Architecture of an FPGA-Based Heterogeneous System for Code-Search Problems

Yuki Hiradate, Hasitha Muthumala Waidyasooriya^(✉), Masanori Hariyama, and Masaaki Harada

Graduate School of Information Sciences, Tohoku University, 6-3-09,
Aramaki-Aza-Aoba, Aoba, Sendai, Miyagi 980-8579, Japan
hiradate.yuki.p5@dc.tohoku.ac.jp, {hasitha,hariyama}@tohoku.ac.jp,
mharada@m.tohoku.ac.jp

Abstract. Code search problems refer to searching a particular bit pattern that satisfies given constraints. Obtaining such codes is very important in fields such as data encoding, error correcting, cryptography, etc. Unfortunately, the search time increases exponentially with the number of bits in the code, and typically requires many months of computation to find large codes. On the other hand, the search method mostly consists of 1-bit computations, so that reconfigurable hardware such as FPGAs (field programmable gate arrays) can be used to successfully obtain a massive degree of parallelism. In this paper, we propose a heterogeneous system with a CPU and an FPGA to speed-up code search problems. According to the evaluation, we obtain over 86 times speed-up compared to typical CPU-based implementation for extremal doubly even self-dual code search problem of length 128.

1 Introduction

Fields such as cryptography, data encoding, error correcting, etc. often requires bit patterns that satisfy particular conditions. However, finding such bit patterns is a very time consuming problem. For example, in order to find 64-bit code that satisfies particular conditions, we have to search 2^{64} bit patterns. Even if we can search one bit pattern in one clock cycle using a 4 GHz CPU, it requires over 146 years to search all combinations. For a 128-bit code search problem, the required processing time exceeds the age of the universe. Therefore, how we can solve such code search problems. Mathematicians propose many algorithms to generate a particular code that satisfies the conditions, instead of search for all possible bit patterns. For example, to find all 64-bit numbers that are divisible by four, we can fix the least significant two bits to zero and generate all combinations of bit patterns of the other 62 bits. This increases the processing speed by four times. For more complex problems, many different methods are available to reduce the amount of searches. Most of those methods use bit operations or fixed-point computations. On the other hand, CPUs and GPUs are specialized for floating-point computations, and using those for such simple bit operations is not an efficient method.

In this paper, we use re-configurable hardware call FPGA (field programmable gate array) [1] to efficiently compute bit operations. FPGAs contain over millions of multi-input logic gates and registers [2]. Since all these logic gates and their interconnections are configurable, we can design custom processing elements and datapaths to efficiently execute the required operation with a massive degree of parallelism. Recently, OpenCL-based FPGA design [3] has been introduced to design accelerators using C-like high-level programming. This design method allows us to exploit the full potential of an FPGA while reducing the design time [4]. OpenCL is not only can be used to design FPGA accelerator, but also can be used to design a whole heterogeneous system including the computations of a CPU and also the data transfers between a CPU and an FPGA.

In this paper, we propose an FPGA-based heterogeneous accelerator to speed-up the “extremal doubly even self-dual code” search problem [5–7]. To solve this problem, the work in [8] proposes a method that contains many bit-operations that can be done in parallel. The proposed FPGA accelerator contains thousands of processing elements to perform bit-operations in parallel while the CPU computes complex but sequential operations in a higher clock frequency compared to the FPGA. The FPGA accelerator design and the heterogeneous system implementation are done using OpenCL. According to the evaluation, we obtain over 86 times speed-up compared to a typical CPU-based implementation for extremal doubly even self-dual code search problem of length 128.

2 Code-Search Problems

In this paper, we consider the acceleration of extremal doubly even self-dual code search [8], as an example to show the efficiency of the FPGA-based heterogeneous system for such problems. Self-dual codes are an important class of linear codes with both theoretical importance and practical applications [7]. It is important in the fields such as cryptography, error correcting, etc. In this section, we briefly explain the extremal doubly even self-dual code search algorithm. Note that, we restrict the details of the mathematical background since it is not in the scope of this paper. Readers can refer [8] for the details. We focus on the types of computations required in such code search problems, and how to accelerate those computations using FPGA-based heterogeneous system.

2.1 Extremal Doubly Even Self-dual Code Search

In the work in [8], the extremal doubly even self-dual code is described as follows. “A binary self-dual code C of length n is a code over $\mathbb{F}_2$ satisfying $C = C^\perp$ where the dual code $C^\perp$ of C is defined as $C^\perp = \{x \in \mathbb{F}_2^n | x \cdot y = 0 \text{ for all } y \in C\}$ under the standard inner product $x \cdot y$. A self-dual code C is *doubly even* if all codewords of C have hamming weight divisible by four, and *singly even* if there is at least one codeword of hamming weight $\equiv 2 \pmod{4}$. Note that a doubly even self-dual code of length n exists if and only if n is divisible by eight. It was

shown in [9] that the minimum hamming weight d of a doubly even self-dual code of length n is bounded by $d \leq 4\lceil n/24 \rceil + 4$. A doubly even self-dual code meeting this upper bound is called *extremal*.”

For example, an extremal doubly even self-dual code C of length 128 satisfies the following three conditions.

1. Hamming weight $\equiv 0 \pmod{4}$
2. $C = C^\perp$
3. $d(C) = 24$

To find such a code, work in [8] proposes the following algorithm that contains four steps.

Step 1: $x \in \mathbb{F}_2^{64}$ and $wt(x) \equiv 3 \pmod{4}$

Step 2: If $AA^T + BB^T \neq I_{32}$ go to step 1. A and B are circulant matrices given by Eq. (1).

$$A = \begin{pmatrix} x_1 & x_2 & \cdots & x_{32} \\ x_{32} & x_1 & \cdots & x_{31} \\ \vdots & \vdots & & \vdots \\ x_2 & x_3 & \cdots & x_1 \end{pmatrix}, \quad B = \begin{pmatrix} x_{33} & x_{34} & \cdots & x_{64} \\ x_{64} & x_{33} & \cdots & x_{63} \\ \vdots & \vdots & & \vdots \\ x_{34} & x_{35} & \cdots & x_{33} \end{pmatrix} \quad (1)$$

Step 3: The matrices G and H in Eq. (2) are the generator matrices of C and $C^\perp$ respectively. If the hamming weight of the sum until the 10^{th} row of G is less than 20, go to step 1. If the hamming weight of the sum until the 10^{th} row of H is less than 20, go to step 1.

$$M = \begin{pmatrix} A & B \\ B^T & A^T \end{pmatrix}, \quad G = (I_{64}, M), \quad H = (M^T, I_{64}) \quad (2)$$

Step 4: A code is found and exit.

In order to satisfy the step 3 of the code search algorithm, the hamming weight of $\{x_1 \cdots x_{64}\}$ must be equal or larger than 19. That is, at least 19 bits of the 64 bits in the code must be ones. Therefore, we have to search k -out-of-64 codes where $19 \leq k \leq 64$. Searching for such a code is a very time consuming problem.

3 FPGA-Based Heterogeneous Architecture

3.1 Exploiting the Parallelism

Since FPGA is a reconfigurable device, we can implement both space parallelism and time parallelism. Space parallelism is similar to SIMD (single instruction multiple data) operations in GPUs where the same operation is performed on multiple data simultaneously. Time parallelism is implemented using pipelines, where multiple operations are performed on different data. In order to design

Fig. 1. The amount of data transferred among each step of the code search method.

an efficient architecture, we have to exploit the parallelism of the code search problem. Figure 1 shows the amount of data transferred among each step of the code search method. As explained in Sect. 2, if the conditions are not met in each step, further computations are terminated and go back the step 1. Therefore, the amount of data transferred to the latter steps become smaller and smaller in each step. To utilize this efficiently, we use a large amount of parallel computations in the initial steps but a small amount of parallel computations in the latter steps. In addition, multiple steps are computed in parallel for different data using a pipelined architecture.

3.2 Overall Architecture of the FPGA-Based Heterogeneous System

Figure 2 shows the overall architecture of the FPGA-based heterogeneous system. It consists of a CPU and an FPGA accelerator. The CPU and the FPGA works together to generate k -out-of- n codes that satisfies the step 1 of the algorithm explained in Sect. 2.1. For each k -out-of- n code, matrix calculations in step 2 is performed in parallel. After the matrix computation is done, hamming weight is calculated as explained in step 3.

Amount of parallel computations are decreases from step 1 to step 3. As shown in Fig. 2, we have used a CPU and 64 bit-shift modules in step 1. There are 64 matrix calculation modules in step 2. However, there are only 10 modules in step 3. Since the amount of data transferred to each stage is getting smaller and smaller, we decrease the amount of parallel computations. This way, we can efficiently use the FPGA resources by spending more resources on bottleneck stages that process a large amount of data.

Fig. 2. Overall architecture.

3.3 k -out-of- n Code Generation

There are a few methods such as [10, 11] for k -out-of- n code generation. However, these methods have a data dependency among code searches. That is, the search of a new bit pattern must be started only after the search of the previous bit pattern is finished. As a result, it is extremely difficult to accelerate such methods using parallel processing. Therefore, we use the “circular permutation generation algorithm” proposed in [12] to accelerate k -out-of- n code generation. A p -ary circular permutation of length n is an n -character string of an alphabet of size p , where all rotations of the string are considered as equivalent. Therefore, we can regard a circular permutation code as a seed and generate the other bit patterns by rotating the bits. Figure 3 shows two seeds and the generated bit patterns of 2-out-of-4 codes. The rotation of bits can be done in parallel using bit-shift operations. Therefore, even if we generate the seeds in serial, we can still have a large amount of parallel operations.

The algorithm to generate circular permutation [12] is a serial one. Therefore, the only way to increase the processing speed of the circular permutation generation is to increase the clock frequency. Unfortunately, the clock frequency of an FPGA is usually less than 300 MHz. Therefore, we use a CPU for the permutation generation which has more than 10 times larger clock frequency compared to that of an FPGA. Once the permutations are generated, those are transferred to the external memory (DRAM) of the FPGA board. The FPGA accelerator access those data and performs 63 shift operations in parallel and select the codes that satisfies the step 1 of the algorithm explained in Sect. 2.1. The permutation generation and bit-shift can be done in parallel as shown in Fig. 4. The time required for a circular permutation generation and parallel bit-shift operations are nearly equal. This way, the CPU and the FPGA are used in parallel manner.

Fig. 3. 2-out-of-4 bit pattern generation using seeds.

Fig. 4. Parallel processing of k -out-of- n code generation using a CPU and FPGA.

3.4 Matrix Calculation and Hamming Weight

In step 2, we do the matrix calculation of $AA^T + BB^T \neq I_{32}$ is a simple bit operation. Since there are 64 codes generated in parallel in step 1, we use 64 modules in parallel in step 2 for matrix calculation. Only a small percentage of codes satisfy this condition, so that the amount of data proceeds to the next step is small. As a result, we use one “hamming weight computation” modules in step 3 for every 8 matrix calculation results. As a result, the number of modules in

```

//The following code is executed in parallel
a = shift_onebit( A ); b = shift_onebit( B );
out.BIT.b30 = popcount( A.w & a.w ^ B.w & b.w ) & 1;
a = shift_onebit( a ); b2 = shift_onebit( b );
out.BIT.b29 = popcount( A.w & a.w ^ B.w & b.w ) & 1;
a = shift_onebit( a ); b2 = shift_onebit( b );
out.BIT.b28 = popcount( A.w & a.w ^ B.w & b.w ) & 1;
...
/** shift_onebit is a function that performs cyclic shift of 32 bits **/

```

Fig. 5. Program code for step 2.

```

for (int r1 = 0; r1 < N; r1++) {
    for (int r2 = r1 + 1; r2 < N; r2++) {
        unsigned long g1, h1;
        g1 = G[r1] ^ G[r2];
        h1 = H[r1] ^ H[r2];
        weight_g = popcount( g1 );
        weight_h = popcount( h1 );
        if (weight_g < wt - 2 | weight_h < wt - 2) {
            return 0;
        }
        for (int r3 = r2 + 1; r3 < N; r3++) {
            unsigned long t2, s2;
            g2 = t1 ^ G[r3];
            h2 = s1 ^ H[r3];
            weight_g = popcount( g2 );
            weight_h = popcount( h2 );
            if (weight_g < wt - 3 | weight_h < wt - 3) {
                return 0;
            }
            for (int r4 = r3 + 1; r4 < N; r4++) {
                ...
                /** calculate upto r10. **/
            }
        }
    }
}

```

Fig. 6. Program code for step 3.

step 2 is reduced without affecting the total processing time. A part of matrix calculation program code is shown in Fig. 5.

In step 3, hamming weight until the 10^{th} row is calculated. However, most codes can be rejected by computing the hamming weight until the first few rows. Therefore, we divide the step 3 into two stages. In the first stage, the hamming weight until the first 5 rows are computed. The codes satisfy this condition go to the second stage. We use only 2 modules in the second stage since a smaller degree of parallelism is required. A part of step 3 program code is shown in Fig. 6.

4 Evaluation

We used two systems for the evaluation, where one contains only one CPU and the other contains one CPU and one FPGA. In the CPU only system, the CPU is Intel Xeon E5-1650 v3 (3.50 GHz). In the heterogeneous system, the CPU is Intel Xeon E5-2643, and the FPGA is Terasic DE5a-net FPGA board [13] with Intel Arria 10 FPGA. FPGA is configured using Quartus prime pro 16.1 and Intel FPGA SDK for OpenCL [14]. CPU codes are compiled using Intel compiler 17.0 with OpenMP directives for parallel computation.

Table 1 shows the comparison of the processing time of k -out-of- n code generation using different methods. In this evaluation, n is 64 and k is 8. The fastest CPU implementation is a nested-loop implementation that search all bit patterns to find the desired code. Some part of the loop can be processed in parallel so that the processing time is reduced. Compared to that, proposed heterogeneous implementation produced over 2.4 times speed-up compared to the nested-loop implementation.

Table 2 shows the comparison of the total processing time of extremal doubly even self-dual code search. Note that the clock frequency of the FPGA is reduced to 207 MHz from 309 MHz in Table 1 due to increased computation. Even with such low clock frequency, the speed-up of the proposed implementation is 86.9 times compared to the CPU-only implementation. This shows that FPGAs are very efficient for bit operations. Moreover, nearly 64 codes can be checked per clock cycle in FPGA due to its massively parallel computations.

Table 3 shows the resource usage of the FPGA. Since only 37% of the logic resources are used, there is a potential to increase the processing speed further by doing more parallel computations. If we increase the degree of parallelism, the bottleneck would be the circular permutation generation in the CPU. Therefore,

Table 1. Comparison of the processing time of k -out-of- n code generation

	[10]	[11]	Nested loops	Proposed
Device	CPU	CPU	CPU	CPU & FPGA
Clock frequency (MHz)	3500	3500	3500	CPU:3300, FPGA:309
Processing time (s)	35	9.04	0.6	0.25

Table 2. Results

	Conventional	Proposed
Device	CPU only	CPU & FPGA
Clock frequency (MHz)	3500	207
Processing time (s)	29.13	0.33
Number of clock cycles (10^9)	10.21	0.07
Codes checked per clock cycle	0.04	63.5

Table 3. FPGA resource utilization

Resource type	Utilization (Percentage %)
Logic	143,186 (34)
Memory bits	3,899,280 (7)
RAM blocks	425 (16)

decreasing the processing time of circular permutation generation is critical for future improvements.

The evaluation is done for only k -out-of-64 codes where k equals 8. We also found similar results for 19-out-of-64 codes. Neither of those search results have given a solution for extremal doubly even self-dual code of length 128. Therefore, it is still an unsolved problem. In order to find a solution for this problem, we have to search other k values and that is one of our future works. Also note that, the data transferred to the FPGA for the CPU are only the circular permutation data generated in CPU. Therefore, the DRAM access by the FPGA is minimal. Although the FPGA board has a small memory bandwidth of 25.6 GB/s bandwidth, it is not a bottleneck for the code search problems.

5 Conclusion

In this paper, we propose an FPGA-based heterogeneous system for extremal doubly even self-dual code search. Although we are yet to solve the problem, there is a great potential to find a solution in near future due to over 86 times of speed-up of the proposed system compared to a conventional one with only a CPU. Moreover, we used only 34% of the FPGA resources, so that further increase of speed is possible. It is very important to exploit the possibility of accelerating other code search problems using FPGAs in future.

References

1. Marchal, P.: Field-programmable gate arrays. *Commun. ACM* **42**(4), 57–59 (1999)
2. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/arria-10/arria_10_aib.pdf
3. Czajkowski, T.S., Neto, D., Kinsner, M., Aydonat, U., Wong, J., Denisenko, D., Yiannacouras, P., Freeman, J., Singh, D.P., Brown, S.D.: OpenCL for FPGAs: prototyping a compiler. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, p. 1 (2012)
4. Waidyasooriya, H.M., Hariyama, M., Uchiyama, K.: *Design of FPGA-Based Computing Systems with OpenCL* (2017)
5. MacWilliams, F.J., Sloane, N.J.A.: *The Theory of Error-Correcting Codes*. North-Holland, Amsterdam (1977)
6. Pasquier, G.: A binary extremal doubly even self-dual code (64, 32, 12) obtained from an extended Reed-Solomon code over F16. *IEEE Trans. Inform. Theory* **27**, 807–808 (1981)

7. Rains, E., Sloane, N.J.A.: Self-dual codes. In: Pless, V.S., Huffman, W.C. (eds.) *Handbook of Coding Theory*, pp. 177–294. Elsevier, Amsterdam (1998)
8. Harada, M.: An extremal doubly even self-dual code of length 112. *Electron. J. Comb.* **15**, 1–5 (2008)
9. Mallows, C.L., Sloane, N.J.A.: An upper bound for self-dual codes. *Inform. Control* **22**, 188–200 (1973)
10. Harbison, S.P., Steele Jr., G.L.: *C: A Reference Manual*. Prentice Hall, Englewood Cliffs (1987)
11. <https://docs.python.org/2/library/itertools.html>
12. Sawada, Joe: A fast algorithm to generate necklaces with fixed content. *Theoret. Comput. Sci.* **301**, 477–489 (2003)
13. Terasic, DE5-Net FPGA Development Kit. <http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=158&No=526>
14. Intel FPGA SDK for OpenCL, Programming Guide. https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.