the rate-limiting step. A processor able to sustain N_I instructions/second can sustain such a computational speed *if and only if* the memory is fast enough to supply the requested operands and to store the produced results. In such a case, the memory bandwidth constitutes the rate-limiting factor for an efficient utilization of the computing resources. In fact, due to the technological trend of microprocessors architectures, poor utilization of memory bandwidth generates a severe under-use of computational resources: the more memory bandwidth is wasted, the worst is the global processor utilization. This is just the case of the LABS problem, which we will discuss in some details.

Let us consider the system bus traffic generated by the algorithm which computes eq.(2). It is possible to recognize four types of operations:

1. multiplication of two string elements s_i and s_{i+k} (elementary product): in such a case the operands are represented with 1 bit;
2. accumulation of the elementary products $y_k = \sum_{i=1}^{N-k} s_i s_{i+k}$: in such a case, the result y_k is contained in the interval [-N,N]. Therefore $n_y = (\lceil log_2 N \rceil + 2)$ bit are required to represent y_k with the two-complement notation. For such operation the result and one operand must be represented with n_y bit while the other operand, i.e. the elementary product, is encoded by 1 bit;
3. multiplication $H_k = y_k \times\ y_k$: such operation involves two integral operands contained in the interval value [-N,N]. Binary coding of H_k with the two-complement notation thus requires $n_H = (2\lceil log_2 N \rceil + 2)$ bits. The operator receives one operand with n_y bit and produces a result with n_H bit;
4. accumulation of the H_k values: it is easy to verify that $\sum_{k=1}^{N} H_k$ is less than N^3 so, in order to perform correctly the accumulation, the accumulated results must be represented with $n_+ = (3\lceil log_2 N \rceil)$ bits. The operator receives two operands encoded through n_H bit and produces a result with n_+ bit.

A detailed evaluation of specific operations involved in the calculation of eq. (2) shows that $n_a = N(N\text{-}1)/2$ operations of type 1), $n_b = N(N\text{-}3)/2$ operations of type 2), $n_c = N$ operations of type 3) and $n_d = (N\text{-}1)$ operations of type 4) are needed.

$H(\boldsymbol{s})$ computation is thus dominated by very simple operations, requiring O(N^2) elementary products and O(N^2) summations of short integer values: such operations are the worse efficiently implemented on standard processor. This results in a severe waste of the processor/memory resources. Let us define the ratio η_B between the effective number of bits transferred to/from the memory to compute eq.(2) and the number of bits which would have been transferred if the length of the operands was compliant with the maximal length allowed by the system bus.

In order to compute η_B we note that:

- operation of type 1) is executed n_a times, thus causing the loading of $2n_a$ 1-bit operands and the storing of n_a 1-bit operands; this causes an I/O traffic of $3n_a$ bits against the maximum I/O traffic allowed, in the same number of bus cycles, which is of $3n_aW$ bits, being W the width of the processor data bus;
- operation of type 2) is executed n_b times, thus causing the loading of n_b 1-bit operands, that of n_y-bit n_b operands and the storing of n_y-bit n_b operands; this produces an I/O traffic of $2n_bn_y + n_b$ bit against the maximum I/O traffic allowed, in the same number of bus cycles, which is of $3n_bW$ bits;
- operation of type 3) is executed n_c times, thus causing the loading of n_y-bit n_c operands and the storing of n_H-bit n_b operands; this causes an I/O traffic of $n_c(n_y + n_H)$ bits against the maximum I/O traffic allowed, in the same number of bus cycles, which is of $2n_cW$ bits;
- operation of type 4) is executed n_d times, thus causing the loading of $2n_d$ operands encoded through n_H bit and the storing of n_d operands of n_+ bit; this causes an I/O traffic of $2n_dn_H + n_bn_+$ bits against the maximum I/O traffic allowed, in the same number of bus cycles, which is of 30_dW bits.

We define the effective amount ET of data transferred as

$$ET = 3n_a + n_b(1 + 2n_y) + n_c(n_y + n_H) + n_d(2n_H + n_+) \quad (4)$$

and the the maximum theoretical allowed data transferred (TT), as

$$TT = 3n_aW + 3n_bW + 2n_cW + 3n_dW \qquad (5)$$

If we set $l = log_2N$, the two previous equations turn out to be

$$ET = N^2(4+\lceil l \rceil) + N(\lceil 3l \rceil + 3\lceil 2l \rceil \text{-}2\lceil l \rceil + l/2) \\ \text{-}\lceil 3l \rceil \text{-}\ 2\lceil 2l \rceil \text{-}\ 4 \qquad (6)$$

and

$$TT = (3N^2 - N \text{-}\ 3)W \qquad (7)$$

If we fix $N = 512$ and $W = 64$ (the typical bus width), the resulting efficiency is $\eta_B = ET/TT = 0.068$, i.e. we use less than 7 percent of the bandwidth of the processor. Indeed, we could thought to a more efficient algorithms which use clever coding to reach a better exploitation of the memory bandwidth. Also in this case, processor utilization remains low because conventional processors are not able to control, in a flexible way, the bit-level parallelism. For example, as we will explain later, in LABS problem N-string characters can be encoded within a word of length N, thus allowing to load N-string elements in a memory cycle; furthermore N-elementary products can be performed in parallel through one XOR operations. Unfortunately, N sequential tests on the word containing the N results of the elementary products are successively needed. This example can be considered paradigmatic: when the architecture does not match the type of parallelism of the given problem, some sequential step must be introduced thus wasting, due to the Amdhal's law, the benefits gained from parallelization of some part of the algorithm. This explain the results reported in Table 1 which shows that, for most COTS platforms, the enhancement of the I/O traffic does not correspond to a substantial increase of the sustained computational power.

This fact reminds the key ingredient to be achieved for the mapping of a computation on a computational platform: an efficient balance between the extent of the I/O traffic and the computational power allowed by the functional units of the available CPU.

5. SARE description of the autocorrelation calculation

According to the SIMPLE syntax, the behavior of the device dedicated to the computation of $H(\mathbf{s})$ has been specified through the SIMPLE program shown in Figure 4

```
Ind[k,i];
Par[N] {N>=1};

Input  s[1] {0<=k<=N-1};

Result y;
Result H;

/*Equation 1: y initialization*/
y[]=yInit();
{i=-1,0<=k<=N-1};

/*Equation 2: y computation   */
y[]=yComp(y[k,i-1],s[i],s[i+k]);
{0<=k<=N-1,0<=i<=N-k-1};

/*Equation 3: y propagation   */
y[]=yProp(y[k,i-1]);
{1<=k<=N-1,N-k<=i<=N-1};

/*Equation 4: y power of two  */
H[]=yPowerOfTwo(y[k,i-1]);
{0<=k<=N-1,i=N};

/*Equation 5: H accumulation  */
H[]=HAcc1(H[k,i-1]);
{k=0,i=N+1};

/*Equation 6: H accumulation  */
H[]=HAcc2(H[k,i-1],H[k-1,i]);
{1<=k<=N-1,i=N+1};

/*Write Output                */
Write(H[]);
{k=N-1,i=N+1};
```

Figure 4: Simple code to compute *H*(s).

where:

- the "indices definition" section specifies the set of indices used to perform the computations;
- the "parameter definition" section specifies the parameters of the algorithm along with their validity domain;
- the "input definition" section specifies the input variable x along with its validity domain;
- the "result definition" section specifies the intermediate/final result of the algorithm;
- the "output definition" section specifies which of the final results must be produced by the algorithm as output.

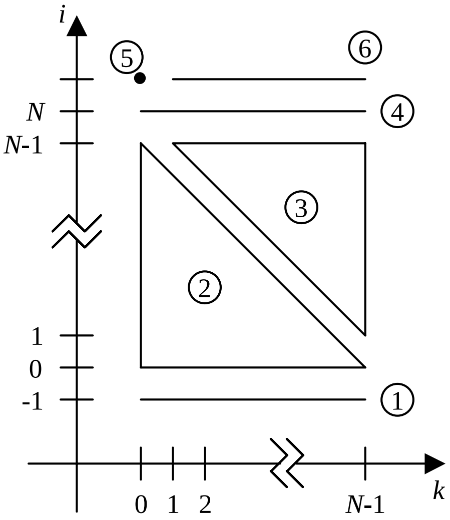

Figure 5: Computing domain for the SARE dedicated to the *H*(s) computation. Each domain is labeled with the number of the corresponding equations.

Equations of the SARE specify how the computations must be performed in order to achieve the final results. Each equation is composed by the symbolic definition of the computing function followed by its validity domain.

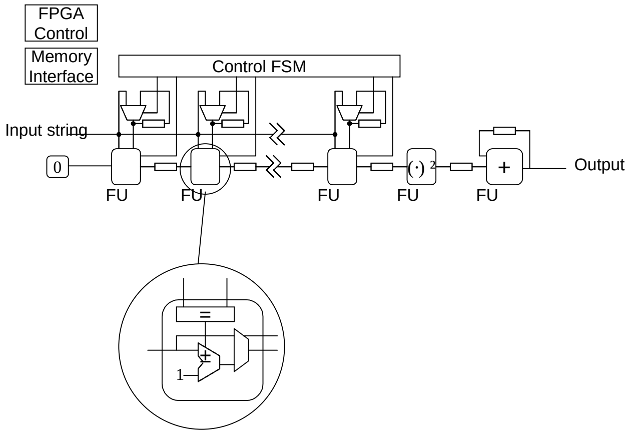

Figure 6: Pipelined architecture designed to compute *H*(*s*)

The SARE algorithm can be represented in the cartesian space, being specified by a set of computations defined over the index set. The SARE algorithm of Figure 4 belongs to the two-dimensional (k,i) space and the shape of the corresponding computing domain is depicted in Figure 5.

Equation 1 initializes the y variable to 0. Equation 2 performs the computation $y(k,i) = y(k,i\text{-}1) + x(i)x(i+k)$. Equation 3 propagates the y values along the i axis, i.e. $y(k,i) = \mathrm{y}(k,i\text{-}1)$. Equation 4 performs the computation $H(k,i) = [y(k,i\text{-}1)]^2$.

Equations 5 and 6 accumulate the $[y(k,i\text{-}1)]^2$ values, i.e. $H(k,i) = H(k\text{-}1,i) + H(k,i\text{-}1)$. The result is contained in the variable $H(N\text{-}1,N\text{+}1)$.

6. Architecture

We designed the circuit to study the case N = 512. Circuits implementing the problem for different values of *N* can be easily generated since the PHG package can design computational devices as a function of the input parameters which describe the size of the problem (like, e.g., the length *N* of the binary sequence, see Figure 4). The architecture obtained applying the PHG to the recurrence equations defining the problem is the pipelined structure sketched in Figure 6. According to the notation and the theory introduced in [14], processor-time coordinates are obtained by using the unimodular transformation matrix $T = \begin{pmatrix}\Lambda\\ \Sigma\end{pmatrix} = \begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix}$. The VHDL source, automatically produced by the PHG tool, contains nearly 63000 lines of code. The architecture is composed by 514 Functional Units (FU). The FU's ranging from 1 to 512 compute eqs.(2-3) (depending on the time step) in the SIMPLE program of Fig.5. FU 513 computes eq.(4) and FU 514 computes eqs.(5-6) (accumulation) of the SIMPLE code. The small boxes are the registers storing the intermediate results of the algorithm. The computing time of the whole pipeline structure is 1026 clock cycles. The pipeline is feed by the string elements received at the Input String port and broadcasted to all the functional units. Such values are stored in a local register to allow local reusing. As in the classical linear pipelining, the output of each FU is sent, through a unitary delay, to the input of the next FU. Last FU is self-connected in order to perform the accumulation (along the time dimension). The whole architecture is controlled by a control FSM which enforces the FU operation scheduling. The architecture has been designed for being hosted by a prototyping board (see Fig.7) equipped with:

- PCI interface;
- 4 independent 2MB SRAM banks (512K*32);
- 1 Xilinx Virtex XV1000 FPGA.

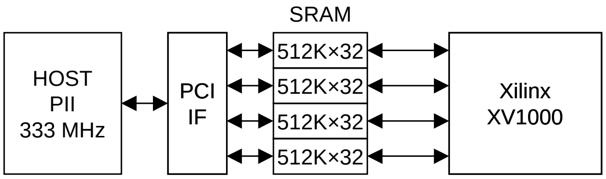

Figure 7: The prototype boards connected to a PCI slot of a Pentium II computer.

Constrained at the speed grade of the FPGA we used (XV1000-4), the synthesized design is clocked at a frequency f_{clk} = 25 MHz. It is worth to note that, once the problem has been defined through the SIMPLE program, all the steps till the final hardware design are executed automatically by the PHG, the VHDL compiler and the standard EDA tools.

Referring to the PT algorithm, the global control flow is demanded to Pentium II host computer; the pseudo-code is sketched in Figure 8. Differently from the code in Fig.3, the iterated execution of MC steps is enclosed in a single procedure call which triggers the FPGA computation.

```
input
  M replicas of MC
  temperature values $T_i$ for each MC
output
  x* | f(x*)<= f(x) ∀x generated
begin
  for i=1 to M x[i] = random
  while not stop_condition
    execute HW_MC
    for i=1 to M-1 do
      swap = false;
      if f(x[i+1])<f(x[i]) then
        swap = true
        else if exp((1/$T_{i+1}$-1/$T_i$)*(f(x[i+1])-f(x[i]))
        <random(0,1) then
        swap = true
      end if
      if swap then swap(x[i+1],x[i])
      /* x[i] becomes the solution for MC at
          temperature $T_{i+1}$ and x[i+1] becomes the
          solution for MA at temperature $T_i$*/
    endfor
  endwhile
end
```

Figure 8: PT pseudo-code with the HW-MC call.

```
HW_MC
  for i=1 to NS
    for i=1 to M
      x'[i] = GN(x[i])
      DO_DMA(HOST2BOARD,x'[i])
      START_FPGA
      WAIT_FPGA_COMPLETION
      DO_DMA(BOARD2HOST,f(x'[i]))
      if f(x') < f(x) then
        accept = true
      else if exp((f(x)-f(x'))/T) > random(0,1) then
        accept = true
      end if
      if accept then x = x'
    end for
  end for
end HW_MC
```

Figure 9: The pseudo-code of the HW-MC routine.

The HW- MC routine (pseudo-code sketched in Fig. 9) performs the MC cycle demanding to the prototyping board the computation of $H(\mathbf{s})$. The computation of cost function requires a sequence of calls to the prototyping board:

- DMA from the host to the board memory in order to communicate the new string *x*'[*i*];
- Computation of the cost function (allotted to the FPGA);
- DMA from the board memory to the host memory of the new cost function.

7. Results

In order to test the advantages derived from the FPGA based design, we implemented the PT algorithm as sketched above on different general-purpose platforms. Moreover we developed two programs to better fit the architecture of the used microprocessor. The first code, whose computational kernel is shown in Figure 8, is based on a straightforward implementation of the autocorrelation function (2): the elements of the binary string are integer numbers whose value is 1 or -1.

```
#define STRING_LENGTH 512

/*Variable declaration*/
...
/* String is a integer array
   containing 1 or –1 values*/

H=0;
for (k=0;k<STRING_LENGTH;k++) {
    Sum=0;
    for (i=0;i<STRING_LENGTH-k;i++)
        if (String[i]==String[i+k]) Sum++;
        else Sum--;
        H+=Sum*Sum;
}
```

Figure 10: Autocorrelation code with string values represented as integer numbers.

The second code, whose computational kernel is shown in Figure 9, is based on a different representation of the binary string values: the elements of the binary string are coded through bits following the convention that a bit set to 0 corresponds to -1 and a bit set to 1 corresponds to 1. In this case a word with 32/64 bit encodes 32/64 string elements while, in the first code, the same word encodes only one string element; as a consequence, when a word is loaded, 32/64 string elements are loaded into the processor.

Expression (2) is computed through basic bit level operations: a) comparison between string elements is performed through the **exclusive or** (**XOR**) operator with parallelism degree fixed by the word length, b) the k-th autocorrelation lag is computed through a sequence of increment/decrement operations driven by the masking of the result of the exclusive or operation. While such a code exploits parallelism at the word level, it suffers from the large number of shift operations required by the autocorrelation algorithm and by the sequential analysis of the **XOR** result to test whether the i-th character of the first operand matches the corresponding character of the second operand. This bit level code runs on 32/64 bit target architectures.

The test processors have been chosen among the most powerful available in our research center and the results have been compared with those achieved on the test bed architecture equipped with a 333 MHz Pentium II processor connected to a Xilinx XV1000 FPGA configured to support the pipelined architecture in Fig.6.

For all the tests we performed, we used the standard processor intruction set. No tests have been performed using special instruction sets like the MMX, VIS, etc.

```
#define WORD_LENGTH 32 /*or 64*/
#define STRING_LENGTH 512
#define WORD_NUMBER (STRING_LENGTH/WORD_LENGTH)

/*Variable declaration*/
...
/* Mask[k]=2**k for k=0,...,WORD_LENGTH-1*/
/* String1 and String2 are initialized to the
   same binary string value */

H=0;
for (i=0;i<STRING_LENGTH;i++) {
  Sum=0;
  index=i/WORD_LENGTH;
  for (j=index;j<WORD_NUMBER;j++) {
    /* Begin Computation of lag j of
       autocorrelation*/
    Aux[j]=String1[j]^String2[j];/*xor operation*/
    if (j==index) {
      for (k=i%WORD_LENGTH;k<WORD_LENGTH;k++) {
        if (Aux[j]&Mask[k]) Sum--;
        else Sum++;
      }
    }
    else {
      for (k=0;k<WORD_LENGTH;k++) {
        if (Aux[j]&Mask[k]) Sum--;
        else Sum++;
      }
    }
    /* End of computation of lag j of
       autocorrelation*/
  }
  H+=Sum*Sum;
  /*Shift String2 up of 1 bit*/
  for (j=WORD_LENGTH-1;j>index;j--) {
    String2[j]>>=1;
    String2[j]+=(String2[j-1]&1)<<(WORD_LENGTH-1);
  }
  String2[index]>>=1;
}
```

Figure 11: Autocorrelation code with string values represented at bit level.

The architectures used in the tests are:

- 333 MHz Pentium II (the same used in the test bed architecture) , MS Visual Studio 6.0 C++ compiler. This processor is quite old and it has been chosen to show the speedup achieved when the same processor is combined with the FPGA;
- 1000 MHz Pentium III, 32 bit architecture, 512K of L2 cache, Windows 2000 OS, MS Visual Studio C++ compiler V6.0;
- 667 MHz Alpha EV6.7 (API UP2000 board), 64 bit architecture, 4M of L2 cache, Linux OS Kernel 2.3.14, Compaq C compiler (ccc) V6.2-506;
- 450 MHz Sun UltraSparc II (Ultra60 workstation), 64 bit architecture, 4M of L2 cache, Solaris 2.7 OS, Sun WorkShop C compiler V5.0;

- 300 MHz SGI Onyx R12000, 64 bit architecture, 4M of L2 cache, Irix64 6.5 OS, MIPSpro C compiler V7.3.1.1m;
- 200 MHz IBM Power3, 64 bit architecture, 4M of L2 cache, AIX 4 OS, C for AIX compiler V4.4;
- 300 MHz Cray SV1 Node, 64 bit vector architecture, 256K L2 cache, Cray UNICOS OS, Cray standard c compiler V6.3.0.0 (the cray C compiler has been used with the option -Oscalar3,vector3 in order to vectorize the code).

We report for each system:

a) the time, in milliseconds, required to perform 16 MC computations with the two kinds of code and, where applicable, results for the bit level code are split between the 32 and 64 bit version;

b) the maximum and minimum speedup of the FPGA based architecture vs. each test architecture.

Table 1: Elapsed times (milliseconds) of the test case with *N*=512, *M*=16 and *NS*=2048 on the different platforms (clock frequencies in parentheses). The last column shows the minimum and maximum speed-up values achieved by the FPGA-based implementation with respect to the specific platform.

	Integer	Bit Level (32 bit)	Bit Level (64 bit)	Min/Max FPGA Speed up
PII(333)+FPGA	3524	-	-	1/1
EV6.7 (667)	25213	24422	23155	6.6/7.2
PIII (1000)	53067	30314	-	8.6/15.2
R12000 (300)	35117	43965	40913	10./12.4
SV1 (300)	36966	58870	-	10.5/16.7
Power3 (200)	81345	41834	-	11.8/23
UltraSparc (450)	59080	69171	75217	16.8/21.3
PII (333)	161981	89408	-	23.4/46

Time reported for the FPGA based architecture is 3524 ms. This time includes: Pentium II control overhead (152 ms), DMA from host main memory to FPGA local SRAM (2030 ms) and FPGA computation (1342 ms). Pentium II control overhead is given by the time to verify the MC acceptance condition, to perform the PT swap and to manage the whole iterative control flow. The time spent in the DMA operation includes *NS* transfers of *M* binary strings from the host memory to the FPGA SRAM and *NS* transfers of *M* integer values, i.e. $H(\mathbf{s})$ values, from the FPGA SRAM to the host memory. Finally FPGA computation is the time spent by the FPGA to compute $NS\times M$ times expression $H(\boldsymbol{s})$ value.

Referring to data reported in Table 1, we see that the FPGA based solution is always considerably faster than the ones based on COTS with speed up ranging from 6.6 (EV6.7 running the code in Figure 9 with WORD_LENGTH = 64) up to 23 (Power3 running the code in figure 8); we obviously have neglected the PII case. May be interesting to note that not all the architectures are able to take advantage from the exploitation of the word level parallelism.

8. CONCLUSIONS

The results of this work enforces our view that dedicated hardware devices can be a key issue to deal with computational problems which cannot be efficiently mapped on the architectures of conventional, general-purpose processors. The specific solution, produced by the PHG package for the LABS problem, highlights the role and the effect arising from a joint use of a COTS platform and a specifically designed architecture. In particular, it emphasizes that a more efficient use of the I/O bandwidth, together with the presence of a large number of parallel functional units able to process the input data, allows to produce large sustained computational power also in the presence of the low clock speed of the device.

Using previous results based on parallelization of algorithm expressed trough the SARE model, we discussed the use of dedicated device to efficiently support the LABS problem solution. Experimental results derived from tests on both a prototype architecture, equipped with a board mounting XV1000 Xilinx FPGA configured to support LABS computation and some commercial systems based on COTS processors, clearly demonstrates the advantages, in terms of sustained performance, which can be achieved when, to COTS processors, is added a device with internal parallel architecture which matches the type of parallelism of the computationally-intensive part of the algorithm. We stress that the same approach based on heterogeneous architecture can be effectively used whenever a computationally intensive problem a) involves operations not efficiently supported by COTS processors, b) under-uses processor bandwidth, c) includes an heavy computational kernel which can be mapped onto a parallel architecture embeddable within a programmable device, d) communications between such a computational kernel and the remaining part of the algorithm has a complexity smaller than that of the computational kernel.

9. REFERENCES

[1] Special Issues of the Computer Physics Communication Journal on “Formal Methods for HW/SW design for Grand Challenge Scientific Applications” - Guest Editors P.Palazzari, V.Rosato - to appear in 2001.

[2] Hillis, W. Daniel. “The Connection Machine”, MIT Press, Cambridge, MA., 1985

[3] Cray SV1 Application Optimization Guide (downloadable from http://www.cray.com/swpubs/manuals/S-2312-36/html-S-2312-36/)

[4] A.Marongiu, "Hardware and Software High Level Synthesis of Affine Iterative Algorithms", Ph.D Thesis in Electronic Engineering, "La Sapienza" University of Rome, February 2000.

[5] Proc. of the 7th Reconfigurable Architectures Workshop (RAW 2000), 1 May 2000, Cancun, Mexico

[6] IEEE Trans on Computer, Special Issue on Configurable Computing, **48** (1999)

[7] C.R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J.M. Arnold, and M. Gokhale: 'The NAPA Adaptive Processing Architecture', Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, 15 - 17 April, 1998, Napa Valley, California

[8] M.B. Gokhale and J.M. Stone: 'NAPA C: Compiling for a Hybrid RISC/FPGA Architecture', Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, 15 - 17 April, 1998, Napa Valley, California

[9] [RecRef] Katherine Compton, Scott Hauck: 'Configurable Computing: A Survey of Systems and Software'. Tech. Report from the Northwestern University, Dept. of ECE, 1999

[10] P.Palazzari, L.Arcipiani, M.Celino, R. Guadagni, A.Marongiu, A.Mathis, P.Novelli, V.Rosato “Heterogeneity as Key Feature of High Performance Computing: the PQE1 Prototype”, Proceedings of "Heterogeneous Computing Workshop", Cancun, Mexico - May 1, 2000.

[11] E.Marinari, G.Parisi, F.Ritort, "Replica Field Theory for Deterministic Models: Binary Sequences with Low Autocorrelation", J. Phys. A: Math. Gen. **27** (1994) 7615.

[12] F.Ferreira, J.Fontanari, P.Stadler, "Landscape statistics of the low-autocorrelation binary string problem", J. Phys. A: Math. Gen. **33** (2000) 8635.

[13] B. Militzer, M. Zamparelli, D. Beule, "Evolutionary search for low--autocorrelation binary sequences", IEEE Trans. on Evol. Comp., **2** (1998) 34.

[14] A.Marongiu, P.Palazzari, "Automatic Mapping of System of Affine Recurrence Equations (SARE) onto Distributed Memory Parallel Systems", IEEE Trans on Soft. Eng., **26** (2000) 262.

[15] IEEE standard VHDL language reference manual. IEEE std. 1076-1993

[16] C.Mongenet, P.Clauss, G.R.Perrin, "Geometrical Tools to Map System of Affine Recurrence Equations on Regular Arrays", Acta Informatica, **31** (1994) 137.

[17] K.H.Zimmermann, "Linear Mapping of n-dimensional Uniform Recurrences onto k-dimensional Systolic Arrays", Journal of VLSI Signal Processing, **12** (1996) 187.

[18] A.Darte, "Regular Partitioning for Synthesizing Fixed-Size Systolic Arrays", INTEGRATION, The VLSI Jounal, **12** (1991) 293.

[19] A. Marongiu, P. Palazzari, L. Cinque and F. Mastronardo, "High Level Software Synthesis of Affine Iterative Algorithms onto Parallel Architectures". Proc.of the 8th Int. Conf. on High Performance Computing and Networking Europe (HPCN Europe 2000). May 2000 Amsterdam, The Netherlands.

[20] A. Marongiu, P. Palazzari, "Optimization of Automatically Generated Parallel Programs", Proc of the 3rd IMACS International Multiconference on Circuits, Systems, Communications and Computers (CSCC'99) - July 1999, Athens.

[21] D.E.Goldberg, "Genetic Algorithms in search, optimization and machine learning" - Addison Wesley 1989.

[22] S.Kirkpatrick, C.Gelatt, M.Vecchi, "Optimization by Simulated Annealing", Science, **220** (1983) 498.

[23] A.Deckers, E.Aarts, "Global Optimization and Simulated Annealing", Mathematical Programming, **50** (1991) 367.

[24] E.Marinari, G.Parisi, "Simulated Tempering: A New Monte Carlo Scheme", Europhysics Letters, **19** (1992) 451.

[25] J. D. McCalpin, "Memory bandwidth and machine balance in current high-performance computers", Newsletters of the IEEE Technical Committee on Computer Architecture, December 1995.