

CYBER-PHYSICAL SYSTEMS PROGRAMMING

Riccardo Gaspari Giulio Vignati Matteo Franchi

SCOPO DEL PROGETTO

Lo scopo del progetto è quello di realizzare ex novo 4 algoritmi crittografici sul cluster PULP:

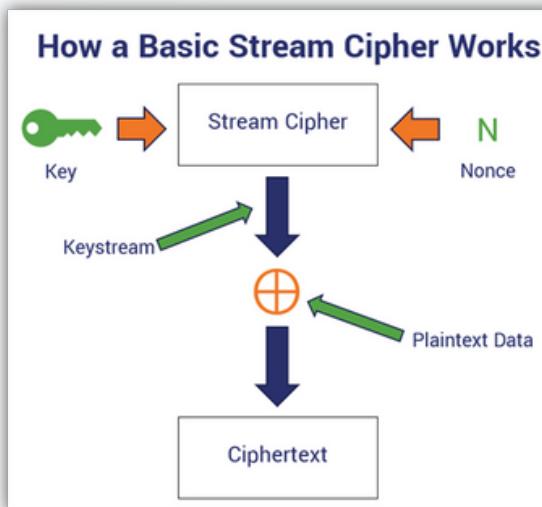
- AES-256-CTR,
- AES-256-GCM,
- ChaCha20,
- ChaCha20-Poly1305,

implementarne una versione parallelizzata sfruttando i core disponibili su GAP8 e valutare il guadagno raggiunto rispetto alla versione sequenziale.

AES CTR:

- 'Algoritmo di crittografia simmetrica con una chiave utilizzata per crittografare gli stessi dati;

- Stream cipher;

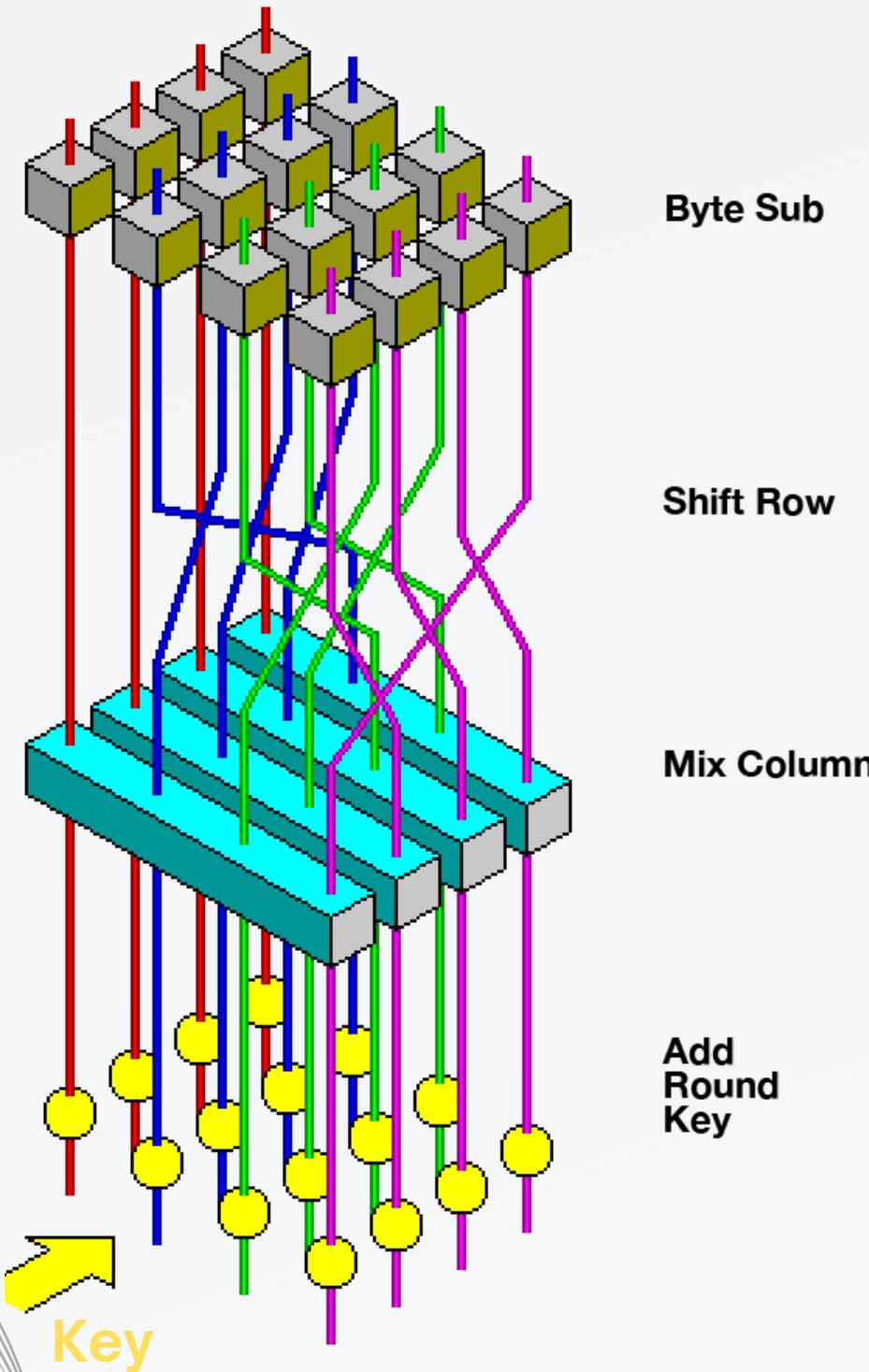


Ogni blocco è organizzato in una matrice 4x4 e viene ripetuto, per 14 round, un set di operazioni.

variante GCM:

Implementa anche l'autenticazione dei dati con un TAG di autenticazione (processo GHASH). Esso è utilizzato per garantire che i dati non siano stati modificati in modo non autorizzato.

CRIPTAZIONE AES



Ci sono 4 operazioni su matrici 4x4 che andremo ad eseguire:

- subBytes()
- shiftRows()
- mixColumns()
- addRoundKey()

CRIPTAZIONE AES

- **subBytes();** sostituisco ogni elemento della matrice con il corrispettivo della matrice sub().

The diagram illustrates the AES subBytes operation. On the left, an input matrix is shown as a 4x4 grid of bytes:

54	71	6b	6f
68	75	20	77
65	69	62	6e
20	63	72	20

An arrow points from this matrix to a second 4x4 matrix on the right, representing the output:

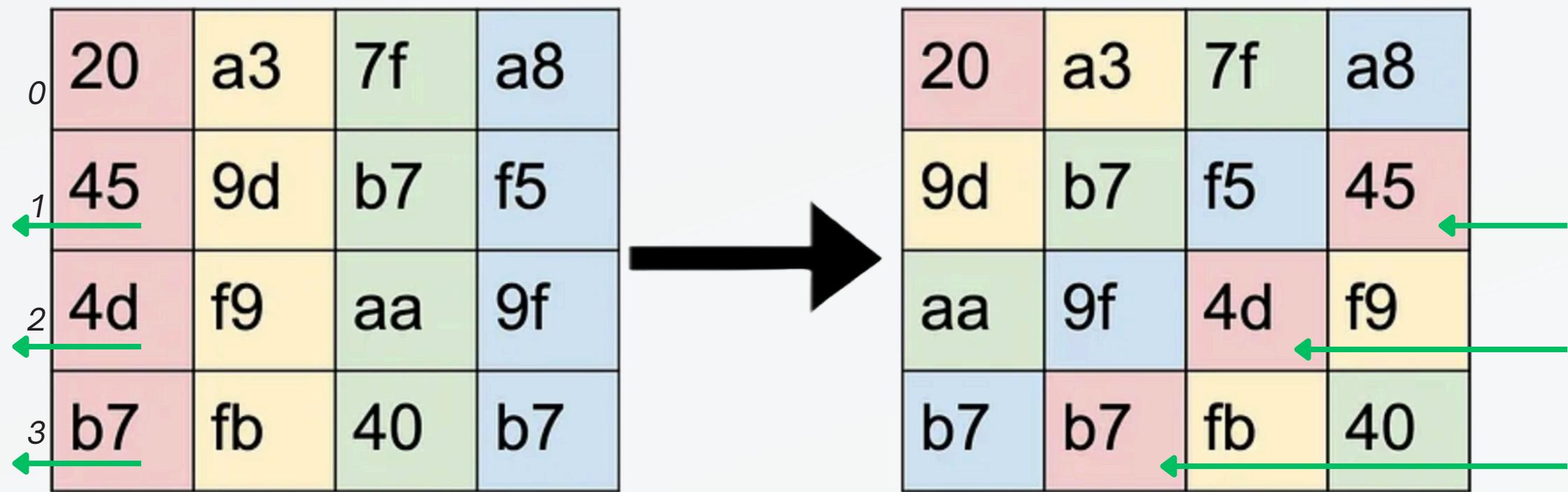
20	a3	7f	a8
45	9d	b7	f5
4d	f9	aa	9f
b7	fb	40	b7

A screenshot of the AES S-box substitution table is shown. The table maps each byte value (0-15) to its corresponding S-box value. The row for input byte 4 is highlighted with a red box, and the output value 20 is also highlighted with a red box.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
/* 0 */	0x63	0x7c	0x77	0x7b	0xf2	0x6b	0x6f	0xc5	0x30	0x01	0x67	0x2b	0xfe	0xd7	0xab	0x76
/* 1 */	0xca	0x82	0xc9	0x7d	0xfa	0x59	0x47	0xf0	0xad	0xd4	0xa2	0xaf	0x9c	0xa4	0x72	0xc0
/* 2 */	0xb7	0xfd	0x93	0x26	0x36	0x3f	0xf7	0xcc	0x34	0xa5	0xe5	0xf1	0x71	0xd8	0x31	0x15
/* 3 */	0x04	0xc7	0x23	0xc3	0x18	0x96	0x05	0x9a	0x07	0x12	0x80	0xe2	0xeb	0x27	0xb2	0x75
/* 4 */	0x09	0x83	0x2c	0x1a	0x1b	0x6e	0x5a	0xa0	0x52	0x3b	0xd6	0xb3	0x29	0xe3	0x2f	0x84
/* 5 */	0x53	0xd1	0x00	0xed	0x20	0xfc	0xb1	0x5b	0x6a	0xcb	0xbe	0x39	0x4a	0x4c	0x58	0xcf
/* 6 */	0xd0	0xef	0xaa	0xfb	0x43	0x4d	0x33	0x85	0x45	0xf9	0x02	0x7f	0x50	0x3c	0x9f	0xa8
/* 7 */	0x51	0xa3	0x40	0x8f	0x92	0x9d	0x38	0xf5	0xbc	0xb6	0xda	0x21	0x10	0xff	0xf3	0xd2
/* 8 */	0xcd	0x0c	0x13	0xec	0x5f	0x97	0x44	0x17	0xc4	0xa7	0x7e	0x3d	0x64	0x5d	0x19	0x73
/* 9 */	0x60	0x81	0x4f	0xdc	0x22	0x2a	0x90	0x88	0x46	0xee	0xb8	0x14	0xde	0x5e	0x0b	0xdb
/* A */	0xe0	0x32	0x3a	0x0a	0x49	0x06	0x24	0x5c	0xc2	0xd3	0xac	0x62	0x91	0x95	0xe4	0x79
/* B */	0xe7	0xc8	0x37	0x6d	0x8d	0xd5	0x4e	0xa9	0x6c	0x56	0xf4	0xea	0x65	0x7a	0xae	0x08
/* C */	0xba	0x78	0x25	0x2e	0x1c	0xa6	0xb4	0xc6	0xe8	0xdd	0x74	0x1f	0x4b	0xbd	0x8b	0x8a
/* D */	0x70	0x3e	0xb5	0x66	0x48	0x03	0xf6	0x0e	0x61	0x35	0x57	0xb9	0x86	0xc1	0x1d	0x9e
/* E */	0xe1	0xf8	0x98	0x11	0x69	0xd9	0x8e	0x94	0x9b	0x1e	0x87	0xe9	0xce	0x55	0x28	0xdf
/* F */	0x8c	0xa1	0x89	0x0d	0xbf	0xe6	0x42	0x68	0x41	0x99	0x2d	0x0f	0xb0	0x54	0xbb	0x16

CRIPTAZIONE AES

- **shiftRows();** spostiamo ogni riga verso sinistra di un numero di spazi corrispondente al numero di riga



CRIPTAZIONE AES

- `mixColumns();` Ora moltiplichiamo ogni colonna per la seguente matrice usando l'aritmetica GF(2^8) (`gmul()` e `XOR` invece delle normali addizioni e moltiplicazioni):

Vantaggi dell'operazione nel Galois Field:

- **velocità computazionale, semplici operazioni bitwise (XOR e shift);**
- **operazioni chiuse, associative, commutative e distributive;**
- **risultati uniformi e prevedibili senza overflow;**

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \cdot \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

$$\begin{aligned} d_0 &= 2 \cdot b_0 \oplus 3 \cdot b_1 \oplus 1 \cdot b_2 \oplus 1 \cdot b_3 \\ d_1 &= 1 \cdot b_0 \oplus 2 \cdot b_1 \oplus 3 \cdot b_2 \oplus 1 \cdot b_3 \\ d_2 &= 1 \cdot b_0 \oplus 1 \cdot b_1 \oplus 2 \cdot b_2 \oplus 3 \cdot b_3 \\ d_3 &= 3 \cdot b_0 \oplus 1 \cdot b_1 \oplus 1 \cdot b_2 \oplus 2 \cdot b_3 \end{aligned}$$

CRIPTAZIONE AES

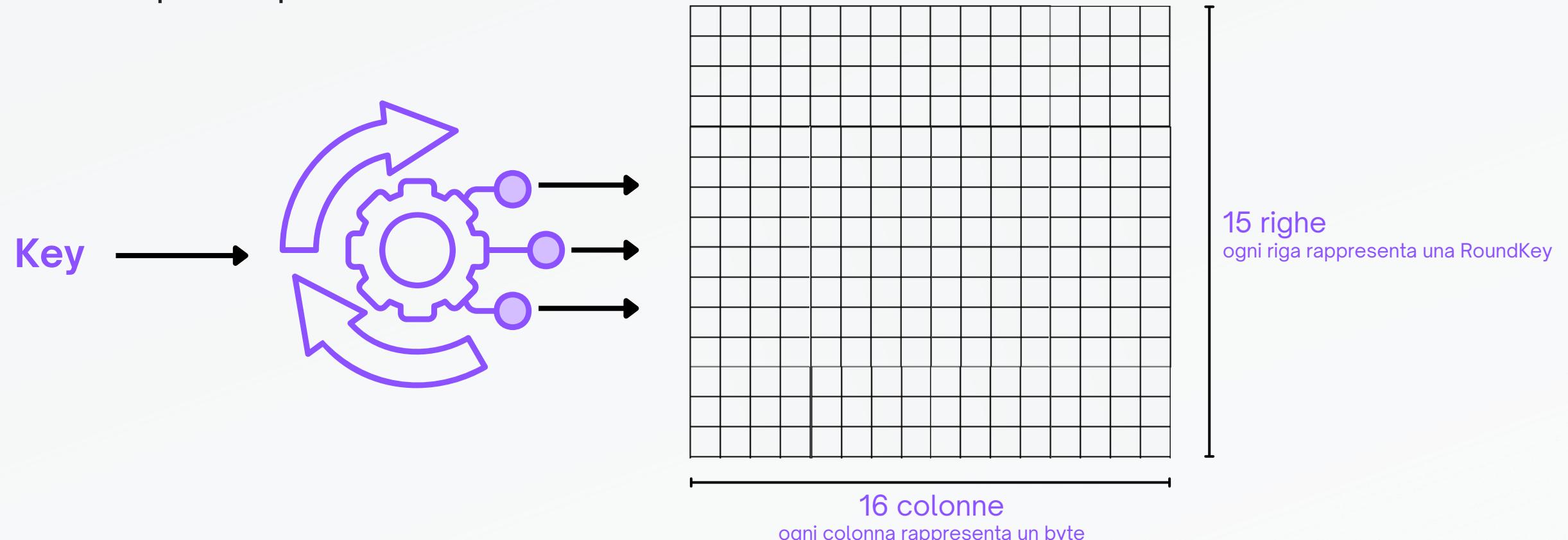
- **addRoundKey();** XOR ogni byte nel blocco con il rispettivo byte della RoundKey.

e1	b7	4c	3d
53	db	a2	72
30	f3	06	c4
22	a3	d4	df

→

80	d2	25	50
31	bd	c8	1c
53	94	d6	ab
46	cb	b8	af

RoundKey è un set di chiavi di round generate a valle di una serie di trasformazioni non lineari e permutazioni, partendo dalla chiave principale.



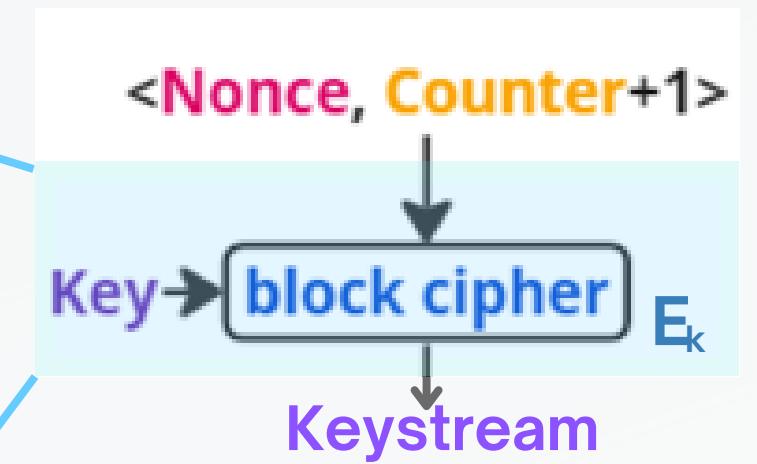
CRIPTAZIONE AES-CTR (COUNTER MODE)

- Generazione di un Counter, incrementato per ogni blocco tale da essere differente per ognuno di essi;
- Nonce/Initialization Vector, elemento arbitrario per inizializzare la criptazione;
- I due elementi vengono criptati tramite algoritmo AES. Dalla key viene internamente ottenuta la RoundKey usata in AES;

```
// Cipher is the main function for AES encryption
void Cipher(state_t* state, const uint8_t* RoundKey)
{
    uint8_t round = 0;

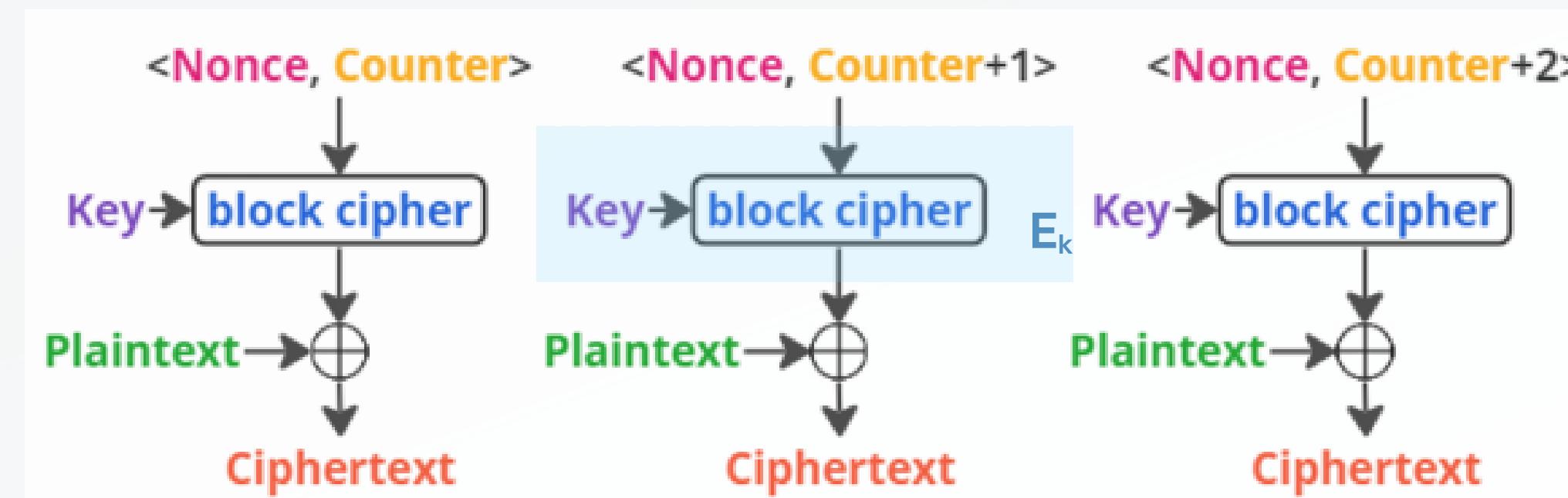
    // Add the First round key to the state before starting the rounds.
    AddRoundKey(0, state, RoundKey);

    // There will be Nr rounds.
    // The first Nr-1 rounds are identical.
    // These Nr rounds are executed in the loop below.
    // Last one without MixColumns()
    for (round = 1; ; ++round)
    {
        SubBytes(state);
        ShiftRows(state);
        if (round == Nr) {
            break;
        }
        MixColumns(state);
        AddRoundKey(round, state, RoundKey);
    }
    // Add round key to last round
    AddRoundKey(Nr, state, RoundKey);
}
```



CRIPTAZIONE AES-CTR

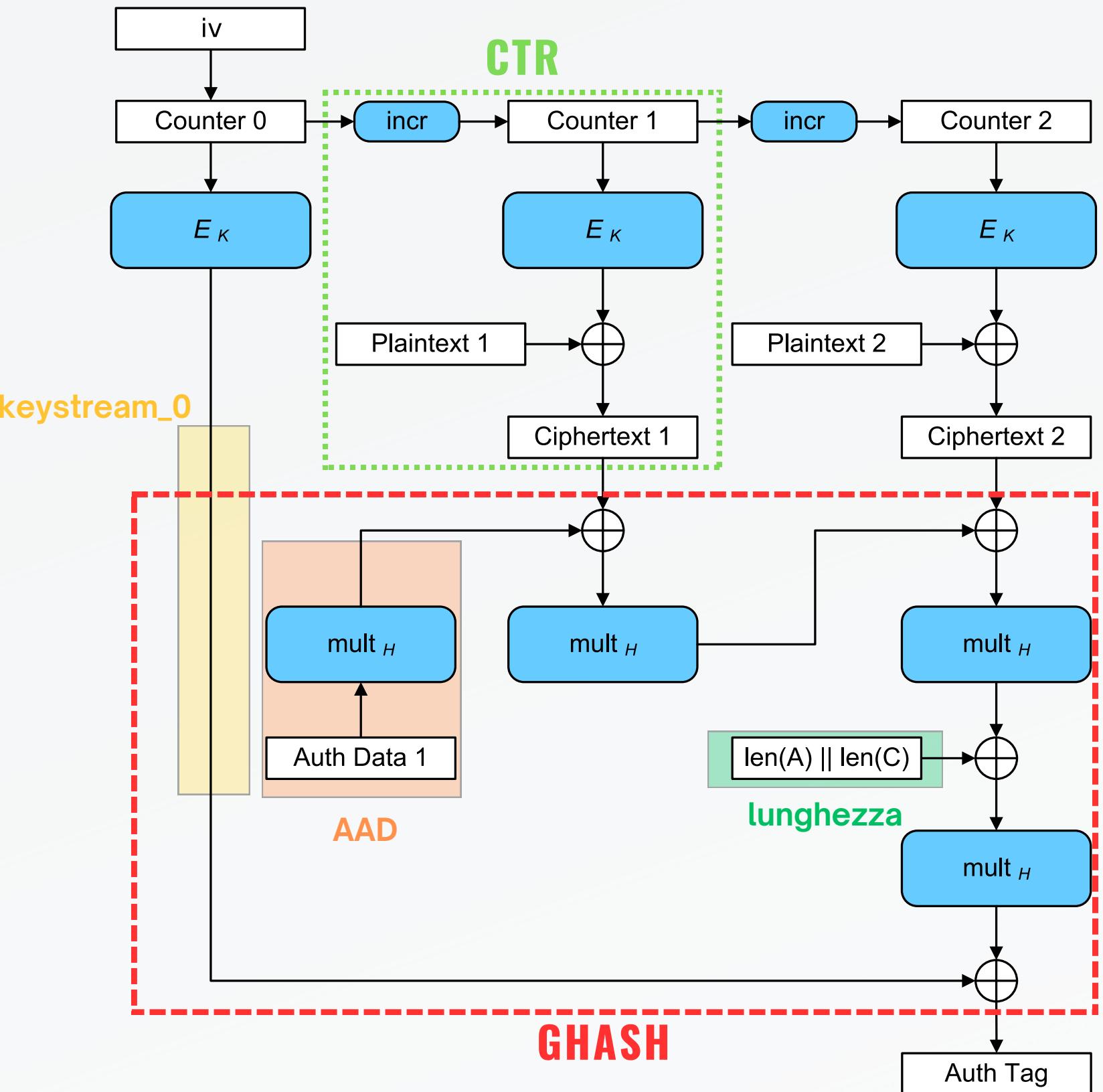
- Per ogni risultato di tale criptazione (**Keystream**) viene svolta l'operazione di XOR con il blocco di **Plaintext** corrispettivo per completare la criptazione



AES GCM (GALOIS/COUNTER MODE)

Processo di autenticazione: verifica basata su:

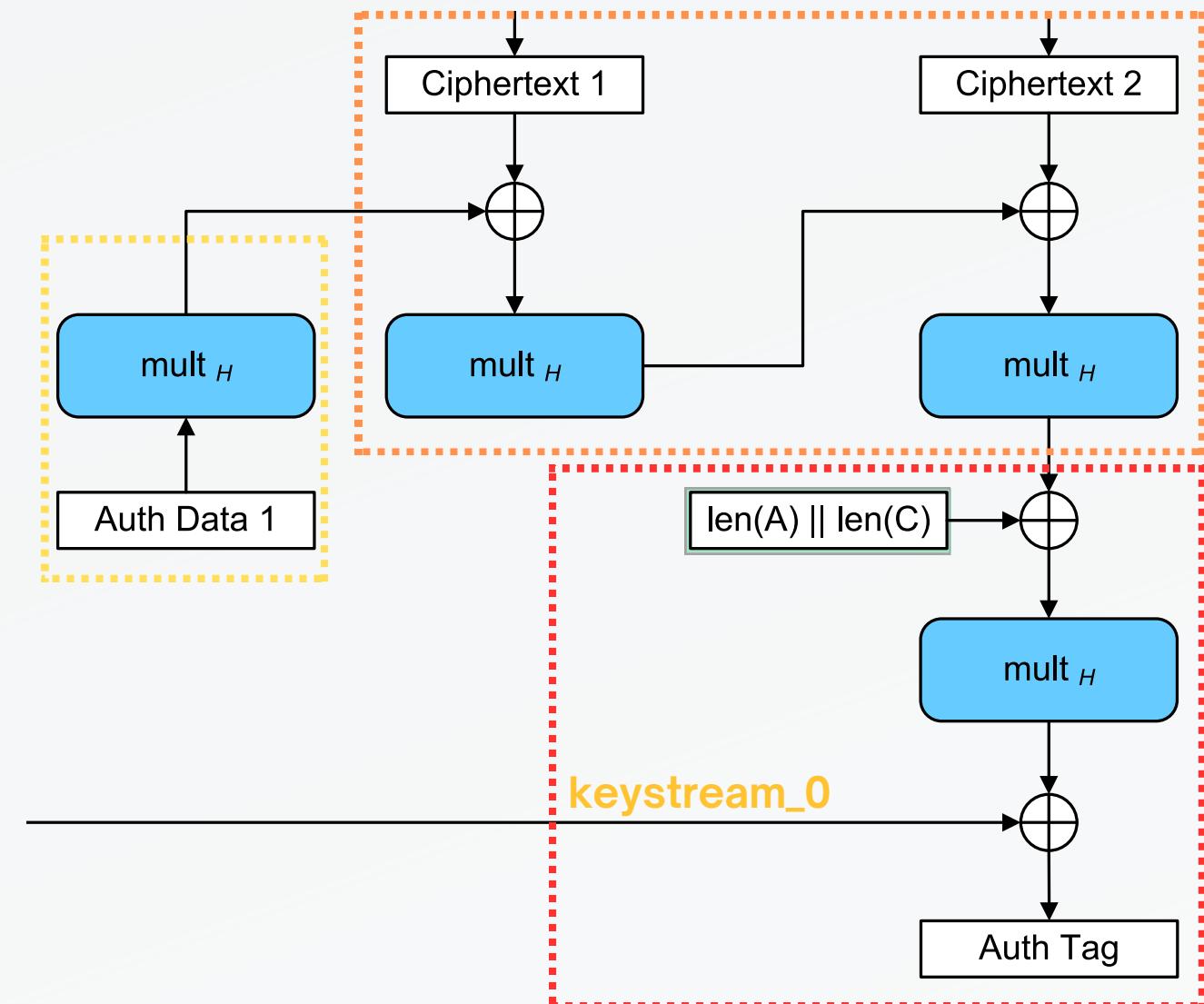
- Lunghezza del messaggio trasmesso ($\text{len}(A)$);
- Dati di autenticazione addizionali (AAD);
- Match del processo di criptazione del Ciphertext;



TAG GENERATION STRUCTURE:

La generazione del TAG è stata suddivisa in:

- INIT gestione del primo blocco di AAD;
- UPDATE con ulteriori blocchi di AAD e Ciphertext;
- FINISH contributo di $\text{len}(A)$ e keystream_0 ;



MOLTIPLICAZIONE IN GF(2^{128})

La hash_key è utilizzata durante l'operazione di moltiplicazione in GF(2^{128}), nella fase di autenticazione.

Moltiplicazione in GF(2^{128}):

1. *Rappresentazione*: Ogni elemento è un polinomio binario di grado < 128.
2. *Moltiplicazione*: Moltiplica i polinomi come normali polinomi binari.
3. *Riduzione Modulo*: Riduci il risultato modulo in un polinomio irriducibile
4. XOR per somma e shift per moltiplicazione

$$X^{128} + X^7 + X^2 + X + 1$$

```
// Define the irreducible polynomial R for GF(2^128)
#define R 0xE1 // R = 11100001

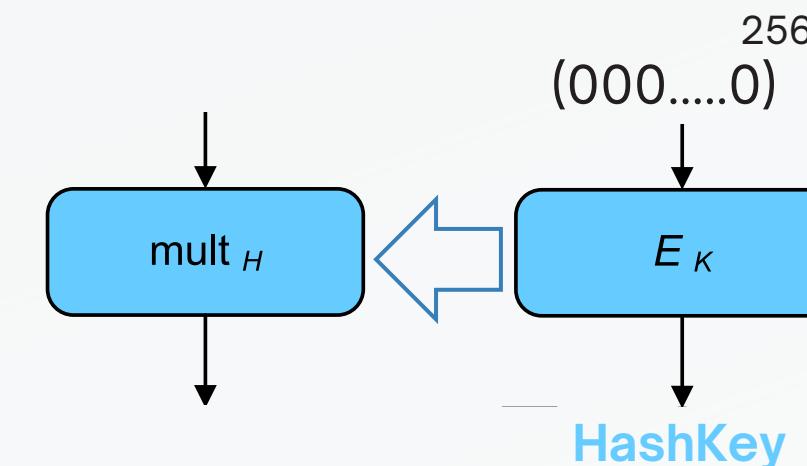
// Function to perform right shift
void right_shift(uint8_t V[16]) {
    for (int i = 15; i > 0; i--) {
        V[i] = (V[i] >> 1) | ((V[i - 1] & 0x01) << 7);
    }
    V[0] >>= 1;
}

// funzione della MUL_H
void gf128_mul(uint8_t AAD[16], uint8_t HashKey[16], uint8_t OUT[16]) {
    uint8_t V[16];

    // Copy AAD to V
    for (int i = 0; i < 16; i++) {
        V[i] = AAD[i];
    }

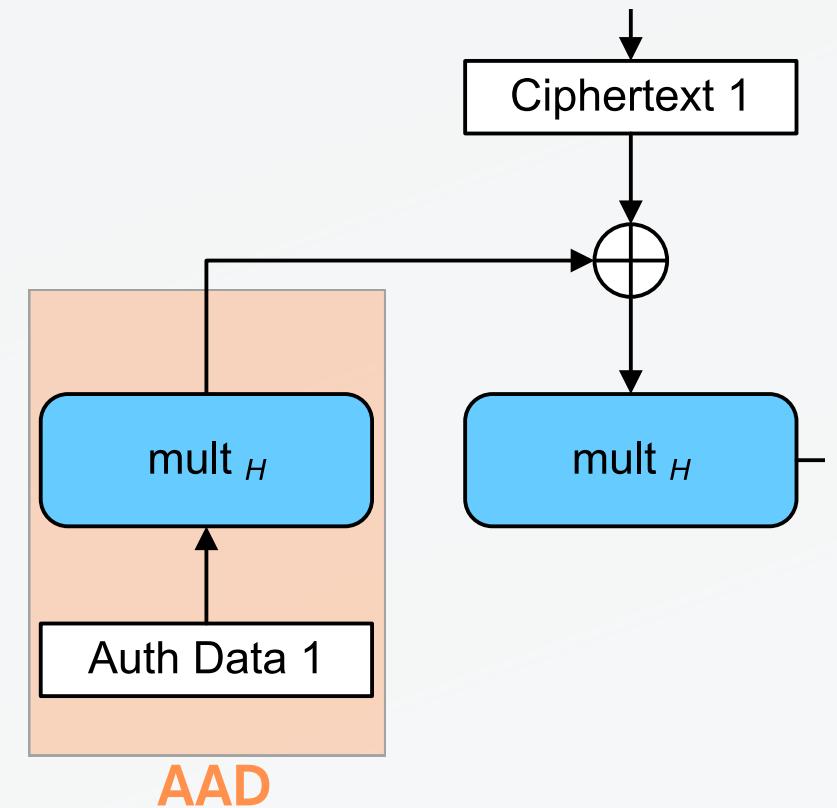
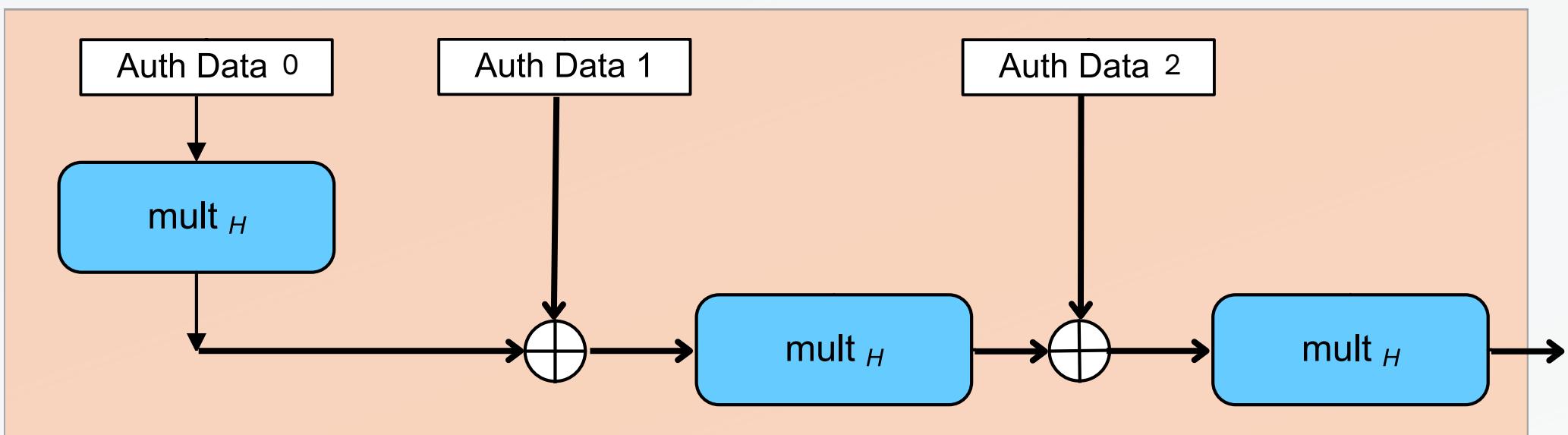
    // Initialize Z to 0
    for (int i = 0; i < 16; i++) {
        OUT[i] = 0;
    }

    // Perform the multiplication
    for (int i = 0; i < 128; i++) {
        if (HashKey[i / 8] & (1 << (7 - i % 8))) {
            // Yi = 1, perform Z ^ V
            for (int j = 0; j < 16; j++) {
                OUT[j] ^= V[j];
            }
        }
        if (V[15] & 0x01) {
            // V127 = 1, perform rightshift(V) ^ R
            right_shift(V);
            V[0] ^= R;
        } else {
            // V127 = 0, perform rightshift(V)
            right_shift(V);
        }
    }
}
```



ADDITIONAL AUTHENTICATED DATA

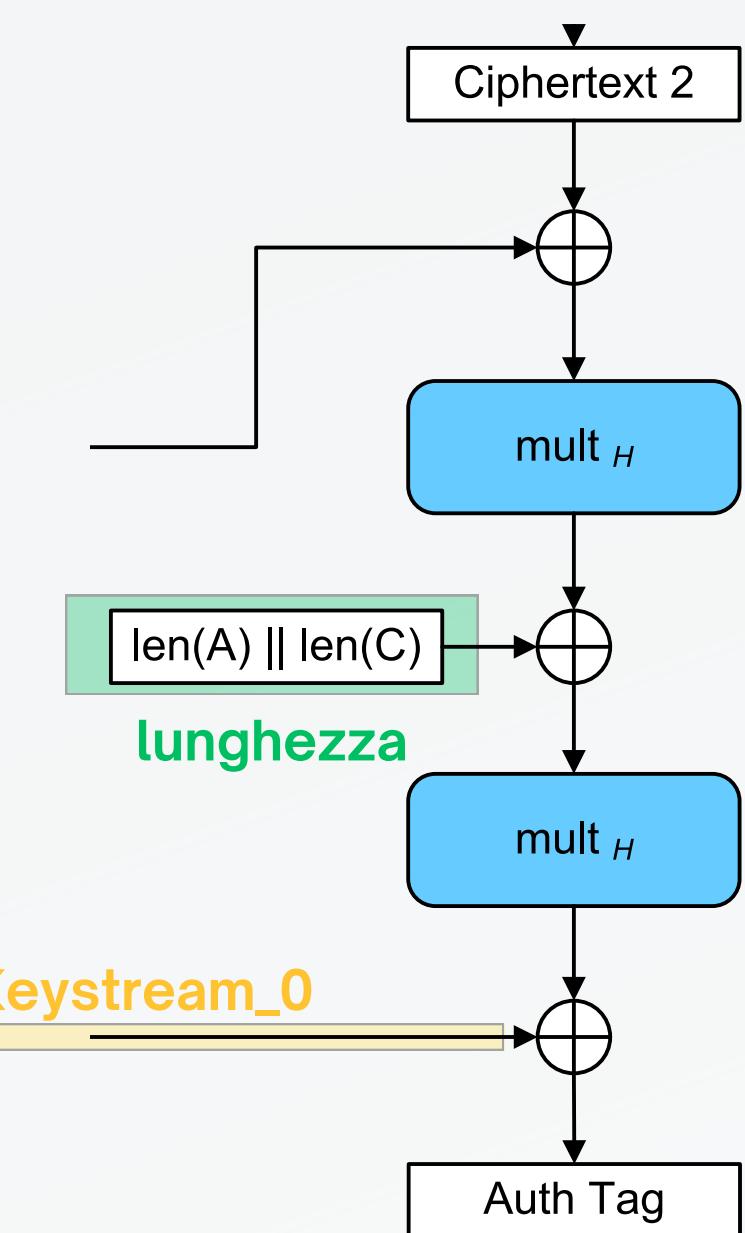
Gli AAD sono a loro volta divisi in blocchi e seguono il seguente processo di criptazione:



LUNGHEZZA E KEYSTREAM_0

Al termine delle operazioni tra AAD e Ciphertext, viene effettuata l'operazione di XOR con la **lunghezza** e infine con il **Keystream_0**.

Termina così il processo di autenticazione **GHASH** da cui si ottiene il Tag di autenticazione.



CHACHA20:

TIPO DI ALGORITMO

Approccio di crittografia simmetrico che crittografa e decrittografa i dati con la stessa chiave a 256 bit.

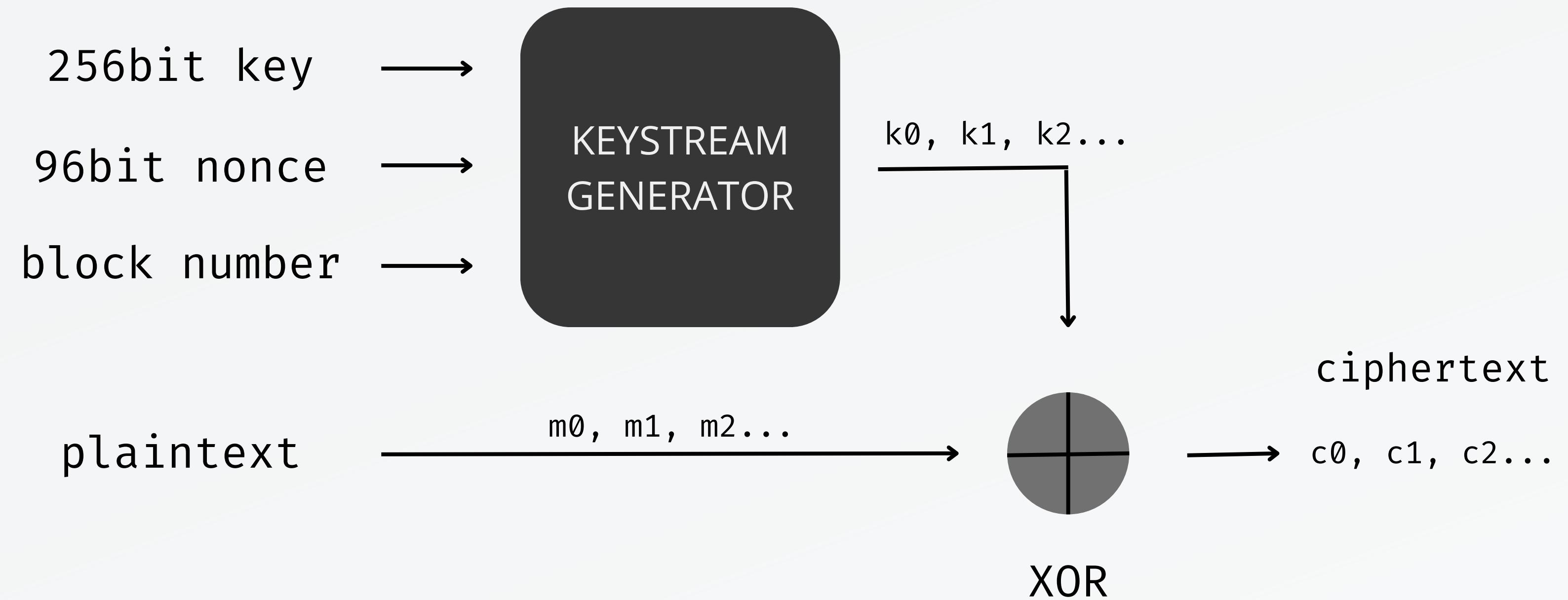
ChaCha20 è uno “stream cipher” (codice a flusso) che crittografa i dati in flussi continui anziché in blocchi di dimensione fissa.

Permette di avere un buon rapporto velocità-sicurezza e permette facilmente implementazioni in parallelo.

STREAM CIPHER WORKFLOW:

- ◆ Cifrario simmetrico
- ◆ Bit che codificano plaintext indipendenti l'uno dall'altro
- ◆ Trasformazione dei simboli successivi varia con il procedere della cifratura
- ◆ Ogni simbolo dipende da uno stato corrente
- ◆ Esecuzione più veloce rispetto alla struttura cifrari a blocchi
- ◆ Richiesta computazionale minore implica necessità di HW meno complesso

CHACHA20 BLOCK DIAGRAM:



QUARTER ROUND OPERATION:

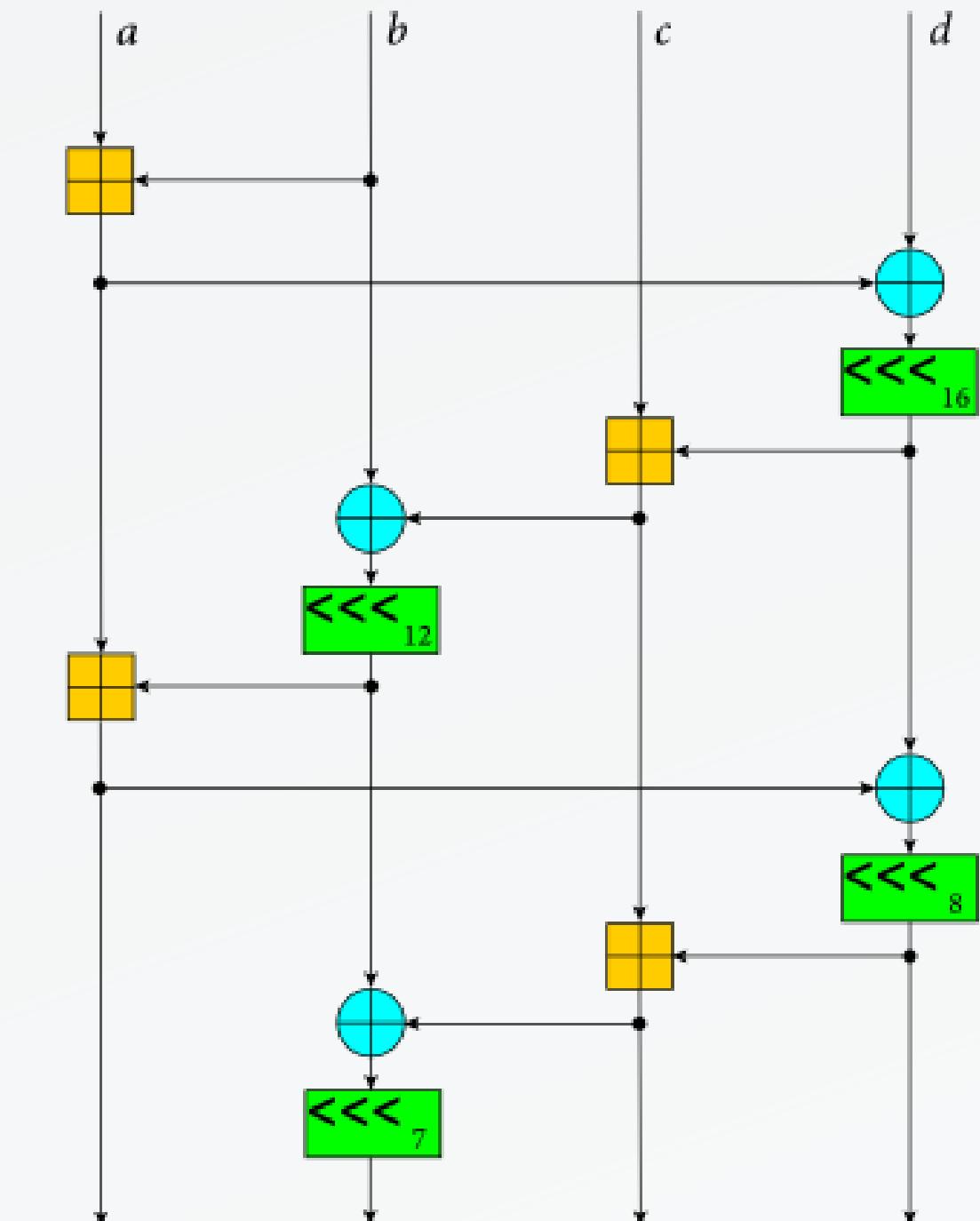
```
#define QUARTERROUND(x, a, b, c, d) \
    x[a] += x[b]; x[d] = ROTL32(x[d] ^ x[a], 16); \
    x[c] += x[d]; x[b] = ROTL32(x[b] ^ x[c], 12); \
    x[a] += x[b]; x[d] = ROTL32(x[d] ^ x[a], 8); \
    x[c] += x[d]; x[b] = ROTL32(x[b] ^ x[c], 7);
```

Opera su integer 32bit

"+" indica una somma tra interi modulo 2^{32}

" \wedge " indica un bitwise Exclusive OR (XOR)

"<<< n" indica n-bit rotazione verso sinistra
(possibile grazie alla funzione ROTL32)



CHACHA20 INITIAL STATE:

Lo stato di C20 è rappresentabile attraverso una matrice 4x4 con ogni cella da 32bit.

cost	cost	cost	cost
key[0]	key[1]	key[2]	key[3]
key[4]	key[5]	key[6]	key[7]
count	nonce[0]	nonce[1]	nonce[2]

La key viene importata in blocchi da 4 byte in little endian.

counter garantisce output diverso per ogni blocco

counter da 32bit può tenere traccia di fino a 2^{32} blocchi di dati prima di esaurire lo spazio, un blocco in C20 è di 64 byte, un contatore a 32 bit è sufficiente per cifrare fino a 256GB di dati.

CHACHA20 KEYSTREAM GENERATION:

ChaCha20 esegue 20 rounds, alternando tra "column rounds" e "diagonal rounds" ognuno consiste di 4 quarter-rounds.

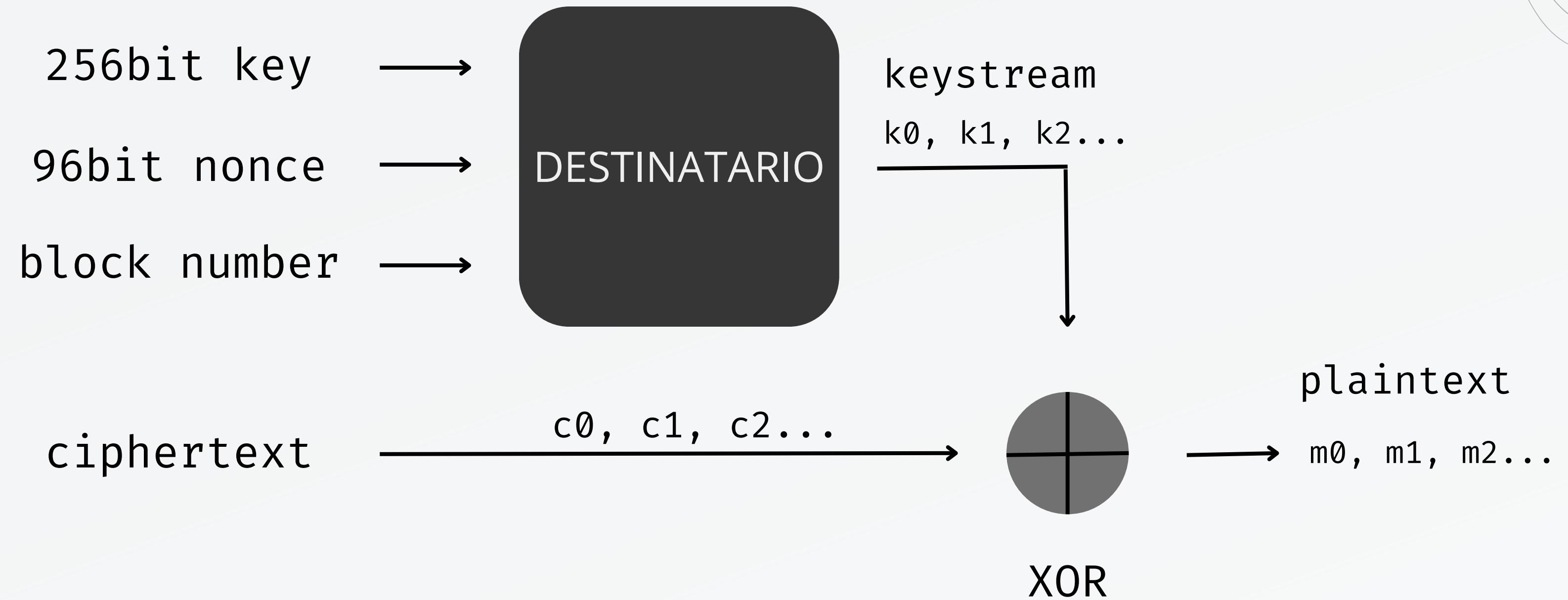
```
for (int i = 0; i < 20; i += 2) {
    QUARTERROUND(state, 0, 4, 8, 12)
    QUARTERROUND(state, 1, 5, 9, 13)
    QUARTERROUND(state, 2, 6, 10, 14)
    QUARTERROUND(state, 3, 7, 11, 15)
    QUARTERROUND(state, 0, 5, 10, 15)
    QUARTERROUND(state, 1, 6, 11, 12)
    QUARTERROUND(state, 2, 7, 8, 13)
    QUARTERROUND(state, 3, 4, 9, 14)
}

for (int i = 0; i < 16; ++i)
    out[i] += state[i];
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

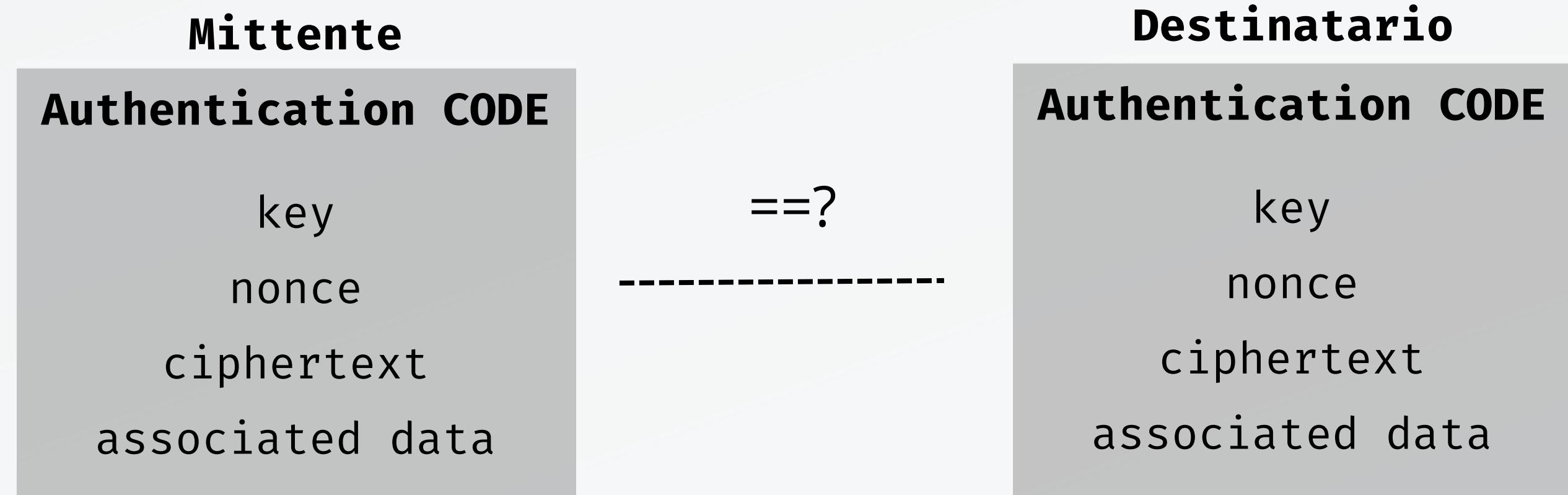
Finita questa operazione viene generato il keystream da mettere in xor con il plaintext.

CHACHA20 DECRYPTION:

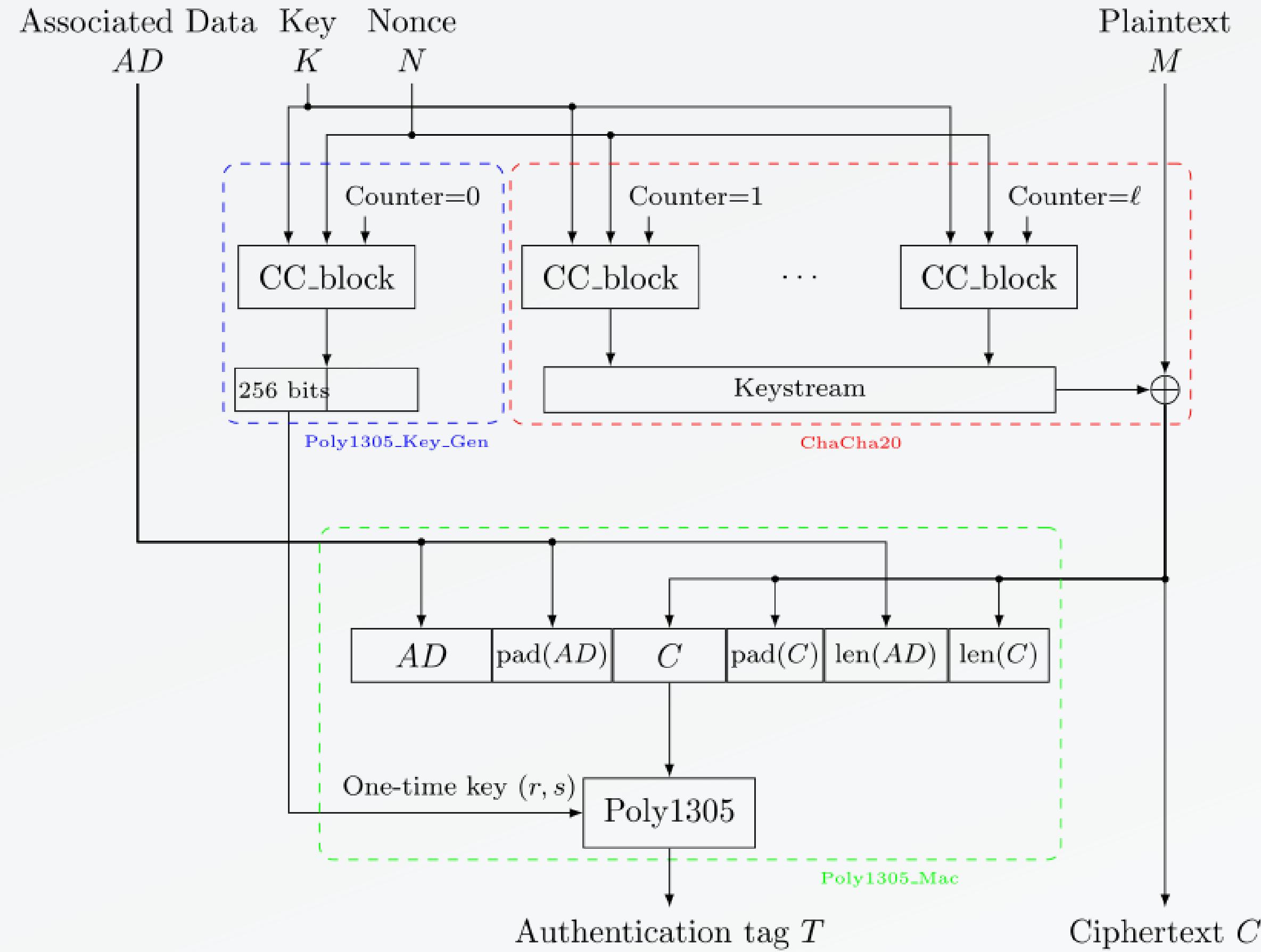


CHACHA20-POLY1305:

- ◆ Combina algoritmo Chacha20 con autenticazione Poly1305
- ◆ Garantisce maggiore sicurezza
- ◆ L'autenticazione verifica l'integrità dei dati



C20P1305 STRUCTURE:



C20P1305 STRUCTURE:

poly1305_context

```
uint32_t r[5];  
  
uint32_t h[5];  
  
uint32_t pad[4];  
  
size_t leftover;  
  
unsigned char buffer[16];  
  
unsigned char final;
```

Secret key Poly1305 → array of five 32-bit integers

Current State → array of five 32-bit integers

Padding → array of four 32-bit integers

Conta bit non processati nel blocco precedente

Buffer dati in ingresso

Flag che indica se blocco è stato processato

C20P1305 STRUCTURE:



Inizializza algoritmo con key e nonce

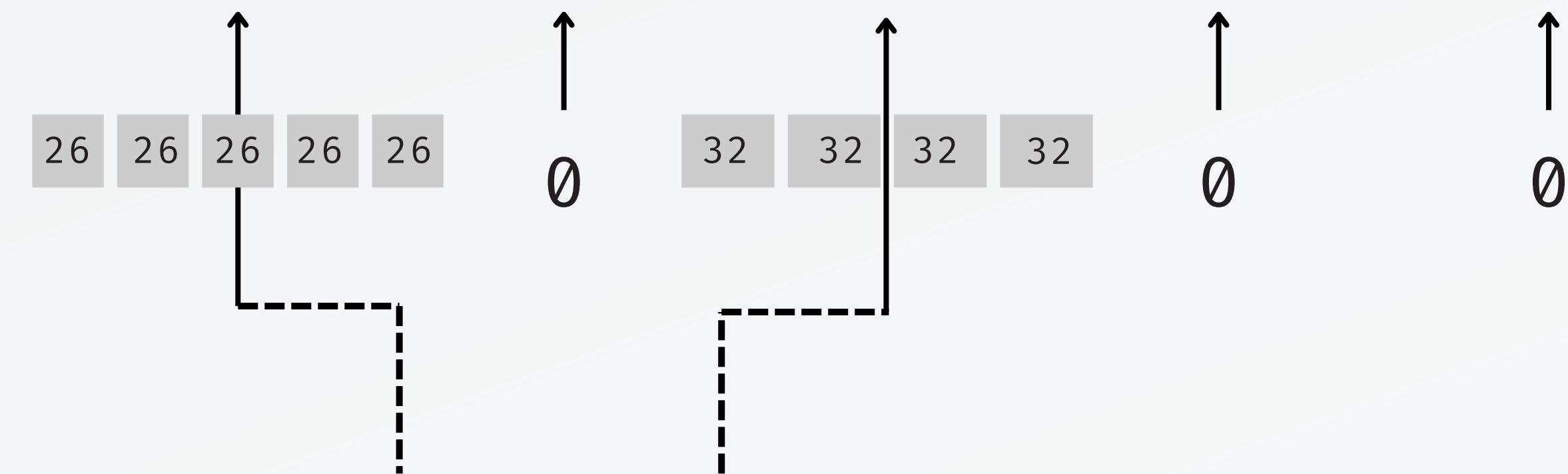
Aggiornamento hash:
chiamata ogni volta che si deve cifrare nuovo blocco

Finalizzazione hash:
restituisce hash finale utilizzato per la verifica

C20P1305 INIT FUNCTION:

init

key state pad leftover final

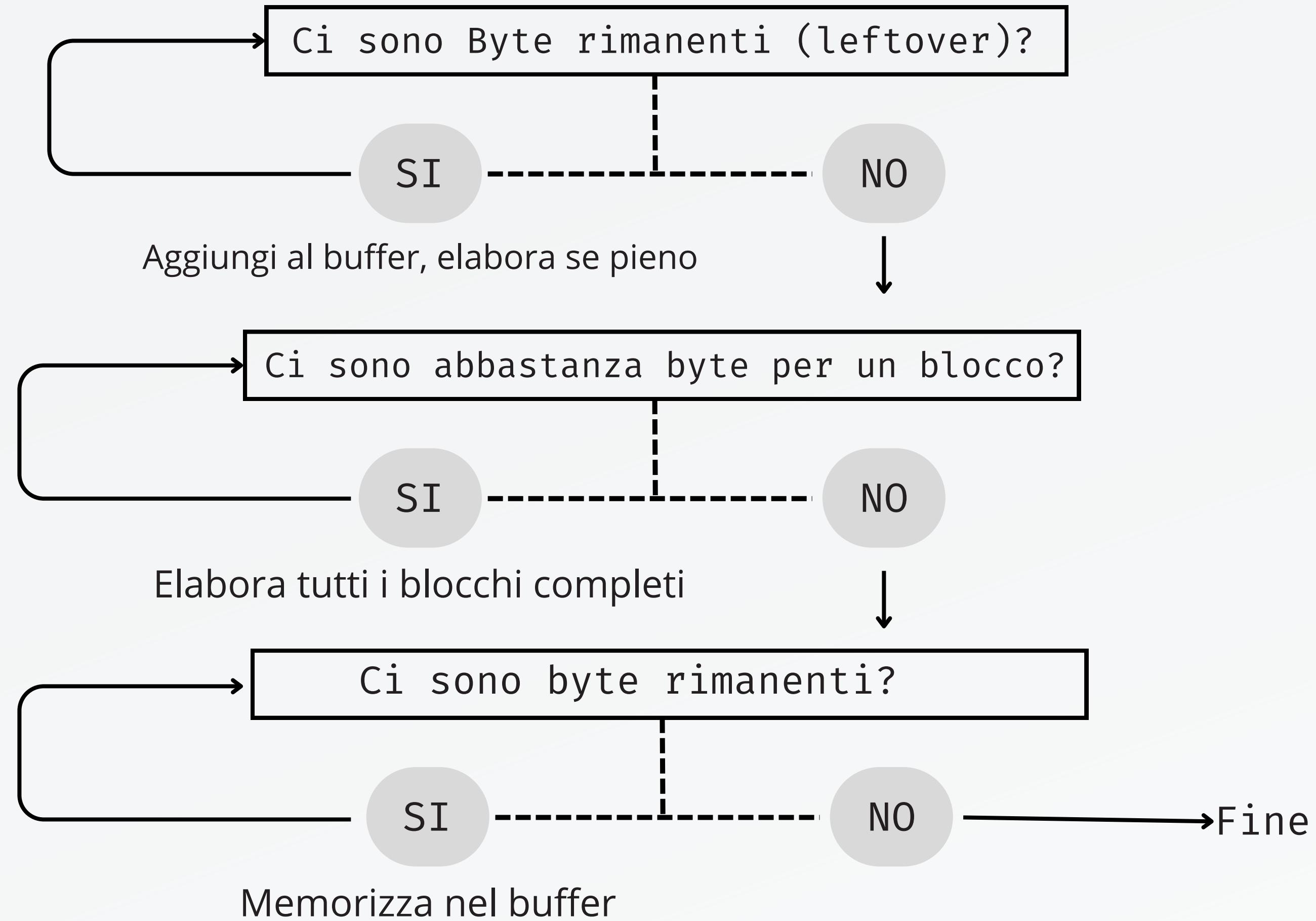


key

16byte 16byte

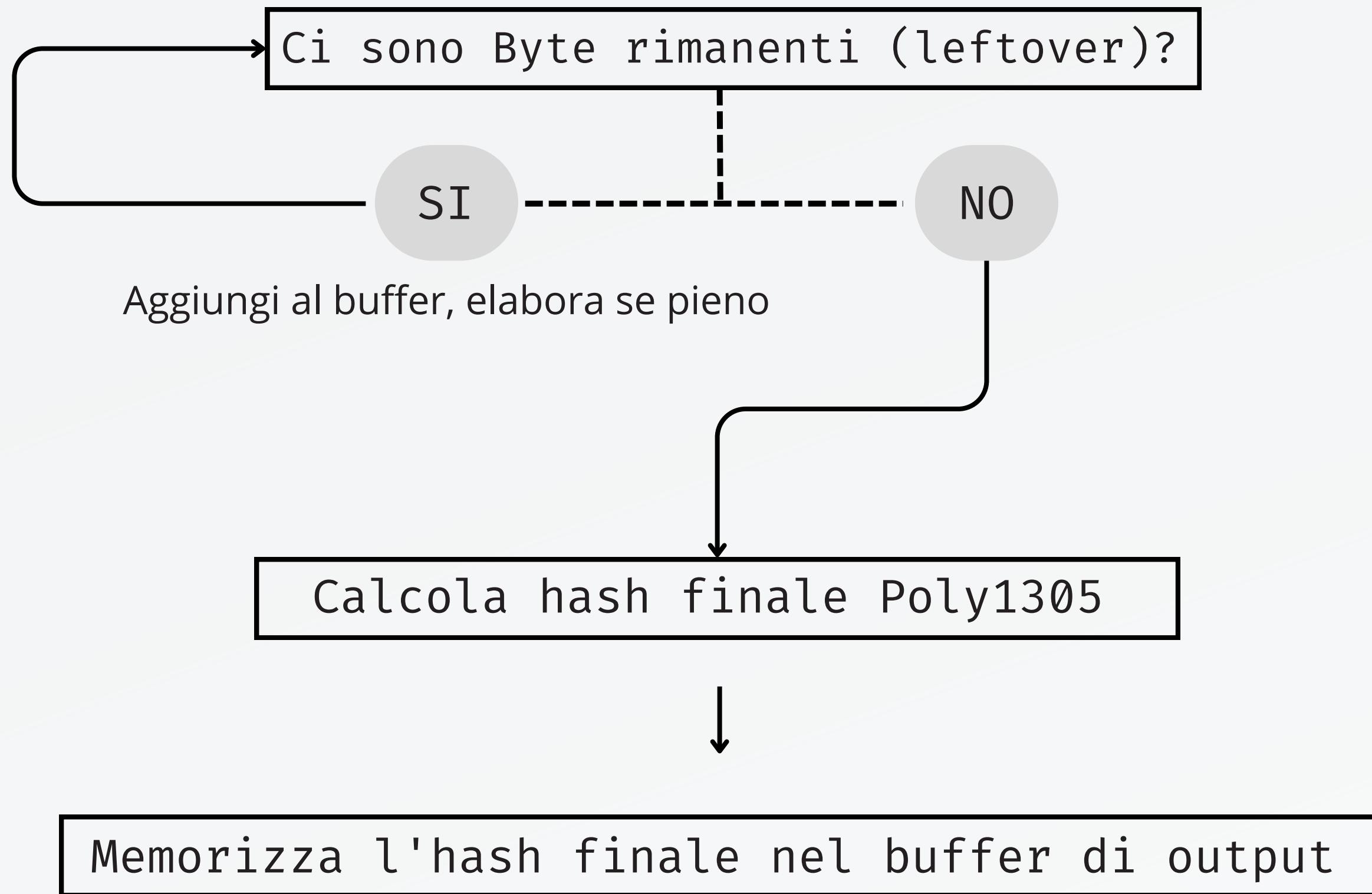
C20P1305 UPDATE FUNCTION:

update



C20P1305 FINISH FUNCTION:

finish



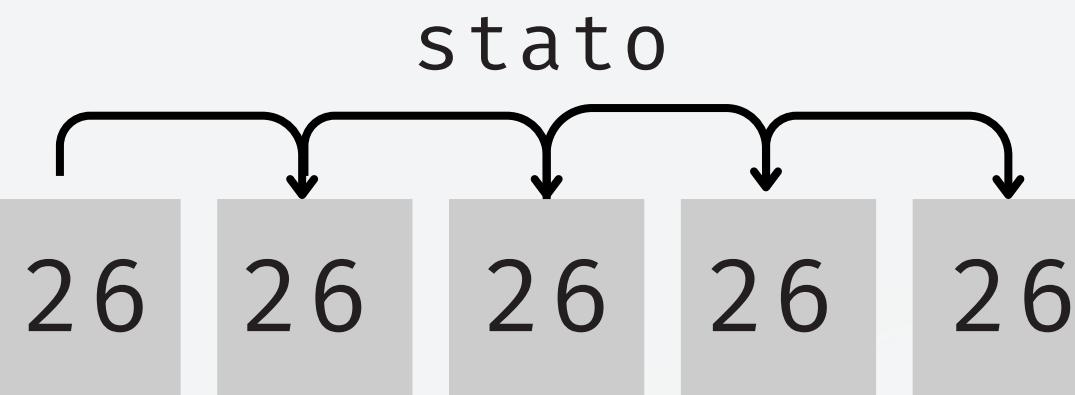
C20P1305 FINISH FUNCTION:

finish

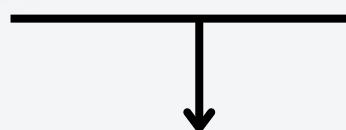
Calcola hash finale Poly1305



Memorizza l'hash finale nel buffer di output



h₀ h₁ h₂ h₃ h₄

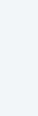


c = h₁ >> 26

h₁ = h₁ & 0x3fffff

h₂ += c

Aggiornamento dello stato corrente



Prevenzione Overflow

Calcolo di g = h + (-p)



g abbastanza grande da diventare negativo?

SI

Uso h

NO

Uso g

C20P1305 FINISH FUNCTION:

finish

Calcola hash finale Poly1305



Memorizza l'hash finale nel buffer di output

"h" o "g"

26 26 26 26 26

§0 §1 §2 §3 §4

#0 = Resto(§0 **or** §1)

#1 = Resto(§1 **or** §2)

⋮
⋮

Ogni elemento del vettore <2^32

Riduzione di valore scelto

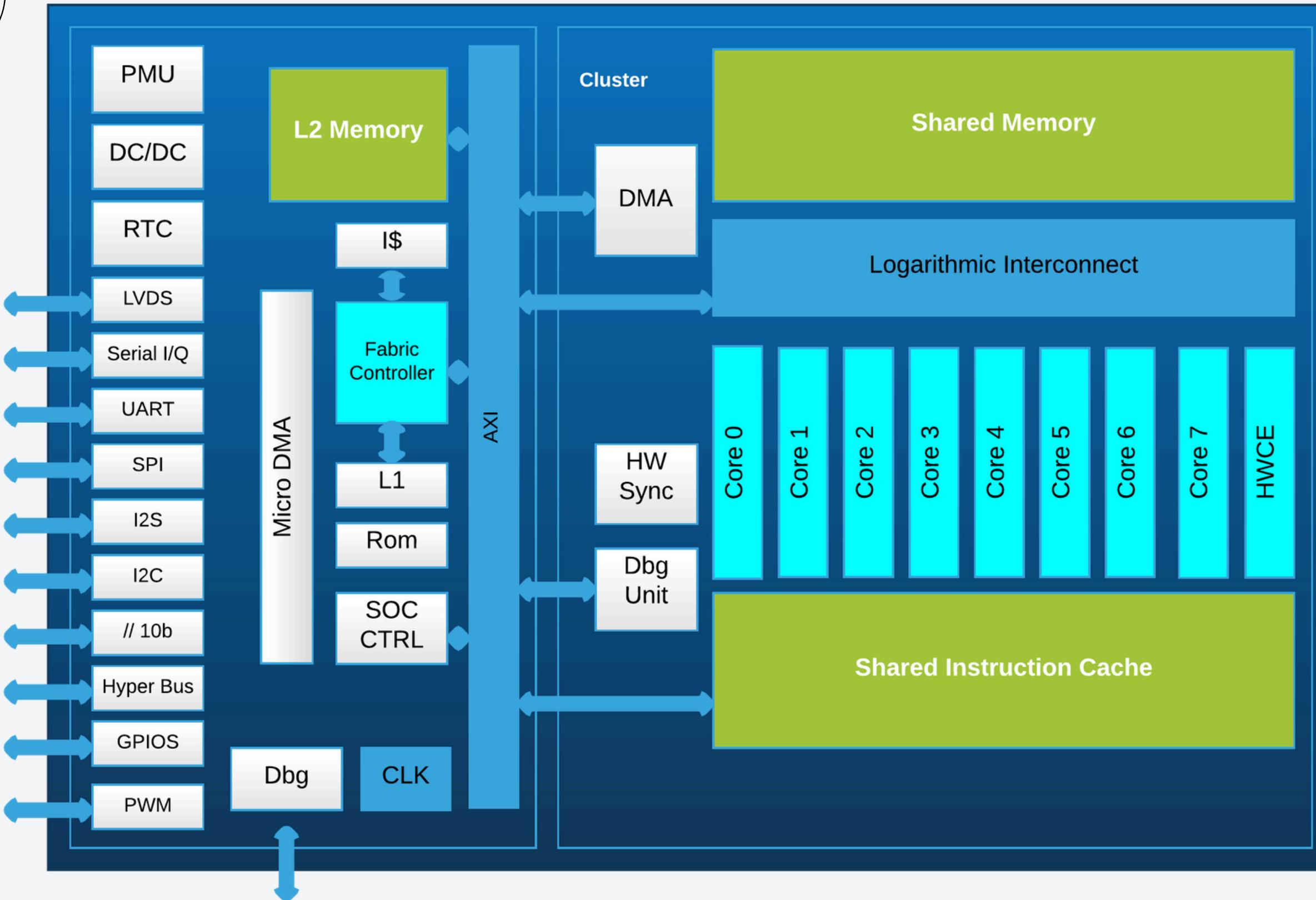
Calcolo TAG:

tag = (§ + pad) % (2^128)



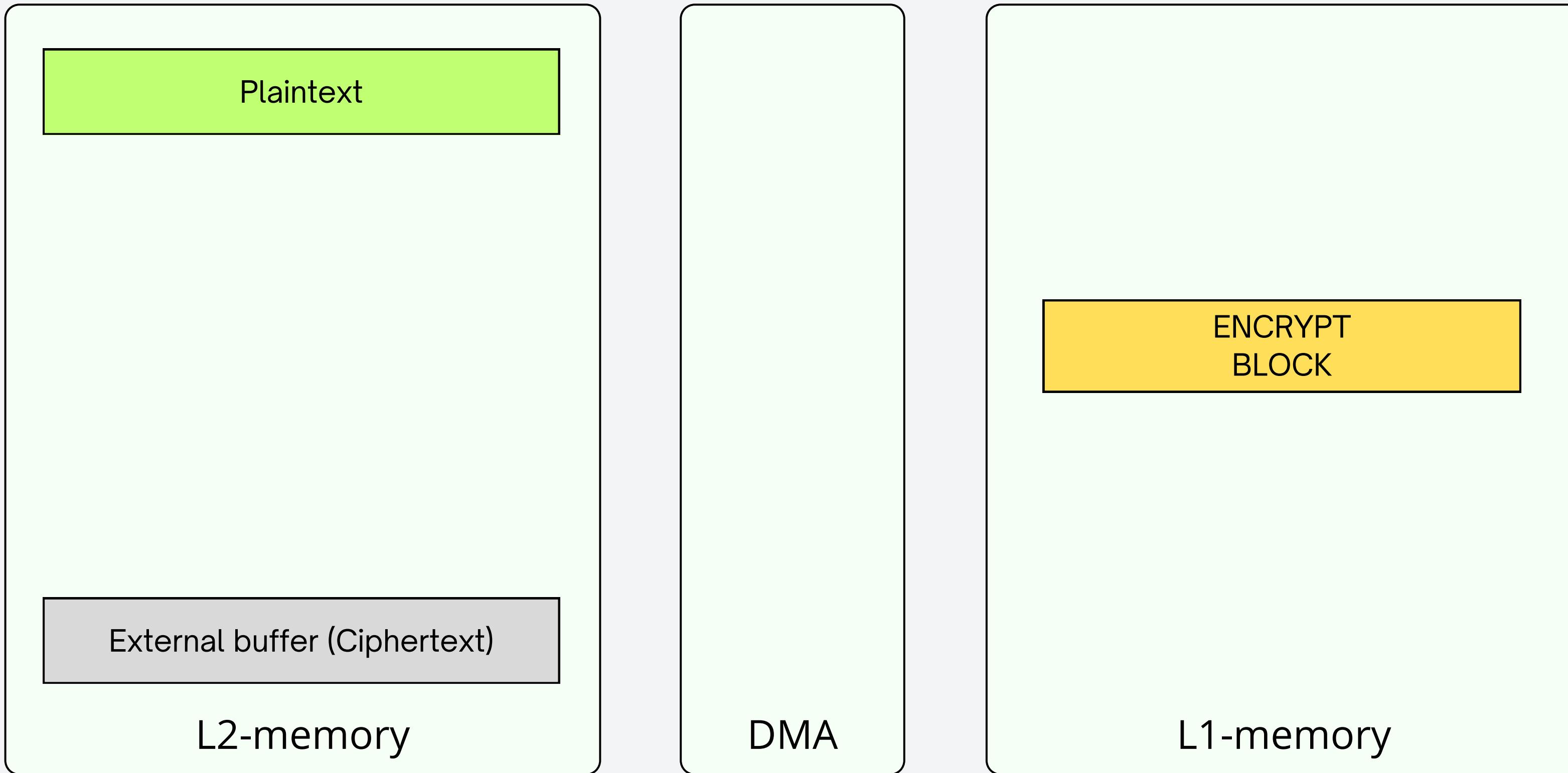
IMPLEMENTAZIONE SU GVSOC

GAP8 MICRO-ARCHITECTURE

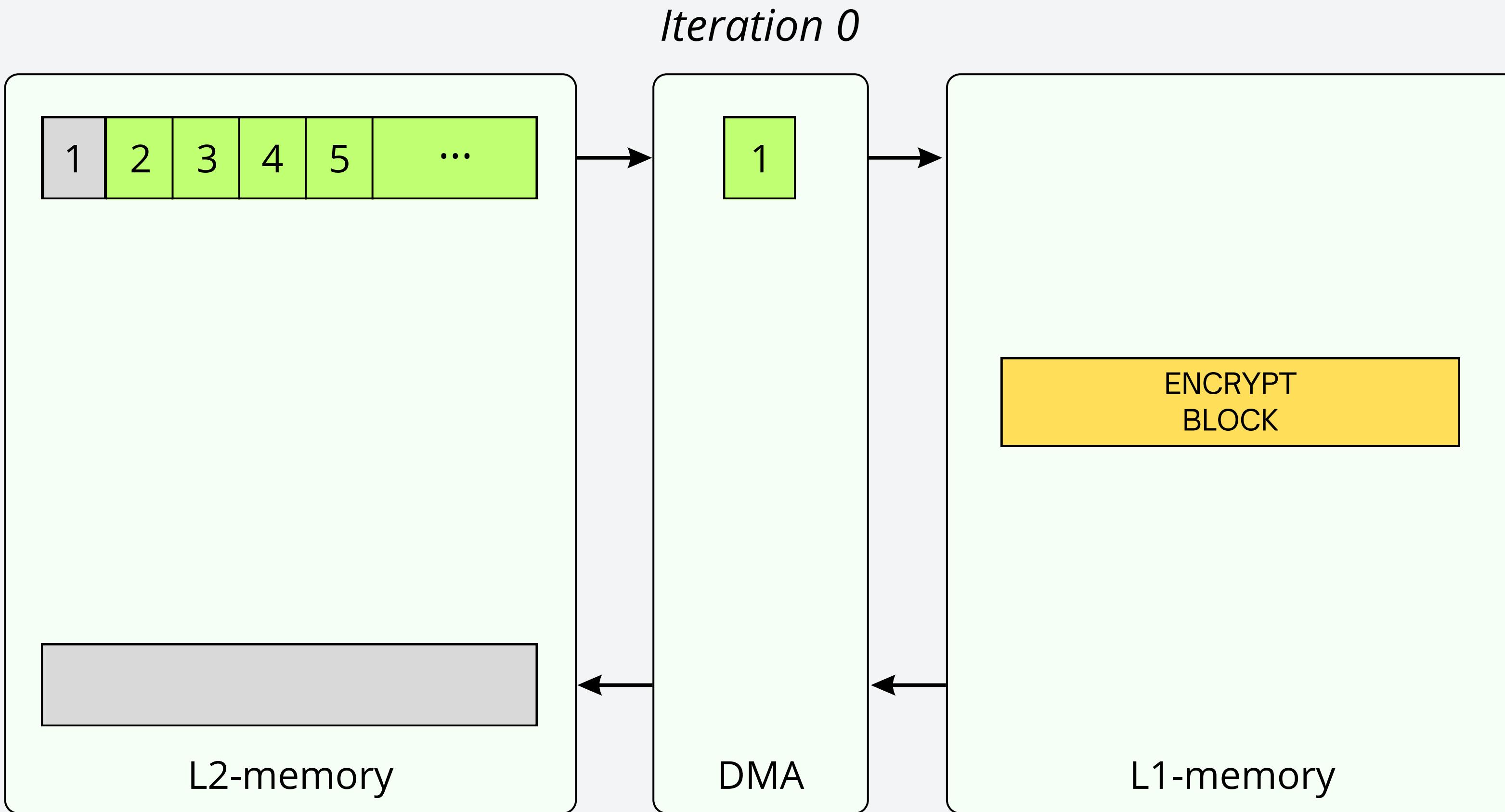


- Fabric controller core (for control, communications and security functions);
- 8 cores cluster;
- Global L2 memory;
- L1 memory shared by the 8 cores;
- Multi-channel cluster-DMA (for the transactions between the L2 Memory and L1 Memory).

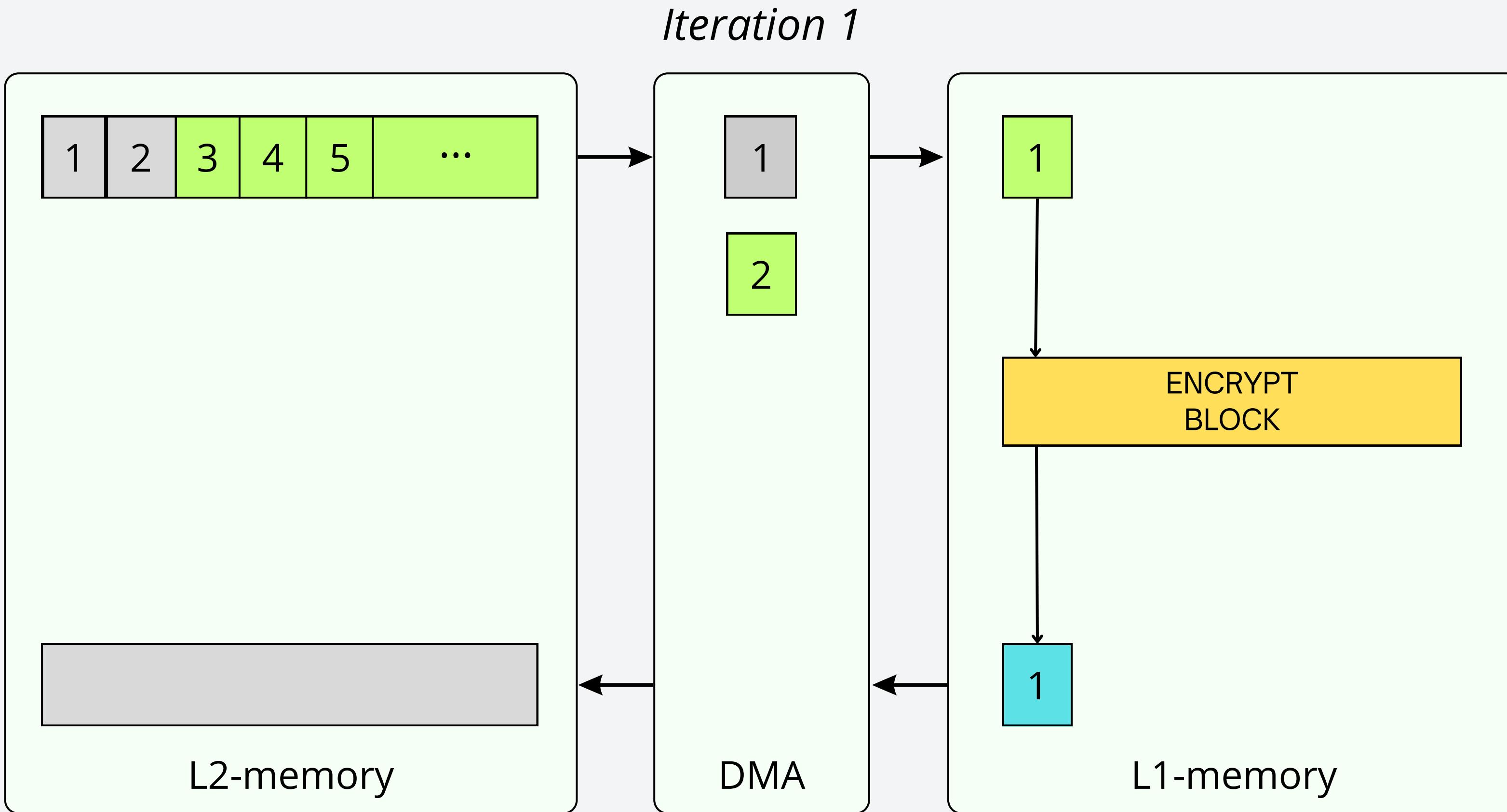
DMA TRANSACTION CONTROL



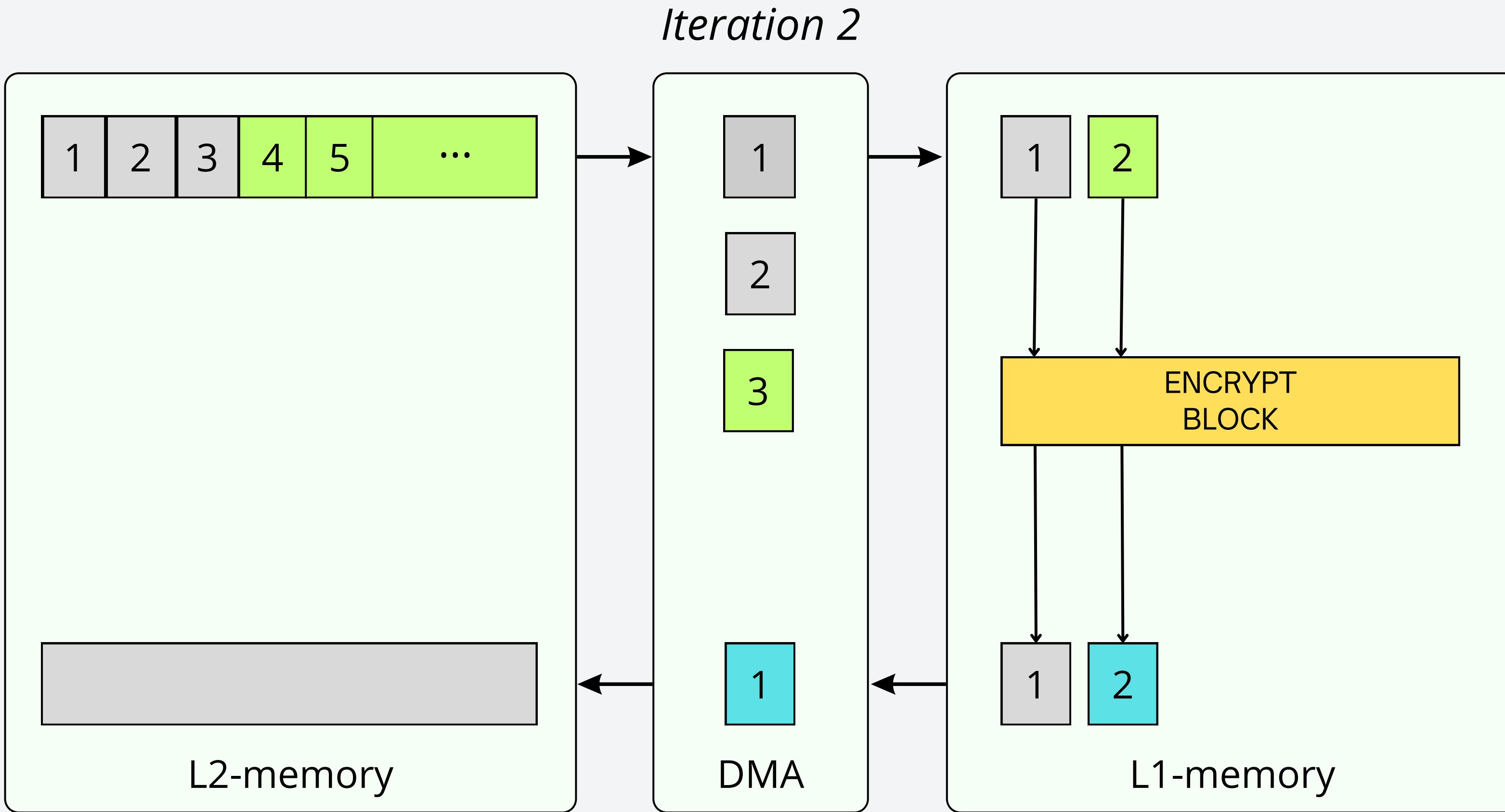
DMA TRANSACTION CONTROL



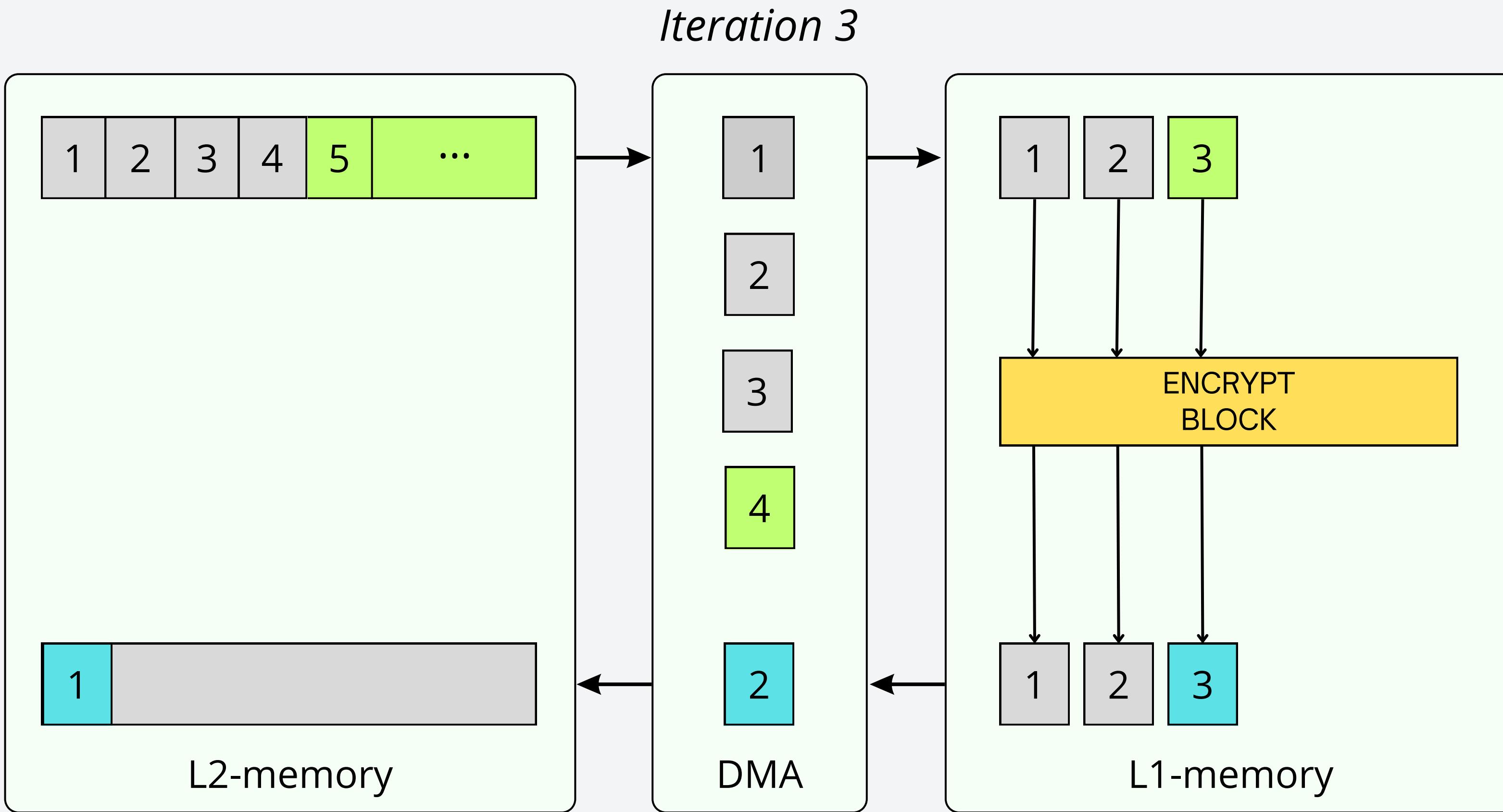
DMA TRANSACTION CONTROL



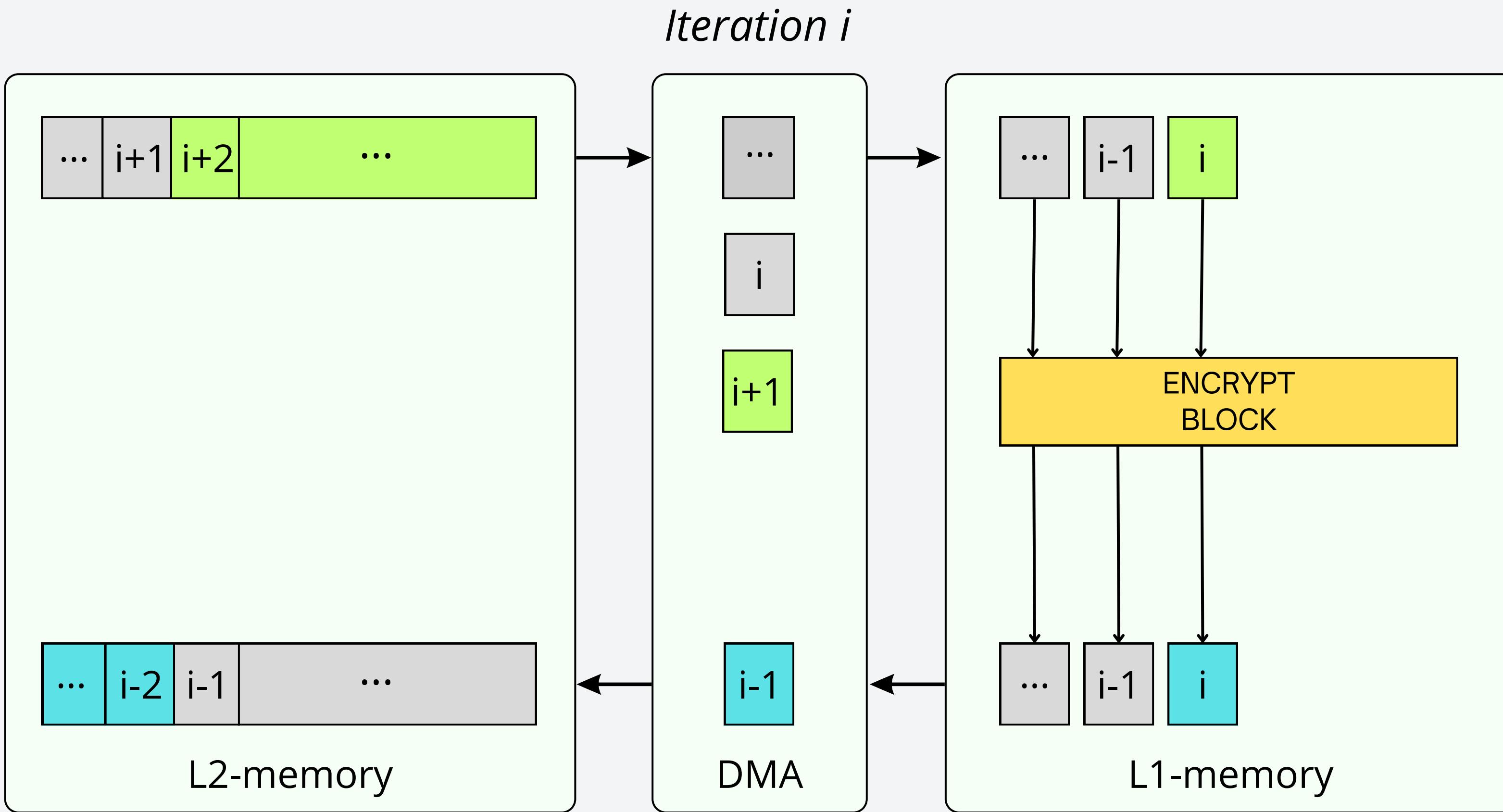
DMA TRANSACTION CONTROL



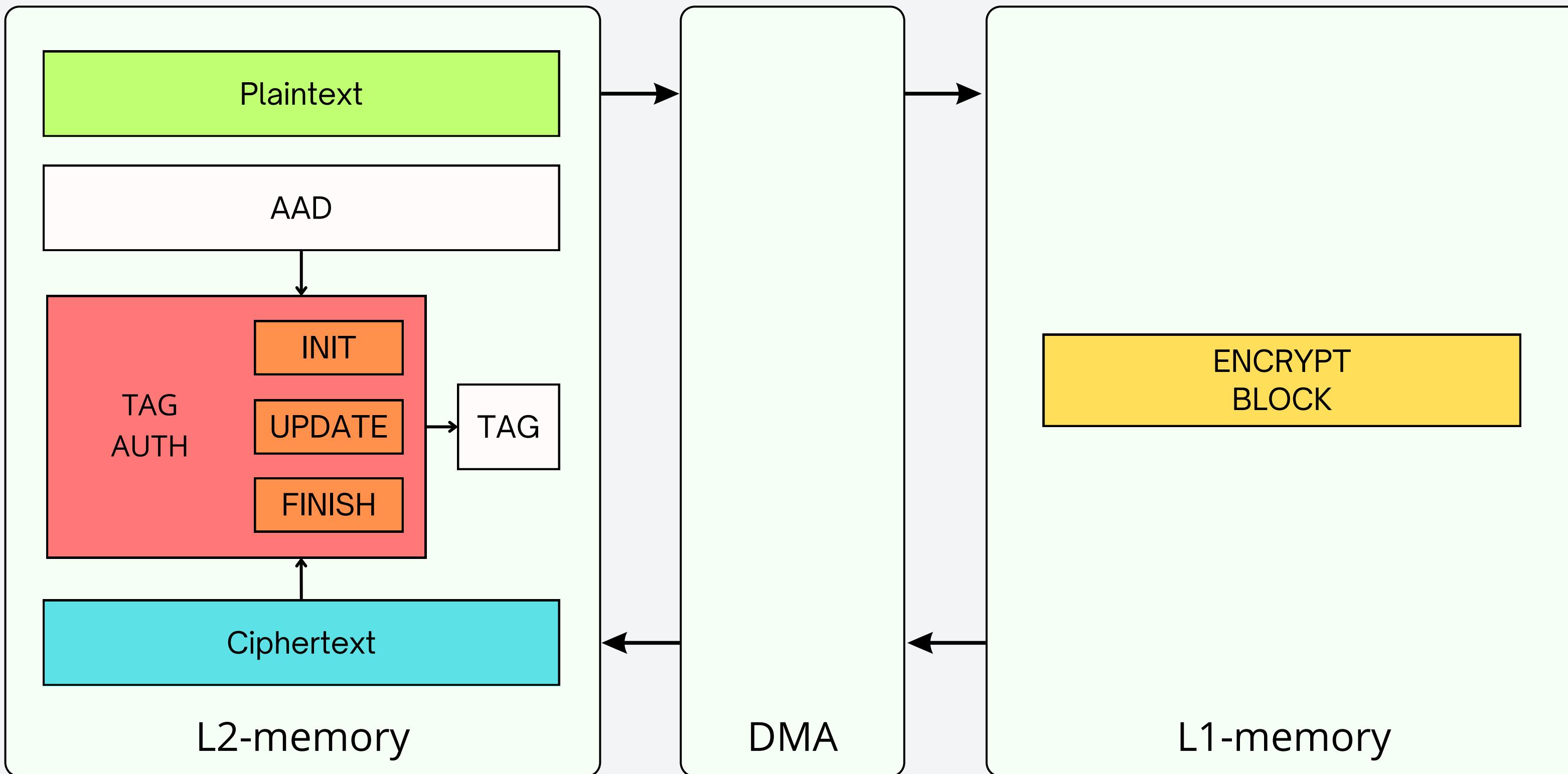
DMA TRANSACTION CONTROL



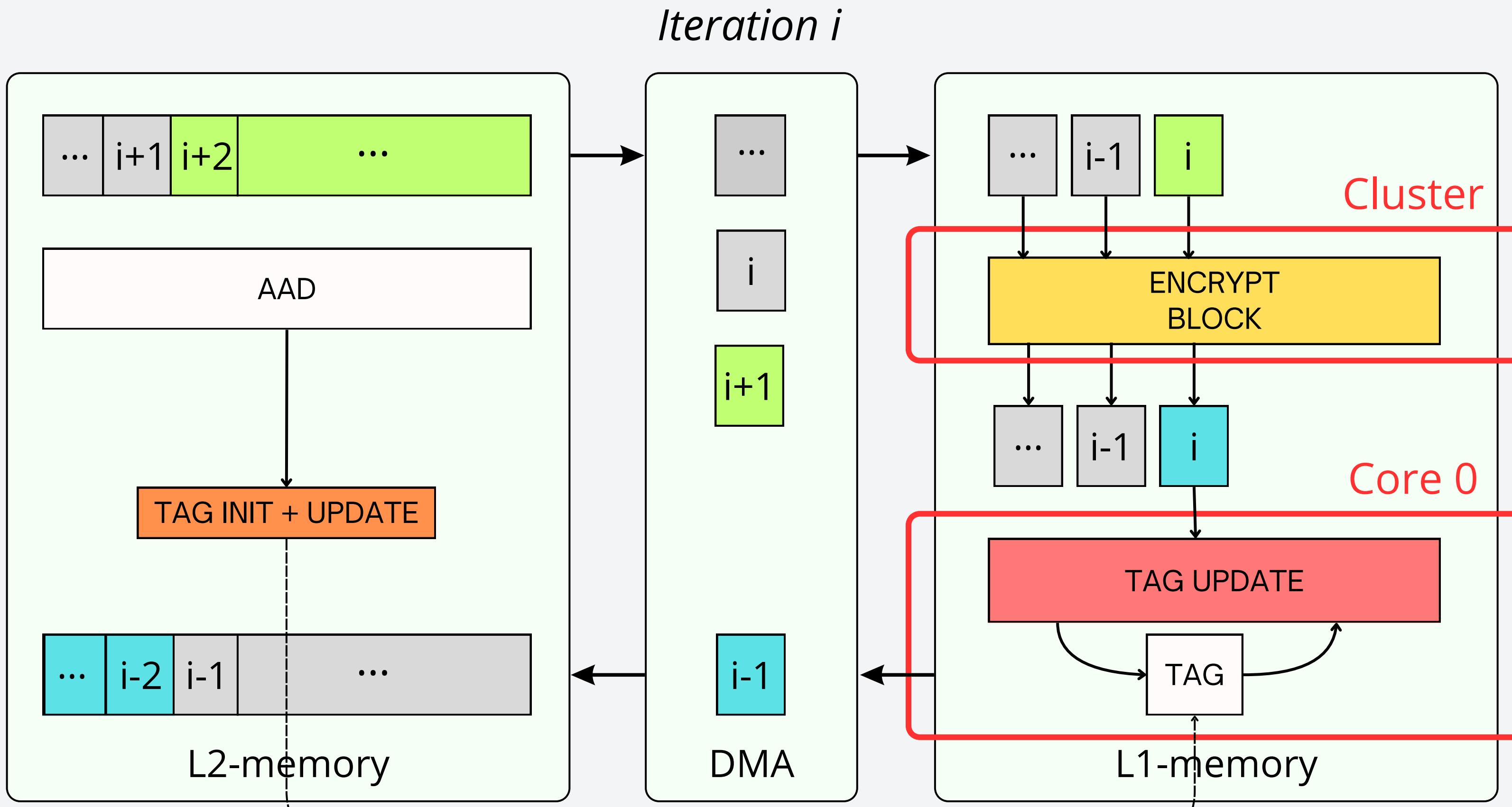
DMA TRANSACTION CONTROL



AUTHENTICATION



ENCRYPTION + AUTHENTICATION



ENCRYPTION + AUTHENTICATION

- Initialization tag computation with aad
- Update every cycle with the encrypted ciphertext's block

Pros:

- Update of the digest done with variables all in L1 memory (faster access)

Cons:

- Update intrinsically sequential for both the algorithms, only 1 of the 8 cores are computing while the others are waiting

But:

- That would have also been the case computing the update of the ciphertext after moving it into L2 memory

ENCRYPTION + AUTHENTICATION: FUTURE STEPS

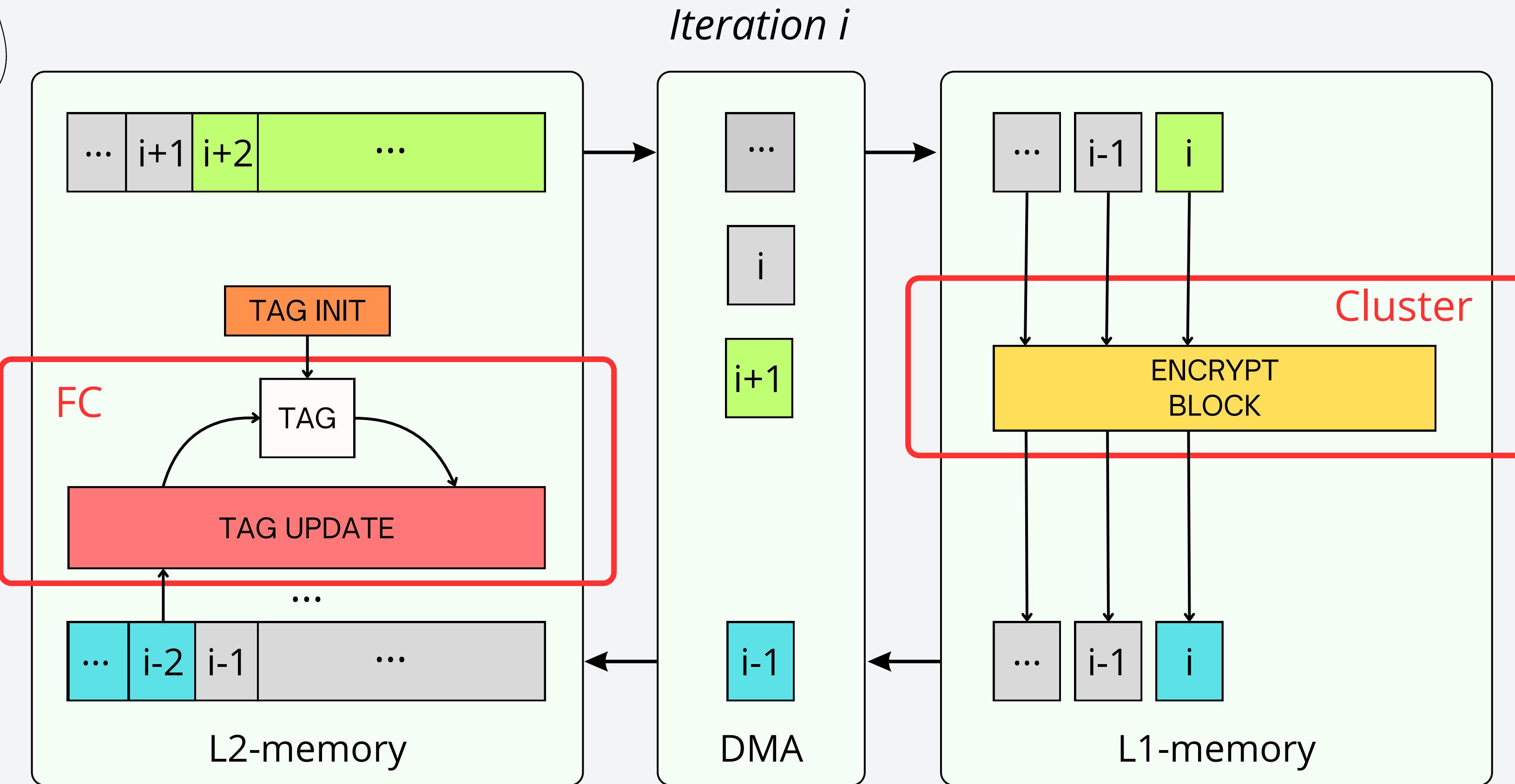
There might be the possibility to maximize the efficiency (and the performances overall) by synchronizing the FC and the cluster:

- CLUSTER: encryption block;
- FC: authentication

The synchronization would be implemented via team synchronization structures.

However, the solution has been attempted with sparse results, so it hasn't been explored.

DMA TRANSACTION CONTROL



PERFORMANCE EVALUATION

PERFORMANCE ASSESSED

- Clock Cycles
- Instruction Count
- CPI / IPC
- Cycles per Byte
- Speedup
- Efficiency

Every measure has been performed with different payload sizes, from 512 bytes up to 32 Kbytes, in order to better analyze the results obtained.

Finally, we'll point to the memory-transfer/execution tradeoff, evaluating the cycles "wasted" for the dma control.

Generation of random TV



Generation of trace files for each
(NUM_CORES, MEM_SIZE) configuration

```
#!/bin/bash

# Directory to store the output files
OUTPUT_DIR="output_files"
mkdir -p "$OUTPUT_DIR"

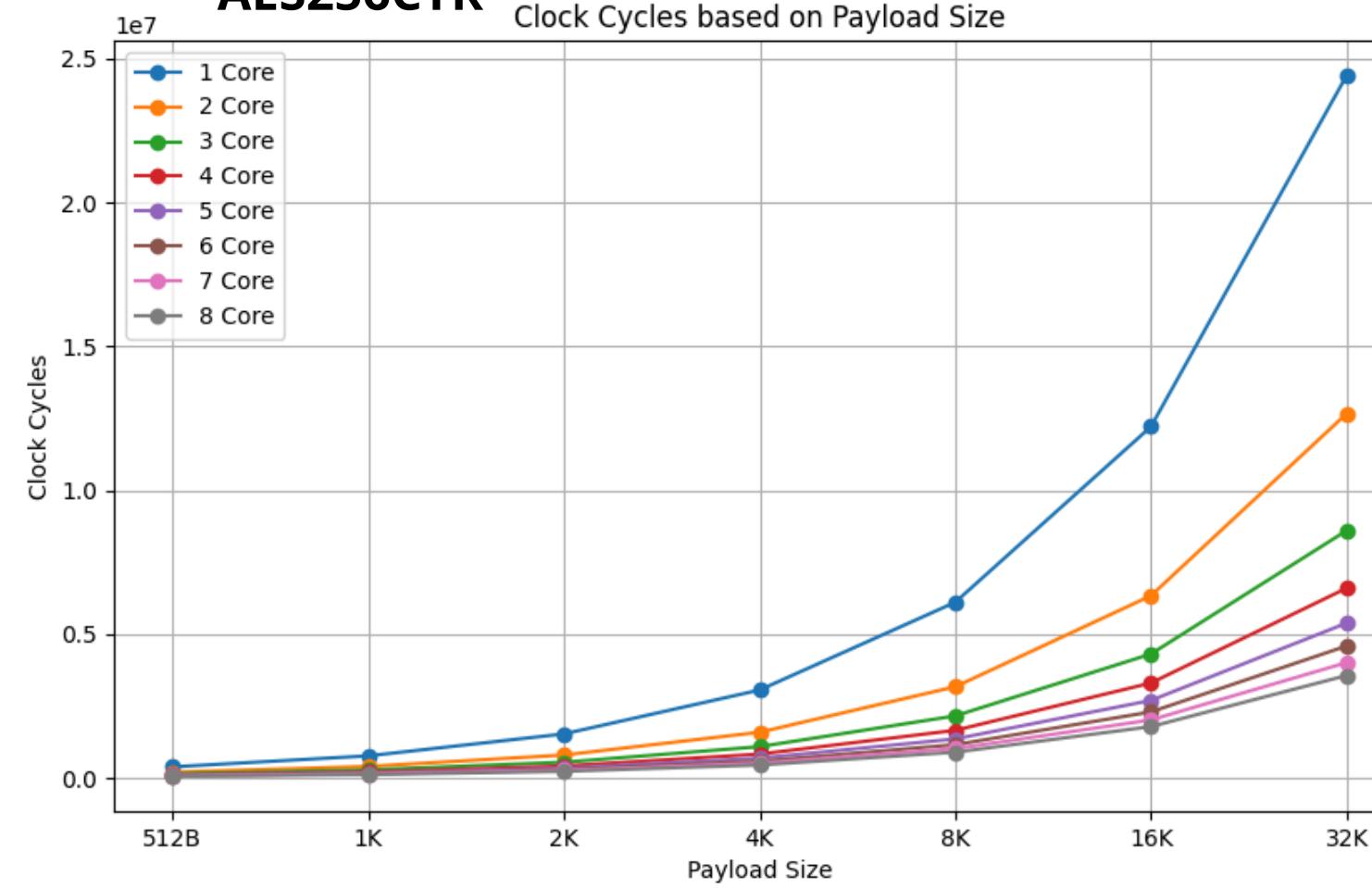
# Array of MEM_SIZE values
MEM_SIZES=(512 1 2 4 8 16 32)

# Loop over NUM_CORES from 1 to 8
for NUM_CORES in {1..8}; do
    # Loop over each MEM_SIZE value
    for MEM_SIZE in "${MEM_SIZES[@]}"; do
        # Run the make command and redirect the output to a file
        OUTPUT_FILE="${OUTPUT_DIR}/${MEM_SIZE}_${NUM_CORES}cores.txt"
        echo "Running with NUM_CORES=$NUM_CORES and MEM_SIZE=${MEM_SIZE}"
        make clean all run USE_CLUSTER=1 NUM_CORES=$NUM_CORES MEM_SIZE=$MEM_SIZE
    done
done
```

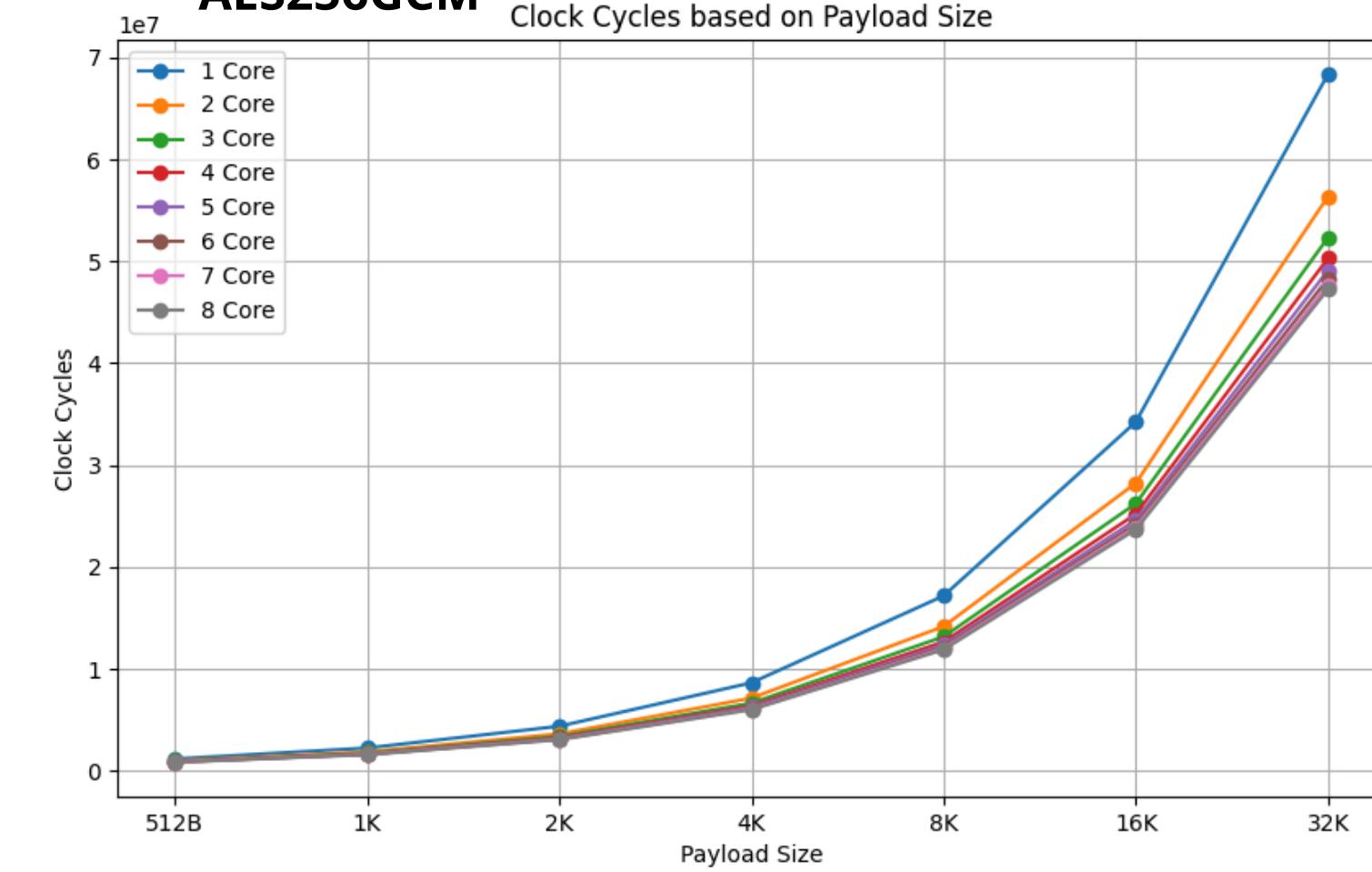
Performance evaluation



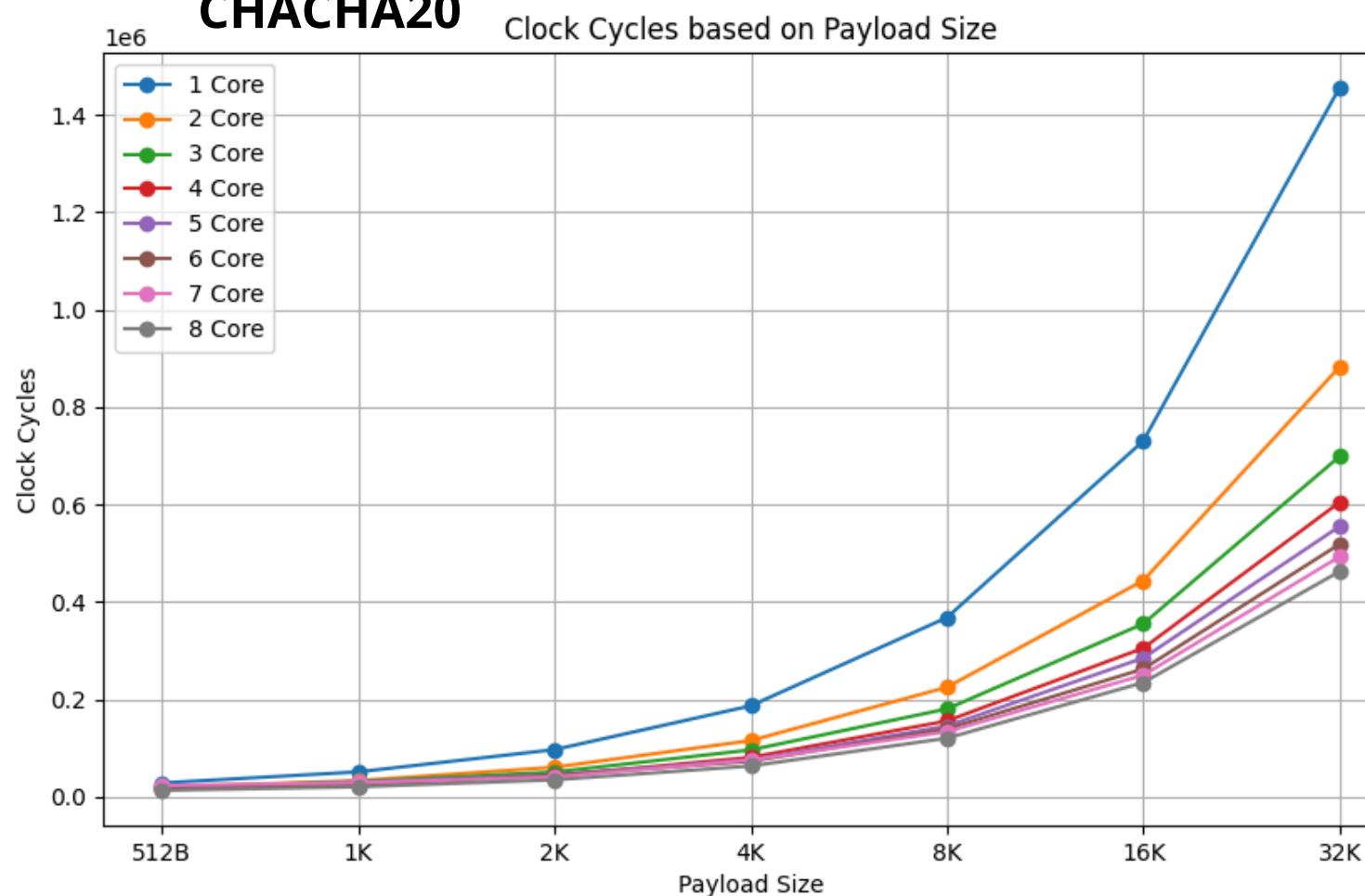
AES256CTR



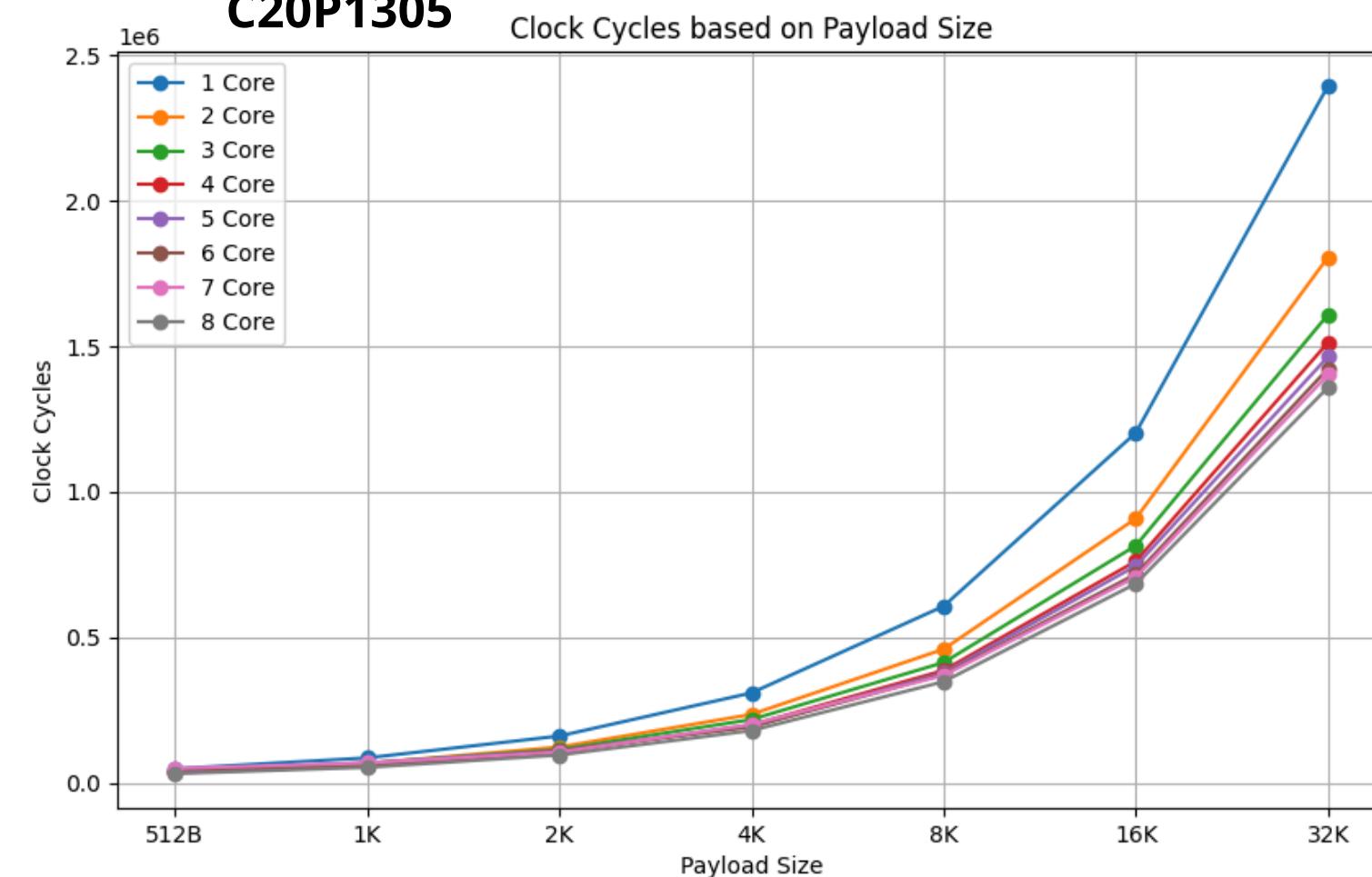
AES256GCM

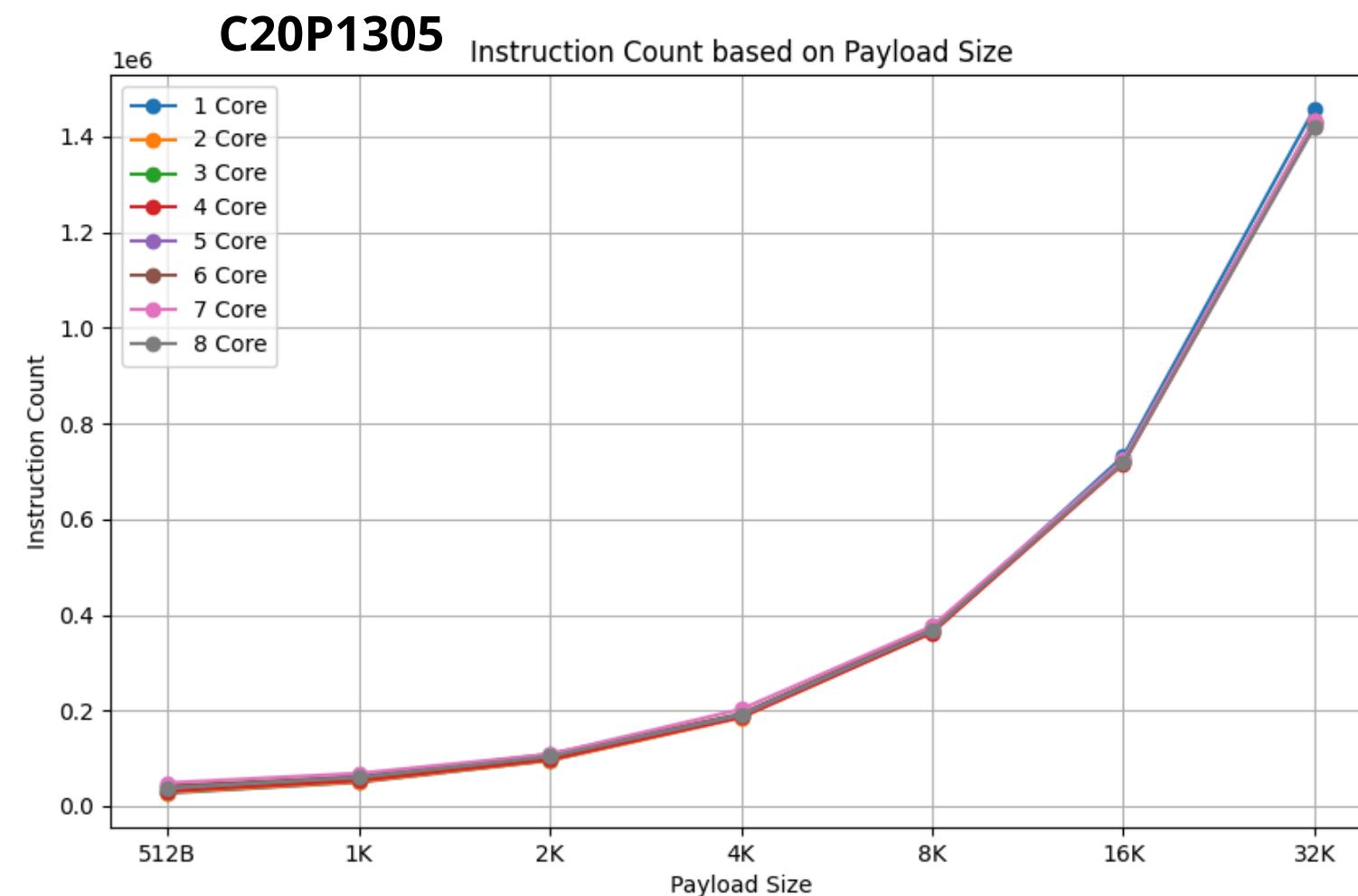
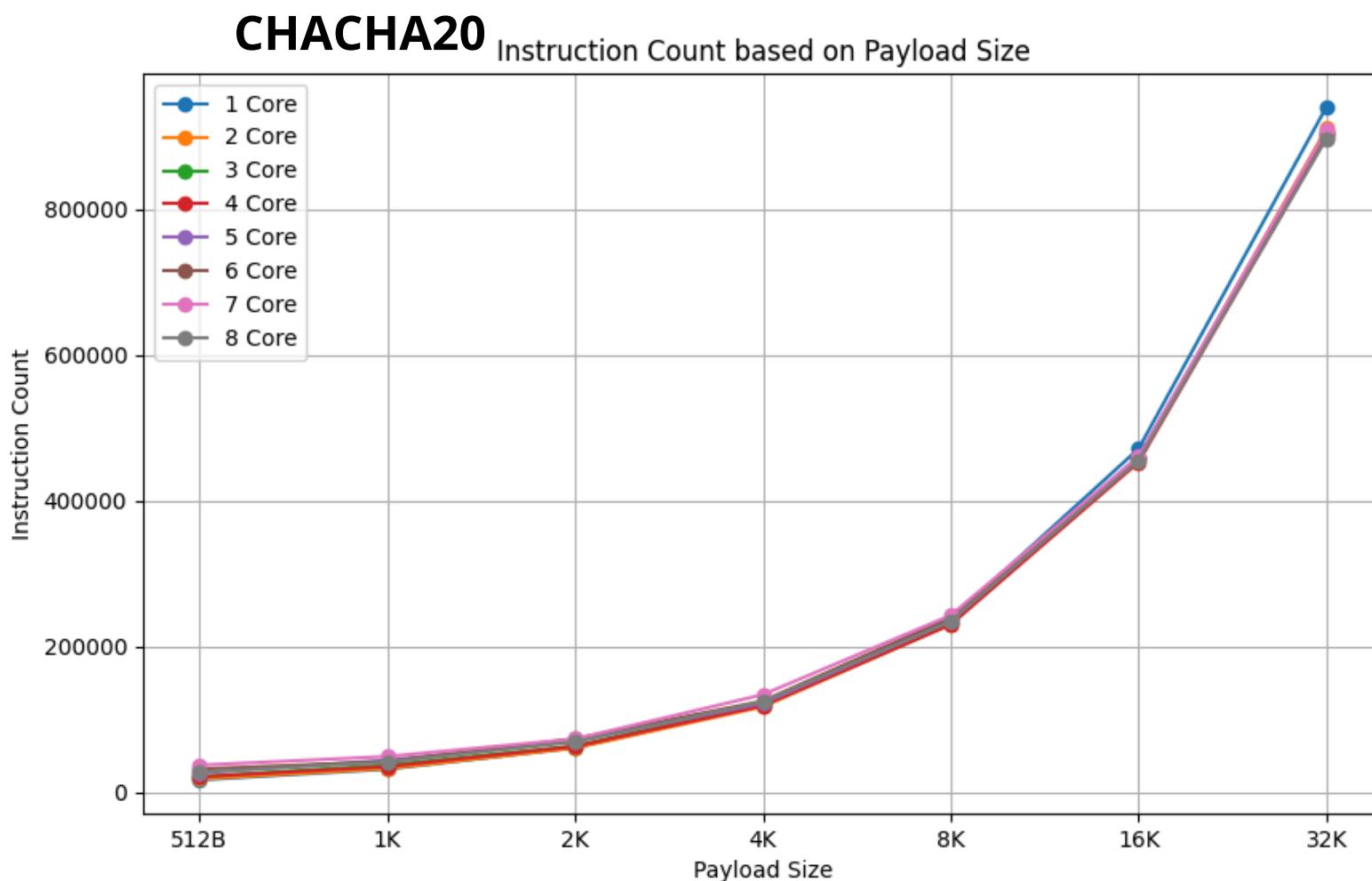
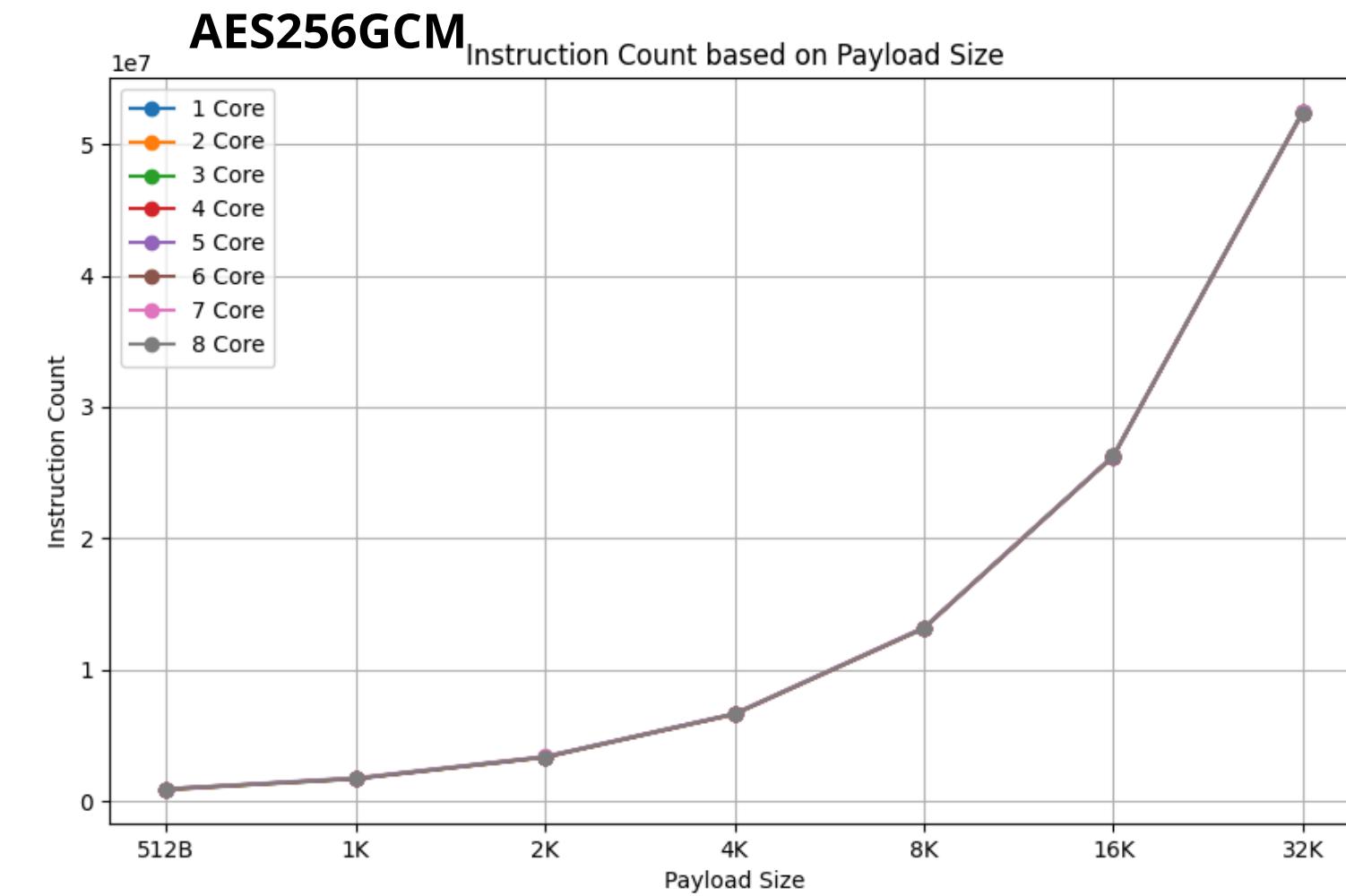
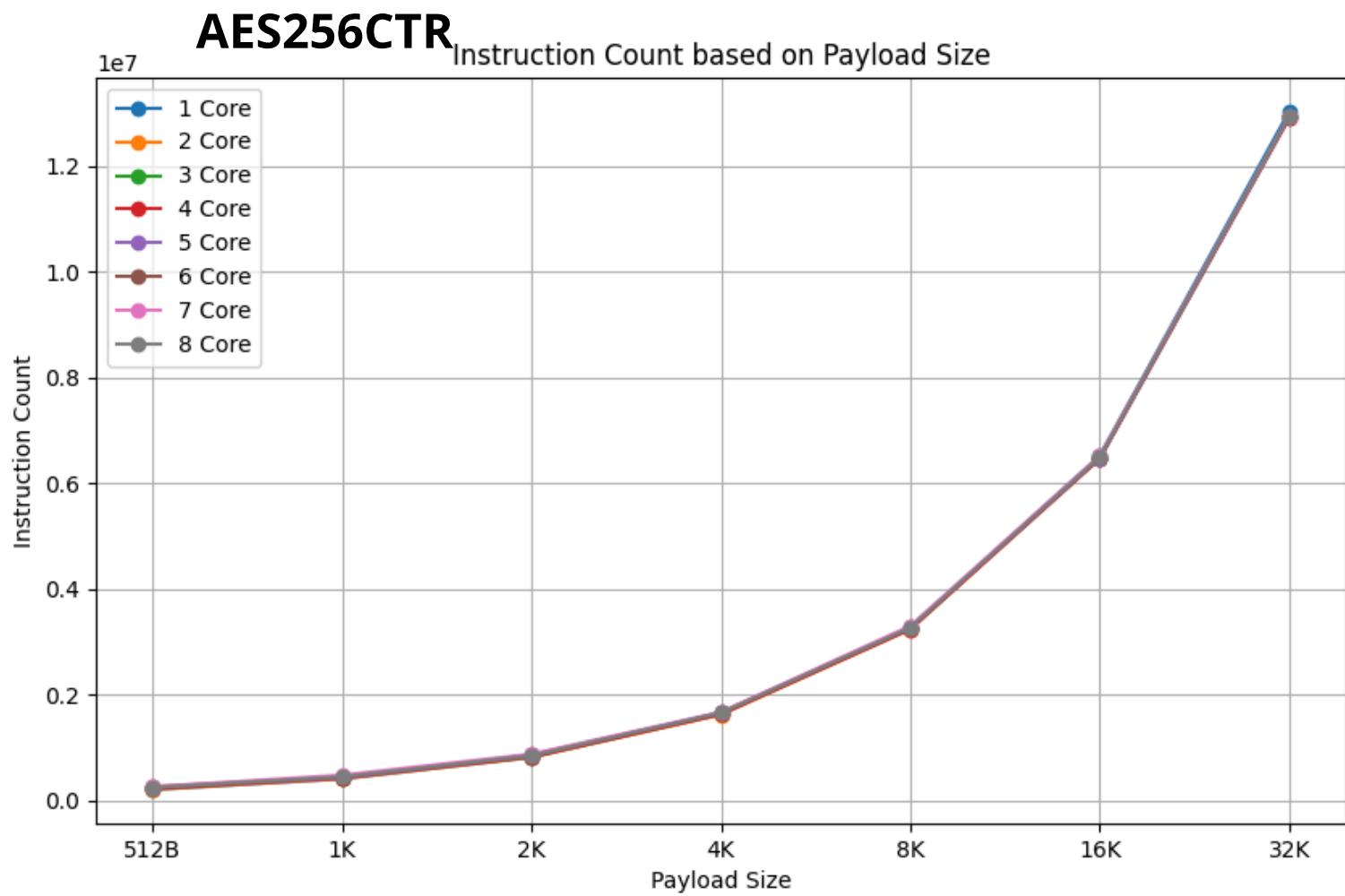


CHACHA20

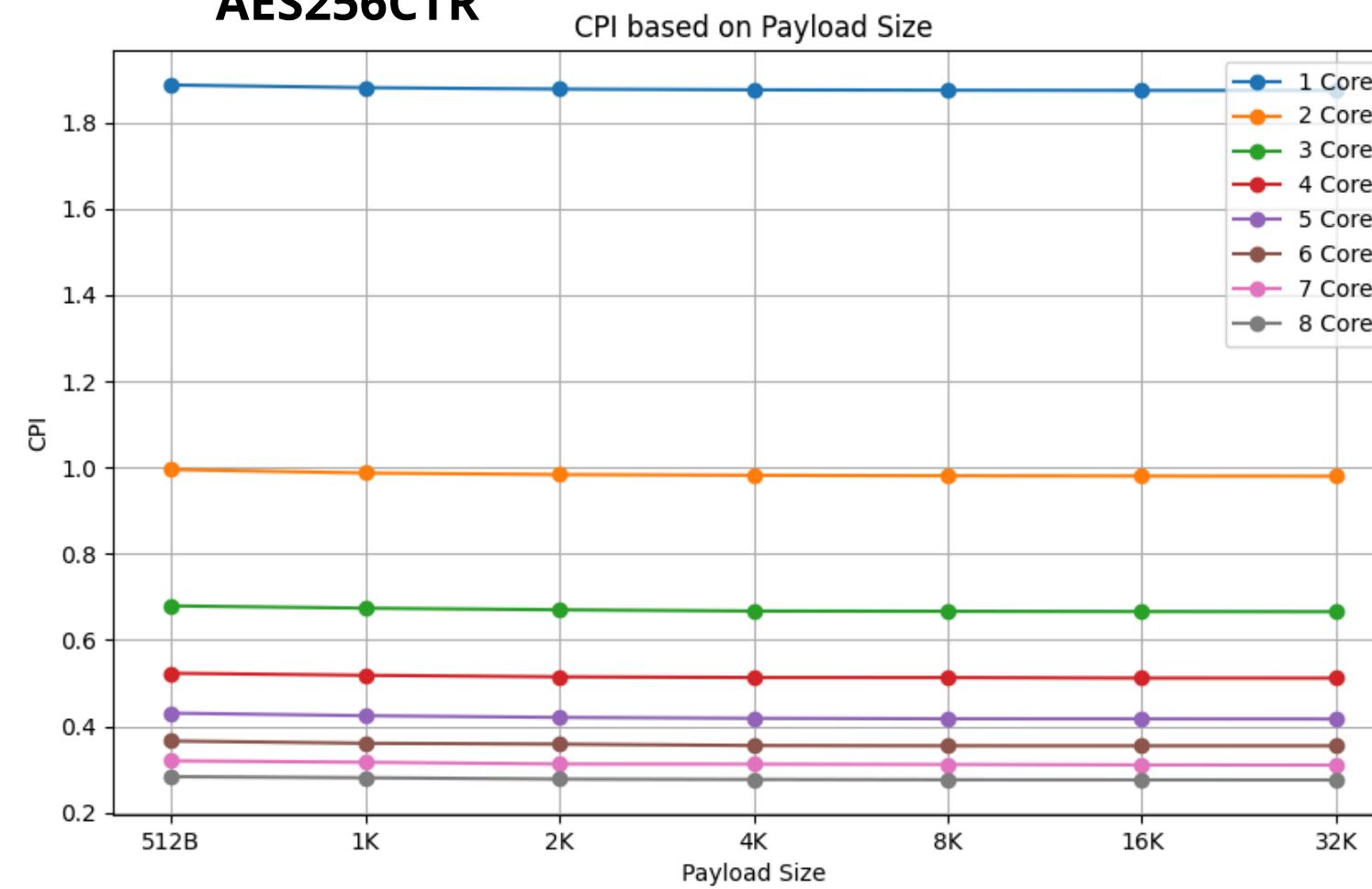


C20P1305

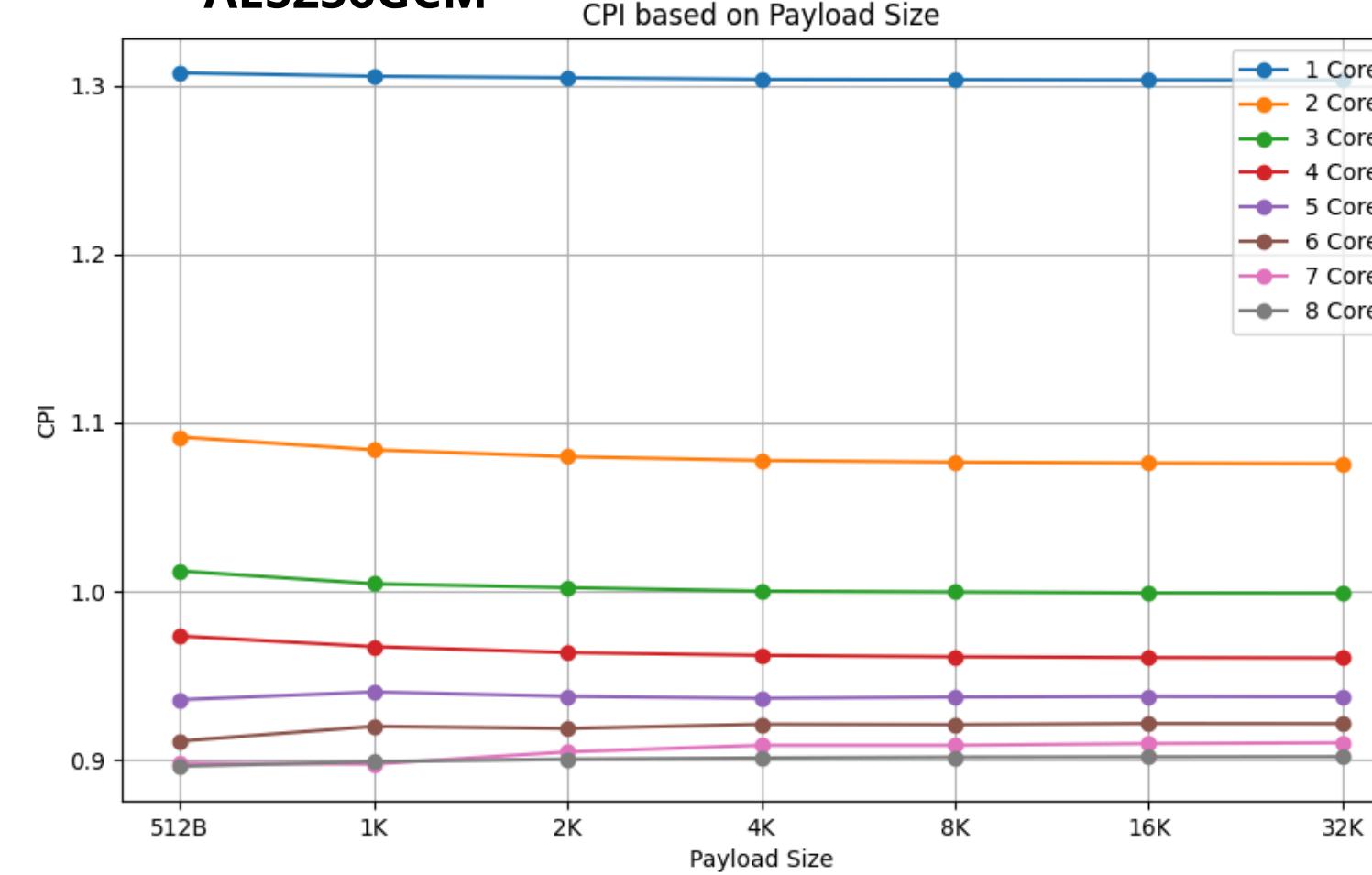




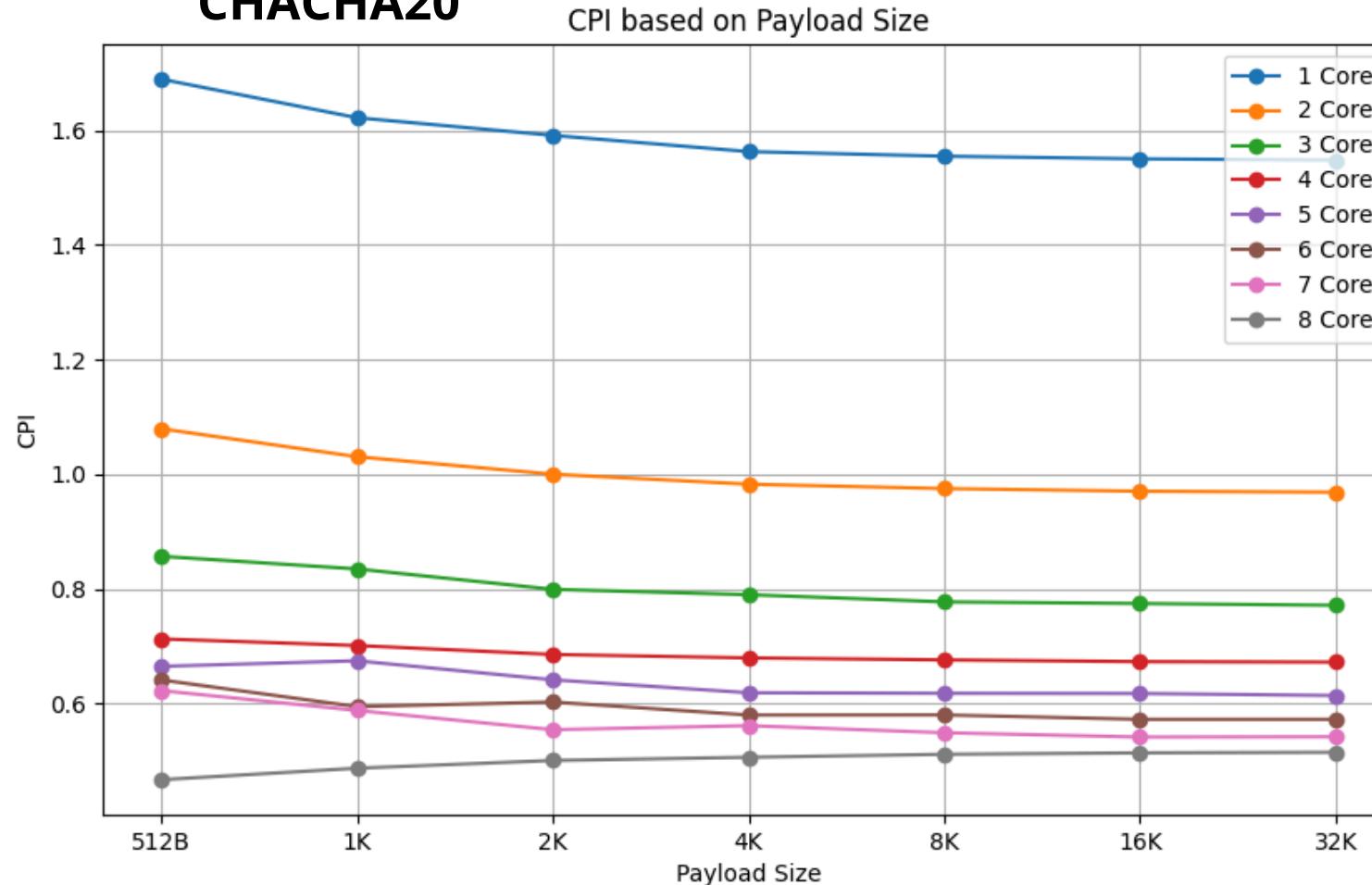
AES256CTR



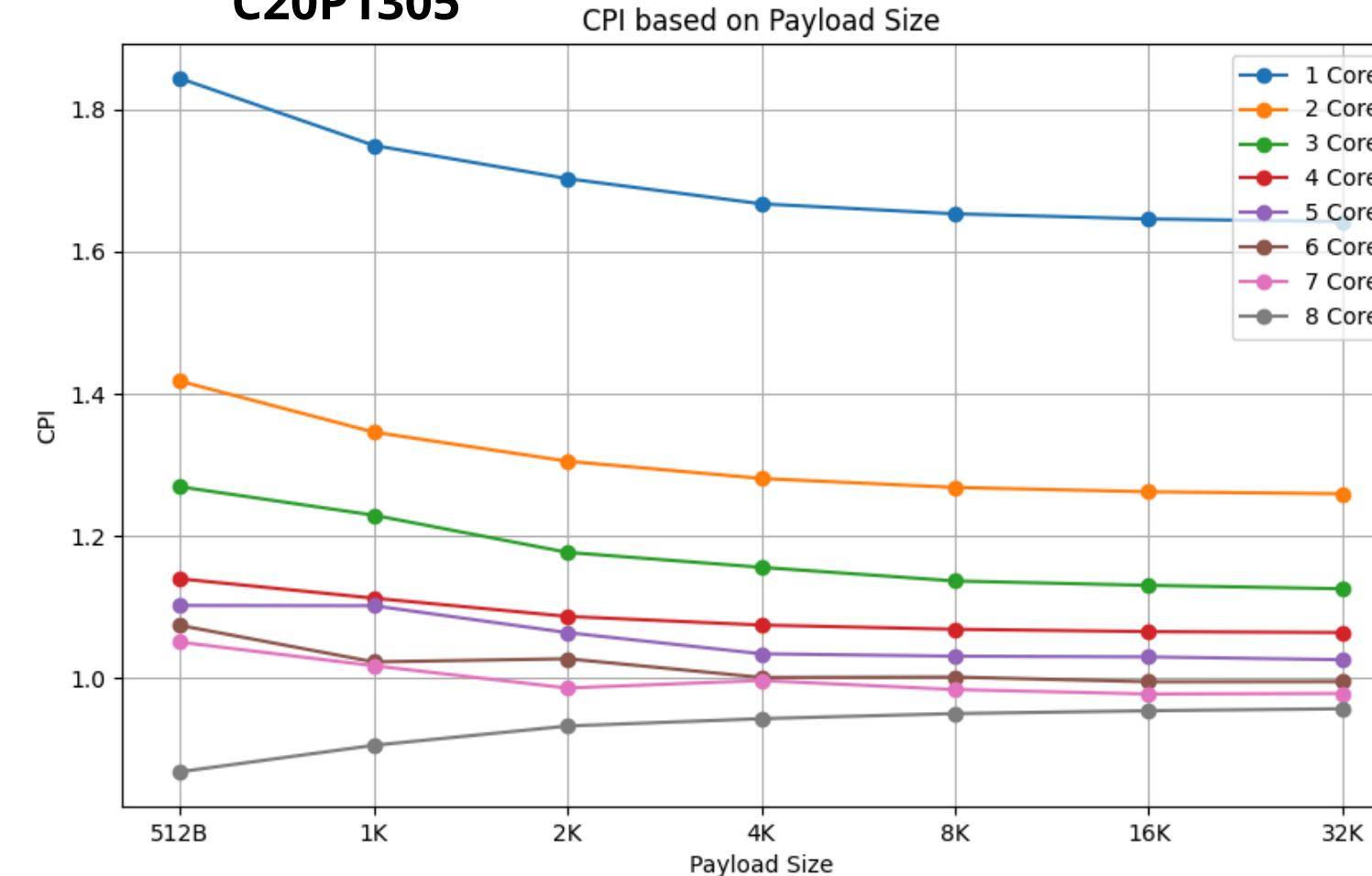
AES256GCM



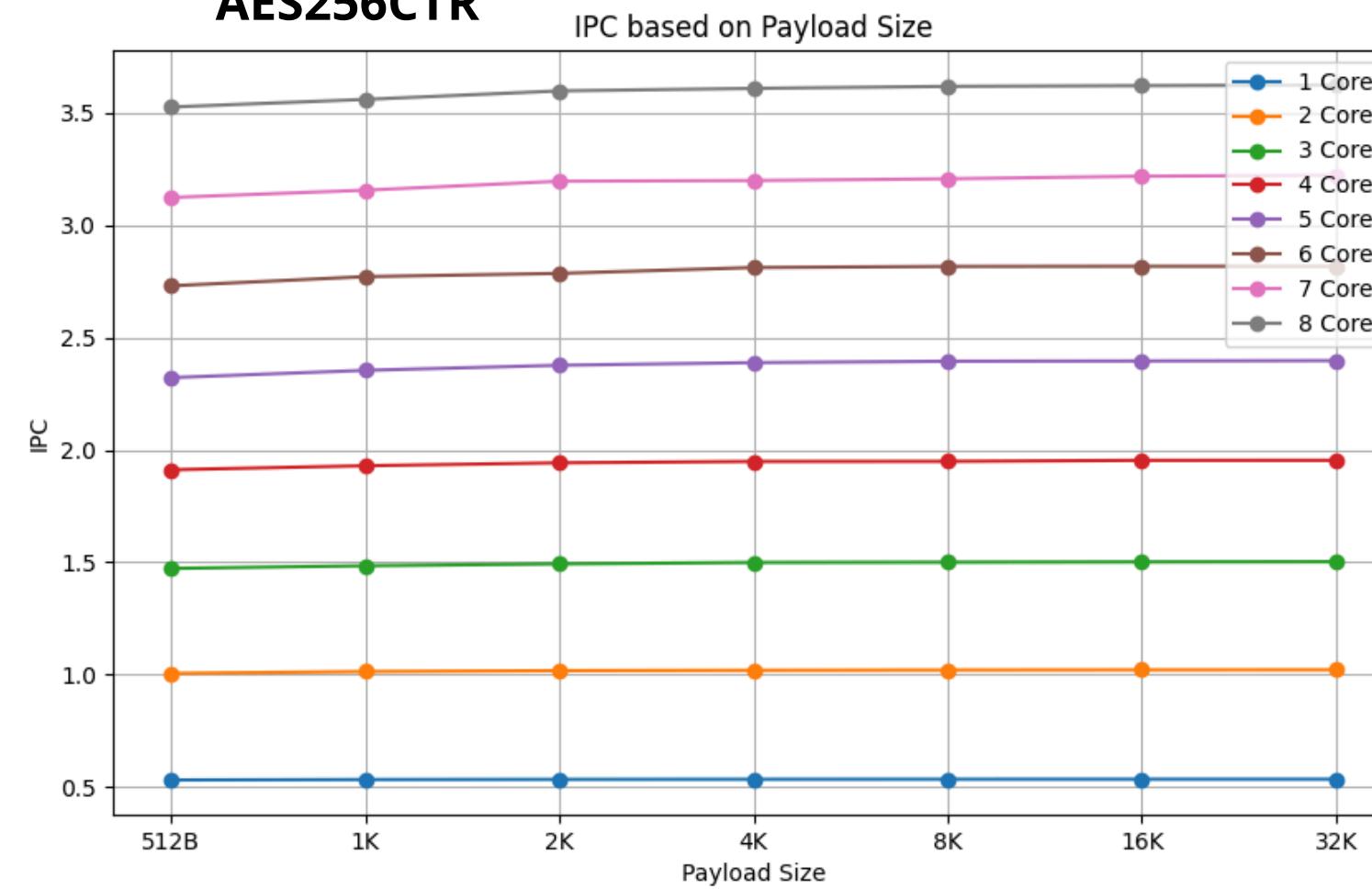
CHACHA20



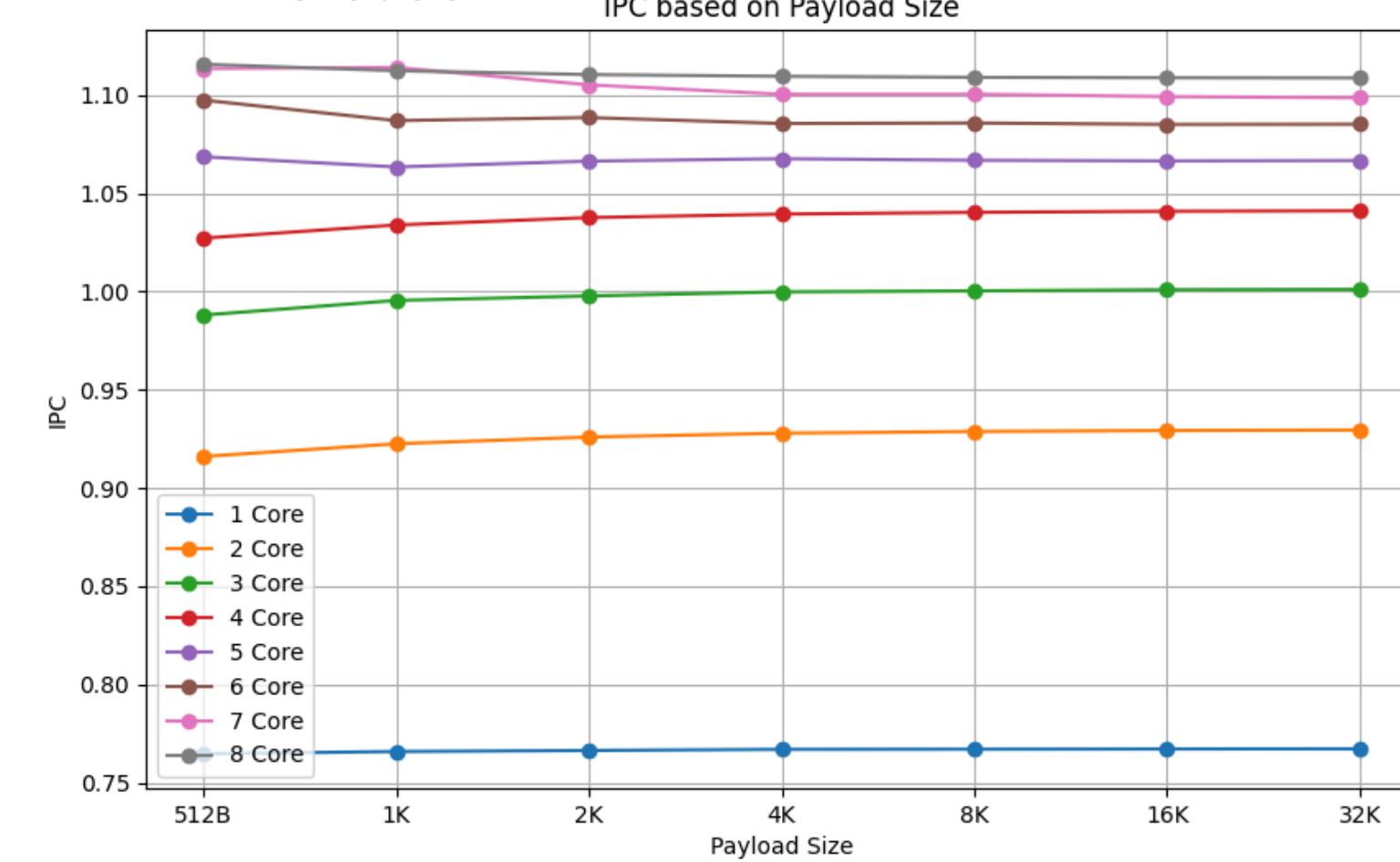
C20P1305



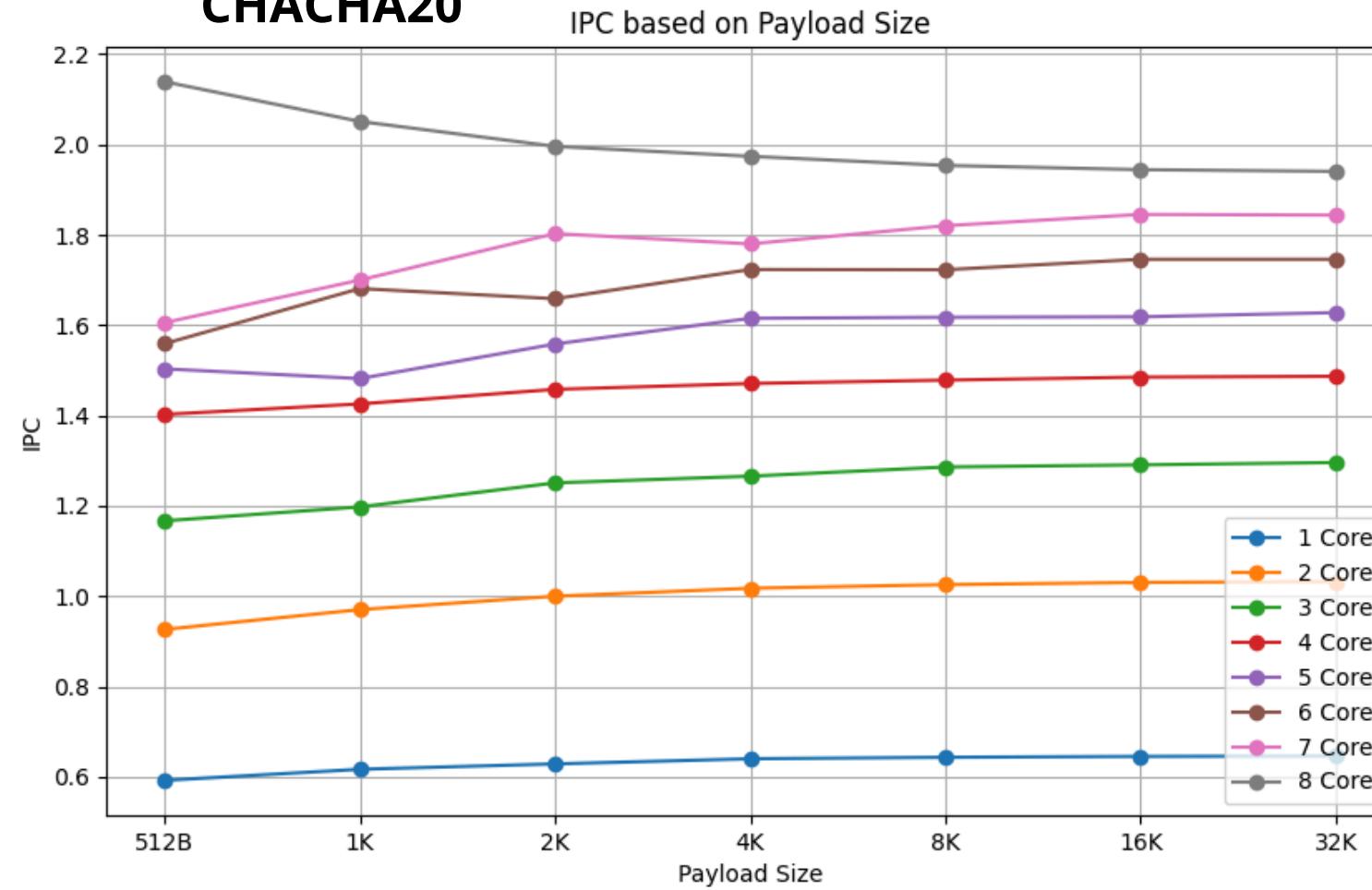
AES256CTR



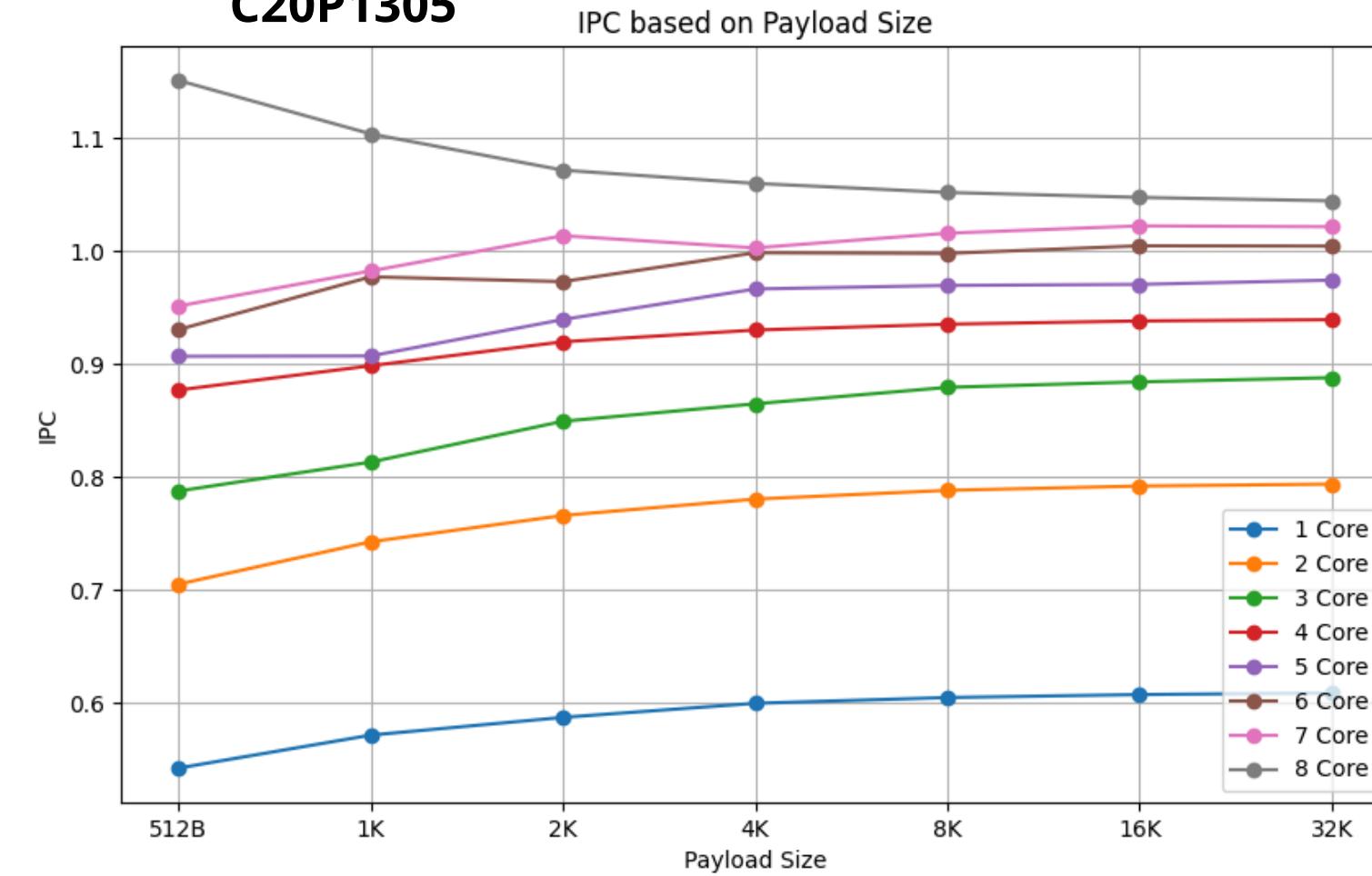
AES256GCM



CHACHA20

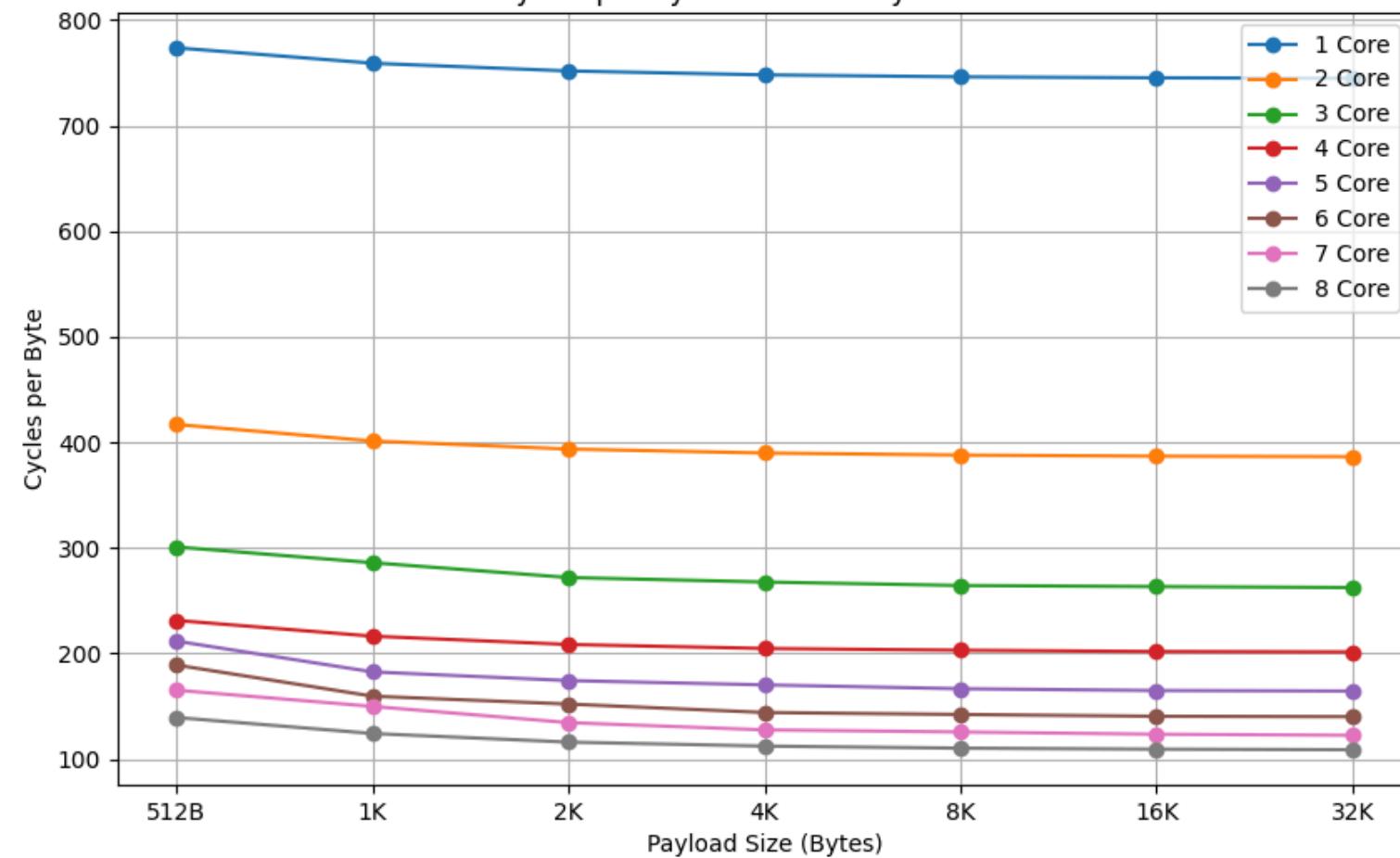


C20P1305



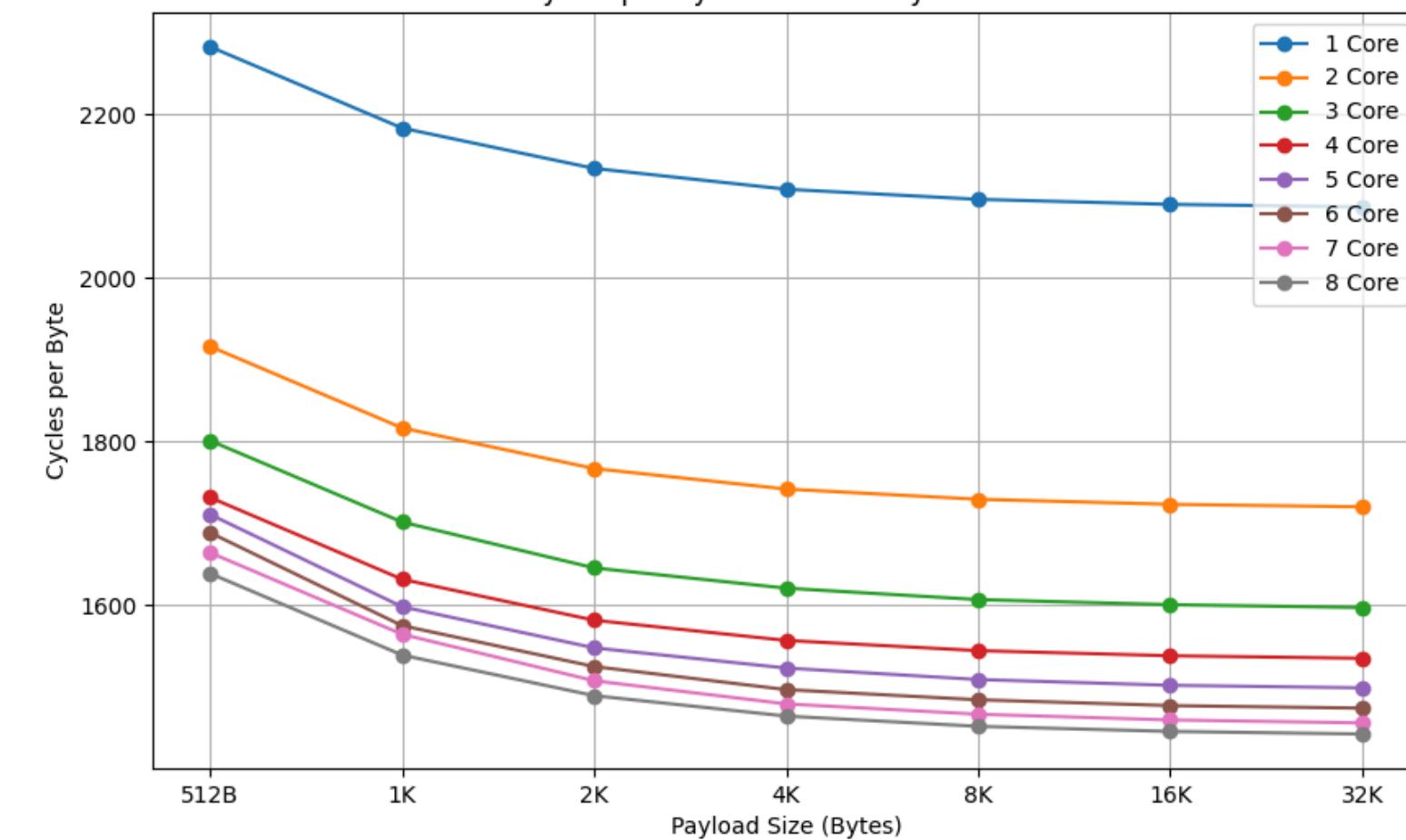
AES256CTR

Cycles per Byte based on Payload Size



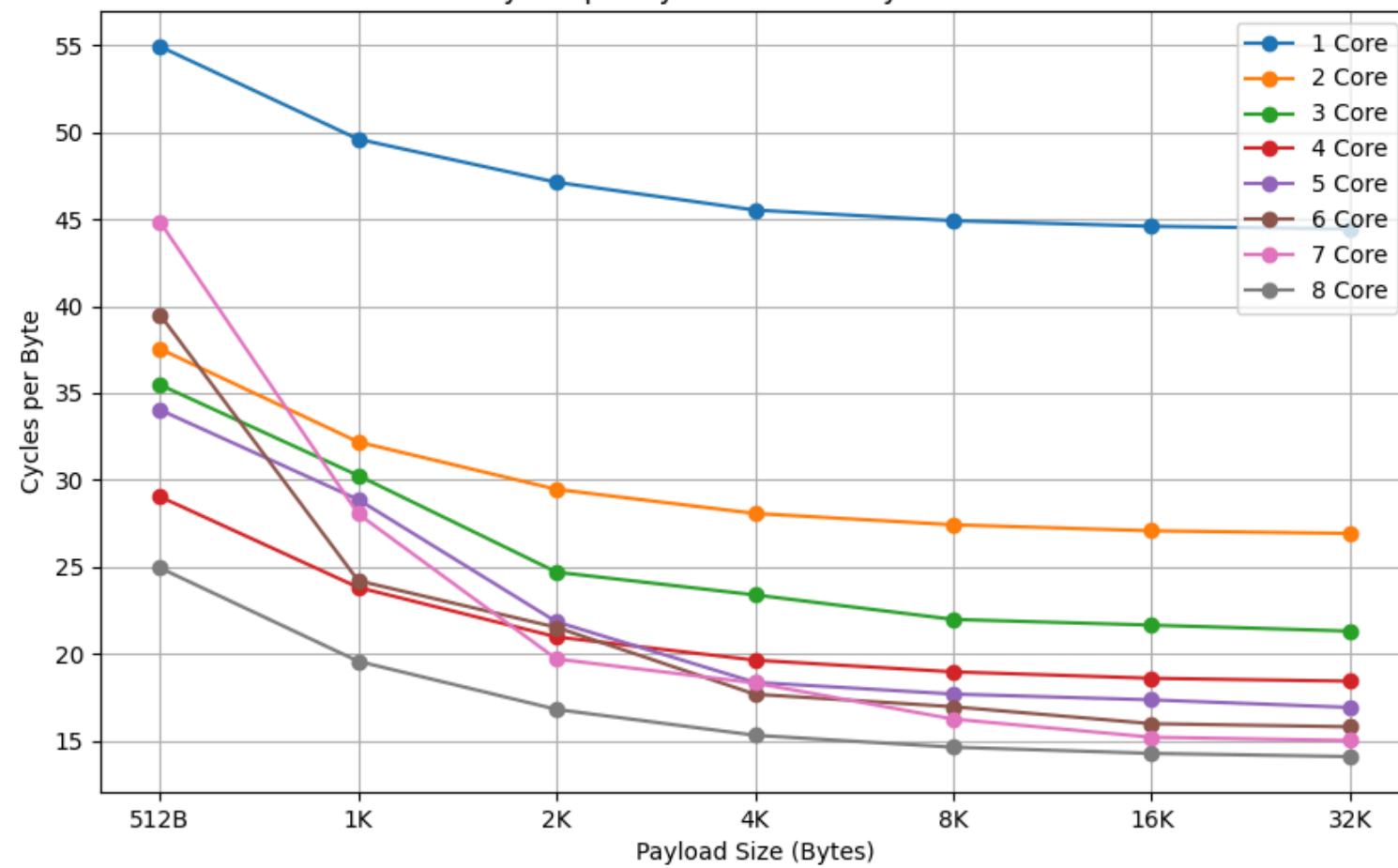
AES256GCM

Cycles per Byte based on Payload Size



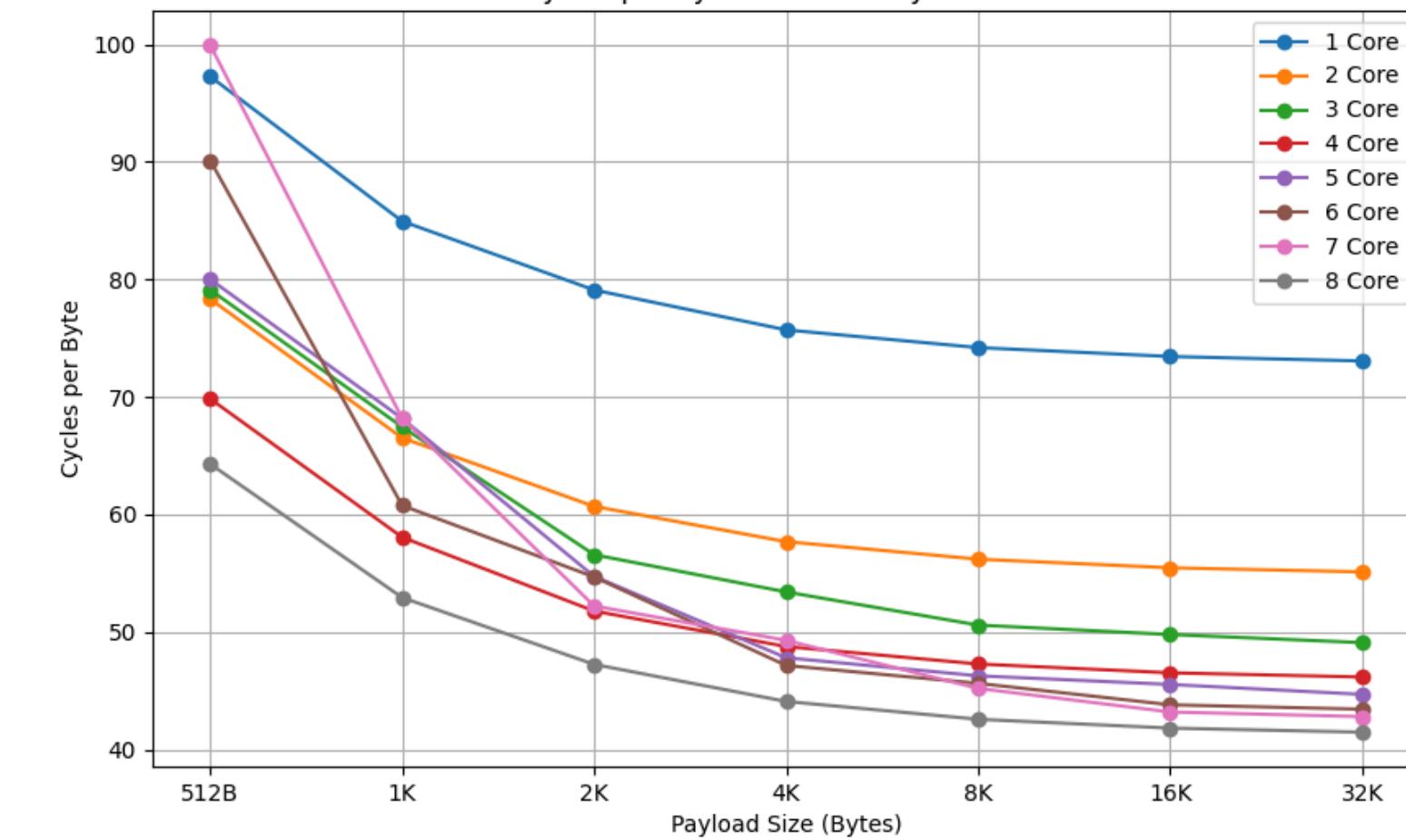
CHACHA20

Cycles per Byte based on Payload Size

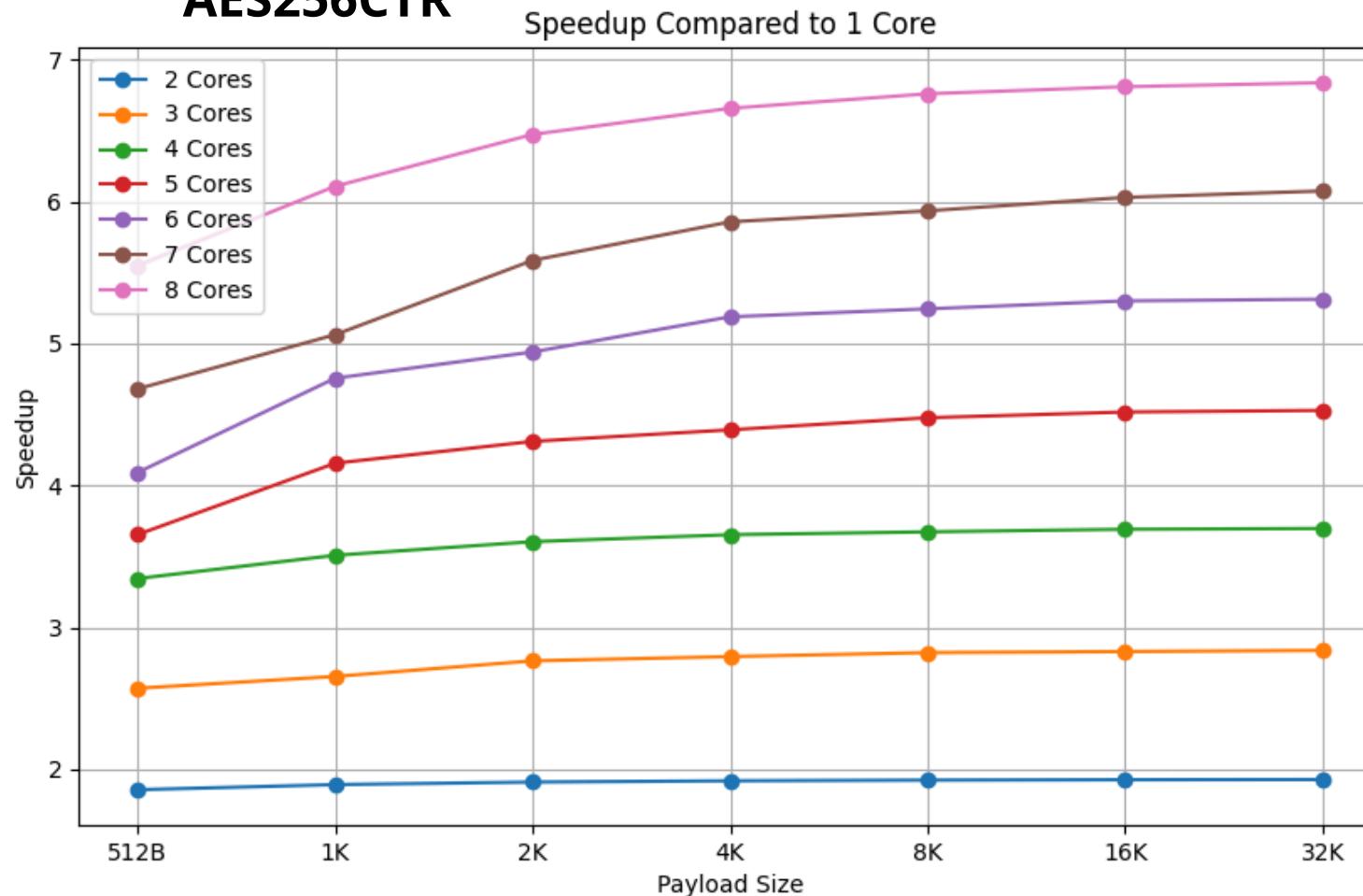


C20P1305

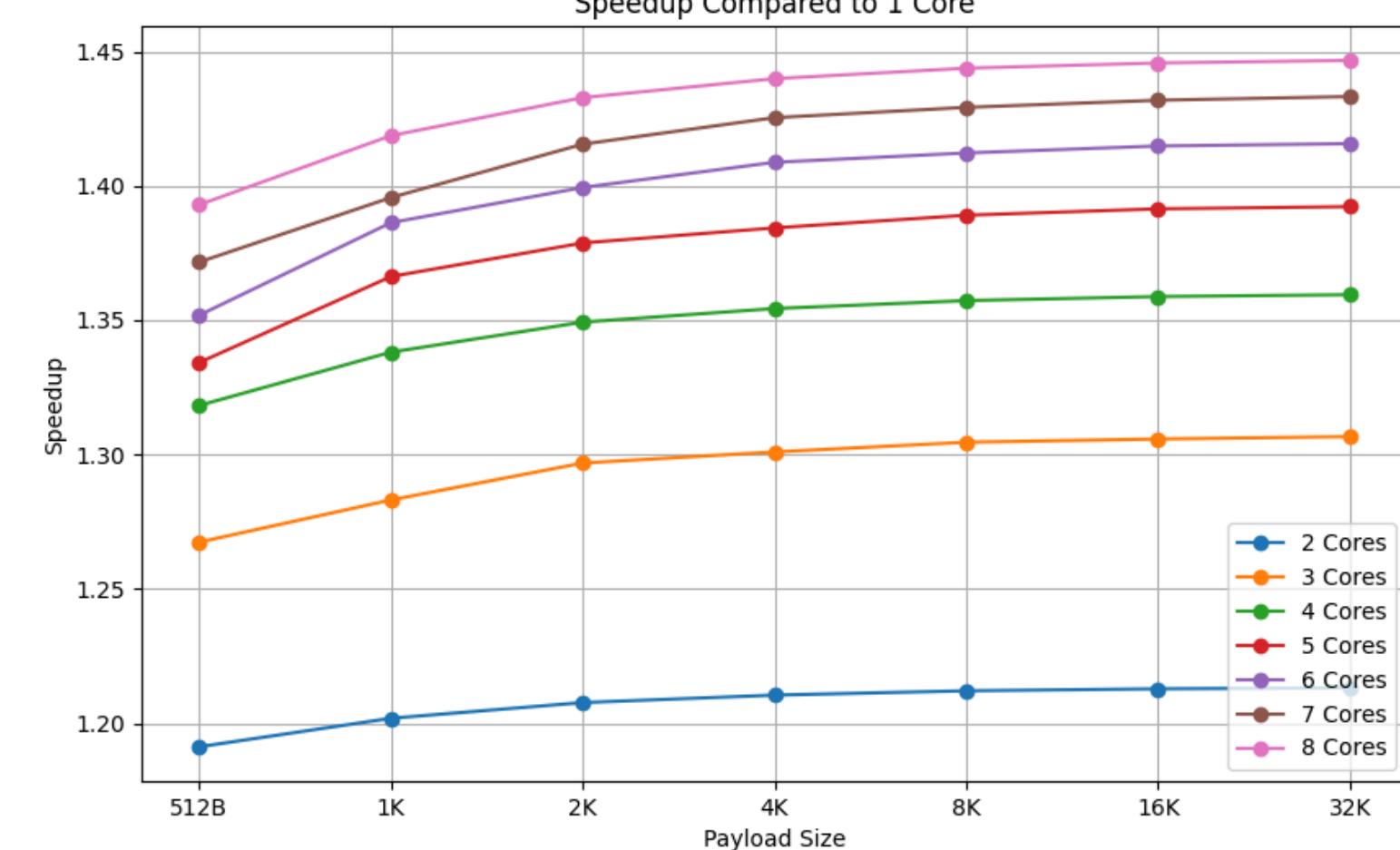
Cycles per Byte based on Payload Size



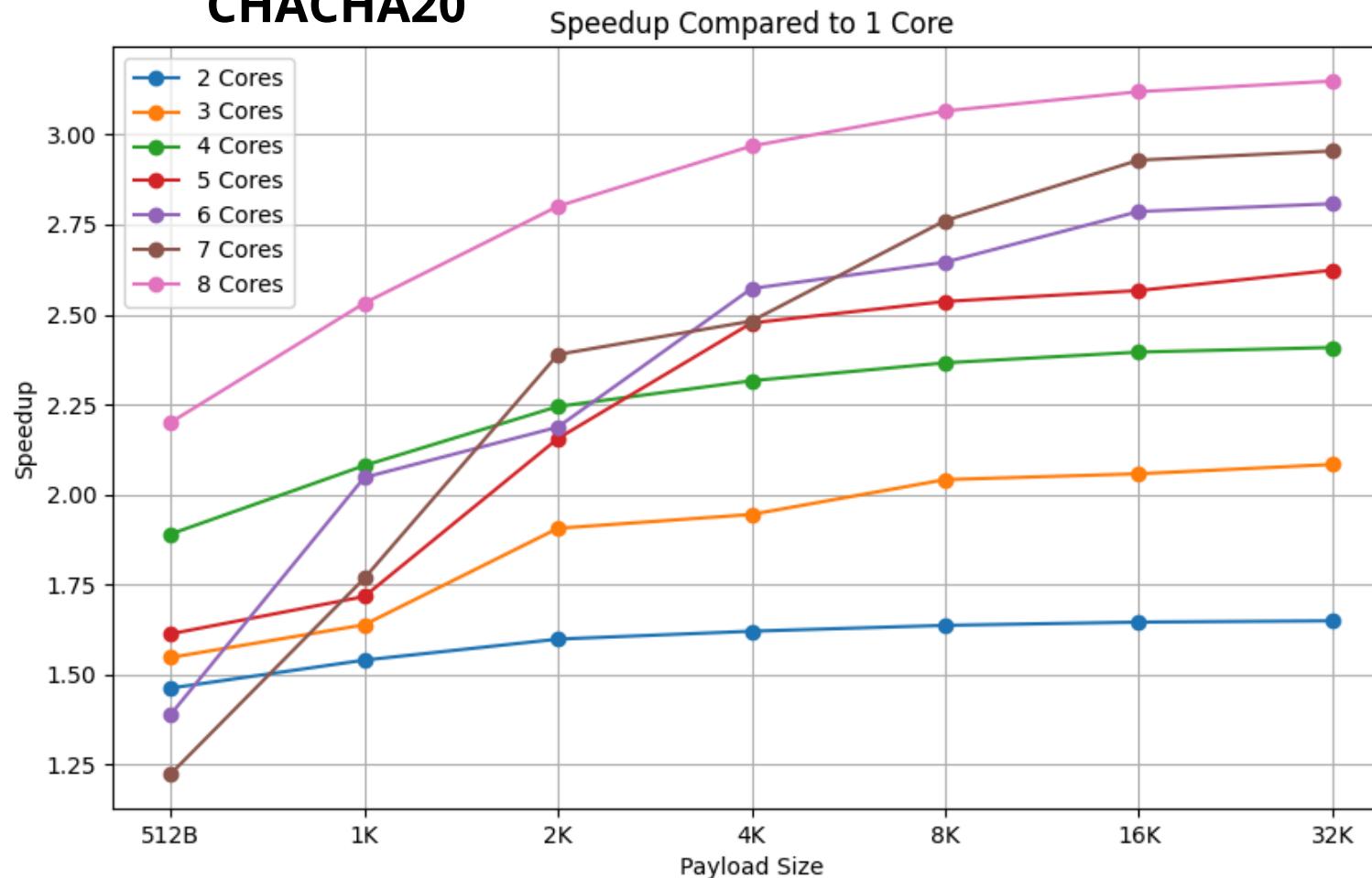
AES256CTR



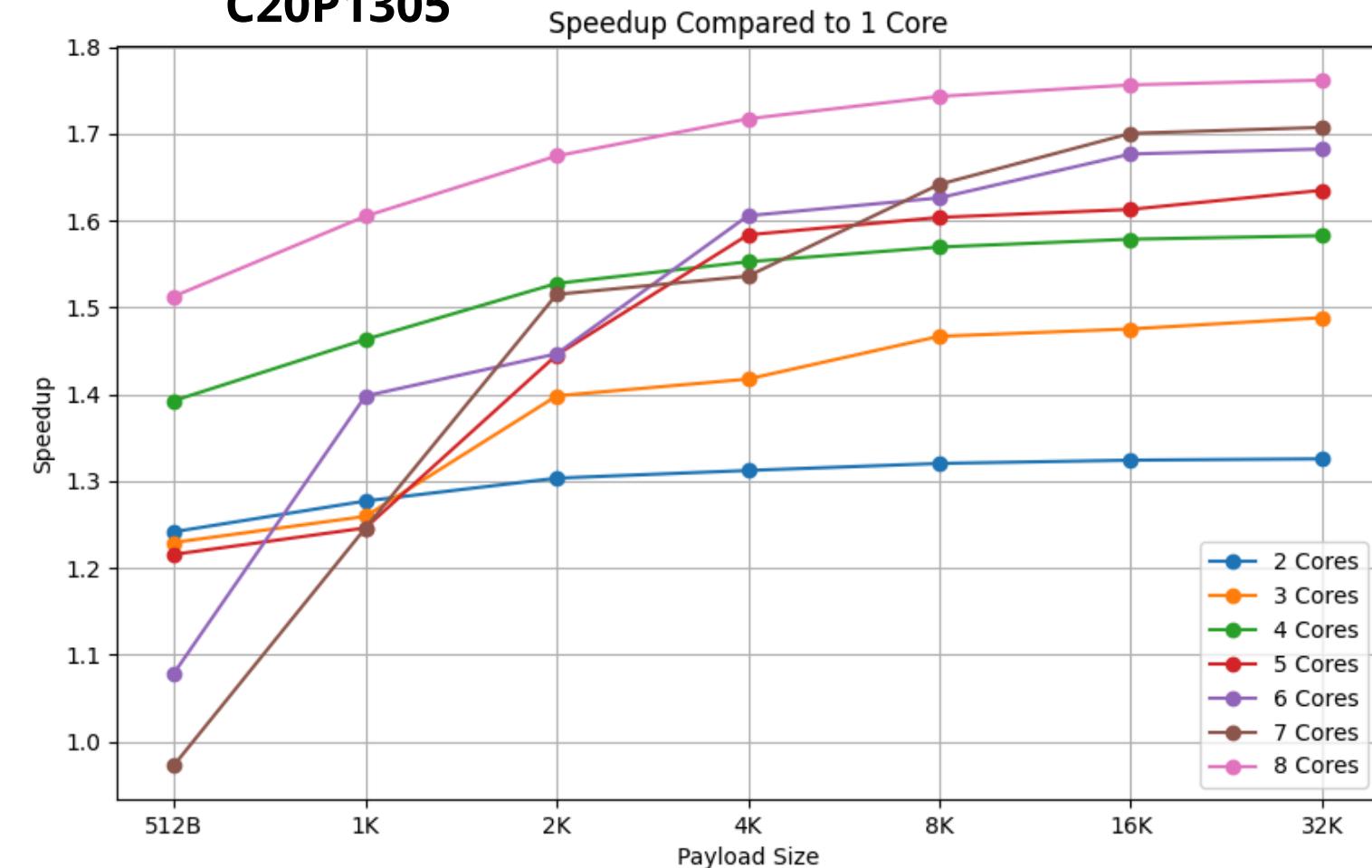
AES256GCM



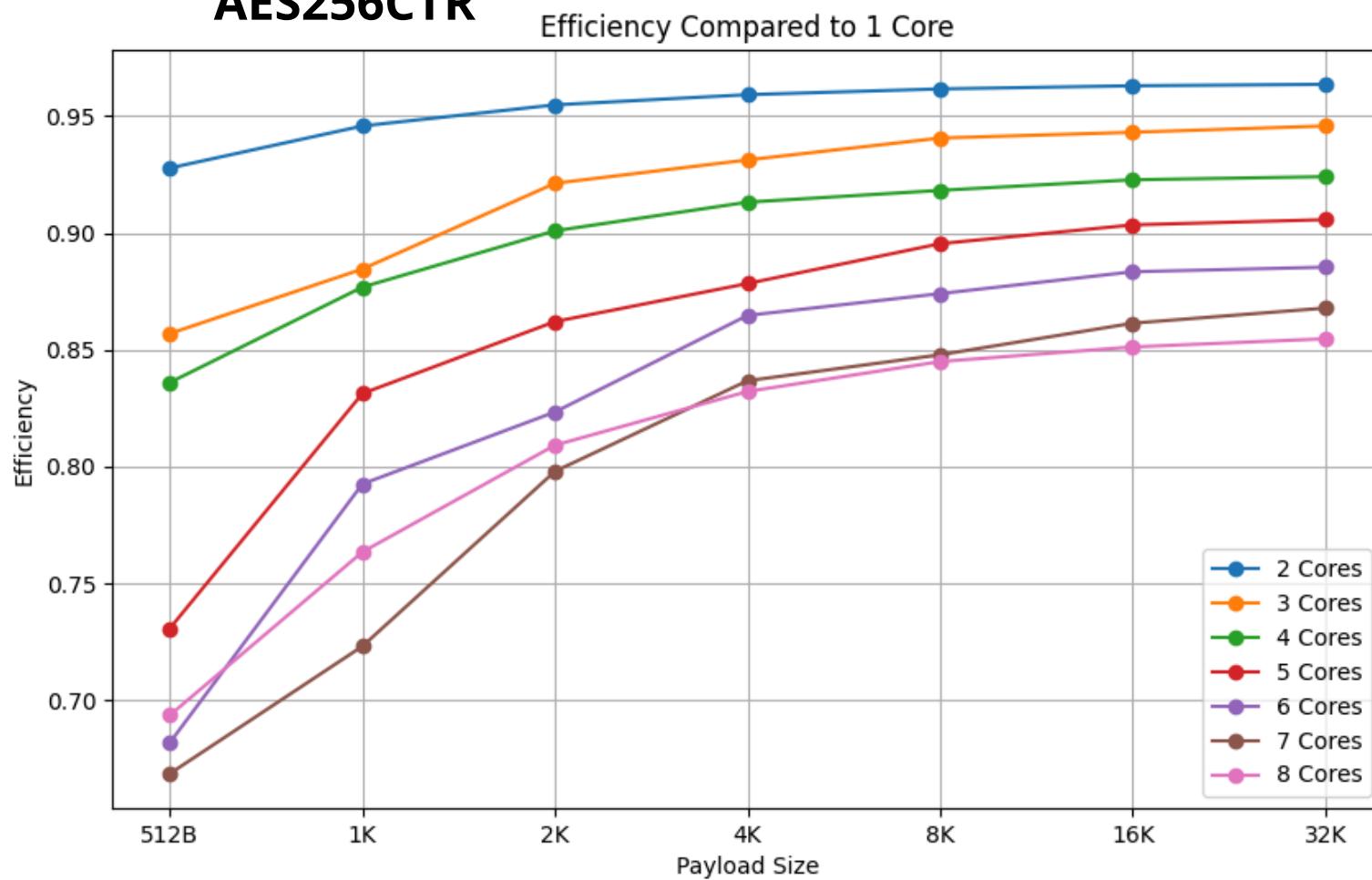
CHACHA20



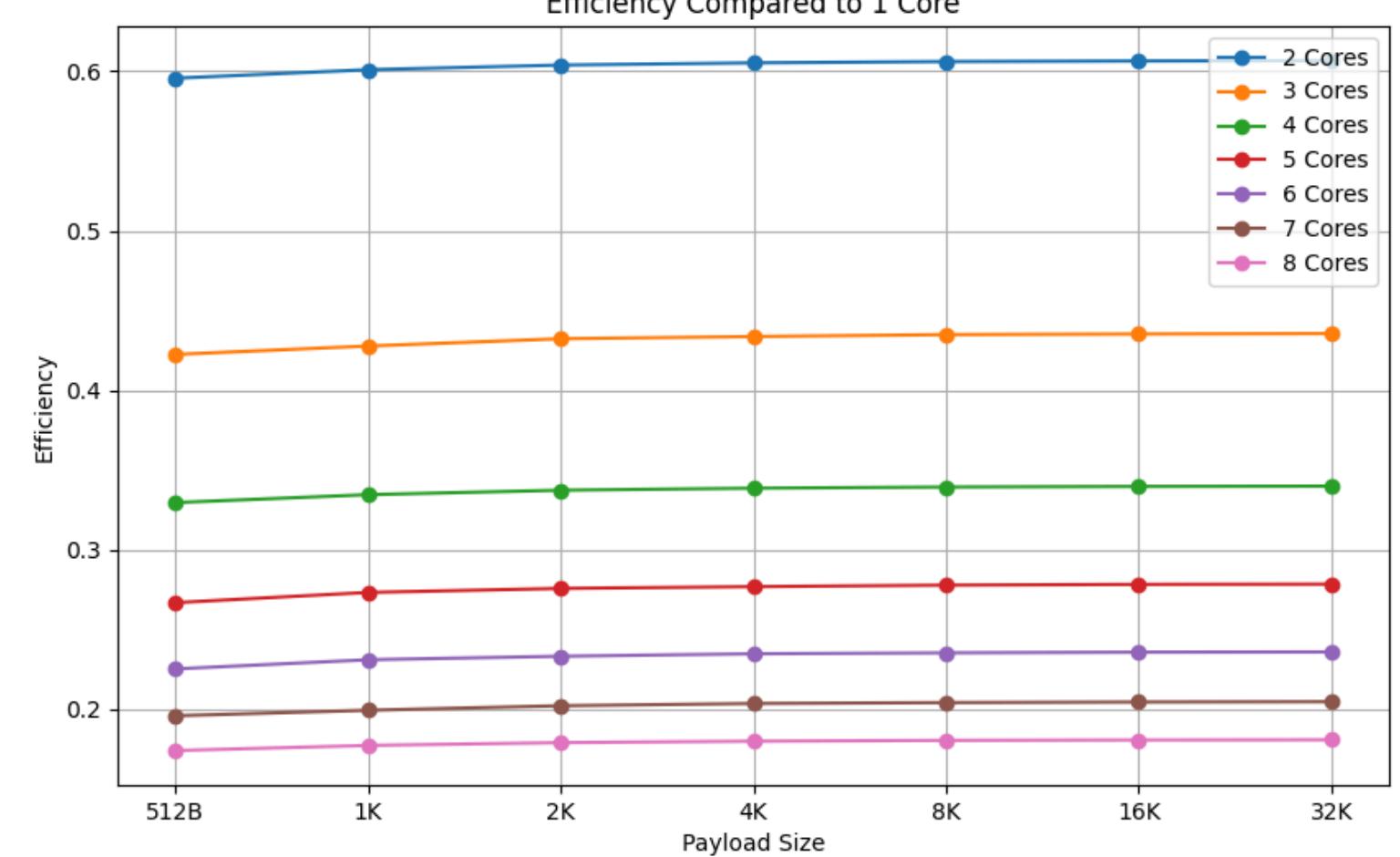
C20P1305



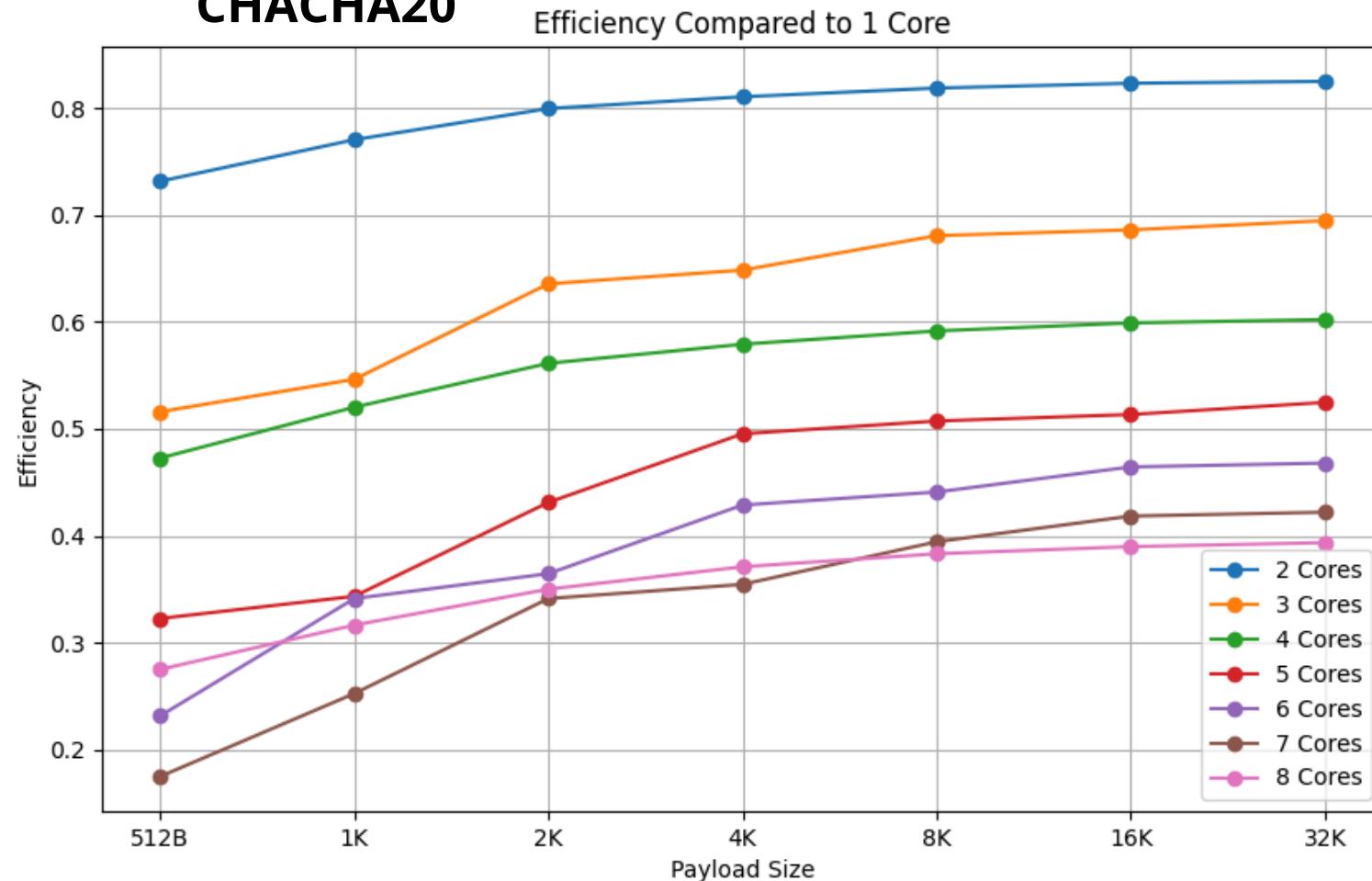
AES256CTR



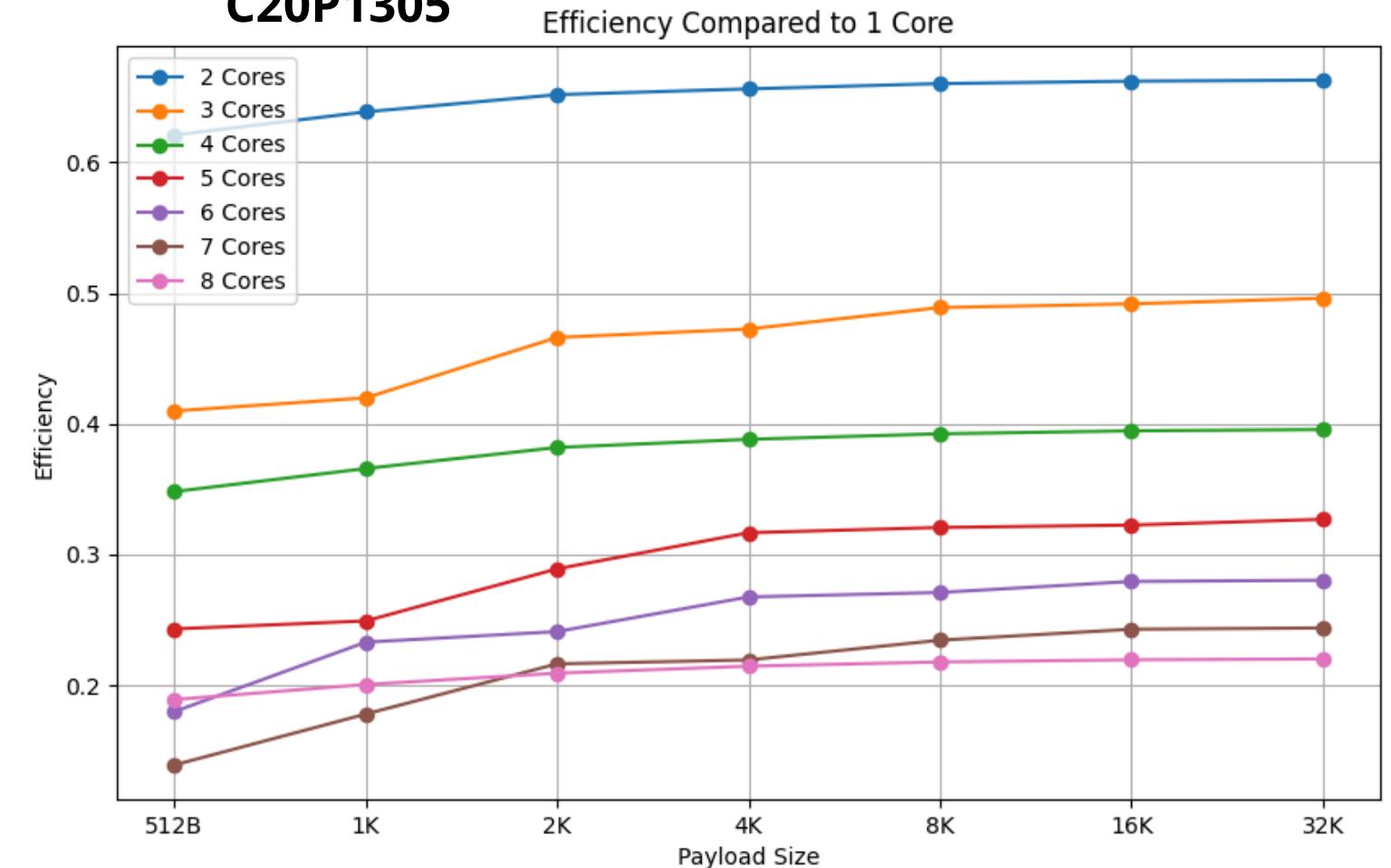
AES256GCM



CHACHA20



C20P1305



NOTE: TRADEOFF EXECUTION/DMA

To understand whether the bottleneck of the implementation is the memory management (DMA transactions) or the execution we need to analyze the trace file comparing the sequence of instructions with the “pi_cl_dma_wait” function, which stalls the execution until the memory transfer is completed.

```
static inline void pi_cl_dma_cmd_wait(pi_cl_dma_cmd_t *cmd)
{
    __cl_dma_wait((pi_cl_dma_cmd_t *)cmd);
}

static inline void __cl_dma_wait(pi_cl_dma_cmd_t *copy)
{
    int counter = copy->id;

    eu_mutex_lock_from_id(0);

    while(DMA_READ(MCHAN_STATUS_OFFSET) & (1 << counter)) {
        eu_mutex_unlock_from_id(0);
        eu_evt_maskWaitAndClr(1<<ARCHI_CL_EVT_DMA0);
        eu_mutex_lock_from_id(0);
    }

    plp_dma_counter_free(counter);

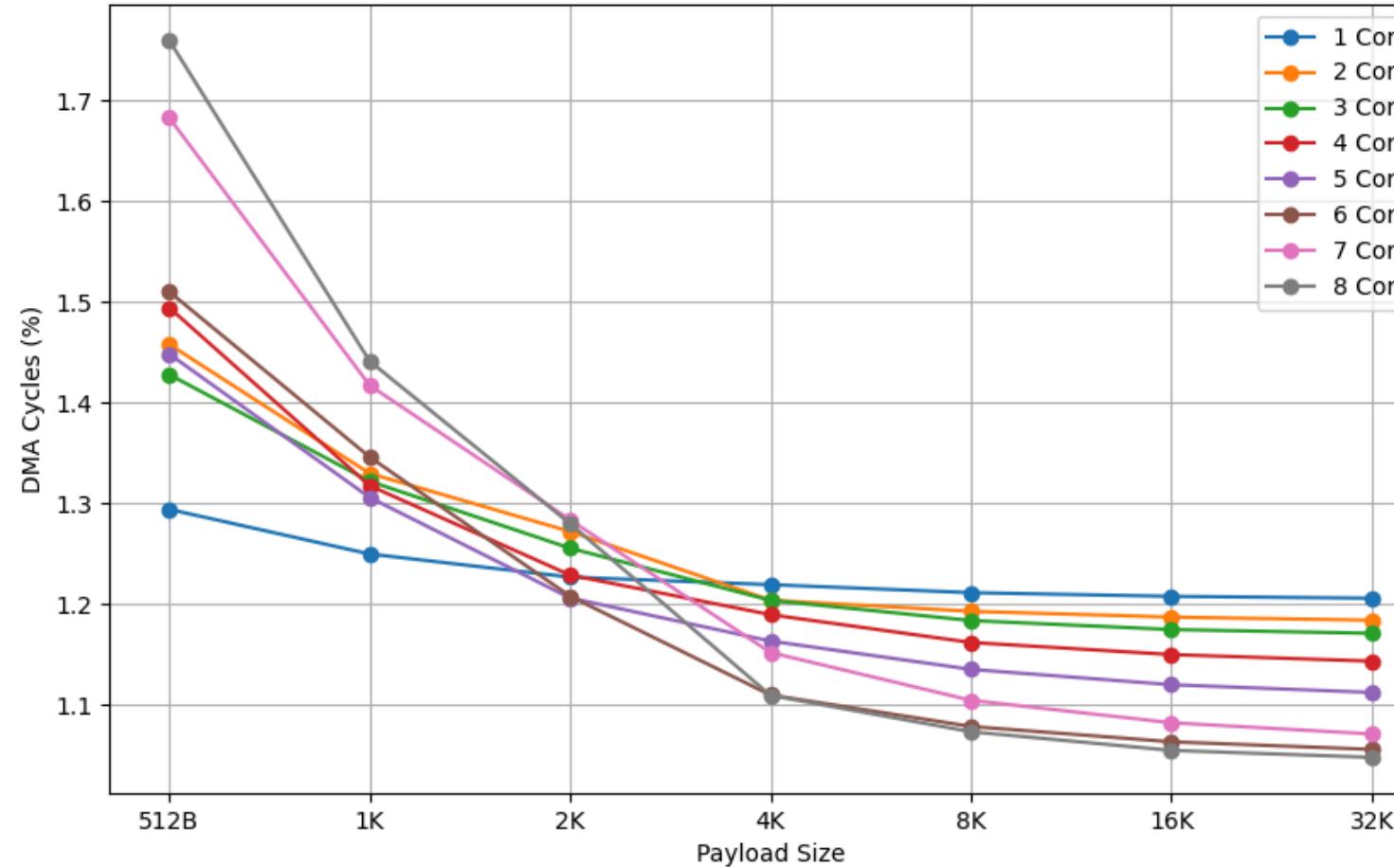
    eu_mutex_unlock_from_id(0);
}
```

As expected, by looking at the trace files we never experienced the fulfilment of the while conditions.

Wait cycles

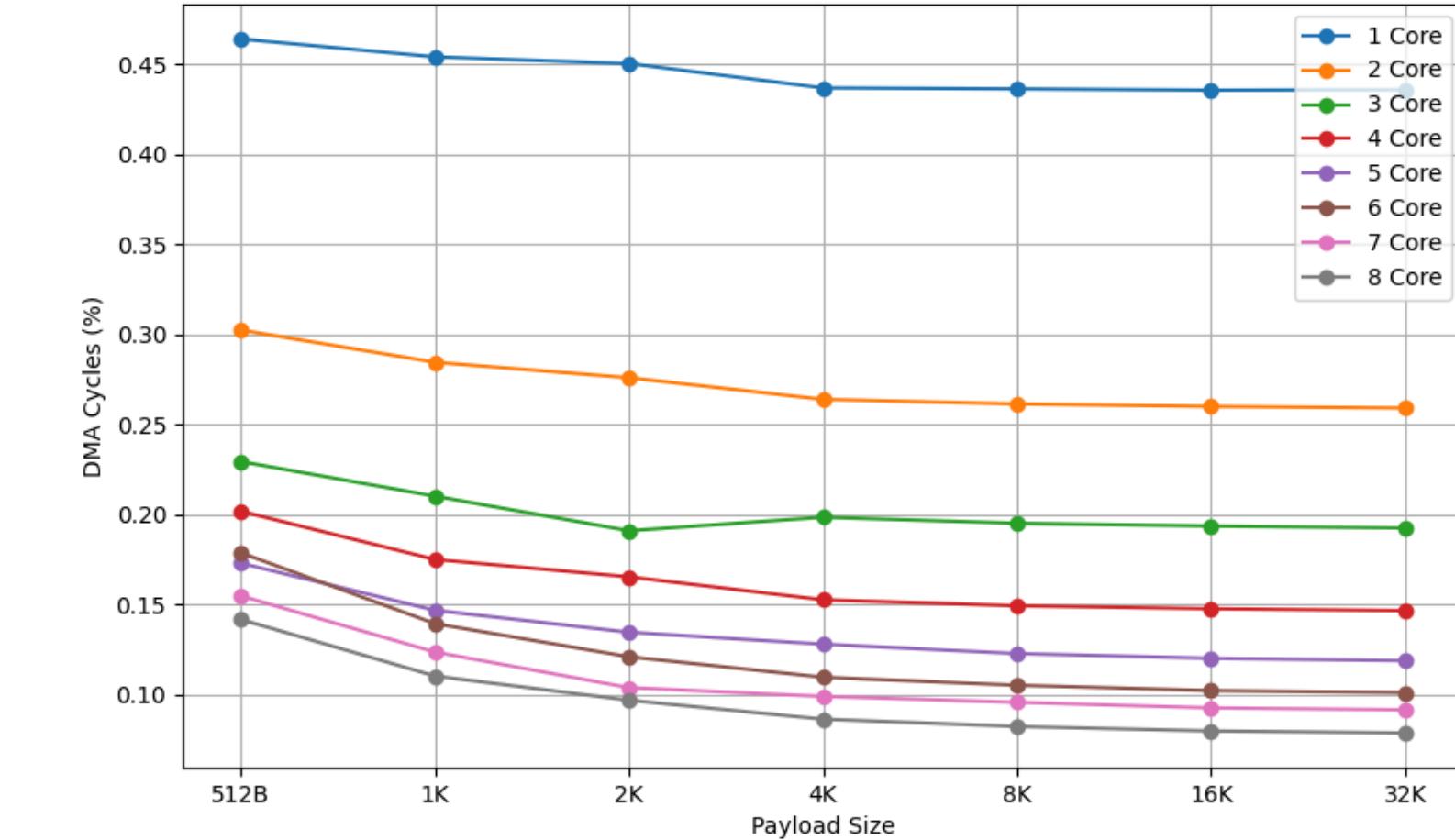
AES256CTR

DMA Cycles as a Percentage of Total Cycles



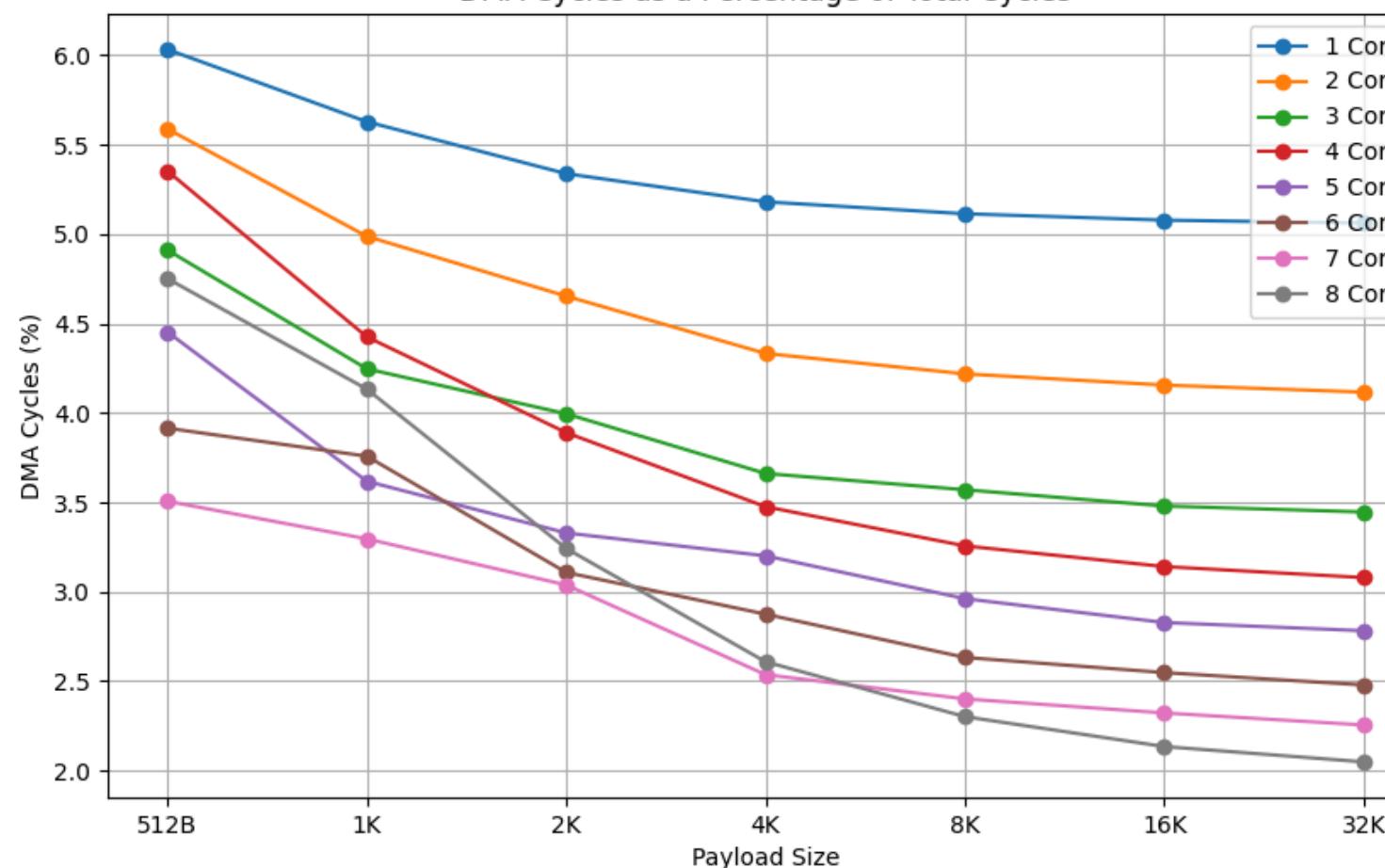
AES256GCM

DMA Cycles as a Percentage of Total Cycles



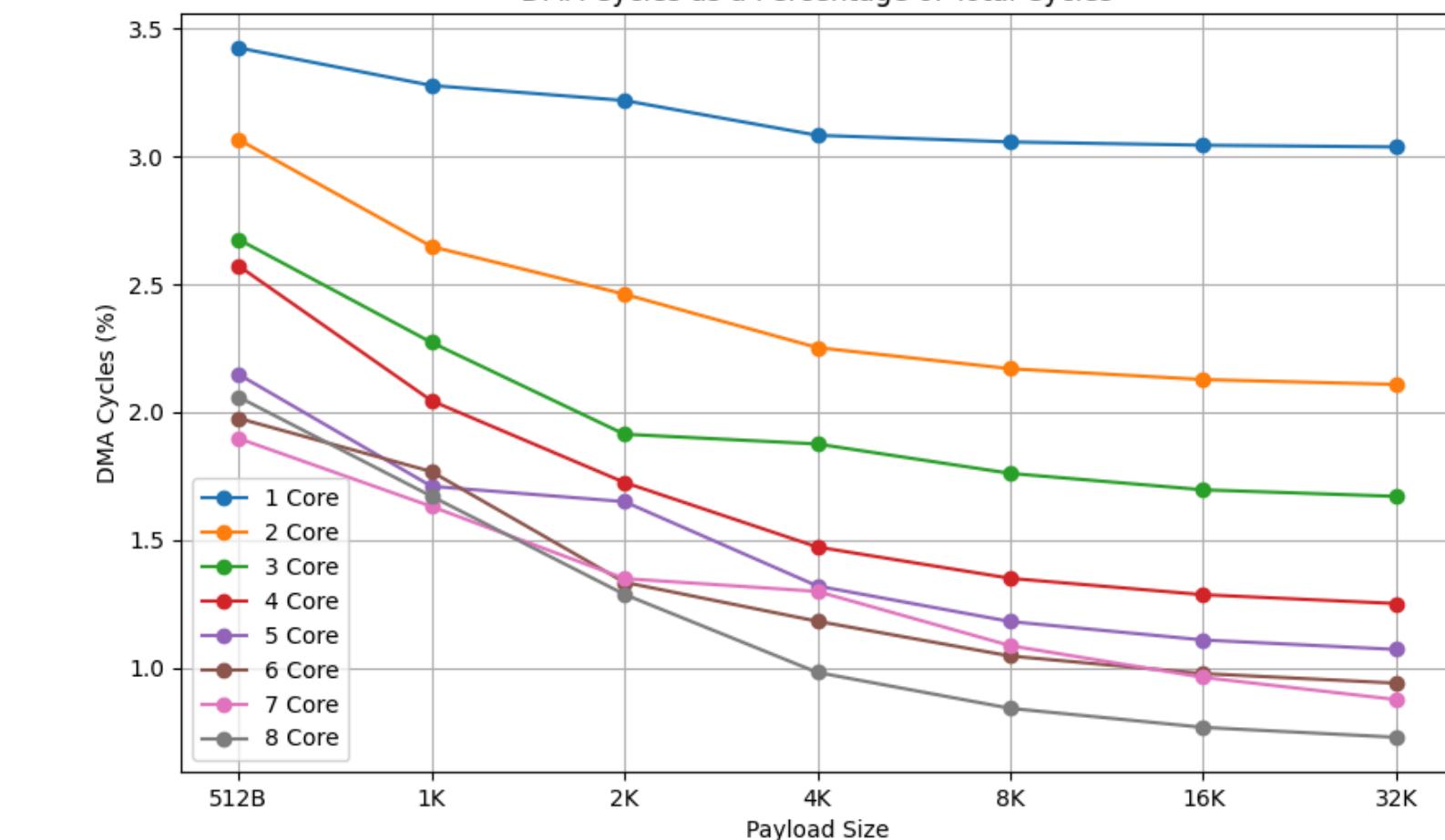
CHACHA20

DMA Cycles as a Percentage of Total Cycles



C20P1305

DMA Cycles as a Percentage of Total Cycles



CONCLUSION

GAP8 architecture has showed as a suitable platform for stream cipher encryption algorithms, exploiting its 8-cores cluster and multi-channel DMA to maximize performances.

Possible future steps will consider FC-Cluster synchronization to achieve maximum efficiency even in case of Authenticated Encryption algorithms.

THANKS FOR YOUR ATTENTION!

More information in our repo:

https://gitlab.com/ecs-lab/projects/cps-crypto-on-pulp/-/tree/final_version

	Matteo Franchi	Riccardo Gaspari	Giulio Vignati
AES CTR	✓	✓	
AES GCM	✓	✓	
CHACHA20			✓
CHACHA20-POLY1305		✓	✓
PARALLELIZZAZIONE		✓	
SCRIPT PERFORMANCE		✓	✓