

Published in IET Computers & Digital Techniques
 Received on 5th May 2008
 Revised on 2nd October 2008
 doi: 10.1049/iet-cdt.2008.0042



Case studies in determining the optimal field programmable gate array design for computing highly parallelisable problems

J.E. Rice¹ K.B. Kent²

¹Department of Math and Computer Science, University of Lethbridge, 4401 University Drive, Lethbridge, AB, T1K 3M4, Canada

²Faculty of Computer Science, University of New Brunswick, PO Box 4400, Fredericton, NB, E3B 5A3, Canada

E-mail: j.rice@uleth.ca

Abstract: Reconfigurable hardware has recently shown itself to be an appropriate solution to speeding up problems that are highly dependent on a particular complex or repetitive sub-algorithm. In most cases, these types of solutions lend themselves well to parallel solutions. The optimal design on field programmable gate arrays (FPGAs) for problems with algorithms or sub-algorithms that can be highly parallelised is investigated. In addition, a classification system is introduced, which categorises FPGA-based solutions into 'instance-specific' and 'parameter-specific'.

1 Introduction

When comparing dedicated hardware and software solutions, it is usually said that hardware is fast and software is slow, although software is more flexible than hardware. Most problems are generally solved in software, since the overhead of writing a program to solve the problem is usually low in comparison to the effort involved in designing a dedicated hardware solution. However, if the resulting solution is too slow, consumes too much power or is physically too large, a move to dedicated hardware might be made. In the past, this would have required fabricating a special-purpose circuit designed specifically to solve the problem at hand. The introduction of reconfigurable hardware now allows us to merge the advantages of both hardware and software. The advent of field programmable gate arrays (FPGAs) has provided researchers and industry alike with a hardware solution, leveraging much of the speed advantages of fabricated circuits that can be reprogrammed in many cases virtually limitlessly. The effort involved in designing hardware solutions is now much more feasible, even if only one instance of the chip will be required. Moreover, it is possible to design a hardware solution specific to only one instance of the problem input data, since the FPGA can then be reprogrammed to deal with subsequent instances of the data [1].

This has led to many interesting changes in the way that problems are solved in hardware. Co-processors that can change their main focus as needed by the user are becoming available, and people researching problems such as DNA matching [2, 3], image compression [4, 5], graph applications [6] and various applications in security [7, 8] are now using FPGAs as their development platform.

Many hardware solutions leverage the ability of FPGAs to provide parallel processing. However, this requires that the problem be subdivided in order to allow the hardware components to simultaneously process the separate parts of the problem. There is overhead in how to subdivide the problem, and in how to merge the results into a final solution. Communication between the separate parts may be needed. And so the question becomes at what point is there too much subdivision; that is, when does the overhead required outweigh the advantages of the parallel processing speed-up? We investigate this problem through the study of FPGA implementations for two problems each well-suited to FPGA implementation.

2 Comparisons to other work

Previous work related to this research includes [2, 9–12]. In each of these works only one particular type of problem was

investigated; this paper extends the work to consider multiple problems that are approached in a variety of ways.

In general, the goal of this work is to determine, for a particular problem, how best to utilise the resources available on the FPGA. The most relevant example in software technology would be the analysis performed within a compilation context to achieve automatic program parallelisation [13]. In order to be effective, parallel computing must efficiently utilise the resources that are made available, and must consider both data dependencies [14, 15] and shared resources [16–18]. Software-based research in this area has filtered into the embedded systems domain with the introduction of novel platforms and paradigms; such work has been carried out on entire systems consisting of both custom hardware and a software processor (System-on-Chip) [19] as well as more complex systems consisting of a network of computing resources (Network-on-Chip) [20]. Several Network-on-Chip architectures have also been researched for communication scalability in order to determine the threshold at which each infrastructure becomes a serious limiting factor [21, 22]; however, this work focuses on topologies, and not on particular problems and their solutions as introduced in our work.

Other related work for FPGA-type devices includes the work by Milder on automatic generation of hardware implementations of the discrete Fourier transform (DFT) [23]. This work allows the user to specify parameters such as size, throughput and latency of the data to be processed, assuming it is known, and the system will choose the best implementation based on this information. Our work did not find that a particular implementation performed better or worse for given datasets, although we are continuing investigations in this area. Milder *et al.* also published earlier work in resource estimation for DFT IP cores [24]. This work estimates the numbers of slices required for particular DFT implementations. Although an application of their work involving determining the fastest DFT is briefly discussed, the authors are mainly considering slice usage estimations, without the added discussion of how maximising slice usage can affect the speed of the implementation.

Cong *et al.* study the optimality of the logic synthesis step in [25, 26], which, although related to this work, is not our primary interest. Our work is mainly focused on the links between additional parallelism to increase computation speed and the resulting overhead in communication resources.

Finally, work in resource utilisation in prime number validation has been published in [27], which addresses similar architectural issues as does our work. Their work also takes note of the trade-off in increasing complexity with additional parallelism, resulting in lowered clock cycles. However, our work continues this investigation with

additional problems, and compares the results from this phenomenon across the problems under investigation.

3 Background

We first introduce a number of concepts so that future discussions may be clearly understood.

3.1 Configurable hardware

The primary focus of this work is ‘configurable hardware’. By this we refer to hardware devices such as FPGAs that may be programmed multiple times to solve different problems. FPGAs are basically what the name describes: hardware consisting of arrays of logic elements that are programmable in the field (i.e. not in a large manufacturing plant, as is usually necessary). The most common type of programmable logic element in FPGAs is called a K-LUT, which is a K-input one-output lookup table (LUT). With such an element, any K-input single-output Boolean function can be implemented. Other resources on a FPGA chip include I/O elements, which are mainly situated around the edges of the chip, and routing resources allowing the LUTs to be connected as needed to implement the desired functionality.

3.2 Problems investigated

Two types of problems were investigated during the course of our studies. These are described in the following subsections.

3.2.1 Computing the autocorrelation coefficients:

The autocorrelation coefficients of a Boolean function F are the result of applying the autocorrelation transform:

$$B(u) = \sum_{v=0}^{2^n-1} F(v) \cdot F(v \oplus u) \quad (1)$$

where n is the number of inputs and $v = \sum_{i=1}^n v_i 2^{i-1}$ [28]. Table 1 shows the autocorrelation coefficients for the majority function, $F(X) = x_1 x_3 + x_1 x_2 + x_2 x_3$. For

Table 1 Autocorrelation coefficients for the majority function, $F(X) = x_1 x_3 + x_1 x_2 + x_2 x_3$

u	$B(u)$
000	4
001	2
010	2
011	2
100	2
101	2
110	2
111	0

example, the value for $B(001)$ is computed by expanding Equation (1) as follows:

$$\begin{aligned}
 B(001) &= F(000) \cdot F(000 \oplus 001) + F(001) \cdot F(001 \oplus 001) \\
 &\quad + \dots + F(111) \cdot F(111 \oplus 001) \\
 &= F(000) \cdot F(001) + F(001) \cdot F(000) \\
 &\quad + \dots + F(111) \cdot F(110) \\
 &= 0 \cdot 0 + 0 \cdot 0 + \dots + 1 \cdot 1 \\
 &= 2
 \end{aligned}$$

Note that the value of u used to compute each coefficient is generally given in its binary format.

The autocorrelation coefficients have been used in applications such as variable ordering for ROBDDs [29], classification and detecting special properties of functions such as symmetries [30]. The application of configurable hardware to this problem was first introduced in [31].

3.2.2 Edit-distance calculation: The determination of the similarity between two DNA or protein sequences is a common problem in bioinformatics [32]. The problem lies in how to compute a 'score' that identifies the number of mutations necessary to change one of the sequences (the query sequence) into the other (the target sequence). One approach to this problem is to compute all pairwise comparisons between the two strings; however, this is quite computationally expensive. There are three actions that can be taken as the comparisons progress [33]:

- mutation, or changing one symbol to another,
- insertion, or inserting an additional symbol, and
- deletion of a symbol.

Each operator is assigned a particular weight, and the total of the required operations is the score. The implementation investigated in this work uses a fixed-weight version of the edit-distance algorithm in [32] with modifications from [34]. The primary concept is that of the processing element, which is defined as follows:

$$d(i, j) = \begin{cases} i & \text{when } j = 0 \\ j & \text{when } i = 0 \\ \min \begin{cases} d(i-1, j-1) + C(i, j) \\ d(i-1, j) + w_i \\ d(i, j-1) + w_d \end{cases} & \text{otherwise} \end{cases} \quad (2)$$

This assumes a two-dimensional array of cells where i and j are the row and column values, respectively, and the edit-distance solution consists of having each of these cells use the formula to compute its value. The other variables used in this equation are

- w_d , which is the cost of deleting a single element from the string,
- w_i , which is the cost of inserting a single element into the string, and
- $C(i, j)$, which is zero if the i th symbol of string S is the same as the j th symbol of string T , otherwise it is w_m , or the cost of mutating an element from one string into an element from the other.

This work imposes the restrictions that $w_m = 2$ and $w_i = w_d = 1$. In the two-dimensional array storing, each value of d , the edit-distance between two strings S and T is found in $d(|S|, |T|)$.

4 Approaches

There are a variety of ways to approach the problems addressed in this research. We have classified these into two groups: instance-specific and parameter-specific approaches.

An instance-specific approach requires that the design incorporate the data for a particular instance of the problem. Computing the solution for another instance of the problem may require redesign and reprogramming of the hardware, but the specificity of the solution may result in quite significant speed advantages. In contrast to this, a parameter-specific approach results in a solution that is suitable for solving a variety of instances of the given problem, provided that those instances fit within given parameters. Parameters are usually variables such as limits on the input length of the data.

Our solution to the edit-distance problem was best suited to a parameter-specific approach, as is detailed in Section 4.3. The computation of the autocorrelation transform was implemented twice; once as a parameter-specific solution and once as an instance-specific solution. These are detailed in Sections 4.1 and 4.2. Section 5 provides some experimental results, and Section 6 focuses on how each of these approaches can be examined in terms of the trade-off between added parallelism, with the accompanying increased speed, and the overhead of the required increased complexity that must support the additional parallel processing capabilities.

In general, the design-flow used in this work consisted of creating a VHDL or other hardware-language description of the desired circuit, which was then processed by the FPGA-design software to generate a configuration file. Experiments on each problem were carried out to determine the optimal CLB usage (maximise parallelism and still place and route in the target device). Particular differences in the design flow for each problem are detailed in the subsequent sections.

4.1 Computation of the autocorrelation coefficients

We first introduce a parameter-specific approach to the computation of the autocorrelation coefficients. The algorithm implemented here is based on the work discussed in [35]. The goal in this implementation was to design hardware that allowed the computation to be carried out for any functions, as long as the functions fit within certain parameters. In the case of the autocorrelation function, the only parameter that limits the circuit is the number of inputs to the function F .

Briefly, the process for computing $B(u)$ is as follows: for each cube in the disjoint cube list

- compute cube $\oplus u$
- search for the new cube or one containing it in the cube list
- if found add two to the sum register as the contribution to the coefficient

Each function is previously expressed as a list of disjoint cubes [35].

As indicated in the algorithm, this solution requires that the function is described as a disjoint cube list [36]. Complete details for the algorithm are given in [35]. The architecture for this solution is shown in Fig. 1.

Because this solution is based entirely on the algorithm above, it could be described in entirety in the hardware-design language. A preprocessing step of converting the benchmark file to a disjoint cube list is, however, required for this approach. This preprocessing was carried out in software (the *espresso* software [37] with option *Dsjoint* was used).

4.2 Another approach to computing the autocorrelation coefficients

The second solution to the computation of the autocorrelation function was to use an instance-specific approach. The

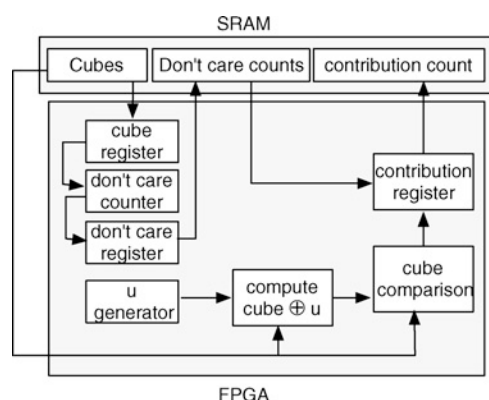


Figure 1 Architecture of the parameter-specific solution for computing the autocorrelation coefficients

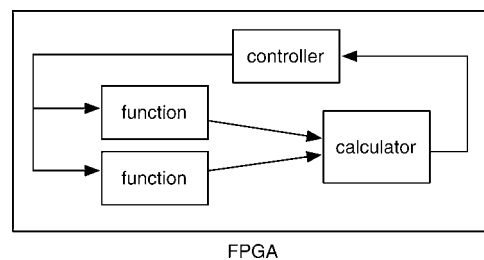


Figure 2 Architecture for instance-specific FPGA solution for computing the autocorrelation coefficients

architecture shown in Fig. 2 was used in this solution. As shown in Fig. 2, there are three main components: the function component, calculator component and the controller. The function components contain the function for which the coefficients are to be computed. Two function components are required, one for each function being compared in the equation. The computation is carried out by performing a comparison between the two functions stored in the function components, thus a large portion of the circuit is dedicated to representing these functions, and so the representation choice is very important.

Binary decision diagrams (BDDs) represent a function's truth table by having a leaf node for every possible combination of values that the input variables can hold. Reduced-ordered binary decision diagrams, or ROBDDs [38], are a canonical form of BDDs that attempt to reduce the exponential size of BDDs by sharing similar parts of the tree. In general, the term BDD is used to refer to a ROBDD.

In this work, a BDD representation provides a fast and compact implementation, since the BDD of the function can be directly translated into a finite-state machine. It is this portion of the design that makes this an instance-specific solution, since the two function components are designed specifically for the given function F and its counterpart $F(v \oplus u)$. The benchmark functions used for our experiments are generally provided in a programmable logic array (PLA) format. This format is similar to a sum-of-products, and is supported by *espresso* [39]. This is used to build a BDD that is then translated to VHDL.

The controller and calculator components do not need to substantially change from instance to instance. The controller component remains the same and simply has its output directed to another set of function components. The calculator component changes slightly since it must accommodate for an additional term in the overall addition step. Addition of the terms is performed in a pair-wise fashion to diminish the penalty for highly parallel architectures (i.e. eight terms requires seven additions performed in three stages). As the output of the functions is a single bit, the addition operation is rather insignificant to the

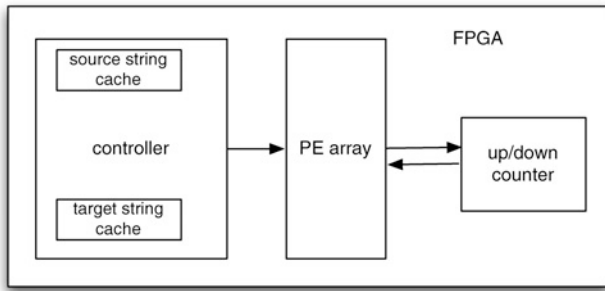


Figure 3 Architecture used for computing the edit-distance of two strings

overall performance. This does, however, result in a small latency increase in the calculator as parallelism is increased.

The underlying technique for this approach was also introduced in [35].

4.3 Computation of the edit-distance score

Computation of the edit-distance between two DNA sequences was the second parameter-specific solution investigated in this work. The algorithm used for this problem is described in Section 2.

A naïve approach would be simply to construct the entire two-dimensional matrix, based on this description. However, if string S is length m and string T is length n , this would require $(m + 1) \times (n + 1)$ processing elements, and is impractical for most instances of the problem. Optimisations suggested by Lipton and Lopresti [34] make a two-channel, two-way systolic array, the most practical solution, and it is such an architecture that is used in this work. The processing element can now be reduced to

$$d(i, j) = \begin{cases} a & \text{if } b = a - 1 \text{ or } c = a - 1 \text{ or } S_i = T_j \\ a_2 & \text{if } b = c = a + 1 \text{ and } S_i \neq T_j \end{cases} \quad (3)$$

In this equation, we have

- $a = d(i - 1, j - 1)$,
- $b = d(i - 1, j)$, and

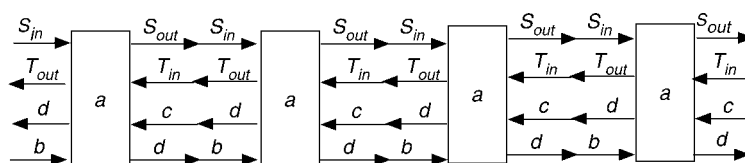


Figure 4 Systolic array of processing elements used to compute the edit-distance of two strings S and T

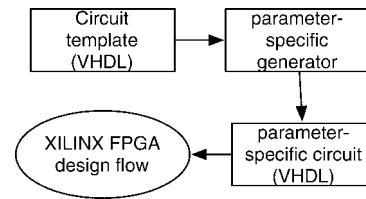


Figure 5 Design-flow for generating FPGA configurations for the edit-distance problem

- $c = d(i, j - 1)$.

Details of the reduction to this simplified version are given in [9].

The implementation consists of an array of processing elements, an up/down counter and a controller, as shown in Fig. 3. Each processing element computes the formula given in Equation (3) and passes the result to its left and right neighbours. Figure 4 illustrates how the elements of the array communicate.

The design-flow used for generating FPGA configurations for this solution is shown in Fig. 5. There are a number of parameters that can be varied when applying this solution to problem instances. In particular, it is common to have strings of varying lengths, and also to require more than two comparison strings. Thus, the design-flow incorporates a template for a general solution to the problem. When the parameters are known then a solution specific to those inputs, but still applicable to general instances of data within those parameters, can be generated and an FPGA configuration file created.

5 Experimental results

Most of the tests reported here were carried out on a Pentium 4, 2.8 GHz, running Windows XP. The target device for the edit-distance problem and the instance-specific autocorrelation computation was the Xilinx Virtex-E 802e, and the Xilinx Integrated Software Environment (ISE) 6.3i tool suite was used to generate designs and simulate the solutions. However, for the parameter-specific autocorrelation computation, a Xilinx Virtex 812E device was targeted. The reason for this is that this chip was packaged on a daughter-board with on-board RAM that was utilised for extra storage in this solution. In this case, the Xilinx ISE 5.2i tool-suite was used to generate the designs, and the results for this solution were obtained by execution on the targeted device.

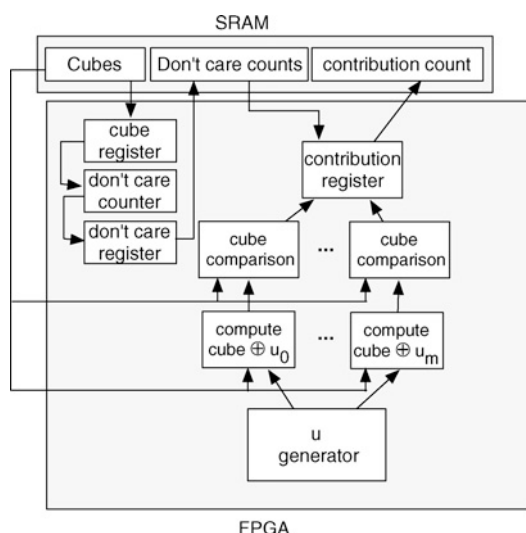


Figure 6 Modified architecture of the parameter-specific solution for computing the autocorrelation coefficients

We should note that some results are reported in terms of ‘slices’. The Xilinx XCV812E device contains 9408 slices, each of which is approximately one-half of a CLB.

The benchmarks used for computation of the autocorrelation function are taken from the MCNC 91 benchmark set [40]; for this work, only single-output functions are considered.

5.1 Parameter-specific autocorrelation computation

The parameter-specific solution to this problem allowed two variations. One was in the number of bits for storage of each cube in the listed used for comparisons. A smaller number of bits limits the solution to work on functions with fewer input variables, but a larger number of bits requires additional computation space on the FPGA. The second variation was in the number of coefficients to be computed in parallel. As shown in Equation (1) the value u specifies

which particular coefficient is desired. This value can range from 0 to $2^n - 1$. This solution assumes that all of these 2^n coefficients are required, and so attempts to compute multiple coefficients in parallel. The modified architecture is shown in Fig. 6. Table 2 contains information regarding the resource usage for various configurations, and Table 3 shows the timing results from these tests. The clock speed for all tests was 26 MHz. It should be noted that in Table 3 the computations were repeated, as necessary, until all 2^n coefficients were computed, and the time for computing all 2^n coefficients is reported.

5.2 Instance-specific autocorrelation computation

In each clock cycle, the hardware architecture used for this solution (shown previously in Fig. 2) can compute only one term of the 2^n required for each coefficient. Replication of the function components, as shown in Fig. 7, allows for multiple terms to be computed in parallel. In general, the experiments on this solution consisted of increasing the number of function components by two until the design could no longer fit into the FPGA. Table 4 gives results for the optimal circuit for each of the benchmark test cases. In every case, the fastest computation time was attained at the highest level of parallelism that the Xilinx tools could successfully place and route, even though this resulted in a clock frequency of as low as 2 MHz. In Table 4, the first three columns give the name of the benchmark, the number of nodes that its BDD representation required and the number of inputs. The optimal level of parallelism achievable is linked to this. The column titled ‘max parallel’ refers to the maximum number of function components that the tools could successfully design for, and the final two columns provide statistics on the number of slices (of a total of 9408 available) required.

There is one extra factor that must be considered with an instance-specific solution, and that is the length of time required to generate the bitstream for configuration of the

Table 2 Space usage of the Xilinx Virtex 812E chip for various scenarios of the parameter-specific approach to the autocorrelation computation problem

Cube bits	Parallel coeffs	LUTs for logic	LUTs for routing	LUT usage (%)	Slice usage (%)
32	64	13 933	891	78	95
26	64	12 696	696	71	84
21	64	11 722	502	62	75
15	64	10 554	307	57	65
10	64	9588	176	51	56
32	32	7412	477	41	51
10	32	5119	114	27	30
32	1	956	81	5	8

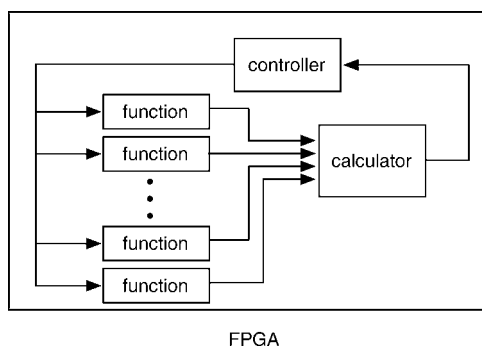
Table 3 Times (in s) to compute all 2^n coefficients as required by the parameter-specific autocorrelation solution

Test case	64 parallel	No parallel
9symml	0.2940	1.0069
Cm152a	0.2680	0.3181
co14	0.2490	0.4480
ex10	0.2650	0.3024
ex20	0.2680	0.2934
ex30	0.3010	0.2968
Life	0.2760	0.6843
majority	0.2680	0.3102
max46	0.2750	0.3349
mux01	24.5830	309.2230
Ryy6	2.1330	32.8776
Sym10	1.1740	27.9384
xor5	0.2700	0.3058

FPGA. In our experiments, this could take anywhere from 10 min to 24 h. This is clearly of a magnitude far larger than we are considering for the run-time comparisons. This certainly must be taken into consideration for solution implementations that are only required to be calculated once. For other problems, however, it is likely that a solution may be used multiple times, thus leveraging the high overhead of creating the instance-specific implementation. It is worth noting that as techniques in incremental synthesis are developed, the time required to re-synthesise the circuit will certainly decrease [41].

5.3 Comparison to software

It is interesting to compare these results to those achieved in software implementations. Table 5 gives the best times as

**Figure 7** Enhanced architecture for the instance-specific computation of the autocorrelation function**Table 4** Test results showing the slice usage for the optimal circuit for each benchmark, as generated by the instance-specific autocorrelation solution

Test case	No. of BDD nodes	No. of inputs	Max. parallel	No. of slices	Slice usage (%)
ex10	6	5	252	9350	99.38
xor5	6	5	252	9350	99.38
majority	8	5	252	9372	99.62
ex30	10	5	252	9341	99.29
ex20	11	5	250	9391	99.82
cm152a	16	11	226	9385	99.76
ryy6	21	16	214	9308	98.94
9symml	25	9	212	8806	93.60
co14	27	14	214	9234	98.15
sym10	31	10	192	8600	91.41
mux01	33	21	190	9333	99.20
max46	75	9	158	8920	94.81

reported in [35] for software computation techniques, and compares them to the best times as resulting from our two hardware implementations.

Table 5 A comparison of the best computation times from [35] to the FPGA solutions reported on in this work

Test case	Software	Param. specific	Instance specific
9symml	39.82	0.2940	0.577
cm152a	18.62	0.2680	9.128
co14	117.0	0.2490	0.59322
ex10	0.64	0.2650	0.00298
ex20	0.64	0.2680	0.00298
ex30	0.64	0.2968	0.00299
life	11.38	0.2760	0.576
majority	0.64	0.2680	0.00298
max46	5.689	0.2750	0.597
mux01	> 200 000	24.58	9941
ryy6	409.6	2.133	9.507
sym10	30.72	1.174	2.340
xor5	0.64	0.27	0.00298

All timings are reported in seconds for computation of 2^n coefficients

Table 6 Test results from the optimal circuit for each case of parallelism for the edit-distance problem

Parallelism	Length	No. of slices	Slice usage (%)	Max. freq. (MHz)	Time (ms)
1	830	9378	99.68	79.789	41.584
2	420	9126	97.00	85.448	9.819
4	220	8993	95.59	80.593	2.723
6	140	8417	89.47	80.919	1.149
8	110	8812	93.66	80.593	0.679
10	90	9120	96.94	81.083	0.441
12	70	8456	89.88	82.590	0.281
13	60	8444	89.75	81.981	0.207
14	50	8005	85.09	85.266	0.145
18	50	9001	95.67	82.488	0.133
20	40	8884	94.43	97.305	0.0812
22	30	7541	80.16	102.722	0.0522
24	30	8204	87.20	101.937	0.0482

5.4 Edit-distance

To generate the results for the edit-distance solution we assumed that there were 1000 target (T) strings to be compared. The source and all target strings are of equal length. The results shown in Table 6 were obtained from both simulation and performance on the FPGA device. The 'Parallelism' column refers to the number of systolic arrays that are implemented, while the 'length' column refers to the number of processing elements in each systolic array. The values presented represent the largest combination of parallelism and length that can 'fit' within the target FPGA.

This computation in software, a C implementation executing on a Pentium 2.8 GHz processor, requires 24 759 μ s – significantly slower than the hardware solutions presented. Faster hardware solutions exist, such as Hookiegene [42], where further improvements are made to the array of processing elements. These continuous improvements focus on the individual processing elements to obtain better performance.

6 Discussion

The problem in general is how to fit more parallel computing elements for each problem into a limited amount of resources while also taking into account the added complexity and overhead of dealing with these additional processors, which will require additional routing and more complex controller components.

The structure employed in solving the edit-distance problem lent itself to a very nice scalable architecture, since communication resources were only required between adjacent processing elements. This is clearly shown in Fig. 4. To increase parallelism for this problem, it was necessary to duplicate the entire array of processing elements. Since these arrays tended to be fairly large, a maximum number of arrays tended to be reached quite quickly, and so the controller was required to control at most 16 parallel components.

In contrast, for the instance-specific autocorrelation computation, each and every processing element is required to communicate with the controller. Thus, as we replicate processing elements, the communications requirements increase far faster.

An interesting problem to examine is that of test case ryy6 for the instance-specific solution to the autocorrelation computation. Table 7 gives some of the results from this test case. As expected, the addition of parallel components increased congestion in the routing and complexity in the controller, leading to a decreased clock frequency. In fact, the clock speed decreases very quickly as we begin increasing parallelism, while at the bottom of the table the clock rate decreases much more slowly. With the increase in parallelism, there is an increase in logic to process the results from the parallel components. The increase in logic for the calculator is insignificant. With the design described in Section 2, each increase in parallelism results in one additional adder. Fig. 8 shows how these rates compare.

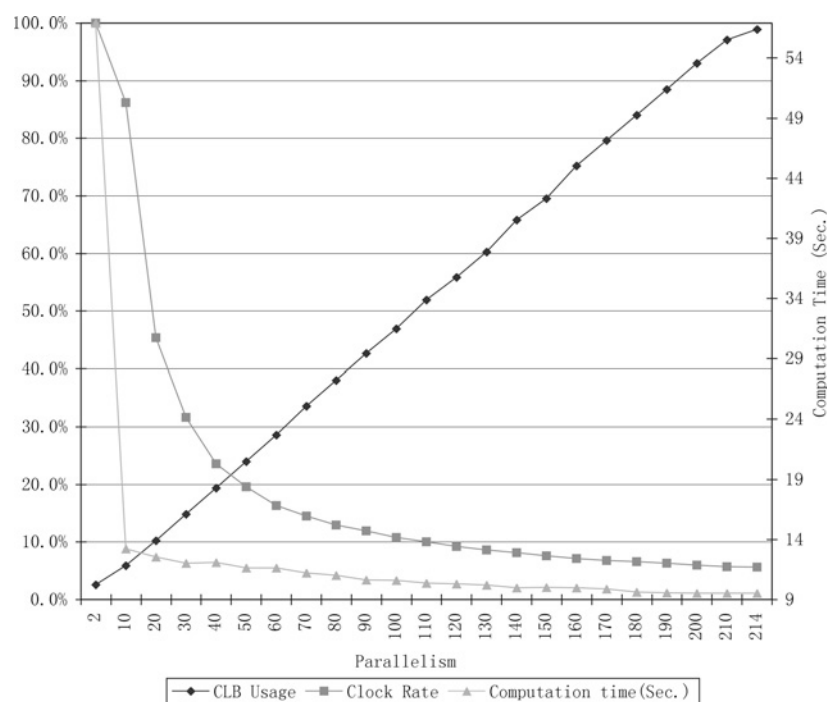
Table 7 A selection of the CLB usage and timing results for benchmark ryy6

Parallelism	CLB usage (%)	Max. freq. (MHz)	Time (s)
2	2.56	75.483	56.90
30	14.84	23.856	12.00
60	28.55	12.317	11.62
90	42.66	8.973	10.64
120	55.84	6.954	10.31
150	69.56	5.728	10.00
180	84.03	4.957	9.63
200	93.05	4.512	9.52
214	98.94	4.222	9.51

The same trend can be seen in the edit-distance circuit, (Fig. 9). Despite this decrease in clock speed as the number of parallel components were increased, we see in the timing column of Table 7 that the increased parallel computation power more than offsets the decrease in the clock speed, and we achieve the best performance with the largest amount of parallel function components that can be fit onto the device. This result held consistently across all the approaches examined in this work.

Parallelism in the parameter-specific autocorrelation computation was achieved by replicating the components that dealt with computing the exclusive or of a particular cube with the current value for u , and then searching for the result in the cube list. One of the variations tested in this work was that of limiting the number of bits used to store each cube. As indicated earlier, limiting this value would reduce the size (in terms of number of input variables) of the functions that could be worked with, but gave interesting results, as shown in Table 8. Additional code optimisations after these values were obtained were able to improve the implementation to the final version in which 32 bits were used to store the cubes, 64 coefficients were computed in parallel and the clock could be run at 26 MHz. The choice of computing 64 coefficients in parallel was made primarily due to being able to maximise the physical resources of the chip most effectively at this size, as is illustrated in Table 2. As shown in Fig. 6, the main overhead in adding parallel components lies in the need to route the cube information to each computational and comparator component. This is limited by our choice of 64 replications of these components.

One question to be asked is why the decrease in cube size allows for a faster clock speed? We suspect that simply freeing up the resources on the FPGA allowed for more or less optimal routing strategies by the CAD tools, thus affecting the clock speed as shown in Table 8.

**Figure 8** Percentages of CLB usage, performance and clock speed in relation to the number of parallel function components used for computing the ryy6 benchmark solution

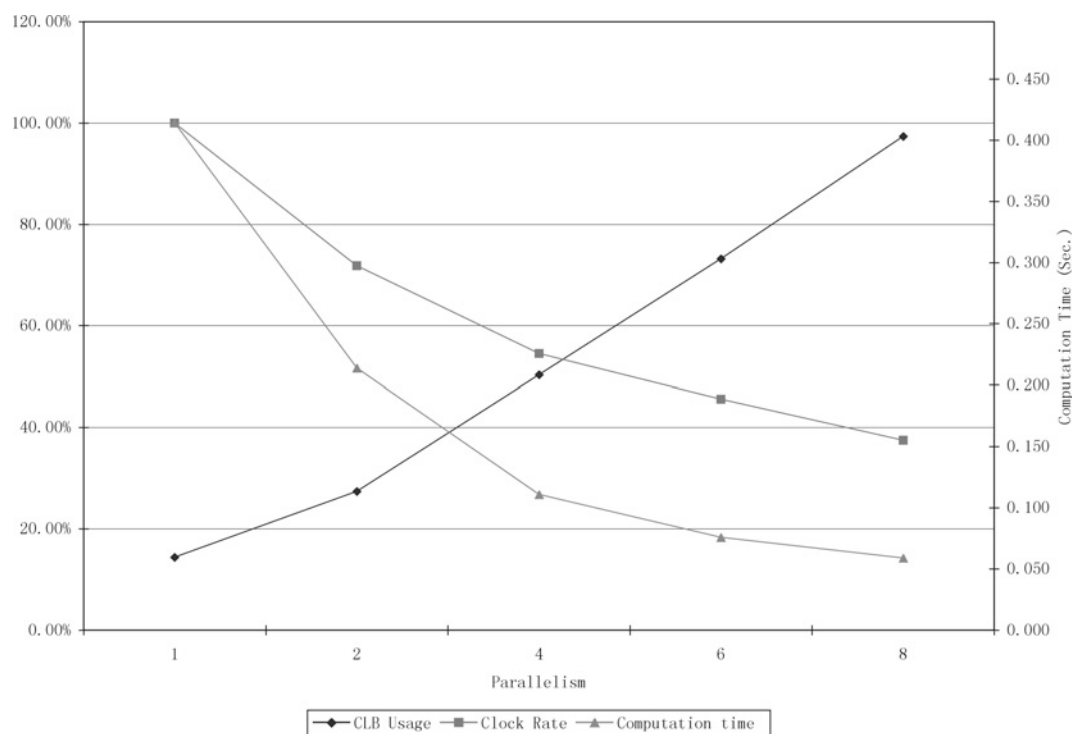


Figure 9 Percentages of CLB usage, performance and clock speed in relation to the number of parallel function components used for computing the edit-distance benchmark solution

Table 8 Achievable clock speeds for varying limits on cube size for the parameter-specific autocorrelation solution

Cube size (bits)	Clock speed (MHz)
10	35
15	27
21	22
32	15

7 Conclusions and future work

In this work we have suggested that there are two types of solutions to problems to be solved in hardware: instance-specific solutions and parameter-specific solutions. Within these categories we have applied and investigated solutions to two particular problems. In applying reconfigurable solutions to these problems, we found in general that, as was expected, as more components were added to the FPGA the clock speed at which the FPGA was capable of processing decreased. This is logical, since additional routing was required to connect parallel components and the component(s) managing them. In our opinions, the extra hardware required for the managing component(s) was negligible, and thus should not have contributed significantly to the reduction in clock speed. We have theorised that the major contributor to this decrease is the additional routing and resulting congestion. Future work is needed to clarify

how these elements interact and contribute to the clock speed reduction.

Despite this, the general result is that additional parallelism generally resulted in faster overall computation despite the slower clock speed – indicating that the computation advantage in the parallelism outweighed the disadvantage of slower communications and additional overhead.

The three solutions that were examined exhibited varying possibilities for parallelism. The edit distance problem required minimal additional communications structures as additional computing components were added, and the parameter-specific autocorrelation solution behaved in a similar manner. However, the instance-specific autocorrelation solution addition of processing elements required communication structures from each one of these to the controller, and so the overhead of adding more elements was higher than in the other two solutions. This overhead was generally reflected in sharp decreases in the achievable clock speed, which imposed a limit on the performance improvements that could be attained by added parallelism.

Future work in this area includes the development of a standardised technique for decomposing problems into sub-problems that can be addressed in this manner, since such success was achieved for these problems. An automated tool is in development that assists with the configuring of a hardware specification to determine the maximum parallelism while

fitting within the target FPGA resources. In addition, we hope to build upon this success by applying our technique to other problems that have not, in the past, been well suited to FPGA speed-up such as those involved with database processing.

8 References

- [1] HAUCK S., DEHON A.: 'Reconfigurable computing: the theory and practice of FPGA-based computation' (Morgan Kaufmann, 2007)
- [2] KENT K.B., RICE J.E., VAN SCHAICK S., EVANS P.A.: 'Hardware-based implementation of the common approximate substring algorithm'. Proc. Euromicro Symp. Digital Syst. Design: Architectures, Methods and Tools (DSD), 2005, pp. 314–320
- [3] YAMAGUCHI Y., MIYAJIMA Y., MARUYAMA T., KONAGAYA A.: 'High speed homology search using run-time reconfiguration'. Proc. Field-Programmable Logic and Applications (FPL) 2002. Springer, Lecture Notes in Computer Science (LNCS, **2438**), pp. 281–291
- [4] RITTER J., MOLITOR P.: 'A partitioned wavelet-based approach for image compression using FPGAs'. Proc. IEEE 2000 Custom Integrated Circuits Conf. (CICC), 2000, pp. 547–550
- [5] SUCHITRA S., LIM C.S., SRIKANTHAN T.: 'Array based architecture for EZW image encoding on FPGA using Handel-C'. Conf. Record of the Thirty-Eighth Asilomar Conf. Signals, Systems and Computers, 2004, vol. 1, pp. 447–450
- [6] SERRA M., KENT K.: 'Using FPGAs to solve the hamiltonian cycle problem'. Proc. Int. Symp. Circuits and Systems (ISCAS'03), 25–28 May 2003, Bangkok, Thailand, IEEE Press), pp. III–228–III–231
- [7] SATO T., FUKASE M.: 'Reconfigurable hardware implementation of host-based IDS'. Proc. 9th Asia-Pacific Conf. Commun. (APCC), 2003, vol. 2, pp. 849–853
- [8] GALANIS M.D., KITSOS P., KOSTOPOULOS G., SKLAVOS N., KOUFOPAVLOU O., GOUTIS C.E.: 'Comparison of the hardware architectures and FPGA implementations of stream ciphers'. Proc. 11th IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS), 2004, pp. 571–574
- [9] KENT K.B., PROUDFOOT R.B., ZHAO Y.: 'Optimizing the edit-distance problem'. Proc. 17th Int. Workshop Rapid System Prototyping (RSP), Chania, Crete, June 2006, pp. 14–16
- [10] KENT K.B., RICE J.E., RONDA T., YONG Z.: 'Instance-specific versus parameter-specific circuit generation'. Proc. Int. Conf. Eng. Reconfigurable Syst. Algorithms (ERSA), 2005, pp. 243–246
- [11] RICE J.E., KENT K.B.: 'Systolic array techniques for determining common approximate substrings'. Proc. Int. Symp. Circuits and Systems (ISCAS), 2006 cdrom paper 1480.pdf
- [12] ZHAO Y.: 'Maximizing performance of configurable hardware resources', Master's Thesis, University of New Brunswick, 2006
- [13] BANERJEE U., EIGENMANN R., NICOLAU A., PADUA D.: 'Automatic program parallelization', *Proc. IEEE*, 1993, **181**, (2), pp. 211–243
- [14] PANDA P., CATTLOOR F., DUTT N.D., ET AL.: 'Data and memory optimization techniques for embedded systems', *ACM Trans. Des. Autom. Electron. Syst.*, 2001, **6**, (2), pp. 149–206
- [15] PANDA P.R., DUTT N.D., NICOLAU A.: 'Local memory exploration and optimization in embedded systems', *IEEE Transactions on Computer-Aided Des. Integr. Circuits Syst.*, 1999, **18**, (1), pp. 3–13
- [16] ANDERSON M., AMARASINGHE S., LAM M.S.: 'Data and computation transformations for multiprocessors', *ACM SIGPLAN Notices*, 1995, **30**, (8), pp. 166–178
- [17] MANJIKIAN N., ABDELRAHMAN T.S.: 'Fusion of loops for parallelism and locality', *IEEE Trans. Parallel Distrib. Syst.*, 1997, **8**, (2), pp. 193–209
- [18] RINARD M.: 'Analysis of multithreaded programs' (Spring Lecture Notes in Computer Science, Germany, 2001), vol. 2126, pp. 1–19
- [19] JERRAYA A.A., YOO S., BAGHDADI A., LYONNARD D.: 'Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip'. Proc. 38th Conf. Design Automation (DAC'01), 2001, pp. 518–523
- [20] JANTSCH A., TENHUNEN H.: 'Networks on chip' (Kluwer Academic Publishers, 2003)
- [21] SALDANA M., SHANNON L., CHOW P.: 'The routability of multiprocessor network topologies in FPGAs'. Proc. IEEE/ACM Int. Workshop System-Level Interconnect, 2006, pp. 49–56
- [22] SALDANA M., SHANNON L., YUE J.S., BIAN S., CRAIG J., CHOW P.: 'Routability of network topologies in FPGAs', *IEEE Trans. Very Large Scale Integration (VLSI) Syst.*, 2007, **15**, (8), pp. 948–951
- [23] MILDER P.A., FRANCHETTI F., HOE J.C., PUSCHEL M.: 'FFT Compiler: from math to efficient hardware HLDVT invited short paper'. Proc. IEEE Int. High Level Design Validation and Test Workshop (HLDVT), 2007, pp. 137–139

- [24] MILDER P.A., AHMAD M., HOE J.C., PUSCHEL M.: 'Fast and accurate resource estimation of automatically generated custom DFT IP cores'. Proc. Fourteenth ACM/SIGDA Int. Symp. Field Programmable Gate Arrays (FPGA), 2006, pp. 211–220
- [25] CONG J., MINKOVICH K.: 'Optimality study of logic synthesis for LUT-based FPGAs'. Proc. Fourteenth ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA), 2006, pp. 33–40
- [26] CONG J., MINKOVICH K.: 'Optimality study of logic synthesis for LUT-based FPGAs', *IEEE Trans. Computer-Aided Des. Integr. Circuits Syst.*, 2007, **26**, (2), pp. 230–239
- [27] CHEUNG R.C.C., BROWN A., LUK W., CHEUNG P.Y.K.: 'A scalable hardware architecture for prime number validation'. Proc. 2004 IEEE Int. Conf. Field-Program. Technol., 2004, pp. 177–184
- [28] KARPOVSKY M.: 'Finite orthogonal series in the design of digital devices' (John Wiley & Sons, 1976)
- [29] CROW J.E., MUZIO J.C., SERRA M.: 'The use of autocorrelation coefficients for variable ordering for ROBDDs'. Proc. 4th Int. Workshop Appl. Reed-Muller Expansion in Circuit Design (RM99), 1999, pp. 185–196
- [30] RICE J.E.: 'Autocorrelation coefficients in the representation and classification of switching functions'. PhD thesis, University of Victoria, 2003
- [31] KENT K.B., RICE J.E.: 'Using instance-specific circuits to compute autocorrelation coefficients'. Proc. 1st Northeast Workshop on Circuits and Systems (NEWCAS), Montreal, Canada, 14–16 June 2003, pp. 61–64
- [32] WAGNER R., FISCHER M.: 'The string-to-string correction problem', *J. ACM*, 1974, **21**, (1), pp. 168–173
- [33] CHURCHILL D., GILLARD P., HAMILTON M., WAREHAM T.: 'Prototyping parallel sequence edit-distance algorithms in FPGA hardware'. Proc. 14th Annual Newfoundland Electrical and Computer Engineering Conference (NECEC), 2004
- [34] LIPTON R.J., LOPRESTI D.: 'A systolic array for rapid string comparison'. Proc. Chapel Hill Conf. VLSI, 1985, pp. 363–376
- [35] RICE J.E., MUZIO J.C.: 'Methods for calculating autocorrelation coefficients'. Proc. 4th Int. Workshop on Boolean Problems, (IWSBP), 2000, pp. 69–76
- [36] THORNTON M., SHIVAKUMARAIAH L.: 'Computation of disjoint cube representations using a maximal binate variable heuristic'. Proc. IEEE Southeastern Symp. System Theory, 2002, pp. 417–421
- [37] RUDELL, R.: 'Espresso minimization tool man pages.' <http://www.fke.utm.my/downloads/espresso/espresso.1.html>
- [38] BRYANT R.: 'Graph-based algorithms for boolean function manipulation', *IEEE Trans. Compu.*, 1986, **C-35**, (8), pp. 677–691
- [39] RUDELL, R.: Tutorial on espresso, 2008. <http://www.fke.utm.my/downloads/espresso/espresso.5.html>
- [40] YANG, S.: 'Logic synthesis and optimization benchmarks user guide version 3.0'. Downloaded from <http://www.cbl.ncsu.edu/xBed/datasets/BCSP/LogSynth91/1991-IWLSUG-Saeyang/1991-IWLSUG-Saeyang.pdf>
- [41] Xilinx Inc. Xst user guide, 2008
- [42] PUTTEGOWDA K., WOREK W., PAPPAS N., DANDAPANI A., ATHANAS P., DICKERMAN A.: 'A run-time reconfigurable system for gene-sequence searching'. Proc. 16th Int. Conf. VLSI Design, 2003, p. 561