High Performance Linear Algebra

Lecture 2.3: Sparse Matrices

Ph.D. program in High Performance Scientific Computing

Salvatore Filippone **Pasqua D'Ambra** **Fabio Durastante**

March 17 2026 — 13.00:15.00

Table of Contents

Sparse Matrices

In most applications A is sparse:

Definition

A matrix $A \in \mathbb{R}^{n \times n}$ is sparse if $nnz(A) \ll O(n^2)$

Definition (Wilkinson)

A matrix is sparse when the percentage of its entries that are zero is sufficiently large that it pays off to devise a scheme to avoid their explicit memorization in the computer

When discretizing a differential equation, most coefficients are "structurally" zero because differential operators are local !
Note: there may exist coefficients that are only "numerically" zero.
Nonzero coefficients are typically $nz = k \cdot n$, with k depending on the discretization technique and local topology, *not* on n.

Sparse Matrices: finite differences

Approximate derivatives (Taylor expansion) for a discrete set of points $x = nh,\ n \in \mathbb{N}$:

$$\frac{du}{dx}(x) = \frac{u(x+h) - u(x)}{h} - \frac{h}{2}\frac{d^2u(x)}{dx^2} + O(h^2)$$

Using the same formula with $-h$, summing and applying the mean value theorem we derive

$$\frac{d^2u(x)}{dx^2} = \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} - \frac{h^2}{12}\frac{d^4u(\xi)}{dx^4}.$$

If we now do the same in both x and y directions we get

$$\begin{aligned}\Delta u(x) \approx \frac{1}{h^2} \quad [\quad & u(x_1 + h, x_2) + u(x_1 - h, x_2) \\ & + u(x_1, x_2 + h) + u(x_1, x_2 - h) - 4u(x_1, x_2)],\end{aligned}$$

a 5-point approximation stencil (i.e.: $k = 5$).

Consider an elliptic problem

$$\begin{cases} -\nabla(\mu \nabla u) = f & \text{in } \Omega, \\ u = g & \text{on } \Gamma = \delta\Omega. \end{cases}$$

Under suitable conditions we can use the weak formulation (in V)

$$\text{find } u \in V : \int_\Omega \mu \nabla u \cdot \nabla v \, d\Omega = \int_\Omega f v \, d\Omega \qquad \forall v \in V.$$

The *finite element* method restricts the search for the solution to the finite-dimensional subspace $V_h \subset V$.

Given a basis

$$V_h = \text{span}\{\Phi_1, \Phi_2, \ldots, \Phi_n\}$$

it suffices to consider the problem on the basis functions $\{\Phi_i\}$; thus we have n equations

$$\int_\Omega \mu \nabla u \cdot \nabla \Phi_i \, d\Omega = \int_\Omega f \Phi_i \, d\Omega \qquad i = 1, \ldots n$$

Moreover we can express;

$$u = \sum_{j=1}^{n} \alpha_j \Phi_j$$

Sparse Matrices: finite elements

Summarizing:

$$\text{find } \boldsymbol{\alpha} \in \mathbb{R}^n : \sum_{j=1}^{n} \alpha_j \int_\Omega \mu \nabla \Phi_j \cdot \nabla \Phi_i \, d\Omega = \int_\Omega f \Phi_i \, d\Omega \qquad i = 1, \ldots n$$

which is a linear system $\sum_j a_{ij} x_j = b_i$ with

$$a_{ij} = \int_\Omega \mu \nabla \Phi_j \cdot \nabla \Phi_i \, d\Omega \qquad b_i = \int_\Omega f \Phi_i \, d\Omega \qquad x_j = \alpha_j$$

For each pair (i, j) such that

$$m(\text{supp}(\Phi_i) \cap \text{supp}(\Phi_j)) = 0$$

we *necessarily* have $a_{ij} = 0$ ("structurally" zero coefficients); e.g. for linear finite elements on the real line, a_{ij} can be nonzero only when $j \in \{i - 1, i, i + 1\}$.

Sparse Matrices: finite volumes

Consider a conservation law like

$$\frac{\partial u}{\partial t} + \nabla \cdot \overrightarrow{F} = Q$$

Applying the weak formulation and integrating over a control volume we get

$$\int_\Omega w \frac{\partial u}{\partial t} dx + \int_\Omega w \nabla \cdot \overrightarrow{F}\, dx = \int_\Omega wQ\, dx$$

Integrating by parts we have

$$\int_\Omega w \frac{\partial u}{\partial t} dx - \int_\Omega \nabla w \cdot \overrightarrow{F}\, dx + \int_\Gamma w\, \overrightarrow{F} \cdot \overrightarrow{n}\, ds = \int_\Omega wQ\, dx$$

Sparse Matrices: finite volumes

Now consider a *control volume* K_i and choose a test function that takes the value 1 on the volume, 0 elsewhere; the second term in the previous equation disappears, and we have

$$\int_{K_i} \frac{\partial u}{\partial t} dx - \int_{\Gamma} \overrightarrow{F} \cdot \overrightarrow{n}\, ds = \int_{K_i} Q\, dx$$

If we also assume that $\overrightarrow{F} = \overrightarrow{\lambda}$ then $\nabla \overrightarrow{F} = \overrightarrow{\lambda} \cdot \nabla u$ and approximate

$$\int_{K_i} \frac{\partial u}{\partial t} dx \approx \frac{\partial u}{\partial t} |K_i|, \qquad \int_{K_i} Q\, dx \approx q_i |K_i|$$

we end up with

$$\frac{\partial u}{\partial t} |K_i| + \overrightarrow{\lambda} \cdot \int_{\Gamma_i} u \overrightarrow{n}\, dx = q_i |K_i|$$

Sparse Matrices: finite volumes

If we now assume that

- we are dealing with "simple" volumes K_i and their contours Γ_i;
- the "length" s_j of an edge is given by $\overrightarrow{s}_j = s_j \cdot \overrightarrow{n}_j$;
- The value of u on the contour can be approximated by some "average" between the values in the adjacent volumes,

we end up with

$$\frac{\partial u}{\partial t}|K_i| + \frac{1}{2}\sum_j (u_i + u_j)\overrightarrow{\lambda} \cdot \overrightarrow{s}_j = q_i|K_i|.$$

Given that $\sum \overrightarrow{s} = 0 \Rightarrow u_i \overrightarrow{\lambda} \sum \overrightarrow{s}_j = 0$, we end up with

$$\frac{\partial u}{\partial t}|K_i| + \frac{1}{2}\sum_j u_j \overrightarrow{\lambda} \cdot \overrightarrow{s}_j = q_i|K_i|,$$

where the summation in j extends over the control volumes adjacent to volume i.

Sparse Matrices: solution methods

First idea: let's use factorizations:

"Direct methods for sparse matrices".

Sparse Matrices: solution methods

First idea: let's use factorizations:
"Direct methods for sparse matrices".

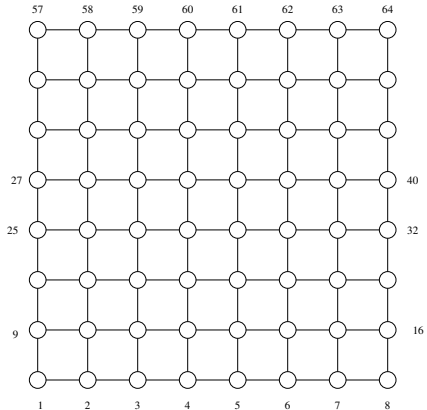
We have a major problem here:

The LU (or QR) factors of a sparse matrix ARE NO LONGER sparse, but dense.

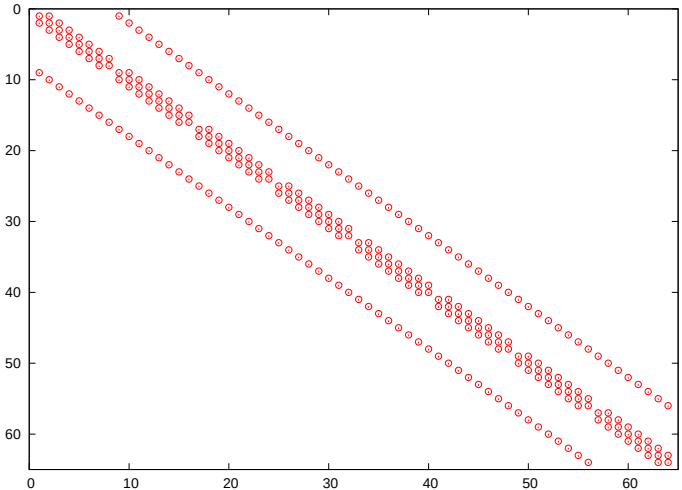
Equally, the inverse A^{-1} is dense (in general)
Therefore the storage requirements grow explosively. Ex: Poisson Equation

$$x_{0,1} + x_{1,0} + x_{0,-1} + x_{-1,0} - 4x_{0,0} = b_{0,0}$$

a square domain with 8 grid points on each side


57
58
59
60
61
62
63
64
27
40
25
32
9
16
1
2
3
4
5
6
7
8

5-point stencil, 8×8 mesh



Fundamental issue: the L and U factors are much denser and the inverse is completely full.

Why?

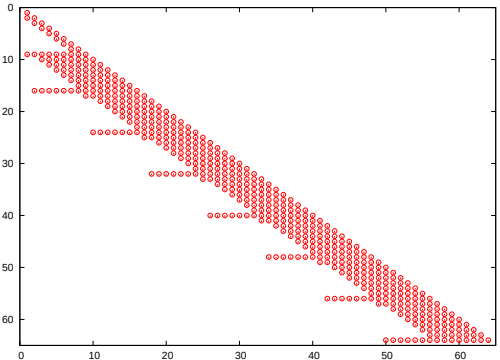


Table of Contents

2 Graphs and Structure of Sparse Matrices

Graph theory: a graph is a set $G = \{V, E\}$ where

$$\begin{aligned} V &= \{v_1, \ldots, v_n\} \\ E &\subseteq V \times V \end{aligned}$$

Let's define the structure of A as a graph [3]

$$\text{struct}(A) = G(A)$$

defined by the vertex and edge sets

$$\begin{aligned} V &= \{i : i = 1, \ldots, n\} \\ E &= \{(i,j) : i \neq j \text{ and } A_{ij} \neq 0\}. \end{aligned}$$

Thus there exist a mapping between matrices and graphs:

- To each row i there corresponds a vertex v_i;
- To each coefficient $a_{ij} \neq 0$ there corresponds an edge (v_i, v_j);

From a graph to a matrix: isomorphism, up to a renumbering of the nodes.

The structure of a vector x is the set

$$\text{struct}(x) = \{i : x_i \neq 0\}.$$

The notation

$$G_1 \subseteq G_2$$

means that both vertex and edge sets of G_1 are subsets of those of G_2.

$i \xrightarrow{A} j$	means $\exists (i,j) \in E$
$i \overset{A}{\Longrightarrow} j$	means there is a path from i to j in $G(A)$

A subset x of the vertices of G is *closed* if no edge of G links a vertex not in x to a vertex in x. The *closure* of x is the smallest closed set containing x

$$\text{closure}(x) = \bigcap \{y : x \subseteq y \text{ and } y \text{ is closed}\}.$$

The *transitive closure* of A is the graph $G^*(A)$ with edges corresponding to paths in $G(A)$, i.e.

$$i \xrightarrow{G^*A} j \text{ if and only if } i \neq j \text{ and } i \overset{A}{\Longrightarrow} j.$$

- A graph G is *strongly connected* if its closure is a complete directed graph.
- A matrix is *irreducible* if $\mathrm{struct}(A)$ is strongly connected.
- A finite set of complex numbers $\{x_1, \ldots, x_n\}$ is called *algebraically independent* if the point $(x_1, \ldots, x_n)$ is not a zero of a nonzero n-variable polynomial with integer coefficients.

Theorem

Let the structures of A and b be given. Then

(i) *Irrespective of the values of the nonzeros in A and b, if A is nonsingular then*

$$\text{struct}(A^{-1}b) \subseteq \text{closure}(b).$$

(ii) *There exist nonzero values for which the above inclusion is actually an equality.*

See [3]

Proof, part (i).

Consider nonzero values such that A is nonsingular; also apply a renumbering such that $\text{closure}(b) = \{1, 2, \ldots, k\}$ for some $k \leq n$. We can then rewrite $Ax = b$ as

$$\begin{pmatrix} B & D \\ C & E \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} = \begin{pmatrix} d \\ 0 \end{pmatrix},$$

with B a $k \times k$ matrix. By definition of closure, there can be no edge (i, j) with $i \notin \text{closure}(b)$ and $j \in \text{closure}(b)$; this implies that $C = 0$. Since A is nonsingular, then so is E, and we also have $z = 0$; therefore $\mathit{struct}(x) \subseteq \{1, 2, \ldots, k\} = \text{closure}(b)$. □

Proof part (ii).

Choose a set of algebraically independent values for the nonzeros of A; this implies that A is nonsingular. Choose also $b_i = 1$ if $i \in \text{struct}(b)$. Let $x = A^{-1}b$ and renumber A so that $\text{struct}(x) = \{1, 2, \ldots, k\}$ for some $k \leq n$. We can then rewrite $Ax = b$ as

$$\begin{pmatrix} B & D \\ C & E \end{pmatrix} \begin{pmatrix} y \\ 0 \end{pmatrix} = \begin{pmatrix} d \\ e \end{pmatrix},$$

with B a $k \times k$ matrix, and all entries in y are nonzero. Now, considering C we have

$$\sum_{1 \leq j \leq k} c_{ij} y_j = e_i.$$

Matrix B is nonsingular because of the choice of the nonzeros; then $By = d$ implies by Cramer's rule that $y_i = \det(B|_j^d)/\det(B)$. □

Proof part (ii) ctd.

Then, this implies that

$$\sum_{1\le j\le k} c_{ij}\det(B|_j^d) - e_i\det(B) = 0.$$

This is a polynomial with rational coefficients in the entries of A, so it can only be the zero polynomial; but $y_j \neq 0$ thus $\det(B|_j^d)$ is not zero, and we must have $c_{ij} = 0$. Therefore x as partitioned above is closed; moreover $\det(B) \neq 0$ hence $e_i = 0$; in summary

$$b = \begin{pmatrix} d \\ 0 \end{pmatrix}, \qquad \mathrm{struct}(b) \subseteq \mathrm{struct}(x) = \mathrm{closure}(x) \Longrightarrow \mathrm{closure}(b) \subseteq \mathrm{closure}(x).$$

Together with the inclusion in part (i), this proves

$$\mathrm{closure}(b) = \mathrm{struct}(x).$$

□

Considering that column j of $G^*(A)$ is $\mathrm{closure}(e^{(j)})$, we immediately derive

Corollary

Let $G(A)$ be given.

(i) *Irrespective of the values of the nonzeros in A, if A is nonsingular then $G(A^{-1}) \subseteq G^*(A)$.*

(ii) *There exist nonzero values for which the above inclusion is actually an equality.*

This implies that the inverse of an irreducible matrix A is full, except possibly for numerical cancellations.

From the definitions given previously, sparse matrices have a memory footprint $O(nnz)$ and since $nnz \leq k \cdot n$ for some k independent of n, storage is *linear*.
But,

- The factorization is more expensive than $O(nnz)$;
- The amount of memory is (much) higher than $O(nnz)$;
- The algorithm is a lot more complicated.

The first two points imply that if we exploit linear storage to increase the system size, we will (likely) run out of memory when we apply a direct method.
Nevertheless, direct methods for sparse matrices are still useful, on their own as well as building blocks in mixed dense/iterative solvers. [1]

[1] This and the next slide courtesy of Alfredo Buttari, CNRS-IRIT

Structure of a direct solver:

- Preprocessing, aka symbolic factorization, aka computing the elimination tree; in particular, search for an ordering that minimizes fill-in;
- Processing, aka numeric factorization;
- Solution (sparse triangular system solve);

These algorithms involve very complicated data structures, based on graphs and trees; versions of them apply to both LU/Cholesky as well as QR factorizations.
In particular, the MUMPS package is capable of computing a Block Low Rank approximation which can be used to implement a preconditioner (or smoother in a MultiGrid environment).
For further details see [1, 2].

Table of Contents

3 Iterative Methods for Sparse Matrices

Using iterative solvers: why?
We have already seen that iterative solvers are somewhat delicate beasts, they require knowledge of some non-trivial matrix property, are difficult to tune, and are not guaranteed to give a reasonable solution.
Why bother then?

Using iterative solvers: why?

The sparse matrix A is only ever used to perform matrix-vector products

$$y = Ax,$$

hence *its structure is not altered*. Sparse Matrix-Vector product: only perform the arithmetic for the nonzeros. Hence the cost of an SpMV operation is *not* $2n^2$ but rather:

$$2 \cdot nnz = 2 \cdot kn = O(n).$$

Therefore the total cost of an iterative method that converges in j iterations will be

$$O(jkn) \ll O(n^3),$$

and it may well be that

$$jkn < n^2.$$

Table of Contents

4 The roofline model

Modern Memory Hierarchy

4 The roofline model

- Computer architectures organized around a **memory hierarchy**
- Designed to balance speed, capacity, and cost

Memory Hierarchy Levels

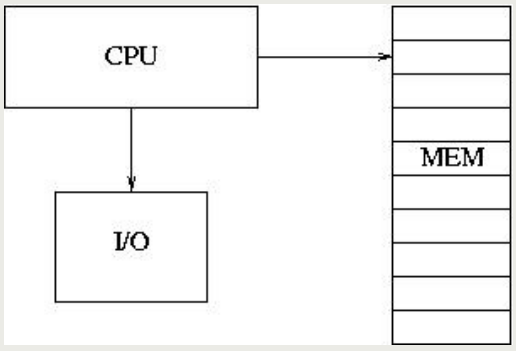
1. Registers and cache (L1, L2, L3) — extremely fast
2. Main memory (RAM) — moderate speed
3. Secondary storage (SSD/HDD) — slower
4. Tertiary storage — archival

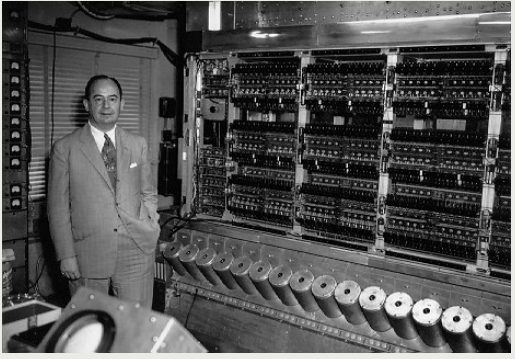
Key parameter: Memory bandwidth — rate of data transfer between memory and processor

1. Evolution of processor vs. memory technology: differential speed;
2. Paging and Virtual Memory;
3. Cache memories;

Essential to proper design of software.

Architettura di Von Neumann (Neumann Janós Lajos)





Unfortunately, technology has driven the frequency of CPUs and memories apart; CPUs and RAM were synchronized at the launch of the original IBM PC in 1981.

Facts of (memory) life:

Memory can be

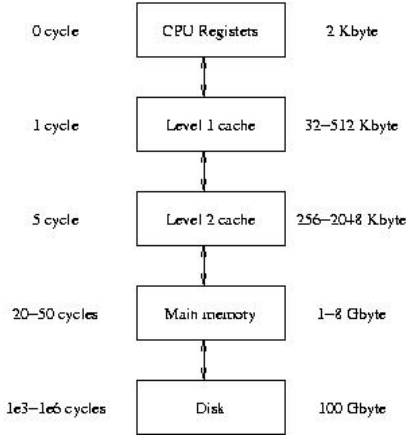
- Large;
- Fast;
- Cheap;

but *you can choose only two*;

Computer architects and programmers want all three anyway!
Hence the concept of *Memory Hierarchy*

Memory Hierarchy

4 The roofline model



Why does it work?

1. Memory access is latency bound, but data are transferred in blocks;
2. Programming *locality* principles:
   - Temporal locality: if a program accesses a memory location now it will likely access it again in the near future;
   - Spatial locality: if a program accesses a memory location now, it will likely access nearby locations as well.

Assume we have two levels:

Cache: Uniform cost: 1 cycle;

Main: 20 cycles for start of blocks of 32 items, with items after the first being made available at full speed.

Average cost of sequential access of 32 items:

$$T_{avg} = \frac{20 \times 1 + 1 \times 31}{32} = 1.59$$

The Problem

Processor speeds have grown much faster than memory bandwidth improvements

- **Memory wall:** memory latency and bandwidth become the primary bottleneck
- Need tools to understand and visualize this limitation
- Enter: the *Roofline Model* [4]

The Roofline Model: Concept

Definition

A visual performance model relating computational throughput to memory bandwidth

Key hardware characteristics:

- Peak floating-point performance: Perf (FLOP/s)
- Peak memory bandwidth: BW (Bytes/s)

Key application characteristic:

- Operational Intensity (OI): FLOP/Byte
- Ratio of floating-point ops to bytes accessed from memory

Roofline Model: The Relationship

Fundamental equation

$$\text{Perf} = \frac{\text{FLOP}}{s} = \frac{\text{FLOP}}{\text{Byte}} \cdot \frac{\text{Byte}}{s} = \text{OI} \cdot \text{BW}$$

- Performance depends linearly on both OI and BW
- Plotted as log-log graph: performance vs operational intensity
- Creates a characteristic “roofline” shape

Roofline Model: Visual Representation

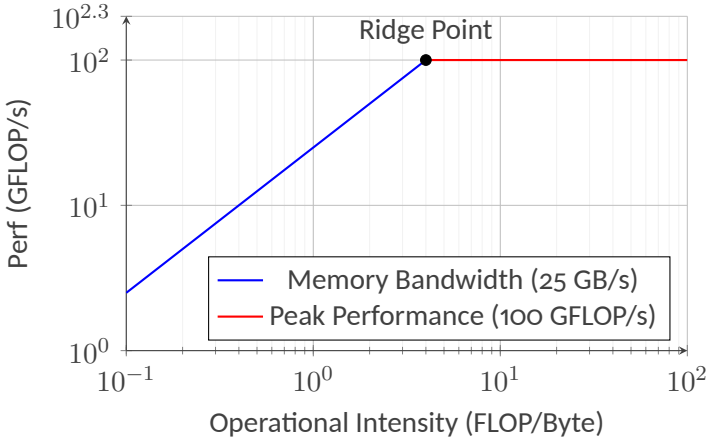


Figure: Roofline model: 100 GFLOP/s peak, 25 GB/s bandwidth

Understanding the Roofline Plot

4 The roofline model

Two regions:

1. Memory-bound (left)
   - Linear increase with OI
   - Limited by bandwidth
2. Compute-bound (right)
   - Horizontal line
   - Limited by peak FLOP/s

Ridge point:

- Intersection of two regions
- Minimum OI to reach peak performance
- In example: 4 FLOP/Byte

Using the Roofline Model

Applications:

- Analyze kernel performance on given architecture
- Identify performance bottlenecks
- Guide optimization efforts

Optimization strategy

Compare kernel's OI to ridge point:

- Below ridge → memory-bound → improve data locality
- Above ridge → compute-bound → optimize computations

Sparse matrices

4 The roofline model

Sparse matrix-vector product performs poorly ...and that's a fact!

- Low ratio between floating point operations and memory accesses;
- High consumption of memory bandwidth
- Indirect addressing;
- Low spatial or temporal locality:
  - The elements of the matrix are accessed sequentially and are not reused;
  - The elements of the destination vector are accessed sequentially and are reused for each element in the corresponding row of the matrix;
  - The elements of the source vector are not accessed sequentially and are not necessarily reused;

Data storage formats are essential! And many of them were invented over the years (more than you care to know).

Table of Contents

A Detour: Multi-Dimensional Arrays

What is a multi-dimensional array?

An array is a collection of objects all of the same type, identified by *a set of one or more* integer indices.

Trouble is:

You can view a 2D array as a set of 1D array (which ones?)

but

That's defeating its purpose (to some extent)

Compiler view

An array is a collection of objects such that you can identify the address in memory of any entry given only the set of indices and the size(s)

A Detour: Multi-Dimensional Arrays

A 1-D example

Consider the following code:

```
double v[10];
i=4;
x=v[i];
```

To access `v[i]` the compiler needs to know the following:

- The starting address in memory of the vector `v`;
- The size of each element of the vector (for a `double`: 8 bytes);
- The starting index `xb` of the first element (in C: 0);

Then it can compute

```
addr = start(v) + size * ( i - xb)
     = start(v)+ 8*(4-0)
```

A Detour: Multi-Dimensional Arrays

A 2-D example

```
double a[10][20];
i=4; j=3;
x=a[i][j];
```

How is the 2D array laid out in memory? In C, *storage by rows*.

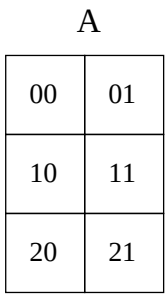
To access `a[i][j]` the compiler needs to know the following:

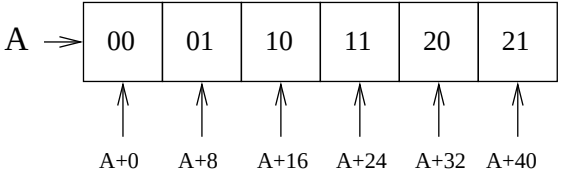
- The starting address in memory of the array `a`;
- The size of each element of the vector (for a `double`: 8 bytes);
- The starting index `xb` in each dimension (in C: 0);
- How many entries `NC` there are in each row (in this case: 20)

Then it can compute

```
addr = start(a) + size * ( (i - xb)*NC + (j-xb) )
     = start(a) + 8*( (4-0)*20+(3-0))
```

A Detour: Multi-Dimensional Arrays





A Detour: Multi-Dimensional Arrays

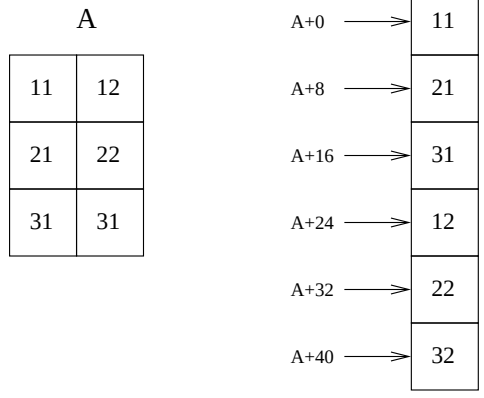
Note that there is another way to store arrays: *by columns*. And, what if the first index is 1 instead of 0?
This is what Matlab and Fortran (and Julia) do, in which case the formula requires the number of *rows* instead of columns, and you have

```
addr = start(a) + size * ( (j - xb)*NR + (i -xb))
     = start(a) + 8*( (3-1)*10+(4-1))
```

- Matlab *requires* the first address to be 1;
- Fortran allows the programmer to choose an arbitrary starting index `xb`, with 1 being the default.

However, it is clear that the formulae are completely equivalent to those we have seen for the C language.

A Detour: Multi-Dimensional Arrays



A Detour: Multi-Dimensional Arrays

What happens if the number of rows/columns is not known at compile time?

For instance, you might have a function accepting the variable `a`: you might want to pass to the function the number of rows and columns, to interpret the variable properly. If you can do this, then your language supports 2D arrays in the most general sense

A Detour: Multi-Dimensional Arrays

What happens if the number of rows/columns is not known at compile time?

For instance, you might have a function accepting the variable `a`: you might want to pass to the function the number of rows and columns, to interpret the variable properly. If you can do this, then your language supports 2D arrays in the most general sense

Unfortunately, this is not what happens in C.

The default action in the C language is to interpret an expression like

```
x = mat[i][j];
```

in two completely different ways depending on whether the size of `mat` is known at compile time or not.

A Detour: Multi-Dimensional Arrays

When the size is only known at runtime...

...the compiler will interpret

```
x = mat[i][j];
```

as meaning that

- `mat` is a (pointer to an) array of pointers; so by `mat[i]` you are selecting the pointer at entry `[i]`;
- Each pointer identifies a separate row array; entries in each row are addressed by `[j]`;
- Each row is allocated independently and there is no obvious relationship between the locations in memory of two different rows.

A Detour: Multi-Dimensional Arrays

When the size is only known at runtime...

...This is different from Matlab and Fortran, where the size is *embedded* in the array object

```
real :: A(:,:)
x = A(i,j)
```

meaning that

- `A` contains details about the size in each dimension;
- The compiler always implements the formula we have seen previously at runtime;

A Detour: Multi-Dimensional Arrays

Do programming languages support multidimensional arrays?

A Detour: Multi-Dimensional Arrays

Do programming languages support multidimensional arrays?

Matlab	Yes (by columns)
Fortran	Yes (by columns)
Julia	Yes (by columns)

A Detour: Multi-Dimensional Arrays

Do programming languages support multidimensional arrays?

Matlab	Yes (by columns)
Fortran	Yes (by columns)
Julia	Yes (by columns)
Java	No (by rows)

A Detour: Multi-Dimensional Arrays

Do programming languages support multidimensional arrays?

Matlab	Yes (by columns)
Fortran	Yes (by columns)
Julia	Yes (by columns)
Java	No (by rows)
C	It's complicated (and it's by rows)
C++	It's complicated (and it's by rows)

A Detour: Multi-Dimensional Arrays

Do programming languages support multidimensional arrays?

Matlab	Yes (by columns)
Fortran	Yes (by columns)
Julia	Yes (by columns)
Java	No (by rows)
C	It's complicated (and it's by rows)
C++	It's complicated (and it's by rows)

In C/C++/Java you can have

```
void compute(double a[][])
```

but then `a` is interpreted as *A 1-D array of pointers (to 1-D arrays)* and those can point to anything (even if the rows all have the same length), i.e. to know where `mat[2][3]` is in memory, you need to look at the *pointer* at location of `mat[2]`, then access its element at index 3. The start address of `mat` and its size(s) are not enough.

Possible solutions to have a 2-D data layout in memory

- If you are using Matlab, Fortran or Julia, you're all set;

Possible solutions to have a 2-D data layout in memory

- If you are using Matlab, Fortran or Julia, you're all set;
- If you are using Java, you are out of luck, there's no way of forcing what you want (but see next item);

Possible solutions to have a 2-D data layout in memory

- If you are using Matlab, Fortran or Julia, you're all set;
- If you are using Java, you are out of luck, there's no way of forcing what you want (but see next item);
- If you are using C or C++, you can declare the array *as if* it was 1-D, then implement the index calculation "by hand"

```
int my2Dindex(i,j,nr,nc) { return(i*nc+j);}
void compute(int nr, int nc, double mat[])
x = mat[my2Dindex(i,j,nr,nc)]
```

  If you put the computation in a function, you *really* want the compiler to do inlining. Especially if `my2Dindex` is declared as a class member function (or method) (See the STL).

But wait, there's more

Possible solutions to have a 2-D data layout in memory

If you are using C, *and if* you can force the compiler to use the *C99* standard semantics, *then* you can write something like

```
void compute(int nr, int nc, double mat[nr][nc])
```

and the compiler will interpret the matrix as a 2D array correctly.
Note that you *must* put `nr,nc` *before* `mat` in the function argument list.

Possible solutions to have a 2-D data layout in memory

If you are using C, *and if* you can force the compiler to use the *C99* standard semantics, *then* you can write something like

```
void compute(int nr, int nc, double mat[nr][nc])
```

and the compiler will interpret the matrix as a 2D array correctly.
Note that you *must* put `nr,nc` *before* `mat` in the function argument list.
And obviously this is incompatible with C++.

If you want to have fun

use a search engine and look for

How do I declare a 2d array in C++ using new?

Kernels we need?

To implement an iterative method we need:

- Matrix-vector product;
- Dot products;
- Scaled sums of vectors;
- Vector norms;
- Matrix norms;
- Solution of sparse triangular systems.

The sparse triangular system is particularly problematic:

1. On the one hand, triangular systems can be parallelized efficiently if they are dense;
2. On the other hand, in an iterative solver context, the matrices should be sparse, but this means the solvers tend to be a lot less efficient.

Sparse linear solvers typically exploit renumbering by wavefronts and coloring, but the available speedup is not very exciting.

References

6 Bibliography

[1] T. A. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006. DOI: 10.1137/1.9780898718881. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898718881. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898718881.

[2] I. Duff, A. Erisman, and J. Reid. *Direct Methods for Sparse Matrices*. Oxford, UK: Oxford University Press, 2017. ISBN: 9780198508380.

[3] J. R. Gilbert. "Predicting Structure in Sparse Matrix Computations". In: *SIAM J. Matrix Anal. Appl.* 15.1 (1994), pp. 62–79.

[4] S. Williams, A. Waterman, and D. Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: https://doi.org/10.1145/1498765.1498785.