Evaluation of Biological Sequence Similarity Using FPGA Technology

Tomáš Martínek*
Department of Computer Systems
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
martinto@fit.vutbr.cz

Abstract

Understanding structure and function of DNA sequences represents one of the most important goals in area of modern biology research. However, algorithms for DNA analysis are usually complicated by mutations caused by an evolution process, which are usually occurred in form of character insertion, deletion and substitution. With respect to these defects the time complexity of the algorithms grows and limits their practical usage. Techniques for an acceleration of the key operations using specific circuits bring a certain expectation into this area. The designed circuits are able to achieve the speedup in orders of thousands in comparison to the most powerful conventional processors. Despite of huge performance of these circuits they are not used in wide range of real word applications. The main reason lies in often variability of input tasks that leads to change of architecture sizes with respect to the target platform properties. These modifications usually require an intervention of experienced designer and thus complicate their practical usage. The objective of this thesis is therefore dedicated to design and implementation of novel methods for automated mapping of circuits into the chips with FPGA technology. At first the problem of mapping is investigated into the detail and formally defined. Based on the formal model a novel mapping technique is designed. In comparison to the previous approaches, this technique is capable to find the optimal sizes of the resulting circuit and it does not limit description of the architecture parts to linear functions only.

*Recommended by thesis supervisor:
Prof. Václav Dvořák
Defended at Faculty of Information Technology, Brno University of Technology on December 2, 2010.

Martínek, T.: Evaluation of Biological Sequence Similarity Using FPGA Technology. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 2, No. 2 (2010) 93-102

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Reliability and Testing-automatic synthesis, hardware description languages, optimization

Keywords

Approximate string matching, approximate palindrome detection, approximate tandem repeat detection, systolic array, programmable hardware, FPGA technology, nested loops mapping, design space exploration, high-level synthesis

1. Introduction

Understanding structure and function of DNA sequences represents one of the most important goals in area of modern biology research. Currently, the scientists have a huge amount of data covering the different kind of species including their various development stages. However, the detailed analysis of these data is complicated by the presence of mutations, which occurs as a consequence of an evolution process. At the lowest level they are usually observed in form of character insertion, deletion or substitution. These defects increase the time complexity of algorithms for sequence analysis and thus complicate their practical usage. To the examples of often used operations in this area belong: sequence alignment and searching for specific secondary structures such as palindromes, tandem repeats, triplexes, quadruplexes etc. This problem becomes more significant in the light of the novel sequencing technologies. While the first human genome assembling took a decade, the new nanopore or SMRT (*Single Molecule Real Time sequencing*) techniques are able to generate billions of base-pairs per minute at cost of hundreds of dollars.

One of the possibilities, how to reduce this problem and achieve the required performance for biological sequence analysis and its filtration lies in utilization of specific hardware circuits. In the last two decades a lot of works [4, 11] were dedicated for acceleration of the one of the most important operations – *sequence alignment*. Several architectures were designed, some of them were implemented to chips with ASIC or FPGA technology and they showed a speedup in orders of hundreds or thousands in comparison to the most powerful conventional processors. Despite of such speedup these accelerators are not widely used in real word applications. One of the main reasons lies in often variability of the tasks using the sequence

alignment operation. This operation distinguishes for example: type of input sequence (DNA vs. proteins), score function, type of alignment (local vs. global) or lengths of both input sequences.

All these parameters affect the sizes of resulting circuit, amount of consumed resources and working frequency. In case that the user has a certain FPGA chip containing limited number of configurable gates, then the question is how to select the circuit sizes to spend computation resources effectively and achieve the maximal speedup for a concrete target application. Unfortunately, existing tools are not able to solve this task and an arbitrary change requires an intervention of experienced designer. This motivated some of researches and the first approaches for automated mapping of circuits with respect to the user application requirements and the target platform parameters were developed [1, 10]. However, none of them is able to find the optimal circuit sizes and they are still subject of research.

The goal of this thesis is therefore to investigate the state-of-the-art methods in this area, identify their weaknesses and design a novel approach. The rest of this paper is organized as follows: Section 2 describes the state-of-the-art methods in this area and identifies their drawbacks. Formal definition of mapping problem including a novel method for searching of optimal architecture sizes is a part of section 3. Finally, the proposed method is evaluated on an example of circuit for sequence alignment (see section 4).

2. State of the art

Techniques for automated mapping of algorithms into the chips with ASIC or FPGA technology belong to area of *High Level Synthesis*. Selection of the appropriate method primarily depends on character of computation. The detail inspection of algorithms for DNA sequence analysis shows, that they usually represent computations inside nested loops. The following part of the text is therefore focused on description of the basic approaches for synthesis of nested loops and summarizes the state-of-the-art methods in this area.

The objective of the nested loops synthesis is to find a suitable mapping of the computation inside the loop on the available computation elements (or processors). An interesting approach, how to represent and partially solve this task is based on integer linear programming theory (ILP). As an input an arbitrary number of loop nests is assumed (e.g. *for* cycles), where the inner loop body is composed of arbitrary number of equations in form:

$$x_i = f_i(\dots, x_j[I - d_{ji}], \dots) \tag{1}$$

where $I \in Z^r$ is a value of index variable, r is the number of loop nests, $x[I]$ is a value of variable x at index I, f_i is an arbitrary function and $d_{ij} \in Z^r$ is a vector of data dependencies between variable x_i and x_j. If all loops are unrolled they represent *index space (IS)*, i.e. the space of all indexes where the loop body has to be evaluated. If boundaries of individual loops are limited and not calculated dynamically during the computation, then IS represents *polytop* or *polyedr* in r-dimensional space. The data dependencies play one of the most important role during the mapping process. If they are supplemented into the index space, we achieve *data dependency graph* $G = (V, E)$ where V is set of vertexes (corresponding to the points of index space) and E is set of edges representing data dependencies between vertexes.

2.1 Scheduling and allocation

The objective of scheduling is to assign a moment $t(I)$ for each index space point I, which it will be processed in. However, during the schedule construction, all data dependencies have to be observed according the above graph. Formally, function $t : Z^r \to Z$ is *schedule* if it satisfies condition 2.

$$(\forall I, I' : I, I' \in IS \wedge (I, I') \in E : t(I) < t(I')) \tag{2}$$

The objective of allocation is to assign a number of processing units for each index space point, which will process it. Concurrently, it is assumed that each index space point is processed by just one processing unit. The allocation function has to be designed with respect to existing schedule because each computation unit can process only single point at the time. Formally, function $a : Z^r \to Z^{r-1}$ is *allocation* with respect to the schedule t if it satisfies condition 3.

$$(\forall I, I' : I, I' \in IS : t(I) = t(I') \Rightarrow a(I) \neq a(I')) \tag{3}$$

The searching of schedule and allocation functions is not trivial task. They have to characterize not only valid functions (with respect to the data dependencies) but quality functions as well. Unfortunately, a number of possible schedules and allocations corresponds to index space size. However several approaches [3] for reducing this space and finding the best solution exist.

2.2 Partitioning

The highest level of parallelism and speed-up of an algorithm is achieved, when the r-dimensional index space is calculated using $r - 1$ dimensional array of computation units. The rest single dimension is time. However, there are not so many computation units in real chips and therefore $r - 1$ dimensional array is called as *virtual processor array*, which is mapped into the *physical processor array*, often limited to 1D or 2D structures. Two basic techniques used during the mapping process are:

- *Local parallel, global sequential (LPGS)*: An array of computation units calculates a part of the index space simultaneously. As this part is finished then the whole array is shifted to another part. This technique is called *tiling* and the given part of index space is called as a *tile*.
- *Local sequential, global parallel (LSGP)*: The index space is split into the several parts where each part is processed by single processing unit sequentially. Then all computations units operate in parallel to calculate the whole index space. This technique is called *clustering* and the given part of index space is called as a *cluster*.

This partitioning process affects the original task of scheduling. It is composed of three major steps: 1) Selection of the partitioning scheme LPGS, LSGP or their combination. 2) Selection of the direction for index space computation. 3) Searching for the optimal sizes of the tile or cluster with respect to required criterion.

Please note that all three parts of scheduling process are related to each other and unsuitable selection in one step automatically affects all other parts. Constrained scheduling is generally NP complete task which has not been solved in a satisfactory manner and it is still subject of research. Published algorithms are usually capable to search a suitable cluster size for a certain type of circuit, however they assume a lot of simplifications and thus they are applicable to limited area of target applications only. Moreover, often used ILP model assumes architecture described using linear functions, which is not usually satisfied in the case of circuits mapped to the FPGA technology. An interesting alternative approach was described in [10], where linear functions are replaced by more general monotonic functions that reflects properties of real components better. However, it has not been formally defined so far and it is not capable to find the optimal cluster or tile size in other way then by using exhaustive computation.

3. Method

Based on the knowledge about state-of-the-art methods in area of nested loops synthesis, we can review the possibilities of techniques for automatic mapping of architectures for DNA sequence analysis, which were described in [8]. Please note that for all designed architectures for sequence alignment, searching of approximate palindromes and tandem repeats, the appropriate partitioning scheme (LPGS, LSGP or combination) and direction of computation were selected. The remaining step is to find the optimal cluster or tile size capable to achieve the lowest computation time or other criterion required by user (i.e. the last step of scheduling process). The objective of this thesis is therefore to follow the approach published in [10] and express the dependencies for amount of consumed resources and computation time with respect to cluster or tile size in form of general functions. For these purposes a novel model called *parametrized architecture* is introduced. Using this model the task for searching of optimal architecture size is defined formally and a novel effective method is designed.

3.1 Parametrized architecture

As an input a template of the circuit corresponding to selected partitioning scheme is assumed. This template can be obtained as a result of automated process for synthesis of nested loops or it can be obtained based on existing architecture designed by experienced designer. Variable parts of the circuit corresponding to the cluster of tile size are specified using *architecture variables* $N_1, ..., N_k$. Except of these variables the sizes of resulting architecture depend on *input task parameters* such as maximal lengths of analyzed sequences, number of the task queries, data width for character representation, etc. and *target platform parameters* such as amount of available resource, input and output bandwidth etc. Both of these groups of parameters will be called *architecture parameters* $P_1, ..., P_l$and it is usual that all of them are known before the synthesis process in form of constants. To find the optimal sizes of architecture, it is necessarily to express

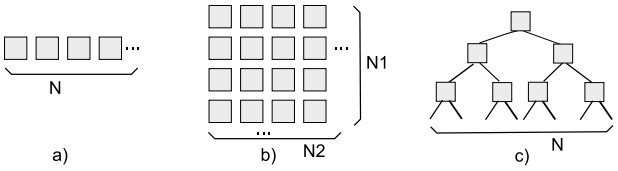

Figure 1: Examples of architectures: (a) linear/systolic array, (b) grid (c) tree structure

relations for amount of consumed resources and computation time with respect to variables $N_1, ..., N_k$ and parameters $P_1, ..., P_l$. At first, we will concentrate on amount of consumed resources.

A model of arbitrary circuit can be described using tree structure where lists represent elementary components and edges express hierarchical relations between components and sub-components. If a certain part of the circuit contains an array of variable size, then the appropriate edge is evaluated by the *architecture variable* N_i representing number of instances of sub-component inside the superordinate component (see examples of linear array and grid architectures on figure 1). In general, the number of instances may not be expressed by variable only but by arbitrary function of variables $N_1, ..., N_k$ and parameters $P_1, ..., P_l$ (see example of tree structure on figure 1c, where overall number of nodes corresponds to $2N - 1$, where N represents the number of lists).

Each of elementary components requires a certain amount of resources for its realization. Therefore, the lists are evaluated by the number representing this amount. Similarly to edges, the amount of resources may not be expressed by scalar value only but by function of variables $N_1, ..., N_k$ and parameters $P_1, ..., P_l$ in general. If more than one kind of resources is used for the circuit realization, then a function is reserved for each type. Thus the evaluation of list is expressed by m-tuple $g = (g_1, ..., g_m)$ of functions in general. The circuits mapped to FPGA technology are typical examples, where look-up tables (LUTs), registers, BlockRAMs, embedded multiplier etc. are considered as a different types of resources.

Definition: (Tree structure of parametrized architecture)
A tree structure of parametrized architecture with k variables realized in technology with m kinds of resources is defined as oriented tree with evaluation of lists and edges. It is 6-tuple $S_{km} = (V, H, F, G, c, d)$ where:

- V is a set of vertexes representing individual components and sub-components of the architecture,
- H is a set of ordered pairs $\{(u, v)\}$ such that $(u, v) \in H$ when v is sub-component of component u,
- F is a set of functions $\{f_i | f_i : Z^k \to Z\}$ representing numbers of instances of sub-components within superordinate components.
- G is a set of ordered m-tuples $\{g_j = (g_{j_1}, ..., g_{j_m})\}$ where each item of m-tuple is function $g_{j_l} : Z^k \to Z$ for all $l \in \{1, ..., m\}$. This set of m-tuples is used for evaluation of lists where each function represents amount of consumed resources of certain kind of resource $1...m$.

- c is a mapping $c : H \to F$ where just one function $f \in F$ is assigned for each edge $h \in H$ (evaluation of edges),
- d is a mapping $d : V \to G$ where just one function $g \in G$ is assigned for each list $v \in V$.

Please note that internal nodes are not evaluated. Amount of resources for their realization can be expressed as a sum of resources of all their sub-components. Amount of resources for arbitrary node of the tree can be expressed recursively as:

$$R(u) = \begin{cases} d(u) & \text{list} \\ \sum_{\forall(u,v_i)\in H} c((u,v_i)) \times R(v_i) & \text{internal} \end{cases} \quad (4)$$

If the above equation is applied to the root, then it represents a general function for computation of amount of resources for the whole architecture. Note that the equation is composed of linear combination of functions $Z^k \to Z$ and the result is also m-tuple of functions.

Except of amount of resources it is necessarily to know the relation for index space computation time with respect to certain size of cluster or tile. This time can be expressed as a function f_T of variables $N_1, ..., N_k$ and parameters $P_1, ..., P_l$ in general.

During the optimal cluster or tile size searching it is necessarily to respect certain restrictions of variables $N_1, ..., N_k$. The first group of restrictions results from ranges of original algorithm loops and represents the fact that it does not make sense to create the bigger cluster or tile than the size of original space. The second group of restrictions depends on target platform properties. To typical examples belong limited amount of computation resources and limited input/output bandwidth, which allows to feed up only limited number of processing elements with the new data or to process the data from a certain number of elements on the opposite side. Neither in this case, it does not make sense to built bigger cluster or tile. All these restrictions will be called as *constraints* and they can be expressed as a set of inequations O composed of functions of variables $N_1, ..., N_k$ and parameters $P_1, ..., P_l$.

Now, we have all necessary relations for optimal cluster of tile size searching. Formally, they are summarized in the following definition of *parametrized architecture*.

Definition: (Parametrized architecture)
Parametrized architecture with k variables realized in technology with m kinds of resources is defined as triplet $A_{km} = (f_R, f_T, O)$ where:

- $f_R = (f_{R_1}, ..., f_{R_m})$ is ordered m-tuple of functions representing amount of resources for the whole architecture with respect to variables $N_1, ..., N_k$ where each item is in form $f_{R_l} : Z^k \to Z$ for all $l \in \{1, ..., m\}$,
- f_T is function $f_T : Z^k \to Z$ representing computation time (in number of steps) with respect to variables $N_1, ..., N_k$,
- O is set of constrained conditions including restrictions to loops ranges as well as restrictions defined by target platform properties. All conditions $o_i \in O$ are described in form of inequations $o_i : Z^k \to Z \; op \; 0$ of variables $N_1, ..., N_k$where $op \in \{>, \geq\}$.

3.2 Mapping of parametrized architectures

A lot of possible instances (sizes of clusters and tiles) can exist for a certain parametrized architecture and capable to satisfy all constrained conditions. All of these instances form space of all realizable circuits – *Design Space* and each of them is uniquely identified by values of variables $N_1, ..., N_k$. At first, lets define the conditions that have to be satisfied for a circuit to be included into this space. Lets $A_{km} = (f_R, f_T, O)$ is parametrized architecture and $P_1, ..., P_l$ are input parameters, then an instance of the circuit defined by the values of variables $(n_1, ..., n_k)$ is included into the design space if the following conditions are satisfied:

- Each component has to consume a positive amount of resources. As amounts of resources of list components are generally expressed by the functions, some of them can achieve a negative values on certain intervals. However, the component with negative amount of resources is not realizable and thus it can not be a part of the design space.

$$g_{i_j}(n_1, ..., n_k) > 0 \text{ pro } \forall g_{i_j} \in g_i \text{ and for } \forall g_i \in G \quad (5)$$

- The results of functions representing the number of instances of sub-components within superordinary components have to be non-negative integer. Similarly to previous condition, the circuit containing negative number of instances of some component is not realizable and thus it can not be a part of design space.

$$f_i(n_1, ..., n_k) > 0 \text{ for } \forall f_i \in F \quad (6)$$

- Amount of resources for the whole circuit can not exceed amount of available resources on the chip $R = (r_1, \ldots, r_m)$ and thus the following condition has to be satisfied:

$$f_{R_i}(n_1, ..., n_k) \leq r_i \text{ for } \forall i \in \{1, \ldots, m\} \quad (7)$$

- Set of constrained conditions has to be satisfied:

$$o_i(n_1, ..., n_k) \text{ have to be satisfied for } \forall o_i \in O \quad (8)$$

If all above defined inequations 5 up to 8 are put together and modified such that they contain zero on the right side and operation $>$ resp. $\geq$ only, then we will obtain the following set of inequations:

$$\begin{array}{rcll} g_{i_j}(n_1, ..., n_k) & > & 0 & \text{for } \forall g_{i_j} \in g_i \text{ and } \forall g_i \in G \\ f_i(n_1, ..., n_k) & > & 0 & \text{pro } \forall f_i \in F \\ r_i - f_{R_i}(n_1, ..., n_k) & \geq & 0 & \text{for } \forall i \in \{1, \ldots, m\} \\ o_i(n_1, ..., n_k) & & & \text{for } \forall o_i \in O \end{array} \quad (9)$$

All solutions of the set 9 represent the design space. The objective of the mapping process is to find a circuit sizes that will not only satisfy the set 9 but they will meet the input criterion the best. Therefore it is necessary to apply an *evaluation function* to the design space and select a solution corresponding to its maximum or minimum. As the objective is to find the circuit capable to process the index space as fast as possible in most cases, the function f_T will be selected as evaluation function.

A task of searching of function extreme $f(x)$ on interval bounded by functions $g_i(x)$ belongs to area of *optimization problems on constrained space*, which are usually described in form 10.

$$\begin{array}{c} P : \{x | g_i(x) \geq 0 \text{ resp. } g_i(x) > 0\} \\ x \in P : f(x) = max[f(z)] \quad \forall z \in P \end{array} \tag{10}$$

An approach for solving of this optimization problem primarily depends on properties of used functions. If all of functions f and g_i are linear then this task belongs to area of *linear programming (LP)* and the space of realizable circuits limited by g_i functions forms *polyedr in n-dimensional space*. Searching of extreme of function f on such space operates in two steps: (1) set of inequations is transformed into *standard matrix form* of LP task, (2) maximum of evaluation function is calculated using simplex method.

If at least one of functions f or g_i is nonlinear, then the optimization problem belongs to the area of nonlinear programing and the method for its solution depends on several factors. For example, it is distinguished whether the evaluation function is continuous or not. In case of continuous evaluation function the methods can use its derivation. A typical example of such method is *Newton method* [6], which scans the space in gradient direction until the local extreme is found. In case the evaluation function is not continuous then other methods such as grid method, simulated annealing or group of genetic algorithms have to be used.

Please note that none of above mentioned methods for nonlinear programing guarantee detection of global extreme and even most of them do not guarantee the convergence as well. Before the certain method is selected, it is necessarily to investigate the functions continuum and other properties, which is not possible to do it automatically without intervention of experienced mathematician so far. If the functions used in parametrized architectures for DNA sequence analysis are analyzed in more detail, then we will observe that they are usually nonlinear but monotonic. Detailed description follows.

Functions representing amount of resources of elementary components
: Amount of elementary component resources (list) is usually constant or grows incrementally depending on some of architecture variables $N_1, ..., N_k$. However, in some cases the amount of resources decreases for longer array of processing units.

Functions evaluating the edges
: Numbers of sub-components instances are usually constant or represent simple growing functions e.g. n for systolic array, n^2 for grid (monotonic for $n \geq 0$), $2n - 1$ for tree architecture etc.

Function representing amount of resources of the whole architecture
: It is composed of linear combinations of functions evaluating edges and lists. As the combination of growing and decreasing functions can occur in internal nodes, then the function for the whole amount of resources is not monotonic, but the combination of monotonic functions bounded on input interval.

Function representing the computation time
: With respect to construction of circuits for DNA sequence analysis the array of processing elements extended in some of its dimensions (sequence alignment, palindrome detection) or using more computation cores (sequence alignment, tandem detection) leads to the computation speed-up. This speed-up usually has linear or incremental character, which represents monotonic function.

All of these findings may not be accepted in general and it is always possible to find an architecture violating monotonicity in some of its part. On the other side, if this condition is satisfied for all functions, then it is possible to use a general bisection method in n-dimensional space or create its new variant for detection of the best solution.

3.3 Basic method scheme

The required inputs are: the parametrized architecture $A = (f_R, f_T, O)$, target platform parameters (amount of resources, input/output bandwidth) and task parameters (lengths of sequences, numbers of queries, data widths). As an output the best or group of the best solutions with respect to evaluation function (f_T) is required. Whereas it is supposed that all used functions are monotonic or linear combinations of monotonic functions. Generally the method can be split into two parts:

1. *Design space computation* – the space of all realizable circuits is calculated based on the parametrized architecture, the platform and the task parameters. This space can be represented in form of intervals or sub-spaces in Z^k.
2. *Application of evaluation function* – selected evaluation function is systematically applied on the design space calculated in previous step. Objective is to find the best or group of the best solutions.

Please note that this model represents a general scheme rather. More effective strategy of design space exploration is achieved by connection of both phases. While the first part identifies the sub-space of realizable circuits the second one evaluates it and shows direction for the next subspace computation and its refinement.

3.4 Design Space Computation

As was mentioned in section 3 the space of all realizable circuits is bounded by set of inequations 9. In case that this set is composed of monotonic functions only then the existing bisection method [7] can be used for detection of all solutions. However, the results of detailed analysis show that some of functions are not monotonic

but in form of linear combination of monotonic functions (LCMF) rather. Therefore, in the following part of the text we will try to generalize the method [7] for LCMF. At first, lets introduce a general bisection method [5], which is independent on function f properties. The method is composed of three major steps:

1. Test whether the input space D contains root of function f or not.
2. Splitting of the space D geometrically into two parts D_1 a D_2 such that $D = \{D_1 \bigcup D_2\}$ a $D_1 \bigcap D_2 = \diamond D_1 \bigcap \diamond D_2$ where $\diamond D$ represents boundary of D.
3. Recursive application of the bisection method to sub-spaces D_1 and D_2.

The most important part of the method is the first step – *Root inclusion test*, which identifies whether the certain space contains the root of function or not, or it is not possible to decide at this level. In case of monotonic function it was proofed that the method will operate correctly even if only two states are distinguished: 1) space does not contain root and 2) it is not possible to decide [7].

In case the input inequation is in form $f_{LCMF} > 0$ or $f_{LCMF} \geq 0$ the root inclusion test can be performed for example using algorithm 1. This algorithm utilizes function $fTreeEval$ for computation of lower bound L and upper bound K of LCMF function. Based on these two values it decides whether 1) the space contains the root and does not satisfy the inequation entirely ($K < 0$ or $K \leq 0$) or 2) it does not contain root and satisfies the inequation entirely ($L \geq 0$ or $L > 0$) or 3) it is not possible to decide (in all other cases).

The situation where it is not possible to decide is caused by two factors: 1) function f goes from negative values to positive ones or in the opposite direction, i.e. it contains the root or 2) the error caused during the lower and upper bound computation is too high, the values of bounds predicate the root presence but it is not present. In the first case it is correct situation, where the bisection method has to split the space into two parts and to refine the root position. In the second case it represents a false positive root detection, which will be disclosed in following steps because this error converges to zero with decreasing space size [8].

The main part of the method is described in algorithm 2. At first, the root inclusion test is performed for each inequation in the set. Based on the results the three situations are distinguished:

1. The space does not contain the root for some of inequation, then this space can not be a part of the design space, it does not make sense to continue and the recursive computation is stopped.
2. The space does not contain the root, however all inequation in the set are satisfied. Then this space is a part of the design space entirely, it does not make sense to continue, the recursive computation is stopped and positions of the space are saved.
3. The space does not contain the root for some of the inequation but satisfies the inequation and for the other inequations (at least one) it is not possible to decide. In this case the space has to be split and recursive computation continues until the space diameter will fall under the threshold ϵ.

Input: Space bounded by a and b points, inequation f in form of LCMF
Output:
NO - does not contain root and space does not satisfy inequation f
YES - does not contain root and space satisfies inequation f
UNKNOWN - it is not possible to decide

```
InclusionTest(a, b, f)
begin
    f_max = fTreeEval(f, a, b, MAX);
    if f_max < 0 then # space < 0
        return NO;
    f_min = fTreeEval(f, a, b, MIN);
    if f_min ≥ 0 then # space ≥ 0
        return YES;
    return UNKNOWN;
end
```

Algorithm 1: Root Inclusion Test

Input: Space bounded by a and b points, set of nonlinear inequations f_{set}
Output: List of all intervals between a and b points, where all inequations from f_{set} are satisfied

```
Bisection(a, b, f_set)
begin
    result = STOP;
    foreach f ∈ f_set do
        inside = InclusionTest(a, b, f);
        if inside = NO then
            return [];
        else if inside = UNKNOWN then
            result = CONTINUE;
        end
    end
    if result = STOP then
        return [a, b];
    else
        if (|a, b| > ε) then
            i = LongestAxis(a, b);
            a_m = a;
            b_m = b;
            a_m[i] = b_m[i] = (a[i] + b[i])/2;
            return
            [Bisection(a, b_m, f_set), Bisection(a_m, b, f_set)];
        else
            return [];
        end
    end
end
```

Algorithm 2: Computation of design space bounded by functions in form LCMF

3.5 Application of Evaluation Function

In previous section we defined the algorithm for design space computation. The last step of the optimization problem is to find the best or group of the best solutions in this space with respect to evaluation function f_{eval}.

At first, it is necessary to analyze evaluation function characteristics. As was mentioned in section 3 this func-

tion is monotonic in most cases and with increasing sizes of architecture the overall computation is accelerated. The monotonicity can be effectively utilized during the evaluation process because a maximum or a minimum of arbitrary part of the design space (specified by interval $I = \langle a, b \rangle$) can be calculated in constant time – extremes can occur on boundary points only. Thus a basic version of algorithm for detection of the best solution could operate in following steps: 1) For each part of the design space find the boundary point with maximum or minimum of the evaluation function and calculate this value. 2) If maximum or minimum is better then the best known so far, then replace the items in an output list with this actual sub-space. I case that the actual sub-space is comparable to the best known, then append it into the output list.

So far, we supposed that the design space exploration task is solved in two steps: design space computation and application of evaluation function. The main disadvantage of this approach is that it calculates the whole design space including the intervals, which never became the part of the best solutions. Similarly the evaluation function is applied to all detected sub-spaces. More effective approach is to interconnect both of these steps. While the bisection method finds the candidate intervals, the evaluation function tests its and routes the computation to perspective areas only. Thus the recursive computation is stopped in areas, which can not achieve better solution then the best known so far. Original bisection method is extended with two rules:

1. From two bisected areas it is selected only that one which achieves the better score. If both results are comparable, then both of them are selected.

2. Both of bisected areas are evaluated preliminary and the recursive computation is applied to that one, which has the higher chance to achieve the better score. Computation of the second part is performed if and only if it has a chance to achieve a better score then the best result from the first part.

3.6 Algorithm time and space complexity

An important property of any algorithm is its time and space complexity and a potential comparison to exhaustive computation. In case of optimal architecture size searching task the exhaustive computation means that each point of k-dimensional space bounded by variables $N_1, ..., N_k$ is tested whether it is a part of the design space or not. Next, all points of the design space are evaluated and the best ones are selected as an algorithm output. For imagination, if the k-dimensional space is scanned and the range of each dimension is n, then the time complexity of exhaustive computation is $O(n^k)$.

Lets concentrate on algorithm 2 for design space computation. Number of steps primarily depends on characteristics of used functions and numbers of roots corresponding to set of inequations. Unfortunately, generally it is possible to design such LCMF function, which will cause the presence of the root in each point of the space. Then the proposed method have to investigate each point of the space and the time complexity is even worse than exhaustive computation because of steps necessary to recursive descent from the whole area to the endpoints ($2n^k - 1$ steps). On the other side, the best case occurs if the computation is directed to single point only and other areas are excluded from the design space. Number of bisections corresponds to the length of the way from root down to the certain endpoint in the list, i.e. $log_2(n^k)$ steps. Please note that the interconnection of both phases (design space computations and application of evaluation function) does not affect the number of steps for the best and the worst case.

However, if the method is applied to a real task, none of above mentioned extremes happens and the method usually achieves better results in comparison to exhaustive computation. As the real value of the time and space complexity depends on used function, we decided to measure it experimentally on artificial and real tasks described by sets of inequations. As the artificial tasks the tree sets of different combinations of stair, parabolic, hyperbolic and other functions were selected. Their detailed description is available in [8]. As the real tasks architectures for sequence alignment, detection of approximate palindromes and tandems were selected.

For all six test sets the numbers of bisection operations (S_{Bis}) and numbers of detected areas ($\#A$) satisfying the set of inequations were measured. The results are summarized in table 1. Numbers of bisection operations performed in case of interconnection of both phases are listed in column S_E. The size of the whole search space D_{Size} can be used for illustration about the number of steps performed by exhaustive computation.

Table 1: Experimental measurement of time and space complexity

Task	D_{Size}	S_{Bis}	$\#A$	S_E
Artif. func. 1	16 384	1 211	141	302
Artif. func. 2	16 384	9 487	960	19
Artif. func. 3	16 384	2 247	279	338
App. palindromes	1 000	17	4	14
App. tandems	1 000	16	4	13
Sequence alignment	16 384	999	157	760

The results in table 1 show: (1) In all examples of artificial and real tasks the proposed method is capable to calculate the design space with lower number of steps then the exhaustive computation. The number of steps is roughly one order lower then the original space. (2) Interconnection of both phases brings significant decrease of number of bisection operations, roughly two orders in comparison to the original space size.

4. Experimental evaluation

For evaluation of the proposed method an example of architecture for sequence alignment (described in more detail in [8]) was selected. This circuit is based on systolic array (SA) of processing elements (PEs), which calculates the whole dynamic programming matrix gradulary. As the real tasks usually produce huge amount of queries more than one SA can be placed on the single chip and operate in parallel fashion. It is supposed that the target platform is in form of acceleration card connected to PCI Express bus. Input strings for individual SAs are transfered from host RAM to the local memories inside the chip using DMA operations. Similarly in the opposite direc-

tion, the calculated scores are aggregated from SAs into the single output stream and transferred back to the host RAM (see figure 2). Therefore additional blocks necessary for DMA operations realization and communication with system bus are included as well.

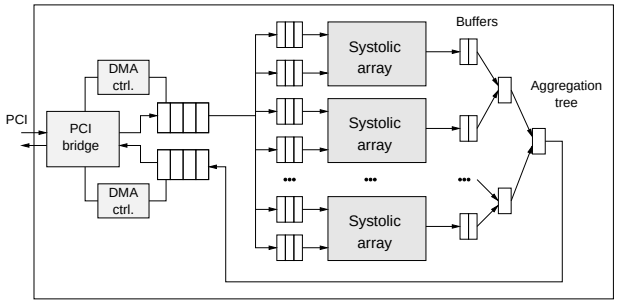

Figure 2: Overall architecture of the system for sequence alignment

A variant of this circuit allowing processing of strings longer than available number of PEs was described in [8]. In this case the computation is performed band by band whereas the intermediate results are stored into the auxiliary FIFO memory. This kind of computations generally allows to build shorter SAs and effectively utilizes the available PEs. On the other side, each array represents a certain overhead and consume additional resources for realization of FIFO memory and other necessary components. Therefore it makes sense to consider this variant of the circuit and search for optimal number of arrays as well as for their length.

4.1 Architecture Model

At the beginning of the model construction, lets define the input task parameters specified by user. To the most important ones belong: ranges of the first input string lengths $L_{1_{min}}$ and $L_{1_{max}}$, ranges of the second input string lengths $L_{2_{min}}$ and $L_{2_{max}}$, number of queries Q and character data width C_{DW} (v bits). Based on these inputs other necessary parameters can be derived (equation 11) such as average lengths of the both input strings $L_{1_{avg}}$ and $L_{2_{avg}}$, data with for storing or distribution of the score value S_{DW} and data width E_{DW} of the item transferred back to the host RAM (it is supposed that it is composed of score value and number of SA which generated its). Target platform parameter are: amount of available resources R_{FPGA}, input B_{Sin} and output B_{Sout} system bandwidth.

$$\begin{array}{lcl} L_{1_{avg}} & = & (L_{1_{min}} + L_{1_{max}})/2 \\ L_{2_{avg}} & = & (L_{2_{min}} + L_{2_{max}})/2 \\ S_{DW} & = & \lceil log_2(max[L_{1_{max}}, L_{2_{max}}]) \rceil \\ E_{DW} & = & \lceil log_2(N_{SA}) \rceil + S_{DW} \end{array} \quad (11)$$

Based on the overall architecture described above, it is possible to build the tree structure of parametrized architecture (see figure 3). The architecture is composed of eleven nodes split into two levels. The function f_R representing the overall amount of resources is expressed as a sum of resources of all elementary components:

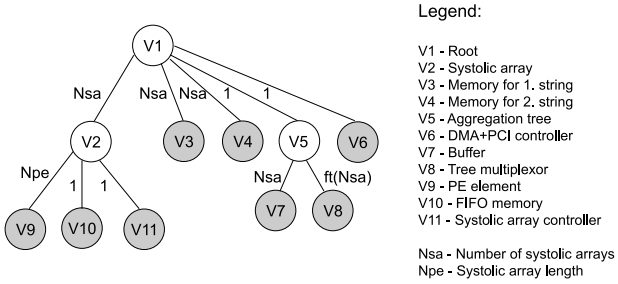

Figure 3: Parametrized architecture

$$\begin{array}{lcl} f_R & = & N_{SA} \cdot [R_{SA} + R_{L_1Mem} + \\ & & R_{L_2Mem}] + R_{Tree} + R_{DMA} \\ R_{L_1Mem} & = & f_{mem}(L_{1_{max}}, C_{DW}) \\ R_{L_2Mem} & = & f_{mem}(L_{2_{max}}, C_{DW}) \\ R_{Tree} & = & N_{SA} \cdot R_{Buf} + f_{tree}(N_{SA}) \cdot R_{Mx} \\ R_{Buf} & = & f_{mem}(Const, E_{DW}) \\ R_{Mx} & = & f(E_{DW}) \\ R_{SA} & = & N_{PE} \cdot R_{PE} + R_{Fifo} + R_{Ctrl} \\ R_{PE} & = & f(S_{DW}, C_{DW}) \\ R_{Fifo} & = & f_{mem}(L_{2_{max}} - N_{PE}, S_{DW} + C_{DW}) \end{array} \quad (12)$$

where R_{L_1Mem} and R_{L_2Mem} represent amount of resources for auxiliary buffers for input strings, R_{Tree} for aggregation tree, R_{Buf} for exported score values buffer, R_{Mx} for aggregation tree multiplexer, R_{SA} for systolic array, R_{PE} for processing element, R_{Fifo} for auxiliary intermediate results memory and R_{Ctrl} for systolic array control logic.

Auxiliary function $f_{mem}(Items, DataWidth)$ expresses an amount of resources for realization of memory blocks composed of given number of items of appropriate data width. The main reason for this function is the fact, that different technologies build memory blocks from different components, f_{mem} function abstracts this dependencies. Auxiliary function $f_{Tree}(n)$ returns a number of internal nodes of the tree, which has n lists, where n is arbitrary number (even out of the power of two).

Average computation time needed for processing of single query T_{avg} is expressed in equation 13. It considers splitting of computation into the bands according to the number of PEs as well as the time for the last band, which can be generally unaligned to array length.

$$T_{avg}(N_{PE}) = \frac{\lceil L_{1_{avg}}/N_{PE} \rceil \cdot L_{2_{avg}} + (L_{1_{avg}} - 1) \bmod N_{PE}}{F} \quad (13)$$

Overall computation time f_T required for the task composed of Q queries is derived in following equation 14. As the lengths of input strings are usually variable, it is not possible to express this time exactly for individual SAs and thus this time represents average value rather.

$$f_T(N_{SA}, N_{PE}) = \frac{Q \cdot T_{avg}(N_{PE})}{N_{SA}} \quad (14)$$

An important parameter of the architecture is a require-

ment for the input system bandwidth B_{In}. This bandwidth has to be sufficient to feed all SAs with the new characters. Relation for B_{In} is derived in following equation 15. It results from bandwidth required by single SA which is multiplied by number of arrays. Then the bandwidth for the single array is calculated as an amount of data required during the query processing.

$$B_{In}(N_{SA}, N_{PE}) = \frac{(L_{1_{avg}} + L_{2_{avg}}) \cdot C_{DW}}{T_{avg}(N_{PE})} \cdot N_{SA} \quad (15)$$

Similarly, the required output bandwidth B_{Out} has to be sufficient to transfer all calculated scores back to host RAM. Equation for B_{Out} is as follows:

$$B_{Out}(N_{SA}, N_{PE}) = \frac{E_{DW}}{T_{avg}(N_{PE})} \cdot N_{SA} \quad (16)$$

A set of constrained conditions O resulting from ranges of loops and properties of the target platform is defined using inequations 17. The first condition represents the fact that it does not make sense to build SAs longer than the length of the input sequences. Similarly, the second condition expresses the fact that it does not make sense to connect more SAs than the number of queries and thus the available resources can be used for example to make the arrays longer. The last two conditions limits the number of SAs and their lengths with respect to available input B_{Sin} and output B_{Sout} system bandwidth.

$$\begin{array}{lcl} L_{1_{max}} - N_{PE} & \geq & 0 \\ Q - N_{SA} & \geq & 0 \\ B_{Sout} - B_{Out}(N_{SA}, N_{PE}) & \geq & 0 \\ B_{Sin} - B_{In}(N_{SA}, N_{PE}) & \geq & 0 \end{array} \quad (17)$$

Thus we have all necessarily parts of parametrized architecture $A = (f_R, f_T, O)$.

4.2 Evaluation and Results

An objective of this subsection is to evaluate the proposed model on a certain task and show how the sizes of resulting circuit will differ on selected chips with Virtex5 LXT technology. The input task is defined as follows: the lengths of both input strings are in range 80-120 characters ($L_{1_{avg}} = L_{2_{avg}} = 100$), DNA sequences will be analyzed ($C_{DW} = 2$) and number of the task queries Q is 100 and more. Used FPGA chips contain huge amount of computation resources in range 7200 slices (xc5lx50t) up to 51840 slices (xc5lx330t). Moreover, all chips contain an embedded IP core for connection to PCI Express x8 bus, where the maximal input and output bandwidth achieve 16Gbps.

At first, individual parts of architecture were implemented. For description of the circuit the VHDL language was used and the synthesis was performed using Xilinx ISE tools. The blocks for controlling of DMA operations and for access to PCI Express bus were adopted from NetCOPE platform [9]. All values representing amount of resources for elementary components were supplemented into the model in form of constants (into the f_R function). An overall amounts of resources were calculate by the model (function f_R) for different values of input task parameters $L_{1_{avg}}$, $L_{2_{avg}}$, C_{DW} and Q, and these amounts were compared with the values obtained from the real implementations. The difference about 5% was measured between the model and real implementations, which can be caused for example by optimization techniques used by tools for synthesis, placing and routing.

Using the model of parametrized architecture and the method described in section 3, an optimal size of circuit, i.e. number of SAs (N_{SA}) and its length (N_{PE}), were found for selected chips with Virtex5 technology. The results listed in table 2 show that the method prefers the array lengths, which divide the $L_{1_{avg}} = 100$ without reminder. It is a correct behavior because the PEs are used the most effective. Only the first row represents an exception, where the method selected a different ratio between number of arrays and their lengths which was caused by the tiny amount of resources. The lack of input/output bandwidth was not observed even in case of the biggest chip with the most number of arrays. Therefore the available bandwidth 16Gbps is sufficient for this type of task.

Table 2: Architecture sizes for selected chips with Virtex5 technology

FPGA	Slices	N_{PE}	N_{SA}	p_{HW}	SUp
xc5vlx50t	7 200	15	6	21,16	423
xc5vlx110t	17 260	20	13	62,62	1 252
xc5vlx220t	34 560	25	23	135,61	2 712
xc5vlx330t	51 840	20	43	207,13	4 142

Additionally, the circuit performance and its speed-up in comparison to the same algorithm implemented in the software was evaluated for individual chips. As the software algorithm calculates DP matrix sequentially, its performance can be measured in number of DP items processed per unit time independently of input strings lengths. At conventional processors it is possible to achieve the performance approximately 0.05 billions of items per second [2] (measured at processor Xeon 3GHz). On the other side, the hardware circuits calculates DP matrix in parallel and thus for the performance evaluation it is necessary to take into account an overhead caused during the initialization and finishing phase. The hardware performance expressed in number of items processed per unit time is derived in equation 18. The resulting values of performance as well as the speed-up are shown in the last two columns of the table 2. As we can see, the circuits are able to achieve the speed-up in orders of hundreds or thousands in comparison to the same algorithm implemented in the software and tested on one of the most powerful conventional processor.

$$P_{HW}(N_{SA}, N_{PE}) = \frac{L_{1_{avg}} \cdot L_{2_{avg}}}{T_{avg}(N_{PE})} \cdot N_{SA} \quad (18)$$

5. Conclusions

An acceleration of algorithms for DNA sequence analysis using dedicated circuits brings a certain expectation how to face up the increasing amount of biological data and requirements for their elaborate processing. Such circuits are capable to achieve the speed-up in orders of hundreds or thousands in comparison to conventional processors.

Despite of this huge speed-up these accelerators are not widely used in real world applications. One of the main reasons relies in the often variability of tasks, which affects the sizes of resulting computation arrays. Technology of programmable chips (FPGA) is able to solve this variability effectively, because the input task parameters can be specified at synthesis time and thus the circuit optimal for certain requirements of target application can be created. However, in case that the user has a certain FPGA chip containing limited number of configurable gates then the question is how to select the circuit sizes to consume the computation resources effectively and achieve the maximal speedup for concrete target application.

Therefore this thesis was focused on designing a new technique for automated mapping of existing architectures to the chips with FPGA technology. The study of state-of-the-art methods shown that existing techniques are not able to find the optimal architecture sizes. Moreover, they require the linearity of all used functions, which is not possible to satisfy for the real circuits. The problem of the optimal sizes searching was investigated in more detailed and formally defined using the parametrized architecture model. Using this formal description a novel method for searching of optimal circuit sizes was designed.

Finally, the proposed method was evaluated on an example of circuit for sequence alignment. The parametrized model of this architecture was built with respect to parameters of the real target platform. An application of the proposed method shown how the sizes of the circuit can change on different chips with Virex5 technology.

Acknowledgements. This research has been partially supported by the Grant Agency of the Czech Republic Research Grant No. 204081560 – In vitro and in silico identification of non-canonical DNA structures in genomic sequences and MSMT Research Grant No.0021630528 – Security-Oriented Research in Information Technology.

References

[1] T. V. Court and M. C. Herbordt. Lamp: A tool suite for families of fpga-based application accelerators. In *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications (FPL), Tampere, Finland, August 24-26, 2005*, pages 612–617. IEEE, 2005.

[2] T. V. Court and M. C. Herbordt. Families of fpga-based accelerators for approximate string matching. *Microprocessors and Microsystems*, 31(2):135 – 145, 2007. Special Issue on FPGA-based Reconfigurable Computing (2).

[3] F. Hannig and J. Teich. Design space exploration for massively parallel processor arrays. In *PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*, pages 51–65, London, UK, 2001. Springer-Verlag.

[4] D. T. Hoang. Searching genetic databases on splash 2. In D. A. Buell and K. L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[5] R. B. Kearfott. Abstract generalized bisection and a cost bound. *Mathematics of Computation*, 49(179):187–202, 1987.

[6] C. T. Kelley. *Solving nonlinear equations with Newton's method*. Fundamentals of algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[7] M. Petr. A bisection method to find all solutions of system of nonlinear equations. In *In Domain Decomposition Methods in Scientific and Engineering Computing*. Providence : AMS, 1994.

[8] T. s Martínek. *Hodnocení podobnosti biologických sekvencí s využitím technologie FPGA*. PhD thesis, Ústav počítačových systému FIT VUT v Brně, 2010.

[9] T. s Martínek and M. K. sek. Netcope: Platform for rapid development of network applications. In *Proc. of 2008 IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, pages 219–224. IEEE Computer Society, 2008.

[10] T. VanCourt and M. Herbordt. Sizing of processing arrays for fpga-based computation. In *Field Programmable Logic and Application (FPL 2006)*, pages 755–760, Madrid, Spain, September 2006.

[11] C. W. Yu, K. H. Kwong, K.-H. Lee, and P. H. W. Leong. A smith-waterman systolic cell. In *Field Programmable Logic and Application (FPL 2003)*, pages 375–384, Lisbon, Portugal, September 2003.

Selected Papers by the Author

T. Martínek and M. Lexa. Hardware acceleration of approximate tandem repeat detection. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 79–86. IEEE Computer Society, 2010.

T. Martínek, M. Lexa, and J. Voženílek. Architecture model for approximate palindrome detection. In *2009 IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 90–95. IEEE Computer Society, 2009.

T. Martínek and M. Lexa. Hardware acceleration of approximate palindromes searching. In *The International Conference on Field-Programmable Technology*, pages 65–72. IEEE Computer Society, 2008.

T. Martínek, M. Lexa, P. Beck, and O. Fučík. Automatic generation of circuits for approximate string matching. In *2007 IEEE Design and Diagnostics of Electronic Circuits and Systems*, pages 203–208. IEEE Computer Society, 2007.

T. Málek, T. Martínek and J. Kořenek. Gics: Generic interconnection system. In *2008 International Conference on Field Programmable Logic and Applications*, pages 263–268. IEEE Computer Society, 2008.