Comparative Study of Keccak SHA-3 Implementations

Subjects: Computer Science, Hardware & Architecture | Engineering, Electrical & Electronic
Contributor: Alessandra Dolmeta, Maurizio Martina, Guido Masera

SHA-3, a pivotal component in modern cryptography, has spawned numerous implementations across diverse platforms and technologies. This text aims to provide valuable insights into selecting and optimizing Keccak SHA-3 implementations. It encompasses an in-depth analysis of hardware, software, and software–hardware (hybrid) solutions. Researchers assess the strengths, weaknesses, and performance metrics of each approach. Critical factors, including computational efficiency, scalability, and flexibility, are evaluated across different use cases. Researchers investigate how each implementation performs in terms of speed and resource utilization. This text aims to improve the knowledge of cryptographic systems, aiding in the informed design and deployment of efficient cryptographic solutions. By providing a comprehensive overview of SHA-3 implementations, it offers a clear understanding of the available options and equips professionals and researchers with the necessary insights to make informed decisions in their cryptographic endeavors.

Keywords: hash function ; SHA-3 ; Keccak ; hardware design ; accelerator ; FPGA ; ASIC ; cryptography ; post-quantum cryptography ; HW/SW co-design

1. Introduction

Open contests have become a preferred method for selecting cryptographic standards in the U.S. and worldwide, beginning with the Advanced Encryption Standard (AES) contest organized by the NIST in 1997–2000. Four typical criteria taken into account in the evaluation of candidates in such contests are security, performance in software, performance in hardware, and flexibility [1]. Security, though crucial, is complex to assess quickly in contests. Hardware performance often serves as a tiebreaker when other criteria fail to declare a clear winner among cryptographic algorithms.

Among the contenders, Keccak, designed by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, stood out for its innovative design and strong security properties, ultimately earning its place as the foundation of SHA-3. This achievement marked a significant milestone in modern cryptography, ensuring robust and efficient hash functions for various security applications. While it currently stands as the leader in resisting recent cryptanalysis attacks and excels in hardware performance, there is a continuous demand for developing an efficient implementation, be it software, e.g., Central Processing Unit (CPU), or hardware, e.g., Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC). Common software implementations on a microcontroller offer high flexibility, but they may not provide the required performance for cryptographic algorithms with high computational demands. Microcontrollers are versatile and programmable, making them suitable for a wide range of applications, but they may struggle with the computational intensity of modern cryptographic algorithms. Moving up the spectrum, an extensible processor, such as an application microprocessor, co-processors (i.e., [2]), or a Digital Signal Processor (DSP), offers more significant performance potential than fixed ones. These processors can be optimized for specific cryptographic operations, providing better throughput and efficiency than a generic microcontroller. However, they are still limited by their general-purpose architecture, which may not match the specialized requirements of specific cryptographic algorithms. A programmable datapath takes the customization a step further by allowing users to design custom hardware accelerators for cryptographic tasks. This approach offers a balance between flexibility and performance. Programmable datapaths enable the efficient execution of cryptographic algorithms through parallel processing and custom hardware instructions (i.e., [3]). Finally, the least flexible but most efficient solution is the hardwired datapath, typically implemented in ASICs (Application-Specific Integrated Circuits). ASICs are designed specifically for a particular cryptographic algorithm or set of algorithms, making them highly efficient in terms of speed and power consumption. However, their lack of flexibility means that any changes or updates to cryptographic algorithms require a new hardware design.

In summary, the choice of implementation approach for cryptographic algorithms depends on the trade-off between flexibility and efficiency, as shown in **Figure 1**. Selecting the most suitable implementation space depends on the specific cryptographic requirements and application constraints.

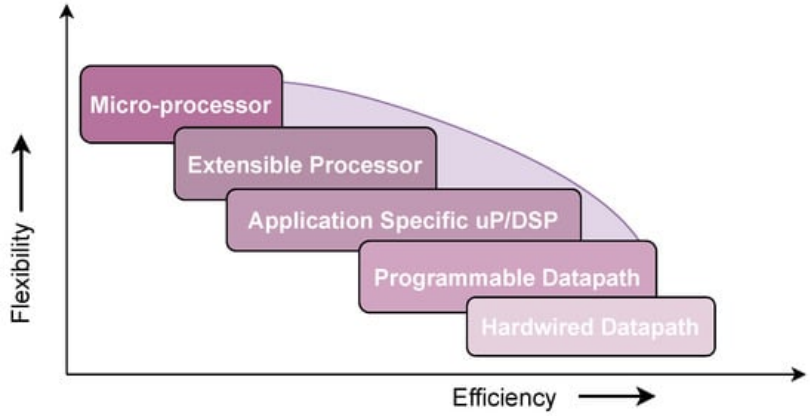

Figure 1. Space of solutions.

2. SHA-3

SHA-3 is a subset of the Keccak family standardized by the NIST. The standard lists four specific instances of SHA-3 and two extendable-output functions (SHAKE128 and SHAKE256). While the SHA-3 functions have a specified output length, the two SHAKE variants permit extraction of a variable length of output data, which makes SHA-3 a suitable candidate for pseudo-random bit generation [4]. All SHA-3 functions operate within a shared foundational framework known as the sponge construction (as shown in **Figure 2**a). This framework is highly adaptable and allows for the generation of hash values with variable length, making it well suited for diverse applications.

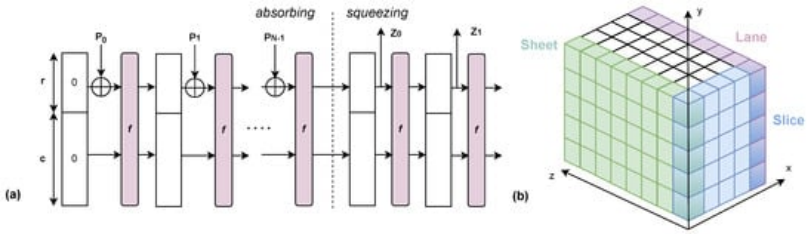

Figure 2. (**a**) Sponge Function. (**b**) Keccak State.

The NIST standard defines four versions of the Keccak sponge function [5] for a message *M* and an output length *d*, as shown in **Table 1**. The algorithm uses two parameters for the sponge construction: the bitrate with *r*-bits, which determines the number of bits absorbed in each step, and the capacity with *c*-bits, which determines the attainable security level (**Figure 2**a).

Table 1. SHA3 instances.

Instance	Output Size *d*	Rate *r*	Capacity *c*
SHA3-224	**224**	**1152**	**448**
SHA3-256	**256**	**1088**	**512**
SHA3-384	**384**	**832**	**768**
SHA3-512	**512**	**576**	**1024**

The flow of a sponge function can be understood through the following steps:

- Initialization: the sponge function is initialized depending on *r* and *c* parameters.
- Padding: The input message is padded to ensure that length is a multiple of *r*. Most of the architectures utilize a software scheduler for preparing the input by splitting and padding long messages into blocks of 1600 bits (multi-block messages) for truncating, if necessary, the output of the hash computation in the appropriate size of the selected mode of operation and for updating the state matrix in the case of multi-block messages. As an example, in [4][6][7][8] the input to the SHA-3 block is assumed to be already padded.

In other works, i.e., [8], the hardware block is not performing only the *f-transform*, but it also has a Versioning and XOR-iring module (VSX) that is responsible for forming the appropriate state per algorithm version.

There are some implementations in which all the steps of the sponge function are supported (padding, mapping, and truncation), but, generally, these architectures assume that the input can only be of a certain length (i.e., [9] considers input messages whose length is fixed to 64 bits), or have a precise application (i.e., [10] considers only the CRYSTALS-Kyber 768 algorithm).

- Absorbing Phase: Here, the padded message is divided into blocks of a size of *r* bits each, and each block is XORed with the current state of the sponge function. The resulting state is then processed through a series of bitwise operations, typically using a permutation function, to mix the input data with the current state. The function *f* acts on the state, with a width of b=r+c.

- Squeezing Phase: After all of the message blocks have been absorbed, the function produces the hash output by repeatedly extracting *r* bits from the state. These bits are concatenated to form the final hash value. The squeezing phase continues until the desired hash length is achieved.

- Finalization: in the end, the sponge function may perform additional operations to finalize the hash value, such as truncating it to the desired length or applying additional cryptographic transformations.

Central to the sponge construction is the concept of state. The state has a length of 1600 bits and consists of a three-dimensional 5×5×645×5×64 table, as shown in **Figure 2**b. Each bit of this cube can be addressed with $A[x,y,z]$. In order to facilitate the description of the applied functions, the following conventions are used: the part of the state that presents the word is also called a lane, a two-dimensional part of the state with a fixed z is called a slice, and all lanes with the same x-coordinate form a sheet.

3. Keccak

The most important part of the SHA-3 and SHAKE primitives is the Keccak permutation function, which calls for 24 rounds of the f-1600 function. Each round is characterized by the five consecutive steps θ,ϱ,π,χ, and ι. These steps have a state array *A* as input and an output *B*, which is a processed new state array. As shown in Equation (1), θ consists of a parity computation, a rotation of one position, and a bitwise XOR.

$$\begin{array}{lll} \theta: & C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4] & 0 \le x \le 4 \\ & D[x] = C[x-1] \oplus ROT(C[x+1],1) & 0 \le x \le 4 \\ & A[x,y] = A[x,y] \oplus D[x] & 0 \le x,y \le 4 \end{array} \quad (1)$$

In Equation (2), ϱ is a rotation by an offset that depends on the word position, and π is a permutation.

$$\rho - \pi: \quad B[y,2x+3y] = ROT(A[x,y], r[x,y]) \qquad 0 \le x,y \le 4 \quad (2)$$

In Equation (3), χ consists of bitwise XOR, NOT, and AND gates.

$$\chi: \quad A\left[x,y\right] = B\left[x,y\right] \oplus \left(\left(\overline{B[x+1,y]} \right) \cdot \left(B\left[x+2,y\right] \right) \right) \quad 0 \le x,y \le 4 \quad (3)$$

Lastly, ι, in Equation (4) is a constant round addition.

$$\iota: \quad A[0,0] = A[0,0] \oplus RC \qquad 0 \le x,y \le 4 \quad (4)$$

When these five are completed, a round is completed. **Table 2** reports the round constant function *RC*[i], which is a 24-value permutation that assigns 64-bit data to the Keccak function. **Table 3** reports the cyclic shift offset *r*[x,y].

Table 2. Values *RC*[i] constants.

***RC*[0] 0x0000000000000001**	***RC*[8] 0x000000000000008a**	***RC*[16] 0x8000000000008002**
***RC*[1] 0x0000000000008082**	***RC*[9] 0x0000000000000088**	***RC*[17] 0x8000000000000080**
***RC*[2] 0x800000000000808a**	***RC*[10] 0x0000000080008009**	***RC*[18] 0x000000000000800a**
***RC*[3] 0x8000000080008000**	***RC*[11] 0x000000008000000a**	***RC*[19] 0x800000008000000a**
***RC*[4] 0x000000000000808b**	***RC*[12] 0x000000008000808b**	***RC*[20] 0x8000000080008081**
***RC*[5] 0x0000000080000001**	***RC*[13] 0x800000000000008b**	***RC*[21] 0x8000000000008080**
***RC*[6] 0x8000000080008081**	***RC*[14] 0x8000000000008089**	***RC*[22] 0x0000000080000001**
***RC*[7] 0x8000000000008009**	***RC*[15] 0x8000000000008003**	***RC*[23] 0x8000000080008008**

Table 3. Values r[x,y] constants.

	X = 3	X = 4	X = 0	X = 1	X = 2
Y = 2	**25**	**39**	**3**	**10**	**43**
Y = 1	**55**	**20**	**36**	**44**	**6**
Y = 0	**28**	**27**	**0**	**1**	**62**
Y = 4	**56**	**14**	**18**	**2**	**61**
Y = 5	**21**	**8**	**41**	**45**	**15**

More information about the Keccak algorithm can be found in [11].

4. Implementation

When developing a real implementation, a diverse array of possibilities within the design space is available. These options encompass entirely hardware-based solutions, pure software implementations, and hybrid approaches, such as Integrated Software Environments (ISE) or Application-Specific Instruction Processor (ASIP). Strictly hardware-based solutions involve dedicated IP cores, while pure software implementations rely solely on software resources. ISEs (Integrated Software Environment) or ASIP, representing a hybrid solution, enhance general-purpose processor cores with specialized hardware and instructions.

Figure 3 shows the different aspects covered in the next sections and proposes for each implementation approach a choice of proper references. Let us now delve into the intricacies of each conceivable approach.

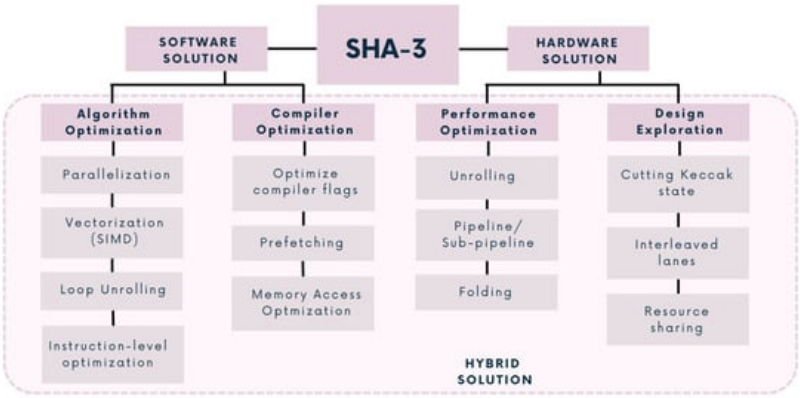

Figure 3. Implementation possibilities scheme.

4.1. Hardware Solutions

Hardware implementations of Keccak demand careful consideration of trade-offs. When implementing Keccak in hardware, the choice of design parameters and strategies heavily depends on the specific goals and constraints of the target application. These objectives typically revolve around factors such as speed, power efficiency, and area utilization.

This section will explore the various aspects that can be considered during the hardware implementation of Keccak, with a focus on these key parameters.

Unrolling. Unrolling is particularly efficient in improving the throughput for single-message hashing. Considering Keccak, the f-permutation block can be replicated and unrolled in the SHA-3 hash function. As an example, Refs. [6][12] implement SHA-3 considering an unrolling factor of two, while, in [7], an even higher degree of unrolling has been analyzed. Moumni et al. [9] and Nannipieri et al. [13] have made several attempts, instantiating from a single instance to twelve, going from 24 clock cycles to 2; however, this resulted in an onerous increase in area.

Pipelining. Pipelining brings the advantages of combined data throughput enhancement in multi-message hashing, where the function processes more than one message concurrently. In addition, two different types of pipelining can be distinguished:

- Classic pipelining, generally used between one round and another;
- Sub-pipelining, inserted instead between two steps of the same round.

For instance, in [8][14], the pipeline is inserted between the π and χ steps, while, in [12], it is inserted between the θ and ϱ steps.

Folding. Towards a more compact SHA-3 structure, folding of the round computation can be considered. In the case of [15], each round is computed over multiple clock cycles, depending on the folding factor.

Cutting the Keccak state. The efficient management of the Keccak state is of paramount importance [16]. There are multiple alternatives, namely using slice-wise, plane-wise, and bit-interleaving techniques. Jungk et al. [17] propose a very compact slice-oriented Keccak hardware, based on the observation that all Keccak steps except ϱ can be performed efficiently with slice-wise processing. However, since input messages for absorption generally arrive in a lane-oriented fashion, the plane-wise partitioning is favorable (adjacent bits in a register belong to the same lane).

Interleaved lanes. Bit-interleaving is a technique that can be used to break large 64-bit lanes of Keccak into smaller chunks [18].

Resource Sharing. Resource area sharing is a crucial optimization technique employed in hardware design, particularly in the context of FPGAs and ASICs. It aims to maximize the efficient utilization of available resources while minimizing the overall hardware footprint, which can lead to cost savings, improved performance, and reduced power consumption. An interesting example is the co-processor presented in [2], named AE$HA-3, which combines two of the NIST's standardized algorithms, i.e., Advance Encryption Standard (AES) and SHA-3. Maache et al. [3] also present a multi-purpose cryptographic system performing both AES and SHA-3, implementing it on the IntelFPGA Cyclone-V device.

To sum up, the hardware implementation of Keccak is a multifaceted task that necessitates careful consideration of various trade-offs and objectives. The specific design choices will be heavily influenced by the unique demands of the target application, whether it be a high-performance cryptographic accelerator, a low-power embedded system, or any other use case in which Keccak is employed. Each implementation will strike a balance between speed, power efficiency, area utilization, and security to meet its intended purpose effectively.

4.2. Software Solutions

Enhancing the software implementation of an algorithm holds the key to unlocking superior performance. By optimizing code, leveraging hardware-specific features, and minimizing resource overhead, software improvements can significantly boost algorithmic efficiency, resulting in faster execution and better utilization of available hardware resources. Accelerating the SHA-3 algorithm on FPGA devices, RISC-V, or ARM without dedicated hardware accelerators involves optimizing the software implementation to maximize performance. Here are some techniques to achieve this.

Parallelization. Using parallelization for implementing multi-threading or multi-processing when having multiple CPU cores can significantly improve performances. Each core can work on a separate chunk of data, improving overall throughput. Pereira et al. [19] present a technique for parallel processing on Graphics Processing Units (GPUs) of the Keccak hash algorithm. They provide the core functionality, and the evaluation is performed on a Xilinx Virtex 5 FPGA.

Vectorization (SIMD). Utilize the SIMD (Single Instruction, Multiple Data) instructions available in modern processors (e.g., ARM NEON, RISC-V RVV) to process multiple data elements in parallel. This can significantly speed up the hashing

process, especially when dealing with large datasets. For example, Ref. [20] proposes a set of six custom instructions for Keccak-f,p [1600, 800, 400, 200] primitives, and, similarly to other crypto-instructions (e.g., Intel AES-NI and SHA), they exploit the wide SIMD (Single Instruction, Multiple Data) registers. Li et al. [21] explore the full potential of parallelization of Keccak-f[1600] in RISC-V-based processors through custom vector extensions on 32-bit and 64-bit architectures.

Loop Unrolling. Unroll loops in the SHA-3 algorithm code to reduce loop overhead and enable the compiler to optimize the code more effectively. This can result in faster execution, especially on CPUs with pipelined execution units. Ref. [22] reports several analyses about security versus area versus the timing of PQC decapsulation algorithms, after loop unrolling, showing how, in most cases, this brings a significant reduction in latency.

Instruction-Level Optimization. Hand-tune critical sections of the code to use processor-specific instructions and features. This may include using assembly language or intrinsic to access specialized instructions for SHA-3 operations. Ref. [23] presents two new techniques for the fast implementation of the Keccak permutation on the A-profile of the Arm architecture: the elimination of explicit rotations in the Keccak permutation through barrel shifting, and the construction of hybrid implementations concurrently leveraging both the scalar and the Neon instruction sets of AArch64.

Optimized Compiler Flags. Use compiler optimization flags (`-O2`, `-O3`, i.e., in [24]) to instruct the compiler to apply various optimizations, including loop unrolling, inline function expansion, and instruction scheduling. Ref. [25] uses `-On` command line flags during GCC compilation to improve performance at the cost of increased compilation times.

Memory Access Optimization. Minimize memory access latency by optimizing data structures and memory access patterns. Cache-friendly data structures and efficient memory layouts can reduce the number of cache misses. Choi et al. [26] discuss optimizations, memory management strategies, and parallelization schemes, aiming to enhance the performance and throughput of SHA-3 operations on graphics processing units (GPUs).

Prefetching. Use prefetching techniques to load data into the cache before it is actually needed, reducing memory access stalls and improving data processing speed. Lee et al. [27], using NVIDIA GPU, exploit the feature for which arithmetic instructions and memory load/store instructions can be executed concurrently, as long as there is no dependency between the executing instruction and data being loaded/stored. They prefetch the input data of Keccak before XORing it into the state, so that address calculation and bitwise XOR operation can run in parallel with the memory copy operation.

The effectiveness of these techniques will depend on the specific platform, compiler, and workload, so thorough testing and profiling are essential to achieve optimal results. Continuously profiling and benchmarking the software implementation will help identify performance bottlenecks and areas for improvement. This iterative process can lead to significant performance gains.

4.3. Hybrid Solutions

Hybrid solutions, which combine both software and hardware components, represent a versatile approach to solving complex problems by harnessing the strengths of each domain. These solutions essentially encompass all of the techniques discussed in the previous section and merge them into a cohesive, integrated system.

In the realm of technology and problem-solving, software and hardware have traditionally been seen as separate entities. Software provides flexibility and adaptability, while hardware offers raw processing power and efficiency. A hybrid solution brings together the computational capabilities of hardware and the logic and adaptability of software to create a powerful and agile system. It allows optimization of the performance by distributing tasks between software and hardware according to their respective strengths. This means that computationally intensive tasks can be offloaded to dedicated hardware accelerators, while software can handle tasks that require flexibility and frequent updates. This balance ensures that the system operates efficiently without bottlenecks. Moreover, being that software is inherently adaptable, it is easier to implement changes and updates to meet evolving requirements. Fritzmann et al. [4] present RISQ-V, an enhanced RISC-V architecture that integrates a set of powerfully coupled accelerators. Here, hardware/software co-design techniques have been combined to develop complex and highly customized solutions, designing tightly and loosely coupled accelerators and Instruction Set Architecture (ISA) extensions. This is an example of how combining the hardware and software provides the flexibility to adjust algorithms, logic, or functionality in response to changing needs while maintaining the stability and speed of the hardware.

References

1. Homsirikamol, E.E.A. Comparing Hardware Performance of Round 3 SHA-3 Candidates using Multiple Hardware Architectures in Xilinx and Altera FPGAs. In Proceedings of the Ecrypt II Hash Workshop, Tallinn, Estonia, 19–20 May 2011.
2. Kundi, D.E.S.; Khalid, A.; Aziz, A.; Wang, C.; O'Neill, M.; Liu, W. Resource-Shared Crypto-Coprocessor of AES Enc/Dec With SHA-3. IEEE Trans. Circuits Syst. Regul. Pap. 2020, 67, 4869–4882.
3. Maache, A.; Kalache, A. Design and Implementation of a flexible Multi-purpose Cryptographic System on low cost FPGA. Int. J. Electr. Comput. Eng. Syst. 2023, 14, 45–58.
4. Fritzmann, T.E.A. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2020, 4, 239–280.
5. Dang, Q. Recommendation for Applications Using Approved Hash Algorithms; US Department of Commerce, National Institute of Standards and Technology: Gaithersburg, MD, USA, 2008.
6. Ioannou, L.; Michail, H.E.; Voyiatzis, A.G. High performance pipelined FPGA implementation of the SHA-3 hash algorithm. In Proceedings of the 4th Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 14–18 June 2015; pp. 68–71.
7. Michail, H.E.; Ioannou, L.; Voyiatzis, A.G. Pipelined SHA-3 Implementations on FPGA: Architecture and Performance Analysis. In Proceedings of the Second Workshop on Cryptography and Security in Computing Systems (CS2 '15), Amsterdam, The Netherlands, 19–21 January 2015; pp. 13–18.
8. Athanasiou, G.S.; Makkas, G.P.; Theodoridis, G. High throughput pipelined FPGA implementation of the new SHA-3 cryptographic hash algorithm. In Proceedings of the 2014 6th International Symposium on Communications, Control and Signal Processing (ISCCSP), Athens, Greece, 21–24 May 2014; pp. 538–541.
9. Moumni, S.E.; Fettach, M.; Tragha, A. High Throughput Implementation of SHA3 Hash Algorithm on Field Programmable Gate Array (FPGA). Microelectron. J. 2019, 93, 104615.
10. Dolmeta, A.; Martina, M.; Masera, G. Hardware architecture for CRYSTALS-Kyber post-quantum cryptographic SHA-3 primitives. In Proceedings of the 2023 18th Conference on Ph.D Research in Microelectronics and Electronics (PRIME), Valencia, Spain, 18–21 June 2023; pp. 209–212.
11. Bertoni, G.; Daemen, J.; Peeters, M.; Assche, G.V. The keccak reference. Submiss. Nist. Round 3 2011.
12. Wong, M.M.; Haj-Yahya, J.; Sau, S.; Chattopadhyay, A. A New High Throughput and Area Efficient SHA-3 Implementation. In Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), Florence, Italy, 27–30 May 2018.
13. Nannipieri, P.; Bertolucci, M.; Baldanzi, L.; Crocetti, L.; Di Matteo, S.; Falaschi, F.; Fanucci, L.; Saponara, S. SHA2 and SHA-3 accelerator design in a 7 nm technology within the European Processor Initiative. Microprocess. Microsystems 2021, 87, 103444.
14. Mestiri, H.; Kahri, F.; Bedoui, M.; Bouallegue, B.; Machhout, M. High throughput pipelined hardware implementation of the KECCAK hash function. In Proceedings of the 2016 International Symposium on Signal, Image, Video and Communications (ISIVC), Tunis, Tunisia, 21–23 November 2016; pp. 282–286.
15. Sundal, M.; Chaves, R. Efficient FPGA Implementation of the SHA-3 Hash Function. In Proceedings of the 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Bochum, Germany, 3–5 July 2017; pp. 86–91.
16. Rawat, H.; Schaumont, P. Vector Instruction Set Extensions for Efficient Computation of Keccak. IEEE Trans. Comput. 2017, 66, 1778–1789.
17. Jungk, B.; Apfelbeck, J. Area-Efficient FPGA Implementations of the SHA-3 Finalists. In Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs, Cancun, Mexico, 30 November–2 December 2011; pp. 235–241.
18. Bertoni, G.; Daemen, J.; Peeters, M.; Assche, G.V.; Keer, R.V. KECCAK Implementation Overview; 2012. Available online: https://keccak.team/index.html (accessed on 15 October 2023).
19. Pereira, F.; Ordonez, E.; Sakai, I.; de Souza, A. Exploiting Parallelism on Keccak: FPGA and GPU comparison. Parallel Cloud Comput. 2013, 2, 1–6.
20. Rawat, H.K.; Schaumont, P. SIMD Instruction Set Extensions for Keccak with Applications to SHA-3, Keyak and Ketje. In Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (HASP '16), Seoul, Korea, 18 June 2016; Article 4; pp. 1–8.
21. Li, H.; Mentens, N.; Picek, S. Maximizing the Potential of Custom RISC-V Vector Extensions for Speeding up SHA-3 Hash Functions. In Proceedings of the 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE),

Antwerp, Belgium, 17–19 April 2023; pp. 1–6.

22. Basu, K.; Soni, D.; Nabeel, M.; Karri, R. NIST Post-Quantum Cryptography- A Hardware Evaluation Study. Cryptol. ePrint Arch. 2019. Available online: https://eprint.iacr.org/2019/047 (accessed on 15 October 2023).
23. Becker, H.; Kannwischer, M.J. Hybrid Scalar/Vector Implementations of Keccak and SPHINCS+ on AArch64. In Proceedings of the International Conference on Cryptology in India, Kolkata, India, 11–14 December 2022; Isobe, T., Sarkar, S., Eds.; Springer: Cham, Switzerland, 2022; pp. 272–293.
24. Dolmeta, A.; Mirigaldi, M.; Martina, M.; Masera, G. Implementation and integration of Keccak accelerator on RISC-V for CRYSTALS-Kyber. In Proceedings of the 20th ACM International Conference on Computing Frontiers (CF ’23), Bologna, Italy, 9–11 May 2023; pp. 381–382.
25. Malik, A.; Aziz, A.; Kundi, D.E.S.; Akhter, M. Software implementation of Standard Hash Algorithm (SHA-3) Keccak on Intel core-i5 and Cavium Networks Octeon Plus embedded platform. In Proceedings of the 2013 2nd Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 15–20 June 2013; pp. 79–83.
26. Choi, H.; Seo, S.C. Fast Implementation of SHA-3 in GPU Environment. IEEE Access 2021, 9, 144574–144586.
27. Lee, W.K.; Phan, R.C.W.; Goi, B.M.; Chen, L.; Zhang, X.; Xiong, N.N. Parallel and High Speed Hashing in GPU for Telemedicine Applications. IEEE Access 2018, 6, 37991–38002.
