Auto-Vectorization for Lattice-Based Cryptographic Primitives

Camille Bossut
cbossut21@gatech.edu
Georgia Institute of Technology
USA

Qirun Zhang
qrzhang@gatech.edu
Georgia Institute of Technology
USA

1 ABSTRACT

As encryption is ubiquitous in today's world, the implementations of cryptographic algorithms must be both secure and fast. Encryption algorithms are often the targets of aggressive and meticulous manual optimization due to this speed requirement. Cryptography has evolved rapidly in recent years. In 2017, NIST (National Institute of Standards and Technology) made a call for *post-quantum* cryptographic protocols. The proposed algorithms should not break under a sufficiently powerful implementation of Shor's algorithm, a quantum-computing based algorithm which achieves integer factorization in polynomial time.

The three finalists for the new post-quantum cryptography standard, as chosen by NIST, are Kyber [4], Saber [5], and NTRU Prime [3]. All three algorithms are lattice-based, and all three incorporate hand-optimized assembly for AVX2 and NEON [11] which use SIMD instructions to exploit instruction-level parallelism in matrix operations.

With each new encryption scheme and algorithm, experts write hand-optimized assembly to achieve competitive performance. These hand-optimizations are expensive, time-consuming, and bug-prone. If an algorithm changes or a modification is released, the assembly must be changed. If the ISA changes for a target architecture, the previous implementation may become deprecated or sub-optimal, meaning the assembly must also be re-written. SIMD instructions are useful for both big-integer operations and matrix-based operations, the latter of which are frequent in lattice-based crytography. However, compilers often do not output the optimal implementation for each target architecture. The efficacy of auto-vectorization in modern compilers like GCC or Clang is heavily affected by the structure of the source code.

To mitigate the issue of human error introduced in handwritten assembly, some work attempts to verify assembly code. For example, the number-theoretic transform, or NTT, is an essential and frequent operation in lattice-based cryptography, and author Hwang verifies an NTT implementation for the Intel AVX2 instruction set [7]. The trouble with assembly verification is that it is usually done on a small portion of an overall scheme, rather than the scheme in its entirety. Ultimately, a compiler must be trusted to link the verified implementation with the rest of the code.

Other work aims to formally verify entire cryptographic protocols. Fiat cryptography [6] automatically generates straight-line code for cryptographic primitives from COQ proofs; thus the generated code is verified. CryptOpt [8] further seeks to optimize this generated code using a random-local-search approach. Neither target cryptographic primitives for lattice-based protocols, and neither support the generation of SIMD instructions. Additionally, the code output from these tools cannot compete with the equivalent hand optimized assembly, though the output performs better than a naively compiled version. HACL* [13] presents a verified set of cryptographic primitives in F, and HACLxN [12] is an extension which presents some verified SIMD cryptographic primitives as well. These HACL libraries also do not contain cryptographic primitives for lattice-based cryptography.

CompCert [9] is a functional verified C-compiler, and authors Barthe and Blazy [2] verified that a slightly modified CompCert is constant-time preserving. However GCC and Clang are used more often to compile deployed versions of cryptographic protocols, since their optimizations are much more aggressive than CompCert, thus their output is much faster than what CompCert can produce.

Our aim is a middle ground of optimization and verifiable correctness for vectorized cryptographic primitives; we choose to trust modern compilers but not handwritten assembly. Developers already put trust in modern compilers, since the official versions of the proposed lattice-based protocols use them in their compilation tool-chain. Our optimization targets straight-line cryptographic primitives for lattice-based cryptography. Our goal is to guide compilers' existing auto-vectorization through hints given via *source-code transformation* to get the best vector instruction generation. Our approach is *iterative*: we learn what vectorization the compiler missed from its optimization pass information, and why it didn't apply certain optimizations. Then we transform the input code to make the parallelism more obvious to the compiler. We want our output to improve upon the default compiler behavior, and ultimately compete with hand-optimized versions. Our insight is that the structure of the source code influences which vector-instruction optimizations the compiler will try to apply. However, writing source code with the compiler in mind is at odds with legibility of the source code, making it easier to introduce bugs just like in handwritten assembly. We verify our source-code transformations to ensure that they are (1) correct and (2) not violating the constant time properties required by cryptographic primitives. The workflow of our tool is as shown in Figure 1.

The code transformer takes advantage of common matrix operation patterns in lattice-based protocols. It attempts to auto-vectorize the primitives in these schemes. It learns data dependency and vectorization information from the output assembly and from the compilers' optimization pass output. For example, the compiler may reveal that vectorization was attempted but not sucessful due

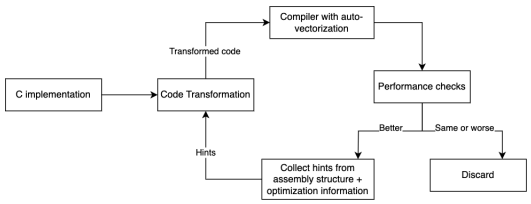

Figure 1: Workflow for source-code transformation.

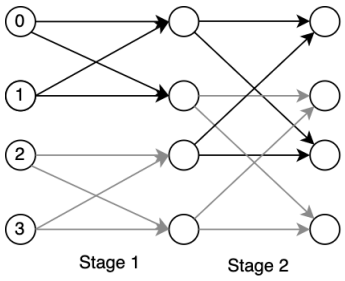

Figure 2: Butterfly pattern of operations in the NTT and inverse NTT algorithms.

to possible aliasing. Then, our transformer modifies the source code to reflect what it has learned or use some domain specific knowledge to rule out the optimization-preventing code property. Hints are given through code structure, compiler-specific annotations, vector intrinsics, and built-in functions, thus telling the compiler how and where to aggressively parallelize. As our input programs are straight-line constant-time programs which operate on large matrices, we can generate large data dependency graphs for nested loops and aggressively parallelize with our knowledge of the loop and matrix structure. Compilers alone may not always unroll loops when possible since loop unrolling is performed based on cost heuristics. Therefore, all vectorization possibilities are not be checked by default. Also, certain vector operations like permutation may not be likely to generate better code in general, but in the domain of lattice-based cryptography vector permutation is particularly useful, so we more aggressively search for places to use this operation.

The code surrounding a cryptographic primitive call is also key to our code transformations. There are many different implementations of cryptographic primitives like addition, multiplication, and modular reduction over finite fields. For instance, many lattice-based cryptography algorithms use Barrett reduction [1] for their modular reduction scheme, however in certain cases Montgomery reduction [10] is preferable. These primitives are chosen within a scheme based on their time complexity and the known input size, rather than with compiler optimizations in mind. The implementation which generates the best performing code may actually depend on the code around it. Another instance where code context is important; if a primitive is called from within a loop, inlining this function may open the door for SIMD instruction generation. We can target primitives used in loops with static analysis to see if inlining is possible. In this same context, it may be worth doing static analysis to see if its arguments are constant or un-aliased. The absence of these explicit conditions in the function header may have prevented possible vectorization within the cryptographic primitive.

Case study of the NTT. The NTT and the inverse NTT both follow a butterfly-type operation interleaving pattern as shown in Figure 2. The nodes labeled zero to three represent matrix indices, and arrows between nodes represent a data dependence between the source node and the destination node. We operate on straight-line code, so we can unroll any loops in the reference C code. We unroll the loop and group iterations to achieve the best parallelism at each stage. The non-interfering operations at the same stage can be done concurrently; we look for this pattern of instruction grouping and encourage the compilers to take advantage of it using the aforementioned hints. With the knowledge of this pattern and the source code loop structure, we can tell the compiler where the barriers between stages should be, and aggressively parallelize within a stage. We could lose this information if we only had access to an assembly implementation of the NTT, as it would be harder to recover the clear barriers between stages as shown in Figure 2.

The primary challenge of this work is ensuring it performs well on different target architectures. We use dynamic instruction count, CPU cycle count, and timing measurements to compare performance of the transformed source code. We use two compilers (GCC and Clang) to compile the transformed code and get optimization pass information, which we ultimately turn into source code hints. Since our process is iterative, it collects information about the vectorization options on the target architecture which influence the source code transformations. We ensure that our approach is general by testing it on architectures with different maximum vector sizes.

The goal of this work is to help bridge the gap between handwritten and auto-generated SIMD code for lattice-based cryptography. We use domain specific knowledge to guide source code transformation and anticipate certain access patterns to aggressively parallelize operations better than the compiler alone. We summarize our contributions as follows:

- Source-code level transformations which reveal instruction-level parallelism to compilers in lattice-based cryptographic primitives.
- Analysis of compiler optimization passes to choose an appropriate source code transformation.
- Verification that the transformations are correct and constant-time preserving.

In future work, we could apply this approach to other applications which benefit from instruction level-parallelism with characteristic patterns.

REFERENCES

[1] P. Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[2] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu. Formal verification of a constant-time preserving c compiler. *Proc. ACM Program. Lang.*, 4 (POPL), dec 2019.

[3] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. Ntru prime: reducing attack surface at low cost. Cryptology ePrint Archive, Paper 2016/461, 2016. https://eprint.iacr.org/2016/461.

[4] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle. Crystals - kyber: A cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroSP)*, pages 353–367, 2018.

[5] J.-P. D'Anvers, A. Karmakar, S. Sinha Roy, and F. Vercauteren. Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. In A. Joux, A. Nitaj, and T. Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2018*, pages 282–305. Springer International Publishing, 2018.

[6] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219, 2019.

[7] S. S. T. W. Y. Hwang, Liu. Verified ntt multiplications for nistpqc kem lattice finalists: Kyber, saber, and ntru. volume 2022, page 718–750, Aug. 2022.

[8] J. Kuepper, A. Erbsen, J. Gross, O. Conoly, C. Sun, S. Tian, D. Wu, A. Chlipala, C. Chuengsatiansup, D. Genkin, M. Wagner, and Y. Yarom. Cryptopt: Verified compilation with randomized program search for cryptographic primitives. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[9] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, jul 2009.

[10] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[11] D. T. Nguyen, K. Gaj, and G. Mason. Optimized software implementations of crystals-kyber, ntru, and saber using neon-based special instructions of armv8. 2021.

[12] M. Polubelova, K. Bhargavan, J. Protzenko, B. Beurdouche, A. Fromherz, N. Kulatova, and S. Zanella-Béguelin. Haclxn: Verified generic simd crypto (for all your favourite platforms). In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 899–918, New York, NY, USA, 2020. Association for Computing Machinery.

[13] J. K. Zinzindohoue, K. Bhargavan, J. Protzenko, and B. Beurdouche. Hacl*: A verified modern cryptographic library. Cryptology ePrint Archive, Paper 2017/536, 2017. https://eprint.iacr.org/2017/536.