Available online at www.sciencedirect.com
ScienceDirect
Journal of Computational and Applied Mathematics 213 (2008) 79–94
JOURNAL OF COMPUTATIONAL AND APPLIED MATHEMATICS
www.elsevier.com/locate/cam

Multigrid solvers in reconfigurable hardware

Safaa J. Kasbah[a,*], Issam W. Damaj[b], Ramzi A. Haraty[a]

[a] *Division of Computer Science and Mathematics, Lebanese American University, Beirut, Lebanon*
[b] *Department of Electrical and Computer Engineering, Dhofar University, Salalah, Sultanate of Oman*

Received 15 July 2006; received in revised form 6 December 2006

Abstract

The problem of finding the solution of partial differential equations (*PDEs*) plays a central role in modeling real world problems. Over the past years, Multigrid solvers have showed their robustness over other techniques, due to its high convergence rate which is independent of the problem size. For this reason, many attempts for exploiting the inherent parallelism of Multigrid have been made to achieve the desired efficiency and scalability of the method. Yet, most efforts fail in this respect due to many factors (time, resources) governed by software implementations. In this paper, we present a hardware implementation of the V-cycle Multigrid method for finding the solution of a 2D-Poisson equation. We use *Handel-C* to implement our hardware design, which we map onto available field programmable gate arrays (*FPGAs*). We analyze the implementation performance using the *FPGA* vendor's tools. We demonstrate the robustness of Multigrid over other similar iterative solvers, such as Jacobi and successive over relaxation (*SOR*), in both hardware and software. We compare our findings with a C++ version of each algorithm. The obtained results show better performance when compared to existing software versions.

Keywords: Iterative methods; Parallelization; *FPGA*; Reconfigurable computing

1. Introduction

Physical, chemical and biological phenomena are modeled using partial differential equations (*PDEs*). Interpreting and solving (*PDEs*) is the key for understanding the behavior of the modeled system. The broad field of modeling real systems has drawn the researchers' attention for designing efficient algorithms for solving (*PDEs*). The Multigrid method has been shown to be the fastest method due to its high convergence rate which is independent of the problem size. However, the computation of such solvers is complex and time consuming. Many attempts for exploiting the inherent parallelism of Multigrid have been made to achieve the desired efficiency and scalability of the method. Yet, most efforts fail in this respect due to many factors (time and resources) governed by software implementations upon parallelizing the algorithm.

Over the past years, researchers have benefited from the continuous advances in hardware devices and software tools to accelerate the computation of complex problems [3]. At early stages, algorithms were designed and implemented to run

* Corresponding author. Tel.: +961 3788402.
E-mail addresses: safaa.kasbah@lau.edu.lb (S.J. Kasbah), i_damaj@du.edu.om (I.W. Damaj), rharaty@lau.edu.lb (R.A. Haraty).

0377-0427/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.cam.2006.12.031

on a general purpose processor (software). Techniques for optimizing and parallelizing the algorithm, when possible, were then devised to achieve better performance. As applications get more complex, the performance provided by processors degenerates. A better performance could be achieved using a dedicated hardware where the algorithm is digitally mapped onto a silicon chip, integrated circuit (*IC*). Though it provides better performance than the processor technology, the *IC* technology (hardware) lacks flexibility.

In the last decade, a new computing paradigm, reconfigurable computing (*RC*), has emerged [15]. *RC*-systems overcome the limitations of the processor and the *IC* technology. *RC*-systems benefit from the flexibility offered by software and the performance offered by hardware [22,15]. *RC* has successfully accelerated a wide variety of applications including cryptography and signal processing [21]. This achievement requires a reconfigurable hardware, such as field programmable gate array *FPGA*, and a software design environment that aids in the creation of configurations for the reconfigurable hardware [15].

In this paper, we present a hardware implementation of the V-cycle Multigrid algorithm for the solution of a 2D-Poisson equation using different classes of *FPGAs*: *Xilinx Virtex II Pro*, *Altera Stratix* and *Spartan3L* which is embedded on the RC10 board from *Celoxica*. We use *Handel-C*, a higher-level hardware design language, to code our design which is analyzed, synthesized, and placed and routed using the *FPGAs* proprietary software (*DK* Design Suite, *Xilinx ISE* 8.1*i* and *Quartus II* 5.1). We demonstrate the robustness of the Multigrid algorithm over the Jacobi and the *SOR* algorithms, in both hardware and software. We compare our implementation results with a software version of each algorithm, since there are no hardware implementations of *MG*, *Jacobi* and *SOR* in the literature.

The rest of the paper is organized as follows: in Sections 2 and 3, we present a general overview of Multigrid solvers and RC, respectively. In Section 4, we describe our hardware implementation of the V-cycle *MG* for the solution of a 2D-Poisson equation. Then, the implementation results are presented in Section 5, where we: (a) report *MG* results, (b) compare these results with a software version written in C + + and running on a general purpose processor, (c) report *Jacobi* and *SOR* hardware implementation results and compare them with their software versions, (d) compare the results obtained in (a) and (c) show how *MG* outperforms *Jacobi* and *SOR*, in both hardware and software versions. Section 6 concludes the work and addresses possible directions to future work.

2. Multigrid solvers

Multigrid methods are fast linear iterative solvers used for finding the optimal solution of a particular class of PDEs. Similar to classical iterative methods (*Jacobi*, successive over relaxation (*SOR*), Gauss Seidel, generalized minimal residual method (*GMRES*), bi-conjugate gradient, etc.), an *MG* method 'starts with an approximate solution to the differential equation; and in each iteration, the difference between the approximate solution and the exact solution is made smaller' [9].

In general, the error resulting from the exact and approximate solution will have components of different wavelengths: high-frequency components and low-frequency components [9]. Classical iterative methods reduce high-frequency/oscillatory components of error rapidly, but reduce low-frequency/smooth components of error much more slowly [39].

The Multigrid strategy overcomes the weakness of classical iterative solvers by observing that components that appear smooth on fine grid may appear oscillatory when sampled on coarser grid [10]. The high-frequency components of the error are reduced by applying any of the classical iterative methods. The low-frequency components of error are reduced by a coarse-grid correction procedure [12,39].

A Multigrid cycle starts by applying any classical iterative method (Jacobi, Gauss Seidel or SOR) to find an approximate solution for the system. The residual operator is then applied to find the difference between the actual solution and the approximate solution. The result of this operator measures the goodness of the approximation. Since it is easier to solve a problem with less number of unknowns [11,27], a special operator-restriction—for mapping the residual to a coarser grid (less number of unknowns)—is applied for several iterations until the scheme reaches the bottom of the grid hierarchy. Then, the coarse grid solver operator is applied to find the error on the coarsest grid. Afterwards, the interpolation operator is applied to map the coarse grid correction to the next finer grid in an attempt to improve the approximate solution. This procedure is applied until the top grid level is reached giving a solution with residual zero. Finishing with several iterations back to the finest grid gives a so-called V-cycle Multigrid [12,19,25] (Fig. 1).

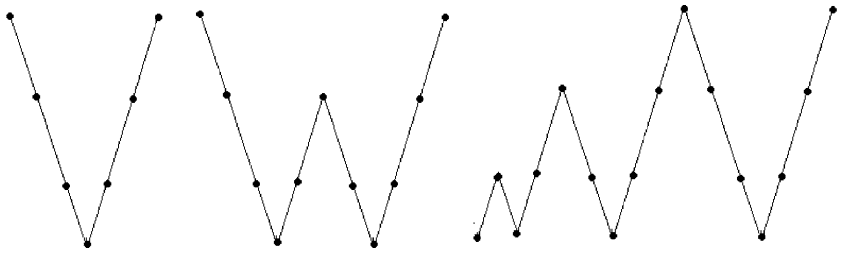

Fig. 1. V-cycle, W-cycle, and full-cycle *MG*.

2.1. Multigrid components

A Multigrid algorithm uses five algorithmic components: smoother/relaxation, residual computation, restriction, coarse grid solver, interpolation.

Relaxation/smoother: This component is responsible for generating an approximate solution by reducing—smoothing/relaxing—the high-frequency error component of the solution imposed upon approximating the solution.

The Gauss–Seidel method can be used in both the pre-smoothing and the post-smoothing steps.

When used in the post-smoothing and pre-smoothing steps in the Multigrid method, the solution is in the form

$$u_{i,j}^{t+1} = \tfrac{1}{4}(t_{i+1,j} + t_{i-1,j} + t_{i,j-1} + t_{i,j+1} + h^2 f_{i,j}), \tag{1}$$

where i and j are the row and column indices of the grid [5].

Residual computation: Let $\hat{u}$ be an approximate solution to the exact solution u, the residual is defined as

$$r_h = f_h - A_h u_h^{(v_1)} = A_h e_h, \tag{2}$$

where $e = u - \hat{u}$ is the error.

The residual must be computed before it can be restricted to the coarser grid.

Restriction: This component is responsible for transporting the residual of the fine grid to the coarser grid using: $r_H = I_h^H r_h$, where $H = 2h$ is the mesh size on a finer grid, r_H is the prolongation of coarser-grid residual to the finer grid, I_h^H is a prolongation, from $h \to H$ (bilinear in our case).

Course grid solver: This operator, often called coarse-grid correction is performed on the coarse grid using

$$u_h^{(v_1+1)} = u^{v_1} + I_H^h e_H. \tag{3}$$

Applying this operator along with the smoothing operator has a substantial effect on the reduction of the residual for all frequencies. However, the coarse grid solver is applied only on the coarsest grid making the cost of this operator negligible to the overall computational cost of the *MG* method [42].

Interpolation/prolongation: Transports the correction obtained on the coarser grid to the fine grid using: $r_h = I_H^h r_h$.

The simplest Multigrid algorithm is based on a two-grid improvement scheme: fine grid and coarse grid. The fine grid, Ω^h, with $N = 2^l + 2$ points and the coarse grid, $\Omega^2 h$, with $N = 2^{l-1} + 2$ points.

In this work, we implement the V-cycle Multigrid to find the solution of a 2D-Poisson equation. Briefly, the V-cycle Multigrid algorithm starts with an initial approximation to the expected solution, goes down to the coarsest grid, and then goes back to the finest grid; as shown in Fig. 2 [12].

2.2. Multigrid solution of Poisson's equation in 2D

The V-cycle Multigrid algorithm is applied to find the solution to a 2D-Poisson equation in the form

$$\frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} = f_{x,y} \tag{4}$$

or in the form $\nabla^2 u = f$ when written in vector notation [32].

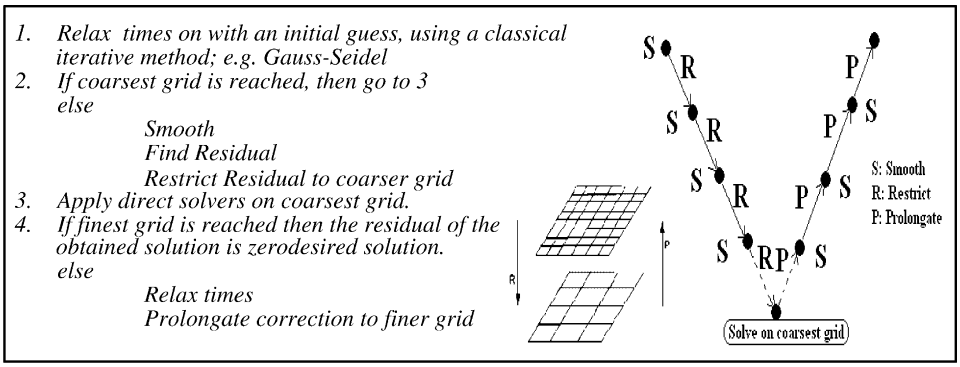

Fig. 2. V-cycle *MG*.

3. Reconfigurable computing

Today, it becomes possible to benefit from the advantages of both software and hardware with the presence of the RC paradigm [15]. Actually, the first idea to fill the gap between the two computing approaches, software and hardware, goes back to the 1960s when Gerald Estrin proposed the concept of *RC* [37].

The basic idea of RC is the 'ability to perform certain computations in hardware to increase the performance, while retaining much of the flexibility of a software solution' [15].

RC systems can be either of fine-grained or of coarse-grained architecture. An *FPGA* is a fine-grained reconfigurable unit while a reconfigurable array processor is a coarse-grained reconfigurable unit. In the fine-grained architecture each bit can be configured; while in the coarse-grained architecture the operations and the interconnection of each processor can be configured. Example of a coarse-grained system is the *MorphoSys* which is intended for accelerating data path applications by combining a general purpose micro-processor and an array of coarse grained reconfigurable cells [2].

The realization of the *RC* paradigm is made possible by the presence of programmable hardware such as large scale complex programmable logic devices (*CPLDs*) and *FPGAs* [34]. RC involves the modification of the logic within the programmable device to suit the application in hand.

3.1. Hardware compilation

There are certain procedures to be followed before implementing a design on an *FPGA*. First, the user should prepare his/her design by using either a schema editor or by using one of the hardware description languages (*HDLs*) such as *VHDL* (very high scale IC HDL) and *Verilog*. With schema editors, the designer draws his/her design by choosing from the variety of available components (multiplexers, adders, resistors, etc.) and connect them by drawing wires between them. A number of companies supply schema editors where the designer can drag and drop symbols into a design, and clearly annotate each component [36]. Schematic design is shown to be simple and easy for relatively small designs. However, the emergence of big and complex designs has substantially decreased the popularity of schematic design while increased the popularity of *HDL* design. Using an *HDL*, the designer has the choice of designing either the structure or the behavior of his/her design. Both *VHDL* and *Verilog* support structural and behavioral descriptions of the design at different levels of abstractions. In structural design, a detailed description of the system's components, sub-components and their interconnects are specified. The system will appear as a collection of gates and interconnects [36]. Though it has a great advantage of having an optimized design, structural presentation becomes hard, as the complexity of the system increases. In behavioral design, the system is considered as a black box with inputs and outputs only, without paying attention to its internal structure [23]. In other words, the system is described in terms of how it behaves rather than in terms of its components and the interconnection between them. Though it requires more effort, structural representation is more advantageous than the behavioral representation in the sense that the designer can specify the information at the gate-level allowing optimal use of the chip area [38]. It is possible to have more than one structural representation for the same behavioral program.

Noting that modern chips are too complex to be designed using the schematic approach, we will choose the *HDL* instead of the schematic approach to describe our designs.

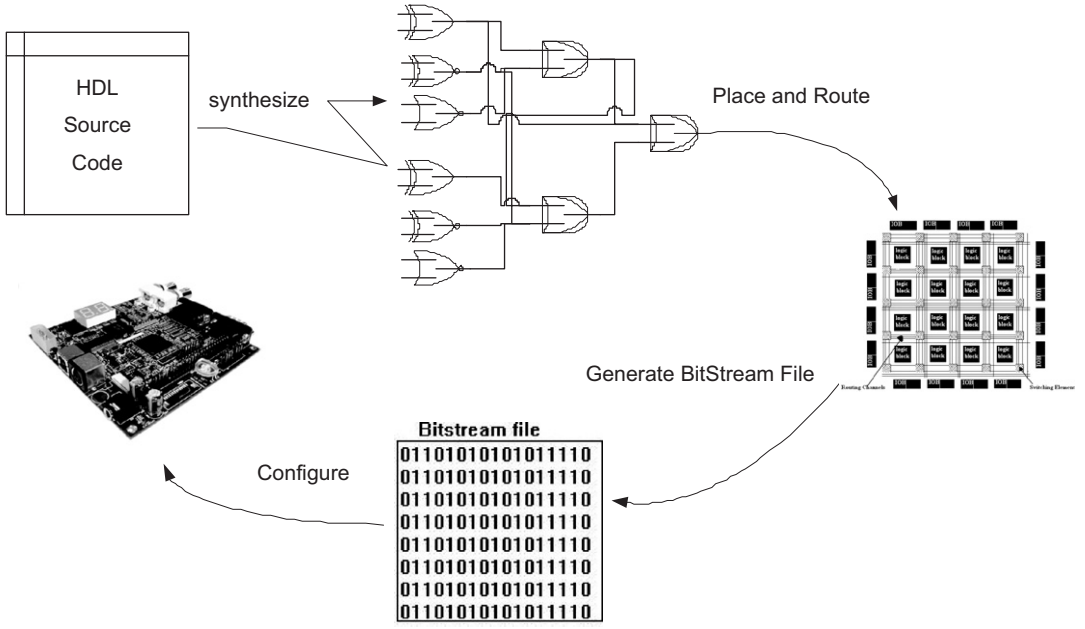

Fig. 3. *FPGA* design flow.

Whether the designer uses a schematic editor or an *HDL*, the design is fed to an electronic design automation (*EDA*) tool to be translated to a netlist. The netlist can then be fitted on the *FPGA* using a process called place and route (PAR), usually completed by the *FPGA* vendors' tools. Then the user has to validate the PAR results by timing analysis, simulation and other verification methodologies. Once the validation process is complete, the binary file generated is used to (re)configure the *FPGA* device. More about this process is found in the coming sections.

Implementing a logic design on an *FPGA* is depicted in the figure below. The above process consumes a remarkable amount of time; this is due to the design that the user should provide using *HDL*, most probably *VHDL* or *Verilog*. The complexity of designing in *HDL*; which have been compared to the equivalent of assembly language; is overcome by raising the abstraction level of the design; this move is achieved by a number of companies such as *Celoxica*, *Cadence* and *Synopsys*. These companies are offering higher level languages with concurrency models to allow faster design cycles for *FPGAs* than using traditional *HDLs*. Examples of higher level languages are *Handel-C*, *SystemC*, and *Superlog* [28,36] (Fig. 3).

3.2. Handel-C language

Handel-C is a high level language for the implementation of algorithms on hardware. It compiles program written in a *C*-like syntax with additional constructs for exploiting parallelism [36]. The *Handel-C* compiler comes packaged with the *Celoxica DK design suite* which also includes functions and memory controller for accessing the external memory on the *FPGA*. A big advantage, compared to other *C* to *FPGA* tools, is that *Handel-C* targets hardware directly, and provides a few hardware optimizing features [13]. In contrast to other *HDLs*, such as *VHDL*, *Handel-C* does not support gate-level optimization. As a result, a *Handel-C* design uses more resources on an *FPGA* than a *VHDL* design and usually takes more time to execute. In the following subsections, we describe *Handel-C* features' that we have used in our design [13,29].

3.2.1. Types and type operator

Almost all *ANSI-C* types are supported in *Handel-C* with the exception of float and double. Yet, floating point arithmetic can still be performed using the floating point library provided by *Celoxica*. *Handel-C* offers additional types for creating hardware components such as memory, ports, buses and wires. *Handel-C* variables can only be initialized if they are global or if declared as static or const. *Handel-C* types are not limited to width since when

Table 1
Effect of using '*par*' construct

	par
$a = 1$;	{
$b = 1$;	$a = 1$
$c = 1$;	$b = 1$
	$c = 1$
	}
No. of clock cycles = 3	No. of clock cycles = 1

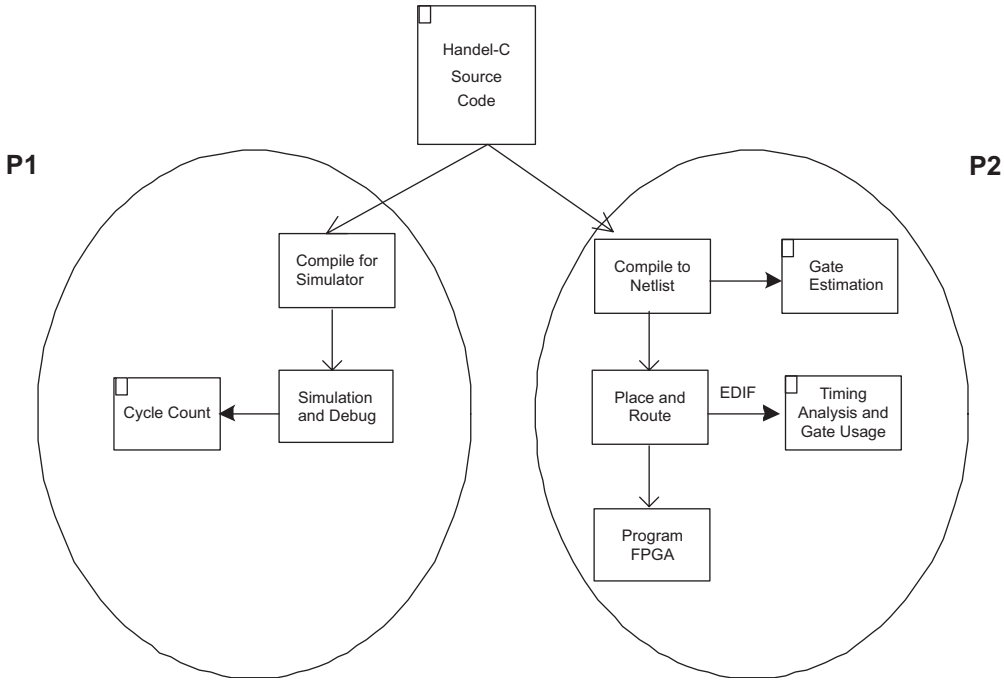

Fig. 4. *Handel-C* targets.

targeting hardware, there is no need to be tied to a certain width. Variables can be of different widths, thus minimizing the hardware usage.

3.2.2. par Statement

The notion of time in *Handel-C* is fundamental. Each assignment happens in exactly one clock cycle, everything else is 'free' [13].

An essential feature in *Handel-C* is the 'par' construct which executes instructions in parallel. Table 1 shows the effect of using '*par*'.

3.2.3. Handel-C targets

Handel-C supports two targets. The first is a simulator that allows development and testing of code without the need to use hardware, P1 in Fig. 4. The second is the synthesis of a netlist for input to PAR tools which are provided by the *FPGA's* vendors, P2 in Fig. 4.

The remaining of this section describes the phases involved in P2, as it is clear from P1 that we can test and debug our design when compiled for simulation.

The flow of the second target involves the following steps:

Compile to netlist: The input to this phase is the source code. A synthesis engine, usually provided by the *FPGA* vendor, translates the original behavioral design into gates and flip flops. The resultant file is called the netlist. Generally, the netlist is in the electronic design interchange format (*EDIF*). An estimate of the logic utilization can be obtained from this phase.

PAR: The input to this phase is the *EDIF* file generated from the previous phase; i.e. after synthesis. All the gates and flip flops in the netlist are physically placed and mapped to the *FPGA* resources. The *FPGA* vendor tool should be used to PAR the design. All design information regarding timing, chip area and resources utilization are generated and controlled for optimization at this phase.

Programming and configuring the FPGA: After synthesis and PAR, a binary file will be ready to be downloaded into the *FPGA* chip [16,31].

3.3. Field programmable gate arrays

An *FPGA* is a programmable digital logic chip. It consists of arrays of logic blocks with an interconnection network of wires. Both the logic blocks and the interconnects can be programmed by the designer so that the *FPGA* can perform whatever logical function is needed. Generally, the internal components of an *FPGA* can communicate with the outside world through the input/output blocks (*IOB*).

The building/logic blocks are the basic elements of an *FPGA*. Each of these blocks is configured to perform a logic function. The interconnection between the logic blocks is provided by the channels of wiring segments of varying lengths [14]. The switching elements are used to determine the choice of active logic modules and their interconnects. The designer can activate or deactivate these elements to suite the requirement of the application in hand [26].

The architecture of an *FPGA* can be classified according to the size and flexibility of the logic cell as well as to the structure of the routing scheme [30]. The four basic architectures are: symmetrical array, row-based, fine grain cellular architecture (sea-of-gates), and complex or hierarchical *PLD*.

3.3.1. Reconfigurability of FPGA

FPGA reconfiguration can be either static, semi-static or dynamic. The dynamic reconfiguration, also known as run-time reconfiguration, is the most powerful form since a dynamically reconfigurable *FPGA* can be programmed/modified on-the-fly while the system is operating [4]. Dynamically reconfigurable *FPGA* may be either partially reconfigured (local run time reconfiguration); where a portion of the *FPGA* continues running while the other portion is being reconfigured or programmed in a full reconfiguration (global run time reconfiguration). In this case, all the system is configured and the intermediate results are stored in an external storage until the configured functions run [33].

4. Hardware implementation of V-cycle *MG*

In recent years, many efforts have been made to develop an efficient, scalable and robust implementation of *MG*. Two major approaches were involved in such development. The first one is implementing the algorithm to run on a general purpose processor; the so-called software implementation. The second approach is implementing it on a dedicated graphics rendering device called graphics processing unit *GPU*. More about *GPUs* can be found in [8,24]. A number of *PDE* solvers have been mapped to *GPUs*, including fast Fourier transform (*FFT*), conjugate gradients (*CG*), and Multigrid. Despite the significant performance that was achieved for these solvers [8,24], our main concern in this paper is to show the superiority of implementing *MG* on *FPGA* over running it on a general purpose processor. Available software packages have been implemented in C, Fortran-77, Java and other languages, where parallelized versions of these packages require inter-processor communication standards such as message passing interface (*MPI*) [6,20,22,35]. Each of these packages attempt to achieve an efficient and a scalable version of the algorithm by compromising between the accuracy of the solution and the speed of realizing the solution.

The V-cycle *MG*, *Jacobi*, and *SOR* algorithms have been designed, implemented and simulated using *Handel-C*. We have preferred the implementation of *Jacobi* and *SOR* over other available iterative solvers since these methods', (*Jacobi* and *SOR*) implementation style is the closest to *MG* and considered to belong to the same gener-

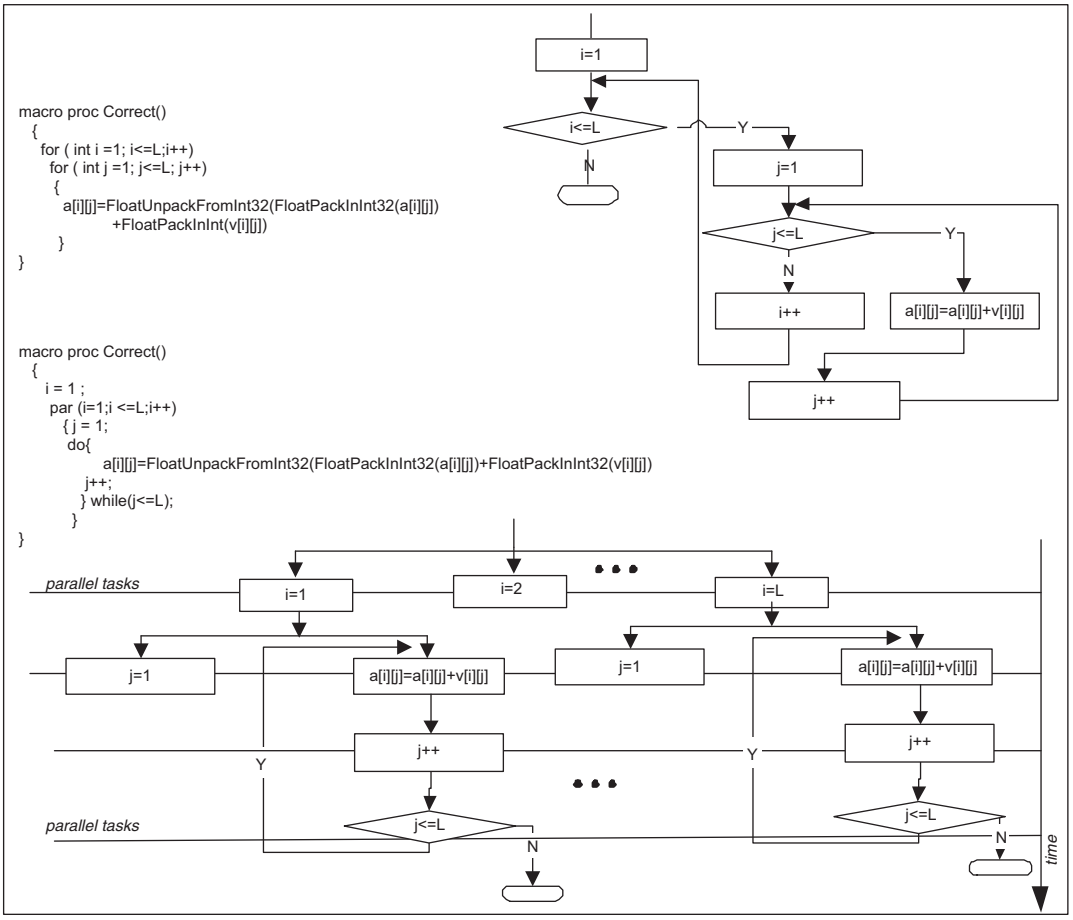

Fig. 5. *MG* correct operator, illustrating the effect of using par construct: (6a), (6b), (6c), and (6d) shows sequential code, flowcharts, parallel code and combined flowchart/concurrent process model, respectively. The dots represent replicated instances in (d). Dashed lines show the parallel tasks.

ation as *MG* [41,7]. Moreover, we wanted to prove that even when implemented in hardware, *MG* outperforms *SOR* and *Jacobi* in certain problems. We have targeted a *Xilinx Virtex II Pro FPGA*, an *Altera Stratix FPGA*, and an *RC10* board from *Celoxica*. The tools provided by the device's vendors were used to synthesize and PAR the design [1,13,40].

Finding the solution to *PDEs* using either of the aforementioned techniques (*MG*, *Jacobi*, *SOR*) requires floating point arithmetic operations which are (1) far more complex and (2) consume more area than fixed point operations. For this reason, *Handel-C* does not support floating point type. Yet, floating point arithmetic can be performed using the Pipelined Floating Point Library provided in the Platform Developer's Kit.

An unexpected crash in the *Handel-C* simulator persists whenever the number of floating point arithmetic operations exceeds four. We were notified, by *Celoxica* people, that a fixed version of the *Handel-C* simulator will be available in a future release of *DK*. The only possible way to avoid the simulator's failure, in its current version, was to convert/Unpack the floating point numbers to integers and perform integer arithmetic on the obtained unpacked numbers. Though it costs more logic to be generated, the integer operations on the unpacked floating point numbers have a minor effect on the total number of the design's clock cycles.

The Multigrid method can be parallelized by parallelizing each of its components; i.e., smoother, coarse grid solver, restriction and prolongation. Each of these components is parallelized by using the *Handel-C* construct '*par*'. This is used whenever it was possible to execute more than one instruction in parallel without affecting the logic of the source code. Figs. 5 and 6 show the two *MG* operators 'restrict residual' and 'correct'. The traditional way of executing

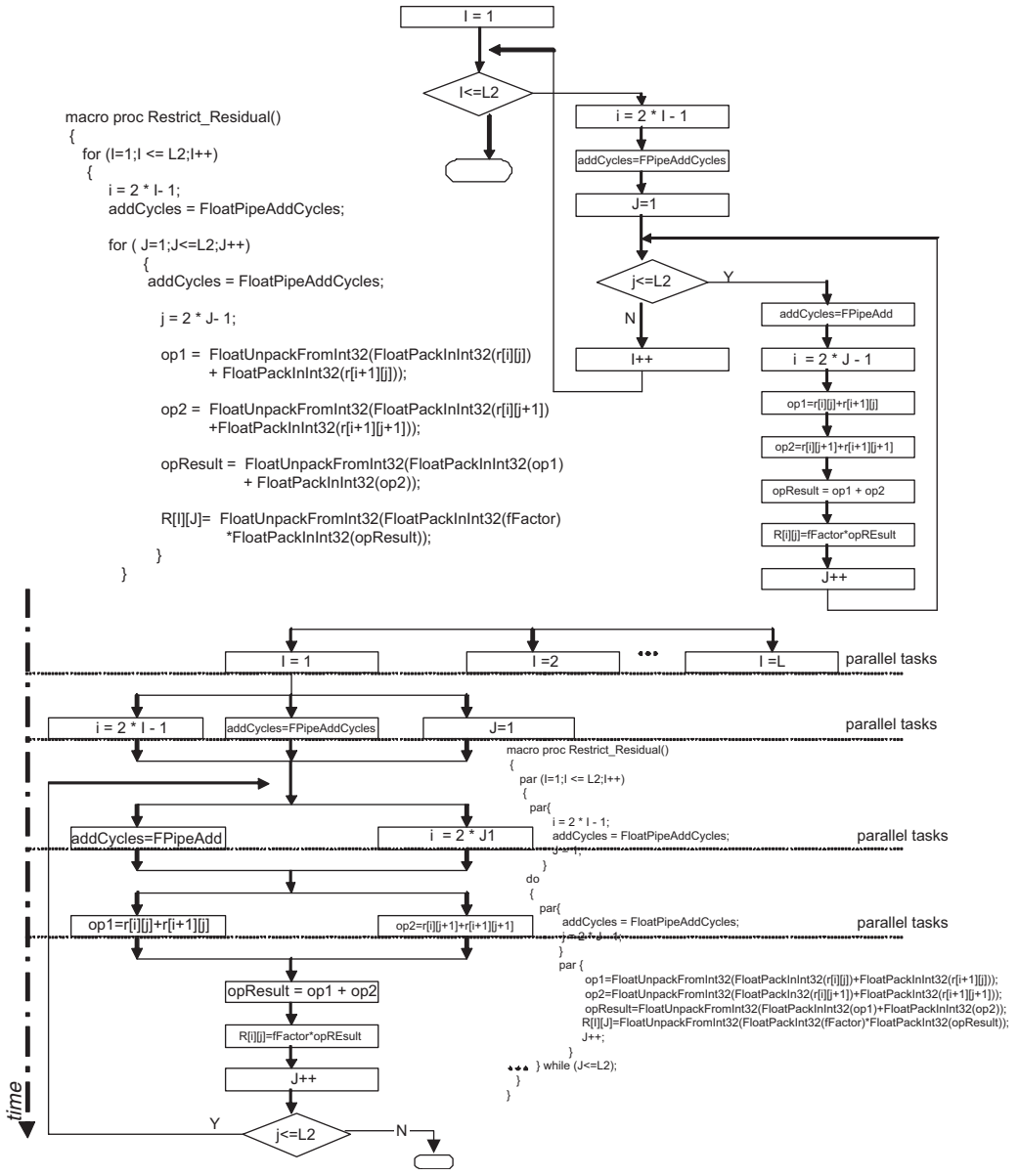

Fig. 6. *MG* restrict residual operator, illustrating the effect of using par construct: (7a), (7b), (7c) and (7d) shows sequential code, flow charts, parallel code and combined flow chart/concurrent process model, respectively. The dots represent replicated instances in (d). Dashed lines show the parallel tasks.

instructions on a *GPP* is shown in the first section, while the second section shows the combined flowchart/concurrent process model of our design. A snapshot of the parallel version of the 'smoother', 'find residual' and 'prolongate' components is shown in Fig. 7. Their implementation style is very similar to that of 'restrict residual' and 'correct' operators.

Both *Jacobi* and *SOR* methods have been parallelized in the same way, i.e., using the par construct whenever possible. The results obtained show: (a) The robustness of *MG* algorithm over *Jacobi* and *SOR* algorithm in both hardware and software implementations. (b) A substantial improvement in the *MG*, *Jacobi*, and *SOR* performance when compared to the traditional way of executing instructions on a *GPP*.

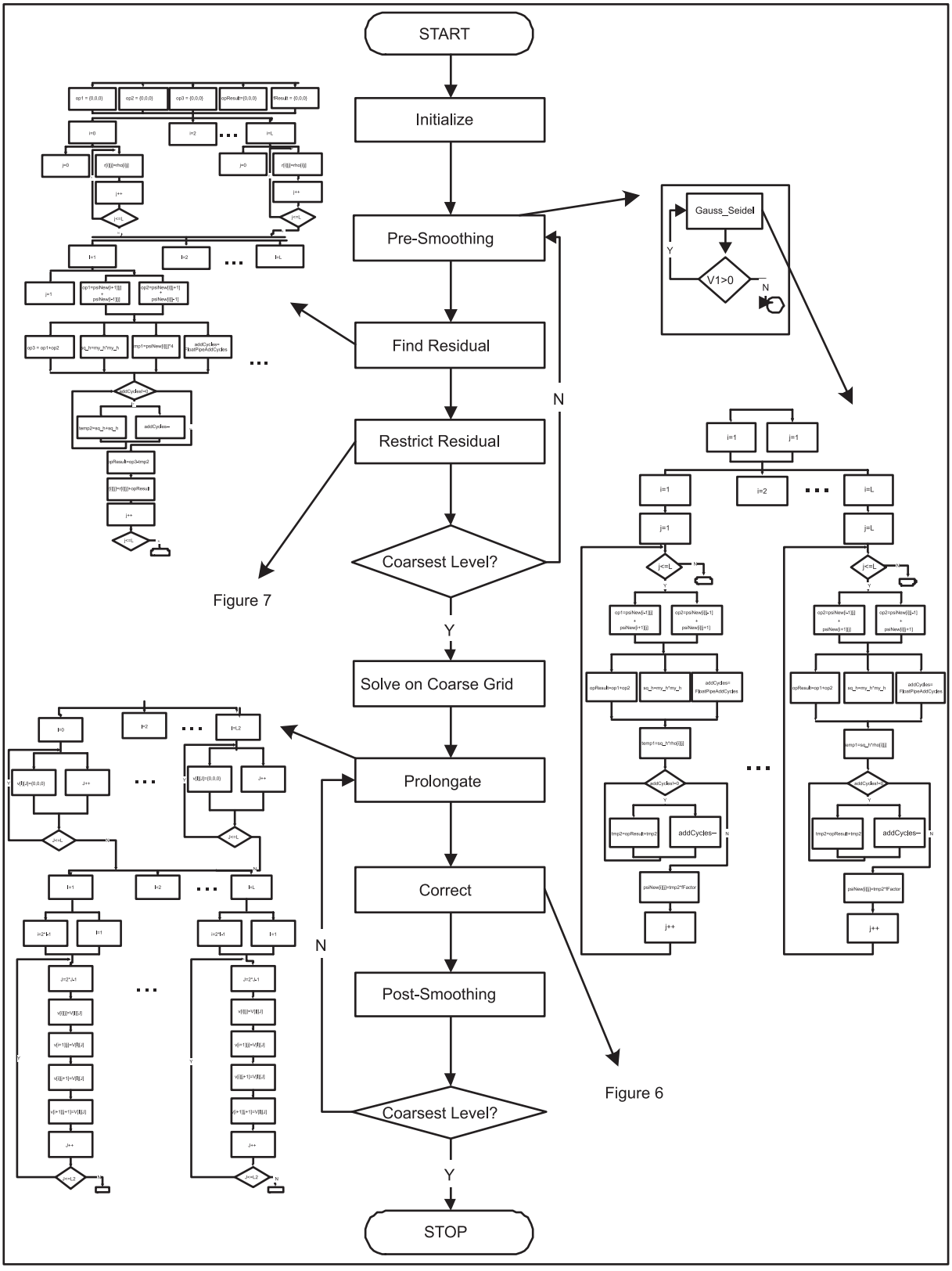

Fig. 7. V-cycle *MG*, iterative version showing each component parallelization. The dots in each of the component's combined flowchart/concurrent process model represent replicated instances.

Table 2
Execution time and max frequency for different problem sizes

Mesh size	Execution time	Fmax
8 × 8	0.000063	159.74
16 × 16	0.00026	153.52
32 × 32	0.00118	136.15
64 × 64	0.00555	115.97
128 × 128	0.031	83.91
256 × 256	0.188	54.60
512 × 512	1.308	31.45
1024 × 1024	9.3	17.60
2048 × 2048	70.97	9.28

5. Experimental results

The *Handel-C* simulator along with the *FPGA* vendor's tools were used to obtain the hardware implementation results. We have chosen C + + to design and code a software version of the algorithms which we compiled using *Microsoft Visual Studio.Net*.

All the test cases were carried out on a Pentium (M) processor 2.0 GHz, 1.99 GB of *RAM*.

The obtained results are based on the following criteria:

- *Speed of convergence*: The time it takes the method of choice to find the solution to the *PDE* in hand. In another word, it is the time needed to execute *MG*, *Jacobi*, or *SOR* algorithm. In our hardware implementation, the speed of convergence is measured using the clock cycles of the design divided by the frequency at which the design operates at. The first parameter is found using the simulator while the second is found using the timing analysis report which is generated using the *FPGA* vendor's tool.
- *Accuracy of the solution*: The convergence of each algorithm is greatly dependent on the accuracy of the solution. The increase in the value of the accuracy results in less computational time and logic utilization. Meaning that, more time and logic are needed for more accurate solution (decrease in the value of the accuracy: high accuracy); while the method terminates faster when a non-accurate solution is needed (increase in the value of the accuracy: low accuracy). Referring to the above criterion (speed of convergence), we can say that increasing the value of the accuracy will speed up the convergence of the algorithm.
- *chip-area*: This performance criterion measures the number of occupied slices on the *FPGA* on which the design is implemented. The number of occupied slices is generated using the *FPGA* vendor's PAR tool.

We compare the timing performance between our hardware implementations of Multigrid, Jacobi, *SOR* and our C + + software version of the same algorithms on *GPPs*.

The following selections were used for all Multigrid performance tests:

- Restriction: full weighting.
- Interpolation: bilinear.
- Number of smoothing steps.
- Smoother used: Gauss–Seidel.
- Accuracy: 0.001 for all *Handel-C* test cases and C + + test cases up to problem size 256 × 256.

As for *SOR* performance tests, the over-relaxation parameters, omega, is set to be 1.5.

The V-cycle *MG* execution time when targeting *Virtex II Pro FPGA*, for different problem sizes, along with the maximum frequency at which each design operates at are shown in Table 2. The execution time is calculated using: No. of clock cycles/Max. Frequency.

Fig. 8 shows the results of comparing the execution time when running our C + + version of the V-cycle Multigrid algorithm and the proposed *Handel-C* version. The Multigrid algorithm is parallelizable in nature, hardware implementation can directly exploit such parallelism to accelerate the algorithm. Such easiness in the parallelization allows

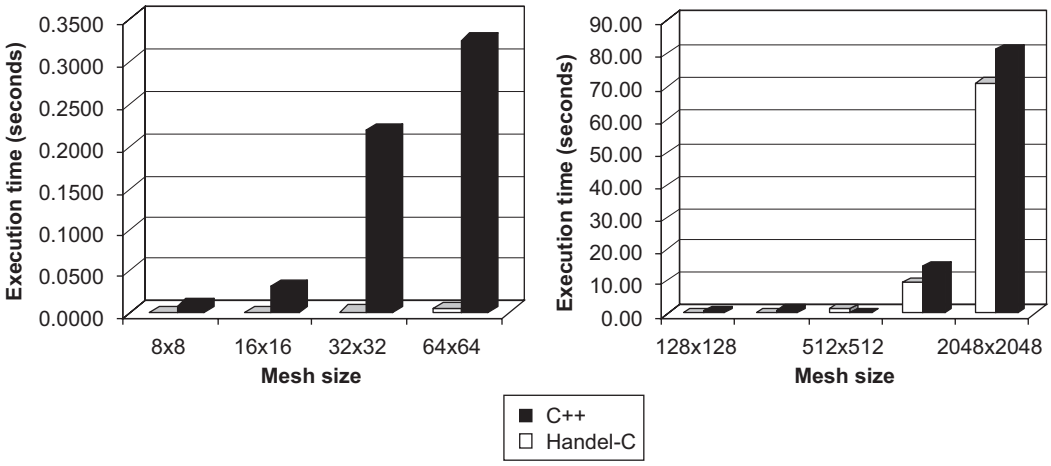

Fig. 8. *MG* execution time results in both versions, *Handel-C* and C + +.

Table 3
Required accuracy of the solution for C + + and *Handel-C* test cases, and the designs speedup

Mesh size	Accuracy		Speedup		
	C + +	*Handel-C*	*MG*	*SOR*	*Jacobi*
8 × 8	0.001	0.001	142.86	1.758	223.81
16 × 16	0.001	0.001	185.59	188	56.21
32 × 32	0.001	0.001	119.23	6.706	5.68
64 × 64	0.001	0.001	58.56	5.69	2.89
128 × 128	0.001	0.001	20.77	1.514	1.41
256 × 256	1	0.001	5.25	1.43	2.29
512 × 512	1.1	0.001	2.92	3.03	2.39
1024 × 1024	1.3	0.001	1.58	2.58	0.75
2048 × 2048	2	0.001	1.14	3.37	1.39

us to reach a problem size of 2048 × 2048 with an accurate solution as needed. Such superiority of the hardware implementation over the software implementation is clear in Fig. 8. Looking back at Figs. 5 and 6, one can expect the number of clock cycles of the hardware version to be less than that of the software version; hence, improving the speed of convergence of the algorithm. However, for a problem size greater than 128 × 128, it becomes difficult to measure the execution time of the software (C + +) version with the same accuracy of 0.001. At that time, our concern was to force our C + + version of *MG* to converge at any price, otherwise it will not converge. This was only possible by sacrificing with the accuracy of the solution; where we had to gradually increase this factor until we reached an accuracy of 2.0 for a problem size of 2048 × 2048, in contrast to an accuracy of 0.001 for a problem size of 8 × 8. On the other hand, *Handel-C* results were independent from the accuracy of the solution. The accuracy was constant all the way from a problem size of 8 × 8 to 2048 × 2048. The degeneration in the speedup is indicated in (b).

As mentioned before, a number of *MG* software packages are available and could have been used in our comparison instead of designing another software version. The point we wanted to raise in this work is that in its simplest form, a hardware implementation of a computationally intensive algorithm, such as (*MG*), can outperform a software implementation of the same form. The hardware version could deal with a problem size of 2048 × 2048, while the software version failed at 256 × 256. It is important to note that these results are based on our implementations and findings and do not by any way imply that the maximum size of the problem to be solved using any of the available software versions is necessary 256 × 256.

In Table 3 we draw a comparison between the accuracy of the solution for each of the C + + and *Handel-C* test cases. The speedup of the design is calculated as the ratio of execution time (C + +)/execution time (*Handel-C*). As

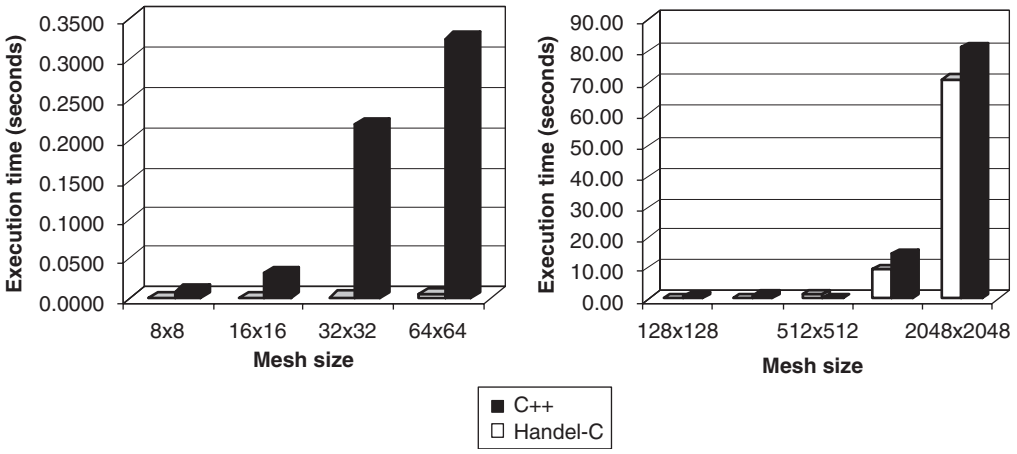

Fig. 9. Jacobi execution time results in both versions, *Handel-C* and C + +.

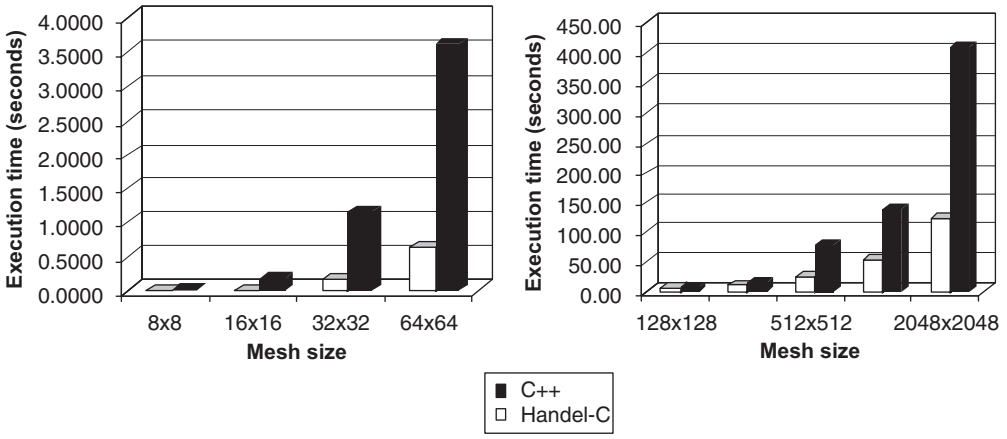

Fig. 10. *SOR* execution time results in both versions, *Handel-C* and C + +.

the table shows, the speedup of the designs degenerates for problem sizes between 8 × 8 and 128 × 128 since the same value of accuracy is kept for the C + + and *Handel-C* versions where the value of the numerator (C + + execution time) and the denominator (*Handel-C* execution time) are not increasing in the same order. As for problem size greater than 128 × 128, the increase in the value of the accuracy (low accuracy) in the C + + version only has decreased the expected speedup.

The superiority of the hardware implementation over the software implementation for *Jacobi* and *SOR* is shown in Figs. 9 and 10. This observation demonstrates the ability of realizing an accelerated version of the algorithm when implemented on hardware (Fig. 11).

Tables 4–6 show, respectively, the *Virtex II Pro* (*2vp7ff672-7*), *Spartan*3*L*(*3s1500lfg320-4*) and *Altera Stratix* (*ep1s10f484c5*) *FPGA* synthesis results for different problem sizes in *MG*, *SOR*, and *Jacobi*. When targeting *Xilinx Virtex II Pro FPGA*, the largest possible problem size that we could achieve was 2048 × 2048, where 99% of the slices were utilized. Meanwhile, the largest possible problem size was 512 × 512 when targeting *Spartan*3*L FPGA*. As for the third targeted *FPGA* which has a different architecture than that of the first two *FPGAs*, we report different metrics for different problem sizes up to 2048 × 2048. A problem size of 4096 × 4096 was successfully synthesized and placed and routed for this *FPGA*.

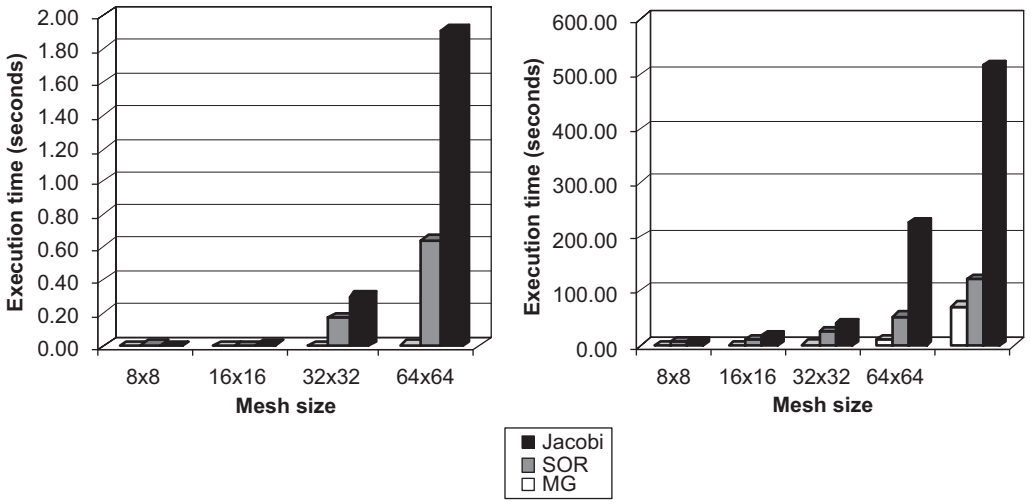

Fig. 11. Robustness of *MG* over *Jacobi* and *SOR*.

Table 4
Xilinx Virtex II Pro synthesis results using *Xilinx ISE*

Mesh size	Number of occupied slices			Total equivalent gate count		
	MG	*SOR*	*Jacobi*	*MG*	*SOR*	*Jacobi*
8 × 8	264	128	146	5990	2918	3229
16 × 16	295	136	159	6497	3033	3397
32 × 32	415	219	299	9321	4807	5090
64 × 64	536	265	380	12 376	5978	7849
128 × 128	789	315	499	18 107	7125	11 864
256 × 256	1247	610	839	29 244	14 538	17 864
512 × 512	2125	1098	1286	51 115	23 012	23 649
1024 × 1024	3875	1601	1890	94 484	31 848	31 327
2048 × 2048	4926	2289	3198	180 879	53 476	35 839

Table 5
Spartan3L synthesis results using *Xilinx ISE*

Mesh size	Number of occupied slices			Total equivalent gate count		
	MG	*SOR*	*Jacobi*	*MG*	*SOR*	*Jacobi*
8 × 8	687	302	416	355 687	279 010	356 109
16 × 16	717	499	599	356 163	281 001	357 631
32 × 32	769	589	7326	357 224	282 997	359 989
64 × 64	832	745	9010	358 921	284 000	342 768
128 × 128	1049	877	1198	361 956	285 872	389 999
256 × 256	1507	1201	1665	367 673	297 134	397 987
512 × 512	3187	2010	2810	375 293	299 858	498 030

Table 6
Altera Stratix synthesis results using *Quartus II*

Mesh size	Total logic elements			LE usage by no. of LUT inputs			Total registers		
	MG	*SOR*	*Jacobi*	*MG*	*SOR*	*Jacobi*	*MG*	*SOR*	*Jacobi*
8×8	725	519	610	402	250	354	228	120	189
16×16	818	601	709	554	310	401	265	155	232
32×32	925	810	880	625	501	556	301	199	300
64×64	1068	999	1001	709	637	681	360	280	385
128×128	1307	1274	1286	841	720	801	467	347	390
256×256	1739	1510	1590	1070	890	950	670	498	476
512×512	2653	2286	2589	1357	1087	1101	816	501	560
1024×1024	3491	2901	3342	1809	1450	1499	1002	569	689
2048×2048	4501	3286	3927	2201	1798	1941	482	640	819

6. Conclusions and future work

In this paper, we have presented a hardware implementation of the V-cycle Multigrid method for solving the Poisson equation in two dimensions. *Handel-C* hardware compiler is used to code and implement our designs (*MG*, *Jacobi*, and *SOR*) and map them onto high-performance *FPGAs*, such as, *Virtex II Pro*, *Altera Stratix*, and *Spartan3L* which is embedded in the *RC*10 *FPGA*-based system from *Celoxica*. The implementation performance is analyzed using the *FPGAs* vendors' proprietary software. Moreover, we compare our implementation results with available software version results running on general purpose processors and written in C + +. The obtained results have demonstrated that (1) *MG* algorithm outperforms the *Jacobi* and the *SOR* algorithms, on both hardware and software and (2) *MG* on hardware outperforms *MG* on *GPP*, where a speedup of 142.86 was achieved for a problem size of 8×8, whereas a speedup of 1.14 was achieved for 2048×2048. This degeneration of the speedup is due to the increase of the value of the required accuracy of the solution. Possible future directions include realizing a pipelined version of the algorithm, moving to a lower-level *HDL* such as *VHDL*, mapping the algorithm into a coarse grain reconfigurable systems (e.g., *MorphoSys*) [17], and benefiting from the advantages of formal modeling [18]. We can also extend the benefit of *MG* by implementing the W-cycle algorithm and the Algebraic *MG*.

References

[1] Altera ⟨http://www.altera.com⟩.
[2] N. Bagherzade, F. Kurdahi, H. Singh, G. Lu, M. Lee, E. Filho, MorphoSys: design and implementation of the MorphoSys reconfigurable computing processor, J. VLSI Signal Process. —Systems Signal Image Video Technol. 24 (2–3) (2000) 147–164.
[3] D. Bailey, J.M. Borwein, Future Prospects for Computer-assisted Mathematics, Canadian Mathematical Society Notes, vol. 37, no. 8, 2005, pp. 2–6.
[4] M. Barr, A Reconfigurable Computing Primer, Freeman Inc., Miller, 1998.
[5] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. Van der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, second ed., SIAM, Philadelphia, PA, 1994.
[6] S. Barsky, BARSKY, 2003 ⟨www.mgnet.org/mgnet-codes-barsky.html⟩.
[7] D. Bertsekas, J. Tsitsiklis, Some aspects of parallel and distributed iterative algorithms: a survey, Automatica 27 (1) (1991) 3–21.
[8] J. Bolz, I. Farmer, E. Grinspun, P. Shroder, Sparse matrix solvers on the GPU: conjugate gradients and multigrid, ACM Trans. Graphics 22 (3) (2003).
[9] A. Borzi, Introduction to multigrid methods, Institut fur Mthematik und Wissenschaftliches Rechnen, 1999 ⟨www.kfunigraz.ac.at/imawww/borzi/mgintro.pdf.⟩
[10] J. Bramble, Multigrid Methods, Pitman Research Notes in Mathematics Series, vol. 294, Longman Scientific, New York, 1993.
[11] A. Brandt, Multi-level adaptive solutions to boundary value problems, Math. Comp. 1 (31) (1977) 333–390.
[12] W.L. Briggs, V.E. Henson, S.F. Mccormick, A Multigrid Tutorial, second ed., SIAM, Philadelphia, PA, 2000.
[13] Celoxica ⟨www.celoxica.com⟩.

[14] M. Chrzanowska-Jeske, Architecture and synthesis issues FPGAs, in: Proceedings of NORTHCON'93 Electrical and Electronics Convention, 1993, pp. 102–105.
[15] K. Compton, S. Hauck, Reconfigurable computing: a survey of systems and software, ACM Comput. Surveys 34 (2002) 171–210.
[16] J. Cong, FPGA Synthesis and Reconfigurable Computing, University of California, Los Angeles, 1997 ⟨www.ucop.edu/research/micro/96$_9$7/96$_1$76.pdf⟩.
[17] I. Damaj, H. Diab, Performance evaluation of linear algebraic functions using reconfigurable computing, Internat. J. Super Comput. 24 (1) (2003) 91–107.
[18] I. Damaj, J. Hawkins, A. Abdallah, Mapping high-level algorithms onto massively parallel reconfigurable hardware, in: IEEE International Conference of Computer Systems and Applications, 2003, pp. 14–22.
[19] K. Datta, S. Merchant, Multigrid methods for titanium heart simulation. Retrieved January 2006, from University of California, Berkeley, Department of Computer Science web site: ⟨http://www.cs.berkeley.edu/∼kdatta/classes/cs267.doc⟩.
[20] C. Douglas, MadPack: a family of abstract multigrid or multilevel solvers, Comput. Appl. Math. 14 (1995) 3–20.
[21] A.J. Elbirt, C. Paar, An FPGA implementation and performance evaluation of the Serpent block Cipher, in: ACM/SIGDA International Symposium on FPGAs, 2000, pp. 33–40.
[22] R. Enzler, The current status of reconfigurable computing, Technical Report, Electronics Laboratory, Swiss Federal Institute of Technology (ETH) Zurich, 1999.
[23] FPGA4Fun, FPGA design entry, 2005 ⟨www.fpga4fun.com/DesignEntry.html⟩.
[24] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, G. Humphreys, A multigrid solver for boundary value problems using programmable graphics hardware, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2003, pp. 102–111.
[25] W. Hackbusch, Multigrid Methods and Application, Springer, Berlin, 1985.
[26] Y. Khalilollahi, Switching elements, the key to FPGA architecture, in: WESCON Conference Record, 1994, pp. 682–687.
[27] MGNet Homepage: ⟨http://www.mgnet.org/⟩.
[28] D. Pellerin, S. Thibault, Practical FPGA Programming in C, in: Prentice-Hall Professional Technical Reference, Prentice-Hall, Upper Saddle River, NJ, 2005.
[29] C. Peter, Overview: Hardware Compilation and the Handel-C language, Oxford University Computing Laboratory, 2002 ⟨http://web.comlab.ox.uk/oucl/work/christian.peter/overview$_h$andelc.html⟩.
[30] A.K. Sharma, Programmable Logic Handbook PLDs, CPLDs and FPGAs, McGraw Hill, New York, 1998.
[31] J. Shewel, A hardware/software co-design system using configurable computing technology, 1998 ⟨http://ipdps.cc.gatech.edu/1998/it/schewel.pdf⟩.
[32] M.L. Smedinghoff, Solving the Poisson equation with multigrid, 2005 ⟨http://cd-amr.fnal.gov/aas/poisson.pdf⟩.
[33] Terminology. Bournemouth University, School of Design, Engineering and Computing ⟨http://dec.bournemouth.ac.uk/drhw$_l$ib/terminology.html⟩.
[34] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, P.Y.K. Cheung, Reconfigurable computing: architectures and design methods, IEE Proceedings—Computers and Digital Techniques 152 (2) (2005) 193–197.
[35] T. Torsti, M. Heiskanen, M. Puska, R. Nieminen, MIKA: multigrid-based program package for electronic structure calculations, Internat. J. Quantum Chem. 91 (2) (2003) 171–176.
[36] J. Turely, How Chips are Designed, in: Prentice-Hall Professional Technical Reference, Prentice-Hall, Upper Saddle River, NJ, 2003 ⟨www.phptr.com/articles/⟩.
[37] F. Vahid, T. Givargis, Embedded Systems Design: A Unified Hardware/Software Introduction, Wiley, New York, 2002.
[38] S.K. Valentina, Designing a digital system with VHDL, Academic Open Internet J. 11 (2004).
[39] P. Wesseling, An Introduction to Multigrid Methods, Wiley, New York, 1992.
[40] Xilinx ⟨www.xilinx.com⟩.
[41] D. Young, A historical review of iterative methods. Report CNA-206, Center for Numerical Analysis, University of Texas at Austin, 1987.
[42] J. Yström, Multigrid methods improve solvers for both speed and memory, 2005 ⟨www.comsol.fr/femlab2005/sample.doc⟩.