36th AIAA Thermophysics Conference
25–28 June 2007, Miami, FL

AIAA Paper # 2007–4085

Large-Scale Numerical Solution of Partial Differential Equations with Reconfigurable Computing

Rafael C. Núñez [†] and Juan G. Gonzalez [‡]
Accelogic, LLC
Weston, FL, USA

José A. Camberos[*]
U. S. Air Force Research Laboratory
WPAFB OH 45433

We describe the shift in paradigm that the algorithm developer must undertake in order to design efficient numerical solutions using Field Programmable Gate Arrays (FPGAs) and reconfigurable computing in general. FPGAs and reconfigurable computing can bring a new dimension to the algorithm design process, with potentially huge gains in algorithm speed, and significant reductions in cost and power consumption. We introduce three levels of parallelism that are possible under this emerging computing paradigm: fine-grained parallelism, coarse-grained parallelism, and algorithm-level parallelism. Furthermore, we illustrate through an example how these levels of parallelism work. Full exploitation of the three levels of parallelism will be a fundamental ingredient for developing the next generation of partial differential equation solvers during the next decade.

I. Introduction

AN incredibly large number of applications in science and engineering have benefited from mathematical computing during the last decades. The influence of mathematical computing has been enormous, to the point that computational modeling and simulation have become important enablers of scientific and engineering progress, on par with physical experimentation and mathematical theory.

Problems in Computational Fluid Dynamics (CFD) usually require large-scale or high-performance computing resources, thrusting this field to be traditionally at the vanguard of scientific computing. The solution of partial differential equations (PDEs), in particular those applied to CFD, has attracted the interest of numerous researchers over the years. PDE-intensive applications like aerospace system design, space power and propulsion, plasma physics, etc., have required extensive computational power, and it is expected that they will require much more during the next years.

Recently, a new programming paradigm is emerging, based on advances in field-programmable gate arrays (FPGAs), that allows a developer to tailor the computer architecture to the numerical problem for optimal performance. This architecture permits the developer to fully exploit the possibilities inherent in parallel computing at a very fundamental level, achieving speedups from hundreds to hundreds-of-thousands or more, depending on the particular application.

[†] Chief Hardware Architect, Accelogic LLC, `www.accelogic.com`, AIAA Student Member
[‡] President & CTO, Accelogic LLC, `www.accelogic.com`, AIAA Member
[*] Research Aerospace Engineer, Air Force Research Laboratory, AIAA Associate Fellow

This document outlines some important concepts and conclusions from our work towards developing path-breaking algorithms for partial differential equations (PDEs) that take advantage of the emerging field of reconfigurable computing.

II. High-Performance Computing

Scientific computing enables the numerical modeling of natural phenomena and engineering machines in terms of mathematical statements, typically a formula or equation to be solved. The equations are traditionally solved by large-scale numerical approximation methods such as finite-differences, finite-volume, and finite-element approaches. Large-scale approximation problems are solved through advanced numerical programming in powerful computing engines – an area known as High Performance Computing (HPC). Most of the numerical computing power in HPC today comes from von Neumann-based processors, usually arranged in very large CPU clusters with optimized communication interfaces.

In the design of algorithms for performance, the trend has been to "tune" the code for optimum execution within the underlying architecture. In the early days of numerical computing, memory was a precious commodity, and the designs were driven by the "save memory" rule. Nowadays, when memory is not an issue anymore, special consideration is given to code optimizations that take advantage of CPU architecture features, such as memory cache hierarchies, multi-core components, and data bus technologies. Code "porting" is a typical term for this type of effort – the code is ported to an architecture that has been pre-built by the hardware manufacturer.

Significantly higher computational power at lower economic cost can be attained through the use of reconfigurable computing (FPGA-based solutions). FPGAs are semiconductor devices for which both architecture and dataflow can be programmed – the architecture is not pre-built, as with the traditional CPU paradigm. When used in HPC, FPGAs can serve as programmable computing units able to execute specialized routines incredibly faster than general-purpose von Neumann machines. Contrary to the traditional practice of HPC, in which algorithm development consists of the design of (possibly threaded) sequences of CPU instructions, the development of algorithms in reconfigurable computing is a dual design game, where both architecture and dataflow are free to be designed to suit the mathematical equation at hand. This new way of doing things, opens tremendous opportunities for the discovery of extremely efficient multidisciplinary algorithms never thought of before, opening a new research field at the intersection of mathematics, computer science, and engineering application domains.

The feasibility of HPC with FPGAs has been already demonstrated in different application domains. Some of the most impressive results are described in References [1], [3], [4], and [9]. The potential for FPGA supercomputing is such that supercomputer vendors have already begun to incorporate FPGAs in their newest hardware products [5], [6].

In the following, we use the method of finite differences as an example to illustrate the advantages of using FPGAs for the solution of PDEs. The proposed solution is based on an intrinsically parallel approach that can solve the problem very efficiently, with significant speedups and cost reductions when compared against the traditional CPU-based supercomputing solutions.

III. Designing Good Reconfigurable Computing Algorithms

Proficiency in algorithm design for reconfigurable computing requires many hours of dedication and concentration from the developer. Even if the algorithm implementation cycle in reconfigurable computing is similar in principle to that of a traditional software solution, mastering the art of reconfigurable computing algorithm design requires a paradigm shift in the thought process of the algorithm developer [2], [7].

When transitioning to reconfigurable computing, one of the first approaches that a novice FPGA developer may attempt would also, unfortunately, is to "port" or "translate" an existing CPU-based algorithms to the FPGA platform. To illustrate this, it is crucial to understand that CPU algorithms are designed for a fixed –and very particular–computing architecture (serial, von Neumann), and solutions that work well on von Neumann machines, are not necessarily the best ones when a more general (customizable) computing architecture is available (reconfigurable computing).

Thus, in order to design good reconfigurable computing algorithms, the designer should unlearn the von Neumann programming "tricks." This unlearning process liberates the designer from previous architecture-dependent misconceptions that can harm his/her ability to build and discover groundbreaking algorithms. Once the algorithm designer is free from the von Neumann way of thinking, he/she must master the three hierarchies of parallelism that can be manipulated in reconfigurable computing: fine-grained parallelism, coarse-grained parallelism, and algorithm parallelism. Each hierarchy represents independent opportunities for new breakthroughs in computing performance. The best algorithms will undoubtedly arise from a balance among these three hierarchies.

A. Fine-grained parallelism:

We refer to "fine-grained" parallelism at a very fundamental level. Fine-grained parallelism refers to the computation of the most basic operations of the algorithm in parallel. It is one of the most important advances enabled by reconfigurable computing programming.

Fine-grained parallelism is defined at the level of single simple operations inside the algorithm that can be executed concurrently in a single clock cycle. The traditional concept of CPU "instructions" for basic operations, as known in the von Neumann paradigm, does not apply at the fine-grained parallelism level. Under the new paradigm, when referring to fine-grained parallelism, we speak in context of dataflow instead of instructions, because data "travels" between computational modules in space[*]. Very complex modules can be designed to create the equivalent of "mega-instructions" that execute with massive parallelism in a single clock cycle.

To illustrate this concept, consider for example an algorithm to calculate the sum of four integer numbers x_i, $i = 1,\ldots,4$, where $x_i \in [0, 1000]$. In a von Neumann architecture, the algorithm can be described simply as

$$y = x_1 + x_2 + x_3 + x_4$$

[*] This "travel in space" occurs typically via physical wires, as opposed to the traditional "travel in time," through memory read/write operations, that occurs in a CPU (von Neumann) dataflow.

using a high-level programming language such as C or FORTRAN. The compiler translates this description into a more detailed algorithm which would look similar to the following data flow in an adder/accumulator inside the CPU:

1) $y = x_1 + x_2$
2) $y = y + x_3$
3) $y = y + x_4$

Note that, using a von Neumann architecture, the execution of the algorithm above requires multiple clock cycles. Each step of the algorithm involves the reading of a variable from memory, and then the addition of two variables. Both operations typically require more than one single clock cycle. The same algorithm is shown in Figure (1) in a reconfigurable computing architecture using "code porting" – *poor algorithm design practice*.

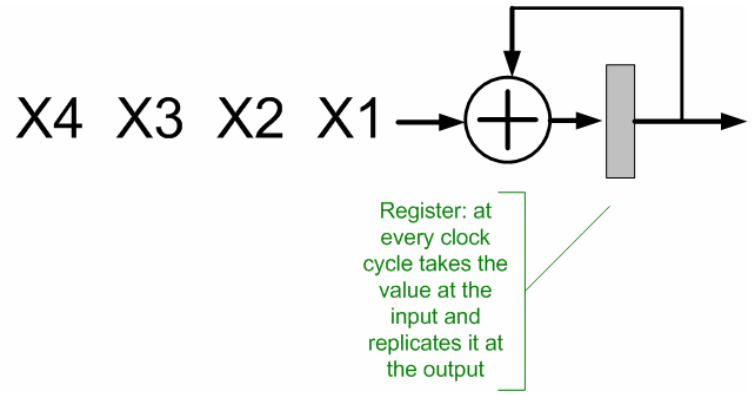

Figure 1: Addition of four integers using a "ported" algorithm from a von Neumann CPU.

The execution of the algorithm in Figure (1) requires four clock cycles in an FPGA-based computing architecture. However, the same problem, i.e., the addition of four numbers, can be solved in a single clock cycle by rethinking the problem in context of reconfigurable computing. This leads to a better, faster solution. It may be obvious that there is no unique algorithmic solution to the problem, and here we describe only two of them. The first solution, shown in Figure (2), takes advantage of the functional capability to instantiate any number of adders in the architecture, all of them executing concurrently in a single clock cycle.

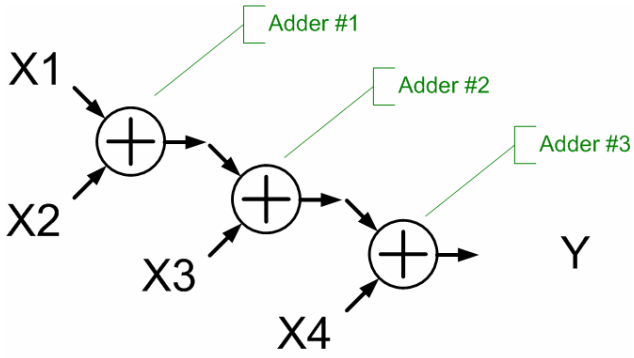

Figure 2: Addition of four integers exploiting fine-grained parallelism (Solution 1).

The time required to execute the algorithm in Figure (2) is proportional to the number of basic blocks that are in the path between the input and the output.[†] In this case, the data must flow through three blocks connected in cascade (serial connection). A more efficient algorithm, which makes better use of the fine-grained parallelism, is depicted in Figure (3), Solution 2.

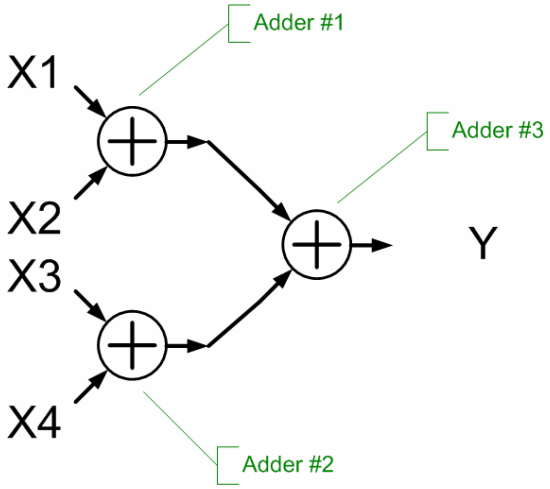

Figure 3: Addition of four integers by exploiting fine-grained parallelism. This solution is faster than Solution 1, while consuming exactly the same amount of resources.

Both of the algorithms described in Figures (2) and (3) can be executed in a single clock cycle. However, since the number of basic blocks from input to output in Figure (3) is reduced to two, the algorithm may lead to faster clock speeds than the one in Figure (2) – the clock speed in reconfigurable computing is a parameter controlled by the user! The essence of the game in reconfigurable computing is to use fine-grained parallelism whenever possible to enable very wide data flow (maximal concurrency) with very short paths between input and output (maximal

[†] Exceptions to this rule exist, which can be further exploited to gain computational speedups. This discussion is, however, beyond the introductory scope of this paper.

clock speed). Good solutions 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, depending on the particular application. Note that even though the clock speed is significantly lower than that of the latest CPUs, 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 on massive scale via electrical 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 overcoming the von Neumann bottleneck in reconfigurable computing.

B. Coarse-grained parallelism:

Fine-grained parallelism enables the design very fast algorithms. Nevertheless, algorithm design using only fine-grained parallelism limits the scope of applications in reconfigurable computing. There are two reasons for this limitation: first, for most designs it is not possible to describe all components with the level of detail required by fine-grained parallelism. Higher levels of abstraction of the problem are required to maximize computational efficiency in the algorithm design process. Second, the use of fine-grained parallelism has a penalty associated with the area and delay in the reconfigurable computing platform. In other words, the available hardware resources are limited.

The algorithm designer is then in charge of determining what are the modules of the design that, once programmed using fine-grained parallelism, enable significant gains in algorithm performance.

There is no clear delineation between fine- and coarse-grained parallelism. However, coarse-grained parallelism is commonly associated with the concurrent execution of multiple, high-level, independent functions. It is closely related to the familiar “domain decomposition” partitioning implemented in parallel algorithms today. Nevertheless, contrary to parallel computing where the network topology is fixed, in reconfigurable computing the designer is free to implement the network topology that is best suited to speed-up the solution of a given problem. The appropriate use of coarse-grained parallelism solves the scalability problem, which is one of the main limitations of von-Neumann-based parallel systems.

C. Algorithm-level parallelism:

This is the highest abstraction level of parallelism. It is available at the top of the parallelization conceptualization, and is related to the capability of implementing parallelized algorithms (combinations of synchronous/asynchronous computations). This kind of parallelization may make no sense in von Neumann architectures, and only exists because of the availability of a fully customizable computing architecture.

This concept is not trivial, and it is easier to explain with an example. The next section introduces the fine grained and coarse grained parallelism concepts, and provides some insights on the design of algorithm-level parallelization using a well-known example from CFD.

IV. Example: Accelerated Algorithms for Elliptic PDEs

In this Section, we illustrate how the reconfigurable computing paradigm can be used to solve a large-scale partial differential equation. The emphasis is on illustrating the different levels of parallelism enabled by reconfigurable computing, rather than on the algorithms themselves. To this end, we limit the discussion to an "easy" equation, namely, the 2D Laplace equation in a rectangle. In particular, we want to solve:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0 \,, \tag{1}$$

with Dirichlet boundary conditions on a square domain. The mesh is uniform, equally spaced in two dimensions, as shown in Figure (4).

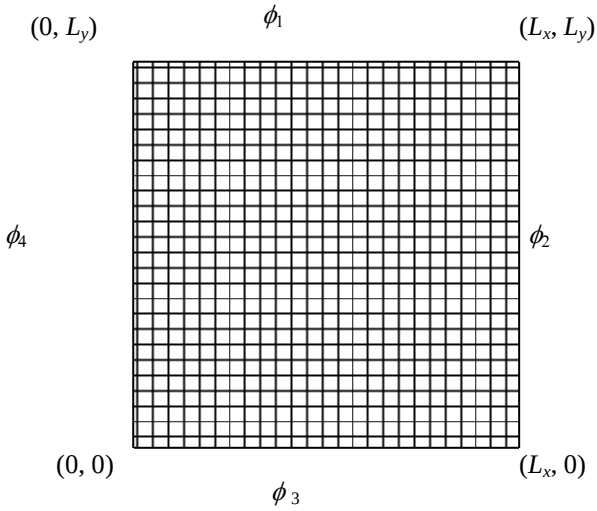

Figure 4: Uniform equally spaced mesh for the 2D Laplace equation in a rectangle.

A numerical approximation to Laplace's equation can be obtained through standard central finite differences,

$$\frac{\partial^2 \phi}{\partial x^2} \cong \frac{\left(\phi_{i+1} - 2\phi_i + \phi_{i-1}\right)_j}{\Delta x^2}, \qquad \frac{\partial^2 \phi}{\partial y^2} \cong \frac{\left(\phi_{j+1} - 2\phi_j + \phi_{j-1}\right)_i}{\Delta y^2}, \tag{2}$$

where the indices i and j denote grid point locations in x and y respectively. There is a wide variety of advanced methods that can be used to tackle this problem. We restrict our attention to the so-called "point Jacobi" explicit finite differences. With this method, we seek to solve the field ϕ as the steady-state solution (n tends to infinity) of the finite differences equation:

$$\phi_{i.j}^{n+1} = \tfrac{1}{4}\left(\phi_{i+1.j}^{n} + \phi_{i-1.j}^{n} + \phi_{i.j+1}^{n} + \phi_{i.j-1}^{n}\right). \tag{3}$$

A typical implementation of this method in a CPU makes use of three "do loops" that sweep the variables *i*, *j*, and *n*, processing expression (3) one grid point at a time. This type of computation can take many CPU clock cycles – cycles cursed by the sluggishness of the von Neumann bottleneck. Although the sluggishness of the von Neumann machine can be compensated by re-writing the code so that it can take advantage of the pipeline of the CPU, the latter algorithm is still an isomorphism of the von Neumann architecture, and as such it cannot get rid of the intrinsic limitations of the architecture. We understand that the Jacobi algorithm is one of the slowest for the iterative solution to this kind of problem. However, it is also well-known that the algorithm often outperforms faster methods (e.g., Gauss-Seidel) when implemented on vector computers (e.g., CRAYs). While we only use the Jacobi method to illustrate the kind of thinking needed to fully exploit the power of reconfigurable computing, it may turn out to be a viable competitor with the fine-grained parallelism now available beyond vectorization.

To speed-up a large-scale computing problem, researchers have cleverly developed a level of parallelism using domain decomposition. In this method, the mesh is divided into several small sub-meshes, and each sub-mesh is assigned to a different processor or CPU. This approach is the most common example of coarse-grained parallelism in high-performance computing today. Although the approach does in effect provide acceleration, it still suffers from the von Neumann bottleneck inside each one of the CPUs. Also, the approach requires the number of CPUs to be relatively small compared to the total number of grid points; otherwise efficiency suffers due to the excessive inter-processor communication required to propagate the solution across the global domain. This communication overhead among the CPUs makes the problem computationally inefficient and difficult to automatically scale effectively for a large number of CPUs.

Reconfigurable computing gives us a new dimension of thinking, by enabling the design of novel and faster algorithms harnessed onto specialized machine architectures that are more "natural" to the algorithm itself. In other words, instead of tuning the algorithm to conform to fixed, general purpose hardware architectures, we tune the hardware itself to solve the problem as best as possible. A direct examination of the Jacobi algorithm, Equation (3), suggests a basic processing "atom" for a reconfigurable computing solution[‡], as depicted in Figure (5). Since this atom is the basic computing unit to solve the Laplace equation, we call it the "Laplace Atom."

Our Laplace Atom is optimized to perform three very fast additions, followed by a multiplier with practically no latency. The configuration of the adders in the atom is the same pyramid configuration that we described as the most efficient 4-integer adder in Section III. Furthermore, the implementation of the multiplier required by Equation (3) is very fast and simple in reconfigurable computing. In general any multiplication (or division) by a power of two, given the binary notation of the underlying arithmetic, is obtained by simply shifting the fractional point to the right (or to the left, in the case of a division) in binary notation.

[‡] Throughout our work, we use the term "atom" to denote a basic conceptually simple computing unit that is typically re-used massively in the algorithm. An efficient (fully concurrent) implementation of the atom means an efficient use of fine-grained parallelism. Atoms are typically executed in a single clock cycle within a data flow.

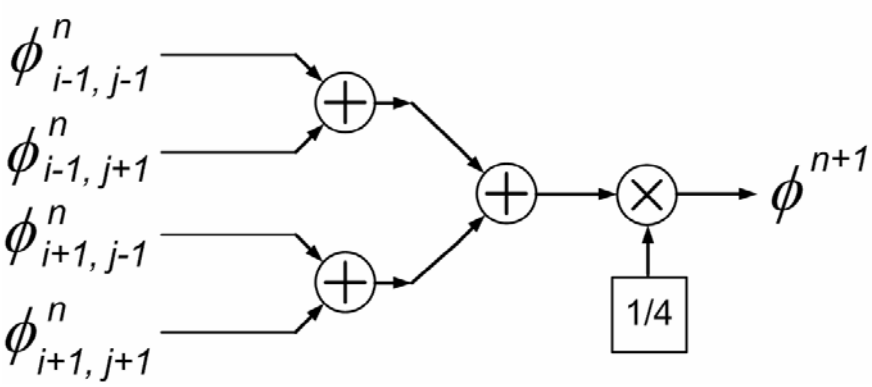

Figure 5: Laplace Atom – Optimized computing unit for the solution of the Laplace equation.

Table I compares the performance of the Laplace atom running in a top-of-the-line CPU vs. a top-of-the-line FPGA. The CPU selected for the test is an Intel Xeon Woodcrest CPU, a powerful von Neumann engine recognized as the fastest today according to the Linpack test [7]. The FPGA used in this experiment is a Xilinx Virtex 5 LX220-3 [8], one of the largest and fastest FPGAs currently available in the market. As can be seen in the table, the Woodcrest runs at a faster clock frequency than that of the Virtex 5. However, the number of clock cycles required for one execution of the Laplace atom is higher in the Woodcrest than in the FPGA, where the design requires only one clock cycle. The result is a reconfigurable computing algorithm that executes the Laplace atom thirty-five times faster than the Woodcrest.

TABLE I: Execution of one operation of the Laplace atom in two different computing engines

Device executing the Laplace atom	Clock frequency (MHz)	Number of clock cycles	Time (ns)
Intel Xeon Woodcrest CPU	3,000	130	43
Xilinx Virtex 5 FPGA	822.37	1	1.22

The implementation of a single Laplace atom in the FPGA used for this study consumes only 0.09% of the computing resources of the FPGA. Thus, we may use the remaining resources via coarse-grained parallelism to increase the processing power even more. This is achieved by replicating the Laplace atom multiple times, so that a large number of points of the Jacobi iteration are computed in parallel at each clock cycle, as shown conceptually in Figure (6) and Figure (7) below.

The replication of the Laplace atom multiple times in the FPGA is a powerful example of coarse-grained parallelism. Table II shows results of the computation of 800 points of the Jacobi algorithm in both the Intel Xeon Woodcrest (CPU with traditional von Neumann software) and the Xilinx Virtex 5 (FPGA with reconfigurable computing software). In this case, the Woodcrest can take advantage of advanced pipelining techniques to reduce the total computing speed, and

the FPGA implements a solution that exploits both fine- and coarse-grained parallelisms. The result: the speedup obtained with the reconfigurable computing unit is 2,523×. We note here that the discussion presented here is simplified and introductory. A final operational solution needs to consider other aspects in the design, such as data communication delay and distributed memory design, which are beyond the scope of the discussion in this paper. After taking those factors into account, the final speedup of the design could be lower.

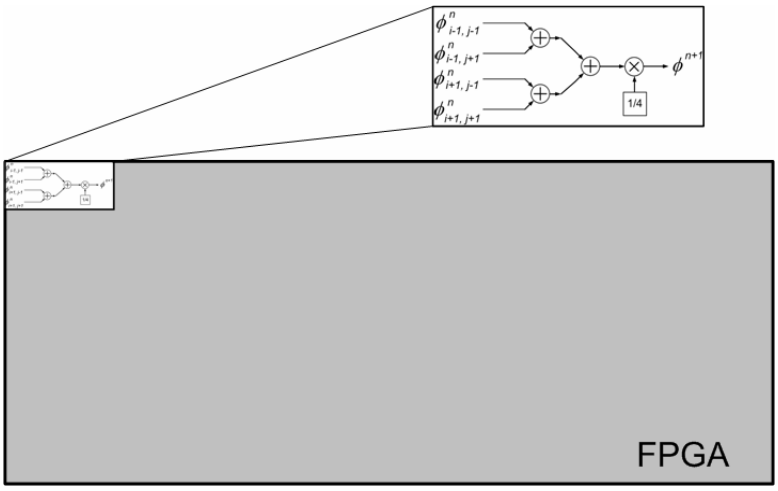

Figure 6: Fine-grained parallelism for the solution of the Laplace equation. A single Laplace atom occupies only a small fraction of the computing resources of the FPGA (only fine-grained parallelism shown here).

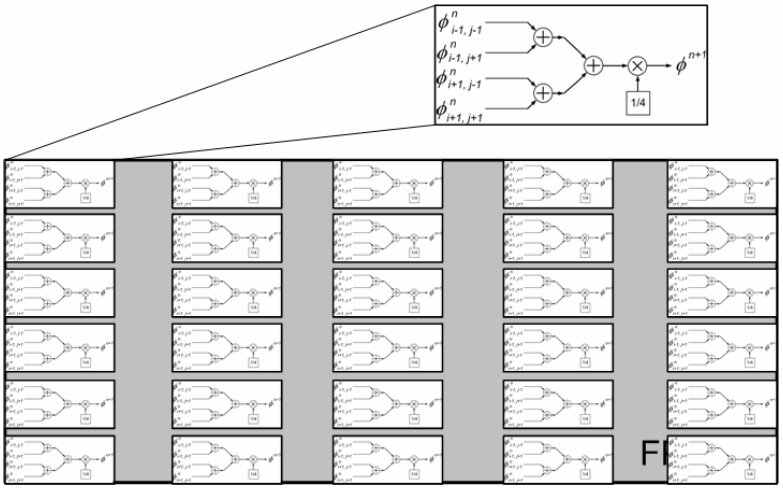

Figure 7: Coarse-grained parallelism for the solution of the Laplace equation. Parallel Laplace atoms are instantiated, thus taking advantage of all the computing resources in the FPGA and providing additional computing power (both fine-grained and coarse-grained parallelism levels are integrated and used).

Even if the performance that can be obtained combining fine- and coarse-grained parallel approaches is on the order of hundreds or thousands for the application addressed in this section, the use of algorithm-level parallelism can provide additional speedup to the solution. Algorithm-level parallelism runs on top of the speedup levels shown above and the most amazing acceleration factors are usually obtained from its use. However, creating solutions that use algorithm level parallelism is not as simple as exploiting fine- and coarse-grained parallelisms. Algorithm-level parallelization requires rethinking the problem entirely, sometimes even at the level of the mathematical foundations behind the algorithm.

TABLE II: 800 operations of the Laplace atom executed in two different computing engines.

Device executing the Laplace atom	Clock frequency (MHz)	Number of clock cycles	Time (ns)
Intel Xeon Woodcrest CPU	3,000	9,214	3,068
Xilinx Virtex 5 FPGA	822.37	1	1.22

An interesting example of algorithm-level parallelization is the development by the authors of LAPACK–RC, a reconfigurable computing library for the accelerated solution of large-scale numerical linear algebra problems. It is expected that this library will provide a speedup factor of 1,000× over CPUs by 2009. Two representative routines in this library are based on LU and Cholesky factorizations, respectively. The linear equation solver based on LU factorization is currently operational, and offers a 60× speedup. The Cholesky-based solver is also available and operational, offering a 50× over the best CPU today [9]. This is an important result, especially considering the fact that the solution of linear equations is an extremely well-studied problem, fully optimized for the execution on von Neumann architectures over the past 50 years. The interested reader may obtain additional information about this emerging technology by contacting the authors directly.

V. Conclusions

Modern scientific computing faces many challenges, including increasingly complex programming and computer hardware architecture. Recently, reconfigurable computing is emerging as a new programming paradigm based on field-programmable gate arrays (FPGAs). Reconfigurable computing enables a developer to tailor the computer architecture to the mathematical problem, thus breaking the limitations of the traditional CPU (von Neumann) architecture. This programmable architecture allows the developer to fully exploit the possibilities inherent in parallel computing at a very fundamental level.

Proficiency in algorithm design for reconfigurable computing is only achieved with practice. Even if the algorithm implementation cycle in reconfigurable computing is similar in principle to that of a traditional software solution, mastering the art of reconfigurable computing algorithm design requires a paradigm shift in the thought process of the algorithm designer [2], [7]. This paradigm shift requires two conditions: First, unlearning the von Neumann programming "tricks;" second, mastering the three levels of parallelism that can be manipulated in reconfigurable computing, namely fine-grained parallelism, coarse-grained parallelism, and algorithmic parallelism.

Once the algorithm designer is set free from the von Neumann bottleneck and in control of the multi-level parallelism enabled by reconfigurable computing, the construction of innovative numerical solutions is limited only by the creativity of the designer. Expertise in reconfigurable computing allows the algorithm designer to create solutions that can easily run hundreds or hundreds-of-thousands of times faster than their von Neumann counterparts.

Acknowledgments

We are grateful to Dr. Duane Veley from the Air Force Research Laboratory for helpful discussions, and to the Air Vehicles Chief Scientist Innovative Research Fund (CSIRF), the Air Force and NASA SBIR Programs, and Accelogic LLC, for partial sponsorship of this work.

References

1 BioXL/S. http://www.biocceleration.com/BioXLS-General.html. September 7, 2006

2 Juan G. Gonzalez and Rafael C. Nunez. Reconfigurable Computing: Algorithm Design Guidebook, Prentice Hall, 2008.

3 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". *Parallel Computing* 32 (2006) 205–221.

4 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.

5 SGI RASC Technology. http://www.sgi.com/products/rasc/. September 2006.

6 Cray XD1 Supercomputer. http://www.cray.com/products/xd1/index.html. September 2006.

7 Accelogic LLC. Algorithm Design in the Era of Reconfigurable Computing – Course Notes. *Accelogic Training Series on HPC with Reconfigurable Computing.* 2006 – 2007. http://www.accelogic.com/training.html.

8 Jack 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.

9 Xilinx. Virtex-5 Family Overview LX, LXT, and SXT Platforms. http://direct.xilinx.com/bvdocs/publications/ds100.pdf. May 2007.

10 D. Veley, J. Gonzalez, R. Nunez. "Reconfigurable Computing in Engineering Mechanics," 18th Engineering Mechanics Division Conference (EMD2007), 2007.