The Art of FPGA Algorithm Design – The Case for the Extreme Acceleration of Linear-Algebra-Intensive Software

Juan Gonzalez and Rafael Nunez
Research Division, Accelogic, LLC
[juan.gonzalez, rafael.nunez]@accelogic.com

In memory of Gene H. Golub, co-founder of Accelogic's Research Program on Acceleration of Numerical Software

Abstract— **The developments seen in the field of reconfigurable computing during the last ten years bring an unprecedented opportunity for the acceleration of supercomputing applications in computational fluid dynamics (CFD). Reconfigurable computing algorithms implemented in Field-Programmable Gate Arrays (FPGAs) have proven to be orders of magnitude faster than traditional CPU-based solutions.**

It is estimated that over seventy percent of supercomputer CPU cycles worldwide are spent solving large-scale linear algebra problems. Accelogic is developing unique algorithmic innovations that will enable a single FPGA chip to surpass the performance of a CPU cluster for solving large-scale linear systems. This technology has the potential to reduce both cost and power consumption by one to two orders of magnitude, while maintaining code portability and ease of use for FORTRAN and C environments.

We show our recent results in this direction, including insights on why and how things can go wrong when designing FPGA supercomputing kernels, and why the common-wisdom approach of "porting" or "translating" the algorithms into the FPGA has not delivered the promised levels of performance for CFD. We discuss the critical factors of success behind Accelogic's latest 60x-speedup narrowband linear solver – the fastest FPGA linear solver at the time of writing.

Index Terms— **Linear algebra, Algorithms, Field programmable gate array, FPGA, Matrix decomposition, Digital arithmetic, LU, Band solver.**

Manuscript written July, 2007. Revised November, 2010, and September, 2014. Released for publication September, 2014 by Accelogic, LLC.

This work is subject matter of U.S. Patent #7,849,126, and was supported in part by NASA under contracts NNX07CA19P and NNX10RA04C.

J. Gonzalez is with Accelogic, LLC, Weston, FL 33326 USA (e-mail: juan.gonzalez@accelogic.com).

R. Nunez is with Accelogic, LLC, Weston, FL 33326 USA (e-mail: rafael.nunez@accelogic.com).

The authors wish to thank Gene H. Golub (R.I.P) from Stanford University, and Jack Dongarra from the University of Tennessee, for their consultation on this project.

I. INTRODUCTION

RECONFIGURABLE computing with FPGAs has emerged within the last years as a viable alternative for very low-cost high-performance supercomputing. Speedups over traditional single-CPU systems on the orders of hundreds, and even thousands, have been demonstrated for specific domain applications during the last years.

The fields of partial differential equations and numerical linear algebra, perhaps the most important ones in high-performance computing, have not escaped to acceleration attempts made by FPGA programmers and developers. However, results in this direction have not been very successful so far. Researchers have been able to demonstrate single-digit speedups at most, and many of the proposed solutions lack both portability and ease of use.

This paper reports the results from Accelogic's reconfigurable computing research program on numerical linear algebra, which aims at conceptualizing and developing the world's fastest FPGA-based solver for large-scale linear equations, a single-FPGA system with performance comparable to that of ScaLAPACK running on a 4,096-CPU supercomputer. Besides the direct breakthroughs in terms of speed performance enabled when using and scaling this technology, this type of solver can also bring a one to two order-of-magnitude reduction in both cost and power consumption, while maintaining code portability and ease of use for Fortran and C environments via simple and easy-to-use APIs.

A working demonstration of the reach and impact of Accelogic's approach is presented through a proof-of-concept prototype that exhibits a 60x performance speedup when compared to LAPACK running on the fastest CPU available at the time of writing. The high performance of this prototype successfully shows the feasibility of the architectural design and the algorithmic innovations proposed by Accelogic to reach a 4,096-node performance target.

The key to the technical and commercial success of this work lies on four mission-critical areas, two of them algorithmic (algorithmic speed balance, and communication speed) and two of them related to the industrial quality of the product (portability, and ease of use). The entirety of the work gravitates around these four mission-critical areas.

We show that the challenge to attaining groundbreaking speedups with FPGAs is algorithmic, and speedup does not come as a direct consequence of merely using the new hardware. The "common wisdom" approach of using traditional von Neumann algorithms and "porting" them or "translating" them into an FPGA code is doomed to fail, as it will likely waste precious opportunities to innovate and take advantage of the non-von-Neumann approach offered by the FPGA. It is thus, in the opinion of the authors, only through strong algorithmic innovations and the invention of new methods enabled only by the reconfigurable computing paradigm, that truly revolutionary speedups will be obtained. The techniques developed and implemented in this work, have rendered, to the authors' knowledge, the highest speedup in the world (at the time of writing) for a single-FPGA linear equation solver. This solver is currently functional and operational in prototype form, and exhibits a 60x speedup over high-end single-CPU systems when compared to LAPACK.

This paper is organized as follows: Section II explains the disadvantages of the von Neumann architecture, Section III explains the atomic LU factorization algorithm, Section IV presents the idea of algorithm-aware adaptive precision (AAAP), critical for attaining the levels of performance of the algorithm, Section V shows the advantages of using arithmetic acceleration at the bit-level (Turbo-LU), Section VI addresses the issues of portability, ease of use, and algorithmic speed balance, Section VII presents a description of the implemented prototype, Section VIII shows several benchmarks and performance results, and finally, Section IX presents the conclusions of the work.

II. THE VON NEUMANN ARCHITECTURE BOTTLENECK

The use of von Neumann architecture-based CPUs for the implementation of numerical linear algebra implies than even the algorithms most suitable for parallelization are executed sequentially. For illustration purposes, let us examine the case of an inner product calculation. In the von Neumann architecture a single hardware multiplier is used to perform the required products, one element at a time. The introduction of the Harvard architecture relieves the problem by allowing the two vectors to reside in different memory spaces, but the multiplier is still a single one. Furthermore, memory caching allows the CPU a faster access to a local memory bank, but no enhancements are introduced as of the number of multipliers in the CPU Core or the sequential read/write operations from memory. Considering that matrix computations reach an optimum processing throughput by the use of as many multipliers as elements in the vectors, researchers have investigated multiprocessor networks as well as multicore/manycore architectures, only to find the von Neumann bottleneck replicated at every machine or core, plus a new breed of problems related to the distribution of the workload among multiple processing units.

Another approach in increasing the processing throughput of von Neumann-based processors is to increase the clock speed. This approach is limited, however, by increased power dissipation.

The solution to this bottleneck is facilitated by reconfigurable hardware architectures suited for each particular problem. FPGAs offer a departing point for the implementation of cost and power effective parallel computing.

III. THE ATOMIC LU FACTORIZATION ALGORITHM

The first and most essential innovation enabling the speedups obtained with this work is the "atomic LU" factorization algorithm. This algorithm can be seen as a sequence of N mega-instructions, each mega-instruction compounded of many (hundreds or even thousands of) atomic arithmetic operations working all at once in a massively parallel network. The approach departs significantly from what is typically done in von-Neumann algorithm design, and is extremely powerful.

Let's begin the discussion with a brief introduction to LU factorization as used in this project:

A. Background

LU factorization is a procedure for decomposing an *NxN* matrix *A* into a product of a lower triangular matrix L and an upper triangular matrix *U*, *A = LU*. When solving the matrix equation *Ax = b*, the LU factorization enables the formulation of the equation as:

$$A\mathbf{x} = (LU)\mathbf{x} = L(U\mathbf{x}) = \mathbf{b} \tag{1}$$

The *x* vector solution can be found by first solving the auxiliary system *Lz = b* for *z* using the so-called forward substitution:

$$z_1 = b_1 \tag{2}$$

and

$$z_i = b_i - \sum_{j=1}^{i-1} l_{ij} z_j \tag{3}$$

for *i = 2, …, N*. The next step is to solve *Ux = z* for *x* using the so-called backward substitution:

$$x_N = \frac{z_N}{u_{NN}} \tag{4}$$

and

$$x_i = \frac{1}{u_{ii}}\left(z_i - \sum_{j=i+1}^{N} u_{ij}x_j\right) \qquad (5)$$

for $i = N\text{-}1, \ \ldots, \ 1$. The methodology described above assumes that all the elements in the diagonal of L are 1, allowing a combined representation of L and U in a single matrix.

A procedure to calculate the LU factorization in N stages is described by the iterative application of the equations:

$$u_{ij} \;=\; a_{ij} - \sum_{r=0}^{i-1} l_{ir}u_{rj} \qquad (6)$$

and

$$l_{ij} \;=\; \frac{1}{u_{jj}}\left(a_{ij} - \sum_{r=0}^{j-1} l_{ir}u_{rj}\right) \qquad (7)$$

where a_{ij} are the components of the intermediate matrix A, such that $a_{ij}(k)$ represents the component ij of the matrix A at the step k, $0 \le k \le N$.

B. *The "Atomic LU" algorithm in reconfigurable computing*

The key concept behind the proposed algorithm is the use of a "mega-instruction", specifically built into the system architecture to exploit the super-modular structure of the LU equations expressed in the equations above. This mega-instruction is made of many (hundreds or even thousands) single arithmetic operators appropriately connected to render the exact mathematical expression of the equations above at each iteration step k.

Massive parallelism is enabled by the concurrent operation of all of these arithmetic operators. The operators are conceived as basic arithmetic processing units at the component level, called atoms, hence the name "Atomic LU." The main atom in the architecture is called the "MA atom," where "MA" stands for "Multiply and Add."

A functional block diagram of this atom is depicted in Figure 1, and the complete mega-instruction, consisting of a massive interconnection of atoms is shown in Figure 2. Note that in this architecture, the atoms at the leftmost side are different from the one shown in Figure 1. Those atoms perform the normalization of the main diagonal of the L matrix, and prepare the delivery of the L matrix.

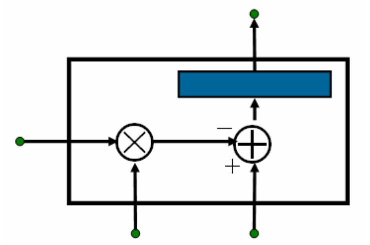

Figure 1. MA Atom

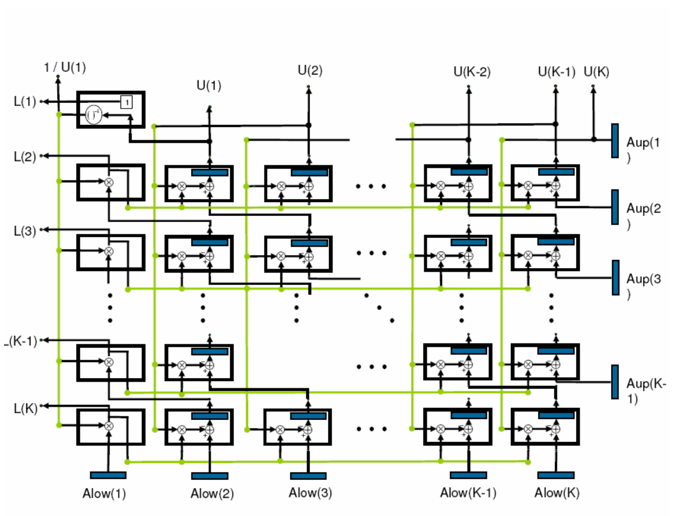

Figure 2. Accelogic's atomic LU architecture

In this architecture, the matrix is fed to the mega-instruction from the bottom (semi-row by semi-row) and the right-hand side (semi-column by semi-column), resulting in a diagonal data flow as shown in Figure 3. After b (half of the matrix bandwidth) clock cycles, the matrix is organized in the array and the calculations can begin. In the next clock cycle, $(b+1)$, the first column of the L and first row of U are calculated at once by the atomic array (or mega-instruction), and can be read as an output from the left and top parts of it. Then, the data is moved upwards and leftwards producing the second column of L and the second row of U. The algorithm continues this way until the last column of L and the last row of U are found. It can be seen that the algorithm requires a total of $(b+N)$ clock cycles to compute the factorization.

The architectures proposed for the forward and backward substitutions are shown in Figure 4 and Figure 5 respectively. The advantage of these architectures is that they can be easily integrated into the core LU core, providing the solution of the linear system only N cycles after the LU factorization is done. The combined LU factorization and solution of the matrix equation $Ax = b$ is shown in Figure 6.

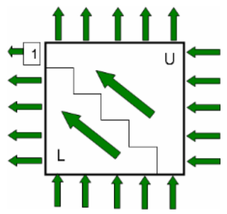

Figure 3. Data flow in the atomic LU architecture

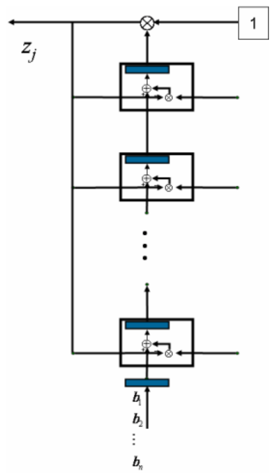

Figure 4. Forward substitution

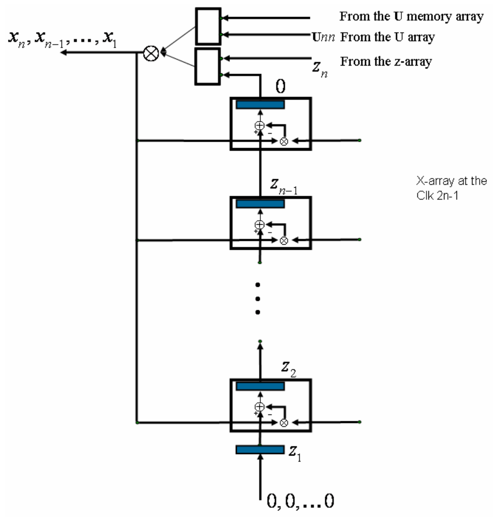

Figure 5. Backward substitution

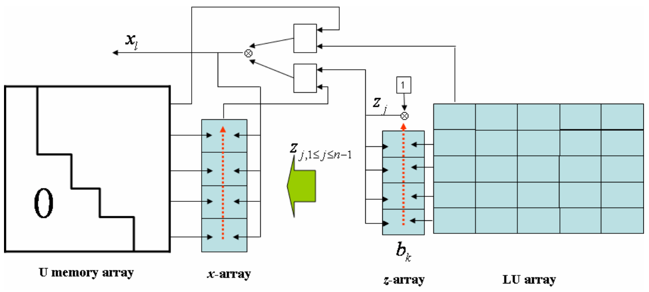

Figure 6. Atomic architecture for the solution of linear equations using LU factorization

IV. ALGORITHM-AWARE ADAPTIVE PRECISION (AAAP)

In the atomic architecture, hundreds, or even thousands of floating-point operations are executed during each clock cycle of the FPGA. At a lower level of the design, those operations are executed as bit operations, defined by the architecture/algorithm design. The bit-level implementation of these floating-point operations plays a key role in the execution speed of the overall system.

In general, from a speed perspective, it is desirable for every floating-point operation to use as few bits as the accuracy requirements of the algorithm allow it. Reducing the number of bits is of primary importance for FPGA algorithms because it has a twofold effect in speed: (1) the reduction simplifies the logic required to complete the arithmetic operations, thus making them faster and able to run at higher clock frequencies (often significantly faster); and (2) by making the operators simpler, each operator will occupy significantly less area in the FPGA, thus freeing FPGA area to allow more parallel processing units – this enables higher parallelism, which results in even higher algorithmic speeds.

Figure 7 and Figure 8 show the effects of arithmetic bit reduction on FPGA area and on FPGA clock speed of the implemented atomic LU algorithm. In this example, reducing the number of bits from 20 to 12, produces a reduction of area better than 3 times, and enables an increase in clock frequency from 41 MHz to 48 MHz. The area reduction enables an additional speedup of 3×, while the clock frequency increase enables an additional speedup of 1.2x. Thus, the overall net effect of reducing the number of bits from 20 to 12 is an additional speedup of 3.6×. Note this speedup comes free and on top of any other architectural or algorithmic speedups in the system.

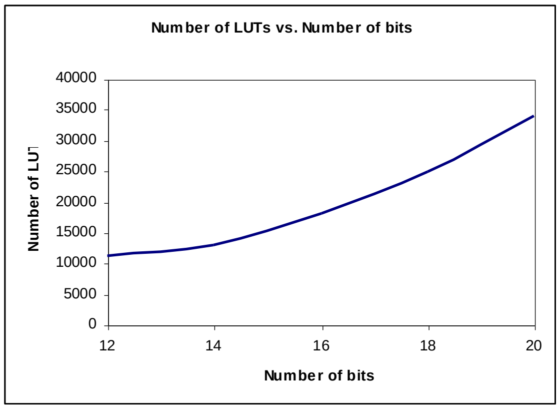

Figure 7. Number of Look up tables (LUT, proportional to the area of the FPGA occupied by the algorithm) as a function of the number of bits used in the floating-point arithmetic.

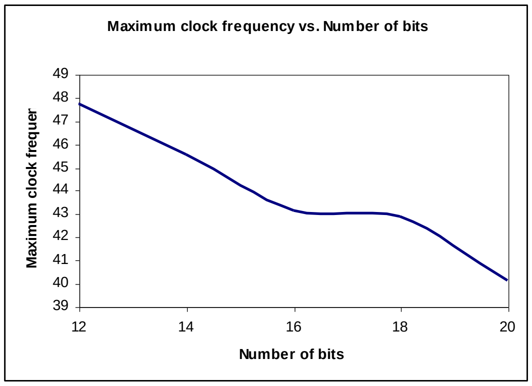

Figure 8. Maximum clock frequency of the atomic architecture as a function of the number of bits in the floating-point arithmetic.

The number of bits required to obtain a given accuracy in linear equation problems, is directly related to the condition number of the matrix *A*. A rule of thumb is $l \approx log_2(\kappa)$, where κ is the condition number of *A*, and *l* is the number of bits of the floating point arithmetic. The use of more than *l* bits does not increase the accuracy of the algorithm.

Given the limitations of von Neumann architectures, only 32- and 64-bit floating-point arithmetic operations are possible in commercial CPU systems.[1] Therefore, the majority of software applications simply use 64 bits for all of their operations, even if such precision is not required (64 bits is usually overkill for most practical problems, and a tremendous waste of computing resources). Reconfigurable computing allows the use of variable precision arithmetic targeted to the application at hand. The atomic LU algorithm implemented in the current prototype is parameterized to allow the easy exploitation of variable precision arithmetic to squeeze speed out of the system.

The key idea of Algorithm-Aware Adaptive Precision (AAAP) is to use different bit-widths for different operations inside the algorithm. There is no reason why the algorithm would need all of its operations to have the same precision. The local precision of arithmetic operations, although correlated with the algorithm's accuracy, is not the same as the global accuracy of the result.

For the atomic LU algorithm, the authors were able to implement AAAP in such a way that most of the floating-point operations (more than 99%) are performed with a precision of only 12 bits, while only a few of them are performed with 64 bit precision (less than 1%). In spite of this precision reduction, the algorithm still presents an end-to-end accuracy of 64-bit arithmetic. This reduction in the number of bits has a tremendous speedup effect on the algorithm, because of the reasons discussed above[2]. This result is achieved by mixing the plain atomic LU factorization done in the FPGA with an iterative "precision enhancement" technique called "iterative refinement" implemented on the CPU side of the system.

Iterative refinement is a technique of the 1960's typically used to correct pivoting errors in sparse systems or to achieve higher than 64-bit accuracy in CPU systems. The algorithm is summarized as [1]:

```
  1.Solve Ax = b in lower precision, save the lower
precision L and U.
  2.Compute in higher precision r = b – A*x .
  3.Stop if residual r meets convergence conditions.
  4.Solve  triangular  systems  (L*U)z  =  r  in  lower
precision.
  5.Update  solution    x+  =  x  +  z    using  higher
precision.
  6.Repeat from 2.
```

It can be demonstrated that, if the number of bits in the low precision arithmetic is appropriate, the number of iterations required to achieve a high accuracy is very small, and the whole iterative process typically accounts for less than 1% of the total time to solution (combined low-precision and iterative refinement algorithm).

The use of iterative refinement for FPGA algorithm speedup is a key enabler of the proposed technology, and an excellent illustration of what AAAP can achieve under the context of this project. The results obtained show that there are several ways of exploiting AAAP in the context of FPGA linear algebra to implement more efficient algorithms.

V. ARITHMETIC ACCELERATION AT THE BIT-LEVEL (TURBO-LU)

Arithmetic acceleration at the bit level is a powerful feature of reconfigurable computing that can be exploited to speedup algorithms significantly. It is neither trivial nor straightforward, and requires significant innovation on the part of the algorithm designer. Moreover, it is a feature not available to the scientist working in the traditional von Neumann paradigm.

In short words, arithmetic acceleration is about designing the algorithm jointly at the macro level (the typical arithmetic operations) and at the bit level (how those arithmetic operations are implemented bit by bit, or logic gate by logic gate). One approach to understand arithmetic acceleration is figuring that, instead of using basic arithmetic operators, the designer implements "mega-operators" that glue the most critical operations of the algorithm, in a way that together they consume less FPGA area and can run at a faster clock speed. The overall effect is to free extra FPGA area for increased and

[1] Single (32 bits) and double (64 bits) floating point precisions are intrinsically defined in commercial CPUs. Arbitrary precisions can be obtained with specialized libraries, at the cost of much higher computation times.

[2] The authors estimate that the core LU algorithm works at 2x speedup, AAAP incorporates an additional 6x, and the arithmetic accelerations explained in the next Subsection account for an additional 5x – the combined effect is multiplicative, hence explaining the resulting 60x.

more massive parallelism, accompanied by a faster clock speed, resulting in an increased speedup similar to the way it is obtained through the use of AAAP.

Bit-level optimizations should always be considered in order to maximize speed performance. Consider for example the addition of two integers. The process is described in Figure 9 and it starts with the binary addition of the Least Significant Bit (LSB) of both numbers. When the first binary addition is completed (after time τ), the LSB bit of the answer is ready and can be used in other arithmetic operations of the parent algorithm if necessary. Also, after time τ, the carry of the first binary addition triggers the addition of the next pair of bits. This process continues until the Most Significant Bits (MSB) of the two integers are added. The time required to compute the complete integer addition is *n*τ (being *n* the number of bits), i.e., this time is proportional to the number of bits of the underlying arithmetic. Now consider the addition of three numbers. The operation is shown in Figure 10, and it begins with the addition of the two LSBs of the first two integers. When the first binary addition is ready, the carry triggers the addition of the next pair of bits (first two integers), but since the LSB of the first addition is ready, this LSB can be added (concurrently) with the LSB of the third integer. The result is that the total time required for the addition of the three integers is only increased by τ, i.e., the total time is *(n+1)*τ, which is just a fraction of the time it takes to compute the addition of two integers. In other words, with the architecture presented, if adding two integers takes time *T*, then adding three integers takes only *(1 + 1/n) T* –a somewhat counterintuitive result, but very beneficial in terms of computational acceleration. In a traditional CPU core, on the contrary, the addition of three integers would typically take (if you do it carefully) at least double the time that the addition of two integers would take.

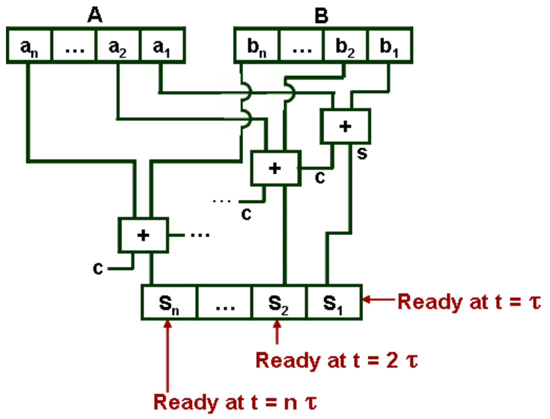

Figure 9. Overview of an implementation of bit-level operations for the addition of two integers. The computing time is proportional to the number of bits *n*.

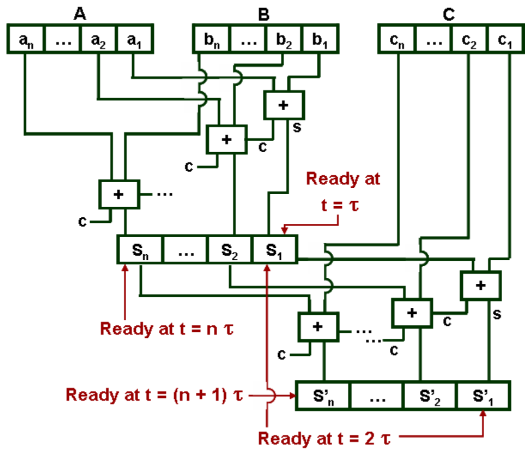

Figure 10. Overview of an implementation of bit-level operations for the addition of three integers. The computation time is proportional to approximately the number of bits *n*.

Good algorithm implementations will have concurrency levels on the order of thousands of basic blocks operating concurrently per clock cycle, and clock speeds on the order of 1 MHz to 500 MHz for the reconfigurable computing platforms available at the time of writing, depending on the particular application. Note that even though the clock speed is significantly lower than that of the fastest CPU, the level of concurrency coupled with the gains provided by coarse-grained and algorithm-level parallelism enables solutions that are thousands of times faster than what the best CPU can do.

Note that, at least in principle, data can be made available to the reconfigurable computing modules massively via wiring, thus reducing the need for storage in memory. This, in addition to the fact that many processing units can operate concurrently without the need for instruction fetch cycles, makes fine-grained parallelism the key ingredient responsible for getting rid of the von Neumann bottleneck in FPGA computing.

For the atomic LU algorithm, the critical operation is that of the so-called "critical path" of the architecture, which is depicted in Figure 11. The careful reader should be able to identify these operations directly from the atomic architecture depicted earlier in Figure 2.

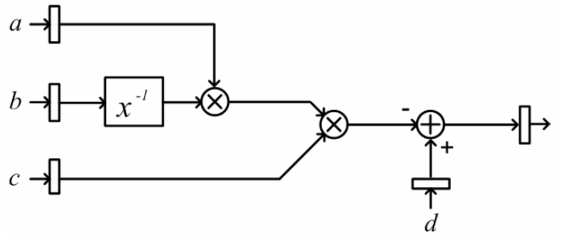

Figure 11. Architecture of the most critical operations in the atomic LU algorithm, known as the "critical path" of the system. Any acceleration implemented in this mega-operator will translate into a speedup of the overall algorithm.

The prototype contains a very aggressive arithmeticaccelerated mega-operator for this critical path, by making use of logarithmic arithmetic (instead of the traditional floating-point operators). This mega-operator, which is termed Turbo-LU, rendered a 5× speedup while providing the same levels of accuracy as if traditional floating-point arithmetic had been used.

VI. PORTABILITY, EASE OF USE, AND ALGORITHMIC SPEED BALANCE

The four mission-critical areas identified at the core of this work as key to groundbreaking and commercially-successful FPGA computing solutions are: portability, ease of use, algorithmic speed balance, and communication speed.

Aspects of the design from the perspective of the mission-critical areas are discussed in the following sections. Before proceeding, it is important to emphasize that both portability and ease of use have been a key requirement during the conception and development of this technology. The use of the system, as it stands today, does not require for the user to know neither VHDL nor FPGA technology. It is possible for a user to exploit the groundbreaking speedups of this technology by simply calling the prototype libraries from a language like C or Fortran, using a traditional development environment in a CPU.

To approach the mission-critical areas using a systematic methodology, the design of the solver is completely modular, in such a way that each component can be seen as an independently entity. Two different devices are part of the solver: a host PC and a reconfigurable computing platform. Figure 12 is a block diagram of the solver, when it is connected to a host PC.

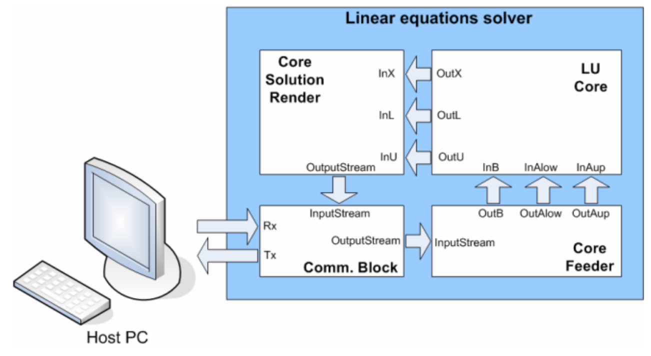

Figure 12. Block diagram of Accelogic LU-based solver as implemented in the current prototype.

VII. DESCRIPTION OF PROTOTYPE

A. *Linear equation solver in the host PC*

The host PC is the device that runs the high-level applications that the final user manipulates. In order to guarantee full compatibility of the linear equation solver, a C++ library that interfaces the reconfigurable computing platform with traditional programming languages was developed. The library, which includes a wrapper that mimics the input and output parameters of the LAPACK function DGBTRS (which solves a system of linear equations using matrix factorization), can be easily called from virtually any high-level programming language, thus guaranteeing both portability and ease of use. The library includes only two internal functions that depend on the communication interface: *"TransmitData"* (reads problem data directly from memory and sends it to the solver platform) and *"ReceiveData"* (receives the result from the solver platform and writes it directly into the host PC memory). Current implementations of *"TransmitData"* and *"ReceiveData"* execute on a variety of communication channels, including very fast PCIe communication. This layout is of primary importance for the system to achieve hardware and interconnect portability.

B. *Linear equation prototype solver in the reconfigurable computing platform*

The linear equation solver that runs in the reconfigurable computing platform (a ML402 FPGA board from Xilinx[3]) is built up from four modules: Communications Block, Core Feeder, LU Core, and Core Solution Render. All the components are implemented using standard VHDL functions, in order to guarantee porting of the source code to a wide number of FPGA systems. Particular definitions that depend on the FPGA system that supports the solver are condensed in a parametric way in a definition package that is part of the source code.

The four blocks communicate through high-speed buses, using a specific protocol and control system that makes the internal definition and design of each of the modules independent from the others. This is especially important in the case of the Communication Block, since different implementations of this module are required depending on the communication interface between the reconfigurable computing platform and the host PC.

Another internal component of key importance is the interface to memory devices. There are different types of memory whose access protocol can change depending on factors like the reconfigurable computing device, host PC, technology, etc. Given the wide variety of memory devices, it is important to make the FPGA solver independent from the memory interface, so to guarantee portability. The current prototype implements memory portability wrappers, called "stacks." A stack in the solver is a parameterizable minimal unit of memory, for which its internal implementation can be modified according to the particular memory device that is being used. Every component of the solver that requires the

[3] A second implementation exists in a very-high-performance hardware system procured from Dini Group, with exactly the same architecture.

use of memory accesses it through the stacks (instead of directly), making the migration between memory architectures completely transparent to the algorithm. This form of building the design is key for the system to achieve full reconfigurable-computing hardware portability.

Figure 13 shows a block diagram of the LU Core, the computational unit that implements the atomic LU architecture/algorithm combined with the solver of linear equations, as described in the subsection “Atomic LU algorithm in reconfigurable computing”. Figure 14 and Figure 15 show the block diagram of Core Feeder and the Core Solution Render, auxiliary modules that provide data to the LU Core, and process the solution delivered from the LU Core to be sent to the Communication Block.

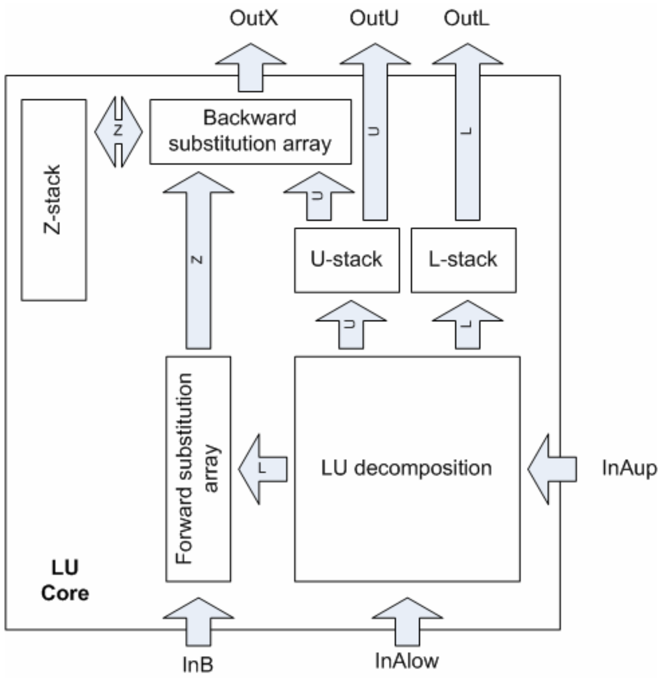

Figure 13. Block diagram of the LU Core architecture/algorithm that runs in the FPGA

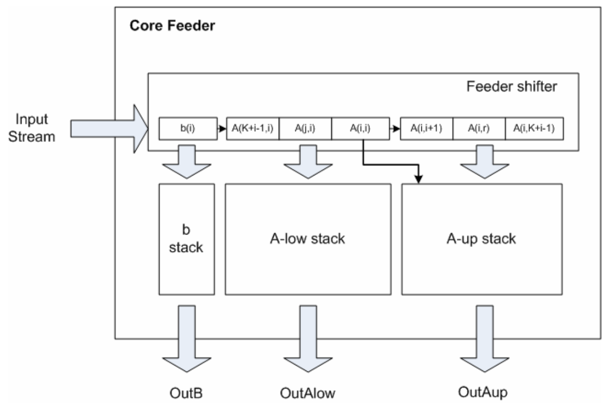

Figure 14. Block diagram of the Core Feeder

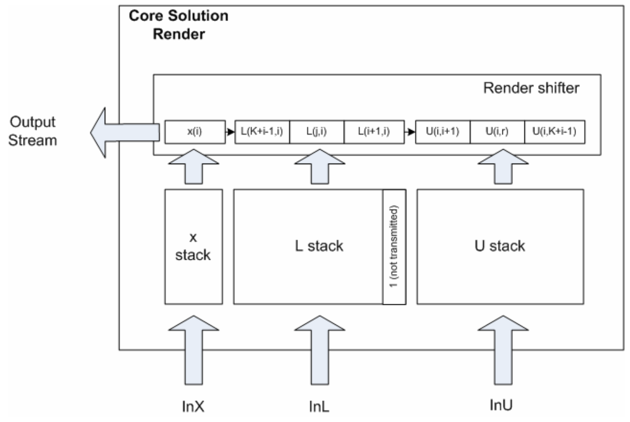

Figure 15. Block diagram of the Core Solution Render

VIII. RESULTS

As a result of this work, a working prototype of an FPGA band linear equation solver was designed and implemented. The prototype was developed using a combination of VHDL and C++ programming languages. VHDL was used for the description of the algorithm in reconfigurable computing, while C++ was used to create the user interface, in such a way that final users only require including a standard C++ library call to integrate the solver in their applications. No knowledge of VHDL or reconfigurable computing is required from end-users of this technology.

Using this C++ library, a demo in MATLAB® was implemented, which runs a benchmark environment comparing the performance of the implemented FPGA algorithms against LAPACK’s solver functions DGBTRF and DGBTRS, which are the standard functions for solving banded systems of linear equations via LU factorization. The prototype was synthesized in the ML402 commercial off-the-shelf FPGA board distributed by Xilinx, as well as in the high-performance Virtex 6 system of Dini Group. The prototype, as it stands today, is portable to any Virtex-type FPGA board, and can be plugged using a variety of communication interfaces to any PC running Windows.

The prototype demonstrated a speedup of 60x when compared against the world's most widely used software for linear equations (LAPACK) running on the Intel Woodcrest CPU. Additional benchmarks are being performed against other CPUs, and using other base FPGA platforms, and will be reported elsewhere. This level of performance is, to the authors' knowledge, the fastest at the time of writing. Figure 18 shows a contour plot of the experimental speedup measurements obtained as a function of both matrix size and matrix bandwidth . The plot on the left shows the speedups obtained using the Xilinx XCV4SX35 prototype board. The plot on the right shows the speedup obtained with the Xilinx XCV5LX220. Both plots were obtained comparing against the Intel Xeon Woodcrest (single core). The "area constraint" dashed line indicates the maximum bandwidth that can be supported by the use of a fully parallel solution with the hardware system. Larger matrix bands require either the use of a multi-FPGA board or a different FPGA algorithm.

A. A remark on the interpretation and usability of the 60x speedup

The results in Figure 16 compare raw processing power in the reconfigurable computing unit without considering the effects of the delay in the bilateral communications that must take place between the CPU and the FPGA. A real practical system should consider not only the FPGA computing speed, but also the communication speed, when assessing the performance of an FPGA algorithm. Suppose an FPGA system with a speedup factor of 1,000×. If the communication channel connecting the FPGA and the host CPU is extremely slow, then the time it would take to transmit the problem data and/or the solution would be longer than the time it would take the CPU to solve the problem by itself, thus making the effective speedup vanish entirely. The FPGA would be useless.

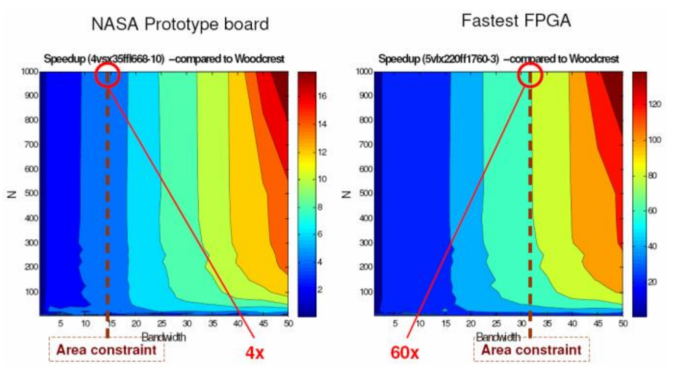

Figure 16. Contour plots of the speedup of the FPGA linear equation solver, as a function of the size of the matrix (N) and the matrix bandwidth (i.e., the number of diagonals that cover all the data in the sparse matrix). For each plot, the thin bar on the right-hand side indicates the mapping associated with the color scale (in general, hot colors like red indicate larger speedup factors).

The communication process plays a key role on the performance of a networked product, and its effect should not be underestimated.

In order to illustrate the effect of the communications channel, Figure 17 shows an estimate of the overall speedups that are achievable when both computation and communication speeds are jointly considered. Note that, when the communication speed is zero, the overall speedup is always zero, whereas when the communication speed is infinite, the overall speedup is equal to 60×, the FPGA computation speedup. The larger the communication speed, the larger the effective speedup. The green vertical lines show typical speedups that are achievable with current (and inexpensive) commercial communication interfaces. PCI Express, for example, renders an effective speedup larger than 45x for the current prototype.

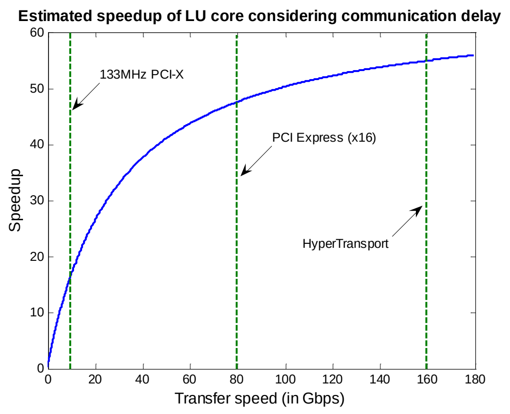

Figure 17. Effect of the communication speed on the performance of the solver when the matrix bandwidth is 30 and N is large. The speed and corresponding speedup of three (currently commercially available) interfaces is shown: PCI-X, PCI Express, and HyperTransport. This plot considers neither data compression nor streaming techniques that will be introduced in this project later. The use of efficient data compression methods and streaming may increase the final speedup.

IX. CONCLUSION

The work demonstrates that a change in the mainstream FPGA programming paradigm is required in order to achieve accelerations of orders of magnitude ahead of implementations over von Neumann computer architectures. Several key innovations at the algorithm level are presented, such as the atomic LU factorization, algorithm-aware adaptive precision, and algorithm optimization at the bit level. Several important commercialization enablers are also presented, such as portability, ease of use, and algorithmic speed balance. The concurrent use of these algorithmic strategies and commercialization enablers, breaks with the tradition of porting a computing algorithm to faster CPUs, and produces an acceleration of 60× for the LU factorization problem, even when available FPGA clock speeds are today much slower that those found in a high-end CPU.

A wide range of applications in high performance computing is envisioned for this technology, encompassing industries as varied as aerospace and automotive vehicle design, oil & gas exploration, circuit simulation, biomechanics, medical image processing, geophysics, ad-hoc networking models, econometric models, linear programming, and many more. This wide collection of applications demonstrates the importance of maturing this area of knowledge and making the results promptly available to industry in the form of reliable commercial solvers.

ACKNOWLEDGMENT

We wish to thank the NASA benchmarking personnel for their collaboration with NPB benchmark reports, as well as Gene H. Golub (R.I.P) from Stanford University, and Jack Dongarra from the University of Tennessee, for their consultation on this project. This material is based upon work supported by NASA, under contract numbers NNX07CA19P and NNX10RA04C. Neither the U.S. Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the U.S. Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

REFERENCES

[1] Golub and C.F. Van Loan. *Matrix Computations*, Johns Hopkins University Press; 3 ed, 1996.

[2] P. Rajesh Kumar, K. Sridharan, S. Srinivasan. *"A parallel algorithm, architecture and FPGA realization for landmark determination and map construction in a planar unknown environment"*. In: *Parallel Computing* 32 (2006) 205–221

[3] J. Gonzalez, A. Upegui, R. C. Nunez, D. Orozco. *"High Performance Non- von Neumann Algorithms for Large-Scale Optimization".* Presentation at the 31st AIAA Dayton-Cincinnati Aerospace Sciences Symposium. Dayton, Ohio. 07 March 2006

[4] Accelogic LLC. *Algorithm Design in the Era of Reconfigurable Computing – Course Notes*. Accelogic Training Series on HPC with Reconfigurable Computing. 2006 – 2007.

[5] J. J. Dongarra. *"Performance of Various Computers Using Standard Linear Equations Software*", (Linpack Benchmark Report), University of Tennessee Computer Science Technical Report, CS-89-85, 2006.

[6] Xilinx. Virtex-5 Family Overview LX, LXT, and SXT Platforms. Available online at: http://direct.xilinx.com/bvdocs/publications/ds100.pdf. May 2007

[7] D. Veley, J. Gonzalez, R. Nunez. *"Reconfigurable Computing in Engineering Mechanics"* presented in the: 18th Engineering Mechanics Division Conference (EMD2007), 2007