

Accelerated Costas Array enumeration using FPGAs

Jim Devlin
Complex & Adaptive Systems Laboratory
University College Dublin
Ireland

Scott Rickard
Complex & Adaptive Systems Laboratory
University College Dublin
Ireland

School of Electrical, Electronic & Mechanical Engineering
University College Dublin
Ireland

School of Electrical, Electronic & Mechanical Engineering
University College Dublin
Ireland

Abstract—Costas array enumeration is an NP-complete problem with a highly parallelize-able solution. This paper examines the implementation of a solution to this problem on an FPGA platform and examines the elements of what makes the most efficient solution to this problem. This paper compares the performance of the hardware solution against the performance of the best known software solution and finds an approximate 40 times speedup from using hardware.

I. INTRODUCTION

A. What is a Costas array

A Costas Array is a square array of dimension N , containing N dots that satisfies three properties:

- 1) One dot per a row
- 2) One dot per a column
- 3) The vectors joining each of the ${}^N C_2$ pairs of dots are distinct

Costas Arrays have a practical application in RADAR and SONAR as they have near ideal range-Doppler ambiguity properties.

Costas Arrays were discovered by John Costas in the 1960s and he published a work on his discovery in 1984 [1]. Techniques for generating a Costas Array have been discovered, the first of these in [2] in 1984. These generation techniques are generally computationally light but they have certain limitations, as they cannot generate every Costas Array and no technique exists for generating Costas Arrays at certain sizes.

							O
	O						
				O			
O			O				
		O					
							O
					O		

TABLE I
AN EXAMPLE 8 X 8 COSTAS ARRAY

B. Significance of Costas array enumeration

Costas Arrays have also been found by inspection, guess work and enumeration. Enumeration is the brute force, exhaustive approach where one finds every Costas Array for a given dimension, N , by trial and error. Enumeration has been used to find many Costas Arrays for which no known generation technique exists. Every Costas Array up to size 26 has been found by enumeration [3]. No Costas Arrays have been found for certain sizes, the lowest of which are 32 and 33.

Predicting the number of Costas Arrays for a given size has been the subject of some debate [2], [4]. Golomb and Taylor [2] predicted an ever-increasing number of Costas Arrays for a given N (denoted hereafter as $C(N)$), in their initial paper in the field. Silverman et al. [4] disproved this when they enumerated all 17×17 Costas Arrays and in fact predicted that there would probably be no Costas Arrays at size 32.

Golomb and Taylor [2] also predicted that Costas Arrays existed for all N . It would be of some interest to enumerate all 32×32 Costas Arrays as that may disprove the above prediction. Enumerating new sizes increases our understanding of Costas Arrays and gives researchers more material to use in the search for new generation techniques.

C. Goals of research

We propose a hardware-based solution to the enumeration of Costas arrays that we hope is more efficient. Our metric for efficiency is the cost of enumerating all Costas arrays of a given dimension or the performance per a euro. We examine exactly how well this solution performs and we aim to significantly extend the affordable search range of complete enumeration, compared to previous software/CPU based solutions. The ultimate goal in this line of research is the complete enumeration of all Costas arrays of dimensions 32 and 33.

We aim to demonstrate that hardware enumeration works effectively on a small scale by enumerating a previously covered result more quickly with an FPGA than can be accomplished with the best available software approach running on a computer, with the value of the computer and the FPGA normalised.

II. TESTING A COSTAS ARRAY

There are two ways of describing a potential Costas array: as a grid of dots and as a series of integers.

								O	
	O								
				O					
			O						
O									
		O							
									O
					O				
4	7	3	5	6	1	8	2		

TABLE II
OUR EXAMPLE 8x8 COSTAS ARRAY

With the graphical representation, one can attempt to test the array by inspection, which is very unreliable at anything but the smallest sizes.

A. The Difference Triangle

With the integer representation, we construct the difference triangle (DT) to test the array, as shown below. If each row of the difference triangle contains distinct entries, then the vectors joining pairs of dots are distinct and property 3 above is satisfied. Properties 1 and 2 should be obvious.

4	7	3	5	6	1	8	2
3	-4	2	1	-5	7	-6	
	-1	-2	3	-4	2	1	
		1	-1	-2	3	-4	
			2	-6	5	-3	
				-3	1	-1	
					4	-5	
						-2	

TABLE III
A DIFFERENCE TRIANGLE

The triangle is constructed as follows: the 0th row is the array, for the first row, find the differences between adjacent elements of the array; for the second row, find the differences between elements of the array with a one-element gap between; for the third row, find the differences between array elements with a two-element gap in between and so on. This represents the standard method of testing an array.

It is in fact only necessary to test the top half of the triangle [5], as there will not be a problem in the bottom half unless there is a problem in the top half. Elements in the first row of the difference triangle are sometimes referred to as first differences or 1-jumps and so on for other rows. So, for a dimension n Costas array it is only necessary to check elements in rows, r , in the range:

$$|r| \leq \left\lfloor \frac{n-1}{2} \right\rfloor \quad (1)$$

6	8	3	7	10	11	4	9	5	2	1
2	-5	4	3	1	-7	5	-4	-3	-1	
	-3	-1	7	4	-6	-2	1	-7	-4	
		1	2	8	-3	-1	-6	-2	-8	
			-4	-3	-1	-2	5	9	3	
				5	-4	6	-2	-8	-10	
					-2	1	2	-5	-9	
					3	-3	-1	-6		
						-1	-6	-2		
							-4	-7		
								-5		

TABLE IV
A DIFFERENCE TRIANGLE WITH NON-ESSENTIAL ELEMENTS IN BOLD UNDER THE LINE

III. HIGH-LEVEL DESIGN DECISIONS

A. Design goal

A modern FPGA has the device resources to support multiple cores, each acting as a separate entity searching for Costas arrays. We feed each of these cores a starting position or stub, the first few dots of a partial Costas array. Each core will enumerate many stubs over the course of a full search. This allows us to have many cores working in parallel on the problem with negligible overhead. Therefore, all design decisions are based on the desire to get the best balance between the size of a core and the speed of a core (in terms of clock cycles per a second and what kind of operation it carries out in each clock cycle).

B. Testing all permutations

There are several ways to approach enumerating Costas arrays. The obvious one seems to be to try every possible permutation of N integers. This gives us $N!$ arrays to check. For 5×5 this is fine, as we have 120 possible candidates.

For $N=32$, this is a hopeless approach. Assuming, we could check 100 million arrays a second and given that $32!$ is about 10^{35} , this means it would take 2.6×10^{27} seconds or 8×10^{19} years. To put this in context the universe is about 13.7 billion years old and it would take 5.8 million times this to enumerate all $32!$ possibilities using this approach.

Clearly, this is not how to proceed.

C. Backtracking

We can vastly reduce the time necessary to enumerate all arrays of a large size by backtracking. Backtracking is an approach where we add dots to a partial array one at a time until we encounter a problem and we have broken one of the rules of Costas arrays, in which case we remove the last dot and try another possibility. This is most simply implemented if the dots are added left-to-right one at a time.

We can see when we stop at 123xx and backtrack to 12xxx to proceed to 124xx then we skip the two possibilities 12345 and 12354. We have saved only two tests but probably generated loads of extra ones. In fact this technique is arguably less efficient at $N=5$, needing about 400 tests in place of $5! = 120$ tests, although the tests on a partial array are less computationally intensive. For 8×8 Costas arrays, when

we stop at 3x3, we have cut out 5! ways of arranging the remaining dots, which is 120 tests. In the case of 32x32, we have saved 80 trillion years of searching by the earlier calculations, every time we stop and find a dead end at our third position in. At the higher sizes, this technique becomes very effective.

D. Lexicographical Ordering

It is necessary to test partial arrays in a non-random order to avoid storing a memory of every partial array tested so far. One obvious order to test them in is to start with the lowest possible arrangement of numbers (i.e. 12345) and finish with the highest (i.e. 54321). This is referred to as lexicographical ordering.

E. Incrementing

There are several improvements that can be made to the above approach. Firstly, we so far have only 2 operations, adding a new dot and taking a bad dot away. We can add a third operation an increment operation. When we go 123xx to 12xxx to 124xx, we could have incremented and gone directly from 123xx to 124xx, saving an operation. This will change every backtrack-newdot into an increment operation.

F. Families and Symmetry

A property of Costas arrays is that if we take a Costas array and flip (i.e. reflect) it left-right, we get a new Costas array. We can flip these two arrays up-down to get four Costas arrays when we thought we had one. Furthermore, we can flip these four arrays diagonally to (sometimes) get four new Costas arrays. Thus Costas arrays are grouped in families. This means if we enumerate all Costas arrays, we will have found every array four or eight times. If we could remove the possibility of finding Costas arrays for which we already had a family member, we could reduce our search field to less than a quarter of its original size.

We refer to the relationship between members of the same family as symmetry.

1) *Frames*: A frame is a partial array with one dot in each of the first and last columns and one dot in each of the top and bottom rows. If we select these frames appropriately, we can eliminate nearly all repetition of Costas arrays in our search. The problem this presents is that it would remove our simple left-to-right method of adding dots to partial arrays. Therefore, we never used it.

2) *Leveraging symmetry to shrink the search tree*: Ideally, we would search only for one member of each family, reducing our search tree to nearly one eighth of its original size for large N. However, some of these symmetries are not trivial to take advantage of and therefore taking advantage of all symmetries would greatly complicate the design.

We may denote an array as (r_1, r_2, \dots, r_n) where there is a dot at (i, r_i) . We can model symmetry between family members as three types of flips:

- 1) an upside-down flip $(i, r_i) \rightarrow (i, n + 1 - r_i)$
- 2) a left-right flip (mirroring) $(i, r_i) \rightarrow (n + 1 - i, r_i)$

3) a diagonal flip (We chose a bottom-left to top-right diagonal axis to flip around) $(i, r_i) \rightarrow (r_i, i)$

3) *The upside-down flip*: By far the easiest symmetry to take advantage of is a flip upside-down of the array.

What this means in practice is that we only search for arrays that (roughly speaking) have their leftmost dot between 1 and $(\frac{n}{2})$. This immediately halves the search space.

The reason for this is simple, if we have already searched arrays with dots in the bottom half of the first column, we can flip them upside down and already have every array that starts with a dot in the top half of the first column.

It turns out, for odd arrays, we do not need to search the middle space in the first column and for even arrays, where the middle lies between two spaces, we need not search either of these spaces. So we only need to search Costas arrays where the first element, r_1 satisfies:

$$1 \leq r_1 \leq \left\lfloor \frac{n-1}{2} \right\rfloor \quad (2)$$

4) *Why we do not search those middle positions (odd arrays)*: Consider 5×5 array searching where we have searched every array starting with either a 1, 2, 4 or 5 in the first column. Now consider that every array ending in a 3 cannot start with a 3, so we have found every array ending in a 3 by searching arrays beginning 1, 2, 4 or 5. If we have searched arrays starting 1 and 2, we can flip vertically for arrays starting 4 and 5 and can then flip horizontally for arrays starting 3 (and delete any double appearances).

This argument can be extended to any odd sized array.

5) *Why we do not search those middle positions (even arrays)*: A similar argument to the odd n argument above holds for not searching middle positions in 6×6 arrays. If we search arrays starting 1, 2, 5 and 6, we have found every array ending in a 3, except those starting in a 4 ($4 \times x \times x \times 3$) and arrays ending in 4 that start in a 3 ($3 \times x \times x \times 4$).

An array starting 3 and ending 4 is $(3 \times x \times x \times 4)$. This flips diagonally to $(x \times x \times 1 \times 6 \times x)$. Similarly, $(4 \times x \times x \times 3) \rightarrow (x \times 6 \times 1 \times x)$.

Therefore, if we perform a diagonal flip of all arrays of the form $(x \times x \times 6 \times 1 \times x)$ or $(x \times x \times 1 \times 6 \times x)$, we get all arrays of the form $(4 \times x \times x \times 3)$ and $(3 \times x \times x \times 4)$. Having run other searches, we will already have all arrays of the form $(x \times x \times 6 \times 1 \times x)$ or $(x \times x \times 1 \times 6 \times x)$, other than those of the form $(3 \times x \times 1 \times 6 \times 4)$ or $(4 \times x \times 6 \times 1 \times 3)$. This is because we have no arrays of the form $(4 \times x \times x \times 3)$ and $(3 \times x \times x \times 4)$ at all, $(3 \times x \times 1 \times 6 \times 4)$ and $(4 \times x \times 6 \times 1 \times 3)$ being a subset of these.

However, $(3 \times x \times 1 \times 6 \times 4)$ and $(4 \times x \times 6 \times 1 \times 3)$ can never be Costas arrays.

3	x	1	6	x	3
x	x	+5	x	x	x
	-2	x	x	-2	

TABLE V

A $(3 \times 1 \times 6 \times 4)$ CAN NEVER BE COSTAS

The same will happen with $(4 \times 6 \times 3)$ with differences of +2. Furthermore, in general, for even n , the $(\frac{n-2}{2})^{th}$ row of the difference triangle will contain a pair of differences of $\pm(\frac{n-2}{2})$ in the first and last positions of that row. This will happen for all arrays of such a form:

$$((\frac{n}{2}), x, x, \dots, x, 1, n, x, x, \dots, x, (\frac{n}{2} + 1)) \text{ or}$$

$$((\frac{n}{2} + 1), x, x, \dots, x, n, 1, x, x, \dots, x, (\frac{n}{2})).$$

So to sum up, if we search all arrays starting 1 and 2 and then flip vertically, we have every array starting 1, 2, 5 or 6. If we flip horizontally, we have every array starting 1, 2, 5 or 6 and every array starting 3 or 4 other than those of the form $(3 \times x \times x \times 4)$ or $(4 \times x \times x \times 3)$. If we flip diagonally, we will have every array starting 1, 2, 5 or 6 and every array starting 3 or 4 other than those of the form $(3 \times 1 \times 6 \times 4)$ or $(4 \times 6 \times 1 \times 3)$, except no such arrays can exist. Therefore we have every array.

G. Looking ahead

It is possible to further prune the search tree considerably by using past known differences to block dead-end positions in the future. Consider the partial array $(1 \ 2 \ 4 \ 3 \ x \ x)$. We can project the 1st row differences from the position of the rightmost dot onto the next column to see where we should not put a dot. We can do the same with the second row differences, projecting them from the position of the second rightmost dot.

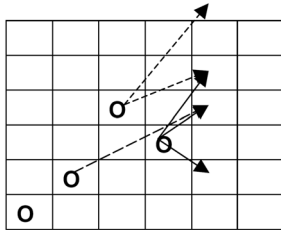


Fig. 1. 1st differences: -1,+1,+2; 2nd differences: +1, +3; 3rd differences: +2

This stops us running several dead-end checks. The extra hardware required is large but the speed boost from using this is much larger.

This can be done quite elegantly in parallel in hardware.

It turns out that the most efficient approach to using look ahead is to only use the top few rows for looking ahead and to check other rows after adding the dot. This gives us a balance between silicon area used and quickness of the search, since lower rows contain less information and we suffer from diminishing returns as positions are blocked multiple times.

H. False positive searching

We noticed that for very large n , the bottom necessary rows of the difference triangle were useful in only the rarest of cases. Therefore, we decided to disclude them from our tests. This saved on chip space very considerably but only increased the search tree quite moderately. The result of the search could then be filtered for false positives in software.

The reason this is effective is that we only check a few hundred or thousand false arrays at the end versus doing millions of checks in the middle while considering partial arrays.

It turns out that an even more efficient approach is to only use the top few rows for looking ahead and to check some other rows after adding the dot and to not check some rows at all until the complete search has finished.

IV. PERFORMANCE, RESULTS AND CALCULATIONS

A. Testing Conditions

Tests were run on a Xilinx Spartan-3 XC3S1000 chip in an FT256 package with a speed grade of -4, on a Digilent Spartan-3 Starter Board (Rev E). The input clock was 25MHz.

We also targeted a Virtex-5 XC5VLX110 in an FF676 package with a -3 speed grade in simulation and clock speed estimates for this are based on information from the Xilinx synthesis tools.

B. Number of blocked paths attributed to each row of the difference triangle

The numbers below indicate which was the highest row to have an error in it, when the checks showed a partial array to be not Costas. This was done with no look ahead and while enumerating a small portion of the 32×32 tree.

Total clock cycles ran	10,004,961
ROW1	4,205,514
ROW2	2,244,571
ROW3	1,176,160
ROW4	552,239
ROW5	273,556
ROW6	129,426
ROW7	65,013
ROW8	9,581
ROW9	1,748
ROW10	137
ROW11	1
ROW12	0
ROW13	0
ROW14	0
ROW15	0
Summation rows 9 to 15	1,886
Summation rows 8 to 15	11,467

TABLE VI
NUMBER OF BLOCKS ATTRIBUTED TO EACH ROW OF THE DIFFERENCE TRIANGLE

C. Applying symmetry

Using symmetry shortcuts, we can reduce out search tree significantly. N is the dimension of the array being enumerated. D is the number of difference triangle rows used for look ahead. $Cores$ is the number of cores placed on the chip. R is the total number of rows used to check arrays.

N	D	Cores	R	Symmetry	Time
15	3	2	7	No	32 seconds
15	3	2	7	Yes	15 seconds

TABLE VII
SAVING OFFERED BY SYMMETRY

D. Enumeration time at various sizes

Below is a table listing times to enumeration for a good set of parameters at various medium-large sizes. These were all timed using a counter inside the chip which stopped when the enumerator had finished. We then calculated the time it had taken by dividing the total number of operations by the number of operations per a second. We employed our symmetry speed-up.

N	D	Cores	R	Time	Time for one core
13	2	1	3	1.5916 seconds	1.5916 seconds
14	3	2	4	2.72 seconds	5.43 seconds
15	3	2	4	15.6574 seconds	31.3148 seconds
16	3	2	4	78.8 seconds	157.5 seconds
17	3	2	4	388.04 seconds	776.07 seconds
18	4	5	5	662.9 seconds	3,314 seconds
19	4	5	5	3,684 seconds	18,422 seconds

TABLE VIII
ENUMERATION TIME AT VARIOUS SIZES

E. Extrapolating to 27 and 32

From the above results, we can calculate the rate of growth in time for one core.

Let x be the overall rate of growth. t_a and t_b denote the times for enumerating $n = a$ and $n = b$. Therefore, the rate of growth is:

$$t_a \times (x)^{b-a} = t_b$$

$$\Rightarrow x = e^{\frac{\ln(\frac{t_b}{t_a})}{b-a}}$$

So for the above results, we could calculate x to be

$$\Rightarrow x = e^{\frac{\ln(\frac{18422}{1.5916})}{19-13}} = 4.76$$

So, extrapolating with this as our typical rate of growth, we can predict a rough enumeration time for each n up to 32 based on one core at 25Mhz and on 20 cores core at 100Mhz, we can see how long each size would take. We chose 20 cores since this is what will fit on a Virtex-5 LX110 at size 27. We also consider this time as years. The unit chip-years is useful as it allows us to compare how long one chip would take with how long many chips would take.

This table was calculated by taking the known result for 19 and multiplying it by our above value of x repeatedly until we got down as far as 32. For other columns, we divided by 3600 seconds per hour and then by 24 hours per a day, then by 365 days per a year. This gave us a result based on a 25MHz core. For a Viretx-5 LX110, we divided the required time by 80 (by

		1core	20 cores	18 cores
		25Mhz	100Mhz	100Mhz
N	sec	years	years	years
19	18422			
20	87689			
21	4.17E+05			
22	1.99E+06			
23	9.46E+06			
24	4.50E+07			
25	2.14E+08	7	0.08	
26	1.02E+09	32	0.40	
27	4.86E+09	154	1.92	
28	2.31E+10	733	9.16	
29	1.10E+11	3488		48.45
30	5.24E+11	16604		230.61
31	2.49E+12	79034		1097.69
32	1.19E+13	376202		5225.02

TABLE IX
EXTRAPOLATION OF ENUMERATION TIME TO 27 AND 32

4 for speed and by 20 for number of cores). We made similar adjustments for the other column, which is more representative of the number of cores per a chip at 32.

1) *Cost comparison for 27 and 32:* At the end of the day, chip-years is useful to know but the real measure of performance is the cost of completing the enumeration. Given a budget, how quickly could a given size be enumerated? Given a time limit, how much would it cost to acquire the necessary equipment? So we decide on a ball-park figure for both PCs and FPGAs, per a unit and translate this with the chip-years (or PC-years) to euro-years for each size.

We decided that 1,000 euro was a fair unit cost for the PCs used to enumerate 26×26 , since they are not state of the art now but we must compare to the computers that were used 2 years ago to compute 26.

An LX-110 costs about 1,000 euro from avnet.com. And a board to mount it onto would cost about another 1,000 euro. This leads us to a unit cost of 2,000 euro.

According to [3], it took 29.52 years for one processor to complete the search of 26×26 . We have found a rate of growth in software searches of something slightly worse than 5 times for each time we increase n by 1. We use a figure of 5.3 times increase for each n .

	Chip	Chip	PC	PC
N	Chip Years	Euro Years	PC-years	Euro years
27	1.92	3,849	156	156,456
32	5225.02	10,450,042	654,292	654,291,941

TABLE X
COST COMPARISON FOR 27 AND 32

V. CONCLUSIONS

A. Effect of leaving rows off at the end

Leaving rows off at the end of the necessary triangle saves space but costs time. The time lost is not very great when leaving between a third and a half of the rows off (given the bottom rows do almost nothing). However, since there is a

fairly consistent saving for each row removed, there should clearly be a balance point somewhere, where these two effects balance out. From looking at the blocked paths experiment in IV-B, it is clear that checking rows 11 through 15 (statistically speaking) is a complete waste of time and that rows 8 through 11 are not very useful either.

B. Symmetry

Although a complete leverage of symmetry (like frames) has not been applied to a hardware approach yet, we were still able to gain very significant savings from using the upside-down flip coupled with a small saving from the other symmetries.

C. Hardware is better than software for this problem

The table in IV-E.1 clearly shows that there is a very significant performance gain offered by using hardware above software in non-trivial examples of this problem. We see an approximate 40 times increase in performance for $n = 27$.

REFERENCES

- [1] J. Costas, "A study of detection waveforms having nearly ideal range-doppler ambiguity properties," in *Proceedings of the IEEE*, vol. 72, no. 8, Aug 1984, pp. 996–1009.
- [2] S. Golomb and H. Taylor, "Constructions and properties of Costas arrays," *Proceedings of the IEEE*, vol. 72, no. 9, pp. 1143–1163, Sep 1984.
- [3] S. Rickard, E. Connell, F. Duignan, and A. Wade, "The enumeration of costas arrays of size 26," in *Proceedings of the 40th Annual Conference on Information Sciences and Systems*, Princeton, NJ, USA, March 2006.
- [4] J. Silverman, V. Vickers, and J. Mooney, "On the number of Costas arrays as a function of array size," in *Proceedings of the IEEE*, vol. 76, no. 7, Jul 1988, pp. 851–853.
- [5] W. Chang, "A remark on the definition of Costas arrays," in *Proceedings of the IEEE*, vol. 75, no. 4, Apr 1987, pp. 522–523.