Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs

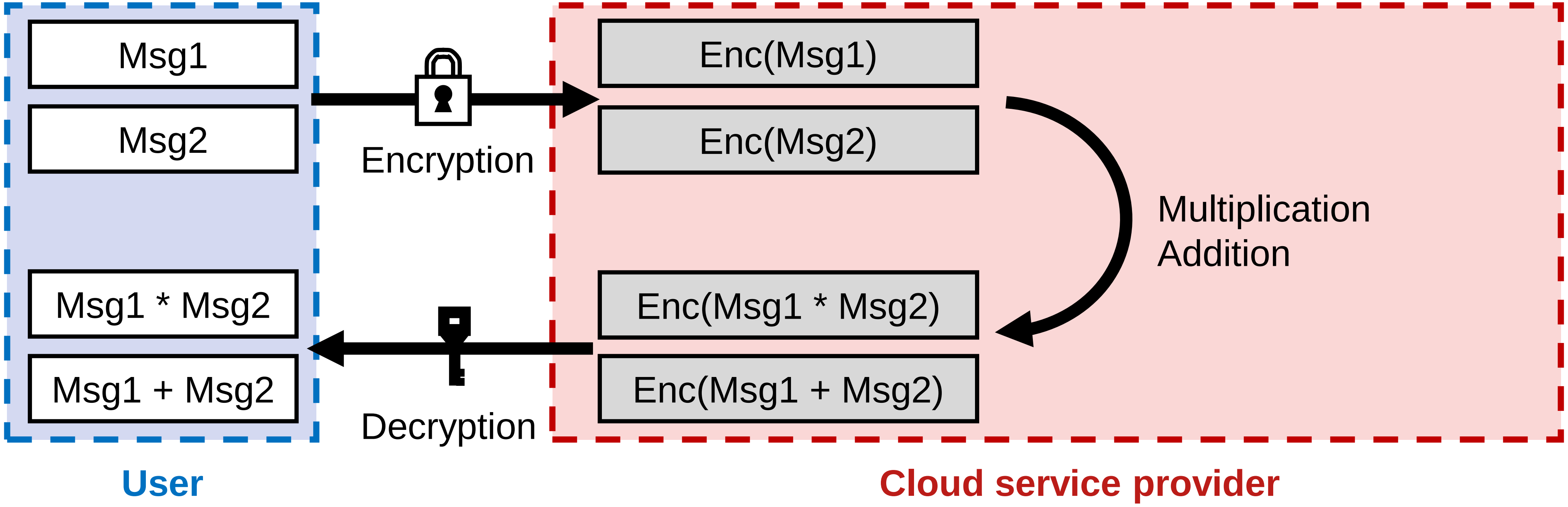
Younho Lee
(Dept. Data Science, SeoulTech)

Thanks to Dr. Jung

Contents

Preliminary: Homomorphic encryption (HE) costs a lot

- HE is a cryptographic scheme that enables computation on encrypted messages (*ciphertexts*).
- Users send their data to a cloud service provider without privacy worries.
- Popular HE schemes
  - BFV (integers), TFHE (binary numbers), **CKKS** (fixed-point numbers)



Preliminary: Homomorphic encryption (HE) costs a lot

- Ciphertexts are large.
  - One ciphertext in CKKS = **10-100s of MBs**
- Slow HE operations.
  - (vs. native operations) 100x **–** 200x in addition
  - (vs. native operations) **1000x – 10000x** in multiplication
- *Bootstrapping* is even heavier.
  - Bootstrapping reduces the noise after HE operations on a ciphertext.
    - = It enables *Fully Homomorphic Encryption* (FHE).
  - It takes **10s of seconds** for a single bootstrapping in CKKS.
    - = (using a single-core of a latest CPU)

Preliminary: Homomorphic encryption (HE) costs a lot

- Each encrypted message is called ciphertext.
- Each ciphertext is represented as a pair of high-degree polynomial in a polynomial ring,

$$\textbf{ax1} = a_0 + a_1 x + a_2 x^2 + \cdots + a_{65535} x^{65535}$$

Ctxt1 = (**ax1**, **bx1**)

Ctxt2 = (**ax2**, **bx2**)

Ctxt3 = (**ax3**, **bx3**)

Multiplication

Preliminary: Homomorphic encryption (HE) costs a lot

- Each encrypted message is called ciphertext.
- Each ciphertext is represented as a pair of high-degree polynomial in a polynomial ring,
  - ...whose coefficients are big-integers.

1200-bit

ax1 = $a_0 + a_1 x + a_2 x^2 + \cdots + a_{65535} x^{65535} \in Z_q[X]/(X^N + 1)$

$q = 2^{1200}, N = 65536$

Ctxt1 = (**ax1**, **bx1**)

Ctxt2 = (**ax2**, **bx2**)

Multiplication

Ctxt3 = (**ax3**, **bx3**)

Multiplication between such polynomials is expensive.

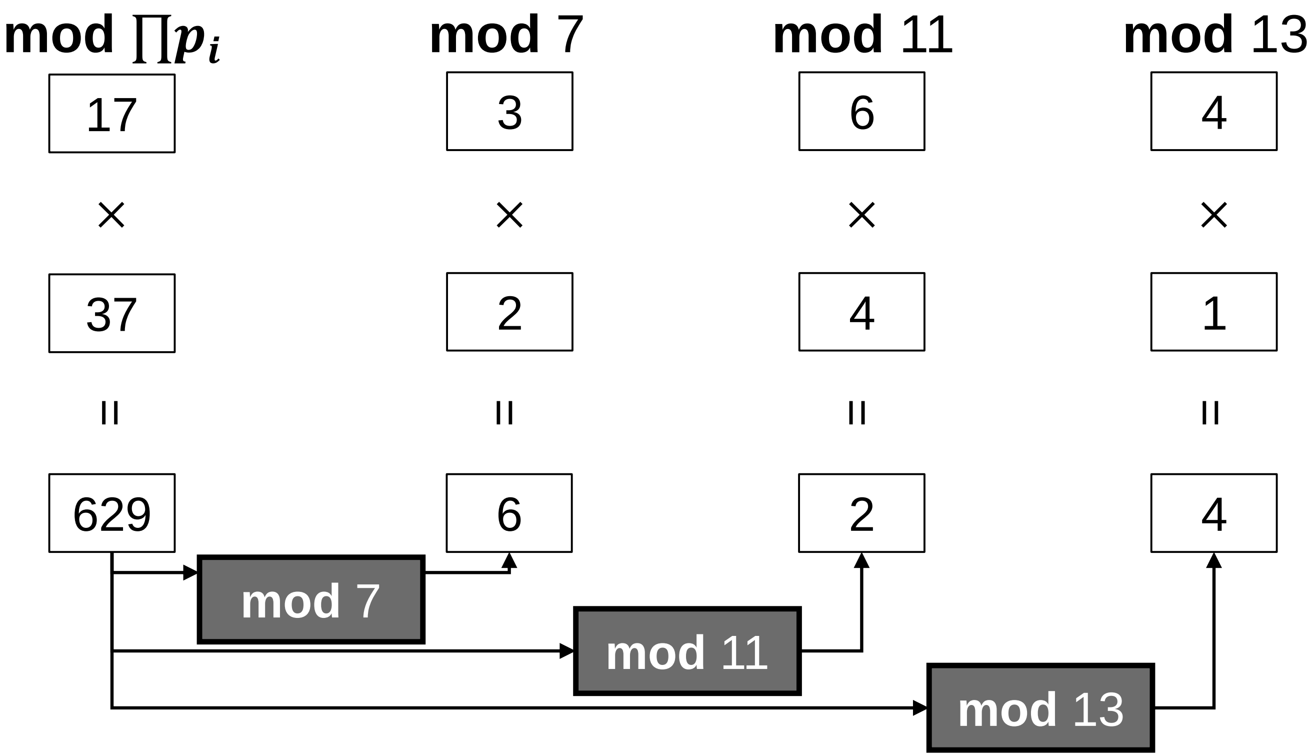
Preliminary: Massive parallelism in HE

- Fortunately, massive parallelism exists in polynomial multiplication:
  - Chinese Remainder Theorem (CRT) to exploit Residue Number System (RNS)
  - Number Theoretic Transform (NTT)

Preliminary: Chinese Remainder Theorem (CRT) in HE

- CRT reduces the computational complexity of multiplication of two big numbers.
  - Ex) multiplying 17 and 37 using coprimes $(p_1, p_2, p_3) = (7,11,13)$

(CRT moduli)



Preliminary: Chinese Remainder Theorem (CRT) in HE

- CRT reduces the computational complexity of multiplication of two big numbers.

  Ex) multiplying 17 and 37 using coprimes $(p_1, p_2, p_3) = (7,11,13)$

(CRT moduli)

mod $\prod p_i$	**mod** 7	**mod** 11	**mod** 13
17	3	6	4
$\times$	$\times$	$\times$	$\times$
37	2	4	1
$=$	$=$	$=$	$=$
$\boxed{629}$	6	2	4

Reconstruct (inverse CRT, iCRT)

Preliminary: Number Theoretic Transformation (NTT) in HE

- Each multiplication between big-integer coefficients can be solved by CRT, but still with O(N^2).

ax1 = $a_0 + a_1x + a_2x^2 + \cdots + a_{65535}x^{65535} \in Z_q[X]/(X^N + 1)$

$q = 2^{1200}, N = 65536$

Ctxt1 = (**ax1**, **bx1**)

Ctxt2 = (**ax2**, **bx2**)

CRT

$(a_0, a_1, \dots, a_{65536})\ mod\ p_1 \quad * \quad (a'_0, a'_1, \dots, a'_{65536})\ mod\ p_1$

$(a_0, a_1, \dots, a_{65536})\ mod\ p_2 \quad * \quad (a'_0, a'_1, \dots, a'_{65536})\ mod\ p_2$

$\dots \qquad \dots$

$(a_0, a_1, \dots, a_{65536})\ mod\ p_{np} \quad * \quad (a'_0, a'_1, \dots, a'_{65536})\ mod\ p_{np}$

Preliminary: Number Theoretic Transformation (NTT) in HE

- Each multiplication between big-integer coefficients can be solved by CRT, but still with $O(N^2)$.
- NTT, an integer-variant DFT, reduces $O(N^2)$ to $O(N\log N)$.

Point-wise mul, $\boldsymbol{O(N)}$

$$(A_0, A_1, \dots, A_{65536}) \quad \odot \quad (A'_0, A'_1, \dots, A'_{65536})$$

NTT, $\boldsymbol{O(NlogN)}$

$$(a_0, a_1, \dots, a_{65536})\ mod\ p_1 \quad * \quad (a'_0, a'_1, \dots, a'_{65536})\ mod\ p_1$$
$$(a_0, a_1, \dots, a_{65536})\ mod\ p_2 \quad * \quad (a'_0, a'_1, \dots, a'_{65536})\ mod\ p_2$$
$$\dots \qquad \dots$$
$$(a_0, a_1, \dots, a_{65536})\ mod\ p_{np} \quad * \quad (a'_0, a'_1, \dots, a'_{65536})\ mod\ p_{np}$$

Preliminary: Variants of Cheon-Kim-Kim-Song (CKKS)

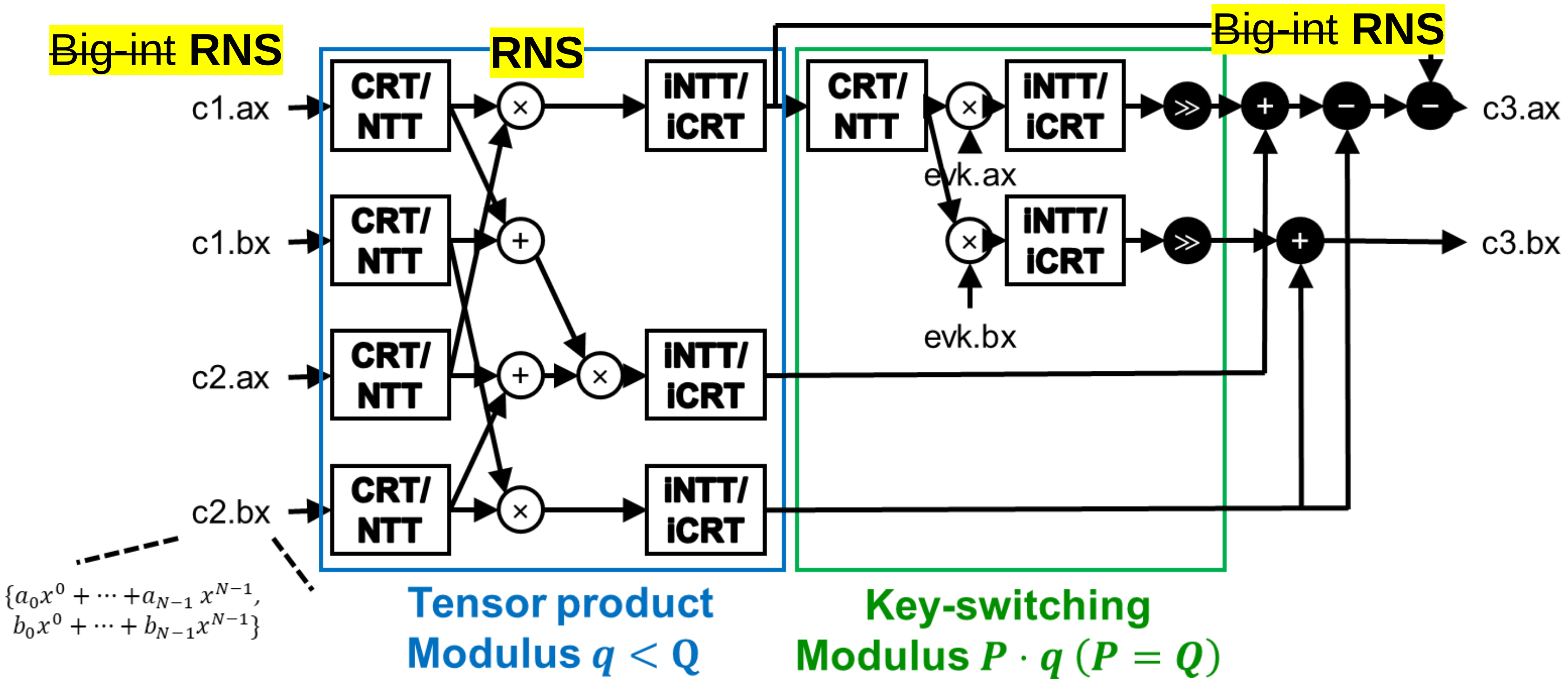
- Representation of plaintext and ciphertext in RLWE:
  - Plaintext $m(X) \in R_q = \mathbb{Z}_q[X]/(X^N + 1),\ q \leq Q(= ciphertext\ modulus\ )$
  - Ciphertext $ct(X) = (b(X), a(X)) \in R_q^2$
- Variants of CKKS
  - Binary CKKS [1]
    - = arbitrary q and Q
    - = needs expensive CRT & iCRT conversion.
  - RNS-CKKS [2]
    - = Fixed q and Q
    - = No (i)CRT; exploits *Fast base conversion*.
  - (Improved) RNS-CKKS [3]
    - = [2] + better algorithms (*key-switching* & bootstrapping)

[1] Cheon at el., Homomorphic encryption for arithmetic of approximate numbers, AsiaCrypt'17
[2] Cheon at el., A full RNS variant of approximate homomorphic encryption, SAC'18
[3] Han at el., Better bootstrapping for approximate homomorphic encryption, CT-RSA'20

Contents

Why Full-RNS CKKS?

- Polynomials are always in RNS domain.
  - no big-int computations



Multiplication of two ciphertexts (c1 * c2)

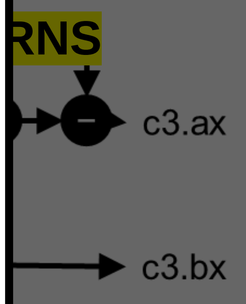
Why Full-RNS CKKS?

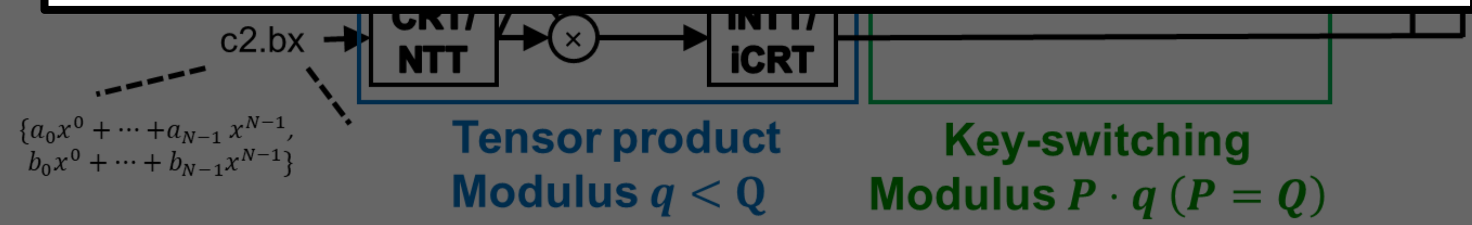
- Polynomials are always in RNS domain.
  - no big-int computations (iCRT+CRT ➔ *Fast base conversion*)

$$iCRT_{\{[x]_{p_i}\}\rightarrow BigInt} \quad \sum \left[r_i \cdot \left(\frac{\prod_j p_j}{p_i}\right)^{-1} \right]_{p_i} \cdot \left(\frac{\prod_j p_j}{p_i}\right) \bmod \prod_j p_J$$

$$Conv_{\{[x]_{p_i}\}\rightarrow\{[x]_{p'_k}\}} \quad \sum \left[r_i \cdot \left(\frac{\prod_j p_j}{p_i}\right)^{-1} \right]_{p_i} \cdot \left[\left(\frac{\prod_j p_j}{p_i}\right)\right]_{p'_k} \bmod p'_k$$

Similar to iCRT, but word-size
precomputed values and modulus





Multiplication of two ciphertexts (c1 * c2)

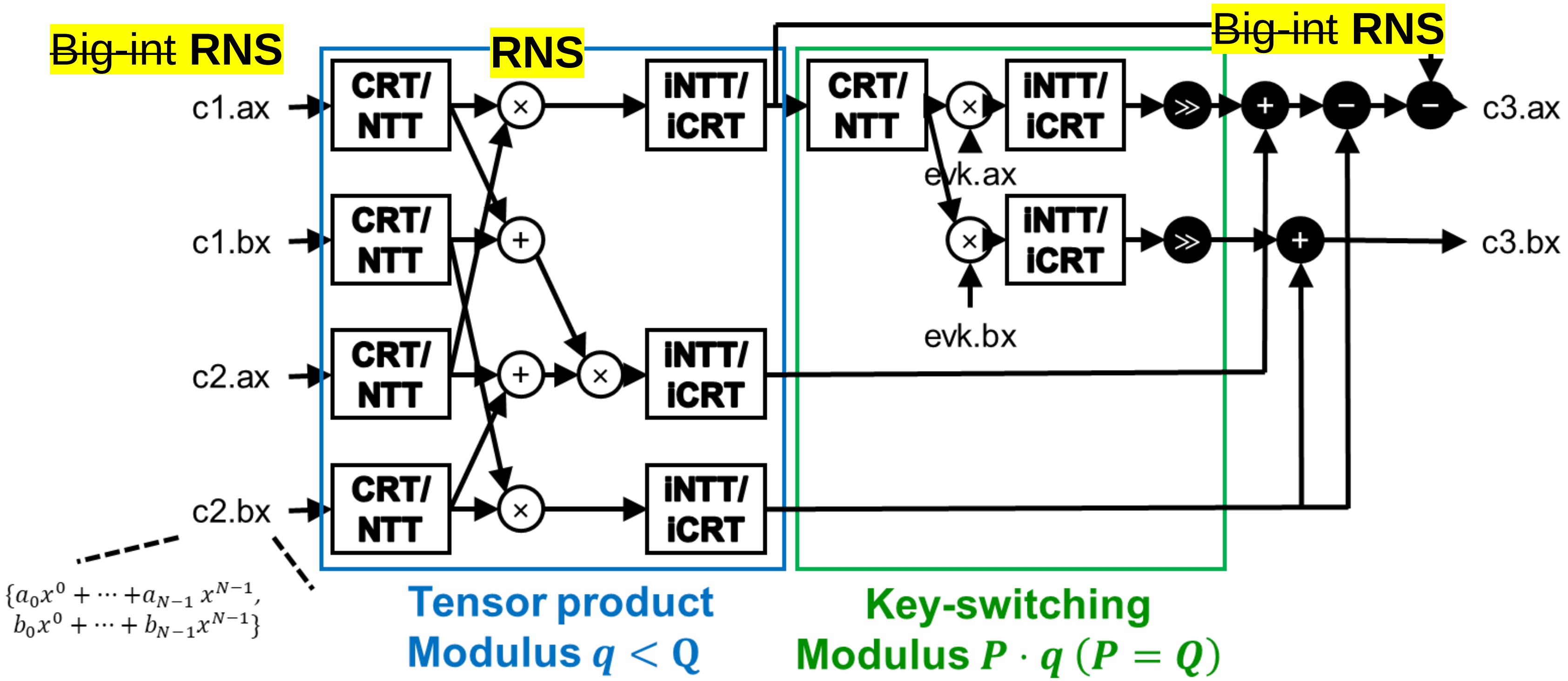
Why Full-RNS CKKS?

- Polynomials are always in RNS domain.
  - no big-int computations (iCRT+CRT ➔ *Fast base conversion*)
- Small *CRT modulus:* ciphertext modulus is a product of moduli ($Q = q_0 q_1 \ldots q_L,\ q = q_0 q_1 \ldots q_\ell$)

(In binary CKKS)
CRT moduli: q^2 and qQP

➔

(In Full-RNS)
q and $P \cdot q$



Multiplication of two ciphertexts (c1 * c2)

Accelerating Full-RNS CKKS

- The first GPU implementation of bootstrapping in the latest Full-RNS CKKS scheme.
- Analyze memory bandwidth bottleneck of the GPU implementation.
- Apply memory-centric optimizations, resulting in sub-second bootstrapping.

  = a speedup of 257x, compared to single-threaded CPU.

Background: a recent Full-RNS CKKS [HK20]

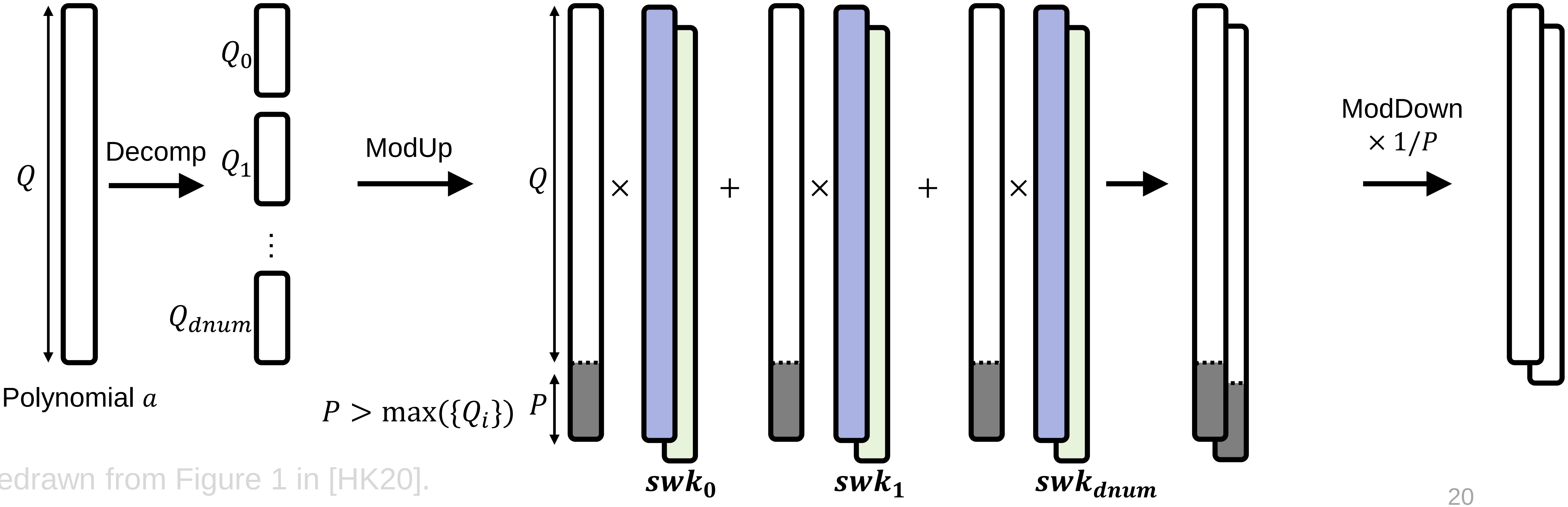
- Parameters
  - Multiplicative level L
  - Primes $q_0, \ldots, q_L, p_1, \ldots, p_k$
  - Modulus $Q = q_0 q_1 \ldots q_L$, $P = p_1 p_2 \ldots p_k$
- Plaintext representation:
  - (RNS form) $m(X) \in R_Q$
  - (NTT form) $m = NTT\big(m(X)\big)$
- Ciphertext representation:
  - (RNS form) $\boldsymbol{ct} = \big(b(X), a(X)\big) \in R_Q^2$
  - (NTT form) $\boldsymbol{ct} = (b, a)$

Background: a recent Full-RNS CKKS [HK20]

- Key operations
  - Plaintext multiplication
    - = $CMult(\boldsymbol{ct}, m) = (ma, mb)$
  - Ciphertext multiplication
    - = $Mult(\boldsymbol{ct_1}, \boldsymbol{ct_2})$
    1. computes a tensor product $(d_0, d_1, d_2) = (b_2 b_1, b_1 a_2 + b_2 a_1, a_1 a_2)$
    2. return $(d_0, d_1) + KeySwitch_{\boldsymbol{MultKey}}(d_2)$
  - Ciphertext rotation by a rotation index r
    - = $Rotate(\boldsymbol{ct}, r)$
    1. computes an automorphism $b'(X) = b\left(X^{5^r}\right), a'(X) = a\left(X^{5^r}\right)$
    2. return $(b', 0) + KeySwitch_{\boldsymbol{RotationKey_r}}(a')$
- [HK20] introduces a new key-switching method for CKKS.
  - Generalized key switching (a.k.a. hybrid key-switching)

Background: a recent Full-RNS CKKS [HK20]

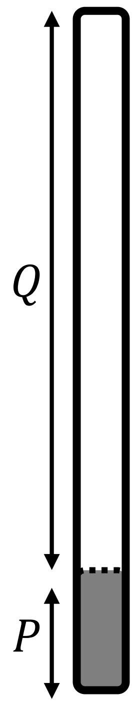
- $KeySwitch_{\boldsymbol{swk}}(a)$
  1. (*Decomp)* decompose into up to ***dnum*** parts, each having α moduli: $Q_0 = q_0 q_1 \cdots q_{\alpha-1}, Q_1 = q_\alpha q_{\alpha+1} \cdots$
  2. (*ModUp)* raise the modulus of each part to PQ.
  3. (*Inner-product)* do inner-product with the key-switching key $\boldsymbol{swk} = (\boldsymbol{swk_0}, \boldsymbol{swk_1}, \dots, \boldsymbol{swk_{dnum}})$.
  4. (*ModDown)* reduce the modulus to Q and multiply $1/P$.



Redrawn from Figure 1 in [HK20].

Background: Choosing a good *dnum* parameter

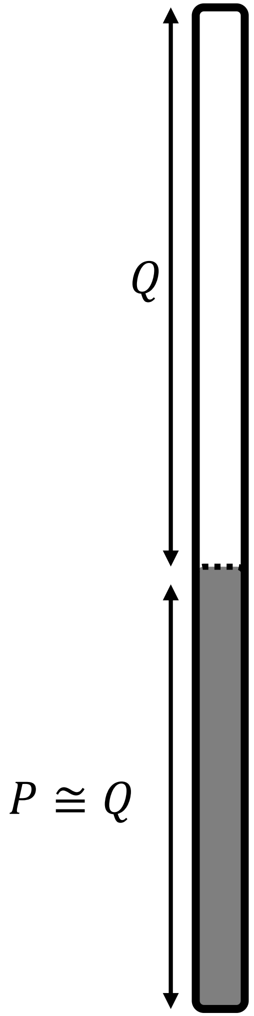
- $\uparrow logPQ$ = $\downarrow$ security
- The choice of **dnum** affects complexity, key size, and security.



(after ModUp)

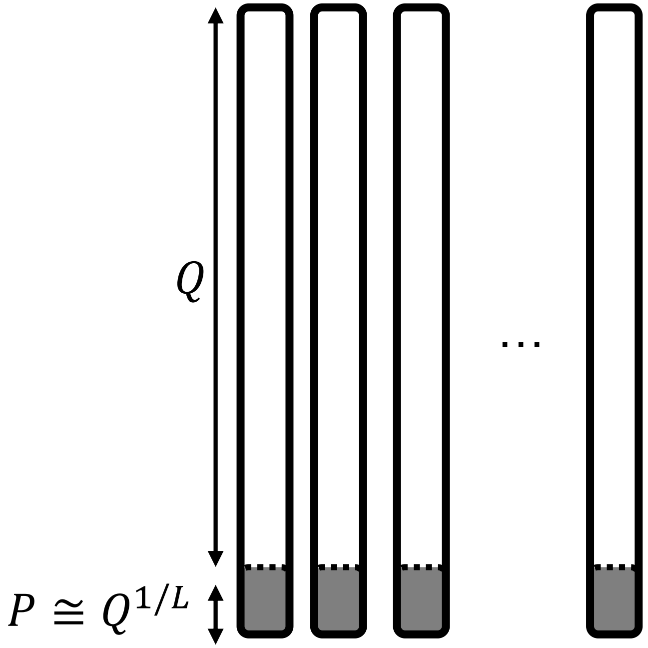
Background: Choosing a good *dnum* parameter

- $\uparrow logPQ$ = $\downarrow$ security
- The choice of **dnum** affects complexity, key size, and security.
- Given Q,
  - **dnum** = 1 (minimum)
    - $\uparrow P$
    - $\downarrow$ security
    - $\downarrow$ size of keys



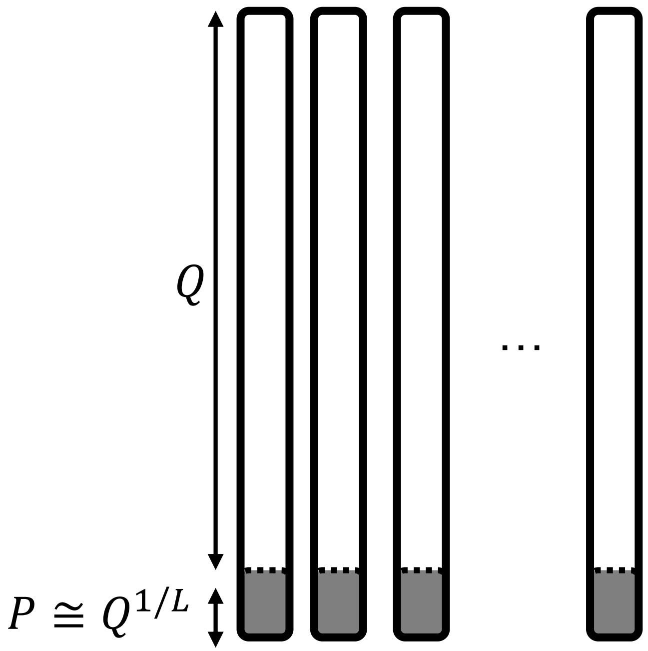
Background: Choosing a good *dnum* parameter

- ↑ $logPQ$ = ↓ security
- The choice of **dnum** affects complexity, key size, and security.
- Given Q,
  - **dnum** = 1 (minimum)
    - ↑ P
    - ↓ security
    - ↓ size of keys
  - **dnum** = L (maximum)
    - ↓ P
    - ↑ security
    - ↑ size of keys



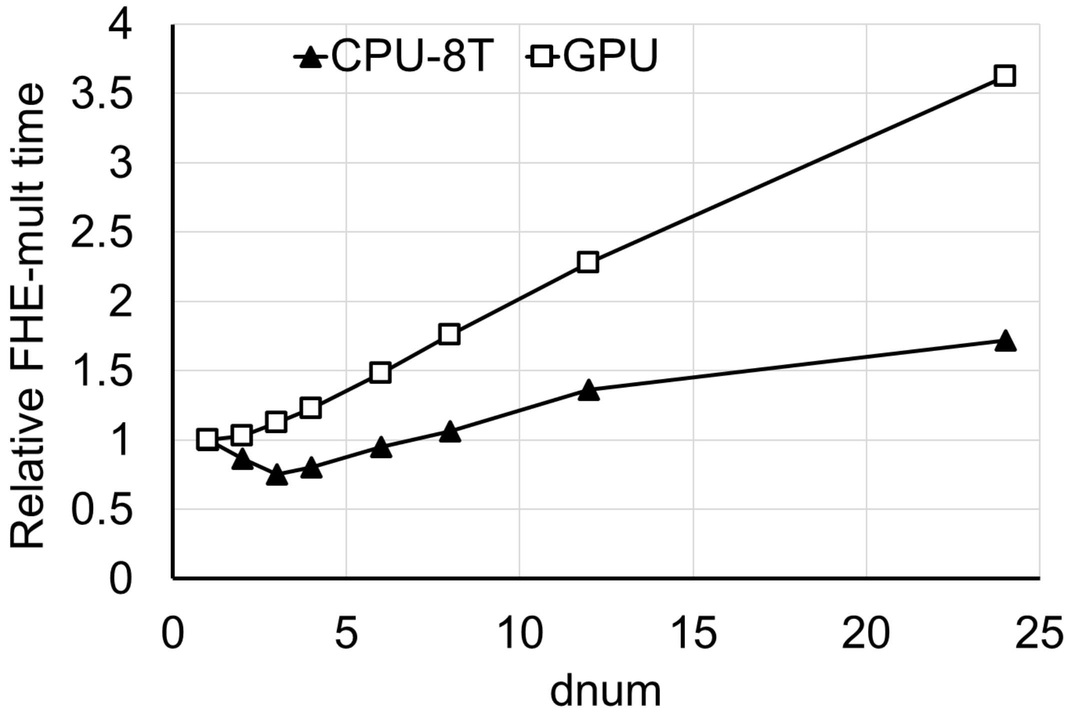
Background: Choosing a good *dnum* parameter

- $\uparrow logPQ$ = $\downarrow$ security
- The choice of **dnum** affects complexity, key size, and security.
- Given Q,
  - **dnum** = 1 (minimum)
    - $\uparrow P$
    - $\downarrow$ security
    - $\downarrow$ size of keys
  - **dnum** = L (maximum)
    - $\downarrow P$
    - $\uparrow$ security
    - $\uparrow$ size of keys
  - Computational complexity becomes minimal somewhere between them.
    - = [HK20] chooses that value.



Motivation: GPU implementations are more memory-bottlenecked

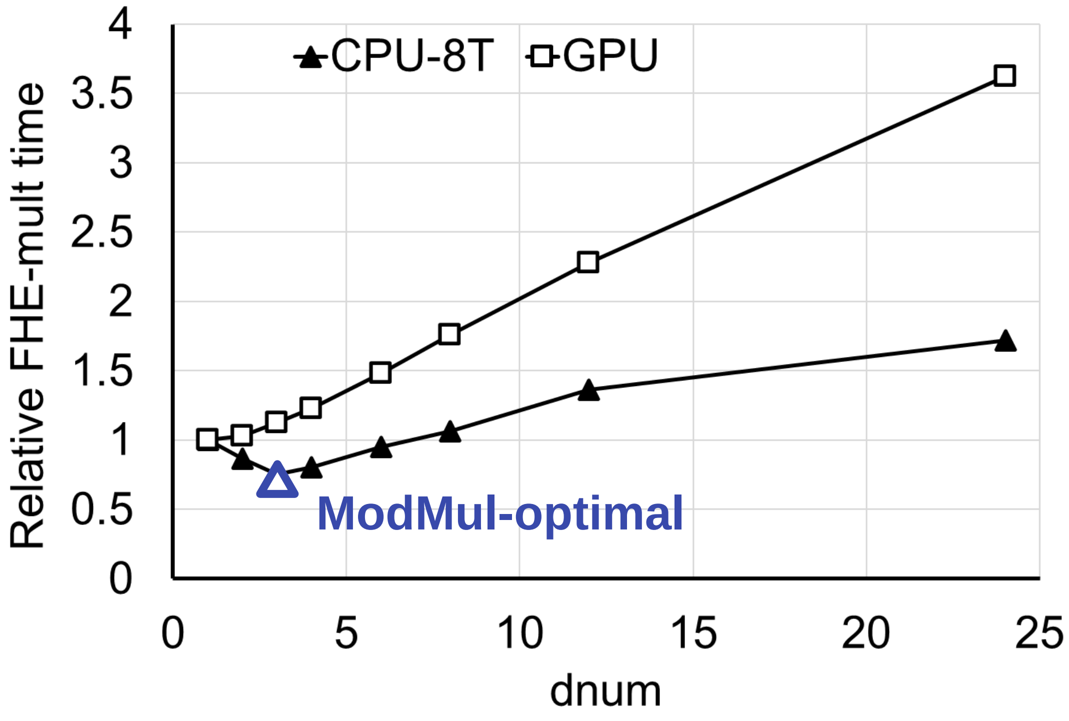
- Last-level cache size of a server class
  - CPU: ~256MB
  - GPU: ~6MB
    - = Hardly accommodates ciphertexts.



dnum vs. multiplication time for a fixed Q

Motivation : GPU implementations are more memory-bottlenecked

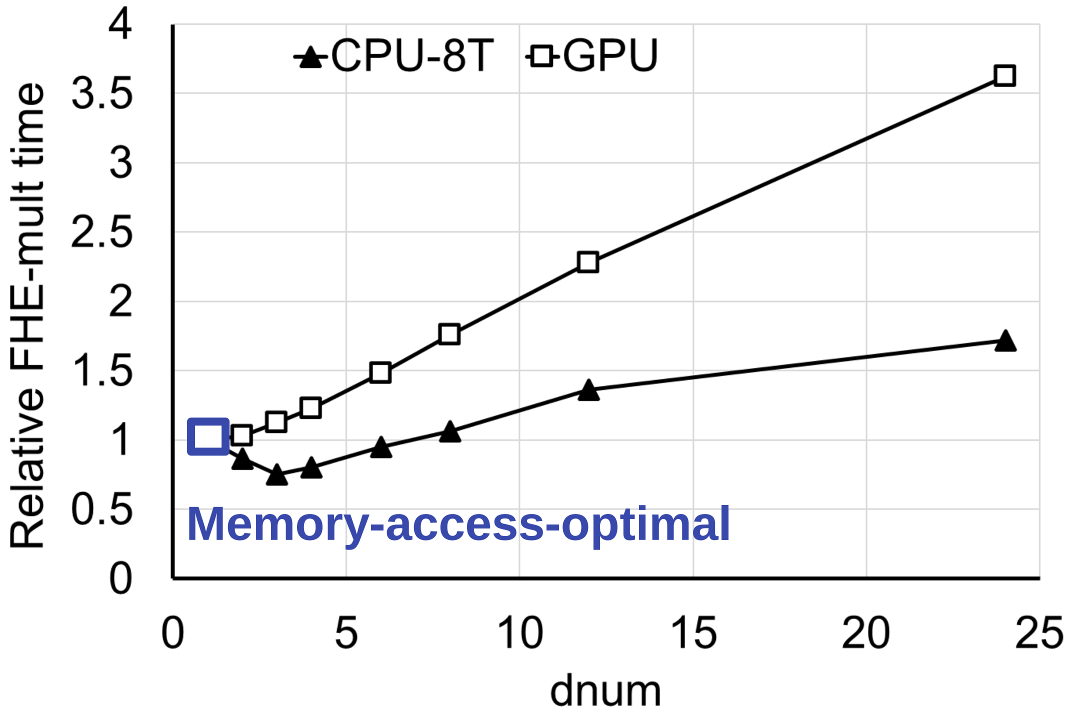
- Last-level cache size of a server class
  - CPU: ~256MB
  - GPU: ~6MB
    - = Hardly accommodates ciphertexts.



dnum vs. multiplication time for a fixed Q

Motivation : GPU implementations are more memory-bottlenecked

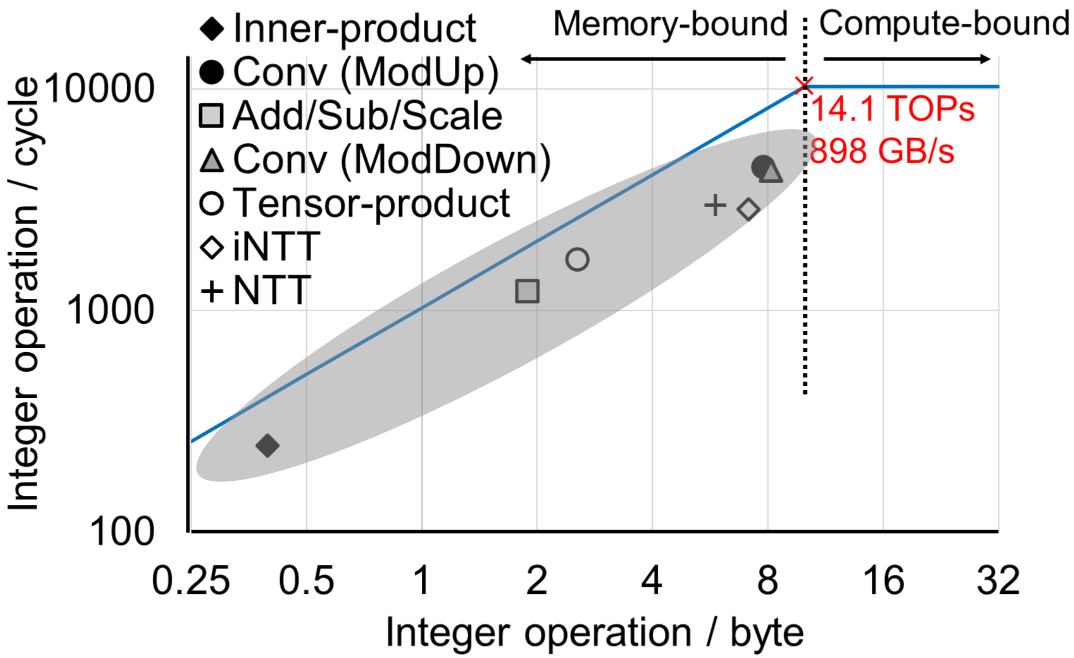
- Last-level cache size of a server class
  - CPU: ~256MB
  - GPU: ~6MB
    - = Hardly accommodates ciphertexts.



dnum vs. multiplication time for a fixed Q

Motivation: GPU implementations are more memory-bottlenecked

- Last-level cache size of a server class
  - CPU: ~256MB
  - GPU: ~6MB
    - = Hardly accommodates ciphertexts.



Roofline plot of a multiplication

HE operations are memory-bound, especially on GPUs.

Contents

Overview: Brief introduction to the programming model in GPUs

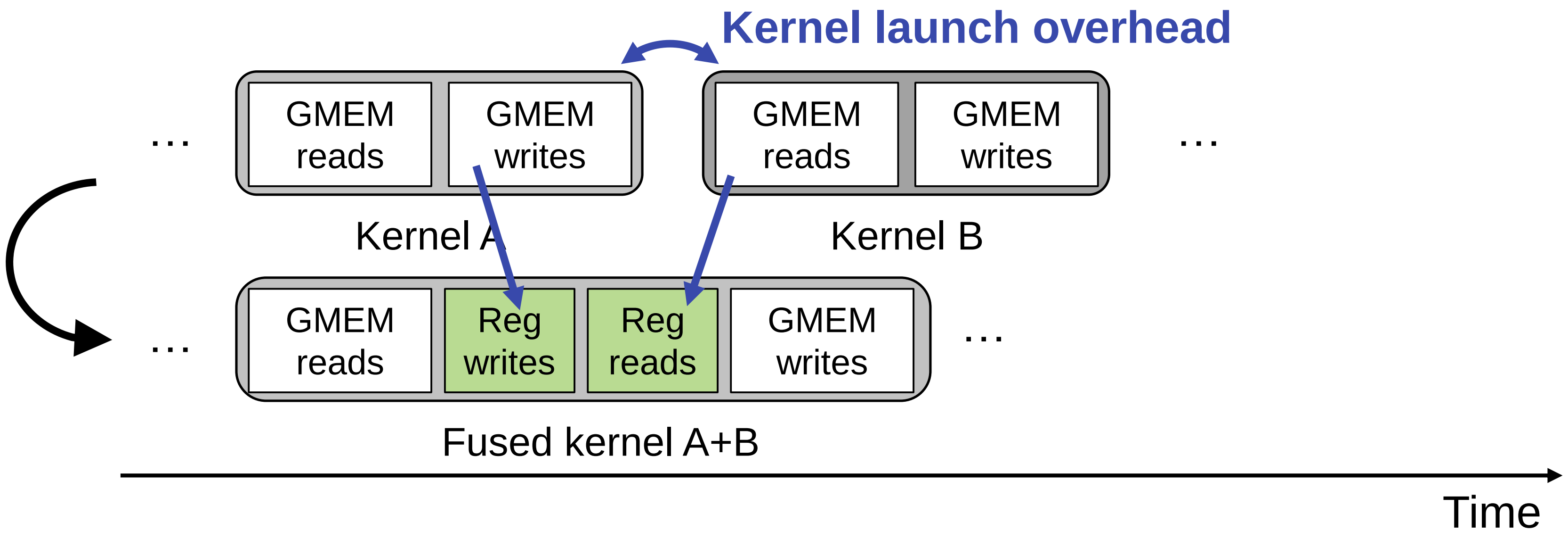
- A GPU has many *Streaming Multiprocessors (SM).*
  - Each SM manages and executes threads in groups of 32 parallel threads.
- A function executed in a GPU is called *Kernel.*
  - Many GPU threads run the same kernel in parallel in SMs.
- Each kernel is configured with
  - the number of threads running on the kernel
  - the amount of the *shared memory* it will use.
- Shared memory is a user-configurable scratch-pad memory.

Overview: Baseline GPU implementation

- Based on prior works:
  - NTT-friendly degree-first data layout [CLP17, HS14, CHK+19]
  - Executing $(L \times N)$ RNS operations in a kernel [BPA+19]
  - Fast NTT/iNTT implementations [KJPA20]
  - …

Overview: Kernel fusion - memory-centric optimization for GPUs

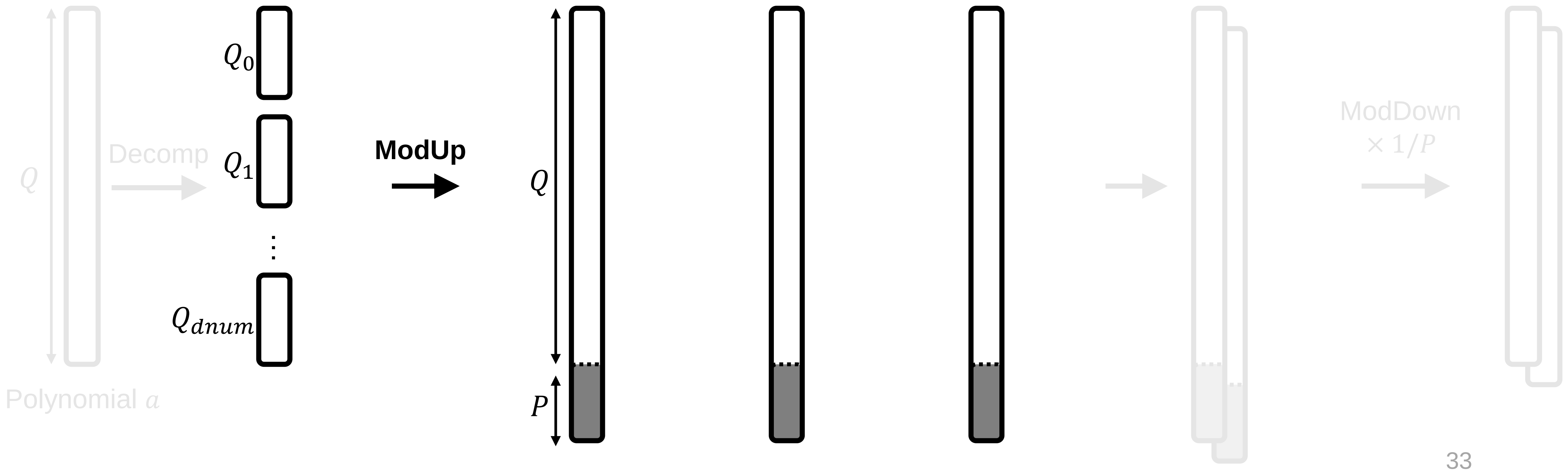
- Kernel fusion (operation fusion)
  – A common technique that fuses multiple GPU kernels into a single kernel
- Kernel fusion can
  – save global memory (DRAM) reads and writes by reusing the data in register file/shared memory.
    = Good for the kernels with low OP/B (memory-bound).
  – reduce *kernel launch overhead* by merging small kernels
- We found many kernel fusion opportunities in **intra- / inter-HE-operation** manner.



Kernel fusion reduces global memory (GMEM) accesses.

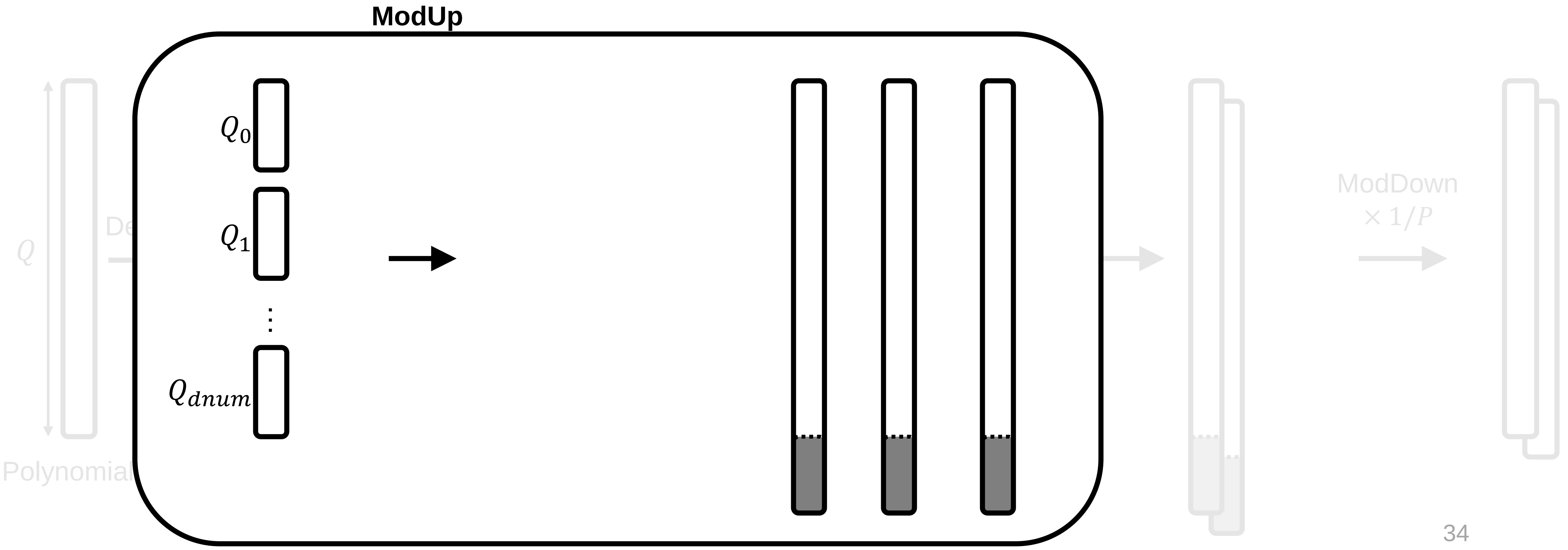
Intra-HE-operation Fusions

- *ModUp Fusion (MF)*



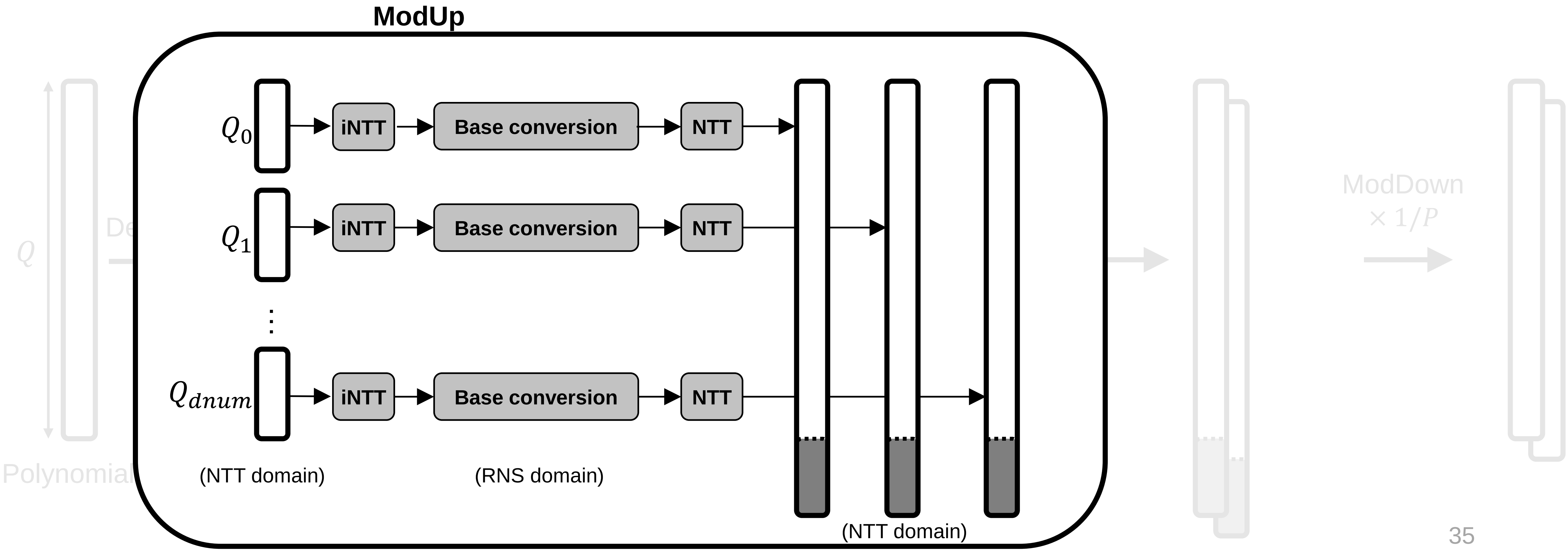
Intra-HE-operation Fusions

- *ModUp Fusion (MF)*



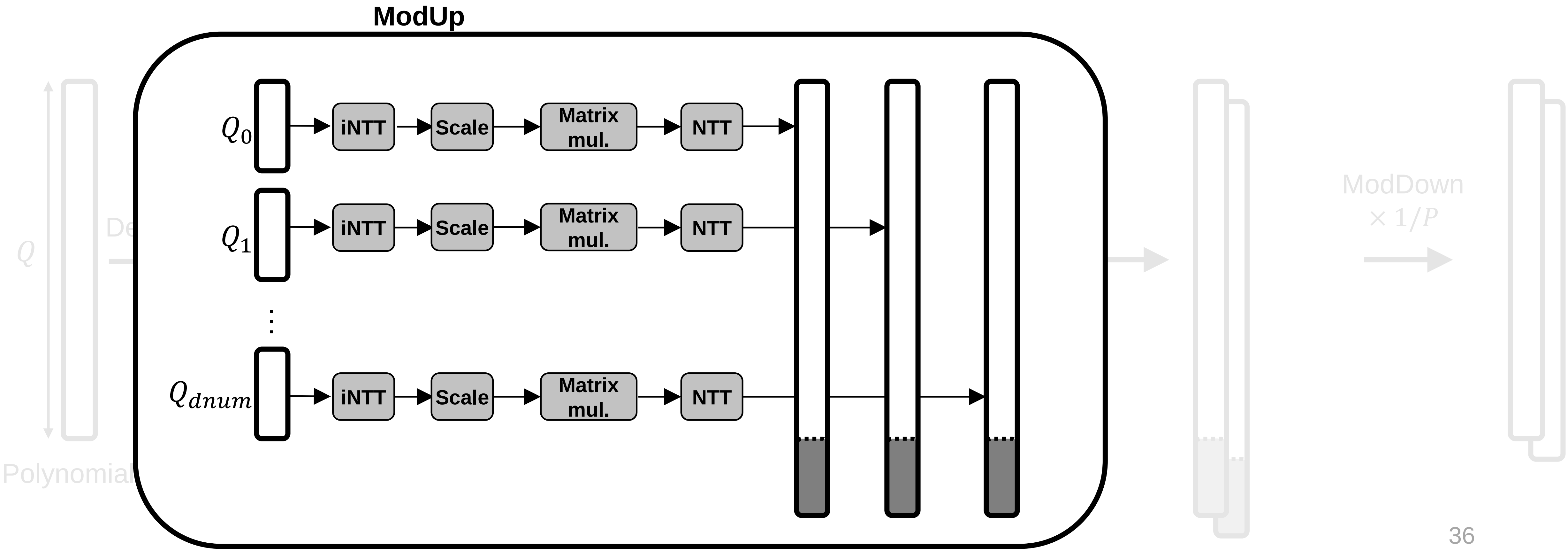
Intra-HE-operation Fusions

- *ModUp Fusion (MF)*



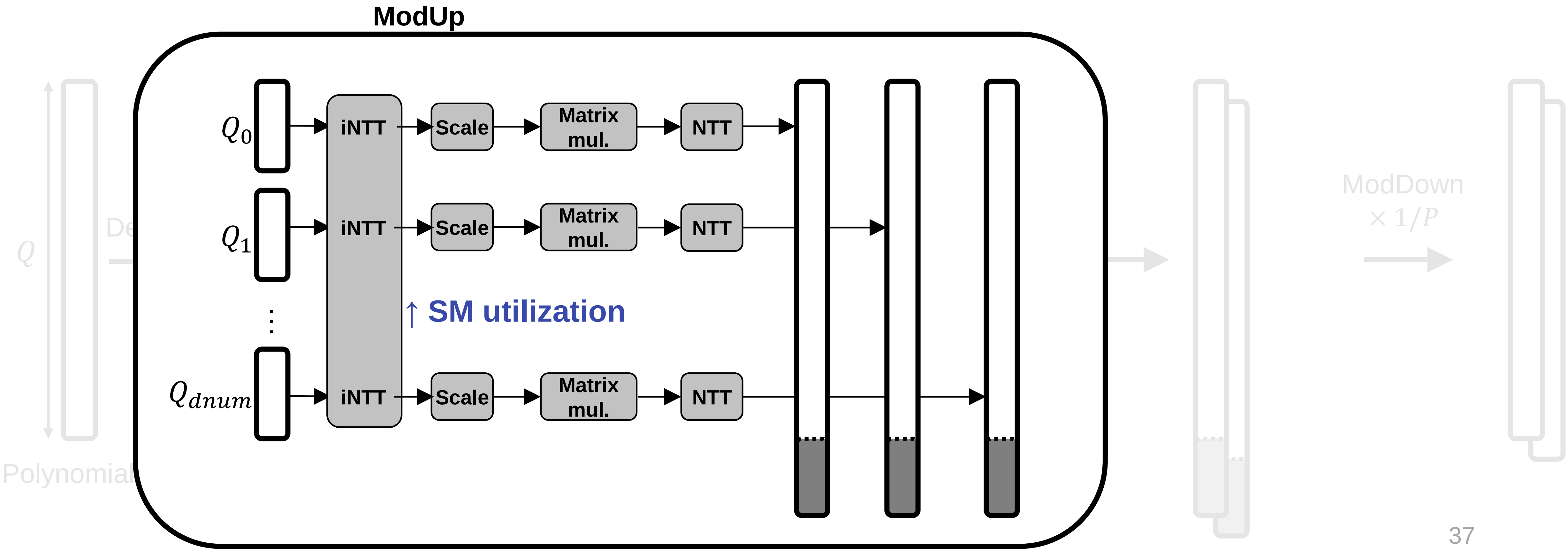
Intra-HE-operation Fusions

- *ModUp Fusion (MF)*
  1. fuses ‘small’ iNTT kernels into a large one.



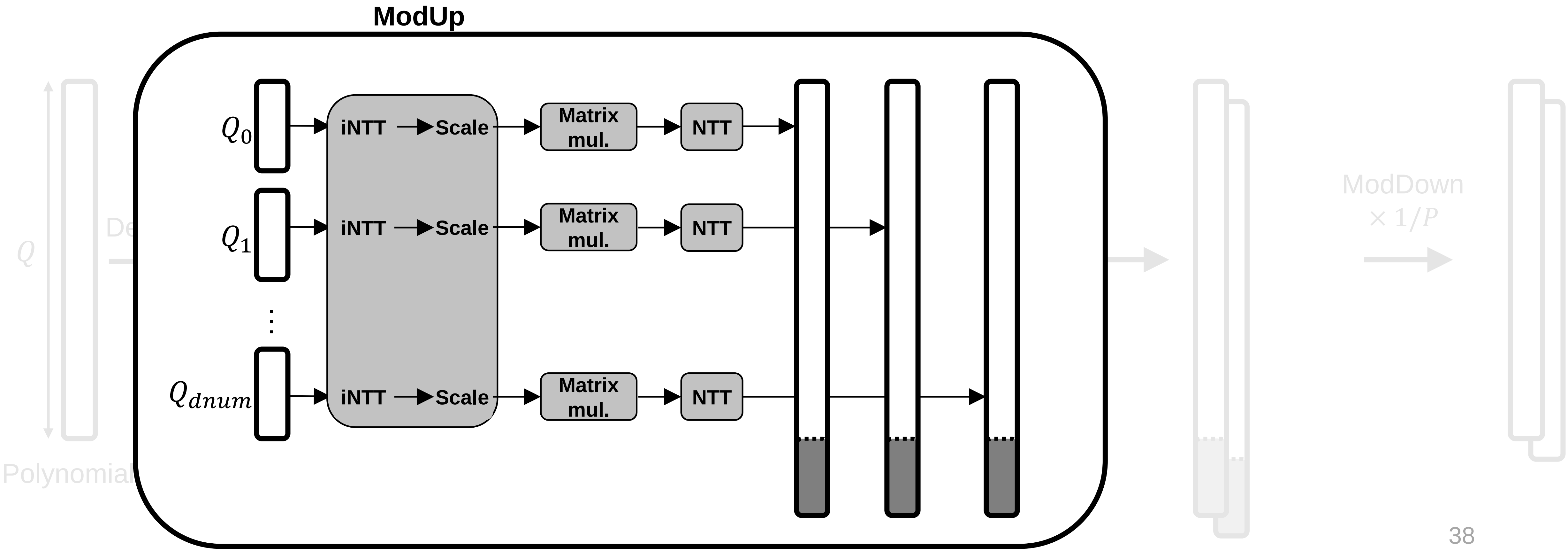
Intra-HE-operation Fusions

- *ModUp Fusion (MF)*
  1. fuses ‘small’ iNTT kernels into a large one.



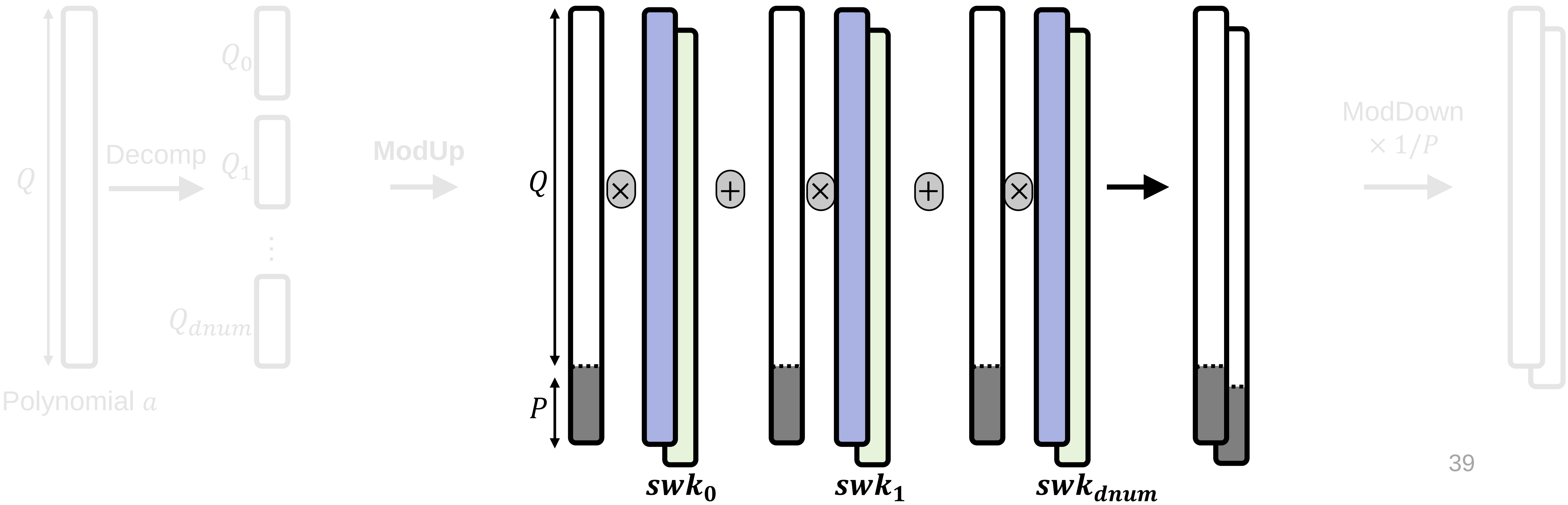
Intra-HE-operation Fusions

- *ModUp Fusion (MF)*
  1. fuses ‘small’ iNTT kernels into a large one.
  2. fuses the scaling kernels with iNTTs.



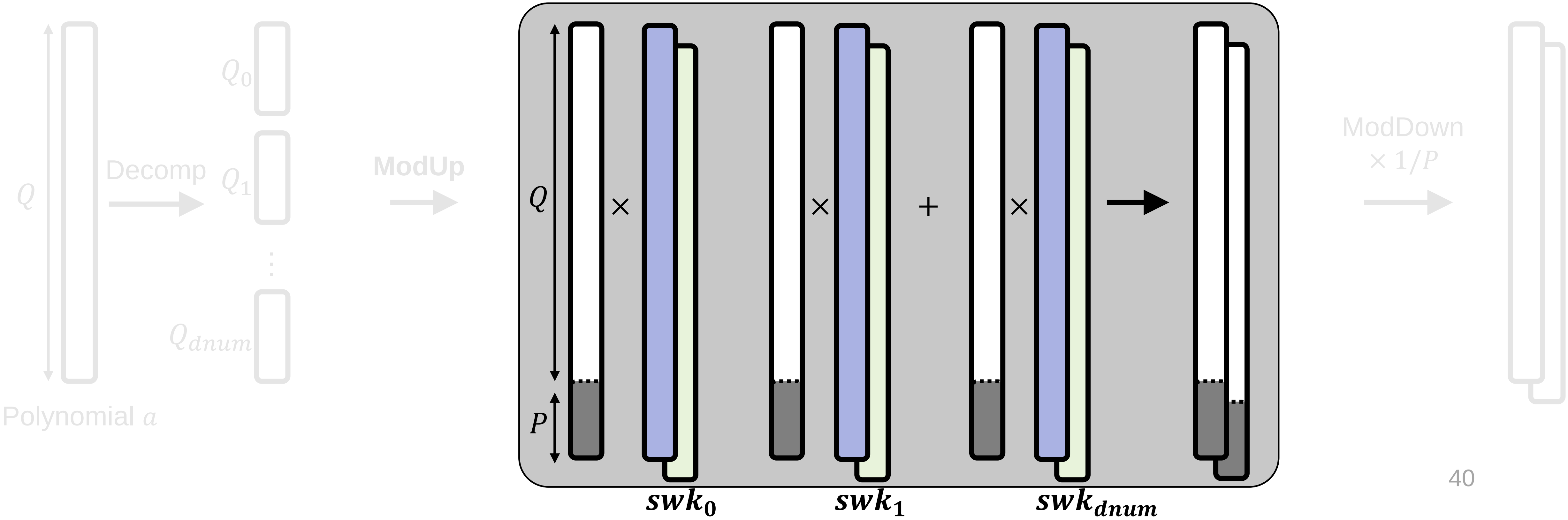
Intra-HE-operation Fusions

- *Inner-product Fusion (IF)*



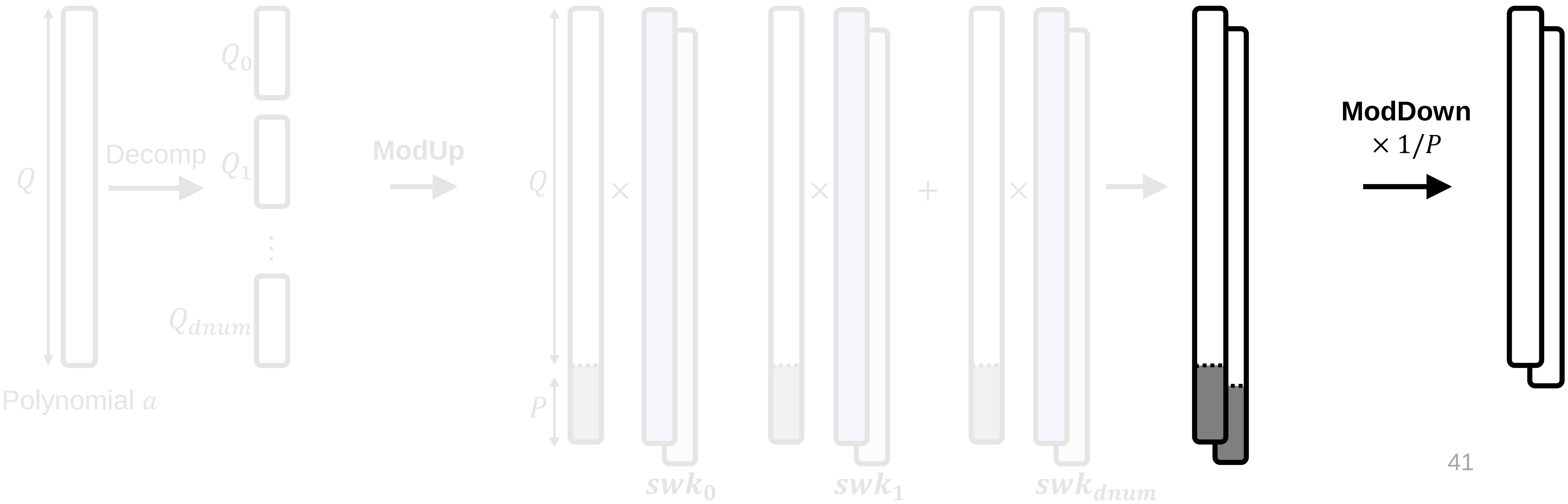
Intra-HE-operation Fusions

- *Inner-product Fusion (IF)*
  - fuses multiplications and additions into one kernel.



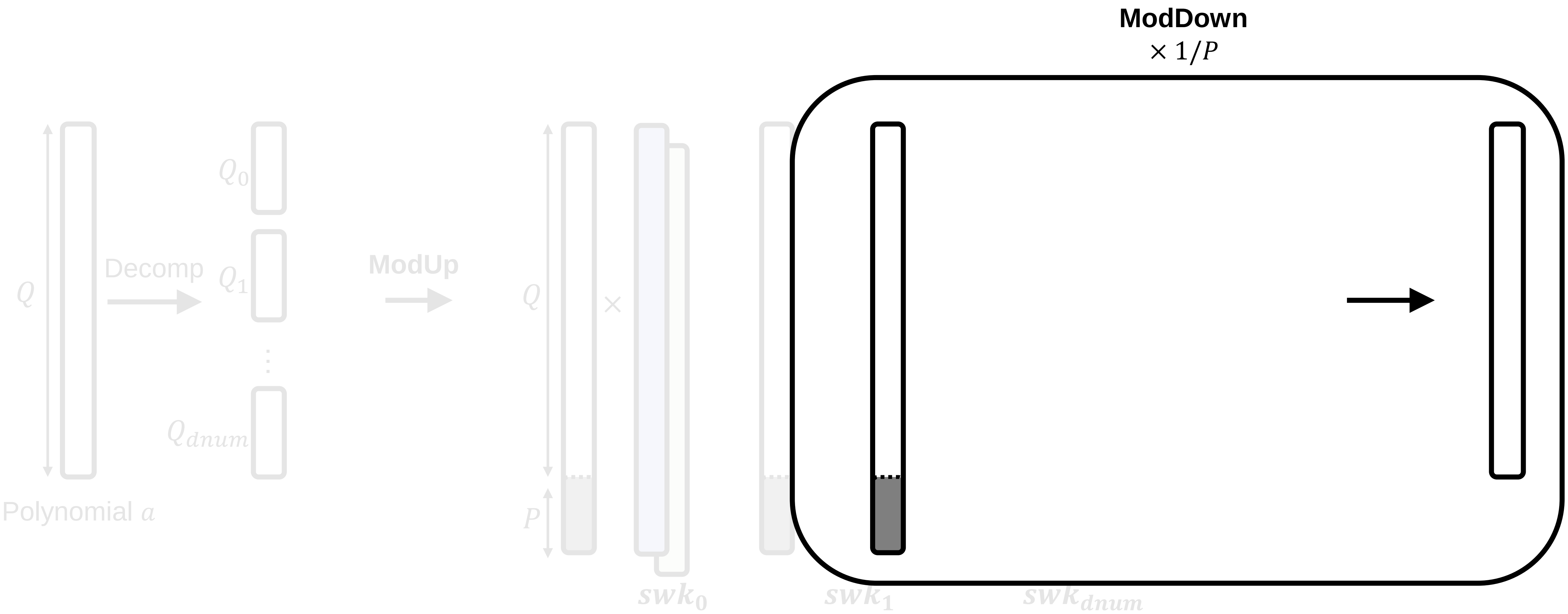
Intra-HE-operation Fusions

- *ModDown Fusion (MDF)*



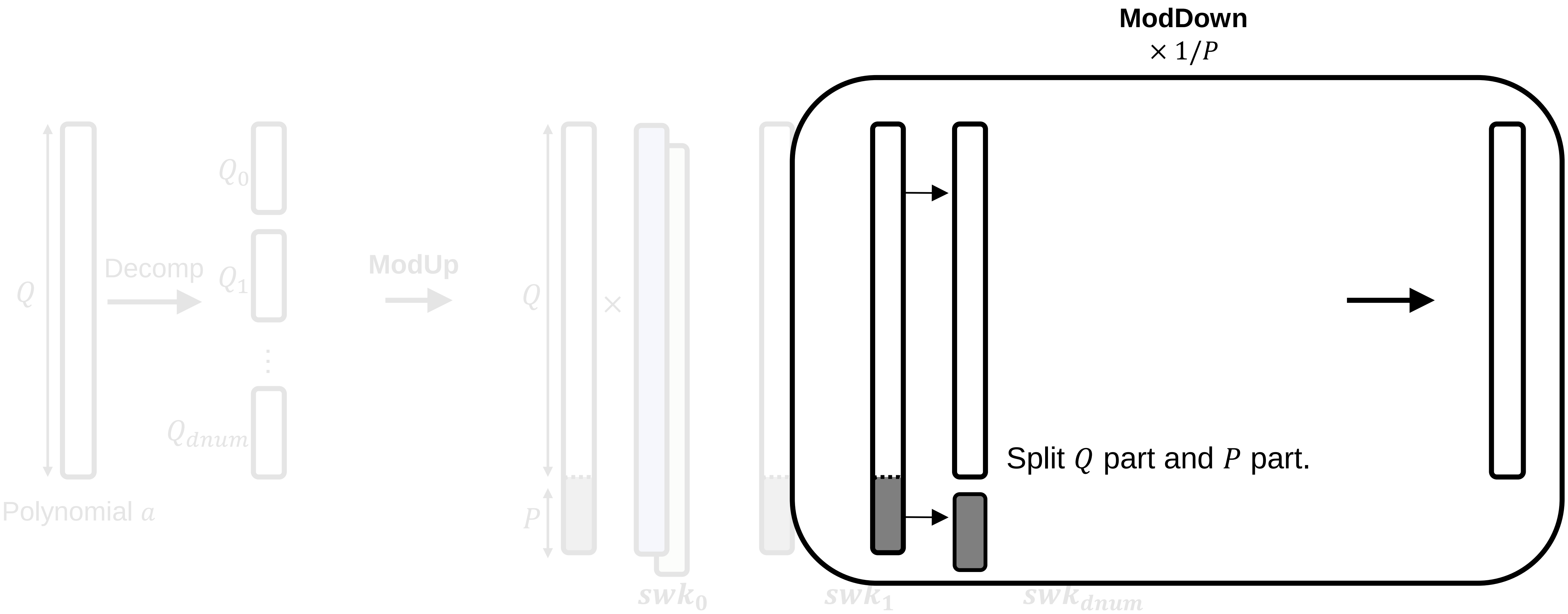
Intra-HE-operation Fusions

- *ModDown Fusion (MDF)*



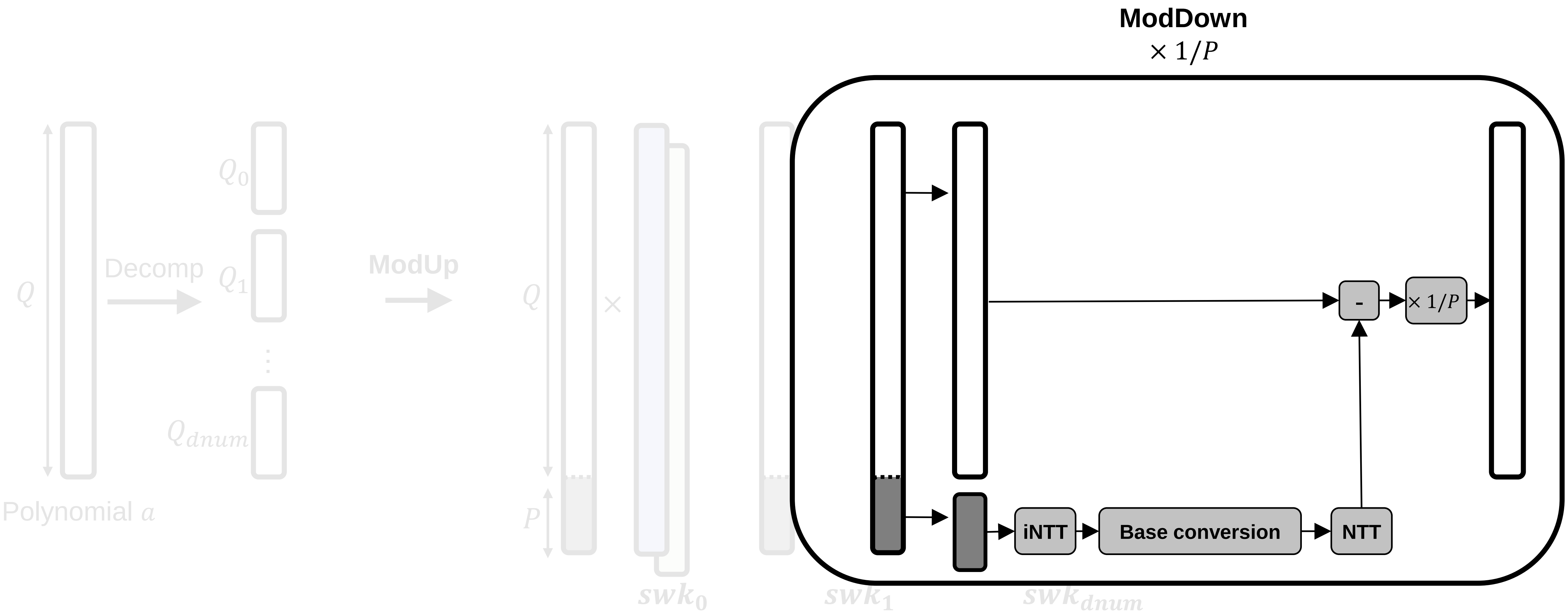
Intra-HE-operation Fusions

- *ModDown Fusion (MDF)*



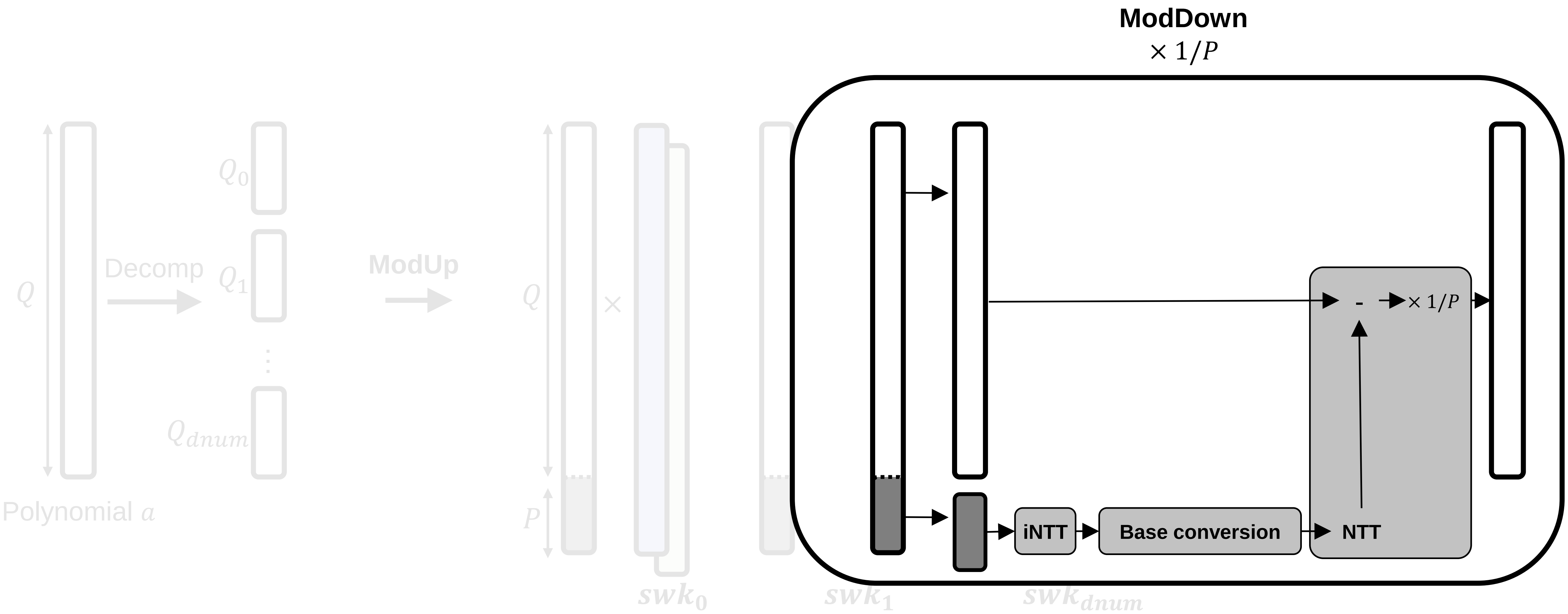
Intra-HE-operation Fusions

- *ModDown Fusion (MDF)*



Intra-HE-operation Fusions

- *ModDown Fusion (MDF)*
  - fuses the element-wise, memory-bound kernels with NTTs.



Experiment results

- Latency of Mult and Rotate after the optimizations.

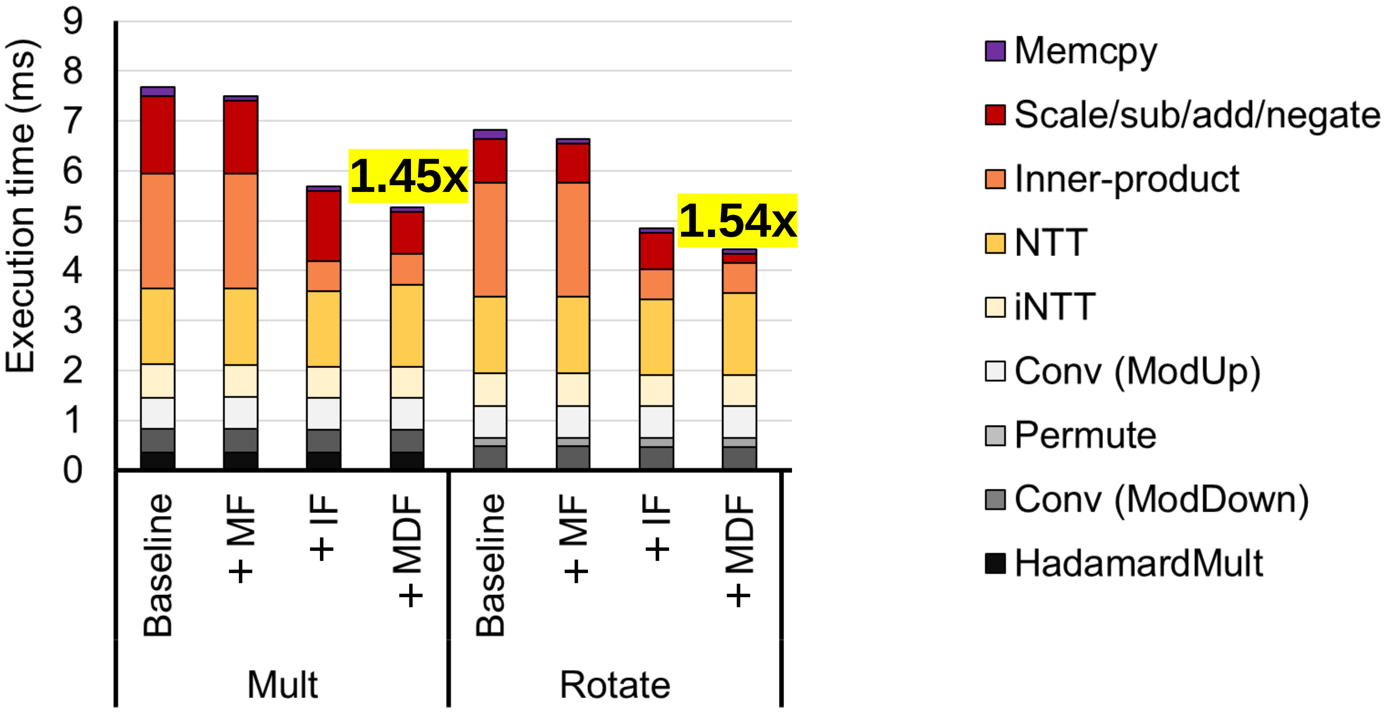
Latency with **dnum** = L (maximum)

$N = 2^{16}$, $\log Q = 2305$, $L = 44$. Used an NVIDIA V100.

Experiment results

- Latency of Mult and Rotate after the optimizations.
- **7.02x** (7.96 vs. 55.88 ms) faster Mult over a prior GPU impl. [BHM+20] which uses **dnum** = L
  - (same security & level)

Latency with **dnum** = L (maximum)

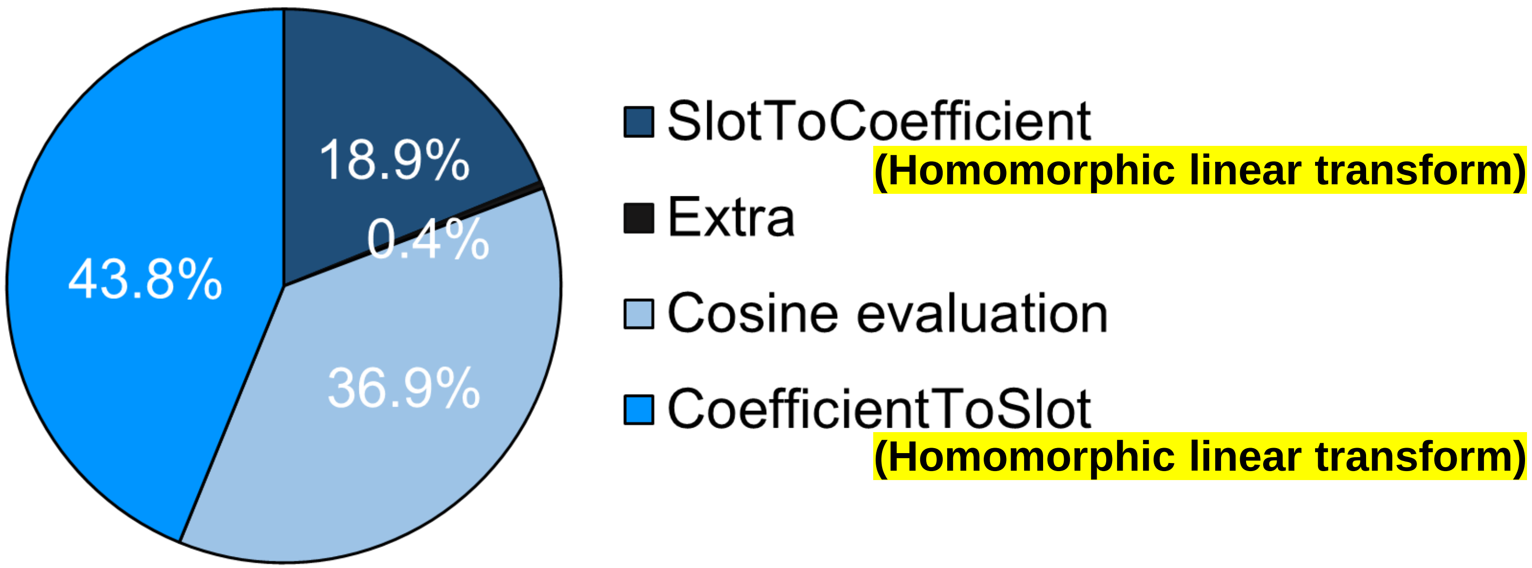


Latency with **dnum** = 3 [HK20]

$N = 2^{16}$, $\log Q = 2305$, $L = 44$. Used an NVIDIA V100.

Bootstrapping in CKKS

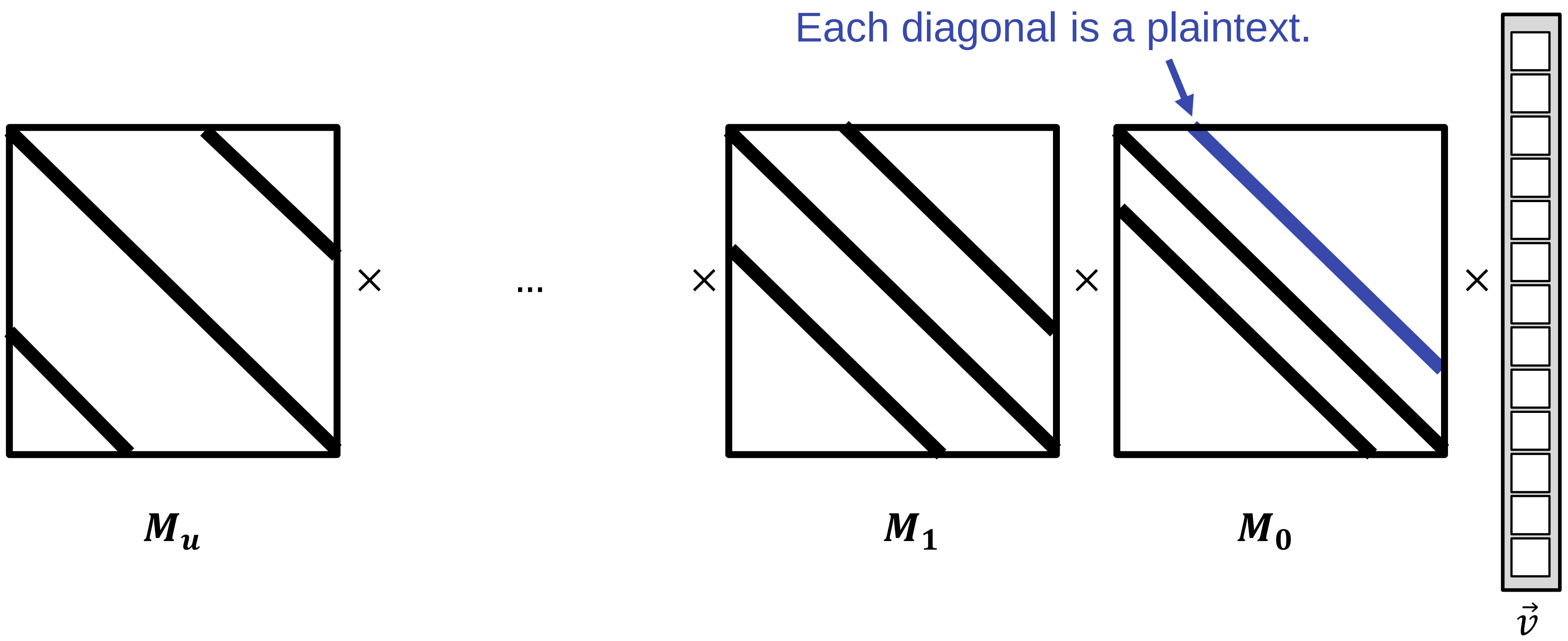
- Bootstrapping is an HE circuit, consisting of Mult, CMult, Rotate, ...
- Breakdown of a bootstrapping latency
  - Homomorphic linear transformations take up of 62.7%.



Bootstrapping latency breakdown on a GPU.

Homomorphic linear transformation in bootstrapping

- Represented as a series of matrix-vector multiplications, where
  - the matrix is sparse-diagonal matrices whose diagonals are plaintexts.
  - the vector is a ciphertext.
- How do we compute this?



ciphertext encrypting complex numbers.

Baby-step Giant-step (BSGS) algorithm

- A core algorithm for homomorphic linear transformation.
  - Used in matrix-vector multiplications in SlotToCoefficient/CoefficientToSlot
  - Setting $\ell, k \cong \sqrt{n}$, the # of rotations becomes $O(n)$➔$O(\sqrt{n})$.

The i-th diagonal element

Vector rotated by i

$$\boldsymbol{M} \cdot \vec{v} = \sum_{i=0}^{n} \mathsf{diag}_i(\boldsymbol{M}) \odot \mathsf{rot}_i(\vec{v})$$

$$= \sum_{i=0}^{\ell} \sum_{j=0}^{k} \mathsf{diag}_{ki+j}(\boldsymbol{M}) \odot \mathsf{rot}_{ki+j}(\vec{v})$$

$$= \sum_{i=0}^{\ell} \mathsf{rot}_{ki} \left(\sum_{j=0}^{k} \underbrace{\mathsf{rot}_{-ki}(\mathsf{diag}_{ki+j}(\boldsymbol{M}))}_{\text{Precomputed plaintext}} \odot \mathsf{rot}_j(\vec{v}) \right)$$

Ciphertext

Improved Homomorphic Discrete Fourier Transforms and FHE Bootstrapping, Kyoohyung Han, Minki Hhan, and Jung Hee Cheon, IEEE Access, 2019

Inter-HE-operation Fusion: Mult-and-add batching in BSGS

- Naïve multiplication-and-add requires multiple memory accesses on a temporal ciphertext.
  - Many reads and writes on $\boldsymbol{ct}$.

$$
\begin{aligned}
\boldsymbol{ct} &= m_1 \times \boldsymbol{ct_1} \\
\boldsymbol{ct} &= m_2 \times \boldsymbol{ct_2} + \boldsymbol{ct} \\
\boldsymbol{ct} &= m_3 \times \boldsymbol{ct_1} + \boldsymbol{ct} \\
&\dots
\end{aligned}
$$

The i-th diagonal element

Vector rotated by i

$$
\begin{aligned}
\boldsymbol{M} \cdot \vec{v} &= \sum_{i=0}^{n} \mathsf{diag}_i(\boldsymbol{M}) \odot \mathsf{rot}_i(\vec{v}) \\
&= \sum_{i=0}^{\ell} \sum_{j=0}^{k} \mathsf{diag}_{ki+j}(\boldsymbol{M}) \odot \mathsf{rot}_{ki+j}(\vec{v}) \\
&= \sum_{i=0}^{\ell} \mathsf{rot}_{ki} \left(\sum_{j=0}^{k} \underbrace{\mathsf{rot}_{-ki}(\mathsf{diag}_{ki+j}(\boldsymbol{M}))}_{\text{Precomputed plaintext}} \odot \overbrace{\mathsf{rot}_j(\vec{v})}^{\text{Ciphertext}} \right)
\end{aligned}
$$

Inter-HE-operation Fusion: Mult-and-add batching in BSGS

- Naïve multiplication-and-add requires multiple memory accesses on a temporal ciphertext.
  - Many reads and writes on $\boldsymbol{ct}$.
- Fusing multiple mul-and-add operations
  - removes most of read and writes on $\boldsymbol{ct}$.

$$\boldsymbol{ct} = m_1 \times \boldsymbol{ct_1}$$
$$\boldsymbol{ct} = m_2 \times \boldsymbol{ct_2} + \boldsymbol{ct}$$
$$\boldsymbol{ct} = m_3 \times \boldsymbol{ct_1} + \boldsymbol{ct}$$
$$\ldots$$

$$\boldsymbol{ct} = m_1 \times \boldsymbol{ct_1} + m_2 \times \boldsymbol{ct_2} + m_3 \times \boldsymbol{ct_1} + \cdots$$

(A single kernel)

The i-th diagonal element

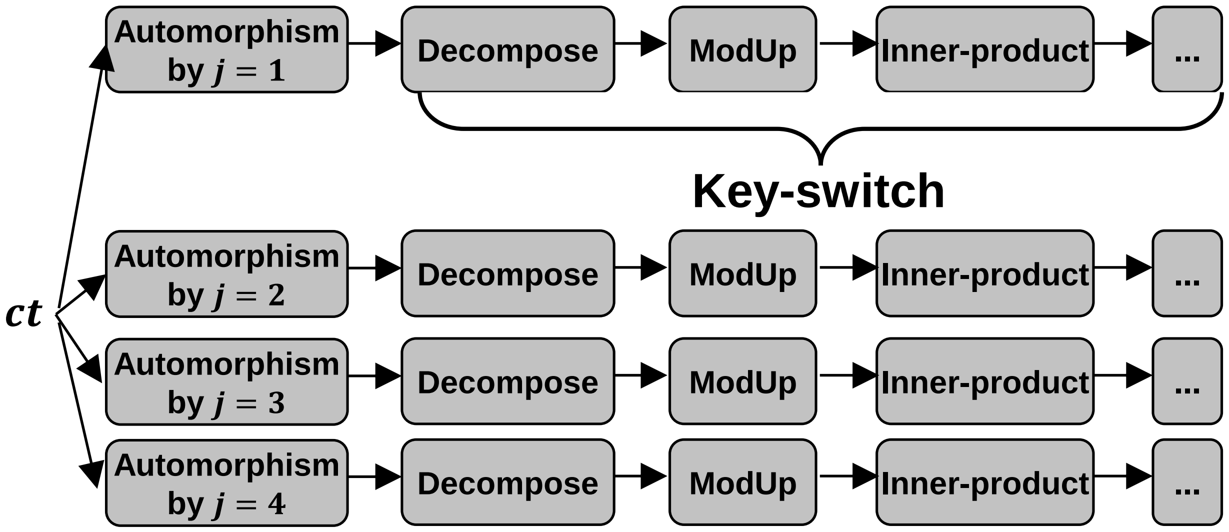
Vector rotated by i

$$\boldsymbol{M} \cdot \vec{v} = \sum_{i=0}^{n} \mathrm{diag}_i(\boldsymbol{M}) \odot \mathrm{rot}_i(\vec{v})$$
$$= \sum_{i=0}^{\ell} \sum_{j=0}^{k} \mathrm{diag}_{ki+j}(\boldsymbol{M}) \odot \mathrm{rot}_{ki+j}(\vec{v})$$
$$= \sum_{i=0}^{\ell} \mathrm{rot}_{ki} \left(\sum_{j=0}^{k} \underbrace{\mathrm{rot}_{-ki}(\mathrm{diag}_{ki+j}(\boldsymbol{M}))}_{\text{Precomputed plaintext}} \odot \mathrm{rot}_j(\vec{v}) \right)$$

Ciphertext

Hoisting [HS18]

- We also evaluated an optimization in [HS18]
- When rotating a single ciphertext multiple times, we save some ModUp computations.
  - Precomputes ModUp first, then permutes by each rotation index.



$$
\begin{aligned}
\boldsymbol{M} \cdot \vec{v} &= \sum_{i=0}^{n} \mathrm{diag}_i(\boldsymbol{M}) \odot \mathrm{rot}_i(\vec{v}) \\
&= \sum_{i=0}^{\ell} \sum_{j=0}^{k} \mathrm{diag}_{ki+j}(\boldsymbol{M}) \odot \mathrm{rot}_{ki+j}(\vec{v}) \\
&= \sum_{i=0}^{\ell} \mathrm{rot}_{ki} \left(\sum_{j=0}^{k} \mathrm{rot}_{-ki}(\mathrm{diag}_{ki+j}(\boldsymbol{M})) \odot \boxed{\mathrm{rot}_j(\vec{v})} \right)
\end{aligned}
$$

Performs rotations with different indices.

Experiment results

- Bootstrapping latency

	Latency (ms)				
Parameter set (N, L, dnum, λ)	Baseline	Intra-HE fusion	Mult-and-add Batching	Hoisting [HS18]	**Speedup (vs. single-thread CPU)**
$(2^{16}, 34, 5, 106)$	428.94	377.78	351.09	328.25	**242x**
$(2^{17}, 29, 3, 173)$	719.87	623.92	568.2	526.96	**257x**

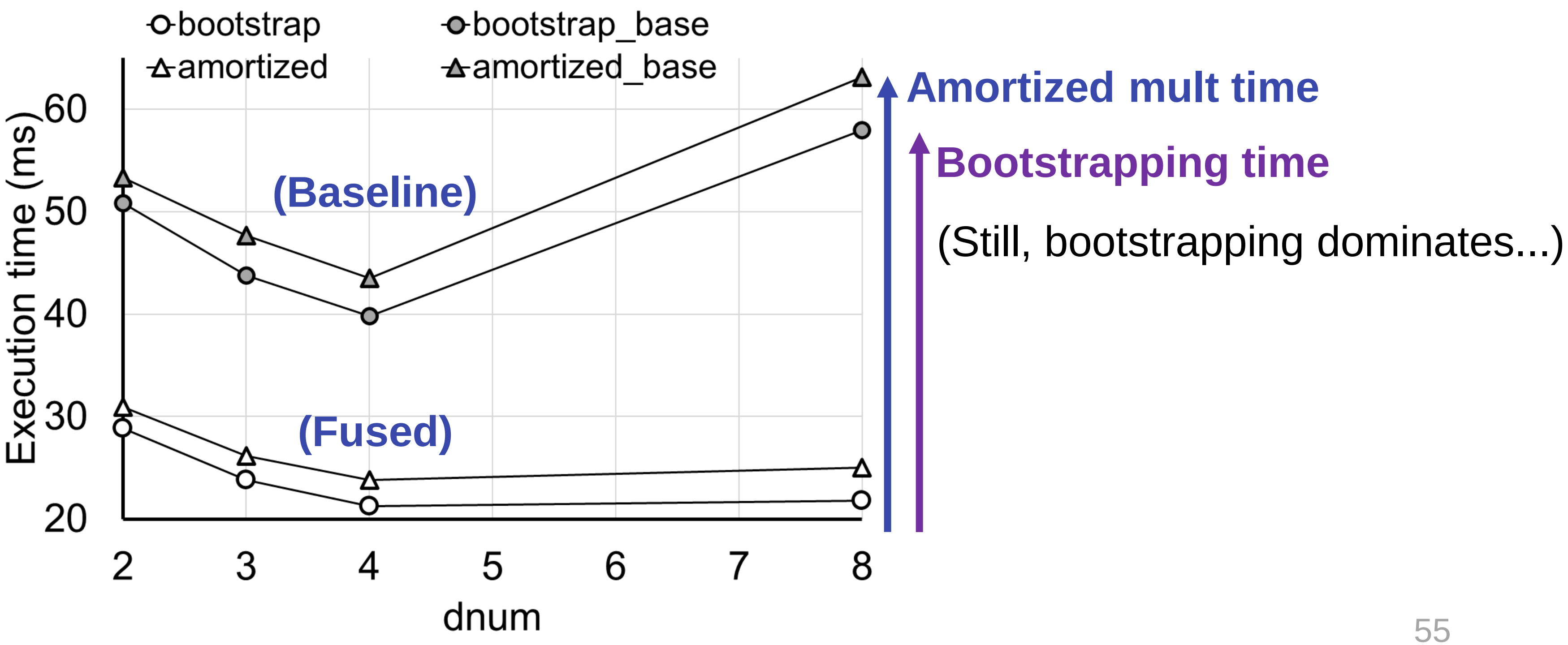
- End-to-end evaluation: training a binary classification (logistic regression) model
  - Compared to 8-threaded CPU case, GPU implementation exhibits a speedup of **40x** in total.

Amortized Mult-time

- We suggest a new metric for performance considering bootstrapping:

$$Amotized\ mult\ time = \frac{Bootstrapping\ time + multiplication\ times}{\#\ of\ multiplications\ after\ bootstrapping}$$

- Our optimizations largely amortize the memory bottleneck.



Summary

- Based on our performance analysis, we accelerated binary CKKS (HEAAN) with
  - AVX-512 & GPU implementation with microarchitecture-aware optimizations.
- Further, we accelerated the recent Full-RNS CKKS using GPU, including
  - the first implementation of bootstrapping
  - analysis on the severe memory-bandwidth bottlenecks
  - applying memory-centric optimizations, giving
    - = 257x of speedup (vs. single-threaded CPU),
    - = application-level speedups (40x vs. 8-threaded CPU in logistic regression)

Reference

- [HK20] Han et al., Better Bootstrapping for Approximate Homomorphic Encryption, CT-RSA, 2020
- [BHM+20] Ahmad Al Badawi, Louie Hoang, Chan Fook Mun, Kim Laine, and Khin Mi Mi Aung. Privft: Private and fast text classification with homomorphic encryption, IEEE Access, 2020
- [CLP17] Hao Chen, Kim Laine, and Rachel Player. Simple encrypted arithmetic library - SEAL v2.1. In FC 2017 Workshops, volume 10323 of LNCS, pages 3–18. Springer, Heidelberg, April 2017.
- [HS14] Shai Halevi and Victor Shoup. Algorithms in HElib. In CRYPTO 2014, Part I, volume 8616 of LNCS, pages 554–571. Springer, Heidelberg, August 2014.
- [CHK+19] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In SAC 2018, volume 11349 of LNCS, pages 347–368. Springer, Heidelberg, August 2019.
- [BPA+19] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme. IEEE Transactions on Emerging Topics in Computing, 2019.
- [KJPA20] Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In IEEE International Symposium on Workload Characterization, IISWC 2020, Beijing, China, October 27-30, 2020, pages 264–275. IEEE, 2020.
- [HS18] Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in HElib. In CRYPTO 2018, Part I, volume 10991 of LNCS, pages 93–120. Springer, Heidelberg, August 2018.

감사합니다