Energy-Efficient Acceleration of Hash-Based Post-Quantum Cryptographic Schemes on Embedded Spatial Architectures

Yanze Wu
ywu42@gmu.edu
George Mason University
Fairfax, VA, US

Md Tanvir Arafin
marafin@gmu.edu
George Mason University
Fairfax, VA, US

Abstract

This work introduces AXIOS, a novel spatial architecture for accelerating hash-based post-quantum cryptography (PQC) primitives. AXIOS demonstrates that structural regularities in hash-based algorithms can be efficiently mapped to spatial accelerators that support FPGA-based programming for granular control and *coarse-grained reconfigurable arrays (CGRA)* for repeated tasks. AXIOS chooses the key generation task of the eXtended Markle Signature Scheme (XMSS) that embodies critical implementation challenges in modern hash-based (PQC) algorithms. AXIOS implementation on the AMD's VCK-190 platform demonstrates **8.54**× improvement in runtime and **71.65**× improvement in energy efficiency compared to a benchmark implementation on Intel®Core™i9-14900K. AXIOS also breaks the current record of XMSS acceleration in terms of execution time on an embedded SoC or an FPGA platform. *To our knowledge, this is the first efficient hardware implementation of compute-intensive hash-based PQC schemes in an embedded spatial architecture.* Albeit complex, this **FPGA+CGRA**-based design is a promising step to support compute-intensive PQC applications at the edge. This work's code and experimental artifacts are publicly available at **https://anonymous.4open.science/r/AXIOS/**.

Keywords

Hash-based Post-Quantum Cryptography, eXtended Merkle Signature Scheme, Winternitz One-time Signature Plus (WOTS$^+$), Adaptive System on Chip (ASoC), Spatial Computing Architecture

ACM Reference Format:
Yanze Wu and Md Tanvir Arafin. 2018. Energy-Efficient Acceleration of Hash-Based Post-Quantum Cryptographic Schemes on Embedded Spatial Architectures. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Reconfigurable computing is experiencing a paradigm shift. On the one hand, leading field-programmable gate array (FPGA) manufacturers, such as AMD-Xilinx, are introducing newer architecture combining FPGA with custom coarse-grained *reconfigurable* array (CGRA) units [2]. On the other hand, accelerator designers such as Tenstorrent are marketing their embedded GraySkull [34] devices containing *reprogrammable* arrays of hundreds of RISC-V cores on a single chip. This newer generation of hardware is targeted towards novel machine-learning (ML) accelerator designs. Interestingly, most of these accelerators share a typical *spatial architecture* pattern where a large number of simple processing elements (PE) are connected in a two-dimensional grid via a fast and reconfigurable network. One of the key appeals of such spatial architecture is that it can provide reconfigurability and coarse granularity for complex workloads, which results in fast operation and low power consumption [1, 7, 19, 26, 38, 42, 43]. However, using such reconfigurable spatial architectures for compute-intensive cryptographic applications remains relatively unexplored.

Cryptography in reconfigurable hardware is primarily limited to traditional FPGA-based solutions. Unfortunately, there are two fundamental issues with the FPGA-only acceleration. First, although reconfigurable, FPGAs are significantly slower when compared with their ASIC or CPU counterparts [24]. Second, FPGAs offer a very granular level of design primitives, which suffers from scalability issues for heterogeneous systems. To address the second issue, CPU+FPGA architectures such as Zynq were commercialized, which removes the computation burden for non-FPGA targeted tasks (such as running an OS or a host program) to the CPU.

However, there still existed a gap that required reconfigurable yet faster-than-FPGA solutions. CGRAs fill that gap by introducing hardened *programmable* PEs that support a handful of operations helpful in accelerating a specific computation. The marriage between FPGA and CGRA offers the best of both worlds for *reconfigurable computing*, where frequent more straightforward operations can be accelerated fast with CGRAs, and the complex tasks can be realized in the FPGA. Thus, the new generation of FPGAs is entering the market that provides **CPU+FPGA+CGRA** solutions [25].

This work pushes the boundary of cryptography on this new generation of reconfigurable hardware. We select the implementation hash-based post-quantum cryptographic (PQC) algorithms on a spatial architecture due to their demanding computation requirements. Standard implementations of hash-based PQC algorithms suffer from fundamental computation bottlenecks. For example, the calculation of modern hash-based algorithms, such as eXtended Markle Signature Scheme (XMSS) and SPHINCS+, depends on a compute-expensive substructure, the *Winternize One-Time Signature Plus(WOTS$^+$)*, which requires thousands of hash function computations during the process [9, 17, 40]. As a result, hash-based PQCs are slower than their classical non-PQC counterparts. For instance, XMSS key generation (with parameter $h = 20$) takes *minutes* for its standard CPU implementation. This computation overhead makes hash-based PQCs poorly suitable for resource-constrained applications at the edge. *The key motivation of this paper is to demonstrate the feasibility and applicability of modern reconfigurable spatial architectures to address the computing challenges of advanced cryptographic schemes under the resource constraints of user/edge devices.*

Several works in the current literature have proposed hardware and software optimization techniques to reduce the computing and

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

storage overhead of hash-based PQCs [4–6, 21, 23, 35, 36]. For example, Buchmann *et al.* [4] proposed a novel node traversal method that significantly speeds up the signature generation process. More recently, Wang *et al.* [35] have demonstrated hash-based PQC applications in embedded devices by utilizing hardware acceleration techniques. Other works such as [5, 6] come up with accelerators for signature generation in FPGAs while [21] complements the accelerators for verification. Moreover, [23] proposes a sub-structure of the hash-based quantum-resistant algorithm (called leaf) in ASICs to achieve further speedup. Unfortunately, only some of these works have evaluated and reported their design's power efficiency, the energy-speed trade-off, and the applicability (of their designs) in edge devices. Thus, fast and efficient implementation of hash-based PQC algorithm on embedded platforms remains a crucial open problem.

Our Contributions

This work implements XMSS, a representative hash-based PQC algorithm, in the spatial AI engine core in AMD's Versal Adaptive System on Chip (ASoC) platform. In recent years, Versal ASoC has shown outstanding performance in multitudes of applications spanning from signal processing to machine-learning problems [11, 32, 37, 39, 41]. However, the application of compute-intensive cryptographic algorithms has yet to be explored in detail in the ASoC and similar architectures.

To the best of our knowledge, *this work is the first to apply ASoC in the domain of hash-based PQC*. It provides a novel FPGA+CGRA-based solution applicable to generic hash-based PQC algorithms, requiring fast calculation with limited hardware resources. We construct the XMSS accelerators on ASoC architecture step by step, from abstracting the data flow to mapping the AI Engine tiles. Not only does the methodology work for XMSS but also for other hash-based cryptography primitives, *e.g.*, SPHINCS+ and LMS, since they have the same WOTS+ and Merkle tree components. Our overall contributions can be summarized as follows:

- We introduce **AXIOS: a novel accelerator for XMSS implementation on spatial architectures**. To address the computation bottleneck in hash-based PQCs, AXIOS utilizes a highly optimized data-flow pattern mapped to the spatial architecture available in ASoC devices.
- We implement AXIOS on the VCK-190 evaluation board and measure the runtime and power consumption of the XMSS algorithm. Our hardware implementation of AXIOS outperforms an Intel(R) Core(TM) i9-14900K processor by **8.54**× in terms of runtime and by **71.65**× in terms of energy efficiency for XMSS key-generation. *We benchmark AXIOS over key-generation since this is the most computation intensive task in the XMSS algorithm.*
- We have published the source codes for deploying XMSS on the Versal ASoC architecture, available at **https://anonymous.4open.science/r/AXIOS/**.

The rest of the paper is organized as follows. The basics of the XMSS and spatial computing architecture are given in the next section. Section 3 presents the details of AXIOS in Versal ASoC using a heterogeneous FPGA+CGRA design. Section 4 provides a thorough performance analysis to evaluate the applicability of spatial acceleration for PQC applications. Section 5 concludes the paper.

2 Preliminaries

This section introduces the fundamental concepts for XMSS that are relevant to hash-based cryptography and the details of spatial computing architectures.

2.1 eXtended Merkle Signature Scheme

The eXtended Merkle signature scheme is a hash-based post-quantum cryptographic algorithm standardized by the National Institute of Standards and Technology (NIST) in 2018 [10, 15]. XMSS has become popular due to its minimal security assumptions, *i.e.*, its security relies on the collision-resistant properties of the hash function [12, 16, 28, 29]. In addition, XMSS supports another hash primitive SHAKE as an equivalent choice. Both can be efficiently constructed to defend against post-quantum computers. As a result, XMSS remains a successful quantum-resistant algorithm with applications proposed in vehicular communication [13, 20], secure boot [21], and quantum-resistant ledgers [3, 30].

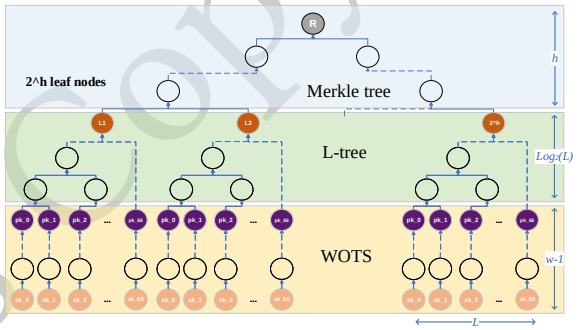

Figure 1: An overview of the XMSS architecture, which is divided into three layers: (1) The bottom (yellow) layer contains chains of WOTS$^+$ instances. Each chain in the WOTS$^+$ instance starts with a secret subkey (orange nodes) and converts it to a public subkey (purple nodes). (2) The green middle layer contains unbalanced binary L-trees. Each tree compresses a set of WOTS$^+$ public keys into L-tree roots (brown nodes). (3) The top (blue) Merkle tree layer takes the L-tree roots as inputs. The root of this Merkle tree is the public key of the XMSS.

Similar to other hash-based PQC algorithms, XMSS computation has a tree-like structure, *i.e.,* the XMSS-tree, as shown in Figure 1. This structure can be divided into three distinct layers: (1) the bottom layer that contains chains of Winternitz one-time signatures, (2) the middle layer consists of unbalanced binary trees, *i.e.,* the L-tree, and (3) a top Merkle tree layer. Details on each of these layers are given below.

2.1.1 Winternitz One-Time Signature Layer. The Winternitz one-time signature scheme is first proposed in [22] to generate one-time signatures (OTSs). The WOTS$^+$ layer comprises of multiple WOTS$^+$ instances. A WOTS$^+$ instance is computed using L number of hash-chains, *a.k.a.* the WOTS$^+$-chains. Each chain starts with a secret key element $\text{WOTS}^+_{sk_i}$, at the bottom node (the orange nodes in Figure 1) and outputs a top node containing the public key element $\text{WOTS}^+_{pk_i}$ (the purple nodes), where $i \in \{0, 1, \ldots, L-1\}$. Every node in a WOTS$^+$ chain is derived using $thash_f$, an XMSS-specific function, *i.e.*,

$$WOTS^{+}_{pk}: (pk_0, pk_1, ..., pk_{L-1}) \leftarrow thash_f^{w-1}(sk_{0\_0}, ..., sk_{L-1\_0})$$

The *thash* functions, *e.g.*, $thash_f$ and $thash_h$, are XMSS-specific transfer functions that generate a node of the XMSS tree from the previous node(s). Each node in the XMSS tree has a unique predefined address. Within a WOTS$^+$ chain, the $thash_f$ function takes three inputs: (1) the address, (2) the value of the previous node, and (3) a public seed. The $thash_f$ function generates the node value for the current node using an XOR operation, two pseudo-random functions, and a core hash function F, as shown in the left sub-figure in Figure 2.

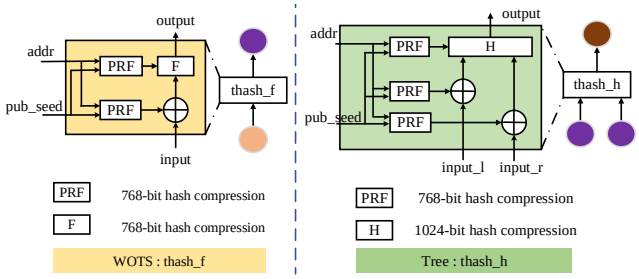

Figure 2: Two XMSS-specified function, *i.e.*, $thash_f$ and $thash_h$, are used in WOTS layer and L-tree layer respectively. PRF, F, and H functions are generally implemented with the SHA-256 function with different input lengths.

To sign a message M of arbitrary length, first, WOTS$^+$ uses a secure hash function to compress the message into n-byte digest D. Then, this digest is split into l_1 k-bit words, *i.e.*, $d_0, d_1, \ldots d_{l1}$, where, $k = log_2(w)$, w is the Winternitz parameter, and l_1 is given by:

$$l_1 = \left\lceil \frac{8n}{log_2(w)} \right\rceil \quad (1)$$

A checksum c is also calculated using the values of words d_i. This checksum is then split into l_2 k-bit words *i.e.*, $c_0, c_1, \ldots c_{l_2}$, where

$$l_2 = \left\lfloor \frac{log_2(l_1(w-1))}{log_2(w)} + 1 \right\rfloor \quad (2)$$

The number of chains, L, for a WOTS$^+$ instance is given by the sum of l_1 and l_2, *i.e.*, $L = l_1 + l_2$. Each chain corresponds to each elements of the set $r = \{d_0, d_1, \ldots d_{l1}, c_0, c_1, \ldots c_{l2}\}$. Thus, the WOTS$^+$ signature $WOTS^{+}_{sig}$ of the digest D is calculated by combining the values of the r_i^{th} node of the i^{th} chain, where $i \in \{0, 1, \ldots, L-1\}$.

2.1.2 L-tree Layer. The L-tree is the middle layer of XMSS, which compresses WOTS$^+$ public keys, *i.e.*, the pk_is, into leaf nodes for the top Merkle tree. Since each node in the L-tree is derived from two lower-level nodes, the $thash_h$ operation now takes (1) two-node values from the lower sibling nodes *i.e.*, $input_l$ and $input_r$, (2) a public seed, and (3) an address value, as shown in the right subfigure of Figure 2. Note that the addresses of the siblings in an L-tree only differ by their least significant bit. Therefore, the address value for the $thash_h$ function is generated by right-shifting the address of either the lower-left or the lower-right node by 1-bit.

2.1.3 Merkle-tree Layer. A Merkle tree sits on the top of the XMSS structure, as shown in Figure 1. To ensure enough root-signature pairs, the Merkle tree chooses a large height h, which means the number of leaves increases exponentially along with the parameter h. Like the L-tree, the nodes of the Markle tree are generated using the $thash_h$ function.

Using this fundamental XMSS tree structure, cryptographic functions such as key generation, signature generation, and verification can be performed.

2.1.4 Key Generation. XMSS accepts two random strings: one as a secret seed and the other as a public seed for key generation. First, WOTS$^+$ secret keys are generated by hashing the secret seed with the addresses of the bottom nodes in the WOTS$^+$ layer. Entire hash chains are generated for the WOTS$^+$ layer using these bottom nodes described in 2.1.1. After that, the L-tree is created using the public keys derived from the WOTS$^+$ chain. Finally, a Merkle tree is derived from the L-tree layer. The root of this Merkle tree is used as the public key for the XMSS algorithm. A generated key pair supports up to 2^h signature generation and verification, key generation will have more space for acceleration than the other two processes.

2.1.5 Signature Generation. The XMSS tree generated by the key generation process is used to sign a message. First, for a message M, a secure hash function creates a digest D with a uniformly random string r. Then, an index, s, is selected to determine which WOTS$^+$ instance of the XMSS tree will be used for the signature. Using this s^{th} WOTS$^+$ instance, a WOTS$^+$ signature, $WOTS^{+}_{sig}$, for the digest is generated. Also, the index is used to create the authentication path (P) from the WOTS$^+$ instance to the root. The index, r, $WOTS^{+}_{sig}$, and P are given as the XMSS signature for the message.

$$XMSS_{sig}: \{M, s, r, WOTS^{+}_{sig}, P\} \leftarrow Sign(M, XMSS_{sk})$$

2.1.6 Signature Verification. To verify an XMSS signature, the verifier generates the same digest D from the message M. Then, using the WOTS$^+$ signature, $WOTS^{+}_{sig}$, and the digest D, the verifier can compute the WOTS$^+$ public keys by traversing the WOTS$^+$ chain. After obtaining the $WOTS^{+}_{pk}$, the verifier can use the index s and the authentication path P (provided in the signature) to recreate the root of the Merkle tree. If the root of the Merkle tree matches with the XMSS public key (generated in the key generation process), the verifier accepts the message.

2.1.7 XMSS Parameter Set. Parameter sets $\{n, h, w\}$ describe the features of an XMSS implementation. n represents the byte in length for each node, which is typically selected from $\{32, 64\}$. w is the Winternitz parameter representing the signature context of WOTS$^+$. It also reflects the distance from the WOTS$^+$ secret key (orange nodes, denoted as $WOTS^{+}_{sk}$) to the WOTS$^+$ public key (purple nodes, $WOTS^{+}_{pk}$). h is the height of the Merkle tree, which denotes the number of signatures a root supports. XMSS generates 2^h WOTS$^+$ instances to obtain a root and compresses all the nodes through the three layers, as shown in Figure 1. AXIOS selects the most common parameter set $\{n, h, w\} = \{32, 10, 16\}$ and primitive SHA256. This selection provides a standard framework to compare the spatial acceleration of hash-based PQC with implementation in other architectures.

2.2 Spatial Computing Architecture

There has been a paradigm shift in hardware accelerator design due to the current progress in ML applications. Although GPUs can handle large ML models' training and inference processes, they suffer from data-movement bottlenecks, inefficient power consumption, and issues originating from the fixed SIMD architecture. Thus, ML-specific accelerators such as Groq's Tensor Streaming chips [14], Tenstorrent's GraySkull processors [34], and AMD's Versal AI Engines [2] are deployed as an alternative. Interestingly, most of these accelerators share a typical spatial architecture pattern,

A spatial architecture (SA) is characterized by a two-dimensional grid of independent processing elements (PE) connected via a reconfigurable network. For example, Tenstorrent's GraySkull boasts a collection of 600 low-power RISC-V cores on a single die. In contrast, AMD's Versal AI Engines provide hundreds of compute tiles arranged in a 2D spatial configuration [2]. Take Versal AI Engines as examples; a few works have [1, 19, 26] applied this platform to the convolutional neural network acceleration and achieved 10-20× higher frames per second (FPS) than the classic FPGA solutions (e.g., ZYNQ). Similar work [7] outperforms programmable logic (PL) by 3.9-96.7× on constructing GNN accelerators. Zhuang et al. [42] improves the throughput of the heavily used kernel, *i.e.*, Matrix Multiplication (MM), by 1.00-32.51× compared to one monolithic accelerator. They further propose efficient MM accelerators in paper [43]. Yang's group proposes arbitrary-precision multiplication accelerators with a 12.6× efficiency over the Intel Xeon Ice Lake 6346 CPU. Thus, the new SA-based architectures demonstrate significant performance gains for computationally complex workloads.

Spatial architectures are becoming popular in accelerating ML workloads for several reasons. First, SA is a many-core architecture that supports core counts in the order of hundreds to thousands and, thus, offers fast and parallel processing of linear layers of a deep neural network (DNN). Second, SAs support coarse-grained programmability for the PEs, which is profoundly different from the SIMD approach taken by the GPUs. This coarse-grained programmability is better suited for targeting compute bottlenecks of non-linear layers in a DNN. Finally, SAs contain low-power PEs or tiles with energy-efficient control mechanisms that can significantly reduce the power consumption for an ML workload. As a result, recent SA designs, such as Groq's Gorq chip, have achieved record-breaking performance in processing large language models [8].

2.2.1 Versal Adaptive System-on-Chip Platform. When integrated into a heterogeneous system with additional control and reconfigurability options, SAs can open better acceleration opportunities for modern computing workloads. AMD has developed the Versal® Adaptive System-on-Chip (ASoC) to explore this opportunity. ASoC provides a heterogeneous platform containing real-time processing elements (the PS side), a large programmable logic (PL) core, and a spatial accelerator built using CGRAs titled the adaptable intelligent engine (AIE). The PS, PL, and AIE are connected using programmable network-on-chips (NoCs) for high-bandwidth communication[31]. Figure 3 presents an overview of the ASoC architecture.

2.2.2 Real-time Processing in Versal ASoCs. The PS side of the Versal ASoC is equipped with ARM processors that run complex applications and control the entire platform. For accelerating complex workloads, the PS side works as the host that (1) manages the startup and synchronization of different elements, *i.e.* the logic

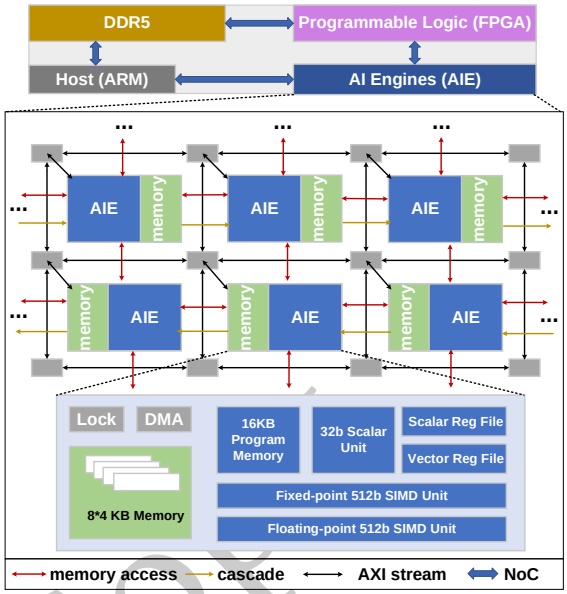

Figure 3: Architecture of the Versal adaptive computing acceleration platform. It consists of (1) host processors that execute complex real-time applications, (2) programmable logic to support FPGA-based hardware acceleration, (3) AI engines for spatial acceleration tasks, and (4) programmable network on the chips (NOC) that ensures fast communication between the components.

blocks in the FPGA core and the tiles in the AIE, and (2) oversees the data movement among the PL, AIE, memory, and the peripherals.

2.2.3 Programmable Logic Elements in Versal ASoCs. The PL side of the ASoC contains building blocks for programable logic *e.g.*, look-up tables (LUT), flip-flops (FF), and block RAMs. By constructing complex FPGA logic blocks, the PL side can accelerate tasks unfriendly to the CPU, such as large integer processing, modular arithmetic, and matrix multiplication. Unfortunately, the PL side is significantly slower than the PS or the AIE, as it works at a relatively low frequency (up to a few hundred megahertz).

2.2.4 Adaptable Intelligent Engine (AIE). The adaptable intelligent engine unit in Versal contains a large array of processing tiles arranged in a checkerboard configuration. Each AIE tile has eight banks of 4 KB local data memory, 16KB program memory, and a 32-bit VLIW processor, as shown in Figure 3. The AIE tiles offer scalar and vector computing capabilities over different data types at 1000-1250 MHz frequency. Tiles in the even rows have their memory bank on the left side, while tiles in the odd rows have memory banks on the right. This memory arrangement allows a tile to directly access the data memory elements of the neighboring tiles in all four spatial directions. The tiles are connected via cascade channels, enabling direct connection among the tiles on the same row. In addition, each tile has an AXI-stream interface that supports communication between non-neighboring tiles. The processors in the AIE tiles maintain a lock mechanism to avoid memory access collisions during communication.

3 Implementation

By studying the construction of the XMSS presented in the previous section, we find several features that make it suitable for implementing XMSS in a spatial architecture. These features are given below.

- F1. WOTS$^+$ computation for a single chain can be divided into small compute kernels deployable in the tiles of a spatial architecture.
- F2. Algorithmic data independence in the *thash* functions can exploit tile-level parallelism.
- F3. Each WOTS$^+$ chain can be constructed to support pipelined operation to increase the throughput.
- F4. Chains in a WOTS$^+$ instance are independent and can be deployed spatially in a parallel manner.
- F5. L-trees in the middle layer of the XMSS architecture can also be individually pipelined and implemented spatially to support parallel operation. Similar acceleration techniques can also be used for the top Merkle tree.

Our accelerator design, AXIOS, exploits all the features F1-F5 to derive an energy-efficient and high-speed implementation of XMSS on an embedded spatial architecture. Next, we provide detailed explanations of each step of the AXIOS design.

3.1 AIE Optimized Hash Kernel

The core of hash-based cryptography relies on efficient computation of the hash functions. Therefore, we first design a general-use kernel for SHA-256 that can be mapped and executed efficiently on a given AIE tile. Since the SHA-256 computation revolves around bitwise operations of 32-bit unsigned integers, AXIOS resorts to an efficient mapping of SHA-256 on the scalar engines of the AIE tiles.

Algorithm 1 AIE optimized SHA-256 Kernel

```
0:  procedure SHA-256(data_out, data_in, length)
1:  unit32_t H ← H_0 ;                          ▷ Initialize vector
2:  size_t new_len ← length + padding_len;      ▷ Padded length
3:  unsigned char buf[new_len] ← padding(data_in);
4:  chunk ← new_len / 64 ;      ▷ The number of message blocks
5:  uint32 w[64];
6:  int i, j = 0 ;
7:  while i < chunk  do
8:     uint32_t w[16] ← parallel_fill(buf[64*i]);  ▷ Fill in vector
9:     w[17-64] ← parallel_expand (w[16]) ;        ▷ Operate in vector
10:    unit32_t A ← H_i ;
11:    for j =0; j<64; j++ do
12:       A ← mapping(A, w[j]);
13:    end for
14:    i++;
15:    H_i ← A;                                    ▷ Update vector
16: end while
17: return : data_out ← H_i;
```

AXIOS implements the SHA-256 kernel in each tile in two steps: a. loading and storing the message into the tile buffer (lines 2-4 of Algorithm 1) and b. bitwise mapping operations (lines 7-16, Algorithm 1). For the loading operation, we assume the input data, *i.e.,* messages, are stored as byte strings in the DRAM. AXIOS builds an accelerator in the programmable logic (PL), *i.e.,* in the FPGA side, to support fast READ operations from the DRAM. The PS side communicates with this PL component to read and transfer data from DRAM to the AI engine (AIE) via the AXI-stream interface. At the AIE side, once the stream FIFO is accessible, the SHA-256 kernel reads the byte strings and fills the tile buffers in a pipelined manner.

In lines 2-4 of Algorithm 1, each kernel loads, packs, and stores a message into a variable buffer, then divides it into 64-byte *chunks*. In the while loop in lines 7-16, AXIOS executes the mapping function on the scalar engines to update internal vector H_i for every *chunk*. The *parallel_fill* and *parallel_expand* functions fill and expand the intermediate variables while the *mapping* function sequentially performs the bitwise mapping computation of the SHA-256 algorithm.

Algorithm 2 SHAKE-256 Kernel on AIE

```
0:  procedure SHAKE-256(data_out, out_len, data_in, in_len)
1:  size_t nblock = out_len / SHAKE_rate ;   ▷ Predifined to 136
2:  uint8_t t[SHAKE_rate] ;
3:  uint64_t s[25];
4:  shake_absorb(s,input,in_len);                ▷ Absorb input
5:  shake_squeezeblocks(data_out,blocks,s);      ▷ Squeeze out
6:  data_out += nblocks * SHAKE_rate;
7:  out_len -= blocks * SHAKE_rate;
8:  shake_squeezeblocks(t,1,s);
9:  if out_len then
10:    for i = 0 ; i < out_len ; ++i do
11:       data_out[i] = t[i];
12:    end for
13: end if
14: return : data_out[i];
```

We also implement SHAKE-256 on AIE as an optional selection. SHAKE processes arbitrary in/output data lengths with *absorb* and *squeezeblocks* functions (lines 4-5). The inner process of SHAKE executes a 24-round state permutation called Keccak F1600. Since SHAKE is not a mandatory choice for XMSS, we offer performance mainly on SHA-256 and provide SHAKE's results as a reference.

3.2 Parallel Computation of the thash Function

XMSS-specified functions $thash_f$ and $thash_h$, depicted in Figure 4, are the node-generating functions for the XMSS signature scheme. In $thash_f$, SHA-256 implements the keyed pseudo-random functions (PRFs) and a core hash function (F). PRF and F are 96-byte compression functions that require two ($\lceil \frac{96}{64} \rceil = 2$ chunks) internal bitwise mappings in SHA-256. Assuming the delay of one SHA-256 mapping as the unit time u, a naïve implementation of $thash_f$ takes $3 \times 2u = 6u$ time to obtain the results with equation,

$$Data\_out \leftarrow thash_f(Addr, Pub\_seed, data\_in) \tag{3}$$

where $thash_f$ can be further divided into sub-functions:

$$Key \leftarrow \text{PRF}(Padding, Addr \otimes bit\_mask_0, Pub\_seed)$$

$$Mask \leftarrow \text{PRF}(Padding, Addr \otimes bit\_mask_1, Pub\_seed)$$

$$Data\_out \leftarrow \text{F}(Padding, Key, data\_in \otimes Mask)$$

On the other hand, $thash_h$ takes three PRF functions and a core hash function H. H executes the mapping function three times to

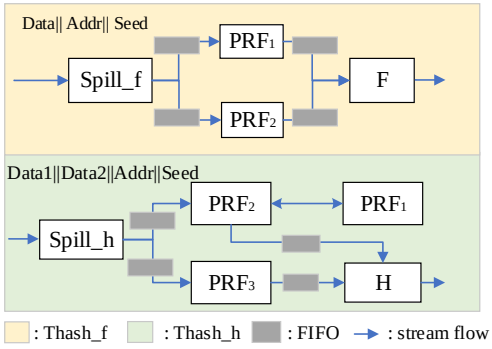

Figure 4: Computation of $thash_f$ and $thash_h$ functions for XMSS in the AI Engine. Each white block in this figure corresponds to an AIE tile. The PRF, F, and H tiles execute the hashing operations related to the corresponding functions. The $Spill_f$ and $Spill_h$ tiles are configured to distribute data streams (containing data, address, and the public seed) to the PRF tiles. The grey blocks represent 128-byte stream FIFOs.

compress a 128-byte input. Therefore, a naïve $thash_h$ implementation takes $3 \times 2u + 3u = 9u$ units of time to compute the results with equations given below:

$$Data\_out \leftarrow thash_h(Addr, Pub\_seed, data\_in_1, data\_in_2) \quad (4)$$

where $thash_h$ consists of the following sub-functions:

$$Key \leftarrow \text{PRF}(Padding, Addr \otimes bit\_mask_0, Pub\_seed)$$

$$Mask_1 \leftarrow \text{PRF}(Padding, Addr \otimes bit\_mask_1, Pub\_seed)$$

$$Mask_2 \leftarrow \text{PRF}(Padding, Addr \otimes bit\_mask_2, Pub\_seed)$$

$$Data\_out \leftarrow \text{H}(Padding, Key, data\_in_1 \otimes Mask_1, data\_in_2 \otimes Mask_2)$$

To improve the efficiency of the *thash* functions, AXIOS uses the tile-level parallelism feature (F2) by allocating different sub-functions across different tiles. Take $thash_f$ as an example. First, a stream control tile, $Spill\_f$ in Fig4, is created that simultaneously distributes the incoming data to the PRF_1 and PRF_2 tiles. Then, the parallel output streams from PRF_1 and PRF_2 are packed and processed by the tile F. This pattern takes a period of $2u + 2u = 4u$ time to compute $thash_f$ which is 33% lower than the naïve approach. Similarly, $thash_h$ computation can be accelerated by allocating the parallel PRF tasks into 3 tiles, reducing the delay from $9u$ to $5u$.

AXIOS chooses the AXI stream to connect the tiles instead of buffers since the AXI bus in Versal ASoC supports communication between non-neighboring tiles. This is required since each AIE tile has only two input and two output streaming ports, whereas $thash_h$ requires streaming from three ports simultaneously. To fit these constraints, the PRF_2 tile curbs out a dedicated path that delivers and receives data from the non-neighboring PRF_1.

3.3 Accelerating the WOTS$^+$ Layer

AXIOS implements a highly pipelined construction of the WOTS$^+$ chain using the features F1 and F3, which are abstracted from the two loops in Algorithm 3. Lines 3-9 in Algorithm 3 shows that the WOTS+ chain computation can be unrolled into a stream crossing $w - 1$ $thash_f$ kernels as

$$pk_i \xleftarrow{thash_f()} sk_{i\_15} \xleftarrow{thash_f()} sk_{i\_14} \xleftarrow{thash_f()} \dots \xleftarrow{thash_f()} sk_{i\_0}$$

where i denotes the index of chains. For feature F3, Lines 1-2 in Algorithm 3 indicate that WOTS+ chains are independent with different addresses. That means the computation of multiple WOTS+ chains can happen simultaneously. AXIOS leverages this and constructs multiple accelerators for the chain.

Algorithm 3 WOTS+ layer

```
0:  procedure WOTS+(pk_{0,1,...,L−1}, sk_{0,1,...,L−1}, Pub_seed, Addr)
1:  for i = 0; i < L; i++ do
2:      Addr.setChain(i);                    ▷ A WOTS+ has L chains
3:      for j = 0; j < w; j++ do
4:          if j == w-1 then
5:              pk_i ← thash_f(Addr,Pub_seed,sk_i);   ▷ Computing
                WOTS+ pk from sk nodes
6:          else
7:              sk_i ← thash_f(Addr,Pub_seed,sk_i);
8:          end if
9:      end for
10: end for
11: return : pk_{0,1,...,L−1};
```

Given the features F1 and F3, AXIOS builds the accelerators for the WOTS+ layer. AXIOS first constructs the WOTS$^+$ chain, which consists of multiple $thash_f$ pipes, depicted as the F pipes in Figure 5. Each $thash_f$ pipe bundles four AIE-tiles together, as 3.2 describes, to compute the $thash_f$ function. A series of $thash_f$ pipes, along with an input tile (SK) and an output tile (PK), computes the WOTS$^+$ chain in a pipeline manner as

$$\dots \xleftarrow{thash_f()} sk_{i+0\_13} \xleftarrow{thash_f()} sk_{i+0\_12} \xleftarrow{thash_f()} sk_{i+0\_11} \xleftarrow{thash_f()} \dots$$
$$\dots \xleftarrow{thash_f()} sk_{i+1\_12} \xleftarrow{thash_f()} sk_{i+1\_11} \xleftarrow{thash_f()} sk_{i+1\_10} \xleftarrow{thash_f()} \dots$$
$$\dots \xleftarrow{thash_f()} sk_{i+2\_11} \xleftarrow{thash_f()} sk_{i+2\_10} \xleftarrow{thash_f()} sk_{i+2\_9} \xleftarrow{thash_f()} \dots$$

The WOTS$^+$ secret keys in Figure 5 are denoted with number and phase to indicate the progress during a highly pipelined process. An example operation is represented. At period t_{15}, the input tile (SK) is ready to send the fifteenth WOTS$^+$ secret key ($sk15\_0$). The first computing tile (F0) is processing the previous secret key ($sk14\_0$) while an earlier secret key sk_{13} is under the next phase ($sk13\_1$). Meanwhile, the first WOTS$^+$ secret key sk_0 segment should be in the last phase, as shown in ($sk0\_14$). At t_{16}, SK will send a new secret key sk_{16}, and different keys in pipes are moving to the next phase simultaneously, *e.g.*, F0 process sk_{15} and F1 computing sk_{14}. At t_{17}, the output tile (PK) has accepted WOTS$^+$ public keys in a stream and distributed them to the following structures.

For the AXIOS implementation with the parameter set defined in Section 2.1.7, it takes 60 AIE tiles to build a WOTS$^+$ chain. Each tile constantly accesses data from the frontward FIFO and writes to the backward FIFO. As mentioned before, it takes $4u$ time for the j^{th} pipe to implement $thash_f^{th}$ to compute the node values, where $i \in \{0, 1, \dots, L-1\}$ and $j \in \{0, 1, \dots, w-1\}$

$$sk_{i\_j+1} \leftarrow thash_f(Addr, Pub\_seed, sk_{i\_j}) \quad (5)$$

and overall $60u$ time is needed to transform a WOTS^+_{sk} to WOTS^+_{pk}.

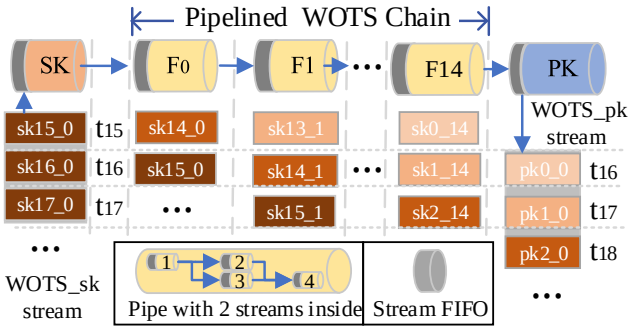

Figure 5: A highly pipelined construction of a WOTS$^+$ chain. An input tile (SK) accepts WOTS$^+_{sk}$ into the pipeline, and an output tile (PK) produces the WOTS$^+_{pk}$. The F-pipes implement the $thash_f$ functions. Grey pipes are 128-byte stream FIFOs that prevent deadlocks in the pipeline. The arrows denote the direction of the flow of the data stream. An example pipeline operation is also depicted. At t_{15}, F_0 process the WOTS$^+_{sk14}$ and at the next period t_{16}, WOTS$^+_{sk14}$ goes to the next pipe F_1 while F_0 processes a new WOTS$^+_{sk15}$ from the SK tile. Meanwhile, at t_{16}, the pipe F_{16} outputs the public key for the chain.

Since AXIOS exploits a pipeline pattern, it only takes an additional $2u$ time for AXIOS to produce a second WOTS$^+_{pk}$ node. With the WOTS$^+$ parameter $w = 16$, there will be 67 WOTS$^+$ chains. This will require $192u$ time to obtain a complete WOTS$^+_{pk}$ as shown in Equation 6. This demonstrates the availability of feature F4 in the spatial architecture.

$$D_1 = 60u + 2u \times 66 = 192u \tag{6}$$

In a general processor, the same computation for building a WOTS$^+$ instance without pipelining and tile-level parallelization will require 6432u time

$$D_2 = 67 \times 16 \times 6u = 6432u \tag{7}$$

The proposed implementation speeds up the calculation by 32 times for one WOTS$^+$ instance with 67 chains. For any arbitrarily large chains n derived from multiple WOTS$^+$ instances, such gain can be understood from the following equations.

$$D_{AXIOS} = 60 + 2u \times (n-1) \tag{8}$$

$$D_{generic\_non-pipelined\_processing} = 16 \times 6u \times (n) \tag{9}$$

where $n \in \mathbb{N}$.

3.4 Orchestrating Dataflow in the L-tree Layer

AXIOS applies the depth-first compression, proposed in [6], to construct the L-tree layer. The basic idea is to prioritize computing nodes of the highest height. As long as the top node in the stack has the same height as the coming node (sibling nodes), it executes $thash_h$ (lines 12-16). The last node (66^{th}) is an exception. It keeps moving to the higher layer unless it finds another sibling node. Given the order 66^{th}, it can be predicted that the sibling nodes are located in layers 1 and 6 (lines 3-10). To further optimize the depth-first computation on AIE, AXIOS leverages the tree-level parallelism in the L-tree layer, *i.e.*, feature F5. By unrolling the loop (lines 12-16), each layer handles the node at a specific height, and these layers process nodes simultaneously.

Algorithm 4 AIE optimized L-tree layer

```
0:  procedure L-TREE(leaf, pk_{0,1,...,L-1}, Pub_seed, Addr)
1:  unsigned len = L;                          ▷ Depth first compression.
2:  for i = 0; i < len; i++ do
3:    node = pk_i
4:    if i == L-1 then
5:      Addr.setLtreeHeight(1);                ▷ Hash for the last node
6:      Addr.setLtreeIndex(ceil(len/2));
7:      node = thash_h(Addr,Pub_seed,Stack.pop(),node);
8:      Addr.setLtreeHeight(6);
9:      Addr.setLtreeIndex(0);
10:     node = thash_h(Addr,Pub_seed,Stack.pop(),node);
11:   else
12:     while Top node in Stack has the same height as node h'
        do
13:       Addr.setLtreeHeight(h');▷ Unrolled into multiple layer
          in AIE
14:       Addr.setLtreeIndex();
15:       node = thash_h(Addr,Pub_seed,Stack.pop(),node);
16:     end while
17:   end if
18:   Stack.push(node);
19: end for
20: return : leaf = Stack.pop();
```

The L-tree is implemented using tile-level parallelization and pipelining techniques and operates independently. The height of the L-tree can be inferred by taking the logarithm of the number of WOTS$^+_{pk}$ nodes, *e.g.*, a 7-layer L-tree is needed for processing 67 WOTS$^+_{pk}$ nodes. Instead of computing a tree layer by layer (as a single processor does), AXIOS schedules 7 $trash_h$ functions working in a pipeline, as demonstrated in Figure 6. Each $thash\_h$ function executes

$$node_{t+1\_i/2} \leftarrow thash_h(Addr, Pub\_seed, node_{t\_i}, node_{t\_i+1}) \tag{10}$$

by bundling five tiles with AXI streams, as discussed in Section 3.2.

An example operation is given. The output stream of the WOTS$^+$ public key (PK in Figure 5) goes into the first l-tree layer (H0). After H0 deals with 2 WOTS$^+$ pk nodes, *e.g.*, 0_0 and 0_1, it continues to process the following WOTS$^+$ pk nodes (0_2, 0_3) and sends the output node 1_0 to the upper layer (H1). Having the complete WOTS$^+$ public key processed through the l-tree pipeline flow, a leaf node of the Merkle tree is generated.

Typically, a general processor takes 7160 unit times the call of hash functions to obtain a leaf node from a secret seed, as equation 11 shows. It engages WOTS$^+$ secret key generation, calculation, and compression(via L-tree). Even using pre-computing technology as [35] does, it achieves a one-third reduction to $4752u$ in equation 12.

$$D_3 = 67 \times 2u + 67 \times 16 \times 6u + 66 \times 9u = 7160u \tag{11}$$

$$D_4 = 68u + 67 \times 16 \times 4u + 66 \times 6u = 4752u \tag{12}$$

However, with a highly pipelined and tile-level parallel computing pattern, the spatial architecture further reduces the unit time of the hash call to $204u$, as equation 13 shows.

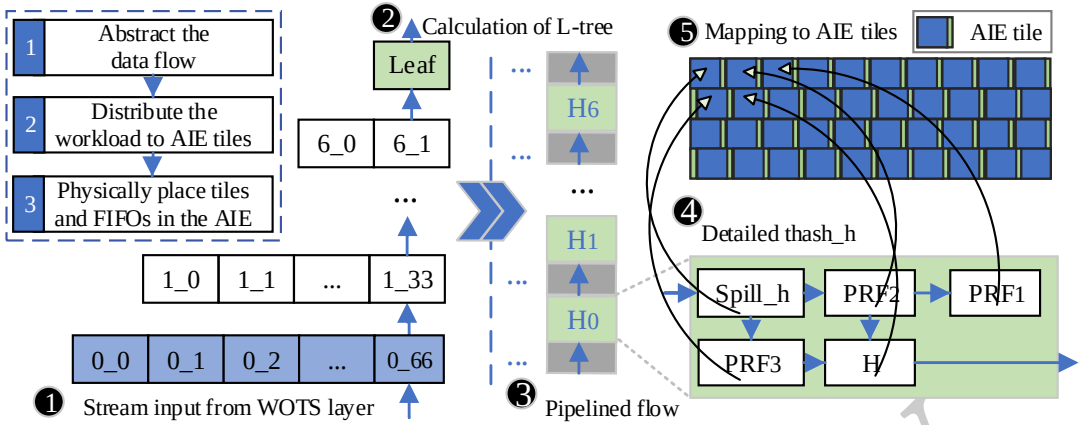

Figure 6: The L-tree design follows the overall development approach of XMSS, which involves abstracting the data, designing each function based on the stream flow, further dividing it into tile-implementable sub-functions, and placing the tiles in suitable locations with FIFO.

$$D_5 = 67 \times 2u + 60u + 2 \times 5u = 204u \tag{13}$$

When placing an L-tree to the AIE, 35 tiles are needed for a tree of height equal to 7. Tile placements for an L-tree should be constrained to optimize adjacent memory access. It should be noted that the current version of the AIE compiler provided by Xilinx may set different FIFOs to one memory bank, resulting in violations in the pipeline's timing constraints. To mitigate this issue, AXIOS provides an optimal AIE tile placement for the XMSS implementation.

3.5 Overall Hardware Implementation

The design approach introduced above complies with the overall XMSS constructing rules. It first abstracts the data flow of the component (*e.g.*, L-tree in Figure 6) before allocating the workload to AIE kernels. Then AXIOS configures kernels according to the workload, *e.g.*, specify the length of stream-in and stream-out. Next, the kernels are mapped to AIE tiles, which run the algorithm optimized. After that, AIE tiles are placed in the proper locations of the AI graph with suitable FIFO resources.

Figure 7 describes a top-level view of the placement of XMSS on the AI Engine and data flow between engines. To generate a leaf node of the Merkle tree, as mentioned before, AXIOS needs one tile to expand WOTS^+_{sk} (the grey blocks), a bundle of tiles to convert WOTS^+_{sk} to WOTS_pk (the orange blocks), and a bundle of tiles for turning WOTS^+_{pk} into leaf nodes (the green blocks). Overall, for the parameter set defined in Section 2.1.7, it takes **96** tiles to build a sub-graph (the yellow block) that calculates one leaf node of the Merkle tree.

Since AXIOS is implemented on a chip containing 400 tiles, it supports up to four leaf nodes in parallel. Hence, shimDMA broadcasts the public/secret seed to four seed expand tiles (the grey blocks in Figure 7), and computation starts simultaneously. The remaining tiles implement the Merkle layer kernel that accepts leaves from the L-tree layers and compresses them into the XMSS root.

As analyzed, the call of the hash function generating a leaf node is D_3 and D_4 for a general processor with or without pre-compute technology. Therefore, with parameter $h = 10$, it takes D_6 and D_7 to obtain an XMSS root.

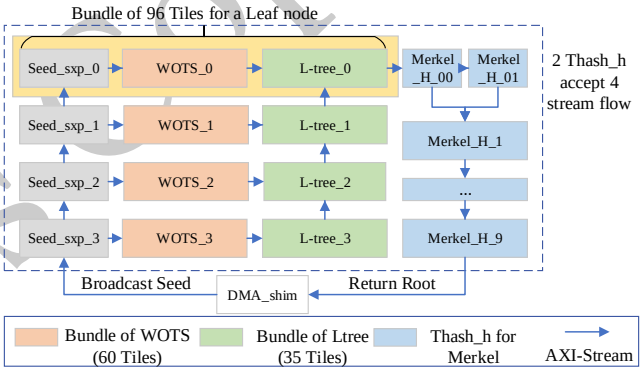

Figure 7: XMSS key generation process in AXIOS. AXIOS bundles seed_expand, WOTS$^+$, and L-tree as a group of leaf structures. Four sets of leaf structures are computed in parallel. Each tile allows two stream flows; thus, two $thash_h$ are needed in Merkle tree's layer_0 to accept four leaf nodes.

$$D_6 = D_3 \times 2^h + (2^h - 1) \times 9u = 7168911u \tag{14}$$

$$D_7 = D_4 \times 2^h + (2^h - 1) \times 6u = 4756994u \tag{15}$$

When looking into AXIOS, the spatial architecture calculates leaf nodes in tile-level parallelism with a pipelined Merkel structure, which eventually takes 51050u time to obtain the public key root. By utilizing the properties of the AIE graph, the proposed implementation achieves a 93.1$times$ theoretical speedup compared with a naive algorithm.

$$D_8 = \frac{D_5 \times 2^h}{4} + h \times 5u = 51050u \tag{16}$$

4 Experimental Results and Evaluation

4.1 Experimental Setup

We benchmark the performance of AXIOS on the XMSS key generation operation on Xilinx's VCK-190 platform containing the Versal AI Core XCVC1902-VSVA2179-2MP-ES chip. We measure

the timing performance of different designs by instrumenting the application code running on the host processor. We monitor and collect real-time power information with the Versal Power Tool [18]. Table 1 lists the configuration of the VCK-190 platform and the local workstation used in this work.

Table 1: Configurations of Versal ASoC and the local workstation.

Versal ASoC	XCVC1902-VSVA2179-2MP-ES Dual-core ARM Cortex-A72 8 GB DIMM DDR4 , 400 AIEs @1.25GHz
CPU	Model:Intel(R) Core(TM) i9-14900K 32 cores @800-6000MHz (Min-Max) 2.2 MiB L1-I/D, 32 MiB L2, 36MiB L3 2× 32GB DIMM DDR5 @4800MHz Ubuntu 22.04.1 x86_64, GNU bash 5.1.16(1)

4.2 Performance Analysis

In this section, we present the performance analysis of AXIOS when implemented in spatial computing architecture. We measure the performance based on computation runtime, power efficiency, and area requirements.

4.2.1 SHA-256 vs SHAKE-256. We first implement XMSS with two primitives, *i.e.*, SHAKE and SHA-2. Table 2 lists the runtime when executing different stages of XMSS. The kernel for SHAKE takes 3.5× more cycles compared to SHA-2 to calculate the PRF function. AXIOS can flexibly change the hash kernel, maintaining the computing pattern.

Table 2: Runtime of SHA-256 and SHAKE-256 on AIE for different stages of XMSS.

Alg.	SHA-256		SHAKE-256		Tiles
	Cycles	Time (μs)	Cycles	Time (μs)	
PRF	4342	3.5	15282	12.23	1
thash_f	18099	14.48	31012	24.81	4
WOTS^+	921902	737.5	1579646	1263.7	60
Leaf	1201422	968.3	4349638	3749	96
Root	1.63×10^8	**0.1303(s)**	5.89×10^8	**0.4715(s)**	395

4.2.2 Runtime Analysis. Table 3 compares the number of cycles required for running different stages of the XMSS algorithm on an AI engine and a high-end CPU separately. Intuitively, as the algorithm becomes more complex, the cycle count increases, and the number of tiles required by the AIE increases. Overall, for the complete execution of the XMSS algorithm, AIE outperforms the CPU by nearly a factor of **8**.

We observe that for calculating a single and multiple SHA-256 implementation, the CPU consistently requires about 550 *us*, whereas the time needed for an AI tile increases linearly. This is due to the sequential nature of the mapping loop in lines 11-13 in Algorithm 1. This helps us estimate the performance of the AI tiles before implementing the complete design in the AIE. Given the baseline computation in Section 3.3, one would require $192 \times u = 672$ *us* for the WOTS^+ public key generation. In reality, to synchronize the data movement between tiles, the actual computation time is slightly bigger (737.5*us*) than estimated.

Table 3: Runtime of the AI engine and the local workstation CPU when executing different stages of XMSS.

Alg.	Imp.	Cycles	Freq.(MHz)	Time (μs)	Tiles
SHA-256	AIE	4342	1250	3.5	1
	CPU	1768957	3100	556	-
SHA-256 (×100)	AIE	502394	1250	401.9	1
	CPU	1825563	3100	574	-
thash_f (×100)	AIE	1809947	1250	1448.0	4
	CPU	1949783	3100	617	-
WOTS^+	AIE	921902	1250	737.5	60
	CPU	4928659	3100	1551	-
Leaf	AIE	1201422	1250	968.3	96
	CPU	4950662	3100	1566	-
Root	AIE	1.63×10^8	1250	**0.1303**(s)	395
	CPU	3.37×10^9	3100	1.11(s)	-

4.2.3 Power Measurements. We record the power consumption of the ASoC chip with/without running the proposed XMSS accelerators as shown in Figure 4. We observe that the static power is about 25.8W, most of which is PL domain power as VCK-190 platform supports a significant number of hardware resources. When launching the XMSS application, the total power consumption rises to 30.2W, *i.e.*, the execution of AXIOS requires only an additional by 4.4 Watts of power consumption. Noting we utilized almost all of the AI tiles, illustrating that the maximum dynamic power of the AIE-based application is about 30 Watts. Among them, one-sixth is used for AIE activities, while the rest of the power consumption supports the whole device.

Table 4: Power consumption of AXIOS. The middle column records the static power of the ASoC; The right column records the dynamic power of the ASoC when executing AXIOS.

Domain	Volts (Static Power)	Amps (Static Power)	Watts (Static Power)	Volts (Dynamic Power)	Amps (Dynamic Power)	Watts (Dynamic Power)
FMC	1.502	0.006	0.0094	1.50	0.0050	0.008
Transceiver	-	-	0.281	-	-	0.268
PL	-	-	19.891	-	-	24.357
System	0.810	6.130	4.965	0.809	6.010	4.861
FPD	0.799	0.185	0.147	0.800	0.190	0.152
LPD	0.799	0.223	0.178	0.799	0.220	0.176
PMC	-	-	0.357	-	-	0.354
Total	-	-	**25.828**	-	-	**30.174**

4.2.4 Area Requirements. Trivial hardware resources, *e.g.*, LUT and FF are used to build communication channels between the PS side and the AIE graph. The overload of XMSS computation falls on the AIE graph.

Figure 8 depicts the complete layout of the AXIOS architecture placed on the AIE of the Versal VCK-190 chip. The sub-stages such as the seed expansion, WOTS^+, L-tree, and Merkle tree require $1 \times 4 = 4$, $60 \times 4 = 240$, $35 \times 4 = 140$, and 11 tiles respectively. Overall, an XMSS tree consumes 395 tiles. The allocation of tiles complies with the computing complexity of different components of XMSS.

We also give out the hardware resources consumed on the PL side when implementing different stages of XMSS as Table 6 lists. With more and more sub-modules assembling for a higher-level structure,

Table 5: Performance of XMSS key generation targeting different platforms. Parameter $\{n, h, w\}$ is set to $\{32, 10, 16\}$

Design	Platform	Cycles ($\times 10^7$)	Freq. (MHz)	Time (s)	Energy(J) Time×Power	Ratio speed/energy
Benchmark	i9-14900K	337	3100	1.11	280.83	1/1
[35]	Murax SoC	2830	152	186	-	0.006/-
[35]	Murax SoC	32.3	93	3.44	-	0.32/ -
[6]	Artix-7	3.17	110	2.88	-	0.35/ -
[33] Minimal	Artix-7	-	100	4.63	-	0.24/ -
[33] Time/Area	Artix-7	-	100	1.68	-	0.66/-
[33] Speed	Artix-7	-	95	0.77	-	1.44/ -
[5]	e5-2650	509	2300	2.21	232.05	0.50/ 1.21
[5] D-1	Artix-7	3.9	102	0.39	-	2.83/-
[5] D-2	Artix-7	4.0	101	0.41	-	2.73/ -
AXIOS	VCK-190	16.3	1250	**0.13**	**3.92**	**8.54/ 71.65**

Speedup = Benchmark/ Time.
Ratio = Benchmark / Energy.

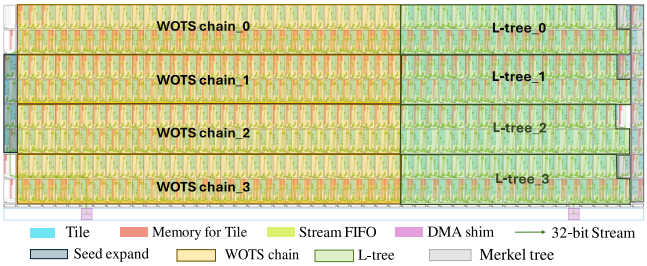

Figure 8: AXIOS layout in AMD's Versal platform. The example implementation presented in this work uses 395 AI engines in the Versal XCVC1902 chip.

hardware resources consumption, *e.g.*, LUTs and Flip-Flops, keeps rising.

Table 6: Area consumption of different modules of XMSS targeting AIE and PL, respectively.

Algorithm	PL(LUTs/FFs/Slices)	AIE(Tiles)
SHA-256	1461/1554/452	1
thash_f	2874/3106/1309	4
thash_h	2037/3622/1388	5
WOTS^+	3028/3591/1536	60
L-tree	3732/7256/2404	35
Leaf	6939/10394/3091	96
XMSS	15738/21638/4820	395

4.3 Comparison with Related Works

Table 3 lists AXIOS's performance and compares AXIOS with recently reported XMSS implementations on embedded devices with parameter $\{n, h, w\} = \{32, 10, 16\}$. It should be noted that most of the work focuses on delay and resource consumption but has not reported details about energy efficiency.

We implement XMSS key generation on the local workstation as a benchmark application since it performs better than several embedded platforms. The benchmark runs the standard reference code provided on [27]. Wang *et al.* [35] run XMSS on Murax, a kind of RISC-V core, which takes 186 seconds to finish whole operations. Then, they propose a software-hardware co-design that improves 54× than running XMSS on Murax. Thoma *et al.* [33] also implement XMSS in software-hardware co-design. Their contribution mainly lies in exploring the trade-offs between the area and runtime performances for the WOTS^+ accelerator. Cao *et al.* [6] first propose a full hardware implementation of XMSS. In [5], Cao *et al.* takes advantage of hardware resources and implements XMSS with multiple hash cores, outperforming the benchmark by 2.83×. They also propose a multi-core implementation whose efficiency focuses on speed and resource consumption rather than a speed-power trade-off. AXIOS achieves the best performance amongst embedded platforms with **8.54**× speed up over the benchmark. Moreover, AXIOS presents **71.65**× improvement in power efficiency over the stock Intel(R) Core(TM) i9-14900K processor.

Interestingly, FPGA-based designs always take fewer cycles than running XMSS on processors. However, the frequency of FPGA-based implementations is restricted to around several 100 MHz due to various factors, *e.g.*, large combinations of logic gates and high fan-outs in their designs. AXIOS works on a spatial computing architecture where the main computing component, *i.e.*, AIE, works at a significantly higher speed (1.25 GHz) than FPGAs. Moreover, AXIOS substantially reduces the computing cycles by designing a highly pipelined computing pattern. AXIOS outperforms the fastest XMSS hardware accelerator [5] by 3×. Overall, AXIOS achieves the best performance reported so far in terms of speed and energy efficiency on embedded platforms.

5 Conclusions & Future Works

This paper demonstrates that spatial architectures commonly found in modern heterogeneous SoCs are promising in handling demanding cryptographic workloads. We choose standard PQC algorithms XMSS and build accelerators for computationally expensive operations, i.e., key generation. As a result, our optimized accelerator, AXIOS, outperforms a modern CPU by 8× in terms of runtime and 71× in terms of power efficiency for handling XMSS. Most of the design optimization for AXIOS was hand-crafted, including pipeline design, intrinsics optimization, FIFO setup, and AIE placement. Therefore, future opportunities exist to automate complex cryptographic accelerator design on FPGA+CGRA platforms. This work should inspire further investigation into these issues to effectively utilize emerging spatial accelerators in modern-day cryptography.

References

[1] Ahmad Al-Zoubi, Gianluca Martino, Fin H. Bahnsen, Jun Zhu, Holger Schlarb, and Goerschwin Fey. 2022. CNN Implementation and Analysis on Xilinx Versal ACAP at European XFEL. In *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. 1–6. doi:10.1109/SOCC56010.2022.9908101

[2] AMD. 2020. Versal: The First Adaptive Compute Acceleration Platform. https://docs.amd.com/v/u/en-US/wp505-versal-acap

[3] Rachid El Bansarkhani, Matthias Geihs, and Johannes Buchmann. 2018. PQChain: Strategic Design Decisions for Distributed Ledger Technologies against Future Threats. *IEEE Security & Privacy* 16, 4 (July 2018), 57–65. doi:10.1109/MSP.2018.3111246 Conference Name: IEEE Security & Privacy.

[4] Johannes Buchmann, Erik Dahmen, and Michael Schneider. 2008. Merkle Tree Traversal Revisited. In *Post-Quantum Cryptography*, Johannes Buchmann and Jintai Ding (Eds.). Springer, Berlin, Heidelberg, 63–78. doi:10.1007/978-3-540-88403-3_5

[5] Yuan Cao, Yanze Wu, Lan Qin, Shuai Chen, and Chip-Hong Chang. 2022. Area, Time and Energy Efficient Multicore Hardware Accelerators for Extended Merkle Signature Scheme. *IEEE Transactions on Circuits and Systems I: Regular Papers* 69, 12 (Dec. 2022), 4908–4918. doi:10.1109/TCSI.2022.3200987

[6] Yuan Cao, Yanze Wu, Wen Wang, Xu Lu, Shuai Chen, Jing Ye, and Chip-Hong Chang. 2022. An Efficient Full Hardware Implementation of Extended Merkle Signature Scheme. *IEEE Transactions on Circuits and Systems I: Regular Papers* 69, 2 (Feb. 2022), 682–693. doi:10.1109/TCSI.2021.3115786

[7] Paul Chen, Pavan Manjunath, Sasindu Wijeratne, Bingyi Zhang, and Viktor Prasanna. 2023. Exploiting On-Chip Heterogeneity of Versal Architecture for GNN Inference Acceleration. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. 219–227. doi:10.1109/FPL60245.2023.00038 ISSN: 1946-1488.

[8] CNN. 2023. CNN Interacts with Cutting-Edge AI Technology. *CNN* (15 11 2023). https://www.cnn.com/2023/11/15/tech/cnn-ai-technology-interaction Accessed: 2023-11-20.

[9] Information Technology Laboratory Computer Security Division. 2017. Selected Algorithms 2022 - Post-Quantum Cryptography | CSRC | CSRC. https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms-2022

[10] David Cooper, Daniel Apon, Quynh Dang, Michael Davidson, Morris Dworkin, and Carl Miller. 2020. *Recommendation for Stateful Hash-Based Signature Schemes*. Technical Report NIST Special Publication (SP) 800-208. National Institute of Standards and Technology. doi:10.6028/NIST.SP.800-208

[11] Tuo Dai, Bizhao Shi, and Guojie Luo. 2024. WideSA: A High Array Utilization Mapping Scheme for Uniform Recurrences on the Versal ACAP Architecture. doi:10.48550/arXiv.2401.16792 arXiv:2401.16792 [cs].

[12] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1986. How to construct random functions. *J. ACM* 33, 4 (1986), 792–807. doi:10.1145/6490.6503

[13] M. Greenspan, L. Pham, and N. Tardella. 1998. Development and evaluation of a real time SAR ATR system. In *Proceedings of the 1998 IEEE Radar Conference, RADARCON'98. Challenges in Radar Systems and Solutions (Cat. No.98CH36197)*. 38–43. doi:10.1109/NRC.1998.677974

[14] Linley Gwennap. 2020. Groq Rocks Neural Networks. *Microprocessor Report, Tech. Rep., jan* (2020).

[15] Aneas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. 2018. *XMSS: eXtended Merkle Signature Scheme*. Request for Comments RFC 8391. Internet Engineering Task Force. doi:10.17487/RFC8391 Num Pages: 74.

[16] Johan HÅstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. 1999. A Pseudorandom Generator from any One-way Function. *SIAM J. Comput.* 28, 4 (Jan. 1999), 1364–1396. doi:10.1137/S0097539793244708

[17] Andreas Hülsing. 2013. W-OTS+ — Shorter Signatures for Hash-Based Signature Schemes, Vol. 7918. 173–188. doi:10.1007/978-3-642-38553-7_10

[18] Jerrywo. 2023. Xilinx/jupyter-pat. https://github.com/Xilinx/jupyter-pat original-date: 2021-05-26T14:23:11Z.

[19] Xijie Jia, Yu Zhang, Guangdong Liu, Xinlin Yang, Tianyu Zhang, Jia Zheng, Dongdong Xu, Zhuohuan Liu, Mengke Liu, Xiaoyang Yan, Hong Wang, Rongzhang Zheng, Li Wang, Dong Li, Satyaprakash Pareek, Jian Weng, Lu Tian, Dongliang Xie, Hong Luo, and Yi Shan. 2024. XVDPU: A High-Performance CNN Accelerator on the Versal Platform Powered by the AI Engine. *ACM Transactions on Reconfigurable Technology and Systems* 17, 2 (2024), 20:1–20:24. doi:10.1145/3617836

[20] Dustin Kern, Christoph Krauß, Timm Lauser, Nouri Alnahawi, Alexander Wiesmaier, and Ruben Niederhagen. 2023. QuantumCharge: Post-Quantum Cryptography for Electric Vehicle Charging. In *Applied Cryptography and Network Security*, Mehdi Tibouchi and XiaoFeng Wang (Eds.). Springer Nature Switzerland, Cham, 85–111. doi:10.1007/978-3-031-33491-7_4

[21] Vinay B. Y. Kumar, Naina Gupta, Anupam Chattopadhyay, Michael Kasper, Christoph Krauß, and Ruben Niederhagen. 2020. Post-Quantum Secure Boot. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1582–1585. doi:10.23919/DATE48585.2020.9116252 ISSN: 1558-1101.

[22] Ralph C. Merkle. 1990. A Certified Digital Signature. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, Gilles Brassard (Ed.). Springer, New York, NY, 218–238. doi:10.1007/0-387-34805-0_21

[23] Prashanth Mohan, Wen Wang, Bernhard Jungk, Ruben Niederhagen, Jakub Szefer, and Ken Mai. 2020. ASIC Accelerator in 28 nm for the Post-Quantum Digital Signature Scheme XMSS. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. 656–662. doi:10.1109/ICCD50377.2020.00112 ISSN: 2576-6996.

[24] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. 2016. Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC. In *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, 77–84.

[25] Omar Ragheb, Stephen Wicklund, Matthew Walker, Rami Beidas, Adham Ragab, and Jason Anderson. 2024. CGRA-ME 2.0: A Research Framework for Next Generation CGRA Architectures and CAD. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 642–649.

[26] Tiago David Sousa Ramos. 2023. Implementation of Convolutional Neural Networks on a Versal Device. (2023).

[27] XMSS Reference. 2024. XMSS/xmss-reference. https://github.com/XMSS/xmss-reference original-date: 2017-07-21T12:43:34Z.

[28] Phillip Rogaway and Thomas Shrimpton. 2004. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In *Fast Software Encryption*, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Bimal Roy, and Willi Meier (Eds.). Vol. 3017. Springer Berlin Heidelberg, Berlin, Heidelberg, 371–388. doi:10.1007/978-3-540-25937-4_24 Series Title: Lecture Notes in Computer Science.

[29] J. Rompel. 1990. One-way Functions are Necessary and Sufficient for Secure Signatures. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing - STOC '90*. ACM Press, Baltimore, Maryland, United States, 387–394. doi:10.1145/100216.100269

[30] Furqan Shahid, Abid Khan, and Gwanggil Jeon. 2020. Post-quantum Distributed Ledger for Internet of Things. *Computers & Electrical Engineering* 83 (May 2020), 106581. doi:10.1016/j.compeleceng.2020.106581

[31] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Bala Jayadev, Jeff Cuppett, Abbas Morshed, Brian Gaide, and Ygal Arbel. 2019. Versal Network-on-Chip (NoC). In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*. 13–17. doi:10.1109/HOTI.2019.00016 ISSN: 2332-5569.

[32] Endri Taka, Aman Arora, Kai-Chiang Wu, and Diana Marculescu. 2023. MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine. In *2023 International Conference on Field Programmable Technology (ICFPT)*. 96–105. doi:10.1109/ICFPT59805.2023.00016 ISSN: 2837-0449.

[33] Jan Philipp Thoma and Tim Guneysu. [n. d.]. A Configurable Hardware Implementation of XMSS. ([n. d.]).

[34] Jasmina Vasiljevic, Ljubisa Bajic, Davor Capalija, Stanislav Sokorac, Dragoljub Ignjatovic, Lejla Bajic, Milos Trajkovic, Ivan Hamer, Ivan Matosevic, Aleksandar Cejkov, et al. 2021. Compute substrate for Software 2.0. *IEEE micro* 41, 2 (2021), 50–55.

[35] Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. 2020. XMSS and Embedded Systems: XMSS Hardware Accelerators for RISC-V. In *Selected Areas in Cryptography – SAC 2019*, Kenneth G. Paterson and Douglas Stebila (Eds.). Vol. 11959. Springer International Publishing, Cham, 523–550. doi:10.1007/978-3-030-38471-5_21 Series Title: Lecture Notes in Computer Science.

[36] Ziheng Wang, Xiaoshe Dong, Heng Chen, and Yan Kang. 2023. Efficient GPU Implementations of Post-Quantum Signature XMSS. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (March 2023), 938–954. doi:10.1109/TPDS.2022.3233348

[37] Victor van Wijhe. 2024. Signal Processing with AMD Adaptive Compute Acceleration Platform (ACAP) for Applications in Radio Astronomy. http://essay.utwente.nl/98354/ Publisher: University of Twente.

[38] Zhuoping Yang, Jinming Zhuang, Jiaqi Yin, Cunxi Yu, Alex K. Jones, and Peipei Zhou. 2023. AIM: Accelerating Arbitrary-Precision Integer Multiplication on Heterogeneous Reconfigurable Computing Platform Versal ACAP. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 1–9. doi:10.1109/ICCAD57390.2023.10323754 ISSN: 1558-2434.

[39] Chengming Zhang, Tong Geng, Anqi Guo, Jiannan Tian, Martin Herbordt, Ang Li, and Dingwen Tao. 2022. H-GCN: A Graph Convolutional Network Accelerator on Versal ACAP Architecture. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, Belfast, United Kingdom, 200–208. doi:10.1109/FPL57034.2022.00040

[40] Kaiyi Zhang, Hongrui Cui, and Yu Yu. 2023. Revisiting the Constant-sum Winternitz One-time Signature with Applications to SPHINCS+ and XMSS. In *Annual International Cryptology Conference*. Springer, 455–483.

[41] Wenbo Zhang, Tianshuo Wang, Yiqi Liu, Yiming Li, and Zhenshan Bao. 2024. New Filter2D Accelerator on the Versal Platform Powered by the AI Engine. In *Advanced Parallel Processing Technologies*, Chao Li, Zhenhua Li, Li Shen, Fan Wu, and Xiaoli Gong (Eds.). Springer Nature, Singapore, 437–449. doi:10.1007/978-981-99-7872-4_24

[42] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. 2023. CHARM: Composing Heterogeneous Accele Rators for Matrix Multiply on Versal ACAP Architecture. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, Monterey CA USA, 153–164. doi:10.1145/3543622.3573210

[43] Jinming Zhuang, Zhuoping Yang, and Peipei Zhou. 2023. High Performance, Low Power Matrix Multiply Design on ACAP: from Architecture, Design Challenges and DSE Perspectives. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. doi:10.1109/DAC56929.2023.10247981