A Hardware Perspective on the ChaCha Ciphers: Scalable Chacha8/12/20 Implementations Ranging from 476 Slices to Bitrates of 175 Gbit/s

Johannes Pfau*, Maximilian Reuter†, Tanja Harbaum*, Klaus Hofmann† and Jürgen Becker*
*Institut für Technik der Informationsverarbeitung, Karlsruher Institut für Technologie
†Integrierte Elektronische Systeme, Technische Universität Darmstadt
Email: pfau@kit.edu, maximilian.reuter@ies.tu-darmstadt.de, harbaum@kit.edu,
klaus.hofmann@ies.tu-darmstadt.de, becker@kit.edu

***Abstract*—AES (Advanced Encryption Standard) accelerators are commonly used in high-throughput applications, but they have notable resource requirements. We investigate replacing the AES cipher with ChaCha ciphers and propose the first ChaCha FPGA implementations optimized for data throughput. In consequence, we compare implementations of three different system architectures and analyze which aspects dominate the performance of those.**

Our experimental results indicate that a bandwidth of 175 Gbit/s can be reached with as little as 2982 slices, whereas comparable state of the art AES accelerators require 10 times as many slices [1]. Taking advantage of the flexibility inherent in the ChaCha cipher, we also demonstrate how our implementation scales to even higher throughputs or lower resource usage (down to 476 slices), benefiting applications which previously could not employ cryptography because of resource limitations.

***Index Terms*—chacha, cryptography, symmetric cipher, fpga, evaluation, stream cipher, arx**

I. Introduction

As private data has recently become increasingly valuable and problems with espionage and data theft arise, data security has become more and more important in various domains. Whereas cryptography has been widely deployed in the IoT (Internet of Things) and consumer device domains, applications in communication, data processing and high-bandwidth sensor data acquisition present higher demands on throughput, limiting the adoption. High-throughput AES accelerators of up to 260 Gbit/s have been proposed [1], but the cost of 35 328 slices is often too high for applications where cryptography is not perceived as an integral part. In order to obtain a better throughput per slice ratio, we provide a detailed analysis of efficient hardware architectures for the ChaCha cipher, a detailed analysis of the performance limiting factors as well as an overview of parallelization possibilities. We describe hardware building blocks for the cipher and three different implementations built on these, then compare performance and hardware resource characteristics for the different implementations. Compared to the state-of-the art, our implementation focuses on throughput, whereas existing implementations focus on low resource utilization. In addition, our implementation can be configured to balance throughput and resource requirements. We further show that a pipeline implementation is much more efficient compared to multiple instances of independent block primitives, as often used in state-of-the-art IP-core implementations.

This work was supported by the German Research Foundation (DFG)

II. Related Work

The ChaCha ciphers are a family of symmetric stream ciphers proposed by D. J. Bernstein [2] as an evolution of the earlier published Salsa ciphers [3]. As endorsed by crypt-analysis publications of Salsa and ChaCha ciphers [4], [5], it is a more conservative design than AES, providing similar or better level of security. ChaCha is an Add-Rotate-XOR cipher: All operations performed by the ChaCha algorithms are based on additions, XOR operations and rotations (cyclic bit shifts) of 32 bit data words. The ChaCha$N$ cipher family provides three standardized variants, which differ only in the number of operations performed on the cipher state ($N = 8$, 12 or 20 rounds). A larger number of rounds increases data diffusion and therefore security of the cipher, but it also increases the processing time (scaling linearly with the number of rounds) when encrypting or decrypting data.

For ciphers such as AES, studies on hardware implementations have been presented [1], [6], but for ChaCha ciphers, publications such as [7] mostly describe optimizations for efficient software implementations. To the best of our knowledge, there are three hardware implementations of the ChaCha cipher: A commercial ChaCha20 implementation by Xiphera, Inc. [8]. This IP core was developed as an TLS accelerator and therefore integrates a Poly1305 implementation to calculate message authentication codes. It does however not allow to customize the number of rounds and does not support ChaCha variations with different counter or nonce sizes. Also, due to it's closed-source IP-core nature, the building blocks can not be reused to implement algorithms based on ChaCha, such as the Blake2 hash function [9]. The second implementation is an open source Verilog implementation available from J. Strömbergson [10]. This implementation allows configuring the number of rounds, it does however not allow to make trade-offs between throughput and hardware resource requirements. The third implementation was published by At et al. as part of

their BLAKE implementation [11]. It is optimized for very-low resource usage (49 slices) at the cost of low throughput of only 595 Mbit/s.

III. A short Summary of the ChaCha Algorithm

In this section, we briefly reiterate the ChaCha algorithm as specified in [2]. As a stream cipher, ChaCha$N$ first generates a stream of key data called keystream. To encrypt data, the bytes of this keystream are combined with the bytes of the data stream by an XOR operation, yielding the cipherstream (Eq. 1). As decryption works the same way (Eq. 2), both the en- and decrypting parties have to generate the same keystream. The whole algorithm for encryption and decryption is therefore identical.

$$\text{cipherstream} = \text{keystream} \oplus \text{datastream} \tag{1}$$

$$\text{datastream} = \text{keystream} \oplus \text{cipherstream} \tag{2}$$

In order to generate the keystream, ChaCha performs various operations on a matrix consisting of 32 bit unsigned integers. The initial matrix M is formed as follows:

$$M = \begin{pmatrix} 61707865 & 3320646e & 79622d32 & 6b206574 \\ \text{key}_0 & \text{key}_1 & \text{key}_2 & \text{key}_3 \\ \text{key}_4 & \text{key}_5 & \text{key}_6 & \text{key}_7 \\ \text{counter}_0 & \text{counter}_1 & \text{nonce}_0 & \text{nonce}_1 \end{pmatrix} \tag{3}$$

The first entries shown are the 16 constant bytes "*expand 32-byte k*" in hexadecimal notation, followed by the symmetric en-/decryption key. The counter is used to provide the current position in the keystream. For the first block (the first 64 bytes in the keystream) it is zero, for the next block it is one, etc. ChaCha allows for random access to the keystream: It is possible to calculate blocks at any stream offset without calculating any previous block. The last entry, the nonce (number-used-once), is a number unique to each keystream.

To process the matrix M, ChaCha$N$ performs $N = 8$, 12 or 20 rounds of operations on the matrix (Lst. 1):

Listing 1: The $rounds_N(M)$ Operation

```
foreach i in (0 .. N-1)
| in = i.odd ? diags(M) : cols(M)
| out(0..4) = qround(in(0..4))
| i.odd ? diags(M)= out : cols(M)= out
```

Listing 2: The $qround(a, b, c, d)$ Operation

```
a += b; d ^= a; d <<<= 16;
c += d; b ^= c; b <<<= 12;
a += b; d ^= a; d <<<= 8;
c += d; b ^= c; b <<<= 7;
```

As shown in Fig. 1, even rounds operate on the four columns of the matrix (column rounds), odd rounds on four diagonal vectors (diagonal rounds). Each round then performs the quarter-round sub-operations ($qround$) once per input vector. Quarter-rounds consist of Add, XOR (^=) and rotate (<<<=) operations and are defined in Lst. 2. After matrix M has been

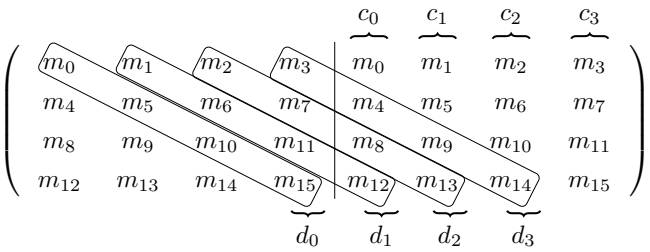


Fig. 1: Columns c_i and Diagonals d_i for $qround$ Inputs

processed through N rounds, the last processing step for the ciphertext block keystream(counter) is to add the processed matrix to the initial matrix:

$$\text{keystream(counter)} := M + rounds_N(M) \tag{4}$$

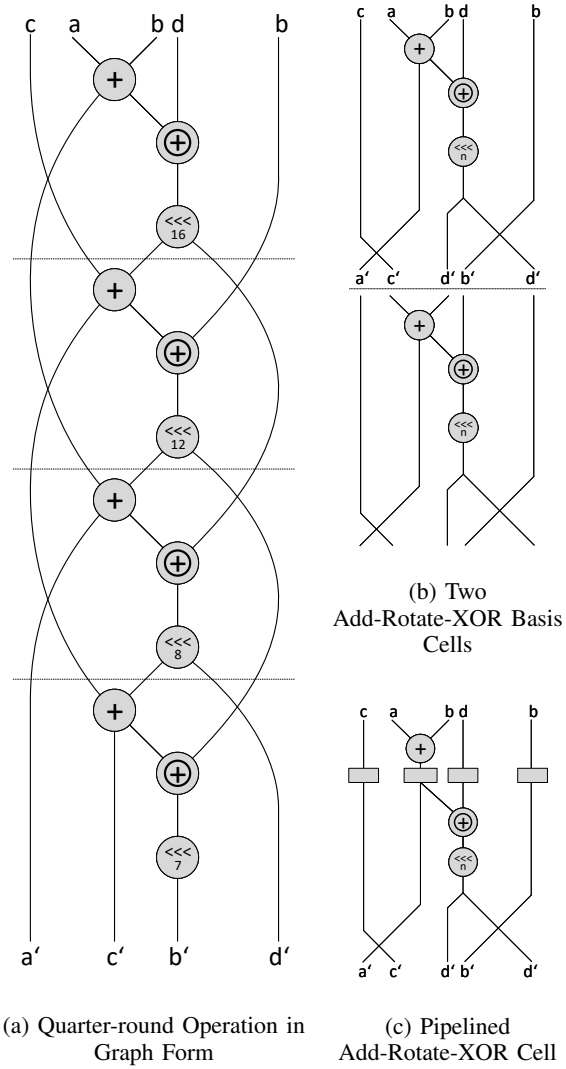
To obtain the next 64 B of the keystream, the counter field in the initial matrix M is incremented by one and the process in Eq. 4 is repeated for the new matrix.

IV. A Hardware Perspective on the ChaCha Cipher

Fig. 2a shows the quarter-round operation of Lst. 2 in a graphical form, depicting both the operations used and the data flow of the algorithm. As can be seen, each output a, b, c and d is dependent on each input and on intermediate results, showing that parallelization of this operation is not easily possible. This restriction does not come as a surprise: The main purpose of the quarter-round operation is to disperse data in the matrix, ensuring that each bit in the final result depends on the whole initial matrix. As such, there is an inherent conflict here: In order to be cryptographically secure, algorithms have to introduce many data dependencies. On the other hands side, these data dependencies limit parallelization opportunities.

Fig. 2b shows a slightly modified form of the first two sections of the graph, suggesting that the whole operation can be built out of four basis cells. These Add-Rotate-XOR (ARX) cells form the base operation used in the ChaCha cipher. As suggested by Fig. 2b, the outputs need to be permuted in order to directly chain these cells: The outputs of the first cell, a', b', c' and d' are updated input variables as described in line one of Lst. 2. Line two of Lst. 2 then applies the same operations on the updated input variables. As it maps the inputs differently to the operations, the basis cell has to perform that permutation. If four basis cells are connected serially, the final result will be in the same order as the input variables.

As shown in the graph, the rotation distance is different for each stage. To handle this in the basis cell, the distance can be required to be a constant parameter. The rotation operation will then be implemented as a simple wire permutation by synthesis tools, introducing no additional logic delay. As such a cell can be used for only one stage in the quarter-round, four physical copies of these cells with different rotation distances will be required. An alternative is to make the rotation distance changeable as a runtime input: In that case, the rotation operation will be synthesized into a 4-input multiplexer structure,



(a) Quarter-round Operation in Graph Form

(b) Two Add-Rotate-XOR Basis Cells

(c) Pipelined Add-Rotate-XOR Cell

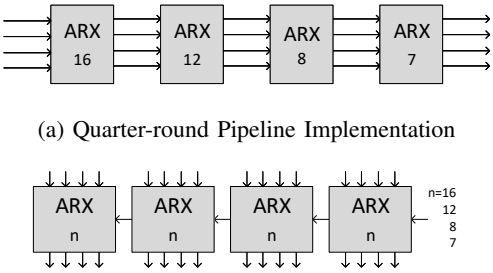
Fig. 2: Quarter-round Decomposed to ARX Basis Cells

requiring additional hardware resources and introducing logic delay.

In Fig. 2c we propose one way to pipeline this ARX basis cell. For clarity, we show the four pipeline registers inserted after the addition operation, but in our implementation, we move one register into the adder. This benefits FPGA implementations which use DSP blocks for the addition, as these DSP blocks usually require internal pipelining registers for enhanced performance. Depending on how the rotate operation is implemented, additional pipelining between the XOR and rotate operation may be beneficial. To keep the implementation simple, we do not explicitly insert a pipelining stage there. Instead, the depth of the post-addition pipeline is adjustable and we rely on retiming optimization to distribute the registers.

A. Quarter-Round and Rounds

We use two different options to combine ARX cell forming the ChaCha cipher, both shown in Fig. 3. Fig. 3a employs a



(a) Quarter-round Pipeline Implementation

(b) Parallel ARX Cells Implementation

Fig. 3: Different Building Blocks for the $rounds_N$ Operation

pipeline structure implicitly suggested by the graph in Fig. 2. In this implementation, the quarter-round is implemented as one pipeline consisting of four physical ARX cells. This pipeline does the complete round processing for a quarter of the matrix and each operation can be mapped to one physical ARX cell. The rotation distance for each cell is constant, reducing logic delay as previously explained. Another benefit of this structure is the possibility of deep-pipelining: As the ARX basis cell is pipelined, an in-series chain of these cells should not pose further restrictions on the critical path and overall performance.

In order to utilize such a deep-pipeline completely, all stages have to be filled with independent data vectors, i.e. any vector in the pipeline may not in any way depend on the final $qround$ result for any other vector in the pipeline. As can be seen from Fig. 1, Fig. 2 and in Lst. 1, the four quarter-rounds in each round are independent. Their inputs depend on the results of the previous round, but not on any result of other quarter-rounds in this same round. Problems however arise after the vectors of one round have been pushed into the pipeline: Assume the cipher is processing round zero. It first pushes the columns c_0, c_1, c_2 and c_3, one each cycle. Now in cycle five, it would have to push one of the diagonals d_0, d_1, d_2 or d_3 of round 1. However, as can be seen in Fig. 1, each of these diagonals depends on some inputs of the first round's columns. One possible solution is to start processing the diagonals only when previous columns have been completely processed. This however reduces the performance of the implementation. As individual keystream blocks in ChaCha are independent, a better solution is to simply prepare the initial matrix M_{n+1} for the next keystream block. We then push the columns of this matrix M_{n+1} into the $qround$ pipeline. This idea can be generalized to any number of matrices, depending on the pipeline depth. In general, processing then alternates between the rounds of multiple matrices.

The alternative in Fig. 3b is closer to the usually employed software approach. Here, at least one ARX cell is used with a configurable rotation distance. This way, the four stages of a quarter-round can be processed consecutively and the result of every step will be written back to the processing matrix. We use four ARX cells in parallel to enhance throughput,

similar to the way software ChaCha implementations use SIMD optimizations. The main difference to the pipelined implementation is that four parallel ARX ciphers will process the whole matrix at once, instead of processing vectors consecutively. As one cycle always yields a complete matrix, there are no pipelining problems when starting to process the next round. It is therefore not necessary to alternate processing between multiple matrices. The main drawback is that this requires a runtime-adjustable rotation distance, as the distance will be different for each processing cycle.

B. Final Data En-/Decryption Operation

After the matrix has been processed through the $rounds_N$ implementation, the final addition of Eq. 4 as well as the encrypt (Eq. 1) or decrypt operation (Eq. 2) need to be performed. As the final operation is an addition followed by an XOR operation, it is possible to reuse the ARX cell for this operation. In the round-processing we use implementations such as in Fig. 3, which provide only four ARX cells. The final addition and XOR are performed for every matrix element, so processing using the round cells would take four clock cycles. To improve throughput, we therefore prefer to implement the finalize operation in dedicated hardware.

V. ChaCha Hardware Architectures

Based on the round building blocks of Fig. 3, we will now present three different implementations. The top level interface is the same for all implementations: It provides an input labeled datastream, an output labeled cipherstream and a configuration port, each with valid and ready handshaking signals. As the encrypt and decrypt operations are identical, the datastream and cipherstream roles can also be swapped. The configuration port is used to configure the contents of the initial matrix, M, which is stored in 14 words of register memory. The remaining two words of the matrix are counter values, which are calculated on demand by the Counter module: This module internally contains two counters with independent enable signals. The $count1$ counter is used for the initial matrix for the $rounds_N$ operation, the $count2$ output is used to form the initial matrix for Eq. 4. This idea is based on the observation that the initial matrix of each keystream block only differs in the counter value. Instead of keeping a backlog of all initial matrices (or count values) which are currently being processed, we use one counter for matrices which have started processing and one for matrices which have finished processing. The Finalizer block contains the final addition and XOR operation of Eq. 4. In cooperation with the FSM block, it also handles valid and ready handshake signals for the datastream input and cipherstream output to disable the Round blocks if one of these ports is not ready.

A. A Pipeline-based Implementation

The Pipeline implementation is shown in Fig. 4. It consists of common elements described previously, as well as of a set of Round blocks. These blocks consist of four parallel quarter-round cells of Fig. 3a, calculating one round completely in parallel. The individual Round blocks are then chained to form a deep pipeline, taking care to permute signals connecting two rounds according to Fig. 1. If the output datastream is not ready or the input datastream is not valid, the FSM accordingly stops the Round blocks to pause keystream generation.

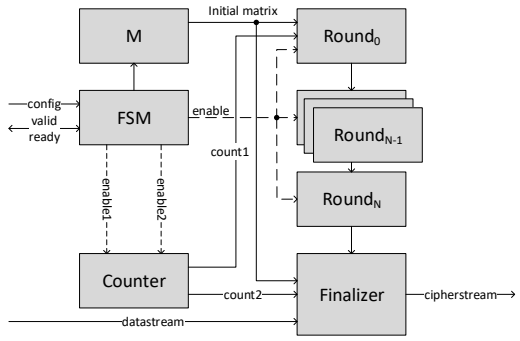


Fig. 4: Pipeline-based ChaCha Implementation

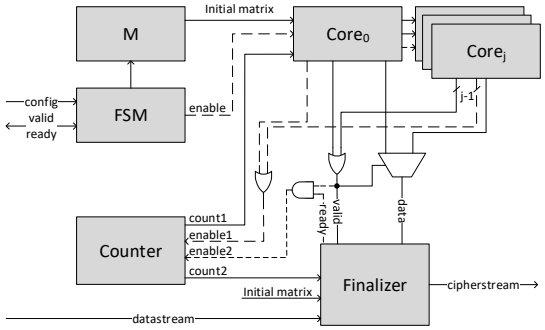


Fig. 5: Block Memory and Register Implementation

B. Block Memory and Register Implementations

Fig. 5 shows the top level architecture common to the Block Memory and the Register implementation. It mostly resembles the Pipeline implementation with one important difference: The Round blocks in the pipeline have been replaced by j Core blocks. Core blocks calculate the complete $rounds_N$ operation and their implementation differs for the Block Memory and the Register implementation. In either case, processing a matrix in a Core block takes a certain amount of cycles. The top level therefore allows to use a configurable number of cores in parallel, interleaving their outputs and enhancing throughput. As the outputs are interleaved, the maximum number of cores is reached when every cycle yields 16 words of data. Any further parallelization then requires duplicating the top-level architecture. The Counter and Finalizer blocks are shared between all blocks, requiring some additional logic to properly drive the enable signals and the Finalizer data input.

For the Block Memory Implementation, one pipelined quarter-round as in Fig. 3a is used. The four output words are then saved to four parallel Block Rams at a certain indices: Each row of the matrix is kept in one Block Ram, which then allows to read all four inputs for a column round or for

a diagonal round in parallel using proper addressing (refer to Fig. 1). The benefit of this implementation is that all operations in the quarter-round are placed into one physical pipeline implementation with constant rotation values. To always keep the pipeline utilized, multiple matrices at different offsets in the Block Ram are processed in parallel. There's one drawback to this approach though: In the final round, the outputs can be directly obtained at the quarter-round output port. However, as the final round is a diagonal round, the matrix elements will be emitted in the order $(0, 5, 10, 15)$, $(1, 6, 11, 12)$, etc.

For the register based implementation, no complex address calculation needs to be done for memory access. Instead, after the initial matrix has been loaded using a multiplexer, the complete matrix is kept in register storage internal to the parallel quarter-round implementation of Fig. 3b. The outputs are then looped back to the inputs. The block operates on the whole matrix, processing one fourth of a round per cycle. Once a round is finished, the multiplexer is configured to permute the inputs according to diagonal vs. column based indexing. This implementation outputs the result as 16 words in one cycle and does therefore not face the output order problem shown of the Bock Memory implementation.

VI. Area and Performance Comparison of the three Architectures

We use Xilinx Vivado 2018.3 and the standard synthesis strategy with retiming enabled, targeting the VC707 board and Virtex 7 XC7VX485T-2FFG1761C FPGA for implementation. Fig. 6 shows the maximum reachable clock frequency and required number of slices when increasing the Core count (see Fig. 5), whereas Tab. I gives a more detailed resource overview for configurations with Core count 1. As can be seen from these figures, maximum reachable clock frequency reduces with Core count, as increasing the number of cores stresses placement and routing phases. Throughput depends on the number of cores and the clock frequency, so Fig. 7 compares different implementations' throughput vs. the required resources. The Pipeline implementation is shown as points, as it does not provide a Core count parameter to balance resources vs throughput. In all these figures, $d =$ denotes whether DSP blocks have been used for the ARX cell (1) or not (0) and $r =$ gives the number of introduced pipeline registers. Tab. I shows that all our implementations surpass all state of the art ChaCha implementations but the low-resource optimized implementation by At et al. in regard to bitrate per slice. The Block Memory and Pipeline implementation also surpass AES state of the art, the Pipeline implementation even by a factor of 8.

These results are for ChaCha8. To calculate numbers for ChaCha12/20, divide throughput by 1.5 and 2.5 for the Register and Memory implementations. For the Pipeline implementation, resource requirements are increased by these factors and the maximum clock frequency and therefore throughput may also be affected.

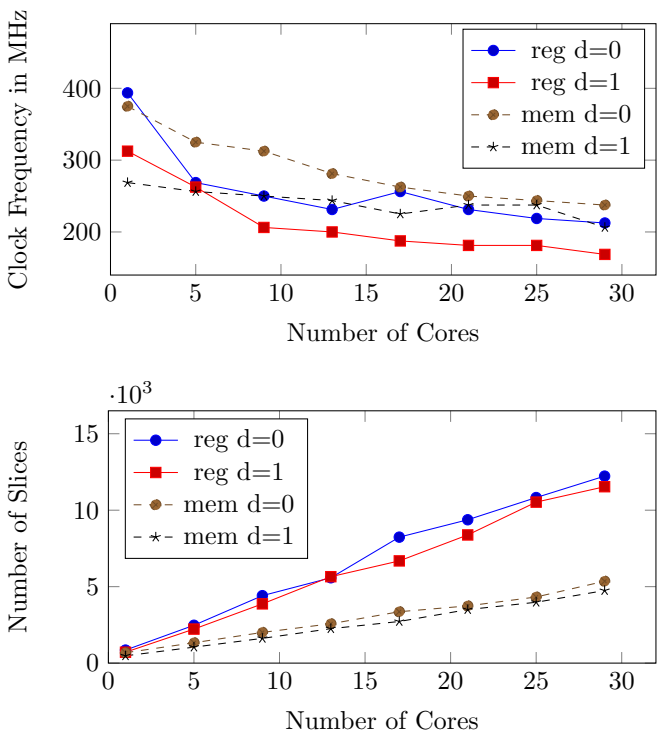


Fig. 6: Results for the Memory and Register Implementation

VII. Conclusion

In this work we presented design considerations to efficiently map the ChaCha cipher to hardware. We demonstrated three different, scalable designs and analyzed their performance and resource requirements, concluding that a fully pipelined architecture is beneficial when high throughput is required.

We have shown that the commonly employed approach of combining multiple processing core blocks to increase throughput is not optimal, as a pipeline based implementation provides better results for high-throughput applications. To obtain even-higher throughput, the Pipeline implementation can be instantiated multiple times without a decrease in the maximum clock rate: N such pipelines can operate completely without synchronization, each one i processing every $i + xN$ matrix. This therefore even enables placing the Pipeline instances into different FPGAs. As our Pipeline implementation provides a significantly higher bitrate per slice ratio than state of the art, this work enables usage of encryption in high-throughput applications which previously could not afford the resource overhead.

References

[1] A. Soltani and S. Sharifian, "An ultra-high throughput and fully pipelined implementation of aes algorithm on fpga," *Microprocessors and Microsystems*, vol. 39, no. 7, pp. 480 – 493, 2015.

[2] D. J. Bernstein, "Chacha, a variant of salsa20," 2008. [Online]. Available: http://cr.yp.to/chacha/chacha-20080120.pdf

[3] ——, "The salsa20 family of stream ciphers," in *New stream cipher designs*. Springer, 2008, pp. 84–97.

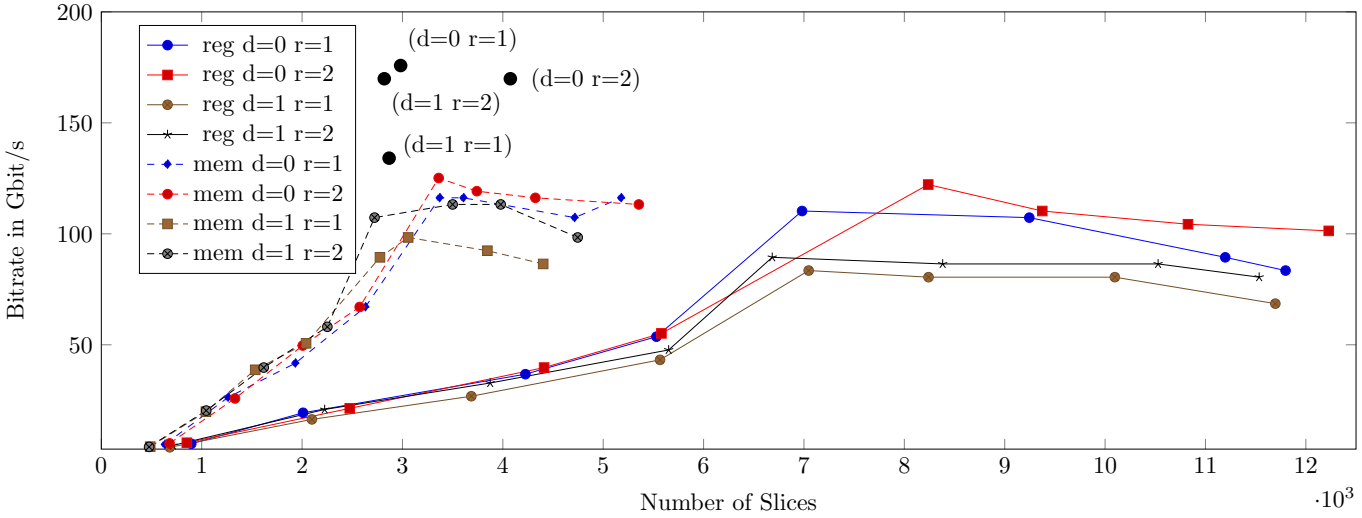


Fig. 7: Comparing Throughput vs. Number of Slices

TABLE I: Resource and Clock Frequency for Select Configurations

Variant	Cores	DSP	Depth	f_{max} MHz	LUT	FF	BRAM18	DSP	Slices	Bitrate Gbit/s	Bitr./Slice (Mbit/s) /slice
chacha_reg	1	yes	1	256.3	1693	1502	0	20	682	3.82	5.74
			2	312.5	1738	2025	0	20	713	4.66	6.69
		no	1	362.5	2369	2152	0	0	896	5.40	6.17
			2	393.7	2392	2873	0	0	852	5.87	7.06
chacha_mem	1	yes	1	275.0	862	1162	4	8	487	4.10	8.62
			2	268.7	871	1260	4	8	476	4.00	8.61
		no	1	343.8	1366	1831	4	0	633	5.12	8.25
			2	375.0	1387	2210	4	0	680	5.59	8.42
chacha_pipe	-	yes	1	281.2	4556	13484	0	144	2867	134.09	47.89
			2	356.3	5633	15707	0	144	2819	169.87	61.71
		no	1	368.7	9138	18004	0	0	2982	175.82	60.38
			2	356.3	10101	25680	0	0	4075	169.87	42.69
Xiphera (ChaCha20) [8] (Xilinx UltraScale+)	-				< 4000					0.50	$>$ ~0.26
At et al. [11] (Xilinx Virtex-6 XC6VLX75T-2)	-			356.3			2		49	0.60	12.54
Strömbergson [10] (Xilinx Artix-7 XC7A200T-3FBG484)	-			100.0	3837	1949			1076	5.96	5.67
Silitonga et al. (AES, HLS) [6] (Xilinx Zynq-7000)	-				11704	8512				43.90	~7.68
Soltani et al. (AES) [1] (Xilinx Virtex-6 XC6VLX240T)	-								35328	260.15	7.54

[4] S. Maitra, "Chosen iv cryptanalysis on reduced round chacha and salsa," *Discrete Applied Mathematics*, vol. 208, pp. 88 – 97, 2016.

[5] J.-P. Aumasson, S. Fischer, S. Khazaei, W. Meier, and C. Rechberger, "New features of latin dances: Analysis of salsa, chacha, and rumba," in *Fast Software Encryption*, K. Nyberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 470–488.

[6] A. Silitonga, F. Schade, G. Jiang, and J. Becker, "Hls-based performance and resource optimization of cryptographic modules," in *Proceedings of the 16th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA2018), Melbourne, Australia, 11th-13th December 2018*. IEEE, 2018, pp. 1009–1016.

[7] F. De Santis, A. Schauer, and G. Sigl, "Chacha20-poly1305 authenticated encryption for high-speed embedded iot applications," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '17. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2017, pp. 692–697.

[8] Xiphera Ltd. Chacha20-poly1305 product brief. [Online]. Available: https://xiphera.com/product_brief/ChaCha20_Poly1305_MPSoC.pdf

[9] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "Blake2: Simpler, smaller, fast as md5," in *Applied Cryptography and Network Security*, M. Jacobson, M. Locasto, P. Mohassel, and R. Safavi-Naini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 119–135.

[10] Joachim Strömbergson. Verilog 2001 implementation of the chacha stream cipher. [Online]. Available: https://github.com/secworks/chacha/

[11] N. At, J. Beuchat, E. Okamoto, I. San, and T. Yamazaki, "Compact hardware implementations of chacha, blake, threefish, and skein on fpga," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 2, pp. 485–498, Feb 2014.