Project Proposal: Optimized Cryptography with LLVM Non-Standard Bitwidth Types

Michael McLoughlin
mcloughlin@cmu.edu

https://mmcloughlin.com/15745

1 Description

1.1 Motivation

An intriguing feature of LLVM IR is its arbitrary bit-width integer types. One might think this is just a curiosity, but in fact *Formal-Methods-Based Bugfinding for LLVM's AArch64 Backend*, they observe:

> Does anyone actually care if the LLVM backends can deal with non-power-of-2 bitwidths? Turns out yes: in an optimized compile of LLVM itself, using LLVM, every integer width from 1 through 64 can be found. The largest integer that occurs when compiling LLVM using LLVM is 320 bits wide.

This invites the question of how we might leverage non-standard bitwidth types for specialist applications? Does LLVM do a good job on these use cases, and what optimization opportunities are there?

Our main target application will be high-performance cryptography, where elliptic curve implementations rely on efficient multi-precision arithmetic modulo fixed prime numbers. For example, one widely deployed elliptic curve is Curve25519, which reduces to operations on 255-bit values modulo the large prime number $2^{255} - 19$. Implementations of such curves are famously complex, but they could be quite easily expressed in LLVM IR using arithmetic operations on the `i255` type. How efficient would the generated code be? Can we implement techniques used in hand-tuned implementations and bring them to the LLVM framework? How close could we get it to the best known performance for these elliptic curves?

1.2 Proposal

The goal of the project is to implement LLVM optimizations targeted at non-standard bitwidth types. Cryptographic finite-field implementations will be the

primary focus, as we have reason to believe these might be ripe for applying domain-specific techniques.

1.2.1 Potential Optimizations

Finite field implementations perform operations on large integers modulo a fixed prime p. These primes are usually chosen to have specific structure. *Crandall primes* have the form $2^n - c$ for a small constant c. Curve25519 mentioned above has a modulus of this form, as does the NIST P521 elliptic curve with modulus $2^{521} - 1$. Alternatively *Generalized Mersenne primes* take the form $f(2^n)$ for some low-weight polynomial $f(x)$ where ideally n is a nice multiple of a machine word width. The NIST P256 curve for example uses the modulus $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. These types of structured primes are amenable to specific optimization techniques, for example Fast Modular Reduction by Solinas, and Word-by-Word Montgomery Reduction. In this project we would hope to leverage techniques such as these to optimize a cryptographic finite-field implementation in LLVM's IR.

A second approach to consider is whether we can optimize one level up in the computation. That is, what optimizations could apply over multiple finite-field operations? This is especially interesting to explore in this context since we expect to have a concise representation of the computation using large integer types, prior to lowering and optimization of multi-precision arithmetic at the machine-word level. A specific optimization we could pursue is so-called "lazy" or "slothful" reductions [13, 10] where we recognize that we can defer reduction until the point at which a reduced value is needed, and therefore perform multiple arithmetic operations in an unreduced form.

Finally, most peak performance finite-field implementations rely on low-level machine-specific instruction scheduling optimizations. For example, they must effectively maintain two carry chains using Intel's ADX Extension. Recent work [9, 4] has shown massive performance improvements with automated program search. It seems likely that LLVM may not achieve these peak results out-of-the-box, and it may be fruitful to explore improvements to instruction scheduling in these refined use cases.

Implementing modulus specific reduction algorithms, lazy reductions and instruction scheduling optimizations are good starting points to consider. As we explore the project, other directions may arise.

1.2.2 Metrics

Early in the project we would establish benchmarks for evaluation. For example, this might contain an implementation of Curve25519's finite-field in LLVM IR, and potentially a suite of similar fields with different parameterizations and forms. Performance of the compiled code in out-of-the-box LLVM would be our baseline, and we'd hope to improve it through our domain-specific optimizations. Performance of best-known implementations in production-grade libraries such

as OpenSSL would serve as a north star. Performance on multiple architectures is of interest, but our focus will be x86-64.

1.3 Goals

Our *100% Goal* is to implement and evaluate multiple optimizations that apply to cryptographic finite fields in LLVM IR.

- Implement multiple optimizations.
- One optimization at each level: high-level optimization such as lazy reduction, mid-level optimization such as modular reduction for a specific family of primes, and low-level instruction scheduling.
- Evaluate performance on a suite of cryptographic finite fields relative to LLVM baseline and external optimized libraries.

Our *75% goal* would be to implement and evaluate one optimization method. There are many possible *stretch goals* for the project:

Multiple Modular Reduction Strategies Implement optimizations targetting multiple families of prime moduli.

Multi-Architecture Explore optimizations on architectures beyond x86-64, most likely AArch64.

Non-Cryptographic Optimizations Explore how LLVM's non-standard bitwidth types are used outside of cryptography.

MLIR Are LLVM's wide types enough to capture the semantics we need for this style of optimization? If not, could we use an MLIR dialect to make compilation easier.

2 Logistics

2.1 Schedule

Tentative schedule:

1. *Experiment setup.* Implement one or more cryptographic implementations in LLVM IR that will serve as our optimization targets. Measure baseline performance in stock LLVM. Gather highly optimized implementations of the same cryptographic primitives for comparison.
2. *Modular reduction optimization.* Implement detection of multi-precision modular reduction in LLVM. Pick one of the modular reduction optimization strategies and implement it. Evaluate results relative to baseline.

3. *Instruction scheduling.* Deep dive into the low-level instruction scheduling for multi-precision arithmetic. Seek optimization opportunities for multi-precision arithmetic with Intel ADX instructions.

4. *High-level optimization.* Pursue high-level optimizations prior to lowering into standard bit width types. Likely optimizations here are "lazy" or "slothful" reduction which attempts to defer and eliminate reductions when it's safe to operate with unreduced values temporarily.

5. *Stretch goals.* Explore stretch goals as time allows. For example, expanding optimization strategies to apply to broader families of cryptographic code, optimizations for non-x86 architectures, or even exploring non-cryptographic uses of non-standard bit width types.

6. *Wrap-up.* Final polish. Report writing and presentation preparation.

2.2 Milestone

At the four week mark, we would like to have met the 75% goal outlined in Section 1.3. That is, to have implemented and evaluated at least one optimization applying to cryptographic finite fields in LLVM IR.

2.3 Resources Needed

The intent is to implement these optimizations within the LLVM framework. We will rely on external libraries such as OpenSSL for evaluation purposes.

Benchmarking will be done on an x86-64 Linux workstation. My personal workstation provided by CMU SCS should be sufficient.

Depending on the direction the project takes, it may be interesting to make use of a small cluster of machines for automated program search, or to have access to an AArch64 server for benchmarking. I expect this will be feasible with CMU resources.

2.4 Getting Started

I have not done any work on this project specifically, though I have run a brief experiment using LLVM wide integer types before, and I've also written an experimental elliptic curve cryptography compiler `ec3`. Therefore I think I am well-placed to start work immediately.

2.5 Literature

[1] Ryan Berger et al. *Formal-Methods-Based Bugfinding for LLVM's AArch64 Backend.* URL: `https://blog.regehr.org/archives/2265`.

[2] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *Public Key Cryptography - PKC 2006*. Ed. by Moti Yung et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228. ISBN: 978-3-540-33852-9. URL: https://www.iacr.org/cryptodb/archive/2006/PKC/3351/3351.pdf.

[3] Joppe W. Bos et al. *Montgomery Multiplication Using Vector Instructions*. Cryptology ePrint Archive, Paper 2013/519. 2013. URL: https://eprint.iacr.org/2013/519.

[4] Jay Bosamiya et al. "Verified Transformations and Hoare Logic: Beautiful Proofs for Ugly Assembly Language". In: *Software Verification*. Ed. by Maria Christakis et al. Cham: Springer International Publishing, 2020, pp. 106–123. ISBN: 978-3-030-63618-0.

[5] Niek J. Bouman. *Multiprecision Arithmetic for Cryptology in C++ - Compile-Time Computations and Beating the Performance of Hand-Optimized Assembly at Run-Time*. 2018. arXiv: 1804.07236 [cs.CR]. URL: https://arxiv.org/abs/1804.07236.

[6] Michael Brown et al. "Software Implementation of the NIST Elliptic Curves Over Prime Fields". In: *Topics in Cryptology — CT-RSA 2001*. Ed. by David Naccache. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 250–265. ISBN: 978-3-540-45353-6. URL: https://delta.cs.cinvestav.mx/~francisco/arith/julio.pdf.

[7] Shay Gueron and Vlad Krasnov. *Fast Prime Field Elliptic Curve Cryptography with 256 Bit Primes*. Cryptology ePrint Archive, Paper 2013/816. 2013. URL: https://eprint.iacr.org/2013/816.

[8] Shay Gueron and Vlad Krasnov. "Software Implementation of Modular Exponentiation, Using Advanced Vector Instructions Architectures". In: *Arithmetic of Finite Fields*. Ed. by Ferruh Özbudak and Francisco Rodríguez-Henríquez. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 119–135. ISBN: 978-3-642-31662-3. URL: https://link.springer.com/chapter/10.1007/978-3-642-31662-3_9.

[9] Joel Kuepper et al. "CryptOpt: Verified Compilation with Randomized Program Search for Cryptographic Primitives". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591272. URL: https://doi.org/10.1145/3591272.

[10] Shuguo Li and Zhen Gu. "Lazy Reduction and Multi-Precision Division Based on Modular Reductions". In: *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 2018, pp. 407–410. DOI: 10.1109/APCCAS.2018.8605566.

[11] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. "Efficient Implementation". In: *Handbook of Applied Cryptography*. CRC Press, 1996. Chap. 14. URL: http://cacr.uwaterloo.ca/hac/about/chap14.pdf.

[12] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. "New software speed records for cryptographic pairings". In: *Progress in Cryptology – LATINCRYPT 2010*. Ed. by Michel Abdalla and Paulo S.L.M. Barreto. Vol. 6212. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2010, pp. 109–123. URL: http://cryptojedi.org/papers/#dclxvi.

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