# Short notes on computational Linear Algebra.

Fabio Affinito

SCAI dept.

CINECA

# BASICS ALGORITHMS AND NOTATION 

$$A \in \mathbb{R}^{m \times n} \iff A = (a_{ij}) = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ & \ddots & \vdots \\ a_{m1} & & a_{mn} \end{bmatrix} \qquad a_{ij} \in \mathbb{R}$$

MATRIX OPERATIONS:

- Transposition: $C = A^T \iff c_{ij} = a_{ji}$
- Addition: $C = A + B \iff c_{ij} = a_{ij} + b_{ij}$
- Scalar-Matrix multiplication: $C = \alpha A \iff c_{ij} = \alpha a_{ij}$
- Matrix-Matrix multiplication: $C = AB \iff c_{ij} = \sum_{k=1}^{r} a_{ik} b_{kj}$

VECTOR NOTATION

$$x \in \mathbb{R}^n \iff x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad x_i \in \mathbb{R}$$

# VECTOR OPERATIONS

- Dot product : $c = x^T y \iff \sum_{i=1}^{N} x_i y_i$
- Vector multiply : $z = x * y$
  $z_i = x_i y_i$ (Hadamard product)
- SAXPY

  $y = ax + y \iff y_i = a x_i + y_i$

... COMPUTATIONALLY SPEAKING ..

Dot product:

```
c = 0
for i = 1:n
    c = c + x(i)y(i)
end
```

$O(n)$

SAXPY

```
for i = 1:n
    y(i) = x(i) a + y(i)
end
```

$O(n)$

# MATRIX - VECTOR MULTIPLICATION (GAXPY)

(generalized)

$$y = Ax + y \iff y_i = \sum_{j=1}^{N} a_{ij} x_j + y_i$$

## GAXPY (ROW VERSION)

$A \in \mathbb{R}^{m \times n}$ $x \in \mathbb{R}^n$ $y \in \mathbb{R}^m$

$y \mapsto y = Ax + y$

```
for i = 1:m
    for y = 1:n
        y(i) = A(i,j) x (j) + y(i)
    end
end
```

We can look at $Ax$ as a combination of A's columns

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} 7 \\ 8 \end{bmatrix} = 7 \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} + 8 \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$$

## GAXPY (COLUMN VERSION)

```
for j = 1:n
    for i = 1:m
        y(i) = A(i,j) x (j)
               + y(i)
    end
end
```

SAXPY!

# MATRIX - MATRIX MULTIPLICATION

- Dot product formulation

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1\cdot 5 + 2\cdot 7 & 1\cdot 6 + 2\cdot 8 \\ 3\cdot 5 + 4\cdot 7 & 3\cdot 6 + 4\cdot 8 \end{bmatrix}$$

- SAXPY version (each column is lin. comb of columns of A)

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \left[ 5\begin{bmatrix} 1 \\ 3 \end{bmatrix} + 7\begin{bmatrix} 2 \\ 4 \end{bmatrix};\ 6\begin{bmatrix} 1 \\ 3 \end{bmatrix} + 8\begin{bmatrix} 2 \\ 4 \end{bmatrix} \right]$$

- Outer product version

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \begin{bmatrix} 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 \\ 4 \end{bmatrix} \begin{bmatrix} 7 & 8 \end{bmatrix}$$

# MATRIX MULTIPLICATION : ijk VARIANT

$C = A \cdot B + C$

```
for i = 1:m
  for j = 1:n
    for k = 1:p
      C(i,j) = A(i,k)B(k,j) + C(i,j)
    end
  end
end
```

} 3 loops : 3! = 6 variations !

Each variant involves the same amount of floating point operations **but** accesses the A, B, C differently

is the $jki$ variant. Each of the six possibilities ($ijk$, $jik$, $ikj$, $jki$, $kij$, $kji$) features an inner loop operation (dot product or saxpy) and has its own pattern of data flow. For example, in the $ijk$ variant, the inner loop oversees a dot product that requires access to a row of $A$ and a column of $B$. The $jki$ variant involves a saxpy that requires access to a column of $C$ and a column of $A$. These attributes are summarized in Table 1.1.1 along with an interpretation of what is going on when the middle and inner loop are considered together. Each variant involves the same amount of floating

| Loop Order | Inner Loop | Middle Loop | Inner Loop Data Access |
|---|---|---|---|
| $ijk$ | dot | vector × matrix | $A$ by row, $B$ by column |
| $jik$ | dot | matrix × vector | $A$ by row, $B$ by column |
| $ikj$ | saxpy | row gaxpy | $B$ by row, $C$ by row |
| $jki$ | saxpy | column gaxpy | $A$ by column, $C$ by column |
| $kij$ | saxpy | row outer product | $B$ by row, $C$ by row |
| $kji$ | saxpy | column outer product | $A$ by column, $C$ by column |

TABLE 1.1.1. *Matrix Multiplication: Loop Orderings and Properties*

point arithmetic, but accesses the $A$, $B$, and $C$ data differently.

### 1.1.12 A Dot Product Formulation

The usual matrix multiplication procedure regards $AB$ as an array of dot products to be computed one at a time in left-to-right, top-to-bottom order.

$\lfloor a_m^T \rfloor$ (6)

and

$$B = [\, b_1, \ldots, b_n \,] \qquad b_k \in \mathbb{R}^p$$

then Algorithm 1.1.6 has this interpretation:

**for** $i = 1{:}m$
  **for** $j = 1{:}n$
    $c_{ij} = a_i^T b_j + c_{ij}$
  **end**
**end**

Note that the "mission" of the $j$-loop is to compu update. To emphasize this we could write

**for** $i = 1{:}m$
  $c_i^T = a_i^T B + c_i^T$
**end**

where

$$C = \begin{bmatrix} c_1^T \\ \vdots \\ c_m^T \end{bmatrix}$$

is a row partitioning of $C$. To say the same thing w we write

**for** $i = 1{:}m$
  $C(i,:) = A(i,:)B + C(i,:)$
**end**

Either way we see that the inner two loops of th row-oriented gaxpy operation.

# MATRIX MULTIPLICATION : DOT PRODUCT VERSION

$C = AB + C$

```
for i = 1:m
  for j = 1:n
    C(i,j) = A(i,:)(:,j) +
             + C(i,j)
```

Partitioned matrix notation:

$A(k,:) = [a_{k1} \; \cdots \; a_{kn}]$

$A(:,k) = \begin{bmatrix} a_{1k} \\ \vdots \\ a_{1m} \end{bmatrix}$

$\Rightarrow$

```
for i = 1:m
  for j = 1:n
    c_ij = a_i^T b_j + c_ij
  end
end
```

$\Rightarrow$

```
for i = 1:m
  c_i^T = a_i^T B + c_i^T
end
```

It's a GAXPY

# MATRIX MULTIPLICATION : SAXPY VERSION

$$c_j = \sum_{k=1}^{p} b_{kj} a_k + c_j \qquad j = 1:n$$

(see slide 4)

$$A = [a_1 \cdots a_p] \quad a_j \in \mathbb{R}^m$$

$$C = [c_1 \cdots c_n] \quad c_j \in \mathbb{R}^m$$

```
for j = 1:n
  for k = 1:p
    C(:,j) = A(:,k)B(k,j)
             + C(:,j)
  end
end
```

```
for j = 1:n
  C(:,j) = A·B(:,j) + C(:,j)
end
```

A·B(:,j) : SAXPY

# MATRIX MULTIPLICATION : OUTER PRODUCT

$A \in \mathbb{R}^{m \times p}$

$B \in \mathbb{R}^{p \times n}$

$C \in \mathbb{R}^{m \times n}$

```
for k = 1:p
  for j = 1:n
    for i = 1:m
      C(i,j) = A(i,k)B(k,j) + C(i,j)
    end
  end
end
```

```
for k = 1:p
  C = A(:,k)B(k,:) + C
end
```

$$C = a_k b_k^T + C$$

# EXPLOIT THE STRUCTURE !

## TRIANGULAR MATRIX MULTIPLICATION

$$C = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ 0 & a_{22}b_{22} & a_{22}b_{23} + a_{23}b_{33} \\ 0 & 0 & a_{33}b_{33} \end{bmatrix}$$

$C = A \cdot B =$ 

$$c_{ij} = \sum_{k=i}^{j} a_{ik} b_{kj}$$

$a_{ik} b_{kj} = 0$
if $k < i$
or $j < k$

```
C=0
for i=1:n
 for j=i:n
  for k=i:j
    C(i,j) = A(i,k)B(k,j) + C(i,j)
  end x3
```

# HOW MUCH DID WE SAVE? COUNT THE FLOPS

FLOP = FLOating Point Operation

Dot product (saxpy) = $2n$ flops

Gaxpy (outer product) = $m \times n$ flops

$C = A \cdot B + C$ = $2 \times m \times n \times p$ flops

$A, B \in \mathbb{R}^{n \times n}$

```
C = 0
for i = 1:n
  for j = i:n
    for k = i:j
      C(i,j) = A(i,k) B(k,j) + C(i,j)
    end
  end
end
```

$$\sum_{i=1}^{n} \sum_{j=1}^{n} 2(j-i+1) =$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{n-i+1} 2j \cong$$

$$\cong \frac{2(n-i+1)^2}{2} = \sum_{i=1}^{n} i^2 = \frac{n^3}{3}$$

$$\sum_{1}^{q} p \cong \frac{q^2}{2}$$

$$\sum_{1}^{q} p^2 \cong q^3/3$$

# BLOCK MATRICES

We can partition a $m \times n$ $A$ matrix in blocks : $A = \begin{bmatrix} A_{11} & \cdots & A_{1r} \\ A_{q1} & & A_{qr} \end{bmatrix} \begin{matrix} m_1 \\ m_q \end{matrix}$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad n_1 \qquad\qquad n_r$

where $m_1 + \cdots + m_q = m$

$n_1 + \cdots + n_r = n$

$A_{\alpha\beta}$ is a block of dimension $m_\alpha \times n_\beta$

$A = (A_{\alpha\beta})$ is a $q \times r$ block matrix

**Th**: If $A = \begin{bmatrix} A_{11} & \cdots & A_{1s} \\ \vdots & & \vdots \\ A_{q1} & \cdots & A_{qs} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix}$ ; $B = \begin{bmatrix} B_{11} & \cdots & B_{1r} \\ & & \\ B_{s1} & & B_{sr} \end{bmatrix} \begin{matrix} p_1 \\ \\ p_s \end{matrix}$

$\qquad\qquad\qquad\qquad\qquad\qquad\quad p_1 \qquad\qquad p_s \qquad\qquad\qquad\qquad\qquad\quad n_1 \qquad\qquad n_r$

and $C = \begin{bmatrix} C_{11} & \cdots & C_{1r} \\ & & \\ C_{q1} & & C_{qr} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix}$ $\Rightarrow$ $\boxed{C_{\alpha\beta} = \sum_{\gamma=1}^{s} A_{\alpha\gamma} B_{\gamma\beta}}$

$\qquad\qquad\qquad\qquad\qquad\qquad\quad n_1 \qquad\qquad n_r$

$\alpha = 1 : q$

$\beta = 1 : r$

(proof : cfr Golub - VL pg. 27)

# BLOCK MATRIX MULTIPLICATION

Let's suppose $A, B, C$ are all $N \times N$ made of blocks of size $l$

$A = (A_{\alpha\beta})$

$B = (B_{\alpha\beta})$

$C = (C_{\alpha\beta})$

$$C_{\alpha\beta} = \sum_{\gamma=1}^{N} A_{\alpha\gamma} B_{\gamma\beta} + C_{\alpha\beta}$$

$\alpha = 1:N \qquad \beta = 1:N$

```
for α = 1:N
    i = (α-1)l+1 : αl
    for β = 1:N
    j = (β-1)l+1 : βl
        for γ = 1:N
        k = (γ-1)l+1 : γl
            C(i,j) = A(i,k)B(k,j)+C(i,j)
        end
    end
end
```

N.B. for $l = 1$

$\alpha \equiv i$

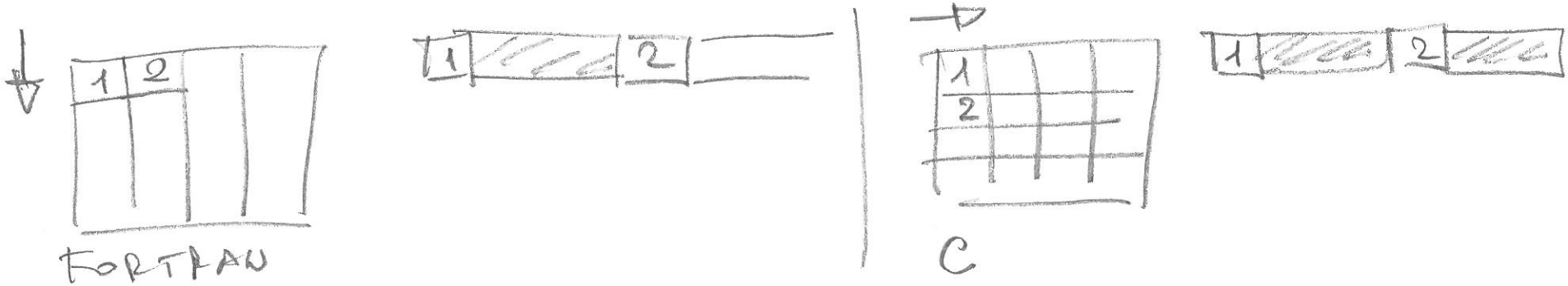
$\beta \equiv j$

$\gamma \equiv k$

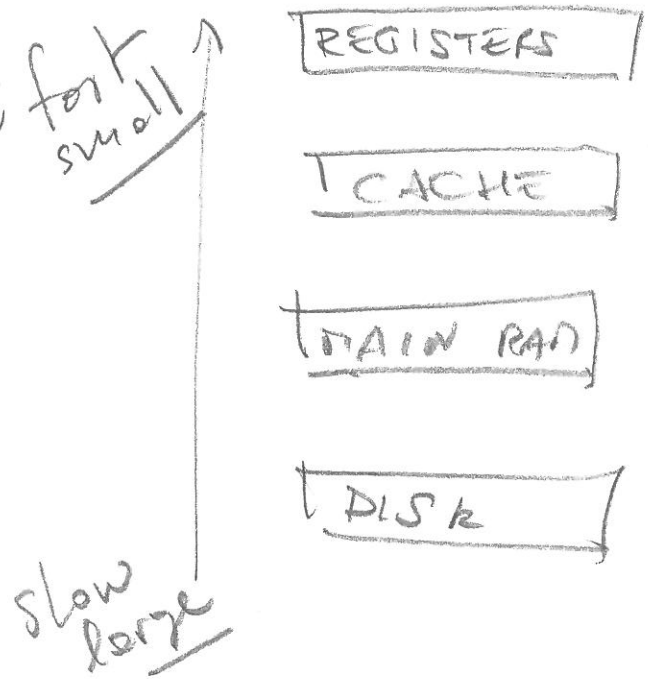
we recover the well know algorithm

# Optimisation KEY FACTORS

1. Computational complexity: how many flop?
2. Access to data: stride, vector units...




3. Exploiting DATA PERSISTENCE; for small
cache-blocking, etc.



# BLOCKING AND RE-USE

$C = A \cdot B + C$

$A, B, C \in \mathbb{R}^{n \times n}$ reside in the main memory

$n >> \text{cache dimension}$ (M)

Strategy:

$$B = \begin{bmatrix} B_1 & \cdots & B_N \\ \ell & & \ell \end{bmatrix}$$

$$C = \begin{bmatrix} C_1 & \cdots & C_N \\ \ell & & \ell \end{bmatrix}$$

$n = N\ell$

$$C_\alpha = A B_\alpha + C_\alpha =$$

$$= \sum_{k=1}^{n} A(:,k) B_\alpha(k,:) + C_\alpha$$

$$C_\alpha = \sum_{k=1}^{n} A(:,k) B_\alpha(k,:) + C_\alpha$$

```
for α = 1 : n
    load B_α and C_α in cache
    for k = 1 : n
        load A(:,k) in cache
        C_α = A(:,k) B_α(k,:) + C_α
    end
    store C_α in main memory
end
```

If $M$ cache size, then we must have:

$$2 \cdot nl + n \leq M$$

$nl$ — $A(:,k)$

$n$ — $B_a, C_a$

$T_1 \equiv$ # of floating point numbers flowing btw cache and RAM

- Every entry of $B$ is loaded into cache once
- Every entry of $C$ is loaded into cache once and stored back in RAM once
- Every entry of $A$ is loaded into cache $N = \frac{n}{l}$ times

$$n^2 + n^2 + n^2 + n^2 \cdot \frac{n}{l} =$$

$$= 3n^2 + \frac{n^3}{l}$$

$$T_1 = 3n^2 + \frac{n^3}{\ell}$$

$$2n\ell + n \leq M$$

We want to maximize $\ell$ with the constraint

$$\ell \simeq \frac{1}{2}\left(\frac{M}{n} - 1\right)$$

$$\Rightarrow \boxed{T_1 \cong 3n^2 + \frac{2n^4}{M-n}}$$

Using $T_1 = 3n^2 + \frac{n^3}{\ell}$ we can see 2 extreme cases

1) $\ell = n$ (large cache, containing the whole matrix)

$$\Rightarrow T \sim n^2$$

2) $\ell = 1$ (small cache, containing only some columns ...)

$$\Rightarrow T \sim n^3$$

$$C_{\alpha\beta} = \sum_{\gamma=1}^{N} A_{\alpha\gamma} B_{\gamma\beta}$$

$\alpha = 1:N \quad \beta = 1:N$

$A = (A_{\alpha\beta}), B = (B_{\alpha\beta}) \in \mathbb{R}^{N\times N}$

```
for α = 1:N
    for β = 1:N
        load Cαβ into cache
        for γ = 1:N
            load Aαγ and Bγβ into cache
            Cαβ = Cαβ + Aαγ Bγβ
        end
        store Cαβ in main memory
    end
end
```

$$T_2 = 2n^2 + \frac{2n^3}{\ell}$$

- each entry in A and B is loaded $N = \frac{n}{\ell}$ times
- each entry of C is loaded once and stored once

$$\Rightarrow T_2' = 2n^2 + 2n^2\left(\frac{n}{\ell}\right) =$$

$$= 2n^2 + 2\frac{n^3}{\ell}$$

Constraint: we want to put a whole block in the

$$3\ell^2 \leq M \quad \Rightarrow \quad \ell \geq \sqrt{\frac{M}{3}}$$

$$T_2' \cong 2n^2 + 2n^3\sqrt{\frac{3}{M}}$$

$$T_1 = 3n^2 + \frac{2n^4}{M-n}$$

$$T_2 = 2n^2 + 2n^3\sqrt{\frac{3}{M}}$$

$$\Rightarrow \frac{T_1}{T_2} = \frac{n^2}{n^2} \; \frac{3 + \frac{2n^2}{M-n}}{2 + 2\sqrt{3}\sqrt{\frac{n^2}{M}}}$$

key quantity is $\left(\frac{n^2}{M}\right)$ = ratio $\frac{\text{matrix size}}{\text{cache size}}$

If $\frac{n^2}{M} >> 1 \Rightarrow$

$$\boxed{\frac{T_1}{T_2} \approx \frac{n}{\sqrt{3M}}}$$

→ The second strategy is superior from the standpoint of data motion to and from the cache.

References:

- GOLUB, VAN LOAN
Matrix computations
John Hopkins Ed.