WHERE **INNOVATION** BEGINS

DESIGN AUTOMATION CONFERENCE

JULY 9–13, 2023

MOSCONE WEST CENTER
SAN FRANCISCO, CA, USA

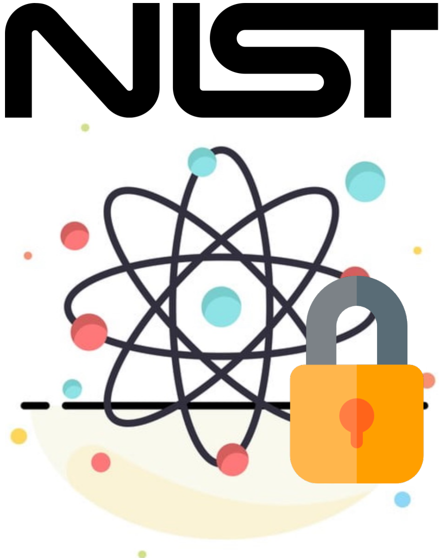
***BP-NTT**: Fast and Compact in-SRAM Number Theoretic Transform with Bit-Parallel Modular Multiplication*

Jingyao Zhang[*], Mohsen Imani[†], Elaheh Sadredini[*]

[*] UC RIVERSIDE [†] UCI University of California, Irvine

DAC 60 WHERE INNOVATION BEGINS DESIGN AUTOMATION CONFERENCE

Lattice-based Cryptography is the Future



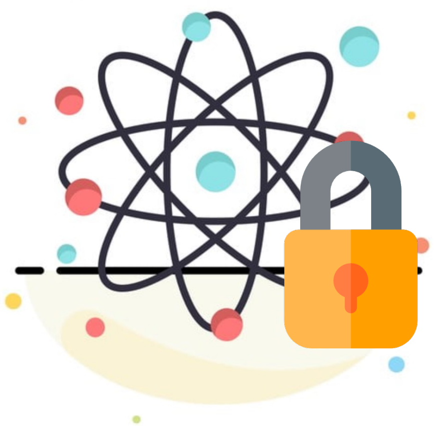
Post-quantum Cryptography

Homomorphic Encryption

Lattice-based Cryptography is the Future

- Polynomial Multiplication is the bedrock

NIST



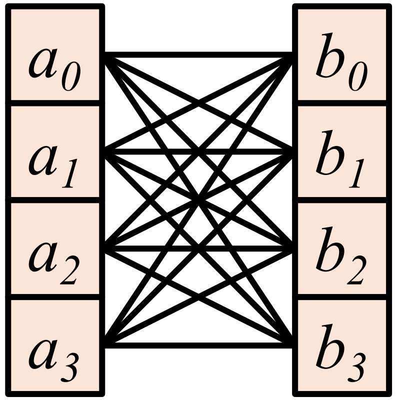
Post-quantum Cryptography

Homomorphic Encryption

Polynomial Multiplication

Polynomial Multiplication is the Bedrock

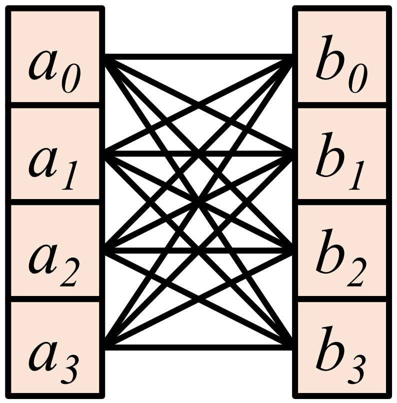
Polynomial Multiplication is the Bedrock



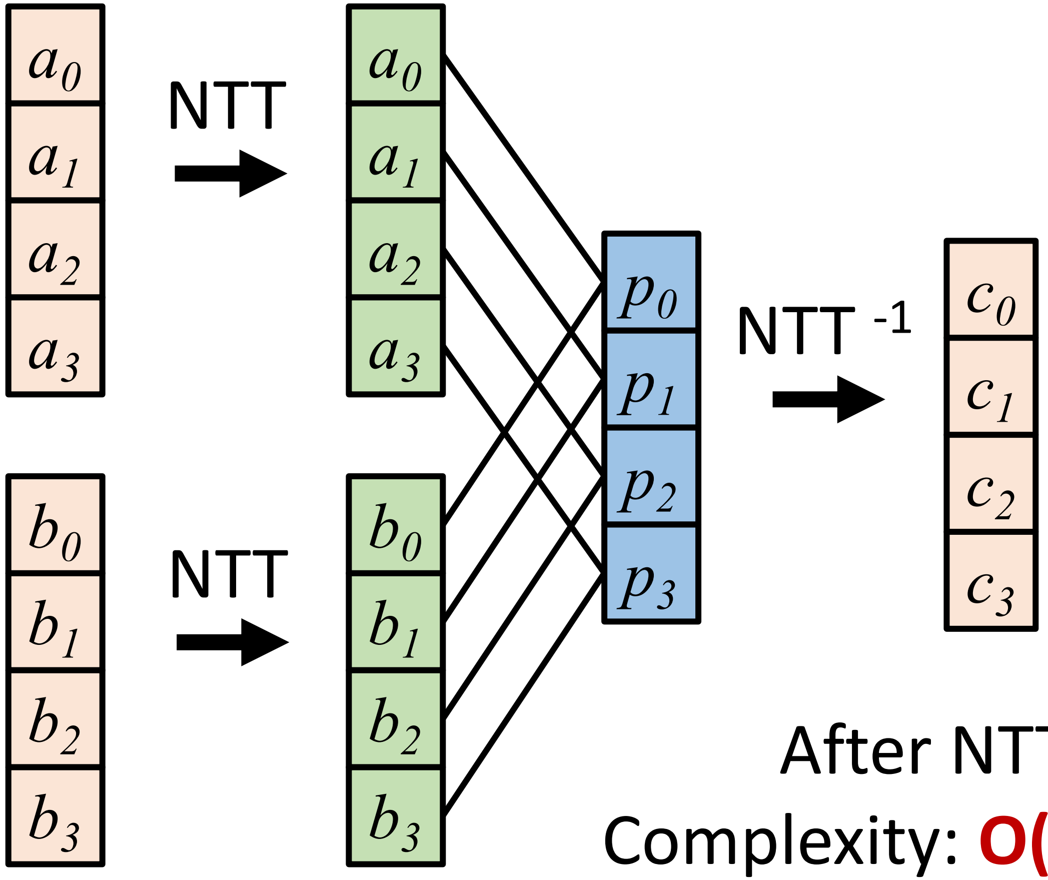
Polynomial Multiplication
Complexity: **$O(n^2)$**

Polynomial Multiplication is the Bedrock

- Number Theoretic Transform (NTT) is necessary



Polynomial Multiplication
Complexity: **O(n²)**



After NTT
Complexity: **O(nlogn)**

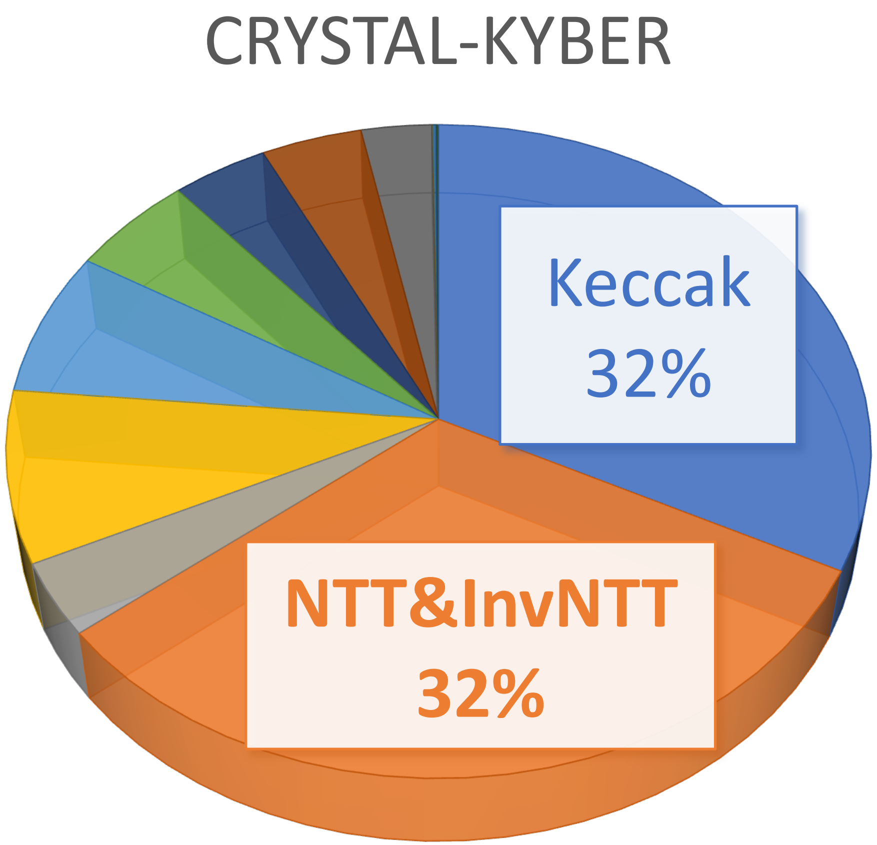
NTT Acceleration is Essential

NTT Acceleration is Essential

- From profiling, 32~50% of execution time is spent on NTT/InvNTT

NTT Acceleration is Essential

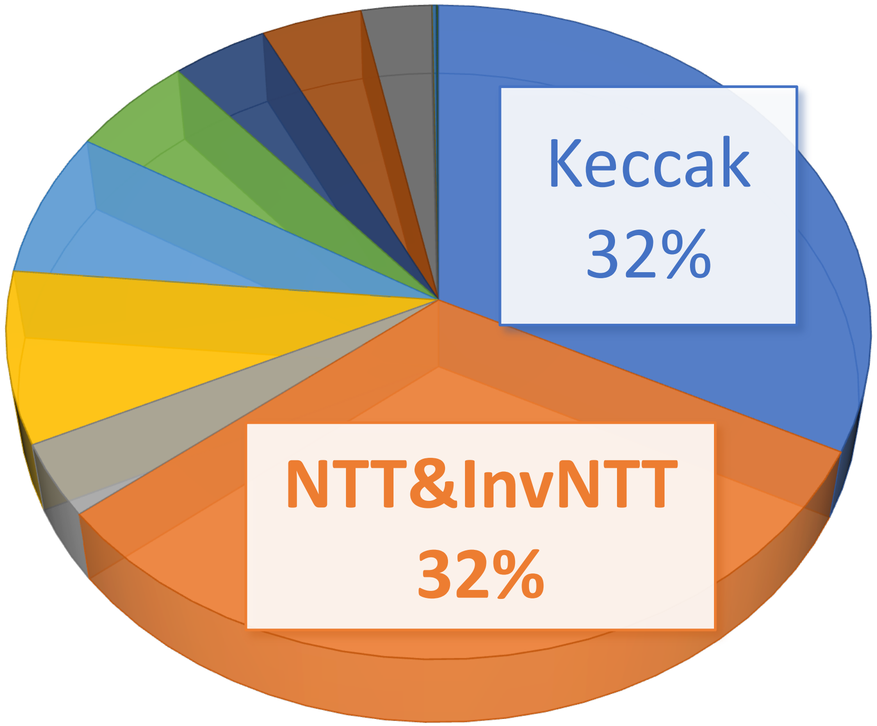
- From profiling, 32~50% of execution time is spent on NTT/InvNTT



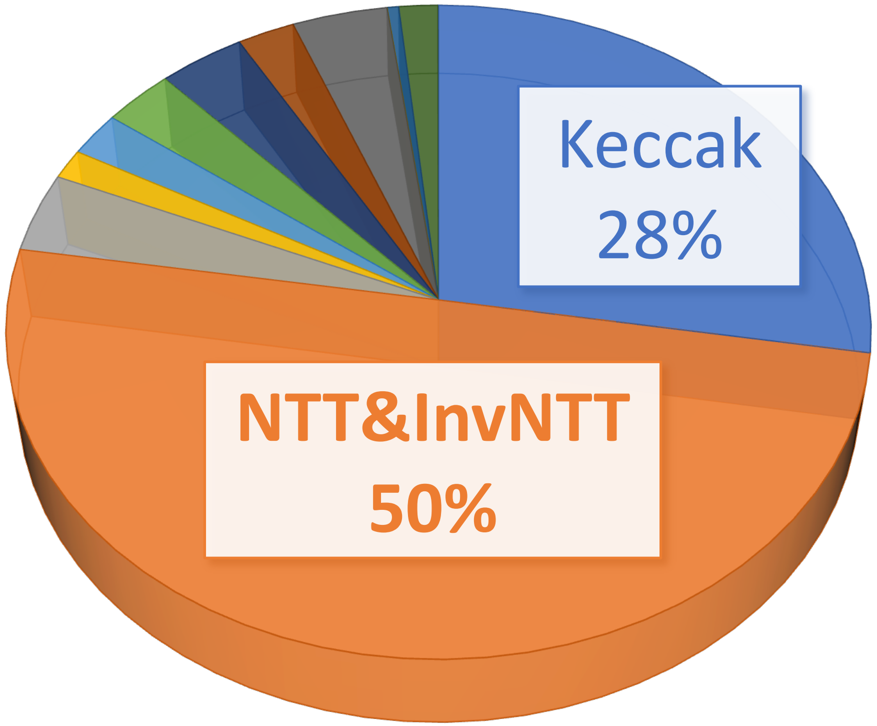
NTT Acceleration is Essential

- From profiling, 32~50% of execution time is spent on NTT/InvNTT

CRYSTAL-KYBER

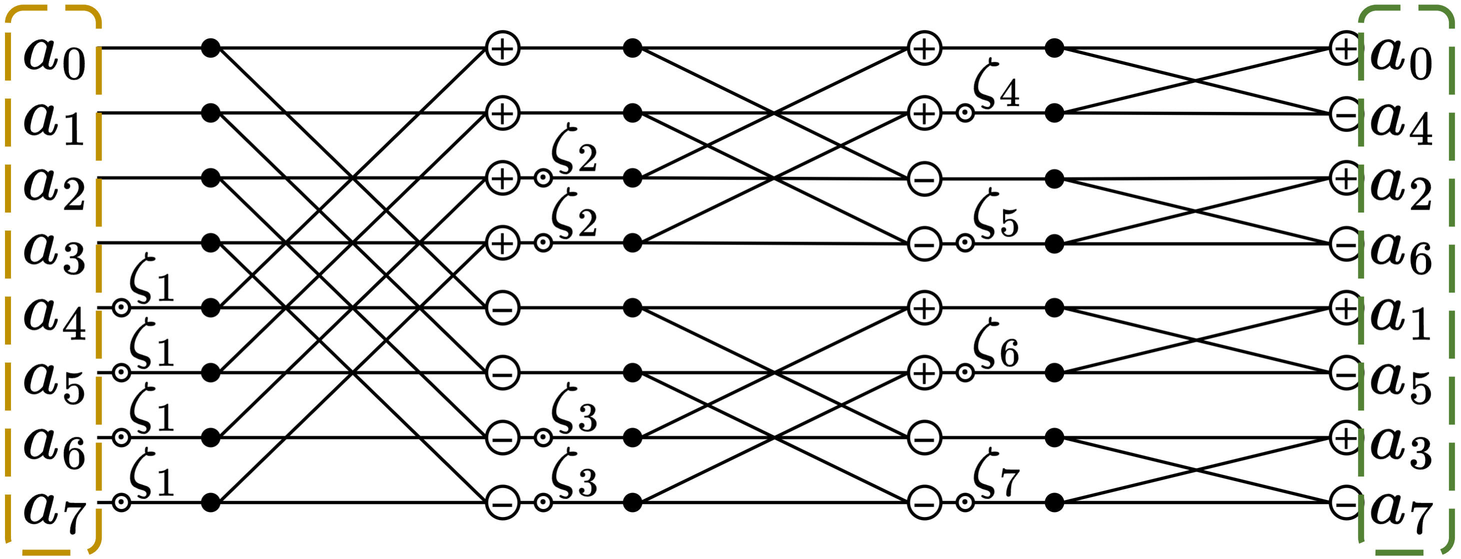


CRYSTAL-DILITHIUM



NTT Acceleration is Hard!

Polynomial coefficients before NTT



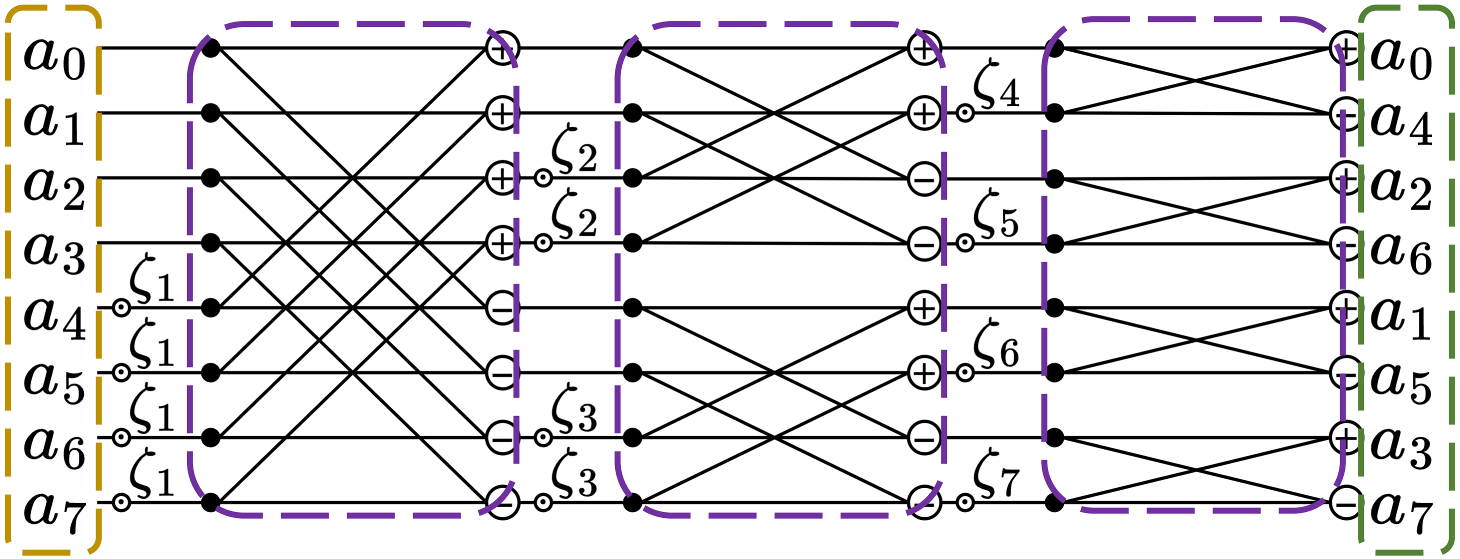
Polynomial coefficients after NTT

3-stage Cooley-Tukey butterfly

NTT Acceleration is Hard!

- Complicated data dependencies

Polynomial coefficients before NTT

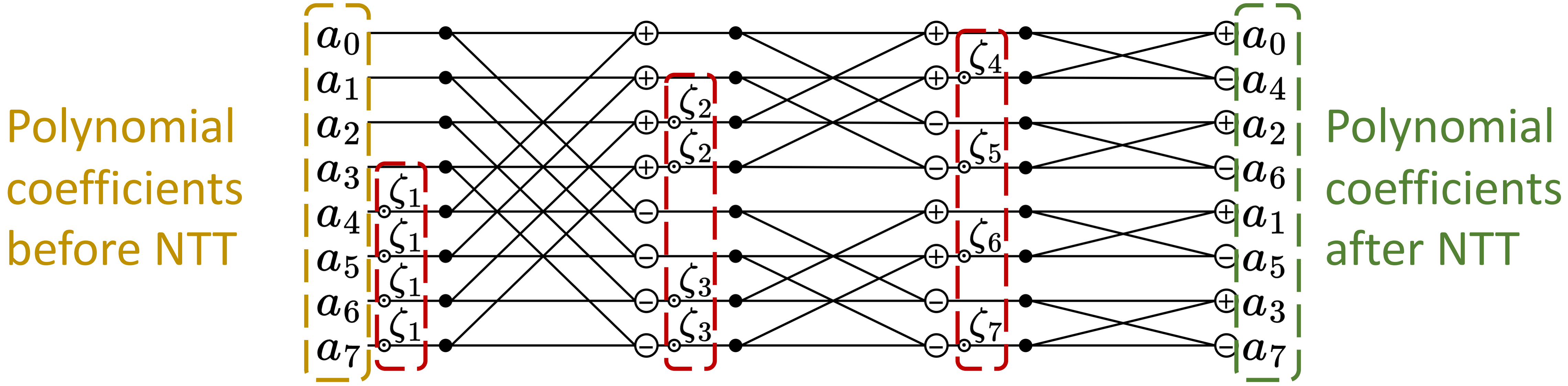


Polynomial coefficients after NTT

3-stage Cooley-Tukey butterfly

NTT Acceleration is Hard!

- Complicated data dependencies
- Heavy multiplication with division-based modulo operation



3-stage Cooley-Tukey butterfly

Existing Solutions

Existing Solutions

ASIC

LEIA [CICC '18], Sapphire [ISSCC '19], ...

Existing Solutions

ASIC

LEIA [CICC ’18], Sapphire [ISSCC ’19], …

- ❑ Low efficiency
  - ○ Frequent data movement

Existing Solutions

ASIC

LEIA [CICC '18], Sapphire [ISSCC '19], …

- ❑ Low efficiency
  - Frequent data movement
- ❑ Limited flexibility
  - Dedicated modular multiplier

Existing Solutions

ASIC

LEIA [CICC '18], Sapphire [ISSCC '19], …

- Low efficiency
  - Frequent data movement
- Limited flexibility
  - Dedicated modular multiplier

How to achieve efficient & flexible NTT acceleration?

Existing Solutions

ASIC

LEIA [CICC '18], Sapphire [ISSCC '19], …

- ❑ Low efficiency
  - Frequent data movement
- ❑ Limited flexibility
  - Dedicated modular multiplier

Compute-in-Memory

Recryptor [JSSC '18], Duality Cache [ISCA '19], …

How to achieve efficient & flexible NTT acceleration?

Existing Solutions

ASIC

LEIA [CICC ’18], Sapphire [ISSCC ’19], …

- ❑ Low efficiency
  - ○ Frequent data movement
- ❑ Limited flexibility
  - ○ Dedicated modular multiplier

→

Compute-in-Memory

Recryptor [JSSC ’18], Duality Cache [ISCA ’19], …

- ❑ Potential high efficiency
  - ○ Reduced data movement

How to achieve efficient & flexible NTT acceleration?

Existing Solutions

ASIC

LEIA [CICC '18], Sapphire [ISSCC '19], …

- ❑ Low efficiency
  - Frequent data movement
- ❑ Limited flexibility
  - Dedicated modular multiplier

→

Compute-in-Memory

Recryptor [JSSC '18], Duality Cache [ISCA '19], …

- ❑ Potential high efficiency
  - Reduced data movement
- ❑ High flexibility
  - General vector processing units

How to achieve efficient & flexible NTT acceleration?

Challenges for Memory-Centric NTT

Challenges for Memory-Centric NTT

Observations:

- Inefficient data layout incurs ***redundant shifts***

Challenges for Memory-Centric NTT

Observations:

- Inefficient data layout incurs ***redundant shifts***

Contribution 1:

Shift-optimized data layout to avoid *redundant shifts*

Challenges for Memory-Centric NTT

Observations:

- Inefficient data layout incurs ***redundant shifts***

Contribution 1:

Shift-optimized data layout to avoid *redundant shifts*

- Bit-parallel modular multiplication has ***unnecessary carry propagations***

Challenges for Memory-Centric NTT

Observations:

- ❑ Inefficient data layout incurs ***redundant shifts***

Contribution 1:

Shift-optimized data layout to avoid *redundant shifts*

- ❑ Bit-parallel modular multiplication has ***unnecessary carry propagations***

Contribution 2:

Carry-save modular multiplication to avoid *carry propagations*

Overview of Our Solution: *BP-NTT*

Overview of Our Solution: *BP-NTT*

BP-NTT **repurposes LLC to perform bitline computing**

Overview of Our Solution: *BP-NTT*

BP-NTT **repurposes LLC to perform bitline computing**

High security ⤶ **High Throughput** ⤶

Overview of Our Solution: *BP-NTT*

BP-NTT **repurposes LLC to perform bitline computing**

High security

High Throughput

BP-NTT **uses shift-optimized data alignment**

Overview of Our Solution: *BP-NTT*

BP-NTT **repurposes LLC to perform bitline computing**

High security ↩ High Throughput

BP-NTT **uses shift-optimized data alignment**

Low Latency ↩ ↪ Low Energy

Overview of Our Solution: *BP-NTT*

BP-NTT **repurposes LLC to perform bitline computing**

High security

High Throughput

BP-NTT **uses shift-optimized data alignment**

Low Latency

Low Energy

BP-NTT **employs bit-parallel modular multiplication**

Overview of Our Solution: *BP-NTT*

BP-NTT **repurposes LLC to perform bitline computing**

High security

High Throughput

BP-NTT **uses shift-optimized data alignment**

Low Latency

Low Energy

BP-NTT **employs bit-parallel modular multiplication**

Low Latency

Small Area

Overview of Our Solution: *BP-NTT*

BP-NTT **repurposes LLC to perform bitline computing**

High security | High Throughput

BP-NTT **uses shift-optimized data alignment**

Low Latency | Low Energy

BP-NTT **employs bit-parallel modular multiplication**

Low Latency | Small Area

BP-NTT: Repurposed LLC

[1] Aga, Shaizeen, et al. “Compute caches.” HPCA 2017

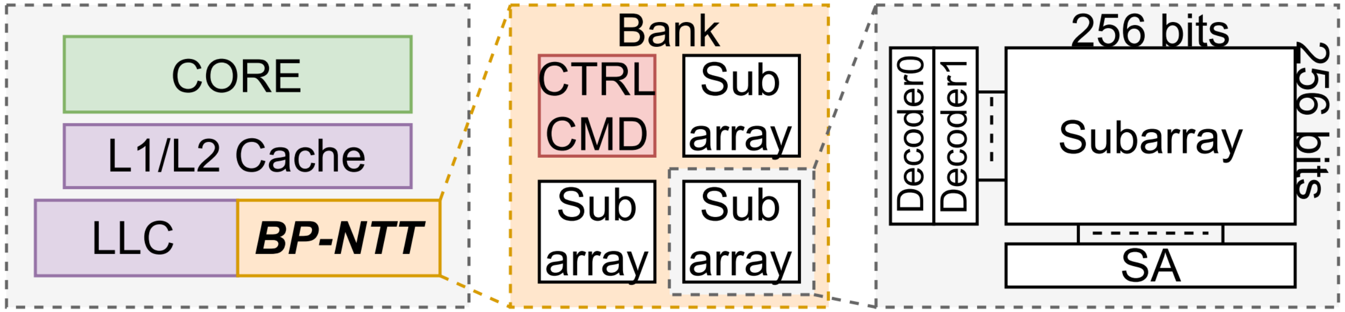
BP-NTT: Repurposed LLC

- *BP-NTT* repurposes LLC to perform bitline computing [1]

[1] Aga, Shaizeen, et al. "Compute caches." HPCA 2017

BP-NTT: Repurposed LLC

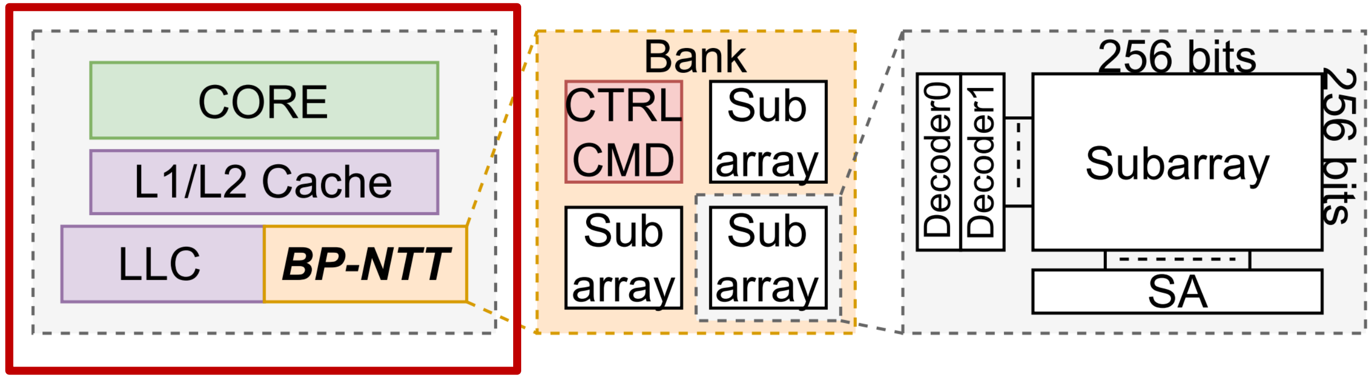
- *BP-NTT* repurposes LLC to perform bitline computing [1]



[1] Aga, Shaizeen, et al. "Compute caches." HPCA 2017

BP-NTT: Repurposed LLC

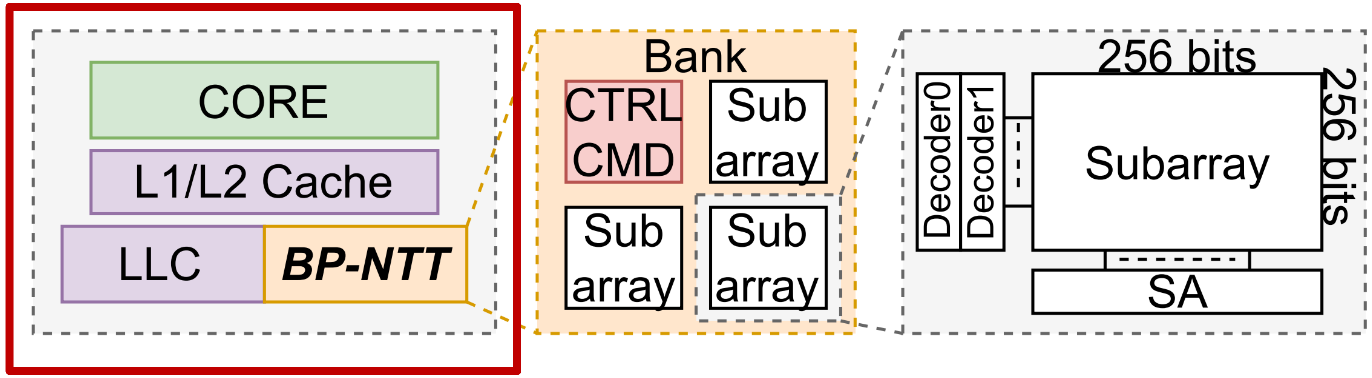
- *BP-NTT* repurposes LLC to perform bitline computing [1]



[1] Aga, Shaizeen, et al. "Compute caches." HPCA 2017

BP-NTT: Repurposed LLC

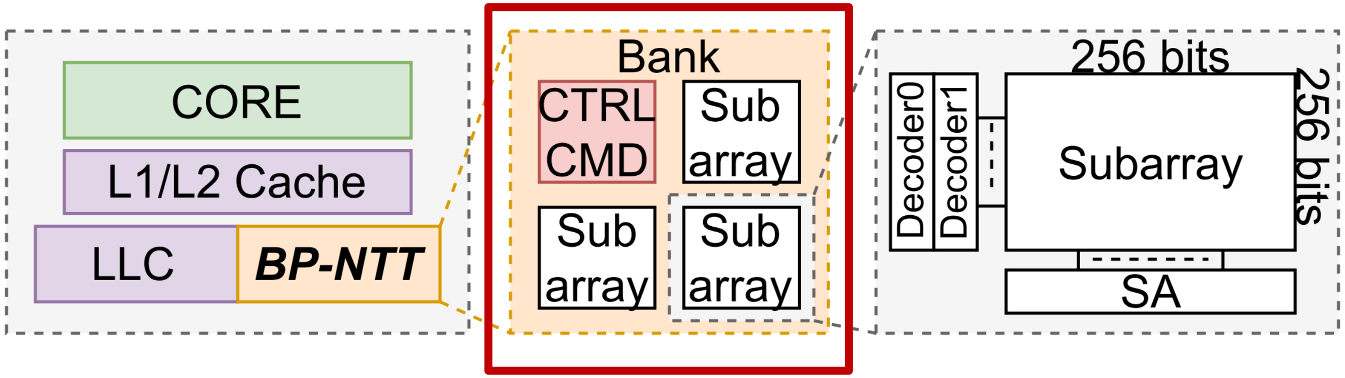
- *BP-NTT* repurposes LLC to perform bitline computing [1]
  - High ***security***



[1] Aga, Shaizeen, et al. "Compute caches." HPCA 2017

BP-NTT: Repurposed LLC

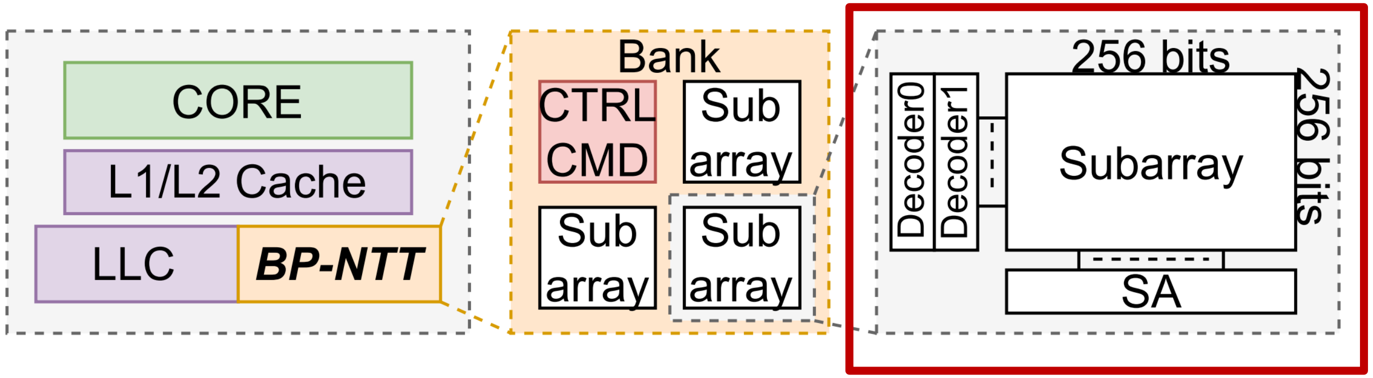
- *BP-NTT* repurposes LLC to perform bitline computing [1]
  - High ***security***



[1] Aga, Shaizeen, et al. "Compute caches." HPCA 2017

BP-NTT: Repurposed LLC

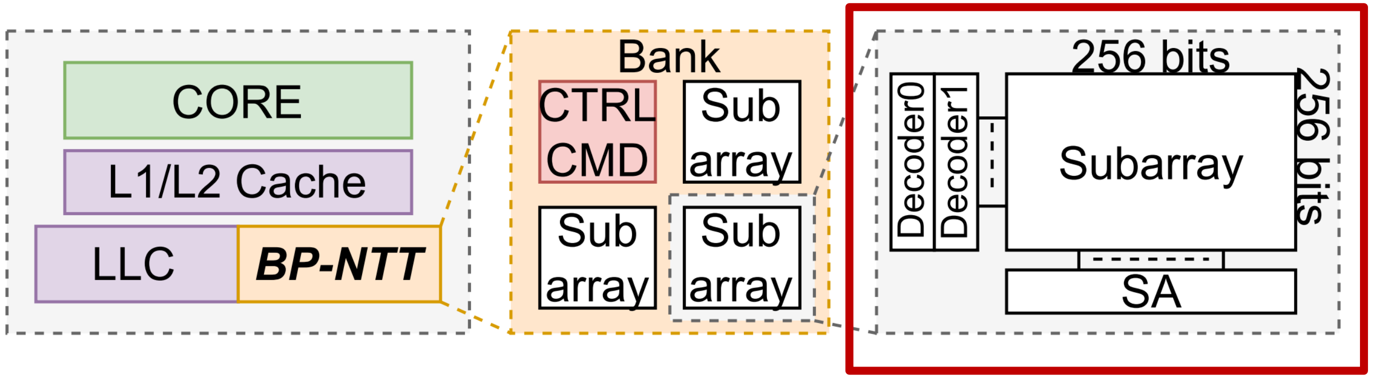
- *BP-NTT* repurposes LLC to perform bitline computing [1]
  - High ***security***



[1] Aga, Shaizeen, et al. "Compute caches." HPCA 2017

BP-NTT: Repurposed LLC

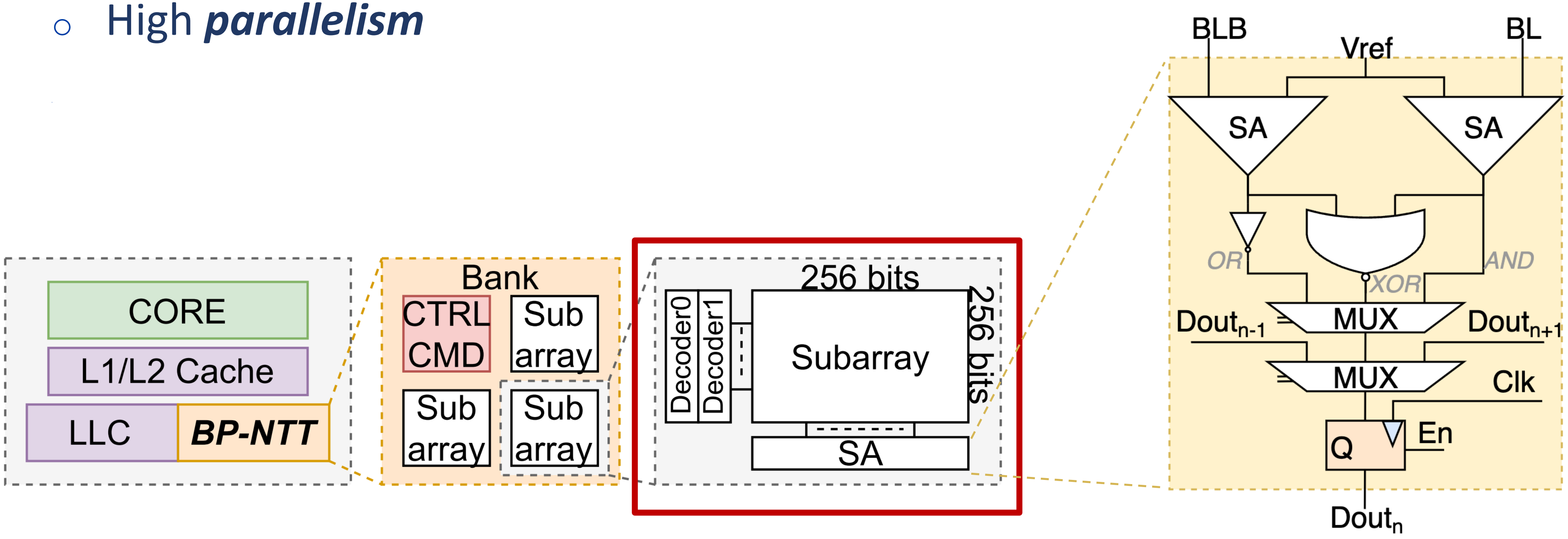
- *BP-NTT* repurposes LLC to perform bitline computing [1]
  - High ***security***
  - High ***parallelism***



[1] Aga, Shaizeen, et al. "Compute caches." HPCA 2017

BP-NTT: Repurposed LLC

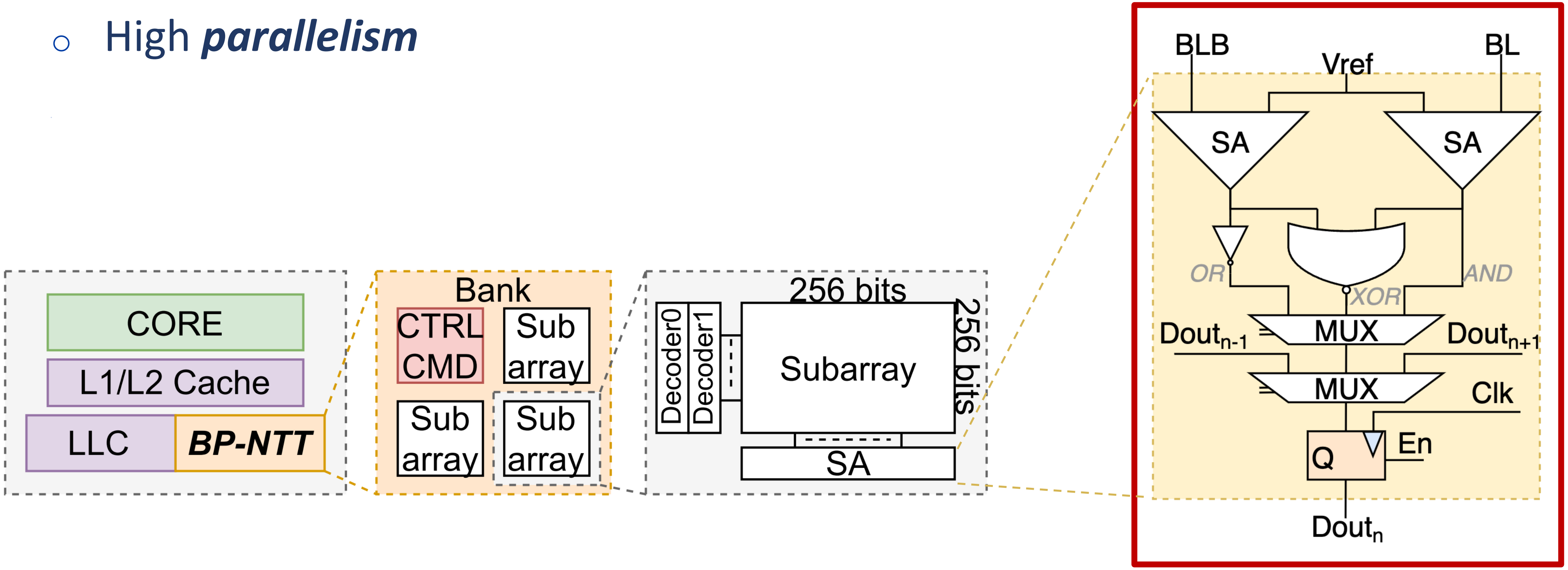
- *BP-NTT* repurposes LLC to perform bitline computing [1]
  - High ***security***
  - High ***parallelism***



[1] Aga, Shaizeen, et al. "Compute caches." HPCA 2017

BP-NTT: Repurposed LLC

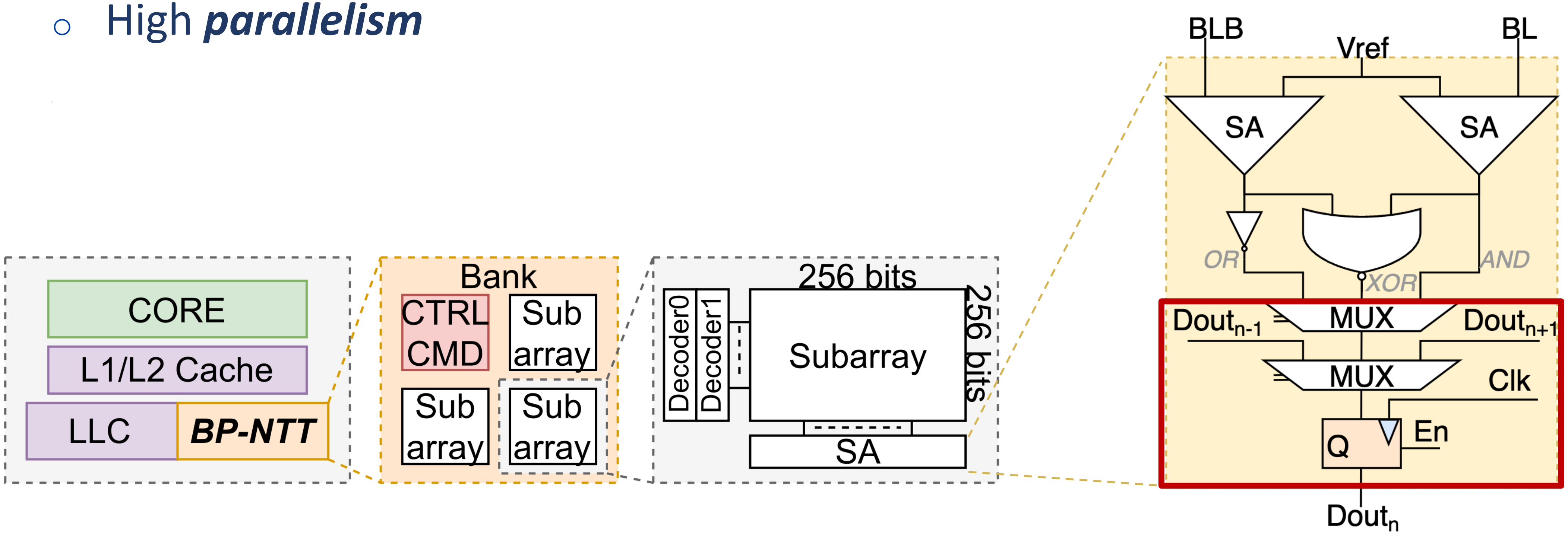
- *BP-NTT* repurposes LLC to perform bitline computing [1]
  - High ***security***
  - High ***parallelism***



[1] Aga, Shaizeen, et al. "Compute caches." HPCA 2017

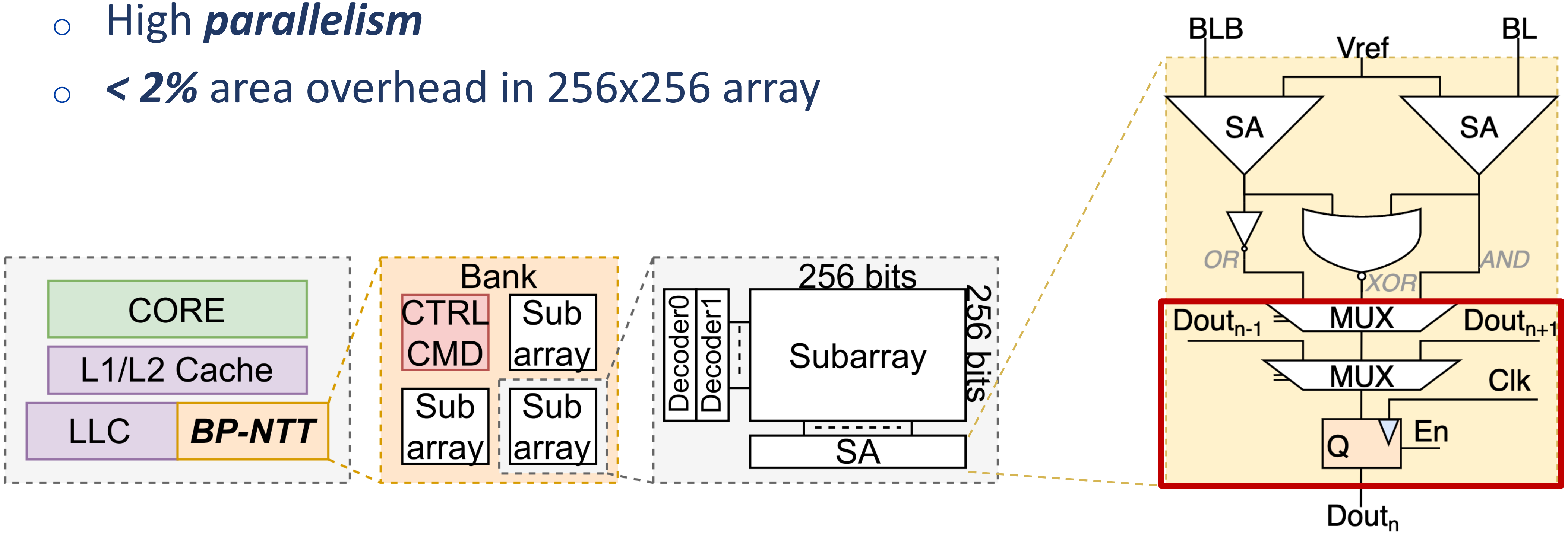
BP-NTT: Repurposed LLC

- *BP-NTT* repurposes LLC to perform bitline computing [1]
  - High ***security***
  - High ***parallelism***



BP-NTT: Repurposed LLC

- *BP-NTT* repurposes LLC to perform bitline computing [1]
  - High ***security***
  - High ***parallelism***
  - ***< 2%*** area overhead in 256x256 array



[1] Aga, Shaizeen, et al. "Compute caches." HPCA 2017

Overview of Our Solution: *BP-NTT*

BP-NTT repurposes LLC to perform bitline computing

High security ↩ High Throughput

BP-NTT uses shift-optimized data alignment

Low Latency ↩ ↪ Low Energy

BP-NTT employs bit-parallel modular multiplication

Low Latency ↩ ↪ Small Area

Motivation for Shift-optimization

Motivation for Shift-optimization

- ~50% operations of 256-point 16-bit NTT are shifting in bit-serial

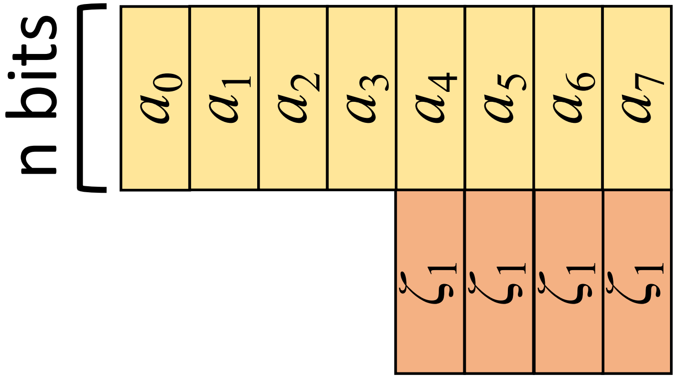
Motivation for Shift-optimization

- ~50% operations of 256-point 16-bit NTT are shifting in bit-serial

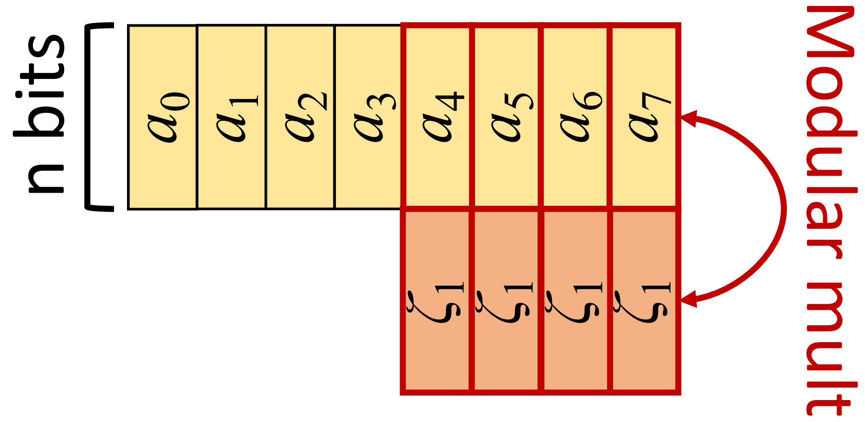
- Shifting destroys parallelism due to bit-by-bit shift fashion

BP-NTT: Shift-optimized Data Alignment

BP-NTT: Shift-optimized Data Alignment

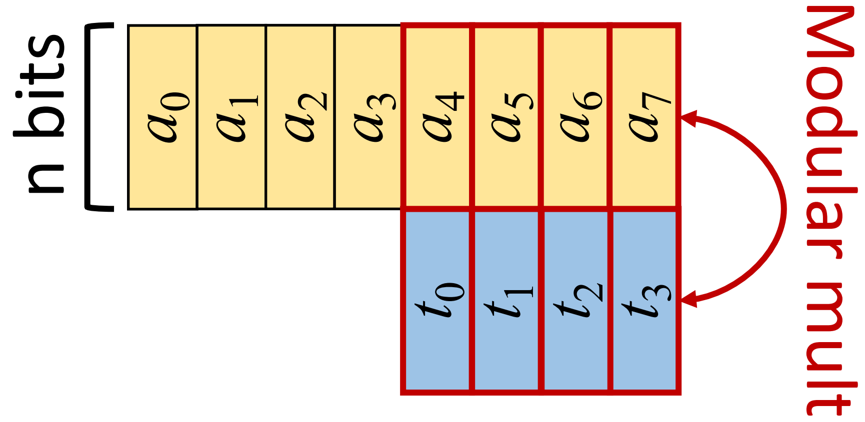


BP-NTT: Shift-optimized Data Alignment



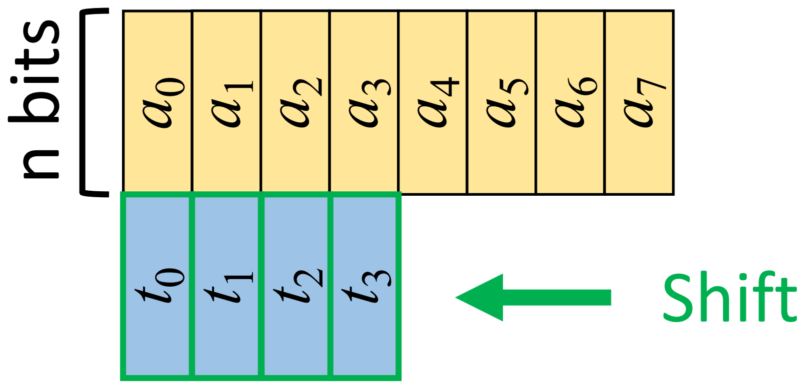
Step 1: Modular mult

BP-NTT: Shift-optimized Data Alignment



Step 1: Modular mult

BP-NTT: Shift-optimized Data Alignment



Step 1: Modular mult

Step 2: Shift & Write back

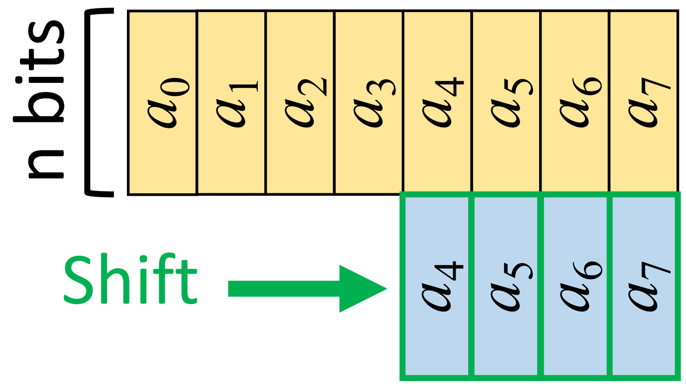
BP-NTT: Shift-optimized Data Alignment

Step 1: Modular mult

Step 2: Shift & Write back

Step 3: Add & Sub

BP-NTT: Shift-optimized Data Alignment



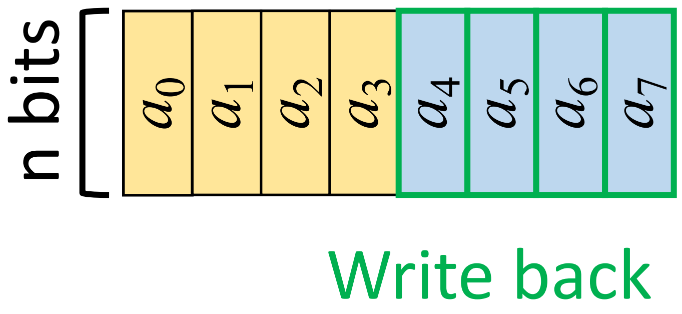
Step 1: Modular mult

Step 2: Shift & Write back

Step 3: Add & Sub

Step 4: Shift & Write back

BP-NTT: Shift-optimized Data Alignment



Step 1: Modular mult

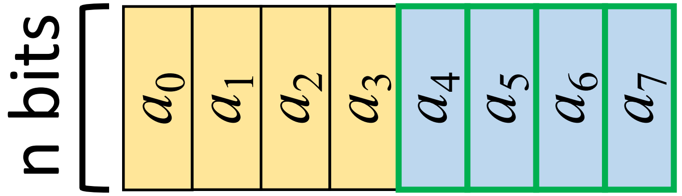
Step 2: Shift & Write back

Step 3: Add & Sub

Step 4: Shift & Write back

BP-NTT: Shift-optimized Data Alignment

- Inefficient data layout incurs ***redundant shifts***



Write back

Step 1: Modular mult

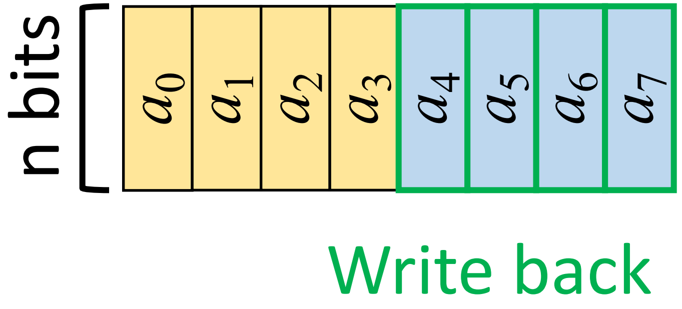
Step 2: Shift & Write back

Step 3: Add & Sub

Step 4: Shift & Write back

BP-NTT: Shift-optimized Data Alignment

- Inefficient data layout incurs ***redundant shifts***



Step 1: Modular mult

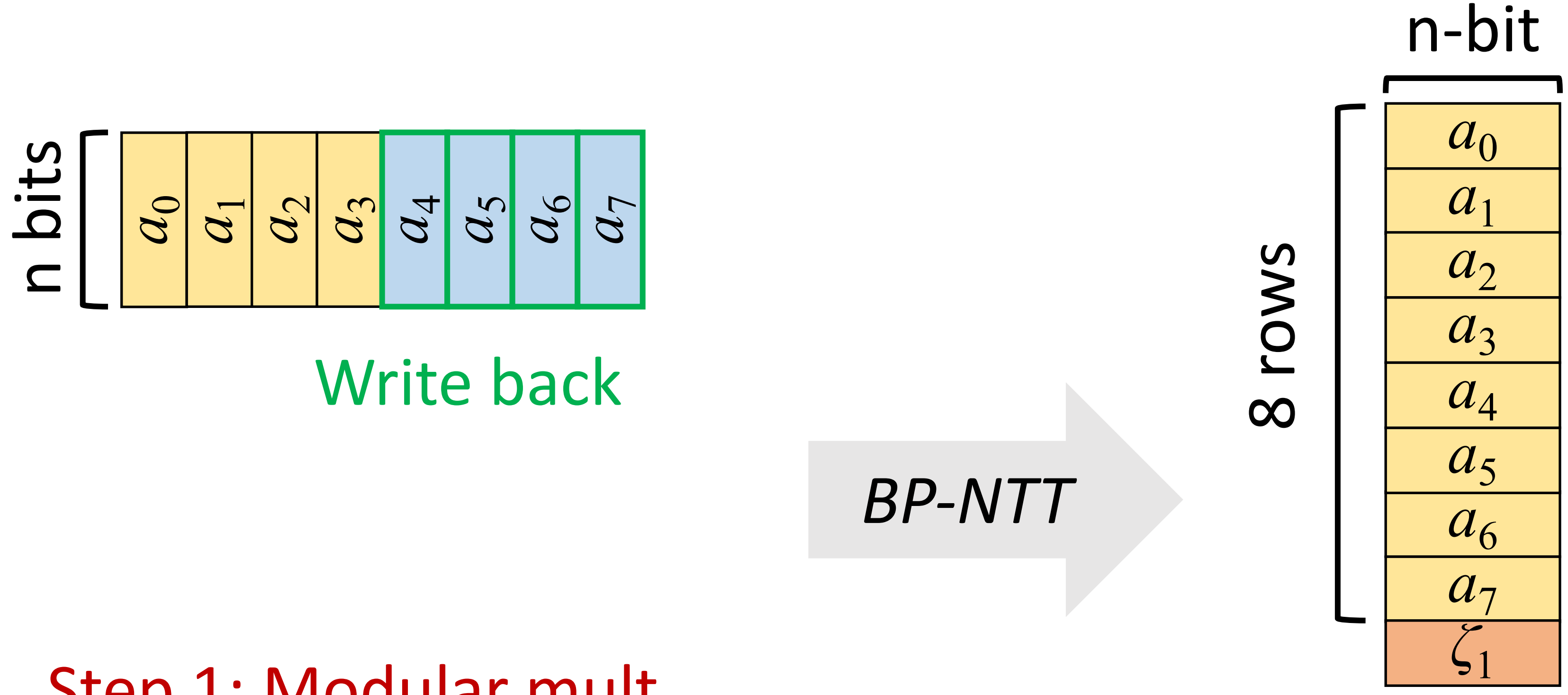
Step 2: Shift & Write back

Step 3: Add & Sub

Step 4: Shift & Write back

BP-NTT: Shift-optimized Data Alignment

- Inefficient data layout incurs ***redundant shifts***



Step 1: Modular mult

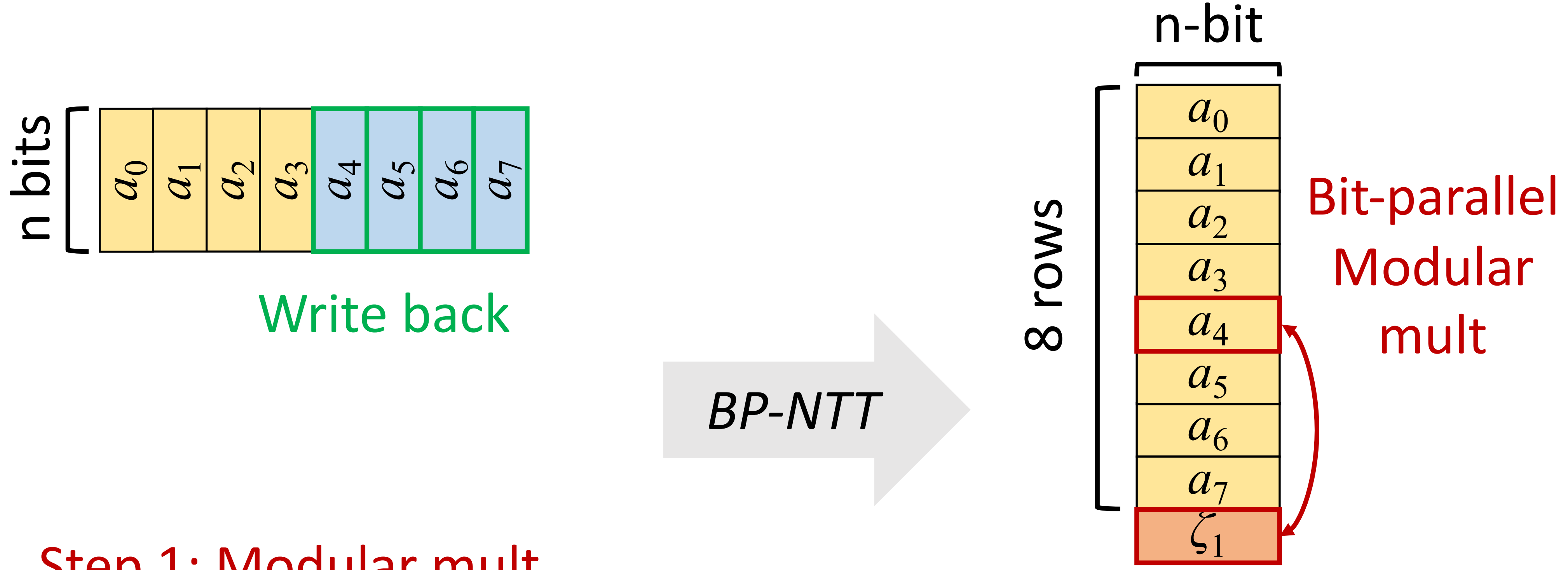
Step 2: Shift & Write back

Step 3: Add & Sub

Step 4: Shift & Write back

BP-NTT: Shift-optimized Data Alignment

- Inefficient data layout incurs ***redundant shifts***

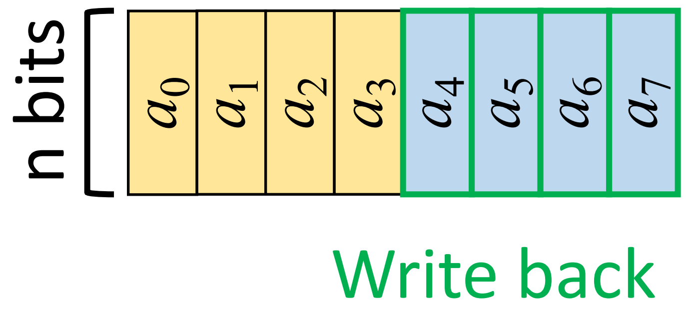


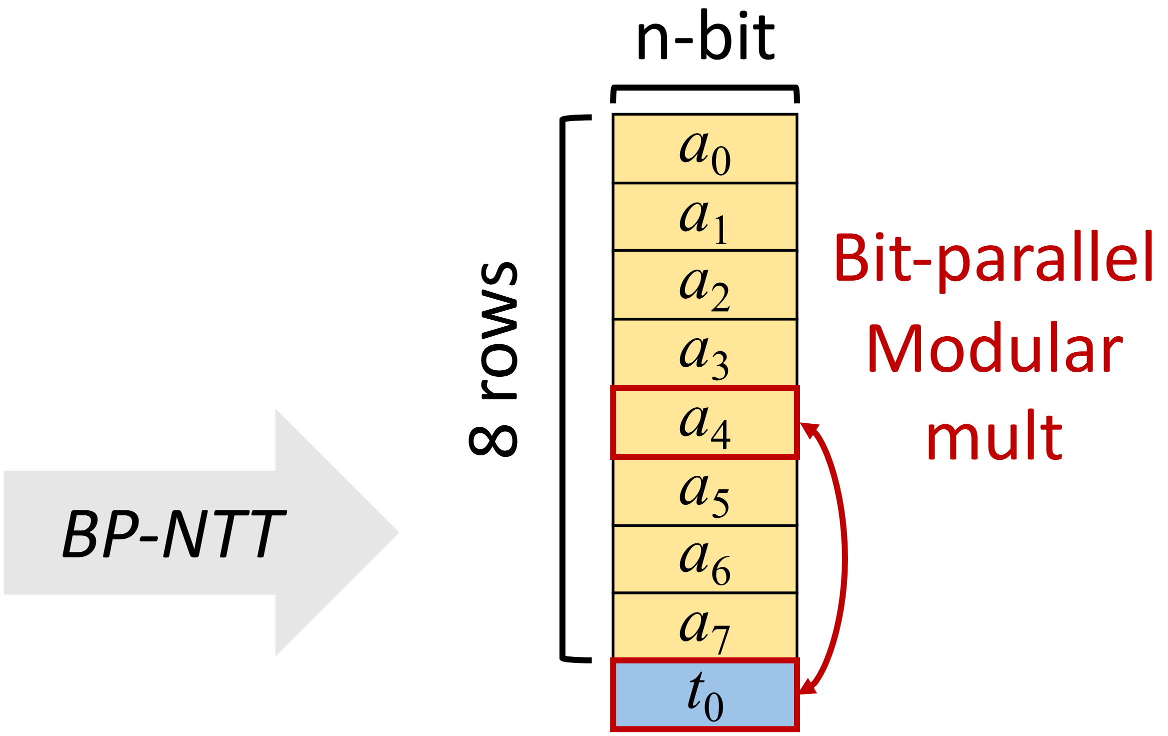
Step 1: Modular mult
Step 2: Shift & Write back
Step 3: Add & Sub
Step 4: Shift & Write back

Step 1: Modular mult

BP-NTT: Shift-optimized Data Alignment

- Inefficient data layout incurs ***redundant shifts***



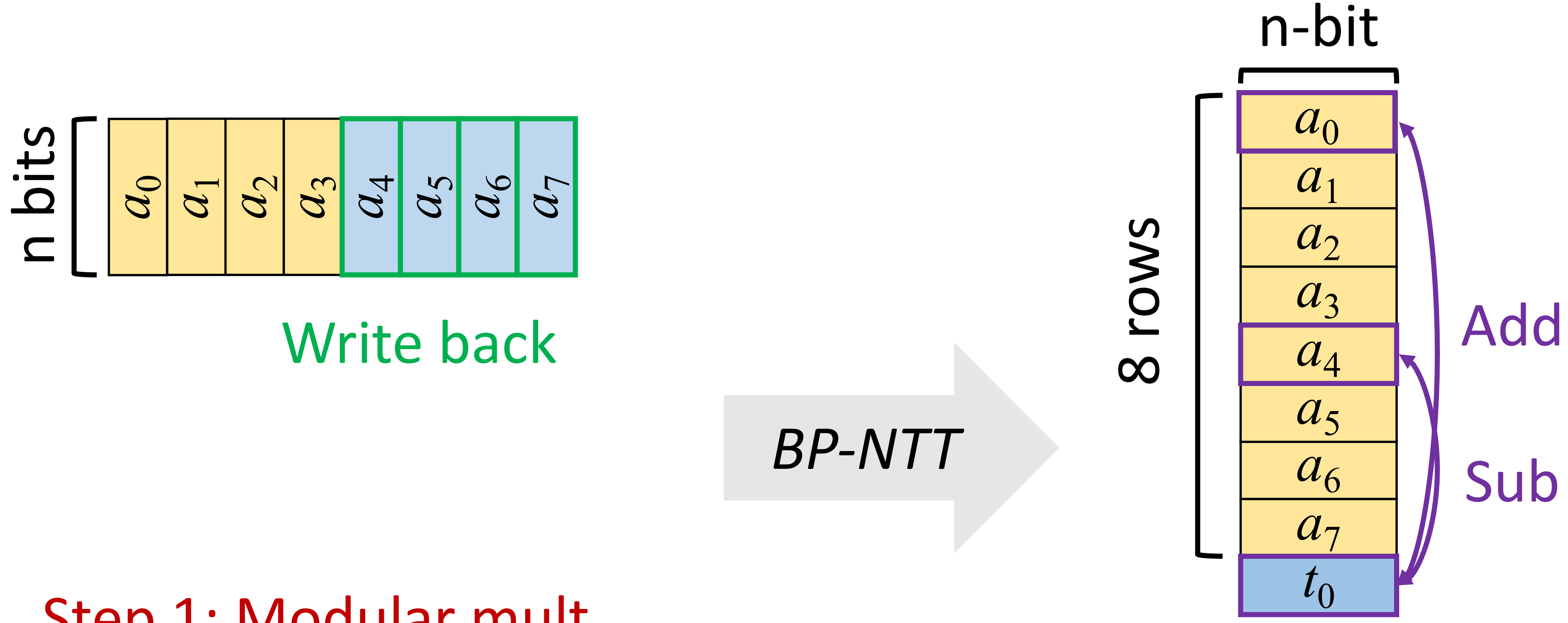


Step 1: Modular mult
Step 2: Shift & Write back
Step 3: Add & Sub
Step 4: Shift & Write back

Step 1: Modular mult

BP-NTT: Shift-optimized Data Alignment

- Inefficient data layout incurs ***redundant shifts***

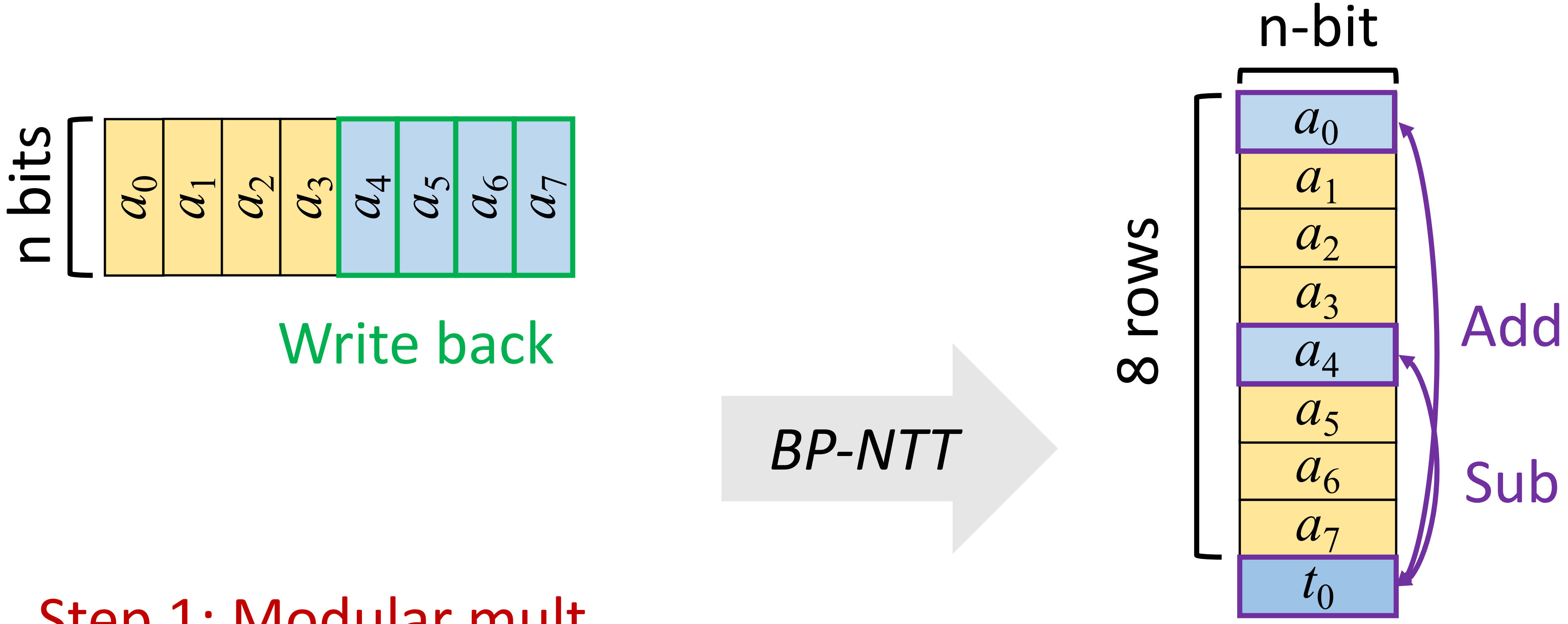


Step 1: Modular mult
Step 2: Shift & Write back
Step 3: Add & Sub
Step 4: Shift & Write back

Step 1: Modular mult
Step 2: Add & Sub

BP-NTT: Shift-optimized Data Alignment

- Inefficient data layout incurs ***redundant shifts***

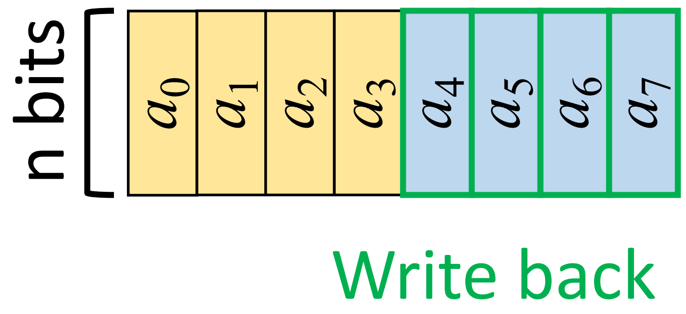


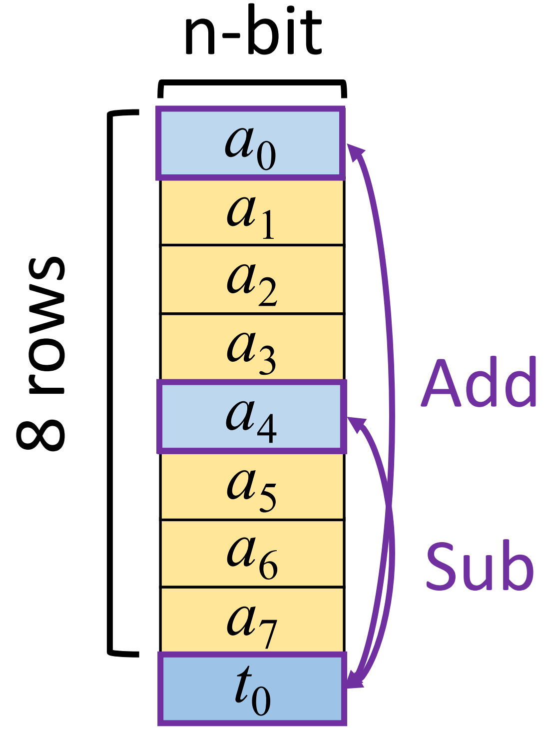
Step 1: Modular mult
Step 2: Shift & Write back
Step 3: Add & Sub
Step 4: Shift & Write back

Step 1: Modular mult
Step 2: Add & Sub

BP-NTT: Shift-optimized Data Alignment

- Inefficient data layout incurs ***redundant shifts***
- **4-step stage** is simplified into **2-step**



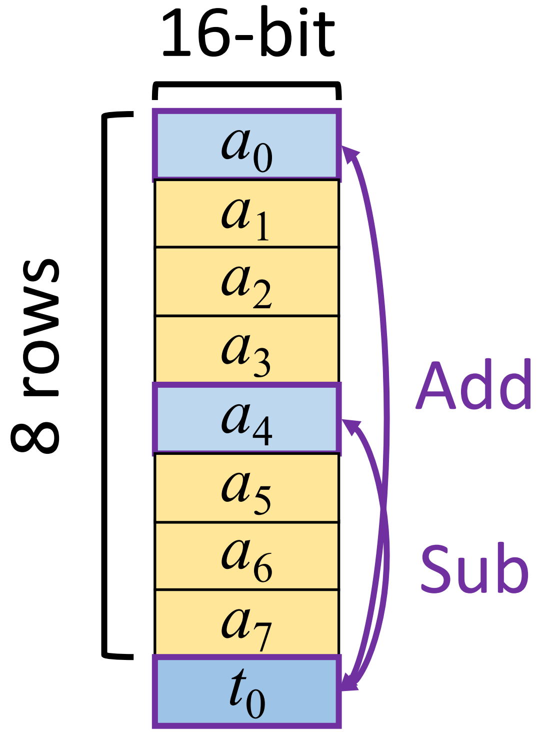


Step 1: Modular mult
Step 2: Shift & Write back
Step 3: Add & Sub
Step 4: Shift & Write back

Step 1: Modular mult
Step 2: Add & Sub

BP-NTT: Shift-optimized Data Alignment

- **Shift-optimized Data Alignment**

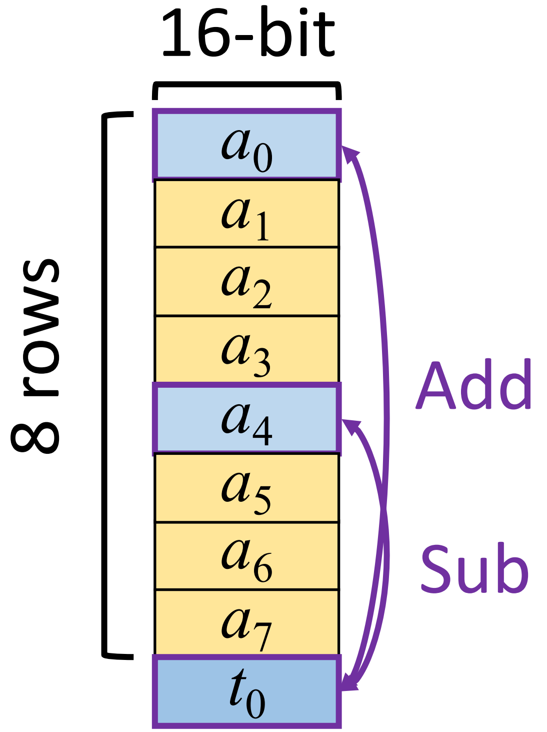


Step 1: Modular mult

Step 2: Add & Sub

BP-NTT: Shift-optimized Data Alignment

- **Shift-optimized Data Alignment**
  - **Place coefficient per row**

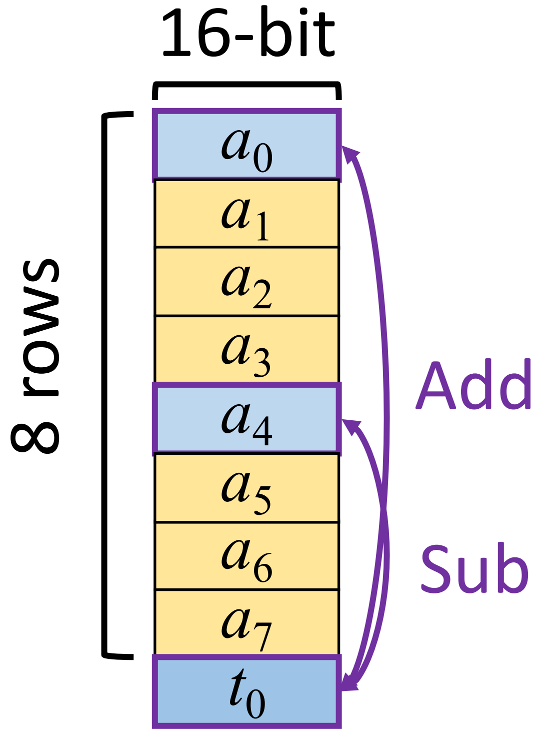


Step 1: Modular mult

Step 2: Add & Sub

BP-NTT: Shift-optimized Data Alignment

- **Shift-optimized Data Alignment**
  - **Place coefficient per row**
  - **No shift operations** to align data

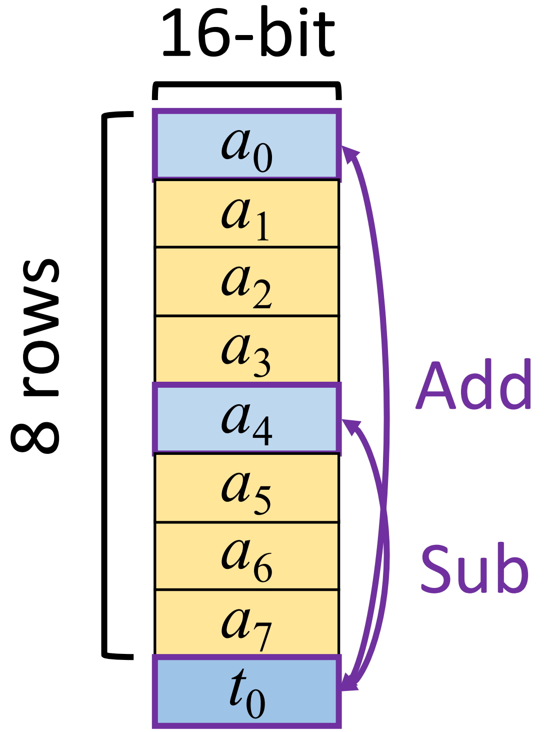


Step 1: Modular mult

Step 2: Add & Sub

BP-NTT: Shift-optimized Data Alignment

- **Shift-optimized Data Alignment**
  - **Place coefficient per row**
  - **No shift operations** to align data
  - Enable bit-parallel multiplication



Step 1: Modular mult

Step 2: Add & Sub

Overview of Our Solution: *BP-NTT*

BP-NTT repurposes LLC to perform bitline computing

High security

High Throughput

BP-NTT uses shift-optimized data alignment

Low Latency

Low Energy

BP-NTT employs bit-parallel modular multiplication

Low Latency

Small Area

Motivation for carry-save multiplication

Motivation for carry-save multiplication

- Multiplication is based on multiple additions

Motivation for carry-save multiplication

- Multiplication is based on multiple additions

```
            1 1 1 1
          × 1 1 1 1
          ---------
            1 1 1 1
        + 1 1 1 1
      + 1 1 1 1
    + 1 1 1 1
    ---------------
    1 1 1 0 0 0 0 1
```

Multiplication

Motivation for carry-save multiplication

- Multiplication is based on multiple additions
- Carry propagation ruins the parallelism from in-SRAM computing

```
          1 1 1 1
        × 1 1 1 1
        ---------
          1 1 1 1
      + 1 1 1 1
    + 1 1 1 1
  + 1 1 1 1
  ---------------
  1 1 1 0 0 0 0 1
```

Multiplication

Motivation for carry-save multiplication

- Multiplication is based on multiple additions
- Carry propagation ruins the parallelism from in-SRAM computing
  - High computing complexity

```
          1 1 1 1
        × 1 1 1 1
        ---------
          1 1 1 1
      + 1 1 1 1
    + 1 1 1 1
  + 1 1 1 1
  ---------------
  1 1 1 0 0 0 0 1
```

Multiplication

Motivation for carry-save multiplication

- Multiplication is based on multiple additions
- Carry propagation ruins the parallelism from in-SRAM computing
  - High computing complexity

```
            1 1 1 1
          × 1 1 1 1
          ---------
            1 1 1 1
        + 1 1 1 1
      + 1 1 1 1
    + 1 1 1 1
    ---------------
    1 1 1 0 0 0 0 1
```

Multiplication

Complexity: **$O(n^2)$**

Motivation for carry-save multiplication

- Multiplication is based on multiple additions
- Carry propagation ruins the parallelism from in-SRAM computing
  - High computing complexity

```
        1 1 1 1
      × 1 1 1 1
      ---------
        1 1 1 1
    + 1 1 1 1
  + 1 1 1 1
+ 1 1 1 1
---------------
1 1 1 0 0 0 0 1
```

Multiplication

Complexity: $O(n^2)$

```
            1 1 1 1
          + 0 0 0 1
          ---------
Sum:        1 1 1 0    XOR  ┐
            0 0 0 1    AND  │
Carry:    0 0 0 1           │
          ---------         │
Sum:      0 1 1 0 0    XOR  ├ n iters
          0 0 0 1 0    AND  │
Carry:  0 0 0 1 0           │
        -----------         │
           ……               ┘
```

Addition

Motivation for carry-save multiplication

- Multiplication is based on multiple additions
- Carry propagation ruins the parallelism from in-SRAM computing
  - High computing complexity

```
          1 1 1 1
        × 1 1 1 1
        ---------
          1 1 1 1
      + 1 1 1 1
    + 1 1 1 1
  + 1 1 1 1
  ---------------
  1 1 1 0 0 0 0 1
```

Multiplication

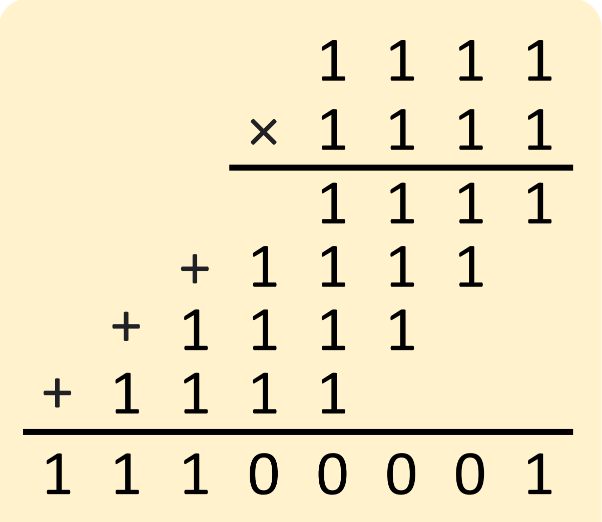
Complexity: **$O(n^2)$**

```
               1 1 1 1
             + 0 0 0 1
             ---------
        Sum:   1 1 1 0    XOR ┐
               0 0 0 1    AND │
      Carry: 0 0 0 1          │
      -----------------       │
       Sum:  0 1 1 0 0    XOR │
             0 0 0 1 0    AND ├ n iters
    Carry: 0 0 0 1 0          │
    -------------------       │
              ......          ┘
```

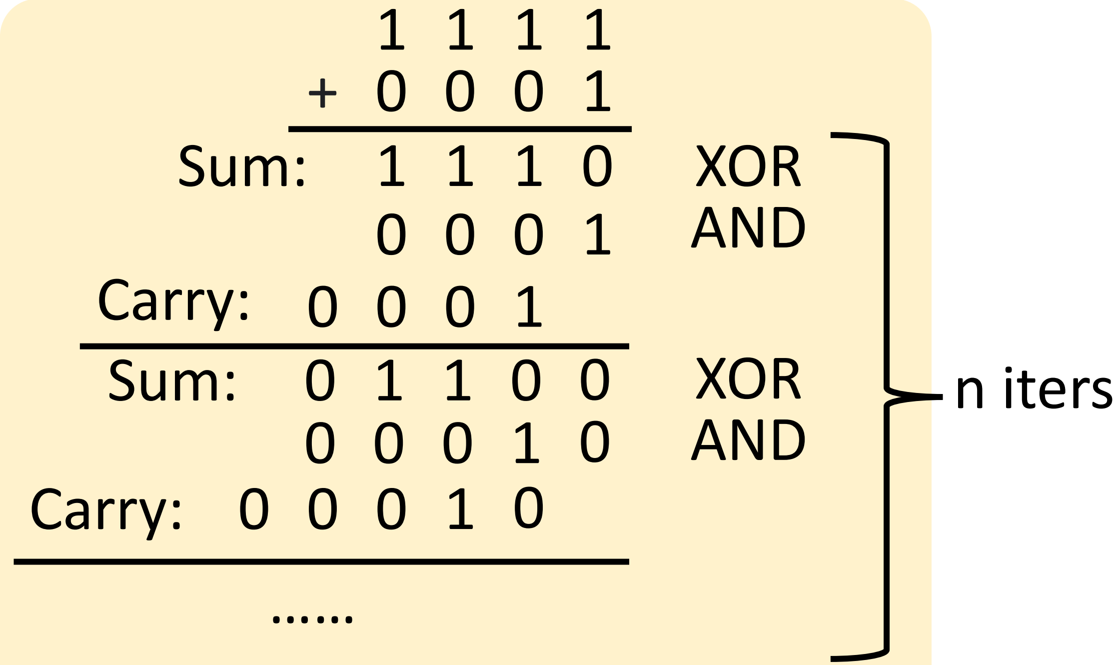
Addition Complexity: **$O(n)$**

Motivation for carry-save multiplication

- Multiplication is based on multiple additions
- Carry propagation ruins the parallelism from in-SRAM computing
  - High computing complexity
  - Low parallelism (require extra columns for overflow bits)



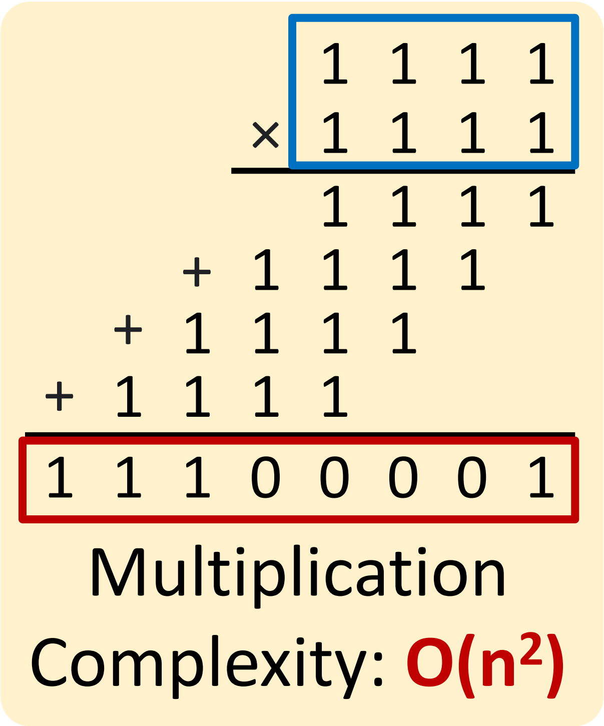
Multiplication Complexity: **O(n^2)**

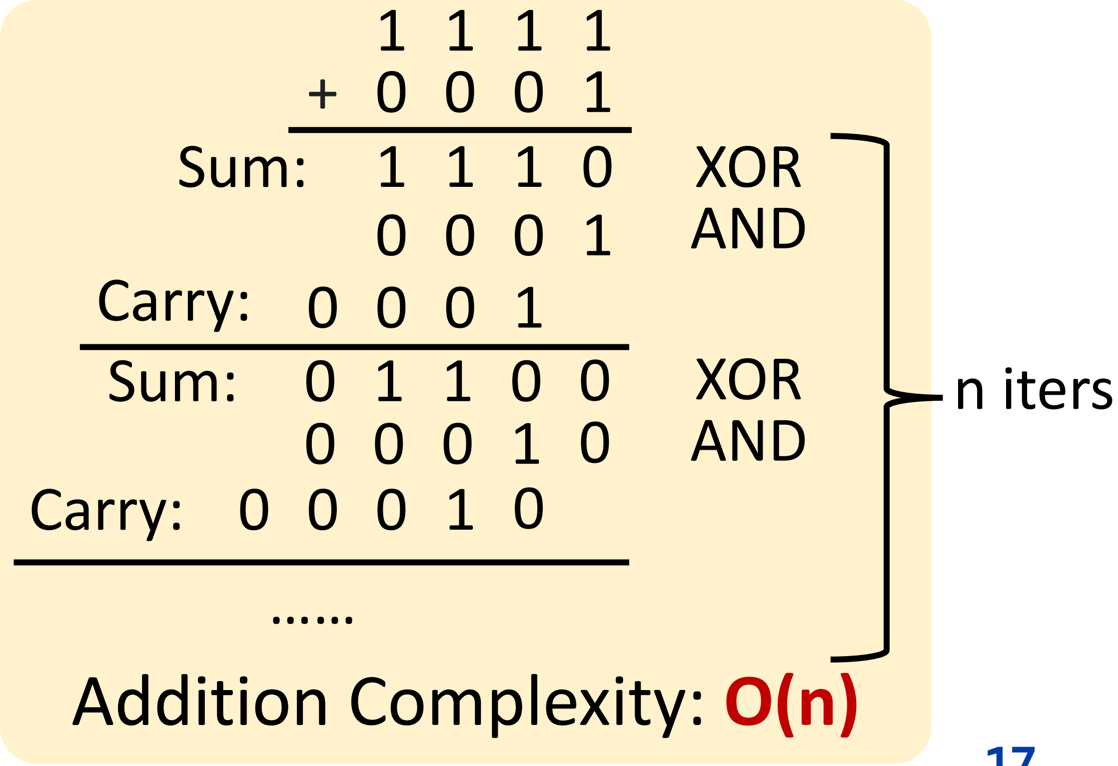


Addition Complexity: **O(n)**

Motivation for carry-save multiplication

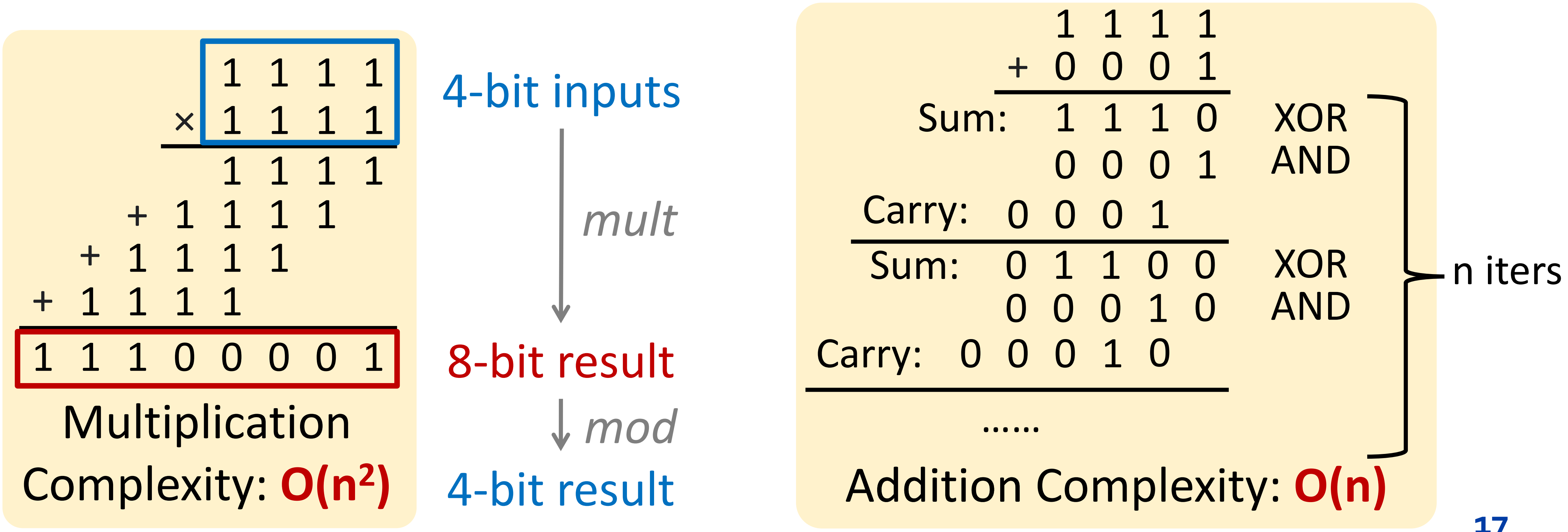
- Multiplication is based on multiple additions
- Carry propagation ruins the parallelism from in-SRAM computing
  - High computing complexity
  - Low parallelism (require extra columns for overflow bits)





Motivation for carry-save multiplication

- Multiplication is based on multiple additions
- Carry propagation ruins the parallelism from in-SRAM computing
  - High computing complexity
  - Low parallelism (require extra columns for overflow bits)



BP-NTT: Bit-Parallel Modular Multiplication

BP-NTT: Bit-Parallel Modular Multiplication

- Bit-parallel modular multiplication has ***unnecessary carry propagations***

BP-NTT: Bit-Parallel Modular Multiplication

- Bit-parallel modular multiplication has ***unnecessary carry propagations***

			1	1	1	1	
		+	0	0	0	1	
Sum:			1	1	1	0	XOR
			0	0	0	1	AND
Carry:		0	0	0	1		
Sum:		0	1	1	0	0	XOR
		0	0	0	1	0	AND
Carry:	0	0	0	1	0		
		……					n iters

Carry-propagation

Addition Complexity: **O(n)**

BP-NTT: Bit-Parallel Modular Multiplication

- Bit-parallel modular multiplication has ***unnecessary carry propagations***

```
               1  1  1  1
            +  0  0  0  1
           ---------------
  Sum:         1  1  1  0   XOR
               0  0  0  1   AND
  Carry:    0  0  0  1
  ------------------------
  Sum:      0  1  1  0  0   XOR
            0  0  0  1  0   AND
  Carry: 0  0  0  1  0
  ------------------------
          ......            n iters
```

Carry-propagation

Addition Complexity: **O(n)**

BP-NTT

BP-NTT: Bit-Parallel Modular Multiplication

- Bit-parallel modular multiplication has ***unnecessary carry propagations***

		1	1	1	1	
	+	0	0	0	1	
Sum:		1	1	1	0	XOR
		0	0	0	1	AND
Carry:	0	0	0	1		
Sum:	0	1	1	0	0	XOR
	0	0	0	1	0	AND
Carry: 0	0	0	1	0		
	……					n iters

Carry-propagation

Addition Complexity: **O(n)**

→ *BP-NTT* →

		1	1	1	1	
	+	0	0	0	1	
Sum:		1	1	1	0	XOR
Carry:		0	0	0	1	AND

Carry-save Addition

Complexity: **O(1)**

BP-NTT: Bit-Parallel Modular Multiplication

- Bit-parallel modular multiplication has ***unnecessary carry propagations***
- Multiplication **O(n^2)** is reduced into **O(n)** inspired by carry-save addition

```
          1 1 1 1
        + 0 0 0 1
        ---------
Sum:      1 1 1 0   XOR
          0 0 0 1   AND
Carry:  0 0 0 1
        ---------
Sum:    0 1 1 0 0   XOR
        0 0 0 1 0   AND
Carry: 0 0 0 1 0
       ----------
        ......      n iters
```

Carry-propagation
Addition Complexity: **O(n)**

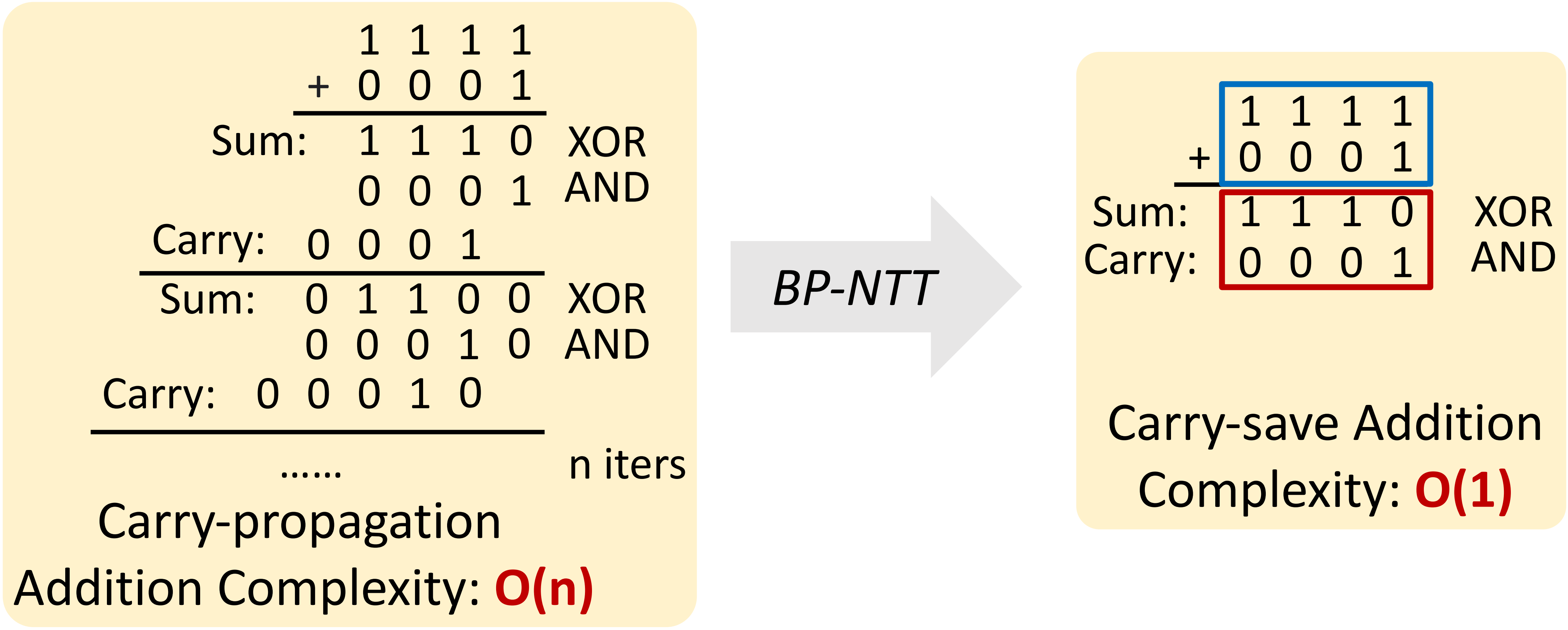
BP-NTT →

```
          1 1 1 1
        + 0 0 0 1
        ---------
Sum:      1 1 1 0   XOR
Carry:    0 0 0 1   AND
```

Carry-save Addition
Complexity: **O(1)**

BP-NTT: Bit-Parallel Modular Multiplication

- Bit-parallel modular multiplication has ***unnecessary carry propagations***
- Multiplication **O(n^2)** is reduced into **O(n)** inspired by carry-save addition



BP-NTT: Bit-Parallel Modular Multiplication

- Bit-parallel modular multiplication has ***unnecessary carry propagations***
- Multiplication **O(n²)** is reduced into **O(n)** inspired by carry-save addition

```
              1 1 1 1
            + 0 0 0 1
            ---------
Sum:          1 1 1 0   XOR
              0 0 0 1   AND
Carry:      0 0 0 1
          -----------
Sum:        0 1 1 0 0   XOR
            0 0 0 1 0   AND
Carry:    0 0 0 1 0
        -------------
          ......        n iters
```

Carry-propagation

Addition Complexity: **O(n)**

BP-NTT →

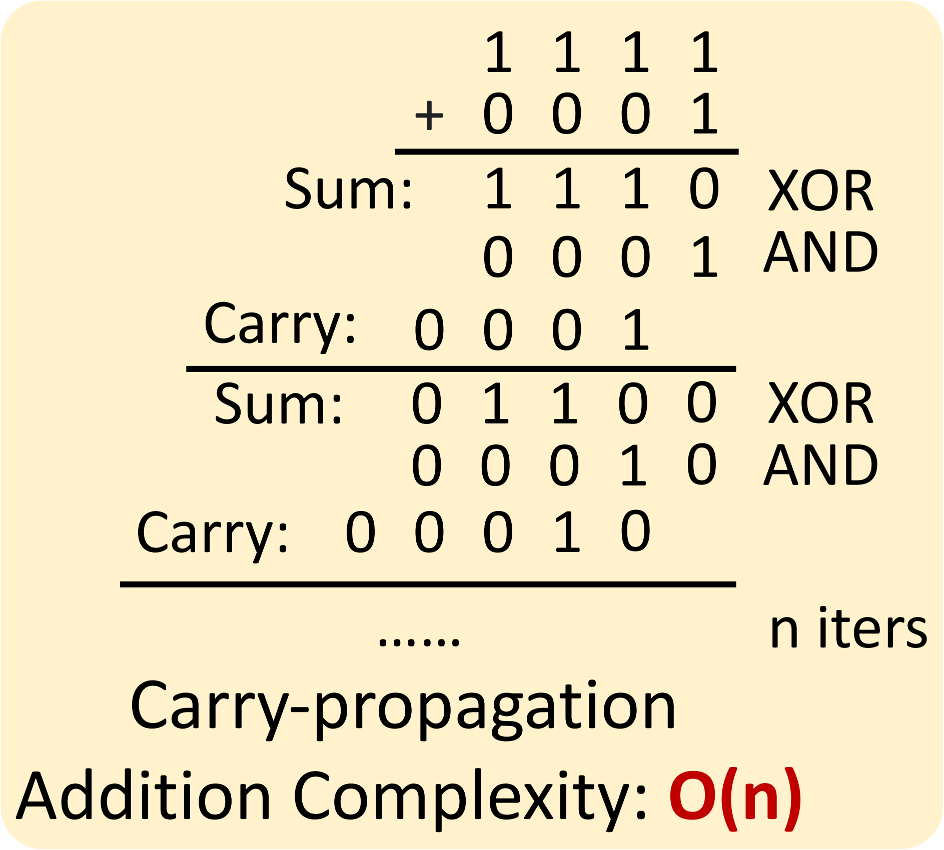
```
          1 1 1 1     4-bit inputs
        + 0 0 0 1
        ---------
Sum:      1 1 1 0     XOR
Carry:    0 0 0 1     AND
                      4-bit results
```

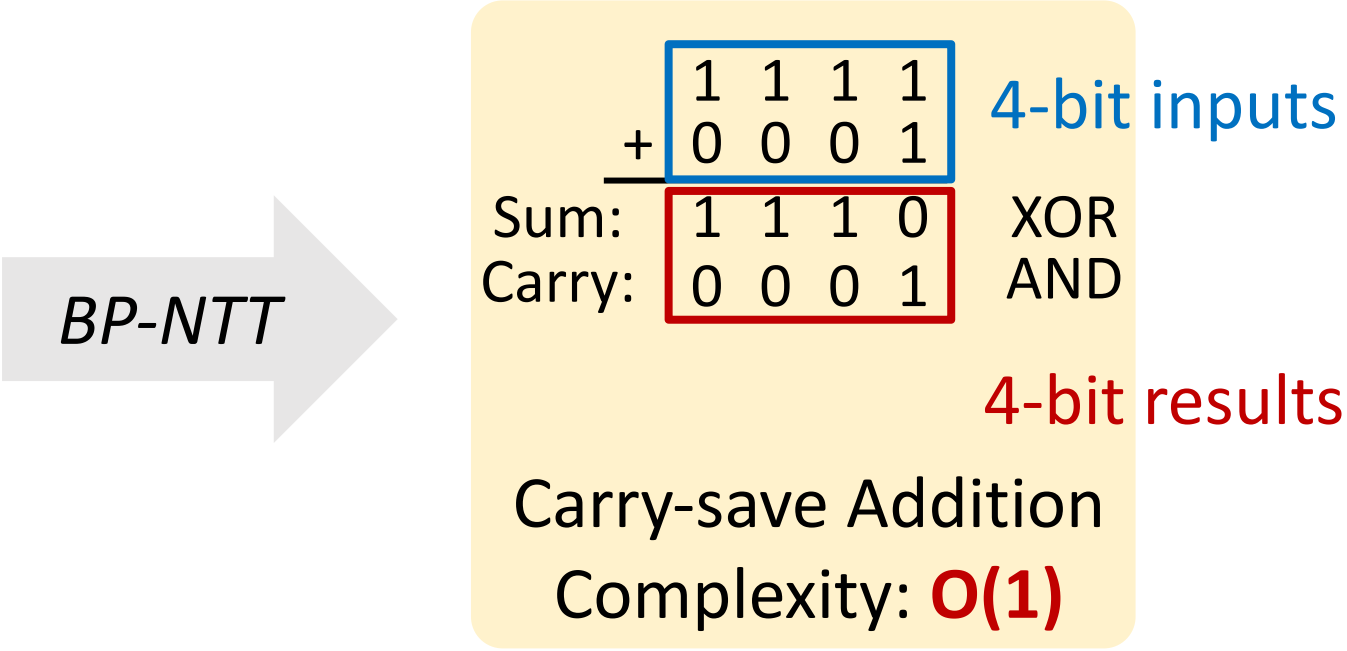
Carry-save Addition

Complexity: **O(1)**

BP-NTT: Bit-Parallel Modular Multiplication

- Bit-parallel modular multiplication has ***unnecessary carry propagations***
- Multiplication **O(n^2)** is reduced into **O(n)** inspired by carry-save addition





Check paper for details

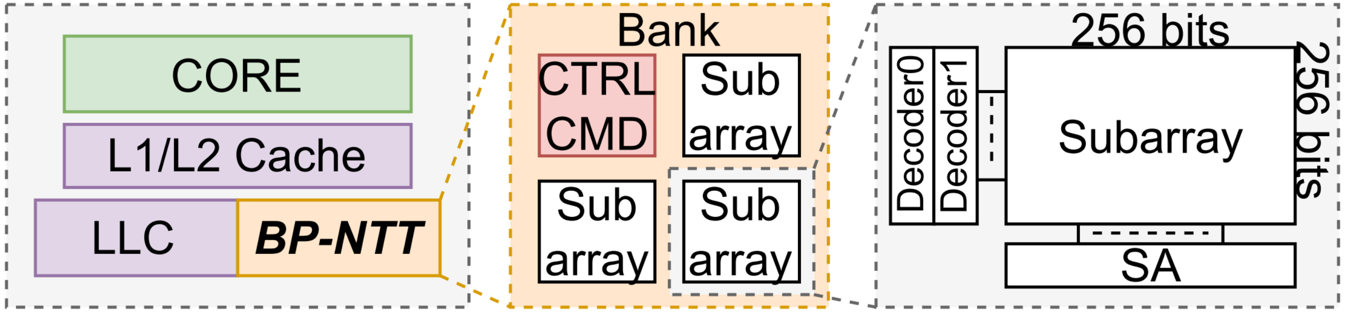
BP-NTT: Overall Architecture

BP-NTT: Overall Architecture

High-performance, low-overhead, energy-efficient and flexible NTT engine

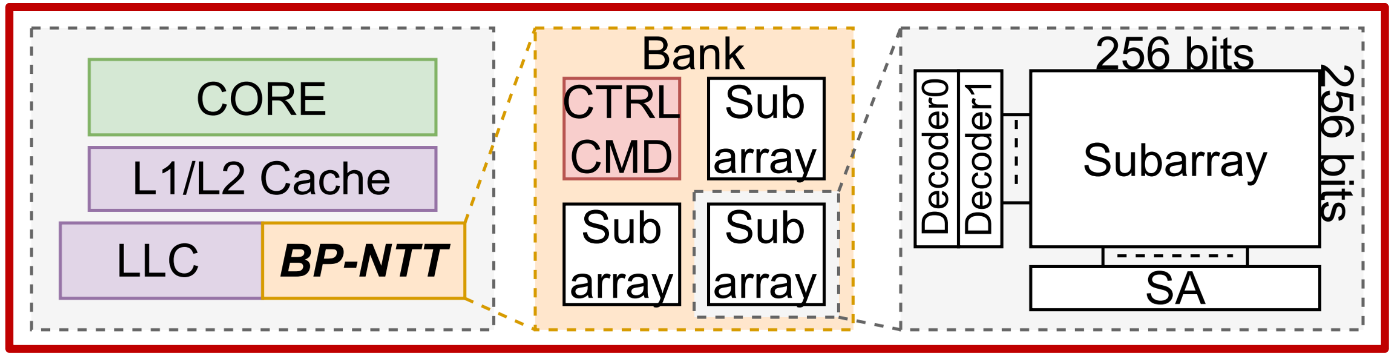
BP-NTT: Overall Architecture

High-performance, low-overhead, energy-efficient and flexible NTT engine



BP-NTT: Overall Architecture

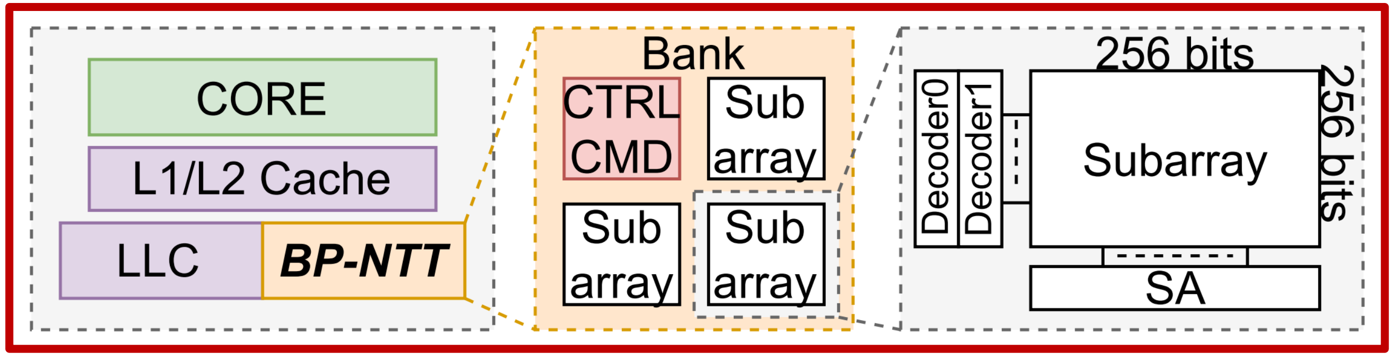
High-performance, low-overhead, energy-efficient and flexible NTT engine



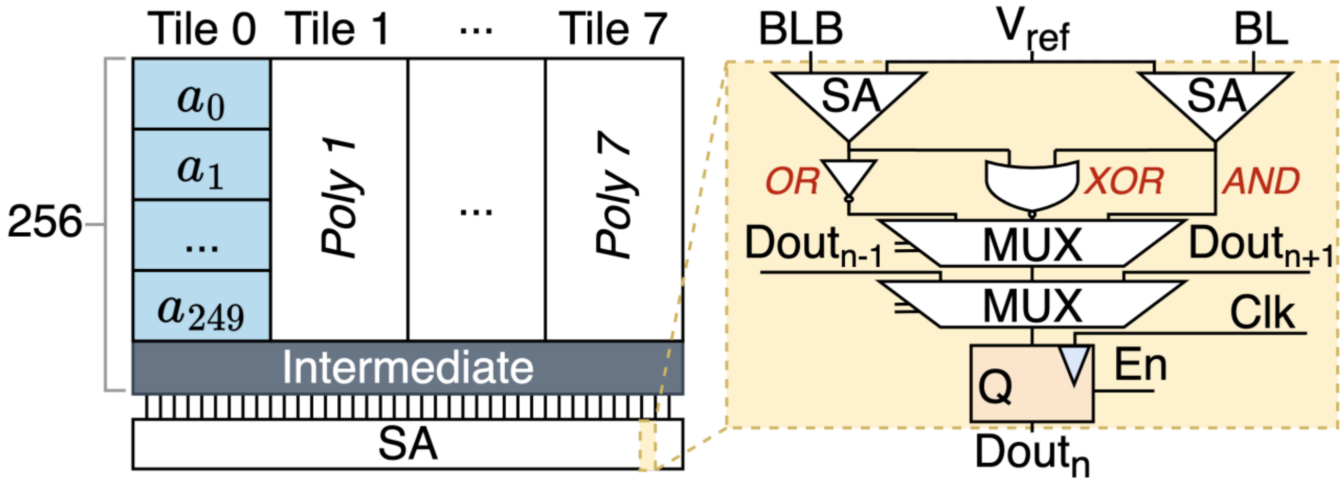
Security & Flexibility

BP-NTT: Overall Architecture

High-performance, low-overhead, energy-efficient and flexible NTT engine

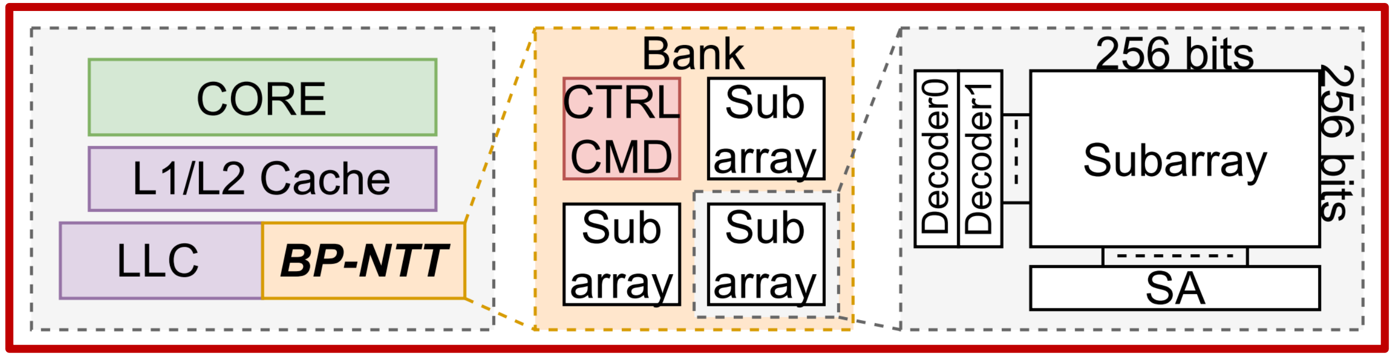


Security & Flexibility

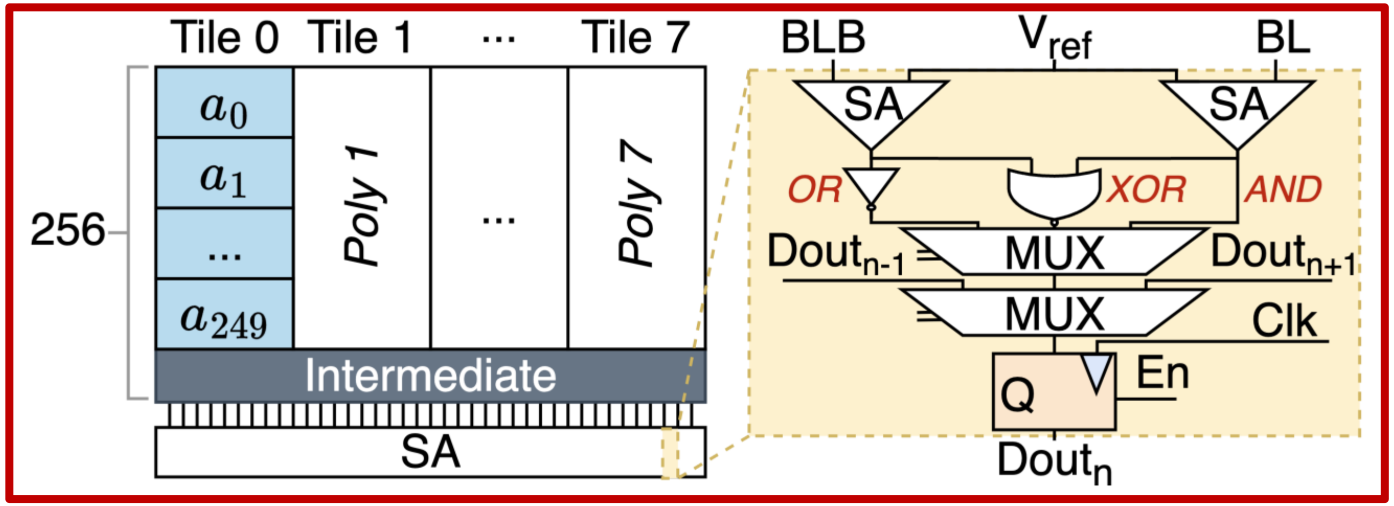


BP-NTT: Overall Architecture

High-performance, low-overhead, energy-efficient and flexible NTT engine



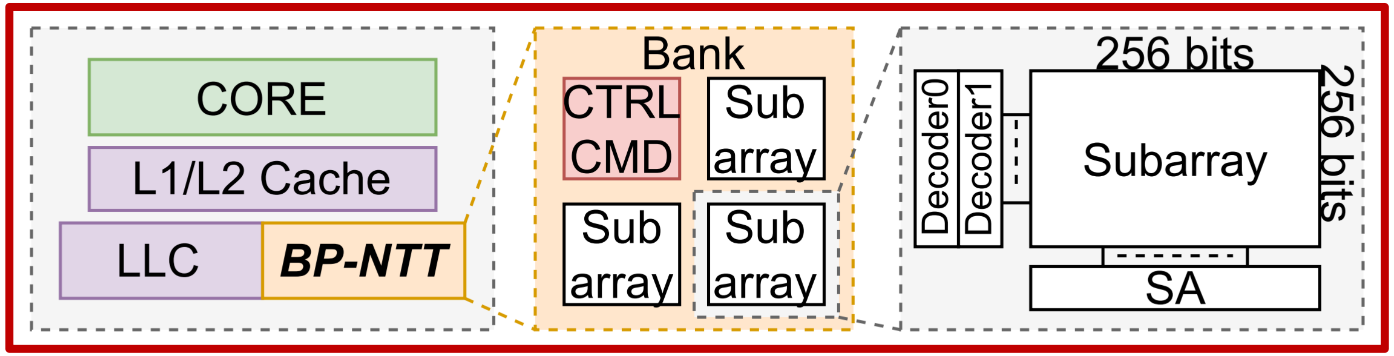
Security & Flexibility



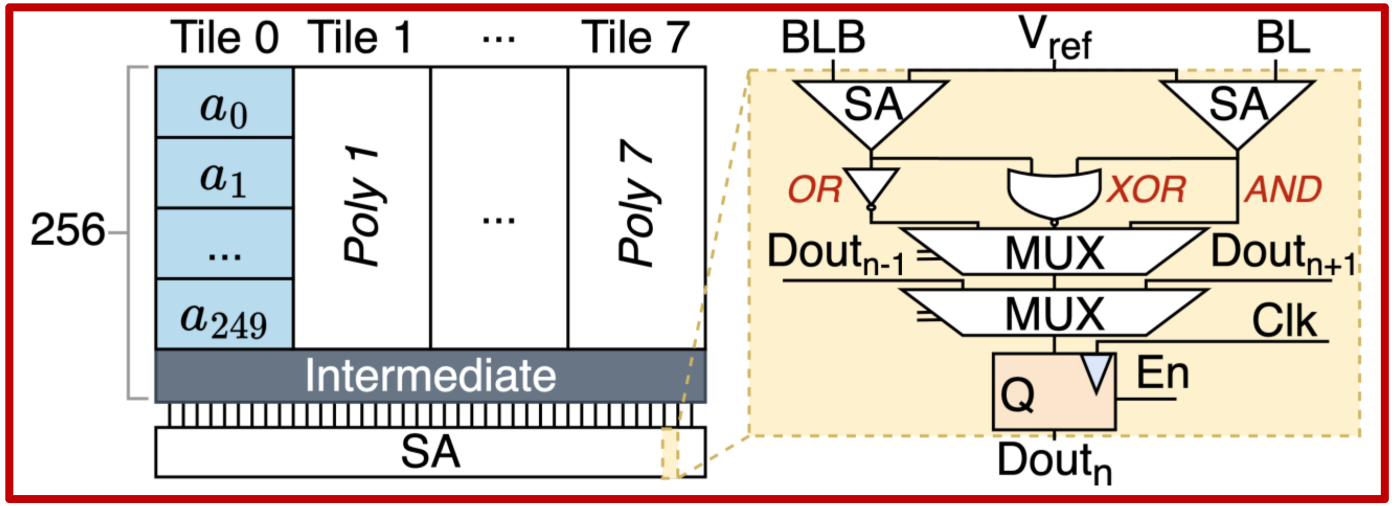
Throughput & Low-overhead

BP-NTT: Overall Architecture

High-performance, low-overhead, energy-efficient and flexible NTT engine



Security & Flexibility



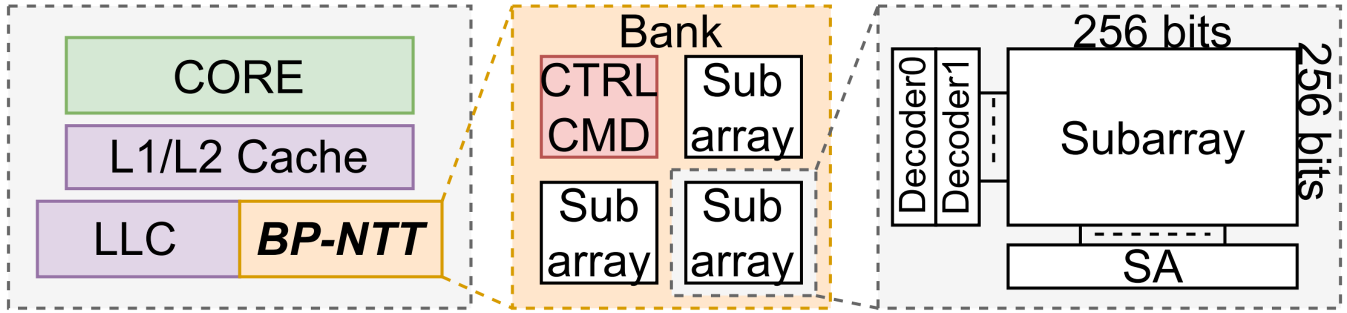
Throughput & Low-overhead

Input: n-bit $A = (a_{n-1}, ..., a_0)$, $B = (b_{n-1}, ..., b_0)$, $M < R = 2^n$, where $n > 2$ and $M \perp R$
Output: $ABR^{-1} \mod M$

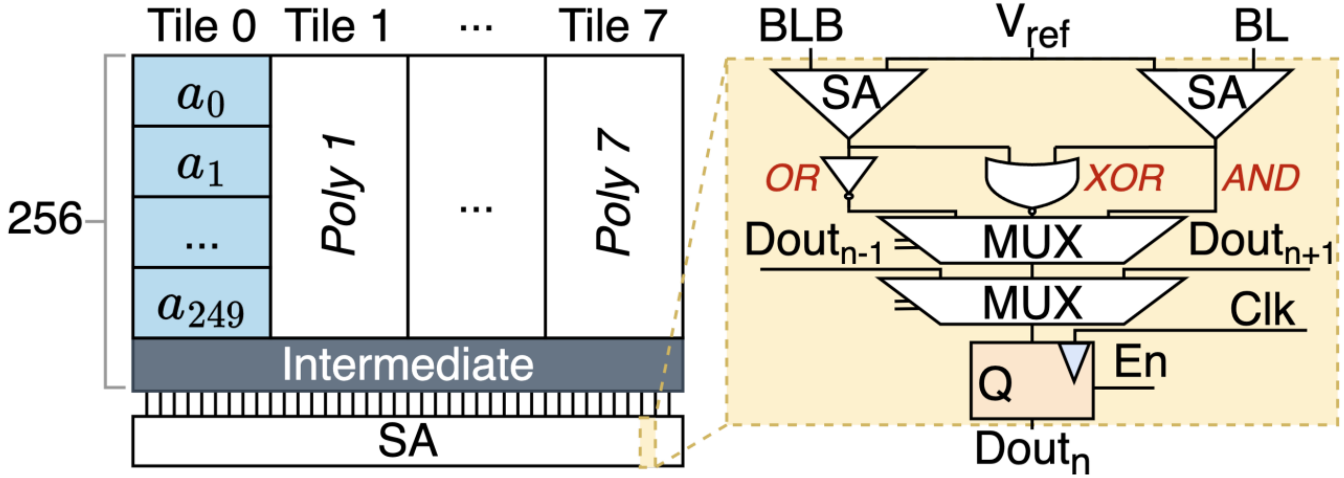
$\texttt{Sum} := (s_{n-1}, ..., s_0) = 0$ // Initialize
$\texttt{Carry} := (c_{n-1}, ..., c_0) = 0$
$P := \texttt{Sum} + \texttt{Carry} << 1$ // $P = 0$
for $i = 0, n-1$ **do**
  if $a_i == 1$ **then** // Implicit compare
    $c1, s1 = \{\texttt{Sum} \,\&\, B, \texttt{Sum} \oplus B\}$
    $\texttt{Carry} << 1$ <— **Oberservation 1**
    $c2, \texttt{Sum} = \{\texttt{Carry} \,\&\, s1, \texttt{Carry} \oplus s1\}$
    $\texttt{Carry} = c1 \,|\, c2$ // $P = P + a_i B$
  end if
  $m = (\texttt{LSB}(\texttt{Sum}) == 1)\ ?\ M\ :\ 0$ // $m = M$ or 0
  $c1, s1 = \{\texttt{Sum} \,\&\, m, \texttt{Sum} \oplus m\}$
  $s1 >> 1$ <— **Oberservation 2**
  $c2, s2 = \{s1 \,\&\, c1, s1 \oplus c1\}$
  $c3, \texttt{Sum} = \{\texttt{Carry} \,\&\, s2, \texttt{Carry} \oplus s2\}$
  $\texttt{Carry} = c2 \,|\, c3$ // $P = P + m$; $P >> 1$
end for

BP-NTT: Overall Architecture

High-performance, low-overhead, energy-efficient and flexible NTT engine



Security & Flexibility



Throughput & Low-overhead

Input: n-bit $A = (a_{n-1}, \ldots, a_0)$, $B = (b_{n-1}, \ldots, b_0)$, $M < R = 2^n$, where $n > 2$ and $M \perp R$

Output: $ABR^{-1} \mod M$

```
Sum := (s_{n-1}, ..., s_0) = 0                          // Initialize
Carry := (c_{n-1}, ..., c_0) = 0
P := Sum + Carry << 1                                   // P = 0
for i = 0, n − 1 do
    if a_i == 1 then                                    // Implicit compare
        c1, s1 = {Sum & B, Sum ⊕ B}
        Carry << 1                                      <— Oberservation 1
        c2, Sum = {Carry & s1, Carry ⊕ s1}
        Carry = c1 | c2                                 // P = P + a_i B
    end if
    m = (LSB(Sum) == 1) ? M : 0                         // m = M or 0
    c1, s1 = {Sum & m, Sum ⊕ m}
    s1 >> 1                                             <— Oberservation 2
    c2, s2 = {s1 & c1, s1 ⊕ c1}
    c3, Sum = {Carry & s2, Carry ⊕ s2}
    Carry = c2 | c3                                     // P = P + m; P >> 1
end for
```

Latency

Evaluation Methodology

- Tools:
  - PyMTL3 and OpenRAM for generating SRAM arrays
  - Synopsys Design Compiler for extracting latencies
  - Cadence Innovus for area consumption
- The array size of BP-NTT is 256×256 following the ARM Cortex-M0+ microcontroller
- Area consumptions of in-ReRAM baselines are from DESTINY simulator
  - Optimistically estimated with only subarray area consumption excluding complex peripheral circuitry

Jiang, Shunning, et al. "PyMTL3: A Python framework for open-source hardware modeling, generation, simulation, and verification." MICRO'20.
Guthaus, Matthew R., et al. "OpenRAM: An open-source memory compiler." ICCAD'16.
Poremba, Matt, et al. "Destiny: A tool for modeling emerging 3d nvm and edram caches." DATE'15.
Nagarajan, Karthikeyan, et al. "SHINE: A novel SHA-3 implementation using ReRAM-based in-memory computing." ISLPED'19

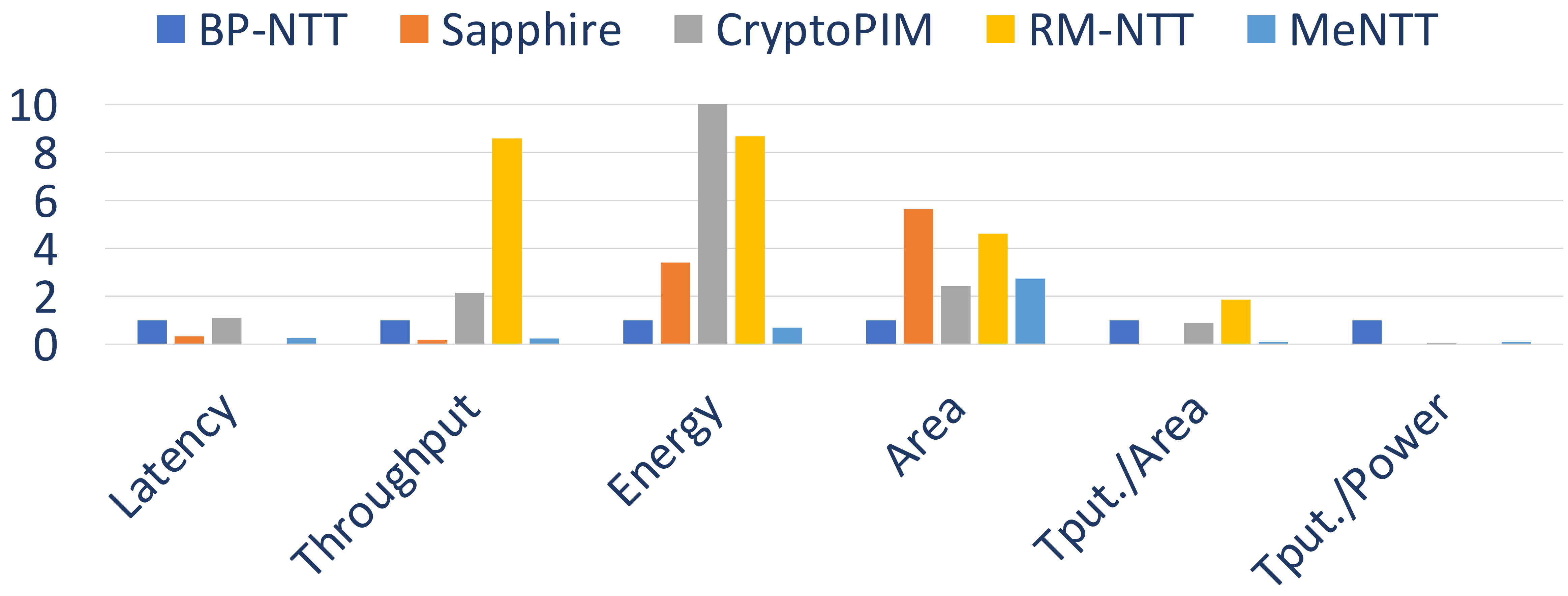
Comparison among Designs

Comparison among Designs

- Results are normalized to BP-NTT

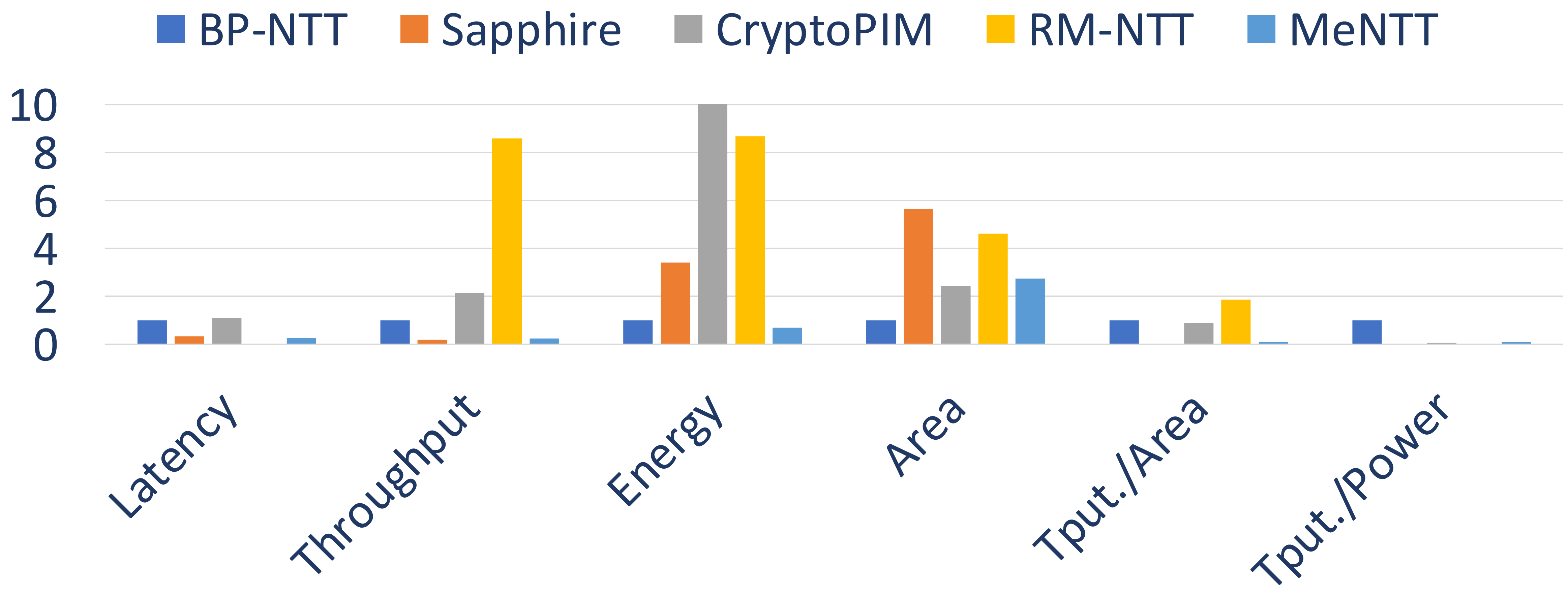
Comparison among Designs

- Results are normalized to BP-NTT



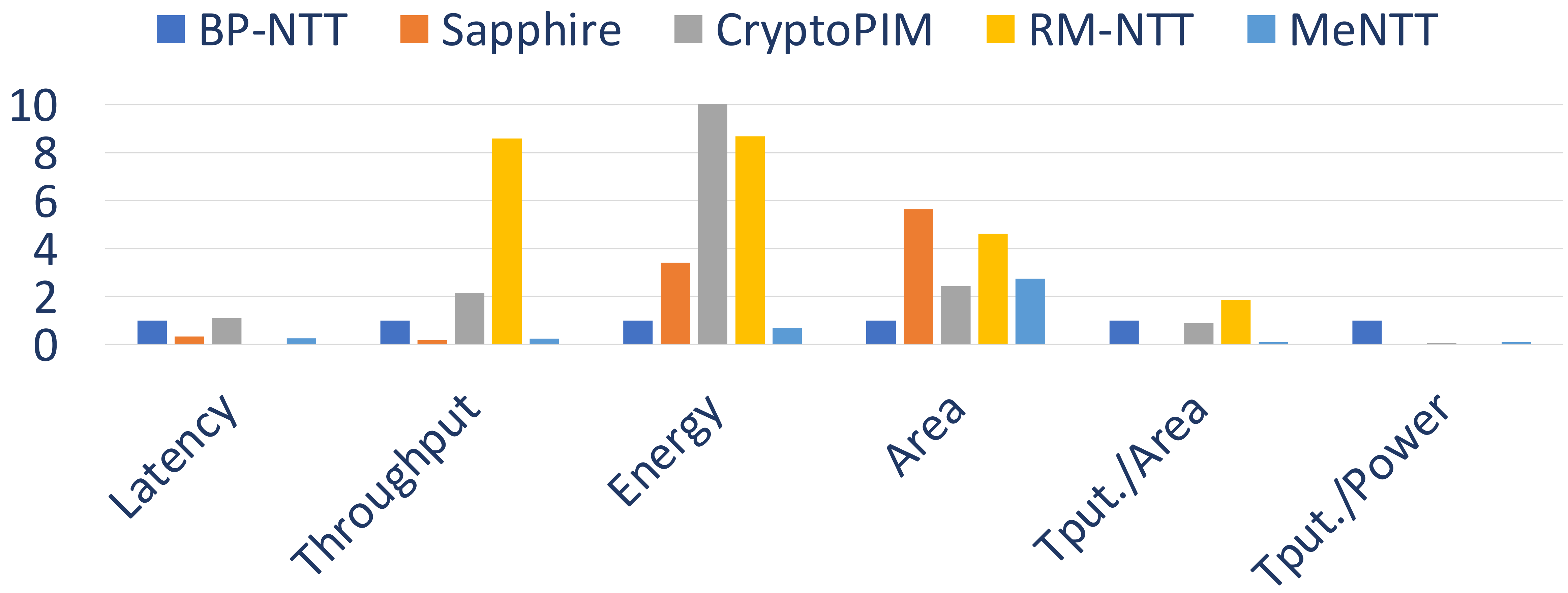
Comparison among Designs

- Results are normalized to BP-NTT



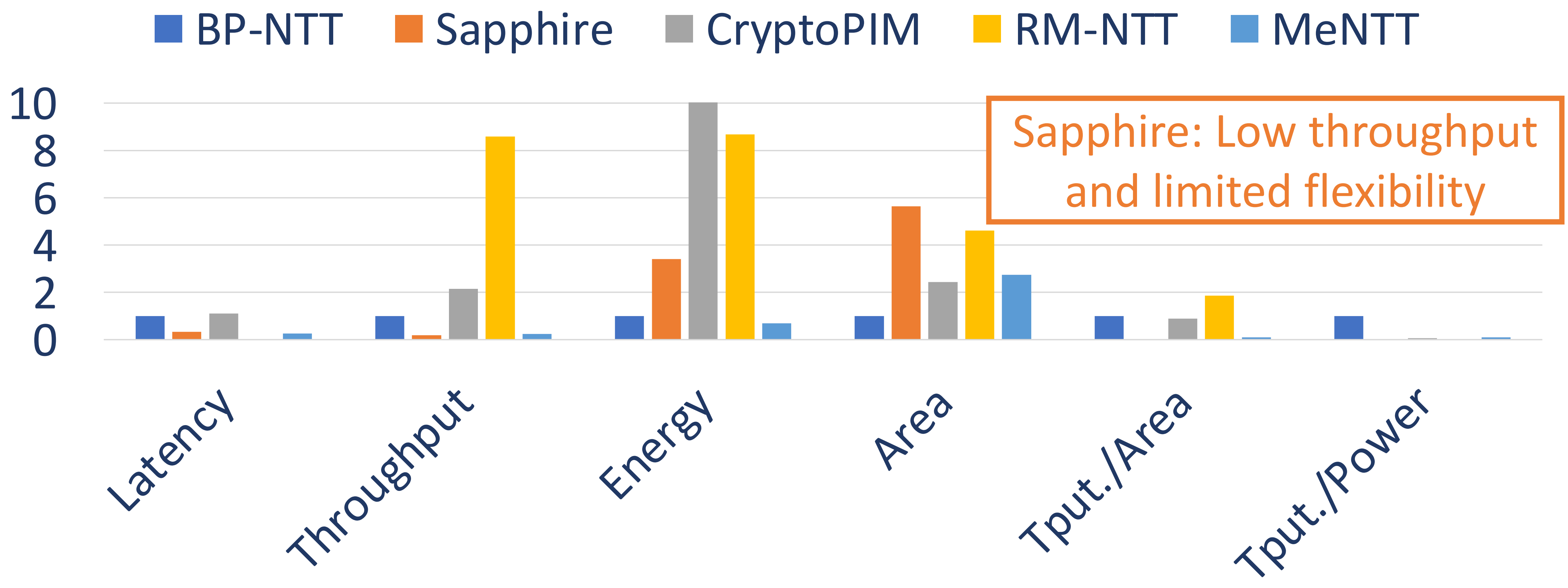
Comparison among Designs

- Results are normalized to BP-NTT



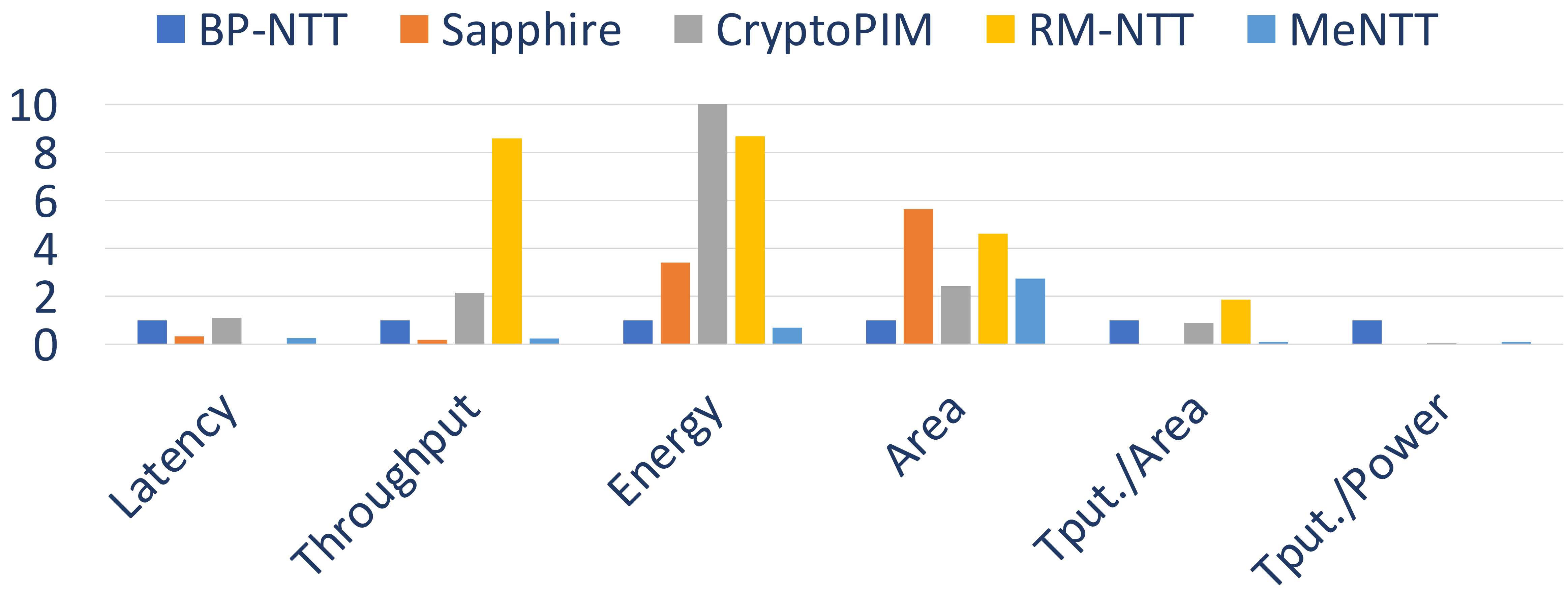
Comparison among Designs

- Results are normalized to BP-NTT



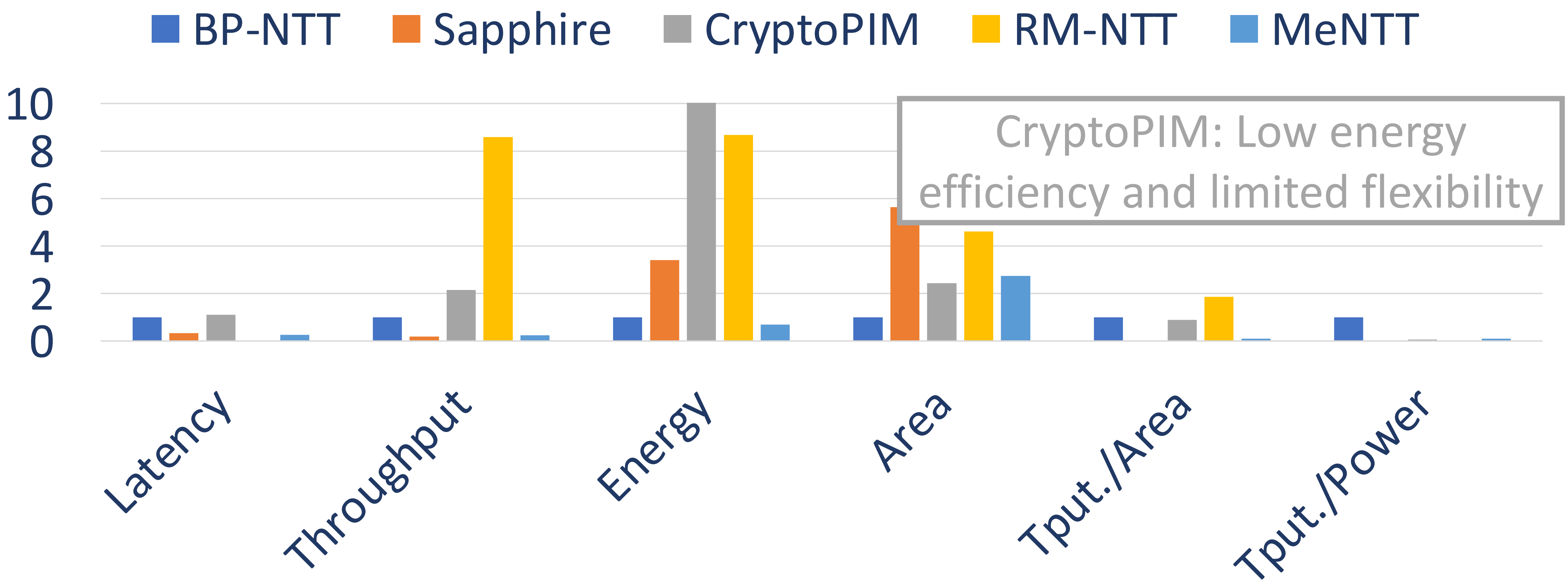
Comparison among Designs

- Results are normalized to BP-NTT



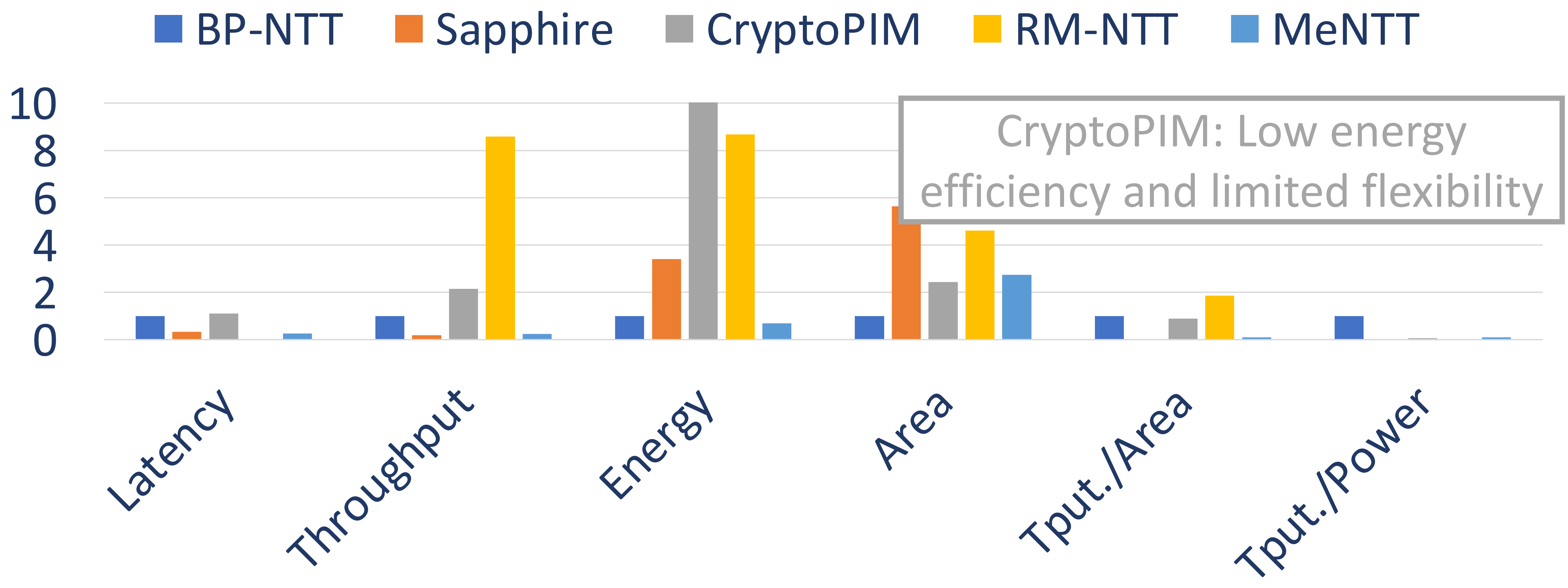
Comparison among Designs

- Results are normalized to BP-NTT



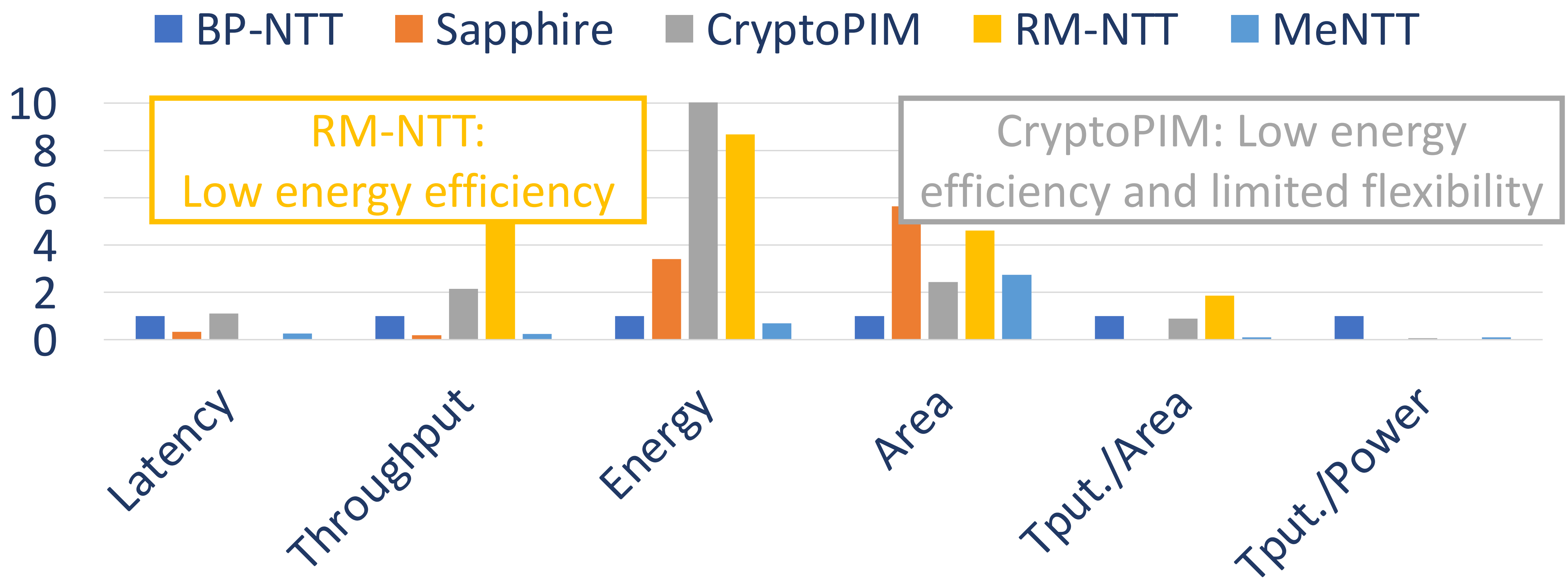
Comparison among Designs

- Results are normalized to BP-NTT



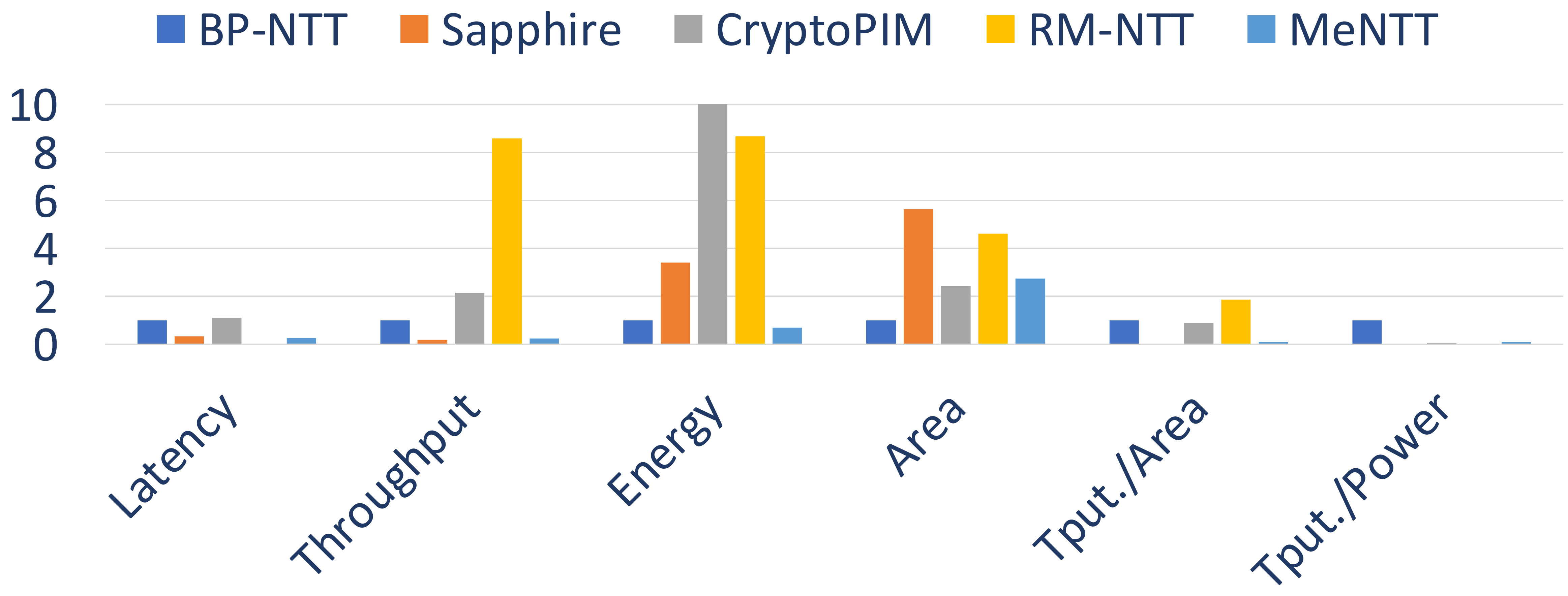
Comparison among Designs

- Results are normalized to BP-NTT



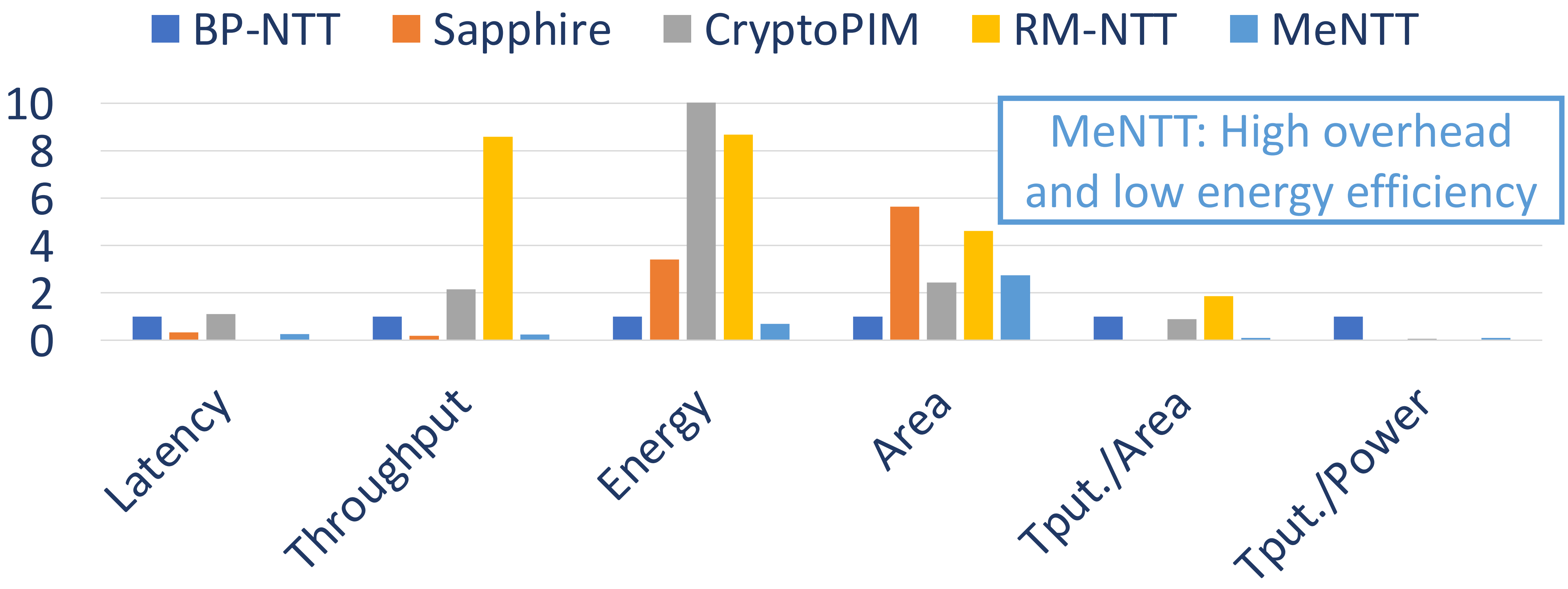
Comparison among Designs

- Results are normalized to BP-NTT



Comparison among Designs

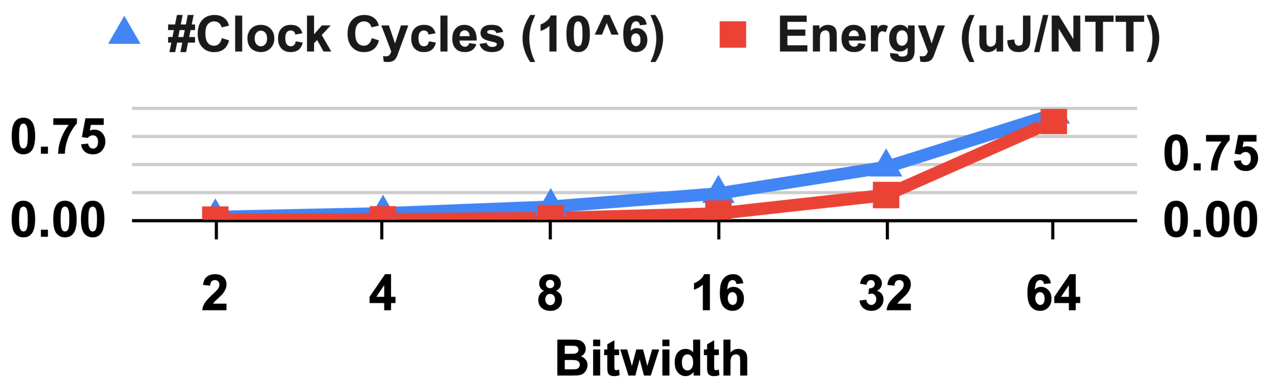
- Results are normalized to BP-NTT



Flexibility Analysis

Flexibility Analysis

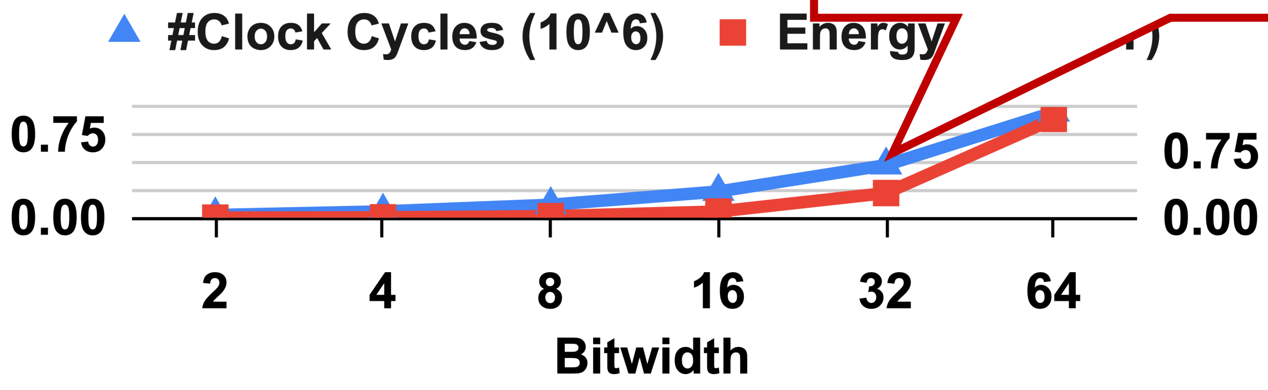
- With fixed polynomial order of 256



Flexibility Analysis

- With fixed polynomial order of 256

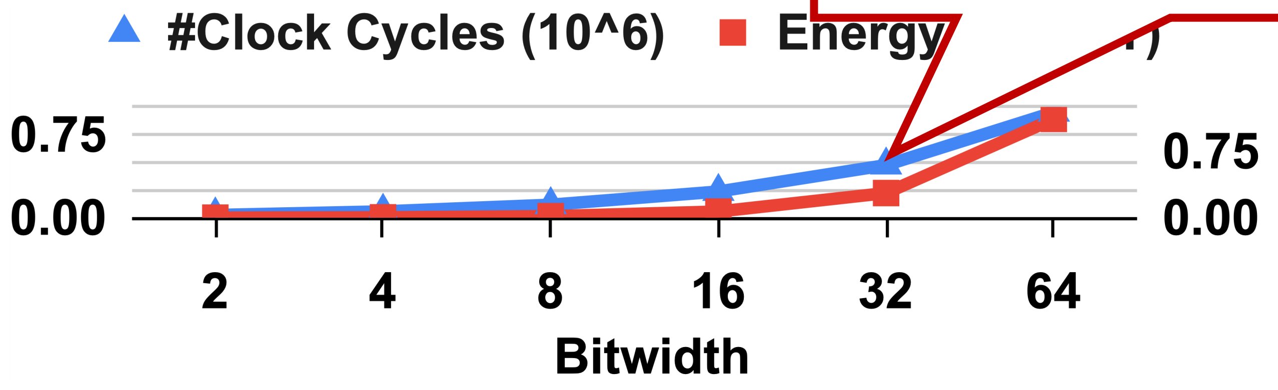
Parallelism decreases with increased bitwidth



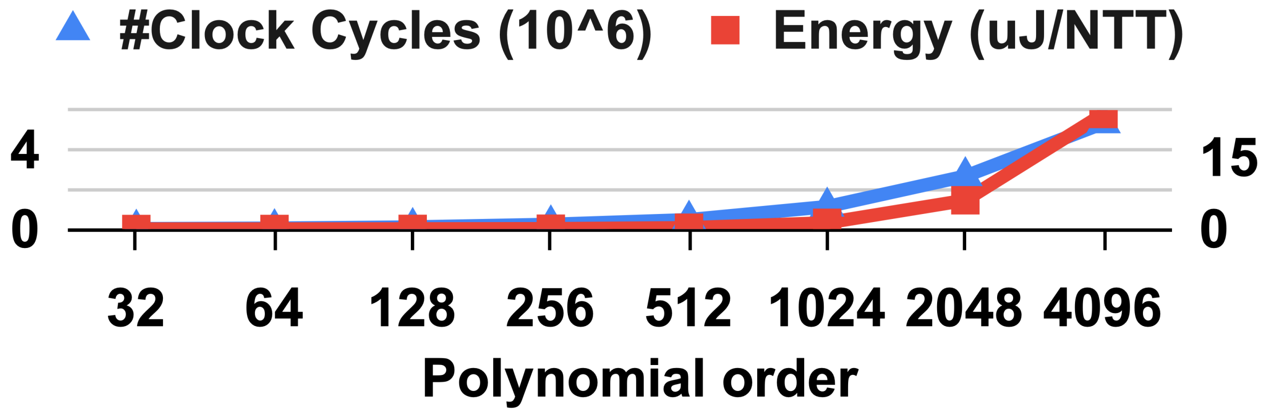
Flexibility Analysis

- With fixed polynomial order of 256

Parallelism decreases with increased bitwidth

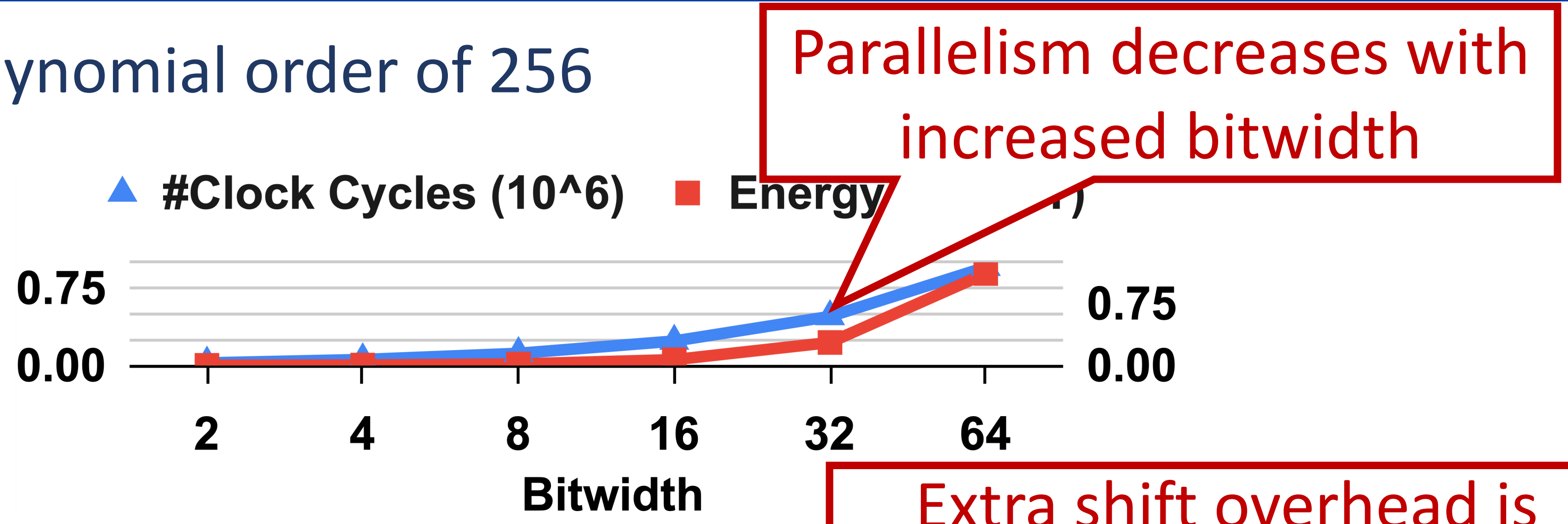


- With fixed bitwidth of 16

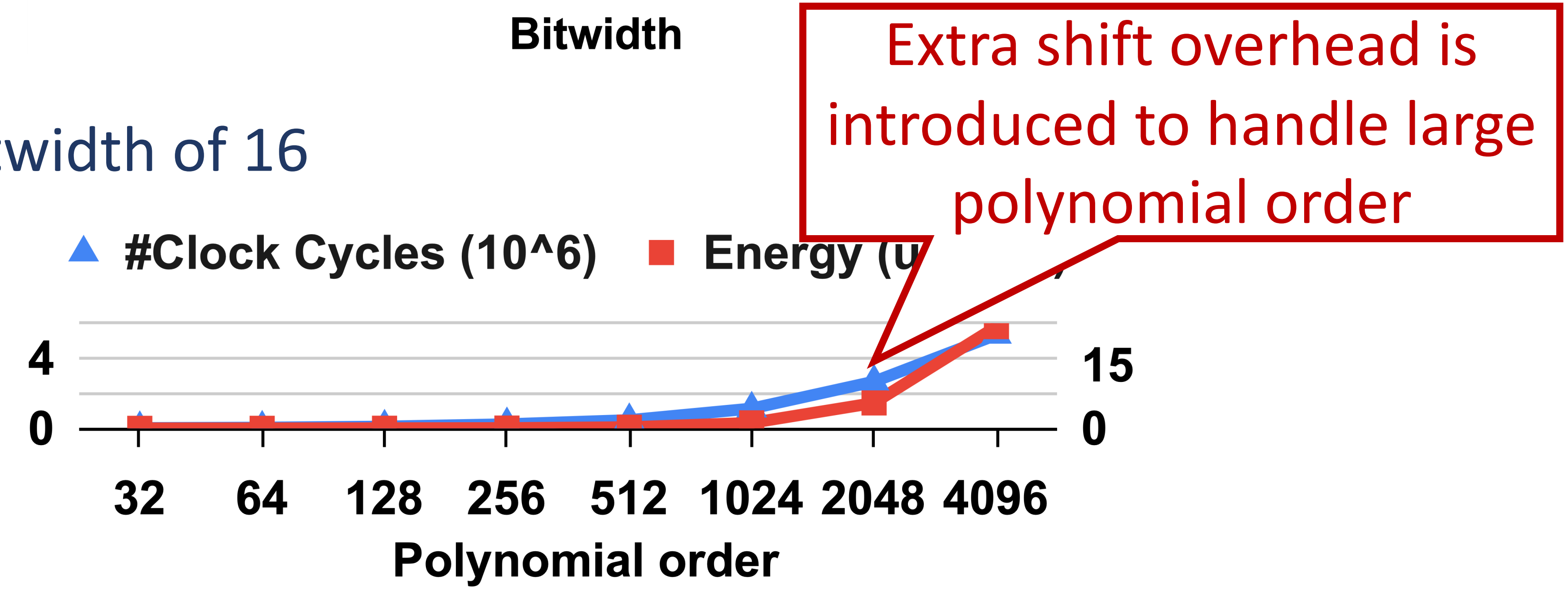


Flexibility Analysis

- With fixed polynomial order of 256



- With fixed bitwidth of 16



Conclusion

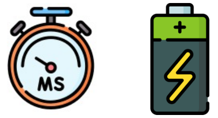
Conclusion

BP-NTT **repurposes LLC to perform bitline computing**

Conclusion

BP-NTT **repurposes LLC to perform bitline computing**

BP-NTT **uses shift-optimized data alignment**



Conclusion

BP-NTT repurposes LLC to perform bitline computing

BP-NTT uses shift-optimized data alignment

BP-NTT employs bit-parallel modular multiplication

Conclusion

BP-NTT **repurposes LLC to perform bitline computing**

BP-NTT **uses shift-optimized data alignment**

BP-NTT **employs bit-parallel modular multiplication**

BP-NTT **can achieve up to 138x throughput-per-power than state-of-the-art with minimal area**

Q&A