Bull. Nov. Comp. Center, Comp. Science, 3 (1995), 39–58
© 1995 NCC Publisher

High-performance heterogeneous processing in concentrating computing system*

Ya.I. Fet and A.P. Vazhenin

In this paper the possibilities of organizing heterogeneous computing in so-called Combined Architecture systems consisting of a basic host subsystem (a massively parallel computer), and a set of high-performance specialized parallel coprocessors (hardware modules) executing the main workload are discussed. To optimize the choice of hardware modules, a classification of massive computations, based on the notion of processing types, which correspond to the character of data processing is suggested. A technique of parallel programming for the suggested concentrated heterogeneous systems, ensuring close matching of the tasks to the hardware modules is introduced.

1. Introduction

During the last decade, the most impressive gain in computer performance has been made by raising the *level of parallelism* and using of microprocessors with improved *physical characteristics*. Such important issue as the *specialization* of hardware was rarely addressed, though it presents a very important source of further improvement of computer systems performance.

Generally speaking, a struggle between the universality and the specialization always existed in the development of computer architecture. The principal cause of this contradiction was in economic reasons of different kinds: buying a single general purpose computer, the user will be able to solve any problem he needs, hence, general-purpose computers could meet a broader market; the specialized computer cannot be evenly loaded with jobs, as compared with a universal one, etc.

As a matter of fact, the most powerful contemporary supercomputers attain striking values of *peak performance*. However, of practical interest is the *real performance* attainable on important applications. Usually one has to make every effort in order to "embed" his algorithms and problems into the given system architecture.

Still, the real problems are usually non-uniform. For different problems and various fragments of the same problem it is expedient to use features of different architectures. Utilization of a rigid architecture leads to a large gap between the peak and the real performance.

*Supported by the Russian Foundation for Basic Research under Grant 95–01–01350a.

Recently, a wide interest was attracted by *Heterogeneous Computing* [1], which implies a *distributed* network system of several *commercially available* computers of diverse architectures. In such environment, the user is able to vary flexibly the style of programming, in accordance with the characteristics of his problems.

At the same time, the network approach in organizing heterogeneous computing seems to have a number of shortcomings (see, for instance [2]):

1. Complete computers of serial production are used, each of them being not perfectly suited for the specific application problem.
2. Exploiting the networks, as well as the node supercomputers, involves large overheads.
3. The internode data transfer through the networks usually causes considerable delays.

The modern developments of VLSI technology obviates most of the mentioned objections against specialized processors. A proper choice of hardware architecture provides the highest speed in solving of corresponding problems.

The best results can be achieved when "computers look like the problem that they are trying to solve" [3]. Following this way, one could design a separate specialized processor for each problem, or a class of problems. However, too many different processors should be included in such a system.

How can be significantly expanded the range of applicability of a special hardware device, without loosing the efficiency of this device?

In this paper, an approach to the solution of this problem based on comparison of the widespread types of batch processing, on the one hand, and the known hardware structures, on the other is suggested. This allows for selection of a limited set of basic *processing types*, covering in common nearly all of hard computations involved in solving of the overwhelming majority of important problems.

This approach leads, in turn, to the conception of so-called *combined architecture* [4, 5], in which different kinds of processing are executed by corresponding dedicated accelerators. Now accelerators are widely used in computing systems for increasing their performance. However, nearly in all cases, the designers come merely to floating-point arithmetic processors, which are in essence *weakly* specialized. In contrast with the known distributed heterogeneous systems, the combined architecture allows for building of Concentrated Heterogeneous Systems (CHSs).

The peculiarities of the present paper are as follows:

1. Use of a highly parallel SIMD computer as a basic (host) subsystem of combined architecture.
2. Broad application of *strongly* specialized accelerators (*hardware modules*) of different intentions and various hardware structures including parallel, pipeline, systolic, associative, etc.
3. Systematic approach to the selection of accelerators based on the classification of massive procedures according to so-called *processing types*.
4. Use of a definite technique of flexible heterogeneous programming ensuring the decomposition of a given application problem into processing types, with subsequent mapping of the designed algorithm onto the available set of hardware modules of a specific system.

5. Balanced interaction of various subsystem of the combined architecture.

In Section 2 the main features of the combined architecture are described. Section 3 is devoted to the issues of classification of massive parallel procedures. In Section 4 some examples of typical hardware modules are presented. The suggested technology of programming in CHSs is described in Section 5. In Section 6 some conclusions are made.

2. Combined architecture

We call *combined architecture* a cooperation of a highly parallel host computer with a set of specialized processors. In this architecture, solving of any problem is considered as interaction of several processes, so that execution of each process is delegated to a specialized subsystem, most efficient in implementation of this process. The subsystems are controlled in such a way that their balanced operation might be ensured, and special complementing features of subsystems might be best exploited. For each subsystem a structure is chosen which best corresponds to the function it should perform.

At choosing the architecture of subsystems, the following considerations should be taken into account.

In the combined architecture (Figure 1), the main working load of the processing is delegated to the coprocessors. Hence, the requirements to the performance of the basic computer can be moderate. However, in order to ensure the effective interaction between the subsystems and the necessary data flows, this computer should have a sufficiently high degree of parallelism, a large capacity of the main memory, and an adequate software.

At the same time, in view of the general objectives of the system, extremely high demands should be made to the performance of each coprocessor. It means that special care is needed in selection of the structures of coprocessors. The most suitable architecture for the basic subsystem seems to be the fine-grained SIMD similar to DAP, or CM. These computers fit the above mentioned requirements (memory size, parallelism, etc.), while they are much cheaper of the other parallel systems because of using simple single-bit processing elements.

The last is, in fact, the weak point of this remarkable architecture preventing it to take a leading place in computation intensive applications. In our case, however, this drawback is not a decisive factor, because hard computations, according to the main idea of combined architecture, are delegated to the coprocessors. On the other hand, the fine-grained SIMD host computer can ensure the necessary input/output and preprocessing of large data arrays, due to such inherent properties as flexible highly parallel memory access, fast manipulation on data structures, and large I/O channel bandwidth.

Efficient operation of high-performance specialized architectures is known to be possible only when proper preparing and staging of data are provided. For instance, in systolic implementation of matrix problems the incoming matrices should go (by row or column) to the inputs of certain PEs in the form of a polygon of definite shape. Sometimes the sequences of coefficients must be interleaved by zeros. Using conventional memory devices and I/O hardware would cause serious diffi-

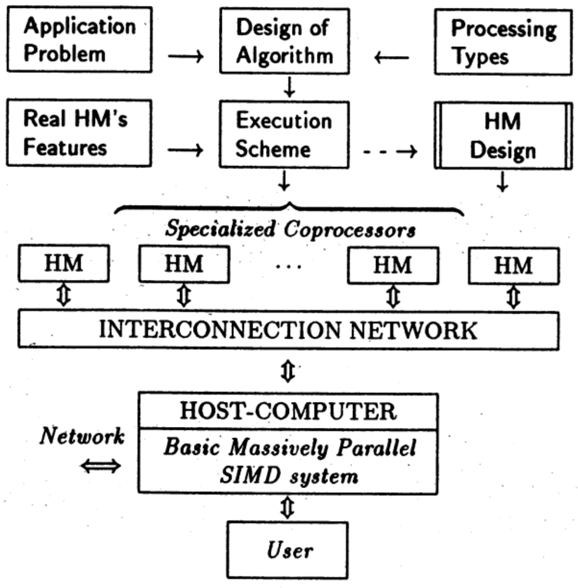

Figure 1. Combined architecture

culties in realizing these elaborated data manipulations. Inevitably, the expected performance of the systolic array would degrade.

In [4] an example of a combined architecture is presented in which a STARAN-like SIMD computer is used as a host, and a set of systolic arrays as a powerful processing subsystem. A family of basic procedures for transformation of data structures is also described in [4], accomplished by the host. It is shown that these procedures provide for a balanced operation of the system as a whole, and allow to utilize the potential speed of specialized hardware. The preprocessing procedures of this kind are fundamental to different dedicated architectures. Another benefit of using the fine-grained SIMD as a host comes from saving of all its features and advantages to be exploited at solving in the system of problems well fitting the "data parallel" paradigm. Thus, the combined architecture might extend the lifetime of existing fine-grained SIMD computers.

Looking at the present development of supercomputers, one can discover two important trends: increasing of the performance of processing elements, and extending of allowable programming styles. The combined architecture pertains to both these trends: the problem-oriented coprocessors of different structure provide a multiarchitectural environment, while the strong specialization and the massive parallelism of these processors ensure high performance.

3. Classification

The novelty of the present approach is that the specific type, or "technology", of processing necessary for efficient execution of the most labor-intensive procedures involved in the implementation of a problem is used as a criterion for the selection of appropriate hardware architecture. As a rule, similar technologies are encountered as well in solving problems of other classes.

Analyzing the common numerical methods, algorithms, and programming languages, one can select a set of recurring general technologies, or styles of data treatment which we call processing types. It such analysis, specific features of existing hardware should be taken into account in order to make possible a reasonable mapping of the processing types onto efficient hardware modules.

3.1. A short history

One of the earliest investigations in massively parallel processing is due to Leonid Kantorovich who described in 1957 the so-called "large-block programming system" [6]. He proposed to consider as basic objects operated by the system ordered sets called *quantities* (such as vectors, matrices, etc.), a single number being the simplest quantity, called an *element*. Some special operations on quantities were introduced: *arithmetical* operations as extensions of usual arithmetic on any element of the quantity, and *geometrical* operations which do not change the values of quantities but only transform their structures.

A significant event in the development of parallel processing was the appearance of APL (A Programming Language) devised by Kennet Iverson [7]. In APL, the variables are *logical, integral, numerical, and arbitrary*. They can be *scalars* as well as rectangular *arrays* of any rank and dimensionality. The following types of operations are defined in APL:

1. *Scalar* operations, with scalar variables both as arguments and results. The scalar operations are subdivided into *unary* and *binary*.
2. *Compound* operations, being an extension of scalar operations to arrays. Four kinds of extensions are provided: *component-wise processing, reduction, inner product*, and *outer product*.
3. *Mixed* operations, in which the ranks of the arguments and the results are different. These operations serve mostly for transforming the array structure.

Later on, some massive, large-block operations were further developed in such programming languages as PL-1, Algol-68, etc. For the time being, the notions of array operands and parallel operators introduced into the mentioned languages remain just conceptual. Up to the 70-ies, in the real computing systems they were executed by means of usual sequential subroutines. With the advent of computers of parallel architecture (Illiac IV, Staran, DAP) the true *parallel* execution of massive processing become a reality, though the extent of parallelism in each case was limited by the possibilities of the particular hardware.

The development of new parallel systems stimulated more precise classification of massive operators, for better exploiting of the advantages of different architectures. One attempt to construct such a classification was made in [8], where five classes of operators were selected:

1. *Numerical component-wise processing*. The arguments are one or two numerical vectors, the result is also a numerical vector, and the processing consists in some computation simultaneously applied to all the components of argument vectors.

2. *Numerical reductive processing.* The argument is a numerical vector, the result is a scalar, the processing consists of application of some binary operation to all the components of the argument.
3. *Logical component-wise processing.* The arguments are two numerical vectors, the result is a binary vector, the processing consists in finding out a specified relation between the corresponding components of array arguments.
4. *Logical reductive processing.* The arguments are an array and a scalar, the result is a subset of elements of the array argument meeting specified conditions. The processing consists in simultaneous comparison of the scalar with all elements of the array.
5. *Data structure transformations.* Both the argument and the result are arrays, the processing consists in some modifying of the structure of the argument array, without changing the values of its elements.

Recently, a valuable contribution to the classification of massively parallel operations has been made by Guy Blelloch [9]. In his book, a set of *primitive instructions* is discussed for the so-called "Scan Vector Model". Three classes of instructions, relying to a considerable extent on the APL operations, are defined: *scalar, vector*, and *vector-scalar* instructions. The vector instructions are divided into *element-wise, scan*, and *permutation* instructions. It should be noted that Blelloch's primitives are virtually directed toward a definite architecture of the Connection Machine.

Following our approach, in selecting of the typical massive procedures to define corresponding processing types one should not limit himself by the features of a specific computer system, but rather examine a broad variety of existing and prospective hardware structures.

3.2. Classification of problems

Among the numerous problems requiring the power of supercomputers to be solved in adequate time, several prevailing classes can be chosen constituting the main part of the computer charge. These problems belong basically to the following two groups.

A. Numerical problems: Numerical approximation, Linear algebra, Ordinary differential equations, Equations of mathematical physics, Numerical simulation, Discrete transforms, Combinatorial problems, Error analysis, Computer graphics and geometry, Signal and image processing, Simulation of complicated objects.

B. Non-numerical problems: Searching and sorting, Symbolic processing, Text processing, Databases, Operational systems, Artificial intelligence (Production systems, Logical inference systems, Pattern and speech recognition, Genetic algorithms, Neurocomputing, Robotics, Computer-aided design).

Of course, this list of problems is incomplete, and needs further improving. Nevertheless, we suppose that the enumerated problems cover the needs of the most important applications. Hence, the analysis of the processing types involved in these problems may be rather representative.

3.3. Processing Types

Our goal here is to provide a classification of processing styles confronting them with the known hardware structures, in order to find a reasonable mapping: *PROCESSINGTYPE* $\rightarrow$ *HARDWARE MODULE*. Indeed, the variety of styles, or technologies involved in machine realization of different application problems is not too large.

We will describe the Processing Type (PT) as a three-tuple: $PT = \{A_1, A_2, T\}$, where $A_1(A_2)$ corresponds to the data type of the first (second) argument, and T corresponds to the transformation to be executed upon these arguments. In massive procedures, the terms A_1 and A_2 usually take the values "Vector (V)", "Scalar (S)", and "Binary (B)". Other possible types of arguments can be: "Array", "Set", "Relation", "Tree", etc. Examples of transformations T are: "Arithmetic (A)", "Logic (L)", "Permute (P)". At this level of discussion, the dimensionalities of the arguments can be ignored. It is supposed that they do not exceed the parallelism of the corresponding hardware modules. If it is not the case, than the usual (program) decomposition techniques should be applied.

Table 1 shows the notations of the most common processing types. In this table, some HMs are also shown suitable for implementation of different PTs. Of course, this list of basic processing types needs further extensions and corrections.

Table 1

A_1	A_2	T	PT	HM(s)
Vector	Vector	Arithmetic	VVA	Pipeline, PVM, PVA, Systolic
Vector	–	Arithmetic	VoA	Pipeline, PVM, PVA, Systolic
Vector	Vector	Logic	VVL	Set Intersection Processor
Vector	Scalar	Arithmetic	VSA	Pipeline
Vector	Scalar	Search	VSS	Associative Processor
Vector	Interval	Search	VIS	Associative Processor
Vector	Vector	Search	VVS	Set Intersection Processor
Vector	–	Order	VoO	Sorting Network, Systolic
Vector	–	maX	VoX	Extremum Selector
Vector	–	miN	VoN	Extremum Selector
Vector	Vector	Permute	VVP	Permutation Network
Vector	–	Permute	VoP	Permutation Network
Vector	–	Compress	VoC	Digital Compressor
Vector	–	Expand	VoE	Permutation Network
Vector	–	Logic	VoL	Associative Processor
Binary	Binary	Logic	BBL	Associative Processor
Binary	Binary	Permute	BBP	Permutation Network
Binary	–	Permute	BoP	Permutation Network

4. Specialized hardware modules

We have already mentioned that the combined architecture is an open system capable of including a variety of hardware modules of different architectures. Further investigations in the computational requirements set by problems, as well as the progress in hardware technology, will supplement the assortment of hardware modules with novel elaborate devices.

The goal of this Section is to give some examples of existing hardware modules. To begin with, a brief survey of well-known structures is made. Then, several specialized processors are described introduced in the previous papers of the present authors.

4.1. Vector Pipeline Architecture

These devices are widely adopted in modern supercomputers (Cray, Convex, Fujitsu FACOM VP, and others), as well as in arithmetical accelerators (Intel860, DEC Alpha, Fujitsu μVP, etc.). The vector pipeline devices are appropriate for implementation of processing types VVA, VSA, VoA.

4.2. Systolic/wavefront arrays

These devices are locally connected networks of homogeneous cells, with a regular directed flow of data and results (see, for instance, [10]). Note, that a systematic method for design and optimization of systolic arrays has been developed in [11]. The systolic/wavefront processors fit well to implementation of processing types VVA, VoA, VoO, etc.

4.3. Permutation Networks

The permutation network (also called an interconnection network or a commutator) is now an indispensable component of any parallel computing system because of the necessity to provide fast data transfer between different nodes of the system. The permutation network may be considered as a specialized processor embedded into proper system.

The most popular permutation networks are: the suffle/exchange network, the Ω-network, the Data Manipulator, the Flip network (see, for instance [12]). Simple meshes are often used as permutation networks: the NEWS-grid (a rectangular four-neighbour mesh), and the X-grid (an eight-neighbour mesh). In the CM-5 system a novel efficient network was implemented called a fat-tree [13]. The permutarion networks realise the processing types VVP, VoP, VoE.

4.4. Sorting devices

A variety of methods and tools of hardware sorting have been discussed in the literature (see, for instance, [14]). A remarkable achievement in this field was Batcher's *sorting network* [15], which is capable to implement the ordering of a N-element array with the time estimation of the order $\log_2^2 N$. Note also, that systolic arrays can be efficiently used for sorting.

The sorting devices implement processing types VVP, VoO.

4.5. Associative Array Processors

Different Associative Array Processors (AAPs) were developed to implement non-numerical problems (database machines, information systems, etc.). An extensive literature is devoted to the architecture and applications of AAPs (see, for instance, [16, 17]).

As it was shown by various authors, the AAP can realize different algorithms and problems. However, they are most efficient for the processing types VVL, VSS, VIS, VoL, BBL.

The operation of AAP is based on the principle of content-addressable memory (CAM). In general, the CAM can be presented as a rectangular $m \times n$ matrix from identical cells, each of which contains a single-bit memory and a logical circuit realizing the equivalence function. In the rows of this matrix (i.e., in the flip-flops of its cells), the m n-bit words of the argument array are stored. In each column of the matrix a bus is provided traversing all the cells of this column, and carrying the corresponding bit of the second, scalar argument (called comparand, or associative tag). In each row, a horizontal bus is provided gathering the logical values of the equivalence functions from all the cells of this row. Evidently, in such a matrix the response signal on the horizontal bus will be produced only in those rows containing words which coincide with the comparand.

The CAM represents a strongly specialized processor oriented to a very important processing type VSS. Here, the algorithm of associative search is simulated in the course of signal flow along a specific distributed logical net of the CAM matrix. This kind of processing might be called *quasi-analogue simulation*. Different logical nets possessing such properties we call *Distributed Functional structures (DF-structures)* [18]. We will describe below two more DF-structures which can be used in designing efficient hardware modules of combined architecture.

4.6. Extremum selector (α-structure)

Consider a two-dimensional homogeneous structure of size $m \times n$ with m n-bit elements (binary numbers) of the processed array written in its memory so that each element occupies one row of the matrix (most significant digits to the left).

For the maximum selection a known algorithm of column-wise (from left to right) inspection of values a of the bits of array elements is used, described as follows.

Step 1. The contents of the first (left) column is looked over, that is, the most significant digits of all m elements. If all these digits are zeros, then at the following step the second digits of all m elements are looked over. If, however, the first column contains both zeros and ones, then at the second step only those elements which had ones in the first position are looked over.

Step j. The contents of the j-th column (j-th digits of all elements) are looked over, in those rows which were singled out at the $(j-1)$-th step. If all these digits are zeros, then at the following step the $(j+1)$-th digits of the same rows are looked over. If there are both zeros and ones in the memory elements looked over at the j-th step, then at the $(j+1)$-th step only rows corresponding to ones are looked over.

The subset of the rows singled out at the last (n-th) step (and this may consist of only one row) contains the maximal elements.

As it was shown in [18], this algorithm is realized when each cell of the distributed logical network implements the functions $z' = z(a \vee \bar{y})$, $x' = x \vee az$, where the variables x and z belong to the vertical and horizontal look-over channels, and y in each column should be set to x'_m.

Evidently, the implementation of minimum selection differs only in that the negations of all binary variables a should be used at the array inspection.

4.7. Digital Compressor

We call digital compressor a functional unit realizing the compression of binary vectors, that is, the conversion of an arbitrary binary vector into a prefix (suffix) vector of the same weight. Various types of digital compressors are known. Consider one of them, based on a simple DF-structure called λ-matrix [19].

This is a two-dimensional homogeneous array (Figure 2a) each cell of which contains two logical gates, AND and OR (Figure 2b) and realizes logical functions $z' = zt$ (the horizontal channel) and $t' = z \vee t$ (the vertical channel).

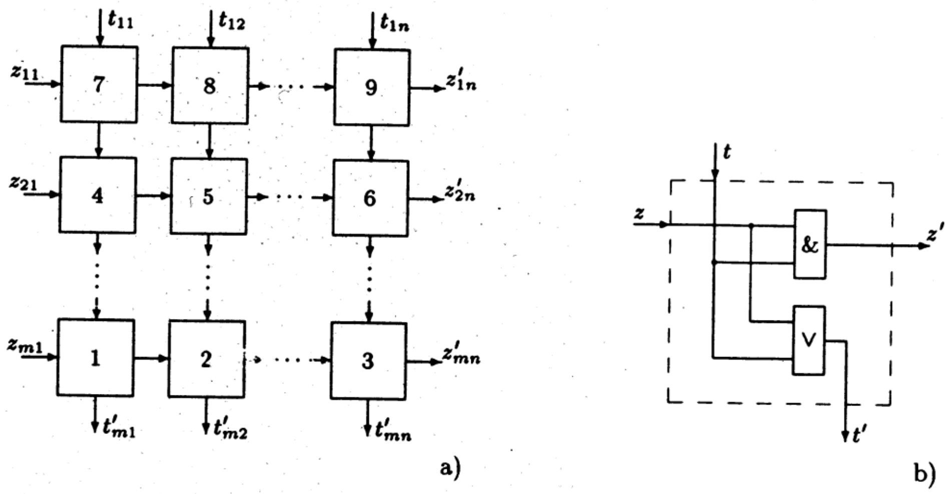

Figure 2. Compression of binary vectors: a) general structure of λ-matrix; b) logical circuit of λ-cell

Let an arbitrary binary vector be applied to the inputs z of the left boundary of λ-matrix.

Consider the first (the left) column of the matrix. The variable t retains its initial value 0 in the vertical channel of this column only till $z = 0$. In some i_1-st cell, where $z = 1$ is encountered for the first time, the value of t changes to 1 which cannot change then till the lower bound. However, the i_1-st cell receives yet the signal $t = 0$. Hence, it is the single cell in the whole column where the combination $z\bar{t} = 1$ is present. This combination may serve as an indication for extracting the "one".

The horizontal channel of thus indicated i_1-st cell is closed by the signal $t = 0$. Hence, the first "one" of the given vector does not propagate further along the current row. In all cells lower than the indicated one, $t = 1$, so that $z' = z$. Thus, to the inputs of the second column a duplicate of the given vector is applied, except for its first "one".

Similar transformations are performed in the second, the third column, and others: in some i_2-nd cell of the 2-nd column the second "one" of the given vector is indicated, in some i_3-rd cell of the 3-rd column the third "one" is indicated, etc. $(i_1 < i_2 < \ldots)$.

Evidently, signals "1" appear at the outputs t' of the lower bound in the 1-st, 2-nd, ... columns of the λ-matrix, and the number of such columns corresponds to the number of "ones" in the given binary vector. Hence, the λ-structure performs the compression of a binary vector, and can realize the processing type VoC.

4.8. Pipelined Vertical Adder

The vertical adder [19] is a pipelined device implementing the reductive summation of m n-bit elements of an array of integers.

The algorithm of vertical addition consists in sequential calculation of the number of "ones" in the bit-slices of the initial array, beginning with the least significant digit. The obtained partial sums are summed, with the systematic one position left shift, taking into account the weights of the ones in the processed bit-slices. This procedure has a pipeline character, which determines the structure of the processor.

The Pipelined Vertical Adder (PVA) consists of four units: a Digital Compressor (DC), a Leading One's Selector (LOS), a Code Transformer (CT), and an Adder-Accumulator (AA). The first three units perform the proper counting of the number of "ones" in bit-slices (the *weighting*), and the last the addition of partial sums.

To realize weighting, the following technique is used. At first, the compressor DC transforms the next bit-slice (an unitary code) into the equivalent prefix vector. Then the selector LOS produces a signal marking the position of the last "one" of the prefix vector. Finally, the transformer CT outputs a binary number corresponding to the weight of the given bit-slice.

Note that here a pipelined compressor should be used.

The Adder-Accumulator should ensure the finishing of the shift and the adding of the next partial sum at one cycle time. Then, the total time necessary to perform reductive summation in the considered device, will depend only on the word length n of the elements of the initial array. Taking into account the filling time $n - 1$ of the pipeline compressor (the "depth" of the pipeline), it follows that the whole procedure time is approximately $2n$ steps.

This device can be used in realizing the processing types VVA, VoA.

4.9. Pipelined Vertical Multiplier

The Pipelined Vertical Multiplier (PVM) described in [20] intended for concurrent component-wise multiplication of two integer vectors is also a two-dimensional matrix of size $m \times n$. Each cell of PVM contains three flip-flops, m, s, and c, a full binary adder, and some control logic.

In PVM, sequential multiplication (beginning from the least significant bits) is performed simultaneously in all rows, while the principle of carry save addition is applied in computing of the sums of products.

The procedure consists of three phases.

Phase 1. Loading of multiplicands. The bit-slices of the array of multiplicands are sequentially loaded from the main memory into the columns of the flip-flops m of PVM.

Phase 2. Proper multiplication. The bit-slices of the arrays of multipliers (located in the main memory) are sequentially fed to corresponding control inputs ("multiplier buses") of PVM rows. In each cycle of the considered procedure in all rows, where the current bits of the multiplier are "ones", an addition is performed of the corresponding multiplicands (from the rows of flip-flops m) with the partial product stored in a carry-save manner in the rows of flip-flops s and c. At this addition, the partial sums s are moved by one position to the left, while the saved carries c do not change their positions. Thus, in each cycle a column of true values of current bits of the array of products is produced (beginning from the least significant bits), and stored in the main memory. At the end of phase 2, n least significant bits of all products are produced.

Phase 3. Output of the $n-1$ most significant bit-slices of the array of products.

It can be easily shown that the component-wise multiplication of two vectors in PVM takes approximately $O(n)$ cycles.

This device can be used in realizing the processing types VVA, VoA.

4.10. Set Intersection Processor

Another example of a specialized processor for non-numeric processing is the Set Intersection Processor (SIP). It is a two-dimensional homogeneous structure of size $m_1 \times m_2$ (Figure 3), each cell of which contains an equivalence circuit, a response flip-flop, and some additional logic, which is needed for implementing in this cell the sequential bit-wise comparison of the corresponding elements of argument arrays M_1 and M_2. After completing the comparison cycle of n steps, where n is the element length, in the two-dimensional response field of SIP the resulting Binary Label Matrix (BLM) is formed.

The presence of "1" in the (i,j)-th node of BLM means that the i-th element of the array M_1 coincides with the j-th element of the array M_2.

The SIP is a quasi-associative processor with a higher level of parallelism compared to conventional associative processors. Whereas in the conventional quasi-associative processors all elements of an argument array coinciding with *one comparand* are singled out during one cycle of memory interrogation, in SIP *a complete intersection* of two arrays is realized at the same time.

The set intersection processor can be used in different hardware modules. Supporting the PT related to non-numerical problems, it can serve as a powerful parallel comparator. Another possible application of this device is in table-lookup computations. In this case, the proper functional table is stored in MU1 while a number of values of current arguments are loaded into MU2. It can be easily shown that in coarse of a single cycle of memory interrogation, a set of values of a given function corresponding to all the arguments of MU2 will be produced in parallel.

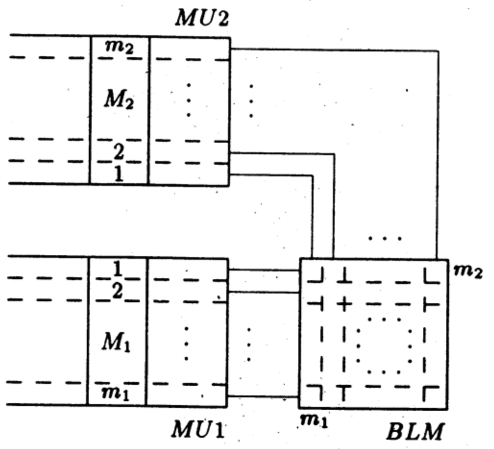

Figure 3. Set Intersection Processor

The SIP can be used in realizing the processing types VVL, VVS.

4.11. Functional Converter

These computations are based on a rather unusual property of the λ-matrix.

Consider a λ-matrix of size $m \times n$ arranged as in Figure 4. If some binary vector **F** is fed to the inputs z of the buttom row of λ-cells, then, in accordance with the basic algorithm of operation of the λ-matrix (see 4.7), each next "one" of this vector marks by a unique logic condition ($z\bar{t} = 1$) a single cell of the next row of the λ-matrix corresponding to its coordinate.

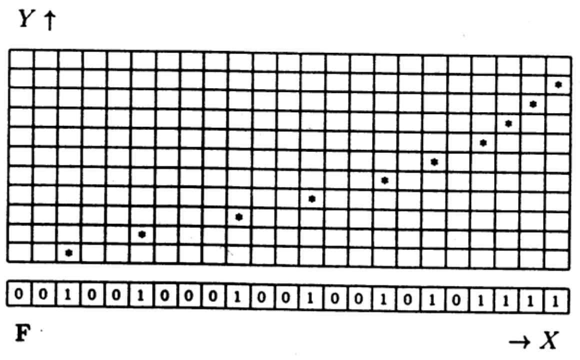

Figure 4. Functional mapping in λ-structure

Suppose a n-bit binary vector **F** represents the "increment flow" of some continuous function $f(x)$, that is, corresponds to the code of the step function approximating the given function. Clearly, in this case the marked cells (asterisks in Figure 4) are tracing, as it were, a representation of the function's graph on the coordinate grid formed by the cells of the λ-matrix.

The feature described above allows to realize, on the basis of a λ-structure, various non-conventional, quasi-analogue devices for computation of functions, A/D and D/A conversions, integration, solving of some equations, etc. The efficiency of these computations is determined by the fact that they are performed by the propagation of signals through the combinational circuit of the λ-matrix, and the results are obtained immediately after the termination of the transition processes.

5. Programming of CHS

5.1. Technique of problem solving

Prior to be executed in suggested CHS, a problem needs definite preparations taking into account the features of this system. The main phases of this work are shown in Figure 1. They are:

1. Design of a "coarse-grained" algorithm by means of decomposition of the given problem into separated procedures (steps), thinking of the PT classification (Table 1).
2. Justification of this algorithm by comparing the required PTs with the assortment of the available HMs of specific CHS.
3. Assigning of PTs to corresponding HMs. If there is no direct hardware support for some PT (absence of corresponding PT), this PT is realized either by combining operation of several existing HMs, or by the resources of the host.
4. Optimization of the execution scheme as a whole, and assembling of the final program taking into account the necessary consistency of data structures and dimensionalities in various procedures, as well as the synchronization of HMs operation.

Evidently, the CHS software should also provide for designing host programs responsible for the control and data exchange in the system.

In this Section the main principles are discussed of a language for the description of algorithms in CHS environment. Such a language should reflect the features of the basic fine-grained SIMD subsystem, as well as of its extension by the set of coprocessors. Some constructs of such language called VEPRAN were described in [21]. Here, we propose an extension of VEPRAN taking into account all requirements of the combined architecture and the classification of PTs.

5.2. A program model of CHS

The fine-grained SIMD architecture is notable for a huge number of simple single-bit processing elements (PEs) working synchronously under the control of a common program unit. Each PE has its own one-bit word local memory, executes bit-sequential data processing, and communicates with other PEs via interconnection network. The aggregate memory of the system can be considered as a bit matrix each row of which is connected with a corresponding PE. Then, the proper processing presents a sequence of bit operations on the binary vectors (slices) fetched/loaded from/in memory.

Three types of slices are usually distinguished in SIMD systems: **vertical** (columns), **word** (segments of a row), and **complex** (different from the first two). Most frequently are used the vertical slices, or simply **slices**. That is why we call such architectures Vertical Processing Systems [22]. A submatrix formed from adjacent vertical slices is called a **field**. In accordance with the type of processed data, the fields may contain numerical vectors, matrices, relational tables, etc.

Each PE contains necessary logic circuits and several flip-flops to store the intermediate results of the bit operations. All PE flip-flops of the same name form specific bit slices considered as programming operational registers (PORs). The input and output buses of the HMs, included in the CHS, can be viewed in the same way. Thus, the extension of the basic SIMD system by coprocessors, from the programmer's point of view, can be considered as enlarging of the number of PORs saving the style of SIMD computations.

5.3. VEPRAN-language

In VEPRAN language, suggested for the description of algorithms, not only the data types like **integer**, **index** (unsigned), **float**, **double**, **char**, **logical**, **structure**, but also their location in the system is used. So, the declaration **scalar** implies that data are placed in the control memory, while declarations **slice** (locate a slice) and **field** (locate a field) serve for the description of data placed in the parallel memory.

The modes of data arranging within the fields of the parallel memory are declared by the following language construct:

place $\langle object \rangle$ **in field** $<$ *field* [*elements*] **by coord** $\langle c_1, c_2, \ldots, c_n \rangle$.

The following example:

scalar index n;
integer $A[n, n]$;
field integer C;
place $A[n, n]$ **in field** $C[n * n]$ **by coord** 1;

implies that the matrix A should be placed by columns in n^2 elements of an one-dimensional field C. The next instructions are an example of placement in the parallel memory of n rows of a relational table:

scalar index n, l;
structure *Person*[n]{**char** *Name*[8], **logical** *Sex*,
  index *Age*, **char** *City*[8]};
index *AgeData*[l];
field structure *PersonTable*{**field char** *Name*[8], **slice** *Sex*,
  field index *Age*, **field char** *City*[8]};
place *Person*[n] **in field** *PersonTable*[n] **by coord** 1;

In this case, the operator

place *AgeData*[l] **in field** *PersonTable.Age*[n] **by coord** 1;

can be considered as placing of l values of the attribute *Age* in the subfield *Age* of the field *PersonTable*.

The operations on scalar data are indicated by a sign ":=". Parallel operations are indicated by "$\leftarrow$", identified with corresponding PT, and belong to the following modifications:

$\langle dest \rangle \leftarrow \langle \mathbf{PT} \rangle$ - for all elements of field;
$\langle dest \rangle \leftarrow [slice] \leftarrow \langle \mathbf{PT} \rangle$ - for elements unmasked by *slice*;
$\langle dest[index] \rangle \leftarrow \langle \mathbf{PT} \rangle$ - for selective loading.

The control constructs in VEPRAN:

```
do ⟨operations⟩ enddo;
if (⟨condition⟩) then ⟨do − part⟩ else ⟨do − part⟩;
for (⟨operation⟩, ⟨condition⟩, ⟨operation⟩) ⟨do − part⟩;
while (⟨condition⟩) ⟨do − part⟩;
procedure (⟨formal parameters⟩);
call (⟨real parameters⟩);
```

are similar to those used in traditional languages. It should be noted only that the statement $\langle condition \rangle$ may include parallel operations and be identified with PT.

5.4. Example CHS algorithms

Consider a well-known problem of matrix multiplication $\mathbf{C} = \mathbf{AB}$.

To solve this problem, n^2 inner products should be computed (where n is the order of the matrices). In our case, the inner product is a basic PT denoted **VVA** in Table 1. The detailed elaboration of the algorithm depends on the relationship between the order n and the level of system parallelism (that is the number of processing channels) m [23]. If $m \geq n^2$, simultaneous computing is possible of n inner products, and the CHS algorithm takes the form shown in Figure 5.

If $m \geq n$, the parallel computation can involve only one inner product, which is reflected by the version of CHS program shown in Figure 6. From these examples, it is easy to pick out the necessary PTs which can be implemented as specialized hardware modules.

As it was shown in [24], when using only the basic SIMD computer with an interconnection network of the Flip, or Hypercube type, the execution of the first algorithm requires $O_A(n \log_2 n)$ arithmetic operations and $O_B(n(s^2 + s \log_2 n))$ bit operations (where s is the wordlength of the data). The second algorithm needs $O_A(n^2 \log_2 n)$ and $O_B(n^2(s^2 + s \log_2 n))$ operations, correspondingly. This is because the procedures with the PTs $\mathbf{VVA}[\ldots, \ldots, *]$ and $\mathbf{VoA}[\ldots, sum(\ldots, n)]$ are realized in the basic architecture in $O_B(s^2)$ and $O_B(s \log_2 n)$ correspondingly.

The use of strongly specialized HMs can sufficiently improve these estimations. Thus, in [21] was shown that application of coprocessors PVM and PVA (see Section 4) ensures execution of the PTs **VVA** and **VoA** in $O_B(s)$ and $O_B(s + \log_2 n)$ bit operations. Moreover, one can organize a programmed "chaining" of these two HMs, thus supporting the pipelining of computations. In this case, the described algorithms can be implemented in $O_B(n(s + \log_2 n))$ and $O_B(n^2(s + \log_2 n))$ bit operations, correspondingly.

```
procedure MulMatr1($A, B, C, n$);
/* Algorithm 1: matrix multiplication for $m \geq n^2$ */
    scalar index $n$;
    integer $A[n,n], B[n,n], C[n,n]$;
    scalar index $k$;   /* the work index */
    integer field $A, B, C, D$;   /* define fields */
    slice $S$;   /* define work slice */
    place $A[n,n]$ in field $A[n*n]$ by coord 2;  /* locate $A$ by rows */
    place $B[n,n]$ in field $B[n*n]$ by coord 1; /* place $B$ by columns */
    $S \leftarrow$ BBL[$S, 0, AND$];  /*clear slice of masks */
    for ($k := 0$;  $k \leq n-1$;  $k := k+1$)  /* set masks*/
        do $S[k*n+k] \leftarrow$ BBL[$S, 1, OR$]; enddo
    for ($k := 1$;  $k \leq n$;  $k := k+1$)
        do
            $D \leftarrow$ VVA[$A, B, *$];  /* $D \leftarrow A * B$ */
            $D \leftarrow$ VoA[$D, sum(n,n)$];  /* compute $n$ sums */
            $C \leftarrow [S] \leftarrow$ VoP[$D, none$];  /* save $n$ elements of $C$ */
            $B \leftarrow$ VoP[$B, cshift(n*n, n)$];  /* exchange columns of $B$ */
            $S \leftarrow$ BoP[$S, cshift(n*n, n)$];  /* exchange bits of mask $S$ */
        enddo
end procedure
```

Figure 5. Matrix multiplication in CHS for $m \geq n^2$

6. Conclusion

In this paper a new approach is suggested to the organization of heterogeneous computing. This approach is based on the conception of combined architecture, which is a composition of a fine-grained SIMD computer with a set of high-performance strongly specialized processors.

The presence of a number of accelerators of various architectures at one site allows to organize heterogeneous computing within a single system. In contrast to the existing distributed heterogeneous systems, the proposed concentrated heterogeneous system does not need for high-bandwidth communications networks, and does not suffer from the delays arising in these networks at the data transfer. The accelerators (hardware modules) of the combined architecture are much cheaper than complete supercomputers and, at the same time, they can be better oriented to the necessary programming styles.

To optimize the choice of hardware modules, a classification of massive computations is suggested, based on the notion of processing types, which correspond to the character of data processing and are applicable for hardware implementation of different classes of problems. Several examples of efficient hardware modules of diverse architecture are listed.

A technique of parallel programming for the combined architecture is introduced based on the language VEPRAN extended by appropriate means to specify the processing types and to control the movement of parallel data in the given concentrated heterogeneous system. Using this language, one can analyze the algorithm of solution of any given problem and present it as a sequence of processing types. In course of the execution of such a program, each processing type is as-

```
procedure MulMatr2($A, B, C, n$);
/* Algorithm 2: matrix multiplication for $m \geq n$ */
    scalar index $n$;
    integer $A[n,n], B[n,n], C[n,n]$;
    scalar index $i$, $j$; /* the work indices */
    structure field $A[n * $(integer field)]; /* define field $A$*/
    structure field $B[n * $(integer field)]; /* define field $B$*/
    structure field $C[n * $(integer field)]; /* define field $C$*/
    integer field $D$; /* define work field $D$*/
    slice $S$; /* define work slice */
    place $A[n,n]$ in field $A[n,n]$ by coord 2; /* locate $A$ by rows */
    place $B[n,n]$ in field $B[n,n]$ by coord 1; /* locate $B$ by columns */
    $S \leftarrow$ BBL[$S, 0, AND$]; /*clear slice of masks */
    do $S[0] \leftarrow$ BBL[$S, 1, OR$]; /* set masks*/
    for ($i := 1$; $i \leq n$; $i := i + 1$)
        for ($j := 1$; $j \leq n$; $j := j + 1$)
            do
                $D \leftarrow$ VVA[$A[i], B[j], *$]; /* $D \leftarrow row_i(A) * column_j(B)$ */
                $D \leftarrow$ VoA[$D, sum(1, n)$]; /* compute sum */
                $C \leftarrow [S] \leftarrow$ VoP[$D, none$]; /* save $c_{ij}$ in $row_i(C)$ */
                $S \leftarrow$ BoP[$S, cshift(n, 1)$]; /* cyclic shift of $S$ in one bit*/
            enddo
end procedure
```

Figure 6. Matrix multiplication in CHS for $m \geq n$

signed to a definite hardware module (or a combination of some modules) taken from the set of real modules included in the system.

The combined architecture is considered as an open system which can be supplemented by additional hardware modules, according to the requirements of the user.

The suggested approach is expected to help to design a family of cost-effective supercomputers providing flexible programming, as well as high performance in a broad range of applications.

References

[1] A.A. Khokhar, et al., *Heterogeneous Computing: challenges and opportunities*, Computer, Vol. 26, No. 6, 18–27, 1993.

[2] R.F. Freund and H.J. Siegel, *Heterogeneous processing*, Computer, Vol. 26, No. 6, 13–17, 1993.

[3] B.M. Boghosian, *Computational physics on the Connection Machine*, Computers in Physics, Vol. 4, No. 1, 14–33, 1990.

[4] A.P. Vazhenin, S.G. Sedukhin, Ya.I. Fet, *High-performance computing systems of combined architecture*, In: "Parallel Computing Technologies (PaCT-91)", Novosibirsk, Russia, 1991 (N.N. Mirenkov, ed.), World Scientific, Singapore, 246–257, 1991.

[5] Ya.I. Fet and A.P. Vazhenin, *Heterogeneous processing: a combined approach*, In: "Workshop on Parallel Scientific Computing (PARA'94-L)", 1994, Lingby, Denmark, Lecture Notes in Computer Science, Vol. 879, Berlin, Springer-Verlag, 194–206, 1994.

[6] L.V. Kantorovich, *On a system of mathematical symbols, convinient for electronic computer operations*, Dokl. Akad. Nauk SSSR, Vol. 113, 738–741, 1957 (In Russian).

[7] K.E. Iverson, *A Programming Language*, New York–London, Wiley, 1962.

[8] Ya.I. Fet, *Hardware support of massive computations*, Optimization, Novosibirsk, Inst. of Mathematics, Siberian Div. of the USSR Acad. Sci., No. 22(39), 115–126, 1978 (In Russian).

[9] G.E. Blelloch, *Vector Models for Data-Parallel Computing*, Cambridge, Mass., MIT Press, 1990.

[10] H.T. Kung, *Why systolic architectures?*, Computer, Vol. 15, No. 1, 37–46, 1982.

[11] S.G. Sedukhin and I.S. Sedukhin, *An interactive graphic CAD tool for the synthesis and analysis of VLSI systolic structures*, Parallel Computing Technologies (PaCT-93), Obninsk, Russia, 1993 (V.E. Malyshkin, ed.), Moscow, ReSCo J.-S. Co., 163–175, 1993.

[12] G. Broomel and J.R. Heath, *Classification categories and historical development of circuit switching topologies*, ACM Computing Surveys, Vol. 15, No. 2, 95–133, 1983.

[13] W.D. Hillis and L.W. Tucker, *The CM-5 Connection Machine: a scalable supercomputer*, Comm. ACM, Vol. 36, No. 11, 31–40, 1993.

[14] D. Knuth, *The Art of Computer Programming*, Vol. 3, Sorting and Searching, New York, Addison-Wesley, 1973.

[15] K.E. Batcher, *Sorting networks and their applications*, In: AFIPS Confer. Proc., 1968 SJCC, Vol. 32, 307–314, 1968.

[16] C.C. Foster, *Content Addressable Parallel Processors*, New York, Van Nostrand Reinhold, 1976.

[17] E. Ozkarahan, *Database Machines and Database Management*, Englewood Cliffs, Prentice-Hall, 1986.

[18] Ya.I. Fet, *Parallel Processing in Cellular Arrays*, Tounton, UK, Research Studies Press, 1995.

[19] Ya.I. Fet, *Digital compessors*, In: Proc. of the VI Int. Workshop on Parallel Processing by Cellular Automata and Arrays (PARCELLA'94), Potsdam, Germany, 1994, Berlin, Akademie Verlag, 13-25, 1994.

[20] Ch. Fernstrom, I. Kruzela, B. Swensson, *LUCAS Associative Array Processor: Design, Programming and Application Studies*, Lecture Notes in Computer Science, Vol. 216, Berlin, Springer–Verlag, 1986.

[21] A.P. Vazhenin, *Hardware and algorithmic support of high-accuracy computations in vertical processing systems*, In: Parallel Computing Technologies (PaCT-93), Obninsk, Russia, 1993 (V.E. Malyshkin, ed.), Moscow, ReSCo J.-S. Co., 149–162, 1993.

[22] Ya.I. Fet, *Vertical processing systems: a survey*, IEEE Micro, Vol. 15, No. 1, 2–12, 1995.

[23] W.F. Tichy, *Parallel matrix multiplication on the Connection Machine*, International Journal of High Speed Computing, Vol. 1, No. 2, 247–262, 1989.

[24] A.P. Vazhenin, *Efficient high-accuracy computations in massively parallel systems*, In: "Workshop on Parallel Scientific Computing (PARA'94-L)", 1994, Lingby, Denmark, Lecture Notes in Computer Science, Vol. 879, Berlin, Springer–Verlag, 505–519, 1994.