Dedicated Hardware Accelerators for the Epistatic Analysis of Human Genetic Data

Fabio Cancare  
Dip. di Elettronica e Informazione  
Politecnico di Milano  
Milano, I-20133, Italy  
Email: cancare@elet.polimi.it

Alessandro Marin  
Dip. di Elettronica e Informazione  
Politecnico di Milano  
Milano, I-20133, Italy  
Email: alessandro.marin@dresd.org

Donatella Sciuto  
Dip. di Elettronica e Informazione  
Politecnico di Milano  
Milano, I-20133, Italy  
Email: sciuto@elet.polimi.it

Abstract—The recent advances in genomic microarrays design provide the possibility to retrieve hundreds of thousands of significative genetic features from patients at affordable costs. Understanding if non-linear interactions (epistatic relationships) between these features determine or not the arising of complex common multifactorial genetic diseases is a critical task for human geneticists. The algorithms able to detect such relationships, like the Multifactor Dimensionality Reduction (MDR) algorithm, are computationally expensive and their practical utility is very often limited by the amount of time required by the analysis. This paper presents three hardware-accelerated implementations of the MDR algorithm, tailored for many-core processors, Xilinx Virtex-5 FPGAs and generic GPUs respectively. These implementations provide timing performance improvements of up to two magnitude orders with respect to the software implementation.

Keywords-FPGA; GPGPU; Bioinformatics; Epistatis; Hardware Accelerator

I. INTRODUCTION

Since its birth, the main goal of Bioinformatics has been to retrieve significant biological information by processing raw experimental laboratory data. If until some years ago the main problem that bioinformaticians were facing was to obtain useful raw biological data, in the last few years the main problem has become to deal with the huge quantity of the relevant available data and, in particular, the main issue is to process them in a reasonable amount of time.

Many bioinformatics algorithms are characterized by a high computational complexity and their execution on a generic workstation can be quickly overwhelmed by an increase in size of the input data. A solution to this problem could be the implementation of the target bioinformatics algorithm exploiting recent digital technologies such as multi-core/many-core processors, graphical processing units applied to general purpose computing (GPGPU) and reconfigurable hardware (FPGA). These devices can greatly improve the temporal efficiency of any algorithm characterized by a certain degree of task and/or data parallelism.

If, on one hand, these devices can provide speed-ups of one, two and sometimes three orders of magnitude, on the other hand they require the modification of the application source code to make it compatible with the chosen hardware architecture. Sometimes altering the source code is equivalent to insert some keywords (pragmas) within the code. Most of the times, however, the code must be partially or completely rewritten in order to expose the parallelism and/or to make use of a programming language that can be compiled targeting the chosen hardware architecture. Thus, it is worthwhile to think about the tradeoff between having a faster application and the effort of coding it. This is particularly true when the bioinformatician has little or no knowledge about FPGAs, GPGPUs and multi-core/many-core processors.

In the light of the considerations reported above, the goal of this paper is to show how the recent hardware technologies can be exploited to accelerate a bioinformatics algorithm characterized by a high-computational complexity. The paper focuses on an application for epistasis detection accelerated using an Intel many-core processor, a Xilinx Virtex-5 FPGA and an Nvidia Tesla GPU.

The target algorithm is called Multifactor Dimensionality Reduction (MDR) [1]–[3]; it is a software tool used by medical geneticists to understand how genetics features and environmental factors determine the susceptibility to complex common multifactorial diseases. In contrast with Mendelian diseases, which are caused by a single genetic variant and which can now be early diagnosed and fought thanks to genetic testing, complex common multifactorial diseases are caused by the interaction between multiple genetic variants and environmental factors; genetic variants contribute to determine disease susceptibility by interacting among them in a non-additive fashion; environmental factors, instead, constitute disease triggers and inhibitors. Examples of these pathologies are: coronary heart disease, hypertension, cancer, diabetes, obesity, bipolar disorder and depression. Differently from Mendelian disorders, medical geneticists were not able, so far, to find the exact genetic causes determining diseases susceptibility. Such findings would constitute an exceptional advancement since they would allow the prevention or early diagnosis of complex multifactorial genetic diseases. The algorithm will be described in Section II; it has been chosen as a good candidate for hardware acceleration due to its current limitations, its practical utility [4], [5] and the fact that it is characterized by a high degree of data parallelism.

The paper is structured as follows: Section II describes how the MDR algorithm works. Section III provides a description of the source code analysis performed for gathering more in-
formation on the algorithm and on how to accelerate it. Section IV shows how the MDR algorithm has been implemented targeting the chosen target platforms while Section V reports a comparison between the timing performance of the accelerated versions and the canonical software implementation written in C. Finally, Section VI wraps up the authors conclusions.

II. THE MDR ALGORITHM

The MDR algorithm is a three-step algorithm that, given as input a case-control dataset reporting, for each person, his/her genotypic and phenotypic features along with his/her status (affected or not affected), and the size \(N\) of the model to be found, is able to detect the N-dimensional model which best represents the dataset. The first step is selection: \(N\) parameters are chosen from the pool of all the parameters; the selection can be exhaustive (i.e. all the parameters combinations are explored) or user-driven. The second step is model building: a model is an N-dimensional space equivalent to the Cartesian product of the parameters domains. The size of a genetic parameter domain is the number of genotypic combinations that a genome locus can assume; given the properties of their genome, in humans this number is fixed to three. Phenotypic variables instead are almost always Boolean variables. During this step each case-control entry is categorized within a cell of the N-dimensional space; for each cell two counters keep track of the number of cases and controls falling within the cell. The third step is model evaluation: for each cell of the model, if the case/control ratio exceeds or is equal to a predefined threshold (usually 1.0) the cell is labeled as “high-risk”, otherwise it is labeled as “low-risk”. Once all cells are labeled, the model is evaluated by estimating its prediction error. This process is repeated until all the parameters combinations are processed.

Based on the description above, given a dataset of \(r\) patients characterized by \(c\) genotypic/phenotypic variables, the algorithm asymptotic complexity, in the exhaustive case, is:

\[
O \left( \binom{c}{N} \cdot N \cdot r \right)
\]

The binomial coefficient represents the number of combinations to be tested by the algorithm; for each combination, \(N\) features of each person must be examined. For every combination, \(N\) parameters of each dataset row must be extracted and evaluated. The data parallelism is huge: there are no dependencies between the different combinations, thus they can be processed in parallel; plus, also groups of rows can be processed independently by aggregating their partial results and then computing the final model accuracy.

III. PRELIMINARY ANALYSIS

The MDR algorithm has been originally developed by Moore et al. [1], [2] and is freely available to researchers as an open source package [6] written in both C and Java. The C version is the starting point for the other implementations; the source code has been analyzed in order to understand which hardware/software partitioning optimizes the timing performance. The analysis has been accomplished by first profiling the source code, then by identifying the data dependencies between the various procedures and finally by modeling the overall system. In this last phase it was assumed that every procedure could be implemented either as a software module or as a hardware module. The model has then been used to decide which is the most efficient hardware/software partitioning.

A. Profiling

Profiling provides information about every procedure called during a software execution. In particular it is possible to know how many times the procedure has been called and its average execution time. This technique requires to compile and link the source code with the profiling option enabled and then to run the obtained executable. The running phase will be slower than normal because the program will also compute the profiling data. The profiling of the MDR algorithm has been performed using the GNU profiling tool: Gprof. The algorithm has been profiled using different datasets, however the weight of each procedure with respect to the overall execution time was almost the same in each run.

The result of the profiling data was the identification of those procedures that contribute most to the total execution time and the number of instructions executed. Table I shows the data of the most significant procedures when \(N\) is set to 3 and the dataset lists 1000 attributes of 1600 different patients (a typical MDR use-case). The analysis of the instructions executed highlights that 90% of instructions belong to the model building procedures while the remaining procedures perform only 10% of the instructions. The analysis of the execution time highlights that 85% of the execution time corresponds to model building procedures, while the remaining 15% is related to the remaining procedures.

<table>
<thead>
<tr>
<th>ID</th>
<th>Total instructions</th>
<th>Number of calls</th>
<th>Total Runtime (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>303,360,518</td>
<td>18,999,115,020</td>
<td>28,363</td>
</tr>
<tr>
<td>1b</td>
<td>10,986,180</td>
<td>166,167,000</td>
<td>1,003</td>
</tr>
<tr>
<td>2b</td>
<td>7,801,300</td>
<td>166,167,000</td>
<td>731</td>
</tr>
<tr>
<td>3b</td>
<td>67,032,000</td>
<td>4,040,361,944</td>
<td>8,195</td>
</tr>
<tr>
<td>4b</td>
<td>76,608,000</td>
<td>4,040,361,944</td>
<td>9,264</td>
</tr>
<tr>
<td>5b</td>
<td>109,440,000</td>
<td>8,080,723,888</td>
<td>8,709</td>
</tr>
<tr>
<td>6b</td>
<td>444,600</td>
<td>166,167,000</td>
<td>123</td>
</tr>
</tbody>
</table>

B. Data dependencies

Besides the information about the number of calls and the average execution time of every procedure, the source code profiling also produces a call graph that turns out to be extremely useful if data dependencies must be analyzed. Identifying the data dependencies is an essential task because the size of these data is one of the main factors affecting the performance of a mixed hardware/software system. The goal of this task is to understand which data are needed by each procedure and which is the previous procedure (or the block of previous procedures) affecting the data needed...
by the one under analysis. This goal has been accomplished thanks to the call graph analysis: for each relationship, the size of the data has been inferred from the algorithm source code; this information will be part of the final system model. The call graph, together with the data dependencies between procedures, is shown in Figure 1.

C. Modeling and solution space exploration

Once the algorithm procedures have been identified and the amount of data transferred among them has been computed, a model of the system has been created; from this model it is possible to infer the timing performance changes when a procedure is implemented either as an hardware module or as a software routine. The model parameters are (1) the speed-up of the hw modules with respect to the software implementations of each procedure and (2) the data transfer rate between the software routines and the hardware modules. The model takes into consideration the size of data transmitted between the caller and the called procedures to assess the impact of the communication time between hw nodes and sw nodes. Through the model it is possible to simulate the performance of the algorithm when the hw/sw partitioning changes.

The model has been used to test all the hw/sw partitions shown in Table II. Figure 2 is a plot showing the execution time (left side) and the speed-up (right side) characterizing each hw/sw solution proposed. The y axis report either data about the execution time or the speed-up while the x axis report the different hw/sw solutions. In each plot there are four different data series. In the first one the hw acceleration is set to 10x, in the second one to 100x, in the third one to 200x and in the fourth one to 1000x.

From the plots it is possible to infer that the hw/sw partitions 3, 4 and 6 require an execution time greater than the pure software execution time if the data transfer rate is less or equal to 80 MB/s; this is caused by the size of the data to be transferred from hw nodes to sw nodes and vice versa. The partitioning number 10, in which most of the procedures are implemented as hardware modules and the size of the data transferred between hardware and software nodes is minimal, is the one achieving the best results.

Table III shows the average results obtained by simulating different partitioning solutions and varying the hw speed-up of each procedure and the data transfer rate. The first row reports the software running time when N is set to 3 and the dataset lists 1000 attributes of 1600 different patients. The other rows show how the running time changes when one or more procedures are implemented as hardware modules. The simulations were made by considering several different hw speed-ups (from 10x to 1000x) and several different sw-hw data transfer rate (from 40MB/s to 120MB/s, experiments with higher transfer rates resulted in no significant speed-up increases).

From Table III it is possible to infer that the hw acceleration of the most time consuming procedures (those used in the model building phase) provide an average speed-up of 2.81x. In order to obtain greater speed-ups it becomes necessary to accelerate also those procedures that are not so time-consuming in software, but which slow down the
TABLE III
DATA OBTAINED THROUGH THE SIMULATION OF THE SYSTEM MODEL.
THE TABLE SHOWS HOW THE AVERAGE TIMING PERFORMANCE CHANGES
WHEN DIFFERENT HW/SW PARTITIONING ARE ADOPTED

<table>
<thead>
<tr>
<th>Running Time (sec.)</th>
<th>Procedures as HW</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.363</td>
<td>None</td>
<td>1.00x</td>
</tr>
<tr>
<td>20.611</td>
<td>5b</td>
<td>1.38x</td>
</tr>
<tr>
<td>15.244</td>
<td>1b,3b,5b</td>
<td>1.86x</td>
</tr>
<tr>
<td>10.095</td>
<td>1b,2b,3b,4b,5b</td>
<td>2.81x</td>
</tr>
<tr>
<td>6.021</td>
<td>1b,2b,3b,4b,5b,6b</td>
<td>4.71x</td>
</tr>
<tr>
<td>4.06</td>
<td>1b,2b,3b,4b,5b,6b</td>
<td>68.78x</td>
</tr>
</tbody>
</table>

hw/sw solution due to data dependencies. If all the procedures
characterized by data dependencies with the model building
procedures are implemented as hardware modules then it is
possible to get a 68.78x speed-up.

IV. IMPLEMENTATION
A. Many-core processors

The profiling data and the fact that both the model building
phase and the model evaluation phase can be carried out in
parallel if different combinations are used lead to a many-core
implementation quite straightforward: a master thread is in
charge of creating the attribute combinations to be evaluated,
then for each combination (or for each set made of a certain
number of combinations) it spawns a thread which has to build
and evaluate the model arising from the given combination;
when a spawned thread terminates, the master thread has to
gather and store the data produced in order to keep track of
the best model found so far.

The multi-thread version of MDR has been obtained using
OpenMP [7]; the C source code has been slightly modified in
order to better expose the data parallelism and then a single
OpenMP pragma has been inserted. The pragma controls a
for loop, it automatically splits the computation over a certain
number of threads and it is able to manage the variables (some
of them are shared between the threads, the others are private)
used within the loop. The pragma determines, on the basis
of the multi-core/many-core processor used, which is the best
number of threads to be run concurrently. Each spawned thread
is in charge of executing all the model building and model
evaluation procedures shown in Figure 1.

There are three nested for loops in the MDR C source
code. All of them were used as target of the OpenMP pragma
in order to understand which is the one that, parallelized,
guarantee the best timing performance boost. Experiments
showed that, with medium and large datasets, parallelizing
the most inner for loop leads to the best timing performance
improvement. The reason is that threads belonging to most
inner for loops can exploit some shared variables, created in
the outer for loops, to be only read and not written during the
thread computation. With OpenMP it is possible to specify that
shared variables are read-only and this improves the access to them.

The effort required to introduce multi-threading exploiting OpenMP is very low (especially if compared to the effort required to obtain a GPU or a FPGA implementation of the same algorithm). Bioinformaticians who want to use OpenMP need just to know the basic of the C/C++ language and to have a good knowledge of the target algorithm.

B. GPUs

Implementing an algorithm exploiting GPGPUs is particularly effective when, as in the case of MDR, the degree of data parallelism is very high. That is because GPUs are arrays of SIMD (Single Instruction, Multiple Data) devices, they are able to execute the same instruction over a very wide set of data at the same time [8].

There are two main programming models for the development of GPGPU applications: CUDA [9] and OpenCL [10]. The CUDA programming model is provided by Nvidia and relies on the CUDA-C programming language. It is compatible with all and solely the CUDA-enabled Nvidia GPUs. For more details about CUDA see [11]. The OpenCL programming model has been developed by the Khronos Group, a consortium of several media-centric companies like AMD, ARM, Nvidia and Intel. The model is based on the OpenCL language and allows targeting a great variety of modern GPUs as well as DSPs and microprocessors. For more details about OpenCL see [12].

In the following it will be described how the MDR algorithm has been implemented for exploiting GPGPUs; there are two different implementations: the first one is written in CUDA-C while the second one is written in OpenCL.

1) CUDA-C implementation: The CUDA-C MDR implementation, firstly proposed in [13], is based on the CUDA-C language [9] and the Python programming language, with a binding called PyCUDA [14]. This model, along with the Parallel Python (PP) library [15] for parallel execution, allows for distributed, networked, high-performance clusters of GPUs that can simultaneously perform a single task. The Numpy library is used for efficient manipulation of the data arrays.

In order to achieve more performance by exploiting data locality, the traditional MDR dataset structure has been changed: instead of having patients as dataset rows and attributes as dataset columns it is better to have attributes as rows and patients as columns. In this way it is possible to get several data (instead of a single datum) per memory access, as schematized in Figure 3.

The structure of the block and thread hierarchy has been defined according to the analysis reported in Section III. To each thread block is assigned a specific attribute combinations. The threads in the block work in parallel on different data, accomplishing the model building and model evaluation phase. The grid is responsible for starting and gathering the results about each attribute combination and additionally runs a reduction that finds the best solution.

One of the main difficulties with writing CUDA code is organizing the memory to maintain high resource utilization and efficiency. In the CUDA programming model there are six different memory spaces: constant memory, global memory, shared memory, texture memory, local memory and registers [11]. In the following paragraphs it will be explained how the available memory spaces have been used to implement the MDR algorithm.

Constant memory, as its name implies, stores constants, values that cannot be changed by kernels running on the GPU. This makes them limited to lookup tables and similar data structures. In the GPU implementation of MDR, they are used to store the phenotypes of all the individuals. Because constant memory is cached and localized, phenotypes are accessed in linear order to ensure spatial locality and cache coherency.

Global memory is the actual RAM that resides on the graphics card, attached via the printed circuit board to the GPU itself. It is slow, but if used correctly can still yield acceptable results. The implementation holds the genotype array directly in global memory. Since only two lookups are used per attribute per run, the overhead is minimal. Other solutions, such as caching global memory in shared memory or using the aforementioned texture memory, were evaluated, but none of these were as fast in a variety of situations as the pure global reads.

Shared memory is a small RAM buffer (16-48 Kb) that can be accessed by all threads within a block. It has a variety of uses (caching, intermediate results, etc), and many of these are used extensively. Most importantly, the parallel reductions which are prevalent in the program design and the buckets which form the main storage component of the program both act on and reside in shared memory. Registers are storage locations which are thread-local, so only a single thread in a block can access their value and they are unique across threads. This makes registers critical for keeping track of which values an individual thread should compute and also for storing results of global reads.

Finally, it is important to note that Python has a number of restrictions that relate to running code in parallel (actually executing two pieces of code simultaneously). Most limiting is the Global Interpreter Lock, in which limits access to I/O resources to only one thread at a time. Practically, this means that only one instance of PyCUDA can run in

![Figure 3. MDR dataset structure for exploiting data locality in GPUs. With a single memory access it is possible to retrieve several contiguous data of different patients](Image)
each Python instance, so only one GPU can be utilized per execution, even though the GPU implementation does not tax the CPU. While there are ways around this single-active-GPU-per-process limitation (by saving and restoring PyCUDA contexts), the better solution is to use an external library to run two Python instances at the same time from the same file. It was chosen to use the Parallel Python library [15], as it allows for seamless parallel execution not only across cores but also across machines.

2) OpenCL implementation: After implementing MDR using CUDA-C [13], it was decided to develop an OpenCL implementation of the MDR algorithm for targeting generic GPUs and not just Nvidia GPUs. The CUDA-C implementation and its python wrapper have been used as a reference for developing the OpenCL based implementation. In this implementation, however, the PyCUDA library has been replaced by the PyOpenCL library [16] and the GPU kernel has been completely rewritten using OpenCL.

The OpenCL implementation spawns as many work-groups as those supported by the target GPU. Each work-group is composed of a variable number of work-items (the number is based on the characteristics of the target GPU); each work-item evaluates the accuracy of the model underlying a single combination. Then a work-group wide reduction phase is performed in order to find out the accuracy of the best model analyzed by the work-items. The computation is repeated until all the combinations are analyzed.

The effort required to code an implementation of a generic algorithm able to exploit GPUs is usually higher than the effort required for coding a multi-threaded implementation; the critical code sections must be partially or completely rewritten and the communication between the host system and the GPU must be correctly handled. Moreover, the memory hierarchy must be used efficiently and the device resource utilization must be maintained high: these last goals can be accomplished only by applying a continuous improvement process to tune both the OpenCL code and the parameters influencing the GPU behavior (for example, the size of the work-groups and the types of memory used by the work-items).

C. FPGAs

The FPGA implementation, differently from the multicore/many-core processor implementation and GPUs implementation, has to be more device-specific. If hardware cores can be written in VHDL or Verilog and then synthesized targeting different FPGAs, some inner components (e.g. multipliers, dividers, ...) should be chosen from the components libraries provided by FPGAs vendors in order to achieve best results.

The first version of the FPGA based MDR implementation was obtained targeting a Xilinx Virtex-5 XC5VFX70T FPGA [17] placed within a Xilinx ML507 evaluation board [18]. First of all a base system architecture has been created using the embedded development kit provided by Xilinx, EDK 12.3 [19]; the base system architecture is composed of a PowerPC 440 processor, a DDR2 memory controller, a RS232 controller and a timer for performance evaluation. The system components are connected using a single PLB bus; the processor is the only bus master, the remaining parts of the system are bus slaves.

The most computational intensive parts of the MDR algorithm have been implemented as a hardware IP-core linked to the bus with a standard PLB interface. Such hardware core has been called mdr_block, described using the VHDL language and it is composed of a state machine and a set of mdr_cores. Each mdr_core is able to compute the accuracy of the model underlying a given attribute combination; the mdr_core receives as input the genotype/phenotype data and provides as output the model accuracy. This component is parametric with respect to N, the number of attributes that constitute the combination to be processed. Once N is fixed, the mdr_core can be synthesized and used; in the following description of the mdr_core, N is set to 3.

The mdr_core component is divided in two different parts: the first one performs the model building phase while the second one performs the model evaluation phase. An high-level schematic of the mdr_core is shown in Figure 4.

![Fig. 4. High level schematic of the mdr_core component](image)

The mdr_core interface is composed of seven I/O signals:

- **control** is used to reset the core and to switch from the model building phase to the model evaluation phase. It is also used to signal that new data are available when the core is in the model building phase;
- **geno0**, **geno1** and **geno2** are 32bit signals encoding N genotype/phenotype data of 16 persons. Each single genotype/phenotype datum can, in fact, be represented with just 2 bits;
- **status** states if the 16 persons currently processed are affected or not by the disorder under analysis;
- **ready** is used by the core to send acknowledgements during the model building phase and to signal whether the model accuracy has been computed during the model evaluation phase;
- **acc** is the double precision model accuracy.

The mdr_core model building logic is in charge of processing the genotype/phenotype data in order to update the value of the N-dimensional model cells. It is a fully pipelined hardware chain composed of (N-1) multipliers and (N-1) adders which continuously interact with the core internal memory. The
model evaluation logic, instead, is a state machine which reads the whole internal memory in order to extract the data for computing the model accuracy. It contains two unsigned-to-double converters and a 64bit divider.

The task of the external wrapper (i.e. the mdr_block) is to distribute the data among the different mdr_core instances and to retrieve the accuracy of each model under evaluation. The mdr_block communicates with the PPC processor through the PLB bus, it receives the data from the PPC and, when no more data are available, it asks to the mdr_cores the model accuracies, which are then forwarded to the PPC processor.

The PPC processor task is limited to reading the input dataset and converting it to an internal compact format in which each datum is represented by two bits; this compact representation will then be used to provide data to the mdr_block. Also, the processor must generate the combinations (selection phase of the algorithm) and keep trace of the best model found during the algorithm execution.

The effort required to create a hardware accelerator is much higher than the effort required to code a many-core or a GPGPU implementation. Hardware accelerators are created using a hardware description language (either VHDL or Verilog), something completely different with respect to traditional software programming languages. Besides that, the synthesis of an FPGA system requires skills in the field of Embedded Design Automation and a good knowledge of the target architecture.

V. Results

The timing performance of the MDR accelerated implementations were computed using a dataset listing 1000 attributes of 1600 different patients and setting the parameter N to 3; the results found by the three implementations were compared to the results produced by the sequential C implementation in order to verify the algorithm consistency. As an initial remark, the C version runs on a linux workstation equipped with a 2.93 GHz Intel® Core i7-870™ processor and 4GB of DDR3-1,333Mhz RAM and compiled using gcc (optimization level set to 3) was able to complete the computation in an average time of 28,840.42 seconds (the test was repeated 10 times and the standard deviation obtained was 50.45 seconds). The following Subsections report the timing performance obtained running the proposed implementations.

A. Many-core results

The many-core implementation was tested using the Intel Many-core Testing Lab (MTL) [20], a computing environment that allows to use the 32-core processor that Intel will commercialize in the second part of 2011. The C source code was compiled enabling the OpenMP flag and setting the optimization level to 3 with both the GNU compiler, gcc, and the Intel compiler, icc. SSE3 vectorization has not been used. Table IV reports the results obtained using the two different executables and varying the number of threads.

B. GPU results

Both the CUDA-C and the OpenCL implementation were tested using a Nvidia Tesla C1060 board interacting with an Intel Xeon quad processor and a 8GB DDR 3 main memory. The CUDA-C version, as stated before, was developed and optimized in [13]. The original code was just adapted (minor modifications) for the Tesla C1060 GPU. The average running time of this GPU-based implementation is 182.4 seconds; the experiment was repeated 10 times and the standard deviation observed was 93 milliseconds.

The first experiments with the OpenCL implementation were instead characterized by a running time of more than an hour. However, after using the CUDA OpenCL profiler and the CUDA occupancy calculator [21] it was possible to optimize the memory accesses and the device utilization, obtaining an average running time of 204.51 seconds. The experiment with the optimized OpenCL implementation was repeated 10 times and the standard deviation observed was 0.80 seconds.

C. FPGA results

The FPGA-based implementation was tested using the previously mentioned Xilinx Virtex-5 XC5VFX70T FPGA. It was decided to place four mdr_cores within the mdr_block and to synthesize the PLB component using ISE 12.3 [22]. Table V summarizes the data about the resources needed for deploying the component onto the FPGA.

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Used</th>
<th>Available</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice LUTs</td>
<td>5074</td>
<td>44,800</td>
<td>20.25%</td>
</tr>
<tr>
<td>BlockRAM/FIFO</td>
<td>16</td>
<td>148</td>
<td>10.81%</td>
</tr>
<tr>
<td>BUFG/BUFGCTRLs</td>
<td>1</td>
<td>32</td>
<td>3.13%</td>
</tr>
<tr>
<td>DSP/PSEs</td>
<td>72</td>
<td>128</td>
<td>56.25%</td>
</tr>
</tbody>
</table>

The synthesized component is characterized by a maximum working frequency of 249 MHz, so it can be connected to the PLB bus working at 125 MHz without experiencing any synchronization issue (in this configuration the bus guarantees a transfer rate of about 83MB/s). Moreover, the low resource requirements allow to plug the component to the remaining parts of the system without exhausting the chip resources. Table VI shows the data about the device utilization when the entire system is deployed onto the FPGA.
The FPGA-accelerated version was able to complete the computation in an average time of 473.23 seconds. The experiment was repeated 10 times and the standard deviation observed was less than 100 milliseconds.

VI. DISCUSSION AND CONCLUSIONS

This paper describes how the Multifactor Dimensionality Reduction algorithm has been accelerated using many-core processors, graphical processing units (GPUs) and fields programmable gate array (FPGAs). The methodology adopted is based on the timing and data dependency analysis of the source code to identify the most critical parts of it.

All the described implementations outperform the sequential C implementation: the many-core implementation achieves a 14.7x speed-up, the GPGPU implementation a 158.12x speed-up (CUDA-C) and a 141.03x speed-up (OpenCL) while the FPGA implementation a 60.9x speedup. The GPU based implementations allow obtaining the best performance improvement; that is because the MDR algorithm is characterized by huge data parallelism. In particular, the CUDA-C solution is characterized by a 12% improvement with respect to the OpenCL solution. The reason behind this difference is the fact that the OpenCL API is built on top of the CUDA-C API. This introduces overheads that justify the gap between the two implementations [23]. On the other side, the OpenCL solution can scale to non-Nvidia GPUs as well. For what concerns the chosen device exploitation (Nvidia Tesla C1060), the GPU computing resources and the fastest memories (registers and local shared memories) are fully used.

Besides the CUDA-C and OpenCL implementations, the many-core one provides a good speed-up and scales well with respect to the number of cores. In the result reported, in fact, the timing performance scale almost linearly until all the 32 cores are used. Also in this case the computing resources are fully used, the highest speed-up is achieved when there are 64 threads and hyper-threading is exploited. The FPGA-based solution is characterized by a good speed-up too, and, differently from the GPGPUs and the many-core ones, it can be improved without the need of using a more advanced device. The resources of the FPGA employed, in fact, are not fully exploited. Moreover some architectural changes can boost the overall speed-up. By now the PLB bus, a central element of the overall architecture, is the bottleneck that prevents further improvements. A possible solution to overcome this limitation could be to connect the hardware accelerator to the general purpose processor using a different bus, like the Fast Simplex Link (FSL) bus. This choice however requires to switch from the embedded powerPC processor to a soft processor.

In conclusion, even if the asymptotic complexity of the algorithm is combinatorial, the speed-up provided by the proposed MDR implementations can help medical geneticists in conducting exhaustive studies on all the three-way interaction models which can be generated using the nowadays available genotypic/phenotypic data.

ACKNOWLEDGMENT

The authors would like to thank professor Jason H. Moore and his research team for his helpfulness and support.

REFERENCES