MARS: A General GPU Optimization Framework for Merkle-Tree–Enabled Cryptography

Yaoyun Zhou[1], Qian Wang[2]
[1]Department of EECS, [2]Department of Electrical Engineering, University of California, Merced, CA, USA
yzhou96@ucmerced.edu, qianwang@ucmerced.edu

***Abstract*—Merkle trees underpin diverse cryptographic systems, from hash-based signatures (e.g., LMS, XMSS) to blockchain (e.g., Bitcoin, Ethereum), post-quantum cryptography (e.g., SPHINCS$^+$), and zero-knowledge protocols. Their strength lies in providing efficient verification with compact authentication paths, making them a fundamental primitive in modern cryptographic systems. Existing acceleration of Merkle trees primarily focuses on assigning parallel processing units (e.g., PEs or CUDA cores) to leaf nodes. However, due to the inherent reduction structure, the number of active computations decreases by half at each upper level, leading to progressively lower hardware utilization, especially for trees with greater depth. Furthermore, existing resource allocation strategies for Merkle tree computations rely on heuristic theoretical approximations, which tend to converge to suboptimal configurations. To address these challenges, we propose MARS, a general GPU-based optimization framework[1], and use the Merkle tree–intensive FORS (Forest of Random Subset) component in SPHINCS$^+$ as a representative case study. MARS leverages the fixed parameters of FORS as guidance to increase the computational load of individual warps, accelerates the tree-reduction process by minimizing shared memory usage and utilizing registers more effectively, and finally, inspired by automated tuning, applies profiling-guided parallel optimization across multiple Merkle trees to achieve architecture-portable configurations. Compared to the latest GPU-optimized FORS signature generation on RTX4090, MARS achieves signature generation throughput improvements of $2.52\times$, $2.39\times$, and $2.33\times$ under three different f-parameter sets.**

***Index Terms*—Merkle Tree, SPHINCS$^+$, GPU, Post-quantum cryptography, Auto Tuning**

I. Introduction

Merkle trees, originally proposed by Merkle in 1979 [1], provide a compact and efficient method for data integrity verification. Their applications span a wide range of domains, from classical hash-based signatures such as the leighton-micali signature system (LMS) and the extended merkle signature scheme (XMSS), to post-quantum digital signatures like SPHINCS$^+$, and even large-scale distributed systems including Bitcoin, Ethereum, and zero-knowledge proofs (ZKP) [2] [3] [4] [5] [6] [7]. In these diverse contexts, Merkle trees provide compact authentication paths and scalable verification, making them indispensable to modern cryptographic systems.

Despite their wide applicability, the computation of Merkle trees introduces significant performance challenges. In hash-based post-quantum cryptography (PQC), such as SPHINCS$^+$, many layered subtrees must be built, resulting in heavy hashing operations and high memory demands. On the other hand, blockchain systems must process large-scale transaction data in real time, where inefficient tree computation may bottleneck system throughput. Existing efforts, whether hardware accelerators or software-based optimizations, have primarily focused on breadth-first strategies to maximize the parallel efficiency of Merkle tree computations. However, as the tree height h increases, the required memory and computing units grow exponentially as $\mathcal{O}(2^h)$. For example, in zero-knowledge proof systems, a Merkle tree of height 64 would demand on the order of 10^{10} GB of memory, even if each node were represented by only one byte, the overhead that is infeasible for both DRAM and the more costly SRAM storage.

TABLE I
Comparison of Merkle tree *counts* and *heights* across representative Merkle-based cryptographic systems.

System	Merkle Tree	Typical Height(s)
LMS	1	5, 10, 15, 20, 25
XMSS	1	10, 16, 20
FORS	33, 35[a]	6, 8, 9[a]
Bitcoin	1 (per block)	$\lceil \log_2 N \rceil$[b]; typically 10–12
Ethereum	1 (per block)	$\lceil \log_2 N \rceil$[b]; typically 12–14
ZKP	1 (global/state tree)	32, 64

[a] SPHINCS+ parameters vary by security level/variant; common FORS counts k include 17, 33, 35, with small-tree heights such as 5, 6, or 9.
[b] N is the number of transactions in a block.

Existing hardware accelerators for Merkle tree computations, such as ASIC and FPGA designs, have achieved significant performance improvements through critical-path optimization, pipelined architectures, and the use of higher-bandwidth memory technologies like SRAM and HBM [8]–[12]. However, those hardware-based implementations must account for fabrication yield and the associated cost overheads. In contrast, GPUs offer a programmable CUDA environment that supports flexible core scheduling and efficient shared memory use. With thousands of parallel CUDA cores, GPUs can process cryptographic workloads much like ASICs with multiple processing elements. By aligning the crypto algorithm's computation logic with GPU features, we design a optimized framework that is well-suited for Merkle-tree-based algorithms while maintaining portability across different GPU architectures.

In this work, we first use the FORS (Forest of Random Subset), a set of Merkle trees of SPHINCS$^+$, as our optimization example. It is the only multi-tree structure shown in Table I, making it well-suited for validating the effectiveness of our

[1]The source code of MARS is available at https://github.com/yaoyunzhou/MARS.

978-3-9826741-1-7/DATE26/© 2026 EDAA

optimizations. However, our proposed optimization method can also be applied to other schemes to balance throughput and GPU resource utilization.

This paper presents the following contributions:

- We analyze the parallelism potential of FORS first at the single-tree level. Specifically, for a single Merkle tree, we employ a warp-centric "load stretching" strategy that increases per-warp work and optimizes the warp–thread setup by utilizing the selective registers.
- We extend an odd–even interleaved access pattern across registers and shared memory to enable buffer reuse. By slightly increasing register usage to reduce shared memory demand, we minimize resource contention and improve signature generation throughput.
- Building on single-tree tuning, we develop a profiling-guided multi-tree optimization using an unrolling-constrained search strategy. This mathematically justified pruning reduces search time by two orders of magnitude. It demonstrates architecture-portable features from Ada (RTX4090) to Pascal, Turing, and Hopper, achieving 2.33×–2.52× speedups on FORS over state-of-the-art GPU baselines.

II. Preliminaries

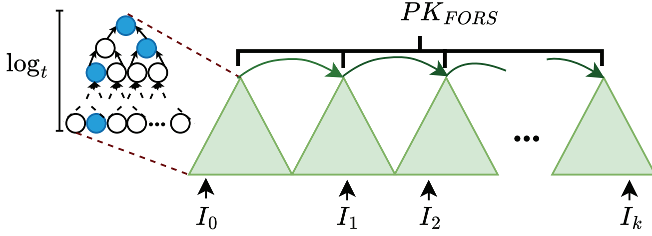

Fig. 1. The Overview of FORS Structure. Blue circles indicate nodes included in the signature within each Merkle tree; inter-tree arrows depict the conventional sequential computation order.

A. Merkle Tree

Let $H : \{0,1\}^* \to \{0,1\}^d$ be a collision-resistant hash. A Merkle tree over n leaves $L_0, \dots, L_{n-1}$ defines leaf nodes $v_{0,j} = H(\text{LeafEncode}(L_j))$ and internal nodes

$$v_{i,j} = H\big(\text{DomSep}_i \,\|\, v_{i-1,2j} \,\|\, v_{i-1,2j+1}\big) \quad (i = 1, \dots, h),$$

where $h = \lceil \log_2 n \rceil$ and the root $r = v_{h,0}$ binds the entire set. An authentication path for index w is the sequence of sibling nodes along the path to r; verification recomputes $u_0 = H(\text{LeafEncode}(L_t))$ and iteratively

$$u_{k+1} = H\big(\text{ord}(u_k, a_k)\big)$$

to recover r, where $\text{ord}(\cdot, \cdot)$ concatenates the current node with its sibling a_k in the correct left/right order. A Merkle tree organizes n leaves into a hierarchy of hashed nodes, where each internal node is derived by hashing its two child nodes, and the final root hash uniquely represents the entire dataset. To prove membership of a leaf, an authentication path provides the sibling nodes along the route to the root. The verifier recomputes the intermediate hashes step by step until reaching the root, confirming data integrity.

In the binary case, the proof size is $h \cdot d$ bits and verification costs h hash evaluations; construction hashes approximately $n - 1$ internal nodes. In practice, leaves may be derived from per-index PRFs and domain-separated hashes (e.g., XMSS/SPHINCS$^+$), or encode transactions/commitments (e.g., blockchains/ZKP), while the verification interface remains identical.

B. FORS

FORS (*Forest of Random Subsets*) is a few-time signature used inside SPHINCS$^+$. Let $H : \{0,1\}^* \to \{0,1\}^d$ be a hash and denote the scheme by $\text{FORS}(k, \log_t)$, comprising k independent binary Merkle trees each of height $\log_t$ (thus t leaves per tree). The signer derives per-leaf secrets $s_{j,i} = \text{PRF}(\mathsf{sk_seed}, \mathsf{addr}_{j,i})$ and computes leaves $\ell_{j,i} = H(\text{DS}_{\text{leaf}} \,\|\, \mathsf{pk_seed} \,\|\, \mathsf{addr}_{j,i} \,\|\, s_{j,i})$, where domain separation and the address scheme ensure uniqueness. To sign a message M, a message-digest mapping deterministically parses M into indices $I_0, \dots, I_{k-1} \in [0, t-1]$; the signature reveals the k corresponding secrets s_{j,I_j} together with their authentication paths π_j of length a. Verification recomputes ℓ_{j,I_j} from each revealed secret, rolls up along π_j to obtain roots r_j, and derives the FORS public key $\mathsf{pk}_{\text{FORS}} = H(\text{DS}_{\text{pk}} \,\|\, \mathsf{pk_seed} \,\|\, \mathsf{addr}_{\text{root}} \,\|\, r_0 \| \cdots \| r_{k-1})$; this key is then propagated to upper layers in SPHINCS$^+$. The signature size is $k(log_t + 1)d$ bits (one leaf secret plus log_t siblings per tree) and verification requires $\mathcal{O}\big(k(\log_t + 1)\big)$ hash calls, exposing two natural parallelism axes, across the k trees and across nodes along each authentication path.

C. GPU Architecture

Modern GPUs comprise many streaming multiprocessors (SMs), each executing threads in groups called *warps* (typically 32 threads) under a SIMT model in which one instruction is issued per warp. A CUDA kernel launches a grid of thread blocks; blocks are scheduled onto SMs, and each block's warps are time-multiplexed by warp schedulers to hide long-latency operations (e.g., global-memory loads) by rapidly switching to ready warps. Synchronization is hierarchical: warp-level barriers (e.g., `__syncwarp(mask)`) coordinate threads within a warp, while block-level barriers (e.g., `__syncthreads()`) coordinate all threads in a block; cross-block synchronization typically relies on kernel boundaries or cooperative groups. The memory hierarchy spans per-thread registers and spill "local" memory, per-block on-chip shared memory (banked, low latency), on-chip L1/texture/constant caches, a chip-wide L2, and off-chip global DRAM (often HBM). Performance hinges on sufficient occupancy to expose many runnable warps for latency hiding, and judicious use of shared memory (static or dynamic) to stage data for reuse within a block.

III. Methodology

A. Inherent parallelism of FORS Structure

In FORS, each subtree performs a level-wise reduction: at level ℓ, parent nodes $v_{\ell,j}$ are computed by concatenating

and hashing disjoint pairs $(v_{\ell-1,2j}, v_{\ell-1,2j+1})$, i.e., $v_{\ell,j} = H\big(v_{\ell-1,2j} \,\|\, v_{\ell-1,2j+1}\big)$. All pairwise computations within the same level are mutually independent, exposing abundant SIMT parallelism. Across subtrees, the k indices $I_0, \ldots, I_{k-1}$ are deterministically derived from the message digest prior to signing; consequently, leaf-secret derivation, leaf hashing, and authentication-path construction for each subtree proceed independently. Hence, absent resource constraints (and aside from the inherent cross-level dependency within a single tree), FORS admits full parallelism both intra-level and across trees, making it an ideal target for GPU mapping.

B. GPU Resource Constraints and Launch-Time Immutability

On NVIDIA GPUs, a kernel's launch geometry is immutable: the grid and block dimensions are fixed at invocation and cannot change during execution. This directly impacts Merkle-tree mapping. If a single tree is mapped with "one thread per leaf" inside a block, the architectural limit of $1024 = 2^{10}$ threads per block implies $2^h \leq 1024 \quad \Rightarrow \quad h \leq 10$. While on-chip SRAM (shared memory) can substantially accelerate tree-reduction staging and signature-path recording, occupancy is bounded by per-SM resources. Let S_{SM} and R_{SM} denote the shared-memory and register budgets per SM, and let a block use s_b bytes of shared memory, t_b threads, and r_t registers per thread. The number of active blocks per SM satisfies $n_b \leq \min\left(\left\lfloor \frac{S_{\mathrm{SM}}}{s_b} \right\rfloor, \left\lfloor \frac{R_{\mathrm{SM}}}{r_t\, t_b} \right\rfloor, \left\lfloor \frac{T_{\mathrm{SM}}}{t_b} \right\rfloor, B_{\mathrm{SM}}\right)$. where T_{SM} is the maximum resident threads per SM and B_{SM} is the architectural cap on resident blocks. Over-allocating shared memory (s_b) or registers (r_t) depresses n_b, reducing runnable warps and latency hiding.

A second inefficiency arises from the tree reduction itself. With $N = 2^h$ leaves, level ℓ (0-indexed) performs $N/2^\ell$ pairwise hashes. Under fixed launch geometry, the fraction of actually active threads at level ℓ is $\eta_\ell = \frac{N/2^\ell}{N} = 2^{-\ell}$, so the average intra-kernel utilization across the h levels is

$$\bar{\eta} = \frac{1}{h}\sum_{\ell=0}^{h-1} 2^{-\ell} = \frac{2}{h}\big(1 - 2^{-h}\big) \approx \frac{2}{h},$$

which decays with tree height. These constraints: block thread limits, shared-memory/register budgets, and launch-time immutability cause level-wise under-utilization, which motivates a balanced mapping and scheduling strategy rather than naively binding threads one-to-one to leaves.

TABLE II
KCS-24 OCCUPANCY AND REGISTER USAGE PER THREAD ACROSS SPHINCS$^+$ f-PARAMETER SETS PROFILED BY NSIGHT COMPUTE WHEN BLOCK NUMBER = 1024. CT/MT: THE COMPUTE-TO-MEMORY THROUGHPUT RATIO.

Scheme	**128f**	**192f**	**256f**
Warp Occupancy	26%	32.30%	31.06%
Theoretical Occupancy	66.67%	33.33%	33.33%
Registers Per Thread	64	128	96
CT / MT	4.01	4.38	2.05

C. A warp-centric "load stretching" strategy

Profiling existing GPU FORS implementations shows modest warp occupancy despite ample headroom in registers; a $\mathrm{CT}/\mathrm{MT} > 1$ further indicates a compute-bound regime dominated by Merkle hashing shown in Table II [13]. However, as the tree grows deeper, memory bandwidth dominates, discouraging heavy shared-memory staging per block (e.g., on RTX 4090 (SM 89), the maximum per-block static shared memory is 48 KB by default). To raise effective utilization and curb warp idling, we *coarsen the mapping*: each thread processes a bundle of g nodes per level. This yields three effects: (i) it *reduces the launched thread count* by a factor of g, i.e., from $T_{\text{launch}} = N$ to $T_{\text{launch}} = N/g$, thereby avoiding inactive lanes as the reduction proceeds; (ii) it *shrinks the effective reduction depth* to $h_{\text{eff}} = h - \log_2 g$, so the average intra-kernel utilization improves from $\bar{\eta} \approx 2/h$ to $\bar{\eta}_{\text{eff}} \approx 2/h_{\text{eff}}$; and (iii) it *cuts synchronization frequency*, reducing level-wise synchronization points from h to h_{eff}. With fewer, busier threads, arithmetic intensity per thread increases while idle-thread occupancy is reduced; the resulting per-block resource footprint can also enable more resident blocks per SM under shared-memory and register budgets. We could tune g to balance added register pressure and locality against occupancy.

D. Memory Optimization

During FORS tree reduction, the dominant operations are leaf-address offsetting, SHA hashing, and frequent loads/stores. We first target a single Merkle tree and rebalance the workload so that the compute-to-memory ratio approaches unity rather than remaining strictly compute-bound. Concretely, intermediate results from each thread's node bundle are kept *register-resident* to maximize reuse (with bandwidth hierarchy: registers(220-300+TB/s), shared memory(10–20 TB/s), global memory(1.0–3.0 TB/s)). To avoid spilling to slow local memory, we cap per-thread registers at r_t under the SM budget R_{SM}.

For the remaining effective reduction depth h_{eff} after load stretching, we stage only the frontier in shared memory. If the original tree has height H and hash-digest size d bytes, the maximum frontier width and the corresponding per-block shared-memory footprint are $W = 2^{H-h_{\text{eff}}}, \quad s_b \approx W \cdot d = 2^{H-h_{\text{eff}}}d$, which reduces the shared-memory usage by a factor of $2^{h_{\text{eff}}}$ compared to staging the bottom level. This register-first, frontier-only staging cuts global traffic, limits shared-memory pressure (preserving occupancy), thereby improving throughput while keeping r_t and s_b within the per-SM budgets $(R_{\mathrm{SM}}, S_{\mathrm{SM}})$.

1) Memory Reuse Pattern: In the stretching operation, we let each thread handle g nodes, so that it locally computes a height of $\log_2 g$ micro-subtree fully in registers. The g nodes are placed in a per-thread register array (logical view) `A[0..g-1]`. At level 0 the thread hashes pairs (`A[0]`,`A[1]`),(`A[2]`,`A[3]`),(`A[4]`,`A[5]`),...,(`A[g - 2]`, `A[g - 1]`) and *overwrites the even slots* `A[0]`, `A[2]`, `A[4]`, ..., `A[g - 2]` with the parents. At level 1 it hashes (`A[0]`,`A[2]`),(`A[4]`,`A[6]`),... (`A[g - 4]`,`A[g - 2]`)

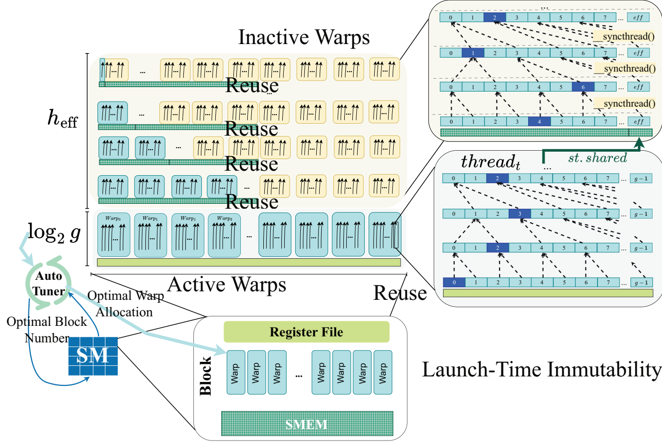

Fig. 2. Load stretching and memory optimizations within a single Merkle tree. Inactive warps during tree reduction are highlighted in light yellow. The parameter g denotes a tuning factor, and blue squares represent signatures. SM denotes a Streaming Multiprocessor. The bottom-left inset illustrates the auto-tuning workflow.

and *writes into the odd slots* `A[1]`, `A[3]`, `A[5]`, ..., `A[g − 3]`. Until at level ℓ, the thread hashes the pair $A[(\ell+1)\%2]$ and $A[(\ell+1)\%2+2]$, and stores the parent into position $A[(\ell+1)\%2+1]$. This odd–even interleaving reuses the same g logical locations without auxiliary buffers and avoids RAW hazards: reads come from the parity written at level $\ell-1$, while writes target the opposite parity at level ℓ. After $\log_2 g$ in-register levels, the thread emits its micro-root to shared memory; we then switch to a one-pair per thread reduction with the same odd–even interleaving, inserting one synchronization point per level. Fig. 2 illustrates warp-centric load stretching and the two memory-access patterns within a single Merkle tree.

2) Should We Use `__shfl_xor_sync`?: Given the higher on-chip bandwidth of registers, we explored replacing the shared memory staging at the remaining depth h_{eff} with register-resident, warp-level exchange via `__shfl_xor_sync`. Therefore, we implemented a Brent–Kung reduction within each warp: at stage ℓ, lanes fetch their partner's digest with XOR-shuffle, compute the parent, and rely on lane masks to avoid block-wide barriers [14]. However, this path underperforms shared-memory staging on SPHINCS$^+$ f-parameter sets. The bottleneck is payload width: security levels use 16/24/32-byte digests, while `__shfl_xor_sync` transfers 32-bit lanes; each pairwise combine therefore requires 4/6/8 shuffles (and accompanying moves) per node, inflating instruction count and register pressure. Over many levels, the accumulated cross-lane traffic outweighs the cost of vectorized 16–32 B loads/stores in shared memory. We thus retain shared memory for inter-thread exchange at depth h_{eff}.

E. Auto Tuning for Multiple Merkle Trees

Building on the single-tree tuning, we extend to *multi-tree* execution within a block. Recall that load stretching assigns $g{=}2^m$ nodes per thread, producing an effective in-register reduction depth $m = \log_2 g$ and a remaining shared-memory depth $h_{\text{eff}} = H - \log_2 g$. The per-tree shared-memory staging for the frontier is $s_{\text{tree}} \approx c_s \cdot 2^{h_{\text{eff}}} d$, where d is the digest size (bytes) and $c_s \geq 1$ accounts for padding/domain-separation metadata. If a block processes b trees concurrently, the dynamic shared memory per block is $s_b = b \cdot s_{\text{tree}} \leq S_{\text{blk}}^{\max}$, yielding the shared-memory bound $b \leq \lfloor S_{\text{blk}}^{\max}/(c_s 2^{h_{eff}} d) \rfloor$. With $W_{\text{blk}} = \texttt{blockDim.x}/32$ warps per block and w_t warps allocated per tree, we also require $b \cdot w_t \leq W_{\text{blk}}$ to maintain occupancy under SM budgets.

A practical complication is that FORS uses *non power-of-two* tree counts (e.g., $k \in \{33, 35\}$(see Table III). Mapping b trees per block induces a tail tile of size $r{=}k \bmod b$; its under-utilization is $\varepsilon_{\text{tail}}(b) = \frac{b-r}{b} \not\Vdash[r \neq 0]$, so we prefer b that minimizes $\varepsilon_{\text{tail}}(b)$ while satisfying the resource bounds above. Too large a b can exceed per-block resources and stall scheduling; too small a b under-exposes parallelism. We therefore adopt a *profiling-guided* autotuner that searches over a constrained space of *(group size g, trees-per-block b, block size, unroll factor u, warps-per-tree w_t)* and selects the configuration that maximizes steady-state throughput under optional penalties for spills, bank conflicts, and low occupancy: $\arg\max_{\theta\in\Theta}$ $\text{Thr}(\theta) - \lambda_1 \text{Spill}(\theta) - \lambda_2 \text{BankConf}(\theta) - \lambda_3\big(1-\text{Occ}(\theta)\big)$.
In practice, we restrict Θ via *block-constrained pruning*: b is capped by $S_{\text{blk}}^{\max}/(c_s 2^{h_{eff}} d)$; (g, u) pairs that exceed a register budget $r_t^{\max}$ are discarded; and we early-stop candidates whose warmup throughput falls below a running percentile. We drive

TABLE III
SPHINCS$^+$-F PARAMETER SETS. n IS THE HASH OUTPUT SIZE IN BYTES. h IS THE HYPERTREE HEIGHT, AND d IS THE NUMBER OF HYPERTREE LAYERS. $\log(t)$ IS THE HEIGHT OF EACH FORS TREE, AND k IS THE NUMBER OF FORS TREES. w DENOTES THE WINTERNITZ PARAMETER IN WOTS$^+$.

Scheme	n	h	d	$\log(t)$	k	w
SPHINCS$^+$-128f	16	66	22	6	33	16
SPHINCS$^+$-192f	24	66	22	8	33	16
SPHINCS$^+$-256f	32	68	17	9	35	16

Algorithm 1 End-to-end autotuning for multi-tree FORS

Require: Candidate sets $\mathcal{G}$ (group sizes), $\mathcal{B}$ (trees-per-block), $\mathcal{D}$ (blockDim), $\mathcal{U}$ (unroll), $\mathcal{W}$ (warps-per-tree); SM budgets $(S_{\text{SM}}, R_{\text{SM}}, T_{\text{SM}}, B_{\text{SM}})$; digest size d; padding factor c_s; total trees k.
1: Best $\leftarrow \varnothing$
2: **for** $g \in \mathcal{G}$ **do** ▷ $h_{\text{eff}} = H - \log_2 g$, $s_{\text{tree}} = c_s 2^{h_{eff}} d$
3: **for** $b \in \mathcal{B}$ **such that** $b \cdot s_{\text{tree}} \leq S_{\text{blk}}^{\max}$ **do**
4: **for** `blockDim` $\in \mathcal{D}$ **do**
5: $W_{\text{blk}} \leftarrow$ `blockDim`/32
6: **for** $w_t \in \mathcal{W}$ **such that** $b \cdot w_t \leq W_{\text{blk}}$ **do**
7: **for** $u \in \mathcal{U}$ **do**
8: *// instantiate templated kernel* `K_FORS<g,u,w_t>` *with dynamic smem* $s_b = b\, s_{tree}$
9: WARMUP(W); DROPCACHES(); ▷ host-side page cache flush; device pre-touch
10: $t \leftarrow$ median runtime of R runs; collect counters: spills, bank-conflicts, occ
11: Thr $\leftarrow \frac{\texttt{blockDim}}{t}$; score $\leftarrow$ Thr $- \lambda_1$Spill $- \lambda_2$BankConf $- \lambda_3(1 - \text{Occ}) - \lambda_4 \varepsilon_{\text{tail}}(b)$
12: **if** Best is $\varnothing$ **or** score $>$ Best.score **then**
13: Best $\leftarrow (g, b, \texttt{blockDim}, w_t, u, \text{score})$
14: **return** Best

the outer loop with a shell script that sets `blockDim`, dynamic shared memory, and environment pins; before each measurement we clear OS page caches to reduce cross-run bias shown in Algorithm 1. This profiling-guided search yields balanced tiles for irregular k value and selects (g, b) that maximize concurrency without over-subscribing SM resources, thereby minimizing tail inefficiency and warp idling.

F. Unrolling-Constrained Pruning in the Autotuner

Algorithm 2 Peak-validated geometric scan for `gridDim` (fixed u)

Require: ratio $\beta > 1$ (e.g., 2), start B_1, upper bound $B_{\max}$, tolerance ε, lookahead K.
1: $B \leftarrow B_1$; measure $\widehat{\mathrm{Thr}}_u(B)$; best $\leftarrow (B, \widehat{\mathrm{Thr}}_u(B))$; $p \leftarrow 0$
2: **while** $\beta B \le B_{\max}$ **do**
3: $\quad B' \leftarrow \lfloor \beta B \rfloor$; measure $\widehat{\mathrm{Thr}}_u(B')$
4: $\quad$ **if** $\widehat{\mathrm{Thr}}_u(B') >$ best.thr $+ \varepsilon$ **then**
5: $\quad\quad$ best $\leftarrow (B', \widehat{\mathrm{Thr}}_u(B'))$; $p \leftarrow 0$ ▷ new higher peak
6: $\quad$ **else if** isLocalPeak(B') **and** $\widehat{\mathrm{Thr}}_u(B') <$ best.thr $- \varepsilon$ **then**
7: $\quad\quad p \leftarrow p + 1$;
8: $\quad\quad$ **if** $p \ge K$ **then return** $B^\star(u) \leftarrow$ best.B
9: $\quad B \leftarrow B'$
10: **return** $B^\star(u) \leftarrow$ best.B

We decouple the 2D search over *unroll* and *block count* by first *fixing the unroll factor* u and then profiling the number of blocks to launch (`gridDim`). Let $t_{\text{blk}}(u)$ be the measured per-block service time under unroll u. For a sufficiently large batch, the sustained throughput (blocks/s) when launching B blocks is well approximated by

$$\mathrm{Thr}_u(B) \;=\; \frac{1}{t_{\text{blk}}(u)}\;\min\{\,B,\; S \cdot n_b(u)\,\}, \tag{1}$$

where S is the number of SMs and $n_b(u)$ is the number of resident blocks per SM bounded by the resource model

$$n_b(u) \;\le\; \min\!\left(\left\lfloor \frac{S_{\text{SM}}}{s_b(u)} \right\rfloor,\; \left\lfloor \frac{R_{\text{SM}}}{r_t(u)\, t_b} \right\rfloor,\; \left\lfloor \frac{T_{\text{SM}}}{t_b} \right\rfloor,\; B_{\text{SM}} \right), \tag{2}$$

with $s_b(u)$ the per-block shared memory, $r_t(u)$ the registers per thread, and t_b the threads per block.

a) Monotonicity and saturation (fixed u).: From (1), $\mathrm{Thr}_u(B)$ is non-decreasing in B and saturates on a plateau once $B \ge S\, n_b(u)$. Define the model-guided upper bound

$$B_{\max}(u) \;=\; \min\{\, M,\; \alpha\, S\, n_b(u)\,\}, \tag{3}$$

where M is the total number of messages and $\alpha \ge 1$ is a small slack to cover tail tiles.

Lemma 1 (Peak-validated scan: ε-optimality)**.** *Assume the measured throughput* $\widehat{\mathrm{Thr}}_u(B) = \mathrm{Thr}_u(B) + \xi(B)$ *has bounded noise* $|\xi(B)| \le \varepsilon_{\text{noise}}$*. Scanning B in an increasing candidate sequence and committing to the* first dominating peak*, the earliest B whose record is not exceeded by the next K peaks by more than a tolerance $\varepsilon \ge 2\varepsilon_{\text{noise}}$, which returns $B^\dagger$ such that* $\mathrm{Thr}_u(B^\dagger) \ge \mathrm{Thr}_u(B^\star) - \varepsilon$*, where* $B^\star = S\, n_b(u)$.

Proof sketch. $\mathrm{Thr}_u(B)$ grows linearly until $B^\star$ and is flat thereafter. With bounded noise and $\varepsilon \ge 2\varepsilon_{\text{noise}}$, any apparent improvement beyond $B^\star$ larger than ε is unlikely; thus the first dominating peak lies within ε of the plateau. □

TABLE IV
THROUGHPUT (KOPS) FOR DIFFERENT (g, WARP) SETTINGS ACROSS SPHINCS$^+$-f PARAMETER SETS. BEST PER SCHEME IN **BOLD** ($\star$).

	Configurations $(g, \textbf{warp})$ and Throughput (KOPS)					
128f	(1, 2) 438.71	**(2,1)** **563.15***	(4, 0.5) 518.21	— —	— —	— —
192f	(1, 8) 124.92	(2, 4) 121.64	(4, 2) 138.25	**(8,1)** **187.28***	(16, 0.5) 121.06	— —
256f	(1, 16) 53.31	(2, 8) 68.32	(4, 4) 79.36	(8, 2) 88.86	**(16, 1)** **102.23***	(32, 0.5) 65.46

b) Peak-validated profiling procedure (fixed u): We use (2)–(3) to bracket the search and then apply a left-to-right, peak-validated scan that is robust to jitter.

c) Discussion of Algorithm 2: Fixing u collapses the 2D search to a 1D profiling over B with $O(\log B_{\max}) + O(K)$ measurements. The resource model guides the bracket and prevents oversubscription; the peak-validated rule (Lemma 1) yields an ε-optimal block count in the presence of noise. In practice we use steady-state medians, clear OS/device caches between runs, and set ε to approximately twice the measured noise floor.

IV. EXPERIMENTAL EVALUATION

A. Environment Setup

We first evaluate on an RTX 4090 (Ada). Section IV-B0d extends the study to commodity GPUs: GTX 1070 (Pascal) and RTX 2080 Ti (Turing) and to a datacenter H100 (Hopper). Warp occupancy and compute/memory throughput (CT/MT) are collected with NVIDIA Nsight Compute. We report throughput in kilo-operations per second (KOPS) and adopt SHA-256 (SHA-2) as the underlying hash primitive.

B. Performance Evaluation

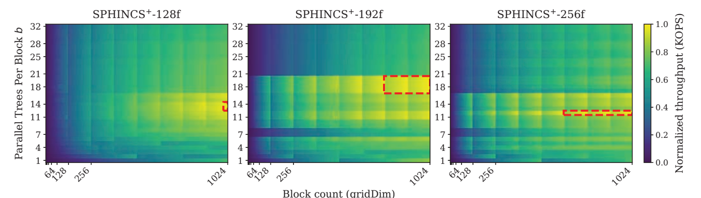

Fig. 3. Heatmaps of normalized throughput (KOPS) over b *parallel trees per block* (y-axis) and *block count / gridDim* (x-axis) for SPHINCS$^+$-128f, -192f, and -256f. Colors are min–max normalized within each panel (higher is better). The red dashed rectangle highlights the near-optimal region, defined as all settings within 2% of the panel's peak throughput.

a) Autotuner Behavior: We first tune the per-tree group size g. Table IV shows that assigning one Merkle subtree to a *single warp* yields the best efficiency; sub-warp mappings (e.g., 16 lanes) require two sub-warps to complete a single subtree, incurring duplicated control and barrier overheads. Fig. 3 visualizes the autotuner landscape: performance peaks appear near mid-range parallel tree counts per block b, consistent with minimizing the tail-inefficiency term $\varepsilon_{\text{tail}}(b)$ introduced earlier. Table V quantifies tuning cost: exhaustive search takes up to 1416 min (≈23.6 h), whereas our pruned search converges in 1.34–2.20 min, reducing wall-time by as much as $10^3\times$ (e.g.,

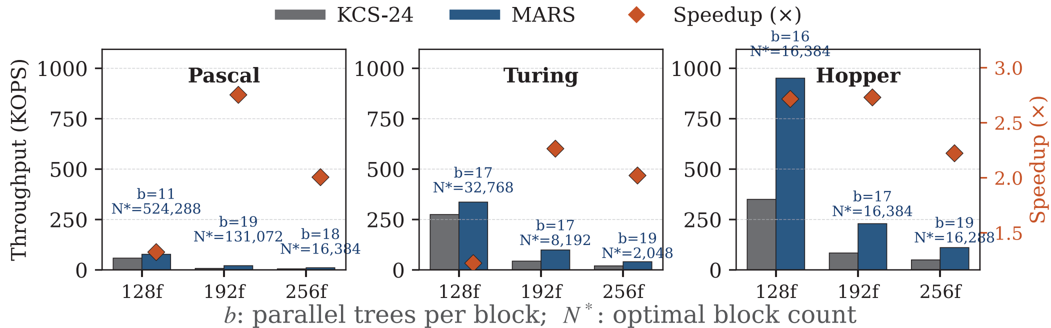

Fig. 4. Cross-architecture throughput (KOPS) for 128f, 192f, and 256f on Pascal, Turing, and Hopper. Bars compare KCS-24 with MARS; annotations above MARS denote *(parallel trees per block, optimal block number)*. Diamond indicates speedup.

TABLE V
AUTOTUNER RUNTIME WITH AND WITHOUT PRUNING, REDUCTION FACTORS, AND THE SELECTED OPTIMAL CONFIGURATIONS. FOLLOWED BY THROUGHPUT (KOPS) AT THOSE OPTIMA FOR KCS-24 AND MARS, INCLUDING THE SPEEDUPS.

Metric	128f	192f	256f
Block Range	(1..1024)		
Auto Tuner (Native) [min]	1248	1354	1416
Auto Tuner (Pruning) [min]	1.49	2.20	1.34
Reduction (Native / Pruning)	838	697	1056
Opt (g, #Trees, Block Size)	(2, 14, 1024)	(8, 18, 1024)	(16, 12, 1024)
Thr (KCS-24)	449.27	124.39	68.26
Thr (MARS)	888.79	210.05	128.72
Speedup	1.98×	1.69×	1.89×

TABLE VI
THROUGHPUT (KOPS) AT OPTIMAL BLOCK SIZE: KCS-24 VS. MARS WITH SPEEDUPS.

Scheme	Optimal Block Size	KCS-24	MARS	Speedup
128f	65552	591.81	1494.02	2.52×
192f	65536	137.25	328.01	2.39×
256f	16384	72.08	167.62	2.33×

1056× on 256f). These results demonstrate the effectiveness of the proposed pruning strategy.

b) Main Results of MARS: We evaluate MARS against KCS-24 as the baseline on RTX4090 [13] . With a fixed block count (`gridDim` = 1024), MARS achieves speedups of 1.98×, 1.69×, and 1.89× for SPHINCS$^+$-128f, -192f, and -256f, respectively (Table V). To probe the single-GPU block capacity, we expand the autotuner's search range up to 10^9 blocks. Thanks to our pruning strategy, the number of measurements scales *logarithmically* with the range, allowing us to identify optimal block counts of 65,552, 65,536, and 16,384 for 128f/192f/256f without triggering OOM (Out of Memory) or incurring stalls due to over-provisioned launches. At these settings, MARS delivers larger gains of 2.52×, 2.39×, and 2.33× (Table VI). These improvements stem from our warp-centric load stretching, which raises effective warp utilization and profiling-guided, resource-aware tuning that balances shared-memory usage and register pressure across blocks.

c) Energy efficiency: We implement an energy-computation script that numerically integrates instantaneous power reported by `nvidia-smi` and report batch energy E_{batch} as the mean over ten kernel runs at the optimal block count. Compared to KCS-24, MARS reduces E_{batch} from 236.98→149.66 J, 675.11→572.40 J, and 441.77→415.26 J for SPHINCS$^+$-128f, -192f, and -256f, corresponding to ∼58%, 18%, and 6% savings. Together with higher throughput, MARS delivers superior performance per joule across f parameter sets.

d) Cross-Architecture Portability: Across Pascal, Turing, and Hopper, MARS consistently surpasses KCS-24, yielding speedups of 1.33–2.75× (Pascal), 1.22–2.26× (Turing), and 2.22–2.73× (Hopper) (Fig.4). The autotuner generalizes across devices by discovering distinct, resource-aware *(parallel trees per block, optimal block count)*. On the higher-clocked RTX 4090 (2.235 GHz), we observe higher throughput than on H100 (1.035 GHz), suggesting that FORS is an instruction-bound scheme. Higher clock frequencies and increased instruction-issue throughput. We observe higher block residency on Pascal GPUs; by contrast, high-end GPUs use more conservative admission thresholds. The autotuner further yields heterogeneous optima across devices, demonstrating hardware-aware effectiveness and portability.

C. Related Work

Prior GPU optimizations for Merkle trees largely fall into two lines: (i) exploiting intra-tree parallelism, and (ii) analytically stacking multiple trees to increase concurrency [13] [15] [16] [17]. These approaches often either under-utilize hardware resources or fail to attain configuration optimality under realistic resource constraints. In contrast, MARS couples a principled resource model with profiling-guided autotuning, providing (to our knowledge) the first GPU framework that jointly optimizes single- and multi-tree execution based on both theory and on-device measurements.

V. CONCLUSION

We presented MARS, a general GPU-centric optimization framework, and used FORS as a case study to evaluate its performance. MARS achieves 2.33×–2.52× speedups over state-of-the-art GPU baselines and migrates across Pascal/Turing/Ada/Hopper without algorithmic changes. Beyond FORS, MARS's insights generalize to Merkle-tree–based cryptographic workloads (e.g., LMS/XMSS, zero-knowledge (ZK) primitives, and blockchain verifiers), offering a practical path toward high-performance deployments.

ACKNOWLEDGMENT

This work is supported by the United States National Science Foundation under Grant No. 2530705.

REFERENCES

[1] R. C. Merkle, "Secrecy, authentication, and public key systems." Ph.D. dissertation, Stanford, CA, USA, 1979, aAI8001972.

[2] D. McGrew, M. Curcio, and S. Fluhrer, "Leighton-micali hash-based signatures," Tech. Rep., 2019.

[3] A. Hülsing, "W-ots+–shorter signatures for hash-based signature schemes," in *International Conference on Cryptology in Africa*. Springer, 2013, pp. 173–188.

[4] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The sphincs+ signature framework," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2129–2146. [Online]. Available: https://doi.org/10.1145/3319535.3363229

[5] P. Franco, *Understanding Bitcoin: Cryptography, engineering and economics*. John Wiley & Sons, 2014.

[6] S. Tikhomirov, "Ethereum: state of knowledge and research perspectives," in *International symposium on foundations and practice of security*. Springer, 2017, pp. 206–221.

[7] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on Computing*, vol. 18, no. 1, pp. 186–208, 1989. [Online]. Available: https://doi.org/10.1137/0218012

[8] R. M. Shadab, Y. Zou, S. Gandham, A. Awad, and M. Lin, "Hmt: A hardware-centric hybrid bonsai merkle tree algorithmnbsp;for high-performance authentication," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 4, Jul. 2023. [Online]. Available: https://doi.org/10.1145/3595179

[9] D. Amiet, A. Curiger, and P. Zbinden, "FPGA-based accelerator for post-quantum signature scheme SPHINCS-256," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 1, pp. 18–39, 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/831

[10] Q. Berthet, A. Upegui, L. Gantel, A. Duc, and G. Traverso, "An area-efficient sphincs+ post-quantum signature coprocessor," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 180–187.

[11] R. Román, R. Arjona, J. Arcenegui, and I. Baturone, "Hardware security for extended merkle signature scheme using sram-based pufs and trngs," in *2020 32nd International Conference on Microelectronics (ICM)*, 2020, pp. 1–4.

[12] C. Wang and M. Gao, "Unizk: Accelerating zero-knowledge proof with unified hardware and flexible kernel mapping," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1101–1117. [Online]. Available: https://doi.org/10.1145/3669940.3707228

[13] D. Kim, H. Choi, and S. C. Seo, "Parallel implementation of sphincs+ with gpus," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. PP, no. 99, pp. 1–14, 2024. [Online]. Available: https://www.researchgate.net/publication/378790147_Parallel_Implementation_of_SPHINCS_With_GPUs

[14] R. P. Brent and H. T. Kung, "The chip complexity of binary arithmetic," in *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, ser. STOC '80. New York, NY, USA: Association for Computing Machinery, 1980, p. 190–200. [Online]. Available: https://doi.org/10.1145/800141.804666

[15] Z. Wang, X. Dong, H. Chen, Y. Kang, and Q. Wang, "Cuspx: Efficient gpu implementations of post-quantum signature sphincs+," *IEEE Transactions on Computers*, vol. 74, no. 1, pp. 15–28, 2025.

[16] J. Wu, Y. Yu, Z. Chen, H. Yang, C. Li, and Z. Liu, "Cbpspx: A cuda-based batch parallel optimization of post-quantum signature sphincs+," *IEEE Internet of Things Journal*, 2025.

[17] T. Lu, Y. Chen, Z. Wang, X. Wang, W. Chen, and J. Zhang, "Batchzk: A fully pipelined gpu-accelerated system for batch generation of zero-knowledge proofs," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 100–115. [Online]. Available: https://doi.org/10.1145/3669940.3707270