ASP-DAC 2025

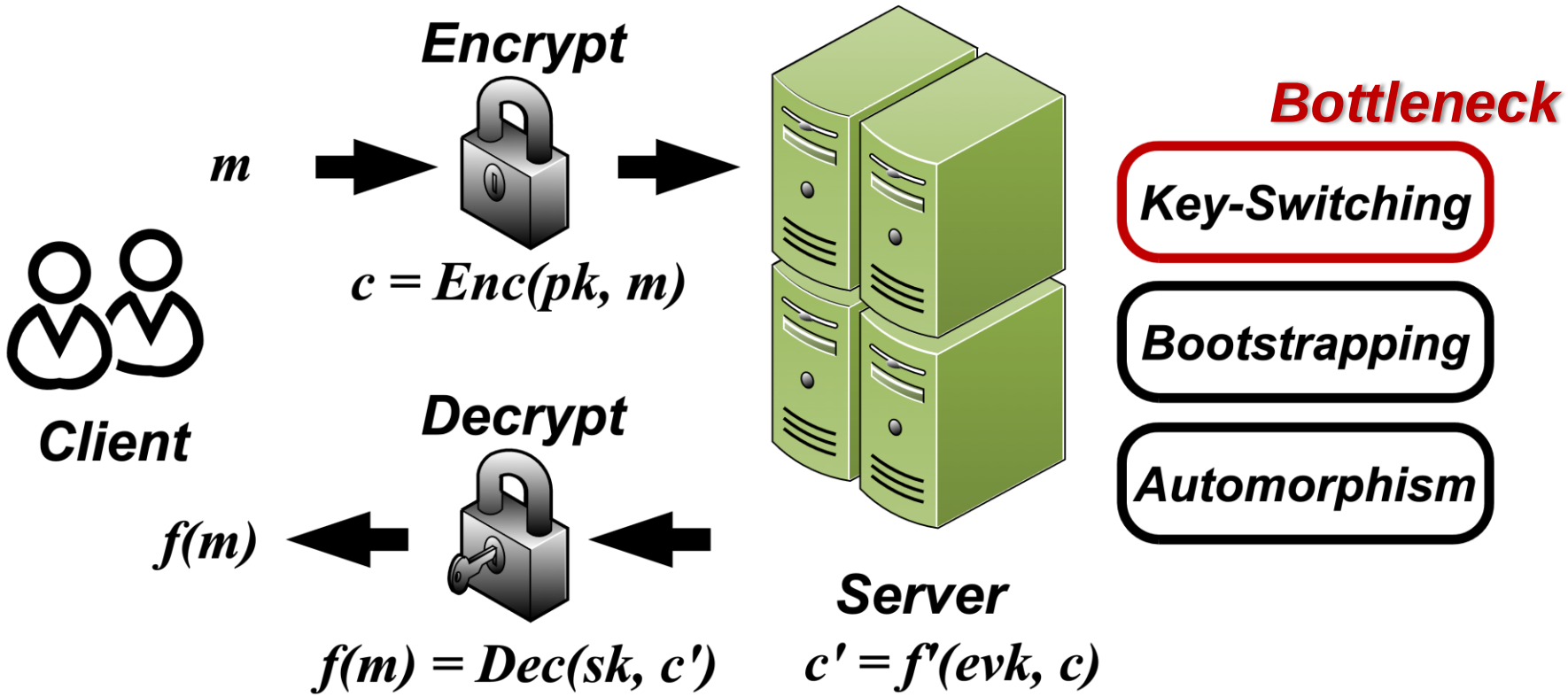
Efficient Key Switching Accelerator for Fully Homomorphic Encryption

Seoyoon Jang, Sungjin Park, Dongsuk Jeon
Seoul National University
Seoul, South Korea

Mobile Multimedia Systems Group

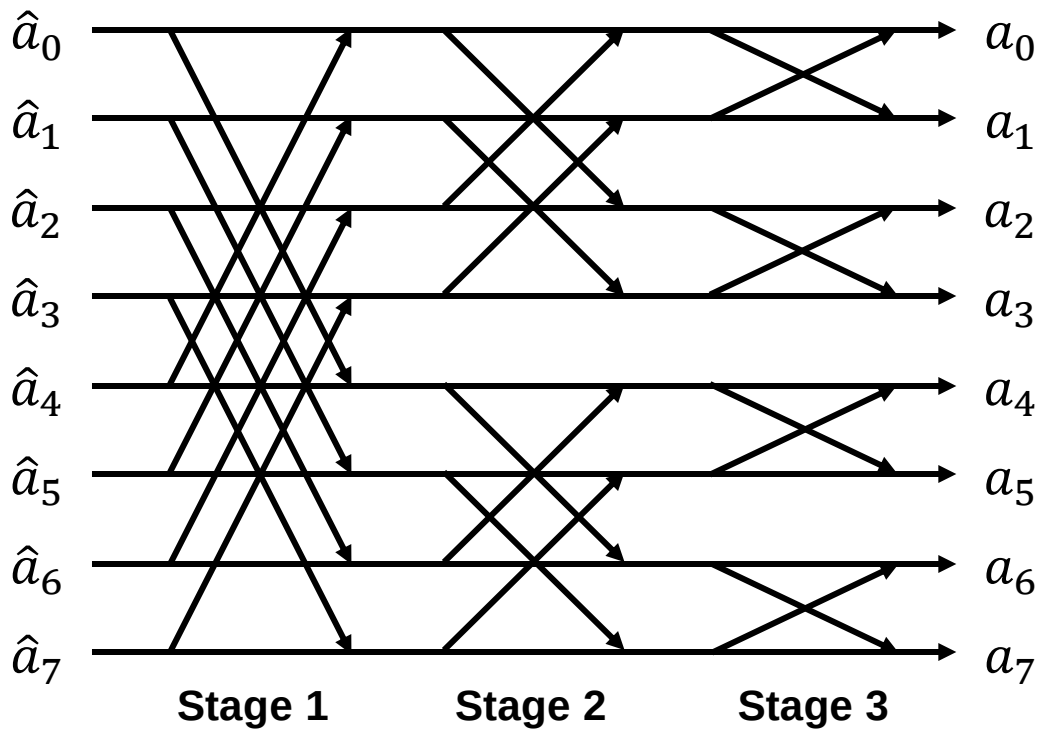
Motivation – Advent of FHE

- Fully Homomorphic Encryption (FHE)
  - The savior of privacy-preserving computation in cloud service
- The main bottleneck operation, Key-Switching (KS)



Motivation – What makes KS expensive?

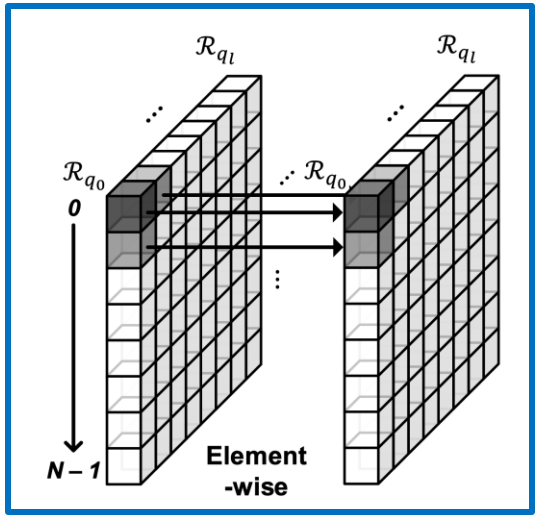
- Expensive Number Theoretic Transform (NTT) and inverse-NTT operations in KS
  - $\{\hat{a}_i\}_i = NTT(\{a_i\}_i) = \left\{\Sigma_{j=0}^{N-1} a_j \omega^{ij} \; mod \; q\right\}_i$
  - At least $O(N \log N)$ computations
  - Irregular memory access pattern



Motivation – What makes KS expensive?

- Expensive Number Theoretic Transform (NTT) and inverse-NTT operations in KS
- Frequent transitions in data access patterns (Element/Ring/Coefficient-wise)

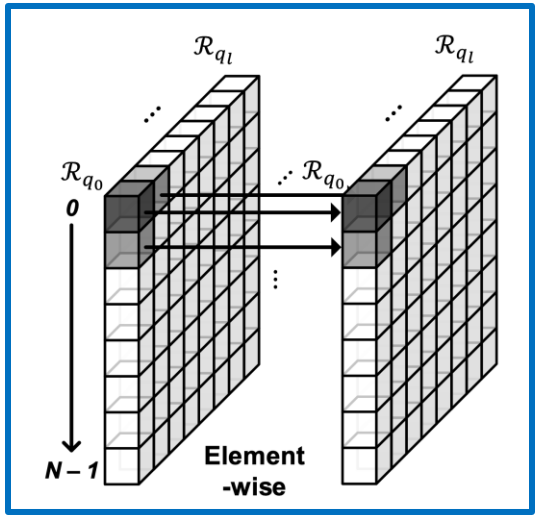
Element-wise



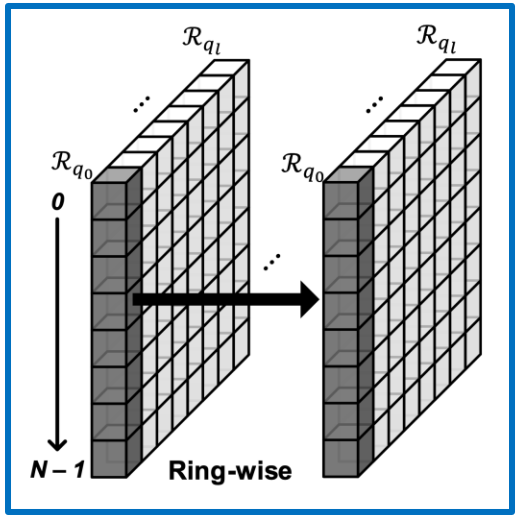
Motivation – What makes KS expensive?

- Expensive Number Theoretic Transform (NTT) and inverse-NTT operations in KS
- Frequent transitions in data access patterns (Element/Ring/Coefficient-wise)

Element-wise



Ring-wise



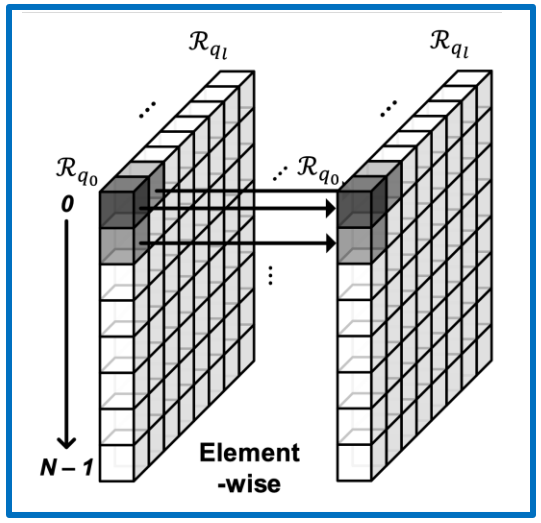
Data access within
each polynomial

(NTT/INTT,…)

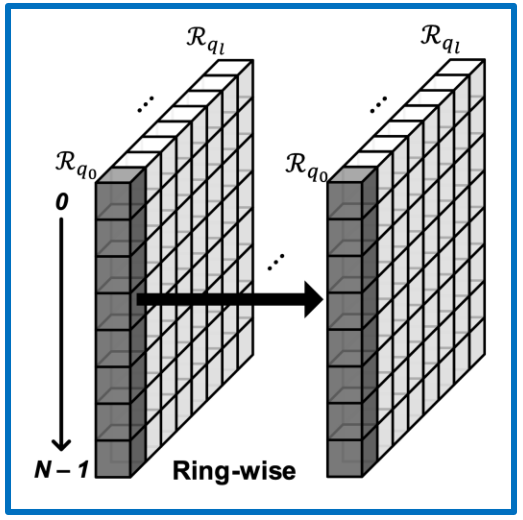
Motivation – What makes KS expensive?

- Expensive Number Theoretic Transform (NTT) and inverse-NTT operations in KS
- Frequent transitions in data access patterns (Element/Ring/Coefficient-wise)

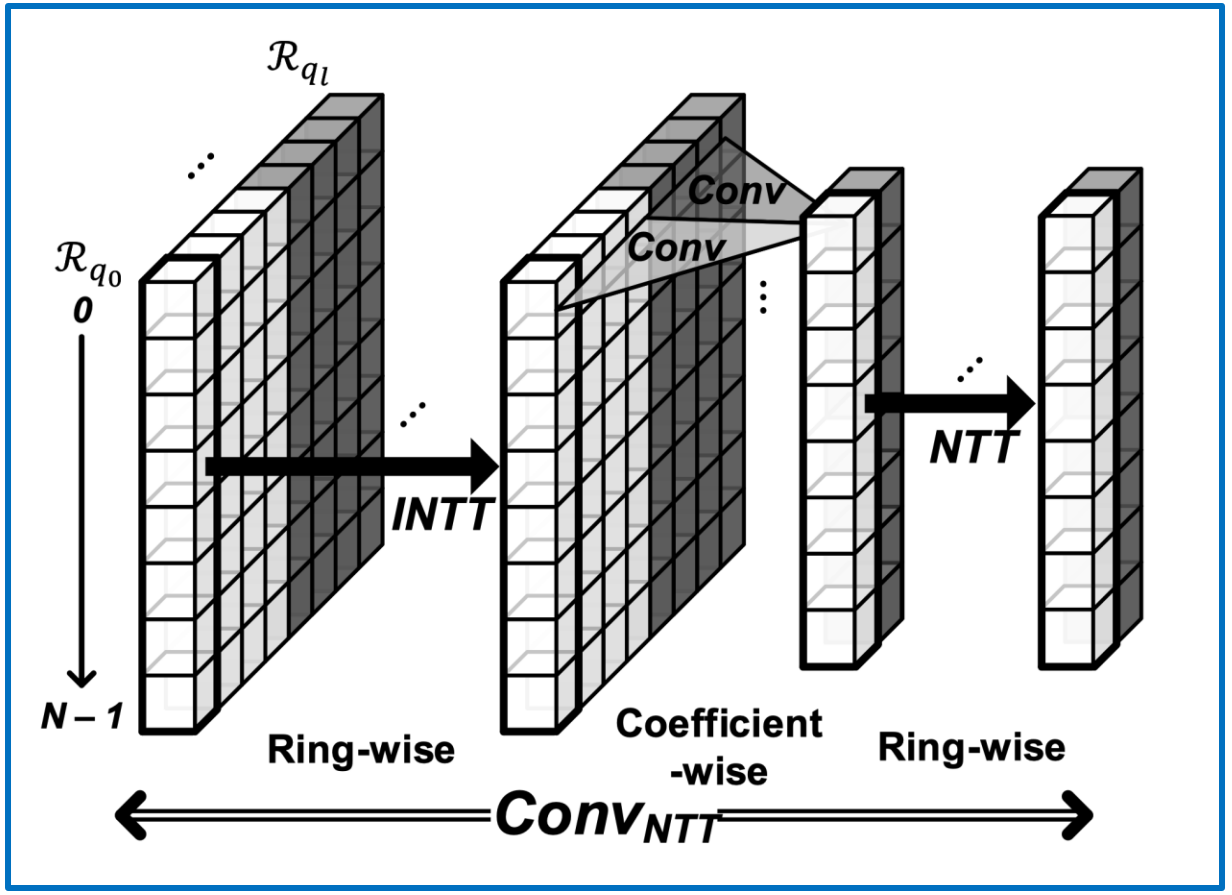
Element-wise



Ring-wise

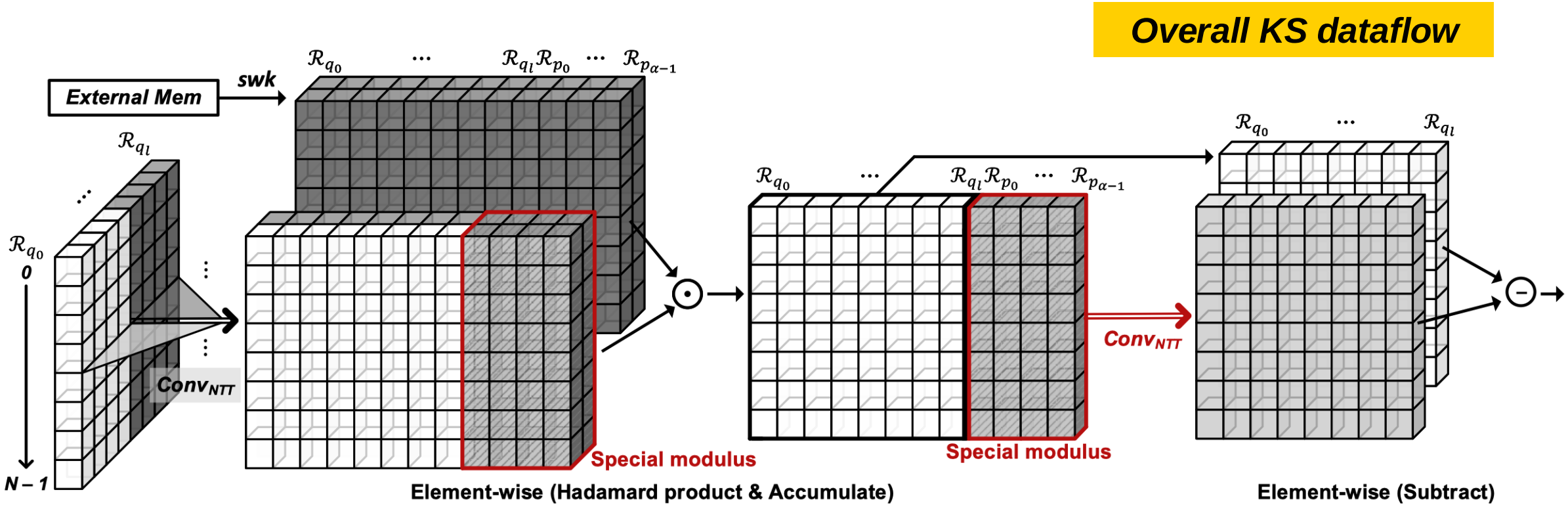


Ring-wise
↓
Coefficient -wise
↓
Ring-wise



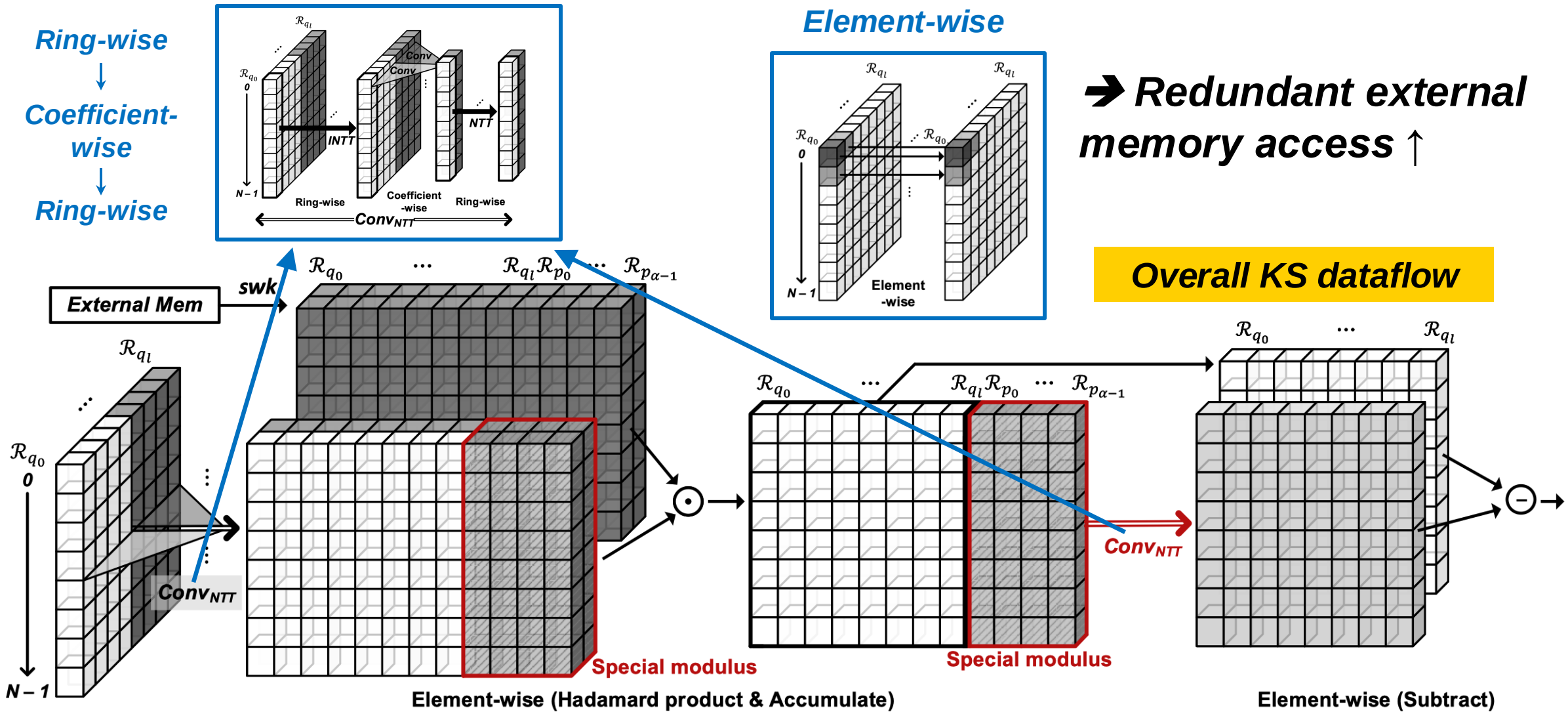
Motivation – What makes KS expensive?

- Expensive Number Theoretic Transform (NTT) and inverse-NTT operations in KS
- Frequent transitions in data access patterns (Element/Ring/Coefficient-wise)



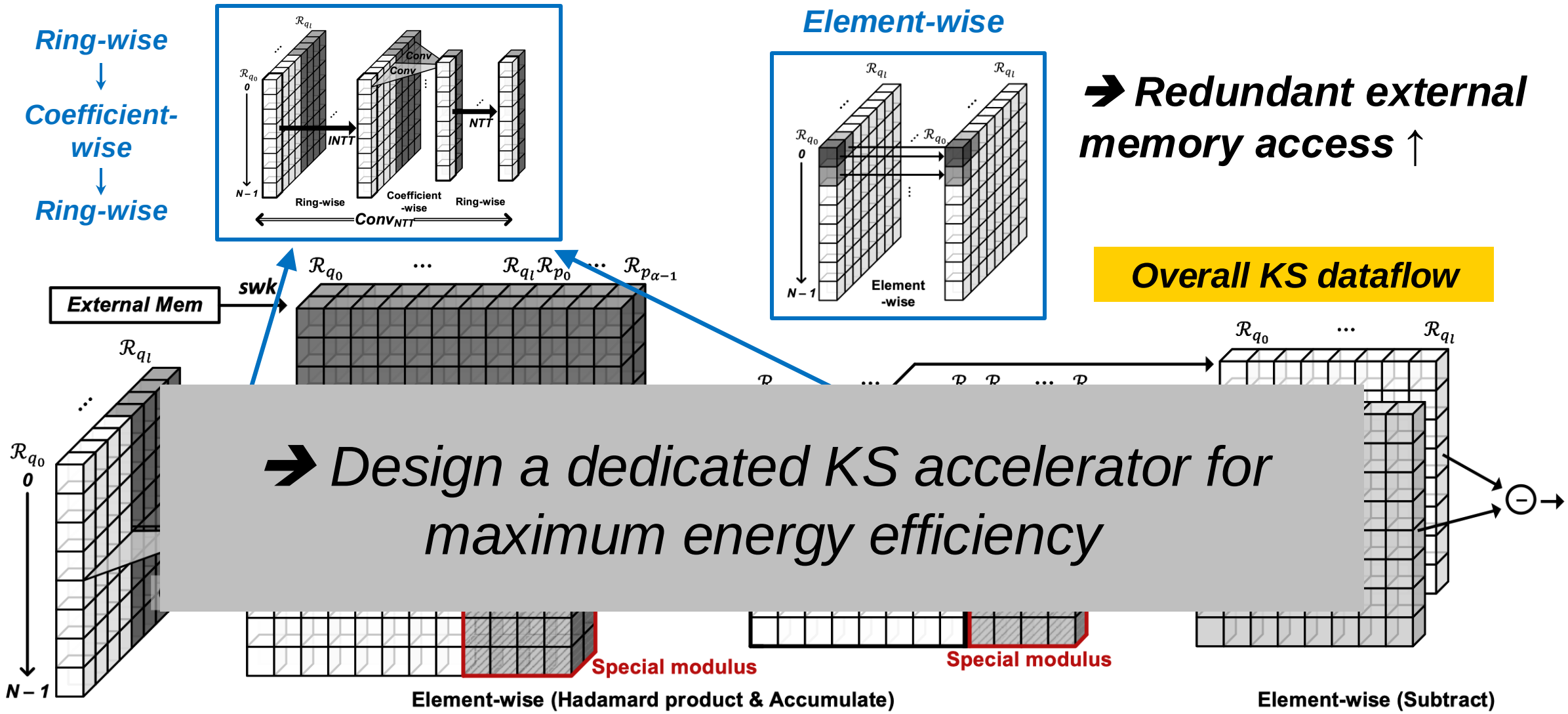
Motivation – What makes KS expensive?

- Expensive Number Theoretic Transform (NTT) and inverse-NTT operations in KS
- Frequent transitions in data access patterns (Element/Ring/Coefficient-wise)



Motivation – What makes KS expensive?

- Expensive Number Theoretic Transform (NTT) and inverse-NTT operations in KS
- Frequent transitions in data access patterns (Element/Ring/Coefficient-wise)



Parameter selected for KS accelerator

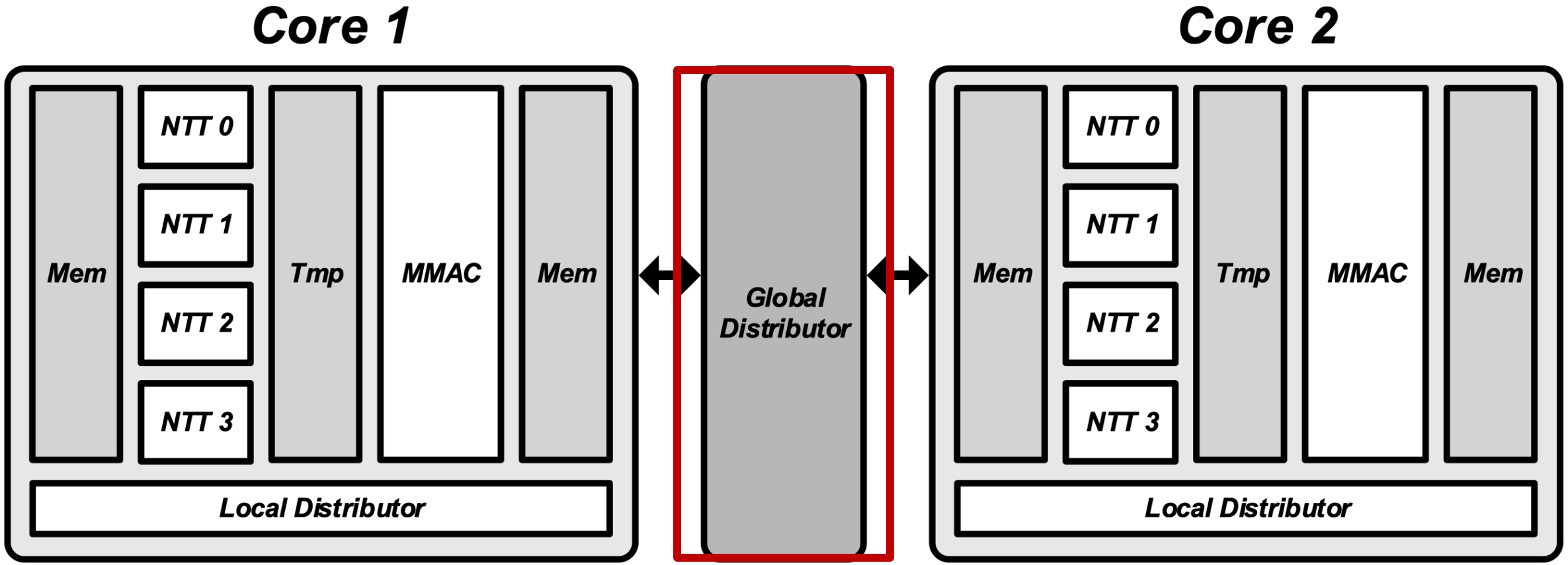
- $(\log N, L, dnum, \alpha) = (17, 35, 9, 4)$
  - N: Number of coefficients for each polynomial
  - L: Maximum circuit depth level
  - $dnum$: decomposition number
    - $\alpha = \lceil (L+1)/dnum \rceil$
    - $\alpha > 1$ → computations ↓, required *swk* ↓

- Constraints for parameters selection
  - Security level $\lambda > 128$
    - Directly related with $N / \log PQ$
  - L large enough to guarantee $L_{boot} = 15 \sim 20$ for bootstrapping that enables FHE
  - → N providing **a sufficient number of lightweight prime moduli for large L**

$$q = 2^{l} \pm 2^{sh_0} \pm 2^{sh_1} \pm 2^{sh_2} + 1$$

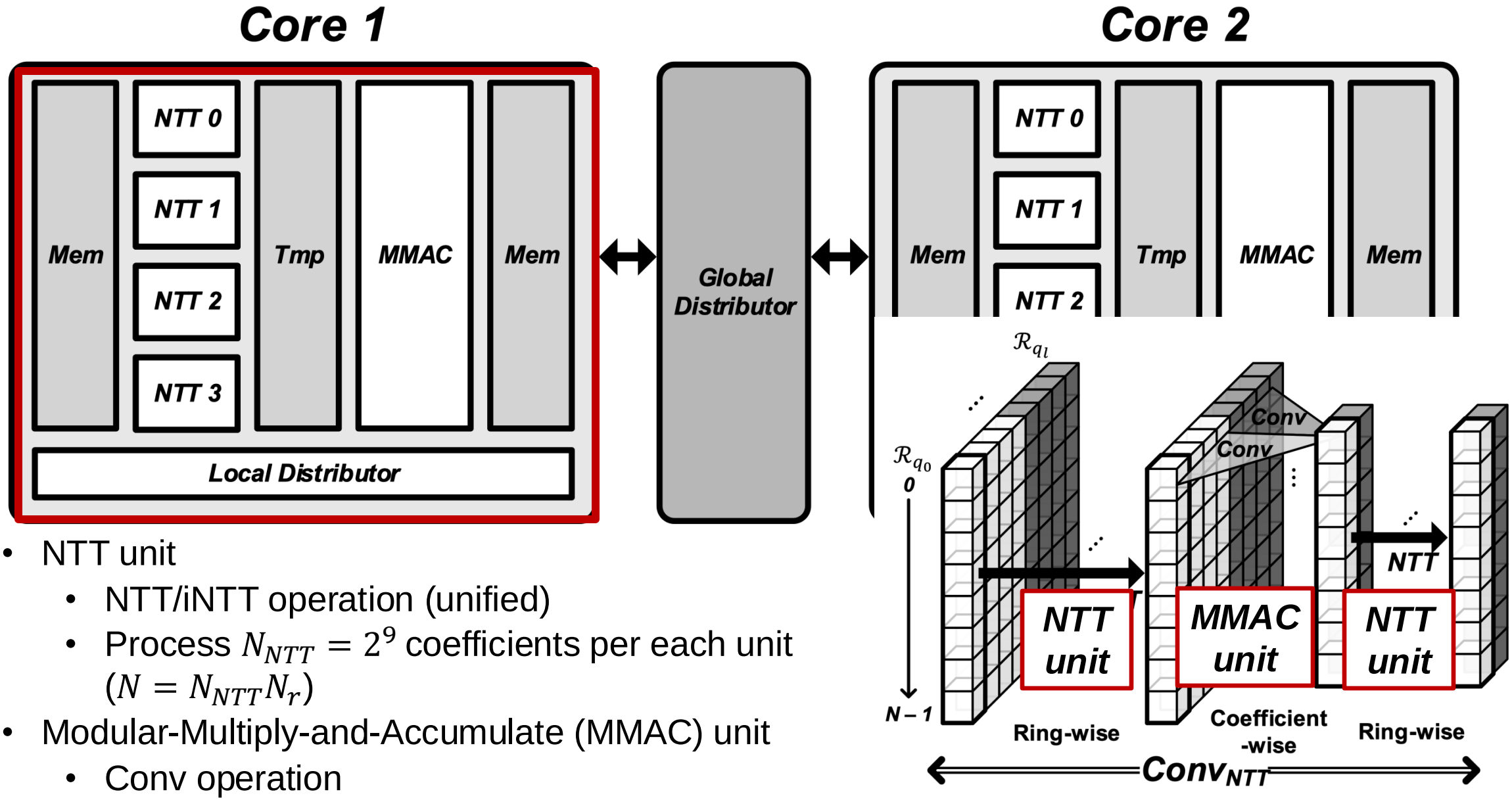
sparse! → enable efficient HW implementation of modular multipliers

Overall Design of KS accelerator



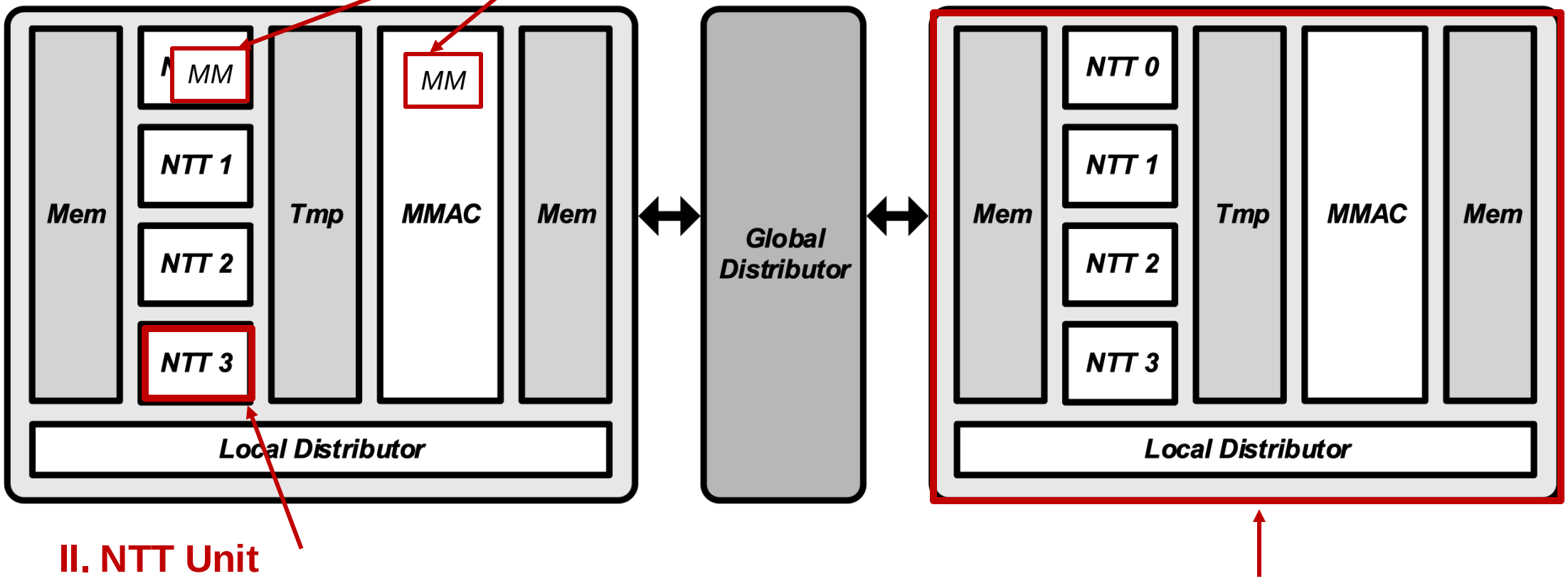
- Router transferring instructions and external data to the target core
- LUT for moduli set and modulus-related constants required by each core

Overall Design of KS accelerator



- NTT unit
  - NTT/iNTT operation (unified)
  - Process $N_{NTT} = 2^9$ coefficients per each unit ($N = N_{NTT}N_r$)
- Modular-Multiply-and-Accumulate (MMAC) unit
  - Conv operation
- Local distributor: internal router

Proposed Design Techniques for Energy Efficiency



I. Modular Multiplier for Sparse Moduli Set

$$q = 2^l \pm 2^{sh_0} \pm 2^{sh_1} \pm 2^{sh_2} + 1 \quad (l > 2 \cdot sh_1 + 1, sh_m > sh_n, \forall m > n)$$

- $* q$ and $* T$ in Barrett modular multiplication

  → replaced with shift-adders

  - $T = \lfloor 2^{2l}/q \rfloor = 2^l \mp 2^{sh_0} \mp 2^{sh_1} \mp 2^{sh_2} - 1$

Benefits of sparsity

I. Modular Multiplier for Sparse Moduli Set

$$q = 2^l \pm 2^{sh_0} \pm 2^{sh_1} \pm 2^{sh_2} + 1$$

$$(l > 2 \cdot sh_0 + 1, sh_m > sh_n, \forall m < n\ )$$

- $* q$ and $* T$ in Barrett modular multiplication
  → replaced with shift-adders
  - $T = \lfloor 2^{2l}/q \rfloor = 2^l \mp 2^{sh_0} \mp 2^{sh_1} \mp 2^{sh_2} - 1$

Benefits of sparsity

Using $l = 59$

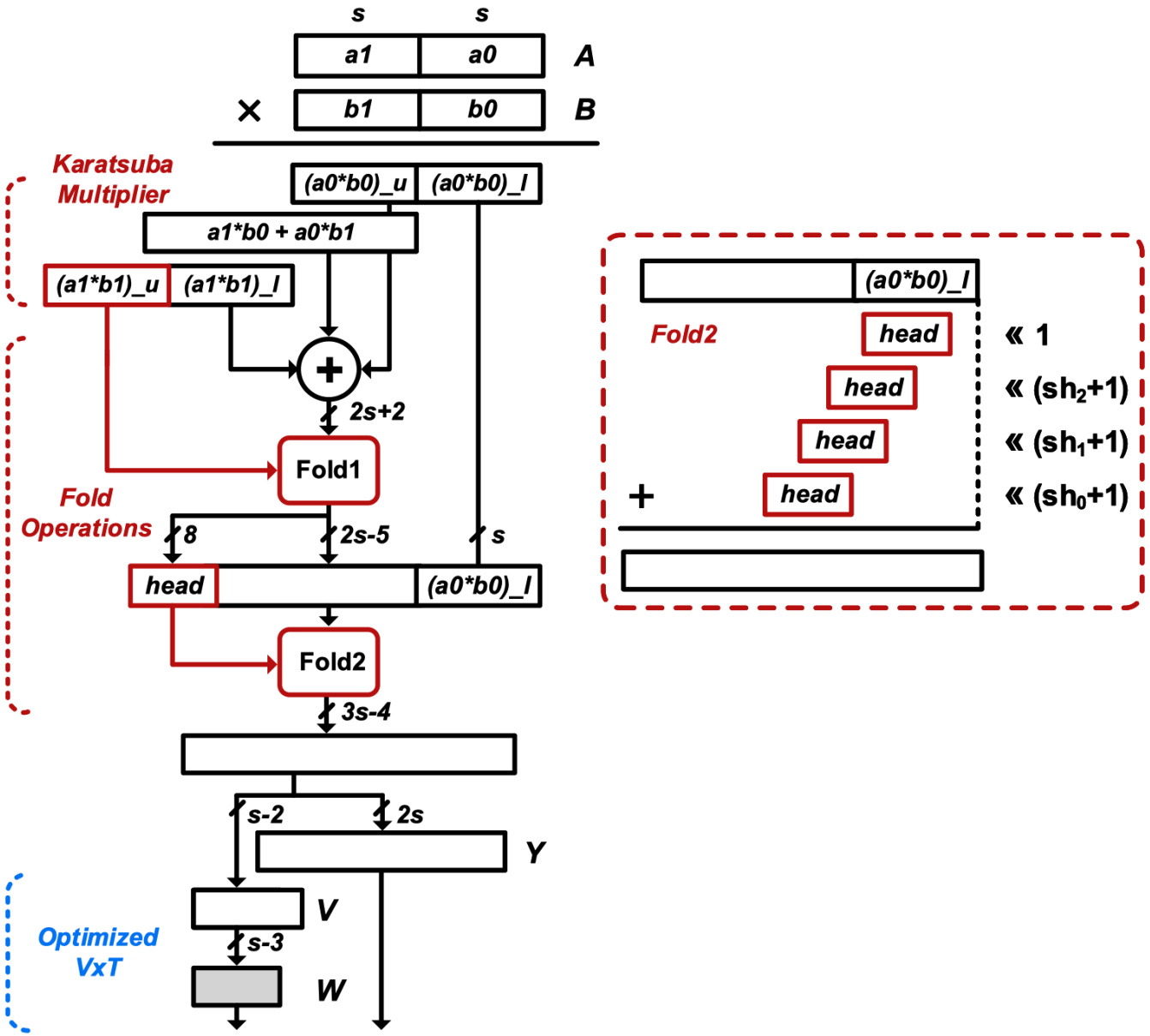
Table 1: Moduli Set.

Index	sh_0	sh_1	sh_2	Index	sh_0	sh_1	sh_2
0	28	-	-	21	27	-21	-18
1	25	19	-	22	27	-23	18
2	26	-21	-	23	-27	23	19
3	-29	20	-	24	27	25	22
4	29	23	-	25	27	26	-19
5	-22	19	18	26	28	20	19
6	-23	21	20	27	28	-22	-18
7	23	22	-19	28	-28	22	20
8	24	-19	-18	29	-28	23	20
9	-24	21	18	30	-28	24	19
10	24	23	-18	31	28	24	23
11						26	24
12						22	-18
13						23	-18
14						23	-20
15	25	24	18	36	29	24	18
16	-26	22	19	37	29	-25	-23
17	26	-22	-21	38	29	26	23
18	26	23	-19	39	29	27	18
19	-26	24	19	40	29	28	26
20	26	25	20	-	-	-	-

41 moduli available
→ *Sufficient for bootstrapping in FHE*

Take the most advantage out of this inherent sparsity!

I. Modular Multiplier for Sparse Moduli Set



$$q = 2^l \pm 2^{sh_0} \pm 2^{sh_1} \pm 2^{sh_2} + 1$$

$$T = \lfloor 2^{2l}/q \rfloor = 2^l \mp 2^{sh_0} \mp 2^{sh_1} \mp 2^{sh_2} - 1$$

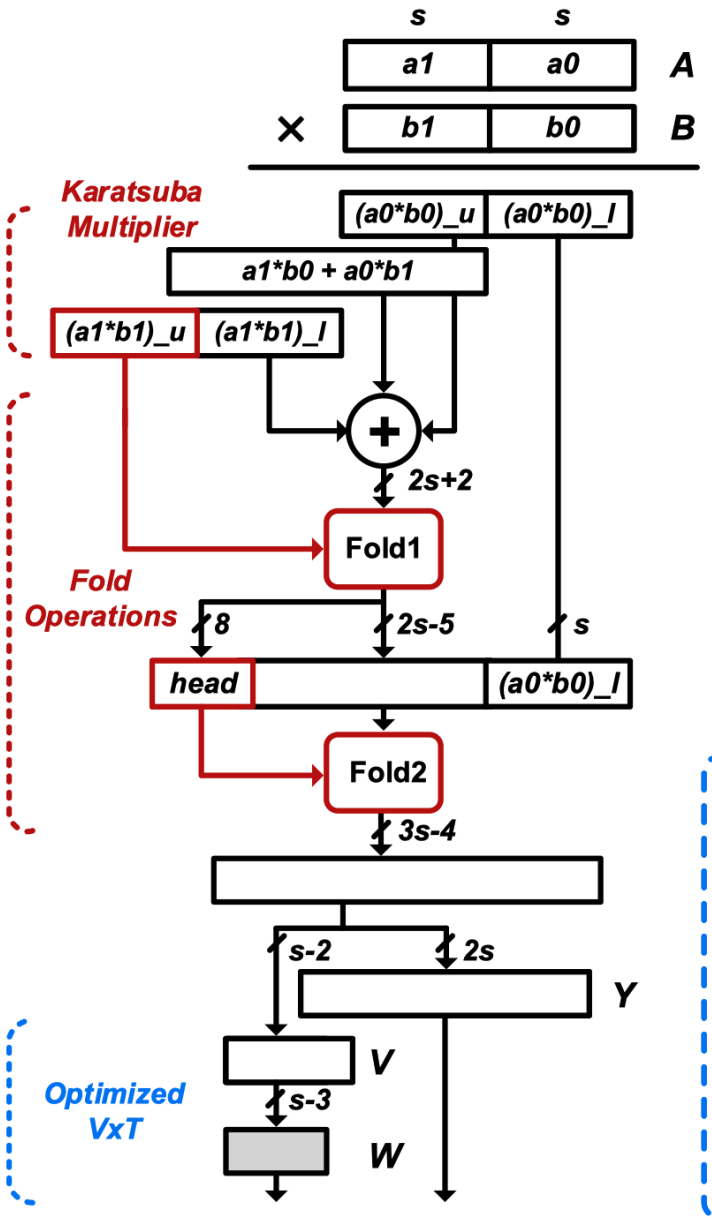
$$2^l = 2^{2s-1} \equiv \mp 2^{sh_0} \mp 2^{sh_1} \mp 2^{sh_2} - 1 \,(mod\ q)$$

$$Fold(X) = [X * 2^{2s}]_q \equiv$$
$$[\mp(X \ll (sh_0 + 1)) \mp (X \ll (sh_1 + 1))$$
$$\mp(X \ll (sh_2 + 1) - (X \ll 1)]_q$$

Simplified computation

- *shift-adding*
- *removing one multiplication*

I. Modular Multiplier for Sparse Moduli Set

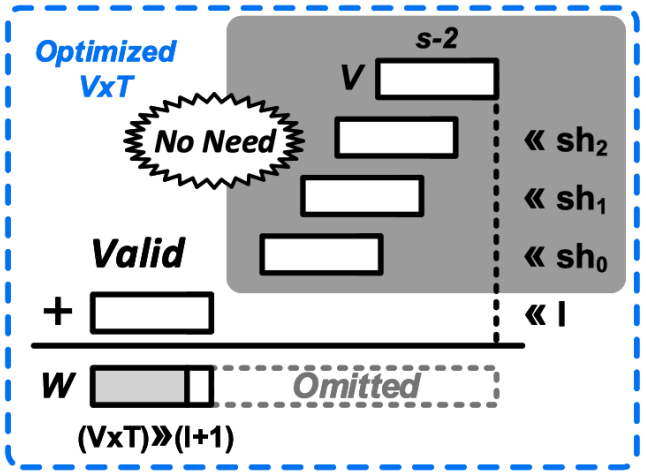


$$W = (V \times T) \gg (l+1)$$

T's sparsity

$$(V-1) \gg 1 \; or \; V \gg 1$$

from sign bit of $\pm 2^{sh_0}$ in q



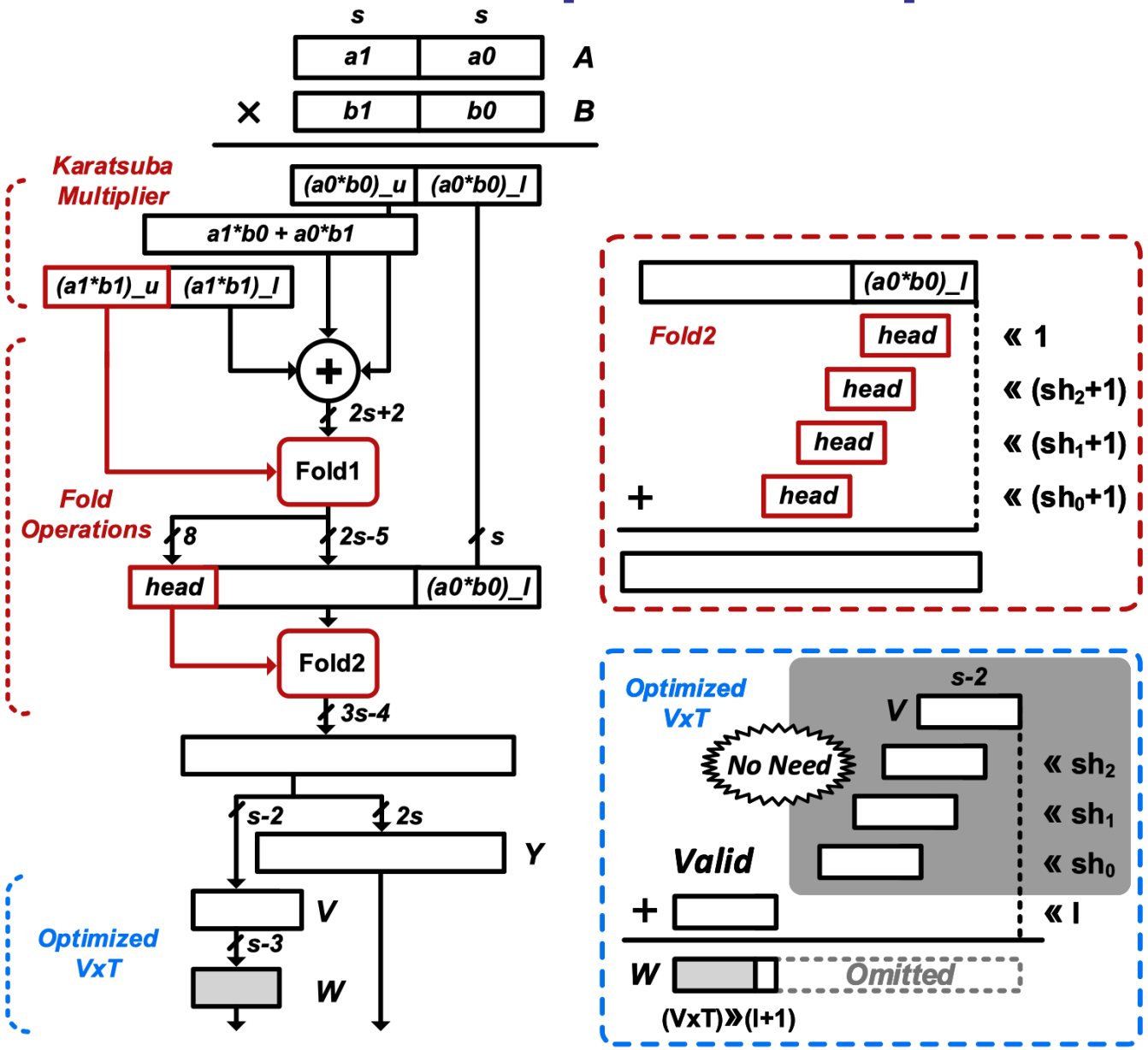
$$q = 2^l \pm 2^{sh_0} \pm 2^{sh_1} \pm 2^{sh_2} + 1$$

$$T = \lfloor 2^{2l}/q \rfloor = 2^l \mp 2^{sh_0} \mp 2^{sh_1} \mp 2^{sh_2} - 1$$

Simplified computation

- *shift-adding*
- *removing one multiplication (V*T)*

I. Modular Multiplier for Sparse Moduli Set



$$q = 2^l \pm 2^{sh_0} \pm 2^{sh_1} \pm 2^{sh_2} + 1$$

$$T = \lfloor 2^{2l}/q \rfloor = 2^l \mp 2^{sh_0} \mp 2^{sh_1} \mp 2^{sh_2} - 1$$

Simplified computation
- *shift-adding*
- *removing one multiplication*

Area (Power)

vs. Non-Sparse:

47.8 (46.0) % ↓

vs. Sparse [1]:

24.6 (22.5) % ↓

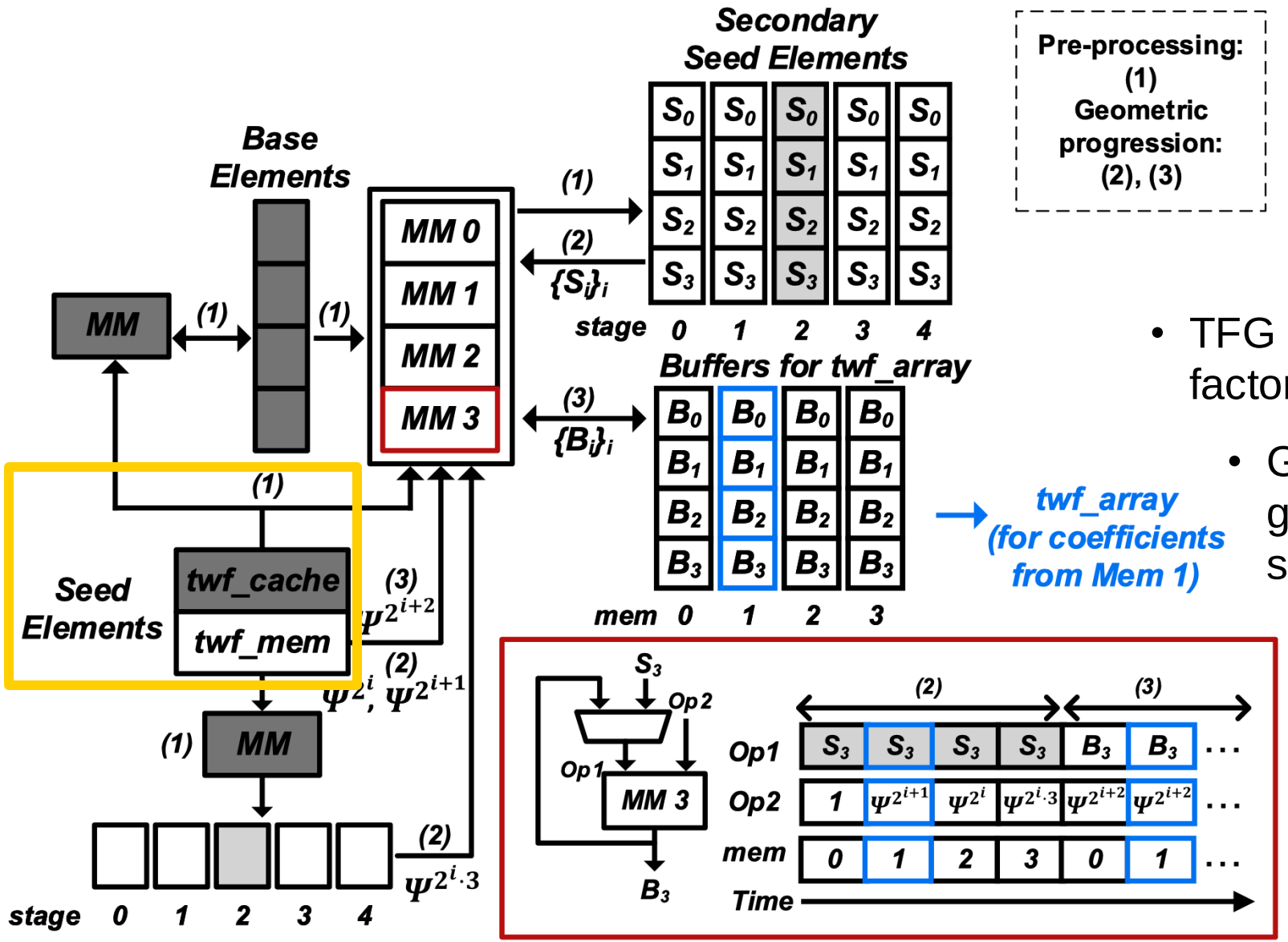
[1] Kim et al., “Fpga-based accelerators of fully pipelined modular multipliers for homomorphic encryption,” ReConFig, 2019.

II-A. Efficient Twiddle Factor Generator (TFG)

$$NTT(\{a_i\}_i) = \left\{\Sigma_{j=0}^{N-1} a_j \boldsymbol{\omega^{ij}} \; mod \; q\right\}_i$$

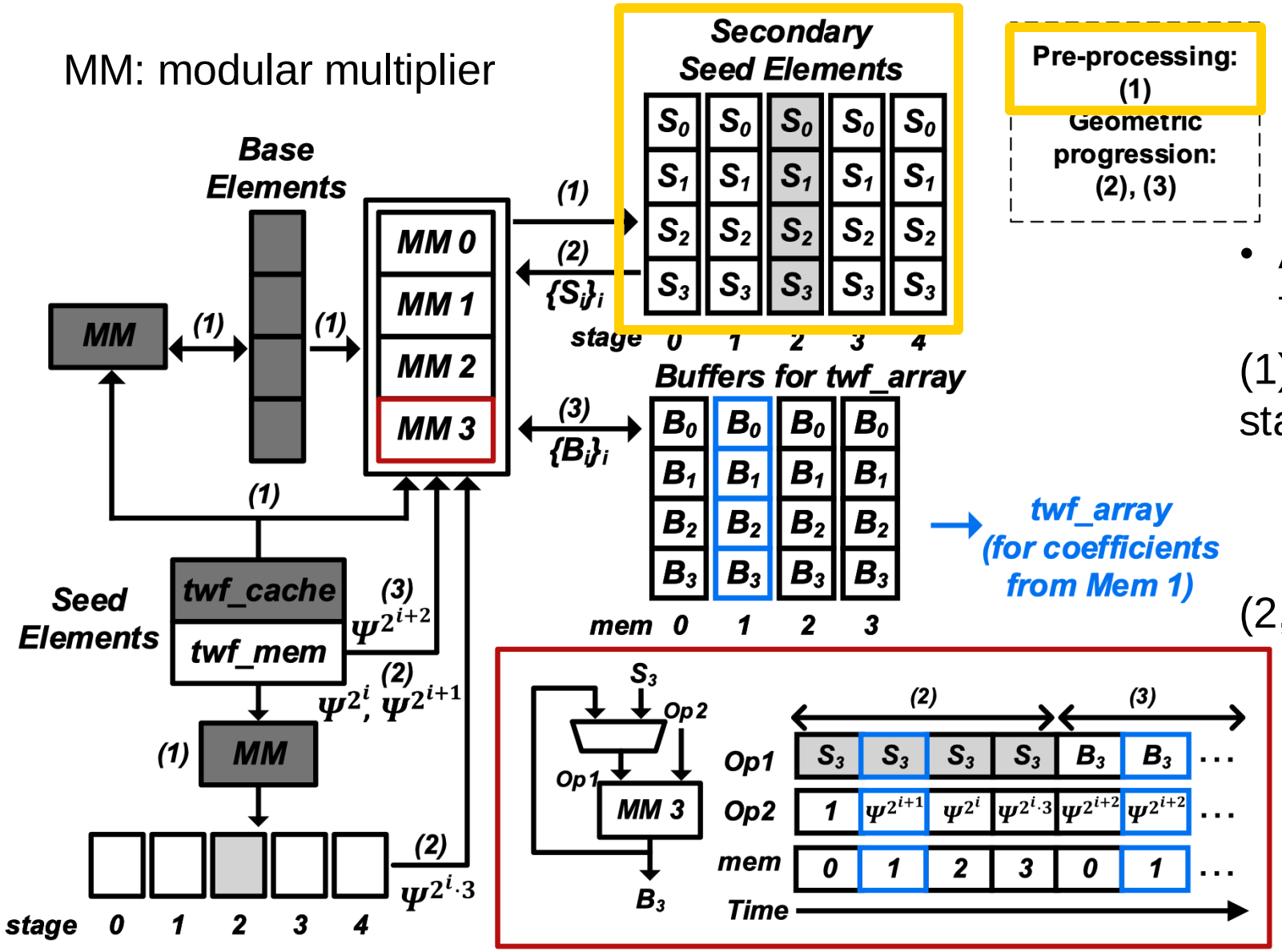
- N twiddle factors required for each NTT/iNTT on N coefficients
  - → Too much overhead in data loading latency and memory area!
- Twiddle Factor Generator
  - Saving memory area for the twiddle factors
  - Generate twiddle factors during NTT/iNTT operations

II-A. Efficient Twiddle Factor Generator (TFG)



- TFG saving memory area for the twiddle factors from $O(N) \rightarrow O(\log N)$
  - Generate twiddle factors using geometric progression with $O(\log N)$ seed elements

II-A. Efficient Twiddle Factor Generator (TFG)



- Additional pre-processing stage for further reduction of seed elements

(1) **Pre-processing** (before NTT/iNTT starts)

Input: Seed elements

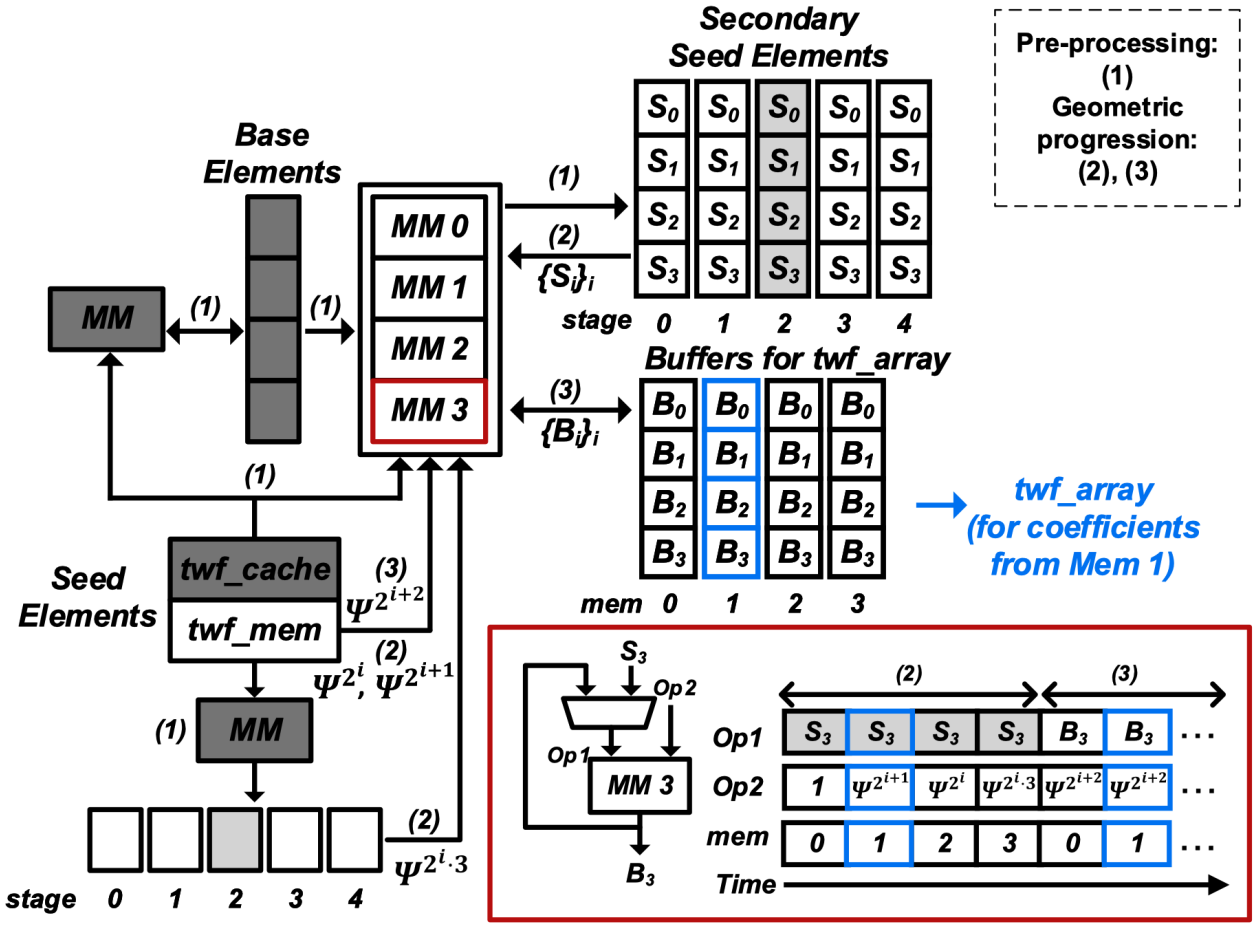
Output: Secondary seed elements

(2,3) Geometric progression (run-time)

Input: Secondary seed elements

Output: Twiddle factors

II-A. Efficient Twiddle Factor Generator (TFG)



- Reduction of twiddle factor memory area $O(N) \rightarrow O(\log N)$
- Additional pre-processing stage for further reduction on seed elements

Table 2: TFG comparison.

	[2]	[6]	[17]	Ours
Modular multipliers	$\sqrt{N}$	$N-2$	$2k-1$	$2k+1$
Area reduction*	**1.62-17.13×**	**23.89-5988.84×**	**0.91×**	
Seed elements ($\log N = 9$)	22.63	1	184	18
Reduction*	**1.26-10.65×**	**0.03-0.06×**	**5.65-10.22×**	

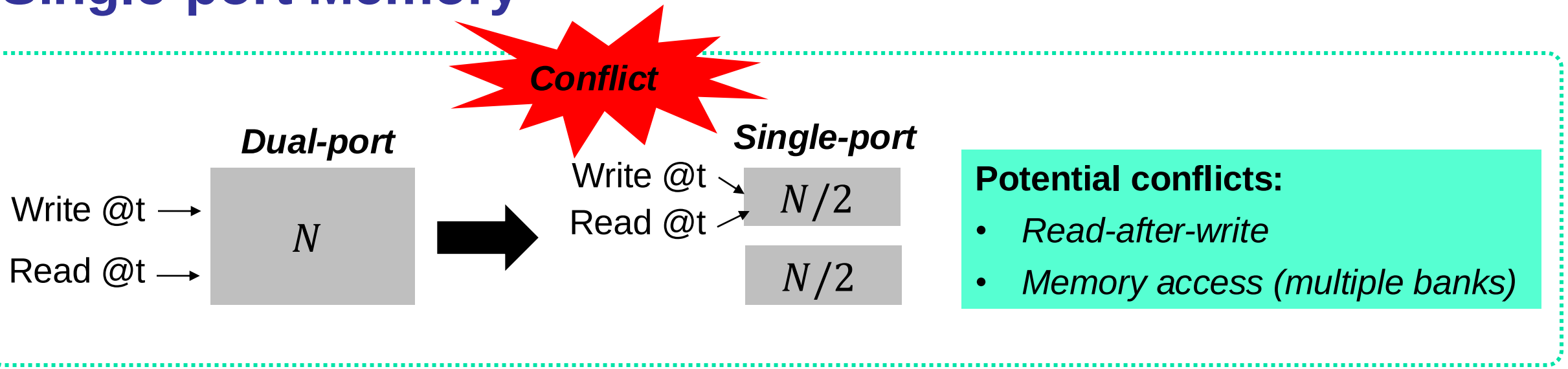
*Reduction for $\log N = 9 - 17$

[2] Kim et al., "Ark: Fully homomorphic encryption accelerator with run-time data generation and inter-operation key reuse," MICRO, 2022.

[6] Geelen et al., "Basalisc: Flexible asynchronous hardware accelerator for fully homomorphic encryption," preprint, arXiv, 2022.

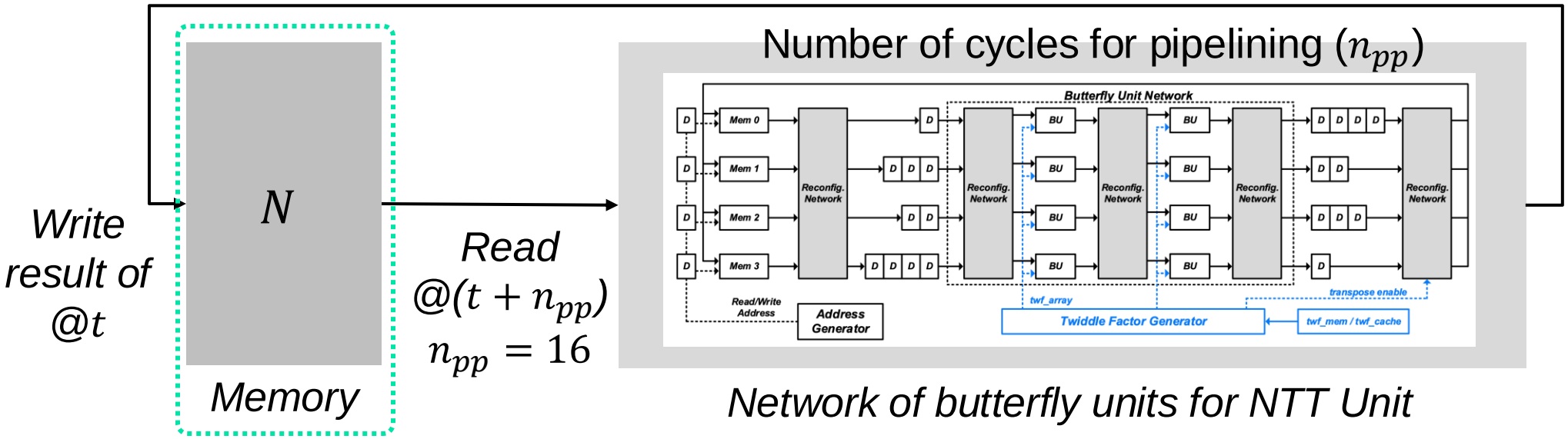
[17] Kim et al., "Hardware architecture of a number theoretic transform for a bootstrappable rns-based homomorphic encryption scheme," FCCM, 2020.

II-B. Conflict-free Addressing (CFA) Scheme for Single-port Memory



Potential conflicts:

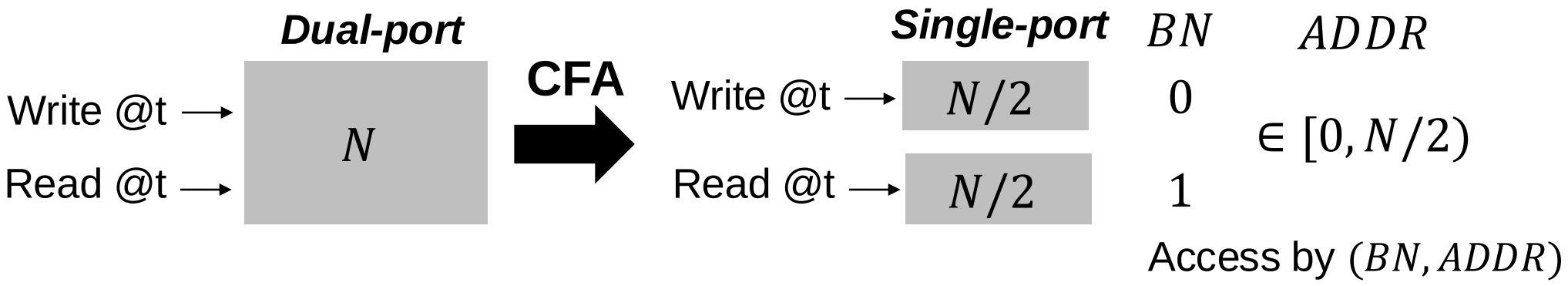
- *Read-after-write*
- *Memory access (multiple banks)*

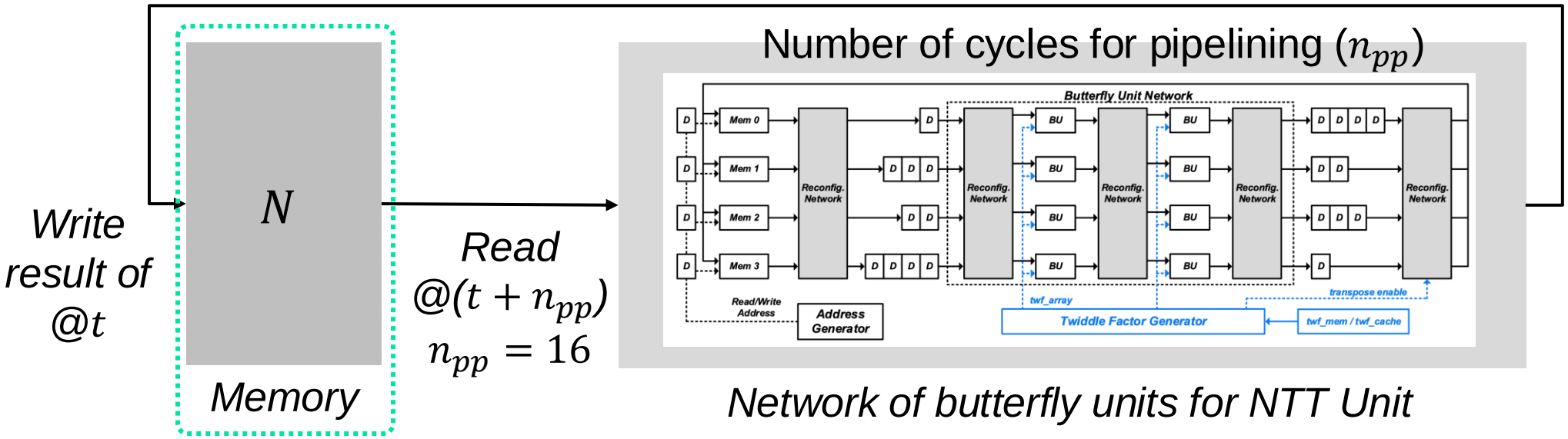


Network of butterfly units for NTT Unit

II-B. Conflict-free Addressing (CFA) Scheme for Single-port Memory

No throughput degradation, Reduce silicon area & power





Network of butterfly units for NTT Unit

II-B. Conflict-free Addressing (CFA) Scheme for Single-port Memory

Algorithm 1 Conflict-free address generation scheme

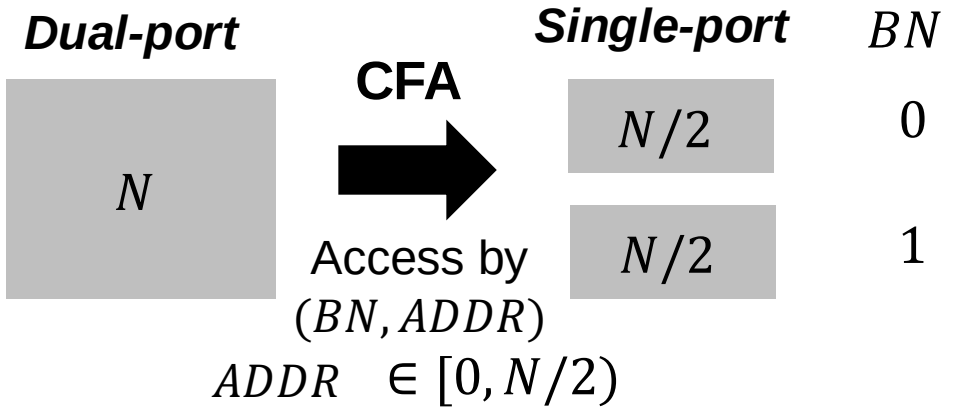
Note $\ll_r$ is a left-rotate operator. $GR = BR \circ Gray \circ BR$ where BR and $Gray$ are a bit-reverse and gray function, respectively.
Input $s,\ addr,\ bn$
Output $BN,\ ADDR$

```
1:  for s = 0; s < 6; s = s + 2 do
2:      for addr = 0; addr < 16; addr = addr + 1 do
3:          for bn = 0; bn < 4; bn = bn + 1 do
4:              tmp ← {BR(addr), BR(bn)}
5:              tmp ← {GR(tmp[5 : 4]), GR(tmp[3 : 2]), GR(tmp[1 : 0])}
6:              orig_addr ← tmp ≪_r s
7:              BN ←^orig_addr
8:              ADDR ← orig_addr[5 : 1]
9:          end for
10:     end for
11: end for
```

Goal: No conflict!

$$BN(t) \oplus BN(t+16) = 1, \forall t$$

Network pipelining stage (n_{pp})



II-B. Conflict-free Addressing (CFA) Scheme for Single-port Memory

Algorithm 1 Conflict-free address generation scheme

Note $\ll_r$ is a left-rotate operator. $GR = BR \circ Gray \circ BR$ where BR and $Gray$ are a bit-reverse and gray function, respectively.
Input $s,\ addr,\ bn$
Output $BN,\ ADDR$

1: **for** $s = 0; s < 6; s = s + 2$ **do**
2: **for** $addr = 0; addr < 16; addr = addr + 1$ **do**
3: **for** $bn = 0; bn < 4; bn = bn + 1$ **do**
4: $tmp \leftarrow \{BR(addr), BR(bn)\}$
5: $tmp \leftarrow \{GR(tmp[5:4]), GR(tmp[3:2]), GR(tmp[1:0])\}$
6: $orig\_addr \leftarrow tmp \ll_r s$
7: $BN \leftarrow \hat{}orig\_addr$
8: $ADDR \leftarrow orig\_addr[5:1]$
9: **end for**
10: **end for**
11: **end for**

Goal: No conflict!

$$BN(t) \oplus BN(t+16) = 1, \forall t$$

Network pipelining stage (n_{pp})

Then, should satisfy:

$$\hat{}addr[3:2](t) \oplus \hat{}addr[3:2](t+16) = \hat{}addr[3:2](t) \oplus \hat{}\ (addr[3:2](t)+1) = 1$$

However, this is not satisfied for all t…

II-B. Conflict-free Addressing (CFA) Scheme for Single-port Memory

Algorithm 1 Conflict-free address generation scheme

Note $\ll_r$ is a left-rotate operator. $GR = BR \circ Gray \circ BR$ where BR and $Gray$ are a bit-reverse and gray function, respectively.
Input $s,\ addr,\ bn$
Output $BN,\ ADDR$

1: **for** $s = 0; s < 6; s = s + 2$ **do**
2: **for** $addr = 0; addr < 16; addr = addr + 1$ **do**
3: **for** $bn = 0; bn < 4; bn = bn + 1$ **do**
4: $tmp \leftarrow \{BR(addr), BR(bn)\}$
5: $tmp \leftarrow \{GR(tmp[5:4]), GR(tmp[3:2]), GR(tmp[1:0])\}$
6: $orig\_addr \leftarrow tmp \ll_r s$
7: $BN \leftarrow \hat{}orig\_addr$
8: $ADDR \leftarrow orig\_addr[5:1]$
9: **end for**
10: **end for**
11: **end for**

Goal: No conflict!

$$BN(t) \oplus BN(t+16) = 1, \forall t$$

Network pipelining stage (n_{pp})

$$\hat{}addr[3:2](t) \oplus \hat{}addr[3:2](t+16)$$
$$= \hat{}addr[3:2](t) \oplus \hat{}\,(addr[3:2](t)+1) = 1$$

Applying Bit-reverse + Gray

$$\hat{}GR\left(BR\left(addr[3:2](t)\right)\right) \oplus \hat{}GR\left(BR\left(addr[3:2](t+16)\right)\right)$$
$$= \hat{}Gray\left(addr[3:2](t)\right) \oplus \hat{}Gray(addr[3:2](t)+1) = 1$$

Now, this is satisfied for all t!

II-B. Conflict-free Addressing (CFA) Scheme for Single-port Memory

Algorithm 1 Conflict-free address generation scheme

Note $\ll_r$ is a left-rotate operator. $GR = BR \circ Gray \circ BR$ where BR and $Gray$ are a bit-reverse and gray function, respectively.
Input $s,\ addr,\ bn$
Output $BN,\ ADDR$

1: **for** $s = 0; s < 6; s = s + 2$ **do**
2: **for** $addr = 0; addr < 16; addr = addr + 1$ **do**
3: **for** $bn = 0; bn < 4; bn = bn + 1$ **do**
4: $tmp \leftarrow \{BR(addr), BR(bn)\}$
5: $tmp \leftarrow \{GR(tmp[5:4]), GR(tmp[3:2]), GR(tmp[1:0])\}$
6: $orig\_addr \leftarrow tmp \ll_r s$
7: $BN \leftarrow$^$orig\_addr$
8: $ADDR \leftarrow orig\_addr[5:1]$
9: **end for**
10: **end for**
11: **end for**

Goal: No conflict!

$$BN(t) \oplus BN(t+16) = 1, \forall t$$

Memory access conflict most likely occur during stage transition

Example of Stage 0 → Stage 1 in INTT

$s = 0, addr = 12, \ldots, 15 \quad s = 2, addr = 0, \ldots, 3$

II-B. Conflict-free Addressing (CFA) Scheme for Single-port Memory

Memory access conflict-free

$$BN(t) \oplus BN(t+16) = 1, \forall t$$

Algorithm 1 Conflict-free address generation scheme

Note $\ll_r$ is a left-rotate operator. $GR = BR \circ Gray \circ BR$ where BR and $Gray$ are a bit-reverse and gray function, respectively.

Input $s,\ addr,\ bn$

Output $BN,\ ADDR$

1: **for** $s = 0; s < 6; s = s + 2$ **do**
2: **for** $addr = 0; addr < 16; addr = addr + 1$ **do**
3: **for** $bn = 0; bn < 4; bn = bn + 1$ **do**
4: $tmp \leftarrow \{BR(addr), BR(bn)\}$ $\{ad, dr, bn\}$
5: $tmp \leftarrow \{GR(tmp[5:4]), GR(tmp[3:2]), GR(tmp[1:0])\}$
6: $orig\_addr \leftarrow tmp \ll_r s$
7: $BN \leftarrow$^$orig\_addr$
8: $ADDR \leftarrow orig\_addr[5:1]$
9: **end for**
10: **end for**
11: **end for**

Same throughput

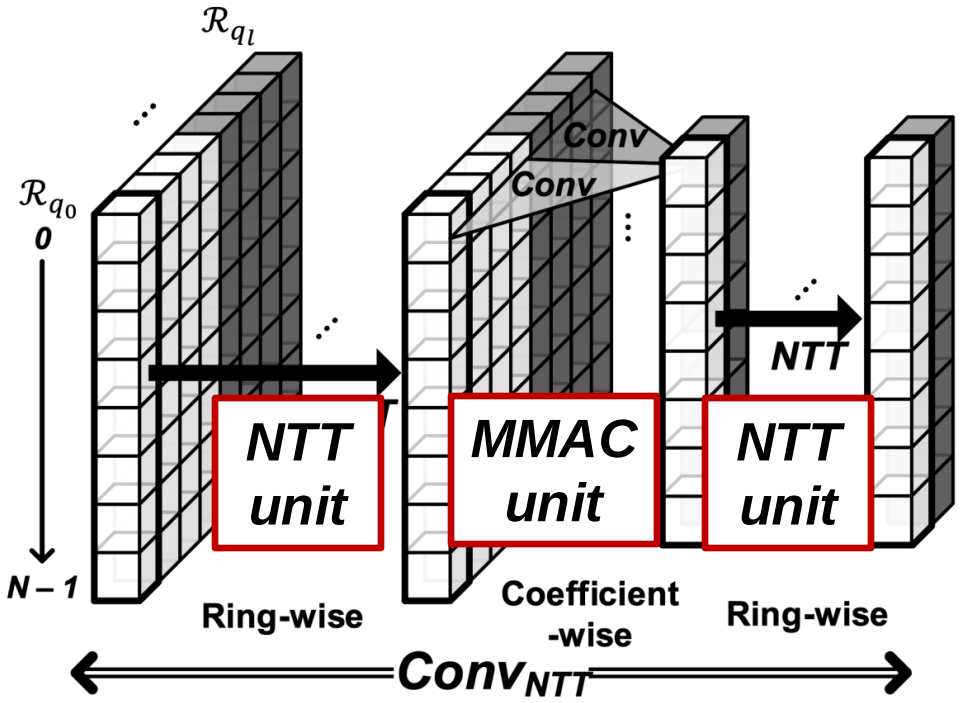
Area (Power) 67.87 (44.39) % ↓

Example of Stage 0 → Stage 1 in INTT

Time	ad	dr	bn	BN	Time	dr	bn	ad	BN
$t_{stg,0} - 16$	00	01	00	1	$t_{stg,0}$	00	00	00	0
$t_{stg,0} - 15$	00	01	10	0	$t_{stg,0} + 1$	00	10	00	1
$t_{stg,0} - 14$	00	01	11	1	$t_{stg,0} + 2$	00	11	00	0
$t_{stg,0} - 13$	00	01	01	0	$t_{stg,0} + 3$	00	01	00	1
$t_{stg,0} - 12$	10	01	00	0	$t_{stg,0} + 4$	00	00	10	1
$t_{stg,0} - 11$	10	01	10	1	$t_{stg,0} + 5$	00	10	10	0
$t_{stg,0} - 10$	10	01	11	0	$t_{stg,0} + 6$	00	11	10	1
$t_{stg,0} - 9$	10	01	01	1	$t_{stg,0} + 7$	00	01	10	0
$t_{stg,0} - 8$	11	01	00	1	$t_{stg,0} + 8$	00	00	11	0
$t_{stg,0} - 7$	11	01	10	0	$t_{stg,0} + 9$	00	10	11	1
$t_{stg,0} - 6$	11	01	11	1	$t_{stg,0} + 10$	00	11	11	0
$t_{stg,0} - 5$	11	01	01	0	$t_{stg,0} + 11$	00	01	11	1
$t_{stg,0} - 4$	01	01	00	0	$t_{stg,0} + 12$	00	00	01	1
$t_{stg,0} - 3$	01	01	10	1	$t_{stg,0} + 13$	00	10	01	0
$t_{stg,0} - 2$	01	01	11	0	$t_{stg,0} + 14$	00	11	01	1
$t_{stg,0} - 1$	01	01	01	1	$t_{stg,0} + 15$	00	01	01	0
Stage 0					Stage 1				

$s = 0, addr = 12, \ldots, 15$ $s = 2, addr = 0, \ldots, 3$

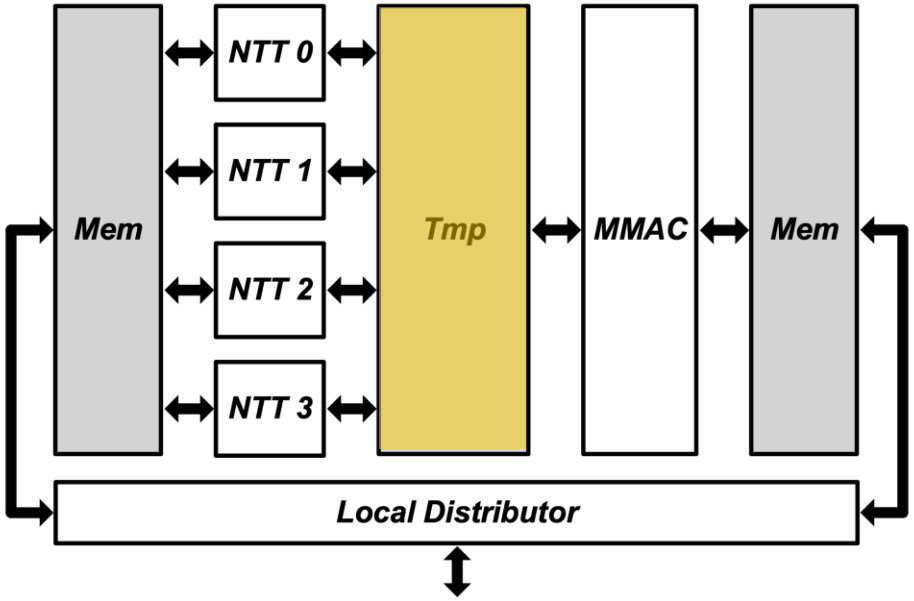
III. Bandwidth-efficient Behavior in Core



Frequent data access pattern transition between NTT and MMAC unit

➔ *Expensive external memory access!*

III. Bandwidth-efficient Behavior in Core



Phase	Operation	Data Transfer
1	IN0	NTTs ↔ Mem
2	IN1	Mem → NTTs → Tmp
	Conv	MMAC ↔ Tmp
	N0	Tmp → NTTs → Mem
3	N1	Mem → NTTs → Tmp
	MAC	Tmp, Mem → MMAC → Mem
4	N1	Mem → NTTs → Tmp
	SM	Tmp, Mem → MMAC → Mem

Dataflow in KS modified for better data utilization, using dedicated buffers (Tmp) between NTT and MMAC units

III. Bandwidth-efficient Behavior in Core

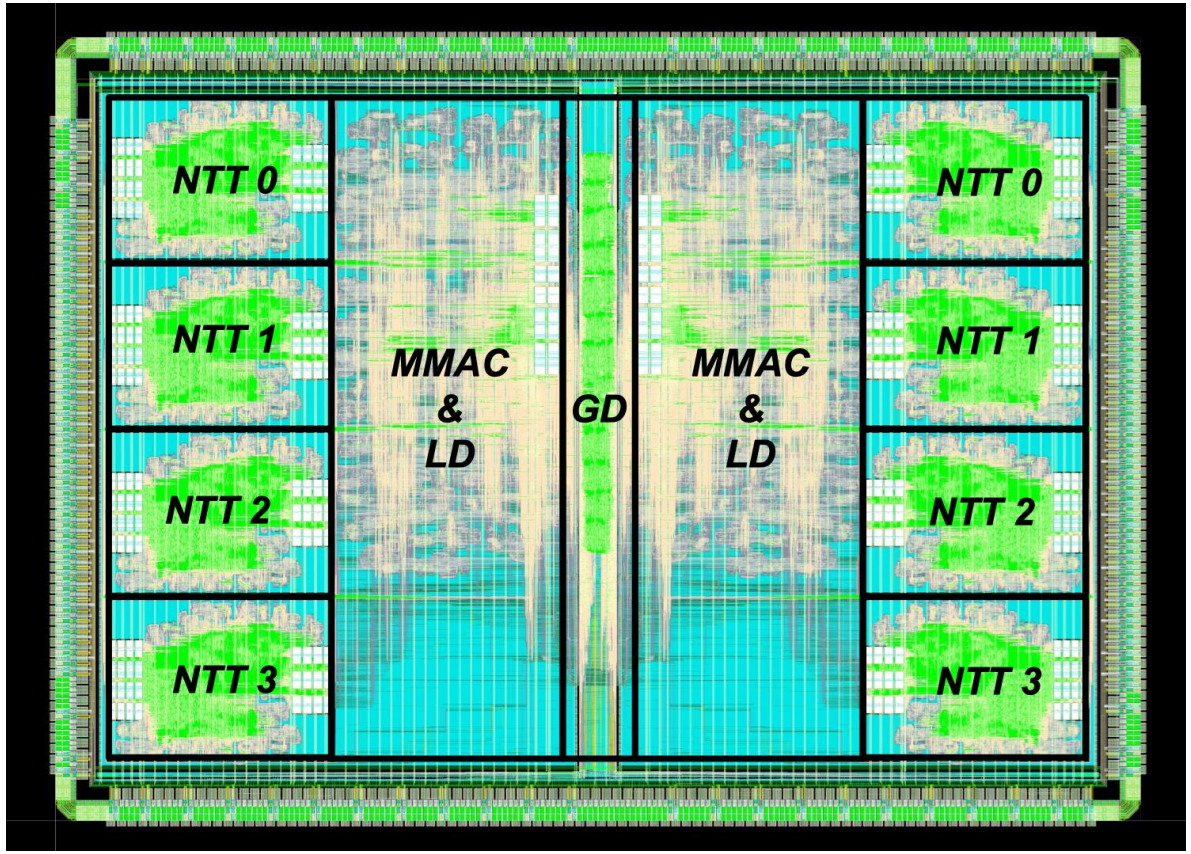
Part	Usage	Naïve		Ours	
		In	Out	In	Out
1	INTT	$N{\cdot}2{\cdot}(L{+}1)$	$N{\cdot}2{\cdot}(L{+}1)$	$N{\cdot}2{\cdot}(L{+}1)$	$N\cdot(L+1)$
	Conv	$N\cdot\alpha\cdot dnum$	$N\cdot dnum$	0	0
	NTT	$N{\cdot}2{\cdot}dnum$	$N\cdot 2\cdot(L+1)\cdot dnum$	$N\cdot dnum$	$N\cdot dnum\cdot(L+1)$
2	(NTT)	$N\cdot dnum\cdot(L+1+\alpha)$	-	0	-
	Key	$2N{\cdot}dnum{\cdot}(L+1+\alpha)$	-	$2N{\cdot}dnum{\cdot}(L+1+\alpha)$	-
	Cxt	0	$2N\cdot(L+1+\alpha)$	0	$2N(L+1+\alpha)$
3	INTT	$2N\cdot 2\cdot\alpha$	$2N\cdot 2\cdot\alpha$	$2N\cdot 2\cdot\alpha$	$2N\cdot\alpha$
	Conv	$2N\cdot\alpha$	$2N$	0	0
	NTT	$2N\cdot 2$	$2N\cdot 2\cdot(L+1)$	$2N$	$2N(L+1)$
4	(NTT)	$2N(L+1)$	-	0	-
	Cxt	$2N(L+1)$	$2N(L+1)$	$2N(L+1)$	$2N(L+1)$

Phase	Operation	Data Transfer
1	IN0	NTTs ↔ Mem
2	IN1	Mem → NTTs → Tmp
	Conv	MMAC ↔ Tmp
	N0	Tmp → NTTs → Mem
3	N1	Mem → NTTs → Tmp
	MAC	Tmp, Mem → MMAC → Mem
4	N1	Mem → NTTs → Tmp
	SM	Tmp, Mem → MMAC → Mem

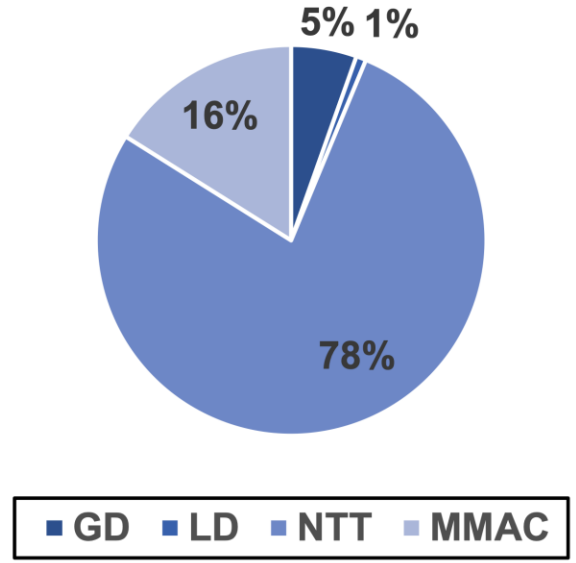
Dataflow in KS modified for better data utilization, using dedicated buffers (Tmp) between NTT and MMAC units

➔ ***External memory access 38.7 % ↓***

Chip Implementation



Area (mm^2)	23.48 (5.70×4.12)
Memory voltage (V)	0.9
Memory size (kB)	240
Core voltage (V)	0.8
Core frequency (MHz)	200
Avg. power (mW)	54.7
Throughput (KS/s)	10.58



GD (LD): Global/Local Distributor

Comparison with Prior Works

Table 3: Comparisons with Prior ASIC NTT Accelerators.

	HOST'19 [22]	CICC'18 [23]	TVLSI'22 [24]	ISSCC'19 [25]	Ours
Technology	65nm	40nm	65nm	40nm	28nm
$(N, \lceil \log q \rceil)$	(512, 14)	(512, 14)	(512, 14)	(256, 13)	(512, 60)
Energy efficiency (Gbit/W/s)	30.308	106.009	51.358	74.989	431.577
Improvement	**14.24×**	**4.07×**	**8.40×**	**5.755×**	
Area efficiency (bit/um^2)	635.846	6371.329	4126.984	1237.580	61033.820
Improvement	**95.99×**	**9.58×**	**14.79×**	**49.32×**	

Table 4: Comparison of Energy Efficiency for KS Operation.

	Platform	$\log N$	E_{norm} (J)	Improvement
HEAX [9]	FPGA	14	0.0082	**12.66×**
BASALIC [6]	ASIC	16	0.0784	**30.32×**
100x [3]	GPU	17	0.3129	**60.52×**
Lattigo [16]	CPU	17	67.7411	**13104.87×**
Ours	ASIC	17	0.0052	

Conclusion

- Designed a dedicated accelerator for KS that requires frequent transitions in data access patterns incurring redundant expensive external memory accesses
- Proposed design techniques on various levels for high energy efficiency – modular multiplier, NTT unit, and data access behavior in core, thus full-stack optimization
- As a result, the design shows significant improvement in performance in energy efficiency compared with prior FHE implementations.
- Although designed specifically for KS, it remains highly applicable since KS operations dominate power, time, and bandwidth across the entire computation.
- Techniques can be also applied to other parameter sets – modular multiplier as long as same moduli set used and $L + \alpha + 1 \leq 41$, NTT unit down to $N = 2^{16}$ (the least amount to support bootstrapping reported in literature)