Solving Large Systems of Linear Equations over GF(2) on FPGAs

Wen Wang, Jakub Szefer
Dept. of Electrical Engineering
Yale University
New Haven, CT, USA
E-mail: {wen.wang.ww349, jakub.szefer}@yale.edu

Ruben Niederhagen
Dept. of Mathematics and Computer Science
Technische Universiteit Eindhoven
Eindhoven, The Netherlands
Email: ruben@polycephaly.org

***Abstract*—This paper presents an efficient systolic line architecture for solving large systems of linear equations using Gaussian elimination on the coefficient matrix. Our architecture can also be used for solving matrix inversion problems and for computing the systematic form of matrices. These are common and important computational problems that appear in areas such as cryptography and cryptanalysis. Our architecture solves these problems efficiently for any large-sized matrix over GF(2), regardless of matrix size, shape or density. We implemented and synthesized our design for Altera and Xilinx FPGAs to obtain evaluation data. The results show sub-μs performance for the Gaussian elimination of medium-sized matrices and performance on the order of tens to hundreds of ms for large matrices. In addition, this is one of the first works addressing large-sized matrices of up to $4{,}000 \times 8{,}000$ elements and therefore is suitable for post-quantum cryptographic schemes that require handling such large matrices.**

I. Introduction

Solving systems of linear equations (SLEs) is an important computational task in many scientific fields. Solving systems over GF(2) is of particular interest in cryptography and cryptanalysis. For example, it can be used as a subroutine for computing inverses of field elements of GF(2^m) [1], which is an essential task in elliptic curve cryptography. Furthermore, solving systems over GF(2) is a relevant step in factorization algorithms, e.g., the number field sieve.

Also in the field of post-quantum cryptography the problem of solving systems over GF(2) arises: it is an important step in key generation of code-based cryptosystems [2], [3]. Increasing the key size is one way to achieve high security levels for such systems. However, when increasing the key size, larger and larger matrices need to be processed. Current high-security proposals consider matrices with 6,960 columns to achieve 128-bit post-quantum security level [4, Sec. 2]. Our work is crucial for accelerating such applications in post-quantum cryptography.

Building systolic architectures for Gaussian elimination is a standard approach for solving SLEs in hardware. Most of the existing publications target small- (about 10×10 elements) to medium-sized (about 50×50 elements) matrices by building a large systolic architecture that matches the matrix size. Due to resource limitations on FPGAs, such designs are not suitable for large matrices (over 200×200 elements) as there are not enough FPGA resources.

In our work, we efficiently break the Gaussian elimination process into a number of *steps* and *phases* that use a systolic architecture, which is smaller than the matrix size, to perform operations on the original, large matrix. Our work is crucial for accelerating multiple applications in post-quantum cryptography. Details of our architecture are presented in Section IV.

In this work, we extend the approach proposed in [5], and improved upon [6], with these contributions:

- *Large Matrix Support* – First work to show results for matrices of up to $4{,}000 \times 8{,}000$ elements. The matrix size is limited by the available on-chip memory; our approach can be extended to use external memory as well.
- *Reduced Runtime* – Sub-μs performance for the Gaussian elimination of medium matrices and performance on the order of tens to hundreds of ms for large matrices.
- *Reduced Resource Utilization* – 1/3 reduction in logic utilization compared to related work and very efficient use of block memory elements.
- *Device Portability* – Easy realization on both Altera and Xilinx FPGAs without any modification since no Xilinx or Altera specific optimizations are involved in the design.
- *Code Availability* – The Verilog source code of our design is available as Open Source at http://caslab.eng.yale.edu/code/gausselim.

II. Gaussian Elimination and Systolic Architectures

Gaussian elimination is a basic method that can be extended and used for, among others, solving systems of linear equations, bringing a matrix into its systematic form, or performing matrix inversions.

Consider solving a system of linear equations in the form $A \cdot x = b$, where A is a square matrix and b is a vector. First, Gaussian elimination is used to transform the system into its equivalent form $U \cdot x = b'$, where U is an upper right triangular matrix. The transformation is done by a sequence of elementary row operations. Once $U \cdot x = b'$ is obtained, the system is solved by using backward substitution, i.e., elementary row operations are applied that convert the system to $I \cdot x = b''$, where I is an identity matrix and b'' is the solution to this system.

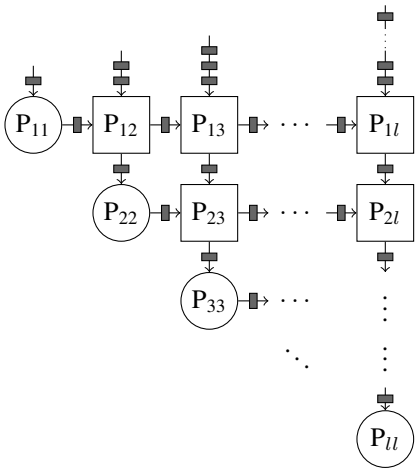


Fig. 1: Systolic array of processor elements from [5].

For matrix systemization, a rectangular matrix G (of size $l \times k$, $k > l$) is divided into the left square part G_1 (of size $l \times l$) and the right part G_2 (of size $l \times (k-l)$). By performing Gaussian elimination and backward substitution on the whole matrix, its left part is reduced to the identity matrix I while its right part is converted to a matrix P. Thus, G is brought to its systematic form $G = [I|P]$.

For matrix inversion, the invertible square matrix A is adjoined to an identity matrix I, i.e. a new matrix $[A|I]$ is created. This matrix is then systematized which reduces its left side to the identity form; the adjoined identity matrix is transformed to the inverse of the original matrix, i.e., $[A|I]$ becomes $[I|A^{-1}]$.

A. Systolic Architectures for Gaussian Elimination

Hardware architectures for Gaussian elimination over finite fields can be divided into three types: systolic array, systolic network, and systolic line.

1) Systolic Array: In 1989, Hochet, Quinton, and Robert introduced a systolic array of processors for doing Gaussian elimination on a matrix over GF(p) with partial pivoting [5]. The general structure of their architecture is shown in Figure 1. They use a processor array with an upper-right triangular shape that has special processors on the diagonal (circular processors) that pick the pivot elements, and general processors (square processors) on the remaining positions that apply transformations for the elimination. The input matrix is fed into the array through a "stairway" of shift registers; after the computation is finished, the resulting matrix is stored in internal registers of the processors. The array is systolic, i.e., all inputs/outputs of the processors are registered, and there are registers between the rows and the columns of the array, as shown in Figure 1. Thus, the critical path of this architecture is determined by the internal logic of the processors. To solve an $l \times l$ linear system, $3l$ clock cycles are needed. Another l cycles are required in order to readout the resulting matrix from the processor's registers in a systolic fashion. Thus, the resulting matrix is available after $4l$ clock cycles.

2) Systolic Network: In 1990, Wang and Lin proposed the idea of a systolic network of processors [1], which eliminates the shift registers for data input/output and the registers between rows and columns in the systolic array. In this case, signals propagate through the whole systolic network within one clock cycle. After $2l$ clock cycles, the solution of an $l \times l$ linear system is available. However, the critical path of the systolic network is determined by the size of the whole network. When l grows bigger, the achievable frequency and thus the performance of the network declines.

3) Systolic Line: In 2011, Rupp et al. discussed a systolic line of processors [7]. This approach is a trade-off between systolic arrays and systolic networks. We adopt this approach in our work. In our architecture, registers are added between different rows, while signals are allowed to propagate through one whole row in one clock cycle. No shift registers are needed neither for data input nor for data output. Compared to systolic arrays, the required time to solve an $l \times l$ linear system is reduced to $3l$. The critical path of this architecture only depends on the width of the rows, which strikes a balance between systolic arrays and systolic networks.

B. Systolic Architecture Size vs. Matrix Size

A key design and implementation detail is the size of the systolic architecture compared to the size of the matrix. Most existing designs focus on small- and medium-sized matrices, as for those sizes the systolic architecture can fully fit on the FPGA. Meanwhile, for large matrices, where the systolic array would be too large to fit on the FPGA, a different approach is needed. Details of our architecture, that addresses this problem, are presented in Section IV.

III. Existing Implementations and Their Applications

There are several publications that describe hardware implementation of Gaussian elimination. Most of the work has been focusing on small-sized or medium-sized matrices over finite fields, especially GF(2). Some publications also deal with GF(2^m) or GF(p).

Jasinski et al. [8] implemented matrix inversion using Gaussian elimination. They implemented a systolic network architecture with a space complexity of O(l^2) for square matrices of size l. Using their approach, matrices up to size 120×120 can be inverted on a Stratix IV FPGA in 1,000ns. However, the authors state that as the synthesized circuit grows larger, the maximum frequency drops heavily due to the increased length of the routing paths.

Bogdanov et al. [9] presented a parallel SMITH (Scalable Matrix Inversion on Time-Area Optimized Hardware) architecture for solving medium-sized SLEs. Their design has an average running time of $2l$ and a worst-case time complexity of O(l^2) for solving a system of l linear equations; our design requires a fixed amount of $3l$ clock cycles. They presented their implementation for SLEs of size up to 50×50. Their design is clocked with a frequency up to 300MHz; inversion of a 50×50 matrix requires 330ns on a Spartan 3 device.

In 2011, Rupp et al. [7] improved upon the work of [9] and proposed a GSMITH (Generalized SMITH) architecture to solve SLEs over GF(2) and GF(2^m). Their GSMITH architecture is capable of solving SLEs of size up to about 150×150 on a Virtex 5 FPGA. They give concrete performance numbers for sizes up to 110×110 SLEs. They solve a 110×110 SLE within 840ns. In this case, the architecture is clocked with a frequency of 391MHz.

In 2010, Shoufan et al. [6] described a novel cryptoprocessor architecture for the McEliece code-based (post-quantum) cryptosystem with about 103-bit pre-quantum security level. In the key generation module, they need to transform a large public key to its systematic form. To achieve this, they built two 11×11 systolic arrays (TRI-SA and SQR-SA) as described in [6, Section 6]. They performed Gaussian elimination on large matrices of size $550\times2{,}048$ on a Xilinx Virtex 5 device. This is the only previous work which discusses Gaussian elimination for large matrices in hardware.

However, they did not provide code nor performance results of their design for Gaussian elimination. Only results for their overall cryptoprocessor design are given.

IV. Design and Implementation

A. Working With Large Matrices

As mentioned in Section I, using one large systolic architecture to do Gaussian elimination on large matrices is not practical due to the resource limitations of FPGAs. Instead of processing the input matrix on the whole, prior work [6] proposes operating on column blocks of the input matrix. Their design uses two systolic processor arrays, TRI-SA and SQR-SA, to simulate the functionality of the original large array by storing and replaying the outputs of the processor array accordingly. A classical (software) implementation of Gaussian elimination sequentially picks a single row as pivot row and eliminates the entries in the corresponding column of the remaining rows. The design in [6] picks a block of n rows at once and eliminates the corresponding columns all together.

The architecture in [6] is composed of two basic processor elements: `processor_A` and `processor_B`, similar to the design in [5]. The processor array TRI-SA has an upper-right diagonal shape similar to the original processor array from [5] (see Figure 1). It contains `processor_A` elements that are in charge of computing the pivot elements for the elimination and `processor_B` elements that apply (together with `processor_A`) the row transformations necessary for elimination. The processor array SQR-SA contains only `processor_B` elements. It is used to perform the row operations on the remaining column blocks of the matrix, as defined by the outputs of TRI-SA.

The design in [6] divides the system-solving process into two passes of Gaussian elimination: one for triangularization (forward elimination) and one for systemization (backward elimination). It iteratively uses the two processor arrays TRI-SA and SQR-SA to process corresponding matrix column blocks. After the first pass, the left part of the matrix is eliminated into an upper-right triangular matrix where the diagonal elements are all one. After the second pass, the partially eliminated matrix is flipped and then eliminated in a similar way as during the first operation. After this second elimination, the left part of the matrix is turned into the identity matrix and the linear system is completely solved.

B. Design of Our Architecture for Large Matrices

Our design is based on [5] and improves upon [6]. We use a similar notation as [6] whenever possible in order to simplify comparison.

We improve the prior design by combining TRI-SA and SQR-SA into one square module `comb_SA` which has diagonal processor elements that can be used either as `processor_A` or `processor_B`. These processor elements are called `processor_AB`. This approach allows us to save about $1/3$ of the logic required by [6] for TRI-SA and SQR-SA. Figure 2 shows the design details of our new `comb_SA` module.

Similar to [6], our algorithm uses several phases where in each phase n pivoting rows are picked at once. Each phase then requires several steps in order to perform the required row operations on all column blocks. To simplify this process, we store the matrix in a column-block format in the on-chip block memory.

To enable a wide range of applications, our implementation is parameterized: the block size n can be chosen according to the requirements, e.g., small in order to reduce resources, large in order to reduce computing time, or according to the memory architecture in case the word size of the memory is fixed. Furthermore, the number of rows (l) and columns (k, where $k \geq l$) can be set as needed. For simplification, both l and k must be multiples of n; otherwise l and k are simply rounded up to the next multiple of n.

C. Implementation

We implemented our design in a hierarchical way: the core are processors of type `processor_AB` and `processor_B`. These processors are organized in an $n \times n$ array structure within the module `comb_SA`. The module `comb_SA` is instantiated in the module `step` that computes the elimination on one column block of width n. In turn, `step` is instantiated in the module `phase` that computes the elimination of a certain row block for all remaining column blocks. Finally, `phase` is instantiated within the module `systemize` that uses `phase` repeatedly in order to eliminate all row blocks.

a) Modules `comb_SA`, `processor_AB`, and `processor_B`: Figure 2 shows `comb_SA` implemented using an array of `processor_AB` and `processor_B` instances. Figure 3a and Figure 3b show input and output ports of `processor_AB` and `processor_B` respectively in more detail. The input data is passed into the processor array by the `data_in` ports of the processors in the top row. The output values of `comb_SA` come from the `data_out` ports of the processors in the bottom row. Within `comb_SA`, the `data_out` ports of processors in one row are connected to the `data_in` ports of the next row by a register. The `start`

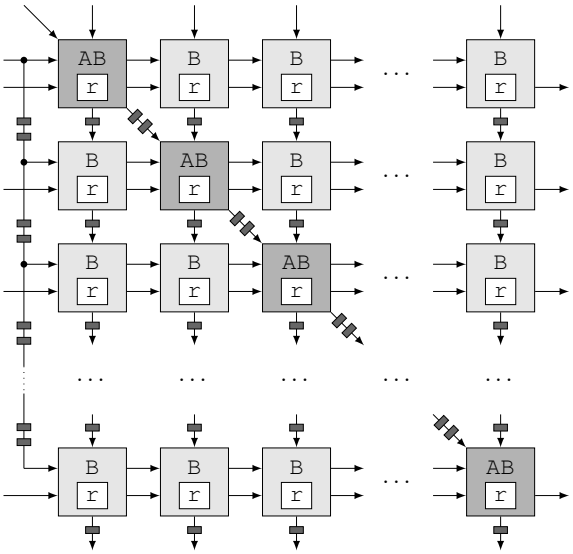
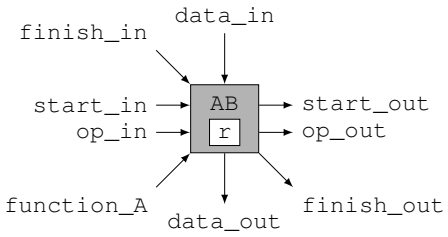
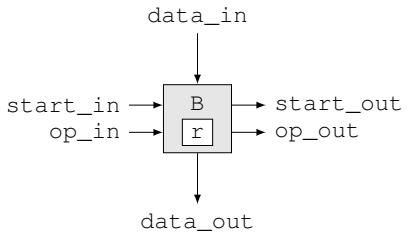


Fig. 2: Layout of module `comb_SA`. Input `function_A` to the `processor_AB`s is not shown. Registers are shown as boxes on the wires connecting processors.



(a) Module `processor_AB`.



(b) Module `processor_B`.

Fig. 3: Details of input/output ports of modules `processor_AB` and `processor_B`.

and `finish` signals are delayed for two cycles between the rows by using two registers as shown in Figure 2.

In contrast to [5] and [6], our design is a systolic line rather than a systolic array: all elements in one row operate logically in the same cycle, there are no registers between processors in the same row. The advantage of our design is that we do not need shift registers for input and output of the data.

Processing the rows of the matrix works as follows: If the signal `function_A` is low, each `processor_AB` behaves exactly like a `processor_B`. Otherwise, `processor_AB` is used in order to find pivot elements for the respective column and to generate instructions for the `processor_B` modules within the same row. The signal `start_in` must be high for one cycle at the beginning of the computation. If `start_in` is high, the value of `data_in` is stored in an internal register `r`. In case the first `data_in` value already is one, the pivot element already has been found. Otherwise, consecutive `data_in` values are forwarded to `data_out` until `data_in` is one. Now, `r` is forwarded to `data_out` and `data_in` is stored in `r`; the register value `r` is "swapped" with the input data. If values are simply forwarded, `op_out` is set to the operation *pass*, i.e., `2'b00`. When `r` and `data_in` are swapped, the processor generates the command *swap*, i.e., `2'b01`. Once the pivot element is found, subsequent rows either need to pass (operation *pass*, i.e., `2'b00`) when `data_in` is already zero or the row needs to be added to the pivoting row, i.e., `data_out` is computed as the sum of `r` and `data_in`. In the latter case, the operation *add*, i.e. `2'b10`, is issued. The operation of `processor_AB` is also shown as a truth table in Table I.

TABLE I: Truth table for `processor_AB`.

inputs			state		outputs	
start	finish	data	r	r+	op	data
1	0	d	x	d	x	0
0	0	0	r	r	*pass*	0
0	0	1	0	1	*swap*	0
0	0	1	1	r	*add*	0
0	1	x	x	x	*swap*	r

TABLE II: Truth table for `processor_B`.

inputs			state		outputs
start	data	op	r	r+	data
1	d	x	x	d	0
0	d	*pass*	r	r	d
0	d	*swap*	r	d	r
0	d	*add*	r	r	d + r

The module `processor_B` applies the operations that have been computed earlier by `procesor_AB` to its input data. If `start_in` is high, the internal register `r` is set to `data_in`. If `op_in` is *pass*, `data_in` is simply forwarded to `data_out`. If `op_in` is *swap*, register `r` is forwarded to `data_out` and `data_in` is stored in `r` instead. If `op_in` is *add*, register `r` is added to `data_in` and the result is forwarded to `data_out`. The operation of `processor_B` is also shown in Table II.

Finally, once all input data has been processed, we have fully eliminated the pivot rows as well. To start the elimination, the signal `finish` of `processor_AB` is set to high for one cycle. In this case, `processor_AB` sets `data_out` to `r` and issues the command *swap*. Therefore, all consecutive instances of `processor_B` in that row also forward their internal state to `data_out` to be processed by the following rows of processors.

b) Modules `step` and `phase`: The purpose of the module `step` is to compute the elimination of one column block of the matrix. The module `phase` invokes `step` repeatedly in order to apply the elimination operations to all column blocks.

The `step` module streams all rows of the targeted column block specified by the `phase` module through `comb_SA` and stores the output in place of the matrix data. In the first step of a phase, the pivot elements are within the column block that is being processed. Therefore, for the first step, the signal `function_A` is set to high by the `phase` module, causing the `processor_AB` instances to pick pivot elements and to compute operation commands for the remaining processors in the row. The `op_out` values are stored by the `step` module in a dedicated memory for processing the next steps. For consecutive column blocks, `function_A` is set to low and the previously stored operations are replayed into `op_in`.

c) Module `systemize`: This module invokes the `phase` module l/n times in order to perform the elimination on all row blocks. Eventually, the final result of the elimination is available in memory.

After the first phase, all rows including the pivot rows have been eliminated with respect to the first n rows. To simplify address calculations, we store the pivot rows in the bottom of the matrix. Therefore, in the second phase, the pivot rows can be picked from the top of the matrix. The last n input rows to `comb_SA` are the pivot rows from the first phase. These rows also get processed with the current pivot rows and therefore they are further eliminated. After the second phase, the last n rows are the pivot rows from the second phase, the second-last n rows ($l-2n+1$ to $l-n$) are the pivot rows from the first phase. Following this scheme, phase by phase all rows are reduced by all pivot rows and finally the systematic form (reduced row echelon form) of the input matrix is computed.

D. Comparison of Single-Pass and Dual-Pass Variants

The algorithm in [6] computes the reduced row echelon form of the input matrix by applying a systolic array design for Gaussian elimination twice in two passes. In both passes, the number of processed rows decreases by n in each phase. This approach is also possible for our systolic line design.

We now show that our single-pass approach that operates on all l rows in each phase is more efficient than a dual-pass approach that operates on n rows less in each phase.

In the dual-pass case, the first phase of Gaussian elimination processes the whole matrix. In this phase, each step takes $l+2n$ clock cycles to finish processing its corresponding n-column block of l rows. After this phase, n rows are in the desired triangular form. For the second phase, since there are n rows less to process, each step requires only $(l-n)+2n = l+n$ cycles. Iteratively, the steps in phase i each require n cycles less compared to steps in the previous phase $i-1$, i.e., $(l-in)+2n$ cycles. Phase i requires $\frac{k}{n}-i$ steps. Thus in total, it takes $2\sum_{i=0}^{\frac{l}{n}-1}(l+2n-in)(\frac{k}{n}-i)$ clock cycles to compute the reduced row echelon form using two passes. (The runtime of [6] is very similar except their design has a $3n$ instead of $2n$ overhead per step because they use a systolic array.)

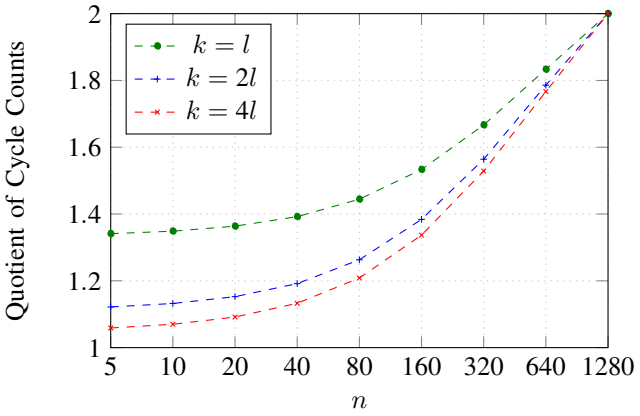


Fig. 4: Quotient of the dual-pass systolic line approach divided by our single-pass systolic line approach ($l = 1280$).

Our design performs both forward and backward elimination in one single pass. The first phase processes l rows of data which takes $l+2n$ clock cycles. In each phase, *all* l rows are reduced with respect to the current pivot rows. Therefore, in the second phase (as well as all following phases), we need all l rows of data as input. Thus, in our design, each step takes a fixed number of $l+2n$ cycles. The first phase requires $\frac{k}{n}$ steps; thereafter, each phase takes one step less compared to the previous phase. In total, we require $\sum_{i=0}^{\frac{l}{n}-1}(l+2n)(\frac{k}{n}-i)$ clock cycles (plus a few cycles of overhead due to pipelining) in order to compute the reduced row echelon form.

Figure 4 shows the theoretical analysis of the cycle count for the two variants for different sizes of n. Our single-pass systolic line approach is always better compared to a dual-pass systolic line approach in terms of number of cycles, especially when the matrix is almost square.

However, the dual-pass approach detects if the matrix is invertible already after its first pass of the Gaussian elimination. Our single-pass approach needs to finish the whole process first. Therefore, the dual-pass approach is a better choice when the matrix is not guaranteed or known to be invertible and when an early abort of the system solving is beneficial.

Note, we use multiples of 5 for n because the embedded memory blocks in our Altera Stratix V FPGA can be used most efficiently when using a word size of this form.

V. Evaluation

A. Trade-off between Area and Time

In our systolic line design, there is a trade-off between area and time, controlled by the width n of `comb_SA`. Bigger n means higher parallelism and less computing time, but at the same time more logic. As mentioned before, the critical path in our architecture is determined by the width of the rows of `comb_SA`. Figure 5 shows that the maximum clock frequency (Fmax) drops as the size of the systolic line architecture (n) grows because of the longer routing paths on the FPGA. However, for moderately large n up to $n \leq 80$, Fmax can

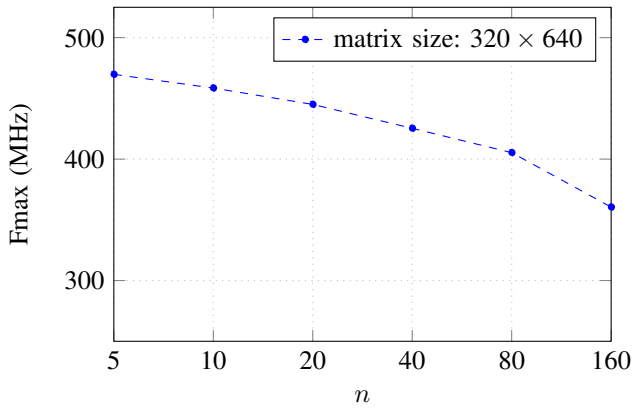


Fig. 5: Maximum clock frequency (Fmax) achieved for different choices of n.

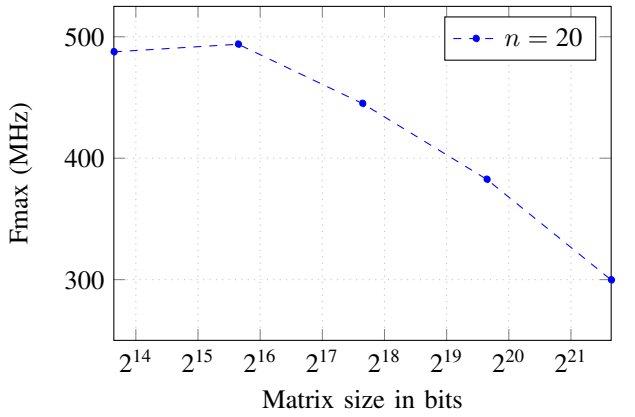


Fig. 6: Maximum clock frequency (Fmax) achieved for different matrix sizes (in bits); see Table III.

be kept above 400MHz, while for larger n, a relatively high Fmax of 360MHz can be maintained as well.

Since we are using a small- to medium-sized systolic line architecture when processing large-sized matrices, logic utilization is no longer a constraint compared to the standard designs discussed in literature. Instead, the available on-chip memory determines the largest size of the matrix that can be processed by our design. Even larger matrices can be processed when using off-chip memory.

To improve the frequency, it is possible to reduce the critical path from the full width of a row of `comb_SA` by insertion of registers every c columns. This is similar to a systolic array that has registers between each column ($c = 1$) but would use larger values of c, such as $n/2$, $n/4$, etc., depending on the desired frequency. Using additional pipelining steps increases the total number of cycles and requires additional shift registers, but it allows higher frequency.

B. Performance and Resource Usage vs. Matrix Size

Figure 6 shows the maximum frequency of our architecture for doing Gaussian elimination from medium-sized to large-sized matrices when the size of the systolic line architecture is fixed to $n = 20$. With an increasing matrix size, the maximum frequency drops significantly. This is caused by the implementation of the memory on the FPGA. Our Stratix V FPGA provides on-chip memory in blocks of 20K bits. Larger memories are composed of several of these blocks which requires additional logic. If the required memory is very large, data paths and additional logic within the memory blocks have a big impact on the maximum frequency that can be achieved. A solution for this problem would be to introduce additional pipeline steps into the data path of the memory. Detailed performance and resource usage is shown in Table III.

C. Comparison with Related Work

Table IV presents a comparison of performance and resource usage of our design with the GSMITH design in [7], the systolic network design in [8], and the SMITH design in [9]. These designs perform Gaussian elimination for medium-sized matrices; their processor array has the same size as the input matrix. Our design is not intended for matrices of this size but optimized for iterative operation on large matrices. To achieve a fair comparison, we compare only our `comb_SA` module using a processor array of a similar size to their designs. The resource usage of [8] and [9] is only provided for Spartan 3 FPGAs. Therefore, we synthesized our `comb_SA` design for this FPGA. Compared with these three designs, our design achieves very good performance in terms of frequency, area, and total runtime.

Shoufan et al. in [6] compute on large matrices of size $550 \times 2{,}048$. They implement a complete crypto system and do not provide details on the performance of their system solver. In order to compare our design with [6], we calculated the expected number of clock cycles for their design based on their description. Since we use a single-pass systolic line approach, while they use a dual-pass systolic array approach, our design takes less clock cycles to finish the elimination process. Since no performance and resource usage data is provided for this part in their paper, no detailed comparison can be made.

VI. Conclusion

Need for Gaussian elimination of large matrices arises in various cryptographic and cryptanalytic algorithms. The presented work is the first to show results of Gaussian elimination for matrices of up to 4,000 × 8,000 elements. The new architecture introduced a new `comb_SA` module and uses multiple *steps* and *phases* to simulate the functionality of the original large systolic architecture. The whole design is configurable both in the size of the `comb_SA` module (n) and the matrix size, $l \times k$. With this design, we can achieve sub-μs performance for the Gaussian elimination of medium matrices and performance on the order of tens to hundreds of ms for large matrices. We also obtained 1/3 reduction in logic use compared to related work [6] and have very efficient use of block memory. Our design can be easily realized on both Altera and Xilinx FPGAs without any modification since no Xilinx or Altera specific optimizations are involved in the

TABLE III: Altera Stratix V synthesis results for different matrix sizes with fixed $n = 20$.

l	k	Clock Cycles[b]	Fmax (MHz)	Runtime (ms)	Logic[a]	Registers	Matrix Size (bits)	Total Memory (bits)
80	160	3,120	488	0.0064	616	1,172	$1.56 \cdot 2^{13}$	$2.15 \cdot 2^{13}$
160	320	20,000	494	0.04	638	1,193	$1.56 \cdot 2^{15}$	$1.81 \cdot 2^{15}$
320	640	141,120	445	0.32	648	1,207	$1.56 \cdot 2^{17}$	$1.67 \cdot 2^{17}$
640	1,280	1,055,360	383	2.8	670	1,226	$1.56 \cdot 2^{19}$	$1.61 \cdot 2^{19}$
1,280	2,560	8,152,320	300	27	726	1,243	$1.56 \cdot 2^{21}$	$1.59 \cdot 2^{21}$
2,560	5,120	64,064,000	229	280	935	1,279	$1.56 \cdot 2^{23}$	$1.57 \cdot 2^{23}$
4,000	8,000	242,804,000	192	1,300	1,458	1,342	$1.91 \cdot 2^{24}$	$1.92 \cdot 2^{24}$

[a] Logic utilization is counted in ALMs (Adaptive Logic Modules).
[b] Theoretical calculation, does not take into account a few cycles of overhead.

TABLE IV: Comparison with existing FPGA implementations of Gaussian elimination.

Design	n	l	k	Clock Cycles	Fmax (MHz)	Runtime (ms)	Logic[a]		Registers	FPGA
							ALMs	Slices		
[7]	50	50	50	150	150	0.00100	(3,106)	3,713	2,574	Xilinx Spartan 3
[8]	50	50	50	50	–[d]	–[d]	(9,256)	11,065	–[d]	Xilinx Spartan 3
[9]	50	50	50	100[e]	300	0.00033[e]	(3,349)	4,004	–[d]	Xilinx Spartan 3
our	50	50	50	150	178	0.00084		3,129	5,236	Xilinx Spartan 3
our	50	50	50	150	413	0.00003	2,618		5,725	Altera Stratix V
[6]	11	550	2,048	5,323,450[c]	–[d]	–[d]	–[d]	–[d]	–[d]	Xilinx Virtex 5
our	11	550	2,048	4,624,100[b]	305	15[b]		246	538	Xilinx Virtex 5
our	11	550	2,048	4,624,100[b]	332	14[b]	437		613	Altera Stratix V

[a] Conversion from Xilinx Spartan 3 Slices to Altera Stratix V ALMs: 1 ALM = 3,129 / 2,618 ≈ 1.2 Slices.
[b] Theoretical calculation, does not take into account a few cycles of overhead.
[c] Theoretical calculation based on design description.
[d] Exact information not provided in reference.
[e] Average depending on input matrix.

design. The Verilog source code of our design is available as Open Source at http://caslab.eng.yale.edu/code/gausselim.

Acknowledgment

This work was supported in part by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005 and by the Commission of the European Communities through the Horizon 2020 program under project number 645622 (PQCRYPTO).

References

[1] C.-L. Wang and J.-L. Lin, "A systolic architecture for computing inverses and divisions in finite fields GF(2^m)," *IEEE Transactions on Computers*, vol. 42, no. 9, pp. 1141–1146, 1993. 1, 2

[2] R. J. McEliece, "A public-key cryptosystem based on algebraic coding theory," *JPL DSN Progress Report*, vol. 44, pp. 114–116, 1978. 1

[3] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Problems of Control and Information Theory*, vol. 15, no. 2, pp. 159–166, 1986. 1

[4] D. Augot, L. Batina, D. J. Bernstein, J. Bos, J. Buchmann, W. Castryck, O. Dunkelman, T. uneysu, S. Gueron, A. ulsing, T. Lange, M. S. E. Mohamed, C. Rechberger, P. Schwabe, N. Sendrier, F. Vercauteren, and B.-Y. Yang, "Initial recommendations of long-term secure post-quantum systems," PQCRYPTO — Horizon 2020 ICT-645622, Tech. Rep., 2015, www.pqcrypto.eu.org/docs/initial-recommendations.pdf. 1

[5] B. Hochet, P. Quinton, and Y. Robert, "Systolic Gaussian elimination over GF(p) with partial pivoting," *IEEE Transactions on Computers*, vol. 38, no. 9, pp. 1321–1324, 1989. 1, 2, 3, 4

[6] A. Shoufan, T. Wink, H. G. Molter, S. A. Huss, and E. Kohnert, "A novel cryptoprocessor architecture for the McEliece public-key cryptosystem," *IEEE Transactions on Computers*, vol. 59, no. 11, pp. 1533–1546, 2010. 1, 3, 4, 5, 6, 7

[7] A. Rupp, T. Eisenbarth, A. Bogdanov, and O. Grieb, "Hardware SLE solvers: Efficient building blocks for cryptographic and cryptanalytic applications," *the VLSI Journal INTEGRATION*, vol. 44, no. 4, pp. 290–304, 2011. 2, 3, 6, 7

[8] R. P. Jasinski, V. A. Pedroni, A. Gortan, and W. Godoy Jr., "An improved GF(2) matrix inverter with linear time complexity," in *International Conference on Reconfigurable Computing and FPGAs — ReConFig*, 2010, pp. 322–327. 2, 6, 7

[9] A. Bogdanov, M. Mertens, C. Paar, J. Pelzl, and A. Rupp, "SMITH — a parallel hardware architecture for fast Gaussian elimination over GF(2)," in *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems — SHARCS*, 2006. 2, 3, 6, 7