DOI 10.1007/s10559-015-9706-0
Cybernetics and Systems Analysis, Vol. 51, No. 1, January, 2015

SOFTWARE–HARDWARE SYSTEMS

SOFTWARE TOOLS FOR AUTOMATION OF PARALLEL PROGRAMMING ON THE BASIS OF ALGEBRA OF ALGORITHMS

F. I. Andon,[a†] **A. E. Doroshenko,**[a‡] **A. G. Beketov,**[a††] **V. A. Iovchev,**[a†‡]
and E. A. Yatsenko[a‡‡]

UDC 681.3

Abstract. *The development of the algebra-algorithmic methodology and tools for automated design and generation of programs for graphics processing units is proposed. A particular feature of the proposed approach is the use of high-level specifications that are close to natural-language specifications and also the application of a method that ensures the syntactical correctness of algorithms and programs being designed. The approach was implemented in a toolkit destined for interactively designing algorithm schemes and generating programs. The use of this toolkit is illustrated by the development of a parallel program in the field of meteorology.*

Keywords: *algebra of algorithms, automated program design and generation, graphics processing unit (GPU), parallel programming, algorithm scheme.*

INTRODUCTION

As a result of the widespread use of multi-core processors in modern computer systems, a topical problem is the creation of special means for the development and reengineering of parallel software that would be used at all stages of the software life cycle.

The theory, methodology, and software tools for the automated design of parallel programs on the basis of algebra of algorithmics [1] are developed at the Institute of Software Systems of the National Academy of Sciences of Ukraine for a long period. The mentioned algebra is a modern direction of computer science going back to fundamental works of academician V. M. Glushkov on the theory of systems of algorithmic algebras (SAAs). In this theory, the subject of inquiry consists of high-level models of algorithms represented in the form of schemes. In [1–9], formal means are proposed for developing efficient parallel programs for multi-core central processors, GPUs, and systems with distributed memory. These means are based on the use of SAAs [1, 2], ontologies [3], algebraic dynamic models and discrete dynamic systems [4, 5], the method of parametric controlled generation of algorithm schemes [6], and also the technique of term rewriting [7–9]. Taking into account the developed theory and methodology, an integrated toolkit for designing and synthesizing programs (IDS) [2, 6–8] and also the system TermWare for symbolic computations [9] are created.

The IDS system is based on the representation of specifications of algorithms in SAAs and generates sequential and parallel programs in the programming languages Java and C++. The TermWare system based on the paradigm of rule rewriting supplements the capabilities of the IDS system and is used for the automation of formal transformations of algorithms with a view to optimizing their performance according to given criteria (memory, speed, equipment loading, etc.). A new IDS version is also developed, namely, an on-line interactive constructor of syntactically correct programs (OISD).

[a]Institute of Software Systems, National Academy of Sciences of Ukraine, Kiev, Ukraine, [†]*andon@isofts.kiev.ua*; [‡]*anatoliy.doroshenko@gmail.com*; [††]*beketov.oleksii@gmail.com*; [†‡]*iovchev.v@gmail.com*; [‡‡]*oayat@ukr.net*. Translated from Kibernetika i Sistemnyi Analiz, No. 1, pp. 162–170, January–February, 2015. Original article submitted September 12, 2014.

 1060-0396/15/5101-0142 ©2015 Springer Science+Business Media New York

This article proposes a further development of the algebra-algorithmic methodology and software tools for automation of designing parallel programs for graphics processing units (GPUs) that make it possible to considerably increase the performance of computations in comparison with usual processors. The present article also considers the problem of automated design and generation of parallel programs for the NVIDIA CUDA platform [10] with the use of SAA means and the OISD system.

The proposed approach is similar to that described in works on algebraic programming [11] and synthesis of programs based on specifications [12, 13] and also on the problem of generation of codes for GPUs. The existing approaches to synthesizing programs for GPUs are based, in particular, on annotations describing properties of data structures and code domains [14], Kahn process networks [15], compiler directives [16], data flow graphs [17], and high level abstractions of data structures, problems, and communication operators [18].

A distinctive feature of the approach considered below consists of using specifications of the Glushkov algebra that are represented in natural-linguistic form facilitating the understanding of algorithms and achievement of the required quality of programs. An advantage of the developed tools is the application of the method for interactively constructing syntactically correct programs that excludes the possibility of syntactic errors in the course of designing schemes.

The application of this approach is illustrated by an example of designing a parallel program in the field of meteorology.

FORMALIZED DESIGN OF PARALLEL PROGRAMS USING ALGEBRA OF ALGORITHMS AND REWRITING RULES

The proposed approach to the design of parallel programs is based on the apparatus of SAAs and their modifications [1]. Modified SAAs (SAA-Ms) are destined for the formalization of processes of multiple job processing that arise in designing software in multiprocessor systems. The described SAA-Ms are underlain by a two-sorted algebra $<U, B; \Omega>$, where U is a set of operators, B is a set of logical conditions (predicates), and Ω is the signature of the operations. Operators are mappings of an information set (IS) into itself, logical conditions are mappings of the IS into the set of values of the three-valued logic $E_3 = \{0, 1, \eta\}$, where 0, 1, and η mean false, truth, and indefiniteness, respectively. The signature $\Omega = \Omega_1 \cup \Omega_2$ consists of a system Ω_1 of logical operations assuming values in the set B and a system Ω_2 of operations assuming values in the set of operators U. Operators and predicates can be basic or compound. The operations belonging to the signature Ω are considered below.

The algorithmic language CAA/1 [1] is based on SAA-Ms and is destined for the multilevel structural design and documentation of sequential and parallel algorithms and programs. Its advantage is the possibility of describing algorithms in the natural linguistic user-friendly form which facilitates the achievement of the required quality of programs. Representations of operators in the CAA/1 language are called SAA schemes.

Compound predicates of the CAA/1 language are constructed from basic ones with the help of the following logical SAA-M operations:

— disjunction *predicate*1 OR *predicate* 2;

— conjunction *predicate*1 AND *predicate*2;

— negation NOT *predicate*.

Compound operators are constructed from elementary operators with the help of the following basic operations:

— sequential execution of operators (composition) *operator*1; *operator*2;

— branch operator IF (*predicate*) THEN *operator*1 ELSE *operator*2 END IF;

— loop operator WHILE (*predicate*) *operator* END OF LOOP.

Specifications of additional SAA-M operations destined for the formalization of multithread computations on multi-core central processors are given in [2, 7, 8].

New operations oriented to the design of parallel programs for the NVIDIA CUDA platform [10] are also added to the signature Ω. Note that the CUDA technology is a hardware-software complex allowing one to develop programs for GPUs. This complex is a computing device, which is used as a coprocessor of a central processor, has own memory, and simultaneously processes some number of threads. A kernel is understood to be a GPU function executed by threads. In CUDA, a programming model presumes the grouping of threads into blocks. Each block is an array of threads interacting with the help of shared memory and synchronization points. Blocks, in turn, are united into a grid of blocks.

New constructions of SAA-Ms for designing programs for GPUs are as follows:

— definition of a kernel function

$$\text{KERNEL } fname(param_list) = function_body,$$

where *fname* is a function name, *param_list* is a list of formal parameters, *function_body* is a function implementation represented in an SAA;

— operation of calling a kernel function

$$fname(N_b, N_{th}, arg_list),$$

where N_b is the number of blocks in the grid of blocks, N_{th} is the number of threads in each block, and *arg_list* is the list of actual parameters of the function;

— the following operators that assume values of global (unique) and local (within a block) indices of a thread and also assign the assumed value to an integer variable i:

$$\text{Get global index of the thread}(i),$$

$$\text{Get local index of the thread}(i);$$

— a synchronizer, i.e., the operation waiting for the completion of computations in all threads

Synchronizer (all threads have completed);

— operations of allocation and deallocation of graphic processor memory for some variable

$$\text{Allocate GPU memory}(var_name,\ size),$$

$$\text{Deallocate GPU memory}(var_name),$$

where *var_name* is a variable name and *size* is the necessary memory capacity (in bytes);

— operation of copying data from the memory of the central processor to the GPU memory and vice versa

$$\text{Copy data from CPU to GPU }(dest, src, count),$$

$$\text{Copy data from GPU to CPU }(dest, src, count),$$

where *dest* is a copy variable, *src* is a variable whose value should be copied, and *count* is the volume of data to be copied (in bytes).

Example 1. We will illustrate the use of some SAA-M operations described above for the construction of a simple algorithm to be executed on a GPU. Below, the SAA scheme of an algorithm that parallelly fills a numerical array with element values is given.

```
SCHEME Parallel GPU algorithm for assigning values to array elements;
KERNEL Set array values (A, N, val) =
    Get global index of the (i)th thread;
    Assign value (A[i], i + val);
main function =
    Declare integer array (h_A, N);
    Declare integer array (d_A);
    Allocate GPU memory (d_A, A_size);
    Set array values (N_b, N_th, d_A, N, 10);
    Copy data from GPU to CPU (h_A, d_A, A_size);
    Deallocate GPU memory (d_A);
    Print array (h_A, N);
```

The scheme includes implementations of the kernel function and main function of the scheme. The kernel function Set array values (A, N, val) assigns the value of $i + val$ to the ith element $A[i]$ of a one-dimensional integer array A of length N,

where i is the index of the current thread and val is a parameter of the function. The implementation of the main function includes the operators of determining the array h_A that should be processed and which is stored in the central processor memory and also the operators for determining and reserving memory for the corresponding array d_A in the GPU memory. Then the kernel function described above is called and, as a result, elements of the array d_A are simultaneously filled with values. The values of the parameters N_b and N_{th} (the numbers of blocks in the grid and threads in each block) are chosen so that $N_b * N_{th} = N$. Then data from the array d_A are copied into the array h_A, and the operator of deallocation of the memory reserved for the array d_A is executed. The last operator of the main function displays the resulting array h_A on a monitor.

In the course of designing schemes of algorithms, rewriting rules are applied together with SAA-Ms to automatedly transform these schemes using the developed system of symbolic computations TermWare [9]. This transformation is based on the application of systems of rules $f(x_1,\ldots,x_n) \to g(x_1,\ldots,x_n)$ to an algorithm represented in the form of a term, where f and g are terms and x_i are variables of terms.

Example 2. Let us apply the technique of rewriting rules to transform the kernel function Set array values from Example 1. We first present this function in the form of the term

$$\begin{array}{l} \text{set_array_values}\,(\text{params}\,(A, N, val), \\ \quad \text{get_global_thread_index}\,(i), \\ \quad \text{assign_value}\,(\text{arr_elem}(A,i),\ \text{summ}\,(i, val)) \\ ). \end{array}$$

Assume that the operator assign_ value presented in the implementation of this function should be included into the block of the branch operator IF so that it is executed only under the condition $i < N$. To this end, we apply the rule

$$\begin{array}{l} \text{set_array_values}(\$x1, \$x2,\ \text{assign_value}(\$x3, \$x4)) \to \\ \text{set_array_values}(\$x1, \$x2, \text{IF}(\text{LESS}(i, N), \text{assign_value}(\$x3, \$x4))), \end{array}$$

to the term described above, where $\$x1$, $\$x2$, $\$x3$, and $\$x4$ are variables. As a result, the initial term will be transformed into the term

$$\begin{array}{l} \text{set_array_values}\,(\text{params}\,(A, N, val), \\ \quad \text{get_global_thread_index}(i), \\ \quad \text{IF}\ \ (\text{LESS}(i, N), \text{assign_value}\,(\text{arr_elem}(A,i), \text{summ}\,(i, val))) \\ ). \end{array}$$

The use of SAA-M operations for designing a parallel algorithm that solves a problem from the field of meteorology is considered below.

SOFTWARE TOOLS FOR AUTOMATED SOFTWARE DESIGN

The developed software tools for automated design and generation of programs are based on the use of SAAs and the method of interactive construction of syntactically correct programs (ISD method) [1]. In contrast to traditional parsers, the ISD method is oriented not to the search for and correction of syntactic errors but to the avoidance of the possibility of their emergence in the course of designing an algorithm. The essence of the method consists of top down level-by-level design of schemes by superposing SAA-M language constructions. At every design step, the system interacts with the user and invites him to choose only the constructions whose substitution into the text being formed does not violate the syntactic correctness of the scheme being designed. The program text is automatically generated from the designed algorithm scheme. The mapping of SAA-M operations into a text in a programming language is represented in the form of templates and is stored in a database.

The described approach is implemented in the IDS system [2, 6–8]. To automatically transform algorithms, the IDS system is used together with the system TermWare [9] based on the paradigm of rewriting rules.

Let us consider a new version of the IDS system, namely, the OISD system destined for the interactive design, generation, and launching of programs. A distinctive feature of the OISD software tools in comparison with the IDS ones is that it is an Internet-based multiuser system and that it has a distributed architecture. Note that the components of OISD support tools are autonomous, flexibly connected, and have an agreed data exchange protocol, i.e., the described model is intrinsically service-oriented.

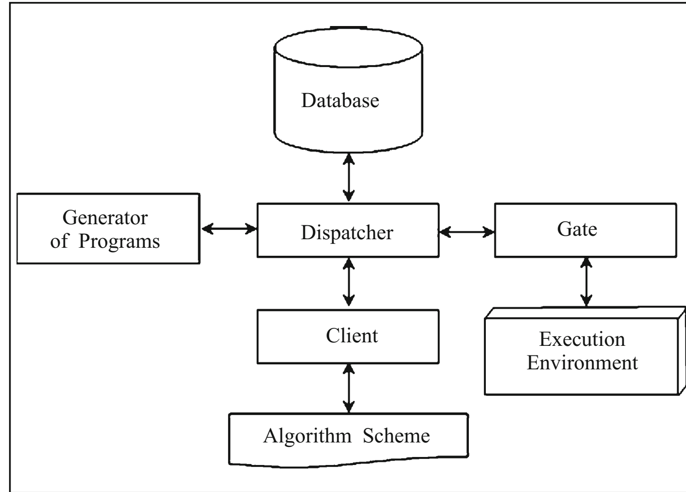

Fig. 1. Architecture of the OISD system.

The OISD system consists of the components presented in Fig. 1. The client is a web interface providing the interaction of the user with the system and its resources. It allows one to design algorithm schemes by choosing necessary elements from the database, to generate codes in the target programming language Java or C ++, and to start programs in the execution environment. In the system, the design of algorithms is based on the interactive mode with the use of lists of constructions and trees of algorithms. Specifications of language constructions are based on an English-language version of the CAA/1 language. Schemes are designed using the top-down method by detailing algorithmic constructions. In the course of designing a scheme, the user selects necessary constructions from their list and adds them to the tree of the corresponding algorithm.

The dispatcher is the core of the software tools and organizes interconnections between the user, generator of programs, gate, and database. The generator of programs generates the text of the program of an algorithm from its scheme constructed with the help of the interface (client). The gate provides the start, analysis, and obtaining of the results of execution of programs in their execution environment. The latter is a software platform installed on the server side of the OISD system and includes an operating system and a software support required to compile and start programs.

The database of the OISD system stores the developed projects of algorithms, descriptions of SAA-M language constructions, frameworks of projects in programming languages, and information on settings specific for starting programs. The description of each SAA-M language construction in the database includes its representation in a natural language, information on the type of the construction, implementation (template) in the chosen programming language, names of variables, and a list of formal parameters.

The OISD system was mainly used earlier for developing parallel sort programs for multi-core central processors. In this work, the database of the system is supplemented with operations oriented towards the design of programs for GPUs.

AN EXAMPLE OF USING ALGEBRA-ALGORITHMIC SOFTWARE TOOLS FOR DESIGNING AN APPLIED PARALLEL PROGRAM

We will illustrate the use of the SAA-M apparatus and OISD system by an example of development of a CUDA program for solving the two-dimensional diffusion-convection problem that arises in mathematical modeling of atmospheric circulation in meteorology. This test problem consists of solving the following collection of convection-diffusion equations:

$$\frac{\partial u}{\partial t}+v_1\frac{\partial u}{\partial x_1}+v_2\frac{\partial u}{\partial x_2}=\frac{\partial}{\partial x_1}\left(\mu_1\frac{\partial u}{\partial x_1}\right)+\frac{\partial}{\partial x_2}\left(\mu_2\frac{\partial u}{\partial x_2}\right)+f \tag{1}$$

when $(x_1,x_2)\in\Theta$, $t\in[0;10]$, and

$$u(0,x_1,x_2)=\sin(x_1+x_2) \tag{2}$$

when $(x_1,x_2)\in\Theta$, $t=0$, and

$$u(t, x_1, x_2) = u_A(t, x_1, x_2) \quad (3)$$

when $(x_1, x_2) \in \partial\Theta,\ t \in [0;10]$,

where

$$\Theta = [0,1] \times [0,1];\ v_k = \sin(x_k);\ \mu_k = 0.001 + 0.1 * \sin^2(x_k) > 0;$$

$$f(t, x_1, x_2) = (v_1 + v_2 - (1 + 0.1 * (\sin(2x_1) + \sin(2x_2)))) * \cos(x_1 + x_2 - t) + (\mu_1 + \mu_2) * \sin(x_1 + x_2 - t).$$

Here, $u = u(t, x_1, x_2)$ is a dependent function (for example, windspeed); t is time; x_1 and x_2 are coordinates; Θ is a two-dimensional spatial domain of definition of the problem; $f = f(t, x_1, x_2)$ is the free term of the equation; v_k is the convection coefficient; μ_k is the diffusion coefficient. In what follows, for the coordinates x_1 and x_2, the notations x and y, respectively, are used.

To solve problem (1)–(3), the numerical finite-difference method described in [19] is used in the CUDA-program. In the course of solution, the domain Θ is divided into S_x and S_y uniform steps (equispaced nodes) along the axes x and y, respectively. Thus, the total number of nodes equals $S_x \times S_y$. The parallelization of computations for this problem consists of the partitioning of the domain of size $S_x \times S_y$ into subsets and parallel solution of the collection of subproblems defined on these subsets.

Below, the general SAA scheme of the developed parallel GPU algorithm for solving this diffusion-convection problem is given. The scheme is constructed in the OISD system according to the interactive design method considered earlier. In addition to the mentioned variables S_x and S_y, the algorithm contains the following basic variables: N_b is the number of blocks in the CUDA grid; N_{th} is the number of threads in each block; $Time = 0$ is the start time; $T = 10$ is the final time; $dT = 0.001$ is the time step; t is time; m is the number of computational steps; h_pX, d_pX, h_pY, and d_pY are arrays of arguments; u_x and u_y are arrays for storing values of the function u.

SCHEME Parallel algorithm for solving the convection-diffusion problem on a GPU;
main function =
  Get command line arguments (S_x, S_y);
  Initialize data for the convection-diffusion problem;
  Start the timer;
  Input initial data (N_b, N_{th});
  Fill the arrays of equation coefficients $(N_b, N_{th}, Time + m * dT)$;
  Fill the array for the function u with initial values (N_b, N_{th});
  Synchronizer (all threads have completed);
  Copy data from GPU to CPU (h_pX, d_pX, d_pX_size);
  Copy data from GPU to CPU (h_pY, d_pY, d_pY_size);
  Synchronizer (all threads have completed);
  Declare integer variable $(t, 0)$;
  WHILE $(m * dT * t < T)$
    Fill the arrays of equation coefficients $(N_b, N_{th}, Time + m * dT)$;
    Fill the boundary conditions with zero values (N_b, N_{th});
    Fill the boundary conditions (N_b, N_{th});
    Fill the arrays of equation coefficients $(N_b, N_{th}, Time)$;
    Fill the arrays u_x and u_y with initial values (N_b, N_{th});
    Synchronizer (all threads have completed);
    Compute u_x (N_b, N_{th});
    Synchronizer (all threads have completed);
    Compute u_y (N_b, N_{th});
    Synchronizer (all threads have completed);
    Compute the average value from u_x and u_y (N_b, N_{th});
    Synchronizer (all threads have completed);
    Increment $(Time, m * dT)$;
    Increment $(t, 1)$;
  END OF LOOP;

Stop the timer and print the execution time;
Copy data from GPU to CPU for the convection-diffusion problem;
Compare the computed function values with the real values and compute the error;
Write the obtained results for the function u into files;
Deallocate the memory occupied by the variables of the convection-diffusion problem;

The implementation of the main function of this SAA scheme begins with the operator reading values of two parameters of the command line and assigning the corresponding values to the variables S_x and S_y. Then data are initialized and time reading begins. Next, the operations with the arguments N_b and N_{th} call kernel functions. These functions input initial data and fill the arrays of coefficients of equations and the array for the function *u*. A special synchronization operator (synchronizer) placed after function calls delays computations until all threads have completed. Note that the algorithm contains WHILE loop whose body contains calls of additional kernel functions filling the arrays of coefficients and boundary conditions and also computing values of u_x and u_y and the average value. The algorithms of kernel functions are represented by individual SAA schemes.

The overall result of execution of the algorithm is the collection of computed values of the function u for coordinates x and y, $(x, y) \in \Theta$. At the end of the program, the computed values of the function u are compared with its real values and the corresponding error is computed. Then values of coordinates and the function u are written into files.

Using the OISD system and based on the constructed algorithm scheme, a program text in the CUDA C language is automatically generated. An experiment was performed that consisted of the execution of the developed parallel program and the corresponding sequential program on one of nodes of a section of the SCIT-4 cluster at the V. M. Glushkov Institute of Cybernetics of NAS of Ukraine [20]. This node contains the NVIDIA Tesla M2075 GPU Accelerator with 448 computing cores and Intel Xeon E5-2670 as the central processor. The numerical experiments were carried out for the problem size values varying from $S_x \times S_y = 64 \times 64$ to $S_x \times S_y = 2048 \times 2048$. Values of the multiprocessor speed-up factor T_s / T_p are obtained, where T_s and T_p are the execution times of the sequential and parallel programs, respectively. The maximum speed-up factor value for $S_x \times S_y = 2048 \times 2048$ amounted to 69.

CONCLUSIONS

The development of the algebra-algorithmic methodology and software tools for the automated design and generation of programs for GPUs are considered. An advantage of the described approach consists of using language constructions close to a natural language and also the application of a method that provides syntactic correctness of algorithms and programs being designed. The approach is implemented in a development support system destined for the interactive construction of algorithm schemes and generation of programs in the basis of reusable software components, i.e., operations of algebra of algorithms. The use of the proposed software tools is illustrated by the example of developing a parallel program for the diffusion-convection problem. An experiment was carried out that consisted of running the developed program on a GPU and, as a result, a good index of paralleling computations is obtained.

REFERENCES

1. F. I. Andon, A. E. Doroshenko, G. E. Tseytlin, and E. A. Yatsenko, Algebra-Algorithmic Models and Methods of Parallel Programming [in Russian], Akademperiodika, Kyiv (2007).
2. A. E. Doroshenko, K. A. Zhereb, and E. A. Yatsenko, “Formalized design of efficient multithread programs,” Problems in Programming, No. 1, 17–30 (2007).
3. A. Doroshenko and O. Yatsenko, “Using ontologies and algebra of algorithms for formalized development of parallel programs,” Fundamenta Inform., **93**, Nos. 1–3, 111–125 (2009).
4. A. Doroshenko, K. Zhereb, and O. Yatsenko, “Formal facilities for designing efficient GPU programs,” in: Proc. Int. Workshop “Concurrency, Specification, and Programming (CS&P’2010)” (Bernau, 27–29 Sept., 2010), Humboldt University, Berlin (2010), pp. 142–153.
5. F. I. Andon, A. E. Doroshenko, and K. A. Zhereb, “Programming high-performance parallel computations: Formal models and graphics processing units,” Cybernetics and Systems Analysis, **47**, No. 4, 659–668 (2011).

6. O. Yatsenko, "On parameter-driven generation of algorithm schemes," in: Proc. Intern. Workshop "Concurrency, Specification, and Programming (CS&P'2012)" (Berlin, 26–28 Sept., 2012), Humboldt University, Berlin (2012), pp. 428–438.
7. A. Doroshenko, K. Zhereb, and O. Yatsenko, "Developing and optimizing parallel programs with algebra-algorithmic and term rewriting tools," in: Proc. Intern. Conf. "Information and Communication Technologies in Education, Research, and Industrial Applications (ICTERI' 2013)" (Kherson, 19–22 June, 2013), Revised Selected Papers, **412**, Springer, Berlin (2013), pp. 70–92.
8. E. A. Yatsenko, "Integrating support tools of algebra of algorithms and term writing for developing efficient parallel programs," Problems in Programming, No. 2, 62–70 (2013).
9. A. Doroshenko and R. Shevchenko, "A rewriting framework for rule-based programming dynamic applications," Fundamenta Inform., **72**, Nos. 1–3, 95–108 (2006).
10. N. Wilt, The CUDA Handbook: A Comprehensive Guide to GPU Programming, Addison-Wesley, Boston (2013).
11. D. Sannella and A. Tarlecki, Foundations of Algebraic Specification and Formal Software Development, Springer, Berlin (2012).
12. P. Flener, "Achievements and prospects of program synthesis," Lecture Notes in Artificial Intelligence, **2407**, 310–346 (2002).
13. S. Gulwani, "Dimensions in program synthesis," in: Proc. 12th Intern. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (Hagenberg, 26–28 July, 2010), ACM, New York (2010), pp. 13–24.
14. S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu, "CUDA-lite: Reducing GPU programming complexity," in: Proc. 21st Intern. Workshop on Languages and Compilers for Parallel Computing (LCPC' 2008) (Edmonton, 31 July–2 Aug, 2008), Springer, Berlin (2008), pp. 1–15.
15. W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele, "Efficient execution of Kahn process networks on multi-processor systems using protothreads and windowed FIFOs," in: Proc. Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia'09) (Grenoble, 15–16 Oct., 2009), IEEE Computer Society Press, Los Alamitos (2009), pp. 35–44.
16. T. D. Han and T. S. Abdelrahman, "hiCUDA: A High-level language for GPU programming," IEEE Transactions on Parallel and Distributed Systems, **22**, No. 1, 78–90 (2011).
17. H. Jung, Y. Yi, and S. Ha, "Automatic CUDA code synthesis framework for multicore CPU and GPU architectures," Lecture Notes in Comp. Sci., **7203**, 579–588 (2012).
18. C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a high-level language for GPUs (via language support for architectures and compilers)," in: Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 12) (Beijing, 11–16 June, 2012), ACM, New York (2012), pp. 1–12.
19. V. A. Prusov, A. E. Doroshenko, and R. I. Chernysh, "A method for numerical solution of a multidimensional diffusion-convection problem," Cybernetics and Systems Analysis, **45**, No. 1, 89–100 (2009).
20. Supercomputer of the Institute of Cybernetics of NASU, http://icybcluster.org.ua.