electronics

MDPI

Article

Methodology of Firmware Development for ARUZ—An FPGA-Based HPC System

Rafał Kiełbik [1,*], Kamil Rudnicki [1], Zbigniew Mudza [1] and Jarosław Jung [2]

[1] Department of Microelectronics and Computer Science, Lodz University of Technology, ul. Wólczańska 221/223, 90-924 Łódź, Poland; rudnickikamil@gmail.com (K.R.); zmudza@dmcs.pl (Z.M.)
[2] Department of Molecular Physics, Lodz University of Technology, ul. Żeromskiego 116, 90-924 Łódź, Poland; jaroslaw.jung@p.lodz.pl
* Correspondence: rkielbik@dmcs.pl

Received: 5 August 2020; Accepted: 8 September 2020; Published: 10 September 2020

Abstract: ARUZ is a large scale, massively parallel, FPGA-based reconfigurable computational system dedicated primarily to molecular analysis. This paper presents a methodology for ARUZ firmware development that simplifies the process, offers low-level optimization, and facilitates verification. According to this methodology, firstly an expanded, generic, all-in-one VHDL description of variable Processing Elements (PEs) is developed manually. GCC preprocessing is then used to extract only the desired target functionality. A dedicated software instantiates and connects PEs in form of a scalable network, divides it into subsets for chips and generates its HDL description. As a result, individual HDL-coded specification, optimized for certain analysis, is provided for the synthesis tool. Code reuse and automated generation of up to 81% of the code economizes the workload. Using well-optimized VHDL for core description rather than High Level Synthesis eliminates unnecessary overhead. The PE network can be scaled inversely proportional to PEs complexity, in order to efficiently utilize available resources. Moreover, downscaling the problem makes verification during HDL simulations and testing the prototype systems easier.

Keywords: computer aided engineering; design automation; high level synthesis; high performance computing; programmable logic arrays; reconfigurable architectures

1. Introduction

Development of massively parallel computation firmware (configuration, bitstream) for FPGA is a relatively difficult and time-consuming process. Even more so if it needs to be run on thousands of co-operating FPGAs performing a common task.

In this paper, the methodology elaborated and used for ARUZ (in Polish: Analizator Rzeczywistych Układów Złożonych—Analyzer of Real Complex Systems) firmware development is presented. ARUZ is a scalable, fully parallel data processing system equipped with approximately 26,000 FPGAs [1]. Its firmware is understood as a configuration of all of these FPGAs.

ARUZ construction was initially motivated by the need to speed-up the molecular simulations based on a Dynamic Lattice Liquid (DLL) algorithm [2,3], but, due to its reconfigurability, ARUZ can be easily and efficiently adapted to computations based on other algorithms (e.g., Lattice Boltzman method [4–6]). This makes ARUZ a very flexible and powerful High Performance Computing (HPC) tool, but demands a novel approach to firmware development, which is a real challenge at this scale.

2. Hardware Architecture

The detailed description of ARUZ architecture is presented in [1]. The essential aspects of its desing required for the purpose of this paper are outlined below.

ARUZ is composed of 20 panels of DBoards (Daughter Boards) [1]. Each panel contains 144 of them, organized in 12 rows (Figure 1). The operation of DBoards is controlled by 1 MBoard (Mother Board) which is connected to the rows of DBoards via CBoards (Connection Boards). Each board is composed of:

- 1 supervisory FPGA called Master (MMaster on MBoard, CMaster on CBoard and DMaster on DBoard),
- some executive FPGAs called Slaves (1 MSlave on MBoard, 1 CSlave on CBoard and 8 DSlaves on DBoard).

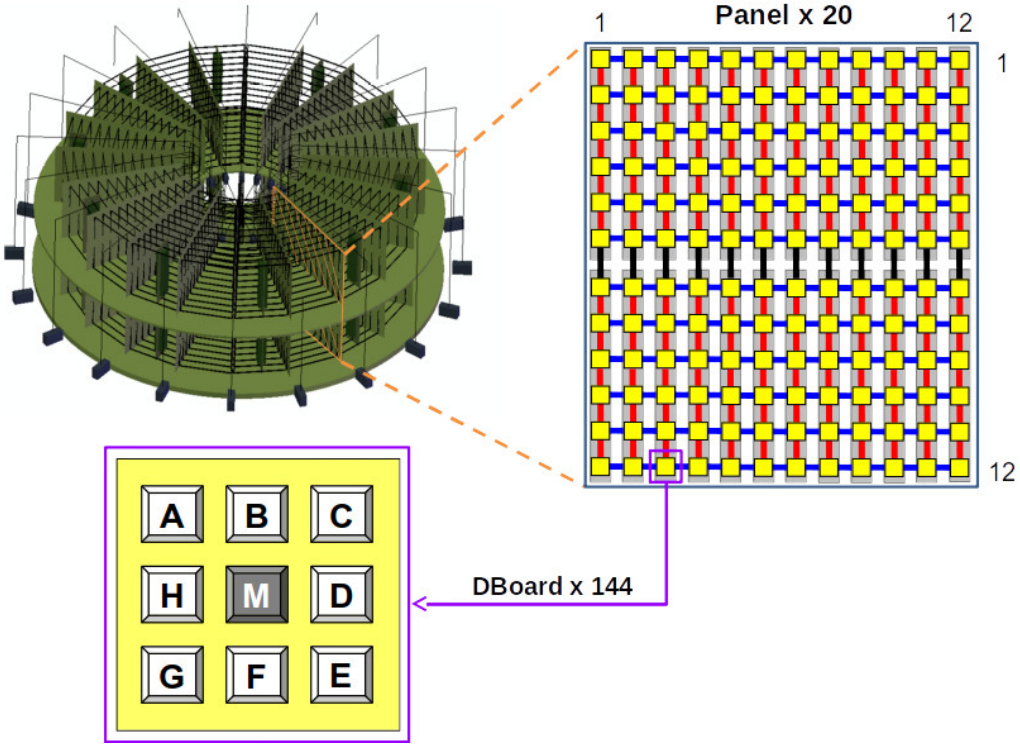

Figure 1. General architecture of ARUZ.

Masters (Zynq XC7Z015 chips) are equipped with the reconfigurable fabric and Dual ARM Cortex-A9 based processor. The configuration of Masters is fixed, and it is uploaded from the on-board flash memory during the power up process. The role of Masters is to manage the operation of Slaves (Artix XC7A200T chips) that is, to upload their configuration from the external servers. This way, the configuration of Slaves is not fixed and can be uploaded on request, thus the internal architecture of ARUZ is flexible and can be adapted to the computations to be performed.

The flexibility of Slaves is crucial in the case of DSlaves (in Figure 1 marked as A-H devices), since ARUZ contains over 23,000 of them. Each DSlave can host a set of Processing Elements (PEs) executing the required computations in parallel (Figure 2). The number of PEs in in a single DSlave depends on their complexity. For example, for a DLL algorithm, each PE represents a Face Centered Cubic (FCC) lattice node in which a molecule (or indivisible group of molecules) can be placed. For a typical molecular simulation variant, 128 nodes can be placed in a single DSlave. For 20 panels, it gives almost 3 million nodes simulated in a fully parallel manner.

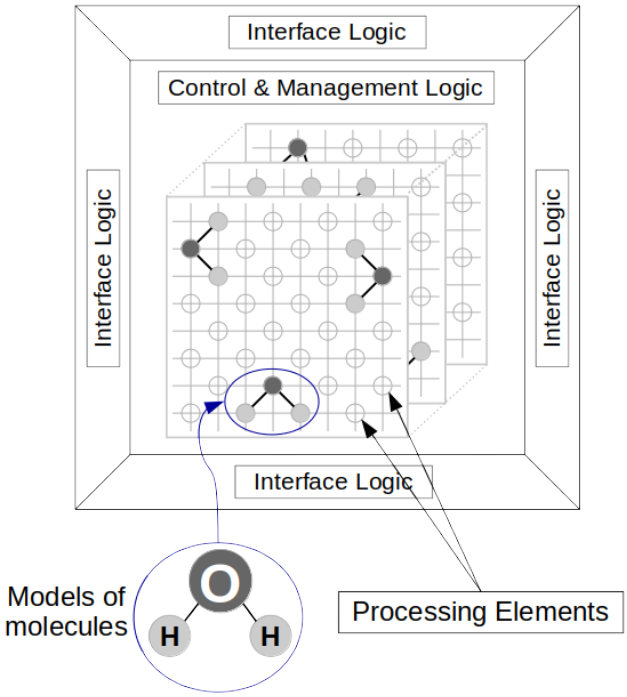

Figure 2. Block diagram of DSlave configuration.

ARUZ FPGAs are interconnected by means of a dense network of cables implementing three types of communication:

- **Global communication** is used for the data exchange between Master chips. It is responsible for configuring Slaves, determining the current state of the whole system (temperatures, hardware defects, etc.), uploading initial data and archiving computation results.
- **Control communication** is dedicated for the control of the computation process. This communication is based on two groups of signal lines i.e., state and command lines. State lines indicate the current state of PEs in DSlaves. On the basis of this information, MSlave issues the commands forcing the next steps of the computations.
- **Local communication** is used for the data exchange between DMaster and DSlaves within 1 DBoard (for the purpose of initialization and acquisition of the results of the computations). However, it is primarily dedicated for the data flow among the PEs located in the neighboring DSlaves. The topology of the local communication interconnections allows for periodic boundary conditions [7,8].

ARUZ is designed for the algorithms for which the global and control communication has a minor influence on its performance and scalability. Therefore, to reduce costs, DMasters in every row of the panel are daisy-chained and only the first DBoard is connected to the server (in the case of global communication) and to the CBoard controlling given panel (in the case of control communication). Such an approach increases message latency; however, it significantly reduces the number of physical links.

On the other hand, the latency in local communication is crucial from the point of view of the ARUZ computational efficiency [1]. To reduce this latency, the dedicated protocol has been developed and the specific placement [7,8] reducing the distance among PEs for periodic boundary conditions has been applied.

3. Challenge

The configuration of ARUZ FPGAs is composed of many components responsible mainly for computations (PEs) but also for communication (e.g., dedicated local communication transceivers),

control (e.g., FSMs—Finite State Machines—controlling the progress of computations) and management (e.g., defect detectors). Some of these components are fixed and can be efficiently used regardless of what computations are being performed. However, most of the components are adapted to the requirements of the specific computations. This mainly concerns:

- The main FSMs responsible for proper algorithm execution;
- The construction of PEs, which—for better efficiency—precisely reflects the needs of the algorithm to be used;
- The resources required to connect these PEs.

Although common components are certainly crucial for the proper operation and overall ARUZ performance, they were developed once and can be easily reused for all computation scenarios. On the other hand, computation specific (lattice-specific and/or algorithm-specific) components may vary greatly depending on what needs to be computed. The real challenge is to provide highly optimized configuration for any desired computation.

The design of ARUZ allows for using it for a wide range of analysis, especially those which require parallel calculations. However, as the primary application of ARUZ is liquid dynamics based on an FCC lattice and DLL algorithm, this paper focuses on the implementation of different variants of such analysis. Even within this narrow field, numerous diverse configurations are required.

Miscellaneous analysis may require different molecular simulation space—from cubic box to plane sheet or approximately a one-dimensional tube. Despite being based on the same FCC principles, variable geometry precludes use of universal PEs' interconnection patterns. Moreover, resource requirements of a single node depend on the algorithm. Thus, the same hardware can be used to simulate more elements of simple construction or fewer objects of greater complexity. Effective mapping of various real-objects to ARUZ requires scalability of the FCC lattice. In order to fully exploit the available resources, dedicated connections should be provided for different geometries and lattice node complexity. The applied maximal distance reduction for periodic boundary condition [7,8] makes interconnections non-trivial and different for individual DSlave chips. In brief, for each geometry variant, eight different lattice structures (connecting up to several hundred PEs) need to be provided.

Moreover, as already mentioned, operation and properties of individual lattice elements are determined by the algorithm being used. Actually, for better optimization, they can even be adapted to the specific variants of the same algorithm. In the discussed case—the DLL algorithm—the desired combination of 13 mechanisms representing various physical phenomena may be chosen. Even though some of the combinations make no physical sense, due to dependencies between mechanisms, there are more than a thousand valid ones.

Theoretically, all functionalities could be implemented in hardware in the form of a single generic solution and any combination of them could be selected (using configuration registers). However, in such a case, hardware resources associated with unused mechanisms would still be utilized. Since molecular analyses are performed on millions of molecules, the node instances need to be multiplied numerously. In the case of the generic approach, it means that the amount of reserved but not used resources (area overhead) grows drastically, which can be considered a major resources mismanagement. Consequently, in order to achieve satisfying performance with minimal overhead in large scale molecular simulations, a specialized approach to firmware optimization had to be adopted. The scale and complexity of the firmware makes its automated generation practically indispensable.

Once an FPGA configuration is synthesized, its operation needs to be verified. In order to detect potential bugs and malfunctions as soon as possible and determine the source of such problems, testing at multiple development stages is recommended. In the case of a generic solution approach, verification procedure is conducted on a single version of firmware in every development cycle. On the other hand, if highly optimized application-specific configurations are to be used, multiple variants need to be synthesized, implemented, and verified. Therefore, automation and simplification of testing procedure might be required to increase productivity. Moreover, the firmware ought to be easy to

maintain and support debugging in the event of any error occurrence (from unexpected behavior to incorrect results).

Due to the unique architecture and unparalleled scale of ARUZ, no off-the-shelf solutions could be used to conduct the entire design flow for the simulator. Most FPGA design tools focus on single chip applications. Even if that was not a problem, generation and processing of scalable interconnection maps supporting periodic boundary conditions are left unresolved. Finally, regardless of the tools used, attempts to improve performance and reduce resource utilization, to increase versatility and exploit adaptability are generally self-contradictory and require several trade-offs. Solving these issues was essential for ARUZ development.

4. Solution

4.1. Language

One of the fundamental decisions that had to be made was the choice of the FPGA design methodology and tools. There are two dominant approaches for digital systems development: based on Hardware Description Languages (HDLs) or more abstract High-Level Synthesis (HLS). Both solutions have their drawbacks and advantages.

Created solely for the purpose of digital circuit designs, the HDL approach is intended for precise description of their architecture. It offers a convenient abstraction of behavioral description with low-level manipulations and control over the way the circuit is synthesized. It allows experienced developers to fully exploit the FPGA chip's capabilities for many applications. The design process is time-consuming but produces excellent results.

HLS, on the other hand, is focused only on functional description, defining what the system does rather than how it is performed. The actual logical circuit structure is generated by automated HLS tools with respect to additional design guideline directives, changes of which may lead to entirely different architecture for the same input. Shifting responsibility from developer to automated tools is not only convenient, but it also reduces design time and makes potential reuse much easier. A parametrizable, complex system desired for ARUZ could benefit greatly from those advantages, provided they are not gained at the expense of significant performance loss nor lattice size reduction.

The quality of the output of the HLS product strongly depends on the code structure. Although very high abstraction level and object-oriented C++ description are supported, hardware aware coding with simplified constructions, flattened hierarchy and appropriate usage of HLS directives allows the achievement of a better outcome. The preliminary test results obtained with Vivado HLS 2014.3 suggested that proper code shaping can lead to substantial reduction of resource utilization ($\sim$30%) with no performance loss.

In order to precisely compare HDL and HLS for ARUZ development purposes, independent descriptions of behaviorally identical FSM and PEs for simple variants of DLL-simulation were prepared using both approaches. Syntheses, targeting the same chips, with identical timing constraints, were performed using the same version of Xilinx toolset (Vivado/Vivado-HLS 2014.3). Resource utilization results are presented in Table 1 (main FSM) and Table 2 (single chip PE set).

Table 1. Resource utilization for main FSM.

Resorce Type	HLS		HDL
	ANSI C	SYSTEM C	VHDL
Look-up Tables (LUT)	104	92	67
LUT-FlipFlop pairs	106	92	67
Multiplexers	0	0	2

Table 2. Resource utilization for 128 PEs.

Resorce Type	HLS	HDL
	SYSTEM C	VHDL
Look-up Tables (LUT)	41,850	39,175
LUT-FlipFlop pairs	46,002	45,428
Block RAM	128	128
Multiplexers	210	36

As a result of independent structural design, the implementation may vary greatly in the way the same functionality is realized. Particularly, it can be observed that some selection-based operations are performed using either Look-Up Table (LUT) based logic or multiplexers. In the case of ARUZ FSM, the VHDL-based solution utilizes significantly fewer LUTs than HLS at the cost of a small number of multiplexers. Such a trade-off is favorable. Thus, HDL results can be considered more satisfactory.

It is worth noting that there is a non-negligible difference in resource utilization between the two HLS designs. Generic, high level description in standard ANSI C might be easier to develop; however, it does not offer much control over synthesis, even considering proper usage of HLS superset and directives. On the other hand, hardware-modelling-dedicated SystemC allows explicit structural description while remaining at a high level of abstraction. As the comparison suggests, it results in a more efficient design.

The FSM for the basic DLL-algorithm variant is quite simple and instantiated once, so its impact on the total resource utilization is marginal compared to the PEs. For further evaluation, complete functional sets of the 128-node networks, using three basic DLL mechanisms, were synthesized from SystemC and VHDL designs. The resources utilized by 128 PEs are presented in Table 2.

Although the total number of the LUT-based logic is similar, it needs to be stressed that some of the multiplexers present in the SystemC design are replaced by the LUT structures in the HDL solution. Even in that case, despite a significant increase in the multiplexer usage, the HLS approach still results in a higher look-up table utilization level.

Overall results confirm that manual HDL logic design produces more efficient output than HLS, even if differences are not significant. Considering that suboptimal resource allocation for PEs may substantially reduce molecular simulation lattice size, the HDL approach was chosen as better suited for ARUZ.

Among several known HDL languages, the two most popular ones are: Verilog and VHDL. They both have their advantages and disadvantages. Due to experience of developers, VHDL was chosen for the development of ARUZ.

4.2. Preprocessor

Due to a high number of mechanism combinations and the requirement for having as many nodes as possible, it was necessary to be able to add/remove mechanisms depending on the algorithm variant. Unfortunately, the VHDL and synthesis tools do not allow such optimization easily, while maintaining code clarity. The *generate* statement cannot be used inside a process block. The *if* statements with generic values make the code complex to debug a specific variant of the FSM. In a behavioral HDL simulation, such code consists of not only the required portions, but also the switched off mechanisms, which are optimized out at the synthesis stage.

The solution to this could be code preprocessing, which is native to Verilog, but not supported in VHDL selected for the development of ARUZ. To achieve such a functionality in VHDL, it is possible to use a C compiler preprocessor (e.g., from a well-established GCC package) or less popular dedicated tools, like VPP [9]. In any case, an extra step of code preprocessing is added to the process flow.

For ARUZ firmware development, the approach based on a GCC preprocessor was selected. Firstly, the complete description of a full set of mechanisms was formed, including the state machine,

enforcing the proper sequence of their execution was formed. The 144 states were divided into 17 state groups, where each state group corresponds to the specific algorithm mechanism or molecular simulation control actions (e.g., loading initialization data, unloading archive data). The order of the state groups (e.g., A → B → C) is always the same (e.g., A → C), even in the absence of the intermediate state group (i.e., B).

The pure VHDL code was then cut into sections with multiple preprocessor *#define* commands. The code in such a form is no longer synthesizeable. In order to make it VHDL compliant again, the code together with extra arguments has to be passed through the C pre-processor. These arguments indicate whose mechanisms should be included/excluded in/from the state machine. The output from the preprocessor is the application-specific truncated code, which can then be fed to the regular HDL synthesis/simulation process.

During code development, it is necessary to verify its correctness, which is done in four steps:

1. Preprocessor—finds and indicates syntax errors related to the *#define* arguments;
2. Synthesis tool—checks the syntax of the remaining code and some logical errors like too aggressive elimination of a portion of code;
3. HDL simulator—enables precise and concise debugging of the logical errors of mechanism/functionality dependencies;
4. Automatic tests—described later on.

Such elimination of portions of the code makes the synthesis process easier for the tool to execute and results in code with higher readability. The latter is especially useful during debugging at the functional verification stage, as the conditions for the presence of the state groups (mechanism dependencies) are usually not trivial.

4.3. Configuration Registers

The smallest, fully-scalable structure of ARUZ theoretically consists of a network of $3 \times 3 \times 3$ DBoards. Each DBoard configuration in such a network is unique (i.e., it represents different boundary conditions: central or terminating position along each axis). In order to increase the machine size in any direction, it is enough to copy the already-present board configuration. However, there are 27 location specific configurations. Combining this number with eight bitstreams (one for each DSlave) gives 216 bitstream files for each molecular simulation variant.

Most of the differences among these variants are caused by the local communication registers and transceivers. Therefore, the unification of these components was proposed. With a little extra resource (less than 1% of LUTs/FFs), those location specific components were provided with extra connectivity functionality and configuration registers. After the configuration of FPGA with a bitstream, a set of configuration bits is sent to the DSlave. Part of this extra configuration specifies if the DBoard is on the edge/side of the DBoard network and, if so, which position it occupies. This way, by the use of configuration registers, one of the 27 location-specific configurations can be achieved. Consequently, the synthesis task was simplified 27-fold and the number of bitstreams was reduced to a much more manageable eight files.

The configuration registers allow for switching on/off the boundary conditions for each board for the specific side. As it is explained further, such functionality effectively simplifies the smallest, fully-scalable ARUZ structure from $3 \times 3 \times 3$ to $2 \times 2 \times 2$ DBoards.

4.4. Dedicated Networks of PEs

As already mentioned, for different molecular simulation variants, various number of PEs can be instantiated in a single DSlave. This number is limited by the PE complexity, but, within this limit, it can be to some extent (as it is explained in [10,11]) adapted to the specific molecular simulation requirements. Furthermore, even for the same number of PEs, the topology of the interconnections can vary, according to the specification for the required molecular simulation space geometry (regular cube,

long pipe, thin layer, etc.). Such a flexibility disqualifies manual elaboration of the VHDL description of each desired interconnection topology. Therefore, for the purpose of ARUZ firmware development, the dedicated application (DLLDesigner) has been prepared. The role of this application is to analyze the molecular simulation assumptions and to generate—if feasible—the part of VHDL code responsible for instantiating and interconnecting as many PEs as required.

Despite its name, DLLDesigner can be used not only for the purpose of the implementation of the DLL algorithm, but also for all other computations exploiting the FCC lattice. The algorithm implemented in the DLLDesigner is based on a mathematical transformation of such lattice of nodes into the network of PEs representing these nodes and mapped on ARUZ hardware resources. The details of this transformation are out of the scope of this paper (they are precisely described in [7,8,10–14]), but it is worth mentioning that:

- The transformation organizes PEs inside DSlave using odd and even layers—the number and the dimension of layers is parameterizable, the order of layers (odd-even-odd-etc. or even-odd-even-etc.) depends on the DSlave (A–H), but the total number of PEs in each DSlave in the entire machine is the same.
- Mapping the nodes on the PEs takes into account periodic boundary conditions—the nodes are distributed in a way ensuring that the ones located in the space-distant lattice boundaries are mapped on PEs located in relatively close proximity. This way, periodic boundary conditions can be achieved without the need for long cables to connect the PEs representing the boundary nodes.

The input for the DLLDesigner application is a set of parameters determining:

- The dimensions of a network of PEs in each DSlave;
- The number of panels, rows, and columns of DBoards to be used in a molecular simulation.

Using these parameters DLLDesigner:

1. Evaluates the dimensions of the entire FFC lattice to be simulated;
2. Creates internal model of such lattice;
3. Interconnects all lattice nodes (virtually) by storing (in each model of node) the pointers to the neighbors (actually, to the neighboring models);
4. Evaluates the location of the PE for each node (by means of mathematical transformation mentioned above);
5. Generates VHDL instantiations of as many PEs as required (independently for each DSlave (A–H));
6. Using the stored pointers to the neighbors generates a VHDL description of interconnections among PEs located in a single DSlave (independently for each DSlave);
7. Evaluates the number of bits which have to be transferred among DSlaves in each communication phase (in each direction, independently for each DSlave);
8. Using the maximum numbers of bits to be transferred parametrizes the width of local communication transceivers (by setting appropriate constants in the dedicated VHDL package, independently for each DSlave).

As a result of these operations, a set of eight VHDL descriptions of the PE networks (to be implemented in each DSlave) is obtained. Each description is dedicated to a given DSlave; thus, it is initially optimized. In addition, the interfaces within the networks are optimized since the widths of the transceivers are statically parametrized before the synthesis.

The automatically generated code—even 166 k lines (Table 3)—constitutes up to 81% of the entire VHDL description (only 38 k lines are written manually). In addition to the description, the detailed report of the created interconnections is generated. It can be used during the debugging phase when the data transfer must be checked in detail.

Obviously, DLLDesigner generates VHDL descriptions but does not verify if they can be implemented in the DSlaves in terms of available resources and demanded clock frequency. For such a verification, a synthesis process must be performed. Thus, the preparation of a new bitstream for the highest possible number of PEs in a single DSlave (for a given molecular simulation variant) may require an iterative process, in which the code generation and the synthesis must be repeated several times (for different dimensions of networks of PEs to be implemented in a single chip). Fortunately, the synthesis process can be distributed across several computers to be executed in parallel and the code generation is relatively fast (Table 3). Furthermore, a reduced number of panels, rows, and columns ($3 \times 3 \times 3$) of DBoards can be used for a feasibility study. This is because during the code generation for the network of $3 \times 3 \times 3$ DBoards the inter-board communication and the boundary conditions in each direction are considered, formed, and multiplexed (using configuration registers). The codes generated in this way are the same as for the network of $18 \times 12 \times 12$ DBoards. As a result, using the network of $3 \times 3 \times 3$ DBoards (scalable to the larger networks), various configurations of networks of PEs can be checked in a reasonable time and without the need of manual re-coding.

Table 3. DLLDesigner statistics.

DBoards	Nodes	PC Resources	VHDL Code [*lines*]	Report [*MB*]
2592 ($18 \times 12 \times 12$)	1,492,992 (72/DSlave)	2 min 58 s RAM: 2.2 GB	60,968 (7621/DSlave)	412
	5,308,416 (256/DSlave)	21 min 10 s RAM: 7.6 GB	165,928 (20,741/DSlave)	1332
27 ($3 \times 3 \times 3$)	15,552 (72/DSlave)	1.8 s RAM: 28 MB	60,968 (7621/DSlave)	4.3
	55,296 (256/DSlave)	13 s RAM: 85 MB	165,928 (20,741/DSlave)	13.8

Sample results of such a feasibility study for both generic and application-specific codes for several variants of molecular simulations are presented in Table 4. They demonstrate the resource usage for various:

- Sizes of PE networks (number of nodes per single DSlave);
- Mechanisms;
- Number of bits dedicated to encode the type of molecules.

Table 4. Resource utilization for different molecular simulation variants (results marked with '*' correspond to unimplementable solutions—too many resources required or routing congestion problems encountered).

Solution	Type Bits	Mechanisms	Nodes per DSlave	LUTs [%]	FFs [%]	BRAMs [%]	DSPs [%]
Generic	5	All (13)	72	88	34	79	19
Optimized	2	All (13)	72	87	34	20	19
		2 basic	144	75	35	-	-
			192 *	99 *	45 *	-	-
			256 *	124 *	58 *	-	-
		1 basic	192	93	43	-	-
			256 *	121 *	55 *	-	-

The analysis of the values presented in Table 4 allows the type of FPGA resource limiting the implementation of the PE network for different molecular simulation settings to be easily determined.

Even though the majority of the DSlave resources is occupied by modules adapted to a specific molecular simulation variant (Table 5), some components are simulation independent, e.g., control/status modules and local communication transceivers. The former consist of control registers, supply voltage and temperature monitors, etc. Transceivers are responsible for exchanging data among FPGAs. They use low-latency, ARUZ-dedicated protocol, and they also perform permanent connection quality monitoring in the background. The usage of transceivers depends on the boundary conditions (their position in the entire system), and it is manipulated by means of the control registers. Transceivers are algorithm independent, and their number is fixed as it is determined by the topology of interconnections among the DBoards. However, the width of transceivers depends on lattice dimensions. It is elaborated by a DLLDesigner, and it reflects the maximum width of data being sent in a given direction. To conclude, the transceivers are in fact algorithm-independent but lattice-optimized. This fact can be seen in Table 5—the utilization of algorithm-independent modules varies with the change of the lattice parameters. The more bits to exchange (more PEs in FPGA), the more resources for transceivers are required. The algorithm complexity (the number of mechanisms) practically do not influence the utilization of algorithm-independent modules. Negligible differences seen in Table 5 are caused by global optimization.

Table 5. Resource utilization of algorithm-dependent and algorithm-independent modules of the design (results obtained for 2 bits dedicated to encode the type of molecules).

Mechanisms	Nodes per DSlave	Utilization of Modules			
		Algorithm-Dependent		Algorithm-Independent	
		LUTs [%]	FFs [%]	LUTs [%]	FFs [%]
All (13)	72	79.2	24.0	6.7	3.1
1 basic	72	23.1	9.2	7.4	3.1
	192	60.8	24.5	9.7	4.3

4.5. Parallel Synthesis and Binary File Tracking

Generation of many configuration sets (1 set = 8 files) requires a parallel approach to the synthesis process which allows distributed synthesis with sufficient tracking information in order to keep track of files, either on a hard drive or in FPGA. To address those challenges, an appropriate synthesis script has been written.

As an input for this script, the user specifies the number of workstations (together with their domains/IP addresses) in the form of a text file and then executes the script on one of the workstations—the origin, which can but does not have to participate in the synthesis. The script automatically and evenly distributes tasks across multiple workstations and starts each synthesis. The user in the meantime fills in the configuration set description. This is particularly useful to keep track of the changes at the development stage, when there are many volatile feature tests. Next, the user waits for the parallel synthesis process to complete. The whole process terminates after the last synthesis finishes and all files are downloaded back to the origin. As a result, the user obtains a ready to use set of configuration files and a log file containing the provided description, some other tracking information and reports.

A multitude of configuration sets requires methods allowing identification of files. This is achieved on two levels. One allows identification of the bitstream file on a hard drive. The other identifies the bitstream that is inside the FGPA.

The former is achieved by the use of MD5 hash check sums, which are calculated after bitstream syntheses and are added to the log file. Those checks sums are also used as the pre-check of machine configuration when starting ARUZ operation.

The latter is implemented with the use of the read-only registers. At the start of the synthesis script execution, synthesis ID numbers are generated, stored in the dedicated VHDL package, and saved in the log file. For each chip the name of the chip, the date of the synthesis start and the login of the developer is coded in a 32-bit value.

During the synthesis process, such a synthesis ID is mapped into a read-only register. It can be read from the operating device in order to check the configuration details.

4.6. Debugging

The development of complex systems requires an advanced and thorough approach to their verification. For ARUZ development, several techniques of debugging are applied:

- Comparison of output files. At the very beginning of the verification process, the final results obtained from the reduced ARUZ model (so called nanoARUZ, composed of $2 \times 2 \times 2$ array of DBoards) are compared with the bit-accurate results obtained from ARUZ emulator (part of DLLDesigner), in which the functional ARUZ model is implemented.
- Comparison of the HDL simulations with the C++ emulations. If any differences between hardware-based molecular simulation and its emulation are detected during the verification of the final results (output files), intermediate results (values of relevant signals) are compared. For that purpose HDL simulations are performed and compared to the results of emulations (obtained by means of emulator executed in the debug mode with appropriate breakpoints set). Such analysis allows correcting VHDL codes (firmware) as well as C++ codes (emulator).
- Dedicated logic analyzer. If the HDL simulations fit the model implemented in the emulator, but hardware-based verification demonstrated some differences, the real signals in the FPGAs are analyzed. For that purpose, a dedicated logic analyzer has been prepared (described in VHDL). It is able to detect some events and latch the values of relevant signals inside the chip. Such values can then be downloaded to a PC for the exhaustive analysis. Dedicated logic analyzer modules are used instead of the standard ones in order to allow the acquisition and analysis of the results from all ARUZ chips simultaneously. This technique proves to be invaluable due to the system complexity.
- Status registers. Once correct behavior of the FPGAs is achieved, the potential run-time errors (caused by e.g., chip damages or interconnection problems) are detected by means of dedicated status registers, which are read (periodically or on demand) during ARUZ operation (transparently).
- Initial readback. In order to verify if the chips are properly configured and initialized, the initialization data are downloaded before the first molecular simulation step and compared with the data used for the initial upload. In case of any mismatch, the molecular simulation is stopped, and the hardware error is reported.

4.7. Automatic Verification

Due to the multitude of molecular simulation variants, a script has been written to verify the correctness of each intended configuration. A set of configurations (written manually or generated at random for various network sizes, shapes, bitfield widths, combination of mechanisms, etc.) can be fed into the script, which processes them one by one. The processing involves:

- generation of the code
- synthesis
- nanoARUZ programming

- initialization data generation
- running molecular simulation in emulator and nanoARUZ with the generated data
- output data comparison
- report generation

The report contains the status (successful or unsuccessful) of each checked configuration. The failed configurations should then be investigated one by one in order to determine and fix the bugs. Afterwards, the script with the same set of configurations has to be re-run to confirm that the fixes do not introduce new bugs. Considering that both machines are bitstream compatible, the automatic verification of nanoARUZ verifies the bitstreams for ARUZ itself as well.

5. Result Discussion

Using HLS allows for raising productivity at a certain expense of efficiency. For ARUZ, it was not a suitable solution since the overall monetary cost of a lower efficiency machine is higher in the long run (longer execution time/smaller molecular simulation space). Because of this, when designing the system, priority was given to efficiency. Shorter execution time means running more molecular simulations and consuming less power, which translates into lower cost per simulation. Greater node density translates into the ability of solving larger problems, which improves quality. Consequently, unlike in HLS methodology, efficient building blocks were developed first. Then, they were interconnected and configured by the dedicated tools, which significantly boosted the productivity and flexibility of the solution.

Using the generic set of bitstream files for all of the molecular simulation variants is convenient from the maintenance point of view. However, it is wasteful from the perspective of FPGA resources and machine efficiency. By using the dedicated bitstreams for each molecular simulation variant, FPGA resource usage can be lowered allowing for implementing larger networks of PEs. When compared with the generic solution, such an optimization results in approximately 2.7 (192/72, see Table 4) times greater number of implementable PEs for one basic mechanism.

As indicated in Table 3, the PE network generation time can be shortened by formation of the smaller DBoard network ($3 \times 3 \times 3$), and then scaling it up, instead of running the process for the $18 \times 12 \times 12$ DBoard network. Such an approach speeds up the code generation by two orders of magnitude (2 min 58 s/1.8 s, 21 min 10 s/13 s).

The implementation of the configuration registers described above improved the runtime of the synthesis process by a factor of 27 (see Section 4.3). This way, assuming the same synthesis platform, the overall synthesis runtime for all of the required molecular simulation variants was reduced from months to single days.

All of the aforementioned factors improving efficiency first and then productivity, combined with parallel synthesis, dedicated debugging mechanisms, and automatic verification techniques, allowed for delivering the whole, efficient system in the span of 14-month-long project. Now, with the proposed methodology, elaborated CAD tools (preprocessor, DLLDesigner), and reusable HDL codes (e.g., inter-chip interfaces), implementation time for new algorithms can be significantly reduced.

6. Related Work

Contemporary high performance computing systems are predominantly constructed using general purpose processors, often supported by additional hardware accelerators [15,16]. In particular, heterogeneous systems combining CPUs with graphics processors have become extremely popular in recent years [15,16]. Other solutions expand processors with reconfigurable logic. Although they are nowhere near as popular as GPUs, several supercomputers with FPGA-based accelerators (Microsoft Catapult, Amazon F1, Padebox Noctua) [17–19] have been presented recently. HPC systems based solely on multi-FPGA platforms are far less common and usually used for either advanced ASIC simulations [20,21] or very specific scientific computations [22–24]. There are a few massively parallel

reconfigurable machines for molecular physics [22,23,25,26], but, to the best of our knowledge, none of them on such a large scale as ARUZ.

The attitude towards firmware development for FPGA-accelerated machines varies depending on their architecture and target application. Unlike multi-processor computations, there are no standardized methodologies, interfaces, or tools for multi-FPGA based platforms. Although there were attempts to introduce a multi-FPGA equivalent of OpenMP [27], no uniform approach has been established yet. Considering the variety of applications and architectures (from FPGA-dominated clusters to servers based on hybrid CPU-FPGA chips) as well as diverse temporal granularity, differences in firmware development methodology are understandable.

In most heterogeneous HPC systems, the entire control-flow is handled by standard CPUs while additional hardware (such as FPGAs) is used only to accelerate certain computations that may benefit from its architecture [20,28]. Implementing only off-the-shelf IPs or coarse DSP soft-cores for certain computations is not uncommon [18]. If application-specific firmware is being developed for FPGAs, stress is usually put on uniformity of software description—to improve productivity and maintenance. Moreover, many of such platforms are designed as general purpose machines, in which temporal granularity of FPGA configuration is much finer than in application-specific systems; therefore, time-to-market might be favored over performance. Hence, high-level languages (HLL) and heterogeneity-oriented frameworks like OpenCL are preferred [17,18,29].

A high-level synthesis approach is generally considered more convenient, simpler, and less-time consuming compared with RTL-level hardware description [30,31]. The ease-of-use is especially important when firmware is supposed to be modified by the HPC system end-users—usually with a science or IT background rather than an electronics and system design one [29]. In FPGA-based machines, not everything can be handled efficiently at a high-level, so some parts of the system are developed using HDL nonetheless—employing a so called hybrid approach [20]. A high-level approach is convenient, but, in spite of constant advancement of HLS tools, it induces area and performance overhead compared with HDL solutions [31,32]. Despite addressing the lower efficiency problem of HLS in their work, many system designers still accept this trade-off to shorten time-to-market [31].

Using an HDL approach allows for efficiently exploiting the full capabilities of hardware resources; thus, it is still preferred wherever no overhead is tolerated [20,31]. It is typically applied to all low-level critical sub-systems and also recommended for IP-core development [18,20]. This approach, however, offers few advantages (code re-usability, limited language in-built code generation and generalization) when it comes to increasing productivity [31,33]. To address that problem, advanced automatic generation of HDL code is highly desirable. Even though some basic mechanisms have been provided by both vendors and scholars, they are not as well-established as in the case of HLL [33].

Many studies regarding firmware development methodologies for multi-FPGA HPC systems focus on partitioning [21,34,35]. Whenever a task or compute kernel is too complex to be implemented utilizing the resources of a single chip, proper distribution of computation across multiple devices is essential. However, task partitioning is of little significance in molecular simulations, as they typically involve an immense number of simple processing elements, performing parallel computations of range-limited interactions corresponding to individual objects or subsets of molecular simulation space [22,23,25,26].

7. Conclusions

This paper presents the methodology used for the automated generation of individual, optimized bitstreams for ARUZ. Bearing in mind the related work in this domain, the discussed solution is unique and well tailored to all efficiency-oriented, massively parallel systems, composed of an immense number of simple processing elements executing the same tasks for different data.

The methodology focuses on the DSlave chips as the one-time developed configuration of the DMaster is fixed, while functionality of the DSlaves varies between individual molecular simulations.

Since the configuration of all DSlaves takes only about 2 min while the time required for molecular simulation is counted in dozens of hours, it is advisable to use individual, optimized bitstreams.

Molecular simulation dedicated DSlave configuration is obtained during the multi-stage hardware description generation process (Figure 3). The starting point of this process constitutes the manual VHDL description of all possible mechanisms of a single PE. Then, such a description is preprocessed in order to select only those mechanisms that are required for a given molecular simulation variant. Finally, such reduced description is automatically multiplied, as many times as required, and interconnected by the DLLDesigner—dedicated application being able to generate ready-to-use, optimized VHDL code. This application and the above-mentioned preprocessor are the software tools developed in order to achieve a low-level ARUZ optimization in an automated manner. The base of this automation is the manual PE description developed with careful considerations regarding resource utilization and timing constraints. This is because each PE is instantiated multiple times, so savings in its resources have a direct impact on a maximum number of PEs, which can be implemented.

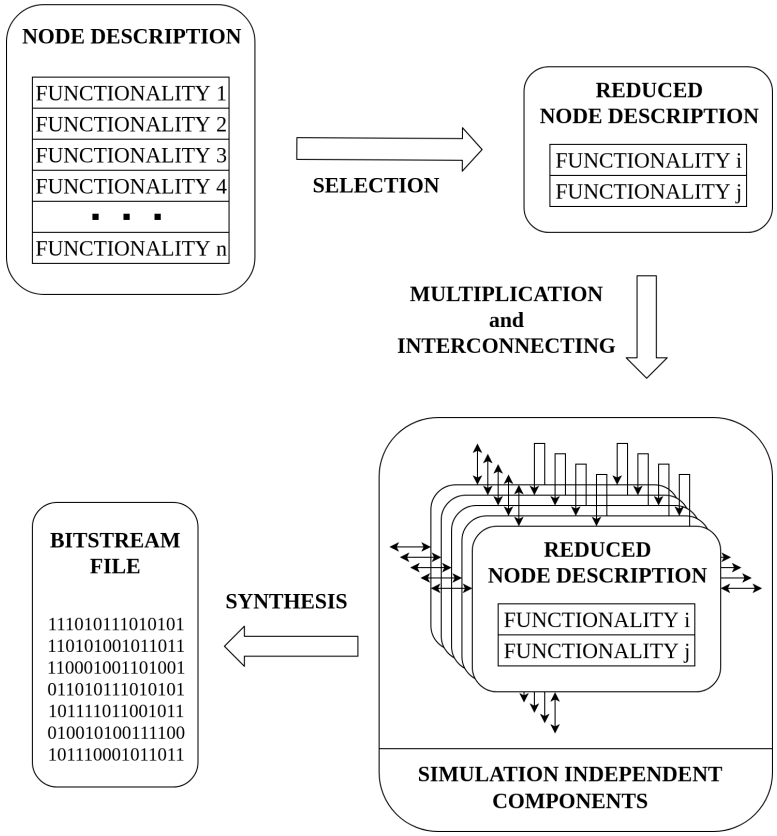

Figure 3. Preparation of the DSlave configuration.

Apart from the algorithm-dependent PEs, the configuration of each DSlave also contains some algorithm-independent components such as control/status modules and communication resources. Although they do not depend on algorithm variants (in other words, they can be rescued for different PE definitions), some of them are optimized for a particular system configuration. For example, the internal and inter-chip interconnections among PEs are optimized for the selected lattice dimensions, the former—by dedicated routing description, the latter—by settings of parametrizable transceivers—both generated automatically by the DLLDesigner.

To conclude, ARUZ efficiency is achieved by means of several optimization efforts taken at various levels of the (highly automated) development process:

- Language selection—there is no acceptance for HLS overhead;
- Dedicated components—PEs and their peripherals are optimized for the algorithm requirements;

- Dedicated protocols—data exchange reflects specific payload characteristics;
- Code preprocessing—only the relevant code is synthesized;
- Dedicated number of PEs—only the PEs which are required are instantiated;
- Dedicated interconnections within a DSlave—routing is generated for the application-specific network of PEs;
- Parametrized interconnections among DSlaves—widths of the transceivers are adapted to maximum data size.

The applied combination of methods addresses all of the challenges and meets productivity expectations. As a result, ARUZ is a high-performance computing machine, which is efficient not only in terms of the speed but also in terms of power consumption [1].

Author Contributions: Conceptualization, R.K., K.R., and J.J.; methodology, K.R. and R.K.; software, R.K., K.R., and Z.M.; validation, K.R., Z.M., and R.K.; investigation, Z.M., K.R., R.K., and J.J.; writing, R.K., Z.M., and K.R.; supervision, R.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by European Union funds in the framework of the Innovative Economy Operational Program (through Polish Agency for Enterprise Development, grant: POIG.05.03.00-00-007/10-00), by the equity capital of the Lodz region (city and voivodeship) and by the Polish National Science Centre (grants: 2014/14/A/ST5/00204 and 2017/25/B/ST5/01970).

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Kiełbik, R.; Hałagan, K.; Zatorski, W.; Jung, J.; Ulański, J.; Napieralski, A.; Rudnicki, K.; Amrozik, P.; Jabłoński, G.; Stożek, D.; et al. ARUZ—Large-scale, massively parallel FPGA-based analyzer of real complex systems. *Comput. Phys. Commun.* **2018**. [CrossRef]
2. Pakuła, T. Collective dynamics in simple supercooled and polymer liquids. *J. Mol. Liq.* **2000**, *86*, 109–121. [CrossRef]
3. Pakuła, T. Simulations on the Completely Occupied Lattice. In *Simulation Methods for Polymers*; CRC Press: Boca Raton, FL, USA, 2004; Chapter 5, pp. 147–176. [CrossRef]
4. Jabłoński, G.; Kupis, J. Performance estimation of Lattice Boltzmann method implementation in ARUZ. In Proceedings of the 2017 MIXDES—24th International Conference "Mixed Design of Integrated Circuits and Systems", Bydgoszcz, Poland, 22–24 June 2017; pp. 308–313. [CrossRef]
5. Jabłoński, G.; Kupis, J. The Application of High Level Synthesis for Implementation of Lattice Boltzmann Method in ARUZ. *Int. J. Microelectron. Comput. Sci.* **2017**, *8*, 36–42.
6. Jabłoński, G.; Kupis, J. Performance Optimization of Implementation of Lattice Boltzmann Method in ARUZ. In Proceedings of the 2018 MIXDES—25th International Conference "Mixed Design of Integrated Circuits and Systems", Gdynia, Poland, 21–23 June 2018.
7. Jung, J.; Polanowski, P.; Kiełbik, R.; Hałagan, K.; Zatorski, W.; Ulański, J.; Napieralski, A.; Pakuła, T. Panel z Układami Elektronicznymi i Zestaw Paneli. Patent RP PAT.223795, 1 March 2016.
8. Jung, J.; Polanowski, P.; Kiełbik, R.; Hałagan, K.; Zatorski, W.; Ulański, J.; Napieralski, A.; Pakuła, T. A Panel with Electronic Circuits and a Set of Panels. European Patent EP3079071B1, 1 August 2018.
9. VHDL Preprocessor Project Home Page. Available online: http://vhdlpp.sourceforge.net/ (accessed on 20 June 2019).
10. Jung, J.; Polanowski, P.; Kiełbik, R.; Hałagan, K.; Zatorski, W.; Ulański, J.; Napieralski, A.; Pakuła, T. A Parallel Machine Having Operational Cells Located at Nodes of a Face Centered Lattice. European Patent Application EP3079073A1, 12 October 2016.
11. Jung, J.; Polanowski, P.; Kiełbik, R.; Hałagan, K.; Zatorski, W.; Ulański, J.; Napieralski, A.; Pakuła, T. Maszyna Równoległa z Komórkami Operacyjnymi Umieszczonymi w Węzłach Sieci Powierzchniowo Centrowanej. Patent RP PAT.227249, 7 June 2017.

12. Jung, J.; Polanowski, P.; Kiełbik, R.; Hałagan, K.; Zatorski, W.; Ulański, J.; Napieralski, A.; Pakuła, T. A Parallel Machine with Reduced Number of Connections between Logical Circuits. European Patent Application EP3079072A1, 12 October 2016.
13. Jung, J.; Polanowski, P.; Kiełbik, R.; Hałagan, K.; Zatorski, W.; Ulański, J.; Napieralski, A.; Pakuła, T. Maszyna Równoległa ze Zredukowaną Liczbą Połączeń Pomiędzy Układami Logicznymi. Patent RP PAT.227250, 7 June 2017.
14. Jung, J.; Polanowski, P.; Kiełbik, R.; Hałagan, K.; Zatorski, W.; Ulański, J.; Napieralski, A.; Pakuła, T. System of Electronic Modules Having A Redundant Configuration. European Patent EP3079066B1, 23 August 2017.
15. TOP500 Supercomputers Site. Available online: https://www.top500.org/ (accessed on 20 June 2019).
16. Véstias, M.; Neto, H. Trends of CPU, GPU and FPGA for high-performance computing. In Proceedings of the 2014 24th International Conference on Field Programmable Logic and Applications (FPL), Munich, Germany, 2–4 September 2014; pp. 1–6. [CrossRef]
17. Caulfield, A.; Chung, E.; Putnam, A.; Angepat, H.; Fowers, J.; Haselman, M.; Heil, S.; Humphrey, M.; Kaur, P.; Kim, J.Y.; et al. A Cloud-Scale Acceleration Architecture. In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, Taipei, Taiwan, 15–19 October 2016; IEEE Computer Society: Washington, DC, USA, 2016.
18. Amazon Web Services EC2 Instances F1. Available online: https://aws.amazon.com/ec2/instance-types/f1/ (accessed on 20 June 2019).
19. Paderborn Universtity, Paderborn Center for Parallel Computing, Noctua Webpage. Available online: https://pc2.uni-paderborn.de/hpc-services/available-systems/noctua/ (accessed on 20 June 2019).
20. Morris, G.R.; Prasanna, V.K.; Anderson, R.D. A Hybrid Approach for Mapping Conjugate Gradient onto an FPGA-Augmented Reconfigurable Supercomputer. In Proceedings of the 2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, USA, 24–26 April 2006; pp. 3–12. [CrossRef]
21. Swaminathan, S.P.; Lin, P.K.; Khatri, S.P. Timing aware partitioning for multi-FPGA based logic simulation using top-down selective hierarchy flattening. In Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD), Montreal, QC, Canada, 30 September–3 October 2012; pp. 153–158. [CrossRef]
22. Kasap, S.; Benkrid, K. A high performance implementation for Molecular Dynamics simulations on a FPGA supercomputer. In Proceedings of the 2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), San Diego, CA, USA, 6–9 June 2011; pp. 375–382. [CrossRef]
23. Kasap, S.; Benkrid, K. Parallel Processor Design and Implementation for Molecular Dynamics Simulations on a FPGA Parallel Computer. *J. Comput.* **2012**, *7*, 1312–1328. [CrossRef]
24. *TimeLogic® DeCypher® System*; Active Motif Inc.: Carlsbad, CA, USA, 2019.
25. Shaw, D.E.; Deneroff, M.M.; Dror, R.O.; Kuskin, J.S.; Larson, R.H.; Salmon, J.K.; Young, C.; Batson, B.; Bowers, K.J.; Chao, J.C.; et al. Anton, a Special-Purpose Machine for Molecular Dynamics Simulation. *Commun. ACM* **2008**, *51*, 91–97. [CrossRef]
26. Shaw, D.E.; Grossman, J.P.; Bank, J.A.; Batson, B.; Butts, J.A.; Chao, J.C.; Deneroff, M.M.; Dror, R.O.; Even, A.; Fenton, C.H.; et al. Anton 2: Raising the Bar for Performance and Programmability in a Special-Purpose Molecular Dynamics Supercomputer. In Proceedings of the SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16–21 November 2014; pp. 41–53. [CrossRef]
27. Baxter, R.; Booth, S.; Bull, M.; Cawood, G.; Perry, J.; Parsons, M.; Simpson, A.; Trew, A.; McCormick, A.; Smart, G.; et al. The FPGA High-Performance Computing Alliance Parallel Toolkit. In Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007), Edinburgh, UK, 5–8 August 2007; pp. 301–310. [CrossRef]
28. Baxter, R.; Booth, S.; Bull, M.; Cawood, G.; Perry, J.; Parsons, M.; Simpson, A.; Trew, A.; McCormick, A.; Smart, G.; et al. Maxwell—A 64 FPGA Supercomputer. In Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007), Edinburgh, UK, 5–8 August 2007; pp. 287–294. [CrossRef]
29. Nabi, S.W.; Vanderbauwhede, W. A Fast and Accurate Cost Model for FPGA Design Space Exploration in HPC Applications. In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, USA, 23–27 May 2016; pp. 114–123. [CrossRef]

30. Nane, R.; Sima, V.; Pilato, C.; Choi, J.; Fort, B.; Canis, A.; Chen, Y.T.; Hsiao, H.; Brown, S.; Ferrandi, F.; et al. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2016**, *35*, 1591–1604. [CrossRef]
31. Wang, G.; Lam, H.; George, A.; Edwards, G. Performance and productivity evaluation of hybrid-threading HLS versus HDLs. In Proceedings of the 2015 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 15–17 September 2015; pp. 1–7. [CrossRef]
32. Pelcat, M.; Bourrasset, C.; Maggiani, L.; Berry, F. Design productivity of a high level synthesis compiler versus HDL. In Proceedings of the 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), Samos Island, Greece, 17–21 July 2016; pp. 140–147. [CrossRef]
33. Pohl, C.; Paiz, C.; Porrmann, M. vMAGIC—Automatic Code Generation for VHDL. *Int. J. Reconf. Comput.* **2009**. [CrossRef]
34. Kerkiz, N.; Elchouemi, A.; Bouldin, D. Multi-FPGA Partitioning Method Based on Topological Levelization. *J. Electr. Comput. Eng.* **2010**, *2010*. [CrossRef]
35. Roy-Neogi, K.; Sechen, C. Multiple FPGA Partitioning with Performance Optimization. In Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 12–14 February 1995; pp. 146–152. [CrossRef]