

High-Performance Implementations of HAETAE and SMAUG-T on Arm Cortex-M55 Using MVE and Slothy

Seung-Won Lee¹[0009–0009–9583–1387], Min-Joo Sim¹[0000–0001–5242–214X], and
Hwa-Jeong Seo¹[0000–0003–0069–9061]

Department of Information Computer Engineering,
Hansung University, Seoul, 02876 Korea
dkajdfhd1@gmail.com, minjoos9797@gmail.com, hwajeong84@gmail.com

Abstract. Efficient execution of Post-Quantum Cryptography (PQC) in resource-constrained embedded environments requires the joint optimization of not only algorithm vectorization but also dataflow and instruction execution order. This paper implements the HAETAE-2/3/5 and SMAUG-T1/3/5 parameter sets of two Korean Post-Quantum Cryptography Competition (KpqC) winners using the M-profile Vector Extension (MVE) of the Arm Cortex-M55 and analyzes the contribution of each optimization stage to performance improvement. We first establish an MVE-intrinsic implementation as the baseline and then present Manually Optimized MVE Assembly that redesigns the data layout, register reuse, and memory access patterns of the principal kernels. Slothy is subsequently applied to this assembly to optimize instruction order and register allocation for the Cortex-M55 pipeline. This process is applied to the Keccak/SHAKE operations shared by the two schemes; the Number-Theoretic Transform (NTT), fixed-point Fast Fourier Transform (FFT), and Gaussian sampling of HAETAE; and the Toom–Cook/Karatsuba polynomial multiplication of SMAUG-T. Measurements on the STM32N657 show that the implementation with Slothy achieves speedups of $2.349\text{--}2.424\times$ for the HAETAE forward NTT, $1.701\text{--}1.732\times$ for the FFT including squared-value accumulation, and $1.322\text{--}1.350\times$ for the SMAUG-T matrix–vector multiplication over the intrinsic implementation. At the complete-scheme level, HAETAE and SMAUG-T achieve speedups of $1.050\text{--}1.212\times$ and $1.108\text{--}1.212\times$, respectively. These results demonstrate that direct dataflow redesign and automated instruction scheduling eliminate different bottlenecks and that combining the two stages can effectively improve the performance of Cortex-M55-based PQC implementations.

Keywords: Post-Quantum Cryptography · HAETAE · SMAUG-T · Cortex-M55 · M-profile Vector Extension · Instruction Scheduling · Slothy

1 Introduction

As the standardization of Post-Quantum Cryptography (PQC) progresses, the efficient implementation of standardized algorithms in resource-constrained embedded environments has become increasingly important. Accordingly, the focus of PQC research has expanded beyond algorithm design to platform-specific implementations and automated code optimization. This paper presents methods for efficiently implementing and optimizing two KpqC winners from the Korean Post-Quantum Cryptography Competition (KpqC) on a Cortex-M55-based embedded platform: HAETAE, a digital signature algorithm [9], and SMAUG-T, a key encapsulation mechanism [8]. Although the two schemes share Keccak/SHAKE, HAETAE primarily uses the Number-Theoretic Transform (NTT), a fixed-point Fast Fourier Transform (FFT), and sampling, whereas SMAUG-T primarily uses Toom–Cook/Karatsuba polynomial multiplication. This enables a joint analysis of M-profile Vector Extension (MVE) optimizations for shared cryptographic operations and distinct arithmetic structures.

The Arm Cortex-M55 supports MVE, also known as Helium, a 128-bit Single Instruction, Multiple Data (SIMD) instruction set. MVE intrinsics enable the use of vector instructions in C code, but actual instruction selection, register allocation, and execution order are determined by the compiler. Given the Cortex-M55’s limited vector execution resources and in-order pipeline, the resulting code can contain unnecessary instructions and data dependencies that cause pipeline stalls [5]. Vectorization using intrinsics alone therefore cannot fully exploit the performance of the Cortex-M55; optimization must consider both instruction count and execution order.

Previous studies have optimized HAETAE and SMAUG-T for other Arm profiles [20,13,14] or analyzed individual kernels, such as the NTT and polynomial multiplication, on the Cortex-M55 [1]. However, to the best of our knowledge, no prior study has evaluated HAETAE-2/3/5 and SMAUG-T1/3/5 on Cortex-M55/MVE while separately quantifying the effects of dataflow redesign and automated instruction scheduling.

The main contributions of this paper are as follows.

1. We present the first implementation and evaluation of HAETAE-2/3/5 and SMAUG-T1/3/5 from the final KpqC specifications on Cortex-M55/Helium.
2. We present Manually Optimized MVE Assembly that restructures data layout, register reuse, operation fusion, and memory access in the principal HAETAE and SMAUG-T kernels.
3. We apply Slothy-based instruction scheduling to the Manually Optimized MVE Assembly to optimize instruction order and register allocation. Across complete operations, the Slothy implementation achieves speedups of 1.006–1.131 $\times$ over the Assembly implementation and 1.050–1.212 $\times$ over the Intrinsic implementation.

2 Background and Related Work

This section summarizes the background necessary to understand the implementation and performance analysis presented in subsequent sections. It first describes the MVE programming model and execution constraints of the Cortex-M55 and introduces the computational structures of HAETAE, SMAUG-T, and the Keccak/SHAKE operations shared by the two schemes. It then explains how Slothy reorders instructions in a given assembly program and summarizes the findings and unresolved limitations of related work.

2.1 Cortex-M55 and MVE

MVE is a 128-bit SIMD vector extension introduced in Armv8.1-M [5]. A single Q register can be used as either four 32-bit lanes or eight 16-bit lanes; consequently, the number of elements processed by one instruction depends on the element bit width. When the number of elements to be processed is not a multiple of the lane count, per-lane predication can restrict processing in the final iteration to valid elements. MVE instructions can be expressed using either C intrinsics or assembly [3]. The Cortex-M55 does not process a 128-bit MVE operation all at once; instead, it processes the operation in two 64-bit stages. To do so, it internally divides a 128-bit vector into four 32-bit processing units and processes two units per cycle. Arm refers to each 32-bit processing unit as a beat and to the simultaneous processing of two beats as dual-beat execution [6].

Because a single vector operation is processed in two stages, the placement of subsequent instructions affects execution efficiency. An instruction that immediately consumes the result of a preceding operation must wait until that result becomes available. By contrast, placing independent instructions between the two operations allows other computations to use this otherwise idle interval. Optimizing execution order with consideration of inter-instruction data dependencies is therefore important on the Cortex-M55.

In addition, the Cortex-M55 provides eight Q registers [4]. Unrolling loops or executing multiple operations concurrently increases the number of intermediate values that must be retained, and values exceeding the available register capacity must be temporarily stored in memory and subsequently reloaded. These additional memory accesses can reduce the performance gains obtained from increased instruction-level parallelism. Cortex-M55 optimization must therefore consider register usage as well as the number of independent operations.

2.2 Target Algorithms

Both HAETAE and SMAUG-T repeatedly perform polynomial matrix-vector operations of degree $n = 256$ and invoke Keccak-based functions, but their principal arithmetic structures differ. Tables 1 and 2 summarize the parameters required for the subsequent implementation and performance analysis.

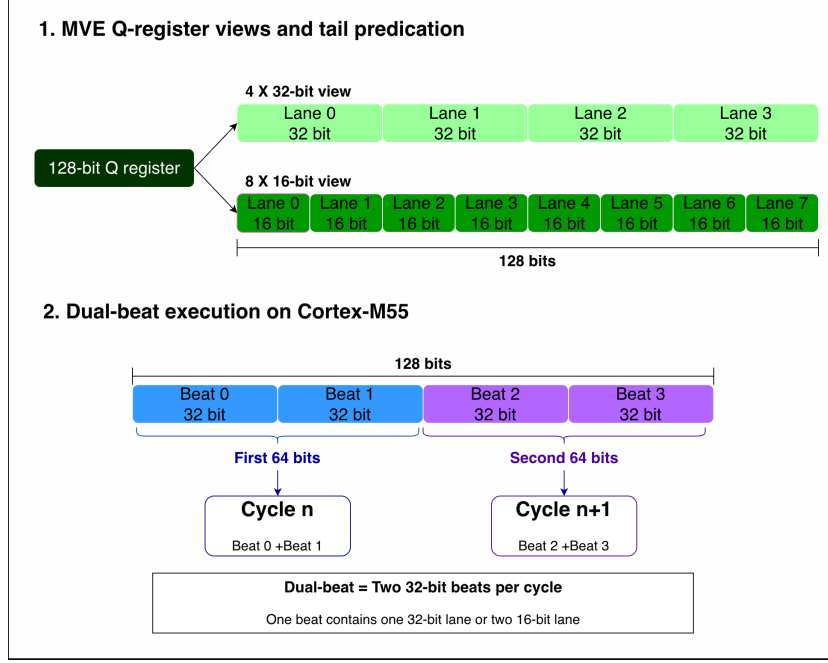


Fig. 1. MVE Q-register organization and dual-beat execution on the Cortex-M55. A 128-bit Q register can be interpreted as four 32-bit lanes or eight 16-bit lanes. The Cortex-M55 divides a 128-bit vector into four 32-bit beats and processes two beats per cycle.

Computational Structure of HAETAE HAETAE is a digital signature scheme in the Fiat–Shamir-with-aborts family based on the hardness of the module variants of Learning With Errors (LWE) and Short Integer Solution (SIS) [9,16,17]. HAETAE operates over the polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. All three parameter sets use $n = 256$ and the prime $q = 64,513$. Because $q \equiv 1 \pmod{512}$, negacyclic polynomial multiplication can be performed using a 256-point NTT.

In the polynomial matrix–vector multiplications used for key generation, signing, and verification, the input polynomials are transformed into the NTT domain, coefficients at corresponding positions are multiplied pointwise, and the products corresponding to each output polynomial are accumulated. When a result in coefficient representation is required, the inverse NTT is applied.

In addition to this modular polynomial arithmetic, HAETAE uses fixed-point operations. Secret-key rejection during key generation applies a 256-point complex FFT to each secret polynomial and accumulates the squared magnitudes at each frequency. The hyperball sampler used in signing scales the samples using the sum of squares of a Gaussian sample vector and its inverse square root, and regenerates candidates that do not satisfy the acceptance range [9]. In Table 1, (k, ℓ) denotes the dimensions of the matrix and vectors. As these values increase, both the number of

Table 1. HAETAE parameters used in this work, taken from the 2026 KpqC final specification. Key and signature sizes are in bytes.

Parameter	HAETAE-2	HAETAE-3	HAETAE-5
NIST security level	2	3	5
Target security (bits)	120	180	260
Ring dimension n	256	256	256
Module dimensions (k, ℓ)	(2, 4)	(3, 6)	(4, 7)
Modulus q	64,513	64,513	64,513
Secret coefficient bound η	1	1	1
Challenge weight τ	58	80	128
Truncated vk bits d	1	1	0
Expected repetitions M	6.0	5.0	6.0
Hint compression factor α_h	512	512	256
Verification key	992	1,472	2,080
Secret key	1,408	2,112	2,752
Signature	1,474	2,349	2,948

invocations of the same kernel and the number of terms in the pointwise accumulation increase. The parameter τ denotes the number of nonzero coefficients in the challenge polynomial.

In Table 1, n and q define the polynomial ring, η bounds the secret-key coefficients, and (k, ℓ) gives the module dimensions of the matrix and vectors. Increasing (k, ℓ) increases both the number of kernel invocations and the number of terms in pointwise accumulation. The challenge polynomial has τ nonzero coefficients. The parameters d and α_h determine verification-key truncation and hint compression, respectively, whereas M is the expected number of signing repetitions and therefore affects end-to-end signing time. The final three rows report the serialized verification-key, secret-key, and signature sizes.

Computational Structure of SMAUG-T SMAUG-T is a key encapsulation mechanism (KEM) that uses the module variants of LWE and Learning With Rounding (LWR)[2]. It repeatedly performs polynomial matrix–vector multiplications during key generation, encapsulation, and decapsulation [8]. In Table 2, k is the module rank, so increasing k increases both the number of matrix elements and the reuse count of each vector operand. The moduli q , p , p' , and t define the public-key, ciphertext, compressed-ciphertext, and plaintext coefficient domains. The parameter h_s is the fixed Hamming weight used to sample the secret, r controls sparse centered-binomial sampling of the ephemeral secret, and σ is the standard deviation of the discrete-Gaussian error. The last four rows report the decryption-failure probability and serialized object sizes. The reference polynomial multiplication in SMAUG-T uses 4-way Toom–Cook[22,10]. It divides each of the two 256-coefficient inputs into four 64-

Table 2. SMAUG-T1, SMAUG-T3, and SMAUG-T5 parameters used in this work, taken from the 2026 KpqC final specification. DFP is shown as $\log_2(\text{DFP})$, and key and ciphertext sizes are in bytes.

Parameter	SMAUG-T1	SMAUG-T3	SMAUG-T5
NIST security level	1	3	5
Ring dimension n	256	256	256
Module rank k	2	3	4
Public-key modulus q	1,024	2,048	2,048
Ciphertext modulus p	256	512	512
Compression modulus p'	32	16	128
Plaintext modulus t	2	2	2
Secret Hamming weight h_s	140	264	348
spCBD parameter r	1/8	1/4	3/16
Error standard deviation σ	1.0625	1.0625	1.0625
$\log_2(\text{DFP})$	-118.3	-179.2	-194.2
Public key	672	1,088	1,440
Secret key	832	1,312	1,728
Ciphertext	672	992	1,376

coefficient parts, performs seven 64×64 multiplications at the evaluation points, and then performs interpolation.

For the product $\mathbf{y} = A\mathbf{s}$ of a polynomial matrix A and a vector $\mathbf{s}$, each output row is computed as $y_i = \sum_j A_{i,j}s_j$. Because the same vector operand s_j is reused across multiple output rows, processing each $A_{i,j}s_j$ as an independent polynomial multiplication also repeats the Toom evaluation of s_j . This reuse structure motivates considering the entire matrix-vector operation, rather than only independent polynomial multiplications, as the unit of optimization.

Shared Keccak/SHAKE Processing Path HAETAE and SMAUG-T repeatedly invoke SHAKE or SHA-3 for matrix expansion[11], sample generation, challenge generation, and key derivation. The underlying Keccak- $f[1600]$ permutation updates a 1600-bit state consisting of 25 64-bit words over 24 rounds [7]. Each round comprises the Theta, Rho, Pi, Chi, and Iota steps[12]. In the Keccak specification, each 64-bit state word is referred to as a lane, which is distinct from an MVE SIMD lane. On a 32-bit Cortex-M processor, each Keccak lane must be processed as two 32-bit words. Bit interleaving, as used in single-stream scalar Keccak, separates the bits at even and odd positions in each 64-bit lane into two words, thereby expressing 64-bit rotations and logical operations in terms of 32-bit instructions.

2.3 Slothy-based Instruction Scheduling

Slothy is a constraint-solving-based tool that takes the assembly code of a target function and a processor model as input and searches for

an instruction order and register allocation that preserve program semantics [21]. It first constructs data dependencies from the input and output operands of each instruction and adds constraints on memory-access ordering and available registers. It then uses the instruction latencies, throughput, and execution resources defined in the target processor model to search for a schedule with a low estimated execution cost.

Slothy can reduce pipeline stalls by overlapping mutually independent load, arithmetic, and store instructions or by software-pipelining multiple loop iterations. However, it does not design algorithms, data layouts, or operation orders that are not provided in the input.

2.4 Related Work

Becker et al. analyzed low-precision polynomial multiplication using MVE on the Cortex-M55 and integrated Toom-Cook/Karatsuba and NTT implementations into Saber [6]. Their study demonstrated the importance of register management and instruction scheduling given the limited number of Q registers. However, the target scheme, modulus, and overall call structure differ from those of HAETAE and SMAUG-T.

Abdulrahman et al. presented a method that uses Slothy to automatically schedule instructions and allocate registers for NTT and fixed-point and floating-point FFT assembly on the Cortex-M55 and Cortex-M85 [1]. Their study models the Cortex-M55’s dual-beat execution, separate execution resources, and store-load hazards, but focuses on scheduling an already selected primitive rather than redesigning its dataflow.

Cheon et al. optimized a complete implementation of HAETAE, including its NTT and Gaussian sampler, on the Cortex-M4F [9]. Although their study identified the embedded bottlenecks of HAETAE, Digital Signal Processing (DSP)-oriented scalar execution on the Cortex-M4 and 128-bit MVE execution on the Cortex-M55 differ in register organization and execution model. Thus, their results characterize HAETAE under scalar DSP-oriented execution rather than the MVE execution model considered here.

Sim et al. optimized the HAETAE NTT, inverse NTT, and pointwise Montgomery multiplication on the AArch64/NEON architecture of the Apple M1 Pro [20]. Although both platforms use 128-bit SIMD, the A-profile provides more vector registers and has different instruction-issue and memory architectures. Moreover, their scope is limited to HAETAE polynomial arithmetic and does not include the fixed-point FFT or Gaussian CDT.

Narlı et al. implemented SMAUG-T1/3/5 on the Cortex-M4 and evaluated their feasibility on lightweight devices [18]. Although this work is a directly relevant embedded implementation of the same scheme, its Cortex-M4 platform does not support MVE and therefore cannot evaluate Cortex-M55 vector dataflow.

Previous studies have also presented streaming, variable-live-range separation, and recomputation techniques to reduce the stack usage of HAETAE and SMAUG-T [15,19]. These studies target low-memory implementations. The present study primarily evaluates execution performance and

reports stack usage as a secondary metric accompanying performance optimization.

In summary, previous studies have either optimized individual polynomial primitives on the Cortex-M55 or implemented HAETAE and SMAUG-T on other Arm platforms. However, within the scope of the literature surveyed, no study has implemented HAETAE-2/3/5 and SMAUG-T1/3/5 on the Cortex-M55 and separately evaluated the effects of dataflow restructuring and instruction scheduling under identical conditions. The following section describes the resulting Cortex-M55 implementation.

3 Implementation

Fig. 2 shows the workflow in which MVE intrinsics, Manually Optimized MVE Assembly, and Slothy scheduling are applied sequentially.

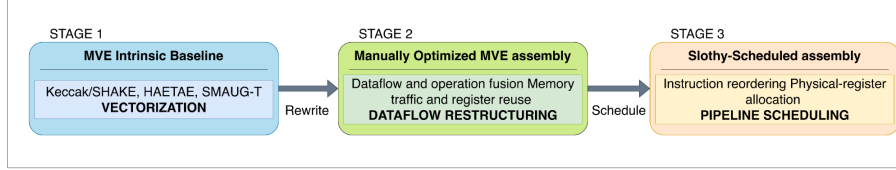


Fig. 2. Three-stage implementation workflow. The intrinsic baseline establishes the vectorized implementation, manually optimized assembly restructures dataflow and memory access, and Slothy changes instruction order and physical-register allocation.

3.1 MVE Intrinsic Implementation

Keccak and SHAKE The Keccak/SHAKE implementation comprises a single-stream path that processes one SHAKE invocation and an x4 parallel path that uses MVE to process four independent SHAKE invocations concurrently. The x4 parallel path does not divide one input into four parts; instead, it performs SHAKE computations for four distinct inputs simultaneously. The single-stream path uses one Keccak state and retains the bit-interleaved representation of 32-bit scalar Keccak from absorption through squeezing.

Independent SHAKE invocations with identical input and output lengths are repeated during polynomial sampling in HAETAE and matrix generation in SMAUG-T. When four such invocations can be grouped, the x4 parallel path is used. As shown in Fig. 3, values at the same position in the four Keccak states are placed in the four 32-bit MVE lanes, one per lane. A single vector instruction can then apply the same permutation step to all four states concurrently. If fewer than four independent invocations are available for concurrent processing, the single-stream path is used.

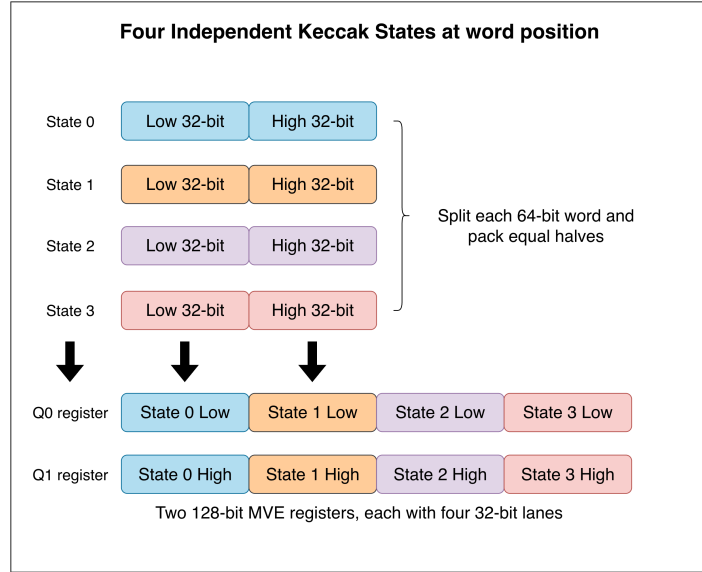


Fig. 3. Four-stream Keccak state layout. Each MVE lane holds the corresponding 32-bit half from one independent state.

With this layout, the logical operations of Theta and Chi are applied simultaneously to the corresponding columns and rows of all four states. The NOT and AND operations of Chi are combined into an MVE AND-NOT operation. Rho constructs a 64-bit rotation by moving bits between two 32-bit vectors, whereas Pi changes the storage locations of the rotated values. Because the Iota round constant is also replicated across the four lanes, the same x4 layout can be maintained throughout all stages of the permutation.

HAETAE Because an MVE vector contains four 32-bit values, the HAETAE NTT, polynomial operations, and fixed-point FFT can process four values together with a single instruction. For the NTT and polynomial operations, four consecutive integer coefficients are placed in one vector. For the FFT, the complex-valued input, in which real and imaginary parts are stored alternately, is separated into real and imaginary vectors.

The butterfly, the basic NTT computation, combines two coefficients using constant multiplication, addition, and subtraction. This implementation loads the four left-hand coefficients and four right-hand coefficients of four butterflies and computes the four butterflies concurrently. Multiplying the required coefficients by an NTT constant (root) and reducing the results to the prescribed residue range are implemented by combining a conventional multiplication with an operation that rounds the upper half of the multiplication result. The same approach processes four pairs

of coefficients at corresponding positions concurrently when multiplying or accumulating their products.

The NTT comprises eight stages. Processing each stage separately would reload and store all 256 coefficients at every stage. The intrinsic implementation instead groups the stages as $3 + 3 + 2$, reducing the number of complete polynomial loads and stores from eight to three. A three-stage group requires eight live coefficient vectors, exactly filling the Cortex-M55's eight Q registers, whereas four stages would require 16.

The real and imaginary parts of the FFT input are each represented as 32-bit integers scaled by 2^{16} . In the input array, the real and imaginary parts of the first complex value and those of the second complex value are stored alternately in the order $\text{Re}[0], \text{Im}[0], \text{Re}[1], \text{Im}[1], \dots$. In this layout, the real and imaginary parts must be separated for every computation. To address this issue, the input array is read once and di-

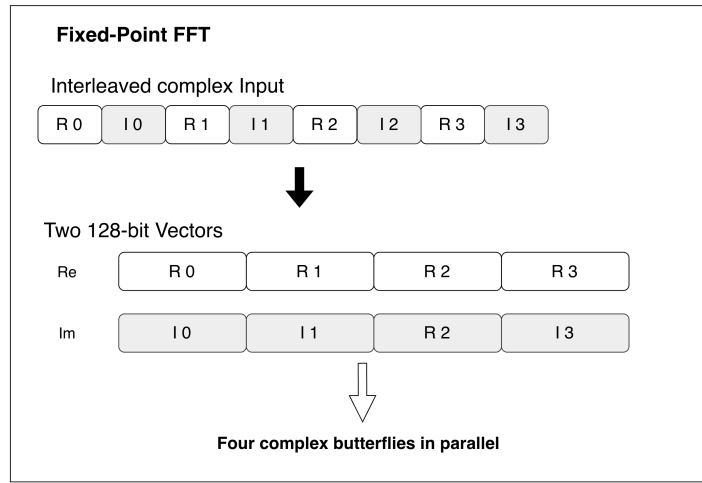


Fig. 4. Fixed-point FFT data layout for MVE processing.

vided into one array containing all real parts and another containing all imaginary parts. All subsequent FFT stages use these two arrays. The first two FFT stages, which require no twiddle-factor multiplication, are fused into a single operation that processes four inputs concurrently. For the squared-norm path, the final-stage output is passed directly to squared-magnitude accumulation without converting the separated real and imaginary arrays back to the interleaved layout.

The Gaussian CDT of the hyperball sampler compares one 16-bit random value against 64 thresholds. By replicating the same random value across eight 16-bit lanes and loading and comparing eight thresholds at a time, 64 scalar comparisons are replaced with eight vector comparisons. Each comparison result is converted to 0 or 1 and accumulated on a per-lane basis.

SMAUG-T To parallelize this multiplication using MVE, the implementation decomposes the large multiplication into subproblems of sizes suited to vector operations and selects 16- or 32-bit operations according to the range of intermediate values at each step.

Following the 4-way Toom–Cook decomposition described in Section 2.2, each 64×64 point multiplication is further decomposed into three 32×32 polynomial products using Karatsuba.

Toom–Cook evaluation and the Karatsuba submultiplications are implemented using 16-bit MVE operations. The intermediate values remain within the 16-bit range when generating the operands for the seven multiplications and performing the multiplications decomposed by Karatsuba. Eight consecutive coefficients are therefore placed in one MVE vector, and their additions, subtractions, and constant multiplications are performed concurrently.

Toom–Cook interpolation is implemented using 32-bit MVE operations. This process uses division implemented through a combination of constant multiplication and bit shifts. Because computing the intermediate values in 16-bit precision would not preserve the required values exactly, the coefficients are widened to 32 bits and placed four at a time in each vector. After all results have been combined, they are converted back to the 16-bit format. Thus, the smaller multiplications process eight 16-bit coefficients at a time, whereas result combination processes four 32-bit coefficients at a time.

The intrinsic implementation retains repeated operand loads across the lower-level multiplications and uses a temporary product array before accumulation. The following subsection directly targets these two costs in assembly.

3.2 Manually Optimized MVE Assembly

The MVE assembly implementation retains the same interfaces and data representations while directly specifying the lifetimes of intermediate values across multiple operations and the order of memory accesses.

Table 3 summarizes the MVE instructions referenced in the following implementation descriptions and listings [5]. The suffix indicates the element width and signedness, and the T suffix denotes execution under the current vector predicate.

Keccak and SHAKE To reduce memory accesses, the assembly implementation retains the stage intermediates in the eight Q registers. During Theta, only the column-wise XOR results are retained temporarily. During Rho and Pi, the current 64-bit state value occupies two vector registers while the remaining 24 positions are updated sequentially. During Chi, the five state values in one row are loaded only once to compute all five results.

The offsets of the other three buffers relative to the address of the first buffer are stored in one vector. The `vldrw.u32` gather instruction in Listing 1.1 uses this address information to load one value from each of four noncontiguous input buffers into a single vector. Conversely, the

Table 3. MVE instructions used in the implementation and code listings.

Instructions	Operation	Use in this work
<code>vmov, vmov.i32</code>	Moves a scalar to one lane or broadcasts an immediate across the vector.	Constructs the SHAKE buffer-offset vector and initializes vector registers.
<code>vldrw.u32, vstrw.32</code>	Loads or stores four 32-bit words; the vector-offset forms gather from or scatter to non-contiguous addresses.	Accesses four SHAKE buffers in parallel and transfers HAETAE state or NTT coefficients.
<code>vldrh.u16, vstrh.16</code>	Loads or stores eight 16-bit halfwords.	Transfers SMAUG-T Karatsuba operands, partial products, and combined results.
<code>veor.u32</code>	Computes a lane-wise bitwise exclusive OR.	Absorbs four independent SHAKE inputs into four Keccak states.
<code>vadd, vsub</code>	Performs lane-wise addition or subtraction.	Implements NTT butterflies and combines Karatsuba partial products.
<code>vmul, vmla</code>	Performs lane-wise multiplication or multiply-accumulate.	Implements NTT modular arithmetic and SMAUG-T polynomial multiplication.
<code>vqrdmulh.s32</code>	Returns the signed, saturating, rounded high half of a doubling multiplication.	Estimates the quotient used by the HAETAE modular reduction.
<code>vctp.32</code>	Marks the lanes corresponding to the remaining 32-bit elements.	Creates the predicate for the final three SMAUG-T coefficients.
<code>vpst</code>	Applies the current vector predicate to the following predicated instruction.	Guards the tail load and store at the polynomial boundary.
<code>vldrht.u32, vstrht.32</code>	Loads active 16-bit values into 32-bit lanes or stores only active lanes under the current predicate.	Prevents the fourth lane from accessing memory beyond the 127-coefficient result array.

`vstrw.32` scatter instruction stores one of the four vector values to each output buffer.

In Listing 1.1, Lines 1–10 compute the three buffer offsets relative to the first buffer, initialize `q4` to zero, and place the offsets in its remaining three 32-bit lanes; `r7` addresses the upper 32-bit word of the first buffer. Lines 11–19 implement absorption. The two `vldrw.u32` gather instructions collect the lower and upper words of four independent inputs into `q0` and `q1`, Lines 14–15 load the corresponding state vectors, and Lines 16–19 XOR and store the updated state. Lines 20–24 implement squeezing by loading the two state vectors and scattering their four lanes directly to the four output buffers. This removes the lane-by-lane insertion and temporary output array required by the intrinsic implementation.

HAETAE The HAETAE assembly implementation retains the operations and data layouts of the intrinsic implementation while changing the

```

1  /* q4 = {0, buffer1-buffer0, buffer2-buffer0,
2     buffer3-buffer0} */
3  sub    r2, r2, r1
4  sub    r3, r3, r1
5  sub    r4, r4, r1
6  vmov.i32 q4, #0
7  vmov    s17, r2
8  vmov    s18, r3
9  vmov    s19, r4
10 add    r7, r1, #4
11 /* Absorb four independent inputs */
12 vldrw.u32 q0, [r1, q4]
13 vldrw.u32 q1, [r7, q4]
14 vldrw.u32 q2, [r0]
15 vldrw.u32 q3, [r0, #400]
16 veor.u32 q0, q0, q2
17 veor.u32 q1, q1, q3
18 vstrw.32 q0, [r0]
19 vstrw.32 q1, [r0, #400]
20 /* Squeeze four outputs using the same address offsets */
21 vldrw.u32 q0, [r0]
22 vldrw.u32 q1, [r0, #400]
23 vstrw.32 q0, [r1, q4]
24 vstrw.32 q1, [r7, q4]

```

Listing 1.1. MVE gather and scatter for four parallel SHAKE streams

NTT stage grouping and the iteration order of the FFT and Gaussian CDT.

Because the intrinsic three-stage grouping already occupies all eight Q registers with coefficient vectors, it leaves no vector register for the temporary values required by the hand-written butterfly sequence. The assembly implementation therefore groups two stages at a time, retaining four coefficient vectors and the arithmetic intermediates in registers. Listing 1.2 shows how four coefficient vectors are loaded once and the results of the first stage are used immediately in the second stage without being stored to memory.

In Listing 1.2, Lines 1–7 define the `CT` macro, which computes four 32-bit Cooley–Tukey butterflies concurrently. Lines 2–4 multiply the four right-hand coefficients by the root and reduce the products: `vqrdmulh.s32` forms the quotient estimate using a precomputed twisted root, and `vmla.s32` applies the correction with $r12 = -q$. Lines 5–6 then form the difference and sum outputs. Lines 8–11 load four coefficient vectors once. Lines 12–17 apply two first-stage macro calls and feed their register-resident outputs directly to two second-stage calls, without an intermediate store and reload. Finally, Lines 18–21 store only the outputs of the second stage. The roots and reduction constants remain in general-purpose registers and are reused across these butterflies.

```

1 .macro CT left, right, root, twisted
2 vmul.s32    q4, \right, \root
3 vqrdmulh.s32 \right, \right, \twisted
4 vmla.s32    q4, \right, r12
5 vsub.u32    \right, \left, q4
6 vadd.u32    \left, \left, q4
7 .endm
8 vldrw.u32 q0, [r0]
9 vldrw.u32 q1, [r1]
10 vldrw.u32 q2, [r2]
11 vldrw.u32 q3, [r3]
12 /* First stage */
13 CT q0, q2, r4, r5
14 CT q1, q3, r4, r5
15 /* Second stage: reuse the first-stage outputs */
16 CT q0, q1, r6, r7
17 CT q2, q3, r8, r9
18 vstrw.32 q0, [r0], #16
19 vstrw.32 q1, [r1], #16
20 vstrw.32 q2, [r2], #16
21 vstrw.32 q3, [r3], #16

```

Listing 1.2. Two consecutive NTT stages in MVE assembly

In the forward NTT, the stages in which the coefficients combined by a butterfly are separated by 128 and 64 positions are grouped first, followed by the stage pairs with separations of (32, 16), (8, 4), and (2, 1). The inverse NTT applies these groups in the reverse order. In the final (2, 1) group, instructions that automatically rearrange coefficients during loads and stores place the coefficients to be processed together in corresponding lanes of each vector.

The assembly-specific change is the traversal order of the twiddle factors. Whereas the intrinsic implementation reconstructs the real- and imaginary-part twiddle vectors for each coefficient group, the assembly implementation processes groups sharing the same twiddle factor consecutively. Each pair of twiddle vectors can therefore be loaded once and reused across multiple coefficient groups.

The intrinsic implementation processes the eight threshold groups in a loop, whereas the assembly implementation fully unrolls these computations. After accumulating the comparison results, the assembly implementation produces one sample through a horizontal addition across the eight vector lanes. This removes the instructions for loop control and threshold-address calculation.

SMAUG-T The SMAUG-T assembly implementation preserves the decomposition and vector formats of the intrinsic implementation while reducing data reloads and stores in the 32-coefficient Karatsuba multiplications.

```

1 /* Pointer setup and loop control are omitted for clarity. */
2 /* Add z0 and z2 directly to the low and high results */
3 vldrh.u16 q0, [r0], #16      /* z0 */
4 vldrh.u16 q1, [r1], #16      /* z2 */
5 vldrh.u16 q2, [r2]           /* accumulated low */
6 vldrh.u16 q3, [r3]           /* accumulated high */
7 vadd.i16 q0, q0, q2
8 vadd.i16 q1, q1, q3
9 vstrh.16 q0, [r2], #16
10 vstrh.16 q1, [r3], #16
11 /* Add the middle term z1-z0-z2 directly */
12 vldrh.u16 q0, [r0], #16      /* z0 */
13 vldrh.u16 q1, [r1], #16      /* z1 */
14 vldrh.u16 q2, [r2], #16      /* z2 */
15 vldrh.u16 q3, [r3]           /* accumulated middle */
16 vsub.i16 q1, q1, q0
17 vsub.i16 q1, q1, q2
18 vadd.i16 q3, q3, q1
19 vstrh.16 q3, [r3], #16

```

Listing 1.3. MVE accumulation of the three Karatsuba terms into the result array

The first difference is how the coefficients used in multiplication are retained in registers. The intrinsic implementation reloads the coefficient vectors of the right-hand polynomial from memory whenever the left-hand coefficient changes. The assembly implementation loads the 32 coefficients of the right-hand polynomial only once into four Q vector registers. It then executes the `vmla.i16` vector multiply-accumulate instruction while changing the left-hand coefficient one at a time. The 32 right-hand coefficients can therefore be reused throughout all 32 iterations.

The second difference is how the results of the smaller multiplications are added. In matrix-vector multiplication, the results of multiple polynomial multiplications must be continuously accumulated into the same array. The intrinsic implementation stores the 127 coefficients produced by multiplying two 64-coefficient polynomials in a temporary array and then reloads this array to add it to the existing result. By contrast, the assembly implementation updates the existing result array directly while combining the three parts z_0 , z_1 , and z_2 produced by Karatsuba. Listing 1.3 shows how the low-degree part z_0 , high-degree part z_2 , and middle part $z_1 - z_0 - z_2$ are added directly to the existing values at their respective positions.

In Listing 1.3, Line 1 states that address initialization and loop control are omitted from the condensed excerpt. Lines 2–10 load eight 16-bit coefficients from each of z_0 and z_2 together with the existing low- and high-degree results, add the vectors, and store the updated results directly. Lines 11–19 load the corresponding vectors from z_0 , z_1 , and z_2 , form the Karatsuba middle term $z_1 - z_0 - z_2$, add it to the existing middle-degree result, and store it in place. Thus, each block processes

eight coefficients concurrently and avoids materializing and reloading a separate 127-coefficient product array. The vector layouts used to generate the inputs to the seven multiplications and combine their results are identical to those of the intrinsic implementation.

3.3 Slothy-based Instruction Scheduling

Slothy is applied to the manually optimized MVE assembly while preserving its data layout, extent of operation fusion, and memory dependencies; only instruction order and physical-register allocation are optimized under the Cortex-M55 model. Fig. 5 illustrates the resulting scheduling principle.

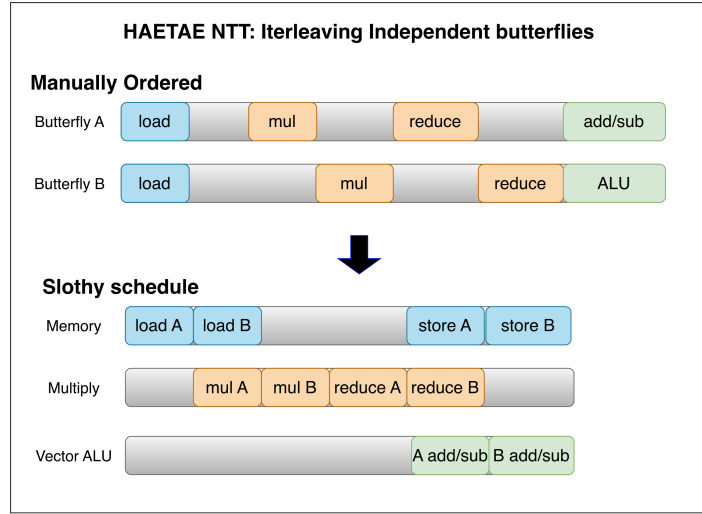


Fig. 5. Representative scheduling patterns used in the Slothy stage.

Keccak and SHAKE In Keccak, operations within a 64-bit state update form a dependency chain, whereas updates to different columns and rows are independent. Slothy interleaves these independent updates and retains the upper and lower 32-bit halves required by the next Theta, Rho/Pi, Chi, and Iota stages in registers.

For SHAKE, Slothy similarly alternates memory accesses and state operations for different positions. As shown in Fig. 5(b), two consecutive 64-bit values are processed as a group so that a state operation on one value is performed while a memory access for the other is in progress. SHAKE128 processes 21 64-bit values per block, leaving one value after grouping them in pairs. This value is processed separately after the loop.

HAETAE Each NTT butterfly forms a dependency chain comprising root multiplication, reduction, addition, and subtraction. Slothy interleaves the four independent butterflies within a stage pair with loads and stores for subsequent coefficients. The same scheduling structure is applied to the inverse NTT, pointwise operations, and residue reduction, while reused addresses and NTT constants remain in registers.

For the FFT, Slothy interleaves twiddle-factor multiplications with butterflies and memory accesses from independent coefficient groups. It also schedules independent vector operations across input separation, output recombination, and squared-magnitude accumulation.

For the Gaussian CDT, Slothy interleaves the loads and comparisons of the eight independent threshold groups with accumulation of their comparison results.

SMAUG-T During Toom–Cook evaluation and Karatsuba multiplication, Slothy interleaves multiplications and accumulations of independent coefficient vectors. During interpolation, it schedules coefficient loads, additions, and subtractions between constant multiplications.

Multiplying two polynomials of 64 coefficients each produces 127 result coefficients. Because the result-combination process handles four 32-bit coefficients per vector, only three coefficients remain in the final vector. If a conventional vector instruction loads or stores all four lanes, the fourth lane may access memory beyond the array boundary. Tail predication, which processes only valid vector lanes, is used to prevent such access. `vctp.32` marks the lanes corresponding to the three remaining coefficients, and `vpst` applies this predicate to the immediately following vector instruction. `vldrht.u32` loads only the marked 16-bit coefficients into a 32-bit vector and fills the remaining lane with zero, whereas `vstrht.32` stores only the marked lanes to memory.

These instructions are already available in Armv8.1-M MVE, but the existing version of Slothy cannot parse `vpst`, `vldrht`, or `vstrht`. We therefore added the syntax, input and output registers, and memory-access behavior of these three instructions to Slothy’s Armv8.1-M instruction model. `vpst` was configured to initiate every cycle, and `vldrht` and `vstrht` were each configured to initiate at two-cycle intervals. The modeled latency of all three instructions was set to one cycle. This extension allows the final three coefficients to be scheduled together with the preceding vector operations without separate code that processes them individually.

4 Performance Evaluation

This section addresses three questions. First, what performance gain is achieved by redesigning operations expressed using MVE intrinsics as Manually Optimized MVE Assembly? Second, does scheduling the same assembly with Slothy provide an additional gain? Third, how much of the two-stage improvement remains in the top-level operations of the complete schemes? To answer these questions, the Intrinsic, Assembly, and Slothy implementations are compared stage by stage.

4.1 Experimental Setup

The evaluation was conducted on the STM32N657 platform described in Table 4. Interrupts were disabled to prevent other tasks from affecting execution time during measurement, and execution time was measured using the processor’s internal cycle counter, the Data Watchpoint and Trace (DWT) Cycle Count Register (CYCCNT).

Kernel measurements used two reversed-order sets of 100 runs, yielding 200 measurements per implementation. End-to-end measurements used four order-balanced sets of 100 runs, yielding 400 measurements per implementation; pooled medians are reported.

Implementation correctness was verified using a Known-Answer Test (KAT), which compares the output against a known correct answer. In addition, the outputs of the Intrinsic, Assembly, and Slothy implementations were compared for 1,000 inputs from each parameter set, confirming that all three implementations produced identical results.

Table 4. measurement environment.

Item	Configuration
Target	STM32N657, Cortex-M55
Core / HCLK / AXISRAM	600 / 200 / 400 MHz
Memory	Secure AXISRAM
Cache	L1 instruction/data enabled
Compiler	GCC 16.1.0, -O3, hard-float ABI
Timer	DWT CYCCNT

4.2 Kernel-Level Performance

Tables 5–7 report matched kernel-level results for the Intrinsic, Manually Optimized MVE Assembly, and Slothy implementations. Assembly speedup is calculated relative to Intrinsic, additional Slothy speedup relative to Assembly, and the final column directly compares Slothy with Intrinsic.

The implementation with Slothy achieved speedups of $2.349\text{--}2.424\times$ for the forward NTT and $1.730\text{--}1.832\times$ for the inverse NTT over Intrinsic. For the forward NTT, the Assembly stage provided a $1.526\text{--}1.621\times$ speedup, followed by an additional $1.449\text{--}1.588\times$ speedup from Slothy. For the inverse NTT, the speedups from the two stages were $1.079\text{--}1.193\times$ and $1.451\text{--}1.698\times$, respectively. The sizable gains from both stages show that the NTT benefits from both reduced memory traffic and scheduling freedom among independent butterflies.

For the pointwise operations, the implementation with Slothy also achieved speedups of $1.191\text{--}1.325\times$ for multiplication and $1.135\text{--}1.689\times$ for accumulation over Intrinsic. For pointwise multiplication, Assembly provided a $1.069\text{--}1.189\times$ speedup, and Slothy added a further $1.101\text{--}1.114\times$ speedup. For pointwise accumulation, the additional Slothy speedup was $0.996\text{--}1.181\times$; however, the gain from the Assembly stage remained, and

Table 5. Median cycles and speedups for HAETAE kernels. Assembly is the hand-optimized MVE code used as input to Slothy. The speedup columns compare Assembly with Intrinsic, Slothy with Assembly, and Slothy with Intrinsic.

Parameter	Kernel	Median cycles			Speedup		
		Intr.	Asm.	Slothy	Asm/Intr.	Slothy/Asm.	Slothy/Intr.
HAETAE-2	Forward NTT	6,089	3,974	2,590	1.532	1.534	2.351
	Inverse NTT	6,899	6,393	3,765	1.079	1.698	1.832
	Pointwise multiplication	1,973	1,846.5	1,657	1.069	1.114	1.191
	Pointwise accumulation	8,842	6,179	5,234	1.431	1.181	1.689
	Fixed-point FFT	15,528	12,665	12,620	1.226	1.004	1.230
	FFT with squared norm	21,582	13,540	12,472	1.594	1.086	1.730
	CDT leaf	203	202	189	1.005	1.069	1.074
	Full Gaussian sampler	607,026	611,472	608,051	0.993	1.006	0.998
HAETAE-3	Forward NTT	6,227	4,080	2,569	1.526	1.588	2.424
	Inverse NTT	6,365	5,338	3,679	1.192	1.451	1.730
	Pointwise multiplication	2,109	1,868.5	1,697	1.129	1.101	1.243
	Pointwise accumulation	28,489	26,582	25,100	1.072	1.059	1.135
	Fixed-point FFT	15,325	12,425	12,392	1.233	1.003	1.237
	FFT with squared norm	21,166	13,529	12,218	1.564	1.107	1.732
	CDT leaf	203	202	189	1.005	1.069	1.074
	Full Gaussian sampler	608,204	611,937	608,941	0.994	1.005	0.999
HAETAE-5	Forward NTT	6,592	4,067	2,806	1.621	1.449	2.349
	Inverse NTT	6,474	5,427	3,633	1.193	1.494	1.782
	Pointwise multiplication	2,201.5	1,852	1,662	1.189	1.114	1.325
	Pointwise accumulation	36,747	27,727	27,846	1.325	0.996	1.320
	Fixed-point FFT	15,532	12,553	12,429	1.237	1.010	1.250
	FFT with squared norm	21,060	13,505	12,382	1.559	1.091	1.701
	CDT leaf	203	236	189	0.860	1.249	1.074
	Full Gaussian sampler	606,755	611,666	608,175	0.992	1.006	0.998

the final implementation outperformed Intrinsic for every parameter set. Slothy can alternate the multiplications of independent coefficients. By contrast, accumulation sequentially performs coefficient multiplication, residue-range adjustment, and accumulation, leaving fewer operations that can be reordered. This difference accounts for the different additional effects of Slothy on the two pointwise kernels.

With Slothy, the fixed-point FFT achieved a 1.230–1.250 $\times$ speedup over Intrinsic, and the path including the final squared-value accumulation achieved a 1.701–1.732 $\times$ speedup. For the FFT itself, Assembly provided a 1.226–1.237 $\times$ speedup, whereas the additional Slothy speedup was 1.003–1.010 $\times$. The Assembly gain reflects the twiddle-vector reuse described in Section 3.2. When squared-value accumulation was included, the Assembly speedup increased to 1.559–1.594 $\times$, and the additional Slothy speedup increased to 1.086–1.107 $\times$. The larger Slothy gain for the squared-norm path reflects the additional scheduling freedom provided by independent butterfly, squaring, and memory operations.

The Slothy implementation achieved a 1.074 $\times$ speedup over Intrinsic for the CDT leaf, whereas the complete Gaussian sampler remained essentially unchanged at 0.998–0.999 $\times$. Assembly alone achieved 0.860–1.005 $\times$ for the CDT leaf and 0.992–0.994 $\times$ for the complete sampler, indicating that unrolling alone did not reduce the underlying compari-

son work. Slothy improved the CDT leaf by $1.069\text{--}1.249\times$ over Assembly, but the unchanged fixed-point operations outside the CDT and rejection resampling diluted this leaf-level gain in the complete sampler. For Keccak/SHAKE, only the effect of instruction scheduling after the Assembly stage is compared separately; Table 6 therefore presents the Assembly and Slothy results.

Table 6. Keccak/SHAKE cycles before and after Slothy. Each entry is a range of medians across the six HAETA and SMAUG-T binaries. Speedup is calculated as Assembly cycles divided by Slothy cycles.

Kernel	Median cycles		Speedup
	Assembly	Slothy	Slothy/ Asm.
Keccak- $f[1600]$ x4	25,754–25,994	25,754–25,994	1.000–1.000
SHAKE128 x4	26,587–26,826	26,582–26,823	0.998–1.000

Keccak- $f[1600]$ x4 required the same number of cycles in Assembly and Slothy, and the difference for SHAKE128 x4 was within 0.2%. These near-unity ratios indicate that the assembly dataflow had already removed the dominant memory-transfer overhead, leaving little additional benefit from instruction reordering.

Table 7. Median cycles and speedups for SMAUG-T polynomial kernels. The speedup columns compare Assembly with Intrinsic, Slothy with Assembly, and Slothy with Intrinsic.

Parameter	Kernel	Median cycles			Speedup		
		Intrinsic	Assembly	Slothy	Asm. / Intrinsic	Slothy / Asm.	*Slothy / *Intrinsic
SMAUG-T1	Polynomial mul.+acc.	41,314	37,145	34,262.5	1.112	1.084	1.206
	Matrix-vector	160,390	131,807	121,354	1.217	1.086	1.322
SMAUG-T3	Polynomial mul.+acc.	41,046	36,928	34,158	1.112	1.081	1.202
	Matrix-vector	351,577	284,493	261,491	1.236	1.088	1.345
SMAUG-T5	Polynomial mul.+acc.	41,221	36,826	34,096	1.119	1.080	1.209
	Matrix-vector	618,516.5	498,793	458,011	1.240	1.089	1.350

With Slothy, SMAUG-T achieved speedups of $1.202\text{--}1.209\times$ for polynomial multiplication and accumulation and $1.322\text{--}1.350\times$ for matrix-vector multiplication over Intrinsic. For polynomial multiplication and accumulation, Assembly provided a $1.112\text{--}1.119\times$ speedup, and Slothy

added a further $1.080\text{--}1.084\times$ speedup. For matrix–vector multiplication, the corresponding stage-wise speedups were $1.217\text{--}1.240\times$ and $1.086\text{--}1.089\times$. Because the optimized polynomial kernel is invoked repeatedly within matrix–vector multiplication, these gains accumulate and produce a larger improvement than for a single polynomial multiplication.

4.3 End-to-End Performance

Table 8 compares the C Reference and MVE Intrinsic implementations, whereas Table 9 reports KeyGen, Sign/Encaps, and Verify/Decaps results for HAETAE-2/3/5 and SMAUG-T1/3/5 under the Intrinsic, Assembly, and Slothy implementations.

Table 8. End-to-end cycle counts for the C Reference and MVE Intrinsic implementations. Values are medians in thousands of cycles (K).

Parameter	Operation	C Ref. cycles	*MVE Intr. cycles	Speedup
HAETAE-2	KeyGen	5,984.0	2,616.4	2.287
	Sign	34,081.5	20,430.7	1.668
	Verify	1,463.0	761.9	1.920
HAETAE-3	KeyGen	13,424.6	5,973.8	2.247
	Sign	35,507.6	21,204.7	1.675
	Verify	2,772.8	1,439.0	1.927
HAETAE-5	KeyGen	12,535.8	5,693.0	2.202
	Sign	70,524.4	42,381.4	1.664
	Verify	3,584.3	1,842.7	1.945
SMAUG-T1	KeyGen	838.1	546.5	1.533
	Encaps	729.6	463.0	1.576
	Decaps	967.8	620.9	1.559
SMAUG-T3	KeyGen	1,687.9	1,042.2	1.619
	Encaps	1,548.3	932.7	1.660
	Decaps	1,914.5	1,165.2	1.643
SMAUG-T5	KeyGen	2,705.9	1,599.2	1.692
	Encaps	2,507.2	1,445.8	1.734
	Decaps	3,002.6	1,758.7	1.707

The Slothy implementation of HAETAE achieved speedups of $1.191\text{--}1.212\times$ for KeyGen, $1.050\text{--}1.055\times$ for Sign, and $1.064\text{--}1.099\times$ for Verify over Intrinsic. Because NTT and FFT account for a large proportion of KeyGen, the Assembly and Slothy improvements to these two kernels were directly reflected in end-to-end performance. By contrast, Sign incurs substantial costs from Gaussian sampling and the rejection loop, and Slothy has little effect on the complete sampler; consequently, most of the maximum kernel-level improvement was diluted. For Verify, the NTT and pointwise improvements remained, but the unchanged Keccak/SHAKE operations and top-level control overhead resulted in a smaller gain than for KeyGen.

Table 9. End-to-end cycle counts for the Intrinsic, Assembly, and Slothy implementations. Values are medians in thousands of cycles (K).

Parameter	Operation	Median cycles			Speedup	
		Intrinsic	Assembly	Slothy	Asm. / Intr.	*Slothy / *Intr.
HAETAE-2	KeyGen	2,616.4	2,422.3	2,158.0	1.080	1.212
	Sign	20,430.7	19,581.5	19,455.6	1.043	1.050
	Verify	761.9	745.3	716.3	1.022	1.064
HAETAE-3	KeyGen	5,973.8	5,647.5	4,992.6	1.058	1.197
	Sign	21,204.7	20,341.4	20,168.2	1.042	1.051
	Verify	1,439.0	1,423.5	1,339.3	1.011	1.074
HAETAE-5	KeyGen	5,693.0	5,375.7	4,780.1	1.059	1.191
	Sign	42,381.4	40,674.6	40,167.3	1.042	1.055
	Verify	1,842.7	1,842.3	1,677.2	1.000	1.099
SMAUG-T1	KeyGen	546.5	514.2	493.1	1.063	1.108
	Encaps	463.0	419.0	392.4	1.105	1.180
	Decaps	620.9	559.9	528.8	1.109	1.174
SMAUG-T3	KeyGen	1,042.2	975.1	922.9	1.069	1.129
	Encaps	932.7	845.3	784.1	1.103	1.190
	Decaps	1,165.2	1,057.8	987.8	1.102	1.180
SMAUG-T5	KeyGen	1,599.2	1,486.3	1,388.4	1.076	1.152
	Encaps	1,445.8	1,301.8	1,192.8	1.111	1.212
	Decaps	1,758.7	1,586.6	1,463.2	1.109	1.202

The Assembly implementation of SMAUG-T achieved a $1.063\text{--}1.111\times$ speedup over Intrinsic, whereas the Slothy implementation achieved a $1.108\text{--}1.212\times$ speedup. Because the optimized polynomial kernel is invoked repeatedly in matrix–vector multiplication, even small kernel improvements accumulate as the parameter size increases. However, packing and scheme-level control remain unchanged, making the end-to-end gain smaller than the maximum kernel-level gain.

4.4 Stack Usage

To determine whether the performance improvements require additional temporary storage, the largest stack usage among the three top-level operations is compared for each parameter set. The stack watermark is read before the cycle samples are printed, and the same measurement procedure is applied to Intrinsic, Assembly, and Slothy.

The maximum stack usage of HAETAE increased by 8 bytes for Assembly and Slothy relative to Intrinsic and was therefore essentially unchanged. This is because the maximum usage is determined by the temporary arrays of the Gaussian sampler rather than by the FFT or smaller assembly kernels. By contrast, Assembly and Slothy used 492 fewer bytes than Intrinsic for SMAUG-T. This reduction resulted from

Table 10. Peak stack use in bytes. Each value is the largest of KeyGen, Sign/Encaps, and Verify/Decaps.

Parameter	Intrinsic	Assembly	Slothy
HAETAE-2	83,368	83,376	83,376
HAETAE-3	126,808	126,816	126,816
HAETAE-5	159,864	159,872	159,872
SMAUG-T1	15,796	15,304	15,304
SMAUG-T3	20,220	19,728	19,728
SMAUG-T5	25,724	25,232	25,232

eliminating the intermediate Toom–Cook product array and adding the results directly to the accumulation array.

5 Conclusion

The principal strength of the proposed staged method lies in assigning Assembly and Slothy distinct roles in solving different problems. Assembly reused intermediate values in registers and eliminated unnecessary memory accesses and temporary arrays, whereas Slothy reduced pipeline idle time by interleaving independent operations within this streamlined dataflow. Consequently, after both stages were applied, the forward NTT achieved a $2.349\text{--}2.424\times$ speedup over Intrinsic, the fixed-point FFT including squared-value accumulation achieved a $1.701\text{--}1.732\times$ speedup, and the SMAUG-T matrix–vector multiplication achieved a $1.322\text{--}1.350\times$ speedup. End-to-end performance improved by $1.050\text{--}1.212\times$ for HAETAE and $1.108\text{--}1.212\times$ for SMAUG-T.

A limitation of the proposed method is that the effectiveness of Slothy depends on how many mutually independent operations remain within a kernel. Although both Assembly and Slothy improved the NTT and SMAUG-T polynomial multiplication, Slothy provided little additional improvement for Keccak/SHAKE and the fixed-point FFT itself, for which Assembly had already reduced data movement. The complete Gaussian sampler also exhibited essentially identical performance to Intrinsic, with a speedup of $0.998\text{--}0.999\times$.

Future work should explore data layouts, loop-unrolling ranges, kernel boundaries, and instruction scheduling jointly to integrate the Assembly design and automated scheduling stages more closely. In particular, for these low-gain paths, the exploration scope could be extended to include top-level call structures and memory accesses. We also leave as future work the application of the extended Slothy instruction model to other Cortex-M processors and the validation of the same staged method in MVE-based PQC implementations beyond HAETAE and SMAUG-T.

References

1. Abdulrahman, A., Becker, H., Kannwischer, M.J., Klein, F.: Fast and clean: Auditable high-performance assembly via constraint solving.

- Cryptology ePrint Archive (2022)
2. Alwen, J., Krenn, S., Pietrzak, K., Wichs, D.: Learning with rounding, revisited: New reduction, properties and applications. In: Annual cryptology conference. pp. 57–74. Springer (2013)
 3. Arm Limited: Helium Programmer’s Guide: Coding for Helium (2020), [https://developer.arm.com/documentation/102095/0100/](https://developer.arm.com/documentation/102095/0100/version-1.1/Document-ID-102095_0100_01_en), version 1.1, Document ID 102095_0100_01_en
 4. Arm Limited: Arm Cortex-M55 Processor Software Optimization Guide, issue 03 edn. (2022), <https://developer.arm.com/documentation/102692/latest/>
 5. Arm Limited: Arm Cortex-M55 Processor Technical Reference Manual (August 2022), <https://developer.arm.com/documentation/101051/0101/>, revision r1p1, Issue 02, Document ID: 101051_0101_02_en
 6. Becker, H., Mera, J.M.B., Karmakar, A., Yiu, J., Verbauwhede, I.: Polynomial multiplication on embedded vector architectures. Cryptology ePrint Archive (2021)
 7. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the indifferenciability of the sponge construction. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 181–197. Springer (2008)
 8. Cheon, J.H., Choe, H., Choi, J., Hong, D., Hong, J., Jung, C.G., Kang, H., Lee, J., Lim, S., Park, A., et al.: Smaug-t: the key exchange algorithm based on module-lwe and module-lwr. Algorithm Specifications Version **3** (2024)
 9. Cheon, J.H., Choe, H., Devevey, J., Güneysu, T., Hong, D., Krausz, M., Land, G., Möller, M., Stehlé, D., Yi, M.: HAETAE: Shorter lattice-based Fiat-Shamir signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems* **2024**(3), 25–75 (Jul 2024). <https://doi.org/10.46586/tches.v2024.i3.25-75>, <https://tches.iacr.org/index.php/TCHES/article/view/11669>
 10. Cook, S.A., Aanderaa, S.O.: On the minimum computation time of functions. *Transactions of the American Mathematical Society* **142**, 291–314 (1969)
 11. Dworkin, M.J., et al.: Sha-3 standard: Permutation-based hash and extendable-output functions (2015)
 12. Guido, B., Joan, D., Michaël, P., Gilles, V.: The k reference (2011)
 13. Hyoungh, Y., Cho, S., Kim, U., Lee, M., Seo, H., Sim, M.: Memory-efficient implementation of smaug-t and haetae. Cryptology ePrint Archive (2026)
 14. Kannwischer, M.J., Krausz, M., Petri, R., Yang, S.Y.: pqm4: Benchmarking nist additional post-quantum signature schemes on microcontrollers. Cryptology ePrint Archive (2024)
 15. Kim, Y., van Vredendaal, C., Banegas, G., Seo, S.C.: Low-stack haetae for memory-constrained microcontrollers. Cryptology ePrint Archive (2026)
 16. Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography* **75**(3), 565–599 (2015)

17. Lyubashevsky, V.: Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In: International conference on the theory and application of cryptology and information security. pp. 598–616. Springer (2009)
18. Narlı, O., Pehlivanoğlu, M.K., Akleylek, S.: A new optimized implementation of *smaug-t* for lightweight devices. In: Nordic Conference on Secure IT Systems. pp. 77–90. Springer (2025)
19. Sim, M., Lee, M., Cho, S., Hyoun, Y., Seo, H.: Evaluating hybrid kem/dsa for kpgc and nist pgc on arm cortex-m4. *Cryptology ePrint Archive* (2026)
20. Sim, M., Lee, M., Seo, H.: Haetae on armv8. *Electronics* **13**(19), 3863 (2024)
21. SLOTHY Project: pqmx: Post-quantum cryptography on arm cortex-m. <https://github.com/slothy-optimizer/pqmx> (2026), software artifact, commit 89465b9
22. Toom, A.L.: The complexity of a scheme of functional elements realizing the multiplication of integers, published in soviet math (translations of dokl. adad. nauk. sssr), 4 (1963)