Milestone Report: Optimized Cryptography with LLVM Non-Standard Bitwidth Types

Michael McLoughlin
mcloughlin@cmu.edu

https://mmcloughlin.com/15745

1 Major Changes

There have not been any major changes to the project. The goal remains to optimize cryptographic finite-field implementations in non-standard bitwidth types. Initial work has suggested this is a fruitful area for optimization. The only changes the project are minor refinements of the scope and expected optimizations to be implemented.

2 Accomplishments

2.1 Experiment Setup

I have selected and implemented the primary optimization target for the project, and established baseline performance in stock LLVM. The primary optimization target is the X25519 cryptographic function [1] implemented as plainly as possible in C with the `_BitInt(255)` type provided by the upcoming C23 standard [2]. The X25519 function is a scalar multiplication on the Curve25519 elliptic curve, which ultimately reduces to arithmetic in the finite field of integers modulo the prime $p = 2^{255} - 19$. X25519 was selected due to its importance as a real-world cryptographic algorithm, and because the implementation strategies required to optimize its finite field operations could be transferable to other cryptographic primitives. Finally, the C implementation with 255-bit types was as simple as reasonably possible to prompt exploration of how much optimization can be automated by the compiler.

Alongside the target X25519 implementation, I have implemented a test suite and benchmark against a popular hand-tuned external implementation. The test suite verifies both the low-level finite field operations, and the X25519 function against test vectors. The test suite is valuable to provide confidence in the initial implementation, and to verify any custom optimizations preserve correctness. The benchmark compares the target implementation against the

hand-tuned version in `libsodium`, a widely deployed and well-optimized cryptographic provider, which we expect to have good though perhaps not absolute peak performance. The benchmark shows that out of the box Clang 17.0.3 produces code that takes over 125x that of `libsodium` for a single X25519 call.

Implementation	Time (ns)	Iterations	Cycles	Instructions
Target	2900123	240	15.4M	81.236M
`libsodium`	23161	30217	124.7k	359.506k

This result suggests there may be some low hanging fruit, but we also have a lot of work to do!

2.2 Understanding Existing LLVM Optimization Passes

Having established target benchmarks of interest, I spent some time understanding how LLVM compiles multi-precision bitwidth arithmetic at the moment. In the mid-end optimizations on LLVM OR, the `SROAPass`, `EarlyCSEPass` and `IPSCCPPass` perform useful cleanup. The `InstCombinePass` and `CorrelatedValuePropagationPass` do perform some optimizations of interest, for example replacing modular reductions with conditional subtraction when the value is known to be only slightly larger than the modulus (such as the result of an addition).

The `llc` backend performs more substantial transformations. Firstly, the `ExpandLargeDivRemLegacyPass` replaces large modular reductions with a looping construct which is poorly suited to our use case. It was interesting to see in source code comments for that pass that "future work includes generating more specialized code". Following this, the `x86-isel` pass does lowering and instruction selection. Most importantly, it's the "Legalize SelectionDAG Types" phase which converts multi-precision arithmetic on wide types into word-by-word operations on machine words. Moreover, the instruction selection phases for x86 do not appear to take advantage of Intel ADX extensions, which are frequently exploited to accelerate multi-precision arithmetic in cryptographic finite fields.

To support these investigations, I developed a helper tool to inspect the transitions of LLVM IR and Machine IR through the mid- and back-end optimization stages. This parses the output of `opt` and `llc` with their `-print-after-all` flags and produces a HTML report with the IR diff for all passes that made a change.

2.3 Crandall Reduction Pass

I have implemented an optimization pass that targets modular reductions by constants with the *Crandall* form. These reductions appear frequently in cryptographic finite field implementations such as X25519. *Crandall primes* are those of the form $p = 2^n - c$ for a small constant c, but the optimization would actually apply for any modulus of that form whether prime or not. The optimization leverages the simple fact that for a modulus $m = 2^n - c$ we have

$2^n \equiv c \pmod{m}$. Therefore, given a value x to be reduced, if we represent it as its low n bits l and remaining high bits h, then:

$$x = 2^n h + l \equiv ch + l \pmod{m}, \tag{1}$$

and $ch + l$ will in most circumstances be much smaller than x. Therefore, this does not complete the reduction by m but it cheaply produces a result equivalent modulo m.

The `CrandallReductionPass` detects and applies this optimization technique to LLVM IR. Specifically, we apply it to `urem` instructions where the modulus is a constant with Crandall form. At this stage the pass only performs the single step Crandall reduction as above, leaving the rest of the reduction to LLVM, but the results are surprisingly good nonetheless. With the Crandall reduction pass applied, our benchmark shows a massive 23x speedup.

Implementation	Time (ns)	Iterations	Cycles	Instructions
Target with Crandall	121530	5760	654.7k	2.80878M

These results are encouraging, but there's much more performance still left on the table.

2.4 Reduction Analysis Pass

One common technique that appears in hand-tuned optimizations of finite field arithmetic is to defer complete modular reduction until the last possible moment. Instead, intermediate computations operate on convenient partially reduced forms. For example in the case of Crandall reduction for X25519, we could use the relation $2^{255} \equiv 19 \pmod{p}$, but it's actually more favorable to use the fact that $2^{256} \equiv 38 \pmod{p}$ since 256 sits nicely on a machine-word boundary. Therefore hand-tuned implementations often maintain 256-bit intermediate values and only do a full reduction modulo p at the end. A related concept is that a Crandall reduction step will leave a value in a nearly reduced form, and it may be beneficial to us to keep it that way as long as it won't affect the correctness of future modular arithmetic on that value. Our simple C version of X25519 doesn't leverage these powerful techniques, since it reduces modulo p after every operation. We would like to be able to automatically detect and apply partial reduction in the compiler, but we need to know when it would be safe to do so.

The `ReductionAnalysisPass` performs an analysis pass that aims to unlock these kinds of optimizations. Specifically it answers the question of when all uses of a value *will be reduced* modulo p in future. This property is intended to be used to detect when reduction modulo p operations are not necessary, and the value could be left in a partially-reduced form. The `ReductionAnalysisPass` is a worklist-based analysis in which the reduction property is generated by `urem` instructions and preserved by selected arithmetic operations. I intend to use the results of this analysis to extend the Crandall reduction pass, and to implement so-called "Slothful Reduction" [3].

3 Milestone

The milestone goal was to have to have "implemented and evaluated at least one optimization applying to cryptographic finite fields in LLVM IR". I have implemented the Crandall Reduction Pass and evaluated it against the X25519 optimization target, therefore the milestone has been met.

4 Surprises

There have not been any major surprises, though some minor unforeseen factors in the project are below.

- *LLVM Baseline Performance.* I expected better performance from LLVM out of the box. This isn't a problem per se, and in fact suggests there might be low-hanging fruit. However, it also suggests there might be a lot of work required to match hand-tuned implementations, which has unfortunately lowered my expectations of doing so in the time-scale of this project.
- *LLVM Backend Complexity.* Coming into the project I had less familiarity with the LLVM backend optimizer than I did with the mid-end passes on LLVM IR. While this is not a blocker, I have less ambitious expectations for this aspect of the project than I did before.

5 Schedule

The remaining time on the project will focus on three areas, with approximately equal time devoted to each.

1. *Mid-end Optimization.* Leverage the results of the Reduction Analysis Pass to perform optimizations on modular reduction at the LLVM IR level. Specific optimizations would be: replacing modular reduction by conditional subtraction, performing crandall reduction at the word boundary, and deferring reductions using the "Slothful Reduction" technique.
2. *Back-end Optimization.* Deep dive into the low-level instruction scheduling for multi-precision arithmetic. Seek optimization opportunities for multi-precision arithmetic with Intel ADX instructions.
3. *Wrap-up.* Evaluation, report writing and presentation preparation.

6 Resources Needed

Since the project has focussed on LLVM and x86, the resource requirements have been minimal. I don't expect to need anything else to complete the project.

References

[1] Adam Langley, Mike Hamburg, and Sean Turner. *Elliptic Curves for Security*. RFC 7748. Jan. 2016. DOI: 10.17487/RFC7748. URL: https://www.rfc-editor.org/info/rfc7748.

[2] *Adding a Fundamental Type for N-bit integers*. Proposal. June 2021. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2763.pdf.

[3] Michael Scott. *Slothful reduction*. Cryptology ePrint Archive, Paper 2017/437. 2017. URL: https://eprint.iacr.org/2017/437.