Motivation

tl;dr: hardware acceleration for cryptography
Cryptographic applications

- "FHE": Fully Homomorphic Encryption (compute on encrypted data)
  - HEIR
  - HECO
  - HECATE
  - Concrete
  - HEaaN. MLIR

- NIST-standard Post-Quantum Cryptography (PQC)
  - Kyber (key encapsulation)
  - Dilithium (digital signature)

Bottleneck: modular polynomial arithmetic

\[ \mathbb{Z}_q[x]/(F(x)) \]
Polynomial modular arithmetic

Example: $q = 8$, $F(x) = x^3 + 1$

$$(1 + 5x + x^2)(3 + x - x^2)$$

Normal product:
$$3 + 16x + 7x^2 - 4x^3 - x^4$$

Mod $x^3 + 1$:
$$7 + 17x + 7x^2$$

"set" $x^3 + 1 = 0$ and reduce

Mod 8 coefficients :
$$7 + x + 7x^2$$

Choice of polynomial & coefficient mod is security + performance critical

FHE Example: $q \approx 952809000096560291$, $F(x) = x^{65536} + 1$
Why put crypto in the compiler?

• Need hardware acceleration for FHE and PQC!
  • GPU/TPU
  • FPGA
  • Optical accelerators

• HW supports poly ops as first-class operations
  • DARPA DPRIVE
Input + computation dialect
No new MLIR infrastructure
Lowers to existing dialects

Prototype implementation at
GitHub: [github.com/google/heir](https://github.com/google/heir)
Types and Attributes

tl;dr: a new polynomial type
Polynomial type

A polynomial is an element of a ring

```plaintext
!polynomial.polynomial<#ring>
```

```plaintext
#ring = #polynomial.ring<
    ctype=i32,  Coefficient (base) type
    cmod=4294967291, Coefficient modulus
    ideal=#p  Polynomial modulus
>
#p = #polynomial.polynomial<1 + x**1024>
```

Modulus must be statically known
Custom attribute: parser, storage

"ring" is the math name for this
Why a new type?

• are polynomials just “tensors with metadata“?
• A polynomial can be stored in many ways
  • Many nonzero coefficients -> dense tensor
  • Large degree and many zero coefficients -> sparse tensor
  • Not necessarily always a list of coefficients (DFT/NTT “evaluation form”)
• Polynomial seems like the right abstraction for passes
Why specify ring on the type?

• "Better for ops to specify semantics"
• Once you agree we need a new type...
• Type conversion needs to pick `tensor<dim x ty>`
• Alternatives to make type conversion work seem overkill
Why not hard-code choices?

• Why not specialize to \((x^N + 1)\)
  • Future proof for new innovations in PQC/FHE crypto
  • Support non-PQC crypto (e.g. secret sharing)
  • Support non-crypto scientific computation
  • Potential use within MLIR in polyhedral analysis

• Post-Quantum Crypto, Private Information Retrieval, Secure Multi-Party Computation
  • Similar crypto-friendly polynomial math
  • Often much smaller security parameters than FHE -> different tradeoffs
Ops & Passes

tl;dr: optimized lowerings for common rings
Obvious polynomial ops

- constant $<1 + x^{1024}>$
- add, sub, mul, div_rem*
- to_tensor, from_tensor
- mul_scalar
- leading_term (degree + leading coefficient)
Less obvious ops

- **monomial** construct a single-term polynomial from data
- **monomial_mul** multiply a polynomial by a monomial (optimized lowerings)
- **dft/idft** compute a forward/reverse complex Fourier transform
- **ntt/intt** compute a forward/reverse integer number theoretic transform (integer-only DFT analogue)

Enables $O(n \log(n))$ multiplication of $f \cdot g$ as $iNTT(NTT(f) \cdot NTT(g))$ with $\cdot$ elementwise!

While this requires a compatible ring, virtually all crypto uses such rings
Ops we probably don't need

- `tensor_mul` use `linalg.generic` with poly ops inside?
Lowering mul

- Generic lowering supporting all* parameters: Naive polymul + modular reduction
  - Naive polymul computed via...
    - Cyclic convolution (linalg.generic)
    - DFT/NTT + pointwise mul + IDFT/INTT
    - Karatsuba, etc.
  - Modular reduction via textbook poly long division (scf.while)
- \( \mathbb{Z}_q[x]/(x^N + 1) \) - machine word-sized coefficients
  - no manual mod reduction step
  - (Nega)cyclic convolution
  - DFT + entry-wise mul + IDFT
  - Tensor mul via Toeplitz matrix trick (TPU)
- \( \mathbb{Z}_p[x]/(x^N + 1) \) - prime coefficients (< 64-bits)
  - NTT + entry-wise mul + INTT
Lowering dft/ntt

- Goal: keep polynomial in coefficient or evaluation form for as long as possible
  - Canonicalize \([\text{dft}, \text{op1}, \text{idft}, \text{dft}, \text{op2}, \text{idft}]\) to \([\text{dft}, \text{op1}, \text{op2}, \text{idft}]\)
- Lowering via
  - Cooley-Tukey
  - Stockham + AVX
  - Dedicated accelerator support
Formal verification!

- Cambridge group (Tobias Grosser) looking into Lean to formalize MLIR dialects & passes
- Polynomial dialect is one of their case studies
- We hope this will allow us to formally verify correctness of polynomial passes and lowerings
Roadmap

tl;dr: lower to LLVM, then accelerators
Milestones

- Focus on HW acceleration
- Nontrivial example implementations as end-to-end tests
- Polynomial interpolation
- Simplified Dilithium scheme
- Lowering polynomial to LLVM via standard dialects
  - Generic Ring $\mathbb{Z}_p[x]/(x^N + 1)$

Focus on HW acceleration
GitHub repo
github.com/google/heir

HEIR meetings
google.github.io/heir/community/

RFC on Discourse
shorturl.at/fNO18
Questions