An FPGA-based Parallel Sorting Architecture for the Burrows Wheeler Transform

José Martínez, René Cumplido, Claudia Feregrino
National Institute of Astrophysics, Optics and Electronics
Computer Science Department
Luis Enrique Erro # 1. Sta Ma. Tonantzintla
Puebla, 72840, México
{josemcr,rcumplido,cferegrino}@inaoep.mx

Abstract

Burrows-Wheeler transform (BWT) has received special attention due to its effectiveness in lossless data compression algorithms. However, implementations of BWT-based algorithms have been limited due to the complexity of the suffix sorting process applied to the input string. Proposed solutions involve data structures combined with hardware architectures aimed at reducing computational complexity. However, advanced data structures are difficult to be implemented directly into hardware architectures as they require sophisticated control units. In this paper we present a novel architecture based on a parallel sorting block to implement the BWT transform. The proposed architecture has been implemented on a Field Programmable Gate Array (FPGA) device providing good performance improvements compared with other reported implementations on FPGAs. Results obtained show a reduction in the number of cycles and an increase in the maximum frequency compared with other works. FPGA implementation results are presented and discussed.

1. Introduction

Several data compression algorithms have been developed based on the BWT transform. By using BWT in the data compression process it is possible to obtain compression ratios close to the best statistical compressors, i.e. the PPM family of algorithms [1]. BWT based compression algorithms have also to be more efficient in terms of computational resources and memory use [2] than PPM type algorithm, however implementations of compression algorithm based on BWT, either in hardware or software, still demand large amounts of memory resources and computational power. Hardware implementations of BWT require a custom-built storage matrix capable of performing shifts and rotations of the input string. The matrix should also allow performing lexicographical sorting of its contents.

FPGA offer a flexible platform for rapid prototyping and implementation of the BWT transform. The reconfigurable hardware property on FPGAs allows easy adaptation and changes in functional requirements. In addition, the FPGA could be used as a first step for prototyping and synthesizing algorithms to very large scale integration (VLSI) technology.

Improvements of traditional *merge sort* and *quick sort* algorithms have been proposed for software implementations of BWT. A suffix list data structure in proposed in [2] leads to antisequential and memory efficient algorithms, the authors also describe a possible architecture to a BWT-based compression system in VLSI, although only few details are given and no hardware results of the complete algorithm are shown.

Direct hardware implementations of merge and quick sort type of algorithms would require sophisticated control units. To tackle this problem, simpler sorting algorithms could be used. In [4], Mukherjee *et al.* implement an area-efficient register-based architecture for solving the suffix sorting problem. In this paper, a scalable FPGA-based architecture for the BWT transform that uses a parallel approach to solve the suffix sorting problem is presented.

The remainder of the paper is organized as follows: Section 2 defines the suffix sorting problem and some strategies to implement it. Section 3 describes the proposed approach based on parallel sorting. Experimental results are shown in Section 4 and finally, conclusions and future work are presented in Section 5.

Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2005)
0-7695-2456-7/05-$20.00 © 2005 **IEEE**

2. Suffix sorting in the BWT

2.1. The suffix sorting

As mentioned, a direct implementation of the BWT implies the use of large amounts of memory resources. For example, an input string of length 100 would require a matrix of 100x100 elements for storing all possible combinations of shifts for that string. Additionally, the matrix would require additional logic to perform the comparison and swapping operations required for sorting its contents.

To avoid the construction of such matrix consider the following: once the matrix is sorted, the last, column denoted as L in Figure 1, corresponds to the output string. Note that the string L is a permutation of the original input string were each character of this output string is a prefix character of the string F in Figure 1. Also note that the string F is a sorted version of the string L and thereby, a sorted version of the input string. For this reason, it is only neccesary to sort the input string to get the last column of the sorted matrix. However, a single sorting iteration might not be enough to get the same the whole matrix sorted. If there are identical characters in the input string, then it is neccesary to replace those characters with their suffix character and to sort again as any times as necessary until identical characters are not present. It is important to say that the sorting is done by each group of identical characters. Those that are not equal remain in their position. In this way, we can recover the output string by only decreasing one unit to these index values. For example, applying the BWT to the string (D,R,D,O,B,B,S) requires two sorting iterations. First, it is necessary to remember the original index of each character: ((0,D),(1,R),(2,D),(3,O),(4,B),(5,B),(6,S)). After a first sorting iteration, the resulting string is ((4,B),(5,B),(0,D),(2,D),(3,O),(1,R),(6,S)). Replacing the corresponding suffix characters where there are identical characters, the following string is obtained: ((4,B),(5,S),(0,R),(2,O),(3,O),(1,R),(6,S)) which should be sorted again to get: ((4,B),(5,S),(2,O,),(0,R),(3,O),(1,R),(6,S)) which is now fully sorted.. The output string corresponds to the indexes (3,4,1,6,2,**0**,5) and that corresponds to the string (O,B,R,S,D,D,B). The key or index, where the first character of the original string is located, in this case it is equal to 5.

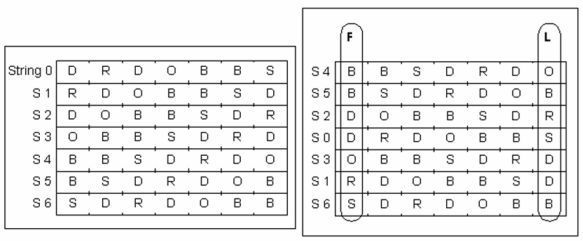

Figure 1. To the left: Matrix built with shifted strings of the original string. To the right: Sorted Matrix with *L* as the output string.

This method is simpler and less expensive in memory resources compared with the direct implementation of BWT. Standard sorting algorithms have around or slightly less than O(*N*log(*N*)) complexity [2]. Their aim is to reduce the computational cost O(n^2) obtained by burble sort or any other simple sorting method. In [5] is reported that a sorting algorithm based on suffix tree structure can be implemented with O(*n*) complexity, specifically for the problem of suffix sorting. A wavesorter algorithm described in [4] has O(*n*) complexity as it needs *4n* steps to sort a string using a single register array. In [2], a novel sorting method based on suffix trees data structures is presented. It also describes briefly an implementation for VLSI technology. However, the proposed architecture requires a large amount of storing memory.

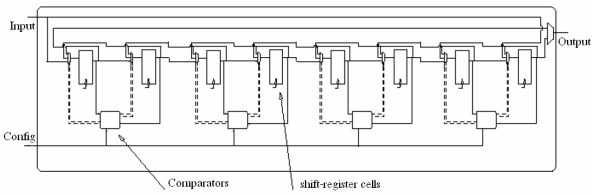

Figure 2. Wavesorter block for 8 data.

2.2. The wavesorter approach

In [4], it is described an FPGA implementation based on wave sorting algorithm proposed in [5]. The wavesorter consists of a group of slightly modified bidirectional shift registers, see Figure 2. The registers can perform shift-right and shift-left operations to store their actual value to the next left or right adjacent register. These registers are grouped into pairs by a comparator block that swaps them if a 'less than' condition is met. The architecture reads the input string, character by character, from memory and stores each character in the wavesorter.

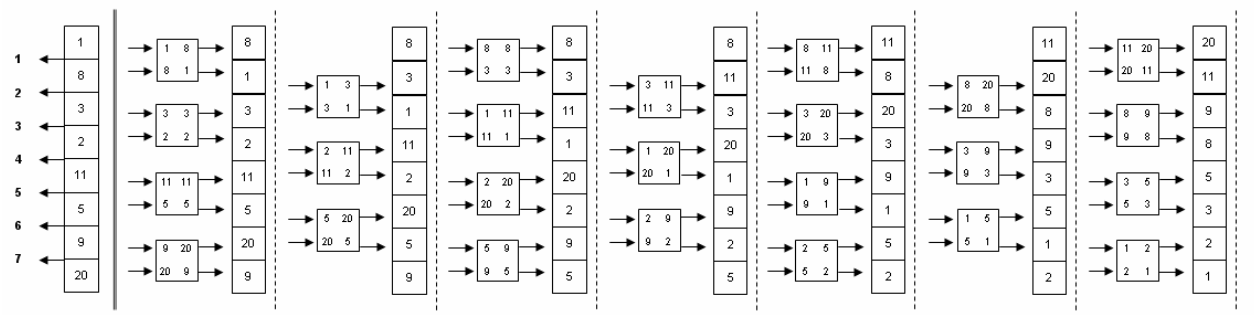

Figure 3. Parallel sorting comparing and swapping in a single sorting iteration.

Every time a character is read, a shift-right operation is performed followed by a comparison operation. When all the characters from the input string are read, a partial sort of the string is found in the registers. To get a complete sort, the string is shifted out by changing the shift direction to the left. When the last character exits, the output string is fully sorted.

2.3. Parallel Sorting strategy

To improve the wavesorter approach, a second level of 2-input comparators were added to the parallel architecture based on shift registers. These shift registers are similar to those used on the wavesorter approach. Assume that there are *n* characters stored in a register array, with *n* being an even number. The adjacency between each two register is enumerated as shown on the left of Figure 3. Registers with odd adjacency number will be referred as *odd adjacency registers*. Registers with even adjacency number will be referred as *even adjacency registers*. Comparisons and swaps are performed only between odd adjacency registers. If the array is not sorted yet, a comparison and swap is performed only between even adjacency registers. If the array is not sorted, then new comparisons and swaps are performed again by switching between the odd and later between the even adjacency registers until the array is fully sorted. Figure 3 shows an example where comparisons and swaps are performed alternating between odd and even adjacency registers.

In figure 3, dotted lines point out that all comparisons and swaps performed to the registers are performed in parallel. In this way, it is possible to perform parallel comparisons and swaps by following the order in which the registers are compared, first odd registers and then even registers. Thus, for an array of *n* data, the number of steps required for a sorting iteration is *n-1*. This number of steps can even be improved by connecting the comparators used with the odd adjacency registers to the comparators used with the even adjacency registers, as shown in Figure 4. Then, the total number of steps for sorting the array is at most *n/2*. This sorting strategy will be referred as *Parallel sorting strategy.*

To make a fair comparison of the parallel sorting strategy against wavesorter strategy in terms of the total number of required steps to sort an array, it is necessary to consider the steps used to read data from memory and the steps required to store the sorted data back to memory. The proposed approach is based on the same structure of the registers array used in the wavesorter strategy. With this kind of array, data can be stored in the array by sending a datum to the first register and later, when the second datum is sent to the first register, the value on the first array is shifted to the second register. Thus, for every datum sent to the array to be stored, values in registers are shifted to their respective adjacent registers. This process requires *n* steps. The same number of steps is required to take data out from the array. This approach allows storing a new set of data in the array while the previous set is being sent back into the memory.

As mentioned in section 2, suffix sorting might imply more than one sorting iterations. If *k* sorts are required, then the parallel sorting requires to ((*n+n/2) * k + n)* to sort an array of *n* data. Thus total number of steps required can be obtained by the following equation:

$$f_{steps}^{PS}(n,k) = n\left(\frac{3}{2}k+1\right) \qquad (1)$$

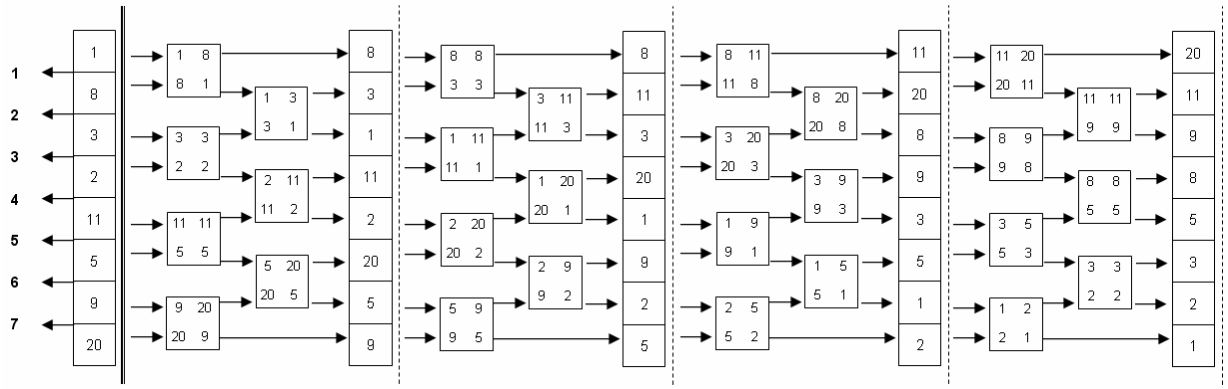

Figure 4. Parallel sorting with two levels of comparators performed in one iteration.

For the wavesorter approach, the authors report the following number of steps to solve the suffix sorting problem:

$$f_{steps}^{WS}(n,k) = 2n(k+1) \qquad (2)$$

The parallel strategy leads to a significant reduction of about 30% compared to the wavesorter approach. Furthermore, in additional sorts the necessary number of steps for sorting is equal to the number of characters in the biggest group of identical characters divided by 2 (remember that an additional sorting is implied if groups of identical adjacent characters appear in the array). This implies that in practice, it is possible to reduce more than 30 % the number of steps to solve the suffix problem. Experimental results confirm this.

3. Proposed Architecture

This section describes the proposed parallel architecture for suffix sorting. Xilinx's System Generator v6.3 for Simulink and ISE v6.3. were used to implement the architecture.

3.1. The parallel sorting block

As it was explained in section 2.3, it is possible to perform a sorting iteration in a single step. The input of the first level of comparators is read directly from the registers and its outputs are then used as inputs for the second level. The output of second level of comparators is written back to the registers so that a new sorting iteration can be started in case the array is not fully yet. A number of multiplexers are used to select the input for the registers. These inputs can be selected form the upper adjacent register or the output of the second level of comparators. Figure 5 shows a block diagram for the proposed parallel sorting block. For the sake of clarity, the design shown has only 8 registers. The first register has a direct input that comes from the memory. A data is read from memory and sent to the first register in the block every clock cycle. When the storage is finished, signals of multiplexers change to '0' and then, in the following clock's cycles, the sorting is performed. When sorting is finished, signals of multiplexer change again to '1' and then, sorted data is read from the block through the last register that is connected to the output port.

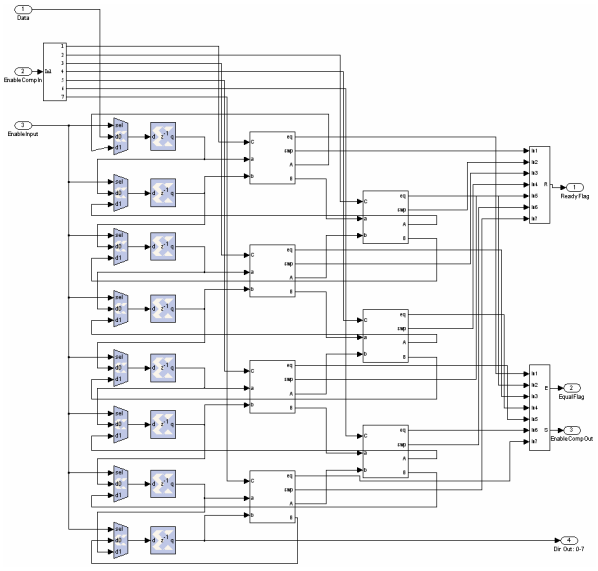

Figure 5. Sorting block for 8 data.

3.2. The comparator block

An underlying block in the parallel sorting block is the comparator block used to build the first and second level of comparators. The comparator block is shown in Figure 6. The inputs of this block are: (1) a signal that enables or disables the comparison, (2) datum *a* and (3) datum *b*. Their outputs are: (1) an *equal flag*

signal that indicates if data *a* and *b* have the same value, (2) an *swp flag* signal that indicates if swapping was performed, (3) datum A and (4) datum B with the corresponding value whether a swapping was performed or not.

The sorting is finished when all *swp flags* are set to '0', i.e. not swapping was performed in the first or second level of comparators. *Equal flags* are used to identify if there are still groups of identical data, in such case a suffix substitution and further sorting are required. These *equal flags* are used to identify these groups and then, for the next sorting, enable the appropriate comparators. The rest of data should remain without changes.

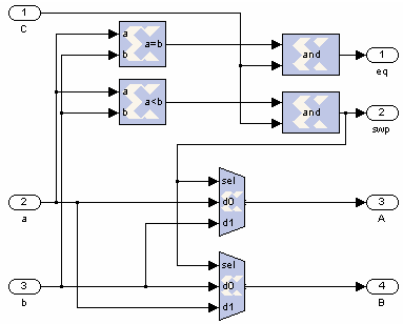

Figure 6. The comparator block.

3.3. Memory and registers

The general memory architecture has an 8-bit serial input port. An extra bit is used to create a sentinel character, namely '$'. This sentinel character has the decimal value 511 which is bigger that any other ASCII character and is used to prevent an infinite sorting cycle. For example, the string "aaaaaaa" is sorted already, but the architecture's control realizes that a suffix substitution is still needed. To avoid this problem, a sentinel character was added as follows: "aaaaaaa$", then the suffix substitution will be "aaaaaa$a", where only the first 7 characters will be sorted since the character in positions 7 and 8 were different and do not need to be sorted again.

When a datum is transmitted to the parallel block, a binary number of 7 bits that represents the address of the data in memory is concatenated. The registers of the parallel sorting block have a word length of 16 bits where the first 9 refer to the ASCII value of the character, and the last 7 are used to store the original memory address. Thus, in suffix substitution it is only replaced the corresponding suffix value but without deleting the original address of the data.

3.4. Architecture description

Figure 7 shows the proposed architecture. An input string of *n* data is stored in memory through the serial port input. Once data is stored in memory, they are transferred to the parallel registers of the sorting block to begin the sorting process. If the control detects that still there are identical data, it gets the address of the data that are being taken out from the sorting block. This means taking the 7 most significant bits of the last register. The number of sorts is added to this number to obtain the correspondent suffix data. This new address is sent to memory to obtain the suffix data that is sent to the parallel sorting block.

While data are being read from the sorting block, suffix data are sent from memory back to the sorting block, thus a new sorting iteration can start. This iteration continues until data are sorted and no groups of identical data remain. Then data from parallel sorting block is read with shifting operation again but instead of adding the sorts counter, address are decreased by one. The datum read from memory is sent directly to one of three output ports of the architecture. Figure 7 shows the three serial output ports of the architecture. The second one is a 1-bit flag that indicates with '1' that the signal of the third output is the final result and with '0' if else. The first output port is 1-bit flag that indicates which character at the output corresponds to the key.

4. Experimental results

A Virtex 2 xc2v2000-6bf957 was used for the FPGA implementation. The parallel sorting architecture works with strings of 128 characters, 127 characters plus the sentinel character. The 128 number was chosen to take advantage of using in full the binary numbers required to address data, in this case 7 bits. The Place&Route report for this implementation is presented in Table 1.

To perform the experiments, 9 random strings of 127 characters were extracted from this text were used as inputs to test the proposed architecture. Results are shown in Table 2 where the performance is compared against the wavesorter approach. Equation 2 was used to calculate the number of steps needed by wavesorter approach. The number of cycles and the maximum clock frequency are included.

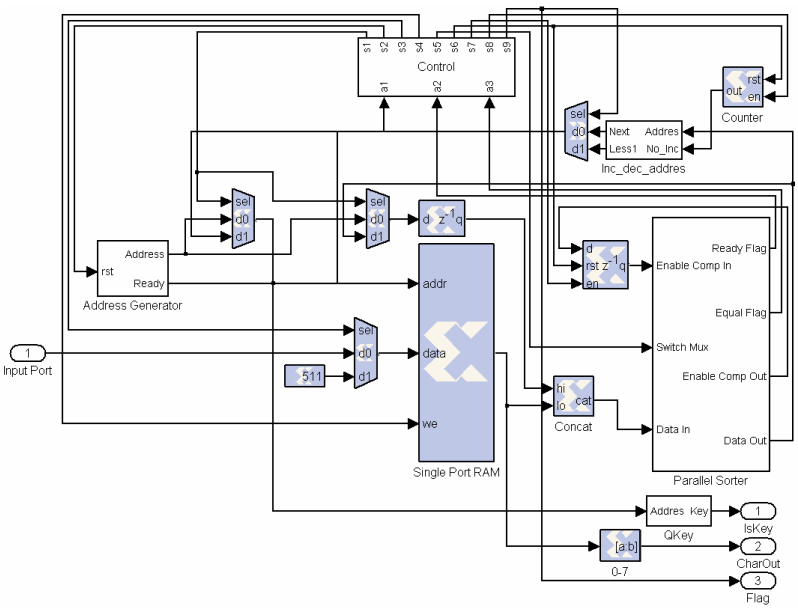

Figure 7. Parallel Sorting Architecture.

The wavesorter approach was implemented for an array of 100 data on a Virtex XCV300-4 BG352 and runs at a frequency of 45 MHz. This implementation used an 88% of slices from a total of 3072 which makes a usage of 2703 slices.

Table 1. Place & Route report

Device	Resources	Usage
xc2v2000-6bf957	**Usage/Total**	**(%)**
External IOBS	19 / 624	3 %
RAMB16s	1 / 56	1 %
Slices	**4316 / 10752**	**40 %**
BUFGMuxs	1 / 16	6 %
Max. path delay:	**Max. Clock Frequency**	
19.351 ns	**51.67** MHz	

In addition, Table 2 shows the results of running the parallel sorting. The total number of cycles used by parallel sorting architecture was obtained by multiplying 129*(sorts+1) and adding the total number of cycles from the second column. When the number of cycles per sort is equal to 1, the control detects that data is already sorted and continues with substitution stage. Last row in Table 2 shows the results of running an especial case where all 127 characters have the same value. It shows the ability of the control to identify that data are sorted and that there is not need to spend more cycles and starting thus the suffix substitution. The wavesorter approach needs to perform the entire sequential storing and taking out of data to perform the sorting without taking advantage of this kind of input. The comparison shows a reduction in the number of cycles in more than 40%.

5. Conclusions

We have presented an architecture that implements the BWT transform based on a parallel sorting block. This approach reduces more than 40% the number of cycles required to perform the complete task compared with previous solutions. This task includes the solution of the suffix sorting problem and the generation of the output string corresponding to the BWT. The proposed architecture can be scalable to more than 128 characters without significant changes on the control and sorting blocks.

Future work includes the implementation of a new storing strategy that allows a reduction in the latency associated with sequential access of data. Also, pipeline registers can be between the two levels of comparators to reduce the critical path delay and thus, increase the maximum frequency.

A full implementation of a BWT compression algorithm is under way; this involves the integration of the proposed architecture with Move to Front, Run Length Encoding and Entropy Encoder blocks.

6. Acknowledgments

Figure 1 and figure 2 were taken from [6] and [4] respectively. We want to thank to anonymous reviewers whose comments were useful to improve this

work. This work is in part supported by a grant from CONACYT under No. 189903.

7. References

[1] P. Fenwick, "Block Sorting Text Compression - Final Report", Technical Report 130, Auckland, New Zealand, ISSN 1173-3500, April 1996.

[2] D. Baron and Y. Bresler, "Antisequential Suffix Sorting For BWT-Base Data Compression", IEEE Transactions on Computers, Vol. 54, No4, April 2005.

[3] M. Burrows and D.J. Wheeler, "A Block-Sorting Lossless Data Compression Algorithm," SRC Research Report 124, Digital Systems Research Center, Palo Alto, Calif., May 1994.

[4] A. Mukherjee, N. Motgi, "Prototyping of Efficient Hardware Algorithm for Data Compression in Future Communication Systems", Proceedings of the 12th International Workshop on Rapid System Prototyping, (RSP'01), IEEE, 2001.

[5] M. Effros, "PPM Performance with BWT Complexity: A Fast and Effective Data Compression Algorithm", Proc. IEEE, vol. 88, no. 11, pp. 1703-1712, Nov. 2000.

[6] M. Nelson, "Data Compression with the Burrows-Wheeler Transform", Dr. Dobb's Journal, September, 1996.

Table 2. Results for the parallel sorting architecture over 10 tests.

Max. Frequency wavesorter 45 MHz. Device Virtex xcv300-4 bg352				
Max. Frequency Parallel Sorting architecture 51.67 MHz. Device Virtex 2 xv2v2000-6bf957				
String Length = 128 characters (127 + sentinel)				
Sorts	**Cycles per sort**	**Total cycles / time (*ms*) for BWT**		**Improvement Cycles / time (%)**
		wavesorter	**Parallel sorting**	
3	57,9,1	1024 / 22.75	583 / 11.28	43.06 / 50.41
3	59,5,1	1024 / 22.75	581 / 11.24	43.26 / 50.59
5	60,7,2,1,1	1536 / 34.13	845 / 16.35	44.98 / 52.09
6	55,8,2,1,2,1	1792 / 39.82	972 / 18.81	45.75 / 52.76
7	61,9,2,1,1,1,1	2048 / 45.51	1108 / 21.44	45.89 / 52.88
8	58,7,3,1,1,1,1,1	2304 / 51.20	1234 / 23.88	46.44 / 53.35
8	56,5,2,1,1,1,1,1	2304 / 51.20	1229 / 23.78	46.65 / 53.55
13	54,8,2,2,1,1,1,1,1,1,1,1,1	3584 / 79.64	1881 / 36.40	47.51 / 54.29
23	56,4,2,1,1,1,1,1,1,1,1,1,…,1	6144 / 136.53	3178 / 61.50	48.27 / 54.95
127	1,1,1,1,1,1,1,1,1,1,1,1,…,1	32768 / 728.17	16639 / 322.02	49.22 / 55.77