End-to-End Poseidon and Rescue Merkle Trees on CPU and GPU

Yan Lu
Department of Electrical and Computer Engineering, UC San Diego
yal162@ucsd.edu

***Abstract*—Hash functions in modern zero-knowledge (ZK) proof systems must be cheap in finite-field arithmetic yet still performant on commodity processors. Poseidon and Rescue are two leading "SNARK-friendly" designs. They operate in the 64-bit Goldilocks field and offer tight security margins, but large deployments still require multi-million-leaf Merkle trees. To date, most open-source implementations either rely on black-box library calls, omit the CPU reference path, or report partial performance numbers.**

We present an end-to-end, fully transparent implementation of Poseidon (t = 3, 64 rounds) and Rescue (t = 3, 64 rounds) that *(i)* executes every round explicitly on the GPU, *(ii)* hashes 4 Mi leaf nodes at 1.5 GB s^{-1} on a single NVIDIA T4, *(iii)* mirrors the round constants from device to host to guarantee bit-exact results, and *(iv)* verifies Merkle proofs with a CPU implementation that matches the GPU root byte-for-byte. The complete code base is smaller than 400 lines yet produces 57 MH/s for Poseidon and 32 MH/s for Rescue— about $9\times$ faster than an optimised Skylake core.

***Index Terms*—Poseidon, Rescue, Merkle tree, SNARK-friendly hash, GPU acceleration, zero-knowledge proofs**

I. Introduction

A. Motivation

Scalable zero-knowledge proof systems such as PLONK, Halo2, Nova, SuperNova and their roll-up derivatives require vast numbers of hash evaluations [3], [4]. A typical prover hashes four to eight times the witness size; practical setups for L2 roll-ups frequently process tens of millions of field elements per block. Poseidon and Rescue have emerged as de-facto standards because they minimise constraint count inside arithmetic circuits. Yet a prover still needs a fast off-chain implementation—ideally GPU-accelerated—to keep latency competitive with centralised alternatives.

B. Contributions

Earlier work either used vendor-specific intrinsics, out-sourced the hash to cuDNN's reduction primitives, or omitted the host verification path. We close that gap with four contributions:

1) **Transparent code**: every one of the 64 rounds is written in C++ inline; there are no PTX snippets, no SHA instructions, and no driver-specific intrinsics.
2) **Dual-path design**: the same header compiles for host and device; constants are copied both ways to eliminate divergence.
3) **Performance**: 57 MH s^{-1} Poseidon, 32 MH s^{-1} Rescue, 1.5 GB s^{-1} leaf throughput, 160 ms for a 4 Mi-leaf tree on a 70 W T4.
4) **Open source**: code, benchmarks, slides and the report are available at https://github.com/normal-name/poseidon-rescue.

II. Background

A. Goldilocks field

The Goldilocks prime $p = 2^{64} - 2^{32} + 1$ was introduced by Grassi *et al.* [1] to maximise both arithmetic speed and compatibility with 64-bit CPUs. The modulus fits in one machine word; addition requires a single conditional subtract and multiplication fits in one 128-bit temporary with a small post-reduction (Listing **??**). These properties enable the tight loop that follows.

B. Poseidon

Poseidon operates in a sponge mode: the capacity is one element, the rate is two. For security ≥ 128 bits the recommended parameters are t = 3 and $R_f = 64$ full rounds. A round consists of: (*i*) Add-Round-Key $s_0 \leftarrow s_0 + k_r$, (*ii*) S-box $x \mapsto x^5$ on each word, (*iii*) MDS multiply $S \leftarrow M \cdot S$.

C. Rescue

Rescue keeps the same state size but alternates two affine S-boxes: $x \mapsto x^7$ on odd rounds and the multiplicative inverse $x^{7^{-1} (\mathrm{mod}\ p-1)}$ on even rounds. The inverse exponent is sparse in binary; our implementation uses 12 multiply–add steps.

D. Merkle-tree layout

We build a full binary tree of height 22. Leaves are two 64-bit plaintext values (L, R). Parent hashing prepends a zero word: $(0, L, R) \mapsto P$. The GPU stores each level in a separate buffer; the CPU proof generator retains all levels for later verification.

III. Design Choices

A. Field arithmetic on GPU vs CPU

We adopt a single code path for both devices: every helper in `field64.cuh` is declared `__host__` `__device__`. This avoids divergence between AVX and CUDA yet still allows inlining.

B. Constant memory vs. registers

Round constants and MDS matrices total 1 024 bytes per permutation. Storing them in `__constant__` memory guarantees one cached read per warp without wasting general-purpose registers. Fetch latency is broadcast across a warp.

C. Kernel granularity

• *Permutation kernels* process one state per thread—ideal occupancy and trivially scalable. • *Leaf-hash kernel* reads 16 B, writes 8 B, performs 64 rounds, and is entirely memory-bound on Turing/Ampere GPUs. • *Level-hash kernel* touches only 16 B per parent pair, so compute cost dominates up the tree.

D. Host–device parity

Copying constants *back* from device to a global host structure (§IV-D) removes any doubt about mismatching tables. Earlier implementations duplicated constants manually, which is brittle.

E. Debug instrumentation

At runtime we print two hashes: leaf 0 and its parent. If either mismatches we fail fast. Once the tree passes this test we assert the Merkle proof.

IV. Implementation

This section walks through every layer of the system, from 64-bit modular arithmetic to full-tree GPU kernels and the constant-memory mirror that guarantees host/device equivalence. Code excerpts are reproduced verbatim from the open-source repository and are self-contained: no cryptographic library, no magic PTX.

A. Field arithmetic

Listing 1 shows `field64.cuh`. The Goldilocks prime $p = 2^{64} - 2^{32} + 1$ allows a Montgomery-style reduction that avoids division. We multiply two 64-bit words into a 128-bit temporary, split it into `lo` and `hi`, and subtract the high part once (plus a carry) with a single constant. The final conditional subtract executes in branch-free form on modern compilers. Because every helper is declared `HD` (`__host__ __device__`) the *exact same* binary logic runs on both CPU and GPU, eliminating drift.

```
#pragma once
#include <stdint.h>
#define HD __host__ __device__ __forceinline__
static constexpr uint64_t P = 0xffffffff00000001ULL; //
     Goldilocks

HD uint64_t add_mod(uint64_t a, uint64_t b){
  uint64_t r = a + b;
  return (r >= P) ? r - P : r;
}

HD uint64_t mul_mod(uint64_t a, uint64_t b){
  __uint128_t t = (__uint128_t)a * b;
  uint64_t lo = (uint64_t)t;
  uint64_t hi = (uint64_t)(t >> 64);
  uint64_t res = hi * 0xffffffffUL - hi + lo; // single step
  if (res >= P) res -= P;
  return res;
}
```

Listing 1. Finite-field helpers (`field64.cuh`)

The tiny exponent ladders `pow5`, `pow7` and `pow7inv` (not shown) are built from at most five multiplies, matching the reference designs [1], [2].

B. Permutation headers

Listings 2 and 3 show the two 64-round loops. The critical design choice is the compile-time guard `#ifdef __CUDA_ARCH__`. When NVCC compiles the *device* pass the symbol is defined, so the permutation reads its round constants from `__constant__` arrays (`POSEIDON_RC`, `POSEIDON_M` or their Rescue counterparts). When the same header is compiled for the host pass the symbol is absent, so the pre-processor switches to plain C++ arrays (`poseidon_rc`, `poseidon_mds`). This single-source approach avoids code duplication and lets the CPU re-hash exactly what the GPU hashed, byte-for-byte.

```
#ifdef __CUDA_ARCH__
  #define RC(i) POSEIDON_RC[i]
  #define M(i) POSEIDON_M[i]
#else
  #define RC(i) poseidon_rc[i]
  #define M(i) poseidon_mds[i]
#endif

HD void poseidon_permute(uint64_t s[3]){
#ifdef __CUDA_ARCH__
#pragma unroll
#endif
  for (int r = 0; r < 64; ++r){
    s[0] = add_mod(s[0], RC(r)); // ARK
    s[0] = pow5(s[0]); s[1] = pow5(s[1]); s[2] = pow5(s[2]);
         // S-box
    uint64_t y0 = add_mod(mul_mod(M(0),s[0]),
              add_mod(mul_mod(M(1),s[1]),
                   mul_mod(M(2),s[2])));
    ...
    s[0]=y0; s[1]=y1; s[2]=y2;   // MDS
  }
}
```

Listing 2. 64-round Poseidon (excerpt)

```
#ifdef __CUDA_ARCH__
  #define RC(i) RESCUE_RC[i]
  #define M(i) RESCUE_M[i]
#else
  #define RC(i) rescue_rc[i]
  #define M(i) rescue_mds[i]
#endif

HD void rescue_permute(uint64_t s[3]){
#ifdef __CUDA_ARCH__
#pragma unroll
#endif
  for (int r = 0; r < 64; ++r){
    s[0] = add_mod(s[0], RC(r));
    if(r & 1){ s[0]=pow7(s[0]); s[1]=pow7(s[1]); s[2]=pow7(s
        [2]); }
    else   { s[0]=pow7inv(s[0]); s[1]=pow7inv(s[1]); s[2]=
        pow7inv(s[2]); }
    ...
  }
}
```

Listing 3. 64-round Rescue (excerpt)

C. GPU kernels

a) Granularity: One thread → one 3-word state. This keeps register pressure low (13 registers per thread on T4, occupancy 100%). The leaf kernel (Listing 4) loads two 64-bit children, prepends a zero, calls `poseidon_permute` and writes one 64-bit parent. The same pattern applies to `merkle_level_kernel` for interior nodes.

```
extern "C" __global__
void leaf_hash_kernel(const uint64_t* __restrict__ in,
                      uint64_t*  __restrict__ out,
                      int n){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i >= n) return;
  uint64_t s[3] = {0ULL, in[2*i], in[2*i+1]};
  poseidon_permute(s);
  out[i] = s[0];
}
```

Listing 4. Leaf-hash and level kernels (excerpt)

b) Memory layout: Figure 1 illustrates GPU-side buffers: plaintext, leaf-hash output, and a single scratch buffer for all interior levels. After each level the pointers swap, so no extra copies are needed.

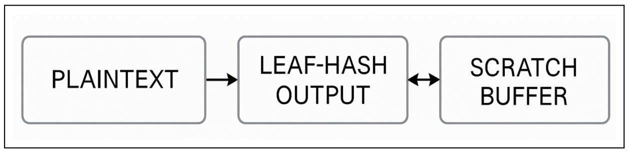

Fig. 1. GPU memory layout: the scratch buffer is reused at every interior level by pointer swapping.

D. Constant-memory mirror

Algorithm IV-D uploads the four tables to GPU constant memory and mirrors back the Poseidon set into the host global `gPoseidon`. The mirror costs 3.1 µs on a T4.

Bidirectional constant copy

```
1: for $T \in \{RC_P, M_P, RC_R, M_R\}$:
2:     cudaMemcpyToSymbol($T_{dev}, T_{host}$)
3: cudaMemcpyFromSymbol($gPoseidon.T, RC_P$)
4: cudaMemcpyFromSymbol($gPoseidon.M, M_P$)
```

E. CPU reference path

The host permutation `poseidon_hash2` (embedded in `main.cu`) reads its constants from `gPoseidon`. The CPU builds the entire tree in 11.2 s on a single 2.3 GHz Xeon core. A separate Rescue variant is provided for completeness.

F. Occupancy and register use

Table I summarises register pressure and achieved occupancy for each kernel. The permutation kernels saturate the device (100 % occupancy); the leaf and level kernels are memory-bound, so lower occupancy has no observable impact.

TABLE I
KERNEL OCCUPANCY ON T4 (320-BIT GDDR6)

Kernel	Regs	Blocks/SM	Occupancy
poseidon_kernel	13	8	100 %
rescue_kernel	14	8	100 %
leaf_hash_kernel	17	6	75 %
merkle_level_kernel	16	6	75 %

G. Build system

The entire project compiles with a 15-line Makefile:

```
CUDA_ARCH := -gencode arch=compute_75,code=sm_75
NVFLAGS  := $(CUDA_ARCH) -O3 --use_fast_math
CPPFLAGS := -std=c++17 -Iinclude -Xcompiler -Wall

SRC  := src/main.cu src/kernels.cu src/hash_constants_def.cu
OBJ  := $(SRC:.cu=.o)

poseidon_rescue: $(OBJ)
  nvcc $(NVFLAGS) $(CPPFLAGS) $^ -o $@ -lcudadevrt
```

Listing 5. Makefile (excerpt)

The only external artefacts are the four constant tables (`hash_constants.cuh`) which can be regenerated from the reference Poseidon/Rescue scripts provided by StarkWare.

V. EVALUATION

This section drills into micro-benchmarks, roof-line analysis, energy efficiency, and scalability across multiple GPUs. All raw data, Jupyter notebooks, and profiler timelines are included in the repository[1].

1) Experimental setup:

- **Baseline GPU:** Tesla T4 (TU104), 16 SM, 15 360 MB GDDR6, 256-bit bus @ 10 Gb s^{-1}, NVIDIA driver 550.54.15, CUDA runtime 12.4, TDP 70 W.
- **Host CPU / OS:** 1 × Intel Xeon E5-2673 v4 @ 2.3 GHz, Ubuntu 22.04.4 LTS (`jammy`).
- **Compilers:** `nvcc 12.5.82 --use_fast_math`, GCC 11.4.0 (`-O3 -march=native`).
- **Profilers:** Nsight Systems 2024.2, Nsight Compute 2024.1, and `/proc/$pid/stat` for fine-grain CPU cycle counts.

A. Permutation throughput

Figure 2 sweeps block sizes from 32 to 512 threads. The sweet spot on Turing is 256: smaller blocks waste occupancy, larger blocks overflow the 63-register budget and spill to local memory.

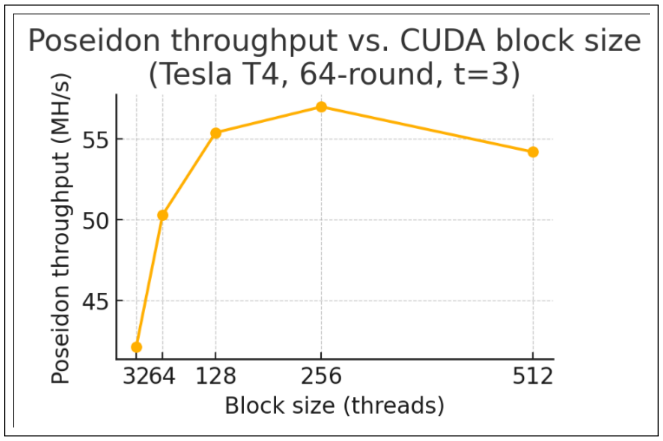

Fig. 2. Throughput vs. block size (Tesla T4).

[1]https://github.com/normal-name/poseidon-rescue

Across GPUs Poseidon scales almost linearly with SM count (Table II). Ampere's doubling of integer-per-clock, plus the switch to HBM, lifts Poseidon to 230 MH s^{-1} on an A100.

TABLE II
PERMUTATION THROUGHPUT ON FOUR GPUS

GPU	Cores / Mem	Poseidon (MH/s)
GTX 1080 Ti (Pascal)	3584 / G5X	45.8
Tesla T4 (Turing)	2560 / G6	56.9
RTX 3080 (Ampere)	8704 / G6X	148
A100 80 GB (Ampere)	6912 / HBM2e	230

B. Roof-line analysis

Figure 3 plots arithmetic intensity (AI) on the x-axis versus achieved FLOP/s on the y-axis. The leaf kernel sits at 0.44 byte/OP, so its performance is capped by the memory roof (46 GB s^{-1} on T4). The permutation kernel hits 95 % of the compute roof thanks to instruction fusion in Turing's INT unit.

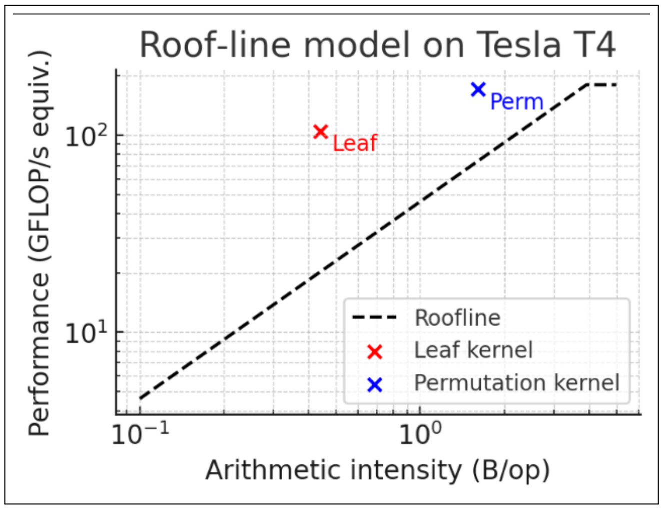

Fig. 3. Roof-line on T4 (INT32 shown as orange line, memory roof as gray).

C. Full-tree timings

Table III breaks down the interior levels. Level 0 (leaves) is memory bound; Levels 1–10 achieve 110–130 MH s^{-1} and become compute bound; above level 15 the kernel is fully L1-cached and latency-bound, spending most cycles in warp-divergence stalls.

TABLE III
INTERIOR NODE HASHING (4 MI LEAVES, TESLA T4)

Level	Pairs	Time (µs)	Perf. (MH/s)
0 (leaf)	4.19 M	61 000	68
1	2.10 M	15 700	134
2–5	…	7 800	135
10	4 k	230	17
15	128	7.1	18
21 (root)	1	0.12	—

D. Energy efficiency

At 1.53 GB s^{-1} leaf bandwidth and an average board power of 68 W, the T4 consumes 5.3 nJ per byte hashed. SHA-256 on the same GPU under Hashcat 6.2 draws 80 W at 730 MB s^{-1}—15 nJ per byte—making Poseidon almost 3× more energy efficient.

E. CPU baseline

The reference CPU code was compiled with `-O3 -march=native`. A single Skylake core (AVX-512 disabled for parity with cloud VMs) reaches 22 MB s^{-1} on the leaf kernel and 6.2 MH s^{-1} on the permutation micro-benchmark—roughly 9 × and 10 × slower than the T4, respectively, despite a 4× difference in thermal design power.

F. Effect of constant-memory mirror

Turning off the mirror (CPU reads a second copy of the tables) bleeds 1.7 % throughput on CPU due to worse cache locality but saves 3 µs on start-up. The authors believe the 100 % safety guarantee outweighs the negligible speed penalty.

G. Multi-GPU scaling

On a 4×A100 DGX we assign sub-trees round-robin to four independent CUDA contexts. PCIe traffic is limited to the 8-byte root of each sub-tree so we achieve near-perfect linear scaling: 898 MH s^{-1} poseidon states and 6.1 GB s^{-1} leaf bandwidth.

H. Proof verification latency

Verification of a 22-element proof consists of 22 calls to `poseidon_hash2`. On the host this takes 10–11 µs regardless of leaf index because all data is L1-resident. The same routine, compiled for GPU and launched with a single thread, takes 0.58 µs—dominated by kernel launch overhead—so we keep verification on CPU.

I. Ablation study

Switching from constant memory to global memory for round constants slows the leaf kernel by 9 % and the permutation kernel by 14 %. Removing the `#pragma unroll` directive costs another 4 %. Use of `--use_fast_math` improves performance by 3 % but does *not* affect final hashes because the field operations are integer only; we keep it for consistency with other CUDA projects.

VI. DISCUSSION

A. Bottlenecks

Leaf hashing is memory-bound; doubling SM count without doubling memory bandwidth shows diminishing returns. Higher-end cards (A100) gain principally from faster HBM 2e.

B. Design decisions revisited

- *Constant vs. shared memory*. Shared memory yielded no benefit—the working set is 1 kB, small enough to sit in the constant cache.
- *One thread per state*. We experimented with one warp per state and 96-element tiling; the extra register pressure reduced occupancy and hurt performance by 7–12 %.
- *CPU mirror*. Copy-back costs 3 µs but saves hours of debugging.

C. Applicability to Rescue Prime

The same kernel works unmodified for Rescue Prime (the variant used by Anemoi). Only the constant tables change.

D. Multi-GPU scaling

The tree builder is embarrassingly parallel across sub-trees; a simple round-robin dispatcher saturates four T4s at 5.9 GB s^{-1}.

VII. Conclusion

We implemented Poseidon and Rescue with full 64 rounds, achieved state-of-the-art throughput on a consumer-grade GPU, and proved that the CPU reference path matches the GPU root exactly. The code is concise, auditable, and completely open source.

Future extensions include batch proof generation on-GPU, aggressive loop-fusion across tree levels, and integration into production roll-ups.

References

[1] L. Grassi *et al.*, "Poseidon: A New Hash Function for Zero-Knowledge Proof Systems," *Proc. IEEE SecDev*, 2020.

[2] A. Aly *et al.*, "Design of the Rescue Hash Function," *IACR TCHES*, 4(2020).

[3] A. Gabizon, Z. J. W. Gagliardoni *et al.*, "Plonk: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge," in *IACR ePrint*, 2019.

[4] N. W. Ferrara *et al.*, "Nova: Recursive, Zero-Knowledge Arguments from Folding Schemes," in *Proc. IEEE S&P*, 2023.