POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Multi-Dimensional Design Space Exploration of ChaCha20 on FPGA

Original
A Multi-Dimensional Design Space Exploration of ChaCha20 on FPGA / Manfredi, E., Piscopo, V., Antognazza, F., Dolmeta, A., Mirigaldi, M., Martina, M., Masera, G., Pelosi, G., Barenghi, A.. - ELETTRONICO. - (2026), pp. 244-251. (2026 29th Euromicro Conference on Digital System Design (DSD) Krakow (Pol) 02-04 September 2026) [10.23919/DSDCPS00089.2026.00041].

Availability:
This version is available at: 11583/3016773 since: 2026-10-20T08:32:24Z

Publisher:
Euromicro

Published
DOI:10.23919/DSDCPS00089.2026.00041

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

18 November 2026

A Multi-Dimensional Design Space Exploration of ChaCha20 on FPGA

Enrico Manfredi[†], Valeria Piscopo[†], Francesco Antognazza[*], Alessandra Dolmeta[†], Mattia Mirigaldi[†], Maurizio Martina[†], Guido Masera[†], Gerardo Pelosi[‡], and Alessandro Barenghi[‡]
[†]Department of Electronics and Telecommunications, Politecnico di Torino, Turin, Italy
[*]Department of Computer Science, University of California, Irvine, Irvine (CA), USA
[‡]Department of Electronics, Information and Bioengineering, Politecnico di Milano, Milan, Italy

***Abstract*—The ChaCha20 stream cipher has emerged as a cornerstone of modern network security, particularly within the TLS 1.3 and WireGuard protocols, due to its high performance and inherent resilience against timing-based side-channel attacks. Addressing the lack of holistic trade-off analysis between throughput and area, this paper presents a comprehensive Design Space Exploration (DSE). We propose a modular, highly parameterized FPGA architecture that allows for the systematic modulation of three key dimensions: the level of algorithmic parallelism, the temporal partitioning of operations, and the granularity of the arithmetic datapath. Our experimental analysis on a Xilinx Artix-7 demonstrates a vast performance spectrum, ranging from resource-constrained 8-bit serialized designs to massively unrolled 32-bit architectures. A key finding of this exploration is that extreme datapath slicing (e.g., 8-bit) is counterproductive on modern FPGAs, as the control overhead outweighs the logic savings. Conversely, we identify a sweet-spot configuration that achieves over 2.3 Gb/s while requiring fewer than 700 equivalent slices, corresponding to the best overall efficiency with an area–time product of only 140.33 eSlice $\times \mu$s. The exploration further establishes a minimum-area implementation (507 eSlices) and a minimum-latency architecture (0.059 μs), providing practical guidelines for both resource-constrained IoT and high-speed networking applications.**

***Index Terms*—ChaCha20, Stream Cipher, Lightweight cryptography and Implementations, FPGA.**

I. Introduction

The transition toward a *zero-trust* security model and the widespread adoption of the Transport Layer Security (TLS) 1.3 protocol have fundamentally reshaped the requirements for symmetric cryptography [1]. To eliminate legacy vulnerabilities, modern protocols have shifted exclusively toward Authenticated Encryption with Associated Data (AEAD) schemes. Within this framework, the ChaCha20 stream cipher serves as the primary alternative to AES, particularly optimized for environments where dedicated hardware acceleration is absent [2], [3]. Unlike AES, which relies on data-dependent lookups (S-Boxes) that are prone to cache-timing attacks, ChaCha20 is built on the Addition-Rotation-XOR (ARX) design philosophy. This structure allows the algorithm to execute in constant time and provides exceptional efficiency across both software and hardware fabrics. As ChaCha20 is increasingly deployed in everything from ultra-low-power IoT sensors to high-bandwidth VPN gateways, the need for hardware implementations that can adapt to varying area and power constraints is paramount. Current literature regarding ChaCha20 hardware implementations is mainly characterized by specialized architectures optimized for specific, isolated metrics, most notably peak throughput or minimal resource utilization. While these bespoke designs establish high-performance benchmarks, they frequently function as localized optima that offer limited insight into the interdependent effects of varying architectural parameters across a continuous design space.

Indeed, the high degree of potential reconfigurability in ChaCha20 hardware stems from the algorithm's repetitive and hierarchical structure. The top-level block function orchestrates eighty repetitive arithmetic transformations, labeled as Quarter-Rounds (QR), that possess varying degrees of spatial independence. Each QR consists of a fixed sequence of arithmetic steps, and this inherent modularity provides natural *cut-points* for architectural optimization. By leveraging these points, designers can trade off spatial parallelism against temporal serialization and modulate the bit-level granularity of the arithmetic units. However, the absence of a systematic mapping of these trade-offs deprives designers of a data-driven framework for diverse deployments, often leading to suboptimal resource allocation where hidden control overheads can yield designs less efficient than native implementations.

Contributions. The primary contribution of this work is a multi-dimensional hardware Design Space Exploration (DSE) of the ChaCha20 cipher. We provide a systematic analysis of the trade-offs between area, latency, and throughput by modulating the internal datapath and control logic.

Specifically, we evaluate the impact of:

- **Algorithmic Parallelism**: Scaling the number of concurrent Quarter Round instances from 1 to 80 to explore the limits of throughput.
- **Temporal Partitioning**: Adjusting the pipeline depth of the core logic to modulate the critical path and operating frequency.
- **Arithmetic Granularity**: Evaluating the efficiency of 8-bit, 16-bit, and native 32-bit datapath widths to determine the impact of serialization on resource utilization.

Our study utilizes a standardized Equivalent Slice (eSlice) metric to compare resource costs across diverse configurations, demonstrating that native 32-bit operations consistently outperform artificial slicing on modern FPGA fabrics. The

architecture is implemented as a standalone accelerator with full synthesis-time configurability, offering a flexible building block for integration into various environments and showing different performance and area profiles.

Organization. The remainder of this paper is organized as follows: in Sec. II, Subsec. II-A details the ChaCha20 algorithm, while Subsec. II-B reviews existing hardware implementations. In Sec. III, the proposed modular architecture and control logic are presented, and Sec. IV analyzes the design space exploration and area-throughput trade-offs. Finally, Sec. V summarizes our findings.

II. Background

This section provides a technical overview of the ChaCha20 algorithm (Subsec. II-A) and a comprehensive summary of prior implementation strategies in the literature (Subsec. II-B).

A. ChaCha Stream Cipher

ChaCha20 was introduced by Daniel J. Bernstein as a refinement of the Salsa20 cipher, specifically designed to increase data diffusion per round without increasing computational complexity [4]. Based on the Addition-Rotation-XOR (ARX) design philosophy, the cipher avoids data-dependent memory lookups or branching, ensuring constant-time execution that neutralizes timing-based side-channel attacks. The core data structure of ChaCha is a 4×4 matrix of 32-bit elements, commonly referred to as the *state*, as shown in Tab. I.

TABLE I
ChaCha Initial State Matrix.

constant[0]	*constant*[1]	*constant*[2]	*constant*[3]
key[0]	*key*[1]	*key*[2]	*key*[3]
key[4]	*key*[5]	*key*[6]	*key*[7]
counter[0]	*nonce*[0]	*nonce*[1]	*nonce*[2]

To populate this matrix, the first four 32-bit words are assigned to the constant value of the ASCII-encoded text *expand 32-byte k*, followed by eight words representing the 256-bit secret key. This large key size provides a robust 128-bit security margin in post-quantum landscapes against Grover's algorithm. The remaining slots are filled with a 32-bit block counter and a 96-bit nonce, aligning with the IETF standardization in RFC 7539 [5] and RFC 8439 [2]. Together, these 16 words establish the initial *state* of the cipher.

At the heart of the cryptographic process lies the **Quarter-Round (QR)** primitive. As detailed in Alg. 1, the QR function receives four independent 32-bit words, denoted as a, b, c, and d, and sequentially modifies them through four successive ARX steps. In a hardware context, this sequential logic can be temporally partitioned to modulate the critical path, an architectural choice governed by the **QRS** (QR Stages) parameter. Furthermore, while the algorithm natively operates on 32-bit words, the implementation of the operations can follow different strategies. For example, the underlying arithmetic granularity of the adders can be adjusted via the **AW** (Adder Width) parameter to balance silicon area and cycle count.

Algorithm 1 Quarter Round (QR) function

Require: a, b, c, d
1: $a \leftarrow a + b; d \leftarrow a \oplus d; d \leftarrow d \lll 16;$
2: $c \leftarrow c + d; b \leftarrow b \oplus c; b \leftarrow b \lll 12;$
3: $a \leftarrow a + b; d \leftarrow a \oplus d; d \leftarrow d \lll 8;$
4: $c \leftarrow c + d; b \leftarrow b \oplus c; b \leftarrow b \lll 7;$

Building upon this primitive, the main ChaCha block function iterates over the *state* for 20 rounds, totaling 80 QR operations. Each round comprises four QR executions, systematically alternating between the matrix columns (odd rounds) and diagonals (even rounds). The degree of spatial parallelism employed to process these operations, ranging from serial execution to full unrolling, is controlled by the **PQR** (Parallel QR) parameter.
The complete logic for this multi-round transformation is described in Alg. 2 and depicted in Fig. 1.

Algorithm 2 ChaCha block function

Require: $in[16]$
Ensure: $out[16]$
1: $x \leftarrow in$
2: **for** $i = 0$ **to** 15 **do**
3: $\quad x[i] \leftarrow in[i]$
4: **end for**
5: **for** $i = 1$ **to** 10 **do**
6: $\quad \text{QR}(x[0], x[4], x[8], x[12])$ ▷ Column round
7: $\quad \text{QR}(x[1], x[5], x[9], x[13])$
8: $\quad \text{QR}(x[2], x[6], x[10], x[14])$
9: $\quad \text{QR}(x[3], x[7], x[11], x[15])$
10: $\quad \text{QR}(x[0], x[5], x[10], x[15])$ ▷ Diagonal round
11: $\quad \text{QR}(x[1], x[6], x[11], x[12])$
12: $\quad \text{QR}(x[2], x[7], x[8], x[13])$
13: $\quad \text{QR}(x[3], x[4], x[9], x[14])$
14: **end for**
15: **for** $i = 0$ **to** 15 **do**
16: $\quad out[i] \leftarrow x[i] + in[i]$
17: **end for**

The standard ChaCha20 configuration performs 20 rounds to ensure a high security margin against cryptanalysis. Upon concluding the last round, the algorithm computes a feed-forward addition of the transformed matrix with the original initial *state*, yielding a 16-word output block. This final addition is a critical non-reversible step that prevents the algorithm from being trivially inverted.

To encrypt a message of arbitrary length, the cipher relies on the 256-bit key, the 32-bit starting counter, and the 96-bit nonce. It repeatedly invokes the block function, incrementing the block counter value for each new invocation while keeping the key and nonce static. The continuous concatenation of the resulting output blocks produces the cryptographic keystream. The actual encryption is achieved by performing an XOR operation between this keystream and the incoming plaintext, discarding any surplus keystream bytes at the end. Because

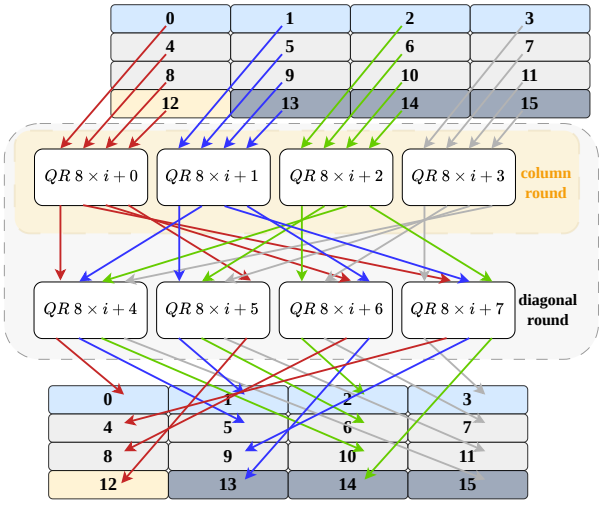


Fig. 1. Composition of column and diagonal rounds in ChaCha.

the operation is symmetric, the decryption phase perfectly mirrors encryption, merely substituting the plaintext with the received ciphertext. Crucially, to maintain cryptographic integrity, the employed nonce must remain strictly unique for every encryption performed under a given key.

B. Related works

The architectural simplicity of ChaCha20 has inspired a wide variety of hardware implementations, ranging from high-speed network accelerators to resource-constrained IoT nodes. However, as identified in the literature, most existing works focus on specialized, point-solution designs rather than a comprehensive exploration of the continuous trade-off space. The following review categorizes these efforts by their architectural priorities and target platforms.

The first ChaCha20 VLSI hardware implementation using a 0.18 μm CMOS technology appeared in [6]. This work explores two architectural dimensions: the level of parallelism, by computing 1, 4, or 8 QR simultaneously, and the operation granularity, allowing for either a full QR or a single ARX step to be executed per clock cycle to modulate the critical path.
In [7] the authors produced a compact co-processor for the Virtex-6 FPGA implementing the ARX logic in three different pipeline configurations. The resulting arithmetic unit is interfacing with a register file to access the ChaCha20 state, to compute the four independent QRs in parallel. Optionally, the 32-bit addition is computed in two cycles to reduce the critical path. Leveraging the algorithmic similarities between the BLAKE hash function and ChaCha20, the authors also presented an unified arithmetic unit compatible with both cryptographic schemes, able to execute two parallel ChaCha20 keystream generations. Similarly, [8] provides a flexible reconfigurable architecture supporting SHA256, BLAKE256, BLAKE2s, and ChaCha20 cryotographic schemes in a single design, discussing the results for the Virtex-5 and Virtex-6 FPGAs along with the synthesis results with the FreePDK45 and the Nangate open cell library.

A detailed analysis of hardware architectures for the ChaCha family of stream ciphers, specifically optimized for high-throughput FPGA implementations, is produced in [9]. This work provides three distinct hardware architectures: pipeline-based, block memory-based, and register-based implementations. Notably, the pipeline architecture shows a 8× improvement in the throughput per area metric compared to the best AES implementation available. [10] presents a hardware-software co-design for ChaCha20 on a Cyclone V SoC, utilizing a non-pipelined architecture that computes 4 parallel QR each clock cycle. To optimize resource utilization, the implementation partitions the cipher by offloading the final summation and XOR stages to the CPU, focusing hardware resources solely on the core round functions.

Various works focused on the hardware implementation of the ChaCha20-Poly1305 Authenticated Encryption with Associated Data (AEAD) [11]–[14] standardized in RFC 7539 [2], [5] by the Crypto Forum Research Group (CFRG) of the Internet Engineering Task Force (IETF) organization. Notably, ChaCha20-Poly1305 and AES-GCM are the only AEADs in the allowed cipher suites defined in the TLS 1.3 protocol [15]. Two commercial hardware Intellectual Properties (IPs) for the ChaCha20-Poly1305 AEAD are provided by [13], [14].
In [11] a ChaCha20 hardware accelerator processing 4 QR in parallel is implemented onto a Zynq-7000 FPGA, with the goal of accelerating the keystream generation for the TLS handshake performed by the Caddy web server executed by the ARM core in the same SoC. Similarly, [12] uses 4 parallel QR units in the ChaCha20 core of their AEAD unit, providing results for a Virtex-7 FPGA for a standalone unit or as an accelerator integrated into the Rocket RISC-V SoC. The same design is also extracted and evaluated on an Artix-7 in their later work [16]. Targeting the same Rocket SoC, the authors in [17] propose three variants of a RISC-V ISA extension for resource-constrained devices without large vector engines, either minimizing the number of executed instructions, maximizing the resource reuse, or performing two parallel QR operations in parallel.

Two works focused on error detection architectures in the ChaCha20 primitive [18], [19]. The former work provides three mechanisms to detect faults by using complementary inverse operations to verify the quarter-round (QR) function, time-redundant recomputation with rotated operands to identify permanent hardware defects, and dual-rail self-checking adders to ensure the integrity of modular arithmetic through signal complementarity. Results are provided for an ASIC synthesis using a TSMC 65 nm standard cell library. The latter work provides a fault-tolerant version of ChaCha20 using Hamming Error Correction Code (ECC) for the datapath logic and triple redundancy for the controller logic. Furthermore, two unprotected designs with 1 or 8 parallel QR are provided as reference, and evaluated onto a Zynq-7000 FPGA.
In [20] authors employ a single QR unit with only two ARX units to reduce resource usage. Notably, they use a sparse parallel prefix adder to perform the 32-bit additions to improve the performance, and provide results with a 0.18 μm ASIC

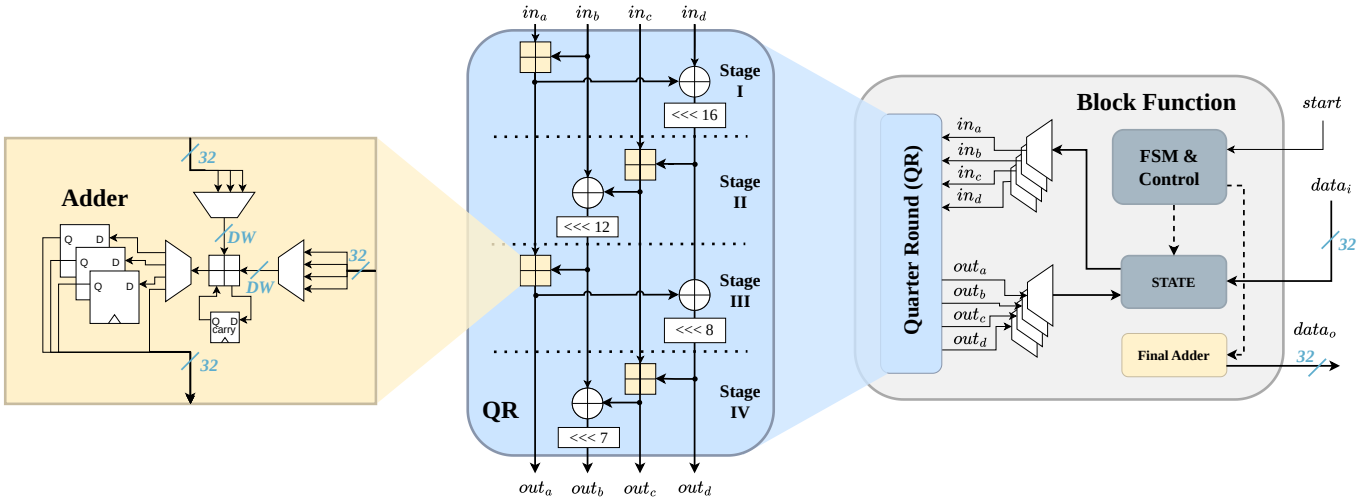


Fig. 2. Hierarchical hardware architecture of the ChaCha core, illustrating a top-down drill-down from the system level to individual arithmetic components.

technology node and the Virtex-7 FPGA.

III. ChaCha20 Design

This section details the architectural implementations and the systematic exploration of the ChaCha stream cipher. First, Subsec. III-A establishes the baseline hardware architecture, defining the core hierarchical structure and the fundamental quarter-round modules. Building upon this baseline, Subsec. III-B investigates the broader hardware design space by adjusting key architectural parameters to evaluate the trade-offs between computational latency and hardware resources.

A. Fundamental Design

The ChaCha hardware is organized as a hierarchical architecture that separates interface handling, state orchestration, and core arithmetic computation. This modularity ensures a stable external interface, facilitating seamless integration into diverse host systems and FPGA-based SOC environments. Simultaneously, this approach preserves the internal flexibility necessary to explore the continuous area-throughput trade-off space by modulating the core's architectural parameters: Parallel Quarter Round (**PQR**), Quarter Round Stages (**QRS**), and Adder Width (**AW**).

At the architectural level, the top wrapper receives the initial 512-bit state and coordinates data movement between host-facing registers and the cryptographic datapath. The core computation is delegated to the Block Function, while arithmetic is implemented by reusable Quarter-Round (QR), adder, and XOR-bank building blocks.

As shown in Fig. 2, the **Block Function**, depicted on the right side, orchestrates the 16-word state matrix. On `start`, the input state is copied into both a working state and a initial state. Then, the scheduler executes 80 quarter-round operations (20 rounds $\times$ 4 quarter-rounds), alternating column and diagonal index patterns to match the ChaCha round structure described in Alg. 2. The parameter PQR controls how many quarter-round computations are issued concurrently at each scheduling step. After all quarter-round operations are complete, 16 adders run in parallel to implement the ChaCha feed-forward (word-wise sum between the transformed and the initial state). The resulting 512-bit vector is the keystream.

The **Quarter-Round** (QR) module, central block in Fig. 2, implements the ARX primitive over four 32-bit lanes (a, b, c, d) with fixed rotations of 16, 12, 8, and 7 bits (Alg. 1). Its micro-architecture is controlled by two parameters:

- QRS: temporal partitioning of the four ARX operations (supported values in this work: 1, 2, 4);
- AW: arithmetic slice width used inside the adder (supported values in this work: 32, 16, 8 bits).

When the QR module is in the single stage configuration, an instance of an adder, a bit-wise XOR, and a rotation unit is added to the design for each one of the four ARX operations. When QRS is set to 2 (respectively, 4), only 2 (respectively, 1) ARX operations are performed each clock cycle, and the adder/XOR/rotation units are reused. In hardware fabrics like FPGAs or ASICs, the rotation by a fixed number of positions is realized through wire re-routing, without gate evaluation delays incurred. However, when QRS > 1, we still need to select the correct wire permutation via a multiplexer.

The adder module, left block in Fig. 2, is configurable in terms of its datapath width to reduce the area and the depth of the critical path. When AW $= 32$, the operation is performed in a single clock cycle. For AW $= 16$ or 8, additions are split across multiple sub-operations, taking care to propagate the carry output between each sub-operation. The micro-architecture implementing the adder is left to the synthesis tool.

With QRS $= 1$ and AW $= 32$, a QR transformation is completed in the shortest control sequence, but with the longest combinational path. The increase of QRS and/or the reduction of AW serializes the computation, reducing local

combinational depth and arithmetic footprint at the expense of additional cycles and control overhead.

The scheduler supports multiple PQR configurations, constrained to values that divide the 80 quarter-round operations. In practice, the explored set includes {1, 2, 4, 8, 16, 40, 80}. Small PQR values reuse fewer QR instances over more iterations (area-oriented operation), while larger values unroll more quarter-round work in time (throughput-oriented operation).

Internally, quarter-round instances are arranged in groups of four operations per ChaCha round step, with additional layers when PQR exceeds four. This layered organization allows the same control structure to represent sequential, partially unrolled, and highly unrolled datapaths with a common state-update interface. For PQR < 8, each QR instance needs to address multiple sections of the state matrix, requiring multiplexers and de-multiplexers at the a, b, c, and d input/output ports. When PQR ≥ 8, each instantiated QR module will be exclusively part of a column or a diagonal round, thus having fixed links to the internal matrix state or to other QR input/output ports. This optimized design micro-architecture is, however, not compatible when 20 QR modules are instantiated, as each QR unit will alternate the computation of a column or a diagonal round, thus we decided to exclude this configuration from the explored ones.

B. Design Space Exploration

The ChaCha hardware architecture enables a continuous trade-off between resource utilization and throughput by modulating the three primary parameters defined in Subsec. III-A: PQR, QRS, and AW. The results of this exploration, visualized in the latency–area plane in Fig. 3, reveal a Pareto frontier ranging from ultra-compact sequential engines to high-performance parallel arrays.

Our design architecture supports a broad design space; however, not all parameter combinations yield viable or efficient hardware. Certain configurations are strictly dominated by others and have been excluded from this analysis to focus on the Pareto-optimal frontier. Specifically, when PQR exceeds 4, configurations utilizing QRS $\neq 1$ or AW $\neq 32$ result in degraded latency, area, and efficiency. This performance drop stems from the inherent data dependencies between column and diagonal rounds: in these serialized setups, only four QR units can effectively run in parallel, leaving additional hardware resources idle while waiting for data propagation in subsequent clock cycles. While computing multiple independent ChaCha keystreams in parallel could resolve this bottleneck, it would require significant area overhead to store multiple state matrices, a trade-off that falls outside the scope of this single-block optimization.

By modulating the internal structure of the state orchestration and arithmetic modules, the design can be tuned to prioritize either high throughput, minimal resource utilization, or higher computational efficiency. The impact of each parameter is summarized below:

- **PQR** (Parallel QR): As PQR increases from 1 to 80, the design transitions from a resource-oriented sequential region to a throughput-oriented parallel region. High PQR values drastically reduce cycle counts, processing a block in as few as 3 clock cycles, but increase the complexity of the interconnection logic and the total eSlice count.
- **QRS** (QR Stages): Increasing QRS from 1 up to 4 allows for the reuse of arithmetic units within a single QR module. As described in Subsec. III-A, this reduces the number of instantiated adders and XOR banks, lowering the area at the cost of increased temporal serialization.
- **AW** (Adder Width): Reducing the AW from 32 down to 16 or 8 bits further shrinks the arithmetic footprint. While this lowers the critical path delay, it requires multiple clock cycles to complete a single 32-bit addition, shifting the design toward the high-latency region.

The execution latency scales predictably with these serialization factors. In the maximum-throughput configuration PQR80_QRS1_AW32 the block function computation is completed in just 1 clock cycle. Reducing the PQR to 1 introduces a $80\times$ time penalty; using QRS $= 4$ multiplies the required cycles by another factor of 4; and restricting AW to 8 bits increases the cycle count by another $\times 4$ factor. Consequently, the fully serialized PQR1_QRS4_AW8 design requires a total of $\approx 80 \times 4 \times 4 = 3,840$ clock cycles.

An interesting configuration is the PQR4_QRS4_AW32. While it shares the cycle count and area profile of a PQR1_QRS1_AW32 setup, it benefits from a substantially shorter critical path due to finer ARX partitioning, despite the slight overhead of rotation-selection multiplexers.

IV. Experimental evaluation

The correctness of our ChaCha design is validated both through extensive behavioral simulations matched against the pycryptodome reference implementation and the execution on the ChipWhisperer CW305 Artix-7 FPGA board.For the experimental campaign, we employed Vivado 2019.2 using the standard synthesis, and performed out-of-context synthesis to gather the resource utilization before place-and-route.

We then converted the heterogeneous set of resources resulting from the synthesis into a synthetic eSlice area metric, refining the formulas from [21]:

$$\mathsf{LUT}_{\mathrm{eq}} = \mathsf{LUT} + 880{\cdot}\mathsf{BRAM36E1} + 714{\cdot}\mathsf{DSP48E1} + 3{\cdot}\mathsf{CARRY4}$$

$$\mathsf{FF}_{\mathrm{eq}} = \mathsf{FF} + 548{\cdot}\mathsf{BRAM36E1} + 230{\cdot}\mathsf{DSP48E1}$$

$$\mathrm{eSlice} = \max\left(\left\lceil \frac{\mathsf{LUT}_{\mathrm{eq}}}{4} \right\rceil, \left\lceil \frac{\mathsf{FF}_{\mathrm{eq}}}{8} \right\rceil\right)$$

Specifically, [21] converts the DSP48E1 realizing an integer multiply and accumulate with pre-addition and BRAM36E1 in a Simple Dual Port RAM configuration into a set of LUT and FF resources. In addition, we provide two key improvements:

- We impeded the synthesis engine from using FPGA-specific MUX7, MUX8, or MUX9 primitives by setting `BLOCK_SYNTH.MUXF_MAPPING` to 0 in the XDC file.
- We converted the CARRY4 primitive into LUTs by setting `BLOCK_SYNTH.ADDER_THRESHOLD` to 128 in the XDC file when implementing a 8-bit incrementer.

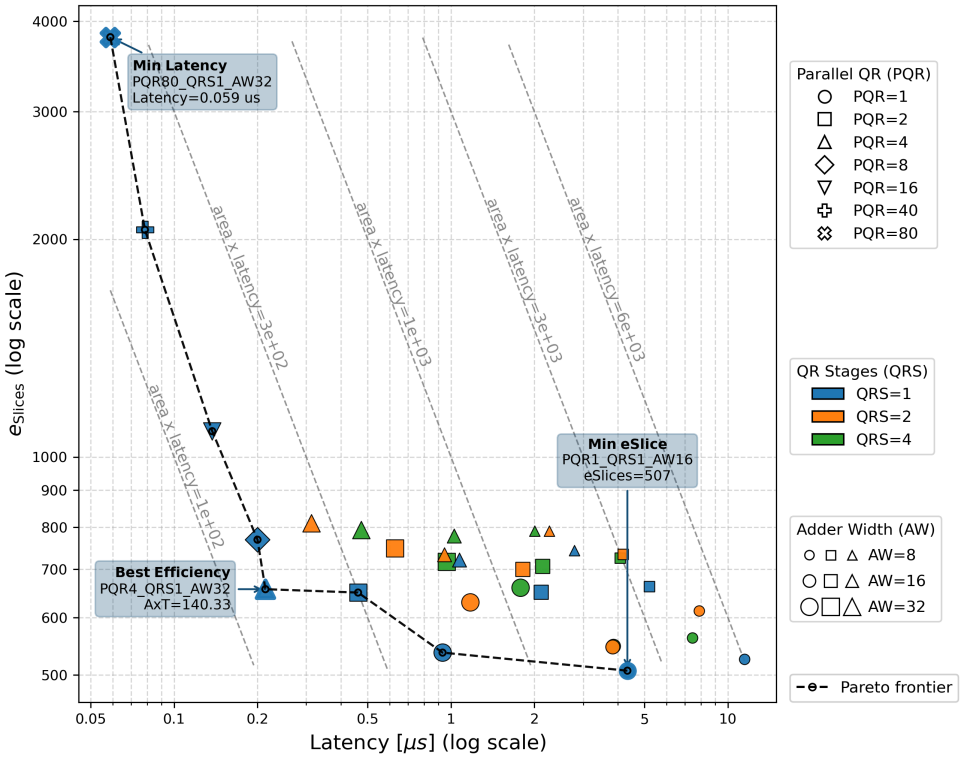


Fig. 3. Latency versus area (eSlices) trade-off for the explored ChaCha hardware configurations. Both axes are plotted on a logarithmic scale. The dashed lines represent constant Area-Time ($A \times T$) product contours, expressed in eSlices $\times$ μs. The visual encoding of the data points maps the architectural parameters as follows: distinct marker shapes represent different Parallel QR (PQR) values, different colors indicate the number of QR Stages (QRS), and the relative size of the markers reflects the Adder Width (AW). The three annotated markers highlight the primary corner cases of the design space: minimum latency (pink), best overall hardware efficiency (green), and minimum area footprint (blue).

Although the specific carry-to-LUT conversion was not strictly critical for this study, we include this mechanism as a general-purpose approach to enhance the accuracy of area estimation for carry-intensive logic, which may prove valuable in other architectural contexts. To foster transparency and facilitate further research, the scripts and synthesis constraints used for our area evaluation are available open-source[1].

The exact cycle count for a single ChaCha20 execution is retrieved during the behavioral simulation managed by cocotb v1.9.2 using Verilator v5.034 simulation tool. We derived the maximum working frequency of each design via a binary search algorithm of the most strict clock period constraint resulting in a successful out-of-context synthesis. Note that most modern synthesis tools, including Vivado, employ rather sophisticated routing delay estimation model during the synthesis timing analysis. Therefore, the result of the synthesis timing analysis highly correlates to the post-implementation result when there are not significant routing congestions. Finally, we derive the efficiency indicator by multiplying the clock cycles latency by the eSlice area.

A. Design Space Exploration results

The design space exploration of the ChaCha hardware architecture yields a diverse set of trade-offs between computational latency and resource utilization, as illustrated by the scatter plot in Fig. 3. By evaluating different combinations of the architectural parameters across the latency–area plane, we can identify a Pareto frontier that highlights the most optimal configurations for various design goals.

Hardware Corner Cases. Analysis of the implementation data identifies three distinct architectural corner cases, each optimizing a specific performance metric:

- **Minimum Latency** (**Min Latency**): fully unrolling the computation of 80 parallel QR units and a full 32-bit datapath, the PQR80_QRS1_AW32 configuration minimizes the processing time to 0.059 μs. This represents the upper-left limit of the design space, prioritizing speed at an area cost of approximately 3,800 eSlices.
- **Best Efficiency** (**Best Efficiency**): Represented by the PQR4_QRS1_AW32 configuration. This design strikes the optimal balance between throughput and resource consumption, achieving the lowest Area-Time product ($A \times T = 140.33$). It demonstrates that moderate parallelism often yields the highest hardware-efficiency.
- **Minimum Area** (**Min eSlice**): the most compact solution is represented by the PQR1_QRS1_AW16 configuration, with a hardware footprint of just 507 eSlices. By reducing parallel units and narrowing the arithmetic datapath, this setup is ideal for area-constrained SoCs

[1]https://doi.org/10.5281/zenodo.21249196

TABLE II
COMPREHENSIVE SUMMARY OF STATE-OF-THE-ART HARDWARE IMPLEMENTATIONS OF THE CHACHA20 CIPHER.

Work	Target	Configuration			Latency	Area					Freq.	Throughput	Efficiency
		PQR	QRS	AW	(cc)	LUT	FF	BRAM	DSP	eSlice	(MHz)	(b/cc)	(eSlice×kCC)
[8]	Virtex-5	N/A	N/A	N/A	10†	N/A	N/A	0	0	3816*	120	51.2	3.81*
	Virtex-6									3503*	135	53.1	3.50*
[7]	Virtex-6	4	8	32	701†	N/A	N/A	2	0	489*	362	0.73	342*
				16						517*	316		362*
[20]	Virtex-5	1	2	32	164	N/A	N/A	0	0	431*	407	3.12	69.2*
	Virtex-6									424*	487		
	Virtex-7					1686	640			422	575		
[12]	Virtex-7	4	1	32	22†	2288	1058	0	0	572	166	23.2	12.5
[9]	Virtex-7	4	4	32	86†	1693	1502	0	20	4004	312	5.96	344
					86†	2369	2152		0	598	393	5.96	51.4
		1	4	32	86†	862	1162	2	8	2083	275	5.96	179
					86†	1366	1831		0	786	375	5.96	67.5
		80	4	32	3†	4556	13484	0	144	27112	356	190	81.3
					3†	9138	18004		0	2284	368	190	6.85
[19]	Zynq-7000	8	1	32	12	2676	518	0	0	669	52.6	42.7	8.02
		1			103	1562	1044			391	81.8	5.01	40.2
[11]	Zynq-7000	4	4	32	83	3064	2604	0	0	766	115	N/A	63.5
[16]	Artix-7	4	1	32	22	2416	1026	0	0	604	N/A	23.2	13.2
Our	Artix-7	80	1	32	3	15210	1046	0	0	3803	51	170	11.4
		4	1	32	22	2625	1064			657	103	23.2	14.4
		1	1	16	643	2027	1439			507	93	1.27	326

N/A: not applicable, or data not available in the original publication.
*: the value contains the Slice count reported in the original publication, which is slightly different from the eSlice metric.
†: the exact value is unavailable, we computed it as $512/\text{bits-per-cycle}$.
We selected from [9] only the pipeline depth configurations which generally performed better in our area and efficiency metrics.
The eSlice formula is not compatible with Virtex-5. Nonetheless, we could not apply it as the authors only provided the Slice count.

where throughput is secondary to resource savings.

B. Comparison with state of the art

In Tab. II, we report the area, latency, efficiency, and throughput results of other ChaCha20 implementations currently available in the literature, which we compare against our three designs selected from the DSE. Where performance metrics were not explicitly stated, we estimated them from the available data in the respective publications. For instance, [9] provides results for the ChaCha8 algorithm, along with a conversion rule to provide ChaCha20 approximate performance. Similarly, while [12] implements a full ChaCha20-Poly1305 AEAD scheme, we isolated the ChaCha20 metrics based on their assessment in the related work [16].

While all works provide a different description of their architectures, most of them can be categorized using our methodology. A notable exception is the Reconfigurable Crypto Core from [8], which provides a unified architecture to implement ChaCha20, BLAKE256, BLAKE2s, and SHA256.

To provide a comprehensive overview, Fig. 4 visualizes the latency versus resource usage (in eSlices) for all works discussed, illustrating the position of our implementations within the broader design space. Given the variety of target FPGAs, we focused on platform-independent results to facilitate a fair comparison of the computing architectures. In this context, we expressed the latency in clock cycles (cc), throughput in bits per cycle (b/cc), area in eSlices, and efficiency in eSlice×cc. Note that the working frequency can only be compared against other works employing the exact same FPGA target.

Note that the eSlice metric could not be derived when LUT and FF counts are not reported, like in [7], [8], [20]. This fact also contributes to their higher efficiency; we mark the affected entries with an asterisk (*) in Tab. II.

Both [8] and [9] utilize deeply pipelined designs that require processing interleaved ChaCha20 keystreams to mitigate data dependencies between column and diagonal rounds. Consequently, their clock latency figures represent the amortized cost of processing a single keystream once the pipeline is saturated.

Our PQR4_QRS1_AW32 configuration has similar results of [12], [16], with the small overhead due to the versatility of our design in supporting multiple micro-architectures at syn-

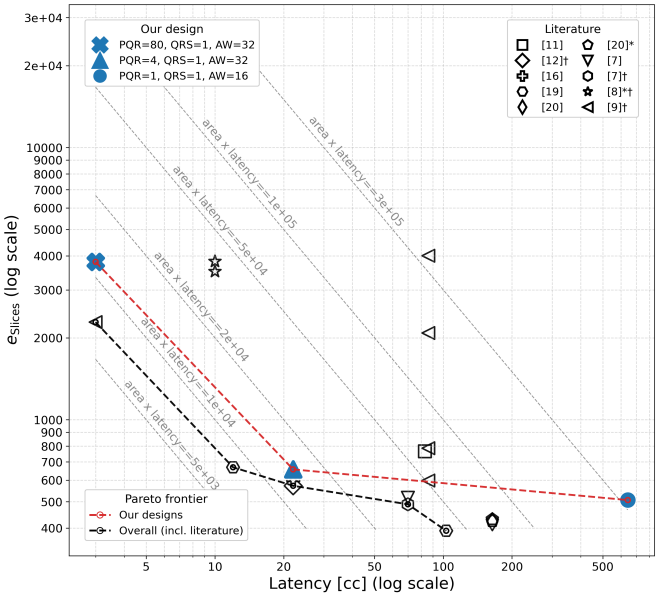


Fig. 4. Latency (x) vs eSlices (y) SoA comparison.

thesis time. An interesting observation arises when comparing it against the PQR8_QRS1_AW32 configuration from [19]. While our design takes approximately twice the clock cycle count, it has a comparable area and simultaneously sustains a 2× higher clock frequency. As both designs are evaluated on identical architectures and 28nm TSMC technology nodes, this frequency scaling fully compensates for the higher cycle latency. While the efficiency computed via the eSlices × μs product is identical, we keep using the eSlices × cc formula to ensure a unified, frequency and technology independent comparison across all the evaluated AMD FPGA families.

Our PQR80_QRS1_AW32 configuration is the fastest design to produce a single keystream block, recalling that [9] interleaves independent keystream computations to solve data dependencies and Tab. II contains the amortized latency.

V. Conclusion

In this paper, we presented a comprehensive design space exploration of the ChaCha20 cipher on FPGA, providing a data-driven framework to navigate the trade-offs between throughput, latency, and hardware resources. By systematically modulating architectural parameters, specifically algorithmic parallelism, temporal partitioning, and arithmetic granularity, we identified a Pareto-optimal frontier that serves as a guide for system architects. Our experimental results, validated on Artix-7, highlighted specific configurations optimized for minimum latency, maximum hardware efficiency, and minimum area. These findings demonstrate that, contrary to some conventional design assumptions, native datapath widths often outperform aggressively sliced architectures, while moderate parallelism proves most effective for general-purpose applications. Ultimately, this work provides a versatile toolkit for tailoring ChaCha20 implementations to meet the strict constraints of diverse environments, from resource-limited IoT sensors to high-performance networking gateways.

References

[1] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3." RFC 8446, Aug. 2018.

[2] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols." RFC 8439, June 2018.

[3] F. D. Santis, A. Schauer, and G. Sigl, "ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017* (D. Atienza and G. D. Natale, eds.), pp. 692–697, IEEE, 2017.

[4] D. J. Bernstein, "ChaCha, a variant of Salsa20," in *SASC 2008 - The State of the Art of Stream Ciphers*, vol. 8, pp. 3–5, 2008.

[5] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols." RFC 7539, May 2015.

[6] L. Henzen, F. Carbognani, N. Felber, and W. Fichtner, "VLSI hardware evaluation of the stream ciphers Salsa20 and ChaCha, and the compression function Rumba," in *2008 2nd International Conference on Signals, Circuits and Systems*, pp. 1–5, 2008.

[7] N. At, J. Beuchat, E. Okamoto, I. San, and T. Yamazaki, "Compact hardware implementations of ChaCha, BLAKE, Threefish, and Skein on FPGA," *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 61-I, no. 2, pp. 485–498, 2014.

[8] V. T. D. Le, H. L. Pham, T. H. Tran, T. S. Duong, and Y. Nakashima, "High-efficiency reconfigurable crypto accelerator utilizing innovative resource sharing and parallel processing," in *16th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSoC 2023, Singapore, December 18-21, 2023*, pp. 576–583, IEEE, 2023.

[9] J. Pfau, M. Reuter, T. Harbaum, K. Hofmann, and J. Becker, "A Hardware Perspective on the ChaCha Ciphers: Scalable Chacha8/12/20 Implementations Ranging from 476 Slices to Bitrates of 175 Gbit/s," in *32nd IEEE International System-on-Chip Conference, SOCC 2019, Singapore, September 3-6, 2019*, pp. 294–299, IEEE, 2019.

[10] I. Semenov, "An implementation of ChaCha20 stream cypher in all-programmable SoCs," *Theses*, Jan. 2020.

[11] M. Cuppens, "Implementation and Analysis of a ChaCha Hardware Accelerator for the Caddy Secure Webserver," Master's thesis, KU Leuven, 2017.

[12] R. Serrano, C. Duran, M. Sarmiento, C. Pham, and T. Hoang, "ChaCha20-Poly1305 Authenticated Encryption with Additional Data for Transport Layer Security 1.3," *Cryptogr.*, vol. 6, no. 2, p. 30, 2022.

[13] Silex Insight, "ChaCha20-Poly1305 Crypto Engine." Silex Insight, acquired by Secure-IC, acquired by Cadence.

[14] Xiphera Ltd., "ChaCha20-Poly1305."

[15] A. Langley, W.-T. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson, "ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)." RFC 7905, June 2016.

[16] R. Serrano, C. Duran, M. Sarmiento, T. Dang, T. Hoang, and C. Pham, "A unified PUF and crypto core exploiting the metastability in latches," *Future Internet*, vol. 14, no. 10, p. 298, 2022.

[17] B. Marshall, D. Page, and T. H. Pham, "A lightweight ISE for ChaCha on RISC-V," in *32nd IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2021, Virtual Conference, USA, July 7-9, 2021*, pp. 25–32, IEEE, 2021.

[18] M. M. Kermani, R. Azarderakhsh, and A. Aghaie, "Fault detection architectures for post-quantum cryptographic stateless hash-based secure signatures benchmarked on ASIC," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 2, pp. 59:1–59:19, 2017.

[19] W. Grignani, K. G. Q. de Santana, D. A. dos Santos, L. Dilillo, and D. R. Melo, "Implementation and reliability evaluation of a ChaCha20 stream cipher hardware accelerator," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, Didcot, United Kingdom, October 8-10, 2024*, pp. 1–4, IEEE, 2024.

[20] B. Rashidi, "High-performance hardware structure of ChaCha20 stream cipher based on sparse parallel prefix adder," *Int. J. Circuit Theory Appl.*, vol. 53, no. 5, pp. 2947–2957, 2025.

[21] F. Antognazza, A. Barenghi, and G. Pelosi, "An efficient and unified RTL accelerator design for hqc-128, hqc-192, and HQC-256," *IEEE Trans. Computers*, vol. 74, no. 7, pp. 2306–2320, 2025.