GPU-Accelerated GGM Tree Construction with Keccak-f1600 and Spongent-128

Anjana Manoj, Mark Sui, Nikhita Neelakanta
Department of Electrical and Computer Engineering
University of California San Diego
La Jolla, California, United States
{anmanoj, tisui, nneelakanta}@ucsd.edu

***Abstract*—This project implements full GGM tree construction on both CPU and CUDA GPU using Keccak-f1600 and Spongent-128 as PRF backends. The implementation uses a flat breadth-first memory layout and expands the tree level by level, with one GPU thread handling one parent node. We compare runtime, throughput, and GPU speedup on a GTX 1080 Ti for depths 8, 12, 16, and 20. The results show that GPU acceleration becomes more effective as the tree deepens, while absolute performance depends strongly on the PRF design. At depth 20, Keccak achieves a $1030\times$ speedup and Spongent achieves a $1675\times$ speedup over the CPU baseline. The primary optimisation for both primitives was replacing runtime-indexed temporary arrays with compile-time-constant indices, moving working state from off-chip DRAM into GPU registers.**

***Index Terms*—GGM tree, pseudorandom function, CUDA, GPU acceleration, Keccak, Spongent**

Source Code:
https://github.com/marksui/ECE268-GGM-Tree-Construction

I. Introduction & Prior Work

The Goldreich-Goldwasser-Micali (GGM) tree is a foundational cryptographic construction [1]. It enables the generation of a large number of pseudorandom values from a single secret seed by recursively applying a length-doubling pseudorandom generator (PRG). The resulting outputs are computationally indistinguishable from true randomness. GGM trees appear throughout protocol design: they are the basis for oblivious transfer (OT) extension [2], private set intersection, secure multiparty computation (MPC), and puncturable pseudorandom functions (PPRFs). In all of these settings, a server-side implementation may need to evaluate millions of leaves per session, making throughput the primary constraint.

Despite their security guarantees, GGM trees present a significant computational bottleneck. Constructing a tree of depth d requires $2^d - 1$ distinct PRF evaluations. Modern GPUs are a natural fit for this workload: every node expansion at a given depth is independent, so all nodes at that level can be computed in parallel. The challenge is that GPU throughput depends heavily on the primitive. A primitive designed for software maps well to GPU hardware; one designed for silicon area minimisation carries structural costs that no kernel optimisation can remove.

Our project focuses on four contributions:

- GPU implementation of the full GGM tree using both Keccak-f1600 and Spongent-128, enabling direct comparison of a software-optimised versus a hardware-native algorithm in a parallel environment.
- A flat BFS memory layout that avoids scattered access patterns and ensures coalesced global memory reads.
- Crossover analysis identifying the tree depths at which GPU execution overcomes kernel launch overhead and outperforms the CPU for each primitive.
- Four GPU-specific optimisations for Spongent that we implemented reduce depth-12 GPU time from 2041 ms to 108 ms

Prior works demonstrate efficient batch-mode Keccak optimization on graphics hardware [3], [4], whereas hardware-oriented primitives like Spongent trade software speed for extreme silicon-area efficiency and physical side-channel resilience [6]. This divergence introduces a fundamental performance-versus-security tradeoff when deploying parallelized tree expansions within modern multi-party computation protocol pipelines [2].

II. Background

A. GGM Tree Construction

The GGM tree [1] builds a secure PRF from a length-doubling PRG $G\colon \{0,1\}^n \to \{0,1\}^{2n}$. A depth-$d$ tree with root seed s produces 2^d leaves. Node (l, i) holds seed $s_{l,i}$; its children are $G_0(s_{l,i})$ and $G_1(s_{l,i})$. The PRF value on input $x = x_1 x_2 \cdots x_d$ is the leaf at the path defined by x. Security reduces to the pseudorandomness of G: any PRF distinguisher yields a PRG distinguisher. We realise G_b as $\mathrm{Hash}(b \,\|\, \text{seed})$ for domain byte $b \in \{0\text{x}00, 0\text{x}01\}$, ensuring left and right children are always distinct.

B. Keccak-f1600

Keccak-f1600 is the cryptographic permutation underlying SHA-3, defined in NIST FIPS 202 [5]. It is designed for high-throughput software implementations on ALU-based hardware.

Its three main components are:

- **Sponge construction:** Input data is XORed into the rate portion of the state; the capacity portion is kept hidden, providing security against extension attacks.
- **State structure:** The permutation operates on a 1600-bit state, organised as a 5×5 array of 64-bit lanes.

- **Round function:** 24 rounds each applying five steps: θ (column parity diffusion), ρ/π (bitwise rotation and lane permutation), χ (nonlinear row mixing), and ι (round constant XOR). All steps reduce to 64-bit XOR and rotation which are native GPU integer ALU operations.

C. Spongent-128

Spongent-128 is a lightweight sponge hash designed for RFID tags and IoT microcontrollers [6]. Its state is 136 bits (17 bytes) with a 1-byte rate, 128-bit capacity, and 70 rounds per permutation call. It prioritises low silicon area over software throughput.

It consists of three main components:

- **PRESENT S-box:** Spongent uses the 4-bit S-box from the PRESENT block cipher [7]. This gives cryptographic nonlinearity with only four logic gates per nibble in hardware.
- **pLayer (permutation layer):** The pLayer is a fixed bit permutation defined by

$$p(i) = (68i) \pmod{135}, \quad 0 \le i < 135, \qquad (1)$$

  with $p(135) = 135$. On an ASIC this is pure wire routing; bits arrive at their destination with zero logic and zero clock cycles. In software, each of the 136 bits must be individually read, its destination computed, and written to the correct position. This 136-operation sequence repeats every round across 70 rounds per permutation call, and is the dominant software cost addressed by the GPU optimisations in Section III-D.
- **Design intent:** Spongent was designed to minimise gate count on silicon. The operations that are free on hardware (wire routing, fixed permutation) are among the most expensive in software. This structural mismatch is the central challenge of the GPU implementation.

D. CUDA Programming Model

We use NVIDIA's CUDA programming model to exploit Single Instruction, Multiple Thread (SIMT) parallelism. Execution is organised into grids of thread blocks; 32 threads execute in lockstep as a warp which is the fundamental scheduling unit.

The memory hierarchy, from fastest to slowest, is: registers ($\approx$1 cycle), shared memory ($\approx$5 cycles), constant memory (cached, broadcast to a warp when all threads access the same address, $\approx$5 cycles), and local/global memory (DRAM, $\approx$400 cycles). A critical rule: if an array is indexed by a runtime value, the NVCC compiler cannot keep it in registers and must allocate it in local memory. Kernels launched in the same CUDA stream execute in order without an explicit synchronisation call between them.

III. Implementation

A. Memory Layout: Flat BFS Array

Pointer-based tree structures on a GPU produce scattered memory access, destroying coalescing and causing L2 cache misses. We instead map the entire GGM tree to a flat one-dimensional array in CUDA global memory. Node (ℓ, i) is stored at flat index

$$\text{flat\_index}(\ell, i) = 2^\ell - 1 + i. \qquad (2)$$

All nodes at the same depth are contiguous. When a kernel processes level ℓ, thread i reads from flat index $2^\ell - 1 + i$ and writes children to $2^{\ell+1} - 1 + 2i$ and $2^{\ell+1} - 1 + 2i + 1$. Adjacent threads therefore access adjacent addresses; the GPU memory controller coalesces these into a single DRAM transaction. Total allocation is $(2^{d+1} - 1) \times$ seed_bytes: 32 MB at depth 20 for Spongent (16-byte seeds) and 64 MB for Keccak (32-byte seeds).

B. Parallelisation Strategy

The naive path-traversal approach assigns one thread per leaf, each walking from root to leaf. This causes severe redundancy: at depth 12, any two sibling leaves share 11 of 12 ancestors, so path traversal performs $12\times$ more hash evaluations than necessary and creates contention on shared ancestors.

We replace this with level-by-level BFS kernel execution. At depth level ℓ, the host launches a CUDA kernel with $N = 2^\ell$ threads, one per parent node. Each thread reads its parent seed, performs the PRF expansion, and writes the two child nodes to global memory. All level kernels are submitted back-to-back in the same CUDA stream; CUDA stream ordering guarantees that kernel ℓ+1 cannot begin until kernel ℓ has finished writing its outputs. A single `cudaDeviceSynchronize()` is issued after the complete level loop. The original code called it after every level, 11 unnecessary CPU-GPU round-trips at depth 12 which was especially wasteful for early levels whose GPU work takes microseconds.

C. Keccak GPU Kernel

One thread computes one node expansion. Each thread owns one Keccak state:

$$A_t = (a_0, a_1, \ldots, a_{24}), \quad a_i \in \{0,1\}^{64}. \qquad (3)$$

The θ, ρ/π, χ, and ι steps depend entirely on 64-bit XOR and rotations, which translate directly to CUDA ALU instructions.

The key optimisation was the ρ+π step:

$$B_{y+5((2x+3y) \bmod 5)} = \text{rotl}(A_{x+5y},\ r_{x,y}). \qquad (4)$$

The original loop used a runtime-valued table index into the temporary array, forcing it into local memory (DRAM). Replacing the loop with 25 literal assignments makes every index a compile-time constant; NVCC promotes the 200-byte state and temporary arrays entirely to GPU registers. Round constants are stored in `__constant__` memory for broadcast across a warp. The Keccak kernel operates without any local-memory DRAM stalls. For depth 12, this changed GPU runtime from approximately 1.87 ms to 0.49 ms:

$$S_{\text{Keccak}} = \frac{1.87}{0.49} \approx 3.8\times. \qquad (5)$$

D. Spongent GPU Kernel Optimisations

The number of permutation calls per node expansion does not change with any of the optimisations below. Each child requires one domain absorb, sixteen seed absorbs, one padding permute, and fifteen squeeze permutes:

$$N_{\text{perm}} = 33 + 33 = 66 \text{ calls per node expansion.} \quad (6)$$

The speedup comes entirely from making each call cheaper on the GPU.

Problem 1: S-box warp divergence. The original `sBoxLayer` used a 16-case switch on each nibble. When 32 warp threads hold different nibble values, the hardware serialises all 16 cases. Fix: a 256-entry byte-level lookup table `c_sbox2` in `__constant__` memory:

$$L(v) = 16\,S\Big(\Big\lfloor \frac{v}{16} \Big\rfloor\Big) + S(v \bmod 16), \quad 0 \le v < 256, \quad (7)$$

where S is the 4-bit PRESENT S-box. Every thread in a warp accesses the same table addresses, so the hardware broadcasts the value with no divergence.

Problem 2: pLayer DRAM spilling. The pLayer loop computed the destination of each bit as $(68i) \bmod 135$ at runtime and then wrote to `tmp[dst]`. A runtime index into an array forces NVCC to place `tmp` in local memory (≈400-cycle DRAM). With

$$66 \times 70 \times 136 = 628{,}320 \quad (8)$$

bit-placement operations per node expansion, virtually every thread cycle was a DRAM stall. Fix: `#pragma unroll` on both the outer byte loop ($b = 0 \ldots 16$) and the inner bit loop ($k = 0 \ldots 7$). After unrolling, $i = 8b + k$ is a compile-time constant, so $p(i)$ from equation(1) is also compile-time, so `tmp[p(i) >> 3]` becomes a literal register access. NVCC keeps the 17-byte temporary state

$$T = (t_0, t_1, \ldots, t_{16}) \quad (9)$$

entirely in registers with zero DRAM traffic.

Problem 3: Shared-prefix register spilling. We attempted an optimisation in which both children share the 16 seed-absorb permutes, forking only at the domain byte (reducing from 66 to 50 permutes per expansion). This required two 17-byte state arrays to be live simultaneously, increasing peak register demand. NVCC responded by spilling one state array to DRAM, causing a $5.7\times$ regression. We reverted to computing the two hashes sequentially so that all registers from the first hash are freed before the second begins.

Problem 4: Instruction cache overflow. Applying `#pragma unroll` to the outer 70-round loop would generate 70 copies of the round body, producing thousands of instructions that exceed the L1 instruction cache. We add `#pragma unroll 1` to keep a single loop body in the instruction cache and prevent fetch stalls.

Before these fixes, Spongent GPU at depth 12 took approximately 2041 ms ($4.1\times$ over CPU). After all four fixes:

$$S_{\text{opt}} = \frac{2041}{108.581} \approx 18.8\times \quad \text{(GPU-side improvement only).} \quad (10)$$

The final CPU-to-GPU speedup at depth 12 is

$$S_{\text{CPU/GPU}} = \frac{11859.152}{108.581} \approx 109\times. \quad (11)$$

The permute count per expansion remains 66. The entire gain comes from replacing DRAM-bound operations with register operations.

E. PRF Construction and CPU/GPU Consistency

The PRF expansion uses domain separation to produce two distinct children:

$$\text{left} = H(0\text{x}00 \,\|\, \text{seed}), \quad (12)$$

$$\text{right} = H(0\text{x}01 \,\|\, \text{seed}). \quad (13)$$

The sponge absorbs the domain byte first, then the 16 seed bytes, applies 10* padding, and squeezes 16 output bytes, 33 permute calls per child. The CPU reference implementation uses the same domain-first construction; mismatched input orderings between CPU and GPU would both produce valid trees but fail byte-for-byte comparison.

Correctness was validated at multiple levels. For Spongent, the core permutation was verified against the official CHES 2011 test vector, pLayer bijectivity was confirmed by checking that each input bit maps to exactly one output bit, and the LFSR sequence was validated against the reference. For Keccak, the SHA3-256 digest of the empty string was checked against NIST FIPS 202, and GPU-versus-CPU equivalence was tested across 10,000 randomly generated states. Three fixed seeds (all-zeros, all-ones, and incrementing bytes) verified byte-for-byte agreement at depths 0, 1, 4, and 8, covering all 511 nodes of a depth-8 tree per seed. A separate dummy-PRF framework test confirmed correct tree structure independent of the hash function. Across 90 named assertions in four test binaries, all pass.

IV. Evaluation

A. Experimental Setup

All experiments ran on an NVIDIA GeForce GTX 1080 Ti (3,584 CUDA cores, 11 GB GDDR5X, Pascal architecture, `sm_61`). GPU kernels were compiled with:

```
nvcc -std=c++11 -O2 -rdc=true
  --generate-code arch=compute_61,code=sm_61
```

CPU (Intel i7-12700K) timings use `std::chrono::high_resolution_clock` on the same host.Reported GPU runtime includes host-to-device (H2D) transfer and kernel execution, but excludes device-to-host (D2H) transfer unless otherwise stated.Results in Table I report the best of one run. Figure plots reporting mean runtime use three repeated runs, except the dense depth sweep (depths 4, 6, 10, 14, 18) and the block-size sensitivity test, which each use five repeated runs with standard deviation reported. Seeds are 16 bytes for Spongent and 32 bytes for Keccak. Speedup is calculated as:

$$\text{speedup} = \frac{T_{\text{CPU}}}{T_{\text{GPU}}}. \quad (14)$$

B. Benchmark Results

Table I gives CPU runtime, GPU runtime, throughput, and speedup for both PRFs at depths 8 through 20.

Spongent CPU throughput is roughly flat at 350–375 leaves/sec at all depths; the CPU processes nodes sequentially and each requires the same number of permute calls. GPU throughput scales with depth as more parallel threads become available. At depth 20, Spongent CPU takes 46 minutes and 44 seconds; the GPU completes the same tree in 1.67 seconds. Keccak CPU is also flat at around 200,000 leaves/sec but two to three orders of magnitude above Spongent. Keccak GPU reaches 202 million leaves/sec at depth 20.

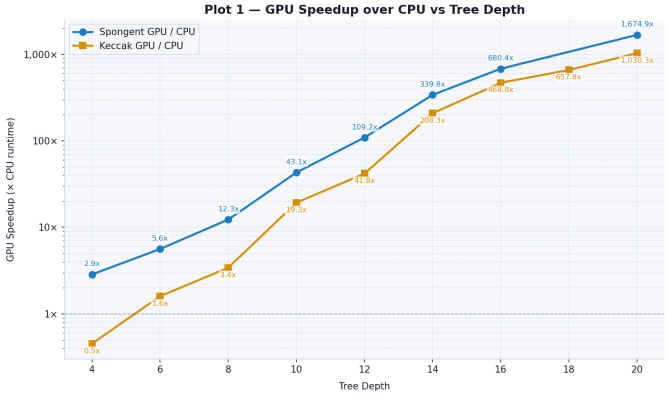

Fig. 1. GPU speedup over CPU vs. tree depth for Spongent and Keccak across all nine measured depths. Both curves grow on a log scale as tree size doubles per level. Spongent reaches 1,675× and Keccak reaches 1,030× at depth 20.

Fig. 1 plots GPU speedup across all nine measured depths. Both curves grow on a log scale, reflecting the fact that tree size doubles with each depth increment while per-thread GPU cost remains roughly constant. Spongent's speedup ratio exceeds Keccak's at depth 20 because Spongent's CPU baseline is far slower not because the GPU handles Spongent more effectively. In absolute wall time at depth 20, Keccak GPU is still 323× faster than Spongent GPU.

Fig. 2 compares absolute throughput on a log scale. At depth 20, Keccak GPU reaches 202 million leaves/sec while Spongent GPU reaches 626,205 leaves/sec, a 323× absolute gap. CPU throughput for both PRFs stays flat throughout, confirming that the CPU cannot exploit the additional parallelism available at deeper levels. This flatness reflects serial execution: a single CPU core processes nodes one at a time at a fixed clock speed. The time required to compute one leaf is constant regardless of tree depth, so leaves-per-second does not change as the tree grows.

Fig. 3 shows raw runtime on a log scale. Execution time doubles with each additional depth level ($O(2^d)$ nodes); the GPU keeps this growth manageable by parallelising all nodes at a given level simultaneously.

Fig. 4 sweeps thread-per-block (tpb) values from 32 to 512 for Spongent GPU at depth 16. Both 64 and 128 threads/block achieve around 373,000 leaves/sec, approximately 10% above

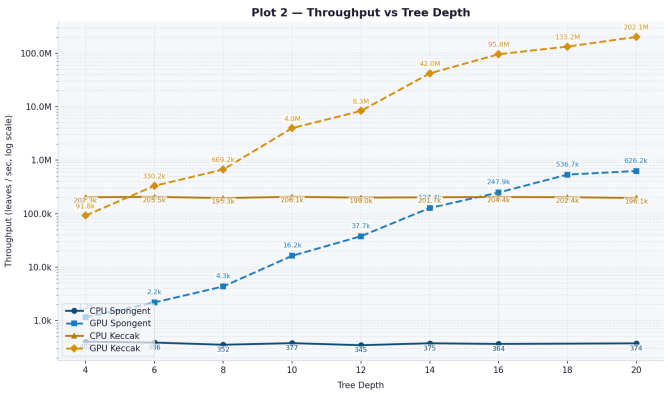

Fig. 2. Throughput (leaves/sec) vs. tree depth on a log scale for all four configurations. Keccak lines sit two to three orders of magnitude above Spongent. Within each PRF, GPU throughput grows with depth while CPU remains flat.

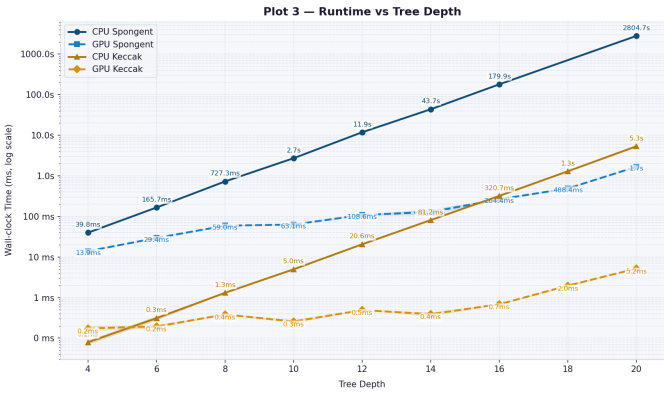

Fig. 3. Wall-clock runtime vs. tree depth on a log scale with standard deviation shading. GPU lines diverge from CPU lines as depth grows.

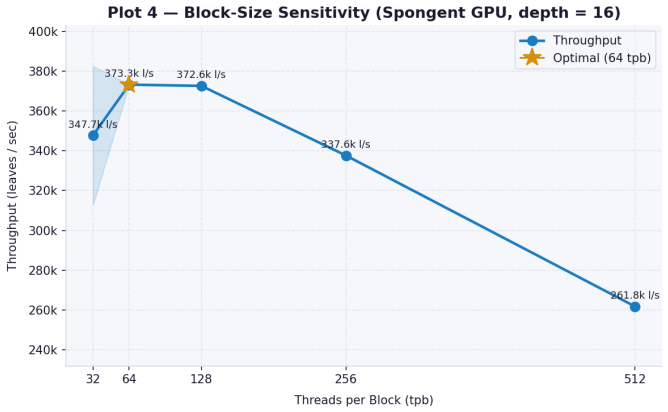

Fig. 4. Block-size sensitivity for Spongent GPU at depth 16 (five repeats). 64 and 128 threads/block achieve approximately 10% higher throughput than the default 256.

TABLE I
CPU AND GPU GGM TREE BENCHMARK RESULTS ON NVIDIA GEFORCE GTX 1080 TI.[1]

PRF	Depth	Leaves	CPU (ms)	GPU (ms)	CPU leaves/s	GPU leaves/s	Speedup
Spongent	8	256	727.317	59.028	351.98	4,336.90	12.3×
Keccak	8	256	1.311	0.383	195,338.30	669,161.46	3.4×
Spongent	12	4,096	11,859.152	108.581	345.39	37,723.06	109×
Keccak	12	4,096	20.581	0.492	199,022.71	8,326,161.79	41.8×
Spongent	16	65,536	179,874.245	264.376	364.34	247,889.16	680×
Keccak	16	65,536	320.665	0.684	204,375.26	95,835,578.72	469×
Spongent	20	1,048,576	2,804,658.541	1,674.492	373.87	626,205.28	1,675×
Keccak	20	1,048,576	5,346.094	5.189	196,138.70	202,068,562.50	1,030×

the 338,000 at the default 256. At 512 threads/block, throughput drops to 262,000 leaves/sec. The decline at high tpb values is consistent with Spongent's register pressure (≈50–60 registers per thread for state and temporary arrays): at 256 threads/block, the per-SM register file may already be saturated, limiting occupancy. The large standard deviation at tpb = 32 (±18.9 ms) reflects scheduling variability when blocks are very small.

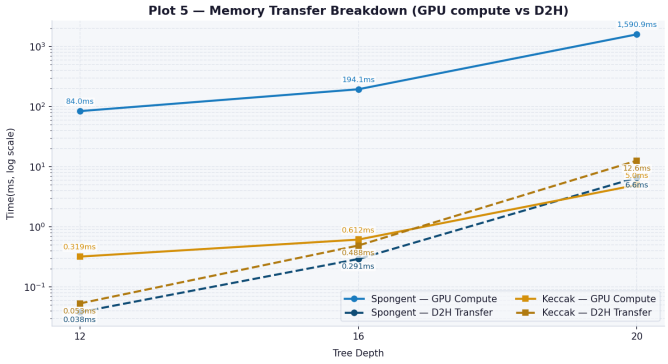

Fig. 5. GPU compute time (H2D seed copy + kernel execution) vs. device-to-host (D2H) copy time at depths 12, 16, and 20.

Fig. 5 separates GPU compute time from device-to-host (D2H) copy time. For Spongent, D2H accounts for under 0.5% of total GPU time at all depths, the bottleneck is compute throughout. For Keccak, the D2H fraction grows from 14.2% at depth 12 to 44.3% at depth 16 and 71.7% at depth 20. At depth 20, the 64 MB Keccak tree takes 3.72 ms to copy back over PCIe while the GPU compute finishes in under 1.47 ms. Keccak's raw compute throughput is therefore even higher than wall-clock figures suggest; at large tree sizes the bottleneck shifts entirely to PCIe bandwidth. The two primitives therefore hit completely opposite hardware limits at scale: Spongent is strictly *compute-bound* (the GPU ALU is the bottleneck, 1667.9 ms(from the figures directly) of computation against only 6.57 ms of transfer at depth 20), while Keccak is strictly *memory-bound*. The D2H transfer time grows predictably with tree size at the fixed physical speed of the PCIe bus; the divergence between the two compute lines reflects the fundamental difference in algorithmic weight.

Fig. 6 examines the crossover zone. Spongent GPU is already 2.9× faster than CPU at depth 4, even 16 nodes is

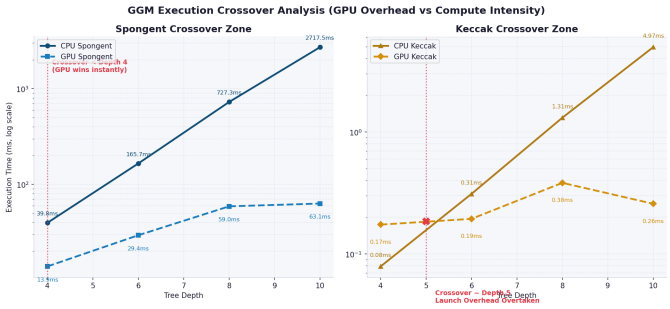

Fig. 6. Crossover analysis at depths 4–10. Spongent GPU is faster than CPU from depth 4 onwards. Keccak GPU is slower than CPU at depth 4 (0.174 ms vs. 0.079 ms) and crosses over at approximately depth 5.

enough work to justify GPU launch overhead. Keccak GPU is slower than CPU at depth 4 because the kernel launch cost (0.174 ms) exceeds the CPU time for a 16-node tree (0.079 ms). The crossover falls at approximately depth 5, and by depth 6 Keccak GPU is already 1.6× faster. The GPU launch overhead is a fixed driver initialisation cost of roughly 0.1 to 0.2 ms, independent of job size. At depth 4, a single CPU core processes all 16 Keccak nodes in 0.079 ms, finishing before the GPU overhead even completes. As depth grows, this fixed cost is amortised across exponentially more nodes and quickly becomes negligible relative to the computation. Spongent never faces this problem because its 70-round permutation makes each node expensive enough that even 16 nodes outweigh the launch cost.

V. DISCUSSION

Why Keccak maps well to GPU. All five Keccak round steps reduce to 64-bit XOR and bitwise rotation, which are native GPU integer ALU operations. The 5×5 word state fits in 25 registers per thread with no memory traffic during the round function. There are no data-dependent branches and no lookup tables beyond the thread's own state. The only non-trivial GPU change was fixing the Rho+Pi step, where the original loop used runtime-valued table indices that forced the temporary array into DRAM. Replacing it with 25 literal assignments by making every index a compile-time constant reduced depth-12 GPU time by 3.8× from a single change.

Why Spongent's ceiling is partially irreducible. Spongent's pLayer is implemented as wire routing in hardware,

requiring zero gates and zero clock cycles. In software, however, it requires 136 extract-and-scatter operations per round across 70 rounds per permutation. This instruction count is dictated by the algorithm specification and cannot be reduced through kernel-level optimisations. The PRESENT S-box exhibits a similar structural limitation: it maps to four logic gates in hardware but requires a lookup or branch in software. Our four GPU optimisations removed the GPU-specific inefficiencies layered on top of these structural costs, including divergence, DRAM spilling, and synchronisation overhead, reducing the depth-12 GPU execution time from 2041 ms to 108 ms. These optimisations were introduced after the baseline GPU implementation failed to deliver sufficient speedup. The remaining throughput gap between Spongent and Keccak therefore reflects the algorithm's inherently higher instruction count on software platforms rather than deficiencies in the GPU implementation.

Two metrics with opposite stories. At depth 20, Spongent's GPU speedup ratio ($1{,}675\times$) exceeds Keccak's ($1{,}030\times$). This can be misleading: the higher ratio reflects Spongent's very slow CPU baseline, not superior GPU acceleration. In absolute wall time, Keccak GPU is $323\times$ faster than Spongent GPU. Relative speedup measures how much the GPU helps compared to the baseline; absolute throughput measures what the hardware actually delivers. Both metrics are informative but neither alone tells the complete story.

Transfer bottleneck at scale. As Fig. 5 shows, Keccak's compute is so fast that PCIe bandwidth becomes the binding constraint at depth 20. This suggests that the practical throughput of Keccak-based GGM in production settings is limited by memory bandwidth rather than compute, and that streaming or in-place tree generation (avoiding the full D2H copy) could yield significant further gains.

Security considerations. Spongent's bit-level operations are constant-time with no data-dependent table lookups, providing resistance against power and electromagnetic side-channel attacks. This is a meaningful property for IoT devices where a physical adversary can probe the hardware. In the GGM tree use case, server-side batch OT on a GPU,the threat model is network-level; a remote adversary cannot measure power traces from a data-centre GPU. For this deployment context, Keccak is the appropriate primitive. Spongent remains the correct choice for the embedded applications it was designed to serve.

Performance versus security tradeoff. The choice between Keccak and Spongent for GGM tree construction represents a tradeoff between computational throughput and physical security. Keccak offers overwhelming performance advantages in software environments, reaching over 202 million leaves/sec on the GPU because its mathematical operations map natively to GPU ALUs. However, Spongent trades this software throughput for strict hardware-level constraints: it prioritizes extremely low gate counts and constant-time bit-level operations that provide inherent resistance against physical side-channel attacks (such as power and electromagnetic analysis).

Therefore, the optimal primitive depends strictly on the deployment threat model. In a server-side deployment (e.g., generating GGM trees for batch Oblivious Transfer on a data-center GPU), the threat model is network-based; a remote adversary cannot probe the GPU's power consumption. In this high-throughput, remote-adversary context, Keccak is the definitive choice. Conversely, if the cryptographic tree generation is taking place on the edge (e.g., IoT microcontrollers or physical tokens) where physical adversaries can probe the device, Spongent's side-channel resilience makes it the superior choice, validating its structural software penalty.

VI. Conclusion

We implemented GGM tree expansion on GPU using Keccak-f1600 and Spongent-128, with full CPU reference implementations. At depth 20, Keccak GPU achieves $1{,}030\times$ speedup (202 M leaves/sec) and Spongent GPU achieves $1{,}675\times$ speedup (626 K leaves/sec). The primary optimisation for both PRFs was the same: identifying that runtime-indexed temporary arrays force NVCC to allocate them in off-chip DRAM and replacing runtime indices with compile-time-constant equivalents. This single structural change of moving working state from DRAM to registers accounts for the majority of the GPU improvement in both kernels. The absolute throughput gap between the two PRFs (Keccak GPU is $323\times$ faster at depth 20) is structural: Keccak's round function consists of native 64-bit ALU operations, while Spongent's pLayer is a bit permutation designed for silicon that costs 136 software instructions per round regardless of how the kernel is written.

References

[1] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *Journal of the ACM*, vol. 33, no. 4, pp. 792–807, 1986.

[2] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending oblivious transfers efficiently," in *Advances in Cryptology – CRYPTO 2003*, ser. Lecture Notes in Computer Science, vol. 2729. Berlin, Germany: Springer, 2003, pp. 145–161.

[3] H. Choi and S.-C. Seo, "Fast implementation of SHA-3 in GPU environment," *IEEE Access*, vol. 9, pp. 144,574–144,586, 2021.

[4] C. Wang and X. Chu, "GPU accelerated Keccak (SHA3) algorithm," arXiv preprint arXiv:1902.05320, Feb. 2019.

[5] National Institute of Standards and Technology, "SHA-3 standard: Permutation-based hash and extendable-output functions," FIPS PUB 202, Aug. 2015.

[6] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede, "SPONGENT: A lightweight hash function," in *Cryptographic Hardware and Embedded Systems – CHES 2011*, ser. Lecture Notes in Computer Science, vol. 6917. Berlin, Germany: Springer, 2011, pp. 312–325.

[7] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems – CHES 2007*, ser. Lecture Notes in Computer Science, vol. 4727. Berlin, Germany: Springer, 2007, pp. 450–466.

[8] NVIDIA Corporation, "CUDA C++ programming guide," NVIDIA Documentation. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[9] NVIDIA Corporation, "NVIDIA introduces the GeForce GTX 1080 Ti," NVIDIA Newsroom, 2017. [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-introduces-the-beastly-geforce-gtx-1080-ti-fastest-gaming-gpu-ever