Computing Goldbach partitions using pseudo-random bit generator operators on a FPGA systolic array

Dominique LAVENIER[1], Yannick SAOUTER[2]

[1] IRISA, Campus de Beaulieu, 35042 Rennes, France
lavenier@irisa.fr
[2] IRIT, 118 route de Narbonne, 31062 Toulouse, France
saouter@irit.fr

Abstract. Calculating the binary Goldbach partitions for the first 128×10^6 numbers represents weeks of computation with the fastest microprocessors. This paper describes an FPGA systolic implementation for reducing the execution time. High clock frequency is achieved using operators based on pseudo-random bit generator. Experiments carried both on the R10000 processor and on the FPGA PeRLe-1 board are reported.

1 Introduction

In 1742 Goldbach claimed the following conjecture: "Every even integer greater than 3 is the sum of two prime numbers". This conjecture is known as the *binary Goldbach conjecture*. Today, it has been verified till the bound 4×10^{14} and partially near various powers of ten up to 10^{300} [3]. But the *exact number* of binary partitions has only been investigated up to 350 000 [1], because of high computation cost.

Such computations are useful for estimating the reliability of probabilistic models and testing their theoretical background. Historically, three models for the number of partitions have been proposed by Hardy and Littlewood, Brun, and Selmer. These three models are compared in [5] together with the values obtained from our FPGA implementation.

The solution we propose for speeding up the computation combines two technics: parallelization and customization.

- the **parallelization** allows to compute simultaneously N consecutive binary Goldbach partitions. It is implemented on a linear systolic array. Such regular arrays map well onto FPGAs: they are both made of locally interconnected regular elements. Furthermore, synchronous designs can reach very high computation speed.
- the **customization** maps into hardware a function which requires the use of several instructions on a sequential machine. Particularities of the function can be extracted to make the best use of hardware resources. In our case, a fast implementation has been achieved with arithmetic operators built with pseudo-random number bit generators and integer modular representation.

The next section introduces the binary Goldbach partition enumeration problem and section 3 shows the parallelization on a linear systolic array. Section 4 details the FPGA implementation and focuses on the pseudo-random bit generator operators. Section 5 compares the experimentations we have done with the R10000 processor and the PeRLe-1 board. Section 6 concludes with some perspectives.

2 Computing binary Goldbach partitions

Computing **one** binary Goldbach partition consists of counting, for an even number K ($K \geq 4$), all the possible ordered pairs such that $K = p_1 + p_2$ where p_1 and p_2 are two prime numbers. For instance, the binary Goldbach partition of 22 (denoted by $G_2(22)$) is 5 since there are five possibilities of getting 22 by summing two prime numbers (3+19, 5+17, 11+11, 17+5, 19+3).

Computing the binary Goldbach partitions of an even number K may be expressed by the following C function:

Algorithm 1:
```
int Goldbach_1(K) int K; {
  int G=0; int i=1;
  while (i<K) {
    if (prime(i) && prime(K-i)) G=G+1;
    i=i+2;
  }
  return(G);
}
```

The function `Goldbach_1(K)` returns $G_2(K)$. `prime(x)` is a function which returns true if `x` is a prime number and false otherwise. The complexity of this algorithm is equal to the numbers of times the loop is executed, that is: $K/2$ One may note the symmetry of the calculation for $i < K/2$ and $i > K/2$, leading to an optimized function:

Algorithm 2:
```
int Goldbach_2(K) int K; {
  int G=0; int i=K;
  while (i<K/2) {
    if (prime(i) && prime(K-i)) G=G+1;
    i=i+2;
  }
  G=G*2;
  if (prime(K/2)) G=G+1;
  return (G);
}
```

Since the case `i=K/2` is not considered when executing `Goldbach_1(K/2)`, it is added to the body of the `Goldbach_2` function. The complexity is then lowered to $K/4$.

Now, the problem of computing N consecutive binary partitions can be stated as the calculation of N partitions in the interval `[P .. P+2N-2]`. The following C function stores the N results in the array `G` such as `G[n]`=$G_2(P+2n)$.

```
Algorithm 3:      Goldbach_3(P,N,G) int P, N, *G; {
                    int K,n;
                    n=0;
                    for (K=P; K<P+2*N; K=K+2) {
                      G[n]=GoldBach_2(K);
                      n=n+1;
                    }
                  }
```

For P$\gg$N (the general case) the complexity of algorithm 3, denoted by $G_{seq}(P,N)$ is equal to $N \times P/4$.

3 Parallelization

The computation of N consecutive binary Goldbach partitions can be parallelized on a linear systolic array of N processors as depicted by the figure 1. Each cell is responsible for the calculation of one binary Goldbach partition: The leftmost cell computes $G_2(P+2N-2)$ and the rightmost cell $G_2(P)$.

The array is supplied with two boolean vectors: `V1` and `V2`. The vector `V2` crosses the array at twice the speed of `V1`. The boolean vectors are formed as follows:

```
   V1[i] = prime(2i-1)                          0  <= i <  (P+2N-2)/2
V2[i] = prime (P+2N-2i-3)       (P+2N-2)/2  <= i <  (P+2N-2)/2 + N-1
```

A cell computing $G_2(k)$ receives two boolean values: `prime(x1)` and `prime(x2)` such that $k = x_1 + x_2$. When `prime(x1)` and `prime(x2)` are both true, a counter (`G`) is incremented by 1. At the end of the computation each counter holds the value of one binary Goldbach partition.

Performing the calculation of N consecutive binary partitions over the interval `[P,P+2N-2]` requires, for $P \gg N$, approximatively $P/2$ systolic steps (denoted by $G_{sys}(P,N)$). The speed-up compared with the sequential implementation, is thus given by:

$$S = \frac{G_{seq}(P,N)}{G_{sys}(P,N)} \times \frac{\delta_{seq}}{\delta_{sys}} = \frac{N}{2} \times \frac{\delta_{seq}}{\delta_{sys}} \qquad (1)$$

δ_{seq} is the time for executing one iteration of the loop of algorithm 1. δ_{sys} is the time of one systolic cycle. From the above equation it can easily be seen that an efficient systolic implementation will aim to both increase N and to reduce δ_{seq}. These are the two sources of increased performance.

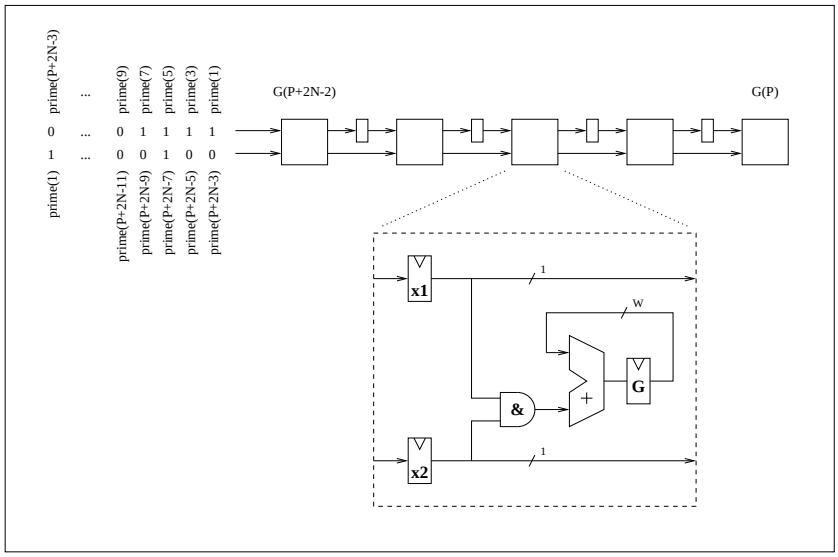

Fig. 1. Linear systolic array for computing N consecutive binary Goldbach partitions. It is composed of N cells, each cell computing one Goldbach partition. The array is supplied with two boolean vectors. The calculation perform by a cell consists simply in incrementing a counter if both inputs of the cell are true.

4 FPGA implementation

This section explains how the architecture of the systolic array has been tailored for the FPGA XC3000 Xilinx family. The three next sub-sections describe respectively the time optimization (how to get a high clock frequency), the space optimization (how to map a cell in a minimum of hardware), and the unloading of the results.

4.1 Time optimization

The clock frequency of a cell – and hence, the clock frequency of the array – is mainly determined by the speed of the counter. This depends of the number K for which $G_2(K)$ is computed. For example, finding the partition for numbers around 10^8 requires a 27-bit counter. Here, a ripple carry adder is too slow for achieving good speed performance, while a more sophisticated architecture, such as a carry save adder, requires too much space to reach a reasonable size array.

Instead of implementing a conventional counter, we use a mechanism which uses two main techniques: Pseudo-random bit generators and modular representation. The advantage is that no carry propagation occurs, allowing high frequency clock. On the other hand, the numbers are not in their natural representation and need to be post-processed before interpretation.

We use pseudo-random bit generator based on the following algorithm [2]:

```
Algorithm 4:    counter = 0;
                while (true) {
                 x = MSB(counter);                  /* MSB */
                 counter = counter << 1;            /* shift left */
                 if (x==0) counter = counter ^ key;
                }
```

With a suitable *key* value, the successive values of the data can have a period long enough to be used as a counter. But interpreting the value of a W-bit counter based on this technique is not obvious: A lookup table of 2^W entries is required, which is unreasonable for the values of W we are considering (W$\geq$25). On the other hand, retrieving natural representation by post-processing (in software) soft is too slow.

The difficulty can be surmounted by using two counters and using modular representation of integers: let p and q two numbers relatively prime to each other such that $M < p.q$. Then any number k in the interval $[1..M]$ is uniquely represented by the pair $(k \ mod \ p, k \ mod \ q)$.

The conversion from the pair representation to the plain integer representation is based on the following theorem:

Theorem 1:

Let p and q two integers relatively prime to each other. Let u and v be such that $u.p - v.q = 1$. A solution to the set of equation:

$$\begin{cases} k = n_1 \ mod(p) \\ k = n_2 \ mod(q) \end{cases}$$

is given by:

$$k = n_1 + (n_2 - n_1).u \ mod(p.q)$$

In other words, given the pair (n_1, n_2), the post-processing task for computing k consists simply of a modular multiplication which actually represents a small part of the overall computation. More details can be found in [5].

As an example, the hardware implementation of a 9-bit counter based on two pseudo-random bit generators and modular integer representation is shown on the figure 2. It is composed of two separate counters: A 4-bit counter and a 5 bit counter. According to Algorithm 4, the F_i functions implement the pseudo-random bit generator as follows:

$$F_i = inc.(\overline{x}.(k_i \oplus c_{i-1}) + x.c_{i-1}) + \overline{inc}.c_i \qquad (2)$$

inc is true if the two inputs of the cell are true. x is the most significant bit of the counter. c_i and c_{i-1} are the current state of respectively the ith and the i-1th bits of the counter. $+, .$ and $\oplus$ stand respectively for logic OR, logic AND and logic exclusive OR boolean functions.

In such a counter, there is no carry propagation. The clock frequency is dictated by the computation time of one elementary 5-input F_i function.

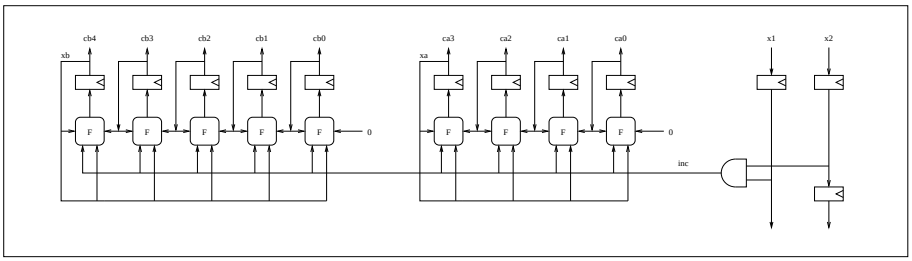

Fig. 2. Systolic cell: two pseudo-random bit generators of respectively 5 and 4 bit wide are used to form a 9-bit counter. The two generators are concurrently activated when input x1 and x2 are both true.

4.2 Space optimization

The goal is to fit a maximum of cells in a FPGA XC3000 Xilinx family component. Such components contain a matrix of CLBs, each CLB containing two 1-bit registers and either two 4-input boolean functions or one and 5-input boolean function.

If W is the width of the counter, a direct implementation of a cell requires W $\times$ 5-input functions (counter), one 2-input function (AND gate) and 3 $\times$ 1-bit registers, that is (W+2) CLBs (the 3 registers and the AND function fit into 2 CLBs). Note that this is an optimistic estimation since it does not consider the hardware for reading back the results.

One may notice that the counter contains two keys which remain stable during the whole computation. It is then possible to encode these keys and provide simpler F_i functions depending of the bit state k_i.

From equation 2, if k_i is equal to zero then:

$$F0_i = inc.c_{i-1} + \overline{inc}.c_i$$

On the same way, if k_i is equal to one, F_i becomes:

$$F1_i = inc.(\overline{x}.\overline{c_{i-1}} + x.c_{i-1}) + \overline{inc}.c_i$$

In this scheme, a counter is composed of two types of elementary functions: $F0_i$ or $F1_i$, according to the key. From a practical point of view, one CLB can now contain two $F0_i$ or $F1_i$ functions (which are respectively 3-input and 4 input functions), lowering a cell to (W/2 + 2) CLBs.

4.3 Unloading the results

The mechanism implemented for collecting the results (the values held by the counter) exploits the shift capability of the counter. Each $F0_i$ and $F1_i$ function is extended with a *read-back* input which acts as a switch:

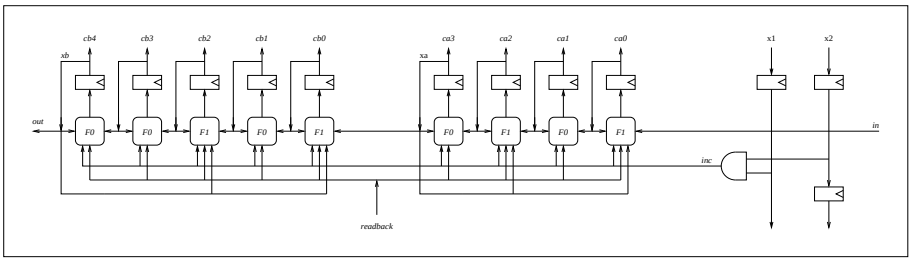

Fig. 3. Read-back mechanism: the shift capability of the counter is used to form a long shift register for outputing data serially.

$$F0_i = readback.c_{i-1} + \overline{readback}.(inc.c_{i-1} + \overline{inc}.c_i)$$

$$F1_i = readback.c_{i-1} + \overline{readback}.(inc.(\overline{x}.\overline{c_{i-1}} + x.c_{i-1}) + \overline{inc}.c_i)$$

Thus, reading back the results consists of connecting all the counters in a long shift register and outputing data serially. This is achieved simply as shown by the figure 3. This mechanism has the main advantage of requiring a very little extra hardware without affecting the clock frequency.

Note that the $F1_0$ function must be different: In the normal mode the shift register input is *zero*, and in the read-back mode its input is the most significant bit of the previous counter. Then the $F1_0$ function is:

$$F1_0 = readback.in + \overline{readback}.(inc.\overline{x} + \overline{inc}.c_i)$$

The $F0_i$ function is now a 4-input function, while the $F1_i$ function is a 5-input function. Consequently, the number of CLBs to fit a complete cell depends on the key since a CLB cannot simultaneously house two 5-input functions. Fortunately, the keys contain a small number of `"1's"`, allowing hardware resources to be greatly minimized.

5 Experiments

A 256 cell linear systolic array has been successfully implemented on the PeRLe-1 board [4]. This board houses a 4×4 matrix of Xilinx XC3090 and is connected through a Turbo Channel interface to a 5000/240 Dec Station.

The counter, which is the critical point in terms of hardware resources, is 27-bit wide. It is split into two counters of 13-bit and 14-bit wide which require respectively a key set to 9 and 7. This configuration allows us to map 16 cells into a single Xilinx XC3090 component. The entire systolic array is then composed of 256 cells running at 30 MHz.

In addition to the systolic array, an interface composed of a small automaton, coupled with a serializer/deserializer mechanism has been implemented for managing the I/O connection with the Dec station. The host alternatively sends two sub-vectors of 16 boolean values corresponding to the two data inputs of the array. This compression mechanism avoids exceeding the Turbo Channel bandwidth.

The computation of the binary Goldbach partitions up to 128×10^6 was performed in 220 hours (9.2 days). The 64 first million binary Goldbach partitions are now available for comparing the three basic probabilistic methods. This mathematical study is beyond the scope of this paper and is not addressed here. Readers Interested by the estimation done on the reliability of the three probabilistic models mentioned in the introduction (Hardy and Littlewood, Brun, and Selmer) can refer to [5].

In order to evaluate the speed-up of the hardware approach, two programmable versions have been tested on a R10000 processor. We used one node of the *Power Challenge Array* of the Charles Hermite Center, Nancy, France. This supercomputer is composed of 40 R10000 processors scheduled at 195 MHz and sharing a common memory of 1.5 Gbytes. Experiments were performed for low values and extrapolated to 128×10^6 with respect to the complexity of the algorithms.

The first algorithm implemented is the naive one, that is, the algorithm presented in section 2 (complexity: $O(N^2)$, where N is the upper value for which the Goldbach partitions are computed). The second one has a lower complexity: For any pair of prime numbers below N, it computes the sum and adds one in the corresponding entry of an array containing the $N/2$ partitions. The prime number theorem states that the number of prime numbers below N is approximatively equal to $N/log(N)$. Consequently the complexity of the second algorithm is raised to $O((N/log(N))^2)$. Note that this algorithm requires an integer table of $N/2$ entries. The following table summarizes the execution time for the naive algorithm, the optimal algorithm and the systolic algorithm.

N	**naive algorithm** R10000	**optimal algorithm** R10000	**systolic algorithm** PeRLe-1
10^6	1:58:59	11:25	2:00
2×10^6	7:50:47	1:11:26	5:30
3×10^6		2:53:01	10:35
4×10^6		4:37:22	17:10
5×10^6		5:42:05	25:30
128×10^6	32500 hours *(3.7 years)*	2928 hours *(4 months)*	220 hours *(9.2 days)*

The last row – for the naive and optimal algorithms only – is an estimation of the execution time calculated as follows:

naive algorithm: $t_{naive} = 7.2 \times 10^{-9} \times N^2$

optimal algorithm: $t_{optimal} = 2.3 \times 10^{-7} \times (N/log(N))^2$

The two constants have been determined from the first measures. The systolic column reports the exact time.

One may have noticed that the comparison between the hardware and software doesn't rely on equivalent technology. The PeRLe-1 board is far from using up-to-date FPGA components compared with the R10000 microprocessor. The PeRLe-1 matrix (16 x XC3090 chips made of 16x20 CLB matrix) contains 10240 4-input boolean functions. This is approximatively the capacity of a single Xilinx XC40125XV component (10982 4-input look-up table)[6]. In other words, an up-to-date board will be able to house a much larger systolic array (4096 cells) which will certainly be clocked with a higher frequency. In that case, the execution time would be reduced to a few hours.

6 Conclusion and Perspectives

As in many other specific applications, the FPGA based co-processor approach has demonstrated its efficiency for enumerating the binary Goldbach partitions. In the present case, the execution time has been reduced from months to days.

Of course, such computation could have been performed on a parallel machine, or on a network of workstations. This is technically feasible. The drawback is just to find such computing resources, i.e. a parallel machine available for several tens of days, exclusively for that specific computation. Designing an ASIC is another solution which is no longer valid: once results have been obtained, the chip become useless. The FPGA technology appears as the right alternative for a domain of research which requires both intensive computation and one-time-use architecture.

Other than the architecture we proposed for enumerating Goldbach partitions, the systolic scheme can be applied to many other similar problems such as, for example, the test of the reliability of Schinzel's conjecture about pairs of prime numbers, or to verify the Hooley's conjecture on the sums of exact powers. Any of these applications would lead to very similar architectures, but operating on different bit-streams. Similarly, the specialization of operators based on pseudo-random bit generators can address other areas for optimizing the clock frequency and reducing the cost of certain circuits such as the Brent's polynomial greatest divider, the Huffman's systolic encoder, etc.

Short term perspectives of this work are concerned with automatic tools for implementing regular arrays onto FPGA. A a matter of fact, many time-consuming applications can be parallelized on a regular array. Generally, the critical section of code is a nested loop from which a parallel hardware co-processor can be synthesized. Being able to map onto a FPGA board an HDL description of a regular architecture, together with the host/co-processor communication interface will constitute a real gain of time and effort. We are currently developing such tools.

References

1. C.E. Bohman, J. Froberg. Numerical results on the Goldbach conjecture. BIT **15** (1975)
2. D. Knuth. The Art of Computer Programming: semi-numerical algorithms, Addison-Wesley (1969).
3. J-M. Deshouillers, Y. Saouter, H.J.J. te Riele. New experimental results concerning the Goldbach conjecture. Proc. of the Annual Number Theory Symposium, Portland, Oregon, USA, 1998.
4. J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age, IEEE Transactions on VLSI Systems, Vol. 4 No. 1 (1996) 56-69
5. D. Lavenier, Y. Saouter. A Systolic Array for Computing Binary Goldbach Partitions. IRISA report 1174 (1998)
6. Xilinx Products, XC4000XV family. `http//www.xilinx.com/products/xc4000xv.htm`