Prototyping a compiler for homomorphic encryption in MLIR

Juneyoung Lee (aqjune@cryptolab.co.kr)
Woosung Song (lego0901@gmail.com)
Homomorphic Encryption?

• An encryption method that allows operations on ciphertexts (encrypted texts)

• “Homomorphic”: \( \text{Encrypt}(x \ast y) = \text{Encrypt}(x) \ast \text{Encrypt}(y) \)
  
  +, *, ReLU, ... on encrypted data: “Private AI”

• Among 5 Impactful Emerging Technologies in 2022 from Gartner

Why is HE Important?

It completes protection of the data

The three states of data

1. At Rest: Secure storage
2. In Transit: HTTPS
3. In Use: **HE**

- HE Libraries: CryptoLab’s HEaaN, Microsoft’s SEAL, Duality Technologies’ Palisade, ...
- Layers using HE Libraries: IBM’s HELayer, AWS’s HIT, ...
Challenge: Space & Speed
Homomorphic Encryption 101 – CKKS scheme

• IntPoly: A polynomial \(a_0 + a_1x + \cdots + a_{N-1}x^{N-1}\) with large integer coefficients
  
  • \(0 \leq a_i < Q\) where \(Q\) is a large integer
  • To avoid using BigInt, \(Q = q_0 \times q_1 \times \cdots \times q_{L-1}\) where \(q_j\) prime (uint64_t)
  • Coefficient \(a_i : \text{uint64}_t[L]\) which is \((a_i \% q_j)\) - called Residual Number System
  • All polynomial operations are modulo \((x^N + 1)\) (“polynomial ring”)
Challenge: Space & Speed
Homomorphic Encryption 101 – CKKS scheme

• IntPoly: A polynomial $a_0 + a_1x + \cdots + a_{N-1}x^{N-1}$ with large integer coefficients
• In C++, IntPoly is `uint64_t[N][L]`, or equiv. `uint64_t[L][N]`, where

1. $N$: the degree of the polynomial ($\sim 2^{17}$)
2. $L$: # of prime numbers used to represent coefficients ($\sim 30$)
Challenge: Space & Speed
Homomorphic Encryption 101 – CKKS scheme (En/Decryption)

- Given a secret key s: IntPoly
- Enc(p, s): Ciphertext = (a, -a*s + p + e) where
  a: fresh random poly  e: fresh error poly (random poly with small coeffs)
- Dec(c, s): Plaintext = c.second + c.first*s ~= p + e
Given a secret key $s$: IntPoly

- $\text{Enc}(p, s): \text{Ciphertext} = (a, -a*s + p + e)$ where
  - $a$: fresh random poly
  - $e$: fresh error poly (random poly with small coeffs)
- $\text{Dec}(c, s): \text{Plaintext} = c.\text{second} + c.\text{first}*s \approx p + e$
Challenge: Space & Speed
Homomorphic Encryption 101 – CKKS scheme (En/Decryption)

- Given a secret key $s$: IntPoly
- Enc$(p, s)$: Ciphertext $= (a, -a*s + p + e)$ where $a$: fresh random poly, $e$: fresh error poly (random poly with small coefficients)
- Dec$(c, s)$: Plaintext $= c.second + c.first*s$

2. Multiplying two polynomials is slow! 😞
   - Naive product requires $O(N^2)$
   - Sol: Number-theoretic transformation!
     - Analogous to FFT
     - Time complexity: $O(N \log N)$
Challenge: Space & Speed
Homomorphic Encryption 101 – CKKS scheme (Other Ops)

1. **Add(ctxt1, ctxt2):** ctxt1 + ctxt2
2. **Mult(ctxt1, ctxt2):** ctxt1 * ctxt2
   - Needs to remove $s^2$ term: ‘evaluation key’ must be pre-calculated & used
   - A scale factor multiplied by encode() is multiplicatively increased: ‘rescale’ op
3. **Rotate(ctxt, i):** [ctxt[i], ctxt[i+1], ..., ctxt[0], ..., ctxt[i-1]]
   - Needs ‘rotation keys’ that are pre-calculated
4. **Bootstrap(ctxt):** very slow
   - ctxt cannot be used after ~L multiplications; bootstrap revives it
Challenge: Space & Speed

In a Nut Shell

1. Space
   - Message size is multiplied by # of primes
   - If a message is not packed (m.size() < N), the factor is worse
   - mult, rotate, ... requires pre-calculated keys that are large

2. Speed
   - Performs a lot of 64-bit (and sometimes 128-bit) int operations
   - Time complexity may be larger than O(N)
   - Even if O(N), it has large constant factors (L, multiple polynomials, ...)
Challenge++: HE Runs on Diverse Environments!

1. It must be fast on diverse environments
   - On-premise is beneficial because $op(ctxt, ptxt2)$ is faster than $op(ctxt, ctxt2)$
   - Encryption & decryption must be done on the device

2. Utilizing both GPUs and CPUs brings benefits
   - GPUs are fast but less cost-effective than deep learning (no FP ops)
   - High-end server CPUs have many cores & good at int benchmarks
   - Ciphertexts are large: sending them to GPU is intensive
   - Pre-calculated keys may not fit in GPU memory!
... but HE Library Developers Are Busy!

- How to efficiently pack a matrix in ciphertexts?
- How to reduce the error of approximated $\log x$?
- How to use less bootstrap operations?

Can we use compiler optimization techniques to help them?
HEaaN.MLIR

- **Src lang - Poly**: a new lang for high-level operations on polynomial rings
- **Tgt lang - {x86-64, AArch, ...} (+OpenMP) x {CUDA, ROCm, ...}
- In a developing stage; being prototyped using MLIR
- Currently, **HEaaN.MLIR** can:
  - Compile encode + encryption (symm. & pub. key) and decryption written in Poly
  - Provide OpenMP offloading (works well) and CUDA (primitive)
The Poly Lang

module attributes { poly.he_param = "FVa" } {
  func @encrypt(%m: !poly.poly_ntt<30>, %s: !poly.poly_ntt<30>) -> (!poly.poly_ntt<30>, !poly.poly_ntt<30>) {
    // (a, -a*s + m + e)
    %e0 = poly.sample_gaussian (): !poly.poly<30>
    %e = poly.forward_ntt %e0: !poly.poly<30> to !poly.poly_ntt<30>
    %a0 = poly.sample_uniform (): !poly.poly<30>
    %a = poly.forward_ntt %a0: !poly.poly<30> to !poly.poly_ntt<30>
    %as = poly.mult_ntt %a, %s : !poly.poly_ntt<30>
    %asm = poly.sub_ntt %m, %as : !poly.poly_ntt<30>
    %b = poly.add_ntt %asm, %e : !poly.poly_ntt<30>
    return %a, %b: !poly.poly_ntt<30>, !poly.poly_ntt<30>
  }
}

The HE parameter: uses predefined N, L, moduli, ...

Polynomial with 30 moduli, NTT conversion applied

Polynomial with 30 moduli, no NTT

Q: %m is poly after encoding; how to do encoding?
The Poly Lang

```mlir
module attributes { poly.he_param = "FVa" } {
  func @encode_without_ntt(%msg: tensor<65536xcomplex<f64>>) -> !poly.poly<30> {
    %y = poly.from_msg %msg: tensor<65536xcomplex<f64>> to !poly.poly<30>
    return %y: !poly.poly<30>
  }
}
```

Now, you can write encode + encrypt!
Poly-to-tensor

- !poly.poly<L> and !poly.poly_ntt<L> are lowered into tensor<LxNxi64>
- tensor<..> and other types are kept intact
- Poly ops are lowered into Linalg + Tensor ops
- Constant tensors that are necessary for (efficient) calculation are inserted
- Operations that cannot be expressed in Linalg are temporarily represented as external fn calls!
  - Ex: NTT conversion loop: cannot be represented in Linalg.generic’s reduction loop
  - Simply insert `call @__external_forward_NTT(..)` & lower it into SCF at a later pass
Pipeline of HEaaN.MLIR for CPU Code Gen

Poly

Tensor + Linalg

Comprehensive Bufferization

MemRef + Linalg

MemRef + Linalg

MemRef + SCF + OpenMP

OpenMP offloading

MemRef + SCF

External calls into SCF loops

MemRef + Affine

Loop fusion,
Optimize heap alloc for sampling,
Mark loops as parallelizable,
Super vectorization

MemRef + SCF

MemRef + Std + OpenMP

Bare pointer conv.

LLVM IR
Loop Fusion: Collects Low-Hanging Fruits

• Benefit 1: Reduces the size of working sets by $L$ in best cases.
• Benefit 2: Facilitates memory optimization → removes dead heap allocs
• Benefit 3: Removes synchronization points of OpenMP offloaded loops

```c
for (i = 0 to L)
  for (j = 0 to N)
    B[i][j] = op(A[i][j]);

// sync. barrier

for (i = 0 to L)
  for (j = 0 to N)
    C[i][j] = op'(B[i][j]);

// B is now dead alloc.
for (i = 0 to L)
  for (j = 0 to N)
    C[i][j] = op'(op(A[i][j]));
```
Implementing Algorithms in MLIR

• Forward/backward NTT, FFT
• Barrett reduction
• Random sampling, ZO sampling, Gaussian sampling
• Many loops that are specifically necessary for encoding/decoding

Special thanks to Woosung for doing a lot of things from these!!
For Better Debugging Experience

• It is tricky to debug the generated code in terms of correctness & performance.

• To facilitate debugging, we:

  1. Defined a debug dialect and used it: assertion, printer, timer

     ```
     "debug.assert_eq"(%x, %y) {msg = "x and y must be equal"}: (i64, i64) -> ()
     ```

  2. Added a ‘sanitizer’ mode: insert bounds-checking assertions whenever creating memory accessing ops (memref.load/store)
Experimental Results

- Competitor: HEaaN (CryptoLab’s proprietary HE library, use HEXL)
- 3 processors: AMD Ryzen 2990WX, AMD Ryzen 3700, 2 Intel(R) Xeon(R) Gold 6242s
- # Threads: 1 vs. full cores (for Gold: 1 vs. 10 vs. 80)
- Ran 50 times & calculated averages
- Disabled ASLR, set CPU to performance mode, ...
Experimental Results

— Experimental Results —

• Single core results
  • Performance benefit was not clear
  • Gold 6242 has AVX512DQ: Intel HEXL gets benefit

• Multi core results
  • Parallelization was successful
  • Consistently got 40% speedups!

Elapsed Time, Encode-and-encrypt (Pub key.)

(μsec.)
Experimental Results

Elapsed Time, Encode-and-encrypt (Symm Key.)

- HEaaN.MLIR
- HEaaN

Elapsed Time, Decrypt

- HEaaN.MLIR
- HEaaN
Future Works

• Faster NTT conversion: directly invoke Intel HEXL if beneficial

• Fully enable GPU offloading: utilize GPUs in smartphones for en/decryption

• Support more HE ops: primary target is rotation!
  • For some benchmarks, about ½ of running time of matmul in HE is from rotations.

• Correctness of compilation: can we formally verify it?
  • SMT-based validation of transformations on structured loops seems to work well.
  • MLIR-TV*: another on-going (personal) project

* https://github.com/aqjune/mlir-tv
Thank you!