High-Level Languages and Floating-Point Arithmetic for FPGA-Based CFD Simulations

Diego Sanchez-Roman, Gustavo Sutter, Sergio Lopez-Buedo, Ivan Gonzalez, Francisco J. Gomez-Arribas, and Javier Aracil
Universidad Autonoma de Madrid

Francisco Palacios
Stanford University

Editor's note:
Computational fluid dynamics is a classical problem in high-performance computing. In order to make use of an existing code base in this field, the use of high-level design tools is an imperative. The authors explore the use of the Impulse C design tools for a Navier-Stokes implementation.
—George A. Constantinides (Imperial College London) and Nicola Nicolici (McMaster University)

Computational Fluid Dynamics (CFD) plays a key role in the design and optimization of many industrial applications. In the case of aeronautics, aircraft design has been traditionally based on costly and time-consuming wind tunnel tests. Computer-based flow simulations would enable much faster and less expensive tests, significantly reducing design costs and allowing for the exploration of new airfoil geometries. More importantly, CFD would also enable shape optimization, thus facilitating the development of safer, less polluting and less fuel-consuming aircrafts. Unfortunately, the huge computational costs of CFD prevent it from being a valid tool for the entire design process. CFD is currently used only at some design steps, and wind tunnel tests are still essential.

These huge costs come from the Navier-Stokes equations that govern the air flow motion. These Navier-Stokes equations derive from the physical laws of mass, momentum, and energy conservation, and they cannot be solved analytically except in concrete cases, so their solutions must be approximated numerically. The drawback with CFD simulation in aeronautical design is that, in many situations, flows develop two physical phenomena: shock waves and turbulence. Such cases require using a fine discretization of the space to obtain accurate results, so the time required to compute the solutions becomes prohibitive, even in the best high-performance computing (HPC) clusters. Because of this, researchers have expended considerable effort to try to accelerate the execution of these algorithms. Developed algorithms include computing parallelization,[1] GPU computing,[2-4] and FPGA solutions.[5,6]

FPGA-based high-performance reconfigurable computing (HPRC) is a promising technology for application acceleration because it creates a synergy between system-level parallelism and lower-level hardware parallelism. Unfortunately, reconfigurable solutions typically require skilled engineers to write hardware description language (HDL) code, and development time and testing usually far exceed that in software solutions. Fortunately, this drawback is being reduced, thanks to the advent of hardware compilers from high-level languages (HLLs), typically C or C dialects.[7,8] Although HLLs reduce development time, manual optimizations are still needed to improve the implementation, and this limits the reduction in design efforts and costs. Additionally, newer FPGA families provide better support for floating-point arithmetic, so it's no longer necessary to go through the painful step of porting the algorithm to fixed-point arithmetic. Finally, the advent of a new generation of acceleration-oriented FPGA platforms, such as in-socket accelerators, has made HPRC far more efficient and simpler to use.

In this article, we show how to employ these new methodologies and tools to significantly improve the performance of complex applications such as aeronautical CFD simulations, while reducing the energy required to perform those computations. We have obtained a 22× speedup and one order of magnitude in energy savings for a 2D Navier-Stokes solver using an approach based on the XtremeData XD2000i In-Socket Accelerator,[9] along with single-precision floating-point arithmetic and Impulse Accelerated Technologies' Impulse C software (http://www.impulseaccelerated.com), which generates VHDL or Verilog from standard ANSI-C code.

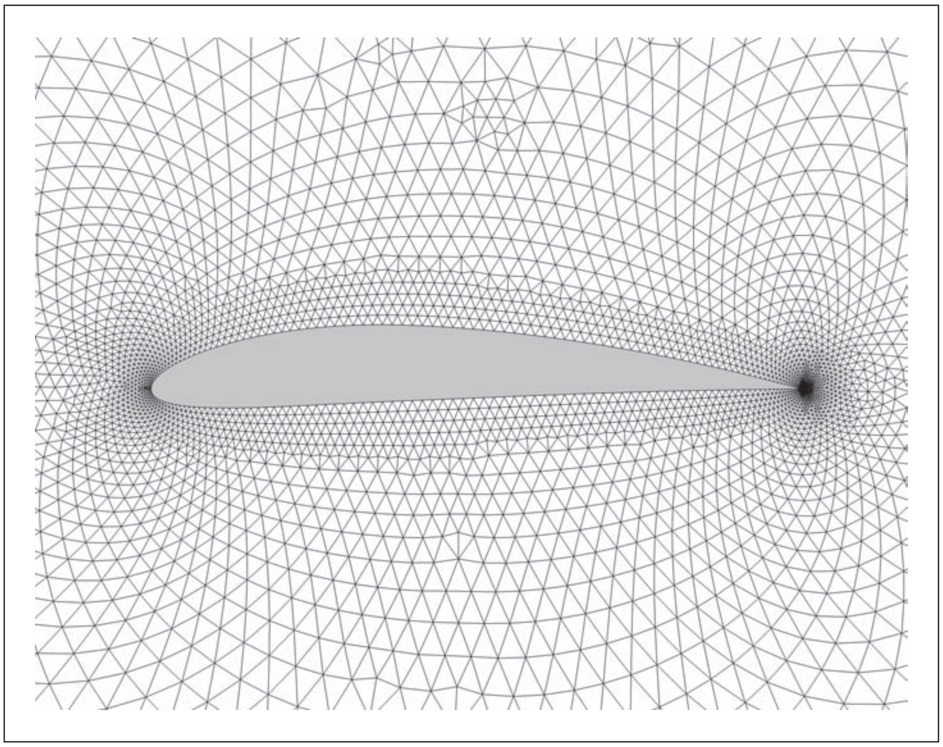
Figure 1. 2D discretization of a National Advisory Committee for Aeronautics (NACA) 4412 airfoil. The space is discretized in finite volumes shown as 2D triangles, where the laws of physics are imposed.

Navier-Stokes solver

We've used an in-house Navier-Stokes solver written in C++ using single floating-point arithmetic, which implements a vertex-centered finite-volume method (FVM) applying the MUSCL (Monotone Upstream-Centered Schemes for Conservation Laws) scheme. The program's input is an unstructured mesh, in which the space is discretized in finite volumes. These finite volumes are triangles or rectangles in the 2D case, and hexahedrons, pyramids, tetrahedrons, or wedges in the 3D case. Figure 1 represents a 2D discretization of a National Advisory Committee for Aeronautics (NACA) 4412 airfoil consisting of triangles. The density, momentum, and energy are associated with each node (we will call them conservative values since they follow conservation laws). These magnitudes are computed in successive time integrations until a convergence criterion is reached.

In the first preprocessing stage, the program reads the mesh and computes the control volumes. It also determines the edge connectivity, calculates the normal vectors (to edges), and initializes the conservative values for each node. After this preprocessing stage, the actual integration loop begins. Figure 2 shows the algorithm flow. The arrows between the boxes represent the dependency between the subroutines, and the numbering reflects the order in the sequential execution of the software solver. The new conservative values for each node are updated in the time integration routine, as a function of the residuals computed in the space integration and the time step. Because of the numerical scheme implemented, one node uses data from two neighborhood layers to update its new conservative values.

Hardware platform

The development system is a workstation with a dual Xeon motherboard populated with one Intel Xeon L5408 quad-core processor and one XtremeData XD2000i In-Socket Accelerator. The XD2000i, installed in the second processor socket, uses the

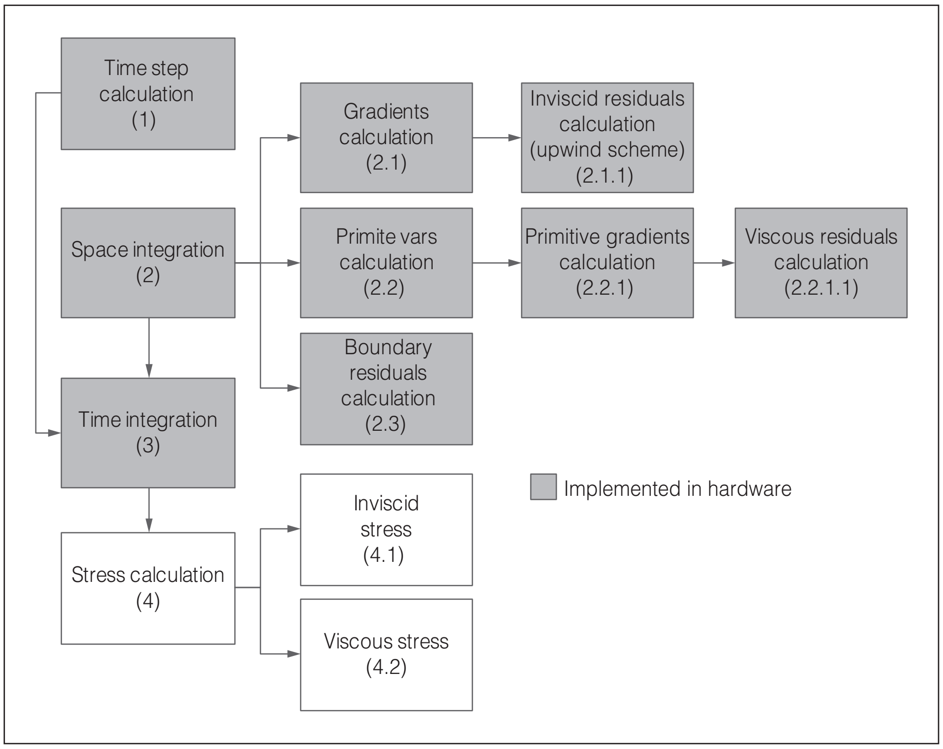
Time step calculation (1)
Space integration (2)
Time integration (3)
Stress calculation (4)
Gradients calculation (2.1)
Primite vars calculation (2.2)
Boundary residuals calculation (2.3)
Inviscid residuals calculation (upwind scheme) (2.1.1)
Primitive gradients calculation (2.2.1)
Viscous residuals calculation (2.2.1.1)
Inviscid stress (4.1)
Viscous stress (4.2)
Implemented in hardware

Figure 2. Algorithm flow for the Navier-Stokes solver that we used.

motherboard's existing CPU infrastructure to create a full-featured environment for FPGA coprocessing. The high-bandwidth, low-latency front-side bus (FSB) link between the coprocessor module and the Intel processor enables tightly coupled FPGA acceleration of x86 applications—previously impossible with legacy PCI-bus–based solutions.[9]

The XD2000i In-Socket Accelerator features three Altera Stratix III FPGAs (http://www.altera.com). One of these FPGAs (Stratix III SL150) serves as a bridge to the FSB, whereas the other two (Stratix III SE260 with 255,000 logic elements, 768 embedded multipliers, and 15 Mbytes of internal memory) are available to implement the user logic. These two application FPGAs are connected through two unidirectional 64-bit buses at 400 megatransfers per second (MT/s). In addition, the XD2000i module includes two QDRII+ SRAM banks, one for each user FPGA.[9] The accelerator system typically works at 100 MHz.

Hardware development methodology using HLLs

Scientific-computing algorithms are mainly written with floating-point arithmetic in C, C++, or Fortran. However, the traditional approach for FPGA design is based on fixed-point arithmetic and HDLs such as VHDL or Verilog. Using this traditional approach, algorithm acceleration in an HPRC system requires a huge effort: analyze the dynamic range of variables to transform floating- to fixed-point arithmetic; manually schedule arithmetic operations; create finite-state automata to control the execution of operations, and so forth. Additionally, validation and debug is especially difficult, making this methodology suitable only for very stable codes. This is a serious drawback for algorithms that are continually evolving because of new scientific knowledge, as with aeronautical CFD applications. Therefore, we use an alternative methodology based on Impulse C and floating-point arithmetic.

Synthesis framework creation

The first step was to create a C synthesis framework optimized for the accelerator platform used. We chose Impulse C because it is the only tool that supports our XD2000i module, since Mitrion-C has been discontinued.[8] To the best of our knowledge, there are only two other HLLs that support floating-point arithmetic out of the box—Xilinx AutoESL and ROCCC (Riverside Optimizing Compiler for Configurable Computing)—but neither of these HLLs provides support for acceleration modules. Another advantage of Impulse C is that it lets users easily change the cores being used for arithmetic operations.

Actually, we found that the floating-point cores originally provided by Impulse C and Altera are deeply pipelined, featuring a high clock frequency but at the cost of increased latency and area. However, the XD2000i module is aimed at 100-MHz designs, so it made no sense to use such deeply pipelined cores. Therefore, we developed our own single-floating-point library based on *IEEE Std.* 754-2008 (*IEEE Standard for Floating Point Arithmetic*) binary32, targeting this clock frequency at the lowest

Table 1. Latency and area of floating-point operations.

	Impulse C			Altera			Our floating-point library		
Operation	**Latency (cycles)**	**Look-up tables**	**Registers**	**Latency (cycles)**	**Look-up tables**	**Registers**	**Latency (cycles)**	**Look-up tables**	**Registers**
Add or subtract	11	554	632	7	548	308	3	435	139
Multiply	11	148	283	5	155	136	2	107	72
Divide	15	2,127	710	6	1,482	758	6	1,320	392
Divide (DSP)*	15	244	491	6	197	260	NA	NA	NA
Square root	NA	NA	NA	16	473	521	5	401	198

* Multipliers in digital-signal processor (DSP) blocks may be used to perform division. Impulse C division uses 14 DSP blocks, whereas Altera division may use either 16 DSP blocks or none. Our division does not use multipliers; therefore, there is no version with DSP blocks.

possible latency. Table 1 shows the area and latency comparison between Impulse C, Altera, and our floating-point library, where FPGA resources are measured in terms of look-up tables (LUTs) and registers.

Hardware-software partitioning and solver code adaptation

The second step was to perform hardware-software partitioning and adapt the original solver code to Impulse C. We began by profiling the code to find the computational bottlenecks. This profiling, along with an analysis of the data flow, let us decide which portions of the code to execute in the FPGA. Most HDL compilers, including Impulse C, are based on the process-stream model of computation, derived from the concurrent sequential processes (CSP) paradigm. In this computation model, data flows from one process to others through streams. Synchronization is automatically resolved because processes run only when there is data available in their input streams. In a well-modularized C or C++ program, the rule of thumb is to map each C or C++ function to one process, although it's always desirable to examine the code to find parallel tasks within a function, which can be separated into concurrent processes.

The major difficulties in adapting the original C code for high-level synthesis are in data access and storage. The HDL compiler might not (as Impulse C does not) support object-oriented paradigms or complex data structures, such as arrays inside C structures, pointers, or dynamic-memory allocation. More decisively, efficient data access is critical for an optimum hardware implementation. Therefore, we advocate using an architectural pattern that isolates computation and data storage into two different types of processes. This solution aims for the best code maintainability and easier optimizations.

Loop unrolling and pipelining

At the end of the second step, a hardware-software solution was running, but it served only for validation purposes, because performance in such solutions is typically very poor. Acceleration can be achieved only when loop parallelism is exploited via loop unrolling and pipelining—the third step in our methodology.

In Impulse C, this is achieved through pragmas placed at the beginning of the loops. However, there are currently some restrictions: Neither nested unrolled loops inside a pipeline loop nor nested pipelined loops are allowed. Moreover, loops must be fully unrolled, and this is only possible when the number of iterations is known at compilation time. Other tools, such as Mentor Graphics' Catapult C,[8] let users unroll loops in sets of *k* iterations, thus providing a maximum speedup factor of *k*. Moreover, Impulse C automatically tries to achieve pipelines with an optimal rate of 1 output result per clock cycle. If input data cannot be provided at such a rate because of data dependencies or access contention, pipeline stalls are automatically added. In contrast, Catapult C offers less aggressive pipelines with a custom issue rate and automatic component sharing.

Architectural optimizations

The last step in our methodology is to apply architectural optimizations to improve area and performance. Impulse C provides the Stage Master Explorer tool to report pipeline throughput and

Table 2. Number of arithmetic operations for the routines given in Figure 2.

	Routines							
Operation	**2.1 and 2.2**	**2.2.1**	**2.3**	**2.1.1**	**2.2.1.1**	**1**	**3**	**Total**
Add or subtract	27	2	6	106	26	13	4	184
Multiply	34	19	17	122	20	16	4	232
Divide	0	3	1	11	3	4	1	23
Square root	0	2	3	7	0	1	0	13
Multiply by 2	0	0	0	1	2	1	0	4
Divide by 2	7	0	3	7	10	2	0	29

data stalls, greatly simplifying the performance optimizations. In our experience, a significant fraction of data stalls originate because FPGA memory elements have very limited parallelism: just two independent read/write ports. Because each process requires at least one port, global array sharing among processes is discouraged. However, the use of array flattening can increase parallelism in data access; that is, we split multidimensional arrays into several 1D arrays, which can be accessed concurrently to improve data throughput. The corollary is that improving performance requires optimizing data access, which in turn requires separating computation and data storage into different processes.

Simple code tweaks were effective for reducing area. First, we transformed arithmetic expressions by extracting common factors. Then, we stored some common calculations that were redundantly computed in auxiliary variables. Another trivial transformation was the suppression of as many divisions as possible, because this arithmetic operator is excessively hungry in terms of latency and area (see Table 1). When more than one division with the same denominator is performed, it's usually preferable to compute the inverse and multiply. Similarly, we developed arithmetic cores for multiplying and dividing by 2, which are significantly simpler operations than regular multiplication and division.

In general, this methodology is valid for most scientific computational algorithms. However, it is best suited for algorithms with an irregular data access pattern and parallelizable loops. Additionally, when using floating-point arithmetic, low-latency operators and arithmetic transformations always help to shrink the circuit by reducing register utilization.

Hardware implementation

We selected the routines to be implemented in hardware (see Figure 2) by profiling the original C++ solver. The changes required from the C++ code were straightforward and related chiefly to the process-stream programming and the methodology described earlier.

However, because the total number of floating-point arithmetic operations was huge (see Table 2), we had to use the area reduction tweaks described in the previous section. We also carefully analyzed the instruction scheduling, and we detected that the reuse of some variables was limiting the code parallelization, thus increasing pipeline latency and register use. Furthermore, we redefined the array sizes to be a power of 2 when possible, so that effective addresses were computed by shifting and concatenation rather than multiplication and addition. Even when we used these optimizations, however, we were left with a design that was too big for one FPGA (see Table 3), so we used both devices available in the XD2000i.

To optimize performance, the main issue concerning us was to prevent pipeline stalling. First, data waits were eliminated by array splitting, as explained earlier. Second, dependency between loop iterations appeared in the form of accumulations over a given memory. We eliminated this loop dependency by developing an out-of-order accumulator. This Impulse C process detects data dependencies, saves the values that cannot be computed, processes stored values when input cannot be processed, and inserts bubbles when necessary. Finally, we had to minimize the number of data transfers in the inter-FPGA communication bus to maximize performance. So, we replicated data in both FPGAs—namely, the edges and the nodes' conservative values. This data-transfer

Table 3. Resource utilization by different FPGAs.

	Look-up tables				
Resource measure	**Combinational**	**Memory**	**Registers**	**DSP blocks**	**Memory (Mbits)**
Available resources in Stratix SE260 FPGA	203,520	101,760	203,520	768	15,040
Resource usage in FPGA 1	61,163 (30.1%)	704 (0.7%)	88,033 (43.3%)	436 (56.8%)	7,502 (49.9%)
Resource usage in FPGA 2	142,677 (70.1%)	7,978 (7.8%)	140,048 (68.8%)	764 (99.5%)	11,715 (77.9%)

minimization imposed restrictions on the partitioning, which caused an imbalance in the area utilization of the two FPGAs (see Table 3).

Figure 3 shows a simplified view of the hardware architecture. The input stream comes to FPGA 1 from the FSB passing through the bridge FPGA. Although data is stored in the internal memory of FPGA 1, it is also forwarded to FPGA 2, which stores it in its internal memory as well. Then, routines 1, 2.1, 2.2, 2.2.1, and 2.3 of Figure 2 are computed in parallel in FPGA 1. Specifically, gradients are computed on FPGA 1 but are stored in FPGA 2. Once gradient computation is finished, inviscid and viscous residuals are computed in parallel in FPGA 2. Finally, time integration is performed in FPGA 1, which returns the target nodes' conservative values to the CPU through the bridge FPGA.

Because our hardware accelerator uses only the internal FPGA memory, we can compute mesh fragments containing up to 12,288 nodes and 24,576 edges. Real meshes are actually much bigger, so we used a hardware-software solution. The mesh is split into fragments so that each part is computed in the hardware accelerator separately, and then each fragment's computed values are merged by the CPU in the software. When all fragments are processed, the node values are updated, and a subsequent time iteration can be computed.

A good mesh partition is crucial to our obtaining good performance results. At least two neighborhood levels must be added to each partition, and each subdomain must be connected to avoid duplicate data and computations. To get these partitions, we used the Metis tool.[10] Another key factor to avoid incurring a penalty in reassembling the returned values from the accelerator is overlapping the work in the XD2000i and CPU. We developed a double-buffering stream system to overcome this challenge. With this solution, the CPU updates the streams for the next iteration while data is being processed in the hardware accelerator.

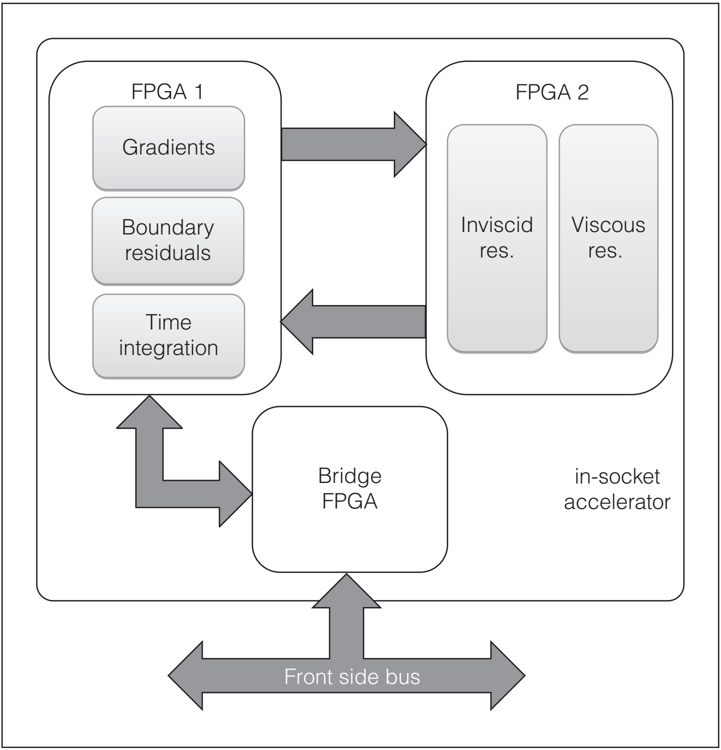

Figure 3. Hardware processes distribution.

A rough estimation of the effort saved from the HLL approach is evident from the source line count. Our original in-house C++ solver has 4,570 lines of code, whereas the Impulse C code is 8,990 lines, which are translated into 102,175 lines of VHDL code. Considering an effort equivalence of three Impulse C lines for each VHDL line,[8] we estimate that the implementation effort has been reduced by a factor of 34. Even if we consider that the VHDL code generated by Impulse C is not optimal and that some of the engineers involved in the study by El-Araby, Merchant, and El-Ghazawi did

Table 4. Mesh characteristics and mean execution time per iteration.

					CPU		1,000 iterations in XD2000i accelerator		1 iteration in XD2000i accelerator	
Mesh	Nodes	Edges	Triangles	Size (Mbytes)	GCC (ms)	ICPC (ms)	Time (ms)	Speedup	Time (ms)*	Speedup
NACA_0012_s	5,233	15,449	10,216	0.64	14.5	13.8	1.0	13.80	1.8 (0.8 + 1.0)	7.67
NACA_4412	6,492	19,207	12,715	0.79	22.6	22.5	1.4	16.07	2.4 (1.0 + 1.4)	9.38
NACA_0012-m	7,908	23,296	15,388	0.97	30.5	30.3	1.6	18.94	2.7 (1.1 + 1.6)	11.22
NACA_0012-1	494,128	1,478,960	984,832	70.83	4,115	3,866	–	–	170 (43 + 127)	22.74

* The numbers in parentheses indicate how the execution time is split between data transfer and raw computation time, respectively.

not have experience with Impulse C or HDL development,[8] the difference is significant enough to justify the use of HLL tools.

Results

We compared the execution of routines 1, 2, and 3 in the CPU against the execution in the XD2000i accelerator, including data transfers and data structure updates. The CPU was an Intel Xeon L5408 (a 45-nm quad-core processor running at 2.13 GHz with 12 Mbytes of Level-2 cache) running a 64-bit CentOS Linux distribution. The system also featured 32 Gbytes of RAM and an FSB running at 1,066 MHz.

We evaluated four test cases. Three of them (NACA_0012_s, NACA_0012_m, and NACA_0012_l) were different discretizations of a NACA_0012 airfoil. The other test case corresponded to a NACA_4412 wing section. All of them consisted of triangle-shaped control volumes. Table 4 includes the mesh sizes in terms of nodes, edges, and triangles, as well as the size of the stream sent to the XD2000i accelerator. The table also shows the execution times and speedups achieved. We compiled our original Navier-Stokes solver with two different compilers: GCC 4.12 and ICPC 12.0.2. Execution times for the software were significantly improved when we enabled compiler optimizations. Specifically, GCC was invoked with the -O3 flag, whereas ICPC was called with -O3-xSSE4.1-parallel-static-intel-axSSE4.1-ip-ipo0.

With regard to execution with the XD2000i accelerator, the NACA_0012_s, NACA_4412, and NACA_0012_m grids fit entirely in the internal FPGA memory. Hence, the algorithm could be executed any desired number of iterations in the hardware module, eliminating data transfers to the host main memory. For this reason, the speedup was better when we performed 1,000 time integrations in the accelerator. Also, this test let us accurately compute the raw execution time inside the XD2000i because the time spent in data transmission was negligible. This raw computation time is the second term given in parentheses in the last column of Table 4; the transfer data time is the first term. As Table 4 shows, the execution time with the XD2000i module is one order of magnitude faster than in our original C++ solver.

The NACA_0012_l mesh did not fit in the internal FPGA memory, so we had to partition it. Thus, we expected degradation in performance for this case. To improve performance, we explored how to minimize the number of data transfers. By adding $2N$ neighborhood layers to each fragment, we can compute N time iterations without interacting with the CPU. There is a trade-off, because the stream size for each fragment grows with N, and redundant computations stem from the greater overlapping between fragments. However, CPU performance degrades considerably more with increasing mesh size. This degradation is due to the increased cache misses from the irregular access pattern of unstructured meshes and the augmented memory utilization. Therefore, the best speedup was for this last case, which was 22.74 times faster. We obtained this value when the mesh was split into 85 fragments and two time integrations were computed in the XD2000i module for each fragment.

The bottlenecks in the current implementation are communication and local-memory capacity. The current communication limits are imposed by the FSB and the inter-FPGA links. We estimate that improving the latter link would allow for a 2× improvement in the raw computation time. Also, a greater amount of internal memory would result in better performance because bigger mesh fragments would be processed in the FPGA.

We also measured AC power consumption. The complete system running the operating system without any computational load consumed 183.9 W, without configuring the FPGAs. (Power consumption increases when configuring the FPGAs.) This value represents all the static-power consumption: fans, disks, and other circuitry. The extra power added by running the software application consumed an average of 36.5 W (including the memory accesses and processor computations). By contrast, the hardware-software solution using the CPU and the XD2000i consumed an average of 43.1 W, of which 18 W corresponded to the XD2000i module.

The preceding measurements show that using the in-socket accelerator adds negligible power consumption. Thus, using the XD2000i module lets users reduce both execution time and energy consumption by more than one order of magnitude.

Our results show that a methodology based on high-level languages, low-latency floating-point arithmetic, and in-socket acceleration is effective for accelerating complex scientific applications. However, without the help of high-level language compilers—more specifically, Impulse C—this project would have required considerably more effort. Impulse C let us reduce the development time from months to weeks, without compromising performance. Moreover, a big part of this success was due to the XD2000i In-Socket Accelerator, which minimized the I/O bottlenecks that are present in other accelerator technologies. ■

Acknowledgments

This work was supported by the Dovres (Design Optimization by Virtual Reality Simulation) project under the Airbus FuSim-E (Future Simulation Concept) Programme initiative.

References

1. L. Xiao et al., "Auto-FCD: Efficiently Parallelizing CFD Applications on Clusters," *Proc. IEEE Int'l Conf. Cluster Computing,* IEEE CS Press, 2003, pp. 46-53.
2. T. Brandvik and G. Pullan, "Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware," *Proc. 46th AIAA Aerospace Sciences Meeting,* Am. Inst. of Aeronautics and Astronautics, 2008, AIAA paper no. 2008-607.
3. J.C. Thibault and I. Senocak, "CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows," *Proc. 47th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition,* Am. Inst. of Aeronautics and Astronautics, 2009, AIAA paper no. 2009-758.
4. V.G. Asouti et al., "Unsteady CFD Computations Using Vertex-Centered Finite Volumes for Unstructured Grids on Graphics Processing Units," *Int'l J. Numerical Methods in Fluids,* 19 May 2010, doi:10.1002/fld.2352.
5. H. Morisita et al., "Implementation and Evaluation of an Arithmetic Pipeline on FLOPS-2D: Multi-FPGA System," *ACM SIGARCH Computer Architecture News,* vol. 38, no. 4, 2010, pp. 8-13.
6. W.D. Smith and A.R. Schnore, "Towards an RCC-Based Accelerator for Computational Fluid Dynamics Applications," *J. Supercomputing,* vol. 30, no. 3, 2004, pp. 239-261.
7. J. Curreri et al., "Performance Analysis with High-Level Languages for High-Performance Reconfigurable Computing," *Proc. 16th Int'l Symp. Field-Programmable Custom Computing Machines* (FCCM 08), IEEE CS Press, 2008, pp. 23-30.
8. E. El-Araby, S.G. Merchant, and T. El-Ghazawi, "A Framework for Evaluating High-Level Design Methodologies for High-Performance Reconfigurable Computers," *IEEE Trans. Parallel and Distributed Systems,* vol. 22, no. 1, 2011, pp. 33-45.
9. "XD2000i FPGA In-Socket Accelerator for Intel FSB," XtremeData; http://www.xtremedata.com/products/accelerators/in-socket-accelerator/xd2000i.
10. G. Karypis and V. Kumar, *METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0,* tech. report, Dept. of Computer Science, Univ. of Minnesota, Minneapolis, 1998.

Diego Sanchez-Roman is pursuing a PhD in Escuela Politecnica Superior at Universidad Autonoma de Madrid. His research interests include computer architecture and high-performance reconfigurable computing. He has an MTech in computer science from Universidad Autonoma de Madrid.

Gustavo Sutter is an associate professor in Escuela Politecnica Superior at Universidad Autonoma de Madrid. His research interests include computer arithmetic and architectures, algorithms in hardware,

reconfigurable devices, high-performance computing, and low-power design techniques. He has a PhD in computer engineering from Universidad Autonoma de Madrid. He is a member of IEEE.

Sergio Lopez-Buedo is an associate professor in Escuela Politecnica Superior at Universidad Autonoma de Madrid, and is also the founder of Naudit HPCN. His research interests include high-performance computing and communication applications of reconfigurable devices. He has a PhD in computer engineering from Universidad Autonoma de Madrid. He is a member of IEEE.

Ivan Gonzalez is an associate professor in Escuela Politecnica Superior at Universidad Autonoma de Madrid. His research interests include parallel algorithms and performance tuning for heterogeneous computing. He has a PhD in computer engineering from Universidad Autonoma de Madrid.

Francisco J. Gomez-Arribas is an associate professor in Escuela Politecnica Superior at Universidad Autonoma de Madrid. His research interests include reconfigurable-computing applications, with a special focus on the design of multiprocessor systems and their integration in heterogeneous clusters. He has a PhD in physics from Universidad Autonoma de Madrid.

Javier Aracil is a full professor in Escuela Politecnica Superior at Universidad Autonoma de Madrid. His research interests include computational mathematics, and traffic analysis and characterization of communication networks. He has a PhD in telecommunications engineering from Universidad Politecnica de Madrid. He is a senior member of IEEE.

Francisco Palacios is an engineering research associate at Stanford University. His research interests include mechanical engineering and applied mathematics. He has a PhD in applied mathematics from Universidad Autonoma de Madrid.

■ Direct comments and questions about this article to Diego Sanchez-Roman, Escuela Politecnica Superior, Universidad Autonoma de Madrid, Cantoblanco, E-28049 Madrid, Spain; d.sanchez@uam.es.