NARLabs

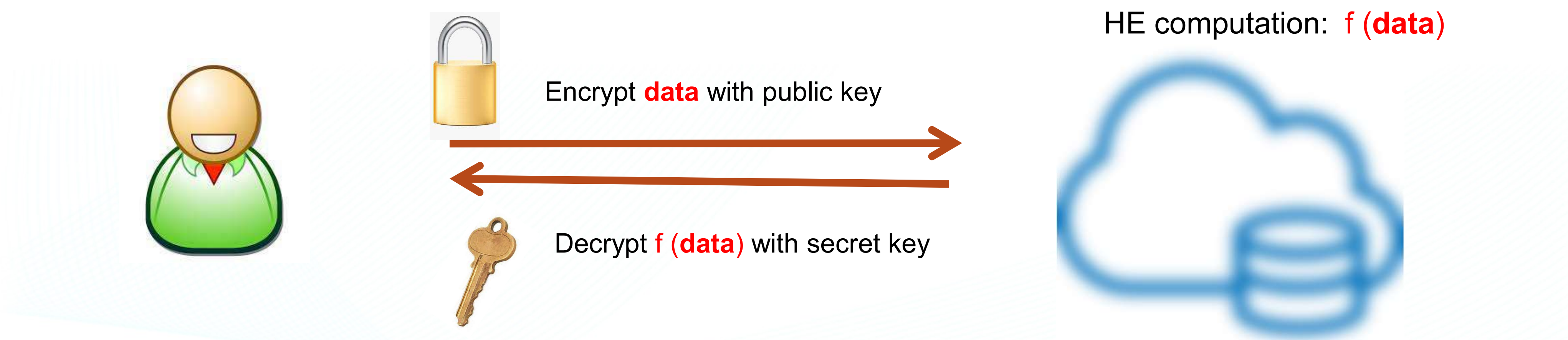
Lattice-based Homomorphic Encryption

Chun-Yu Lin (NCHC)

Jul 6, 2021 @ NCHC NVIDIA Joint-Lab

Homomorphic Encryption

- A privacy-enhanced tech (PET) that allows computation of encrypted data without decryption → f (Enc(x), Enc(y),…) = Enc (f(x,y,…))
- Scenario: computation outsourcing



- E.g. : Private information retrieval
  - Secret database query, to get a raw of DB = {A,B,C,D,E,F,G,……..}
  - Calculate Enc(DB * {0,0,0,1,0,0,0,….})

Types of HE shemes

- Partially HE (only + or ×): El Gamal(*), Paillier(+)…
- Somewhat HE (only **finite** steps)
- Fully HE

Raw RSA as a multiplicative-HE

- Choose coprime p, q, N=pq, and e coprime to ϕ(N)=(p−1)(q−1)
- Secret key: d, the inverse of e (d*e=1 mod ϕ(N).)
- Public key: (e,N)
- Encrypt: $\mathrm{c} \equiv Enc(m) = m^{e} \bmod N$
- Decrypt: $Dec(c) = c^{d} \bmod N$

$$Enc(m_1) \times Enc(m_2) = (m_1 m_2)^{e} = Enc(m_1 m_2)$$

- Based on the hardness of factorization. (Not quantum-safe)
- Not semantic secure ! → If $m_1 = m_2$, then $c_1 = c_2$ (not IND-CPA)

Applications

- (Modern) HE supports many algorithm:
  - Linear algebra, kNN clustering, FFT, and **limited** order statistics (sorting, comparison)
  - Logistic regression, ANN, CNN, and possibly more complicated DL model
  - … inversion and nonlinear functions approximated by polynomials.

- Extension:
  - Multipartite HE (Threshold HE),
  - Proxy Re-Encryption,
  - Attribute-based/Identity base HE….

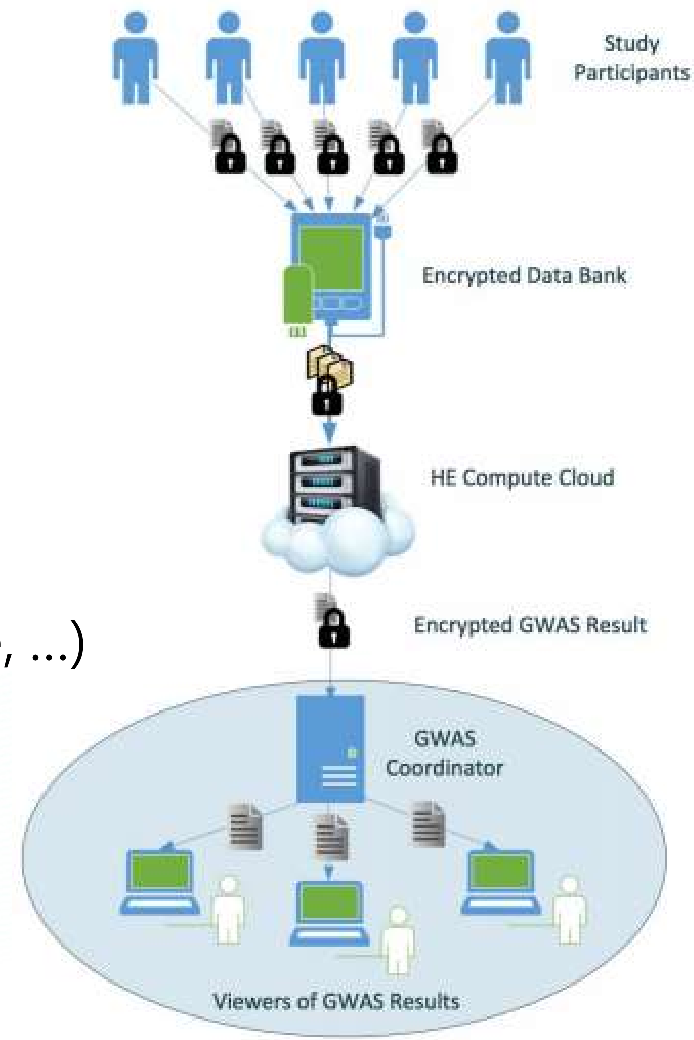
- Scenarios:
  - E-voting (Helios).
  - Private information retrieval,
  - Secure info aggregation
  - Medical data analysis ….

Application: GWAS for iDASH'18 competition

GWAS (Genome Wide Association Studies)
for the age-related macular degeneration (AMD)

- 27 k samples * 263 k SNP markers
- Computing tasks
  - allelic chi-square test
  - logistic regression with covariates (sex, age, …)
- HE ~6x faster than SMPC reported

https://eprint.iacr.org/2020/563



Homomorphic Encryption based on Lattice

- Lattice problem can be reduced to learning-with-error (LWE) problem…
  - Each operations increase noise in ciphertexts
- Proper noise management become crucial:
  - Bootstrapping:
    - Homomorphically decrypt with encrypted secret key and re-encrypt
  - w/o bootstrapping for finite level scheme:
    - Modulus switching + key switching
- Fully HE proposed by Gentry 2009

  → Bootstrappable somewhat HE + Bootstrapping

  (decryptable within capacity)

- Proposed tech standard by community (2019)

Example: Noise in the simple integer-based HE

- Key: odd, large q
- Plaintext m in $[0,1]$
- $\mathrm{Enc}(m) = c = kq+2e+m$
- $\mathrm{Dec}(c) = m = c \bmod q \bmod 2$

- $\mathrm{Eval}(c1 + c2) = q(k1+k2) + 2(e1+e2) + \mathbf{m1+m2}$
- $\mathrm{Eval}(c1{*}c2) = q(..) + 2(e1{*}e2+e1{*}m2+e2{*}m1) + \mathbf{m1{*}m2}$

- Decryption works for noise $< q$

Fully Homomorphic Encryption over the Integers, *van Dijk, Gentry, Halevi and Vaikuntanathan (2009)*

If too much noise in ciphertext…

- Bootstrapping
- Reduce "capacity" dropping
  (HE w/ high a level)

```
Round: 0
 multiplication: 0     ctxt.bitCapacity: 332
 multiplication: 1     ctxt.bitCapacity: 320
 multiplication: 2     ctxt.bitCapacity: 308
 multiplication: 3     ctxt.bitCapacity: 296
 multiplication: 4     ctxt.bitCapacity: 284
 multiplication: 5     ctxt.bitCapacity: 271
 multiplication: 6     ctxt.bitCapacity: 259
 multiplication: 7     ctxt.bitCapacity: 247
 multiplication: 8     ctxt.bitCapacity: 235
 multiplication: 9     ctxt.bitCapacity: 223
 multiplication: 10     ctxt.bitCapacity: 211
 multiplication: 11     ctxt.bitCapacity: 198
 multiplication: 12     ctxt.bitCapacity: 186
 multiplication: 13     ctxt.bitCapacity: 173
 multiplication: 14     ctxt.bitCapacity: 161
 multiplication: 15     ctxt.bitCapacity: 149
 multiplication: 16     ctxt.bitCapacity: 137
 multiplication: 17     ctxt.bitCapacity: 125
 multiplication: 18     ctxt.bitCapacity: 113
 multiplication: 19     ctxt.bitCapacity: 101
Before recryption, capacity=88.9707, p^r=2
After recryption, capacity=332.265, p^r=2
```

What’s the scheme “w/o bootstrapping” …

NARLabs

Key-switching : re-linearize ciphertexts   (symmetric version in BV11)

https://eprint.iacr.org/2011/344

- Let consider the BV scheme

$$c = \mathsf{Encrypt}(m) = (\mathbf{A}, b = \mathbf{A} \cdot \mathbf{s} + 2e + m)$$

$$m = \mathsf{Decrypt}(c) = (b - \mathbf{A} \cdot \mathbf{s}) \mod q \mod 2.$$

N+1 components

$A, s \in \mathbb{Z}_q^n$

- Under multiplication, Eval(c1*c2):

$$(b - \mathbf{A} \cdot \mathbf{s})(b' - \mathbf{A}' \cdot \mathbf{s}) = bb' + h_i s_i + h_{ij} s_i s_j$$
$$= bb' + h_i(b^{(i)} - \mathbf{A}^{(i)} \cdot \mathbf{t}) + h_{ij}(b^{(ij)} - \mathbf{A}^{(ij)} \cdot \mathbf{t})$$
$$= bb' + h_i b^{(i)} + h_{ij} b^{(ij)} - (h_i \mathbf{A}^{(i)} + h_{ij} \mathbf{A}^{(ij)}) \cdot \mathbf{t}$$

- Solution: add extra keys:

$$b^{(i)} = \mathbf{A}^{(i)} \cdot \mathbf{t} + 2e^{(i)} + s_i$$

$$b^{(ij)} = \mathbf{A}^{(ij)} \cdot \mathbf{t} + 2e^{(ij)} + s_i s_j$$

- Under L-level *, the key chain si={s0, s1,…,sL} and ci = (A′, b′=A′ si + 2e′+m) → Multiplication key.
- One can switch any key with the same encrypted message but extra errors. → Rotation key

Modular switching : reduce noise

- Consider fixed q and c with noise level B, after L-level multiplication gives B^2^L

  → log q > 2^L log(B)   grows exponentially

- Instead, use dynamical $q_\ell = \{q_0, q_1, q_2, \ldots, q_L\}$ and rescale $c' = (q_{\ell+1}/q_\ell)c$ after each *.

  → Noise level reduced by $q_{\ell+1}/q_\ell$

- For example,

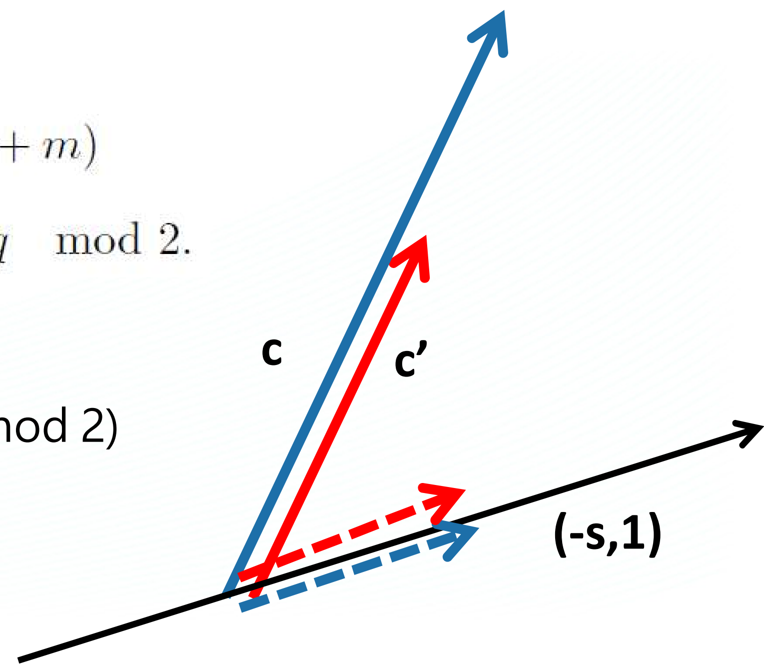
$$c = \text{Encrypt}(m) = (\mathbf{A}, b = \mathbf{A} \cdot \mathbf{s} + 2e + m)$$

$$m = \text{Decrypt}(c) = (b - \mathbf{A} \cdot \mathbf{s}) \mod q \mod 2.$$

  to reduce q → p, chose c' ~ p/q c (mod 2)

  → same decryption: (b'-A's) (mod p) = (b-As) (mod q)   (mod 2)

  → lower noise



https://eprint.iacr.org/2011/277

Packed plain/ciphertexts : a performance breakthrough

- Plain-/cipher-text space allows 1-to-many mapping to the messages

a0	a1	a2	….	

⇄

plain-/ciphertext = f(x) = $a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$

coefficient-basis ←DFT→ evaluation-basis [1]

- Typically >8k +,* SIMD-style operation at a time.

- Rotation within slot:

0	1	2	3

3	0	1	2

- Arbitrary permutation = rotation + masking-* + addition.

[1] For example, see Prof. Tsai's lecture note on digital signal analysis (NTU 2017)

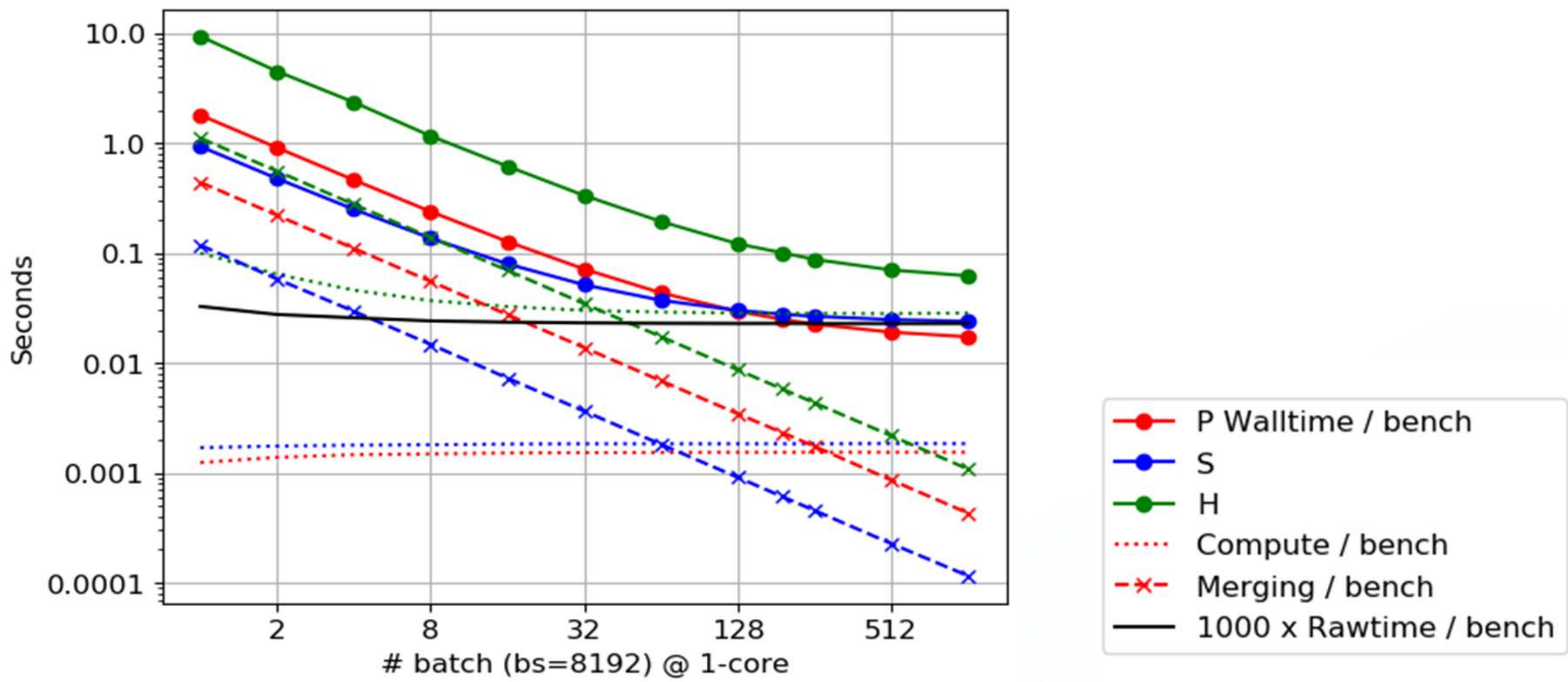
Popular implementations & their functionality

Projects \ Scheme	BGV [int]	CKKS [real]	BFV [int]	FHEW [bool]	CKKS Bootstrapping	TFHE [bool]
HELib (2013-)	Yes	Yes	No	No	No	No
MS SEAL (2018-)	No	Yes	Yes	No	No	No
Palisade (2017-)	Yes	Yes	Yes	Yes	No	Yes
HEAAN (2016-)	No	Yes	No	No	Yes	No
FHEW (2014-)	No	No	No	Yes	No	No
TFHE (2016-)	No	No	No	No	No	Yes
FV-NFLlib	No	No	Yes	No	No	No
NuFHE	No	No	No	No	No	Yes
Lattigo	No	Yes	Yes	No	No	No

- cuHE (2015-): Accelerate polynomial-based HE with GPU
- Google's transpiler for FHE (2021-): compile user code into encrypted Boolean arithmetic based on TFHE

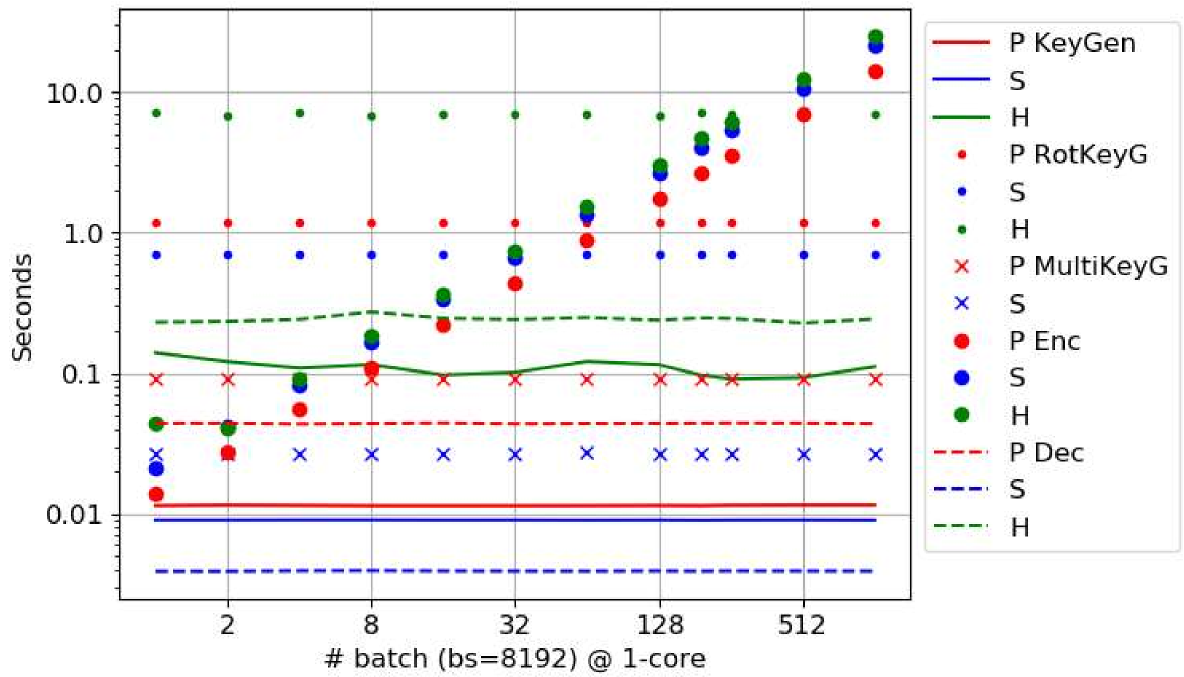
https://en.wikipedia.org/wiki/Homomorphic_encryption

Squared sum of 8M items : Compare HELib / Palisade / Seal



- HE ~ O(1000x) slower than native computation for this depth-1 multiplication case.
- Palisade marginally beats SEAL's for batch number >128 (BV key switching w/o auto-rescale ~2x than Hybrid key switching).
- HELib is ~3-4x slower! Still under investigation.
- Testing code at https://github.com/chunyulin/he/tree/main/compare_reorder

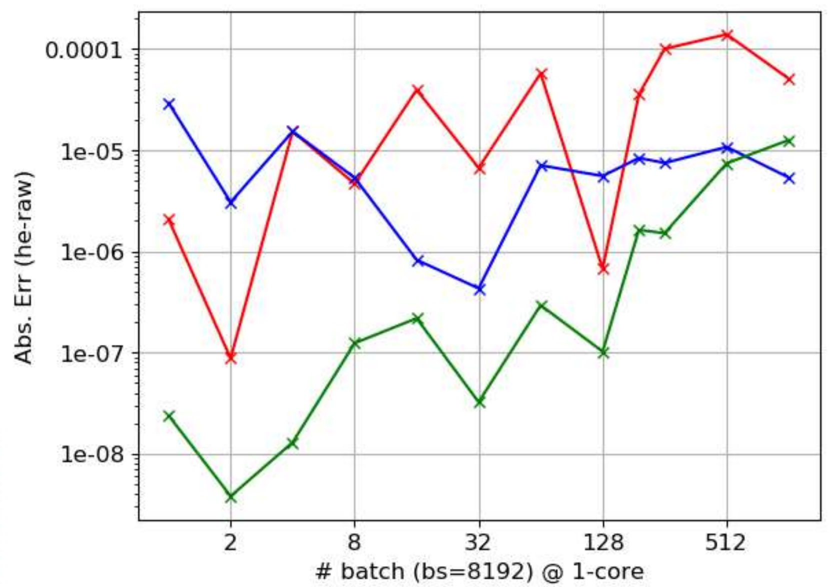
Further profiling



HELib combines MultKeyG/RotKeyG
Seal faster in KeyGen, Decoding,
Palisade faster in Encoding, Computing (dominated)

Accuracy

Still not a complete fair comparison….



Benchmarks (Palisade)

- Measure **effective FLOPS per core** = Exact flops / evaluation time / core

	BGV: Sum(x^2)	BFV: Sum(x^2)	CKKS: Sum(x^2)	CKKS: Sum(sigmoid(x))
Exact flops	2 N (integer)		2 N	7 N
Walltime HE for 8M terms	30 s	40 s	16 s	91 s
Eval-only performance	HE: 1576 KF Raw: 1379 MF **875x**	HE: 668 KF Raw: 1374 MF **2155x**	HE: 9707 KF Raw: 813 MF **84x**	HE: 238 KF Raw: 807 MF **3387x**
Note: (–O3 –std=c++17)	Rd=8192 nMult=1 (*), maxdepth=1		Rd = 2*8192 nMult=1 (*), maxdepth=1 APPROXRESCALE, BV, rw=10, sf=39, fb=60	Rd = 2*8192 nMult=3 (*), maxdepth=1 APPROXRESCALE, HYBRID, rw=10, sf=39, fb=60

Raw: native computation w/o HE
MF = Mega FLOPS
KF = Kilo FLOPS
All timing is for 1-core on ThunderX2@2GHz
(nice OpenMP scaling; ~2x faster on Twnia-2)

Sigmoid: $\sigma(x) \approx \frac{1}{2} + \frac{x}{4} - \frac{x^3}{48}$

Example: delayed relineralization and rescaling

- $\sigma(x) \approx \frac{1}{2} + \frac{x}{4} - \frac{x^3}{48} \;=\; \frac{1}{2} + (\frac{1}{4}x) + (-\frac{1}{48}x)(xx)$

```
auto bsum =
for (i: num_

   xo48  = Eva
   x2    = Eva
   x3    = Eva
   xo4   = Eva
   xo4   = Eva
   tmp   = Eva
   bsum  = Eva
}


auto sum = E
```

```
auto bsum = 0
for (i: num_b

   xo48  = Eval
   x2    = Eval
   x3    = Eval
   xo4   = Eval
   xo4   = Eval
   tmp   = Eval
   bsum  = Eval
}

bsum = Reline
auto sum = Ev
```

```
auto bsum = 0;
for (i: num_batch) {      // ~8k terms in each batch

   xo48 = Rescale( EvalMult(-0.0208333333333333, ctx[i])); // D1
   x2   = Rescale( EvalMult(ctx[i], ctx[i]));               // D1
   x3   = Rescale( EvalMultNoRelin(xo48, x2) );             // D1
   xo4  = Rescale( EvalMult(ctx[i], 0.25) );                // D1
   xo4  = EvalAdd(xo4, x3);                           // D1+D1  OK
   tmp  = EvalAdd(0.5, xo4);
   bsum = EvalAdd(bsum, tmp);
}

bsum = Relinearize(bsum);
auto sum = EvalSum(bsum);
```

Concluding remarks

- Lattice-based HE has almost become usable.
- Leveled HE are widely supported.
- Yet CKKS bootstrapping implementation is still limited.
- Modern schemes mostly includes optimization like Reduce Number System and reduced error approaches
- Performance can vary a lot with order of operation / parameters / schemes.
- HE could be an wide, interesting research topic both from the aspect of fundamental study, computation, and application scenario.

~ Thank you ~