Matrix Chain Multiplication Algorithmic Design

Alberto Casagrande Email: acasagrande@units.it

a.y. 2020/2021

Consider the matrices A_1, A_2, A_3

- A_1 having dimension 50×5
- A_2 having dimension 5×100
- A_3 having dimension 100×10

How many scalar multiplications does $A_1 \times A_2 \times A_3$ require?

Matrix product is associative i.e., $(A_1 \times A_2) \times A_3 = A_1 \times (A_2 \times A_3)$



Matrix product is associative i.e.,
$$(A_1 \times A_2) \times A_3 = A_1 \times (A_2 \times A_3)$$

• if we compute $(A_1 \times A_2) \times A_3$

$$50 * 100 * 5 = 25000$$
 (to compute $A_1 \times A_2$)
 $50 * 10 * 100 = 50000$ (to compute $(A_1 \times A_2) \times A_3$)

if we compute
$$A_1 \times (A_2)$$

$$5 * 10 * 100 = 5000$$
 (to compute $A_2 \times A_2$)

$$50 * 10 * 5 = 2500$$
 (to compute $A_1 \times (A_2 \times A_3)$)

$$75000 ((A_1 \times A_2) \times A_3) \text{ vs}$$

Matrix product is associative i.e., $(A_1 \times A_2) \times A_3 = A_1 \times (A_2 \times A_3)$

• if we compute $(A_1 \times A_2) \times A_3$

$$50 * 100 * 5 = 25000$$
 (to compute $A_1 \times A_2$)
 $50 * 10 * 100 = 50000$ (to compute $(A_1 \times A_2) \times A_3$)

• if we compute $A_1 \times (A_2 \times A_3)$

$$5*10*100 = 5000$$
 (to compute $A_2 \times A_3$)
 $50*10*5 = 2500$ (to compute $A_1 \times (A_2 \times A_3)$)

75000
$$((A_1 \times A_2) \times A_3)$$
 vs 7500 $(A_1 \times (A_2 \times A_3))$

Consider the chain of matrices $\langle A_1, \ldots, A_n \rangle$ where

Consider the chain of matrices $\langle A_1, \ldots, A_n \rangle$ where A_i has dimensions $p_{i-1} \times p_i$ for all $i \in [1, n]$

Consider the chain of matrices $\langle A_1, \dots, A_n \rangle$ where A_i has dimensions $p_{i-1} \times p_i$ for all $i \in [1, n]$

Compute a parenthesization that minimizes the # of scalar products for the chain multiplication

Why Are We Interested in Matrix Chain Multiplication?

- Deep neural networks evaluation depends on matrix multiplication (ever heard it?)
- May bring an important speedup in data preparation pipeline
- A good example for its class of solution strategies



Recursive Solution

We may try to search among all the possible parenthesizations

- if n = 1, the parenthesization is obvious

Recursive Solution

We may try to search among all the possible parenthesizations

- if n = 1, the parenthesization is obvious
- if n > 1, the chain can be parenthetized as

$$(A_1 \times \ldots A_k) \times (A_{k+1} \times \ldots A_n)$$

for any $k \in [1, n-1]$. Recursively produce the parenthesizations for $\langle A_1, \dots, A_k \rangle$ and $\langle A_{k+1}, \dots, A_n \rangle$

Recursive Solution

We may try to search among all the possible parenthesizations

- if n = 1, the parenthesization is obvious
- if n > 1, the chain can be parenthetized as

$$(A_1 \times \ldots A_k) \times (A_{k+1} \times \ldots A_n)$$

for any $k \in [1, n-1]$. Recursively produce the parenthesizations for $\langle A_1, \dots, A_k \rangle$ and $\langle A_{k+1}, \dots, A_n \rangle$

How many parenthesizations has $\langle A_1, \dots, A_n \rangle$?

$$\langle A_1, \dots, A_n
angle$$
 has

$$P(n) = \begin{cases} 1 & \text{if } n = 1\\ \sum_{k=1}^{n-1} P(k) * P(n-k) & \text{if } n > 1 \end{cases}$$

different parenthesizations

Counting Parenthesizations

$$\langle A_1,\ldots,A_n
angle$$
 has

$$P(n) = \begin{cases} 1 & \text{if } n = 1\\ \sum_{k=1}^{n-1} P(k) * P(n-k) & \text{if } n > 1 \end{cases}$$

different parenthesizations

It can be proved that $P(n) \in \Omega(2^n)$

Counting Parenthesizations

 $\langle A_1,\ldots,A_n \rangle$ has

$$P(n) = \begin{cases} 1 & \text{if } n = 1\\ \sum_{k=1}^{n-1} P(k) * P(n-k) & \text{if } n > 1 \end{cases}$$

different parenthesizations

It can be proved that $P(n) \in \Omega(2^n)$

Too many parenthesizations to be enumerated!!! (if you don't believe it, try for n = 8)

- if $(A_1 \times ... \times A_k) \times (A_{k+1} \times ... \times A_n)$ is optimal for the chain, the 1st part is optimal for $(A_1, ... A_k)$ the 2nd part is optimal for $(A_{k+1}, ... A_n)$
 - may bridge s of a recursive approach perform the
 - e.g. or every parenthesization of $A_1 \times \ldots \times A_k$, the parenthesization of $A_{k+1} \times \ldots \times A_n$

- if $(A_1 \times ... \times A_k) \times (A_{k+1} \times ... \times A_n)$ is optimal for the chain,
 - the 1st part is optimal for $\langle A_1, \dots A_k \rangle$
 - the 2nd part is optimal for $(A_{k+1}, \dots A_n)$
- many branches of the "naive" recursive approach perform the very same computations
 - e.g. for every parenthesization of $A_1 \times ... \times A_k$, the parenthesizations are recomputed $A_{k+1} \times ... \times A_n$

- if $(A_1 \times ... \times A_k) \times (A_{k+1} \times ... \times A_n)$ is optimal for the chain,
 - the 1st part is optimal for $\langle A_1, \dots A_k \rangle$
 - the 2nd part is optimal for $\langle A_{k+1}, \dots A_n \rangle$
- many branches of the "naive" recursive approach perform the very same computations
 - e.g. for every parenthesization of $A_1 \times ... \times A_k$, the parenthesizations are recomputed $A_{k+1} \times ... \times A_n$

- if $(A_1 \times ... \times A_k) \times (A_{k+1} \times ... \times A_n)$ is optimal for the chain,
 - the 1st part is optimal for $\langle A_1, \dots A_k \rangle$
 - the 2nd part is optimal for $\langle A_{k+1}, \dots A_n \rangle$
- many branches of the "naïve" recursive approach perform the very same computations
 - e.g., for every parenthesization of $A_1 \times ... \times A_k$, the parenthesizations are recomputed $A_{k+1} \times ... \times A_n$

- if $(A_1 \times ... \times A_k) \times (A_{k+1} \times ... \times A_n)$ is optimal for the chain,
 - the 1st part is optimal for $\langle A_1, \dots A_k \rangle$
 - the 2nd part is optimal for $\langle A_{k+1}, \dots A_n \rangle$
- many branches of the "naïve" recursive approach perform the very same computations
 - e.g., for every parenthesization of $A_1 \times ... \times A_k$, the parenthesizations are recomputed $A_{k+1} \times ... \times A_n$

Idea:

Recursively compute optimal parenthesizations and use dynamic programming



Dynamic Programming Solution

Store the minimum # of products for all the sub-chains in m

Recursively, compute m[i,j] as:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{k \in [i,j-1]} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

Dynamic Programming Solution

Store the minimum # of products for all the sub-chains in m

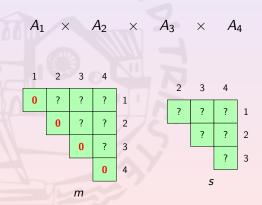
Recursively, compute m[i,j] as:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{k \in [i,j-1]} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

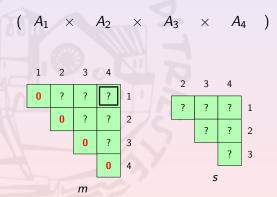
For each i, j also store in s[i, j] the k that minimizes

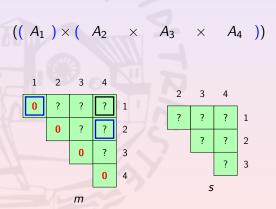
$$m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$$

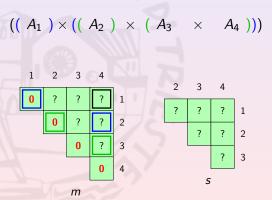
i.e., the parenthesization for the current level

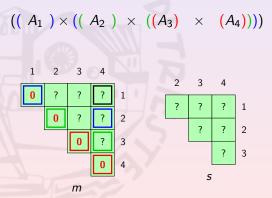


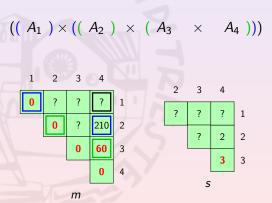
Problem Definition

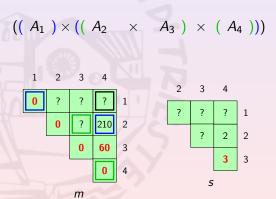


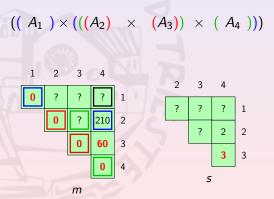


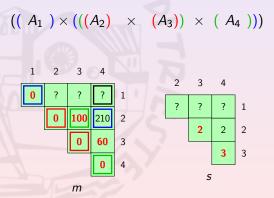


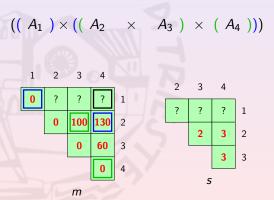


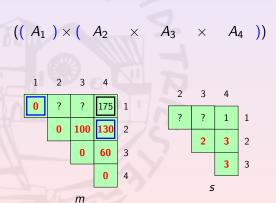


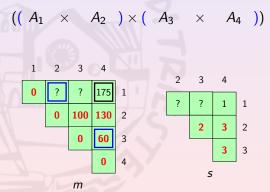


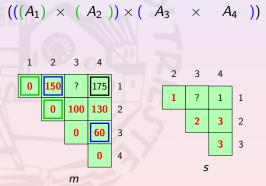


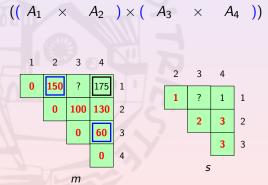


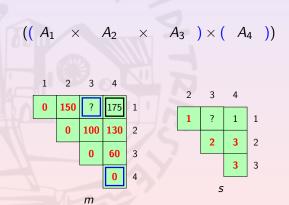


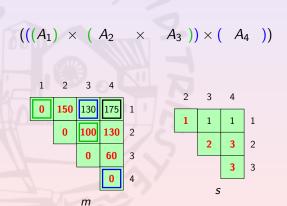


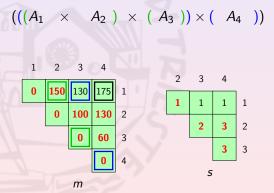


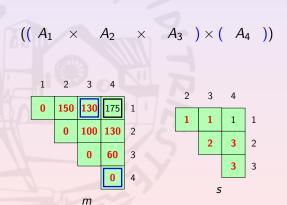


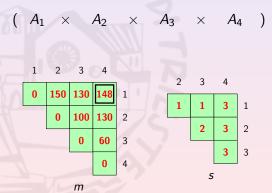


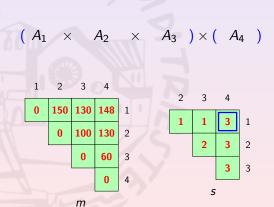


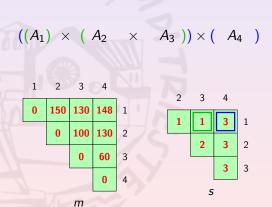


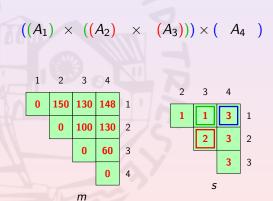




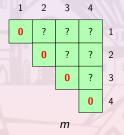


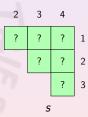






Both m and s can be computed iteratively from the shortest sub-chains to the longest one.

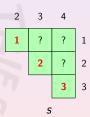




Dynamic Programming Solution: An Iterative Version

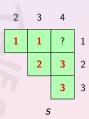
Both m and s can be computed iteratively from the shortest sub-chains to the longest one.

	1	2	3	4		
	0	150	?	?	1	
		0	100	?	2	
		廛	0	60	3	
				0	4	
m						



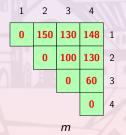
Both m and s can be computed iteratively from the shortest sub-chains to the longest one.

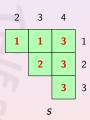
1	2	3	4			
0	150	130	?	1		
	0	100	130	2		
	Æ	0	60	3		
		0	4			
m						



Dynamic Programming Solution: An Iterative Version

Both m and s can be computed iteratively from the shortest sub-chains to the longest one.





Dynamic Programming Solution: Code

```
def MatrixChain(P):
   m \leftarrow allocate(1..n, 1..n)
    s \leftarrow allocate(1..n-1, 2..n)
    for i \leftarrow 1..n:
       m[i, i] \leftarrow 0
    for 1 \leftarrow 2, n:
        for i \leftarrow 1..(n-l+1):
            i \leftarrow i + l - 1
            MatrixChainAux(P,m,s,i,j)
        endfor
     endfor
     return (m, s)
enddef
```

Dynamic Programming Solution: Code

```
def MatrixChainAux(P,m,s,i,j):
   m[i,j] \leftarrow INFINITY
    for k \leftarrow i ...(j-1):
       q \leftarrow m[i,k] + m[k+1,j] +
                 P[i-1]*P[k]+P[i]
        if q < m[i,j]:
           m[i,j] \leftarrow q
           s[i,i] \leftarrow k
        endif
    endfor
enddef
```

The computation of m[i,j] takes time:

$$\sum_{k=i}^{(j-1)} \Theta(1) = \Theta(j-i)$$

Since $i \in [1, n]$ and $j \in [i, n]$,

$$T_C(n) = \sum_{i=n}^n \sum_{j=i}^n \Theta(j-i) = \Theta\left(\sum_{i=1}^n \left(\sum_{j=i}^n j\right) - n * i\right)$$
$$= \Theta\left(\sum_{j=1}^n \frac{n * (n+1)}{2} - \frac{i * (i+1)}{2} - n * i\right) = \Theta\left(n^3\right)$$