AUTOMATYKA • 2011 • Tom 15 • Zeszyt 3

Maciej Wielgosz*, Ernest Jamro*, Paweł Russek*, Kazimierz Wiatr*

FPGA Implementation of Exchange-Correlation Potential Calculation for DFT

1. Introduction

Quantum chemistry is a computationally demanding branch of science. Most of the experiments conducted involve processing huge volumes of data and the calculation time plays a critical role due to the highly competitive nature of this field. Throughout the last decade, many various systems [1, 2, 3] have been deployed aiming to streamline quantum chemistry computations. However most of them were cluster solutions exploiting course-grain parallelism. This, in turn, leaves a broad range of potential improvements at the fine-grain level. Therefore several initiatives [4, 5, 6, 7] have been developed to deal with this issue by means of reconfigurable hardware like FPGAs (Field Programmable Gate Arrays). Adopting such an approach means that facing a growing complexity of quantum chemistry routines together with a rise in granularity level. Moreover, floating-point precision is also a primary concern when dealing with elementary operations, thus some measures have been taken to optimize them both in terms of resource consumption and the calculation speed. At first glance, these goals seem to be in opposition to each other, but FPGA technology allows manipulation at the bit level, which gives a huge advantage over GPP (General Purpose Processor) and GPGPU (General-Purpose Computing on Graphics Processing Units) solutions. This also justifies the choice of HDL (Hardware Description Language) as the design language instead of one of the HLLs (High Level Languages) which, despite their huge flexibility, do not provide access to low level bit operation.

2. Algorithm consideration

A comprehensive study of numerical methods for computational chemistry is beyond the scope of this paper. Nevertheless, some introductory information is provided to set

* AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Department of Electronics, al. A. Mickiewicza 30, 30-059 Krakow, Poland

a fragment of the presented algorithm in the context of a quantum chemistry SCF computational procedure.

The Schrödinger equation is a primary mathematical formula in computational quantum chemistry. It provides an exhaustive molecular description in the form of a time-dependent wave function that reflects the spatial coordinates of all component elements of a molecule and time.

Despite the fact that a wave function $\Psi(x, t)$ is a complete description of a molecule, it does not make sense by itself. The square of the absolute value of the wave function denotes the probability density where the particle will be detected. Essentially, all the interesting physical parameters may be derived from a wave function.

The time-dependent Schrödinger equation is given by:

$$\hat{H}\psi(x,t) = i\hbar\frac{\partial}{\partial t}\psi(x,t) \tag{1}$$

where $\Psi(x, t)$ is an orbital function, h stands for the Planck constant divided by 2π. H is the Hamiltonian operator referring to the total energy of the system. The final formula of the operator depends on mass, charge and relative positions of the particles that compose a molecule. In quantum chemistry only a certain set of possible molecule states are considered. These are stationary states in which probability density is not determined by time (2).

$$\hat{H}\psi(x) = E\psi(x) \tag{2}$$

Eigenfunctions of the (2) are wave functions of the stationary states and eigenvalues are energy values corresponding to these states.

Equation (2) can be analytically solved only for very basic cases. Hydrogen and lithium (He^+, Li^{2+}, ...) atoms are considered to be the upper limit of that approach. Therefore, some approximate methods of solving the Schrödinger equation have been adopted. One of the most common is Hartree–Fock algorithm which is also one of the simplest approximation theories for solving the many-body Hamiltonian.

The general procedure for solving the Hartree–Fock equations is to make orbitals self-consistent with the potential field they generate. This is achieved through an iterative trial-and-error computational process, for which reason the entire procedure is called the self-consistent field (SCF) method.

SCF procedure for solving the Hartree–Fock equation leads to the following set of eigenvalue equations in matrix formulation.

$$FC - SCE = 0 \tag{3}$$

F is the Fock-operator, C the matrix of the unknown coefficients, S the overlap matrix and E is the energy eigenvalues. All matrices are of the same size.

Solving the Hartree–Fock equation is an iterative process. The coefficients (corresponding to an electric field) are used to build the Fock-operator F, with which the system

of linear equations is solved again to get a new solution (a new electric field). The procedure is repeated until the solution no longer changes. However, it does not take into account some vital particle interactions which in turn affect calculation accuracy.

There are several other methods which do not have such limitations. One of them is DFT (*Density Functional Theory*) [8] which was partially implemented in the FPGA and is presented in the next sections of this paper.

In the typical implementation, the DFT calculations require integrating the exchange-correlation (XC) potential matrix into the Gaussian-type orbital basis set

$$\chi_{klm}(\boldsymbol{r}) = r_x^k r_y^l r_z^m \sum_i C_i e^{-\alpha_i \boldsymbol{r}^2} \tag{4}$$

where r_x, r_y, r_z denote atom-centered spatial coordinates within the molecule. C_x, C_y, C_z are normalization coefficients. The k, l and m indices depend on the type of atomic shell (s, p, d or f). The atomic base used for calculation is represented by C_i and α_i coefficients.

The integration is performed numerically due to the complicated, highly nonlinear structure of the XC potential. This requires calculating the orbital values at each grid point. It has been shown that speeding up this phase should result in considerable acceleration of the whole DFT calculations.

3. Exchange-correlation potential generation

The generation of exchange-correlation potential is considered to be a hot spot of the whole DFT routine, due to its large overall contribution to total calculation load. Therefore this part of the algorithm was chosen to be implemented on FPGA.

Atomic basis functions and spatial grid coordinates are input data for the algorithm that generates exchange-correlation potential. Points are divided into blocks according to their location within the grid. The size of a single block ranges from 128 to 512 points. Such a division allows the blocks to be spread across computational nodes, so the calculations are conducted concurrently and results are merged afterwards. Such a coarse-grained approach is not supported directly by FPGAs yet, due to an insufficient amount of resources. However we can expect that the future FPGA will provide enough resources to implement the feature.

The focus of this work is to accelerate the calculation within a single “grain” which is a block of points. The average size of a grid for a molecule composed of a few atoms is 500 000 points.

Electron density is spread nonlinearly, which is reflected by a strongly nonlinear distribution of the grid points in the three dimensional space. Substantially more points are located closer to the atom’s kernel and their number gradually drops towards the molecule’s border.

Calculating exchange-correlation potential is carried out in several steps:

Firstly, orbital values for all points within the block are calculated

$$\chi_1(P),\chi_2(P),\chi_3(P)..... \tag{5}$$

$$\chi_{klm}(r) = C_x C_y C_z r_x^k r_y^l r_z^m \sum_i c_i N_i e^{-\alpha_i r^2} \tag{6}$$

This part of the algorithm was implemented in FPGA and is described in detail later in this paper.

Secondly, matrix S is generated.

$$S = \begin{bmatrix} \chi_1(P)*\chi_1(P) & \chi_2(P)*\chi_1(P) & \chi_3(P)*\chi_1(P) \\ \chi_1(P)*\chi_2(P) & \chi_2(P)*\chi_2(P) & \chi_3(P)*\chi_2(P) \\ \chi_1(P)*\chi_3(P) & \chi_2(P)*\chi_3(P) & \chi_3(P)*\chi_3(P) \end{bmatrix} \tag{7}$$

This is an example of generating *S* matrix for just three orbitals. In case of real molecules, the number of orbitals is significantly larger.

Subsequently, the electron density is calculated at each point of the grid, where Δ is a density matrix. Both Δ and *S* matrices are symmetrical, as well as all the other matrices.

$$\rho = Tr(S \cdot \Delta) \tag{8}$$

where

$$\begin{aligned} Tr(S*\Delta) &= \chi_1(P)*\chi_1(P)*\Delta_{11} + \chi_2(P)*\chi_1(P)*\Delta_{21} + \chi_3(P)*\chi_1(P)*\Delta_{31} \\ &+\chi_1(P)*\chi_2(P)*\Delta_{12} + \chi_2(P)*\chi_2(P)*\Delta_{22} + \chi_3(P)*\chi_2(P)*\Delta_{32} \\ &+\chi_1(P)*\chi_3(P)*\Delta_{13} + \chi_2(P)*\chi_3(P)*\Delta_{23} + \chi_3(P)*\chi_3(P)*\Delta_{33} = \\ &2*\chi_2(P)*\chi_1(P)*\Delta_{21} + 2*\chi_3(P)*\chi_1(P)*\Delta_{31} + 2*\chi_3(P)*\chi_2(P)*\Delta_{32} + \\ &\chi_1(P)*\chi_1(P)*\Delta_{11} + \chi_2(P)*\chi_2(P)*\Delta_{22} + \chi_3(P)*\chi_3(P)*\Delta_{33} \end{aligned}$$

After the electron density has been calculated, exchange-correlation potential for a given point is evaluated as follows:

$$v = k*\rho^{\frac{1}{3}} \tag{9}$$

where the constant *k* represents a whole set of different operands (subject to strong variations) and ρ is the electron density (obtained at the previous step). The presented operation is carried out by the general purpose processor due to its indeterminate execution flow (all the other operations discussed hereby are implemented in reconfigurable logic).

FPGA implementation of this step would result in substantial resource consumption and a small performance gain.

Finally, the contribution to exchange-correlation matrix V is computed.

$$V = V + S \cdot v \cdot w \tag{10}$$

where w is the grid point value, v – scalar value generated at the previous step of the algorithm execution.

All of the presented steps of the procedure for generating exchange-correlation potential involved designing dedicated logic of an adjustable precision to ensure sufficient accuracy and low resource consumption.

4. The platform

The RASC hardware blade contains two computational FPGAs (Xilinx Virtx-4 LX200s) and two TIO ASICs. The platform is also equipped with 10 synchronous static RAM dual in-line memory modules (SSRAM DIMMs), which are grouped into three logical memory structures (Fig. 1).

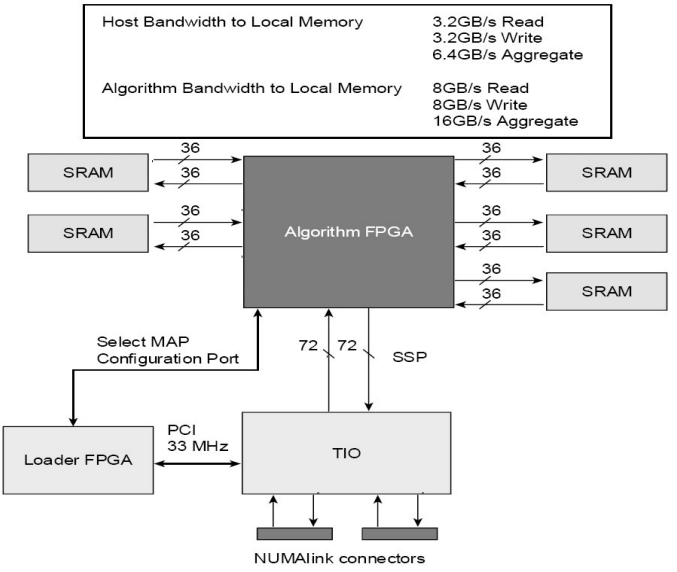


Fig. 1. Block diagram of the RASC slice [10]

The RASC communication module is based on an application-specific integrated circuit (ASIC) called TIO which attaches directly to the Altix system NUMAlink data bus.

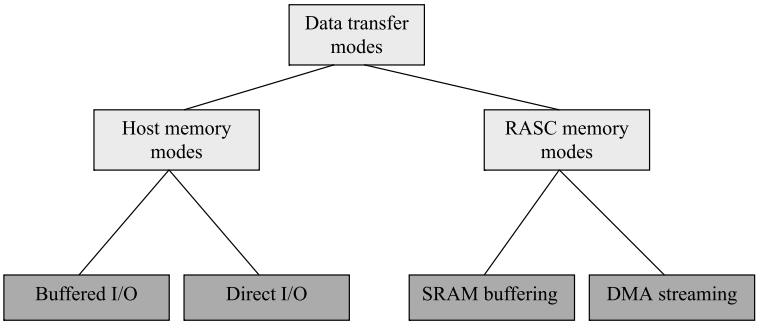


Fig. 2. Transfer modes

There are two FPGAs available on the RASC platform, but no interconnecting link on the board between them has been provided. Nevertheless it is still possible to communicate across both FPGAs through an external hub which, however, adds some overhead.

The SGI RASC v2.1 [10] device has four transmission modes. These options cover a large spectrum of potential memory division and managing methods. It is worth emphasizing that the selection of data transfer mode is done on the Host side as well as on the RASC side (Fig. 2).

According to SGI documentation, it is possible to transfer data between the RASC platform and the host processor at a speed of 3.2 GB/s in each direction. Several speed tests have been done in order to determine the real transfer rate. Consequently, two exp() functions have been implemented [9], each of which processes a 64-bit data chunk, so each clock cycle one 128 bit input word is fed into the implemented twin module structure and one result is obtained. An effective transfer is about 1.2 GB/s.

Among all the available transfer modes, Streaming DMA is the most efficient and reduces time of single exp() execution to 7.8 ns (Fig. 3).

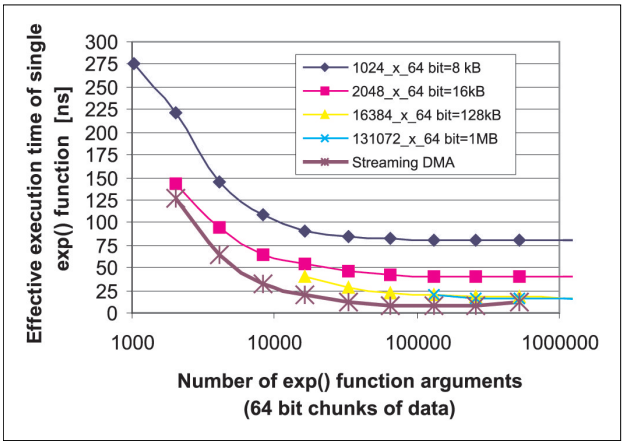


Fig. 3. Effective execution time of single exp() function

Execution of a hardware algorithm from the host processor perspective is composed of several steps. The most time-consuming one is FPGA programming. Therefore it is recommended to perform it as rarely as possible. This, in turn, imposes some additional requirements on the accelerated algorithm that must be taken into account when frequent reconfiguration is considered.

5. Architecture of the orbital module

Implementation of the orbital generation function requires designing hardware modules as well as software routines to be run on the host processor. As mentioned above, an FPGA accelerator – RASC RC100 together with Itanium 2 1.6 GHz processor hosted by Altix 4700 computer have been employed.

Orbital function is given by the equation (1), which can be broken down into two sections, the exponential part:

$$\chi_e(r) = \sum_i C_i e^{-\alpha_i \boldsymbol{r}^2} \tag{11}$$

and the polynomial part:

$$\chi_p(r) = r_x^k r_y^l r_z^m \tag{12}$$

Consequently, the orbital logic is composed of two main units – exponential and polynomial units, each of which is a fully-pipelined floating-point reconfigurable unit capable of generating results every clock cycle.

The host processor handles FPGA control messages and manages data streaming. Therefore several routines must be executed on the host processor in order to carry out the computations with the RASC accelerator. The size of a singe block of points ranges from 128 to 512. A single molecule can contain up to 32 different atoms. Furthermore, it is assumed that the upper limit of atomic base coefficients (C_i oraz α_i) is 64.

A uniform input data stream is well suited for FPGA implementation, but unfortunately it is not the case for the algorithm described in this paper. The complexity of quantum chemistry computational formulas is also reflected by the diverse structure of input data which requires an appropriate compacting scheme in order to take a full advantage of available data throughput. The presented calculations involve transferring ten vectors' input data to the RASC memory. Vectors and their maximum sizes are listed below:

1. *T_Ci* – vector of atoms' *Ci* coefficients, (number of atoms)×(average number of *Ci* per atom)= 32×64=2048 double.
2. T_α_i – vector of atoms' α_i coefficients, (number of atoms)×(average number of *Ci* per atom) = 32×64=2048 double.

3. T_r_x – vector of x coordinate of atom-centered points, (number of points which make up a single block)×(average number of atoms in a single block) = 512×32 =16384.
4. T_r_y – vector of y coordinate of atom-centered points, (number of points which make up a single block)×(average number of atoms in a single block) = 512×32 =16384 double.
5. T_r_z – vector of z coordinate of atom-centered points, (number of points forming a single block)×(average number of atoms in a single block) = 512×32 =16384 double.
6. T_r^2– vector of the square of the spatial coordinate sum ($r^2 = r_x^2 + r_y^2 + r_z^2$), (number of points which make up block)×(average number of atoms in a single block) = 512×32 = = 16384 double.
7. *T_kw* – vector of relative location of atom *Ci* and α_i coefficients used to calculate the radial (exponential) part of the orbital function, (average number of α_i, *Ci* coefficients per atom)×(average number of α_i, *Ci* coefficients per exponential part) = 32× 64/4 = = 512 byte.
8. *T_tp* – vector of control data for internal FSM – the shell type (s, p, d or f) currently being processed, (average number of atoms)×(orbital number per atom)/(average number of orbitals per shell) = 32× 64/4 = 512 byte.
9. *T_a* – vector of atom *Ci* and α_i coefficient number used to calculate the polynomial part of the orbital function, number of atoms = 32, byte.
10. *Delta* – vector used in the next step of calculations (S matrix generation), (number of orbitals)×(number of orbitals per single point)/2 = = 512×512/2 = 131072 double.

A dedicated formatting method for the input data was introduced and implemented on the host processor which performs the following operations (Fig. 4):

– Building vector $T\_\alpha_i\_C_i$ by merging vectors T_C_i and T_α_i
– Building vector $T\_r_x\_r_y$ by merging vectors T_r_x and T_r_y

Control signals are also appropriately formed before they are sent over the bus to the RASC platform. This process involves building vector *T_kw_tp_a* out of *T_kw*, *T_tp* and *T_a* (Fig. 5).

All the data is transferred to the FPGA from the host processor in two steps. Initially, data is sent to the accelerator's local memory of 40 MB size, and is subsequently fetched by the orbital calculation module implemented on the FPGA. The RASC platform can also work in a streaming mode which does not involve storing data in local memory, but due to the non-uniform input data structure, the multi-buffering mode is more efficient.

It is worth noting that the capacity of Xilinx Virtex-4 LX200 internal memory is insufficient to store all the data. Additionally, BRAM and FIFO memories share a single 128 bit-width bus.

Therefore a custom mechanism for request tracking has been introduced to handle data dispatching among internal memories (Fig. 6). Furthermore, this mechanism prevents units from getting stalled due to a lack of input data. The key issue while designing the orbital

module was the right choice of FIFO size and data request thresholds. Because of the molecule-dependent structure of input data, this is not a trivial task and leads to some critical trade-offs.

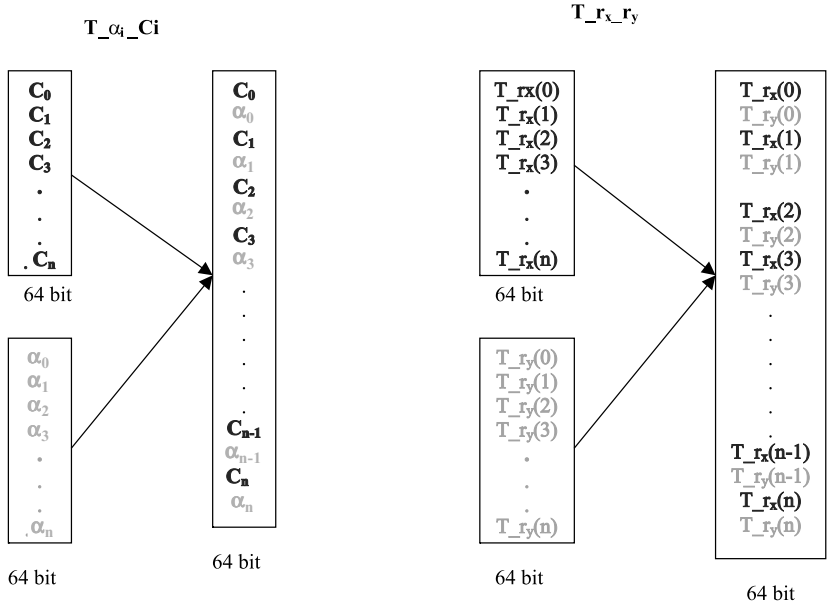
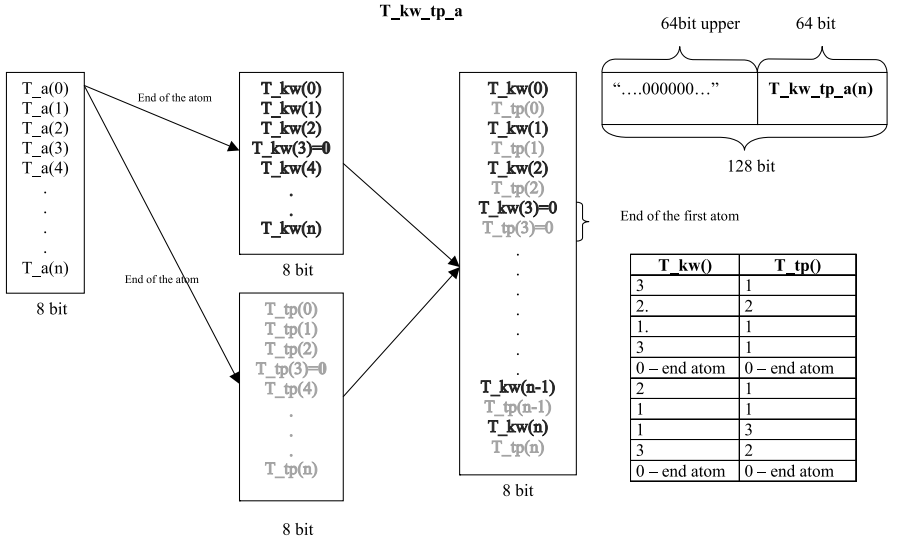


Fig. 4. Input data serialization



T_kw()	T_tp()
3	1
2.	2
1.	1
3	1
0 – end atom	0 – end atom
2	1
1	1
1	3
3	2
0 – end atom	0 – end atom

Fig. 5. Control data serialization

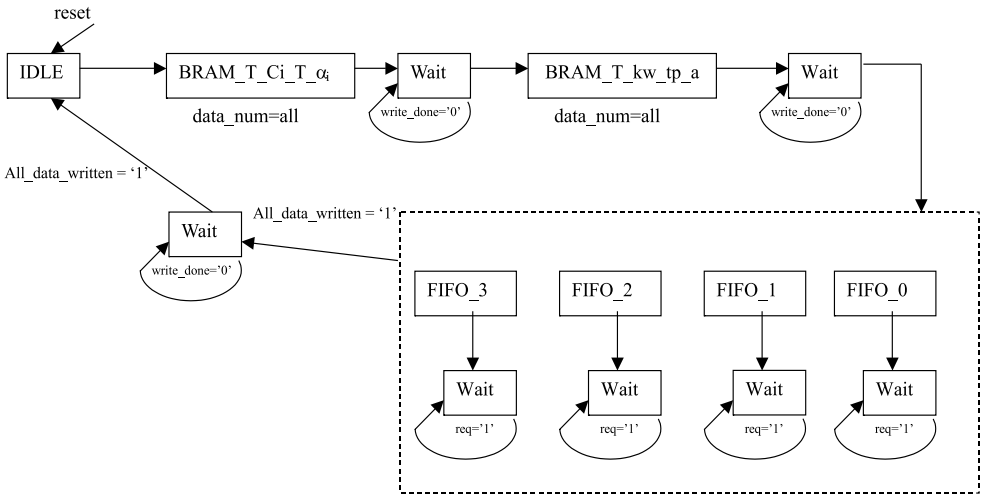


Fig. 6. Block diagram of the RASC slice

The amount of data sent to the accelerator varies depending both on the size of the molecule and the single data block. Given the maximum molecule size and 512 points in a single block, the amount of data to be sent would be roughly 1.5 MB.

The procedure presented in Figure 7 shows a complete execution flow of the hardware-accelerated algorithm. The accelerator starts off after a launch command has been issued by the host processor and the system runs in the loop till calculations have been performed for all the blocks of data. It is worth noting that the accelerator initialization process, which is very time consuming (7.5 ms) is performed only once for every block size.

As depicted in Figure 8, the orbital calculation module is composed of two core units – EP and PP. Both of them were developed separately and can act independently, but this particular implementation requires their tight cooperation, and so the pipelined data interlink has been provided.

Several computational tests were conducted to compare the RASC performance against an Intel Itanium 2 processor. Some of the tests were done for the water molecule calculated in a block composed of 512 grid points. It took the Itanium 2 processor roughly 2885 μs to perform the calculation of the orbital function, which is close to the 3174 μs consumed by FPGA. It is worth noting that the predominant shell of the water molecule is s (the 6-31G atomic base) which is actually the worst case in terms of the acceleration achieved.

Due to the architectural constraints of the hardware module, shell types have an impact on the overall performance. Consequently, an increase of the atom shell size leads to better utilization of the LUT and pipeline mechanisms implemented on the FPGA. This, in turn, results in a noticeable acceleration. For the f shell and 6 coefficients, an acceleration of roughly 3× is achieved over the Itanium 2 processor. Exemplary FPGA implementation results are presented in Table 1.

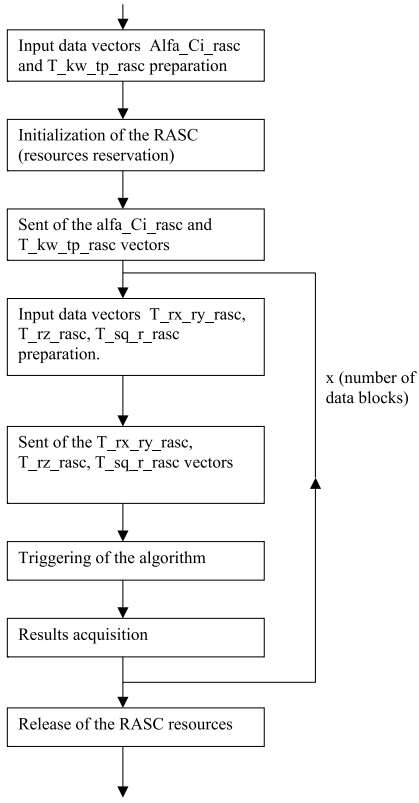


Fig. 7. The algorithm execution flow

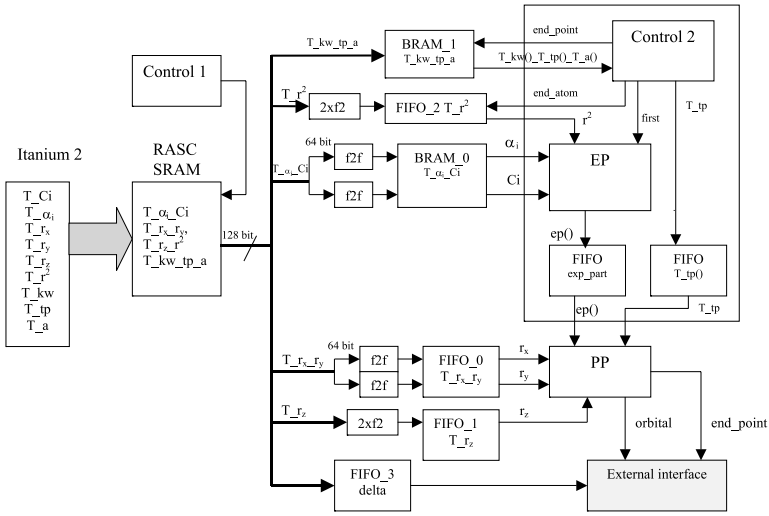


Fig. 8. Block diagram of the complete system

Table 1
Implementation results of the orbital module for a single data precision
(Xilinx Virtex-4 LX200)

Implementation results	#4-input LUT	#FF	# 18-Kb BRAMs
Orbital module	4164 (2%)	6257 (3%)	6 (0.06%)
Orbital module + core services	13495 (11%)	20204 (11%)	29 (8%)

6. Architecture of the S matrix generator

The S matrix (7) is generated for each grid point fed into the hardware module. χ vectors (orbitals) are input data to the module. Those vectors are transferred to the BRAM memories in a ping-pong manner. Which means that while one port of memory provides the S matix generation logic with data (Fig. 9), the second one accepts data transferred from the orbital calculation module. Such an approach allows for constant data processing at the expense of a two-fold increase in the amount of memory employed.

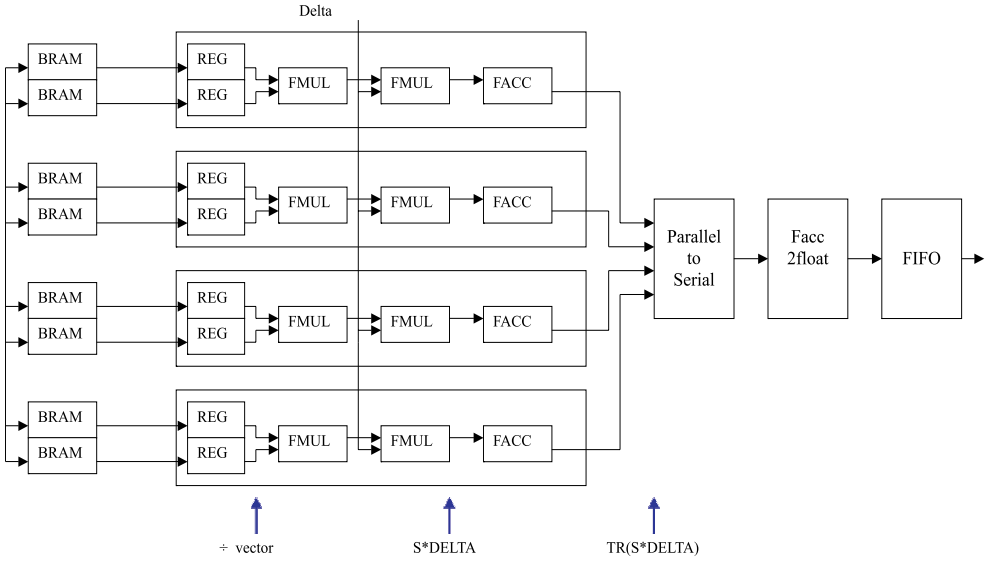


Fig. 9. S matrix generation module

Due to the symmetry of the S matrix, it is sufficient to perform just half of the all operations that are theoretically required.

To boost the module's performance, a few parallel modules have been employed which compute the S matrix for different χ vectors. The number of those units is set as a generic parameter in VHDL source code. All the results are converted to floating-point IEEE-754 format before there are passed to a subsequent module (exchange-correlation potential logic).

Table 2 contains FPGA implementation results for two different numbers of parallel modules (Fig. 9)

Table 2
Implementation results of the S matrix generation module

NUM_TR_CALC	#4-input LUT	#FF	# 18-Kb BRAMs
1	1552 (0.8%)	2314 (1.2%)	2 (0.5%)
16	27756 (15%)	19159 (10%)	32(9%)

7. Conclusions

This paper presents the hardware implementation part of the DFT algorithm. The ultimate goal of the authors is to implement a complete calculation routine of exchange-correlation potential. The presented modules are pipelined floating-point units optimized to meet both precision and speed requirements. The orbital calculation module allows us to achieve a 3× acceleration, and the S matrix generation module works up to 16× faster than an Itanium 2 depending on the number of concurrently working hardware threads incorporated into the module. It is estimated that the whole exchange correlation will perform much faster than an Itanium 2 processor. A single 128 bit RASC interface is considered to be a bottleneck of the RASC system. But it can be overcome in the future by employing a platform with more data transfer links (e.g. DRC Computers, Xtreme-Data).

It is worth emphasizing that nowadays FPGAs are considered to be the most energy effective HPC solutions. Therefore we may expect that in next few years they may surpass other platforms (GPUs, GPPs) in this regard.

Acknowledgements

This scholarly work was made thanks to POWIEW project. The project is co-funded by the European Regional Development Fund (ERDF) as a part of the Innovative Economy program.

References

[1] http://www.gaussian.com/.
[2] http://www.msg.chem.iastate.edu/gamess/.
[3] http://www.molpro.net/.
[4] Gothandaraman A., Peterson G., Warren G., Hinde R., Harrison R., *FPGA acceleration of a quantum Monte Carlo application*. Parallel Computing, 34(4–5), 2008, 278–291.
[5] Gothandaraman A., Warren G., Peterson G., Harrison R., *Reconfigurable accelerator for quantum Monte Carlo simulations in N-body systems*. Proc. of the 2006 ACM/IEEE Conference on Supercomputing (Tampa, Florida, November 11–17, 2006). SC '06. ACM, New York, NY, 177.

[6] Ramdas T., Egan G.K., Abramson D., Baldridge K.K., *On ERI Sorting for SIMD Execution of Large-Scale Hartree-Fock SCF.* Computer Physics Communications, vol. 178, 2008, 817–834.
[7] Ramdas T., Egan G., Abramson D., Baldridge K., *Towards a special-purpose computer for Hartree-Fock computations.* Theoretical Chemistry Accounts, vol. 120, 2007, 133–153.
[8] Koch W., Holthausen M., *A Chemist's Guide to Density Functional Theory, Wiley-VCH*. 2nd ed. (Aug. 21 2001).
[9] Wielgosz M, Jamro E., Wiatr K., *Highly Efficient Structure of 64-Bit Exponential Function Implemented in FPGAs.* ARC 2008, Lecture Notes in Springer-Verlag, London LNCS 4943, 274–279.
[10] Silicon Graphics, Inc. Reconfigurable Application-Specific Computing User's Guide, Ver. 005, January 2007, SGI.