Towards perfect CRYSTALS in Helium

Hanno Becker and Fabien Klein

Arm Limited,
hanno.becker@arm.com,
fabien.klein@arm.com

Abstract. In this work, we present a tool for the automated super optimization of Armv8.1-M Helium assembly on Cortex-M55. It consists of two parts: Firstly, a generic framework SLOTHY – <u>S</u>uper (<u>L</u>azy) <u>O</u>ptimization of <u>T</u>ricky <u>H</u>andwritten assembl<u>Y</u> – for expressing the super optimization of small pieces of assembly as a constraint satisfaction problem which can be handed to an external solver – concretely, we pick CP-SAT from Google OR-Tools. Secondly, an instantiation HeLight$_{55}$ of SLOTHY with the Armv8.1-M architecture and aspects of the Cortex-M55 microarchitecture. We demonstrate the power of SLOTHY and HeLight$_{55}$ by using it to optimize two workloads: First, a radix-4 complex Fast Fourier Transform (FFT) in fixed-point arithmetic, fundamental in Digital Signal Processing. Second, the instances of the Number Theoretic Transform (NTT) underlying CRYSTALS-Kyber and CRYSTALS-Dilithium [SAB$^+$22],[LDK$^+$22], two recently announced winners of the NIST Post-Quantum Cryptography standardization project [NIS16].

Keywords: Superoptimization · Software Pipelining · Constraint Solving · Phase ordering problem · Arm · MVE · Helium · Post-Quantum Cryptography · Fast Fourier Transform · FFT · Number Theoretic Transform · NTT

1 Introduction

The Armv8.1-M architecture [Armc] introduced the M-Profile Vector Extension (MVE), or Arm® Helium™ technology, which brings the performance promise of vector extensions to embedded microcontrollers, while taking into account their tight power/area profile. The way Helium solves this conundrum is by putting emphasis on the efficient usage of execution resources: Notably, MVE instructions may run for multiple cycles, *but* they may overlap if they operate on different execution resources. Similarly, Helium's vector register file is smaller than that of (say) the Neon vector extension, *but* it offers scalar-vector instructions which use vector and general purpose register files at the same time. To leverage those capabilities, good Helium code must be carefully scheduled to mix different classes of instructions – such as vector load/store vs. vector arithmetic – and manage the register files very efficiently.

Autovectorization with standard C/C++, or C intrinsics for Helium, provide a means to offload the complexities of instruction ordering and register allocation to the compiler, but they naturally constitute a tradeoff between convenience and performance, as the heuristics and time budgets of compilers aren't sufficient to always find optimal solutions within the huge search spaces for scheduling and register allocation problems.

In this work, we demonstrate the use of *constraint programming* for the *practical* superoptimization of medium-sized kernels (≈50 instructions) of Helium assembly, tailored for the Arm® Cortex®-M55 CPU. Specifically, we express the simultaneous optimization of (a) register allocation, (b) instruction scheduling, and (c) software pipelining (in the case of loops) as a mixed integer/boolean constraint satisfaction problem which can be passed to an external solver – here, we use CP-SAT from Google OR-Tools [PF] (version v9.3). While our initial focus is on Helium and Cortex-M55, we decouple specifics of the architecture and microarchitecture from the general constraint modelling, and expect our approach and software to be useful for other (micro)architectures as well. We call the general approach SLOTHY – Super (Lazy) Optimization of Tricky Handwritten assemblY; its extension for Helium and Cortex-M55 is called HeLight$_{55}$.

Contributions. Our contributions are threefold:

1. We describe and implement a (micro)architecture-agnostic superoptimizer SLOTHY for the tasks of register allocation, instruction ordering and software pipelining based on constraint solving.
2. We instantiate SLOTHY with the Armv8.1-M architecture and software optimization aspects of the Cortex-M55r1 microarchitecture, resulting in the HeLight$_{55}$ superoptimizer for Helium assembly on Cortex-M55.
3. We demonstrate the practicality of HeLight$_{55}$ by superoptimizing a radix-4 complex Fast Fourier Transform (FFT) in fixed-point arithmetic, as well as two instances of the Number Theoretic Transform (NTT) underlying the post-quantum cryptography key encapsulation and signature schemes CRYSTALS-Kyber and CRYSTALS-Dilithium.

Future work. There are (at least) two avenues of future work:
First, while we demonstrate the power and use of SLOTHY + HeLight$_{55}$ in complex real world examples, limitations remain, such as the modelling of superscalar microarchitectures (like the Cortex-M85 CPU) and the automated and optimal introduction of stack spills in case the register file is not large enough for the kernel under consideration. Nonetheless, we hope that our work will stimulate interest and further use and extension of SLOTHY + HeLight$_{55}$ to address those limitations, as well as extending it to other (micro)architectures.
Second, we encourage research in the application of SLOTHY + HeLight$_{55}$ for further workloads. In the context of post-quantum cryptography, for example, we do deliberately *not* build complete implementations of Kyber and Dilithium. We believe that doing so, under consideration the numerous implementation techniques available (such as [AHKS22], base multiplication strategies, CT vs. GS butterflies, ...), would make for an attractive piece of research, while it exceeds the scope of this paper.

Structure. In Section 2, we briefly discuss some preliminaries on Armv8.1-M+Helium, Google OR-Tools, software pipelining and super optimization. In Section 3, we describe how we express simultaneous optimization of register allocation, instruction scheduling and software pipelining as a mixed Boolean/integer constraint satisfaction problem. Section 4 then discusses how we extend SLOTHY to HeLight$_{55}$ by pairing it with (micro)architectural information on Armv8.1-M+Helium and Cortex-M55, and touch on the interfaces it provides to the user. Finally, in Section 5 and Section 6 we work through the examples of the Fast Fourier Transform and Number Theoretic Transform. We conclude with Section 8 containing some reflections and outlook.

Software. SLOTHY and HeLight$_{55}$ are freely available under MIT license on https://gitlab.com/arm-research/security/pqmx/.

Related work. There is a rich literature on superoptimization in general and the potential to use Integer Linear Programming (ILP) for it in particular [Mas87, SSA13, KL99, WGB94, GW96, KW98, Rau94, Lam88a, Lam88b]. Yet, to the best of our knowledge, our approach to using constraint solving for *simultaneously* addressing instruction scheduling, register

```
vldrw.u32  q0, [inA]        vldrw.u32  q0, [inA]        vldrw.u32  q0, [inA]
vadd.u32   q0, q0, q0       vadd.u32   q0, q0, q0       vadd.u32   q0, q0, q0
vadd.u32   q0, q0, q0       vadd.u32   q0, q0, q0       vldrw.u32  q1, [inB]
vstrw.u32  q0, [inA]        vstrw.u32  q0, [inA]        vadd.u32   q0, q0, q0
vldrw.u32  q0, [inB]        vldrw.u32  q1, [inB]        vmul.u32   q1, q1, q1
vmul.u32   q0, q0, q0       vmul.u32   q1, q1, q1       vstrw.u32  q0, [inA]
vmul.u32   q0, q0, q0       vmul.u32   q1, q1, q1       vmul.u32   q1, q1, q1
vstrw.u32  q0, [inB]        vstrw.u32  q1, [inB]        vstrw.u32  q1, [inB]
```

Listing 1: Left: Two logically independent code paths using the same register. Middle: Register renaming to separate register usage and enable interleaving. Right: Interleaving.

allocation, and software pipelining, is new. For software pipelining, prior art seems to focus on achieving high loop initiation rates for small loops on superscalar microarchitectures, while our initial goal is to optimize complex loops on the (largely) single-issue Cortex-M55 microarchitecture by interleaving at most 3 iterations at a time.

2 Preliminaries

2.1 Phase ordering problem

During compilation, two important code generation phases are *instruction scheduling* and *register allocation*. Instruction scheduling assigns a linear order to the instructions in a computational flow graph. Register allocation assigns architectural register names to logical instruction arguments. Instruction scheduling and register allocation influence each other, as the choice of scheduling restricts the set of valid register allocations, and vice versa: An instruction must not overwrite a register (an aspect of register allocation) if that instruction is placed in between a producer and consumer of said register (an aspect of scheduling). The relation and ordering between instruction scheduling and register allocation is an extensively studied problem called the *phase ordering problem*.

2.2 Software pipelining

Software pipelining [Lam88a, RG81] is a software optimization technique whereby multiple iterations of a loop are interleaved to create instruction level parallelism and thereby facilitate execution on the underlying microarchitecture. A popular approach is iterative modulo scheduling [Rau94]. While originally devised for Very Long Instruction Word (VLIW) processors, software pipelining is a well-known optimization technique also for classical microarchitectures: When the execution of one loop iteration cannot progress due to latency constraints or lack of availability of functional units, instructions from the next iteration(s) may be pulled forward to fill the gaps. This is conceptually similar to how out-of-order microarchitectures reorder instructions during execution, but explicit software pipelining may still be beneficial even for such microarchitectures.

Software pipelining puts pressure on the register file since iterations may only be interleaved once there is no collision in their register use. In an out-of-order microarchitecture, this is a consequence of register renaming assigning a fresh physical register to each instruction output, thereby benefiting from a physical register file that's larger than the architectural register file. Software pipelining, however, has to perform manual register renaming within the *architectural* register file, which can be challenging. See Listing 1 for a toy example of an interleaving opportunity created through register renaming.

2.3 Superoptimization

The term superoptimization was introduced in [Mas87] as finding "the shortest program that computes the same function as the source program by doing an exhaustive search over all possible programs". Here, we use the term more broadly for approaches to software optimization that find *optimal* solutions rather than approximations. Superoptimization can be studied from multiple angles, such its position within the overall software development and compilation flow, the scope of superoptimization, and techniques for its (efficient) implementation – we briefly comment on them and explain where our approach resides.

First, in terms of positioning, the original [Mas87] studies superoptimization at the level of assembly. In contrast, the more recent [SCC+17] discusses superoptimization at the level of the LLVM intermediate representation (IR), making it more broadly applicable. Our superoptimizer operates at the level of assembly.

Second, in terms of techniques, numerous approaches have been explored, such as brute force enumeration [Mas87], stochastic search [SSA13], and integer linear programming (ILP) [KL99, WGB94, GW96]. Our approach falls into the last category, applying mixed Boolean/integer constraint programming to express and solve the problem of finding optimal code.

Third, we comment on the scope of superoptimization. Following [Emb15, Section 11], one can broadly distinguish two separate optimization phases: First, the search for optimal (e.g. short) sequences of assembly expressing a given, typically loop-free, piece of functionality – this requires awareness of instruction semantics. Second, once instructions have been fixed, the search for an optimal scheduling and register allocation strategy – in contrast to the first approach, this only requires knowledge of the architectural signature of instructions, as well as microarchitectural information like the available functional units, latencies and throughputs. Here, we focus on the second approach: Finding optimal solutions for (simultaneous) instruction scheduling and register allocation. Further, we also include software pipelining into the scope of our superoptimizer.

2.4 Google OR-Tools

Google OR-Tools [PF] is a software for combinatorial optimization, tailored at solving problems such as vehicle routing, flows, integer and linear programming, and constraint programming. We find that Google OR-Tools's CP-SAT is very well suited to our assembly superoptimization problem, in two ways: First, it's fast. Second, CP-SAT's API allows for the specification of a mix of Boolean, integer, and interval variables, and moreover offers convenient constraints such as non-overlapping for intervals, or mutual difference for a set of integer variables. However, we expect our modeling approach to apply to other constraint solvers as well, and encourage further research and comparison.

2.5 M-Profile Vector Extension/ Helium

The M-Profile Vector Extension (MVE) is a Single Instruction Multiple Data (SIMD) extension that was introduced as part of the Armv8.1-M architecture [Armc]. Its primary goal is to enable higher performance for signal processing and machine learning applications. So far, MVE has been implemented in the Cortex-M55 CPU as well as the recently announced Cortex-M85 CPU.

MVE is also referred to as Arm® Helium™ technology, in alignment with the Arm® Neon™ Technology architecture extension for A-profile processors [Arma, Section C.3.5], or Neon for short. However, despite the similarity in name, Helium is a new ground-up architecture designed specifically for the tight area/power constraints of the embedded market. We refer to [BBMK+21, Armd, Armf, Armg] for introductions to Armv8.1-M+Helium and to the reference manual [Armc] for the details.

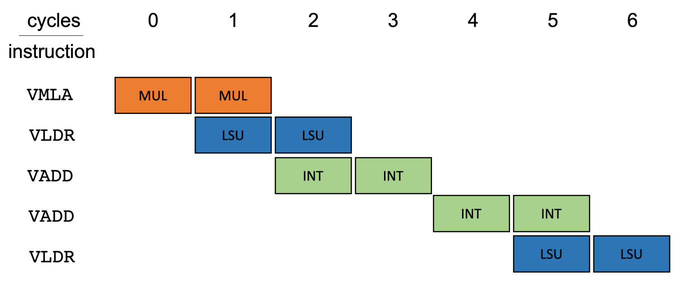

Figure 1: Illustration of instruction overlapping for instructions relying on separate functional units.

```
vldrw.u32   q0, [inA]
vldrw.u32   q1, [inA, #16]  //-
vldrw.u32   q2, [inA, #32]  //-
vldrw.u32   q7, [inB], #16  //-
vmulh.u32   q0, q0, q7
vmulh.u32   q1, q1, q7      //-
vmulh.u32   q2, q2, q7      //-
vadd.u32    q0, q0, q0
vadd.u32    q0, q0, q7      //-
vadd.u32    q1, q1, q1      //-
vadd.u32    q1, q1, q7      //-
vadd.u32    q2, q2, q2      //-
vadd.u32    q2, q2, q7      //-
vstrw.u32   q1, [inA, #16]
vstrw.u32   q2, [inA, #32]  //-
vstrw.u32   q0, [inA], #48  //-
```

```
vldrw.u32 q1, [inB], #16
vldrw.u32 q2, [inA, #16]  //-
vmulh.u32 q7, q2, q1
vldrw.u32 q4, [inA, #32]
vadd.u32  q7, q7, q7
vmulh.u32 q6, q4, q1
vadd.u32  q4, q7, q1
vldrw.u32 q2, [inA]
vadd.u32  q7, q6, q6
vmulh.u32 q6, q2, q1
vadd.u32  q7, q7, q1
vstrw.u32 q4, [inA, #16]
vadd.u32  q4, q6, q6
vstrw.u32 q7, [inA, #32]
vadd.u32  q7, q4, q1
vstrw.u32 q7, [inA], #48
```

Listing 2: Left: Poorly written snippet of Helium assembly with little potential for instruction overlapping. Right: Improved scheduling + register allocation. An //- annotation indicates a structural hazard preventing instruction overlapping.

What is most important about Helium for the sake of this work is how it carefully introduces software constraints to lower hardware complexity and thereby retain suitability for embedded microcontrollers: Most notably, instruction overlapping and the compact vector register file of 8×128-bit vector registers.

First, instruction overlapping allows to achieve up to $2\times$ performance with the same execution resources compared to non-overlapping, single-issued execution. However, it requires instructions to be scheduled in such a way that they run on different functional units and are therefore amenable to overlapping. Figure 1 provides an illustration, assuming an implementation of Helium where each vector instruction takes two cycles ("dual beat implementation") and where there are separate functional units for vector load/store (LSU), addition/logical operations (INT), and multiply operations (MUL) – the Cortex-M55 is an example for this. We can see how the first instructions overlap, leading to high resource utilization, but how the consecutive pair of `VADD` instructions stalls the pipeline. Typically, there is significant flexibility in instruction scheduling, so that good interleaving of different instruction types is possible. For example, Listing 2 shows two versions of the same piece of (meaningless) Helium assembly, one poorly scheduled, making little use of instruction overlapping, and another with a good mix of instructions, facilitating overlapping.

Second, the compact vector register file reduces the cost of CPUs implementing Helium, but requires developers or compilers to manage register usage very carefully, and balance it with the general purpose register file through the use of scalar-vector instructions.

The primary goal of this work is to demonstrate how to automate the process of solving the software constraints posed by Helium through constraint solving.

3 SLOTHY: Super optimization via constraint solving

This section, which is the heart of our work, describes SLOTHY – Super (Lazy) Optimization of Tricky Handwritten assemblY – our approach to modeling assembly superoptimization as a constraint solving problem. First, Section 3.1 clarifies the scope of our model and provides an overview. We then separately discuss constraints for functional correctness (Section 3.2, Section 3.3), software pipelining (Section 3.4, Section 3.5), and microarchitectural performance (Section 3.6, Section 3.7). Section 3.8 discusses the modeling of memory and stack, while Section 3.9 and Section 3.10 describe how we approach minimization of stalls, and other objectives. Finally, Section 3.11 comments on the soundness of SLOTHY.

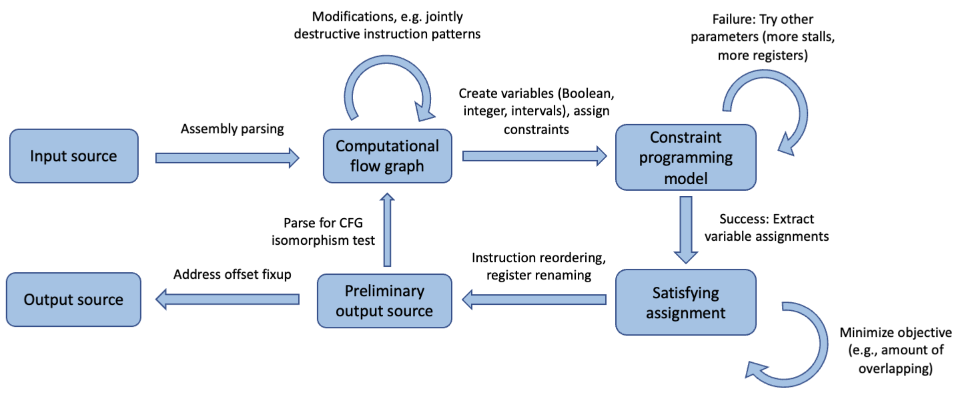
Modifications, e.g. jointly destructive instruction patterns
Failure: Try other parameters (more stalls, more registers)
Create variables (Boolean, integer, intervals), assign constraints
Assembly parsing
Input source
Computational flow graph
Constraint programming model
Parse for CFG isomorphism test
Success: Extract variable assignments
Instruction reordering, register renaming
Address offset fixup
Output source
Preliminary output source
Satisfying assignment
Minimize objective (e.g., amount of overlapping)

Figure 2: High-level overview of operation of SLOTHY

3.1 Scope and overview

SLOTHY optimizes instruction scheduling, register allocation and software pipelining. That is, it considers reordering of instructions and change of the use of registers in search for some functionally equivalent but optimally performing variant of the code. Importantly, instruction scheduling and register allocation are considered *simultaneously*, avoiding the phase ordering problem (Section 2.1). In case of a loop, SLOTHY also simultaneously searches for suitable interleavings of iterations. Put differently, SLOTHY retains the (isomorphism class of the) source's computational flow graph. Figure 3 shows the computational flow graph underlying both the naïve and optimized code in Listing 2.

Figure 2 provides an overview of SLOTHY's operation: First, the input is parsed and converted to a computational flow graph. Next, a constraint model is derived as explained below, and passed to an external solver. Upon success, the (optimal) satisfying assignment found by the solver is converted back into the output source. This source is then subject to some post-processing and self-check before being returned to the caller.

Multiple aspects of optimization are currently out of scope for SLOTHY. First, SLOTHY does not change instructions except for register renaming. In fact, it has no knowledge of the semantics of instructions beyond their signatures, that is, the number and types of inputs, outputs, and input/outputs. It thus remains the responsibility of the user to find suitable ways to express the target computation in terms of the underlying instruction set architecture (ISA). Second, SLOTHY has no notion of memory, and is therefore not able to introduce stack spills in situations where the register files are not large enough to hold all temporaries used by a computation. It is the responsibility of the user to introduce stack spills in this case and re-run SLOTHY. See Section 3.8.

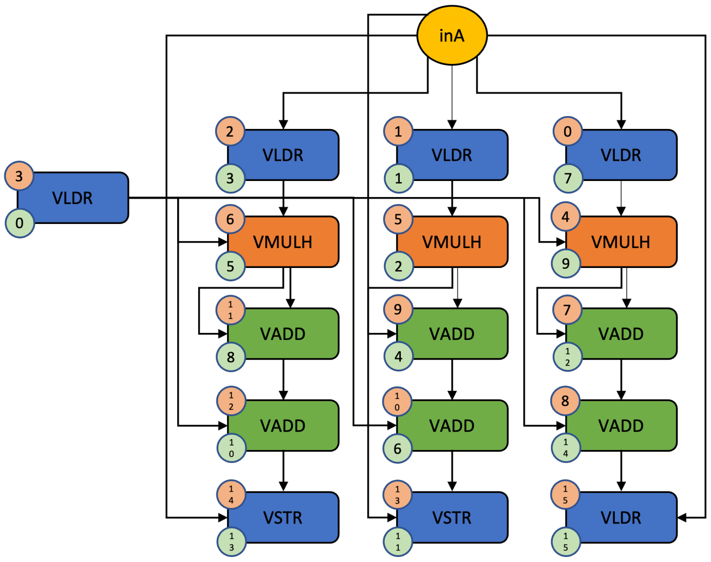

Figure 3: The shared computational flow graph for the naïve and optimized versions of Listing 2. The numbers indicate the program order used by both examples.

3.2 Correctness/Architecture constraints: Instruction scheduling

Every instruction `I` of the input source is assigned an integer variable `I.pos` defining where the instruction is placed in the output code. To get a unique program order in the output, we require $\{\mathtt{I.pos}\}_{\mathtt{I}}$ to be mutually distinct. Further, to maintain functional correctness, consumers must come after producers: If instruction `J` consumes output `O` produced by instruction `I` – that is, we have an edge $\mathtt{I} \xrightarrow{\mathtt{O}} \mathtt{J}$ in the computational flow graph – then $\mathtt{I.pos} < \mathtt{J.pos}$.

3.3 Correctness/Architecture constraints: Register Allocation

For every instruction `I`, every output `O` of `I`, and every possible register `R` that `I` can use for `O`, we assign a Boolean variable `alloc(I,O,R)` indicating whether the instruction `I` uses register `R` for the output `O`. In analogy with the operation of out-of-order microarchitectures, we call this process *register renaming*. Simultaneous input/output arguments are not subject to register renaming. Note that the input source's choice of registers is irrelevant here – it is only used initially to construct the computational flow graph from the source.

Multiple constraints need to be satisfied: First, for the uniqueness of register renaming, we require that for fixed `I,O`, exactly one Boolean variable in the family $\{\mathtt{alloc}(\mathtt{I},\mathtt{O},\mathtt{R})\}_{\mathtt{R}}$ is set. Second, for functional correctness, we need to express that in between a register being produced and consumed, no other instruction uses the register for register renaming. We model this as a disjointness constraint as follows: First, for any instruction `I` and any output `O`, we add a interval $[\mathtt{I} \xrightarrow{\mathtt{O}}\,]$ starting at `I.pos`, and bound its endpoint below by `J.pos` for any $\mathtt{I} \xrightarrow{\mathtt{O}} \mathtt{J}$, as well as by $\mathtt{I.pos} + 1$ (in case `O` has no consumers). For any choice of output register `R`, we then add a copy $[\mathtt{I} \xrightarrow{\mathtt{O}}\,]_{\mathtt{R}}$ of $[\mathtt{I} \xrightarrow{\mathtt{O}}\,]$ as a *conditional* interval constrained by `alloc(I,O,R)`. Functional correctness then requires that for any `R`, the set of conditional intervals $\{[\mathtt{I} \xrightarrow{\mathtt{O}}\,]_{\mathtt{R}}\}_{\mathtt{I},\mathtt{O}}$ is non-overlapping. For simultaneous input/output variables – for which we cannot rename – $[\mathtt{I} \xrightarrow{\mathtt{O}}\,]_{\mathtt{R}}$ is instead conditioned on `alloc(K,U,R)`, where `K,U` is the transitively computed source of `O`, with `U` being an *output* (not merely an input/output). For example, if `I` is an `MLA` in a `MUL; MLA; ...; MLA` chain and `O` is the accumulator, we'd always go back to the initial MUL which made the choice for which register to use for the accumulator.

We note that our modelling of register usage is an instance of the flexible job shop problem, with jobs being instructions and "machines" being registers.

Restricted instructions. Sometimes there are restrictions on the registers that an instruction can use. Such restrictions often apply to individual arguments (such as requiring even or odd registers), but there are also more complex cases: For example, the output of VCMUL in Helium must not coincide with an input, and the four (!) vector arguments to the de-interleaving load VLD4x are constrained to the set $\{\texttt{Qi}, \texttt{Q(i+1)}, \texttt{Q(i+2)}, \texttt{Q(i+3)}\}$ for $i = 0, 1, 2, 3, 4$. Individual restrictions on output arguments are straightforwardly added to the register renaming model by accordingly restricting the Boolean variables alloc(I,O,R). To model restrictions for multiple register outputs, such as for VLD4x/VST4x, we introduce Booleans for the various choices, constrain that precisely one is set, and then add implications to the respective alloc(I,O,R) variables.

Jointly destructive instruction patterns. Armv8.1-M+Helium has instruction patterns where each instruction individually overwrites only part of the destination register, but where the sequence as a whole overwrites it entirely. Examples are blocks of VLD4{0,1,2,3}, or pairs of VQDMLSDH+VQDMLADHX. In this case, the destination register is by default modeled as an input/output argument, which leads to unnecessary dependencies which in turn reduce reordering flexibility. To address this, SLOTHY allows architecture specifications to modify instructions within the context of an entire computational flow graph. For $\textsf{HeLight}_{55}$, we leverage this to detect and mark the destination register in aforementioned patterns as a pure output for the first instruction of the sequence, thereby allowing to reorder it past previous instructions which write to the same register.

3.4 Loop interleaving aka Software pipelining

A powerful feature of SLOTHY is software pipelining, that is, interleaving multiple iterations in a loop: Some early instructions, such as initial loads, are moved into the previous iteration, while some late instructions, such as final stores, are deferred to the next iteration. To avoid having to unroll the entire loop, periodicity of code has to be maintained.

We model software pipelining as follows: To begin, for each instruction I, we add three Boolean variables I.pre/I.late and I.core, indicating whether I will be an early/late instruction for the previous/next iteration, or whether it stays in its original iteration. Precisely one of $\{\texttt{I.pre}, \texttt{I.core}, \texttt{I.late}\}$ must be set. For periodicity, we first double the loop body C, say as $\texttt{C}_1$ and $\texttt{C}_2$; for an instruction I, we denote $\texttt{I}_1$ and $\texttt{I}_2$ its copies in $\texttt{C}_1$ and $\texttt{C}_2$, respectively. Note that this duplication is internal only and does, by default, not enforce an unrolling of the loop in the final code. We then relate $\texttt{I}_1$ and $\texttt{I}_2$ as follows: First, the choices of $\texttt{I.pre}, \texttt{I.core}, \texttt{I.late}$ and of register allocations must be the same for $\texttt{C}_1$ and $\texttt{C}_2$: $\texttt{I}_1\texttt{.\{pre,core,late\}} \Leftrightarrow \texttt{I}_2\texttt{.\{pre,core,late\}}$ and $\texttt{alloc}(\texttt{i}_1, \texttt{O}, \texttt{R}) \Leftrightarrow \texttt{alloc}(\texttt{I}_2, \texttt{O}, \texttt{R})$ for all $\texttt{I} \in \texttt{C}$. Second, "core" instructions are placed exactly n instructions apart, where n is the length of C: $\texttt{I.core} \Rightarrow \texttt{I}_2\texttt{.pos} = \texttt{I}_1\texttt{.pos} + n$. Third, and perhaps somewhat non-canonically, we force early instructions in $\texttt{C}_1$ to be "seen" as early instructions for the *next* iteration $\texttt{C}_3$, and late instructions in $\texttt{C}_2$ to be "seen" as late instructions for the *previous* iteration $\texttt{C}_0$, by requiring $\texttt{I.pre} \vee \texttt{I.late} \Rightarrow \texttt{I}_2\texttt{.pos} = \texttt{I}_1\texttt{.pos} - n$ (note the sign!) and by "cutting" the constraints derived from $\texttt{I}_1 \xrightarrow{\texttt{O}} \texttt{J}_1$ if $\texttt{I.pre} \wedge \texttt{J.core}$, and from $\texttt{I}_2 \xrightarrow{\texttt{O}} \texttt{J}_2$ if $\texttt{I.core} \wedge \texttt{J.late}$. Finally, we require that $(\texttt{I} \xrightarrow{\texttt{O}} \texttt{J}) \wedge \texttt{I.pre} \Rightarrow \neg\texttt{J.late}$ – otherwise, cross-iteration dependencies could be entirely and incorrectly cut by never setting I.core (a previous version of SLOTHY got this wrong and had astonishing optimization abilities). See Figure 4 for an illustration. Note in particular how the constraints that we "cut" are retained through symmetry.

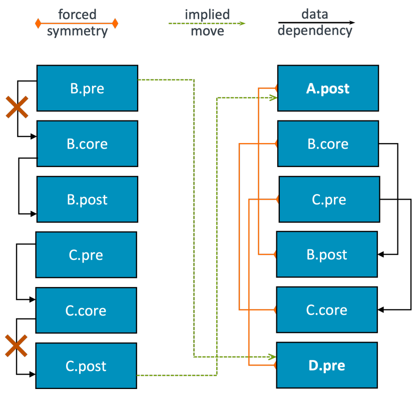

Figure 4: Illustration of how SLOTHY models software pipelining.

We do not model looping instructions themselves, such as counter decrements and compare+jumps: First, we don't expect those to have a meaningful impact on the optimization. Second, on Armv8.1-M+Helium in particular, inner iterations in low overhead loops do execute as if the loop had been unrolled, including the potential for instruction overlapping. By modelling the last and first instructions of two successive iterations as adjacent, HeLight$_{55}$ correctly takes into account the overlapping for those instructions.

3.5 Address modifications.

A typical loop will modify the base address for the data to be operated on with each iteration: For example, in Helium, data might be loaded via `VLDR Qdest, [Raddr]` for some address register `Raddr`, and later stored via `VSTR Qdst, [Radd], #16`, incrementing the address register for the next iteration. Such patterns make deep software pipelining impossible without further modelling of the semantics of the address increments, as there is a data dependency between the final store of one iteration, incrementing the address register, and the initial load for the next iteration. Since SLOTHY does not know about the semantics of instructions beyond their signature and dependencies, it cannot reorder load and store in this situation. *With* further semantic knowledge, however, one *could* of course reorder load and store in such situations, leveraging commutativity relations such as `VSTR Qa, [Rptr], #16; VLDR Vb, [Rptr]` ≡ `VLDR Qb, [Rptr, #16]; VSTR Qa, [Rptr], #16`.

We address this problem as follows: First, SLOTHY does not model *any* address increments, but always treats address registers for load/store operations as input-only. This gives SLOTHY the flexibility to freely reorder load/stores and identify complex software pipelining opportunities. Second, *after* the optimization, we then iterate through pairs of load/stores which depend on the same address register and have been reordered by SLOTHY, and fix up their address offsets, leveraging commutativity relations such as the one above. This approach works well for a variety of workloads, but it has some limitations, as we now explain.

First, we currently require that any address register is only modified by at most one load/store instruction per iteration. This is for simplicity only and could be easily overcome if necessary. Second, *manual* address increments via `add Raddr, Raddr, #16` do currently still prevent software pipelining in SLOTHY: For loops were such increments can not be realized as pre/post increments, further work on SLOTHY would be needed to optimize them. Third, and most importantly, SLOTHY's post-optimization address offset fix up assumes that there is no aliasing between memory pointed to by different address registers. In C terminology, it behaves as if every pointer was qualified as `restrict`.

Listing 3 shows a simple example for a software pipelining opportunity requiring the reordering of load/store instructions, and a corresponding address offset fix up.

3.6 Performance/μArch constraints: Latencies

To model performance of code on in-order microarchitectures, such as Cortex-M55 currently targeted by HeLight$_{55}$, we constrain the scheduling of instructions according to their latencies: If $\texttt{I} \overset{\texttt{O}}{\to} \texttt{J}$, we demand $\texttt{J.pos} \geq \texttt{I.pos} + \texttt{latency}(\texttt{I} \overset{\texttt{O}}{\to} \texttt{J})$, superseding the previously introduced functional correctness constraint $\texttt{J.pos} \geq \texttt{I.pos}$. We note that while $\texttt{latency}(\texttt{I} \overset{\texttt{O}}{\to} \texttt{J})$ often depends on `I` only, dedicated forwarding paths provide examples where $\texttt{latency}(\texttt{I} \overset{\texttt{O}}{\to} \texttt{J})$ may be smaller than the "generic" latency of `I`: In Cortex-M55, for example, while vector multiplication instructions typically have a latency of 2 cycles, sequences `VMULx; VSTRx` run stall-free.

The above approach to modelling latencies hinges on the program ordering `I.pos` being the same as the execution time unit of `I`. For microarchitectures which support multi-issuing for the code under consideration, further thought is required how to relate `I.pos` to a notion of execution time which can then be used to express latency constraints.

3.7 Performance/μArch constraints: Instruction overlapping

Instruction overlapping is modeled by assigning to each instruction `I` a *functional unit* `Unit(I)` (depending on the target microarchitecture) and a usage time `block`(I) capturing for how long `I` keeps `Unit(I)` busy. We then demand that for any fixed functional unit `U`, the set of intervals $\{[\texttt{I.pos}, \texttt{I.pos} + \texttt{block}(I)) \mid \texttt{Unit(I)} = \texttt{U}\}_{\texttt{I}}$ is non-overlapping. This is a straightforward instance of the job shop problem. If an instruction may run on multiple functional units, we create Boolean variables tracking where they actually run, and condition the usage intervals accordingly – that is, we reduce to a *flexible* job shop problem. On Cortex-M55, an example for an instruction that can run on multiple functional units is the double vector-to-register move `VMOV Ra, Rb, Qn[i], Qn[j]`.

We highlight that the latency of an instruction may be smaller than its usage time: On Cortex-M55, for example, many vector instructions (e.g. `VADD`) occupy their functional units for 2 cycles, but have 1 cycle latency: This is possible since instruction overlapping supports data dependencies between the overlapping instructions.

3.8 Memory, register pressure and stack spills

SLOTHY does not have a notion of memory: Store instructions are instructions without output, while load instructions are instructions without input. As a consequence, SLOTHY is not sound for code relying on memory as a temporary storage, as it does not track data dependencies through memory. To at least accommodate the optimization of code which relies on stack spills, the following trick can be used: Stack locations are modeled as a separate "register type" which are written to / read from via virtual instructions that that have the same properties as loads/stores (e.g. in terms of latencies and functional units used). With this simple trick, which fits into the existing framework of SLOTHY, data dependencies can be tracked across stack spills, without having to model linear memory.

3.9 Modelling stalls

We consider two approaches for modelling and minimizing stalls. First, "externally": We pad the input code with a number of `nop` instructions and then search for a perfect variant of the padded code. If such does not exist, we increase the `nop`-count and try again. Via binary search, we quickly find the minimum number of nops required. Alternatively, we model stalls internally as a flexible integer variable which defines the amount of "slack" allowed in the reordering of instructions: That is, rather than requiring that the reordering be a permutation of $\{0, 1, \ldots, \texttt{codesize}\}$, we define it as map $\{0, 1, \ldots, \texttt{codesize}\} \to \{0, 1, \ldots, \texttt{codesize} + \texttt{pad}\}$. We then ask the solver to minimize pad.

Our experiments suggest that modelling stalls "externally" is faster. Moreover, it allows us to set other optimization objectives, as we discuss next.

3.10 Other objectives

Beyond the minimization of stalls, the following are useful objectives: If software pipelining is enabled, it is natural to maximize $\sum_{\texttt{I}} \texttt{I.core}$ – that is, to minimize the amount of interleaving between successive iterations. Second, register usage can be minimized. This is particularly interesting if the code in question does not fit into the register file, and "virtual registers" are used to approximate the amount of stack spilling needed.

3.11 Soundness

There are two forms of soundness for SLOTHY: Functional soundness and performance soundness. The former means that SLOTHY emits code that is functionally equivalent to the original code. For the most part, this is easy to check: As we do not change instructions but only their order and register use, functional soundness amounts to the produced code permutation yielding an isomorphism of computational flow graphs – which is easily checked. However, our handling of address offsets (Section 3.5) has to be treated separately, as we elide address increments during load/stores when passing to computational flow graphs. We also emphasize again that address registers are assumed to be non-aliasing.

Performance soundness is more subtle: We do not expect SLOTHY to be used with fully exact microarchitectural models. Instead, one only models key software optimization aspects, such as instruction latencies or throughput, and optimizes code according to those. Unmodeled microarchitectural aspects may therefore still lead to stalls in code which the (micro)architectural model underlying SLOTHY considers "stall-free". It is thus important to validate the performance of code produced via SLOTHY.

4 HeLight$_{55}$: Assembly optimization for Cortex-M55

Section 3 explained our generic approach for modeling assembly optimization via constraint solving. We now describe our implementation for Armv8.1-M+Helium + Cortex-M55.

4.1 SLOTHY + (micro)architecture

HeLight$_{55}$ combines our optimization framework SLOTHY with architectural information about Armv8.1-M+Helium and microarchitectural aspects of Cortex-M55r1.

First, HeLight$_{55}$ models and provides parsing/writing functionality for a subset of Armv8.1-M+Helium: For now, it focuses on supporting basic Helium load/store/arithmetic instructions required for Section 5, Section 6, but it can easily be extended. What SLOTHY requires from HeLight$_{55}$ is knowledge of the available register files (here, the general purpose and vector register files, and potentially a "virtual" register file for stack spills Section 3.8), and the *signatures* of instructions – that is: The number and types of arguments, and the information of whether they are an input, an output, or a simultaneous input/output.

Second, HeLight$_{55}$ models basic aspects of the Cortex-M55 microarchitecture important for software optimization: The set of functional units and, for each instruction, the assignment of functional unit(s) it belongs to, as well as its latency and throughput. We

```
> ./helight-cli examples/simple0.s
INFO:Attempt with 0 stalls...
...
INFO:Status: INFEASIBLE
INFO:Attempt with 1 stalls...
...
INFO:helight:Status: OPTIMAL
vldrw.u32 q4, [inB], #16
nop
vldrw.u32 q0, [inA, #16]
vmulh.u32 q3, q0, q4
vldrw.u32 q2, [inA, #32]
vadd.u32  q3, q3, q3
vmulh.u32 q1, q2, q4
vadd.u32  q2, q3, q4
vldrw.u32 q0, [inA]
vadd.u32  q3, q1, q1
vmulh.u32 q1, q0, q4
vadd.u32  q3, q3, q4
vstrw.u32 q2, [inA, #16]
vadd.u32  q2, q1, q1
vstrw.u32 q3, [inA, #32]
vadd.u32  q3, q2, q4
vstrw.u32 q3, [inA], #48
```

```
> ./helight-cli examples/simple0.s --loop
INFO:Attempt with 0 stalls...
...
INFO:helight:Status: OPTIMAL
INFO:helight:Wall time: 0.659182 s
vmulh.u32 q3, q3, q2     // ....*...........
vadd.u32  q4, q4, q2     // ............*...
vldrw.u32 q0, [inA, #16] // .*..............
vadd.u32  q5, q3, q3     // .......*........
vstrw.u32 q4, [inA, #32] // ..............*.
vadd.u32  q5, q5, q2     // ........*.......
vmulh.u32 q1, q0, q2     // .....*..........
vldrw.u32 q3, [inA, #48] // e...............
vadd.u32  q0, q1, q1     // .........*......
vldrw.u32 q6, [inA, #80] // ..e.............
vadd.u32  q7, q0, q2     // ..........*.....
vldrw.u32 q2, [inB], #16 // ...e............
vmulh.u32 q1, q6, q2     // ......e.........
vstrw.u32 q7, [inA, #16] // .............*..
vadd.u32  q4, q1, q1     // ...........e....
vstrw.u32 q5, [inA], #48 // ...............*
```

Listing 3: Example for the usage of command line tool `helight-cli` to optimize the snippet from Listing 2 (left hand side), with and without software pipelining. Note how the address offsets for early loads have been adjusted on the right hand side.

also model some exceptional conditions such as special latencies for specific instruction pairs (such as `VMULx; VSTRx` which typically run stall-free even though an integer multiply instruction usually has 2 cycles latency), or `ST-LD` hazards described below.

4.2 Convenience functions

In addition to providing the mandatory (micro)architectural complement to SLOTHY, HeLight$_{55}$ provides a number of convenience functions which make it more practical to use than the bare bones optimization capabilities of SLOTHY.

First, when using software pipelining, HeLight$_{55}$ automatically optimizes the preamble and postamble of the loop and adds looping instructions to produce a working piece of assembly that's functionally equivalent to the original loop. Second, HeLight$_{55}$ can operate on specific parts of assembly files, making it convenient to maintain a set of functionally correct and readable, yet poorly performing reference source files, and have them auto-optimized to high performance assembly by HeLight$_{55}$. Third, HeLight$_{55}$ supports parsing and unfolding of assembly macros, as well as a mixture of register names: Undefined symbolic register names, defined symbolic register names (via `.req` directives), and raw register names. Listing 5 (right) shows an assembly snippet using symbolic register names.

4.3 Interface

SLOTHY and HeLight$_{55}$ are implemented in Python, building on the existing Python interface to Google OR-Tools. They can be used from within Python or through a small command line application, as we illustrate now.

HeLight$_{55}$ via the command line. For quick experimental optimization of small assembly snippets, `helight-cli` can be used. Listing 3 shows an example usage of `helight-cli` for the optimization of the naïve snippet from the left hand side of Listing 2, with and without software pipelining. Without software pipelining, we can see that a stall is unavoidable due to the initial `VLDR` bottleneck. With software pipelining, those loads can

```
helight = Helight()
helight.load_source_from_file("examples/naive/ntt_dilithium.s")
# Tell HeLight about the types of symbolic register names for which
# there is no unambiguous architectural typing.
# For example, in `vmul a, b, c`, c could be either a GPR or a vector.
helight.config.typing_hints = { r : RegisterType.GPR for r in
   [ "root0",         "root1",         "root2",
     "root0_twisted", "root1_twisted", "root2_twisted" ] }
# Optimize a specific loop in the source. This replaces the respective loop
# by its optimized version, padded with optimized preamble and postamble.
helight.optimize_loop("layer12_loop")
helight.print_code()
helight.write_source_to_file("examples/opt/ntt_dilithium.s")
```

Listing 4: Example for the usage of HeLight_{55} in Python.

be pulled into the previous iteration, enabling a stall-free variant (for Cortex-M55). The tool minimizes the interleaving, so there is no stall-free implementation with less than 5 early instructions. We also note how the address offsets have been adjusted as early VLDRs are reordered before VSTR with post-increment, as discussed in Section 3.5.

HeLight_{55} via Python. HeLight_{55} can also be used from Python. For example, Listing 4 shows a script to optimize the layer12 loop from Listing 8 (further discussed below). For more information, we refer to the source code.

5 Example: Fast Fourier Transform

5.1 Introduction

The *Fourier Transform* is a transformation for the decomposition of signals into frequency components. It has numerous incarnations – such as Fourier series or the Number Theoretic Transform (NTT) discussed in Section 6 – and a vast range of applications. Simply put, the importance of the Fourier Transform cannot be overstated. Here, we are interested in the Fourier Transform over the complex numbers, which can be viewed as $\mathbb{C}^n \to \mathbb{C}^n, (x_i) \mapsto (\sum_j x_i \zeta_n^{ij})$ where $\zeta_n = \exp(2\pi i/n)$ is the standard primitive n-th root of unity (algebraically, this is the same as the splitting $(\mathrm{ev}_{\zeta_n^i}) : \mathbb{C}[X]/(X^n - 1) \xrightarrow{\cong} \prod_i \mathbb{C}[X]/(X - \zeta_n^i))$.

The *Fast Fourier Transform* (FFT) is a method for the fast computation of the Fourier Transform. While the above description of the Fourier Transform suggests quadratic complexity, the FFT splits the computation into a logarithmic number of *layers* of linear complexity each, giving an overall complexity of $\mathcal{O}(n \log n)$. Each FFT layer operates on strided blocks of r data units via so-called *butterflies*, and r is called the *radix* of the FFT. Common choices are radix-2 and radix-4. Efficient implementations of the FFT are an essential component of any Digital Signal Processing (DSP) library, with complex numbers presented in either floating point or fixed point format.

In this section, we look at the optimization of a radix-4 layer of a fixed-point FFT in Armv8.1-M+Helium: In Section 5.2, we show an implementation in C intrinsics and look at the resulting assembly when compiled with Arm Compiler 6.18. In Section 5.3, we then use HeLight_{55} to optimize an assembly version of the same code, and compare the results.

5.2 Radix-4 fixed-point FFT in intrinsics and handwritten assembly

Listing 5 (left) shows one layer of a radix-4 fixed-point FFT written in C Helium intrinsics taken from [Arme][1]. The right hand side shows the same code naïvely translated into

[1]Link to C intrinsics based implementation of radix-4 fixed-point FFT

```c
#define MVE_CMPLX_MULT_FX_AxB(A,B)              \
  vqdmladhxq(vqdmlsdhq(                          \
     vuninitializedq(A), A, B), A, B)
#define MVE_CMPLX_SUB_FX_A_ixB(A,B)             \
   vhcaddq_rot270(A,B)
#define MVE_CMPLX_ADD_FX_A_ixB(A,B)             \
   vhcaddq_rot90(A,B)
vA = vld1q(inA); vC = vld1q(inC);
while (blkCnt > 0U) {
  vB = vld1q(inB); vD = vld1q(inD);
  vSm0 = vhaddq(vA, vC);
  vDf0 = vhsubq(vA, vC);
  vSm1 = vhaddq(vB, vD);
  vDf1 = vhsubq(vB, vD);
  vTmp0 = vhaddq(vSm0, vSm1);
  vst1q(inA, vTmp0); inA += 4;
  vTmp0 = vhsubq(vSm0, vSm1);
  vW = vld1q(pW2); pW2 += 4;
  vTmp1 = MVE_CMPLX_MULT_FX_AxB(vW, vTmp0);
  vst1q(inB, vTmp1); inB += 4;
  vTmp0 = MVE_CMPLX_SUB_FX_A_ixB(vDf0, vDf1);
  vW = vld1q(pW1); pW1 += 4;
  vTmp1 = MVE_CMPLX_MULT_FX_AxB(vW, vTmp0);
  vst1q(inC, vTmp1); inC += 4;
  vTmp0 = MVE_CMPLX_ADD_FX_A_ixB(vDf0, vDf1);
  vW = vld1q(pW3); pW3 += 4;
  vTmp1 = MVE_CMPLX_MULT_FX_AxB(vW, vTmp0);
  vst1q(inD, vTmp1); inD += 4;
  vA = vld1q(inA); vC = vld1q(inC);
  blkCnt--;
}
```

```asm
loop_start:
 vldrw.s32      vA,   [inA]
 vldrw.s32      vC,   [inC]
 vldrw.s32      vB,   [inB]
 vldrw.s32      vD,   [inD]
 vhadd.s32      vSm0, vA,    vC
 vhsub.s32      vDf0, vA,    vC
 vhadd.s32      vSm1, vB,    vD
 vhsub.s32      vDf1, vB,    vD
 vhadd.s32      vT0,  vSm0,  vSm1
 vstrw.s32      vT0,  [inA], #16
 vhsub.s32      vT0,  vSm0,  vSm1
 vldrw.s32      vW,   [pW2], #16
 vqdmladhx.s32  vT1,  vW,    vT0
 vqdmlsdh.s32   vT1,  vW,    vT0
 vstrw.s32      vT1,  [inB], #16
 vhcadd.s32 vT0, vDf0, vDf1, #270
 vldrw.s32      vW,   [pW1], #16
 vqdmladhx.s32  vT1,  vW,    vT0
 vqdmlsdh.s32   vT1,  vW,    vT0
 vstrw.s32      vT1,  [inC], #16
 vhcadd.s32 vT0, vDf0, vDf1, #90
 vldrw.s32      vW,   [pW3], #16
 vqdmladhx.s32  vT1,  vW,    vT0
 vqdmlsdh.s32   vT1,  vW,    vT0
 vstrw.s32      vT1,  [inD], #16
 le lr, loop_start
loop_end:
```

Listing 5: Left: One layer of radix-4 fixed-point FFT using C intrinsics. Right: Straightforward translation into pseudo-assembly using symbolic register names.

pseudo-assembly, using symbolic register names; this will be the input to HeLight$_{55}$ below.

Listing 6 (left) shows the disassembly of the result of compiling Listing 5 (left) using Arm Compiler 6.18. A `//-` annotation indicates a structural hazard, while `//?` indicates the risk of a *`ST-LD`-hazard* resulting from a `VSTR; xxx; VLDR` sequence – whether such a sequence actually leads to a stall depends on data alignment. Overall, we achieve good instruction overlapping, but (unsurprisingly) there is potential for improvement left.

Next, Listing 6 (right) shows handwritten assembly from [Armb][2]. We observe only one structural hazard on Cortex-M55, but three alignment dependent `ST-LD` hazards.

5.3 HeLight$_{55}$-optimized assembly implementation

Listing 7 shows the result of optimizing Listing 5 (right) through HeLight$_{55}$. The comments indicate where instructions were originally placed, and if they are early instructions for the next iteration. We see that in each iteration, *seven* instructions are being pulled into the previous iteration, enabling a *stall-free* optimized loop body – which we confirm by running the code on real hardware. Further, HeLight$_{55}$ has extracted and separately optimized the loop preamble and postamble. Little reordering was necessary for postamble, which is expected since the original postamble is a subset of the perfectly optimized loop body.

6 Example: Number Theoretic Transform

6.1 Introduction

The *Number Theoretic Transform* (NTT) is a variant of the Fourier Transform that's defined over the integers rather than the complex numbers: While the latter splits $\mathbb{C}[X]/(X^n - 1)$ as $\prod_i \mathbb{C}[X]/(X - \zeta_n^i)$ for the complex n-th root of unity $\zeta_n = \exp(2\pi i/n)$, the NTT splits

[2]Link to handwritten assembly implementation of radix-4 fixed-point FFT

```
.LBB0_7:
  vldrw.u32     q0, [r8]          //-
  vldrw.u32     q1, [r1]          //-
  vhadd.s32     q3, q1, q0
  vldrw.u32     q2, [r2]
  vhsub.s32     q0, q1, q0
  vldrw.u32     q4, [r0]
  vhadd.s32     q5, q2, q4
  vhadd.s32     q6, q3, q5        //-
  vstrb.8       q6, [r1], #16
  vhsub.s32     q3, q3, q5
  vldrw.u32     q5, [r9], #16     //?
  vqdmlsdh.s32  q6, q5, q3
  vhsub.s32     q1, q2, q4
  vqdmladhx.s32 q6, q5, q3
  vstrb.8       q6, [r2], #16
  vhcadd.s32    q2, q0, q1, #270
  vldrw.u32     q3, [r7], #16     //?
  vqdmlsdh.s32  q4, q3, q2
  vqdmladhx.s32 q4, q3, q2        //-
  vstrb.8       q4, [r8], #16
  vhcadd.s32    q2, q0, q1, #90
  vldrw.u32     q0, [r11], #16    //?
  vqdmlsdh.s32  q1, q0, q2
  vqdmladhx.s32 q1, q0, q2        //-
  vstrb.8       q1, [r0], #16
  le lr, .LBB0_7
```

```
  vldrw.32      q1, [in0]
  vldrw.32      q6, [in2]
2:
  vhadd.s32     q0, q1, q6
  vldrw.32      q4, [in1]         //?
  vhsub.s32     q2, q1, q6
  vldrw.32      q5, [in3]
  vhadd.s32     q1, q4, q5
  vhsub.s32     q3, q4, q5        //-
  vldrw.32      q7, [t1], #16
  vhadd.s32     q4, q0, q1
  vstrw.32      q4, [in0], #16
  vhsub.s32     q4, q0, q1
  vldrw.32      q5, [t0], #16     //?
  vqdmlsdh.s32  q0, q4, q5
  vhcadd.s32    q6, q2, q3, #270
  vqdmladhx.s32 q0, q4, q5
  vstrw.32      q0, [in1], #16
  vqdmlsdh.s32  q0, q6, q7
  vldrw.32      q1, [in0]         //?
  vqdmladhx.s32 q0, q6, q7
  vstrw.32      q0, [in2], #16
  vhcadd.s32    q4, q2, q3, #90
  vldrw.32      q5, [t2], #16
  vqdmlsdh.s32  q0, q4, q5
  vldrw.32      q6, [in2]
  vqdmladhx.s32 q0, q4, q5
  vstrw.32      q0, [in3], #16
  le            lr, 2b
```

Listing 6: Left: Compilation of Listing 5 (left) using Arm Compiler 6.18. Right: Handwritten implementation of FFT from github.com/ARM-Software/EndpointAI.

a modular polynomial ring $\mathbb{F}_q[X]/(X^n - 1)$ as $\prod_i \mathbb{F}_q[X]/(X - \omega^i)$, for $\omega \in \mathbb{F}_q$ a *modular* primitive n-th root of unity, that is, $\omega^n = 1$ modulo q, but $\omega^i \neq 1$ modulo q for all $i < n$.

Structurally, the NTT is the same as the Fourier Transform, and in particular can be implemented in $\mathcal{O}(n \log n)$ time through a variant of the Fast Fourier Transform. However, the underlying coefficient arithmetic relies on modular integer arithmetic rather than floating point or fixed point arithmetic.

Fast implementations of the Number Theoretic Transform are essential for high performance implementations of post-quantum cryptography, which use the NTT for polynomial multiplication (using the analogue of the convolution theorem in digital signal processing). The recently designated winners Kyber and Dilithium [SAB$^+$22],[LDK$^+$22] of the NIST Post-Quantum Cryptography standardization project [NIS16] both rely on the NTT.

Basic aspects of NTT implementations are the merging of layers (depending on register pressure), the growth of coefficients and insertion of modular reductions (depending on the underlying prime), and the underlying primitive for modular multiplication.

6.2 Previous implementations

[BHK$^+$21] explores how to map modular arithmetic in the NTT to the Arm architecture, including Armv8.1-M+Helium. [BBMK$^+$21, BHK$^+$22] leverage those primitives for complete implementations of 32-bit NTTs in Helium. These implementations keep the data in vector registers but load constants in general purpose registers, allowing them to maintain a good balance between the register files, and to merge two radix-2 layers at a time.

Coming up with the implementations from [BBMK$^+$21, BHK$^+$22] appears challenging due to the complex interleaving necessary to achieve good overlapping. Loc.cit. achieves this through loop unrolling and scripted register usage tracking. While the resulting code is of very high performance, it is difficult both to read and to adapt to other contexts.

In the following, we demonstrate how HeLight$_{55}$ can be used to derive high performance Helium assembly implementations for the NTT from *readable*, handwritten base implementations. Beyond being considerably less effort and easier to reproduce, the resulting

```
vldrw.s32 q0, [inB]           //*........
nop                           //.......*.
vldrw.s32 q2, [inD]           //....*....
vhadd.s32 q1, q0, q2          //......*..
vldrw.s32 q7, [inA]           //..*......
nop                           //........*
vldrw.s32 q4, [inC]           //.*.......
vhadd.s32 q6, q7, q4          //...*.....
vldrw.s32 q5, [pW2] , #16     //.....*...
loop_start:
  vhadd.s32 q3, q6, q1        //........*................
  vstrw.u32 q3, [inA] , #16   //.........*...............
  vhsub.s32 q6, q6, q1        //..........*..............
  vqdmladhx.s32 q3, q5, q6    //............*............
  vldrw.s32 q1, [pW1] , #16   //................*........
  vqdmlsdh.s32 q3, q5, q6     //.............*...........
  vldrw.s32 q5, [pW3] , #16   //.....................*...
  vhsub.s32 q6, q0, q2        //.......*.................
  vldrw.s32 q0, [inB, #16]    //..e......................
  vhsub.s32 q2, q7, q4        //.....*...................
  vstrw.u32 q3, [inB] , #16   //..............*..........
  vhcadd.s32 q7, q2, q6, #270 //...............*.........
  vqdmladhx.s32 q3, q1, q7    //.................*.......
  vldrw.s32 q4, [inC, #16]    //.e.......................
  vqdmlsdh.s32 q3, q1, q7     //..................*......
  vldrw.s32 q7, [inA]         //e........................
  vhcadd.s32 q1, q2, q6, #90  //....................*....
  vstrw.u32 q3, [inC] , #16   //...................*.....
  vqdmladhx.s32 q3, q5, q1    //......................*..
  vhadd.s32 q6, q7, q4        //....e....................
```

```
  vldrw.s32 q2, [inD, #16]    //...e.....................
  vqdmlsdh.s32 q3, q5, q1     //.......................*.
  vldrw.s32 q5, [pW2] , #16   //...........e.............
  vhadd.s32 q1, q0, q2        //......e..................
  vstrw.u32 q3, [inD] , #16   //........................*
  le lr, loop_start
loop_end:
vhadd.s32 q3, q6, q1        //*...................
vstrw.u32 q3, [inA] , #16   //.*..................
vhsub.s32 q6, q6, q1        //..*.................
vqdmladhx.s32 q3, q5, q6    //...*................
vldrw.s32 q1, [pW1] , #16   //....*...............
vqdmlsdh.s32 q3, q5, q6     //.....*..............
vhsub.s32 q6, q0, q2        //.......*............
vldrw.s32 q5, [pW3] , #16   //......*.............
vhsub.s32 q2, q7, q4        //........*...........
vstrw.u32 q3, [inB] , #16   //.........*..........
vhcadd.s32 q7, q2, q6, #270 //..........*.........
vqdmladhx.s32 q3, q1, q7    //...........*........
nop                         //...................*
vqdmlsdh.s32 q3, q1, q7     //............*.......
vstrw.u32 q3, [inC] , #16   //..............*.....
vhcadd.s32 q1, q2, q6, #90  //.............*......
vqdmladhx.s32 q3, q5, q1    //...............*....
nop                         //..................*.
vqdmlsdh.s32 q3, q5, q1     //................*...
vstrw.u32 q3, [inD] , #16   //.................*..
```

Listing 7: HeLight$_{55}$ optimization of pseudocode listing Listing 5 (right) for radix-4 fixed-point FFT. No stalls remain on Cortex-M55 for the core of the loop.

code is also much smaller than that in the previous works.

We target the NTTs used for the post-quantum cryptography schemes Kyber and Dilithium. Kyber uses a 7-layer incomplete 16-bit NTT with $n = 256$ and $q = 3329$. Dilithium uses an 8-layer complete 32-bit NTT with $n = 256$ and $q = 2^{23} - 2^{13} + 1$.

6.3 Intrinsics-based implementation

Listing 12 shows an implementation of the last two layers of the Kyber NTT using C intrinsics for Helium (left), as well as the resulting assembly after compilation with Arm Compiler 6.18. We observe again how the compiler manages to interleave the different kinds of instructions well, but that without software pipelining for this loop, the final block of `VST4x` constitutes a large bottleneck. We also annotate by `//l` two instances of a stall because of the 2-cycle latency of multiply instructions.

6.4 Readable base implementations

We now comment on the handwritten assembly that we will use as input to HeLight$_{55}$.

Initial layers of Dilithium. Listing 8 shows a naïve implementation of the first two layers of the forward NTT for Dilithium, using macros to keep the implementation readable. While readable, however, it performs very poorly: The back-to-back instances of load/store and multiplication instructions mean that only little use is made of instruction overlapping – see the unfolded version of the code displayed on Listing 8 (right).

Last layers of Kyber. The most challenging part in the NTT are layers which require intra-vector shuffling, like the last two layers in the Kyber NTT displayed in Listing 9 (right): We notice that `VLD4x` and `VST4x` are used for (de)interleaving at load/store time, and that additional vector registers are needed for the roots. This puts significant pressure on the vector register file: Not only are less vectors available for the actual data, but the arguments of `VLD4x` and `VST4x` are constrained to {`Qi,Q(i+1),Q(i+2),Q(i+3)`} for $i = 0, 1, 2, 3, 4$. Listing 9 also shows an alternative where the `VLD4x` in layers 6,7 are

```
// Barrett multiplication
.macro mulmod dst, src, c, c_twist
  vmul.s32     \dst,  \src, \c
  vqrdmulh.s32 \src,  \src, \c_twist
  vmla.s32     \dst,  \src, q
.endm
// Cooley-Tukey butterfly
.macro ct_butterfly a, b, r, r_twist
  mulmod    tmp, \b, \r, \r_twist
  vsub.u32  \b,  \a, tmp
  vadd.u32  \a,  \a, tmp
.endm
...
  ldrd r0, r0_twist, [r_ptr], #+8
  ldrd r1, r1_twist, [r_ptr], #+8
  ldrd r2, r2_twist, [r_ptr], #+8
layer12_loop:
  vldrw.u32 d0, [in_lo]
  vldrw.u32 d1, [in_lo, #256]
  vldrw.u32 d2, [in_hi]
  vldrw.u32 d3, [in_hi, #256]
  ct_butterfly d0,d2,r0,r0_twist
  ct_butterfly d1,d3,r0,r0_twist
  ct_butterfly d0,d1,r1,r1_twist
  ct_butterfly d2,d3,r2,r2_twist
  vstrw.u32 d0, [in_lo],#16
  vstrw.u32 d1, [in_lo, #-240]
  vstrw.u32 d2, [in_hi],#16
  vstrw.u32 d3, [in_hi, #-240]
  le lr, layer12_loop
layer12_loop_end:
```

```
layer12_loop:
  vldrw.u32    d0, [in_lo]
  vldrw.u32    d1, [in_lo, #256]  //-
  vldrw.u32    d2, [in_hi]        //-
  vldrw.u32    d3, [in_hi, #256]  //-
  vmul.s32     tmp,   d2, r0
  vqrdmulh.s32 d2, d2, r0_twist   //-
  vmla.s32     tmp,   d2, q       //-
  vsub.u32     d2, d0, tmp
  vadd.u32     d0, d0, tmp        //-
  vmul.s32     tmp,   d3, r0
  vqrdmulh.s32 d3, d3, r0_twist   //-
  vmla.s32     tmp,   d3, q       //-
  vsub.u32     d3, d1, tmp
  vadd.u32     d1, d1, tmp        //-
  vmul.s32     tmp,   d1, r1
  vqrdmulh.s32 d1, d1, r1_twist   //-
  vmla.s32     tmp,   d1, q       //-
  vsub.u32     d1, d0, tmp
  vadd.u32     d0, d0, tmp        //-
  vmul.s32     tmp,   d3, r2
  vqrdmulh.s32 d3, d3, r2_twist   //-
  vmla.s32     tmp,   d3, q       //-
  vsub.u32     d3, d2, tmp
  vadd.u32     d2, d2, tmp        //-
  vstrw.u32    d0, [in_lo],#16
  vstrw.u32    d1, [in_lo, #-240] //-
  vstrw.u32    d2, [in_hi],#16    //-
  vstrw.u32    d3, [in_hi, #-240] //-
  le lr, layer12_loop
layer12_loop_end:
```

Listing 8: Naïve implementation of two merged layers of a 32-bit, radix-2 Number Theoretic Transform. Left: Using macros for readability. Right: Unfolded.

replaced by plain `VLDR`s, and where we instead use `VST4x` in layers 4,5, thereby spreading the complexity of `VLD4x`/`VST4x` over two separate loops.

Merging three layers. We also study merging *three* radix-2 NTT layers a time, which for Helium has not been considered before. Merging layers is natural to save load/store operations, and particularly attractive for Kyber, which has an odd number of layers.

Despite the limited amount of 8 vector registers, we find that layers $4, 5, 6$ in Dilithium and $3, 4, 5$ in Kyber can be merged with only 3 stack spills per iterations – a complete load/store sequence would amount to 8 spills. Merging layers $3, 4, 5$ in Kyber allows to split its 7-layer NTT into $2 + 3 + 2$ layers. For Dilithium, we also merge layers $1, 2, 3$, but we find that large pressure also on the *general purpose registers* requires 3 more spills. Listing 10 shows a symbolic implementation layers $3, 4, 5$ of a Dilithium NTT.

6.5 Auto-optimized implementation

We now comment on the HeLight$_{55}$ optimizations of our base implementations.

Initial layers of Dilithium. Listing 11 shows the result of optimizing the first two layers of the Dilithium NTT (Listing 8) through HeLight$_{55}$. As before, comments indicate where instructions were originally placed, and if they are early instructions for the next iteration – this time, five instructions are being pulled into the previous iteration. The body of the loop is stall-free, while the preamble and postamble have stalls, which should be attempted to be filled through further interleaving with the surrounding code.

Last layers of Kyber. We next look at the last four layers of the Kyber NTT (Listing 9). As explained before, they are very difficult to schedule due to the constraints on `VLD4`,

```
layer45_loop:
  load_next_roots
  vldrw.u32 d0, [in]
  vldrw.u32 d1, [in, #16]
  vldrw.u32 d2, [in, #32]
  vldrw.u32 d3, [in, #48]
  ct_butterfly d0, d2, r0, r0_tw
  ct_butterfly d1, d3, r0, r0_tw
  ct_butterfly d0, d1, r1, r1_tw
  ct_butterfly d2, d3, r2, r2_tw
  vstrw.u32 d0, [in], #64
  vstrw.u32 d1, [in, #-48]
  vstrw.u32 d2, [in, #-32]
  vstrw.u32 d3, [in, #-16]
  // ALTERNATIVE:
  // vst40.u32 {d0, d1, d2, d3}, [in]
  // vst41.u32 {d0, d1, d2, d3}, [in]
  // vst42.u32 {d0, d1, d2, d3}, [in]
  // vst43.u32 {d0, d1, d2, d3}, [in]!
  le lr, layer45_loop
layer45_loop_end:
```

```
layer67_loop:
  vld40.u32 {d0, d1, d2, d3}, [in]
  vld41.u32 {d0, d1, d2, d3}, [in]
  vld42.u32 {d0, d1, d2, d3}, [in]
  vld43.u32 {d0, d1, d2, d3}, [in]
  // ALTERNATIVE:
  // vldrw.u32 di, [in, #...] i=0,1,2,3
  vldrh.u16 r0,    [r_ptr] ,#96
  vldrh.u16 r0_tw, [r_ptr, #-80]
  ct_butterfly d0, d2, r0, r0_tw
  ct_butterfly d1, d3, r0, r0_tw
  vldrh.u16 r1,    [r_ptr, #-64]
  vldrh.u16 r1_tw, [r_ptr, #-48]
  ct_butterfly d0, d1, r1, r1_tw
  vldrh.u16 r2,    [r_ptr, #-32]
  vldrh.u16 r2_tw, [r_ptr, #-16]
  ct_butterfly d2, d3, r2, r2_tw
  vst40.u32 {d0, d1, d2, d3}, [in]
  vst41.u32 {d0, d1, d2, d3}, [in]
  vst42.u32 {d0, d1, d2, d3}, [in]
  vst43.u32 {d0, d1, d2, d3}, [in]!
  le lr, layer67_loop
layer67_loop_end:
```

Listing 9: Naïve implementation of last four layers of a Kyber NTT. The use of `VLD4x/VST4x` puts pressure on the vector register file.

`VST4`. Indeed, when we run HeLight$_{55}$ on it, it runs out of time without finding a solution.

However, for the alternative variant in Listing 9 – the one trading `VLD4x` in layer 6,7 for `VST4x` in layer 4,5 – there is indeed a stall-free version on Cortex-M55: First, running HeLight$_{55}$ unmodified does not find a solution. However, HeLight$_{55}$ by default uses an overapproximation of the `ST-LD`-hazard, forbidding instances of `VSTx; ?; VLDx` because they *may* lead to memory bank conflicts depending on data alignment. Ignoring `ST-LD` hazards leads to the solution in Listing 13: This solution *does* have instances of the `ST-LD` patterns, but the displayed ordering of `VST4x` does not actually trigger any bank conflicts. Even though HeLight$_{55}$ minimizes interleaving, we need *nine* early instructions.

Triple merged layers. Optimizing our Kyber and Dilithium NTT implementations with $3+3+2$ and $7 = 2+3+2$ layer splittings, respectively, takes considerably longer than the $2+2+2+2$ and $1+2+2$ splittings: between 5 and 20 minutes on our machine. However, we find that the triple merged layers admit almost stall-free interleavings even *without* software pipelining (some ST-LD hazards remain). This leads to compact code, and the lost cycles are compensated for by loads and stores saved by merging three layers.

7 Results

Table 1 compares our HeLight$_{55}$-generated code to prior art. Beyond performance, we consider readability and μarch-flexibility as metrics for the practicality of code.

For the complex fixed-point FFT, we observe a speedup of 14% compared to intrinsics, and 5.7% compared to prior handwritten assembly. Given that the FFT is an exceptionally important and well-studied workload for DSP, we consider this improvement proof of the capabilities of HeLight$_{55}$. Our code is readable and the approach adaptable to other (Helium-based) microarchitectures, similar to intrinsics-based code.

For the 32-bit Dilithium NTT, we compare our $3+3+2$-layer and $2+2+2+2$-layer implementations to the 32-bit NTT from [BBMK+21] and to the Cortex-M4 implementation from [AHKS22]. For 16-bit Kyber NTT, we compare our $1+2+2+2$-layer and $2+3+2$-layer implementations to the recent Cortex-M4 implementations from [AHKS22, HZZ+22]; there is no prior implementation in Helium.

```
layer456_loop:
  ldrd rt0, rt0_tw, [r_ptr],#(7*8)
  ldrd rt1, rt1_tw, [r_ptr, #(-6*8)]
  ldrd rt2, rt2_tw, [r_ptr, #(-5*8)]
  ldrd rt3, rt3_tw, [r_ptr, #(-4*8)]
  ldrd rt4, rt4_tw, [r_ptr, #(-3*8)]
  ldrd rt5, rt5_tw, [r_ptr, #(-2*8)]
  ldrd rt6, rt6_tw, [r_ptr, #(-1*8)]
  vldrw.32 data0, [in]
  vldrw.32 data1, [in, #16]
  vldrw.32 data2, [in, #32]
  vldrw.32 data3, [in, #48]
  vldrw.32 data4, [in, #64]
  vldrw.32 data5, [in, #80]
  vldrw.32 data6, [in, #96]
  vldrw.32 data7, [in, #112]
  ct_butterfly data0, data4, rt0, rt0_tw
  ct_butterfly data1, data5, rt0, rt0_tw
  ct_butterfly data2, data6, rt0, rt0_tw
  ct_butterfly data3, data7, rt0, rt0_tw
  qsave QSTACK4, data4
  qsave QSTACK5, data5
  qsave QSTACK6, data6
```

```
  ct_butterfly data0, data2, rt1, rt1_tw
  ct_butterfly data1, data3, rt1, rt1_tw
  ct_butterfly data0, data1, rt2, rt2_tw
  ct_butterfly data2, data3, rt3, rt3_tw
  vstrw.32 data0, [in], #128
  vstrw.32 data1, [in, #(-128+16)]
  vstrw.32 data2, [in, #(-128+32)]
  vstrw.32 data3, [in, #(-128+48)]
  qrestore data4, QSTACK4
  qrestore data5, QSTACK5
  qrestore data6, QSTACK6
  ct_butterfly data4, data6, rt4, rt4_tw
  ct_butterfly data5, data7, rt4, rt4_tw
  ct_butterfly data4, data5, rt5, rt5_tw
  ct_butterfly data6, data7, rt6, rt6_tw
  vstrw.32 data4, [in, #(-128+64)]
  vstrw.32 data5, [in, #(-128+80)]
  vstrw.32 data6, [in, #(-128+96)]
  vstrw.32 data7, [in, #(-128+112)]
  le lr, layer456_loop
```

Listing 10: Symbolic implementation of three merged layers of a Dilithium NTT. Three stack spills are introduced to ensure realizability within the Armv8.1-M register files.

We see that our Dilithium NTT matches the performance of that from [BBMK+21], while achieving a much smaller code size and ensuring maintainability through the automated derivation from a readable base implementation. We do not consider the implementation [BBMK+21] practical from the latter perspective.

For our Kyber NTT, we observe a speedup of 6.4× compared to the Cortex-M4 implementation of [AHKS22] and of 4.8× compared to [HZZ+22], while maintaining readability and code compactness.

8 Conclusion

In this work, we have demonstrated the practicality of the use of constraint programming for the superoptimization of instruction scheduling, register allocation and software pipelining (periodic loop interleaving). Based on the CP-SAT solver from Google OR-Tools, we developed a (micro)architecture agnostic core SLOTHY and an instantiation HeLight$_{55}$ for Armv8.1-M+Helium and Cortex-M55r1. We showcased the power of our approach by optimizing, in a matter of seconds to minutes, two complex workloads from Digital Signal Processing and Post-Quantum Cryptography. We believe that SLOTHY + HeLight$_{55}$ can be generalized and applied to other (micro)architectures and workloads.

For Armv8.1-M+Helium and Cortex-M55 we confirm once again that very efficient implementations exist for critical and complex workloads, despite the numerous software constraints to be solved. Somewhat surprisingly, we even identify highly non-trivial opportunities for software pipelining despite the comparatively low number of vector registers, further increasing performance while maintaining a compact code-size.

We also consider the use of intrinsics as a more convenient and generally perferred alternative to handwritten assembly, and find very good results in terms of the compiler's leverage of instruction interleaving (using Arm Compiler 6.18). Yet, we consider our superoptimization approach as a viable alternative for code that's of very high complexity or for which it's essential to unlock the last % of performance. It is also a useful vehicle to understand the implications of different (micro)architectural properties on theoretical performance limits.

```
vldrw.u32 q0, [in_hi, #256]        //.*....
vqrdmulh.s32 q5, q0, r0_tw         //...*..
vldrw.u32 q3, [in_lo, #256]        //*.....
vmul.s32 q0, q0, r0                //..*...
nop                                //.....*
vmla.s32 q0, q5, modulus           //....*.
layer12_loop:
  vsub.u32 q4, q3, q0         //............*...............
  vmul.s32 q6, q4, r2         //...................*........
  vldrw.u32 q1, [in_hi]       //..*.........................
  vmul.s32 q5, q1, r0         //....*.......................
  vadd.u32 q7, q3, q0         //.............*..............
  vqrdmulh.s32 q3, q1, r0_tw  //.....*......................
  vldrw.u32 q0, [in_lo]       //*...........................
  vmla.s32 q5, q3, modulus    //......*.....................
  vldrw.u32 q3, [in_lo, #272] //.e..........................
  vmul.s32 q1, q7, r1         //..............*.............
  vsub.u32 q2, q0, q5         //.......*....................
  vqrdmulh.s32 q7, q7, r1_tw  //...............*............
  vadd.u32 q5, q0, q5         //........*...................
  vmla.s32 q1, q7, modulus    //................*...........
  vldrw.u32 q7, [in_hi, #272] //...e........................
  vqrdmulh.s32 q0, q4, r2_tw  //....................*.......
  vadd.u32 q4, q5, q1         //..................*.........
  vmla.s32 q6, q0, modulus    //.....................*......
  vstrw.u32 q4, [in_lo], #16  //........................*...
  vadd.u32 q4, q2, q6         //.......................*....
  vstrw.u32 q4, [in_hi], #16  //..........................*.
  vmul.s32 q0, q7, r0         //.........e..................
  vsub.u32 q5, q5, q1         //.................*..........
  vqrdmulh.s32 q1, q7, r0_tw  //..........e.................
  vstrw.u32 q5, [in_lo, #240] //.........................*..
  vsub.u32 q7, q2, q6         //......................*.....
  vmla.s32 q0, q1, modulus    //...........e................
  vstrw.u32 q7, [in_hi, #240] //...........................*
  le lr, layer12_loop
```

```
layer12_loop_end:
  vldrw.u32 q4, [in_hi]       //..*.....................
  vmul.s32 q5, q4, r0         //...*....................
  vadd.u32 q1, q3, q0         //....*...................
  vmul.s32 q6, q1, r1         //........*...............
  vsub.u32 q7, q3, q0         //*.......................
  vqrdmulh.s32 q3, q4, r0_tw  //.....*..................
  vldrw.u32 q0, [in_lo]       //......*.................
  vmla.s32 q5, q3, modulus    //.......*................
  nop                         //.......................*
  vqrdmulh.s32 q1, q1, r1_tw  //..........*.............
  vsub.u32 q2, q0, q5         //.........*..............
  vmla.s32 q6, q1, modulus    //............*...........
  vadd.u32 q4, q0, q5         //...........*............
  vmul.s32 q5, q7, r2         //.*......................
  vadd.u32 q1, q4, q6         //..............*.........
  vstrw.u32 q1, [in_lo], #16  //................*.......
  vqrdmulh.s32 q0, q7, r2_tw  //.............*..........
  vsub.u32 q1, q4, q6         //...................*....
  vmla.s32 q5, q0, modulus    //...............*........
  vstrw.u32 q1, [in_lo, #240] //....................*...
  vsub.u32 q1, q2, q5         //.....................*..
  vstrw.u32 q1, [in_hi, #256] //......................*.
  vadd.u32 q1, q2, q5         //.................*......
  vstrw.u32 q1, [in_hi], #16  //..................*.....
```

Listing 11: Optimal (for Cortex-M55) scheduling for two layers of the NTT for CRYSTALS-Dilithium. Automatically derived via HeLight$_{55}$ from Listing 8.

```
static void ct_butterfly( int16x8_t* d0,
  int16x8_t* d1, int16x8_t r, int16x8_t r_tw)
{
  int16x8_t tmp; const int16_t mod = -3329;
  tmp     = *d1 * r;
  *d1   = vqrdmulhq(*d1, r_tw);
  tmp    += *d1 * mod;
  *d1   = *d0 - tmp;
  *d0   = *d0 + tmp;
}
void ntt_kyber_layer67_loop_intr(
          int16_t* in, int16_t* r_ptr)
{
  int16x8x4_t data;
  int16x8_t r0, r0_tw, r1, r1_tw; r2, r2_tw;
  for (int i = 0; i < 8; i++) {
    data.val[0] = vld1q(in);
    data.val[1] = vld1q(&in[16 / 2]);
    data.val[2] = vld1q(&in[32 / 2]);
    data.val[3] = vld1q(&in[48 / 2]);
    r0 = vld1q(r_ptr);
    r0_tw = vld1q(r_ptr + 8);
    ct_butterfly(&data.val[0], &data.val[2],
                 r0, r0_tw);
    ct_butterfly(&data.val[1], &data.val[3],
                 r0, r0_tw);
    r1 = vld1q(r_ptr + (8* 2));
    r1_tw = vld1q(r_ptr + (8* 3));
    ct_butterfly(&data.val[0], &data.val[1],
                 r1, r1_tw);
    r2 = vld1q(r_ptr + (8* 4));
    r2_tw = vld1q(r_ptr + (8* 5));
    ct_butterfly(&data.val[2], &data.val[3],
                 r2, r2_tw);
    vst4q(in, data);
    in += (8* 4); r_ptr += (96 / 2);
} }
```

```
.LBB0_1:
 vldrh.u16    q2, [r1], #96  //-
 vldrh.u16    q1, [r0, #32]  //-
 vmul.i16     q0, q2, q1
 vldrh.u16    q3, [r1, #-80]
 vqrdmulh.s16 q1, q1, q3
 vldrh.u16    q6, [r0]
 vmla.u16     q0, q1, r2
 vldrh.u16    q1, [r0, #48]
 vmul.i16     q4, q2, q1
 vldrh.u16    q2, [r1, #-64]
 vqrdmulh.s16 q1, q1, q3
 vldrh.u16    q3, [r0, #16]
 vmla.u16     q4, q1, r2
 vadd.i16     q1, q4, q3    //l
 vmul.i16     q5, q1, q2
 vldrh.u16    q2, [r1, #-48]
 vqrdmulh.s16 q1, q1, q2
 vsub.i16     q3, q3, q4
 vmla.u16     q5, q1, r2
 vadd.i16     q1, q0, q6
 vldrh.u16    q4, [r1, #-32]
 vsub.i16     q2, q1, q5
 vadd.i16     q1, q5, q1    //-
 vmul.i16     q5, q4, q3
 vldrh.u16    q4, [r1, #-16]
 vqrdmulh.s16 q3, q3, q4
 vsub.i16     q0, q6, q0
 vmla.u16     q5, q3, r2
 vsub.i16     q4, q0, q5    //l
 vadd.i16     q3, q5, q0    //-
 vst40.16     {q1-q4}, [r0] //-
 vst41.16     {q1-q4}, [r0] //-
 vst42.16     {q1-q4}, [r0] //-
 vst43.16     {q1-q4}, [r0]!//-
 le           lr, .LBB0_1
```

Listing 12: Left: Intrinsics implementation of last two layers of Kyber NTT. Right: Result of compilation with Arm Compiler 6.18.

```
layer45_loop:
vadd.u16 q3, q2, q0     //................*..............
vmul.s16 q1, q3, r2     //.................*.............
vldrw.u32 q4, [in, #32] //.....*.........................
vmul.s16 q7, q4, r4     //.......*.......................
ldrd r2, r1, [r, #-8]   //..*............................
vqrdmulh.s16 q6, q4, r6 //........*......................
vsub.u16 q0, q2, q0     //...............*...............
vmla.s16 q7, q6, mod    //.........*.....................
vldrw.u32 q5, [in]      //...*...........................
vqrdmulh.s16 q4, q3, r8 //..................*............
vadd.u16 q2, q5, q7     //...........*...................
vmla.s16 q1, q4, mod    //...................*...........
vsub.u16 q7, q5, q7     //..........*....................
vmul.s16 q5, q0, r2     //......................*........
vadd.u16 q3, q2, q1     //.....................*.........
vqrdmulh.s16 q0, q0, r1 //.......................*.......
vsub.u16 q4, q2, q1     //....................*..........
vmla.s16 q5, q0, mod    //........................*......
vldrw.u32 q0, [in,#112] //......e........................
vsub.u16 q6, q7, q5     //.........................*.....
vldrw.u32 q2, [in, #80] //....e..........................
vadd.u16 q5, q7, q5     //..........................*....
ldrd r4, r6, [r] , #24  //e..............................
ldrd r2, r8, [r, #-16]  //.e.............................
vst40.u32 {q3-q6}, [in] //...........................*...
vqrdmulh.s16 q1, q0, r6 //.............e.................
vst41.u32 {q3-q6}, [in] //............................*..
vmul.s16 q0, q0, r4     //............e..................
vst42.u32 {q3-q6}, [in] //.............................*.
vmla.s16 q0, q1, mod    //..............e................
vst43.u32 {q3-q6},[in]! //..............................*
le lr, layer45_loop
...
```

```
layer67_loop:
vmla.s16 q7, q1, mod    //........*.........................
vldrw.u32 q5, [in, #16] //.*................................
vsub.u16 q3, q5, q0     //..............*...................
vldrh.u16 q2, [r, #-32] //.......................*..........
vadd.u16 q1, q5, q0     //...............*..................
vmul.s16 q4, q3, q2     //.........................*........
vldrh.u16 q5, [r, #-16] //........................*.........
vqrdmulh.s16 q2, q3, q5 //..........................*.......
vldrh.u16 q5, [r, #-48] //.................*................
vmla.s16 q4, q2, mod    //...........................*......
vldrw.u32 q0, [in]      //*.................................
vsub.u16 q3, q0, q7     //.........*........................
vqrdmulh.s16 q6, q1, q5 //...................*..............
vsub.u16 q5, q3, q4     //............................*.....
vldrh.u16 q2, [r, #-64] //................*.................
vadd.u16 q4, q3, q4     //.............................*....
vmul.s16 q1, q1, q2     //..................*...............
vadd.u16 q3, q0, q7     //..........*.......................
vmla.s16 q1, q6, mod    //....................*.............
vldrh.u16 q7, [r] , #96 //....e.............................
vadd.u16 q2, q3, q1     //......................*...........
vldrw.u32 q6, [in,#112] //...e..............................
vsub.u16 q3, q3, q1     //.....................*............
vldrh.u16 q1, [r, #-80] //.....e............................
vmul.s16 q0, q6, q7     //...........e......................
vst40.u32 {q2-q5}, [in] //..............................*...
vqrdmulh.s16 q6, q6, q1 //............e.....................
vst41.u32 {q2-q5}, [in] //...............................*..
vmla.s16 q0, q6, mod    //.............e....................
vldrw.u32 q6, [in, #96] //..e...............................
vmul.s16 q7, q6, q7     //......e...........................
vst42.u32 {q2-q5}, [in] //................................*.
vqrdmulh.s16 q1, q6, q1 //.......e..........................
vst43.u32 {q2-q5},[in]! //.................................*
le lr, layer67_loop
```

Listing 13: Optimal (for Cortex-M55) scheduling for the periodic parts of the last four laysers of the NTT for CRYSTALS-Kyber. Note the significant amount of interleaving.

		Type	Cycles	Code size	Readable	Flexible µarch
32-bit Dilithium NTT		Cortex-M4				
	[AHKS22]	Handwritten ASM	8093	1.5 KB	✓	✗
		Cortex-M55				
	[BBMK+21]	Scripted ASM	2027	7.8 KB	✗	✗
	Our work 2+2+2+2 layers	HeLight$_{55}$	2088	1.1 KB	✓	✓
	Our work 3+3+2 layers	HeLight$_{55}$	2037	1.1 KB	✓	✓
16-bit Kyber NTT		Cortex-M4				
	[AHKS22]	Handwritten ASM	5992	2.2 KB	✓	✗
	[HZZ+22]	Handwritten ASM	4474‡	?‡	?‡	✗
		Cortex-M55				
	Our work 1+2+2+2 layers	HeLight$_{55}$	1018	0.9 KB	✓	✓
	Our work 2+3+2 layers	HeLight$_{55}$	926	1.0 KB	✓	✓
CFFT Q.31		Cortex-M55				
	[Arme]	Intrinsics	5000	1.1 KB	✓	✓
	[Armb]	Handwritten ASM	4560	1.3 KB	✓	✗
	Our work	HeLight$_{55}$	4300	1.4 KB	✓	✓

‡: The code for [HZZ+22] is not publicly available yet.

Table 1: Comparison of various FFT and NTT implementations.

References

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Daan Sprenkels. Faster kyber and dilithium on the cortex-m4. In *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*, page 853–871, Berlin, Heidelberg, 2022. Springer-Verlag. 2, 18, 19, 21

[Arma] Arm Limited. Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/latest. 4

[Armb] Arm Limited. ARM-Software Repository. https://github.com/ARM-Softwaregithub.com/ARM-Software/EndpointAI. 14, 21

[Armc] Arm Limited. Armv8-M Architecture Reference Manual. https://developer.arm.com/documentation/ddi0553/latest. 1, 4

[Armd] Arm Limited. Blog series: “making Helium: Why not just add Neon?”. Part 1: https://community.arm.com/developer/research/b/articles/posts/making-helium-why-not-just-add-neon, Part 2: https://community.arm.com/developer/research/b/articles/posts/making-helium-sudoku-registers-and-rabbits, Part 3: https://community.arm.com/developer/research/b/articles/posts/making-helium-going-around-in-circles, Part 4: https://community.arm.com/developer/research/b/articles/posts/making-helium-going-around-in-circles. 4

[Arme] Arm Limited. CMSIS-DSP Repository. https://github.com/ARM-software/CMSIS-DSP. 13, 21

[Armf] Arm Limited. Whitepaper: Introducing Cortex-M55. https://www.arm.com/resources/white-paper/cortex-m55-introduction. 4

[Armg] Arm Limited. Whitepaper: Introduction to the Armv8.1-M Architecture. https://www.arm.com/resources/white-paper/intro-armv8-1-m-architecture. 4

[BBMK+21] Hanno Becker, Jose Maria Bermudo Mera, Angshuman Karmakar, Joseph Yiu, and Ingrid Verbauwhede. Polynomial multiplication on embedded vector architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):482–505, Nov. 2021. 4, 15, 18, 19, 21

[BHK+21] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, Nov. 2021. 15

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Lorenz Panny, and Bo-Yin Yang. Efficient multiplication of somewhat small integers using number-theoretic transforms. In *Advances in Information and Computer Security: 17th International Workshop on Security, IWSEC 2022, Tokyo, Japan, August 31 – September 2, 2022, Proceedings*. Springer-Verlag, 2022. 15

[Emb15] Embecosm Limited, supported by Innovate UK. Superoptimization – Feasibility Study, 2015. https://www.embecosm.com/appnotes/ean15/ean15.html. 4

[GW96] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0–1 integer programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996. 2, 4

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636, Aug. 2022. https://tches.iacr.org/index.php/TCHES/article/view/9833. 18, 19, 21

[KL99] Daniel Kästner and Marc Langenbach. Code optimization by integer linear programming. pages 122–136, 03 1999. 2, 4

[KW98] Daniel Kästner and Reinhard Wilhelm. Operations research methods in compiler backends. *Mathematical Communications (mc@mathos.hr); Vol.3 No.2*, 01 1998. 2

[Lam88a] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 318–328, New York, NY, USA, 1988. Association for Computing Machinery. 2, 3

[Lam88b] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. *SIGPLAN Not.*, 23(7):318–328, jun 1988. 2

[LDK+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, 2022. https://pq-crystals.org/dilithium. 1, 15

[Mas87] Henry Massalin. 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, page 122–126. Association for Computing Machinery, 1987. 2, 4

[NIS16] NIST Computer Security Division. Post-Quantum Cryptography Standardization, 2016. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography. 1, 15

[PF] Laurent Perron and Vincent Furnon. Google OR-Tools. https://developers.google.com/optimization/. 2, 4

[Rau94] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, page 63–74, New York, NY, USA, 1994. Association for Computing Machinery. 2, 3

[RG81] B. Rau and Christopher Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. volume 12, 12 1981. 3

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, 2022. https://pq-crystals.org/kyber. 1, 15

[SCC+17] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *CoRR*, abs/1711.04422, 2017. 4

[SSA13] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 305–316, New York, NY, USA, 2013. Association for Computing Machinery. 2, 4

[WGB94] T.C. Wilson, G.W. Grewal, and D.K. Banerji. An ilp solution for simultaneous scheduling, allocation, and binding in multiple block synthesis. In *Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 581–586, 1994. 2, 4