Enabling Microarchitectural Agility: Taking ML-KEM & ML-DSA from Cortex-M4 to M7 with SLOTHY

Amin Abdulrahman
amin@abdulrahman.de
Max Planck Institute for Security and Privacy (MPI-SP)
Bochum, Germany

Matthias J. Kannwischer
matthias@kannwischer.eu
Quantum Safe Migration Center, Chelpis Quantum Corp
Taipei, Taiwan

Thing-Han Lim
han.lim@chelpis.com
Quantum Safe Migration Center, Chelpis Quantum Corp
Taipei, Taiwan

ABSTRACT

Highly-optimized assembly is commonly used to achieve the best performance for popular cryptographic schemes such as the newly standardized ML-KEM and ML-DSA. The majority of implementations today rely on hand-optimized assembly for the core building blocks to achieve both security and performance. However, recent work by Abdulrahman et al. takes a new approach, writing a readable assembly implementation first and leaving the bulk of the optimization work to a tool named SLOTHY based on constraint programming. SLOTHY performs instruction scheduling, register allocation, and software pipelining simultaneously using constraints modeling the (micro-)architectural details of the target platform.

In this work, we extend SLOTHY and investigate how it can be used to migrate already highly hand-optimized assembly to a different microarchitecture, while maximizing performance. As a case study, we optimize state-of-the-art Arm Cortex-M4 implementations of ML-KEM and ML-DSA for the Arm Cortex-M7.

Our results suggest that this approach is promising: For the number-theoretic transform (NTT) – the core building block of both ML-DSA and ML-KEM – we achieve speed-ups of 1.97× and 1.69×, respectively. For Keccak– the permutation used by SHA-3 and SHAKE and also vastly used in ML-DSA and ML-KEM – we achieve speed-ups of 30% compared to the M4 code and 5% compared to hand-optimized M7 code. For many other building blocks, we achieve similarly significant speed-ups of up to 2.35×. Overall, this results in 11 to 33% faster code for the entire cryptosystems.

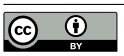
CCS CONCEPTS

• **Security and privacy → Public key (asymmetric) techniques**.

KEYWORDS

ML-KEM, ML-DSA, Arm Cortex-M7, pqm4, Constraint Solving

ACM Reference Format:
Amin Abdulrahman, Matthias J. Kannwischer, and Thing-Han Lim. 2025. Enabling Microarchitectural Agility: Taking ML-KEM & ML-DSA from Cortex-M4 to M7 with SLOTHY. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3708821.3736210



ASIA CCS '25, August 25–29, 2025, Hanoi, Vietnam

ACM ISBN 979-8-4007-1410-8/2025/08
https://doi.org/10.1145/3708821.3736210

1 INTRODUCTION

Modern implementations of cryptographic algorithms strive to achieve the best possible performance on a given platform, while at the same time being protected against timing side-channels or more powerful attacks. For these reasons, handwritten assembly is commonly employed in cryptographic implementations. However, this comes at the cost of the implementation being less portable as it is often specialized for a certain platform. If such specialization includes heavy microarchitectural optimization, it often results in relatively poor performance on different microarchitectures.

Recent work by Abdulrahman et al. [2] proposes an alternative path: A readable 'clean' assembly implementation is written first which is then optimized by a super-optimization tool performing register allocation, instruction scheduling, and software pipelining. The tool proposed in their work – named SLOTHY– uses constraint-solving (instantiated with CP-SAT from Google's OR-Tools [42]) to model both the architectural and microarchitectural details. It has been used to transform clean implementations targeting the Arm Neon and Arm Helium instruction sets into fast implementations for two different microarchitectures each. All the resulting implementations exceed or match the performance of their hand-optimized assembly counterparts.

However, the approach taken in [2] currently relies on a clean implementations written by the authors, requiring full control over the structure of the input source code. Thus, it remains an open question whether their tool can also be applied to existing hand-optimized implementations. In particular, in this work we study if heavily hand-optimized assembly for one microarchitecture can be automatically optimized for a different microarchitecture and yield satisfactory performance.

A prime example of such heavily hand-optimized assembly are implementations of the recently published post-quantum cryptography (PQC) standards ML-KEM and ML-DSA which were the results of a multi-year standardization effort by the US National Institute of Standards and Technology (NIST). Early on in the process, NIST asked the submission teams as well as the community to study high-speed software implementations – which in most cases resulted in handcrafted assembly implementations. A particularly popular platform for PQC is the Arm Cortex-M4 – a 32-bit microcontroller. It was also recommended by NIST as the primary microcontroller optimization target [5] sparking a vast number of research projects studying the performance of various post-quantum algorithms. The pqm4 library [30] compiles most of the state-of-the-art PQC implementations for the Arm Cortex-M4 into a single repository and provides a common benchmarking framework.

Both ML-KEM [3, 6, 14, 26, 45] and ML-DSA [3, 22, 23, 25] were the subject to a series of research papers studying highly efficient Arm Cortex-M4 implementations. At the same time, with each new publication and its implementation, the number of hand-optimized routines commonly increased resulting in a myriad of functions written in assembly for both ML-KEM and ML-DSA. In total, state-of-the-art implementations of ML-KEM and ML-DSA make use of 3633 and 3009 lines of assembly, respectively. As this code was written for the Arm Cortex-M4 microarchitecture implementing the Armv7E-M architecture, the code can also run on other microarchitectures implementing the same or newer versions of the architectures. For example, the Arm Cortex-M7 also implements the Armv7E-M architecture, while the Cortex-M33, and Cortex-M35P implement the newer Armv8-M. Consequently, all these cores can run the optimized Cortex-M4 implementations. However, whether those implementations perform well is unclear.

A particularly interesting microarchitecture for our case study is the Arm Cortex-M7[1] as it implements the exact same instruction set as the Cortex-M4 and it is, hence, conceivable that optimal Cortex-M4 code also performs well on the Cortex-M7. Yet, in reality, this is not the case and Cortex-M4 code often performs poorly on the Cortex-M7. This is primarily due to the Arm Cortex-M7 being a dual-issue central processing unit (CPU) with a substantially more complex pipeline design than the Cortex-M4. For certain well-scheduled workloads, the Arm Cortex-M7 can execute 2 instructions per cycle (IPC). Code not optimized for the dual-issuing capability often runs at only 1 IPC and, thus, performance often falls short by a factor of two or even more. While optimal performance of 2 IPC is not always achievable, careful scheduling respecting the latencies of the individual instructions and restrictions concerning which instructions can execute in parallel can result in much faster code.

In this work, we undertake a case study, evaluating if SLOTHY can be used to aid with the process of migrating hand-optimized assembly code between different microarchitectures while maximizing the performance with ML-KEM and ML-DSA as examples. We chose to pick the Arm Cortex-M4 and Cortex-M7 as our hardware targets because they (a) offer a wide variety of hand-optimized assembly routines, (b) share the same instruction set architecture (ISA), and (c) differ significantly in their microarchitectural properties. As we integrate our work directly into the SLOTHY tool, it can be used for any (micro-)architecture already supported by SLOTHY, and further (micro-)architectures can be added easily.

Our contributions are the following:

- We extend the SLOTHY superoptimizer to support a much broader class of assembly programs. This includes extending the loop capabilities, resulting in much more powerful software pipelining than before. We also augment SLOTHY to automatically replace specific instruction patterns that may perform well on one microarchitecture but not on another.
- We add support for the Armv7-M architecture to SLOTHY, complementing existing support for AArch64 and Armv8-M. Alongside this, we provide a microarchitectural model of the Cortex-M7. As Arm does not provide any documentation on this matter, we base our model on our own experiments as well as on 3rd party reverse engineering.
- We present a Keccak implementation that is 30% faster than a hand-optimized Cortex-M4 implementation running on the M7 and 5% faster than the existing hand-optimized Cortex-M7 [4] assembly.
- We apply SLOTHY to all assembly routines present in state-of-the-art ML-KEM and ML-DSA implementations. Besides Keccak which is dominating the performance of both ML-KEM and ML-DSA, this covers scheme-specific functions such as the number-theoretic transforms. For the vast majority of sub-routines, we get significant performance improvements. We achieve up to 2.3× performance improvement for certain sub-routines, resulting in full-scheme speed-ups of up to 32% for ML-DSA, and up to 27% for ML-KEM when comparing to implementations from pqm4.

Code. Our modifications to SLOTHY (integrated into its source tree) are available at https://github.com/slothy-optimizer/slothy under MIT license. Our benchmarking framework pqm7 can be found at https://github.com/mupq/pqm7.

Related Work. The literature that relates to our work can be divided into two groups. On the one hand, there exists a vast amount of prior work from the domain of software optimization and superoptimization. Work that considers superoptimization that may also take the selection of instructions into account has been studied for a long time [28, 37]. More recently, [44] presents a superoptimizer operating on LLVM IR level, making use of satisfiability modulo theories (SMT) solving to find missing opportunities for replacement patterns. [31] presents "CryptOpt", a tool that produces formally verified high-speed cryptographic code for x86-64 CPUs using random program search. Techniques from the domain of deep learning are employed by the authors of [35], who formalize the search for fast sorting algorithms as a single-player game, being played by a deep reinforcement learning agent. The area of superoptimization through constraint programming or Integer Linear programming (ILP) that excludes the instruction selection has received lots of interest in the 1990s, for example in [7, 8, 20]. The most recent and most relevant publication to this work is [2], presenting the tool SLOTHY, which makes use of constraint programming to simultaneously address instruction scheduling, register allocation, and software pipelining. It has been applied to several examples from the domain of cryptographic software.

The other group of related literature concerns high-speed implementations of PQC on embedded devices – most notably the Arm Cortex-M4 microcontroller. For ML-KEM, an initial implementation was provided by the Kyber submission team [45]. Later, Botros et al. [14] wrote a faster implementation focusing on the performance of the number-theoretic transform. Alkim et al. [6] improved this implementation by proposing a faster Montgomery multiplication. The implementation was further improved by Abdulrahman et al. [3]. Most recently, Huang et al. [26] proposed a yet faster NTT using Plantard multiplication. ML-DSA has a similar history of heavy hand-optimization on the Cortex-M4. An initial implementation was proposed by Krausz et al. [23] which was later improved by Greconici et al. [22]. This was again improved by Abdulrahman et al. [3], with the most recent performance improvements being

[1]The Cortex-M7 is a popular microarchitecture, with commercial chips available from all major vendors. However, it has not been as well studied as the Cortex-M4 in the cryptography literature yet.

part of [25] by Huang et al. A first evaluation of PQC schemes on an Arm Cortex-M7 CPU has been done in [24], targeting the round-3 variants of Dilithium and Falcon. The initial implementation of Keccak used within most PQC implementations on the Arm Cortex-M4 is provided by [17] and already tailored to Armv7-M. It was improved by [4], applying the lazy rotation technique from [11] to the Cortex-M4, while also introducing hand-optimized Keccak code for the Cortex-M7.

Structure. We structure our paper by starting off with a description of relevant background information on PQC, software optimization, and the tool SLOTHY in Section 2. We move on to a description of our extensions to SLOTHY in Section 3, and continue explaining our case study using the modified tool with workloads from the PQC-domain in Section 4. Following, we present the results of our efforts in Section 5 before concluding our work in Section 6.

2 PRELIMINARIES

In this section, we introduce the relevant background on the cryptographic algorithms (ML-KEM, ML-DSA, and Keccak) and introduce the Arm Cortex-M7 microarchitecture and compare it to the much more popular Arm Cortex-M4 microarchitecture. We follow the notation used in the NIST PQC standards [39, 40].

2.1 ML-KEM

The key encapsulation mechanism (KEM) Kyber [13] has been specified by NIST in Federal Information Processing Standard (FIPS) 203 [39] under the name ML-KEM. Its IND-CCA2 property is guaranteed by applying a tweaked Fujisaki-Okamoto (FO) transform to the underlying IND-CPA secure public-key encryption (PKE) scheme. The hardness of the scheme is based on the module learning with errors (MLWE) problem, which allows to scale the security level by varying the module rank k. The three available security levels are called ML-KEM-512, ML-KEM-768, and ML-KEM-1024. ML-KEM operates over the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, where n is 256 and the modulus q is the 12-bit prime 3329 making polynomial arithmetic a core part of the scheme.

2.2 ML-DSA

Similarly, the digital signature scheme Dilithium [19] has been standardized in FIPS 204 [40] and named ML-DSA. Just as ML-KEM, ML-DSA relies on the MLWE problem. In addition, it is based on a variant of the module short integer solution (MSIS) problem. Its construction is based on the Fiat–Shamir with aborts pattern [34] and believed to fulfill the SUF-CMA security property – even against quantum adversaries. The scheme operates over a similar polynomial ring as ML-KEM, namely $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $n = 256$ but a larger 23-bit q chosen as 8380417. The security level can be adjusted by varying the lattice dimensions k and ℓ leading to the three security levels ML-DSA-44, ML-DSA-65, and ML-DSA-87.

2.3 Polynomial Arithmetic

With both, ML-KEM and ML-DSA operating over the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, polynomial arithmetic is a core part of the schemes. Especially polynomial multiplications are generally expensive; their naive implementation using the school-book method comes with a time complexity of $O(n^2)$. To counteract this inefficiency, the specification of both schemes prescribes the use of the number-theoretic transform (NTT), a variant of the discrete Fourier transform (DFT) defined over finite fields. After transforming the input polynomials into the NTT domain, the polynomial multiplication comes down to a pointwise multiplication with complexity of $O(n)$ followed by an inverse number-theoretic transform (INTT) to retrieve the result: $f \cdot g = \mathsf{iNTT}(\mathsf{NTT}(f) \circ \mathsf{NTT}(g))$, where $f, g \in R_q$. Key to the efficiency is that the transformation itself can be computed efficiently in $O(n \log n)$ time using a fast Fourier transform (FFT) algorithm, shifting runtime away from the actual multiplication towards the switch to a different domain. Thus, the NTT and its inverse are the most critical and also most costly operations when considering the performance of the polynomial arithmetic in ML-KEM and ML-DSA. Note that in the case of ML-KEM, where only $n \mid (q-1)$ but not $2n \mid (q-1)$, the NTT is called "incomplete" and thus, the multiplication inside NTT-domain amounts to the multiplication of linear polynomials over $\mathbb{Z}_q$. The umbrella-term to refer to both, this, and the pointwise multiplication inside NTT-domain is called "base multiplication", or short basemul.

In order to understand some of our performance results, it is crucial to revisit the core operation, that is part of the FFT-based NTT and INTT algorithms: The "butterfly" operation. It comprises one addition, one subtraction, and one modular multiplication with a constant. Figure 1 depicts the most common types, the Cooley–Tukey (CT) [16] and Gentleman–Sande (GS) [21] butterflies, and can be found in Appendix A.

2.4 Keccak

Next to the polynomial arithmetic being a core part of ML-KEM and ML-DSA, both schemes make heavy use of hashing using functions from the sponge-construction based SHA-3 family [38]. It comprises the fixed-length functions SHA3-{224, 256, 384, 512} as well as the extended output functions (XOFs) SHAKE, denoted by SHAKE-{128, 256}. Both the fixed-length functions and the XOFs are based on the Keccak permutation [12], more precisely the Keccak-f1600 function that operates on a 1600 bit state over a course of 24 rounds, where in each round, a sequence of five transformations is applied: $(\theta, \rho, \pi, \chi, \iota)$.

All of these five steps mostly consist of bitwise logical operations, such as the "exclusive or" XOR, or rotations. As we abstain from making any algorithmic changes to the hash function's implementation itself, we omit a more detailed description of the algorithm except for mentioning one optimization technique that will be relevant for the interpretation of our results.

Lazy Rotations. The concept of lazy rotations was introduced in [11]: The idea behind it is to trade a larger number of explicit rotations for fewer in a later step of the Keccak permutation, as well as a number of inline barrel shifts applied to the second operand of logical instructions, that come at no performance penalty on many Arm CPUs (e.g., Cortex-M4, many AArch64 cores). This concept was applied to Armv7-M in [4], where the author shows performance gains for the Arm Cortex-M4, but conjecture that the approach will be detrimental to the performance on Arm Cortex-M7 due to its different pipeline structure.

2.5 Arm Cortex-M7 and Arm Cortex-M4

The Arm Cortex-M family is a series of microcontroller CPUs designed as low-power embedded devices. Both, the Cortex-M4 and Cortex-M7, implement the Armv7-M ISA with the additional digital signal processor (DSP) extension. The combination is referred to as Armv7E-M. Armv7-M offers sixteen 32-bit general-purpose registers (GPRs), `r0` to `r15`, where `r13` is designated as the stack pointer, `r14` as the link register, and `r15` as the program counter. In addition, there is potentially a floating point unit (FPU), which adds 32 floating-point registers (FPRs) `s0` to `s31` with a width of 32 bits each. While on both CPUs the presence of the FPU is optional, most commercially available Cortex-M7 devices implement it.

Some notable features that are shared between both microarchitectures are:

Barrel Shifter The barrel shifter allows shifting or rotating the second input to many logical and arithmetic instructions before the operation is executed without any additional latency to the instruction.

Thumb-2 The Thumb-2 instruction set allows for a more compact code size by enabling a shorter 16-bit encoding for some instructions. While this feature aids with code size, it may cause performance penalties due to accesses to the instruction memory not being 4-byte aligned.

In the following, we will introduce some key differences between the two microarchitectures.

2.5.1 Arm Cortex-M4. The Arm Cortex-M4 offers a simple 3-stage pipeline with most instructions taking a single clock cycle, except branches and load instructions, which may take longer [9]. The load instruction `ldr` typically takes two clock cycles to complete, however, n subsequent `ldr` instructions can pipeline their address and data phases taking only $n + 1$ clock cycles to complete.

2.5.2 Arm Cortex-M7. The Arm Cortex-M7 has a more complex 6-stage pipeline and offers dual-issuing capabilities [18]. This means that the Cortex-M7 can issue two instructions in every clock cycle. In contrast to the Arm Cortex-M4, no detailed information about the CPUs's performance characteristics is available by Arm itself. However, attempts to reverse-engineer the pipeline structure and to gather details about the Cortex-M7 have been made by independent individuals [27, 41]. General information about the pipeline structure was revealed in [18], which we will describe here, while we defer the exact microarchitectural specifics to Section 3.1.

The most important takeaways from [18] are:

- There are two 32-bit load-pipes.
- There are two arithmetic logic unit (ALU)-pipes, where only one is capable of barrel shifting (`ALU0`), while the other one, `ALU1`, is "skewed".
- There is a single multiply-accumulate (MAC)-pipe capable of computing a $32 \times 32 + 64$ bit product per cycle.
- There is a single 64-bit store-pipe (skewed).
- There is one FPU-pipe for ALU operation, and another one capable of multiplication and division.

For a pipe to be considered "skewed", it means that an instruction can be sent to such pipes in an earlier stage of the execution, making its result also available one stage earlier than usual, allowing another pipe to consume it within the same cycle.

2.6 Software Optimization

We introduce the background on automated software optimization underlying the SLOTHY superoptimizer that we extend in this work. When optimizing the performance of a piece of code, there are multiple different aspects to take into account.

Instruction Selection. In almost any case, there exist multiple ways to implement the same functionality using a different sequence of instructions. Compilers commonly start this process from a higher level language that gets translated into assembly, oftentimes relying on heuristics to deliver acceptable results with a restricted time budget. For high-speed implementations, developers frequently write assembly code themselves and, doing so, handle the instruction selection on their own. Due to a more comprehensive understanding of the optimization target's semantic and a less restrictive time budget, humans can often find better-suited sequences than compilers do. In addition, by hand-writing the code, developers can ensure security properties such as secret-independent timing – those are crucial for cryptographic implementations, but not enforced by most compilers.

Instruction Scheduling. The order in which instructions are scheduled on a CPU can have a significant impact on the performance. Reasons for this may be an instruction's latency characteristics or throughput limitations. Especially on processors with multiple execution units, a sensible ordering of instructions can lead to a higher utilization of the available resources, and thus, faster code. Even though out-of-order (OoO) execution is a common feature in powerful CPUs that allows for ad-hoc instruction rescheduling, there exists a large set of microarchitectures for which the order within the source still matters, especially in-order microarchitectures. *Fixed-instruction superoptimization* aims at finding an optimal instruction scheduling without altering the set of instructions employed.

Register Allocation. The number and size of available registers determines how much data can be kept at hand, avoiding spills to memory. The choice of register allocation also impacts the flexibility of instruction re-scheduling. *Register renaming* is the process of changing the choice of register allocation, and is typically considered alongside instruction (re-)scheduling. OoO microarchitectures conduct both on the fly; on in-order microarchitectures, however, they have to be done in software.

Software Pipelining. As introduced in [32, 43], software pipelining is a technique to overlap two or more iterations of a loop in order to aid with data dependencies or exhaustion of processing resources and to increase instruction level parallelism. It is typically applied when a given loop iteration cannot progress and a stall would occur. Then, instructions from the next iteration may be issued in the current iteration, provided they are independent of the current iteration's data stream. A limiting factor to this technique is the size of the register file: More parallelism can only be introduced if there are enough registers available to hold the data required for the additional instructions.

2.6.1 SLOTHY. The tool SLOTHY [2] automates the process of instruction scheduling, register allocation, and software pipelining. It does so by constructing a constraint-programming problem

based on the input assembly and models of the target architecture and microarchitecture. Notably, it considers instruction scheduling, register allocation, and software pipelining simultaneously. SLOTHY deploys CP-SAT from Google's OR-Tools [42] to solve the constraint problem, yielding optimal solutions based on the provided microarchitectural model.

Why SLOTHY. In this work, we decide to use and extend SLOTHY as a starting point for our migration process for multiple reasons. Given the large amount of hand-optimized Cortex-M4 code for ML-KEM and ML-DSA, automated optimization for a different microarchitecture promises fast implementations with limited manual effort. First, we deem the scheduling of instructions, as well as the allocation of registers as the most crucial parts of the optimization process for the Arm Cortex-M7. We argue that the selection of instructions – which SLOTHY is incapable of – is a task humans generally excell at, especially on simpler reduced instruction set computer (RISC) architectures like Armv7E-M. In particular, the routines of ML-KEM and ML-DSA we care about have been steadily improved in the aforementioned publications and are now readily available as a starting point for our optimization process. Second, we believe that many of the sub-routines we will consider are amenable to a complete search, i.e., finding an optimal solution based on the given microarchitectural model.

Prerequisites. For SLOTHY to be able to run on a certain piece of assembly code, the user is required to provide two models: The *architectural model* for the target ISA is used to parse the source code, provide basic information about the ISA, and to deliver information about the instructions' data flow, declaring inputs and outputs. In addition, SLOTHY requires a *microarchitectural model* in order to represent the target CPU's intricacies in terms of latencies, throughput, use of execution units, and other performance-relevant characteristics such as forwarding paths, hazards, or slot restrictions. Note that both models can be built lazily, meaning that only the instructions and features that are relevant for the optimization process need to be defined.

Heuristics. As it was noted in the original paper on SLOTHY [2, Section 7.4], there exist certain limits to the complexity of the optimization problems that SLOTHY can solve optimally. These limits depend on a number of different factors, e.g., the number of instructions, the register pressure, the complexity of additional constraints, esp. for data or structural hazards. To counteract this limitation, SLOTHY offers multiple different types of heuristics. Most importantly, the "splitting heuristic", where SLOTHY only considers a (small) "sliding window" of instructions at a time.

Address Offset Fixup. The address offset fixup is a feature of SLOTHY that tries to maximize the possibilities for re-ordering of load and store instructions that use an immediate offset to access memory. Oftentimes, it is possible to change the order of two load instructions using the same address register even if one of them does modify the address register. It works by first ignoring the exact offsets and then, after the optimization process, reconsidering them to semantically match the input again. This increases the flexibility in case one of the instructions increments the output pointer, overwriting it and thus, posing as a natural barrier for reordering of other memory operations to the same pointer. Ignoring the offsets is implemented by removing the address register as an output from load or store instructions that would increment the address, thus, removing the dependency on the address register and allowing for more freedom in the scheduling.

3 EXTENDING SLOTHY

We provide several extensions and improvements to SLOTHY to enable and aide its use for automatic migration of code from Cortex-M4 to Cortex-M7.

3.1 New Models

First, we add support for the Armv7E-M architecture to SLOTHY. This is necessary for SLOTHY to parse the input code from the pqm4 library and to generate the output code for the Cortex-M7.

Next, we build a microarchitectural model for the Cortex-M7, which is a prerequisite for the optimization process. In the following, we summarize the state-of-the-art public knowledge about the Cortex-M7 pipeline details and its performance characteristics that are relevant to the examples we consider in this work. We also describe how to model these constraints in SLOTHY. The following microarchitecture details are based on our own microbenchmarks and we point to prior third party profiling [18, 27] confirmed by our benchmarks.

We initially experimented with deriving a model from the microarchitecture model present in LLVM.[2] Unfortunately, we found the LLVM model to be too inaccurate to obtain satisfactory results. For example, the model restricts issuing of store instructions to one of the two issue slots, which we have found to be incorrect in our microbenchmarks and also contradicts the findings in [27].

ALU. The CPU offers two ALU units meaning that most operations on the ALU can dual-issue and complete with a latency of 1 cycle. One ALU offers a Barrel shifter at an earlier stage, and instructions that use this barrel shifter require the shifted operand to be available one cycle earlier than usual. Modeling this in SLOTHY is straightforward, as the routine to infer the latency called during construction of the constraint model receives the source and destination instructions as inputs.

Multiplications. The Cortex-M7 has a single MAC unit, so multiplications do not dual-issue. The latency of multiplications is 2 cycles, except for MAC-chains, where the latency into the accumulator is reduced to 1. We model this in SLOTHY by setting the default latency to 2 cycles, and adding a special case for cases where both the consumer and producer of a value are MAC instructions. An additional restriction for multiplications is that they cannot dual-issue with store instructions. In SLOTHY, we model this by making store instructions also occupy the MAC unit.

Bit-Field & DSP. Some bit-field instructions like `pkhtb`, `pkhbt`, and `ubfx` can only be issued on the ALU unit with the barrel shifter. Moreover, neither bit-field, nor DSP instructions can dual-issue with respect to each other. We express this in SLOTHY's model by adding a "fictional" execution unit for DSP instructions, which is used in addition to an ALU unit for the respective instructions.

[2] https://github.com/llvm/llvm-project/blob/78a871abf7018f4a288b773c9c89f99cd5c66b9c/llvm/lib/Target/ARM/ARMScheduleM7.td

Memory. By default, loads have a latency of 2 clock cycles. However, due to the skewed design of the pipes (see Section 2.5.2) the latency is reduced to just one cycle, in case the destination is an arithmetic or logical instruction without a barrel shift. With two load units, `LOAD0` and `LOAD1`, they can even dual-issue – but only if they target different memory banks, i.e., even and odd indexed words. We model this using the following "best-effort" approach: We assume that all pointers are aligned to 8 bytes. Then, based on the immediate offset inside the load instruction, we evaluate which of the memory banks is targeted by assessing whether the index of the word is even or odd. If the index of the word is even, we assign the load to the `LOAD0` unit, otherwise to the `LOAD1` unit. Instructions that load or store n words at a time have a latency of $n + 1$ clock cycles. Thanks to the skewed store-pipe, store instructions can store results with zero cycles latency, meaning within the same clock cycle. To evade load-after-store hazards, we augment the model with a constraint that, for each time a load happens after a store within the same memory region, mandates an offset of 8 clock cycles between the two instructions. This value has proven itself empirically and accounts for, e.g., transferring elements from the store buffer into random access memory (RAM).

Floating-Point. The only floating-point instructions relevant to us – namely just different varieties of `vmov` – go down the FPU ALU pipe. For the variant moving the value of an FPR to a GPR, the latency is 1, while the `vmov` variants moving from a GPR to an FPR have a latency of 3.

3.2 Instruction Splitting & Fusion

When considering two different microarchitectures, there commonly exist patterns of instructions that perform well on one, but suboptimal on the other. For example, when it is desired to load multiple 32-bit words from memory, the `ldm` instruction can be used on an Armv7-M CPUs. On the Arm Cortex-M4, this instruction takes $n + 1$ clock cycles to load n words. This is exactly the same performance as one would get by issuing several individual `ldr` instructions in a row. On the Cortex-M7, however, using `ldm` may be detrimental to the performance even though a sequence of `ldr` instructions and the `ldm` would also take the same time to complete. This is due to the M7's dual-issuing capabilities, the Cortex-M7 can already start to compute, e.g., arithmetic operations on the output of one `ldr` from the sequence, while the other `ldr` instructions are still pending. This is not possible when using the `ldm` instruction, which would stall the pipeline until all words are loaded. To address this issue, we introduce a *splitting* and *fusion* feature to SLOTHY. This allows the merging of multiple instructions into fewer, or – more useful in our case – the splitting of one instruction into multiple. Using this feature, we can split `ldm` instructions into multiple `ldr` instructions, which can be scheduled in parallel to, e.g., additions, in the following code, making better use of the entirety of the CPU's resources. Let us consider the output of two SLOTHY optimization runs in Listing 1 for an illustration of the power of this feature. The input to both runs has been one `ldm` instruction loading eight registers, followed by eight `uadd16` instructions that add the register `r1` onto the loaded data. This would be a performance-wise valid approach on the Cortex-M4. Naively optimizing this code for the Cortex-M7 yields the output given in Listing 1a, which does not improve over the original input as there is no meaningful way to alter the scheduling. By enabling the splitting feature, SLOTHY can generate the code in Listing 1b, which splits the `ldm` instruction into multiple `ldr` instructions, which can be scheduled in parallel to the `uadd16` instructions. By considering the comment at the top of SLOTHY's outputs, we can see that the expected number of clock cycles based on our microarchitectural model has been reduced from 12 clock cycles to just 9 – a 33% improvement. We confirm this by benchmarking both snippets on-device. Each column in the comments next to the output source code represents one clock cycle. An asterisk symbolizes the issuing of the respective instruction, meaning two "stacked" asterisks indicate that two instructions are dual-issued.

A note on security: It's the developers responsibility to ensure that the fusion they add to SLOTHY is (a) semantically correct and (b) does not alter the program in a way that introduces side-channel leakage. If it would be desired to implement an "unsafe" replacement, a possible, future addition to SLOTHY could be the ability to mark registers as either public or secret, and to enforce that the fusion may not interact with secret registers in a way that would introduce leakage.

3.3 Re-Worked Loop Handling

SLOTHY supports the ability to parse loops in the input assembly in order to enable the use of software pipelining. This is necessary since the pre- and post-amble need to be sensibly embedded into the optimized code and modifications to the counter need to be made. However, SLOTHY has so far only been able to operate on loops that used a static counter register that gets decremented by one in each iteration using `subs` with a following conditional branch instruction `bne`. This instruction pair is subsequently not considered as part of the loop kernel, as the so called "loop boundary" which SLOTHY is blind to during the optimization process.

There exist two reasons why we decided to take on SLOTHY's abilities to detect and handle loops during the optimization process.

(1) While the authors of [2] could work with the aforementioned limitation in their "clean" code, we found that the code present in the pqm4 library makes use of a multitude of different structures for loops, which would not be recognized by SLOTHY. A prominent example for this is that the loop counter is not a dedicated register but the last iteration of the loop is instead inferred based on the value a pointer, which gets incremented with each iteration.
(2) When tuning code for the highest performance, every instruction counts. For this reason, hiding multiple instructions as part of the loop boundary from SLOTHY and thus, loosing opportunities for scheduling, limits performance gains. A logical consequence was to merge the loop boundary into the loop and to make it part of the constraint model that SLOTHY is building.

Every architectural model can now be extended with multiple different loop classes with SLOTHY picking a suitable one for the code at hand. The most general form of loop we add to the Armv7-M model only requires a start label as well as a branch instruction back to the label to be present in the input source. Based on that,

we extended SLOTHY to automatically infer details about the structure of the loop, enabling it to take all instructions, including the branch, into account when scheduling. The information we infer automatically, for example, includes by how much the loop counter is modified in each iteration. This is an important piece of information, as the loop counter needs to be modified accordingly ahead of time in case software pipelining is deployed and thus, one iteration of the loop gets "unrolled".

The performance advantage of this technique can be easily understood based on an example. While Arm Cortex-M7 has the capability to dual-issue instructions, it can only issue one multiplication in each clock cycle as there is only one execution unit for these available. However, many of the functions related to polynomial multiplications are dominated by instructions occupying the MAC unit, meaning that in case there are more multiplications than any other type of instruction, there will inevitably occur stalls due to resource hazards. Having one or more additional instructions from the loop boundary that can be used to balance out the multiplications can potentially save multiple cycles per iteration.

Limitation. Two limitations to the loop handling still remain. First, when making use of the address offset fixup feature in combination with a loop, that relies on a pointer that gets incremented using a load or store instruction, it is required to manually annotate the input source code to inform SLOTHY that the load or store instruction used for the increment needs to appear before the flag-setting instruction (e.g., `cmp`). The reason for this is that the address offset fixup removes the address-output from the load or store instruction to allow for the flexible reordering during the fixup. However, this also removes the dependency between the incrementing instruction and the flag-setting instruction, meaning they could be swapped – resulting in incorrect code. Second, the loop handling is not yet capable of determining the counter register on its own, we require the order of arguments to the `cmp` instruction to follow the convention of the first register being the one that gets incremented, while the second marks the end-value.

3.4 Further Changes

0-Latency. A new capability we added to SLOTHY is the ability to model instructions that have a latency of 0, which was unsupported before but is a feature for certain instructions on the Cortex-M7.

Assembly Directives. With respect to parsing, we also extended the support for assembly directives, adding the ability to resolve (recursive) `.if ... .else` directives and to interpret definitions made through the `.equ` directive.

3.5 Security Considerations

In order to rule out timing side channels in it's output, SLOTHY relies on the input code being constant-time to begin with. For example, memory accesses must be secret-independent and no variable time instructions may be used on secret data. For the latter, special care has to be taken when using the splitting and fusion feature: It is the microarchitectural model-developer's responsibility to ensure that the replacement patterns they define are constant-time on their target microarchitecture. All replacement patterns provided in this work fulfill this property. In case no splitting or fusion is applied, SLOTHY acts as a fixed-instruction superoptimizer. As no branches or additional memory accesses are introduced, the code will remain constant-time.

For, e.g., power side channels, the situation is different: In this case, leakage heavily relies on microarchitectural details of the pipeline. Such pipeline leakage commonly depends on the the ordering of the instructions as well as assignment of registers [36]. Thus, SLOTHY's optimization may introduce additional pipeline leakage. However, addressing this issue is out of scope for this work. As mentioned in [2, Section 5.1], a possible solution could involve modeling a device's leakage characteristics as part of SLOTHY's microarchitectural model and incorporating this information in the construction of the optimization problem.

4 CASE STUDY

In this section, we will describe the process of deploying SLOTHY, including our extensions, for the task of optimizing the ML-KEM and ML-DSA implementations for the Arm Cortex-M7. We will start by explaining our starting point for the process, giving context to the functions we are considering, and then describe the changes we made to the input assembly files to make them compatible with SLOTHY.

As a starting point for the optimization process, we use the assembly files that are present in the pqm4 library [30] (as of Jan 16, 2025, commit `49ce5bea`).

4.1 Target Routines

Next, we will introduce the routines that are subject to our optimization efforts. As the most time-consuming operations within both ML-KEM and ML-DSA are actually symmetric primitives based on the Keccak permutation, we start by looking at Keccak as even small speed-ups in Keccak often have more impact on the performance of the full scheme.

4.1.1 Keccak. Only a single assembly implementation of Keccak is part of pqm4 and it is shared among the other cryptosystems using it. Currently, it is an implementation specifically tuned for the Arm Cortex-M4, presented in [4], which is based on [17] adapting recent techniques from [11]. The implementation from [17] was previously used in pqm4. Next to these implementations targeting the Arm Cortex-M4, a hand-optimized implementation for the Arm Cortex-M7 was introduced in [4] as well.

In this work, we decide to play out one advantage of SLOTHY: The ability to swiftly evaluate how amenable different implementations of the same algorithm are to the optimization for a certain target platform. We, hence, evaluate all three implementations.

For all three implementations, we target the core of the algorithm, the Keccak-f permutation KeccakF1600_StatePermute. In the following, we will suffix the original implementation from [17] with "xkcp", the Cortex-M4-tuned implementation from [4] with "adomnicai_m4", and the Cortex-M7-tuned implementation from [4] with "adomnicai_m7".

4.1.2 ML-KEM. The assembly functions contained in pqm4 for ML-KEM can be split in three groups:

Polynomial Multiplication Most crucial to the polynomial multiplication are the functions for the NTT and INTT. They

are responsible for transforming between the NTT-domain and "regular" domain. Alongside these, we have a set of basemul functions targeting the multiplication inside the NTT-domain. Multiple variants to these exist: Functions that carry the prefix "frombytes", implicitly unpack the wire-representation of the polynomial before performing the multiplication itself. Functions containing either combination of $\{16, 32\}$ in their name belong to the speed-optimized implementation that makes use of the "better accumulation" strategy [15].

Polynomial Arithmetic Next to the polynomial multiplications, we consider functions for polynomial addition and subtraction poly_add and poly_sub. A function implementing the Barrett reduction is also part of the code.

Polynomial Sampling Functions prefixed with "matacc" aid with the acceleration of the sampling of the public matrix $A \in R_q^{k \times k}$ whilst multiplying it with a vector of polynomials on the fly. Again, functions carrying a suffix containing $\{16, 32\}$ are deploying a performance optimization pattern using caching in memory.

It is important to note that pqm4 contains both a stack- and a speed-optimized implementation. Some of the functions we target are only used in one of the two. For example, all sub-routines containing $\{16, 32\}$ are only used in the speed-optimized implementation, while the stack-optimized implementation uses a more stack friendly variant not using the "better accumulation".

4.1.3 ML-DSA. For ML-DSA, we can see a similar picture:

Polynomial Multiplication Routines computing the NTT, INTT, and basemul using the ML-DSA-prime 8380417 belong to the most performance critical routines and are thus subject to our optimizations. Proposed in [3], some of the transformations and basemuls in ML-DSA can be replaced by more efficient ones, making use of smaller moduli 257 or 769. In the case of the Fermat number 257, the transformation is then called Fermat number transform (FNT), instead of NTT. If a basemul routine contains the term "asymmetric", it is designed to make use of an optimization technique introduced in [10], which re-uses part of the result from the NTT/FNT during the basemul.

Other Routines Just as for ML-KEM, a routine for Barrett reduction is part of the implementation. In addition, the function caddq is used to transform polynomial coefficients that are reduced to a representative that may be negative, to one that is in the range $[0, q)$.

Similar as for ML-KEM, there is a speed- and a stack- optimized implementation. However, those use mostly the same assembly. Although it has been shown to be inferior in performance to the 769 arithmetic [26], we also consider the 257 arithmetic as it has been part of pqm4 up until very recently[3]. However, we exclusively deploy the 769 arithmetic for our full scheme implementations.

4.2 Changes to Input Assembly

Although large parts of the transition to the Cortex-M7 are automated, some manual changes to the input assembly files have been necessary or helpful with easing the process. We deem all of these changes as minor but still explain them in the following.

First, we split files from pqm4 in such a way, that always only one global function remains per file. This eases experimentation across multiple optimization runs. Further, we renamed some functions and symbols in the input assembly to avoid naming collisions or to make the function names more descriptive.

In rare cases, the LLVM assembler, used for a self-test feature of SLOTHY[4], would not accept the "short" notation of an instruction being forced into the 32-bit wide encoding, e.g., `add.w r0, #<imm>` would cause an error, while `add.w r0, r0, #<imm>` passed just fine. In these cases, we manually expanded the short notation into the long one. The other way around, sometimes, when giving the "long" notation of an instruction and simultaneously forcing it to the 32-bit encoding would cause an error as well, e.g., `neg.w r0, r0` would fail, while `neg r0, r0` would work while still being encoded as a 32-bit instruction.

The majority of changes are due to limitations in SLOTHY's abilities of parsing the assembly source and in order to circumvent major engineering efforts. We had to rename some constants defined through the `.equ` directive, as SLOTHY does not support overwriting these in case they get re-defined within a function. Also, SLOTHY does not support parsing macros which take other macro names as arguments. We resolved this by directly invoking the appropriate macro. Further, we sometimes switched the order of arguments in the `cmp` instruction at the end of a loop to allow SLOTHY to automatically infer the loop structure as we identify the counter register based on the ordering of the arguments. As a last point on the parsing, we removed the use of the `.rept` directive from three input files as it (a) complicates the parsing, esp. for general cases in combination with other directives, (b) does not add much value, and (c) is trivial to replace (i.e., just deleting the directive). Note that this change makes the input code the slightest bit less performant as the inlining spared some loop handling (e.g., comparisons and branches), while at the same time reducing the code size as a side effect.

Further, we added SLOTHY tag annotations to the input assembly files to give certain hints to SLOTHY, e.g., about the interaction with the memory or about the loop structure. Lastly, we push more callee-save registers to the stack at the beginning of some functions – in case not all have been pushed anyways – in order to avoid restrictions with respect to renaming and unlock more flexible scheduling.

4.3 SLOTHY Configuration

After integrating the input assembly files into SLOTHY, the optimization process could be started. This requires to provide a configuration on the parameters used during optimization – at least providing the target platform, input file, and the region or loop of the file to be optimized.

However, there are several configuration options that may impact the performance. In the following, we go over the performance-relevant parts of the configuration for the critical routines.

[3]It was changed in `1a04a91`

[4]The self-test feature within SLOTHY assembles the code before and after optimization and executes it in the Unicorn emulator on random inputs to check that it produces the same outputs. While using this feature is not strictly necessary, it is very helpful in finding mistakes in the architecture model and SLOTHY itself as early as possible.

4.3.1 ML-DSA NTT & INTT. For the NTT and INTT in ML-DSA using q as 8380417, the number of instructions per loop varies between 33 and 86 and we are able to optimize all of the loops without the use of heuristics and with software pipelining enabled, terminating within approximately one hour (on an Apple M1).

In contrast, the implementation of the FNT and its inverse contain some loops that cover up to 407 instructions, which is beyond SLOTHY's capabilities for optimization without the use of the splitting heuristic. We use a splitting factor between 6 and 8, with a step size between 10 and 15%. In addition, we set a timeout of 180 seconds that interrupts the solver and continues with the solution found at that point. Further, we make use of our new fusion-feature in order to split `vldm` instructions, loading multiple FPRs at once, into a sequence of `vldr` instructions.

As the 769 NTT and INTT are based on to the ML-KEM implementations using 3329 as the modulus and thus require very similar configuration, we skip the description here and refer the reader to the next subsection.

4.3.2 ML-KEM NTT & INTT. The code for the (inverse) NTT in ML-KEM consists of loops with 113–304 instructions. In case of our model for the Arm Cortex-M7, we found that the optimization of the ML-KEM NTT and INTT is not feasible without the use of the splitting heuristics. Therefore, we configure it to split the code into 3–6 parts, moving the window by 10–20% in each step with a timeout of 360 seconds. In the case of ML-KEM, we use fusion to split `ldrd` instructions, loading two GPRs at once, into pairs of `ldr` instructions, and do just the same for `vldm` instructions as we did in the FNT.

4.3.3 Keccak. Since high-performance implementations of Keccak such as the ones presented in [4] commonly rely on heavy loop-unrolling, the number of instructions to consider in the optimization of Keccak is significantly higher than for the NTTs. The Cortex-M4-tuned implementation from [4] consists of 1521 instructions in its main loop, which by far exceeds the threshold for SLOTHY to be able to optimally solve the problem. Thus, we again deploy the splitting heuristic and configure it to split the code into 22 parts, with a step-size of 5%. Moreover, we allow SLOTHY to take two passes over the code using the heuristic, to further facilitate the interleaving. We disable software pipelining as it would (1) be too complex to solve in this case and (2) blow up the code size significantly.

While optimizing the Cortex-M4 Keccak code from [4], SLOTHY identified a useless instruction buried deeply in the Keccak implementation's macros storing to the same memory location twice. While it is very hard to spot this instruction manually, SLOTHY will by default flag such useless instructions. We removed the useless instruction.

4.3.4 Other Functions. For all other functions, optimization is more straightforward and we can optimize the main loop of each function without the use of heuristics. One exception are the "matacc" functions within ML-KEM. Those include branches to external functions (SHAKE) as well as conditionals for sampling values $< q$ using rejection sampling. Neither sub-routine calls, nor conditionals are currently supported by SLOTHY and adding support does not promise much room for improvement as there is close to no flexibility in scheduling these instructions. Thus, we limit the optimization to only the arithmetic part of these function.

4.4 Implementation Security

All code used as input to optimizations in this work has secret-independent timing. The same holds for all resulting code as SLOTHY does not introduce branches or secret dependent memory accesses. Further, we ensure that our splitting or fusion replacement patterns only apply transformations that do not introduce secret dependent timing and are thus, safe to use.

5 RESULTS

In this section, we present the performance of the target functions on the Arm Cortex-M7 before and after optimization. For reference, we also present cycle counts on the Arm Cortex-M4 and from prior work.

The development board we used is a Nucleo-F767ZI which features a STM32F767ZI microcontroller. It has 2048 KiB of flash memory, 512 KiB of SRAM of which 128 KiB are tightly-coupled memory (DTCM). It runs at a frequency of up to 216 MHz offering a significant boost over common Cortex-M4 microcontrollers such as the STM32F407VG.

To perform component benchmarks and tests, we make use of the pqmx [1] framework that comes with SLOTHY. For scheme benchmarks and tests, we extend the popular benchmarking framework pqm4 [29]. For both, we add support for the STM32F767ZI as well as the QEMU Cortex-M7 platform mps2-an500. The former allows for realistic benchmarking on actual hardware, while the latter allows running functional tests without the actual hardware, easing development and allowing to perform tests in a continuous-integration environment.

As usual on microcontrollers, we run the devices at a reduced frequency of 24 MHz to eliminate wait states when fetching instructions from flash memory. We only use the DTCM memory as it offers better performance and 128 KiB of memory is sufficient for all of our benchmarks. Like pqm4, we make use of libopencm3 [33] easing setup of serial communication, clock configuration, and use of the hardware random number generator. For SHA-3 and SHAKE, we use the code available in pqm4 except for the KeccakF1600 permutation itself which we replace as detailed in the following. With this setup we were able to reproduce cycle counts reported in prior work [4] up to only 5 clock cycles difference.

For obtaining cycle counts prior to optimization on the Arm Cortex-M4, we make use of the common STM32F407-DICOVERY board with an STM32F407VG and use the default configuration in pqm4. For pqm4 comparison, we use pqm4 as of Jan 16, 2025 (Commit `49ce5bea`).

For all benchmarks, we use the Arm GNU toolchain version 13.3.Rel1.[5]

5.1 Component benchmarks

To demonstrate the effectiveness of SLOTHY, we first present results for the individual functions that were optimized. For stable

[5] `arm-none-eabi-gcc` from https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads

Table 1: Keccak benchmarking results before and after optimizations. We report speed in clock cycles (cc) and code size (cs) in in kilobytes.

Impl.	Before	Before M7			After M7			Speed-
	M4 cc	cc	IPC	cs	cc	IPC	cs	up
[17]	13368	6262	1.59	6.6	5376	1.85	6.6	1.16×
M4 [4]	9397	6691	1.37	6.0	5149	1.77	6.0	1.30×
M7 [4]	12399	5573	1.79	2.9	5322	1.87	3.3	1.05×

benchmarking, we run the code in a loop 100 times, and repeat the experiment 100 times reporting the median cycle counts.

Keccak. Our results for Keccak are shown Table 1. Prior to optimization, the best performing implementation is the hand-optimized Cortex-M7 implementation presented in [4]. Note that this implementation already achieves 1.79 IPC, which is close to optimal. Nonetheless, SLOTHY manages to find a 5% improvement, resulting in 1.87 IPC. Surprisingly, however, this is not the fastest implementation we found: Optimizing the Cortex-M4 implementation presented in [4] results in even fewer cycles. This implementation has a lower IPC of 1.77, yet it achieves better performance than the 1.87 IPC implementation. This is primarily due to the use of lazy rotations as described before in Section 2. This confirms that SLOTHY can outperform manual hand-optimization even when starting with code written for another microarchitecture. We also tried to run SLOTHY on a previous version of Keccak from [17], but it did not result in an implementation outperforming the other two after optimization. Code size remains mostly unaffected by our optimizations. Some differences are expected due to occasional switching between 16-bit and 32-bit instruction encoding because of other register choices by SLOTHY. The differences between the input implementations are due to varying degrees of unrolling.

ML-DSA. Table 2 contains the performance benchmarks of the core polynomial arithmetic in ML-DSA. For polynomial arithmetic modulo the ML-DSA prime, we see the Cortex-M4 code performing very poorly, achieving only around 1 IPC or even less. SLOTHY finds vastly better code for these functions with speed-ups ranging from 1.75× to 1.97×. For small polynomial multiplication, we evaluate both the 257 and 769 arithmetic proposed in [3] (with the former only being applicable to ML-DSA-44 and ML-DSA-87). SLOTHY achieves significant speeds-up for all those functions. We see particularly good speed-ups for the pointwise multiplications of 2.35× and 2.14×. The reader may be surprised to see a speed-up of larger than two. However, this can be explained by the particularly poor performance of the original code achieving only 0.74 IPC and 0.78 IPC. This poor performance is mainly due to consecutive multiplications depending on each other. As multiplications cannot dual-issue and have a latency of two cycles, this leads to an extra stall and at worst 3 unused issue slots. This effect occurs in other places as well but is particularly pronounced for these pointwise multiplications. In addition, this choice keeps the comparability of the results to prior work such that differences all relate to optimizations by SLOTHY. We also see small speed-ups for the Barrett reduction and conditional addition. Code size increases by a factor of almost 2 for all examples that make use of software pipelining. It also increases moderately in the cases where an `ldm` gets split into multiple instructions. Small variations in the code size are also expected due to switching between 16-bit and 32-bit instruction encoding.

ML-KEM. We present the component results for ML-KEM in Table 3. Similar to ML-DSA, the Cortex-M4 code for the NTT and iNTT performs poorly on the Cortex-M7, not exceeding 1.04 IPC. SLOTHY finds code that performs 1.62× and 1.64× faster. We achieve similar speed-ups for the various base multiplications deployed in ML-KEM, as well as the Barrett reduction. For the Barrett reduction, we see a vast increase in IPC as well, however, this is partly due to the splitting of `ldm` instruction which improves the IPC more than the performance. Lastly, we see only very small improvements for the matacc functions. These functions inline the sampling of the matrix $\mathbb{A}$ into the computation of the rejection sampling and base multiplication resulting in complex assembly including multiple loops with function calls. We can only apply SLOTHY to a share of this computation, namely the base multiplication which greatly limits the speed-ups we see. Note that these functions also perform function calls to Keccak to sample the matrix. We exclude the cycles spent in Keccak in the benchmarks here to ease interpretation. Code size changes similarly as for the ML-DSA.

5.2 ML-DSA results

We report the full scheme results for NIST security level 3 in Table 4. The results for the other parameter sets can be obtained from Table 5 in Appendix C For fairness reasons we mainly compare our implementation to two others: One referred to as 'pqm4' which is the current implementation in pqm4 (Commit `49ce5bea`) including the Cortex-M4 Keccak implementation. This is the Keccak implementation that was used to obtain our fast Cortex-M7 implementation. We also compare to a second implementation called 'pqm4*' which is the same implementation but using the Cortex-M7 Keccak from [4]. This can be seen as the state-of-the-art prior to our work. Compared to 'pqm4', we achieve speed-ups of 11% to 33%. Compared to 'pqm4*', we achieve speed-ups of 2% to 21%. We also present the results from [24] which benchmarks Cortex-M4 implementations on the Cortex-M7. However, these results were obtained from an older Cortex-M4 implementation and do not include Keccak optimized for the M7; hence, the performance is slower than the state of the art. As a reference, we also report Cortex-M4 cycles obtained from pqm4. Compared to those our Cortex-M7 implementations require 1.58× to 1.82× fewer cycles.

5.3 ML-KEM results

Table 4 contains our full scheme results for ML-KEM's NIST security category 3. Results for all parameter sets can be found in Table 6 in Appendix C. We present the same comparisons as for ML-DSA except that there are no previous results published on the Cortex-M7. Compared to the 'pqm4' implementation (M4-optimized code and M4 Keccak), we achieve speed-ups of 22% to 27%. Compared to the 'pqm4*' implementation (M4-optimized code and M7 Keccak), we achieve speed-ups of 9% to 14%. Comparing to cycles on the less powerful Cortex-M4, we require 1.66× to 1.74× fewer cycles.

Table 2: ML-DSA components results. We report performance of the input code in clock cycles on both the Cortex-M4 and the Cortex-M7 prior to optimization. Performance after optimization is reported for the Cortex-M7. Code size before and after optimization is reported in bytes.

Function	Before M4	Before M7			After M7			Speedup
	clock cycles	clock cycles	IPC	code size	clock cycles	IPC	code size	
NTT	8145	8139	0.95	860	4141	1.87	1704	1.97×
iNTT	8683	8207	0.95	1052	4547	1.72	1498	1.80×
Basemul	1914	2080	0.82	132	1063	1.61	212	1.96×
Basemul Acc.	2511	2166	1.03	164	1235	1.80	268	1.75×
NTT 769	4446	4013	1.04	1692	2418	1.75	1702	1.66×
iNTT 769	4575	4194	1.00	2008	2555	1.69	2060	1.64×
Pointmul 769	1025	1053	0.79	124	448	1.87	228	2.35×
Basemul asym. 769	1700	1887	0.82	124	1007	1.56	220	1.87×
FNT 257	5480	5130	0.99	1864	3903	1.34	2090	1.31×
iFNT 257	5552	4988	1.04	1680	3411	1.56	1738	1.46×
Pointmul 257	1155	1311	0.74	136	612	1.59	256	2.14×
Basemul asym. 257	1184	862	1.19	76	608	1.69	140	1.42×
Barrett reduction	1439	989	1.37	190	833	1.62	340	1.19×
caddq	1183	668	1.64	140	636	1.72	150	1.05×

Table 3: ML-KEM components results. We report performance of the input code in clock cycles on both the Cortex-M4 and the Cortex-M7 prior to optimization. Performance after optimization is reported for the Cortex-M7. We split functions by whether they are used in the speed- or stack-optimized ML-KEM implementation or both. Code size before and after optimization is reported in bytes. For the matacc functions, we report cycle counts excluding the Keccak function calls.

	Function	Before M4	Before M7			After M7			Speedup
		clock cycles	clock cycles	IPC	code size	clock cycles	IPC	code size	
all	NTT	4444	4011	1.04	1692	2380	1.78	1704	1.69×
	iNTT	4617	4247	1.00	2176	2586	1.68	2236	1.64×
	Barrett reduction	1391	1213	0.90	292	717	1.84	652	1.69×
	poly add	722	412	0.63	68	386	1.46	134	1.07×
	poly sub	722	412	0.63	68	360	1.57	136	1.14×
m4fspeed	basemul 16 32	1184	862	1.19	76	605	1.70	138	1.42×
	basemul 32 32	1568	1053	1.22	92	797	1.61	172	1.32×
	basemul 32 16	2087	2207	0.82	144	1059	1.71	254	2.08×
	frombytes + basemul 16 32	2468	2272	0.90	156	1251	1.64	278	1.82×
	frombytes + basemul 32 32	3236	2401	0.96	170	1378	1.68	306	1.74×
	frombytes + basemul 32 16	3688	3712	0.83	228	1895	1.63	420	1.96×
	matacc_asm_cache_16_32	8454	6203	1.01	506	5449	1.15	506	1.14×
	matacc_asm_cache_32_32	9158	6542	1.00	530	5867	1.11	534	1.12×
	matacc_asm_cache_32_16	9551	7646	0.94	634	5967	1.20	634	1.28×
	matacc_asm_opt_16_32	7939	5281	1.06	422	5269	1.07	418	1.00×
	matacc_asm_opt_32_32	8433	5517	1.06	450	5291	1.11	446	1.04×
	matacc_asm_opt_32_16	8779	6610	0.99	550	5560	1.17	550	1.19×
m4fstack	basemul_asm_acc	2787	2847	0.79	164	1633	1.45	308	1.74×
	basemul_asm	2403	2721	0.73	148	1440	1.47	274	1.89×
	frombytes_mul_asm_acc	3364	3648	0.81	212	1864	1.38	406	1.96×
	frombytes_mul_asm	2980	3392	0.76	188	1793	1.43	348	1.89×
	matacc_asm_acc	8833	7093	0.94	546	5757	1.15	546	1.23×
	matacc_asm	8513	6946	0.92	514	5604	1.14	514	1.24×

Table 4: ML-DSA & ML-KEM full scheme results for NIST security strength category 3. The 'pqm4' implementation refers to the current implementation in pqm4 including the Keccak which is the Cortex-M4 implementation described in [4]. 'pqm4*' refers to the same implementation but using the Cortex-M7 Keccak permutation from [4]. The 'pqm4' Cortex-M4 results are directly taken from the pqm4 tables. We report the mean of 5000 executions.

		CPU	impl	KeyGen		Sign		Verify	
ML-DSA m4f	65	M7	Ours	1446421	(×1.00)	3412174	(×1.00)	1380098	(×1.00)
		M7	pqm4	1808627	(×1.25)	4489455	(×1.32)	1753730	(×1.27)
		M7	pqm4*	1591914	(×1.10)	4064951	(×1.19)	1552298	(×1.12)
		M7	[24]	2566000	(×1.77)	6009000	(×1.76)	2453000	(×1.78)
		M4	pqm4	2516006	(×1.74)	6193171	(×1.82)	2415944	(×1.75)
ML-DSA m4fstack	65	M7	Ours	1985399	(×1.00)	15439643	(×1.00)	3527430	(×1.00)
		M7	pqm4	2526270	(×1.27)	17213084	(×1.11)	3925753	(×1.11)
		M7	pqm4*	2252520	(×1.13)	16650119	(×1.08)	3726781	(×1.06)
		M4	pqm4	3412622	(×1.72)	24421526	(×1.58)	5732397	(×1.63)

		CPU	impl	KeyGen		Encaps		Decaps	
ML-KEM m4fspeed	768	M7	Ours	372064	(×1.00)	385371	(×1.00)	417224	(×1.00)
		M7	pqm4	462147	(×1.24)	469808	(×1.22)	511840	(×1.23)
		M7	pqm4*	414551	(×1.11)	420946	(×1.09)	462237	(×1.11)
		M4	pqm4	642096	(×1.73)	658754	(×1.71)	707827	(×1.70)
ML-KEM m4fstack	768	M7	Ours	371048	(×1.00)	386586	(×1.00)	423908	(×1.00)
		M7	pqm4	469856	(×1.27)	482068	(×1.25)	526182	(×1.24)
		M7	pqm4*	422102	(×1.14)	433121	(×1.12)	476824	(×1.12)
		M4	pqm4	644195	(×1.74)	664654	(×1.72)	714194	(×1.68)

6 CONCLUSION

In this paper, we have shown that migrating highly hand-optimized code to a new microarchitecture with SLOTHY is feasible and can yield significant performance improvements. As we demonstrated based on the ML-KEM and ML-DSA code from pqm4 and Keccak from [4], with only little to no manual modifications to the source, it is possible to obtain fast implementations for the target of our case study, the Arm Cortex-M7. While building the architectural and microarchitectural models has been time-consuming, this is a one-time-effort that can be approached lazily. Note that for platforms, where the CPU designer provides a software optimization guide, this task would be vastly simplified.

Beyond the general result from our study, we were able to disprove a conjecture from [4], claiming that lazy rotations are not beneficial for Keccak on the Cortex-M7, while at the same time providing the currently fastest, open-source Keccak implementation for the Cortex-M7. We also identified a useless instruction in the state-of-the-art Cortex-M4 implementation in Keccak.

Moreover, we provide the first open-source implementation of ML-KEM and ML-DSA specifically tuned for the Arm Cortex-M7, with many subroutines delivering near-optimal performance with close to 2 IPC. This, together with our improved Keccak implementation, yields performance gains of up to 32 % compared to the original pqm4 code on the Cortex-M7.

ACKNOWLEDGMENTS

We thank Hanno Becker for explaining the inner workings of SLOTHY and helping us extend it.

REFERENCES

[1] Amin Abdulrahman, Hanno Becker, Matthias Kannwischer, and Fabien Klein. 2024. pqmx: Post-Quantum Cryptography on Arm Cortex-M. https://github.com/slothy-optimizer/pqmx.

[2] Amin Abdulrahman, Hanno Becker, Matthias J. Kannwischer, and Fabien Klein. 2024. Fast and Clean: Auditable high-performance assembly via constraint solving. *IACR TCHES* 2024, 1 (2024), 87–132. https://doi.org/10.46586/tches.v2024.i1.87-132

[3] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. 2022. Faster Kyber and Dilithium on the Cortex-M4. In *ACNS 22International Conference on Applied Cryptography and Network Security (LNCS, Vol. 13269)*, Giuseppe Ateniese and Daniele Venturi (Eds.). Springer, Cham, 853–871. https://doi.org/10.1007/978-3-031-09234-3_42

[4] Alexandre Adomnicai. 2023. An update on Keccak performance on ARMv7-M. Cryptology ePrint Archive, Report 2023/773. https://eprint.iacr.org/2023/773

[5] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. 2020. NISTIR8309 – Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. https://doi.org/10.6028/NIST.IR.8309.

[6] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. 2020. Cortex-M4 optimizations for {R,M}LWE schemes. *IACR TCHES* 2020, 3 (2020), 336–357. https://doi.org/10.13154/tches.v2020.i3.336-357

[7] Erik Richter Altman. 1995. Optimal software pipelining with function unit and register constraints. Thesis, McGill University, Montreal.

[8] Erik R. Altman, R. Govindarajan, and Guang R. Gao. 1998. A Unified Framework for Instruction Scheduling and Mapping for Function Units with Structural Hazards. *J. Parallel and Distrib. Comput.* 49, 2 (1998), 259–293.

[9] ARM Limited. 2020. *Cortex-M4 Technical Reference Manual Revision r0p1.* https://documentation-service.arm.com/static/5fce431be167456a35b36ade.

[10] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. 2022. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR TCHES* 2022, 1 (2022), 221–244. https://doi.org/10.46586/tches.v2022.i1.221-244

[11] Hanno Becker and Matthias J. Kannwischer. 2022. Hybrid Scalar/Vector Implementations of Keccak and SPHINCS$^+$ on AArch64. In *INDOCRYPT 2022 (LNCS, Vol. 13774)*, Takanori Isobe and Santanu Sarkar (Eds.). Springer, Cham, 272–293. https://doi.org/10.1007/978-3-031-22912-1_12

[12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2013. Keccak. In *EUROCRYPT 2013 (LNCS, Vol. 7881)*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer, Berlin, Heidelberg, 313–314. https://doi.org/10.1007/978-3-642-38348-9_19

[13] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy*. IEEE Computer Society Press, 353–367. https://doi.org/10.1109/EuroSP.2018.00032

[14] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. 2019. Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4. In *AFRICACRYPT 19 (LNCS, Vol. 11627)*, Johannes Buchmann, Abderrahmane Nitaj, and Tajje eddine Rachidi (Eds.). Springer, Cham, 209–228. https://doi.org/10.1007/978-3-030-23696-0_11

[15] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. 2021. NTT Multiplication for NTT-unfriendly Rings. *IACR TCHES* 2021, 2 (2021), 159–188. https://doi.org/10.46586/tches.v2021.i2.159-188

[16] James W. Cooley and John W. Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301. https://www.jstor.org/stable/2003354.

[17] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. 2024. eXtended Keccak Code Package. https://github.com/XKCP/XKCP.

[18] Charlie Demerjian. 2015. ARM goes into great detail about the M7 core. https://semiaccurate.com/2015/04/30/arm-goes-great-detail-m7-core/.

[19] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR TCHES* 2018, 1 (2018), 238–268. https://doi.org/10.13154/tches.v2018.i1.238-268

[20] G. Gao and Q. Ning. 1992. Loop storage optimization for dataflow machines. In *Languages and Compilers for Parallel Computing*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 359–373.

[21] W. Morven Gentleman and G. Sande. 1966. Fast Fourier Transforms: for fun and profit. In *AFIPS Fall Joint Computing Conference 1966*. AFIPS, 563–578. https://www.cis.rit.edu/class/simg716/FFT_Fun_Profit.pdf.

[22] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. 2021. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR TCHES* 2021, 1 (2021), 1–24. https://doi.org/10.46586/tches.v2021.i1.1-24

[23] Tim Güneysu, Markus Krausz, Tobias Oder, and Julian Speith. 2018. Evaluation of Lattice-Based Signature Schemes in Embedded Systems. In *ICECS 2018*. 385–388. https://www.seceng.ruhr-uni-bochum.de/media/seceng/veroeffentlichungen/2018/10/17/paper.pdf.

[24] James Howe and Bas Westerbaan. 2023. Benchmarking and Analysing the NIST PQC Lattice-Based Signature Schemes Standards on the ARM Cortex M7. In *AFRICACRYPT 23 (LNCS, Vol. 14064)*, Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne (Eds.). Springer, Cham, 442–462. https://doi.org/10.1007/978-3-031-37679-5_19

[25] Junhao Huang, Alexandre Adomnicai, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. 2024. Revisiting Keccak and Dilithium Implementations on ARMv7-M. *IACR TCHES* 2024, 2 (2024), 1–24. https://doi.org/10.46586/tches.v2024.i2.1-24

[26] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. 2022. Improved Plantard Arithmetic for Lattice-based Cryptography. *IACR TCHES* 2022, 4 (2022), 614–636. https://doi.org/10.46586/tches.v2022.i4.614-636

[27] jnk0le. 2024. cortex m7. https://github.com/jnk0le/random/blob/4b8968be72c09b2e2a7c2ec983b1668a1c184658/pipelinecycletest/doc/cortex_m7.md.

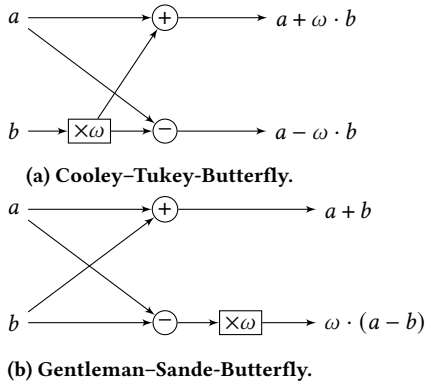
[28] Rajeev Joshi, Greg Nelson, and Keith H. Randall. 2002. Denali: a goal-directed superoptimizer. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*.

[29] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. [n. d.]. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[30] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. 2019. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844. https://eprint.iacr.org/2019/844

[31] Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom. 2023. CryptOpt: Verified Compilation with Randomized Program Search for Cryptographic Primitives. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1268–1292. https://doi.org/10.1145/3591272

[32] Monica Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, Richard L. Wexelblat (Ed.). ACM, 318–328. https://doi.org/10.1145/53990.54022



(a) Cooley–Tukey-Butterfly.

(b) Gentleman–Sande-Butterfly.

Figure 1: Common Butterfly Operations.

[33] libopencm3 authors. 2024. Open source ARM Cortex-M microcontroller library. https://github.com/libopencm3/libopencm3.

[34] Vadim Lyubashevsky. 2009. Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In *ASIACRYPT 2009 (LNCS, Vol. 5912)*, Mitsuru Matsui (Ed.). Springer, Berlin, Heidelberg, 598–616. https://doi.org/10.1007/978-3-642-10366-7_35

[35] Daniel Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, and David Silver. 2023. Faster sorting algorithms discovered using deep reinforcement learning. *Nature* 618 (06 2023), 257–263. https://doi.org/10.1038/s41586-023-06004-9

[36] Ben Marshall, Dan Page, and James Webb. 2022. MIRACLE: MIcRo-ArChitectural Leakage Evaluation A study of micro-architectural power leakage across many devices. *IACR TCHES* 2022, 1 (2022), 175–220. https://doi.org/10.46586/tches.v2022.i1.175-220

[37] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS II)*. Association for Computing Machinery, 122–126.

[38] NIST. 2015. FIPS 202: Secure Hash Algorithm-3. https://doi.org/10.6028/NIST.FIPS.202

[39] NIST. 2023. FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard. https://doi.org/10.6028/NIST.FIPS.203

[40] NIST. 2024. FIPS 204: Module-Lattice-Based Digital Signature Standard. https://doi.org/10.6028/NIST.FIPS.204

[41] Mark Owen. 2024. cortex m7. https://www.quinapalus.com/cm7cycles.html.

[42] Laurent Perron and Vincent Furnon. [n. d.]. Google OR-Tools. https://developers.google.com/optimization/.

[43] B. Ramakrishna Rau and Christopher D. Glaeser. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th annual workshop on Microprogramming, MICRO 1981, Chatham (Cape Cod), Massachusetts, USA*, Dick Eckhouse (Ed.). IEEE/ACM, 183–198. http://dl.acm.org/citation.cfm?id=802449

[44] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *CoRR* abs/1711.04422 (2017). arXiv:1711.04422 http://arxiv.org/abs/1711.04422

[45] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. 2017. *CRYSTALS-KYBER*. Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions.

A BUTTERFLY OPERATIONS

Figure 1 shows a depiction of the two most common butterfly operations.

B INSTRUCTION SPLITTING & FUSION

Listing 1 compares two SLOTHY-optimized code snippets, one with splitting, the other without.

C FULL SCHEME RESULTS

Tables 5 and 6 provide the full results for the scheme benchmarks for Kyber and Dilithium.

```
// Expected cycles: 12
ldm r0, {r2-r9}   // *...........
uadd16 r8, r8, r1 // ....*.......
uadd16 r5, r5, r1 // .....*......
uadd16 r7, r7, r1 // ......*.....
uadd16 r6, r6, r1 // .......*....
uadd16 r4, r4, r1 // ........*...
uadd16 r3, r3, r1 // .........*..
uadd16 r2, r2, r1 // ..........*.
uadd16 r9, r9, r1 // ...........*
```

(a) SLOTHY-optimized code without splitting.

```
// Expected cycles: 9
ldr r10, [r0, #0]  // *........
uadd16 r2, r10, r1 // .*.......
ldr r11, [r0, #28] // .*.......
uadd16 r9, r11, r1 // ..*......
ldr r14, [r0, #4]  // ..*......
uadd16 r3, r14, r1 // ...*.....
ldr r4, [r0, #8]   // ...*.....
uadd16 r4, r4, r1  // ....*....
ldr r8, [r0, #12]  // ....*....
uadd16 r5, r8, r1  // .....*...
ldr r12, [r0, #24] // .....*...
uadd16 r8, r12, r1 // ......*..
ldr r6, [r0, #16]  // ......*..
uadd16 r6, r6, r1  // .......*.
ldr r7, [r0, #20]  // .......*.
uadd16 r7, r7, r1  // ........*
```

(b) SLOTHY-optimized code with splitting.

Listing 1: Example for SLOTHY's splitting feature.

Table 5: ML-DSA full scheme results. The 'pqm4' implementation refers to the current implementation in pqm4 including the Keccak which is the Cortex-M4 implementation described in [4]. 'pqm4*' refers to the same implementation but using the Cortex-M7 Keccak permutation from [4]. The 'pqm4' Cortex-M4 results are directly taken from the pqm4 tables. We report the mean of 5000 executions.

		CPU	impl	KeyGen		Sign		Verify	
m4f	44	M7	Ours	813378	(×1.00)	2127982	(×1.00)	813339	(×1.00)
		M7	pqm4	1017840	(×1.25)	2821229	(×1.33)	1032625	(×1.27)
		M7	pqm4*	902168	(×1.11)	2564933	(×1.21)	923587	(×1.14)
		M7	[24]	1437000	(×1.77)	3658000	(×1.72)	1429000	(×1.76)
		M4	pqm4	1425492	(×1.75)	3822701	(×1.80)	1421600	(×1.75)
	65	M7	Ours	1446421	(×1.00)	3412174	(×1.00)	1380098	(×1.00)
		M7	pqm4	1808627	(×1.25)	4489455	(×1.32)	1753730	(×1.27)
		M7	pqm4*	1591914	(×1.10)	4064951	(×1.19)	1552298	(×1.12)
		M7	[24]	2566000	(×1.77)	6009000	(×1.76)	2453000	(×1.78)
		M4	pqm4	2516006	(×1.74)	6193171	(×1.82)	2415944	(×1.75)
	87	M7	Ours	2449052	(×1.00)	4509525	(×1.00)	2401930	(×1.00)
		M7	pqm4	3063089	(×1.25)	5926412	(×1.31)	3040658	(×1.27)
		M7	pqm4*	2687624	(×1.10)	5424423	(×1.20)	2677843	(×1.11)
		M7	[24]	4368000	(×1.78)	8157000	(×1.81)	4287000	(×1.78)
		M4	pqm4	4274513	(×1.75)	8204023	(×1.82)	4193228	(×1.75)
m4fstack	44	M7	Ours	1029294	(×1.00)	7633106	(×1.00)	1983998	(×1.00)
		M7	pqm4	1314162	(×1.28)	8512457	(×1.12)	2215408	(×1.12)
		M7	pqm4*	1170583	(×1.14)	7787117	(×1.02)	2097750	(×1.06)
		M4	pqm4	1799062	(×1.75)	12134284	(×1.59)	3242333	(×1.63)
	65	M7	Ours	1985399	(×1.00)	15439643	(×1.00)	3527430	(×1.00)
		M7	pqm4	2526270	(×1.27)	17213084	(×1.11)	3925753	(×1.11)
		M7	pqm4*	2252520	(×1.13)	16650119	(×1.08)	3726781	(×1.06)
		M4	pqm4	3412622	(×1.72)	24421526	(×1.58)	5732397	(×1.63)
	87	M7	Ours	3327940	(×1.00)	20661879	(×1.00)	6091186	(×1.00)
		M7	pqm4	4262581	(×1.28)	23991841	(×1.16)	6824998	(×1.12)
		M7	pqm4*	3790676	(×1.14)	21699864	(×1.05)	6450158	(×1.06)
		M4	pqm4	5820537	(×1.75)	33357899	(×1.61)	9911514	(×1.63)

Table 6: ML-KEM full scheme results. The 'pqm4' implementation refers to the current implementation in pqm4 including the Keccak which is the Cortex-M4 implementation described in [4]. 'pqm4*' refers to the same implementation but using the Cortex-M7 Keccak permutation from [4]. The 'pqm4' Cortex-M4 results are directly taken from the pqm4 tables. We report the mean of 1000 executions.

		CPU	impl	KeyGen		Encaps		Decaps	
m4fspeed	512	M7	Ours	226715	(×1.00)	228740	(×1.00)	253999	(×1.00)
		M7	pqm4	283897	(×1.25)	281338	(×1.23)	313772	(×1.24)
		M7	pqm4*	253159	(×1.12)	251441	(×1.10)	283602	(×1.12)
		M4	pqm4	392423	(×1.73)	390881	(×1.71)	428167	(×1.69)
	768	M7	Ours	372064	(×1.00)	385371	(×1.00)	417224	(×1.00)
		M7	pqm4	462147	(×1.24)	469808	(×1.22)	511840	(×1.23)
		M7	pqm4*	414551	(×1.11)	420946	(×1.09)	462237	(×1.11)
		M4	pqm4	642096	(×1.73)	658754	(×1.71)	707827	(×1.70)
	1024	M7	Ours	590805	(×1.00)	601298	(×1.00)	639868	(×1.00)
		M7	pqm4	729330	(×1.23)	732494	(×1.22)	784625	(×1.23)
		M7	pqm4*	651372	(×1.10)	653261	(×1.09)	705017	(×1.10)
		M4	pqm4	1018976	(×1.72)	1031565	(×1.72)	1094008	(×1.71)
m4fstack	512	M7	Ours	225529	(×1.00)	229598	(×1.00)	258008	(×1.00)
		M7	pqm4	285456	(×1.27)	285014	(×1.24)	318696	(×1.24)
		M7	pqm4*	255300	(×1.13)	256111	(×1.12)	289345	(×1.12)
		M4	pqm4	392224	(×1.74)	392864	(×1.71)	430202	(×1.67)
	768	M7	Ours	371048	(×1.00)	386586	(×1.00)	423908	(×1.00)
		M7	pqm4	469856	(×1.27)	482068	(×1.25)	526182	(×1.24)
		M7	pqm4*	422102	(×1.14)	433121	(×1.12)	476824	(×1.12)
		M4	pqm4	644195	(×1.74)	664654	(×1.72)	714194	(×1.68)
	1024	M7	Ours	592612	(×1.00)	607109	(×1.00)	653197	(×1.00)
		M7	pqm4	744941	(×1.26)	754554	(×1.24)	809372	(×1.24)
		M7	pqm4*	668936	(×1.13)	676800	(×1.11)	731518	(×1.12)
		M4	pqm4	1020202	(×1.72)	1037953	(×1.71)	1100982	(×1.69)