

DP-Guided Schedule Selection for $\mathbb{F}_2[x]$ Multiplication across ISAs

Junyu Zhou¹, Xiao Lan², Jing Wang³, Hao Ren¹, Weiran Liu⁴ and Si Gao⁵

¹ School of Cyber Science and Engineering, Sichuan University

² Cyber Science Research Institute, Sichuan University

³ School of Integrated Circuits, Huazhong University of Science and Technology, Wuhan, China

⁴ Individual Author

⁵ Trusted Computing and Information Assurance Laboratory,
Institute of Software, Chinese Academy of Sciences

helloworld [at] junyu33.me; {lanxiao, hao.ren} [at] scu.edu.cn

nick585108 [at] 163.com; liuweiran900217 [at] gmail.com

gaosi [at] iscas.ac.cn

Abstract. Efficient multiplication over $\mathbb{F}_2[x]$ is a core primitive in classical and post-quantum cryptographic software. As ARM and RISC-V become increasingly relevant for open-source cryptographic libraries such as OpenSSL and liboqs, arithmetic kernels must be retuned across a wider range of Instruction Set Architectures (ISAs). High-performance arithmetic libraries recursively apply Karatsuba- and Toom-style decomposition rules, each splitting the operands into smaller subproblems and recombining the resulting products. The resulting decomposition schedule is the recursive tree of rule choices from the top-level multiplication down to the base kernels. However, hand-tuned schedules can miss better decompositions at large operand sizes, while ISA-specific primitive costs complicate cross-platform retuning. In this paper, we formulate recursive $\mathbb{F}_2[x]$ multiplication as a schedule-selection problem. The recursive rule space has optimal substructure, allowing dynamic programming (DP) over a compact ISA profile that accounts for base multiplication, XORs, memory traffic, and recursive overhead. The resulting schedules are materialized as dispatch tables or fixed-size kernels. Across x86-64, ARM64, RISC-V64, and RP2040, DP costs track runtime trends across input sizes, and the full rule set achieves $1.04\text{--}1.18\times$ geomean speedup over **gf2x**. HQC and BIKE integrations on the three 64-bit platforms yield $1.06\text{--}1.21\times$ and $1.01\text{--}1.70\times$ end-to-end speedups, respectively, against their platform baselines. We also evaluate Classic McEliece on these platforms. With prepared profiles, HQC/BIKE schedule selection takes $0.075\text{--}0.762$ seconds, avoiding candidate compilation and benchmarking during search. The scheme provides a systematic alternative to hand tuning and measurement-driven per-size empirical search, reducing the effort required to port and retune carry-less multiplication kernels.

Keywords: $\mathbb{F}_2[x]$ arithmetic · carry-less multiplication · software implementation · post-quantum cryptography · schedule selection

1 Introduction

Binary-field arithmetic is central to classical and post-quantum cryptography. In many polynomial-basis implementations, multiplication is computed by carry-less multiplication over $\mathbb{F}_2[x]$, followed by reduction modulo the field polynomial. For example, polynomial multiplication accounts for approximately 49% and 72% of key-generation time in the optimized x86-64 implementations of HQC-5 [AAB⁺22] and BIKE-L5 [DGK24], respectively.

Consequently, improving $\mathbb{F}_2[x]$ multiplication remains a critical performance challenge. This challenge is amplified by the growing use of ARM and RISC-V ISAs in cryptographic software, where high-performance arithmetic kernels are often less mature or less extensively tuned than on x86. $\mathbb{F}_2[x]$ multiplication therefore has to be considered across several implementation contexts, not just as a fixed x86 kernel.

Optimizing $\mathbb{F}_2[x]$ multiplication involves three intertwined questions, which organize the following discussion: which recursive formulae are available, how implementation variants should be selected, and how the choice changes across target ISAs.

Carry-less multiplication and recursive decompositions. The schoolbook method computes carry-less multiplication by shift-and-add in $O(m^2)$ time. Karatsuba [KO62] and Toom–Cook [Too63, CA69] reduce the asymptotic complexity to $O(m^{1.585})$ and $O(m^{\log_k(2k-1)})$, respectively, at the cost of evaluation and interpolation overhead. This trade-off typically favors Karatsuba or small Toom instances except at large operand sizes. Montgomery’s five-, six-, and seven-term formulae [Mon05] provide further decompositions beyond classical Karatsuba and standard Toom–Cook.

Other approaches include word-parallel comb methods [LD00] and FFT/NTT-based or algebraic-transform methods, which can achieve quasi-linear complexity in suitable algebraic settings [CT65, Pol71, CK91].

Performance depends on both the base kernel and the recursive decomposition. Brent et al.’s benchmarks [BGTZ08] show that the most efficient strategy varies with operand size and hardware. Bodrato [Bod07] further shows that different splits and interpolation sequences can change operation counts even within a fixed Toom–Cook scheme.

Automatic tuning and planning. Numerical libraries have long used planner- or generator-based approaches to avoid relying on a single fixed algorithm. FFTW [FJ05] uses a planner to choose decompositions and codelets for a given transform size and machine. ATLAS [WPD01] applies empirical search to linear-algebra kernels, while SPIRAL [PMJ+05] generates transform implementations from algebraic descriptions.

Superoptimization systems search directly over program fragments or compiler IR to discover missed low-level optimizations, starting from early exhaustive-search systems and the GNU superoptimizer line of work [Mas87, GK92]. STOKE [SSA13] formulates binary superoptimization as a stochastic search problem, while Souper [SCC+17] and Minotaur [LMR24] use synthesis-oriented techniques to improve compiler-generated code, including SIMD code in Minotaur.

Cross-platform cryptographic implementations. BIKE studies optimize Haswell/Cortex-M4 and carry-less-instruction paths [CCK21, CGKT22]. Robert and Véron [RV23] explore Karatsuba/Toom for BIKE and HQC; Chen et al. [CCPY26] compare Toom–Karatsuba and additive-FFT HQC on x86 and ARM, exposing ISA- and size-dependent crossovers.

Beyond these scheme-specific kernels, pqm4 provides testing and benchmarking infrastructure for PQC on Cortex-M4 [KRSS19]. On RISC-V, Stoffelen studies optimized cryptographic primitives and multiprecision arithmetic [Sto19], while Seo et al. optimize polynomial multiplication using operand-length choices, Karatsuba, and instruction extensions [SKE+21]. The VecFalcon work [ZZ26] vectorizes Falcon’s Gaussian sampler across x86, NEON, and RVV, and optimizes FFT/iFFT on RISC-V.

Jang et al.’s *GfmulOpt* performs measurement-driven bottom-up search on AVX2 and Neon [JLK+26a]. At each size, it generates, compiles, and benchmarks branch-free Karatsuba/Toom candidates using previously selected child schedules, retaining the fastest candidate. Their Cortex-M4 work instead combines radix-16 and FAFFT multiplication with operation-count selection for HQC [JLK+26b].

Gap and positioning. Together, these lines of work show that high-performance carry-less multiplication depends on the available recursive formulae, the selection among implementation variants, and the costs of the target ISA. Yet they still leave a practical tuning problem. The best decomposition schedule depends on operand size and ISA-specific primitive costs. **gf2x** empirically tunes low-level kernels and thresholds [BGTZ08], but its general-purpose path is not designed for fixed-size cryptographic kernels. For example, optimized HQC uses dedicated Toom–Karatsuba kernels [AAB⁺22]. However, these kernels use **manually selected decomposition schedules**, which may miss faster alternatives. Jang et al.’s *GFmulOpt* addresses this limitation through bottom-up generation and benchmarking of candidate kernels [JLK⁺26a]. It produces strong schedules, but retains **target-side measurements inside the per-size search loop**, repeated across targets.

These approaches rely on hand tuning or target-side empirical search at different levels. Our goal is to reduce this dependence by separating the algebraic decomposition-rule library from a compact ISA cost profile **calibrated independently of candidate-schedule runtimes**. We formulate recursive $\mathbb{F}_2[x]$ multiplication as a schedule-selection problem. Once a top-level rule is fixed, its recursive subproblems are independent in the model, exposing optimal substructure that can be solved by dynamic programming. The resulting scheme selects per-size decomposition schedules without compiling and benchmarking candidates during search, reducing the effort required to port and retune multiplication-heavy cryptographic software across ISAs. Specifically:

- We formulate recursive carry-less multiplication as a schedule-selection problem, separating an algebraic rule library from a reusable ISA cost profile. The resulting optimal-substructure recurrence is solved by DP, producing variable-size dispatch tables and compiler-generated fixed-size kernels.
- We evaluate model accuracy and schedule quality on x86-64, ARM64, RISC-V64, and RP2040. Costs track runtime trends across input sizes; the full rule set achieves 1.04–1.18× geomean speedup over **gf2x**.
- We integrate schedules into HQC, BIKE, and Classic McEliece on x86-64, ARM64, and RISC-V64. HQC/BIKE achieve 1.06–1.21×/1.01–1.70× end-to-end speedups against platform baselines, with subsecond selection from prepared profiles.

The rest of the paper is organized as follows. Section 2 introduces word-level multiplication, decomposition rules and schedule selection. Section 3 formulates the DP-guided multiplication scheme. Section 4 evaluates the model accuracy, schedule performance, and end-to-end impact of our scheme. Finally, Section 5 concludes the paper and discusses future directions.

2 Preliminaries

2.1 Word-Level Multiplication over $\mathbb{F}_2[x]$

Table 1 summarizes the key notations used in the paper.

An element of $\mathbb{F}_2[x]$ is represented by a bit string, with one bit per coefficient. Addition is therefore bitwise XOR, and multiplication before reduction is carry-less multiplication. Software implementations group coefficients into machine words, so a large polynomial multiplication is built from word-level carry-less products and linear XOR recombination.

In this work, m denotes the input size in machine words. On 64-bit ISAs, the base product uses the local carry-less instruction when available; RP2040 uses a portable 32-bit scalar base kernel. This word-level view is the basis for our cost model: recursive multiplication schedules are compared by counting base products, XOR work, memory traffic, and recursive overhead.

Table 1: Key notations and their meanings

Notation	Meaning
m	Input size, measured in machine words unless otherwise stated.
K_i, K_i^*	Input-length-indexed and unbalanced Karatsuba-like rules.
K_{ioj}	Explicitly composed (fused) Karatsuba-like rule.
T_k	k -way Toom–Cook decomposition rule.
$\mathcal{S}$	Allowed decomposition-rule set supplied to the DP.
$C(m)$	Modeled multiplication cost for size m .
$\mathcal{L}, C_{\text{leaf}}(m)$	Leaf region and fixed-kernel cost for $m \in \mathcal{L}$.
c_{clmul}, c_{32}	Base carry-less primitive and 32-bit scalar base-kernel costs.
$c_{\oplus}, c_{\text{rd}}, c_{\text{wr}}, c_{\text{call}}$	Word-level XOR, memory read/write, and recursive-call overhead costs.

2.2 Recursive Decomposition Rules and Schedules

In general, a *decomposition rule* is one algebraically valid local step that replaces an m -word multiplication by smaller recursive multiplications plus linear recombination work. A k -way decomposition splits each operand into k top-level word-aligned blocks and recombines the resulting products through fixed linear combinations. We use K_i for a Karatsuba-like rule specialized to an *in-word* input. We write K_{ioj} for an explicitly fused composition; for example, K_{2o2} and K_{2o2o2} expose two and three binary stages, respectively. A superscript $*$ denotes an unbalanced split.

We write T_k for a standard k -way *Toom–Cook rule*, where operands are split into k blocks, evaluated at fixed points, multiplied recursively, and interpolated back. Our word-level T_3 implementation evaluates at $0, 1, x, x + 1, \infty$: a $3n$ -word input therefore creates three n -word subproblems and two $(n + 1)$ -word subproblems.

A *decomposition schedule* is the full recursive tree obtained by applying decomposition rules until reaching a base multiplication kernel, with each node labeled by its size and its outgoing edges determined by the chosen rule. When the main recursive sizes form a simple path, we use chain notation such as $4 \xrightarrow{K_2} 2 \xrightarrow{K_2} 1$ to summarize the schedule. A padding edge (written $\xrightarrow{\text{pad}}$) denotes temporary zero-padding to a larger size; the result is interpreted for the original unpadded length. In general, especially with ragged rules, the schedule is a tree with branches of different sizes.

The *allowed rule set* $\mathcal{S}$ is the collection of decomposition rules made available to the DP for a given experiment or implementation. A *leaf* is an implementation-specific base case at which recursion stops and a fixed multiplication kernel is invoked. The *leaf region* $\mathcal{L}$ contains these base-case sizes, with fixed-kernel cost $C_{\text{leaf}}(m)$ for $m \in \mathcal{L}$.

3 Dynamic-Programming Schedule Selection

Before diving deeply into our schedule selection scheme, we first provide a toy example showing the complexity of schedule selection.

A toy example. Consider multiplying two 3-word operands on x86. One schedule applies a direct 3-way rule, denoted as $3 \xrightarrow{K_3} 1$. Another schedule pads the input to 4 words and applies two levels of 2-way Karatsuba, written as $3 \xrightarrow{\text{pad}} 4 \xrightarrow{K_2} 2 \xrightarrow{K_2} 1$. Counting only carry-less multiplications may favor one of them, but the actual runtime also depends on the numbers of XOR operations and memory accesses, as well as their relative costs across ISAs. The better schedule is therefore not obvious without a platform-aware cost model.

Our key insight is that schedule selection has a natural **dynamic-programming structure**. A **calibrated word-level cost model** captures the main arithmetic and memory-traffic tradeoffs well enough to select **competitive decomposition schedules**,

despite microarchitectural effects such as cache behavior and out-of-order execution. The DP therefore uses modeled cost as a ranking criterion for selecting low-cost decomposition schedules at each input size.

Our scheme proceeds in three steps. First, it builds a word-level cost estimator for each decomposition rule. Second, it uses dynamic programming to relax these costs over all subproblem sizes. Finally, it reconstructs the minimum-cost decomposition schedule, as summarized in Figure 1.

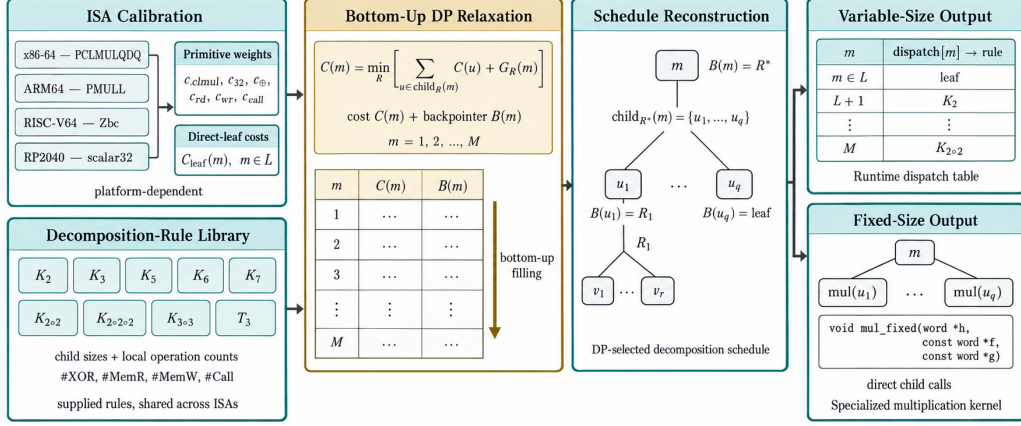


Figure 1: Overview of the proposed scheme. ISA-specific calibration and the supplied decomposition-rule library parameterize the DP relaxation. The resulting backpointers reconstruct a schedule that is materialized either as a variable-size dispatch table or as a fixed-size specialized kernel.

3.1 Word-Level Cost Estimator

The first step is to express different decomposition rules in a common quantitative unit. We therefore assign each local decomposition rule a word-level cost. For a rule applied to an m -word input, the estimator records the recursive submultiplications it creates and the nonrecursive work needed around them: word-level XORs, memory reads and writes, and fixed recursive overhead (the glue term, e.g., argument setup and register save/restore around recursive calls).

We adopt a *parameterized word-level cost model*, in which these operation counts are combined with platform-specific weights such as $c_{\oplus}$, c_{rd} , c_{wr} , and c_{call} . The concrete weight values and their sources used in our experiments are listed in Appendix A; implementers can replace these weights with a profile calibrated for the target implementation environment. To instantiate the counts for the implemented rules, we reconstruct their micro-operation schedules directly from the source code. Table 2 summarizes their recursive subproblems and nonrecursive operation counts.

Table 2 lists the implemented rules used in our scheme, not all algebraically valid decompositions. The composition rows are fused multi-stage rules: their child counts contain only the final n -word products, while their glue counts include work from every fused stage. The explicit $K_{3 \times 2}$ notation distinguishes that composition from the separate K_6 rule at the same input length. Additional Toom–Cook variants or other fused decompositions can be added by specifying their child sizes and nonrecursive operation counts. Thus, the DP optimum is relative to the supplied allowed rule set $\mathcal{S}$.

Using the parameterized word-level cost model introduced above, we express the cost of each decomposition rule as a *structural recurrence* that combines the costs of its recursive submultiplications with a linear overhead term accounting for nonrecursive recombination

Table 2: Estimated operation counts for the core decomposition rules.

Rule	Input	#submults	#XOR	#MemR	#MemW	#Call
K_2	$2n$	$3 \times n$	$7n$	$12n$	$5n$	3
K_3	$3n$	$6 \times n$	$21n$	$24n$	$10n$	6
$K_{2 \circ 2}$	$4n$	$9 \times n$	$35n$	$60n$	$25n$	9
K_5^*	$5n$	$1 \times n + 5 \times 2n$	$39n$	$82n$	$43n$	6
K_5	$5n$	$15 \times n$	$90n$	$180n$	$100n$	15
K_6	$6n$	$17 \times n$	$162n$	$304n$	$154n$	17
$K_{3 \circ 2}$	$6n$	$18 \times n$	$102n$	$192n$	$102n$	18
K_7^*	$7n$	$1 \times 3n + 2 \times 4n$	$28n$	$58n$	$30n$	3
K_7	$7n$	$22 \times n$	$236n$	$428n$	$206n$	22
$K_{2 \circ 2 \circ 2}$	$8n$	$27 \times n$	$133n$	$228n$	$95n$	27
$K_{3 \circ 3}$	$9n$	$36 \times n$	$270n$	$540n$	$324n$	36
T_3	$3n$	$3 \times n + 2 \times (n + 1)$	$76n + 36$	$92n + 22$	$56n + 18$	5

Note. The K_5 , K_6 , and K_7 rules are adapted from Montgomery’s five-, six-, and seven-term Karatsuba-like formulae [Mon05].

work, memory traffic, and recursive-call overhead. These recurrences provide a calibrated **analytical framework for comparing recursive structures**, rather than a **cycle-level simulator** for a specific processor.

Let $C(m)$ denote the modeled cost of multiplying two m -word operands. For a given decomposition rule applied at size m , the total cost is written as the sum of the costs of its recursive subproblems plus a linear overhead in the child sizes.

For a decomposition rule R , the cost is

$$C_R(m) = \sum_{m' \in \text{child}_R(m)} C(m') + G_R(m), \quad (1)$$

where $\text{child}_R(m)$ is the multiset of recursive subproblem sizes induced by the rule, and $G_R(m)$ is the nonrecursive glue cost from Table 2. For the Karatsuba-style rules, the glue term is instantiated as

$$G_R(m) = c_{\oplus} \# \text{XOR}_R + c_{\text{rd}} \# \text{MemR} + c_{\text{wr}} \# \text{MemW} + c_{\text{call}} \# \text{Call}_R. \quad (2)$$

Here, $\# \text{MemR}$ and $\# \text{MemW}$ count word-level memory reads and writes, respectively. For instance, applying $K_{3 \circ 2}$ to a $6n$ -word input gives $\text{child}_{K_{3 \circ 2}}(6n) = \{n\}^{18}$ and $G_{K_{3 \circ 2}}(6n) = 102n c_{\oplus} + 192n c_{\text{rd}} + 102n c_{\text{wr}} + 18c_{\text{call}}$. At the leaves of the recurrence, the boundary cost is defined by

$$C(m) = C_{\text{leaf}}(m), \quad m \in \mathcal{L}. \quad (3)$$

For x86-64 and RP2040, $C_{\text{leaf}}(m)$ combines direct-kernel operation counts with the basic-instruction weights such as $c_{\oplus}$, c_{rd} , c_{wr} , and c_{call} . For ARM64 and RISC-V64, it is obtained from target-local measurements of the direct kernels for $m = 1, \dots, 9$.

For supported non-exact sizes, the rule is evaluated with rounded word-aligned blocks. For example, a 3-way split uses block size $\lceil m/3 \rceil$ and a shortened final block when m is not divisible by three. The child multiset in Eq. (1) is then formed from the full blocks and the shortened block, and the XOR/read/write counts are evaluated with those word lengths.

3.2 Dynamic-Programming Relaxation

Given the local costs from Section 3.1, the next step is to choose which decomposition rule should be used at each operand size. Let $\mathcal{S}$ denote the allowed rule set for a given experiment, such as $\{K_2\}$, $\{K_2, K_3\}$, or any larger subset of the implemented rules in

Table 2. For each word size m , the DP stores two values: the cost array $C[m]$, which records the minimum modeled cost for multiplying two m -word operands, and the backpointer array $B[m]$, which records the top-level decomposition rule attaining that minimum.

For each candidate rule $R \in \mathcal{S}$ applicable at size m , the model cost is the local nonrecursive cost plus the already-computed costs of the child subproblems:

$$\text{cand}_R[m] = G_R(m) + \sum_{m' \in \text{child}_R(m)} C[m']. \quad (4)$$

The DP transition is then

$$C[m] = \min_{R \in \mathcal{S}_m} \text{cand}_R[m], \quad B[m] = \arg \min_{R \in \mathcal{S}_m} \text{cand}_R[m], \quad (5)$$

where $\mathcal{S}_m \subseteq \mathcal{S}$ is the subset of rules applicable at size m . This recurrence has the usual optimal-substructure property [Bel10]: once the top-level rule is fixed, the remaining choices are independent optimal schedules for the child sizes. Since every child size is smaller than m , the dependency graph is acyclic and the table can be filled in increasing order of m .

Algorithm 1: DP relaxation for decomposition schedules

Input: maximum size M , allowed rule set $\mathcal{S}$, leaf region $\mathcal{L}$, leaf costs

Output: cost table $C[1..M]$ and backpointer table $B[1..M]$

foreach $m \in \mathcal{L}$ **do**

$C[m] \leftarrow C_{\text{leaf}}(m); B[m] \leftarrow \perp;$

for $m \leftarrow 1$ **to** M **do**

if $m \in \mathcal{L}$ **then**

continue;

$C[m] \leftarrow \infty; B[m] \leftarrow \perp;$

foreach $R \in \mathcal{S}_m$ **do**

$v \leftarrow G_R(m) + \sum_{m' \in \text{child}_R(m)} C[m'];$

if $v < C[m]$ **then**

$C[m] \leftarrow v; B[m] \leftarrow R;$

return $C, B;$

Algorithm 1 summarizes the relaxation. A fixed rule order resolves ties deterministically. The arrays C and B store the minimum costs and corresponding top-level rules, while $\mathcal{S}_m$ encodes experiment-specific applicability constraints.

3.3 Decomposition-Schedule Reconstruction

After the relaxation, the backpointer table B is expanded recursively to obtain an executable decomposition schedule. We use this reconstructed schedule in two settings. For variable-size multiplication, the schedule is evaluated on the word grid and allows non-exact child sizes. For a fixed target size, the schedule can instead be specialized to one padded size, which removes runtime rule selection and branch decisions from the generated kernel.

3.3.1 Variable-Size Dispatch

For the variable-size multiplication case, the reconstructed schedule is materialized as a direct word-grid dispatch table. Each input size m stores the top-level rule $B[m]$. At runtime, the dispatcher reads this entry, invokes the corresponding decomposition rule, and the recursive calls repeat the same process on their child sizes. Non-exact sizes are

handled by the word-grid child-size convention of the selected rule; no top-level padding is introduced in this mode. We use this variable-size dispatch in Sections 4.1 and 4.2. The details are shown in Algorithm 2:

Algorithm 2: Reconstructing a decomposition schedule from backpointers

Input: backpointer table $B[1..M]$, target size m
Output: root node of the decomposition-schedule tree
Function $\text{ReconstructSchedule}(B, m)$:
 if $B[m] = \perp$ **then**
 return $\text{Node}(m, \text{base}, [])$;
 $R \leftarrow B[m]$;
 $\mathcal{C} \leftarrow []$;
 foreach $m' \in \text{child}_R(m)$ **do**
 $u \leftarrow \text{ReconstructSchedule}(B, m')$;
 Append u to $\mathcal{C}$;
 return $\text{Node}(m, R, \mathcal{C})$;

Figure 2 shows a concrete reconstructed schedule for a 100-word x86 input. The relevant backpointers are $B[100] = K_2$, $B[50] = K_3$, and $B[17] = B[16] = K_2$. The root produces three 50-word children, each yielding five 17-word children and one 16-word child. These split into $(9, 9, 8)$ and $(8, 8, 8)$, respectively, and terminate in direct multiplication kernels.

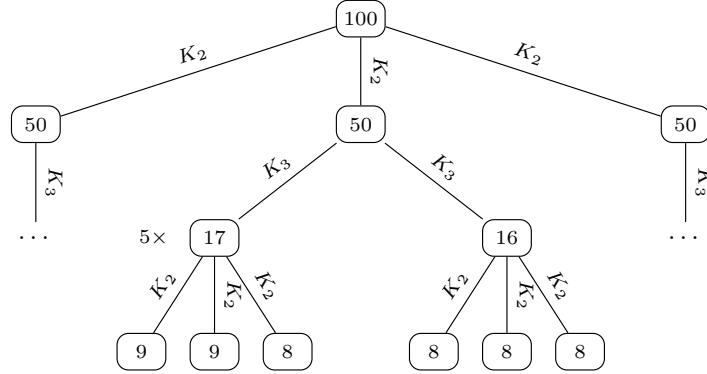


Figure 2: Decomposition schedule reconstructed from the backpointers for the 100-word x86 schedule. Nodes contain operand sizes, edge labels show the selected decomposition rules, multipliers group identical children, and $\dots$ abbreviates a repeated subtree. Boxed bottom-row nodes use direct kernels in the leaf region $\mathcal{L} = \{1, \dots, 9\}$.

3.3.2 Fixed-Size Specialization

For fixed-size specialized kernels, reconstruction can use explicit top-level padding. Algorithm 3 formalizes this variant: given a target size m , it searches a configurable padded interval, chooses the padded size m^* with the smallest modeled cost, and reconstructs the schedule from $B[m^*]$. In our fixed-size case studies in Sections 4.3 and 4.4, we search $m \leq t < 2m$, subject to the available buffer capacity. The wrapper retains a size- m multiplication interface, zero-pads the inputs to size m^* , and executes the corresponding exact-size schedule, allowing the offsets, child sizes, and recombination pattern to be written directly into the specialized kernel.

Thus the DP output is not only a scalar cost. The same backpointers also give an explicit decomposition schedule, either as a runtime dispatch table for variable sizes or as a fixed-size tree for specialized kernels.

Algorithm 3: Selecting a padded schedule for a fixed target size

Input: cost table $C[1..M]$, backpointer table $B[1..M]$, target size m
Output: padded size m^* and root node of the padded schedule

$m^* \leftarrow m;$
for $t \leftarrow m + 1$ **to** $\min(2m - 1, M)$ **do**
 if $C[t] < C[m^*]$ **then**
 $m^* \leftarrow t;$
 $T \leftarrow \text{ReconstructSchedule}(B, m^*);$
return $m^*, T;$

4 Evaluation

The cross-platform experiments use four representative targets: an x86-64 machine with PCLMULQDQ support, an ARM64 cloud platform with PMULL support, a RISC-V64 development board with Zbc carry-less multiplication support, and an RP2040 microcontroller using 32-bit scalar arithmetic. We also include RP2040 to highlight the effect of schedule selection on a low-end MCU platform without carry-less multiplication support. The hardware platforms are summarized in Table 3.

Table 3: Evaluation platform specifications.

Platform	Processor	Effective freq.	Relevant support
x86-64	Intel Core i7-13620H	3.8 GHz	PCLMULQDQ, AVX2
ARM64	Arm Neoverse N1	2.98 GHz	PMULL, AES
RISC-V64	XuanTie C908	1.6 GHz	RV64, Zbc, VLEN=128
RP2040	Arm Cortex-M0+	125 MHz	32-bit scalar

All implementations are compiled with `-O3`. For the 64-bit systems, we enable the locally supported carry-less multiplication primitive: PCLMULQDQ on x86-64, PMULL on ARM64, and Zbc carry-less multiplication on RISC-V64. The RP2040 experiments use the portable 32-bit scalar path. The x86 core measurements are CPU-pinned with turbo enabled and no fixed-frequency cap. Unless stated otherwise, cross-platform multiplication benchmarks cover input sizes from 1 to 128 machine words. This corresponds to 64 to 8192 input bits on x86-64, ARM64, and RISC-V64, and 32 to 4096 input bits on RP2040. This range covers the small and medium-sized carry-less multiplications commonly encountered as inner kernels in cryptographic software. The implementation, scripts, and raw benchmark data are available in a public repository (link to be added).

4.1 Correlation with Measured Runtime

We first evaluate how well the DP cost tracks measured runtime for the full allowed rule set $\mathcal{S}_{\text{full}} = \{K_2, K_3, K_5, K_6, K_7, K_{2 \circ 2}, K_{3 \circ 2}, K_{2 \circ 2 \circ 2}, K_{3 \circ 3}, T_3, K_5^*, K_7^*\}$. On RP2040, we retain $\mathcal{S}_{\text{full}} \setminus \{K_7\}$ to limit the footprint of fully unrolled kernels. Each strategy uses three independent runs with three timing samples per size; we report the median of the per-run medians. Spearman correlation measures whether the DP model preserves the runtime ordering across input sizes; Pearson correlation and relative error measure how closely the DP model cost follows the runtime curve. Table 4 reports the result for input sizes $1 \leq m \leq 128$; the corresponding model weights are listed in Appendix A. Appendix C reports the full error statistics.

Table 4: Runtime-vs-theory correlation for the full rule set.

Platform	Word	Spearman	Pearson	Rel. MAE	MAPE
x86-64	64	0.9960	0.9918	7.26%	9.63%
ARM64	64	0.9986	0.9917	8.41%	16.38%
RISC-V64	64	0.9980	0.9990	2.52%	7.50%
RP2040	32	0.9998	0.9999	0.82%	1.47%

The correlations are consistently high across all four systems. In particular, all Spearman values are above 0.98, indicating that the DP model preserves the runtime ordering across the tested input sizes. The RP2040 result is especially clean: its simple microcontroller core and fixed-frequency execution make word-level operation counts closely match the measured runtime ordering. Overall, these results show that the DP cost tracks the runtime trend of the selected full schedules across input sizes on all four platforms.

4.2 Schedule Selection and gf2x Comparison

We next compare schedules produced under different rule-set strategies. Let **pure2** denote $\{K_2\}$, **d23** denote $\{K_2, K_3\}$, and **full** denote $\mathcal{S}_{\text{full}}$ above. These three strategies use the variable-size dispatch described in Section 3.3.1. We use the sampling and aggregation protocol from Section 4.1. For each platform and each size $1 \leq m \leq 128$, we compare these three strategies and the **gf2x** reference, and record which one is fastest. We also define a within-1% band: a strategy is included when its runtime is within 1% of the empirical best among the four paths. This captures near-ties caused by run-to-run variation and small microarchitectural effects. If multiple strategies have the same measured runtime at one size, all tied strategies are counted as fastest. The concrete *full* schedules selected for each platform are summarized in Appendix B.

Table 5 reports the fastest and within-1% counts over all 128 sizes, while Table 6 gives measured per-invocation times at $m = 23, 32, 75, 128$, chosen to contrast irregular and power-of-two inputs across the measured range, with *pure2* as the reference schedule. Over all 128 sizes, the geometric-mean speedups of *full* over **gf2x** are $1.043\times$, $1.179\times$, $1.166\times$, and $1.089\times$ on x86-64, ARM64, RISC-V64, and RP2040, respectively. The corresponding ratios over *pure2* are $1.022\times$, $1.038\times$, $1.127\times$, and $1.116\times$.

The modest x86 gain is consistent with the limited headroom against **gf2x**’s mature, highly optimized x86 backend. On x86-64, ARM64, and RISC-V64, the empirically fastest of *pure2*, *d23*, and *full* varies with input size, consistent with microarchitectural effects beyond the coarse model. The largest within-1% count belongs to *d23* on RISC-V64, and *full* on x86-64, ARM64, and RP2040 (Table 5). On RP2040, which lacks carry-less multiplication, *full* wins more consistently (94 of 128 sizes).

Table 5: Empirical comparison of three DP strategies and **gf2x** over 128 input sizes.

Platform	Fastest count (of 128)				Within 1% (of 128)			
	pure2	d23	full	gf2x	pure2	d23	full	gf2x
x86-64	17	10	48	56	34	38	70	59
ARM64	35	35	58	0	53	84	107	0
RISC-V64	15	57	56	0	32	114	109	0
RP2040	1	9	94	32	4	17	104	37

4.3 HQC Implementation Case Study

The official HQC v5.0.0 release [HQC25] provides portable **ref** and optimized x86-64 **AVX2** implementations, but no dedicated optimized ARM64 or RISC-V64 backend. Table 7 summarizes the three experimental comparisons.

Table 6: Per-invocation time and speedup over *pure2* for the DP strategies and **gf2x** at representative sizes. Times are in ns.

Platform	m	pure2	d23	full	gf2x
x86-64	23	143.3	158.0 (0.907 $\times$)	158.2 (0.906 $\times$)	158.6 (0.904 $\times$)
	32	221.0	220.8 (1.00 $\times$)	207.6 (1.06$\times$)	258.3 (0.855 $\times$)
	75	1186.6	1231.3 (0.964 $\times$)	1237.8 (0.959 $\times$)	1095.7 (1.08$\times$)
	128	2074.7	2063.8 (1.01 $\times$)	1910.3 (1.09$\times$)	2384.6 (0.870 $\times$)
ARM64	23	293.0	268.2 (1.09 $\times$)	267.9 (1.09$\times$)	363.6 (0.806 $\times$)
	32	445.0	445.8 (0.998 $\times$)	417.2 (1.07$\times$)	514.6 (0.865 $\times$)
	75	2093.7	2024.9 (1.03$\times$)	2025.6 (1.03 $\times$)	2262.4 (0.925 $\times$)
	128	4116.2	4124.1 (0.998 $\times$)	3872.1 (1.06$\times$)	4500.2 (0.915 $\times$)
RISC-V64	23	1856.0	1557.5 (1.19$\times$)	1562.1 (1.19 $\times$)	1945.3 (0.954 $\times$)
	32	2770.0	2758.1 (1.00 $\times$)	2686.5 (1.03$\times$)	3069.1 (0.903 $\times$)
	75	13600.4	10830.9 (1.26$\times$)	10870.4 (1.25 $\times$)	13499.0 (1.01 $\times$)
	128	27690.7	27697.1 (1.00 $\times$)	27186.5 (1.02$\times$)	28158.8 (0.983 $\times$)
RP2040	23	389.1k	329.2k (1.18 $\times$)	329.2k (1.18$\times$)	377.7k (1.03 $\times$)
	32	506.1k	506.1k (1.00 $\times$)	501.0k (1.01 $\times$)	496.9k (1.02$\times$)
	75	2737.8k	2507.8k (1.09 $\times$)	2241.8k (1.22$\times$)	2662.2k (1.03 $\times$)
	128	4626.8k	4626.8k (1.00 $\times$)	4548.0k (1.02 $\times$)	4529.0k (1.02$\times$)

Note. The **gf2x** reference uses upstream commit `e175111`. On x86-64, ARM64, and RISC-V64, it was tuned on each target with `tune-lowlevel` and `tune-toom`. RP2040 uses the `generic32` path with a persistent multiplication pool.

Table 7: HQC fixed-size experimental design.

Platform	Backend	Baseline	Ours	External reference
x86-64	AVX2/PCLMULQDQ	Upstream Toom-3	DP	[JLK ⁺ 26a, CCPY26]
ARM64	Neon/PMULL	pure K_2	DP	[JLK ⁺ 26a, CCPY26]
RISC-V64	scalar/Zbc	pure K_2	DP	—

We modify `vect_mul` in these files: x86-64, `src/x86_64/common/hqc-{1,3,5}/gf2x.c`; ARM64, `artifact/neon/hqc-{1,3,5}/src/gf2x.c`; and RISC-V64, `src/ref/gf2x.c`. For each baseline–DP pair, the representation, direct leaves, reduction, serialization, and surrounding KEM implementation are fixed; only the recursive multiplication schedule changes. All DP-selected trees use the fixed-size specialization in Section 3.3.2. Empirical references use fixed trees from Jang et al.’s public optimizer, rerun through each target length on the evaluation platform. On x86-64 and ARM64, we also compare with Chen et al.’s additive-FFT implementation [CCPY26], with and without key caching. We omit RP2040 because this case study retains 64-bit-word multiplication kernels without a dedicated 32-bit adaptation.

Each implementation uses three CPU-pinned runs in rotating order, with 100 warmups and 1,000 samples per operation. We archive samples and report the median of run medians. Local measures one call to `vect_mul`, including reduction and cleanup, in a standalone benchmark. E2E measures complete key generation, encapsulation, and decapsulation calls, with no timers or counters inside the KEM code. Timer overhead is retained. We report ns, using 2.9184-GHz x86 TSC and 25-MHz ARM counters; RISC-V uses `CLOCK_MONOTONIC_RAW`.

Besides decomposition schedules, we report **selection cost**: the time to select a target’s decomposition schedule. DP times are medians of 21 invocations, including process startup, selection, and reconstruction. One-time profile calibration is excluded and reported in Appendix A. Jang et al.’s cumulative wall time includes candidate generation, compilation, and benchmarking through each target length.

4.3.1 Optimized x86-64 AVX2 Implementation

The DP uses 256-bit vector words with leaf region $\mathcal{L} = \{1\}$, costed using the upstream AVX2 `karat_mult_1` kernel. The candidate set is $\mathcal{S}_{\text{AVX2}} = \{K_2, K_3, K_5, K_6, K_7, K_5^*, K_7^*, K_{2\circ 2}, K_{3\circ 2}, K_{2\circ 2\circ 2}, K_{3\circ 3}, T_3^{(64)}, T_3^{(128)}, T_3^{(256)}\}$. The Toom-3 superscripts denote the bit shift used in evaluation and interpolation. For N vector words ($N \approx m/4$), the model searches $[N, 2N - 1]$ and compares the selected schedule with upstream Toom-3. The input lengths are $N = 70, 141, 226$ for HQC-1/3/5, respectively; Table 8 summarizes the decomposition schedules with representative chains. Table 9 reports local and end-to-end timings.

Table 8: Upstream, DP-selected, and empirical HQC AVX2 decompositions, with N in 256-bit vector words.

Param.	Variant	Decomposition schedule	Selection (s)
HQC-1	Upstream	$70 \xrightarrow{\text{pad}} 72 \xrightarrow{T_3} 24 \xrightarrow{T_3} 8 \xrightarrow{K_2} 4 \xrightarrow{K_2} 2 \xrightarrow{K_2} 1$	Manual
	DP	$70 \xrightarrow{T_3^{(128)}} 24 \xrightarrow{K_6} 4 \xrightarrow{K_{2\circ 2}} 1$	0.087
	[JLK+26a]	$70 \xrightarrow{T_3^{(128)}} 24 \xrightarrow{T_3^{(128)}} 9 \xrightarrow{K_2} 5 \xrightarrow{K_5} 1$	1.21k
HQC-3	Upstream	$141 \xrightarrow{\text{pad}} 144 \xrightarrow{T_3} 48 \xrightarrow{T_3} 16 \xrightarrow{K_2} 8 \xrightarrow{K_2} 4 \xrightarrow{K_2} 2 \xrightarrow{K_2} 1$	Manual
	DP	$141 \xrightarrow{T_3^{(256)}} 49 \xrightarrow{K_7} 7 \xrightarrow{K_7} 1$	0.112
	[JLK+26a]	$141 \xrightarrow{T_3^{(128)}} 48 \xrightarrow{T_3^{(128)}} 17 \xrightarrow{K_2} 9 \xrightarrow{K_2} 5 \xrightarrow{K_5} 1$	8.01k
HQC-5	Upstream	$226 \xrightarrow{\text{pad}} 234 \xrightarrow{T_3} 80 \xrightarrow{K_5} 16 \xrightarrow{K_2} 8 \xrightarrow{K_2} 4 \xrightarrow{K_2} 2 \xrightarrow{K_2} 1$	Manual
	DP	$226 \xrightarrow{T_3^{(256)}} 77 \xrightarrow{T_3^{(256)}} 27 \xrightarrow{T_3^{(128)}} 10 \xrightarrow{K_2} 5 \xrightarrow{K_5} 1$	0.142
	[JLK+26a]	$226 \xrightarrow{T_3^{(256)}} 77 \xrightarrow{T_3^{(256)}} 27 \xrightarrow{T_3^{(128)}} 10 \xrightarrow{K_2} 5 \xrightarrow{K_5} 1$	30.7k

Table 9: HQC AVX2 local and end-to-end timings in ns. Parentheses give speedup over upstream.

Param.	Implementation	Local	KGen	Enc	Dec
HQC-1	Upstream	6.51k	19.4k	37.5k	79.4k
	Ours	5.38k (1.21×)	18.0k (1.08×)	34.6k (1.08×)	74.9k (1.06×)
	[JLK+26a]	6.17k (1.06×)	18.9k (1.03×)	36.4k (1.03×)	77.6k (1.02×)
	[CCPY26]	14.3k (0.456×)	26.1k (0.744×)	47.6k (0.788×)	91.7k (0.865×)
	[CCPY26] ^c	10.9k (0.595×)	56.6k (0.344×)	35.6k (1.05×)	73.9k (1.07×)
HQC-3	Upstream	19.8k	47.3k	91.0k	169k
	Ours	16.2k (1.22×)	43.0k (1.10×)	82.7k (1.10×)	157k (1.08×)
	[JLK+26a]	17.7k (1.12×)	44.8k (1.05×)	86.1k (1.06×)	162k (1.05×)
	[CCPY26]	34.3k (0.578×)	60.4k (0.782×)	110k (0.826×)	194k (0.873×)
	[CCPY26] ^c	27.3k (0.726×)	123k (0.384×)	86.3k (1.06×)	156k (1.09×)
HQC-5	Upstream	49.2k	96.8k	189k	343k
	Ours	34.7k (1.42×)	81.0k (1.19×)	157k (1.20×)	296k (1.16×)
	[JLK+26a]	36.8k (1.34×)	83.3k (1.16×)	161k (1.17×)	302k (1.13×)
	[CCPY26]	49.6k (0.992×)	95.8k (1.01×)	171k (1.10×)	310k (1.11×)
	[CCPY26] ^c	35.2k (1.40×)	215k (0.451×)	126k (1.50×)	236k (1.45×)

Note. [CCPY26] uses AVX2/PCLMULQDQ without GFNI; c denotes caching. Cached Local (FFT-1) excludes one input's pretransform but includes the other's conversion and modular reduction. E2E caches FFT representations of fixed key polynomials; new inputs are transformed during Enc/Dec. Cached KGen includes ordinary key generation plus public-key and secret-key cache construction.

4.3.2 Native Neon Implementation on ARM64

The input lengths are $N = 139, 281, 451$ 128-bit vector words, where $N = \lceil m/2 \rceil$ because each vector word contains two 64-bit machine words. The leaf region is $\mathcal{L} = \{1\}$, and the candidate set is $\mathcal{S}_{\text{Neon}} = \mathcal{S}_{\text{AVX2}} \setminus \{T_3^{(256)}\}$. The superscripts denote the bit shift used in Toom-3 evaluation and interpolation.

Table 10 summarizes the decomposition schedules with representative chains: DP and Jang et al.’s empirical optimizer select Toom-3 roots but can differ in the shift and lower levels. Table 11 reports local and end-to-end timings.

Table 10: Pure- K_2 , DP-selected, and empirical HQC Neon decompositions, with N in 128-bit vector words. Ellipses omit repeated K_2 or Toom-3 levels.

Param.	Variant	Decomposition schedule	Selection (s)
HQC-1	pure K_2	$139 \xrightarrow{K_2} 70 \xrightarrow{K_2} 35 \xrightarrow{K_2} 18 \xrightarrow{K_2} 9 \xrightarrow{K_2} \dots \xrightarrow{K_2} 1$	Manual
	DP	$139 \xrightarrow{\text{pad}} 142 \xrightarrow{T_3^{(128)}} 49 \xrightarrow{K_7} 7 \xrightarrow{K_7} 1$	0.157
	[JLK+26a]	$139 \xrightarrow{T_3^{(128)}} 48 \xrightarrow{T_3^{(64)}} 17 \xrightarrow{T_3^{(128)}} 7 \xrightarrow{K_2} 4 \xrightarrow{K_2} 2 \xrightarrow{K_2} 1$	1.91k
HQC-3	pure K_2	$281 \xrightarrow{K_2} 141 \xrightarrow{K_2} 71 \xrightarrow{K_2} 36 \xrightarrow{K_2} 18 \xrightarrow{K_2} \dots \xrightarrow{K_2} 1$	Manual
	DP	$281 \xrightarrow{T_3^{(128)}} 95 \xrightarrow{T_3^{(128)}} 33 \xrightarrow{K_2} 17 \xrightarrow{K_2} 9 \xrightarrow{K_3} 3 \xrightarrow{K_3} 1$	0.237
	[JLK+26a]	$281 \xrightarrow{T_3^{(128)}} 95 \xrightarrow{T_3^{(128)}} 33 \xrightarrow{T_3^{(64)}} 12 \xrightarrow{K_3} 4 \xrightarrow{K_2} 2 \xrightarrow{K_2} 1$	11.0k
HQC-5	pure K_2	$451 \xrightarrow{K_2} 226 \xrightarrow{K_2} 113 \xrightarrow{K_2} 57 \xrightarrow{K_2} 29 \xrightarrow{K_2} \dots \xrightarrow{K_2} 1$	Manual
	DP	$451 \xrightarrow{T_3^{(128)}} 152 \xrightarrow{T_3^{(128)}} 52 \xrightarrow{K_2} 26 \xrightarrow{K_2} 13 \xrightarrow{K_2} 7 \xrightarrow{K_7} 1$	0.334
	[JLK+26a]	$451 \xrightarrow{T_3^{(64)}} 151 \xrightarrow{T_3^{(64)}} 51 \xrightarrow{T_3^{(64)}} 18 \xrightarrow{T_3^{(64)}} 7 \xrightarrow{K_2} \dots \xrightarrow{K_2} 1$	45.3k

Table 11: HQC native Neon local and end-to-end timings in ns. Parentheses give speedup over pure K_2 .

Param.	Implementation	Local	KGen	Enc	Dec
HQC-1	pure K_2	17.8k	39.1k	77.1k	144k
	Ours	14.2k (1.26$\times$)	35.2k (1.11$\times$)	69.5k (1.11$\times$)	133k (1.09$\times$)
	[JLK+26a]	14.8k (1.21 $\times$)	35.9k (1.09 $\times$)	70.9k (1.09 $\times$)	135k (1.07 $\times$)
	[CCPY26]	33.6k (0.532 $\times$)	54.9k (0.713 $\times$)	101k (0.764 $\times$)	181k (0.797 $\times$)
	[CCPY26] ^c	25.4k (0.703 $\times$)	120k (0.325 $\times$)	76.6k (1.01 $\times$)	145k (0.992 $\times$)
HQC-3	pure K_2	54.6k	104k	204k	349k
	Ours	37.0k (1.47$\times$)	86.3k (1.21$\times$)	169k (1.21$\times$)	295k (1.18$\times$)
	[JLK+26a]	41.3k (1.32 $\times$)	90.6k (1.15 $\times$)	177k (1.15 $\times$)	308k (1.13 $\times$)
	[CCPY26]	81.1k (0.673 $\times$)	132k (0.790 $\times$)	241k (0.847 $\times$)	406k (0.860 $\times$)
	[CCPY26] ^c	62.9k (0.868 $\times$)	275k (0.378 $\times$)	199k (1.03 $\times$)	335k (1.04 $\times$)
HQC-5	pure K_2	106k	194k	382k	649k
	Ours	83.6k (1.27$\times$)	172k (1.13$\times$)	337k (1.13 $\times$)	582k (1.11 $\times$)
	[JLK+26a]	92.6k (1.15 $\times$)	181k (1.07 $\times$)	356k (1.07 $\times$)	609k (1.07 $\times$)
	[CCPY26]	124k (0.852 $\times$)	213k (0.911 $\times$)	381k (1.00 $\times$)	657k (0.988 $\times$)
	[CCPY26] ^c	87.0k (1.22 $\times$)	484k (0.401 $\times$)	282k (1.35$\times$)	501k (1.29$\times$)

Note. We compile the Neon/PMULL.P64 backend of [CCPY26] on Neoverse N1. Superscript c and the cached Local/E2E timing scopes follow Table 9.

4.3.3 Native Scalar Zbc Implementation on RISC-V64

On the C908, the tuned `clmul/clmulh` direct kernels define $\mathcal{L} = \{1, \dots, 9\}$.

The input lengths are $m = 277, 561, 901$ 64-bit machine words. For each target, the model searches exact padded sizes in $[m, 2m - 1]$ using $\mathcal{S}_{\text{Zbc}} = \{K_2, K_3, K_5, K_6, K_7, K_5^*, K_7^*, K_{2 \circ 2}, K_{3 \circ 2}, K_{2 \circ 2 \circ 2}, K_{3 \circ 3}, T_3\}$. Table 12 shows the selected decomposition schedules. Table 13 reports local and end-to-end timings.

Table 12: DP-selected HQC decompositions on RISC-V64, with m in 64-bit machine words.

Param.	Decomposition schedule	Selection (s)
HQC-1	$277 \xrightarrow{\text{pad}} 288 \xrightarrow{K_{2 \circ 2}} 72 \xrightarrow{K_{2 \circ 2 \circ 2}} 9$	0.719
HQC-3	$561 \xrightarrow{\text{pad}} 576 \xrightarrow{K_{2 \circ 2 \circ 2}} 72 \xrightarrow{K_{2 \circ 2 \circ 2}} 9$	0.738
HQC-5	$901 \xrightarrow{\text{pad}} 972 \xrightarrow{K_{2 \circ 2}} 243 \xrightarrow{K_3} 81 \xrightarrow{K_3} 27 \xrightarrow{K_3} 9$	0.762

Table 13: HQC scalar-Zbc local and end-to-end timings in ns. Parentheses give speedup over pure K_2 .

Param.	Implementation	Local	KGen	Enc	Dec
HQC-1	pure K_2	124k	420k	817k	1.71M
	Ours	71.8k (1.73×)	369k (1.14×)	713k (1.15×)	1.55M (1.10×)
HQC-3	pure K_2	389k	1.17M	2.26M	3.96M
	Ours	223k (1.74×)	1.00M (1.17×)	1.92M (1.18×)	3.45M (1.15×)
HQC-5	pure K_2	924k	2.50M	4.82M	8.31M
	Ours	553k (1.67×)	2.09M (1.20×)	4.01M (1.20×)	7.13M (1.17×)

4.4 BIKE Implementation Case Study

BIKE provides a second fixed-size case study, with polynomial multiplication occupying a substantial fraction of key-generation time. Table 14 summarizes the comparisons.

Table 14: BIKE fixed-size experimental design.

Platform	DP backend	Baseline	Ours	External reference
x86-64	AVX2/PCLMULQDQ	Upstream K_2	DP	[JLK ⁺ 26a]
ARM64	Neon/PMULL	Upstream K_2	DP	–
RISC-V64	scalar/Zbc	Upstream K_2	DP	–

Note. The x86-64 baseline is the official optimized implementation [DGK24]. ARM64 and RISC-V64 retain upstream K_2 recursion with native direct leaves.

We use Jang et al.’s BIKE integration [JLK⁺26a] and rerun its public optimizer with its own rule library and code generator through each target length on our x86-64 platform. In `src/gf2x/gf2x_mul.c`, the upstream `gf2x_mod_mul_with_ctx` function invokes ragged Karatsuba recursion with AVX2 recombination and the runtime-selected PCLMULQDQ base kernel. We replace this internal multiplication with the DP-selected fixed-size tree, retaining reduction, the external interface, KEM code, and cleanup traffic.

The input lengths are $N = 49, 97, 161$ 256-bit vector words, where $N = \lceil m/4 \rceil$ for an m -word input. Our variant searches exact padded sizes that fit the existing BIKE buffers using the same $\mathcal{S}_{\text{AVX2}}$ rule set as Section 4.3.1, with $\mathcal{L} = \{1\}$ in the vector-word unit.

Appendix B summarizes the decomposition schedules and selection times (Table 20).

We follow the runtime measurement protocol in Section 4.3. Local measures the complete `gf2x_mod_mul` entry. All DP-selected trees use the fixed-size specialization in Section 3.3.2.

Table 15 shows that analytical selection reaches performance comparable to the empirical reference without compiling and benchmarking every candidate.

On ARM64 and RISC-V64, we replace the multiplication inside the same `src/gf2x/gf2x_mul.c` entry with compiler-generated fixed-size functions, retaining the reduction, cleanup, and surrounding KEM code. ARM64 uses $N = 97, 193, 321$ 128-bit vector words, $\mathcal{S}_{\text{Neon}}$ (Section 4.3.2), and $\mathcal{L} = \{1\}$. RISC-V64 uses $m = 193, 386, 641$ 64-bit words, $\mathcal{S}_{\text{Zbc}}$ (Section 4.3.3), and $\mathcal{L} = \{1, \dots, 9\}$. Table 16 follows the same runtime protocol; Appendix B lists the selected decomposition schedules (Table 20).

Table 15: BIKE x86-64 local and end-to-end timings in ns. Parentheses give speedup over upstream.

Param.	Implementation	Local	KGen	Enc	Dec
L1	Upstream	4.88k	119k	29.9k	512k
	Ours	3.34k (1.46$\times$)	101k (1.18$\times$)	28.4k (1.05$\times$)	496k (1.03$\times$)
	[JLK ⁺ 26a]	3.68k (1.33 $\times$)	107k (1.11 $\times$)	30.1k (0.992 $\times$)	502k (1.02 $\times$)
L3	Upstream	14.6k	360k	69.3k	1.52M
	Ours	10.2k (1.43$\times$)	299k (1.21$\times$)	65.7k (1.06$\times$)	1.46M (1.04$\times$)
	[JLK ⁺ 26a]	10.9k (1.35 $\times$)	314k (1.15 $\times$)	66.6k (1.04 $\times$)	1.47M (1.03 $\times$)
L5	Upstream	39.4k	950k	149k	4.20M
	Ours	21.4k (1.84$\times$)	664k (1.43$\times$)	134k (1.12$\times$)	3.90M (1.08$\times$)
	[JLK ⁺ 26a]	22.2k (1.77 $\times$)	682k (1.39 $\times$)	136k (1.09 $\times$)	3.93M (1.07 $\times$)

Table 16: BIKE ARM64 and RISC-V64 local and end-to-end timings in ns. Parentheses give speedup over upstream.

Param.	Variant	Local	KGen	Enc	Dec
ARM64					
L1	Upstream	12.0k	493k	120k	2.24M
	Ours	9.00k (1.33$\times$)	442k (1.11$\times$)	117k (1.03$\times$)	2.19M (1.02$\times$)
L3	Upstream	37.5k	1.47M	328k	6.90M
	Ours	22.8k (1.64$\times$)	1.19M (1.24$\times$)	314k (1.05$\times$)	6.67M (1.03$\times$)
L5	Upstream	99.4k	3.37M	818k	17.4M
	Ours	50.4k (1.97$\times$)	2.39M (1.41$\times$)	770k (1.06$\times$)	16.7M (1.04$\times$)
RISC-V64					
L1	Upstream	78.4k	2.44M	455k	32.0M
	Ours	43.9k (1.79$\times$)	1.90M (1.29$\times$)	427k (1.07$\times$)	31.6M (1.01$\times$)
L3	Upstream	243k	7.55M	1.14M	107M
	Ours	140k (1.73$\times$)	5.65M (1.34$\times$)	1.08M (1.06$\times$)	106M (1.01$\times$)
L5	Upstream	650k	18.2M	2.77M	255M
	Ours	277k (2.35$\times$)	10.7M (1.70$\times$)	2.31M (1.20$\times$)	249M (1.02$\times$)

4.5 Classic McEliece Integration

We additionally evaluate five parameter sets of libmceliece [BL25]. On ARM64 and RISC-V64, we replace the multiplication in $\text{GF}(2^k)[z]$ performed by `crypto_kem/PARAM/vec/gf_2mt_mul.c`'s `GF_mul` with a compiler-generated tree. The baseline uses pure K_2 ; both variants share the same scalar `gf_mul` leaf and outer field reduction. The DP uses $\mathcal{S}_{\text{full}}$ (Section 4.1) adapted to coefficient multiplication, with $\mathcal{L} = \{1\}$. The coefficient lengths are $t = 64, 96, 128, 119, 128$, respectively, with $k = 12$ for 348864 and $k = 13$ otherwise.

On x86-64, we compare a compiler-generated fixed-size PCLMUL+DP tree with the native AVX2 multiplication entry and integrate it into `genpoly_gen`, keeping intermediate powers in coefficient form before batch conversion to bitslices. Local compares the native entries in their respective representations; KeyGen includes the integration’s conversion work. Appendix B summarizes the decomposition schedules in Table 21.

We report only Local and KeyGen because the modified multiplication is used in key generation, not encapsulation or decapsulation. For each parameter set and variant, we report the median of three per-run medians. Each run uses 8 KeyGen samples and 65 Local batches of 8 calls, with each batch time divided by 8 before taking the median. Table 17 reports Local and KeyGen timings. Local gains have little effect on complete key generation, illustrating the limited end-to-end effect when the optimized multiplication occupies only a small part of key generation.

Table 17: libmceliece Local and KeyGen timings in ns: baseline $\rightarrow$ ours (speedup). The x86-64 baseline is AVX2; ARM64 and RISC-V64 use pure- K_2 VEC multiplication.

Param.	Local	KGen
x86-64		
348864	2.15k $\rightarrow$ 1.86k (1.158 $\times$)	8.52M $\rightarrow$ 8.28M (1.029 $\times$)
460896	4.80k $\rightarrow$ 4.16k (1.155 $\times$)	80.2M $\rightarrow$ 79.9M (1.004 $\times$)
6688128	6.33k $\rightarrow$ 5.73k (1.105 $\times$)	103M $\rightarrow$ 103M (1.001 $\times$)
6960119	5.88k $\rightarrow$ 5.63k (1.045 $\times$)	72.3M $\rightarrow$ 72.8M (0.993 $\times$)
8192128	6.33k $\rightarrow$ 5.73k (1.104 $\times$)	541M $\rightarrow$ 542M (0.999 $\times$)
ARM64		
348864	6.24k $\rightarrow$ 6.28k (0.994 $\times$)	35.1M $\rightarrow$ 35.1M (1.000 $\times$)
460896	14.3k $\rightarrow$ 11.6k (1.231 $\times$)	306M $\rightarrow$ 302M (1.011 $\times$)
6688128	19.0k $\rightarrow$ 19.0k (1.000 $\times$)	423M $\rightarrow$ 423M (1.001 $\times$)
6960119	18.8k $\rightarrow$ 18.3k (1.024 $\times$)	335M $\rightarrow$ 333M (1.006 $\times$)
8192128	19.0k $\rightarrow$ 19.0k (1.000 $\times$)	2311M $\rightarrow$ 2295M (1.007 $\times$)
RISC-V64		
348864	42.7k $\rightarrow$ 42.8k (0.996 $\times$)	472M $\rightarrow$ 472M (1.000 $\times$)
460896	103k $\rightarrow$ 79.1k (1.303 $\times$)	4251M $\rightarrow$ 4231M (1.005 $\times$)
6688128	130k $\rightarrow$ 131k (0.996 $\times$)	6323M $\rightarrow$ 6333M (0.999 $\times$)
6960119	131k $\rightarrow$ 125k (1.046 $\times$)	5530M $\rightarrow$ 5439M (1.017 $\times$)
8192128	130k $\rightarrow$ 131k (0.996 $\times$)	40222M $\rightarrow$ 40234M (1.000 $\times$)

4.6 Discussion and Limitations

Taken together, the experiments separate model accuracy, schedule quality, and application-level impact. The cross-platform correlation results in Section 4.1 show that modeled costs track runtime trends across input sizes; the per-size comparisons in Section 4.2 show that DP-selected schedules are often competitive with the measured alternatives, including `gf2x`. The HQC and BIKE case studies show that the resulting fixed-size trees remain effective after integration into complete KEM implementations. Compared with measurement-driven optimization, the main advantage is selection cost: the DP uses calibrated rule costs and completes in 0.075–0.762 seconds, whereas empirical search repeatedly generates, compiles, and benchmarks candidates.

The results also expose the limits of this advantage. No strategy is fastest at every size, and the mature x86 `gf2x` backend leaves limited headroom in the variable-size comparison. End-to-end gains further depend on the fraction of each KEM operation spent in polynomial multiplication. Classic McEliece demonstrates that local gains need not improve complete key generation (Table 17). Consequently, the scheme reduces schedule-tuning effort but

does not eliminate the need for efficient ISA-specific leaves and careful integration.

Finally, the DP optimum is relative to the supplied rule library and the accuracy of its operation counts. The model accounts for explicit memory traffic but does not simulate cache behavior, instruction scheduling, or out-of-order execution. It selects and reconstructs decomposition schedules; low-level kernel optimization remains a separate task.

5 Conclusion and Future Work

This work formulates recursive multiplication in $\mathbb{F}_2[x]$ as a dynamic-programming schedule-selection problem. A calibrated word-level model makes Karatsuba- and Toom-style rules searchable choices, with outputs for variable-size dispatch and compiler-generated fixed-size kernels. Across four platforms, selected schedules remain competitive with tuned `gf2x`. The fixed-size integrations cover HQC, BIKE, and Classic McEliece on x86-64, ARM64, and RISC-V64, improving local multiplication across the evaluated case studies. With prepared profiles, HQC/BIKE selection takes 0.075–0.762 seconds without compiling and benchmarking candidates, contrasting with Jang et al.’s measurement-driven search. These results support analytical schedule selection as a practical way to reduce cross-ISA tuning effort.

The same idea may also extend beyond $\mathbb{F}_2[x]$. Rings such as $\mathbb{Z}/2^m\mathbb{Z}$ would require different leaf kernels and cost parameters because carries and modular wraparound differ from the XOR-only setting. The separation between decomposition search and ring-specific primitive costs nevertheless suggests a broader path for multiplication-rule selection in post-quantum settings such as NTRU [CDH⁺20] and SABER [DKR⁺20].

A Cost Parameters

Table 18 lists the fixed platform-specific weights used by the DP for schedule ranking. These model weights are calibrated independently at instruction level; neither weights nor schedules are fitted to multiplication runtimes.

Table 18: Platform cost parameters used by the DP model.

Platform	c_{clmul}	c_{32}	$c_{\oplus}$	c_{sh}	c_{rd}	c_{wr}	c_{perm}	c_{call}
x86-64	1.000	–	0.300	0.333	1.000	1.167	0.417	13.333
ARM64	1.000	–	0.263	0.264	0.409	0.545	0.263	3.037
RISC-V64	1.000	–	0.174	0.174	0.495	0.615	0.500	9.497
RP2040	–	200.0	1.000	1.000	2.000	2.000	1.000	75.000

For the 64-bit profiles, c_{clmul} is the normalized cost of one carry-less multiplication primitive. For RP2040, we normalize $c_{\oplus}$ instead of c_{32} . A dash marks an inactive term. The four platform profiles are derived from published instruction data, target-local microbenchmarks, or processor documentation, as detailed below.

- x86-64 uses Intel’s Golden Cove P-core data as the closest official proxy [Int22]: $c_i = L_i/L_{\text{clmul}}$, where $L_{\text{clmul}} = 3$ for PCLMULQDQ and $L_i = (0.9, 1, 3, 3.5, 1.25)$ for XOR, shift, read, write, and permutation classes. The wrapper setup is counted as 40 simple integer/address-control operations, giving $c_{\text{call}} = 40/3$.
- ARM64 uses Neoverse-N1 microbenchmarks with eight operations per iteration and PMULL normalized to one, giving $(c_{\oplus}, c_{\text{sh}}, c_{\text{rd}}, c_{\text{wr}}, c_{\text{perm}}) = (0.263, 0.264, 0.409, 0.545, 0.263)$. We set $c_{\text{call}} = 3.037$ from the median incremental cost of child dispatch over matched direct leaves for $m = 1, \dots, 9$.

- RISC-V64 uses target-local C908 microbenchmarks normalized to one `clmul+clmulh` leaf, giving $(c_{\oplus}, c_{sh}, c_{rd}, c_{wr}, c_{perm}) = (0.174, 0.174, 0.495, 0.615, 0.500)$. The median dispatch increment over the nine direct-leaf sizes gives $c_{call} = 9.497$.
- RP2040 uses the RP2040 datasheet’s Cortex-M0+ timing table directly [Ras24]: $c_{32} = 200$ from a 201-cycle `mul_32` wrapper, $c_{\oplus} = c_{sh} = c_{perm} = 1$, $c_{rd} = c_{wr} = 2$, and $c_{call} = 72 + 3 = 75$.

Calibration cost. The x86-64 and RP2040 profiles are derived offline from processor documentation and assembly inspection. For the target-local profiles, one cold primitive-weight and dispatch calibration pass, including source compilation, measurement, and profile generation, took 3.42 s on ARM64 and 61.65 s on RISC-V64. These profiles are reused across all input sizes and case-study targets.

B Selected Schedules

Table 19 summarizes the paper-facing *full* schedules for $1 \leq m \leq 128$. The complete per-size choices are archived as `bench/data/summary/dispatch_choices*_dispatch29.csv` and can be regenerated with `build/model-dp -emit-dispatch-choices` using the platform profile and allowed-rule set from the benchmark scripts.

Table 19: Summary of selected *full* rules for $1 \leq m \leq 128$.

Platform	Selected rules
x86-64	$K_3 : \{22, \dots, 27, 50, 51, 69, \dots, 81, 98, 99\};$ $K_{2 \circ 2} : \{20, 30, \dots, 36, 86, \dots, 97, 104, \dots, 108, 118, \dots, 126\};$ $K_{2 \circ 2 \circ 2} : \{64\};$ $K_7^* : \{13, 14, 21, 28, 37, \dots, 42, 55, \dots, 58, 83, 84, 110, \dots, 115\}.$
ARM64	$K_3 : \{19, \dots, 27, 58, 60, \dots, 63, 66, \dots, 81\};$ $K_{2 \circ 2} : \{28, \dots, 36, 82, \dots, 110, 114, 118, 119, 121, 122\};$ $K_{2 \circ 2 \circ 2} : \{56, 64\}.$
RISC-V64	$K_3 : \{19, \dots, 27, 38, 58, 60, \dots, 63, 73, \dots, 81\};$ $K_{2 \circ 2} : \{28, \dots, 36, 82, \dots, 110, 114, 118\};$ $K_{2 \circ 2 \circ 2} : \{48, 56, 64, 72\};$ $K_7^* : \{122, \dots, 126\}.$
RP2040	$K_3 : \{10, \dots, 12, 22, \dots, 24\};$ $K_6 : \{48\};$ $K_{2 \circ 2} : \{26, 29, \dots, 32\};$ $K_{2 \circ 2 \circ 2} : \{64\};$ $T_3 : \{18, \dots, 21, 33, \dots, 45, 50, \dots, 53, 58, \dots, 60, 65, \dots, 128\};$ $K_7^* : \{27, 28, 54, \dots, 56\}.$

This core table records per-size root choices. Tables 8, 10, 12, and 20 summarize fixed-size decomposition schedules for x86-64 HQC, ARM64 HQC, RISC-V64 HQC, and BIKE across the three platforms, respectively. For Neon, the analytical DP and Jang et al.’s empirical optimizer often choose Toom-3 roots but differ in the shift and lower levels.

Table 21 records the libmceliece schedules used in the performance comparisons. BIKE chains follow a largest child at each split; libmceliece chains highlight nonbinary or fused rules. Other children use their own selected schedules.

C Additional Benchmark Tables

Table 22 gives the full runtime-vs-theory statistics for the *full* schedules. Spearman’s ρ measures rank agreement across input sizes, Pearson’s r measures linear association, and the error metrics use a zero-intercept least-squares conversion from model units to ns fitted on these same 128 sizes. This reporting conversion does not affect DP schedule selection.

Table 20: BIKE decomposition schedules and selection times. Length units are specified for each platform.

Param.	Variant	Decomposition schedule	Selection (s)
x86-64 (256-bit vector words)			
L1	DP	$49 \xrightarrow{K_7} 7 \xrightarrow{K_7} 1$	0.075
	[JLK+26a]	$49 \xrightarrow{T_3^{(256)}} 18 \xrightarrow{K_2} 9 \xrightarrow{K_2} 5 \xrightarrow{K_5} 1$	490
L3	DP	$97 \xrightarrow{\text{pad}} 98 \xrightarrow{K_2} 49 \xrightarrow{K_7} 7 \xrightarrow{K_7} 1$	0.085
	[JLK+26a]	$97 \xrightarrow{T_3^{(256)}} 34 \xrightarrow{K_2} 17 \xrightarrow{K_2} 9 \xrightarrow{K_2} 5 \xrightarrow{K_5} 1$	2.84k
L5	DP	$161 \xrightarrow{\text{pad}} 162 \xrightarrow{T_3^{(256)}} 56 \xrightarrow{K_2} 28 \xrightarrow{K_{2 \circ 2}} 7 \xrightarrow{K_7} 1$	0.107
	[JLK+26a]	$161 \xrightarrow{T_3^{(256)}} 55 \xrightarrow{T_3^{(256)}} 20 \xrightarrow{K_2} 10 \xrightarrow{K_2} 5 \xrightarrow{K_5} 1$	11.6k
ARM64 (128-bit vector words)			
L1	DP	$97 \xrightarrow{\text{pad}} 98 \xrightarrow{K_2} 49 \xrightarrow{K_7} 7 \xrightarrow{K_7} 1$	0.134
L3	DP	$193 \xrightarrow{T_3^{(64)}} 65 \xrightarrow{T_3^{(128)}} 23 \xrightarrow{K_2} 12 \xrightarrow{K_2} 6 \xrightarrow{K_{3 \circ 2}} 1$	0.189
L5	DP	$321 \xrightarrow{T_3^{(128)}} 109 \xrightarrow{T_3^{(128)}} 38 \xrightarrow{T_3^{(128)}} 14 \xrightarrow{K_2} 7 \xrightarrow{K_7} 1$	0.261
RISC-V64 (64-bit machine words)			
L1	DP	$193 \xrightarrow{\text{pad}} 216 \xrightarrow{K_{2 \circ 2 \circ 2}} 27 \xrightarrow{K_3} 9$	0.715
L3	DP	$386 \xrightarrow{\text{pad}} 432 \xrightarrow{K_2} 216 \xrightarrow{K_{2 \circ 2 \circ 2}} 27 \xrightarrow{K_3} 9$	0.728
L5	DP	$641 \xrightarrow{\text{pad}} 648 \xrightarrow{K_{2 \circ 2 \circ 2}} 81 \xrightarrow{K_3} 27 \xrightarrow{K_3} 9$	0.747

Table 21: Nontrivial DP-selected libmceliece schedules; pure- K_2 schedules are omitted. Lengths count $\mathbb{F}_{2^{12}}$ coefficients for 348864 and $\mathbb{F}_{2^{13}}$ coefficients for the other parameter sets.

Platform	Param.	Decomposition schedule
x86-64	460896	$96 \xrightarrow{K_2} 48 \xrightarrow{K_2} 24 \xrightarrow{K_2} 12 \xrightarrow{K_2} 6 \xrightarrow{K_2} 3 \xrightarrow{K_3} 1$
	6960119	$119 \xrightarrow{K_2} 60 \xrightarrow{K_2} 30 \xrightarrow{K_2} 15 \xrightarrow{K_2} 7 \xrightarrow{K_2} 3 \xrightarrow{K_3} 1$
ARM64	460896	$96 \xrightarrow{T_3} 34 \xrightarrow{K_2} 17 \xrightarrow{K_2} 9 \xrightarrow{K_3} 3 \xrightarrow{K_3} 1$
	6960119	$119 \xrightarrow{K_7} 17 \xrightarrow{K_2} 9 \xrightarrow{K_3} 3 \xrightarrow{K_3} 1$
RISC-V64	348864	$64 \xrightarrow{K_2} 32 \xrightarrow{K_2} 16 \xrightarrow{K_2} 8 \xrightarrow{K_2} 4 \xrightarrow{K_{2 \circ 2}} 1$
	460896	$96 \xrightarrow{T_3} 34 \xrightarrow{K_2} 17 \xrightarrow{K_2} 9 \xrightarrow{K_3} 3 \xrightarrow{K_3} 1$
	6688128	$128 \xrightarrow{K_2} 64 \xrightarrow{K_2} 32 \xrightarrow{K_2} 16 \xrightarrow{K_2} 8 \xrightarrow{K_2} 4 \xrightarrow{K_{2 \circ 2}} 1$
	6960119	$119 \xrightarrow{K_2} 60 \xrightarrow{T_3} 22 \xrightarrow{K_2} 11 \xrightarrow{K_2} 6 \xrightarrow{K_2} 3 \xrightarrow{K_3} 1$
	8192128	$128 \xrightarrow{K_2} 64 \xrightarrow{K_2} 32 \xrightarrow{K_2} 16 \xrightarrow{K_2} 8 \xrightarrow{K_2} 4 \xrightarrow{K_{2 \circ 2}} 1$

Table 22: Runtime-vs-theory statistics for *full*, $1 \leq m \leq 128$.

Platform	ρ	r	MAE	RMSE	rel. MAE	MAPE
x86-64	0.9960	0.9918	70.99 ns	100.86 ns	7.26%	9.63%
ARM64	0.9986	0.9917	145.76 ns	196.37 ns	8.41%	16.38%
RISC-V64	0.9980	0.9990	262.09 ns	378.63 ns	2.52%	7.50%
RP2040	0.9998	0.9999	15688.85 ns	21040.03 ns	0.82%	1.47%

References

- [AAB⁺22] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, and Pascal Veron. HQC. Technical report, National Institute of Standards and Technology, 2022. Round 4 submission to the NIST Post-Quantum Cryptography Standardization Project. <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-4-submissions>.
- [Bel10] Richard Bellman. *Dynamic Programming*. Princeton University Press, 2010. Originally published in 1957. <https://doi.org/10.1515/9781400835386>.
- [BGTZ08] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster Multiplication in $\text{GF}(2)[x]$. In *Algorithmic Number Theory: 8th International Symposium, ANTS-VIII*, volume 5011 of *Lecture Notes in Computer Science*, pages 153–166, Berlin, Heidelberg, 2008. Springer. https://doi.org/10.1007/978-3-540-79456-1_10.
- [BL25] Daniel J. Bernstein and Tanja Lange. libmceliece, 2025. Accessed June 25, 2026. <https://lib.mceliece.org/>.
- [Bod07] Marco Bodrato. Towards Optimal Toom–Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0. In *Proceedings of the 1st International Workshop on the Arithmetic of Finite Fields (WAIFI 2007)*, volume 4547 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2007. https://doi.org/10.1007/978-3-540-73074-3_10.
- [CA69] Stephen A. Cook and Stål O. Aanderaa. On the Minimum Computation Time of Functions. *Transactions of the American Mathematical Society*, 142:291–314, 1969. <https://doi.org/10.1090/S0002-9947-1969-0249212-8>.
- [CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing BIKE for the Intel Haswell and ARM Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):97–124, 2021. <https://doi.org/10.46586/tches.v2021.i3.97-124>.
- [CCPY26] Ming-Shing Chen, Chun-Ming Chiu, Chun-Tao Peng, and Bo-Yin Yang. Accelerating HQC with Additive FFT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):520–544, 2026. <https://doi.org/10.46586/tches.v2026.i2.520-544>.
- [CDH⁺20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRU. Technical report, National Institute of Standards and Technology, 2020. Round 3 submission to the NIST Post-Quantum Cryptography Standardization Project. <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>.
- [CGKT22] Ming-Shing Chen, Tim Güneysu, Markus Krausz, and Jan Philipp Thoma. Carry-Less to BIKE Faster. In *Applied Cryptography and Network Security*, pages 833–852. Springer International Publishing, 2022. https://doi.org/10.1007/978-3-031-09234-3_41.

- [CK91] David G. Cantor and Erich Kaltofen. On Fast Multiplication of Polynomials over Arbitrary Algebras. *Acta Informatica*, 28:693–701, 1991. <https://doi.org/10.1007/BF01178683>.
- [CT65] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19:297–301, 1965. <https://doi.org/10.1090/S0025-5718-1965-0178586-1>.
- [DGK24] Nir Drucker, Shay Gueron, and Dusan Kostic. Additional Implementation of BIKE. GitHub repository, 2024. Commit f150a1c. <https://github.com/awslabs/bike-kem>.
- [DKR⁺20] Jan-Pieter D’Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. SABER. Technical report, National Institute of Standards and Technology, 2020. Round 3 submission to the NIST Post-Quantum Cryptography Standardization Project. <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>.
- [FJ05] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. <https://doi.org/10.1109/JPROC.2004.840301>.
- [GK92] Torbjörn Granlund and Richard Kenner. Eliminating Branches Using a Super-optimizer and the GNU C Compiler. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, PLDI ’92*, pages 341–352. ACM, 1992. <https://doi.org/10.1145/143095.143146>.
- [HQC25] HQC Team. HQC Code Repository, Version 5.0.0, 2025. Release v5.0.0, commit f46e5422, accessed June 21, 2026. <https://gitlab.com/pqc-hqc/hqc/-/tags/v5.0.0>.
- [Int22] Intel Corporation. *Intel Processors and Processor Cores Based on Golden Cove Microarchitecture: Instruction Throughput and Latency README*, March 2022. Revision 2.0.
- [JLK⁺26a] Jihoon Jang, Myeonghoon Lee, Suhri Kim, Seog Chung Seo, and Seokhie Hong. Faster GF(2)[x] Polynomial Multiplication Using SIMD Instructions. *Journal of Cryptographic Engineering*, 16(1), 2026. <https://doi.org/10.1007/s13389-026-00392-0>.
- [JLK⁺26b] Jihoon Jang, Myeonghoon Lee, Donggeun Kwon, Seokhie Hong, Suhri Kim, and Sangjin Lee. Efficient Polynomial Multiplication for HQC on ARM Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(4):998–1021, 2026. <https://doi.org/10.46586/tches.v2026.i4.998-1021>.
- [KO62] Anatolii A. Karatsuba and Yuri P. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akademii Nauk SSSR*, 145:293–294, 1962. English translation in *Soviet Physics Doklady* 7 (1963), 595–596. <https://www.mathnet.ru/eng/dan26729>.
- [KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. *Cryptology ePrint Archive*, Report 2019/844, 2019. <https://eprint.iacr.org/2019/844>.

- [LD00] Julio López and Ricardo Dahab. High-Speed Software Multiplication in $\mathbb{F}_2^m$. In *Proceedings of the First International Conference on Progress in Cryptology, INDOCRYPT '00*, pages 203–212, Berlin, Heidelberg, 2000. Springer-Verlag. https://doi.org/10.1007/3-540-44495-5_18.
- [LMR24] Zhengyang Liu, Stefan Mada, and John Regehr. Minotaur: A SIMD-Oriented Synthesizing Superoptimizer. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2), 2024. <https://doi.org/10.1145/3689766>.
- [Mas87] Henry Massalin. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS II*, pages 122–126. ACM, 1987. <https://doi.org/10.1145/36206.36194>.
- [Mon05] Peter L. Montgomery. Five, Six, and Seven-Term Karatsuba-Like Formulae. *IEEE Transactions on Computers*, 54(3):362–369, March 2005. <https://doi.org/10.1109/TC.2005.49>.
- [PMJ⁺05] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005. <https://doi.org/10.1109/JPROC.2004.840306>.
- [Pol71] John M Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of computation*, 25(114):365–374, 1971. <https://doi.org/10.1090/S0025-5718-1971-0301966-0>.
- [Ras24] Raspberry Pi Ltd. *RP2040 Datasheet: A Microcontroller by Raspberry Pi*, 2024. <https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf>.
- [RV23] Jean-Marc Robert and Pascal Véron. Faster Multiplication over $\mathbb{F}_2[X]$ Using AVX512 Instruction Set and VPCLMULQDQ Instruction. *Journal of Cryptographic Engineering*, 13(1):37–55, 2023. <https://doi.org/10.1007/s13389-021-00278-3>.
- [SCC⁺17] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A Synthesizing Superoptimizer, 2017. <https://doi.org/10.48550/arXiv.1711.04422>.
- [SKE⁺21] Hwajeong Seo, Hyeokdong Kwon, Siwoo Eum, Kyungbae Jang, Hyunjun Kim, Hyunji Kim, Minjoo Sim, Gyeongju Song, and Wai-Kong Lee. All the Polynomial Multiplication You Need on RISC-V. Cryptology ePrint Archive, Report 2021/1117, 2021. <https://eprint.iacr.org/2021/1117>.
- [SSA13] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13*, pages 305–316. ACM, 2013. <https://doi.org/10.1145/2451116.2451150>.
- [Sto19] Ko Stoffelen. Efficient Cryptography on the RISC-V Architecture. Cryptology ePrint Archive, Report 2019/794, 2019. <https://eprint.iacr.org/2019/794>.
- [Too63] Andrei L Toom. The Complexity of a Scheme of Functional Elements Simulating the Multiplication of Integers. In *Doklady*

- Akademii Nauk*, volume 150, pages 496–498. Russian Academy of Sciences, 1963. <https://web.archive.org/web/20061014211747/http%3A/de.ufpe.br/~toom/articles/rusmat/MULT-R.PDF>.
- [WPD01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001. [https://doi.org/10.1016/S0167-8191\(00\)00087-9](https://doi.org/10.1016/S0167-8191(00)00087-9).
- [ZZ26] Jipeng Zhang and Jiaheng Zhang. Vectorized Falcon-Sign Implementations using SSE2, AVX2, AVX-512F, NEON, and RVV. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(1):302–324, 2026. <https://doi.org/10.46586/tches.v2026.i1.302-324>.