Integrating Fully Homomorphic Encryption Into the MLIR Compiler Framework

Sanath Govindarajan
Mentor: William Moses
Fully Homomorphic Encryption

Run arbitrary functions on private data

Applications: medical, security, cloud computing, etc.

\[ F_{HE}: F(E(\mu)) = E(f(\mu)) \]

Alice (sender)\[ \mu \rightarrow E(\mu) \]

encryption

\[ \mu \rightarrow E(\mu) \]

decryption

Eve (untrusted receiver)\[ E(\mu) \]

\[ F \text{ is an encrypted function that corresponds to } f \]

\[ F(E(\mu)) \]
Pros and Cons of Fully Homomorphic Encryption

- At no point is your data known to anyone except yourself
- At no point are your results known to anyone except for yourself
- At no point is the computation known to anyone except the receiver
- **Theoretically**, ability to translate any existing program into an encrypted program

- FHE libraries are unwieldy and complicated to learn and use
- Not as widespread as other encryption methods
- Tools for fully homomorphic encryption (FHE) still in early stages
- Homomorphically encrypted programs must be constructed from primitives like binary gates, addition, and multiplication
- Very slow compared to other encryption methods because of the need for high-level and low-level optimizations (scheduling etc.)
The Current State of FHE Libraries

- Currently popular libraries: SEAL, HELib, PALISADE
- FHE operations are called using library functions, one primitive at a time
- Different libraries for different schemes
- No cross-operation optimization
  - PRIMES project last year: enabled cross-operation optimization in the GSW (2013) scheme using Halide
  - Still difficult to write complicated functions
  - Limited to bit-wise optimizations
MLIR

- A compiler, not a library
- An SSA-based Multi-Level Intermediate Representation that sits on top of the LLVM IR
- A multi-level optimizer
- Language-independent
- Language-specific
Levels of Optimization

- Normal compiler optimizations (incl. language-specific)
- DAG rewrites (highly language- and scheme-specific)
- Loop scheduling
- Parallelization
- Overall, three levels: syntax/high level, HE level, low (scheduling) level
How Does MLIR Work?

- Opaque operations, not instructions
- Dialects: sets of operations at a similar level
- Progressive lowering: translating from a higher-level dialect to a lower-level dialect (lowest is LLVM IR) and optimizing along the way
- Can mix and match dialects within a single MLIR module
- Example:

```
  HE Dialect
  └── LinAlg
  └── Affine
  └── Loop
  └── Standard
      └── LLVM IR
```
// add x + x, unoptimized
func @add(%x : i64) -> i64 {
    %0 = addi %x, %x : i64
    return %0
}

// add x + x, optimized
// x + x = x << 1
func @add_opt(%x : i64) -> i64 {
    %cst_1 = constant 1 : i64
    %0 = shift_left %x, %cst_1 : (i64, i64) -> i64
    return %cst_1
}
// Take the NAND of a ciphertext with itself
func @self_nand(%input : memref<20x20x128>) {
    "HE.NAND" (%input, %input, %input) {mod = 11 : i128} : (memref<20x20x128>, memref<20x20x128>, memref<20x20x128>) -> ()
        return
}

// optimized: NAND(a, a) = NOT(a)
// removes two unnecessary operations under the hood
// (aka 1 modular matrix multiplication)
func @self_nand_opt(%input : memref<20x20x128>) {
    "HE.NOT" (%input, %input) {mod = 11 : i128} : (memref<20x20x128>, memref<20x20x128>) -> ()
        return
}
Low Level (Scheduling)

MLIR Standard Dialect

Custom FHE Dialect(s)

LLVM IR

// regular loop nest
func @regular() {
...
...
loop.for %arg0 = %c0 to %c200 step %c1 {
    loop.for %arg1 = %c0 to %c200 step %c1 {
        loop.for %arg2 = %c0 to %c200 step %c1 {
            %3 = load %1[%arg2, %arg1] : memref<200x200x i128>
            %4 = load %0[%arg0, %arg2] : memref<200x200x i128>
            %5 = muli %4, %3 : i128
            %6 = load %2[%arg0, %arg1] : memref<200x200x i128>
            %7 = addi %6, %5 : i128
            store %7, %2[%arg0, %arg1] : memref<200x200x i128>
        }
    }
}
return
}

// GPU loop nest
func @gpu {
...
...
    gpu.launch blocks(%arg0, %arg1, %arg2) in (%arg6 = %3, %arg7 = %c1_0, %arg8 = %c1_0) threads(%arg3, %arg4, %arg5) in (%arg9 = %4, %arg10 = %c1_0, %arg11 = %c1_0) {
        %3 = addi %c0, %arg0 : index
        %4 = addi %c0, %arg3 : index
        loop.for %arg12 = %c0 to %c200 step %c1 {
            %7 = load %1[%arg12, %6] : memref<200x200x i128>
            %8 = load %0[%5, %arg12] : memref<200x200x i128>
            %9 = addi %8, %7 : i128
            %10 = load %2[%3, %6] : memref<200x200x i128>
            %11 = addi %10, %9 : i128
            store %11, %2[%3, %6] : memref<200x200x i128>
        }
    }gpu.terminator
}
return
Think about \( f \), not \( F \)!!
Current Work & Results

- GSW (2013) and B/FV (2012) FHE schemes: custom dialects and lowering implemented
  - Custom dialects allow for highly optimized, custom high-level operations such as “HE.identity_minus”, “HE.flatten”, and “BFV.ntt”
- Optimizations across operations, including DAG rewrites: building off my previous work with Walden Yan
- Language- and scheme-specific optimizations, e.g. removing redundant flatten’s and NTT’s
Future Work

- Write dialects and lowering for more FHE schemes such as BFV RNS and CKKS
- Implement “raising” step for all Standard dialect operations - this will allow encryption of any arbitrary program with just one or two compiler flags
- Implement parallelization / multithreading
Conclusion

- The MLIR compiler framework can be used to easily encrypt any program in any compatible programming language by simply passing a flag.
- MLIR also provides a powerful framework for language-specific optimizations - we can take advantage of this to speed up FHE.
- The entire system is modular, allowing you to swap out the FHE scheme that you use, the set of lowering passes, and/or the architecture that you are targeting.
Acknowledgements

- My mentor, William Moses
- My parents
- The PRIMES program and Dr. Srini Devadas
Sources

Questions?