Suitability Analysis of FPGAs for Heterogeneous Platforms in HPC

Fernando A. Escobar, *Member, IEEE*, Xin Chang, and Carlos Valderrama, *Member, IEEE*

Abstract—High performance computing (HPC) systems currently integrate several resources such as multi-cores (CPUs), graphic processing units (GPUs) and reconfigurable logic devices, like field programmable gate arrays (FPGAs). The role of the latter two has traditionally being confined to act as secondary accelerators rather than as main execution units. We perform a deep survey around state of the art research and implementation of HPC algorithms; we extract features relevant to each family and list them as key factors to obtain higher performance. Due to the broad spectra of the survey we only include the most complete references found. We provide a general classification of the 13 HPC families with respect to their needs and suitability for hardware implementation. In addition, we present an analysis based on current and future technology availability as well as in particular aspects identified in the survey. Finally we list general guidelines and opportunities to be accounted for in future heterogeneous designs that employ FPGAs for HPC.

Index Terms—High performance computing, FPGA, heterogeneous architectures, design approach, suitability

1 Introduction

Nowadays, there is an abundance of heterogeneous, specialized and powerful processing platforms composed by multi-cores, GPUs, FPGAs and Networks on Chip (NoCs), offering a certain degree of parallelism and scalability. Typically, CPUs are mainly good for control tasks, GPUs for linear vector/matrix operations and FPGAs for streaming applications; mastering these platforms has opened the door to a broad spectrum of research on areas such as computer architecture, parallel programming, algorithm refinement and platform design methodologies [1]. Indeed, to cite only one example, the utilization of many-core architectures such as the general purpose GPU (GP-GPU) is popular today in the grid computing domain, as it merges the advantages of instruction and data parallelism in a single and simple programming language.

Although the majority of studies conclude that GPUs are easier to program than FPGAs and at lower costs [2], other considerations in favor of FPGAs have become relevant in recent years. Recent studies agree that, in order to develop faster and more energy-efficient architectures, algorithmic evaluation should highlight the most suitable memory organisation, operators set and interconnect topology [3], [4], [5], [6], [7]. On top of that, it has been estimated that half the lifetime cost of HPC platforms is devoted to electrical power [4]; a relevant criteria for companies and research centres to consider the use of FPGAs. A similar survey notes that, regardless the system processing speed, major bottlenecks in high performance architectures comes from data transfers, limited by the memory hierarchy and the inter-nodes communication efficiency [8], [9].

The use of FPGAs implies a very low-level design, a much more detailed and slow debugging process, as it goes through logic simulations and post-synthesis verifications. FPGA programming also requires expertise in digital design as well as access to expensive vendor electronic design automation EDA tools. Moreover, FPGAs lack dedicated hard-wired floating point units (typically required in HPC) or, when available, they work at frequencies surrounding a few hundreds of MHz, whilst CPUs/GPUs do it at GHz offering greater operations/cycle. In contrast with general purpose platforms, FPGAs deliver the performance of algorithm dedicated architectures but when new processing is needed, even considering reuse, a new slow and complex design process restarts [10], [11], [12].

First approaches that used FPGAs for HPC (2006) focused on creating multi embedded-core architectures to assemble on-chip computing grids. Assisted by the message passing interface (MPI) programming paradigm, applications were developed as in traditional grid environments. Among their main drawbacks, these platforms cannot exploit all the parallelism capabilities of FPGAs since processors are inherently sequential [13], [14], [15], [16].

In 2008, multi-FPGA architectures designed with traditional hardware techniques and languages such as VHDL/Verilog were proposed [17], [18], [19]. Although effective, they suffered from the aforementioned drawbacks of traditional hardware design. Proposals based on high-level languages and synthesis tools were also reported in with promising results [20].

More recent research [7], [21] focused on library development composed by fine and coarse grain operators to create customizable FPGA-based architectures in shorter time. Industrial approaches have invested a considerable amount of resources in EDA tools at the system level (ESL) encouraging the design of hardware architectures from software descriptions in languages like C/C++ and SystemC. These

• *The authors are with the Department of Electronic and Microelectronics, Universite de Mons, Mons 7000, Belgium. E-mail: {fernando.escobarjuzga, xin.chang, carlos.valderrama}@umons.ac.be.*

Manuscript received 24 Feb. 2014; revised 25 Jan. 2015; accepted 20 Feb. 2015. Date of publication 26 Feb. 2015; date of current version 20 Jan. 2016. Recommended for acceptance by E. Leonardi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2407896

tools, such as Vivado HLS (Xilinx), Catapult C (Mentor Graphics), Dime-C (Natallech) automatically map software functions to FPGA logic. There is also Altera's OpenCL SDK which extends the technique by implementing software code into CPUs, GPUs and FPGAs without additional human effort. Their main drawbacks are long compilation/synthesis times, the volume of design exploration, reuse and results that do not to fit into target devices.

In this survey we present an updated and compact analysis of HPC platforms and algorithms so that it can be employed as reference in future research; we specifically focus on performance and scalability aspects of current platforms. In Section 2, we provide an extension of [5], by showing a deep description of each HPC family, highlighting their bottlenecks, main characteristics and current behaviour in CPU/GPU and FPGA devices according to the latest research available; we summarize our findings with a subclassification of the dwarves. Section 3 presents an analysis that remarks guidelines and opportunities of FPGA technology for HPC. Section 4 lists our conclusions.

2 HPC Families in Software and Hardware

The number of algorithms addressed by HPC is vast, traversing several application domains such as commercial, financial, medical or scientific, among others. While thousands algorithms may exist, common processing patterns can be found amidst them; Berkeley sorted them into 13 dwarves or patterns [5] that hold common characteristics. We extend the this study with up-to-date references and remark their computation tasks, data processed, memory requirements and current bottlenecks for CPU, GPUs and FPGA implementations. A table summarizing key facts is presented at the end.

2.1 Dense Linear Algebra—(DLA)

It covers one of the most important categories of HPC algorithms since a significant amount of problems are formulated and solved using linear algebra concepts. Computation wise, its complexity can be split into three operations: vector-vector $(O(n))$, vector-matrix $(O(n^2))$ and matrix-matrix $(O(n^3))$ multiplications [22]. Data processed by this family can be of integer or floating point nature, in which case, it is usually represented with the IEEE-754 standard; floating point values may have single (SP), double (DP) or (QP) quadruple precision, e.g., using 32, 64 or 128 bits respectively. Libraries like BLAS, LAPACK, CUBLAS, etc, exist to exploit a processing platform capabilities when solving three main problems:

- Linear Systems: $Ax = b$
- Least Squares Minimize: $\|Ax - b\|_2^2$
- Eigenvalues: $Ax = \lambda x$.

Decomposition and factorization methods like LU, Cholesky, QR, or singular value decomposition, are included in the aforementioned libraries with various software routines: *block algorithms*, which divide matrices into sub-matrices to perform panel factorization and updates to ever smaller arrays; update operations are computing-bounded whilst panel factorization are memory-bounded. Additionally, *tile algorithms* divide block computations into several cores leading to an increase of floating point operations per second (FLOPS) compensated with higher parallelism [23]. Task dependencies, matrix size, memory accesses, throughput, numerical stability and FLOPS, limit the speed-up of the previous methods [24].

Matrix multiplication complexity could be reduced using Coppersmith-Winograd $O(n^{2.375477})$ or Williams $O(n^{2.3727})$ algorithms but current hardware lacks efficient and scalable communication schemes [25]. Other research suggests identifying memory stalls of basic BLAS operations and extrapolate their effects on bigger routines to develop, hardware-aware, efficient algorithms [26].

Roughly, the running time on a platform with γ time per FLOP and β time per word treated, is given by

$$\gamma \cdot (\#flops) + \beta \cdot (\#words) \tag{1}$$

but unfortunately, memory transfers are more expensive than FLOPS ($\beta \gg \gamma$): the time per FLOP improves at 59 percent (per year) versus the 26 and 5.5 percent increase in network bandwidth and latency [27].

FPGA approaches have exploited parallelism through pipelining and have focused on the three basic operations as well as in QR and LU decompositions; the latter, usually limited by its low scalability and memory re-use. Reported research in [28] showed sustained DP performance of 8.50 GFLOPS at 133 MHz (matrix size of $16,384$) by integrating fine-grain tasks, loop analysis, blocking algorithms and dedicated memory transfers. Nonetheless, it was still $4.8\times$ slower than their best cited work, a multicore at 3 GHz. Likewise, authors in [29] profited from tiling and column-first processing to obtain 11 GFLOPS at 200 MHz (DP) but again they were 14.96 $\times$ behind their best competitor, a CPU-GPU system (2.3, 1.2 GHz).

In another work, to bypass square root computation in FPGAs for Cholesky decomposition, an iterative algorithm reached 20 GFLOPS at 180 MHz for both SP and DP; however, for matrices bigger than 256^2, it was six times slower than a CPU-GPU system (2.6, 1.4 GHz) [30]. Finally, analysis and models to foresee Dense LA routines performance on FPGAs presented in [31] remark the need to carefully balance pipeline depth with operating frequency for higher performance.

2.2 Sparse Linear Algebra—(SLA)

This group deals with the same mathematical problems but works with *sparse* matrices, that is, with few non-zero values. The challenges are indirect and irregular memory accesses, few FLOPS per access (less than 5 percent of theoretical peak efficiency) and low data reuse. Iterative methods, most belonging to the Krylov space solvers, are used to provide approximate, converging solutions: CG, GMRES, BiCG, BiCGStab, QMR and TFQMR [32]. Software libraries included in this family are MKL, cuBLAS and cuSPARSE among others.

To enhance processing speed, preconditioning factorization is commonly used; it encompasses the incomplete LU (ILU), incomplete Cholesky (IC), modified IC (MIC), block Jacobi, multicolor and polynomial factorization methods [33]. Certainly, the study of preconditioning has increased in the last decade as it helps determining efficiency and

convergence as well as degrees of parallelism [34]. Using them, $3-8\times$ speed-ups for CPU/GPU systems have been obtained [35].

Globally, sparse LA performance is constraint by the sparse matrix-vector (SpMV) multiplication whose complexity, for n_{nz} non-zero matrix elements, is $O(n_{nz}^2)$. Along with it, vector dot product and vector addition also affect the efficiency. Proposed algorithmic optimizations range from bandwidth and latency minimization [36], to direct methods like the *superLU* [37] which processes data blocks with dense LA routines. Lastly, storage schemes have shown direct influence over final performance as they can be customized to exploit hardware architectures; accelerations of 20-30$\times$ in GPUs versus multicore have been reported using this technique [38].

Efforts to implement SpMV in FPGAs usually focus on parallel and partial computations that are later combined; recently, with a column-wise approach and a custom memory manager, a Virtex-5 design reached 17.2 GFLOPS (SP) and 91 percent efficiency at 150 MHz, indeed, $4-9\times/1-3\times$ faster than the reported CPU/GPU solutions [39]. Similarly, with an efficient communication-computation scheme authors in [40] provide mathematical models that show up to 4$\times$ improvements of FPGAs over optimized GPU designs for the SpMV. Furthermore, a single-precision, pipelined CG iterative solver achieved up to 35 GFLOPS at 287 MHz and was 1-35$\times$ faster than the optimized CPU reference design [41]. Recent works also include an automated C-to-HDL synthesis of a (SP) Jacobi method in a heterogeneous platform with a 3$\times$ wall-clock speed-up over software [42]. Fixed-point designs are also possible solutions under flexible error margins, as in a Lanczos Kernel that can hit Tera Operations Per Second (TOPS) at 400 MHz in a Virtex-7 [43].

2.3 Spectral Methods—(SM)

In certain applications like fluid dynamics, quantum mechanics or weather prediction, partial (PDE) and ordinary (ODE) differential equations, as well as, eigenvalue problems are solved in the spectral domain. Solutions are expressed as a finite, high-order polynomial expansion of basic functions [44]. Different methods are used to treat PDEs; *collocation* ones must satisfy a set of collocation points whilst *Galerkin* and *Tau* methods work equations in their *weak* form. Most of them are Fourier-based and are recommended for periodic problems. On the contrary, the Lengendre and Chebyshev polynomials are better suited for bounded domains [45].

Although some algorithms use the Wavelet Transform, most employ a fast fourier transform (FFT) kernel. The FFT has a fixed complexity of $O(nlog_2 n)$ for an n-points array. In practice bottlenecks arise from data access since the output depends on all inputs, thus, memory can be overflown for big arrays. In addition, the dimensionality increase (2D, 3D FFTs) leads to all-to-all interconnections that degrade performance. Improvements are mostly focused on cache memory optimization, vectorization and parallelism.

Chip vendors offer 1D-FFT cores for computations in FPGAs. Compared to them, with a 12.5 percent slower performance, research approaches have proposed stream-based, row-column 2D-FFTs with pre-fetching and DMA engines for on-off chip data transfers [46]. Contrastingly, a block based solution for matrices of up to $4{,}096^2$ size and 24-bit data at 100 MHz, was up to 1.95$\times$ faster than row-column counterpart FGPA designs [47]. For 3D-FFTs, an architecture for hard butterfly blocks for FPGAs, estimates improvements of $300-1{,}000\times$ at 300 MHz over an Intel Core 2 Duo at 2GHz; moreover, the study sustains that with higher off-chip memory bandwidths, speed-ups will be constraint by on-chip memory resources [48]. Other designs for latest FPGA devices show they can outrun GPUs and Sandy Bridge CPUs in 3D-FFT computations, by $1.3\times$ for matrices of up to 32^2 and 64^2 respectively [49].

2.4 N-Body Methods—(NBM)

It contains algorithms that compute particle's interactions due to gravitational forces exerted on each other. The formal problem definition [50] is to evaluate

$$f(x_i) = \sum_{j=1}^{n} K(x_i, u_j)s(y_j), i = 1, \ldots, N \tag{2}$$

where f is the potential, s the source density, x and y, the particles coordinates and K the interaction kernel. For an n particle system, the computational complexity is $O(n^2)$ with traditional methods. To ease the workload, the Barnes-Hutt, $O(nlog(n))$, and the fast multipole method (FMM), $O(n)$, are currently employed. The first one arranges data in a tree structure where the leaves represent the particles and the node or *pseudo-particle* acts as their center of mass.

The Barnes-Hutt procedure can be divided into four steps: the space is split into squared regions of equal sizes containing at most one particle (1), the *pseudo-particle's* mass and position are computed (2), the interaction force is evaluated (3), and all particle's positions and velocities are updated (4) [51]. To determine whether the tree should be opened or not, a criteria based on distances is employed. Similarly, FMM performs the same procedure but, when sufficiently separated, interactions are computed between clusters; this way, all particles inside it are updated with the resultant force effect of the distant group. Four types of interactions are found in the FMM method: cluster-cluster (*V interaction*), cluster-body (*W* and *X interactions*) and body-body (*U interactions*) [52].

Sequential CPU profiles show around 97 percent usage consumed in force evaluation (Eq. (2)) [51], which is also affected by *if-else* statements that lead to irregular memory accesses. This, along with scalability issues, defy parallel architectures like multi-core CPUs and GPUs [51], [53], [54]. The latter also suffers from portability and automation issues as they do not support OpenMP directives and require explicit data transfers. Their *single instruction multiple data* (SIMD) structure also makes it harder to handle algorithm conditionals; nonetheless, some of the latest platforms offer a *voting* function that reduces condition evaluations from all threads.

Cluster speed-ups are also undermined by network bandwidth and communication protocols usually dominated by the message passing interface. Among the studies reviewed, up to 4.45 PFLOPs have been attained with a careful selection of MPI primitives, load balancing

techniques and, algorithm based, physical distribution of computation in a cluster [55].

There are not many FMM implementations on FPGA due to the complexity of square root and division operations. Work in [56] synthesized two of its processing stages using MATLAB Simulink and Xilinx SysGen on a Virtex-5 at 160 MHz, obtaining $60\times$ speed-up over a dual-core CPU at 2 GHz. Other comparisons have shown FPGAs $60\times$ slower than GPUs yet with $15\times$ better GFLOPS/Watt [57]. Coinciding with these findings, N-Body FPGA designs mainly use fixed-point data and can be up to $14\times$ faster than CPUs but always slower than GPUs [58]. More recent solutions include a soft-vector processor (100-200 MHz) integrated with dedicated pipelines synthesized from C code for force computation. Overall, it was as fast as an optimized Intel Core i7 version at 3 GHz [59]. A MapReduce scheme, with slave GPU and FPGA boards over PCI Express (to compute particle's acceleration), found equal FPGA-GPU performance [60].

2.5 Structured Grids—(SG)

Includes algorithms where data can be arranged into n-dimensional arrays that are directly related with the problem physical geometry. Each cell value is updated according to its interactions with surrounding elements, as in the Von Neumann and Moore neighbourhoods, establishing repeated and regular memory access patterns. The iterative processes are called *Stencil codes* and the cell update formula is called *Stencil*. Jacobi and Gauss-Seidel kernels are the most used ones.

Since all points' values can change at each time step, performance bottlenecks for multi-core architectures arise when this number exceeds cache size [61]. In GPUs, limitations are found when updating global (off-chip) memory since these transfers are much more costly. In addition, synchronization to ensure GPU threads to read data after it has been refreshed are also stalling factors [62]. In distributed systems, performance falls because of data dependencies between compute nodes that require transfers at each time step to maintain coherency [63].

Certain single node optimizations are oriented to cache reuse and, automated, aligned storage layouts for efficient memory access [64]. Considerations to the influence of the type of stencil on memory behaviour are also important: Jacobi computations require data to be read from one location and written to another while Gauss-Seidel use the same memory space [65]; nonetheless, all improvements have to be tuned platform wise as their memory hierarchies can lead to different results [66].

Molecular dynamics (MD) computation using FPGAs has been a common practice for some years; an implementation of the Lattice-Boltzmann method on a 2-FPGA system obtained (SP) 36.5 GFLOPs at 125 MHz for a $2{,}880 \times 960$ grid; no comparison with SW is reported though [67]. In other studies, theoretical estimations of a 2-D FPGA mesh array that uses inter-node serial communication and pipelining for Stencil computations, estimates 2.24/573 GLOPS on a per-node/256-FPGA system at 160 MHz [68]. Finally, a, bandwidth independent, scalable and pipelined architecture, with serial connections intra-FPGA and dedicated memory managers, hit 280 GFLOPS peak performance and 83.9 percent efficiency for 2D and 3D Jacobi stencil computations. Compared with an optimized Intel Core i7 version at 3.2 GHz, it was $13.7\times$ faster, with estimations of 17.7 TFLOPS with 100 (newer) FPGAs [69].

2.6 Unstructured Grids—(UG)

This family differs from the previous one because elements geometry cannot be easily organized in regular structures. Data is therefore stored in custom formats leading to irregular accesses. Added to the intensive arithmetic needs, extra time for address computation and data retrieval is required. In some cases, structure irregularities are stored using graph formats and data is recovered with breadth-first (BF) and depth-first algorithms.

FPGA-based solutions have included data compression and encoding to reduce on-off chip bandwidth needs conjointly with mesh sorting to reduce memory transfer time and avoid processing stalls due to data dependencies [70]. Similarly, using quantization to reduce data size and other compression techniques, a 34 percent decrease in memory traffic and a potential $5.2\times$ faster execution versus cached-based systems is reported by [71]. Within possible overheads, authors highlight data preprocessing, low FPGA resource utilization and board-to-board transfer times. Lastly, an FPGA cluster (FLOPS-2D) has been also used to accelerate the FaSTAR package for unstructured mesh, fluid dynamics computations; read-after-write hazards that cause pipelining stalls are alleviated through out of order executions and FIFO-based synchronization. A $2.95\times$ acceleration over an Intel Core 2 Duo and $2.6\times$ over in-order execution is obtained [72].

2.7 Map Reduce—(MR)

Includes distributed algorithms that are implemented in a cluster to treat large databases with no strong dependencies allowing maximum parallelism. Initially, the *map* procedure splits the problem into sub-problems and distribute them over the workers, then in the *reduce* step the master gets the answers form the slaves and produces the final output. Popular applications are Monte Carlo simulations and search engines like Google or Yahoo!. A distributed file system is employed in this family as almost no data is shared between nodes.

The most popular MR implementation is called Hadoop which features automatic and scalable parallelization. It does not provide the programmer low-level directives to establish how to split the jobs; it also assumes a full and dedicated cluster control, which is usually not the common case in HPC systems [73]. Any algorithm that can be divided in parts is a candidate to embrace this pattern (e.g., matrix multiplication, page rank, machine learning or data mining) [74], however, MR has mainly been oriented to supply Web services. CPU architectures have been enough for most applications yet GPUs have been increasingly employed with certain adjustments. Recent proposals offer automatic offloading of parallel computations to GPUs [75], [76]. Its use is recommended when the application has many loops, few conditional branches, low data dependency per iteration and when it is of numerical, graph or text type [77].

RankBoost was implemented on FPGA with hand-coded accelerators connected through PCI express to a CPU and with parallel FIFO interfaces for DRAM access; at 125 MHz a $33.5\times$ speedup over a Pentium 4 at 3.2 GHz was reported and FPGA size along with bandwidth were identified as bottlenecks [78]. A cluster composed by CPUs connected to FPGAs through Ethernet links used Linux virtual tasks and UDP packets for workload and data sharing with FPGAs [79]. A report of $10\times$ speed up for matrix multiplication and $4\times$ for page rank is obtained at with 3 FPGAs at 125 MHz versus an Intel Core Duo at 2.66 GHz. A scalable approach for CPU-FPGA clusters (125 MHz) with asynchronous MR updates, that compared n-CPU versus n-FPGA systems, reached quasi-linear speed-ups from $0.7 - 40\times$ against Maiter and up to $360\times$ against Hadoop. Authors used three different applications, generic processing element replication, direct memory access and multiple network interfaces [80].

2.8 Combinational Logic—(CL)

Database related operations like cyclic redundancy check (CRC), hashing and (de) encryption, are relevant computations in electronic transactions and cloud computing for higher privacy and security. Cryptographic algorithms transform data using a *key* for encryption and decoding through two main operations: *confusion*, whose intend is to make the relationship between the ciphertext and the key, as complicated as possible; and *diffusion*, which does the same with the plaintext (data) and the ciphertext [81]. Hashing on the contrary skips the decoding step as it is used in validation (password checking) rather than in data communication. Popular algorithms in this category include the DES, AES, RC4, SHA-1 and RSA.

The basic AES algorithm divides data into 128-bit blocks arranged as 4×4 matrices and applies 11 rounds of bitwise manipulation. In the first one an XOR (exclusive or) operation between the matrix and the key is performed; the following nine iterations execute byte replacements, row shifting, column mixing and *subkey* adding (xor); the final iteration skips column mixing [82]. Bottlenecks arise with the amount of data to be processed and, since logic operations are the simplest and fastest ones, architectural improvements have been added to CPUs to increase performance. GPU limitations come from data transfers and memory sizes [83].

Extensive work in FPGAs has been invested in this family. A thorough review of AES implementation techniques like pipelining, loop unrolling, Look-Up Table based, Mix-Column, less bit datapaths, etc. is shown in [84]; authors apply two-slow retiming to obtain 86 Gb/s throughput at 550 MHz. Similar research is reviewed by [85] for SHA functions and multi-mode architectures on FPGAs; their proposal merges SHA-1 with SHA 256/512 in a general architecture with $12 - 16$ Gbps throughput. Finally, [86] shows a CPU-integrable coprocessor for AES-128 encryption with parallel pipelined units and 45 Gb/s peak throughput, $4.38\times$ faster than an UltraSPARC T2.

2.9 Graph Traversal—(GT)

Several applications that involve data relationships can be solved through graph traversal/exploration algorithms. A graph (G) can be defined as a data structure of nodes (vertices V) interconnected by links (edges E). Formally, it is defined as

$$G = (V, E \subseteq (V \times V)). \tag{3}$$

When all vertices denote the same kind of objects and edges represent the same type of relationships, the graph is *single-relational*, else it is *multi-relational*. Some applications of the first kind are applied in certain social relations (people interactions), biological (between proteins, food, etc), transportation fields (routes-places), etc. The second type can be found in the Web of Data, social networks, etc, where multiple connections are created between data for efficient access. The increasing amount of information added to the need to classify it, has promoted the use of these algorithms.

Two main methods are employed to traverse a graph: the *breadth-first search* (BFS) and the *depth-first search* (DFS). BFS performs a *layer by layer* exploration by first looking at same-level nodes (distance from the origin) i, whilst DFS only goes back if it cannot go deeper. Due to its wide applicability, BFS has been selected as a Graph500 benchmark, which ranks supercomputers according to their traversed edge per second (TEPS) metric. The trade-off between them is called *Best-First Search*.

BFS usually suffers from excessive data access and show irregular structures that lead to poor locality [87]. Low amount of arithmetic computation, constant synchronization, load imbalance and long-latency memory access degrades performance in current multi-core and GPU platforms [88]. Sequential BFS examines vertex by vertex labelling them with the distance to the first node; for a graph with n vertices and m edges, it has a complexity of $O(n + m)$. The *quadratic parallelization* optimization inspects every edge at each iteration with a $O(n^2 + m)$ complexity and it's mainly employed for GPU and multi-GPU implementations.

Implementing BFS on FPGA has included full graph on chip storage, graph-shaped circuits, and multiple DRAM/SRAM usage [89]. Repetitive pipeline stalls caused by irregular memory accesses are compensated with several (runtime scheduled) parallel graph processing units and concurrent memory accesses; up to $10\times$ better performance versus an Intel Xeon (quad-core) CPU at 2.5 GHz was obtained with four Virtex-5 FPGAs at 150 MHz [90]. Using custom sparse storage formats and a cyclic, token synchronized, network of processing units on a similar platform, it was $5\times$ faster than the previous work [91].

2.10 Dynamic Programming—(DP)

Problems that can be solved through overlapping subproblems lie in this category. Computations are data and context dependent and it's difficult to establish global operations that applies to all algorithms. Dynamic programming divides a problem into situations/states (x_i) over time that determine the choices/actions (a_i) to take in each sub-problem.

Other applications such as graph traversal also employ DP algorithms like the Bellman-Ford and Floyd Warshal ones. DP has also been applied to bioinformatics in sequence alignment, protein folding, RNA structure

prediction and DNA-binding. In general DP algorithms can be classified based on two characteristics: if problems at a stage only depend on solutions from the previous level (serial) and if the right hand side of the optimization equation contains only one recursive term (monadic); this leads to four groups in total [92]. For an n sized problem the time complexity in the serial (SMDP) and non-serial monadic (NMDP) categories is $O(n^2)$ while for serial (SPDP) and non-serial polyadic (NPDP) is $O(n^3)$ [93]; difficulties in data dependencies, memory access and load balancing are more complex in the latter [94].

Most solutions for speeding up NPDP algorithms focus on cache reuse and employ block partitions to maximize parallelism; since problems use matrix forms, cache size is one of the major performance limitations. Data layouts for more efficient on-chip memory access have reached 44-fold improvements on modern CPUs [95] and it has been applied to GPUs too [94]. Fine grained parallelism in GPUs has included assigning parallel and independent operations to GPU block threads while coarse grained methods use bigger computations per thread. Synchronization between threads from different GPU blocks, un-coalesced and indirect memory accesses, and tiling, stand as the main research topics [96], [97].

FPGA designs have been mostly oriented to bioinformatic applications using systolic arrays and pipelining with $20\times$ faster performance than CPUs [98] with 1.75 Giga Cell Updates per Second (GCUPS) and up to 24.7 [99]. In addition, using a divide and conquer algorithm, an architecture for the knapsack $0/1$ problem achieved 10 times enhancements over a Pentium 4 at 3.2 GHz on a Virtex-II [100]. In general design re-use is a major drawback when using FPGAs in DP as the scope of problems treated with it is vast and can require many different architectures. Moreover, the recursive nature of DP requires creating complex management units to handle off-chip memory as done by operating systems in CPUs.

2.11 Backtrack and Branch-n-Bound—(BB)

Combinatorial optimization problems whose solution exists in a very large space can be solved by dividing them into subregions (branching) and computing bounds in each of those subspaces until a (near-) optimal result is obtained. The algorithm can be split into four phases [101]: Selection, where the next node to be explored is chosen according to a *depth-first* strategy with variable memory and less sub-problems (used in backtracking), *breadth-first* one, with large memory and more runtime or *best-first* stragey with big memory and shorter runtime. Branching is the action of decomposing the solution space into two or more disjointed subspaces and bounding the one of computing the bound value of each branch. Prunning is used to eliminate the subproblems that exceed the best solution found.

A coarse grained taxonomy of parallel B&B algorithms proposed long ago divides them into four categories: Synchronous single pool (SSP), where all processors are synchronized to update the bound information and share a pool of sub-problems; asynchronous single pool (ASP) where they communicate at their will. Synchronous multiple pool (SMP), where each processor has its own sub-problem pool and communicates synchronously, and lastly asynchronous multiple pool (AMP) where transmissions are done asynchronously [102]. Synchronization provides more efficient prunning but incurs in communication excesses, degrading performance. The same considerations hold for single and multiple pool scenario. Continuous bound updates better suit shared pool systems.

Due to the irregularity, unpredictability and amount of data treated in this family, GPU implementations are scarce. Relevant works have focused on optimizing thread divergence, memory access and GPU-CPU communication; since tree creation is a very irregular and uncertain task, load balancing and synchronization remain the most challenging issues to port applications to high-end platforms [101], [103].

Very few FPGA solutions are found for the B&B family. A method for solving the maximum clique problem with a compressed data storage format, simple logic circuits and pipeline registers was more than $1{,}000\times$ at 40 MHz faster than a Pentium 4 at 2.4 GHz according to the reported data in [104]. Other optimizations treated with B&B like the quadratic assignment problem, has been solved through Tabu Search which maps easier to FPGA technology reaching speed-ups of $400\times$ versus the same CPU configuration [105]. Backtracking algorithms like the Boolean Satisfiability Problem was implemented on FPGA with parallel, pipelined, deduction engines [106]. Authors propose the use of runtime reconfiguration to modify the amount of engines to get speed ups between 1-5 versus single engine case.

2.12 Graphical Models—(GM)

Bayesian Networks, hidden Markov models (HMM) and Neural Networks are included in this family. Graphic representations with directed acyclic or undirected (cyclic) graphs are used to predict or explain situations or phenomena. In Bayesian networks, edges represent dependencies between nodes which are associated to probability functions; three mathematical concepts are iteratively employed to find the required probabilities in the network [107]:

- Bayes theorem: $P(H \mid E) = \frac{P(E \mid H) \cdot P(H)}{P(E)}$
- Marginalization: $P(H) = \sum_{g_i} P(H \wedge G = g_i)$
- Factorization: $P(V) = \Pi_{i=1}^{n} P(V_i \mid \pi(V_i))$.

Algorithms focus on finding *most probable explanation* (MPE), computing the *belief propagation* (calculate marginal distribution on unobserved nodes by inference) and *learning Bayesian Network* structure as in the Markov chain Monte Carlo (MCMC). Learning a Bayesian Network, i.e., finding its graph structure and parameters, is an exponentially growing problem that depends on the number of variables (NP-Complete) [108]; an exact inference requires $O(n^2 \cdot 2^n)$ time plus $O(n \cdot 2^n)$ scoring function evaluation (to select the right network) [109]. As there are several choices to select a scoring function, parallelization may vary from one approach to another; some GPU approaches compute this function in parallel [110].

Belief propagation belongs to the *factor graph* group and focuses on message transmission between graph edges; its complexity increases with the amount of variables states and nodes. For an acyclic graph with m factor nodes, d

neighbours per-node and r nodes, the complexity is $O(m \cdot d^2 \cdot r^d)$ and the result is known to be exact. Independent computations on non-shared data and storage formats in multi-core architectures have been proposed for speeding up the acyclic scenario [111]. In cyclic graphs, where messages are iteratively passed between nodes, results are approximated; the algorithm is known as embarrassingly parallel and falls in the Map Reduce family.

In HMMs a process can take a *state* at each time t that is determined by the one at time $t-1$. The model is defined by *transition* and *emission probabilities*; the first one refers to the next state probability given the current one, generating an $N \times N$ transition matrix (Markov Matrix); the second term is the probability of the value taken by a variable y given that it is in state x.

In summary, the abundance of applications enclosed by graphical models limits possible generalizations to identify bottlenecks and drawbacks.

An FPGA bayesian learner on a Virtex-5 with deep (>20 stages) pipelined nodes, automatic block schedulling, and content-addressable memories was $80\times$ and $15\times$ faster than an Intel Core 2 Duo at 2.4 GHz and a GeForce 9400 M, yielding 20.4 GFLOPs [112]. Neural networks designs on FPGAs have used fixed point data, parallel LUT-based neurons and shared arithmetic units for higher parallelism as in [113], where $41\times$ enhancements were obtained versus an Intel Quad CPU at 2.4 GHz. Artificial neural networks with coarse-grained arithmetic pipelines, different data bit-width and temporal on chip storage estimate speed-ups of $110\times$ versus a dual core CPU at 2.4 GHz on different FPGAs [114].

2.13 Finite State Machines—(FSM)

Encloses processes where the system takes only one state at a time. Inputs determine the next state as well as the outputs at each one. Parallelism is exploited in the computation of transition conditions and state outputs.This family covers a wide spectra of systems.

Some FSMs are used in speech recognition requiring probability computation, FSM composition and minimization with million transitions and states [115]. Other applications include expression matching, text processing, Huffman decoding and feature extraction.

In cases where the amount of states is huge, redundant ones are eliminated [116]. In other approaches like *non-deterministic finite automatas* (NFA), more than one state can be activated at the same time and transitions do not require input changes; the objective in those cases is to obtain a *deterministic finite automata* (DFA) through FSM partitions and dependency reduction algorithms [117].

Some parallelizations focus on speculative transition simulations sometimes with high computation overheads, but exploiting the benefits of parallel machines [118]. Supplementary studies have developed algorithms to profit from multi-core and shared memory architectures. In text pattern matching, input data is split to run a FSM separately; afterwards, sub-results from each execution is joined [119].

A method to synthesize parallel FSMs on FPGA is shown in [120]. A review of speech recognition on hardware and a solution using weighted finite state transducers presents memory size and latency as a big challenge; node reduction and adaptive prunning are proposed to alleviate this drawback [121]. For the same case, duplicated dual port memory, hash tables and pipelining are employed with $3.5\times$ improvements over previous solutions in [122].

2.14 Summary

Dwarves mainly suffer from memory related issues that have been treated in software layers. GPUs have boosted the treatment of the first seven ones with their floating point flexibility conjointly with a powerful vector architecture. On the other hand, the main CPUs advantage has been their ability for memory management and (shared, complex, dynamic) data structure handling required by dwarves 9-12. FPGAs have been mostly used in dwarf 8 that better suits its operators' granularity. As floating point performance are common drawbacks for CPUs and (specially) FPGAs, GPUs lack of network interfaces and slow PCI-express connectivity, has limited their data exchange rates affecting their scalability and efficiency (sustained/peak performance). Nonetheless, upcoming NVidia's NVLink could change this scenario with estimates of $5-12\times$ faster CPU-GPU and GPU-GPU data exchange rates. Correspondingly, to overcome network protocols and memory latency issues, remote direct memory access (RDMA) (present in InfiniBand), where accelerators need no CPU intervention to retrieve memory from other devices, has been proposed. Currently, FPGAs dedicated high-speed links can support this technology.

Many FPGA works have not been included in our survey as they comprise heuristics like Genetic Algorithms or procedures that enclose different dwarves. We finalize this section with Table 1, where the main information from this review is classified according to our criteria for FPGA developments.

3 FPGA Opportunities For HPC

Contrary to CPUs/GPUs, to use FPGAs, a processing architecture must be first created, simulated, debugged and synthesized. As stated above it implies specialized skills to deal with synchronization issues, bit-level operations, etc. To cope with this, industry as well as academy now offer high level synthesis (HLS) tools can generate HDL from C/C++ source code yet they may incur in long compilation times, size restrictions and can require certain hardware expertise to setup the platform.

Moreover, FPGA designs usually fail in offering standardized and abstract models that can be easily used by the non hardware-specialized community; besides, due to the broad spectra of HPC applications, these considerations need to be more carefully studied to provide a competitive processing infrastructure. Developers need to evaluate whether providing customised user/application solutions is more practical than using an already available library. Indeed, the success of CPUs is due to the widespread adoption of vendor libraries tuned to maximize the effectiveness of their hardware architectures. GPUs, initially conceived for graphic processing, have also succeeded in HPC thanks to their processing power and library abstraction along with their offered tools for quick user development.

TABLE 1
HPC Families Profile and Classification for FPGA Environments

Dwarf	Main Algorithms	Core operations	Performance Exploitation Strategies	Memory Access	Current Limitations
GROUP 1: High arithmetic computation + fairly regular memory access patterns. Most efficient in: GPU					
DLA	Cholesky, LU, QR, Eigenvalues, SVD	Vector-vector, vector-matrix, matrix-matrix multiplication	Systolic arrays, blocking, tiling*	Regular	Area expensive FPU, square root and divider units. OnChip memory size, OffChip bandwidth.
SM	Collocation, Tau, Garlerkin	FFT and Wavelet transforms	Prefetching, blocking, DMA, load balancing, cache policies*	Non sequential	Area expensive FPU, memory size and hierarchy*
NBM	Barnes-Hutt, FMM, tree construct	Dot product, divisions, square root	Fixed point data, load balancing, data sorting	Shared	Area expensive FPU, square root units, memory access, network bandwidth, conditional execution*
GROUP 2: High arithmetic computation + irregular memory access patterns. Most efficient in: FPGA					
SLA	Direct (SuperLU), Iterative (Krylov)	Matrix-vector multiplication sparse	Fixed point data, preconditioning*, data storage format, DMA	Irregular	Area expensive FPU, low FLOPS/access
SG	Jacobi, Gauss-Seidel	Arithmetic	Single precision floating point, blocking, tiling*, storage format*	Repetitive, shared	Area expensive FPU, memory size, synchronization, network bandwidth
UG	Jacobi, Graph algorithm	Arithmetic, graph search	Blocking, compression, encoding, tiling*	Repetitive, shared	Area expensive FPU, OnChip memory size, synchronization
GROUP 3: Low arithmetic computation + complex memory handling. Most efficient in: Multicore					
MR	Web services, Monte Carlo	Compare, Count, Search. Algorithm dependent	Node count, hardware coprocessors*, custom network protocols	Independent, application dependent	Node count, design re-use, CPU dependency, network bandwidth.
GT	BFS, DFS	Memory access	Node count*, parallel memory access, graph-shaped circuits	Data dependent	OnChip memory size and irregular access
DP	Non/Serial- Mon/ Polyadic	Matrix fill, backtracking	Systolic arrays, blocking, recursion*	Irregular	Memory size, design re-use, recursion
BB	Tree construction, memory access	Branching, Bounding, Prunning	Load balancing	Unpredictable	Design re-use, complexity, synchronization, memory access
GM	Bayesian Network learning, MCMC, Belief Propagation	Scoring, probability function evaluation	Data parallelism	Repetitive, dependencies	Area expensive FPU, design re-use
GROUP 4: Low arithmetic computation + operators directly map to FPGAs. Most efficient in: FPGA					
CL	Cryptographic	Logical	Custom data width, loop unrolling, data partition	Regular	Logic resources, node count
FSM	Pattern detection, decoding	State transition, Output computation	Custom memory access, data partition, speculative analysis*	Irregular, Data dependent	OnChip memory size, design re-use

Pipelining is employed in all of them. Efficiency is defined as sustained versus peak utilization percentage.
** Software related item.*

Based on the previously established requirements we consider that FPGA designs need to equally focus on two aspects that will determine its adoption or rejection in the HPC world: *user view*, related to the software environment, and *platform efficacy*, determined by the hardware. Complementing and extending the versatility provided by the software through simple, abstract, scalable and portable constructions, can facilitate their access to end-users. FPGA based systems should support user exploration through algorithmic evaluation, data formats and flexible programming approaches. Services that automatically exploit FPGA internal characteristics, adapt to technology changes and allow upgradable enhancements can also promote their adoption in the field. To see the potential of FPGAs, a brief description of the technology and its perspectives is presented below.

3.1 FPGA Technology for HPC Platforms

FPGAs offer a diversity of resources that serve different purposes. In line with the survey, we classify the most relevant ones in two groups:

1) *Communication oriented*
   a) Internal RAM (BRAM).
   b) Distributed memory (registers).
   c) Serial transceivers.
2) *Computation oriented*
   a) Logic cells (LUTs).
   b) Hard-wired CPUs.
   c) Digital signal processing blocks (DSP).

The combined use of those elements facilitates parallelism exploitation; for instance, time parallel or pipelined

TABLE 2
Suggested Habits in FPGA Designs for HPC for Future Developments in Terms of Performance (P), Scalability (S), User Accessibility (UA), Design Reuse (DR) and Portability (Po)

Focus	Objective	Contribution				
		P	S	UA	DR	Po
Adoption of Software Practices						
Library based hardware	User accessible, pre-synthesized modules			•	•	•
Task oriented hardware	Simpler abstractions with automated parallelism			•	•	•
Runtime schedulers in HW/SW	Automatic and efficient hardware management	•	•		•	•
Automatically scalable designs	Sustained performance with low effort	•	•		•	
Parametrizable designs	Customization and re-use	•			•	
Automatic code generation	Lower programming effort and design time		•	•		
High-level abstractions	Lower programming effort with high performance		•	•		
Benchmark comparable systems	Establishing performance versus high-end systems	•		•		
Updatable/Upgradable designs	User friendly architectural support	•		•	•	
Bandwidth performance independency	Easier and faster performance increase	•	•		•	
Hardware Aware Designs						
Memory disposition exploitation	Designs that profit from physical BRAM location	•	•			
Memory structure exploitation	Use of dual-port, asymmetric RW widths, FIFOs, etc	•		•	•	
Area balanced pipelining	Modules with low performance loss in bigger designs	•	•		•	
Off-chip memory management	Dedicated computation-communication overlapping	•	•			
Custom network interfaces	Use of high performance serial links in clusters	•	•			
High-level synthesis	Faster module development		•	•		
Partially reconfigurable modules	Real exploitation of FPGAs unique capabilities	•		•	•	
Algorithm Optimized Architectures						
Balanced arithmetic precision	Smaller circuits with just enough arithmetic precision	•	•			
Mathematical exploration	Identify faster algorithms, create new architectures	•	•			
Communication avoiding algorithms	Automatic performance with technology changes	•	•			
Problem geometry considerations	On/off chip architectures with efficient communication	•	•		•	

platforms require registers, logic cells and usually DSP (multiply-accumulate) blocks. BRAMs, in addition to storage, can be used as flexible memory interfaces such as FIFOs, scratchpads, shared, single or dual-port memories. Transceivers are used for inter-chip data transfers and CPUs for general purpose processing and user interfacing.

As equally important for high performance, the two major FPGA vendors, Xilinx and Altera, have nearly doubled these resources at every generation of their (high-end) Virtex and Stratix devices: consider Virtex-5 and Stratix-III that increased from 18 and 16 Mbits (∼4 MB) BRAM in 2010 to 85 and 52 Mbits (∼20 MB) in Virtex-7 and Stratix-V (2014). Compared to CPUs, these values are much bigger than cache capacity but they lack their coherency and automation abilities. Equivalently, on these two devices, DSP count has increased from 1,056 to 3,600 in Virtex-7 while Stratix passed from 896 to 704 multipliers combined with 399 DSPs. Logic gate evolution changed from 330 K to 2 million (2 M) in Virtex-7 and 1 M in Stratix-V.

The market also offers heterogeneous systems on chip (SoCs) that merge an FPGA portion with a (hard-wired) multi-core ARM, memory controllers and peripherals. Presently, both companies integrate a dual-core ARM Cortex-A9 that can run at 800 MHz maximum. Future announced devices also foresee great improvements in HPC applications: Altera has promoted their 10 series Stratix family with 64-bits, quad-core ARM Cortex-A53 at 1.5 GHz, more than 4 M logic elements and hardened (SP) FPU units, along with many other advances; in fact, they anticipate around 10 Tera FLOPs (SP) on a single chip. On the other hand, Xilinx UltraScale FPGAs offer 4.4M cells where 10, soft ARM Cortex-A9s at 50 MHz can be programmed [123]. They keep doubling the amount of resources but to this date, their next SoC system has not been presented.

The above statistics show how in the last years FPGA manufacturers have strengthen their portfolio to offer more competitive architectures for HPC applications. Also, it seems the trend of synthesizing CPU cores like NIOS or MicroBlaze on FPGA logic is being replaced for the utilization of hard-wired, powerful units like ARM, at least in this field. However, the provision of such a large amount of unconnected resources is but half way in conceiving efficient and competitive architectures for HPC. Hence, the next section will point out the opportunities to better exploit these advances guided by the survey we conducted.

3.2 FPGA Design Guidelines for HPC

In general the limits of reconfigurable logic can not be easily established for most of the dwarves presented. Although their peak FLOP value is much smaller than GPUs, their efficiency (sustained/peak performance) is higher in irregular memory access dwarves (Group 2), thus producing faster results. FPGAs advantages in families that require per-algorithm analysis as in Group 3 are equally uncertain.

Moreover, FPGA benefits can be diminished by the resource investment in routing purposes and synthesis of blocks unavailable in hard-wired form. Recent press critics [124] sustain that FPGA datasheet numbers are over estimated and cannot be fully attainable. Expert designers coincide that underutilization of FPGAs is a common industry

standard and designs still require "exhausting trial-and-error iterations". The last can also be true for CPUs/GPUs and sometimes hardware knowledge is needed to fully exploit their processing power.

In order to find opportunities for FPGAs in the HPC world, we will begin by highlighting some of the most effective approaches encountered through this research. As the success of software platforms is indisputable, our first group of suggestions relates to the *adoption of software practices* that increased productivity with different programming approaches. We also identified frequent attention to *hardware-aware designs* i.e., moulding the architecture according to the underlying hardware for maximum exploitation. Finally, *algorithm-optimized architectures* also represent a unique opportunity to FPGAs due to its inherent flexibility.

Five main factors in which reconfigurable logic can enhance performance in HPC (performance, scalability, user accessibility, design re-use and portability), are used to qualify the guidelines in Table 2. Most suggestions are framed around performance and scalability issues as they are key to the field. We also believe that for this area, user accessibility to the platform is a determining factor for the adoption of the technology. Lastly, design re-use and portability remain challenging issues for FPGAs, specially for complex data handling (trees, lists, structures, etc) usually applied in Group's 3 dwarves.

Table 2 includes software practices that have been boarded through different programming models (structured, object-oriented, actor-oriented, task-based, etc), contributing to better software productivity. Complimentary to the HLS approach methodologies can focus on the listed items for better productivity. On the other hand, what concerns hardware-aware practices is more a summary of what engineers have used through the years for FPGA development. The third part of the list are a general suggestions we found in the state of the art review that apply for both hardware and software platforms.

Despite the effort level invested in FPGA development, their overall enhancements can also depend on *the way* it is employed on a platform: if used merely as an accelerator, progress is more likely to happen in groups 2, 3 and 4; speed-ups gained in solving irregular memory access can overcome their lack of FPU units. For the same reason, along with the need for higher floating point precision (double, quad), it is very unlikely that dwarves from group 1 obtain performance gains with FPGA-based operators. A different scenario could be possible with the (aforementioned) upcoming Altera's SoC though.

A different situation can occur if FPGAs occupy a more central role in the platform. Allowing them main memory and network access has been proven beneficial in many dwarves. With proper work distribution among CPUs and FPGAs, they could eventually compete with GPU systems even in Group's 1 dwarves.

4 Conclusion

A deep study about patterns, design approaches and current status of HPC algorithm families was performed. We listed core algorithms, computations, optimizations and bottlenecks of each Berkeley's proposed dwarf list and propose a general classification of them to study FPGAs suitability for HPC.

We provided an overview of the challenge for future heterogeneous designs and how FPGA logic can be exploited to solve them. General guidelines to integrate reconfigurable resources in the field of HPC were provided and included special attention to software practices adoption, physical hardware awareness, and algorithmic optimized systems. We also emphasize the importance of more abstraction and less customization.

Dwarves 2, and 6-11 have good potential for speed-ups with the use FPGAs and the latest Systems On Chip as they integrate high-end CPUs mixed with FPGA logic.

Acknowledgments

Work granted by the Wallonia Region FEDER project. The authors would also like to acknowledge the contribution of the COST Action IC1303 AAPELE. Fernando A. Escobar is the corresponding author.

References

[1] L. Carloni, F. D. Bernardinis, C. Pinello, A. Sangiovanni-Vincentelli, and M. Sgroi, "Platform-based design for embedded systems," in *Embedded Systems Handbook*, 1st ed. San Francisco, CA, USA: CRC Press, 2005.

[2] S. Craven and P. Athanas, "Examining the viability of FPGA supercomputing," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 13–13, Jan. 2007.

[3] M. C. Herbord, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello, "Achieving high performance with FPGA-based computing," in *Proc. NIH Public Access*, 2007, pp. 50–57.

[4] R. Inta, D. J. Bowman, and S. M. Scott, "The chimera: An off-the-shelf CPU/GPGPU/FPGA hybrid computing platform," *Int. J. Reconfigurable Comput.*, vol. 2012, no. 1, pp. 10–10, 2012.

[5] A. Krste, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[6] I. Lebedev, C. Fletcher, S. Cheng, J. Martin, A. Doupnik, D. Burke, M. Lin, and J. Wawrzynek, "Exploring many-core design templates for fpgas and asics," *Int. J. Reconfigurable Comput.*, vol. 2012, no. 1, pp. 15–15, 2012.

[7] B. Navas, I. Sander, and J. Oberg, "The recoblock SOC platform: A flexible array of reusable run-time-reconfigurable ip-blocks," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, 2013, pp. 833–838.

[8] R. Bittner, E. Ruf, and A. Forin, "Direct GPU/FPGA communication via PCI express," *Cluster Comput.*, vol. 17, pp. 339–348, 2013.

[9] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires: A query compiler for FPGAS," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 229–240, Aug. 2009.

[10] R. Mueller, J. Teubner, and G. Alonso, "Data processing on FPGAS," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 910–921, Aug. 2009.

[11] D. Perera and K. F. Li, "FPGA-based reconfigurable hardware for compute intensive data mining applications," in *Proc. Int. Conf. P2P, Parallel, Grid, Cloud Internet Comput.*, 2011, pp. 100–108.

[12] I. Sourdis and G. N. Gaydadjiev, "HiPEAC: Upcoming challenges in reconfigurable computing," in *Reconfigurable Computing*, 1st ed. New York, NY, USA: Springer, 2011.

[13] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P. Yves Droz, "Ramp blue: A message-passing manycore system in fpgas," in *Proc. Int. Conf. Field Programm. Logic Appl.*, 2007, pp. 27–29.

[14] W. V. Kritikos, A. G. Schmidt, R. Sass, and E. K. Anderson, "Redsharc: A programming model and on-chip network for multi-core systems on a programmable chip," *Int. J. Reconfigurable Comput.*, vol. 2012, no. 1, pp. 11–11, 2012.

[15] P.-A. Mudry, F. Vannel, G. Tempesti, and D. Mange, "Confetti : A reconfigurable hardware platform for prototyping cellular architectures," in *Proc. IEEE Int. Int. Symp. Parallel Distrib. Process.*, 2007, pp. 1–8.

[16] R. Sass, W. Kritikos, A. Schmidt, S. Beeravolu, and P. Beeraka, "Reconfigurable computing cluster (RCC) project: Investigating the feasibility of FPGA-based petascale computing," in *Proc. 15th Annu. IEEE Symp. Field-Programm. Custom Comput. Mach.*, 2007, pp. 127–140.

[17] Y. Dorz and E. Sava, "Constructing cluster of simple FPGA boards for cryptologic computations," in *Proc. Reconfigurable Comput.: Archit., Tools Appl.*, 2012, vol. 7199, pp. 320–328.

[18] T. Guneysu, T. Kasper, M. Novotny, C. Paar, and A. Rupp, "Cryptanalysis with copacobana," *IEEE Trans. Comput.*, vol. 57, no. 11, pp. 1498–1513, Nov. 2008.

[19] O. Mencer, K. H. Tsoi, S. Craimer, T. Todman, W. Luk, M. Y. Wong, and P. Leong, "Cube: A 512-FPGA cluster," in *Proc. 5th Southern Conf. Programm. Logic*, 2009, pp. 51–57.

[20] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "Fcuda: Enabling efficient compilation of cuda kernels onto FPGAS," in *Proc. IEEE 7th Symp. Appl. Specific Process.*, 2009, pp. 35–42.

[21] A. Grudnitsky, L. Bauer, and J. Henkel, "Partial online-synthesis for mixed-grained reconfigurable architectures," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, 2012, pp. 1555–1560.

[22] S. Kestur, J. D. Davis, and O. Williams, "BLAS comparison on FPGA, CPU and GPU," in *Proc. 2010 IEEE Annual Symp. VLSI (ISVLSI '10)*, pp. 288–293, 2010.

[23] H. Ltaief, P. Luszczek, and J. Dongarra, "Profiling high performance dense linear algebra algorithms on multicore architectures for power and energy efficiency," *Comput. Sci.-Res. Develop.*, vol. 27, no. 4, pp. 277–287, 2012.

[24] K. Underwood and K. Hemmert, "Closing the gap: Cpu and FPGA trends in sustainable floating-point BLAS performance," in *Proc. 12th Annu. IEEE Symp. Field-Programmable Custom Comput. Mach.*, 2004, pp. 219–228.

[25] P. D'Alberto and A. Nicolau, "Adaptive Strassen and Atlas's dgemm: A fast square-matrix multiply for modern high-performance systems," in *Proc. 8th Int. Conf. High-Perform. Comput. Asia-Pacific Region*, 2005, pp. 8–52.

[26] R. Iakymchuk and P. Bientinesi, "Modeling performance through memory-stalls," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 2, pp. 86–91, Oct. 2012.

[27] G. Ballard, "Avoiding communication in dense linear algebra," Ph.D. dissertation, EECS Dept., Univ. of California, Berkeley, Berkeley, CA, USA, Aug. 2013.

[28] G. Wu, Y. Dou, J. Sun, and G. Peterson, "A high performance and memory efficient lu decomposer on FPGAS," *IEEE Trans. Comput.*, vol. 61, no. 3, pp. 366–378, Mar. 2012.

[29] Y.-G. Tai, C.-T. D. Lo, and K. Psarris, "Scalable matrix decompositions with multiple cores on {FPGAs}," *Microprocessors Microsyst.*, vol. 37, no. 8, Part B, pp. 887–898, 2013.

[30] D. Yang, G. D. Peterson, and H. Li, "Compressed sensing and cholesky decomposition on {FPGAs} and {GPUs}," *Parallel Comput.*, vol. 38, no. 8, pp. 421–437, 2012.

[31] S. Skalicky, S. Lpez, M. Lukowiak, J. Letendre, and M. Ryan, "Performance modeling of pipelined linear algebra architectures on fpgas," in *Proc. Reconfigurable Comput.: Archit., Tools Appl.*, 2013, vol. 7806, pp. 146–153.

[32] M. H. Gutknecht, "Block krylov space methods for linear systems with multiple right-hand sides: An introduction," Seminar for Applied Mathematics, pp. 1–22, 2006, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.189.5036&rep=rep1&type=pdf

[33] L. Oliker, X. Li, P. Husbands, and R. Biswas, "Effects of ordering strategies and programming paradigms on sparse matrix computations," *SIAM Rev.*, vol. 44, no. 3, pp. 373–393, Mar. 2002.

[34] M. Ferronato, "Preconditioning for sparse linear systems at the dawn of the 21st century: History, current development and future perspectives," *ISRN Appl. Math.*, vol. 2012, no. 1, pp. 49–49, 2012.

[35] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *J. Supercomput.*, vol. 63, no. 2, pp. 443–466, 2013.

[36] M. Hoemmen, "Communication-avoiding Krylov subspace methods," Ph.D. dissertation, EECS Dept., Univ. of California, Berkeley, Berkeley, CA, USA, Aug. 2010.

[37] X. Li, J. Demmel, J. Gilbert, iL. Grigori, M. Shao, and I. Yamazaki, "SuperLU users' guide," Lawrence Berkeley Nat. Lab., Berkeley, CA, USA, Tech. Rep. LBNL-44289, Sep. 1999.

[38] F. Vazquez, J. J. Fernandez, and E. M. Garzon, "Automatic tuning of the sparse matrix vector product on GPUs based on the ellr-t approach," *Parallel Comput.*, vol. 38, no. 8, pp. 408–420, 2012.

[39] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAS," in *Proc. ACM/SIGDA Int. Symp. Field-Programm. Gate Arrays*, 2014, pp. 161–170.

[40] A. Rafique, G. Constantinides, and N. Kapre, "Communication optimization of iterative sparse matrix-vector multiply on gpus and fpgas," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 24–34, Jan. 2015.

[41] A. Roldao and G. A. Constantinides, "A high throughput FPGA-based floating point conjugate gradient implementation for dense matrices," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 1, pp. 1:1–1:19, Jan. 2010.

[42] G. Morris and K. Abed, "Mapping a Jacobi iterative solver onto a high-performance heterogeneous computer," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 1, pp. 85–91, Jan. 2013.

[43] J. Jerez, G. Constantinides, and E. Kerrigan, "A low complexity scaling method for the lanczos kernel in fixed-point arithmetic," *IEEE Trans. Comput.*, vol. 64, no. 2, pp. 303–315, Feb. 2015.

[44] B. Costa, "Spectral methods for partial differential equations," *CUBO Math. J.*, vol. 6, no. 4, pp. 1–32, 2004.

[45] D. Kopriva, "Survey of spectral approximations," in *Implementing Spectral Methods for Partial Differential Equations*, ser. Scientific Computation. New York, NY, USA: Springer, 2009, pp. 91–147.

[46] W. Wang, B. Duan, C. Zhang, P. Zhang, and N. Sun, "Accelerating 2d fft with non-power-of-two problem size on FPGA," in *Proc. 2010 Int. Conf. Reconfigurable Comput.*, 2010, pp. 208–213.

[47] C.-L. Yu, J.-S. Kim, L. Deng, S. Kestur, V. Narayanan, and C. Chakrabarti, "Fpga architecture for 2d discrete fourier transform based on 2d decomposition for large-sized data," *J. Signal Process. Syst.*, vol. 64, no. 1, pp. 109–122, 2011.

[48] B. S. C. Varma, K. Paul, and M. Balakrishnan, "Accelerating 3d-fft using hard embedded blocks in FPGAS," in *Proc. 26th Int. Conf. VLSI Design 12th Int. Conf. Embedded Syst.*, 2013, pp. 92–97.

[49] B. Humphries, H. Zhang, J. Sheng, R. Landaverde, and M. C. Herbordt, "3d FFTs on a single FPGA," in *Proc. IEEE 22nd Int. Symp. Field-Programm. Custom Comput. Mach.*, 2014, pp. 68–71.

[50] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *Proc. Conf. High Perform. Comput. Netw., Storage Anal.*, 2009, pp. 58:1–58:12.

[51] H. Hannak, H. Hochstetter, and W. Blochinger, "A hybrid parallel Barnes-hut algorithm for GPU and multicore architectures," in *Proc. Euro-Par Parallel Process.*, 2013, vol. 8097, pp. 559–570.

[52] D. Blackston and T. Suel, "Highly portable and efficient implementations of parallel adaptive n-body methods," in *Proc. Conf. Supercomput.*, 1997, p. 4.

[53] Q. Hu, N. A. Gumerov, and R. Duraiswami, "Scalable fast multipole methods on distributed heterogeneous architectures," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, p. 36.

[54] R. Yokota and L. A. Barba, "A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems," *CoRR*, abs/1106.2176, 2011, http://arxiv.org/abs/1106.2176

[55] T. Ishiyama, K. Nitadori, and J. Makino, "4.45 pflops astrophysical n-body simulation on k computer: The gravitational trillion-body problem," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 5:1–5:10.

[56] Z. Zheng, Y. Zhu, X. Wang, Z. Que, T. Huang, X. Yin, H. Wang, G. Rong, and M. Qiu, "Revealing feasibility of fmm on asic: Efficient implementation of n-body problem on FPGA," in *Proc. IEEE 13th Int. Conf. Comput. Sci. Eng.*, Dec. 2010, pp. 132–139.

[57] T. Hamada, K. Benkrid, K. Nitadori, and M. Taiji, "A comparative study on asic, FPGAS, GPUs and general purpose processors in the o(n2) gravitational n-body simulation," in *Proc. NASA/ESA Conf. Adaptive Hardware Syst.*, Jul. 2009, pp. 447–452.

[58] M. Khan, M. Chiu, and M. Herbordt, "FPGA-accelerated molecular dynamics," in *High-Performance Computing Using FPGAs*, W. Vanderbauwhede and K. Benkrid, Eds. New York, NY, USA: Springer, 2013, pp. 105–135.

[59] A. Severance, J. Edwards, H. Omidian, and G. Lemieux, "Soft vector processors with streaming pipelines," in *Proc. ACM/SIGDA Int. Symp. Field-Programm. Gate Arrays*, 2014, pp. 117–126.

[60] K. H. Tsoi and W. Luk, "Axel: A heterogeneous cluster with FPGAS and GPUs," in *Proc. 18th Annu. ACM/SIGDA Int. Symp. Field Programm. Gate Arrays*, 2010, pp. 115–124.

[61] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *Proc. 33rd Annu. IEEE Int. Comput. Softw. Appl. Conf.*, 2009, vol. 1, pp. 579–586.

[62] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proc. 26th ACM Int. Conf. Supercomput.*, 2012, pp. 311–320.

[63] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 11:1–11:12.

[64] J. Jaeger and D. Barthou, "Automatic efficient data layout for multithreaded stencil codes on CPU sand GPUs," in *Proc. 19th Int. Conf. High Perform. Comput.*, 2012, pp. 1–10.

[65] S. M. F. Rahman, Q. Yi, and A. Qasem, "Understanding stencil code performance on multicore architectures," in *Proc. 8th ACM Int. Conf. Comput. Frontiers*, 2011, pp. 30:1–30:10.

[66] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for GPUs: Automatic parallelization using trapezoidal tiles," in *Proc. 6th Workshop General Purpose Processor Using Graph. Process. Units*, 2013, pp. 24–31.

[67] K. Sano, Y. Kono, H. Suzuki, R. Chiba, R. Ito, T. Ueno, K. Koizumi, and S. Yamamoto, "Efficient custom computing of fully-streamed lattice boltzmann method on tightly-coupled FPGA cluster," *SIGARCH Comput. Archit. News*, vol. 41, no. 5, pp. 47–52, Jun. 2014.

[68] R. Kobayashi, S. Takamaeda-Yamazaki, and K. Kise, "Towards a low-power accelerator of many FPGAS for stencil computations," in *Proc. 3rd Int. Conf. Netw. Comput.*, Dec. 2012, pp. 343–349.

[69] K. Sano, Y. Hatsuda, and S. Yamamoto, "Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 695–705, Mar. 2014.

[70] T. Mitra, "An FPGA implementation of triangle mesh decompression," in *Proc. IEEE Int. Symp. Field Programm. Custom Comput. Mach.*, 2002, pp. 22–31.

[71] P. Barrio, C. Carreras, J. A. Lpez, Ô. Robles, R. Jevtic, R. Sierra, P. Barrio, C. Carreras, J. A. Lopez, O. Robles, R. Jevtic, and R. Sierra, "Memory optimization in FPGA-accelerated scientific codes based on unstructured meshes," *J. Syst. Archit.*, vol. 60, no. 7, pp. 579–591, 2014.

[72] T. Akamine, K. Inakagata, Y. Osana, N. Fujita, and H. Amano, "Reconfigurable out-of-order mechanism generator for unstructured grid computation in computational fluid dynamics," in *Proc. 22nd Int. Conf. Field Programm. Logic Appl.*, Aug. 2012, pp. 136–142.

[73] M. Neves, T. Ferreto, and C. Rose, "Scheduling mapreduce jobs in HPC clusters," in *Proc. Parallel Process.*, 2012, vol. 7484, pp. 179–190.

[74] D. Yin, G. Li, and K.-D. Huang, "Scalable mapreduce framework on FPGA accelerated commodity hardware," in *Proc. Internet Things, Smart Spaces, Next Generation Netw.*, 2012, vol. 7469, pp. 280–294.

[75] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating mapreduce with graphics processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 4, pp. 608–620, Apr. 2011.

[76] Y. Lin, S. Okur, and C. Radoi, "Hadoop+aparapi: Making heterogenous mapreduce programming easier," Dept. Comput. Sci., Univ. Illinois Urbana Champaign, Urbana Champaign, IL, USA, 2012.

[77] M. Xin and H. Li, "An implementation of GPU accelerated mapreduce: Using hadoop with opencl for data- and compute-intensive jobs," in *Proc. Int. Joint Conf. Service Sci.*, 2012, pp. 6–11.

[78] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "Fpmr: Mapreduce framework on FPGA," in *Proc. 18th Annu. ACM/SIGDA Int. Symp. Field Programm. Gate Arrays*, 2010, pp. 93–102.

[79] D. Yin, G. Li, and K.-D. Huang, "Scalable mapreduce framework on FPGA accelerated commodity hardware," in *Proc. Internet Things, Smart Spaces, Next Generation Netw.*, 2012, vol. 7469, pp. 280–294.

[80] D. Unnikrishnan, S. Virupaksha, L. Krishnan, L. Gao, and R. Tessier, "Accelerating iterative algorithms with asynchronous accumulative updates on FPGAS," in *Proc. Int. Conf. Field-Programm. Technol.*, Dec. 2013, pp. 66–73.

[81] C. E. Shannon, "Communication theory of secrecy systems," *Bell Syst. Tech. J.*, vol. 28, no. 4, pp. 656–715, 1949.

[82] S. of Commerce, *Advanced encryption standard (AES)*, Federal Information Processing Standards, vol. 197, 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[83] H. Jo, S.-T. Hong, J.-W. Chang, and D. H. Choi, "Data encryption on GPU for high-performance database systems," *Proc. Comput. Sci.*, vol. 19, pp. 147–154, 2013.

[84] R. R. Farashahi, B. Rashidi, and S. M. Sayedi, "FPGA based fast and high-throughput 2-slow retiming 128-bit AES encryption algorithm," *Microelectron. J.*, vol. 45, no. 8, pp. 1014–1025, 2014.

[85] H. Michail, G. Athanasiou, G. Theodoridis, and C. Goutis, "On the development of high-throughput and area-efficient multimode cryptographic hash designs in FPGAs," *Integr. VLSI J.*, vol. 47, no. 4, pp. 387–407, 2014.

[86] M. I. Soliman and G. Y. Abozaid, "FPGA implementation and performance evaluation of a high throughput crypto coprocessor," *J. Parallel Distrib. Comput.*, vol. 71, no. 8, pp. 1075–1084, 2011.

[87] H. Lv, G. Tan, M. Chen, and N. Sun, "Understanding parallelism in graph traversal on multi-core clusters," *Comput. Sci.-Res. Develop.*, vol. 28, nos. 2/3, pp. 193–201, 2013.

[88] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, 2012, pp. 378–389.

[89] S. Ni, Y. Dou, D. Zou, R. Li, and Q. Wang, "Parallel graph traversal for FPGA," *IEICE Electron. Express*, vol. 11, no. 7, p. 20130987, 2014.

[90] B. Betkaoui, D. Thomas, W. Luk, and N. Przulj, "A framework for FPGA acceleration of large graph problems: Graphlet counting case study," in *Proc. Int. Conf. Field-Programm. Technol.*, Dec. 2011, pp. 1–8.

[91] O. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno, "Cygraph: A reconfigurable architecture for parallel breadth-first search," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp. Workshops*, May 2014, pp. 228–235.

[92] A. Grama, A. Gupta, G. Karypis, and V. Kumar, "Chapter 12: Dynamic programming," in *Introduction to Parallel Computing*, Pearson, 2nd ed. Reading, MA, USA: Addison-Wesley, 2003, pp. 515–532.

[93] J. Li, G. Tan, and M. Chen, "Automatically tuned dynamic programming with an algorithm-by-blocks," in *Proc. IEEE 16th Int. Parallel Distrib. Syst. Conf.*, Dec. 2010, pp. 452–459.

[94] C.-C. Wu, J.-Y. Ke, H. Lin, and W. chun Feng, "Optimizing dynamic programming on graphics processing units via adaptive thread-level parallelism," in *Proc. IEEE 17th Int. Parallel Distrib. Syst. Conf.*, 2011, pp. 96–103.

[95] L. Liu, M. Wang, J. Jiang, R. Li, and G. Yang, "Efficient nonserial polyadic dynamic programming on the cell processor," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Phd Forum*, May 2011, pp. 460–471.

[96] S. Xiao, A. M. Aji, and W.-C. Feng, "On the robust mapping of dynamic programming onto a graphics processing unit," in *Proc. 15th Int. Conf. Parallel Distrib. Syst.*, 2009, pp. 26–33.

[97] K.-E. Berger and F. Galea, "An efficient parallelization strategy for dynamic programming on gpu," in *Proc. IEEE 27th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, May 2013, pp. 1797–1806.

[98] H. Shah, L. Hasan, and N. Ahmad, "An optimized and low-cost FPGA-based dna sequence alignment. a step towards personal genomics," in *Proc. 35th Annu. Int. Conf. IEEE Eng. Med. Biol. Soc.*, Jul. 2013, pp. 2696–2699.

[99] S. O. Settle, "High performance dynamic programming on FPGAS with opencl," in *Proc. 17th IEEE High Perform. Extreme Comput. Conf.*, 2013.

[100] K. Nibbelink, S. Rajopadhye, and R. McConnell, "0/1 knapsack on hardware: A complete solution," in *Proc. IEEE Int. Conf. Appl.-Specific Syst., Archit. Process.*, Jul. 2007, pp. 160–167.

[101] I. Chakroun, "Parallel heterogeneous branch and bound algorithms for multi-core and multi-gpu environments," Ph.D. dissertation, Ecole Doctorale Sciences Pour l'Ingenieur, Universit de Lille, France, Jun. 2013.
[102] B. Gendron and T. G. Crainic, "Parallel branch-and-bound algorithms: Survey and synthesis," *Oper. Res.*, vol. 42, no. 6, pp. 1042–1066, 1994.
[103] J. Herrera, L. Casado, R. Paulavicius, J. Zilinskas, and E. Hendrix, "On a hybrid mpi-pthread approach for simplicial branch-and-bound," in *Proc. IEEE 27th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, May 2013, pp. 1764–1770.
[104] S. Wakabayashi and K. Kikuchi, "An instance-specific hardware algorithm for finding a maximum clique," in *Proc. 14th Int. Conf. Field Programm. Logic Appl.*, 2004, vol. 3203, pp. 516–525.
[105] S. Wakabayashi, Y. Kimura, and S. Nagayama, "FPGA implementation of tabu search for the quadratic assignment problem," in *Proc. IEEE Int. Conf. Field Programm. Technol.*, Dec. 2006, pp. 269–272.
[106] A. Dandalis, V. Prasanna, and B. Thiruvengadam, "Run-time performance optimization of an FPGA-based deduction engine for sat solvers," in *Proc. 11th Int. Conf. Field-Programm. Logic Appl.*, 2001, vol. 2147, pp. 315–325.
[107] J. Kwisthout, "Most probable explanations in Bayesian networks: Complexity and tractability," *Int. J. Approximate Reasoning*, vol. 52, no. 9, pp. 1452–1469, 2011.
[108] I. Pournara, C. Bouganis, and G. Constantinides, "Fpga-accelerated Bayesian learning for reconstruction of gene regulatory networks," in *Proc. Int. Conf. Field Programm. Logic Appl.*, Aug. 2005, pp. 323–328.
[109] O. Nikolova, J. Zola, and S. Aluru, "A parallel algorithm for exact Bayesian network inference," in *Proc. Int. Conf. High Perform. Comput.*, Dec. 2009, pp. 342–349.
[110] Y. Wang, W. Qian, S. Zhang, and B. Yuan, "A novel learning algorithm for Bayesian network and its efficient implementation on gpu," *CoRR*, abs/1210.5128, 2012, http://arxiv.org/abs/1210.5128
[111] N. Ma, Y. Xia, and V. Prasanna, "Data parallel implementation of belief propagation in factor graphs on multi-core platforms," *Int. J. Parallel Programm.*, vol. 42, no. 1, pp. 219–237, 2014.
[112] M. Lin, I. Lebedev, and J. Wawrzynek, "High-throughput bayesian computing machine with reconfigurable hardware," in *Proc. 18th Annu. ACM/SIGDA Int. Symp. Field Programm. Gate Arrays*, 2010, pp. 73–82.
[113] F. Ortega-Zamorano, J. Jerez, and L. Franco, "FPGA implementation of the c-mantec neural network constructive algorithm," *IEEE Trans. Ind. Inform.*, vol. 10, no. 2, pp. 1154–1161, May 2014.
[114] L.-W. Kim, S. Asaad, and R. Linsker, "A fully pipelined FPGA architecture of a factored restricted boltzmann machine artificial neural network," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 1, pp. 5:1–5:23, Feb. 2014.
[115] M. Mohri, F. Pereira, and M. Riley, "Weighted finite-state transducers in speech recognition," *Comput. Speech Language*, vol. 16, no. 1, pp. 69–88, 2002.
[116] A. Tewari, U. Srivastava, and P. Gupta, "A parallel DFA minimization algorithm," in *Proc. High Perform. Comput.*, 2002, vol. 2552, pp. 34–40.
[117] V. Slavici, D. Kunkle, G. Cooperman, and S. Linton, "Finding the minimal DFA of very large finite state automata with an application to token passing networks," *CoRR*, abs/1103.5736, 2011, http://arxiv.org/abs/1103.5736
[118] R. Sinya, K. Matsuzaki, and M. Sassa, "Simultaneous finite automata: An efficient data-parallel model for regular expression matching," in *Proc. 42nd Int. Conf. Parallel Process.*, Oct. 2013, pp. 220–229.
[119] J. Holub and S. Stekr,"On parallel implementations of deterministic finite automata," in *Proc. Implementation Appl. Autom.*, 2009, vol. 5642, pp. 54–64.
[120] V. Sklyarov and I. Skliarova, "Design and implementation of parallel hierarchical finite state machines," in *Proc. 2nd Int. Conf. Commun. Electron.*, Jun. 2008, pp. 33–38.
[121] L. Aubert, R. Woods, S. Fischaber, and R. Veitch, "Optimization of weighted finite state transducer for speech recognition," *IEEE Trans. Comput.*, vol. 62, no. 8, pp. 1607–1615, Aug. 2013.
[122] J. Choi, K. You, and W. Sung, "An FPGA implementation of speech recognition with weighted finite state transducers," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, Mar. 2010, pp. 1602–1605.
[123] D., and Reuse, "Xilinx delivers the industry's first 4m logic cell device," 2015, http://www.design-reuse.com/news/36309/xilinx-virtex-ultrascale-vu440-fpga.html
[124] K. Morris, "The FPGA is half full," 2013, http://www.eejournal.com/archives/articles/20130924-halffull/

Fernando A. Escobar received the MSc degree in electronic engineering from Universidad de Los Andes, Colombia, 2011. He is currently working toward the PhD degree in the SEMI Department, University of Mons, Belgium. His research interest includes digital design for HPC systems, computer architecture, and embedded systems. He is a member of the IEEE.

Xin Chang received the MSc degree in embedded computing from the University of Turku, Finland, 2012, and is currently working as a research assistant in the SEMI Department, University of Mons, Belgium. His research interest includes on-chip interconnection, high-level synthesis, multiprocessor system-on-chip, and signal processing.

Carlos Valderrama received the PhD degree from the Institut National Polytechnique de Grenoble (INPG, France) in 1998, and the MSc degree from the Federal University of Rio de Janeiro (UFRJ), Brazil, in 1993. He is the director of the Electronic and Microelectronics Department, University of Mons (UMons/SEMI, Belgium). His research interests includes architectures, methodologies and tools for reconfigurable, embedded and multi-core devices. He is a member of the IEEE.