

FAEST on GPUs: Addressing Field Multiplication as a Bottleneck*

Minseop Choi

NORMA, Inc. Seoul, Republic of Korea

minsabb13@norma.co.kr

Abstract. FAEST is a post-quantum signature scheme that combines VOLE-in-the-Head with QuickSilver to prove knowledge of a secret key. It relies on well-established symmetric primitives and offers small public keys, but signing and verification are computationally expensive. We present an NVIDIA GPU implementation of the key-generation core and the QuickSilver procedures `OWFProve` and `OWFVerify` for the standard and EM constructions in both s and f variants. Profiling shows that once AES is accelerated by hardware, field multiplication in QuickSilver becomes the largest cost throughout this parameter grid. In FAEST-128f, field multiplication takes about 82% of signing time. To address this bottleneck in our GPU implementation of FAEST, we use PTX `clmad` for native carry-less arithmetic and Karatsuba to reduce the number of limb products. The `clmad` build improves throughput across all twelve `OWFProve` and `OWFVerify` paths, and its advantage grows with the field width. On an RTX A5000, PTX `clmad` raises standard `OWFProve` throughput to 1.99×, 2.45×, and 2.82× that of the software-multiplication build at 128, 192, and 256 bits, respectively. Together, PTX `clmad` and Karatsuba deliver 26.6× the throughput of the software-multiplication build in the field-multiplication-heavy kernel of standard 256-bit `OWFProve`. When all `OWFProve` stages are included, the combined build reaches 2.9× and 3.1× the software throughput for standard and EM at 256 bits, respectively. These results establish native carry-less arithmetic as a key design choice for GPU implementations of FAEST, with Karatsuba providing an additional improvement for 256-bit proving.

Keywords: FAEST · Post-Quantum Cryptography · GPU · QuickSilver · VOLE-in-the-Head · Carry-less Multiplication

1 Introduction

Advances in quantum computing have accelerated research and standardization in Post-Quantum Cryptography (PQC). On a large-scale quantum computer, Shor’s algorithm could efficiently solve integer factorization and discrete logarithms [Sho94], compromising RSA and elliptic-curve cryptography. Grover’s algorithm could also provide a quadratic speedup for exhaustive key search against symmetric cryptography [Gro96]. Transitioning today’s public-key infrastructure to quantum-resistant alternatives will take time, so these alternatives must be standardized and deployment-ready before large-scale quantum computers pose a practical threat. NIST began its PQC standardization process in 2016 and published ML-KEM, ML-DSA, and SLH-DSA as FIPS 203, FIPS 204, and FIPS 205 in 2024 [Nat24b, Nat24a, Nat24c]. ML-KEM provides key encapsulation, while ML-DSA and SLH-DSA provide digital signatures. ML-KEM and ML-DSA rely on the hardness

*This is the version submitted for peer review on 15 July 2026, independently of and concurrently with ePrint 2026/1409. A revised version is in preparation.

of structured-lattice problems, whereas SLH-DSA relies on the security properties of cryptographic hash functions. These foundations give ML-DSA and SLH-DSA distinct performance and size characteristics [Nat24a, Nat24c]. NIST is also running a separate Additional Digital Signatures process to offer more choices for different settings and reduce reliance on any single security foundation. FAEST, one of nine Round 3 candidates announced in 2026, is based on symmetric primitives [ABC⁺26].

FAEST uses VOLE-in-the-Head [BBdSG⁺23] and QuickSilver [YSWW21] to prove knowledge of a secret key that maps a public input to its corresponding output under a one-way function. Its security rests on information-theoretic primitives and symmetric primitives such as AES and SHAKE. A FAEST public key is small, containing only an input-output pair for the one-way function. At each of the three NIST security categories, FAEST provides the standard and Even-Mansour (EM) constructions and the **s** and **f** parameter sets, offering choices among security level, signature size, and execution cost. Together, FAEST’s reliance on symmetric primitives and its small public keys distinguish it from current PQC standards and make it attractive when public-key size matters.

However, this design makes signing and verification computationally expensive, with substantial work in both symmetric primitives and zero-knowledge proofs. The VOLE-in-the-Head stage repeatedly invokes an AES-based PRG to generate pseudorandom values, and the one-way function in standard FAEST also uses AES. A single signature requires many AES block computations, making symmetric computation a potentially major cost. QuickSilver represents the nonlinear operations of the one-way function as multiplication constraints over $\text{GF}(2^\lambda)$. The QuickSilver routines **OWFProve** and **OWFVerify** perform many field multiplications. As the field width increases, each field multiplication requires a larger number of smaller component multiplications. AES and QuickSilver create two distinct sources of cost, and accelerating one can leave the other to limit overall performance. This interaction is a major reason why efficient FAEST signing and verification are difficult to implement.

Related Work. Prior work on accelerating FAEST has focused mainly on CPUs and FPGAs. Lee et al. [LKS⁺26] use a register-resident AES S-box and NEON table lookups on ARMv8 processors without the Crypto Extensions, and they parallelize VOLE tasks with pthreads. Their approach improves FAEST implementations dominated by software AES but does not target QuickSilver field arithmetic. Medina Anton [Med24] presents an FPGA implementation of the core inner functions of FAEST. Deshpande et al. [DHSY24] implement the proof computations of SDitH on dedicated hardware. Liu et al. [LJWJ24] implement the proof computations of MPCitH and Picnic3 on dedicated hardware. These works demonstrate the feasibility of hardware implementations for MPCitH-family signatures but target neither GPU instructions nor the GPU memory hierarchy.

GPU acceleration of PQC signatures has focused mainly on lattice schemes. Shen et al. [SYD⁺24] optimize the NTT, sampling, and memory access of Dilithium for GPUs. Lee et al. [LZS⁺24] compare GPU implementations of Falcon and Mitaka. The NTT and sampling in these schemes have a computational structure different from FAEST’s field-arithmetic workload. Early AES-GCM work precomputes the counter-mode AES encryptions of GCM on a GPU [SF11], and recent work processes the binary-field multiplication in GHASH on GPUs [LPACGSR26]. GCM optimizations based on the CLMUL instruction use Karatsuba to reduce the number of carry-less multiplications and aggregated reduction to reduce the number of reductions [GK14]. These studies do not examine when a local arithmetic speedup translates into overall GPU throughput for the QuickSilver computation in FAEST.

Our Contributions. We implement FAEST’s key-generation core, **OWFProve**, and **OWFVerify** on a GPU. Our main contributions are as follows.

1. **Identifying field multiplication as FAEST’s next bottleneck after AES acceleration.** By comparing CPU implementations, we find that field multiplication becomes the next major cost when AES acceleration is enabled, accounting for about 82%, 97%, and 96% of signing time in 128f, 192f, and 256f, respectively. This result shows that the operations available on the target hardware determine which part should be optimized first.
2. **Quantifying GPU-native carry-less multiplication.** Replacing software field multiplication with the native GPU instruction increases standard OWFProve throughput by $1.99\times$, $2.45\times$, and $2.82\times$ at 128, 192, and 256 bits, respectively. The same trend appears in all twelve execution paths formed by the three field widths, the two one-way functions, and OWFProve and OWFVerify, and the benefit grows with the field width. These results show that native carry-less multiplication is the most important arithmetic optimization in our GPU implementation.
3. **Determining when multiplication-count reduction improves throughput.** Karatsuba further reduces the number of limb products in field multiplication. Combined with PTX `clmad`, it delivers $26.6\times$ the throughput of the software-multiplication build in the standard 256-bit field-multiplication-heavy computation. When data preparation is included, complete-path throughput reaches $2.9\times$ the software baseline. Karatsuba provides an additional procedure-level improvement only for standard and EM 256-bit OWFProve. Accelerating field multiplication also exposes data rearrangement as the next bottleneck.
4. **Implementing FAEST’s core on GPUs.** Our GPU implementation covers the key-generation core, OWFProve, and OWFVerify for all twelve FAEST parameter sets. For every parameter set, the outputs of our GPU implementation match those of the reference implementation byte for byte. This agreement confirms the functional correctness of these GPU implementations within the evaluated scope, and all reported performance results use these validated implementations.

The rest of this paper is organized as follows. Section 2 provides the necessary background. Section 3 defines the scope of the implementation and evaluation. Section 4 analyzes the computational bottlenecks that motivate our GPU design, and Section 5 presents our GPU implementation and arithmetic optimizations. Section 6 presents the experimental evaluation. Section 7 concludes the paper.

2 Preliminaries

2.1 Notation

The computational security parameter is $\lambda \in \{128, 192, 256\}$, and QuickSilver operates over the finite field $\text{GF}(2^\lambda)$. We denote the irreducible polynomial that defines this field by P_λ . A FAEST key pair consists of $pk = (x, y)$ and $sk = k$, with $E_k(x) = y$. We write the extended witness as $w \in \{0, 1\}^\ell$, where ℓ denotes the bit length of the extended witness and is fixed by each parameter set. The signer’s VOLE values are (u, V) , the verifier’s VOLE value is Q , and the global key linking the two views is Δ . The masked witness is $d = w \oplus u[0..\ell]$, and the three Fiat–Shamir challenges are denoted by *chall1*, *chall2*, and *chall3*.

We denote the coefficient vectors of the QuickSilver constraint polynomial by (a_0, a_1, a_2) and the QuickSilver proof elements by $(\tilde{a}_0, \tilde{a}_1, \tilde{a}_2)$. The specification names the two QuickSilver procedures `ZK.OWFProve` and `ZK.OWFVerify`. Algorithms 1 and 2 retain these formal names, while the surrounding text uses the short names `OWFProve` and `OWFVerify`.

Algorithm 1 Computational flow of FAEST.Sign, adapted from [BBB⁺25]

Require: message msg , secret key sk , public key pk
Ensure: signature σ

```

1: parse  $pk$  as  $(x, y)$ ;  $ctr \leftarrow 0$ ; sample  $\rho$ ;  $\mu \leftarrow H_0^2(pk \parallel msg)$ 
2:  $(r, iv_{pre}) \leftarrow H_3(sk \parallel \mu \parallel \rho)$ ;  $iv \leftarrow H_4(iv_{pre})$ 
3:  $(com, decom, \{c_i\}, u, V) \leftarrow \text{FAEST.VOLECommit}(r, iv, \ell + 2\lambda + B)$ 
4:  $chall1 \leftarrow H_1^2(\mu \parallel com \parallel \{c_i\} \parallel iv_{pre})$ 
5:  $\tilde{u} \leftarrow \text{VOLEHash}(chall1, u)$ ;  $\tilde{V} \leftarrow \text{VOLEHash}(chall1, V)$ 
6:  $w \leftarrow \text{AES.ExtendWitness}(sk, x)$ ;  $d \leftarrow w \oplus u[0..\ell]$ 
7:  $chall2 \leftarrow H_2^2(chall1 \parallel \tilde{u} \parallel \tilde{V} \parallel d)$ 
8:  $(\tilde{a}_0, \tilde{a}_1, \tilde{a}_2) \leftarrow \text{ZK.OWFProve}(w, u[\ell..\ell + 2\lambda], V[0..\ell + 2\lambda], pk, chall2)$ 
9: repeat
10:    $chall3 \leftarrow H_3^2(chall2 \parallel \tilde{a}_0 \parallel \tilde{a}_1 \parallel \tilde{a}_2 \parallel ctr)$ 
11:   if the grind bits of  $chall3$  are nonzero then
12:      $ctr \leftarrow ctr + 1$ ; continue
13:   end if
14:    $I \leftarrow \text{DecodeAllChall3}(chall3 \text{ without its grind bits})$ 
15:    $decom_I \leftarrow \text{BAVC.Open}(decom, I)$ 
16:   if  $decom_I = \perp$  then
17:      $ctr \leftarrow ctr + 1$ 
18:   end if
19: until  $decom_I \neq \perp$ 
20: return  $\sigma = (\{c_i\}, \tilde{u}, d, \tilde{a}_1, \tilde{a}_2, decom_I, chall3, iv_{pre}, ctr)$ 

```

We use B for the VOLE padding parameter used in the specification and use N_{batch} for the number of independent calls processed in one GPU batch.

2.2 FAEST

FAEST is a digital signature scheme in which the signer proves knowledge of a secret key that maps a public input to its stated output under a one-way function. The three choices of λ correspond to NIST security categories 1, 3, and 5. At each security level, both standard FAEST and FAEST-EM provide an **s** and an **f** parameter set. Combining the two families, three security levels, and two parameter choices gives the twelve parameter sets defined in the specification [BBB⁺25]. These parameter sets allow the same QuickSilver computation to be compared across different field widths and one-way functions.

Key generation samples a candidate secret key k and rejects it if its two least significant bits, $k[0]$ and $k[1]$, are both one. Once a valid candidate is found, it samples a uniformly random public input x , which is 128 bits in standard FAEST and λ bits in FAEST-EM, and returns $sk = k$ and $pk = (x, y)$ with $y = E_k(x)$.

Standard FAEST-128 uses $y = \text{AES}_{128_k}(x)$. FAEST-192 and FAEST-256 concatenate the outputs of AES evaluated on x and $x \oplus 1$, where 1 is the 128-bit little-endian encoding of the integer one. FAEST-EM instead uses $E_\lambda^{\text{EM}}(k, x) = \text{Rijndael-}\lambda_x(k) \oplus k$.

FAEST.Sign first constructs a VOLE commitment, then computes a QuickSilver proof, and finally opens the commitment views selected by the final challenge. Algorithm 1 summarizes this flow while omitting detailed hash lengths and parameter passing so that the position of ZK.OWFProve remains clear. The algorithm shows AES.ExtendWitness for standard FAEST; FAEST-EM uses Rijndael-EM.ExtendWitness in the same position. If the grinding check or BAVC.Open fails, the signer increments the counter and derives a new $chall3$. The QuickSilver proof is fixed before the signer searches for a final challenge that yields a valid opening and completes the signature.

Algorithm 2 Computational flow of FAEST.Verify, adapted from [BBB⁺25]

Require: message msg , public key pk , signature σ **Ensure:** **true** or **false**

```

1: parse  $\sigma$  as  $(\{c_i\}, \tilde{u}, d, \tilde{a}_1, \tilde{a}_2, decom_I, chall3, iv_{pre}, ctr)$ 
2:  $\mu \leftarrow H_0^2(pk \parallel msg); iv \leftarrow H_4(iv_{pre})$ 
3: if the grind bits of  $chall3$  are nonzero then
4:   return false
5: end if
6:  $(com, Q) \leftarrow \text{FAEST.VOLEReconstruct}(decom_I, chall3 \text{ without its grind bits}, \{c_i\}, iv)$ 
7: if reconstruction returns  $\perp$  then
8:   return false
9: end if
10:  $chall1 \leftarrow H_1^2(\mu \parallel com \parallel \{c_i\} \parallel iv_{pre})$ 
11:  $\tilde{Q} \leftarrow \text{VOLEHash}(chall1, Q)$ 
12: parse  $chall3$  as  $(\delta_0, \dots, \delta_{\lambda-1})$ 
13:  $D \leftarrow \tilde{Q} \oplus [\delta_0 \cdot \tilde{u} \cdots \delta_{\lambda-1} \cdot \tilde{u}]$ 
14:  $chall2 \leftarrow H_2^2(chall1 \parallel \tilde{u} \parallel D \parallel d)$ 
15:  $\tilde{a}_0 \leftarrow \text{ZK.OWFVerify}(d, Q[0..\ell + 2\lambda], chall2, chall3, \tilde{a}_1, \tilde{a}_2, pk)$ 
16:  $chall3' \leftarrow H_3^2(chall2 \parallel \tilde{a}_0 \parallel \tilde{a}_1 \parallel \tilde{a}_2 \parallel ctr)$ 
17: return  $chall3 = chall3'$ 

```

FAEST.Verify reconstructs, from the verifier's view, the VOLE values and $\tilde{a}_0$ omitted from the signature. In Algorithm 2, the verifier first reconstructs Q from the partial decommitment and derives $chall1$ and $chall2$ from the same hash inputs used by the signer. ZK.OWFVerify then reconstructs $\tilde{a}_0$, and the verifier checks whether the final challenge derived from the three QuickSilver proof elements matches $chall3$ in the signature. Verification succeeds only when the signer and verifier evaluate the same QuickSilver constraints consistently from their respective VOLE views.

In both algorithms, QuickSilver lies between VOLE preparation and the final commitment opening. The standard and EM variants use different witness-extension procedures and one-way-function constraints, but the outer flow containing ZK.OWFProve and ZK.OWFVerify remains the same. This common structure lets us analyze the field-arithmetic path shared by the two one-way-function families.

2.3 QuickSilver

QuickSilver uses a VOLE correlation to check whether a committed witness satisfies a given set of constraints [YSWW21]. The signer knows a bit $u_i \in \text{GF}(2)$ and a field element $v_i \in \text{GF}(2^\lambda)$, while the verifier knows $q_i \in \text{GF}(2^\lambda)$ and the global key $\Delta \in \text{GF}(2^\lambda)$. For every i , these values satisfy

$$q_i = u_i \cdot \Delta + v_i.$$

QuickSilver uses this property to evaluate linear computations without revealing the witness. The random value v_i hides the signer's bit, while the single global key Δ links the signer and verifier views. Because the relation is linear, committed values and constants can be added without further interaction.

Unlike linear operations, multiplication increases the degree of a commitment. A degree- d commitment to x can be viewed as a polynomial $\rho_x(X) = \rho_0 + \dots + xX^d$, with x as the coefficient of X^d . The signer holds the coefficients of this polynomial, while the verifier holds its evaluation at $X = \Delta$. Multiplying two committed values requires multiplying their commitment polynomials, which introduces $\text{GF}(2^\lambda)$ multiplications for

both the signer and the verifier. QuickSilver’s nonlinear computation leads directly to finite-field multiplication.

FAEST uses QuickSilver to check that the extended witness correctly evaluates the AES or Rijndael one-way function. The witness-extension procedure records the secret key and intermediate cipher values in w , and `ZK.OWFConstraints` constructs the constraints for the key schedule and encryption. FAEST expresses the inversion-based S-box using constraints of maximum degree 3. Linear transformations do not increase the commitment degree, but combining the S-box constraints requires repeated commitment multiplications. These constraints are the direct source of the repeated $\text{GF}(2^\lambda)$ multiplications in the QuickSilver OWF computation [BBB⁺25].

`OWFProve` and `OWFVerify` evaluate the same constraints in different forms. `OWFProve` evaluates the constraints using the witness and the signer’s VOLE values, then applies `ZKHash` to the coefficient vectors (a_0, a_1, a_2) together with their VOLE-derived masks to produce $(\tilde{a}_0, \tilde{a}_1, \tilde{a}_2)$. `OWFVerify` evaluates the same constraints using (Q, d, Δ) and reconstructs $\tilde{a}_0$ from $\tilde{a}_1$ and $\tilde{a}_2$ in the signature. This shared constraint evaluation is the common target of the analysis and implementation that follow.

2.4 Finite-Field Multiplication

FAEST defines $\text{GF}(2^\lambda)$ as binary polynomials modulo an irreducible polynomial [BBB⁺25]. Each bit string maps to polynomial coefficients in little-endian order. The three security parameters use the following irreducible polynomials:

$$\begin{aligned} P_{128}(\alpha) &= \alpha^{128} + \alpha^7 + \alpha^2 + \alpha + 1, \\ P_{192}(\alpha) &= \alpha^{192} + \alpha^7 + \alpha^2 + \alpha + 1, \\ P_{256}(\alpha) &= \alpha^{256} + \alpha^{10} + \alpha^5 + \alpha^2 + 1. \end{aligned}$$

Finite-field multiplication consists of two steps: polynomial multiplication and reduction. First, two binary polynomials of degree at most $\lambda - 1$ are multiplied to produce a polynomial of degree at most $2\lambda - 2$. Because coefficient addition is XOR and multiplication produces no carries in a binary polynomial, this step is called carry-less multiplication. The product is then reduced modulo P_λ to obtain another λ -bit field element. A field multiplication includes both steps.

When a λ -bit field element is represented using 64-bit limbs, elements of 128, 192, and 256 bits contain two, three, and four limbs, respectively. A schoolbook product multiplies every pair of limbs and requires 4, 9, and 16 carry-less 64-by-64-bit limb products at the three field widths.

2.5 GPU Execution Model

A GPU achieves high throughput by executing many threads concurrently. CUDA groups threads into thread blocks and organizes multiple blocks into a grid. NVIDIA GPUs divide the threads of a block into warps of 32 threads and execute them on streaming multiprocessors. If threads in the same warp follow different control-flow paths, the GPU must execute those paths separately, leaving some threads idle. This hierarchy defines the basic execution units for mapping independent computations to a GPU.

A CUDA thread can keep frequently used values in registers. Excessive register demand can cause some values to spill to local memory. CUDA local memory resides in device memory and is more expensive to access than registers. Intermediate state that must survive a kernel boundary must be written to global memory and read back by the next kernel. Kernel organization determines both how much state remains within a kernel and how much data moves between kernels.

PTX and SASS represent GPU instructions at different levels. PTX is a virtual instruction set between a CUDA program and the machine code of the target GPU, whereas SASS consists of the machine instructions executed by a specific GPU. The PTX `clmad` instruction performs carry-less multiply-and-add and is available from compute capability 8.0 [NVI26]. The `clmad.lo.u64` and `clmad.hi.u64` instructions can compute the low and high halves of the carry-less product of two 64-bit operands.

3 Design Overview

This section defines our GPU implementation boundary, the batch execution model, and the byte-for-byte output validation used throughout the evaluation.

Our GPU implementation covers the key-generation core, OWFProve, and OWFVerify. For key generation, our GPU implementation covers one-way-function evaluation and key-pair serialization, while random sampling and rejection sampling remain on the host. Key generation does not use QuickSilver field multiplication, so the analysis and optimization focus on OWFProve and OWFVerify.

Our GPU implementations of OWFProve and OWFVerify preserve the inputs and outputs of the reference implementation’s internal dispatchers `aes_prove` and `aes_verify`, respectively. All remaining stages, including the GGM tree, PRG, VOLE commitment, SHAKE, and commitment opening, remain unchanged and run on the host through the reference implementation. The reported performance covers only these two GPU paths within the broader signing and verification workflows.

Evaluating OWFProve and OWFVerify separately lets us measure the cost of field arithmetic and run many independent calls in parallel on the GPU. The reference implementation exposes OWFProve and OWFVerify through independent functions, allowing GPU replacements without changing the rest of the protocol. Each OWFProve or OWFVerify call uses its own state. We group N_{batch} independent calls into one GPU batch.

We evaluate a GPU build only after it produces the same outputs as the CPU reference. Each test case supplies identical inputs to the reference and GPU implementations and compares their outputs byte for byte. The comparison covers the key-generation core, OWFProve, and OWFVerify across all twelve FAEST parameter sets. The `s` and `f` variants at the same security level and in the same mode share a compiled GPU kernel, but their outputs are checked separately. These byte-for-byte matches confirm the functional correctness of our GPU implementations within the evaluated scope and provide the basis for the performance comparisons in Section 6.

Making either procedure faster reduces the time spent in that part of signing or verification, while the other stages remain unchanged. Let f be the fraction of total execution time spent in the selected procedure and s its speedup. If execution time outside the procedure remains unchanged, Amdahl’s law [Amd67] gives the following overall speedup:

$$S_{\text{total}}(f, s) = \frac{1}{(1 - f) + f/s}.$$

The overall speedup cannot exceed $1/(1 - f)$ even if that part becomes arbitrarily fast. This bound assumes that the procedure does not overlap with other stages and that GPU data transfer and reintegration into the host pipeline add no overhead. Under these optimistic assumptions, the bound helps interpret procedure-level performance within the complete signing and verification workflows.

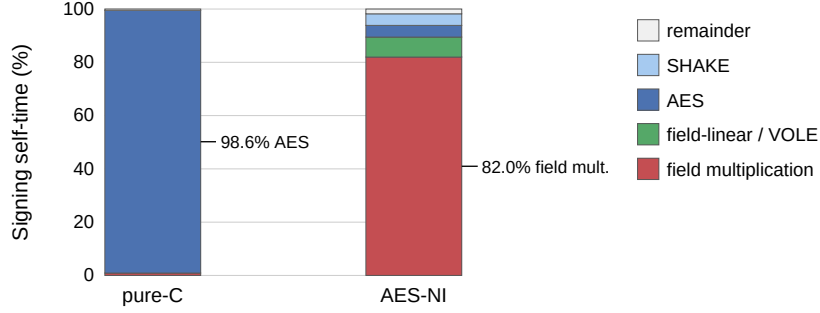


Figure 1: Signing-time breakdown of the pure-C and AES-NI builds for FAEST-128f.

4 The Bottleneck

This section identifies the operation that dominates FAEST signing after AES is accelerated in hardware. We first compare two builds of the reference implementation [FAE26] to find the next bottleneck after AES acceleration. The pure-C build disables AES instruction-set extensions, while the AES-NI build enables them. We then compare the AES-NI build with the `faest-avx` build, an optimized AVX2 implementation of FAEST [FAE25], to determine why the same bottleneck no longer dominates. These results define the main GPU design variable and the predictions evaluated in Section 6.

4.1 The Bottleneck after AES Acceleration

The pure-C and AES-NI reference builds expose different bottlenecks in FAEST signing. Lee et al. [LKS⁺26] identify repeated AES-CTR-based PRG calls in the VOLE-in-the-Head phase as a major bottleneck in software FAEST implementations. Our pure-C profile shows the same pattern, with AES dominating execution time. In the AES-NI build for standard FAEST-128f, field multiplication takes about 82% of total signing time, whereas the symmetric stage combining AES-CTR and SHAKE represents about 8.9%. Hardware AES exposes field multiplication, previously hidden by symmetric computation, as the next bottleneck. Figure 1 shows this shift for FAEST-128f.

With AES acceleration enabled, field multiplication becomes the largest component for every parameter set, as Figure 2 shows. During signing, most field multiplications occur within OWFProve. QuickSilver operations within OWFProve consist largely of field multiplication, while some VOLE and leaf processing outside it uses the same operation. Figure 2 also separates the shares of field multiplication inside and outside OWFProve for each parameter set.

Field multiplication is intrinsic to the algorithm because QuickSilver represents the AES S-box as multiplication constraints over $\text{GF}(2^\lambda)$. However, its actual cost depends strongly on the instructions and multiplication algorithm available on the target hardware. The central implementation question is not whether field multiplication occurs, but how efficiently the target platform can execute it.

4.2 Native Carry-Less Arithmetic

Figure 3 shows that field multiplication no longer dominates total signing time in the optimized `faest-avx` build. For 128f, OWFProve accounts for only 13.4% of execution time, while most of the remaining time is spent on BAVC leaf hashing and VOLE-correlation generation outside OWFProve. These stages are dominated by AES-based processing and SHAKE, with BAVC leaf hashing emerging as the largest cost, consistent with the

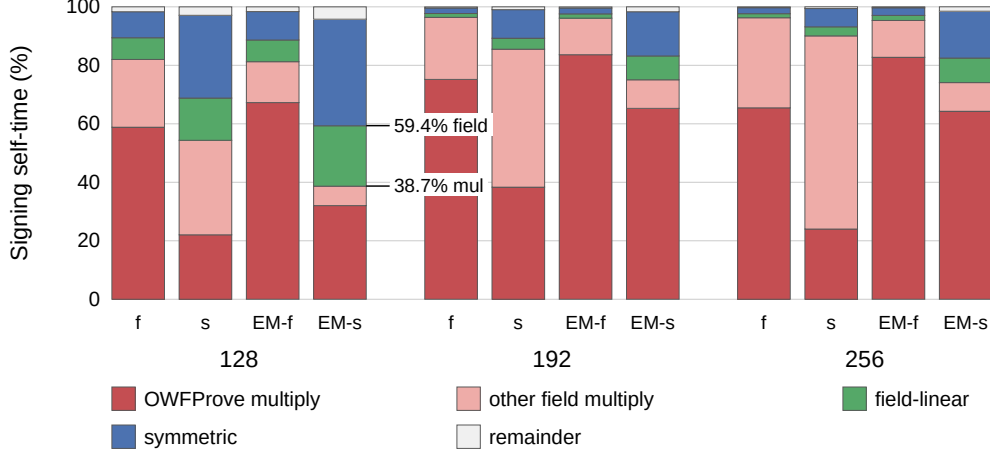


Figure 2: Signing-time breakdown of the AES-NI build across all twelve FAEST parameter sets.

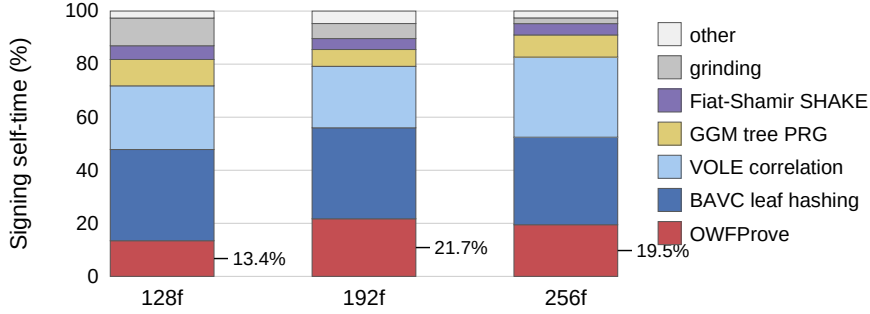


Figure 3: Signing-time breakdown of the optimized `faest-avx` build for the standard 128f, 192f, and 256f parameter sets.

algorithm structure described in Section 2. The bottleneck in the `faest-avx` build shifts from field multiplication to BAVC and VOLE processing outside OWFProve.

The `faest-avx` build profile in Figure 3 shows a shift from field multiplication to BAVC and VOLE processing. The AES-NI build implements field multiplication with software shifts and XORs, and its executable contains no carry-less multiplication instruction. In contrast, the `faest-avx` build uses `VPCLMULQDQ`, a vector carry-less multiplication instruction, for field multiplication. Static disassembly shows that about 74% of its `VPCLMULQDQ` instructions are located in OWFProve and OWFVerify, where they accelerate the field multiplication that accounts for most of the execution time in the AES-NI build. `VPCLMULQDQ` is one of several optimizations in this build, so the comparison characterizes the stack as a whole.

4.3 GPU Design Questions

The CPU analysis shows that native carry-less arithmetic can change both the cost of field multiplication and the overall bottleneck. Whether native carry-less arithmetic produces a similar bottleneck shift on a GPU must be measured separately. GPUs cannot use x86 `VPCLMULQDQ` directly, but the PTX ISA provides the `clmad` instruction for carry-less multiplication on NVIDIA GPUs with compute capability 8.0 or higher. Section 5 implements software multiplication and PTX `clmad` as separate builds and treats native carry-less arithmetic as the main GPU design variable.

This analysis leads to three questions for the GPU evaluation in Section 6.

- **Effect of `clmad` across Field Widths.** As the field grows, each field multiplication requires more component multiplications. Field reduction and the remaining operations do not grow at the same rate. We expect the share of cost removed by native `clmad` to increase with the field width. Section 6 measures how its effect changes at 128, 192, and 256 bits.
- **Differences across One-Way Functions and Procedure Paths.** Standard FAEST and EM use different one-way functions, but OWFProve and OWFVerify share the same QuickSilver field-arithmetic structure in both constructions. We expect field-multiplication optimizations to benefit both constructions. OWFVerify spends a smaller share of its time on field multiplication than OWFProve, so such optimizations may have a smaller effect on the complete OWFVerify path. Section 6 tests whether their effects are limited to a single one-way function or to OWFProve alone.
- **The Bottleneck after Field Multiplication.** Once field multiplication becomes faster, other operations and data preparation account for a larger share of execution time. Further reductions in field multiplication may produce a smaller gain in complete-path throughput. Section 6 measures both the speedup of field arithmetic and where execution time remains after the optimization.

Together, these questions organize the evaluation across field widths, one-way functions, OWFProve, and OWFVerify, and test whether the bottleneck shift found on the CPU also appears on the GPU.

5 Implementation and Optimization

This section presents our GPU implementation of OWFProve and OWFVerify and the field-multiplication optimizations motivated by Section 4. We proceed in three steps: i) mapping batched calls to GPU threads, ii) integrating native PTX `clmad`, and iii) applying Karatsuba to reduce the number of limb products per field multiplication. The discussion focuses on the operations removed and the costs introduced by each design.

5.1 Kernel Organization

Most kernels assign one independent call to one CUDA thread. The host places a batch of N_{batch} input sets in device memory and launches enough threads to process them. Thread i uses only the i -th input set and its assigned scratch buffer, then writes `a0`, `a1`, and `a2` to a separate region of the output buffer. CUDA schedules these threads in 32-thread warps, but the implementation uses no warp-level cooperation. This mapping provides batch parallelism without sharing state between calls. The standard 256-bit input-assembly kernel is the only exception, using one 256-thread block for each input set.

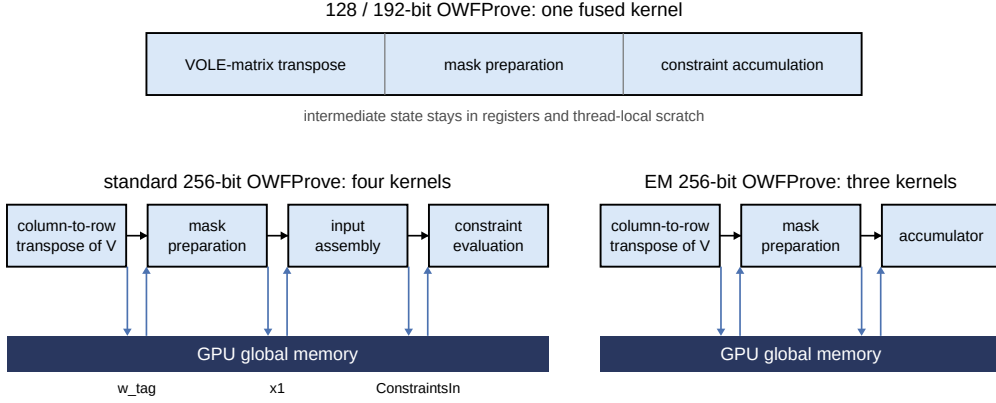


Figure 4: Kernel organization of the fused 128/192-bit and split 256-bit OWFProve implementations.

Figure 4 shows how kernel organization differs by field width and between OWFProve and OWFVerify. The 128- and 192-bit OWFProve implementations and all OWFVerify implementations use a fused kernel. Running the VOLE-matrix transpose, mask preparation, and constraint accumulation in one kernel keeps intermediate state in the registers and scratch space of one thread. This avoids writing the state to global memory and reading it back across kernel boundaries. We retain a fused kernel wherever it produces correct output.

The standard 256-bit OWFProve implementation does not use a fused kernel. The initial monolithic kernel compiled and ran without reporting an error, but only one of the three QuickSilver output elements matched the reference output. The same mismatch appeared with NVCC 11.5 and 13.3. The transpose and mask outputs matched the reference, while the mismatch appeared during constraint evaluation. The large constraint function also showed high register pressure and register spilling. The monolithic `aes_256_prover` in the CPU reference produced build-dependent output under GCC `-O2`. Validation compares the GPU output with the stable output produced by composing the stage-level reference functions.

Therefore, we split standard 256-bit OWFProve into four kernels: transpose, mask preparation, input assembly, and constraint evaluation. The constraint kernel reads all inputs from a single `ConstraintsIn` base, and this organization restores byte-for-byte agreement with the reference output. The split adds global-memory traffic for intermediate values such as `w_tag`, `x1`, and `ConstraintsIn`.

The EM-256 OWFProve implementation uses a different three-kernel organization. It still uses four-limb arithmetic, but its input structure differs from that of the standard 256-bit OWFProve implementation. Its 32-byte one-way-function output and round keys pass directly to an accumulator kernel, eliminating the separate input-assembly kernel. The resulting path contains a transpose kernel, a mask kernel, and an accumulator kernel. This organization limits register pressure in the large accumulator while avoiding an unnecessary fourth kernel boundary.

All final kernel organizations match the reference output byte for byte. Section 6 follows these kernel boundaries. For standard 256-bit OWFProve, we report both the constraint kernel and the complete four-stage path. For EM-256 OWFProve, we report both the accumulator kernel and the complete three-stage path. The primary measurements for 128- and 192-bit OWFProve and all OWFVerify paths cover the complete fused kernel. A

Table 1: Limb-product and `clmad` instruction counts by field width. Entries of the form $a \rightarrow b$ give the schoolbook and Karatsuba counts. Each 64-by-64-bit limb product uses `clmad.lo` and `clmad.hi`.

Field	64-bit limbs	Limb products	Product-stage <code>clmad</code>
$\text{GF}(2^{128})$	2	$4 \rightarrow 3$	$8 \rightarrow 6$
$\text{GF}(2^{192})$	3	$9 \rightarrow 6$	$18 \rightarrow 12$
$\text{GF}(2^{256})$	4	$16 \rightarrow 9$	$32 \rightarrow 18$

separate lean kernel is used only for the auxiliary standard 256-bit OWFVerify analysis.

5.2 Native Carry-Less Multiplication

The software and `clmad` builds differ only in their implementation of 64-by-64-bit carry-less multiplication. The software build composes each product from shifts and XORs. The `clmad` build computes the same product with `clmad.lo.u64` and `clmad.hi.u64` in CUDA C++ inline PTX [NVI26]. Both builds retain the same kernel organization and all other operations, so their comparison isolates the effect of native carry-less multiplication. The GGM tree and PRG remain in the host reference path defined in Section 3.

At the source level, the GPU `clmad` multiplication path is a straight-line sequence independent of the input values and removes the software routine’s data-dependent branch. This source-level property is limited to the arithmetic path. The key-generation one-way function retains the reference table-based AES implementation, and a constant-time characterization of the complete implementation requires a separate evaluation.

5.3 Karatsuba Multiplication

Prior work on GHASH optimization treats the product and reduction stages as separate optimization targets [GK14]. Karatsuba reduces the number of limb products required for each field product, while aggregated reduction performs one reduction for several products. We first apply Karatsuba to the PTX `clmad` path.

We call this configuration the `clmad`+Karatsuba build. Field elements of 128, 192, and 256 bits are represented by two, three, and four 64-bit limbs, respectively. The baseline schoolbook multiplication multiplies every pair of limbs, requiring 4, 9, and 16 limb products at the three field widths. Karatsuba reduces these product counts.

For example, consider two field elements, each represented by two limbs, $A = a_0 + a_1X^{64}$ and $B = b_0 + b_1X^{64}$. Schoolbook multiplication uses four limb products $(a_0b_0, a_0b_1, a_1b_0, a_1b_1)$. Karatsuba instead computes

$$z_0 = a_0b_0, \quad z_2 = a_1b_1, \quad z_1 = (a_0 \oplus a_1)(b_0 \oplus b_1) \oplus z_0 \oplus z_2,$$

and constructs the result from three products [KO62]. This reduces the number of limb multiplications from four to three, while introducing additional XOR operations.

The `clmad`+Karatsuba build replaces only the product stage of the `clmad` build, leaving the reduction and all call sites unchanged. The register count remained 255 per thread at all three field widths, and SASS disassembly confirmed the intended reduction in `clmad` instructions. In $\text{GF}(2^{128})$, for example, Karatsuba reduces the limb-product count from four to three. This reduces the product-stage `clmad` count from eight to six, while the five `clmad` instructions used for reduction remain. The total count per field multiplication falls from 13 to 11. Table 1 summarizes the limb-product and product-stage `clmad` counts at all three field widths.

Karatsuba reduces the `clmad` count for field multiplication at all three field widths, with the largest reduction at 256 bits, where field multiplication is also the most time-consuming.

5.4 Applicability of the Design

Another approach, aggregated reduction, reduces the number of reductions in GHASH [JL11, GK14]. We exclude it from our final design because of its additional storage cost. It precomputes powers of a fixed multiplier and delays reduction, combining several products before performing one reduction per window. We implemented window sizes of 2, 4, and 8.

In our implementation, aggregated reduction did not improve throughput over per-product reduction across 128, 192, and 256 bits because its additional storage cost outweighed the savings from fewer reductions. Our final design retains per-product reduction at all three widths, using `clmad` at 128 bits and the faster shift-and-XOR reduction at 192 and 256 bits.

Within our 64-bit-limb model, the minimum counts are 3, 6, and 9 limb products for two-, three-, and four-limb inputs [BCS97, FP19, Kam85]. Our Karatsuba product stages meet all three bounds, while the complete field-multiplication routine and surrounding GPU implementation may still offer further optimization opportunities.

The final arithmetic design combines PTX `clmad` with Karatsuba and excludes aggregated reduction. PTX `clmad` lowers the cost of each limb multiplication, while Karatsuba reduces the number of limb multiplications. Section 6 evaluates how these choices affect batched GPU throughput and identifies the next bottleneck in the complete 256-bit OWFProve path.

6 Evaluation

6.1 Experimental Setup

We conduct the experiments on an NVIDIA RTX A5000 with 24 GB of GDDR6 memory and 768 GB/s of memory bandwidth, using Ubuntu 22.04.5 LTS and CUDA 13.3. The CPU analysis in Section 4 uses an Intel Xeon w3-2435. The three GPU builds share the same source code and kernel organization and differ only in field multiplication.

Each batch contains N_{batch} independent calls. Inputs are placed in device memory before timing, and only the kernels in each measured path are timed. Because the `s` and `f` variants share a GPU kernel for a given one-way function and field width, we evaluate twelve distinct OWFProve and OWFVerify paths.

We measure batch sizes of 256, 1,024, and 4,096 and use $N_{\text{batch}} = 4,096$ as the common comparison point. Each configuration is repeated ten times in a single measurement session with the display server stopped, and we report the sample mean; at $N_{\text{batch}} = 4,096$, the maximum coefficient of variation is 1.04%.

6.2 OWF Path Throughput

The `clmad` build outperforms the software build across all twelve OWF paths shown in Table 2.

For standard OWFProve, the resulting speedups over the software build are $2.0\times$, $2.5\times$, and $2.8\times$ at 128, 192, and 256 bits, respectively. Wider fields require more limb products per field multiplication, while the remaining operations do not grow at the same rate. This result supports the prediction from Section 4 that native carry-less arithmetic becomes more important as the field grows.

The relative effect of `clmad` is smaller for OWFVerify than for OWFProve at the same one-way function and field width. OWFVerify uses the same field layer but evaluates each constraint only once, so field multiplication accounts for a smaller share of its work.

Table 2: Throughput across all twelve OWF paths at $N_{\text{batch}} = 4,096$, in calls per second.

	Path	Software	clmad (vs. software)	clmad+Karatsuba (vs. software)
OWFProve	Standard 128-bit	24,860.04	49,380.91 (2.0 ×)	50,057.64 (2.0 ×)
	Standard 192-bit	4,560.35	11,194.19 (2.5 ×)	11,404.75 (2.5 ×)
	Standard 256-bit	1,798.43	5,066.23 (2.8 ×)	5,200.05 (2.9 ×)
	EM 128-bit	27,960.64	65,851.64 (2.4 ×)	67,020.15 (2.4 ×)
	EM 192-bit	6,164.40	14,897.54 (2.4 ×)	15,187.87 (2.5 ×)
	EM 256-bit	1,920.65	5,752.33 (3.0 ×)	5,911.86 (3.1 ×)
OWFVerify	Standard 128-bit	32,918.55	47,840.74 (1.5 ×)	48,306.04 (1.5 ×)
	Standard 192-bit	6,837.71	11,609.91 (1.7 ×)	11,701.97 (1.7 ×)
	Standard 256-bit	2,721.70	5,228.01 (1.9 ×)	5,300.56 (1.9 ×)
	EM 128-bit	39,844.06	64,831.03 (1.6 ×)	65,539.86 (1.6 ×)
	EM 192-bit	9,234.45	15,322.84 (1.7 ×)	15,460.17 (1.7 ×)
	EM 256-bit	3,021.42	5,946.00 (2.0 ×)	6,046.95 (2.0 ×)

Karatsuba provides an additional procedure-level improvement only for standard and EM 256-bit OWFProve.

Figure 5 shows that standard OWFProve throughput increases with the batch size for every build and field width. The measured ordering of the three builds remains the same at all three batch sizes in every panel. Thus, using $N_{\text{batch}} = 4,096$ as the common comparison point preserves the qualitative comparison seen at smaller batch sizes.

6.3 Field Arithmetic and Complete-Path Throughput

For standard 256-bit OWFProve, the substantial speedup in the constraint kernel does not fully carry over to the complete path. Table 3 shows that stages outside arithmetic limit the effect of **clmad** on complete-path throughput.

Karatsuba shows the same gap between local arithmetic and complete-path throughput. It reduces the limb-product count in the product stage but does not affect the transpose or intermediate-data preparation, which limits its effect on the complete OWFProve path.

Profiling with NVIDIA Nsight Compute (**ncu**) identifies the column-to-row transpose as the next bottleneck after arithmetic is accelerated. As Table 4 shows, its longer duration and memory-stall profile indicate that the final build spends more time waiting for memory while rearranging the VOLE matrix than performing carry-less multiplication.

These results show that optimizing the complete OWFProve path must address both field arithmetic and data layout. **clmad** and Karatsuba reduce the constraint-kernel cost and expose the transpose as the dominant part of this path. Improving complete-path throughput further requires redesigning the transpose and intermediate layout.

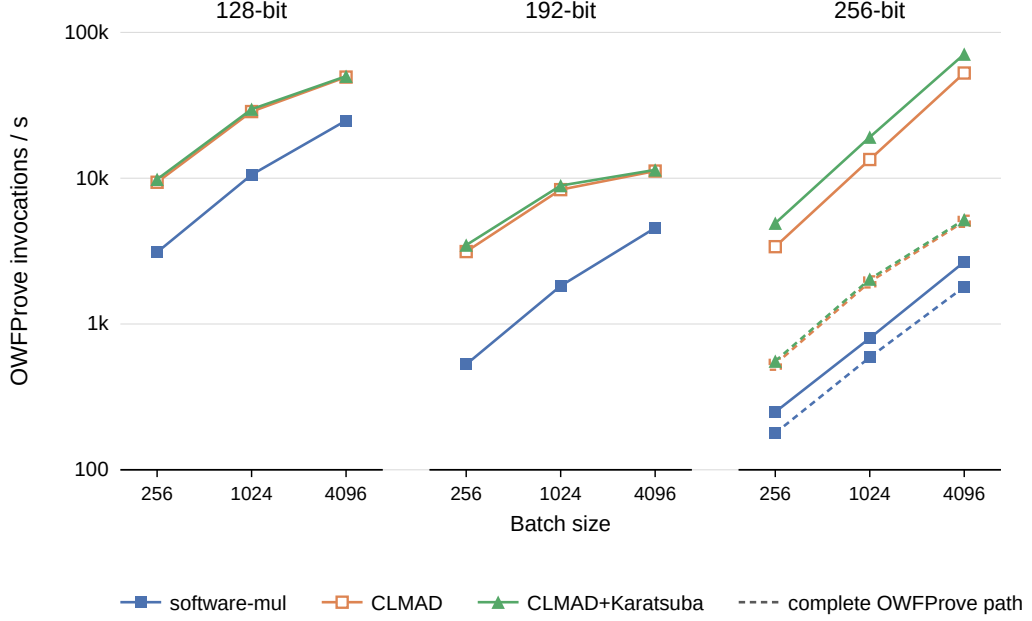


Figure 5: Throughput of the three GPU builds for standard 128f, 192f, and 256f OWFProve across batch sizes. The 256f panel shows both the constraint kernel and the complete OWFProve path for each build.

Table 3: Throughput of standard 256-bit OWFProve at $N_{\text{batch}} = 4,096$, in calls per second.

		c1mad	c1mad+Karatsuba
256-bit path	Software	(vs. software)	(vs. software)
Constraint kernel	2,664.66	52,639.86 (19.8×)	70,948.27 (26.6×)
Complete OWFProve path	1,798.43	5,066.23 (2.8×)	5,200.05 (2.9×)

6.4 Key Findings

Together, these results answer the three questions raised in Section 4. First, the advantage of `c1mad` grows with the field width: standard OWFProve runs 2.0×, 2.5×, and 2.8× faster than the software build at 128, 192, and 256 bits. Second, the improvement holds across all twelve standard and EM OWFProve and OWFVerify paths, with a smaller relative effect on OWFVerify. Third, accelerating field multiplication shifts the bottleneck to data preparation. For standard 256-bit OWFProve, the `c1mad+Karatsuba` build delivers 26.6× the throughput of the software-multiplication build in the constraint kernel and 2.9× in the complete path, while the transpose becomes the dominant component.

7 Conclusion and Future Work

This work shows that the performance bottleneck in FAEST’s QuickSilver OWF computation on a GPU depends on how field multiplication is implemented. The `c1mad` build improves throughput across all twelve OWF paths, covering the standard and EM constructions, all three field widths, and both OWFProve and OWFVerify. For standard OWFProve,

Table 4: Nsight Compute profile of the standard 256-bit `clmad`+Karatsuba OWFProve path at $N_{\text{batch}} = 4,096$.

Kernel	Duration per batch	L2 hit rate	Long-scoreboard stall
Column-to-row transpose	734.22 ms	1.55%	94.21%
Mask preparation	0.30 ms	70.03%	84.29%
Input assembly	1.25 ms	55.96%	96.91%
Constraint evaluation	60.28 ms	73.07%	43.08%

the speedups over the software build are $2.0\times$, $2.5\times$, and $2.8\times$ at 128, 192, and 256 bits. Only standard and EM 256-bit OWFProve gain an additional procedure-level improvement from Karatsuba, reaching $2.9\times$ and $3.1\times$ the software throughput, respectively.

In the standard 256-bit OWFProve path, accelerating field arithmetic exposes the column-to-row transpose as the next bottleneck. These results show that native carry-less arithmetic, multiplication density, and data organization must be considered together. Future work should also redesign the transpose and intermediate layout, integrate the GPU implementations of OWFProve and OWFVerify into complete signing and verification, and measure end-to-end performance. Evaluation on other GPU architectures and CUDA toolchains will show how broadly these design principles apply. The source-level properties discussed in Section 5 apply only to the arithmetic path, and the complete implementation requires a separate side-channel evaluation.

References

- [ABC⁺26] Gorjan Alagic, Maxime Bros, Pierre Ciadoux, Quynh Dang, Thinh Hung Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Hamilton Silberg, Daniel Smith-Tone, and Noah Waller. Status report on the second round of the additional digital signature schemes for the NIST post-quantum cryptography standardization process. NIST Internal Report NIST IR 8610, 2026.
- [Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Joint Computing Conference*, 1967.
- [BBB⁺25] Carsten Baum, Ward Beullens, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Kloof, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. FAEST v2: Algorithm specifications (version 2.0). NIST Post-Quantum Cryptography: Additional Digital Signature Schemes, Round 2, 2025.
- [BBdSG⁺23] Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Kloof, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-head. In *Advances in Cryptology – CRYPTO 2023*, pages 581–615, 2023. IACR ePrint 2023/996.
- [BCS97] Peter Bürgisser, Michael Clausen, and Mohammad Amin Shokrollahi. *Algebraic Complexity Theory*. Grundlehren der mathematischen Wissenschaften. Springer, 1997.

- [DHSY24] Sanjay Deshpande, James Howe, Jakub Szefer, and Dongze Yue. SDitH in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):215–251, 2024.
- [FAE25] FAEST team. faest-avx: Optimized AVX2 implementation of FAEST. <https://github.com/faest-sign/faest-avx>, 2025. Commit 09b2ffb, accessed 2026-07-07.
- [FAE26] FAEST team. faest-ref: FAEST reference implementation. <https://github.com/faest-sign/faest-ref>, 2026. Commit 77caab6, accessed 2026-07-07.
- [FP19] Magnus Gausdal Find and René Peralta. Better circuits for binary polynomial multiplication. *IEEE Trans. Computers*, 68(4):624–630, 2019.
- [GK14] Shay Gueron and Michael E. Kounavis. Intel carry-less multiplication instruction and its usage for computing the GCM mode. Intel White Paper 323640-002, Revision 2.02, 2014.
- [Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing (STOC)*, pages 212–219, 1996.
- [JL11] Krzysztof Jankowski and Pierre Laurent. Packed AES-GCM algorithm suitable for AES/PCLMULQDQ instructions. *IEEE Trans. Computers*, 60(1):135–138, 2011.
- [Kam85] Michael Kaminski. A lower bound for polynomial multiplication. *Theoretical Computer Science*, 40(2–3):319–322, 1985.
- [KO62] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962. In Russian. English translation: Soviet Physics Doklady 7 (1963), 595–596.
- [LJWJ24] Guoxiao Liu, Keting Jia, Puwen Wei, and Lei Ju. High-performance hardware implementation of MPCitH and Picnic3. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):190–214, 2024.
- [LKS⁺26] Seung-Won Lee, Ha-Gyeong Kim, Min-Ho Song, Si-Woo Eum, and Hwa-Jeong Seo. Accelerating FAEST signatures on ARM: NEON SIMD AES and parallel VOLE optimization. *Applied Sciences*, 16(8):3782, 2026. Also IACR ePrint 2026/499.
- [LPACGSR26] Samuel Lemes-Perera, Miguel R. Alarcon, Pino Caballero-Gil, and Miquel Serra-Ricart. A high-throughput AES-GCM implementation on GPUs for secure, policy-based access to massive astronomical catalogs. arXiv:2602.23067, 2026.
- [LZS⁺24] Wai-Kong Lee, Raymond K. Zhao, Ron Steinfeld, Amin Sakzad, and Seong Oun Hwang. High throughput lattice-based signatures on GPUs: Comparing Falcon and Mitaka. *IEEE Trans. Parallel Distributed Syst.*, 35(4):675–692, 2024. IACR ePrint 2023/399.
- [Med24] Miguel Medina Anton. Hardware implementation of the FAEST PQC digital signature scheme. MS thesis, George Mason University, 2024. Handle 1920/14292, DOI 10.13021/MARS/14571.

- [Nat24a] National Institute of Standards and Technology. Module-lattice-based digital signature standard. FIPS PUB 204, 2024.
- [Nat24b] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. FIPS PUB 203, 2024.
- [Nat24c] National Institute of Standards and Technology. Stateless hash-based digital signature standard. FIPS PUB 205, 2024.
- [NVI26] NVIDIA Corporation. Parallel thread execution ISA version 9.3. <https://docs.nvidia.com/cuda/parallel-thread-execution/>, 2026. Accessed 2026-07-07.
- [SF11] Georg Schönberger and Jürgen Fuß. GPU-assisted AES encryption using GCM. In *Communications and Multimedia Security (CMS)*, pages 178–185, 2011.
- [Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 124–134, 1994.
- [SYD⁺24] Shiyu Shen, Hao Yang, Wangchen Dai, Hong Zhang, Zhe Liu, and Yunlei Zhao. High-throughput GPU implementation of Dilithium post-quantum digital signature. *IEEE Trans. Parallel Distributed Syst.*, 35(11):1964–1976, 2024. IACR ePrint 2024/1365.
- [YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2986–3001, 2021. IACR ePrint 2021/076.