
Introduction to FHE Acceleration

Todd Austin
University of Michigan

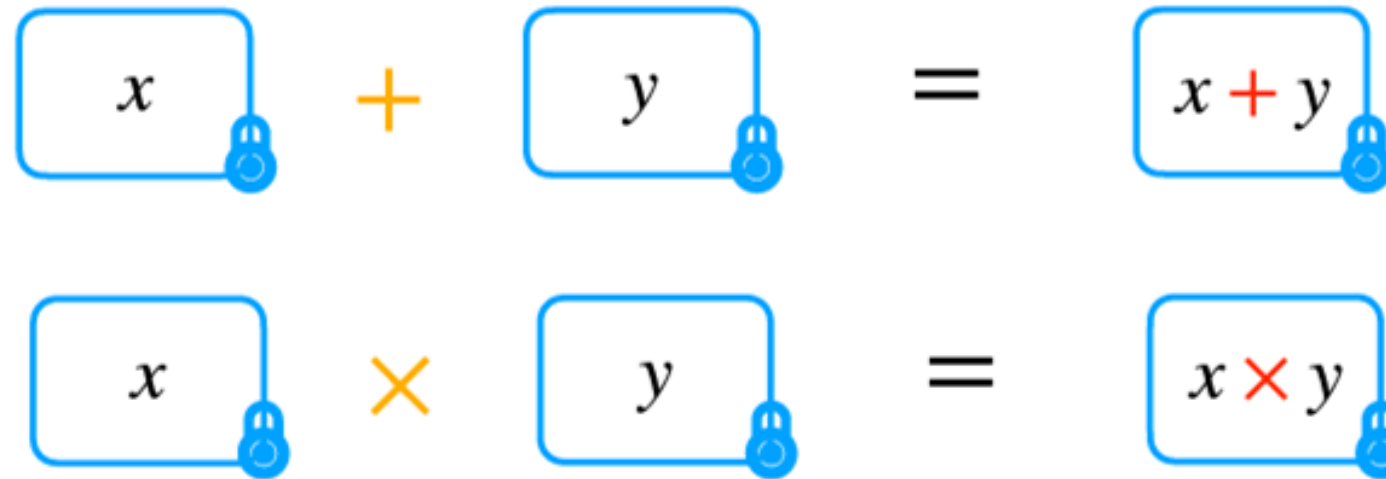
Agenda for the Today's Lecture

- FHE Recap
- Core FHE Optimizations
- Accelerating FHE
- Comparative Performance Analysis

Agenda for the Today's Lecture

- FHE Recap
- Core FHE Optimizations
- Accelerating FHE
- Comparative Performance Analysis

FHE Recap



FHE = Computations over encrypted messages

- Possibly any function (“Fully”)
- Bit, integer, real messages
- Secret key and public key encryption

A world full of noise

An example: DGHV

2010 - Van Dijk, Gentry, Halevi, Vaikuntanathan

Security
Approximate GCD
problem

- $m \in \{0,1\}$ message
- $p \in \mathbb{Z}$ large odd secret
- $q \in \mathbb{Z}$ way larger than p
- $e \in \mathbb{Z}$ way smaller than p , called *noise*

Decryption without p requires factoring pq , which has complexity $O(\sim e^{N \log N})$ where N is the number of bits needed to represent p

Encryption: $m \mapsto c = pq + 2e + m$

Decryption: $c \mapsto m = (c \bmod p) \bmod 2$

$$m = ((pq + 2e + m) \bmod p) \bmod 2$$

$$m = ((2e + m) \bmod 2)$$

$$m = m$$

A world full of noise 🌡️

An example: DGHV

$$c_1 = pq_1 + 2e_1 + m_1$$

$$c_2 = pq_2 + 2e_2 + m_2$$

Multiples of p

Homomorphic addition
(XOR)

$$c_1 + c_2 = p \cdot (q_1 + q_2) + 2 \cdot (e_1 + e_2) + m_1 + m_2$$

$$(m_1 + m_2) \bmod 2 = m_1 \wedge m_2$$

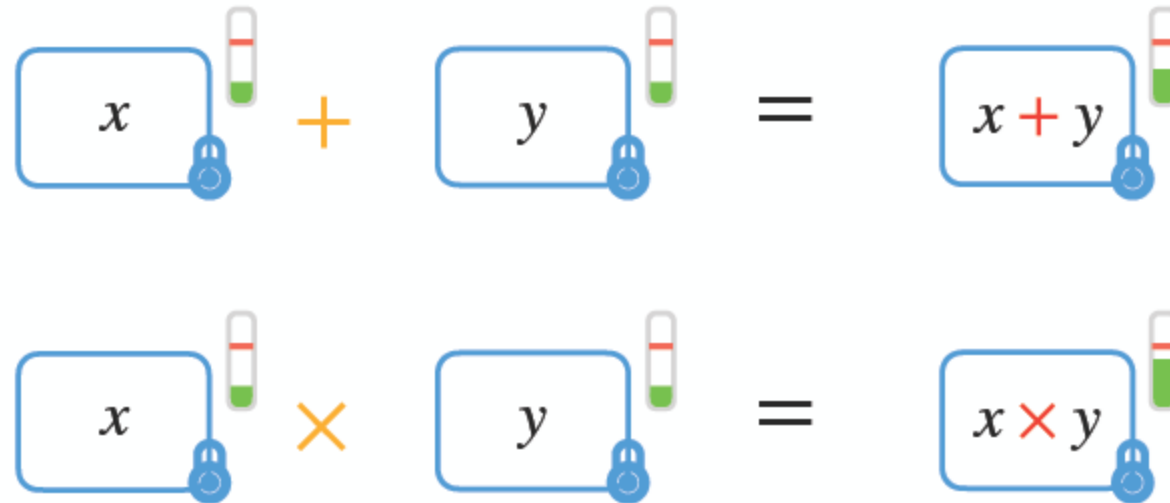
Homomorphic multiplication
(AND)



$$c_1 \cdot c_2 = p \cdot (pq_1q_2 + \dots) + 2 \cdot (2e_1e_2 + \dots) + m_1m_2$$

$$(m_1m_2) \bmod 2 = m_1 \& m_2$$

Noise grows too much 🌡️ \Rightarrow decryption incorrect 🚨

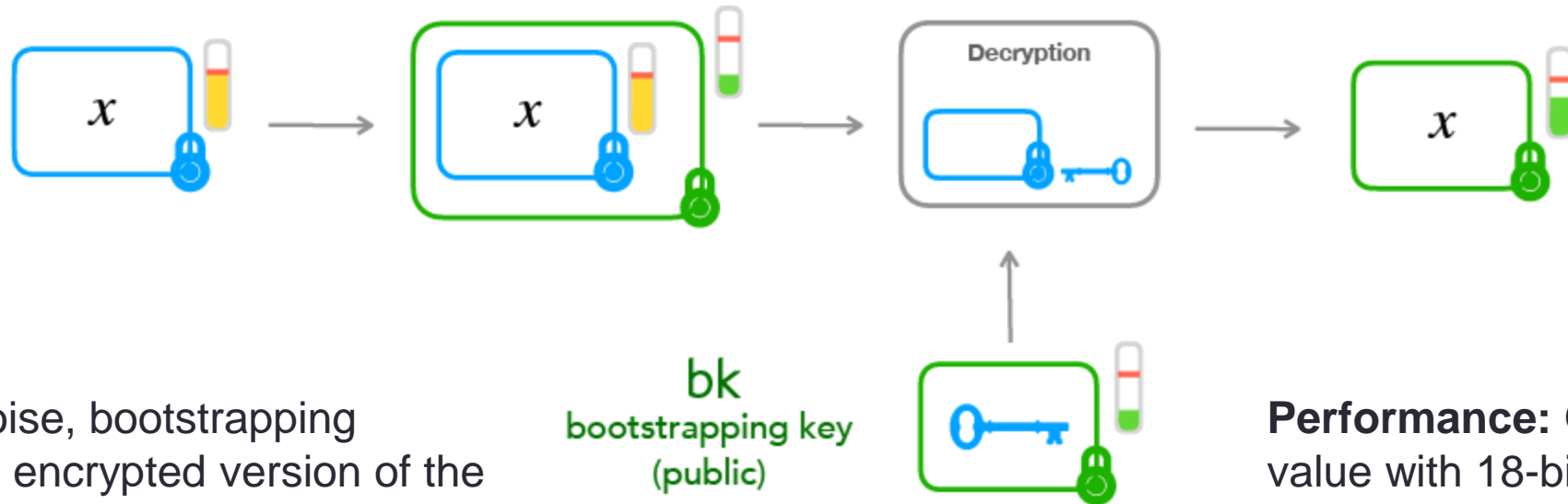
Noise



Noise grows too much  \Rightarrow decryption incorrect 

Some FHE systems utilize statistical noise models, where decryptions can fail with a small probability.

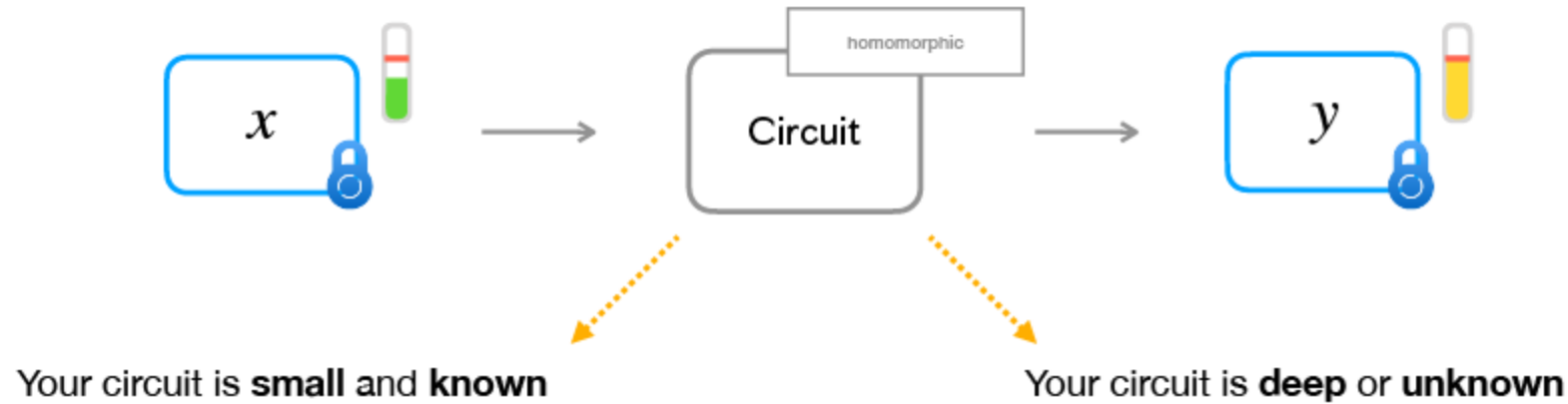
Bootstrapping [Gen09]



To reduce noise, bootstrapping leverages an encrypted version of the secret key to "decrypt" the ciphertext under encryption. This is possible because FHE allows evaluating functions on encrypted data.

Performance: OpenFHE CKKS value with 18-bit precision, single-threaded CPU: **670 ms**

To bootstrap or not to bootstrap?



Leveled approach

- The larger the circuit, the larger the crypto parameters, the slower the evaluation
- Circuit depth must be known in advance

Bootstrapped approach

- No depth limitations
- Bootstrap when needed



Agenda for the Today's Lecture

- FHE Recap
- **Core FHE Optimizations**
- Accelerating FHE
- Comparative Performance Analysis

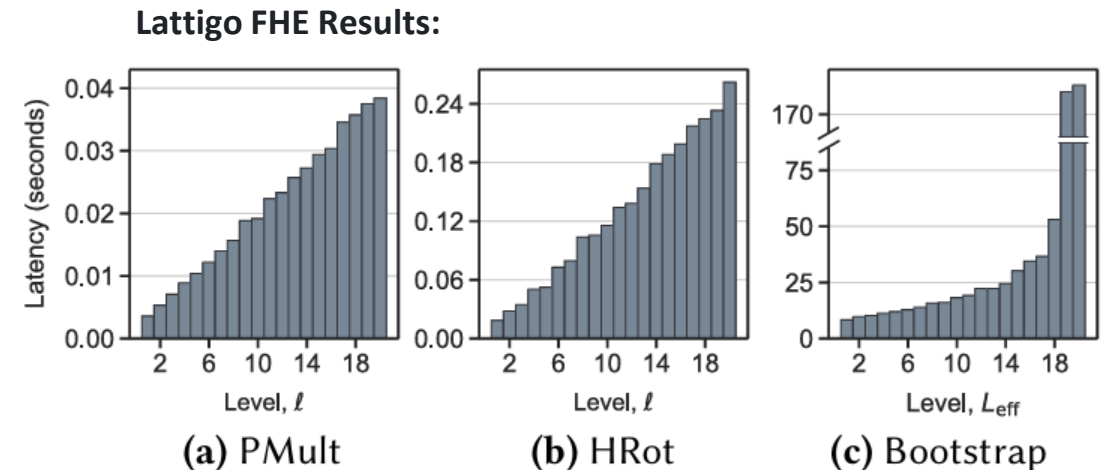
How is FHE Ciphertext Represented

- Ciphertexts are expressed as a set of polynomial elements
 - Stored as a series of integer coefficients
- x is not an actual input
 - It is a symbolic variable used to define
 - Think of it as a placeholder between coefficients
- Storing ciphertexts as polynomials simplifies algebraic operations
 - First, encode, then encrypt
 - **Batching** puts multiple messages into one plaintext polynomial
 - Typical size: $2^{12} - 2^{16}$ degree!

$$ct = (c_0(x), c_1(x), \dots, c_k(x))$$

$$c_i(x) = c_{i,0} + c_{i,1}x + c_{i,2}x^2 + \dots + c_{i,n-1}x^{n-1}$$

$$m \longrightarrow m + 0 \cdot x + 0 \cdot x^2 + \dots + 0 \cdot x^{n-1}$$



Key Ciphertext Parameters

- **Polynomial Degree (N)**
 - Degree of polynomial rings used (often powers-of-two, e.g., $N=2048, 4096, 8192, \dots$).
 - Determines ciphertext size, security, and computational complexity
 - Increasing N increases security but also computational cost and ciphertext size
- **Ciphertext Modulus (Q)**
 - Large integer modulus under which polynomial are computed (often $Q \approx 2^{200} - 2^{2000+}$)
 - Dictates the maximum allowed **noise margin** and ciphertext **precision**
 - Larger Q provides more noise budget but reduces security and increases ct size
 - Q is typically a product of many machine-sized co-primes to support parallel RNS computation
- **Noise Margin (L)**
 - **Multiplicative depth** before your ciphertext becomes undecipherable
 - $L=5$ means your ciphertext can perform 5 dependent multiplies before data is corrupted

Core Opt #1: Residue Number Systems (RNS)

- Break a large coefficient into multiple **co-prime** factors
- Perform modulus of the factors
 - Each factor is eval'ed individually
 - Use **CRT reconstruction** to get wrt Q
- Advantages
 - Fast – typically uses built-in 64-bit ints
 - Parallel – all ops can be in parallel
- Why does this work
 - Due to Chinese Remainder Theorem (CRT)

$$Q = q_0 \times q_1 \times q_2 \times \cdots \times q_L$$

A single large number modulo Q :

$$x \bmod Q$$

Using RNS:

We represent $x \bmod Q$ as multiple residues modulo smaller primes q_i :

$$x \rightarrow [x \bmod q_0, x \bmod q_1, x \bmod q_2, \dots, x \bmod q_L]$$

Chinese Remainder Theorem (CRT)

Formal Definition:

Given pairwise coprime integers m_1, m_2, \dots, m_k (meaning no common factors other than 1), and given integers a_1, a_2, \dots, a_k , the Chinese Remainder Theorem guarantees a unique solution modulo the product $M = m_1 \times m_2 \times \dots \times m_k$ to the following congruences:

$$\begin{aligned}x &\equiv a_1 \pmod{m_1} \\x &\equiv a_2 \pmod{m_2} \\&\vdots \\x &\equiv a_k \pmod{m_k}\end{aligned}$$

Specifically, the CRT states:

- There is exactly **one unique integer** solution x , modulo $M = m_1 m_2 \dots m_k$.
- This integer solution can always be computed efficiently.

Example of CRT (Concrete):

Solve the congruences:

$$\begin{aligned}x &\equiv 2 \pmod{3} \\x &\equiv 3 \pmod{5} \\x &\equiv 2 \pmod{7}\end{aligned}$$

Step-by-step solution:

- First, calculate $M = 3 \times 5 \times 7 = 105$.
- Compute each partial modulus:
 - $M_1 = 105/3 = 35$, $M_2 = 105/5 = 21$, $M_3 = 105/7 = 15$
- Find inverses modulo each m_i :
 - $35 \pmod{3} = 2$, inverse of $2 \pmod{3} = 2$ since $2 \times 2 = 4 \equiv 1 \pmod{3}$.
 - $21 \pmod{5} = 1$, inverse of $1 \pmod{5} = 1$.
 - $15 \pmod{7} = 1$, inverse of $1 \pmod{7} = 1$.
- Construct the solution:

$$x = a_1 M_1 y_1 + a_2 M_2 y_2 + a_3 M_3 y_3 \pmod{105}$$

Plugging in the values:

$$x = (2 \times 35 \times 2) + (3 \times 21 \times 1) + (2 \times 15 \times 1) \pmod{105}$$

Thus,

$$x = (140 + 63 + 30) \pmod{105} = 233 \pmod{105} = 23$$

Verify:

$$23 \pmod{3} = 2, 23 \pmod{5} = 3, 23 \pmod{7} = 2. \text{ Correct!}$$

Core Opt #2: Point-Value Multiplication

- Polynomial multiplication, which occurs when HE multiplies occur has $O(N^2)$ complexity

$$O(N^2) \quad \boxed{c_k = \sum_{i=0}^k a_i b_{k-i}, \quad \text{for } k = 0, 1, 2, \dots, 2n}$$

- Pointwise multiplication, has $O(N)$ complexity
 - Polynomial is represented by $N+1$ points

$$C(x_i) = A(x_i) \times B(x_i), \quad \text{for all } i = 0, 1, \dots, n-1$$

- Converting to point-wise form requires an NTT with $O(N \log N)$
 - NTT is FFT modulo a prime
 - Overall complexity then is $O(N \log N)$

$$O(N \log N) \quad \boxed{\begin{aligned} A(x) &\xrightarrow{\text{NTT}} A(\omega^0), A(\omega^1), \dots, A(\omega^{n-1}) \\ B(x) &\xrightarrow{\text{NTT}} B(\omega^0), B(\omega^1), \dots, B(\omega^{n-1}) \\ C(\omega^i) &= A(\omega^i) \times B(\omega^i), \quad \text{for } i = 0, 1, \dots, n-1 \\ C(\omega^0), C(\omega^1), \dots, C(\omega^{n-1}) &\xrightarrow{\text{iNTT}} C(x) \end{aligned}}$$

Polynomial vs. Point-Value Multiply

Example Polynomials:

Let's multiply two simple polynomials:

$$A(x) = 1 + 2x + 3x^2$$

$$B(x) = 4 + 5x + 6x^2$$

$$(1 + 2x + 3x^2)(4 + 5x + 6x^2)$$

$$C(x) = 4 + 13x + 28x^2 + 27x^3 + 18x^4$$

Point-value Multiplication

Evaluate $A(x)$:

x	$A(x)$
0	$1 + 0 + 0 = 1$
1	$1 + 2 + 3 = 6$
-1	$1 - 2 + 3 = 2$
2	$1 + 4 + 12 = 17$
-2	$1 - 4 + 12 = 9$

Evaluate $B(x)$:

x	$B(x)$
0	$4 + 0 + 0 = 4$
1	$4 + 5 + 6 = 15$
-1	$4 - 5 + 6 = 5$
2	$4 + 10 + 24 = 38$
-2	$4 - 10 + 24 = 18$

Multiply corresponding point-values directly:

x	$A(x) \times B(x)$
0	$1 \times 4 = 4$
1	$6 \times 15 = 90$
-1	$2 \times 5 = 10$
2	$17 \times 38 = 646$
-2	$9 \times 18 = 162$

Number Theoretic Transform (NTT) Primer

- Efficiently converts from **coefficient form** into **point-value form**
 - Simple evaluation is $O(N^2)$, and N get LARGE
 - NTTs accomplish this task with $O(N\log N)$
 - NTTs leverage structure to divide-and-conquer
- Steps for $O(N\log N)$ multiplication
 - Transform polynomial into a point-value via NTT — $O(N\log N)$
 - Doing pointwise multiplication — $O(N)$
 - Doing an inverse NTT — $O(N\log N)$

♦ What is NTT?

The NTT is a way to transform a sequence of numbers into a different form that makes certain operations (like convolution or polynomial multiplication) much faster. It's like a "modular FFT".

- **Input:** A list of numbers (usually coefficients of a polynomial).
- **Output:** A transformed list of numbers (the NTT of the input).

It operates over a **finite field**, meaning integers modulo a prime number.

NTT's Core Trick: Exploit Symmetry

- NTT picks symmetric evaluation points to allow recursion and reuse of, for $O(N\log N)$ complexity
- $f(x)$ is evaluated at **roots of unity**
 - Roots are chosen to create patterns that repeat and cycle
 - Allows one to divide the problem into smaller ones that reuse x-values
 - At each level, combine results from the smaller problems — no recomputation
 - All computations are reused, and the total work drops from n^2 to $n\log n$

◆ The Key Idea: Use Roots of Unity

Instead of random points, we evaluate the polynomial at the powers of a primitive root of unity:

$$\omega^0, \omega^1, \omega^2, \dots, \omega^{N-1}$$

Where:

$$\omega_n^{n/2+k} = -\omega_n^k$$

- ω is a **primitive N-th root of unity** modulo a prime p
- This means $\omega^N \equiv 1 \pmod{p}$, and no smaller power equals 1

These points are **evenly spaced on a circle**, algebraically speaking — just like FFT uses complex roots of unity, NTT uses modular roots of unity.

Coolley-Tukey Algorithm

Step 2: Divide and Conquer with Even/Odd Splitting

Goal

Given a polynomial or sequence:

$$a(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$$

We want to compute its DFT of length n , i.e.,

$$A_k = \sum_{j=0}^{n-1} a_j \cdot \omega^{jk}, \quad \text{for } k = 0, 1, \dots, n-1$$

Where:

- ω is a **primitive n -th root of unity**
 - For FFT: $\omega = e^{2\pi i/n}$
 - For NTT: $\omega \in \mathbb{Z}_q$, satisfying $\omega^n = 1 \pmod q$

Step 2: Recursive Splitting Using Even and Odd Terms

We exploit a **trick**: we can express the DFT of size n in terms of two DFTs of size $n/2$.

Let's split $a(x)$ into:

- $a^{(even)}(x) = a_0 + a_2x + a_4x^2 + \dots$
- $a^{(odd)}(x) = a_1 + a_3x + a_5x^2 + \dots$

Then, the original polynomial becomes:

$$a(x) = a^{(even)}(x^2) + x \cdot a^{(odd)}(x^2)$$

This polynomial identity is the heart of the divide-and-conquer.

Coolley-Tukey Algorithm

Now, Apply the DFT to Both Halves

We want to evaluate $a(x)$ at all powers of ω : $\omega^0, \omega^1, \dots, \omega^{n-1}$

Use the identity:

$$a(\omega^k) = a^{(even)}(\omega^{2k}) + \omega^k \cdot a^{(odd)}(\omega^{2k})$$

This lets us compute the DFT of size n by:

1. **Evaluating the DFTs of even and odd subsequences** at ω^{2k} are $n/2$ -th roots of unity)
2. **Combining them** using twiddle factors ω^k

Setting Up the Coolley-Tukey Butterfly

Let:

- $E_k = \text{DFT}_{n/2}(a^{(odd)})$

- Then:

$$A_k = E_k + \omega^k \cdot O_k$$

$$A_{k+n/2} = E_k - \omega^k \cdot O_k$$

This gives you the full DFT of length n , but built from two DFTs of length $n/2$.

NTT Butterfly Evaluation

- Radix-2 Cooley-Tukey NTT structure
 - Performs NTT for degree $N=8$
- Stage 1: Butterfly consecutive values
 - Sum: $a+b$
 - Difference: $a-b\cdot\omega$
- Stage 2: Butterfly results 2 apart
 - Same operations, but with new ω 's
- Stage 3: Butterfly results 4 apart
 - Same operations, but with new ω 's
- Inverse works as well
 - Total work: $O(N\log N)$

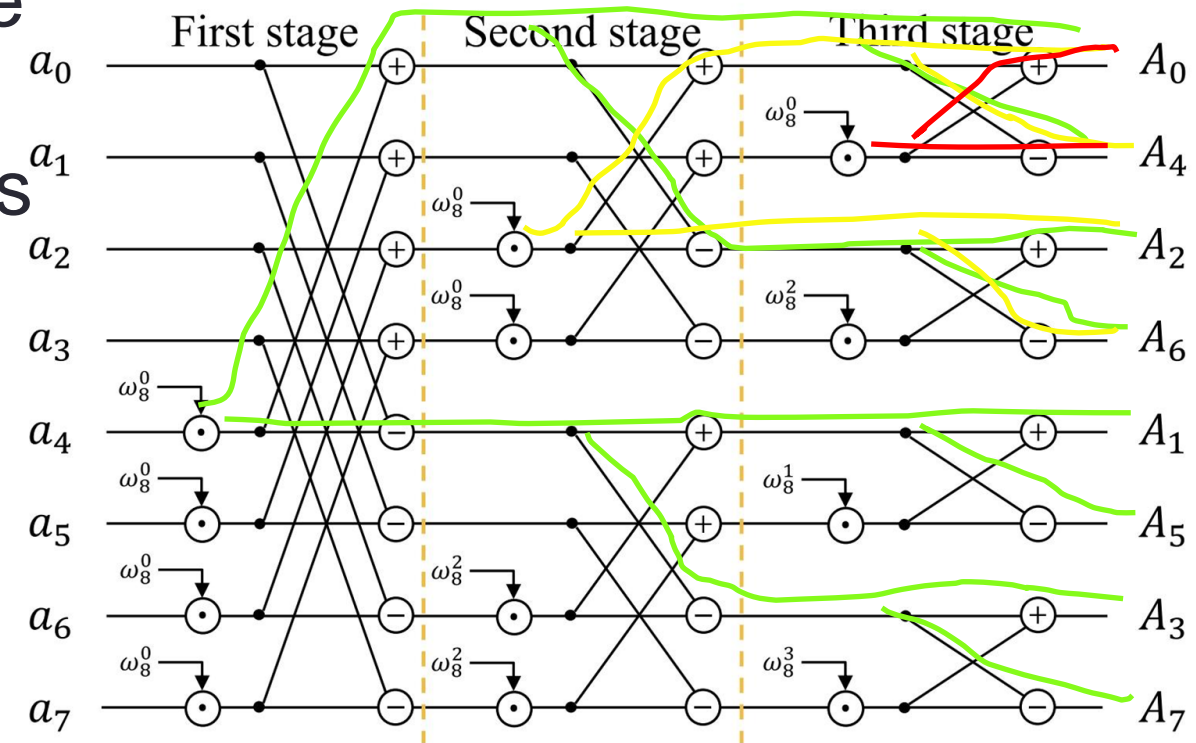


Fig. 4. Workflow of NTT/FFT with $n = 8$ with CT butterfly

Agenda for the Today's Lecture

- FHE Recap
- Core FHE Optimizations
- **Accelerating FHE**
- Comparative Performance Analysis

Orion FHE Compiler for DNNs

- What is Orion? An FHE framework for efficient deep learning inference, automatically converting PyTorch models into CKKS-based FHE computations
- What compile-time optimizations are introduced?
 - Efficient ciphertext packing: Diagonal-encoded MV multiplication with baby-step giant-step (BSGS) and Double Hoisted Rotation (DHR) to reduce ciphertext rotations
 - Optimal bootstrap placement: Proactive bootstrap placement using a latency-aware DAG for minimal inference latency
 - Automatic management of low-level details: Including polynomial approximations, fixed-point scaling, and large data structures
- Performance results: Orion achieves 2.38x overall speed-up over SotA compilers, reduces ciphertext rotations by up to 6.4x, and optimizes bootstrap placement to run up to 255x faster



Figure 8. The first homomorphic object detection and localization results. Labels indicate each object’s predicted class and its confidence score in $[0, 1]$.

Table 3. A comparison of ciphertext rotation counts in CIFAR-10 networks between Lee et al. [52] and Orion.

Work	ResNet-20	ResNet-110	VGG-16	AlexNet
Lee et al. [52]	1382	7622	9214	9422
Orion (us)	836	4676	1771	1470
Improvement	1.65×	1.64×	5.20×	6.41×

HEXL: HE on Intel AVX-512

- Intel HEXL speeds up homomorphic encryption using AVX-512
- Focuses on fast modular multiplication and NTT
- Integrates with SEAL and PALISADE libraries
- Uses SIMD vectorization for higher performance on Intel CPUs

Benchmark	Speedup
Forward NTT	4.70x
Inverse NTT	5.46x
BFV Encrypt	1.54x
BFV Decrypt	2.48x
BFV Multiply	1.43x
BFV Multiply Relinearize	1.56x
BFV Rotate 1	2.38x
CKKS Encode	1.68x
CKKS Decode	1.23x
CKKS Encrypt	1.87x
CKKS Decrypt	2.80x
CKKS Multiply	2.66x
CKKS Multiply Relinearize	2.22x
CKKS Rescale	3.26x
CKKS Rotate 1	2.08x

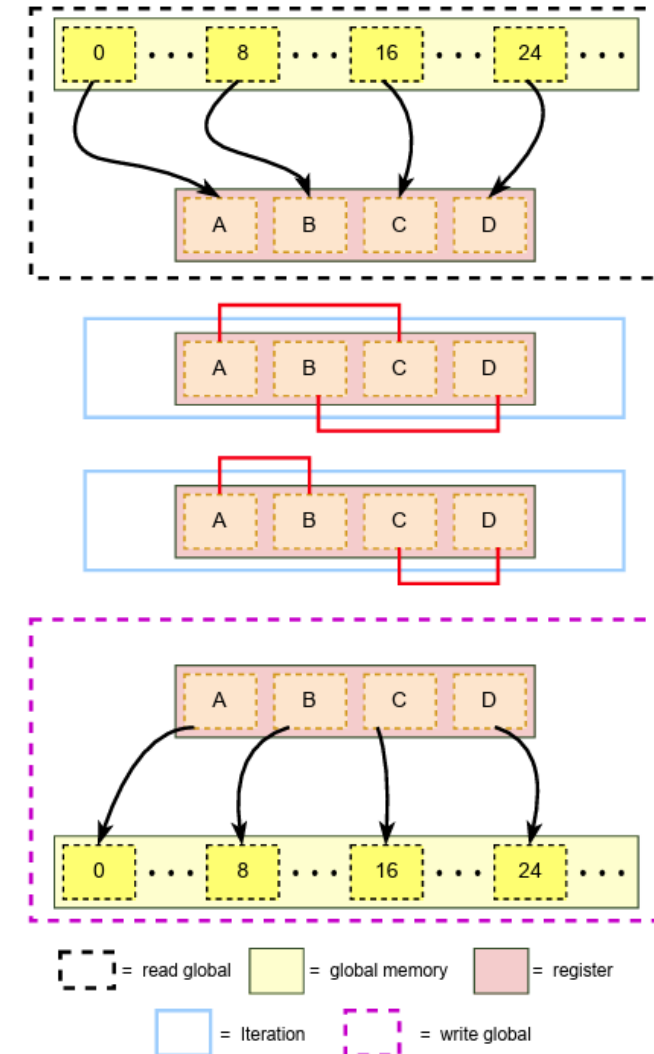
FHE on a GPU

- **Butterfly gap:** In iteration 1, pairs are 8 apart (e.g., (0,8), (1,9))
- **Thread stride:** Threads load data every 8 elements to match pair spacing
- **Speed:** This enables fast, register-based computation with minimal global memory use

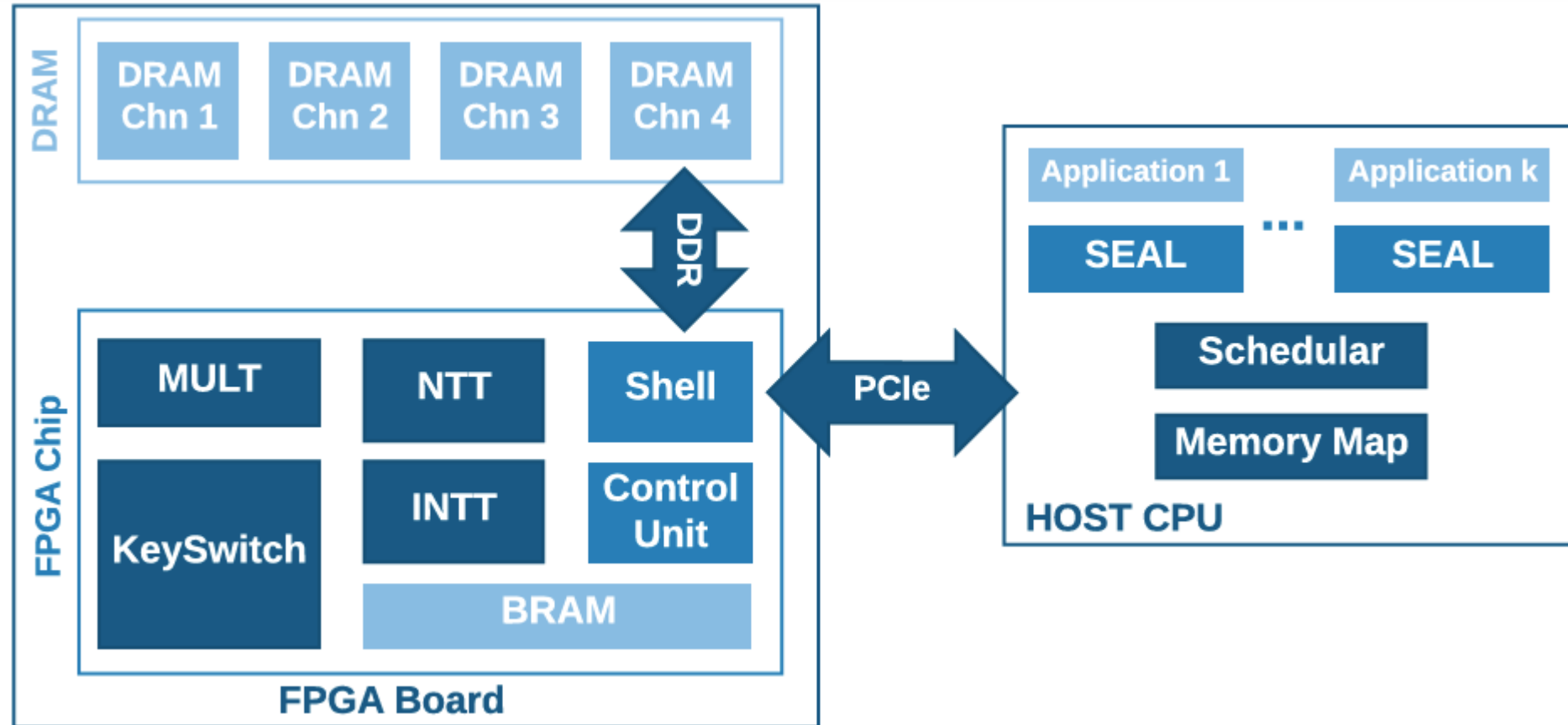
Gradient Boosting (XGBoost) Results

n	$\log_2 q$	SEAL		T.W		
		S.T.	M.T.	RTX 3060Ti	T	S
2^{13}	218	25.62 s	3.4 s	0.596 s	$42.98\times$	$5.7\times$
2^{14}	438	127.028 s	19.27 s	2.41 s	$52.7\times$	$8\times$

S.T.:Single Thread **M.T.:**Multi Thread.(16 threads) **T.W.:**This Work. T :The ratio of single-thread results over this work. S : The ratio of multi-thread results over this work.

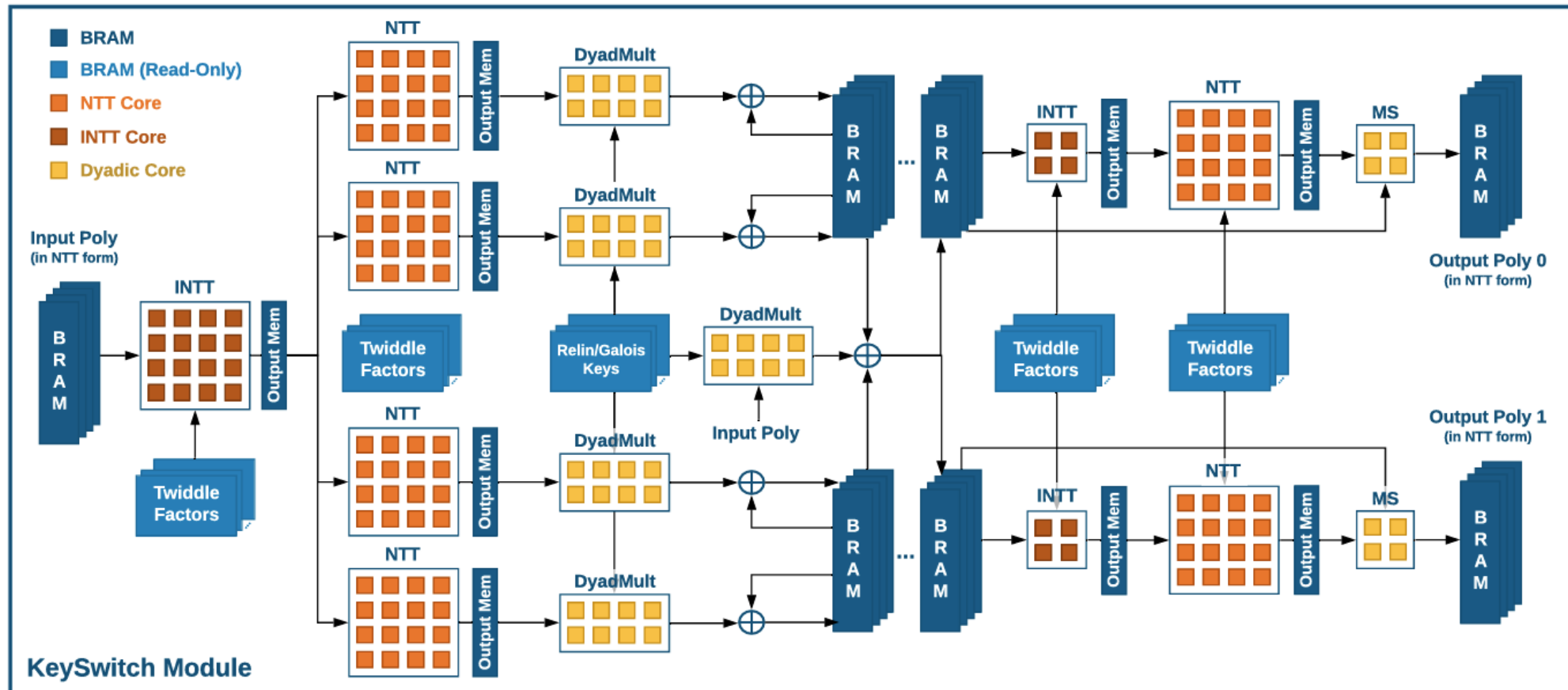


HEAX FGPA-Based FHE Accelerator



- Multi-card system with integrated HB DRAM memory
- Deployed on FGPA cards

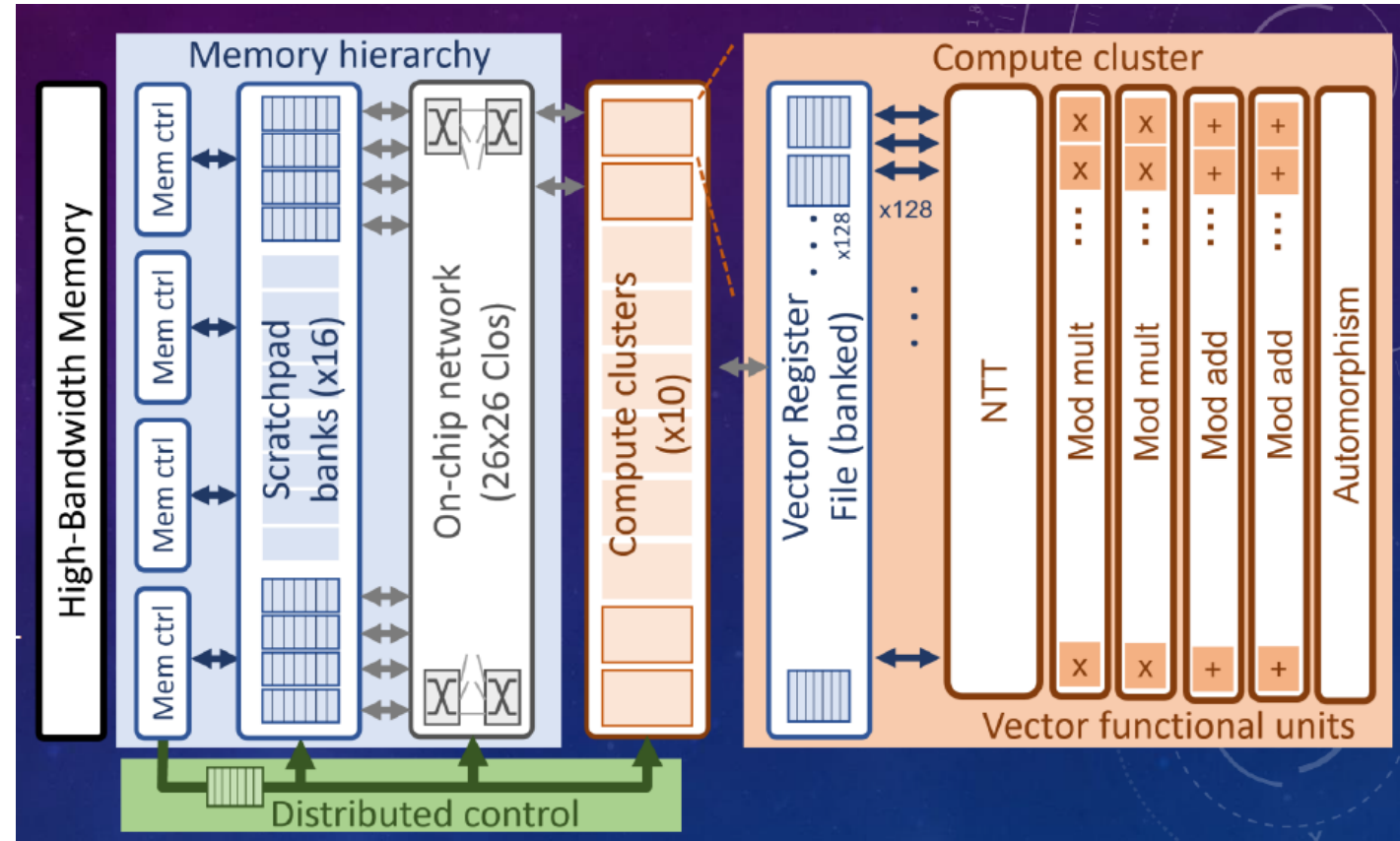
HEAX's Keyswitching Architecture



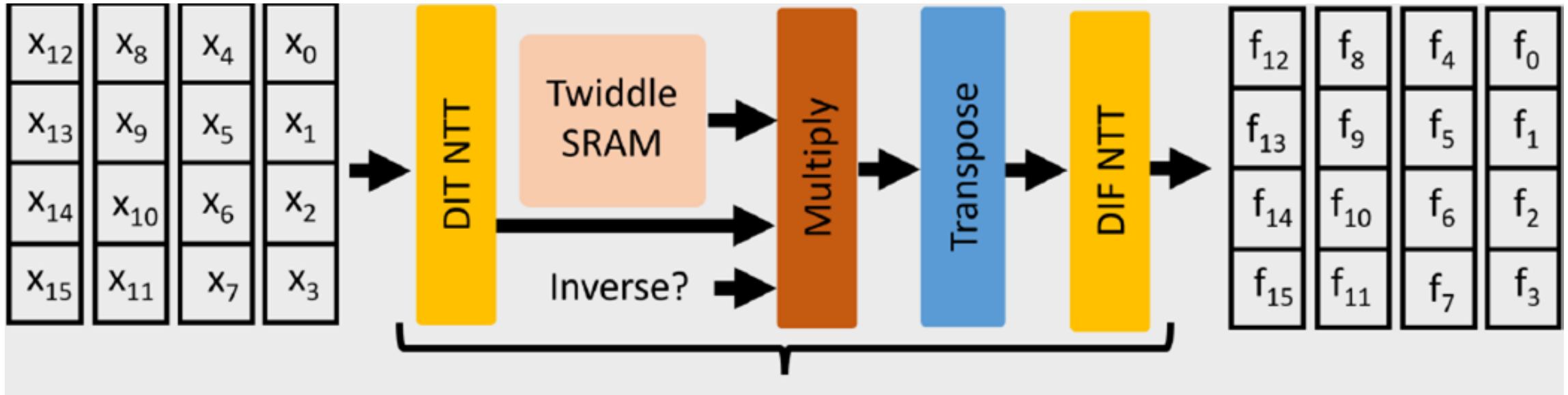
- Used for multiplies and rotates, heaviest op on the accelerator
- Heavily pipelined for high throughput

F1 HE Accelerator Architecture

- Small Design: 151 mm^2
- Large scratch pad: 64MB
 - Note, for $N=16K$, $L=16$, a ct is 2MB
- Compute clusters with FUs for NTT, mod mult, mod add, and automorphism
- Each FU operates on an RNS polynomial, or a vector of 24-bit values



F1 NTT Architecture



- 4-step NTT processes 16 inputs
- Twiddle SRAM holds roots of unit (ω^n)
- Includes forward (DIT) and reverse (DIF) butterflies
- Transpose unit implements HE permutations

F1's Published Performance

Execution time (ms) on	CPU	F1	Speedup
LoLa-CIFAR Unencryp. Wghts.	1.2×10^6	241	5,011×
LoLa-MNIST Unencryp. Wghts.	2,960	0.17	17,412×
LoLa-MNIST Encryp. Wghts.	5,431	0.36	15,086×
Logistic Regression	8,300	1.15	7,217×
DB Lookup	29,300	4.36	6,722×
BGV Bootstrapping	4,390	2.40	1,830×
CKKS Bootstrapping	1,554	1.30	1,195×
gmean speedup			5,432×

*LoLa's release did not include MNIST with encrypted weights, so we reimplemented it in HELib.

Agenda for the Today's Lecture

- FHE Recap
- Core FHE Optimizations
- Accelerating FHE
- Comparative Performance Analysis

Comparative Studies

- Comparison of open-source designs
 - HEAX and F1 show previously published results
- 1Mx range overall (NTT)
 - GPU 100x over CPU
 - GPU 10x over FPGAs
 - F1 14nm ASIC 100x over GPU
 - F1 ASIC 30Kx over CPU
- Key-switching range larger

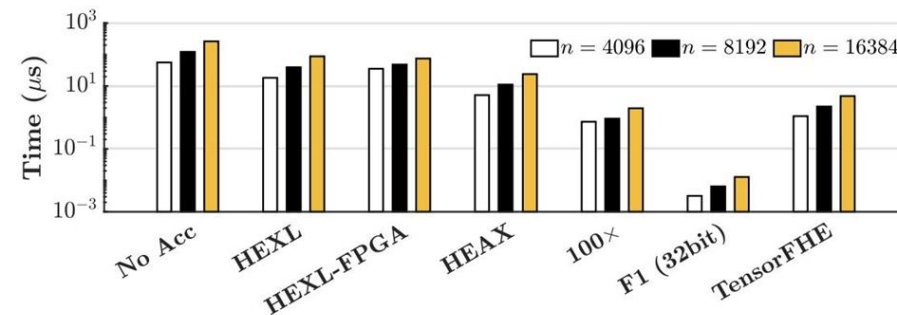


Fig. 5. Performance of NTT. The performance of HEXL, HEXL-FPGA, and 100x is measured on our testbed, whereas the performance results of HEAX and F1 are from their original works.

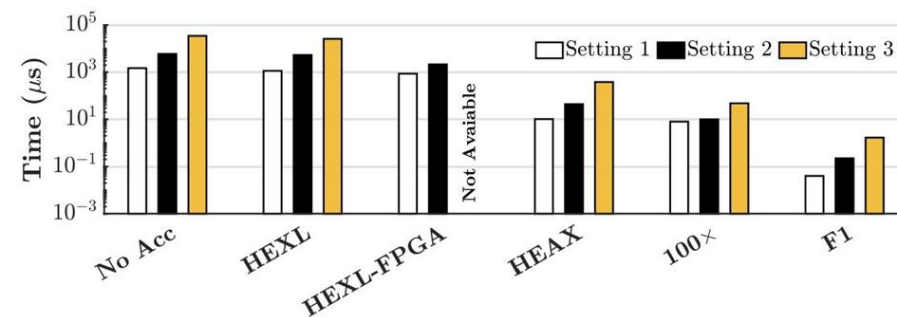


Fig. 6. Performance of key-switching. The performance of HEXL, HEXL-FPGA, and 100x is measured on our testbed, whereas the performance results of HEAX and F1 are from their original works. HEXL-FPGA does not support Setting 3.

End-to-end Paints a New Picture

- End-to-end performance less clear for targets
 - F1 slows down due to poor bootstrap performance
 - BTS and ARK focus on bootstrap performance
- Note Resnet20 performance
 - ~10ms on CPU, ~0.3ms on GPU
 - 300ms to 5sec on FHE accelerator
 - 30-500x slower than CPU native
 - 1000-16,667x slower than GPU native
 - Orion is ~1Mx slower than GPU native

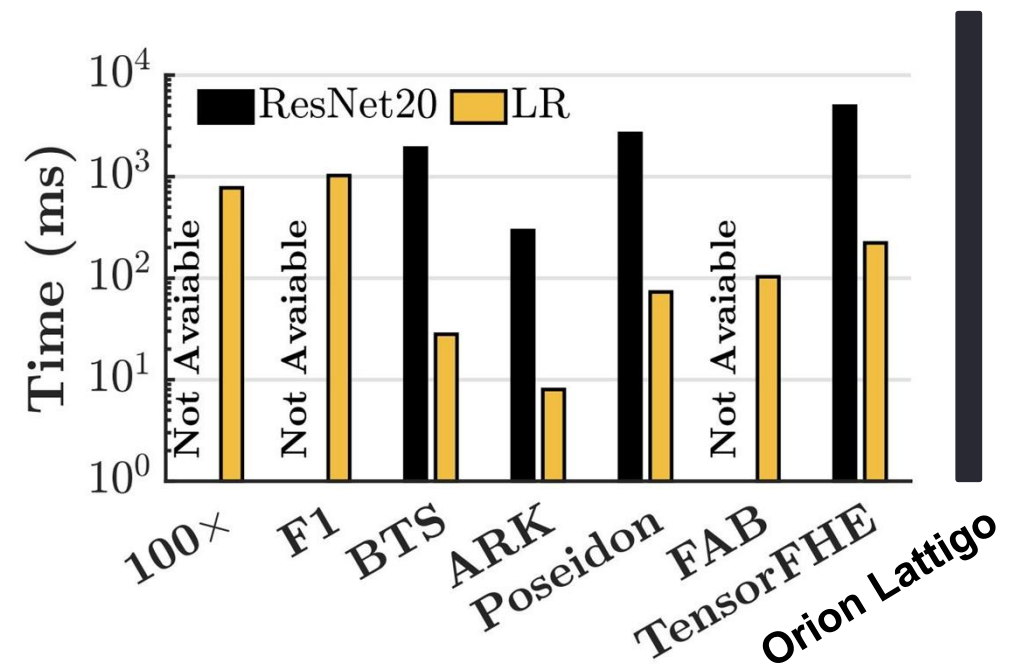


Fig. 8. Performance of end-to-end applications.

Boolean FHE Performance Lags

- Per-gate performance is very slow relative to integer
 - 60ms - 150ms per gate
- Benchmarks:
 - 32-bit equality test - 95 gates
 - 5.7s - 14.3s to evaluate
 - 32-bit ripple adder - 160 gates
 - 9.6s - 24s to evaluate

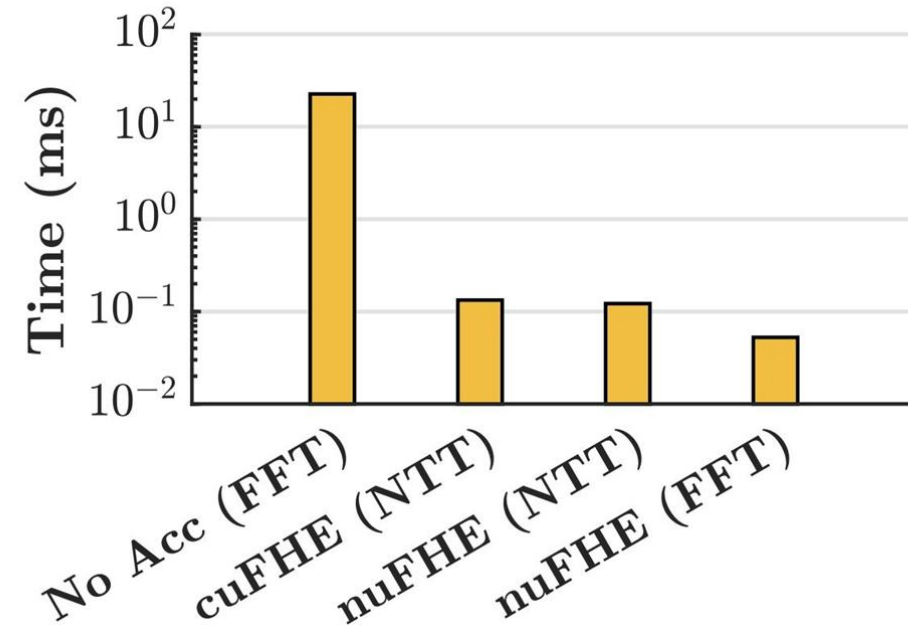


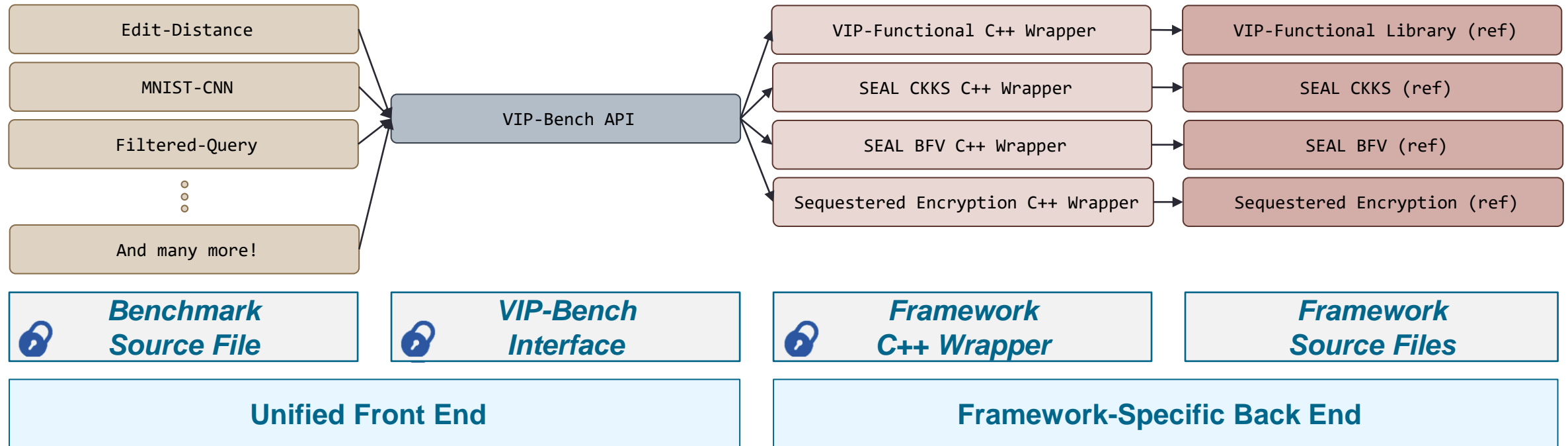
Fig. 7. Performance of the NAND gate with NTT/FFT.

VIP-Bench A Privacy Benchmark Suite

- **Unified workload** source files that map to different privacy frameworks at compile time
- **Flexible interfaces** that enable porting of new technologies to the benchmark suite
- **30+ workloads** built with varying complexity

VIP-Bench Workloads				
	Operator Classes			
	ARITH	REL	BOOL	CMOV
Hamming-Distance	●			
Dot-Product	●			
Linear Regression	●			
Mersenne	●		●	
Nonlinear-NN	●	●		●
Edit-Distance	●	●		●
Kepler	●	●		●
MNIST-CNN	●	●	●	●
<i>And more...</i>				

VIP-Bench A Privacy Benchmark Suite



 = Included in VIP-Bench

VIP-Bench A Privacy Benchmark Suite

- Enables **commensurate comparisons** to native execution and across different privacy technologies
- Facilitates comparing frameworks and identifying performance bottlenecks

VIP-Bench Comparisons

