Understanding how to implement sparse matrix operations efficiently

Simon Dobson

simon.dobson@st-andrews.ac.uk
https://simondobson.org
https://mastodon.scot/@simoninireland

Overview

A talk in three parts

- Matrices
- Why sparse matrices are different, and why this causes problems
- Our plan to improve the situation

Wherever there is a *pattern* (if you can find it)...

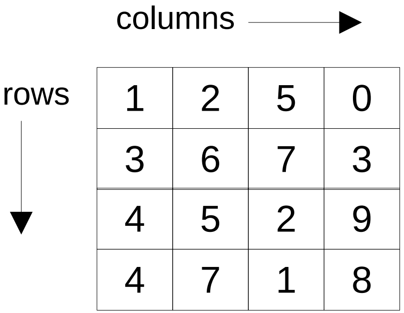
...there is (potentially) an *optimisation* (if you can find it)

...and this is hopefully how we find it

Don’t expect to see any results

Matrices

Not necessarily square
(but often are)



columns →

rows ↓

1	2	5	0
3	6	7	3
4	5	2	9
4	7	1	8

A block of numbers that are treated as a whole, with their own algebra of operations

Matrices

1	2	5	0
3	6	7	3
4	5	2	9
4	7	1	8

A block of numbers that are treated as a whole, with their own algebra of operations

$x + 2y + 5z$
$3x + 6y + 7z + 3p$
$4x + 5y + 2z + 9p$
$4x + 7y + z + 8p$

The operations mirror the way that linear equations are handled, with the numbers in the matrix corresponding to coefficients in the corresponding equations

Matrix multiplication

$$\begin{array}{|c|c|c|c|}\hline 1 & 2 & 5 & 0 \\ \hline 3 & 6 & 7 & 3 \\ \hline 4 & 5 & 2 & 9 \\ \hline 4 & 7 & 1 & 8 \\ \hline \end{array} \quad \text{x} \quad \begin{array}{|c|c|c|c|}\hline 5 & 2 & 5 & 7 \\ \hline 5 & 8 & 1 & 2 \\ \hline 8 & 6 & 3 & 1 \\ \hline 8 & 3 & 5 & 9 \\ \hline \end{array}$$

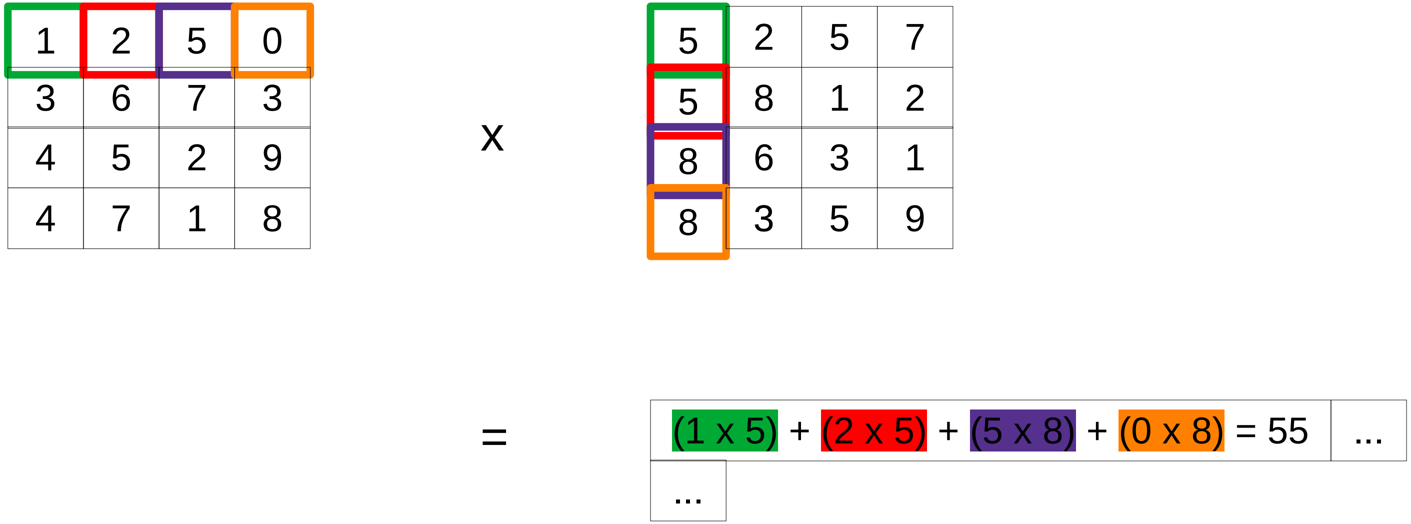
Matrix multiplication

$$\begin{array}{|c|c|c|c|}\hline \boxed{1} & 2 & 5 & 0 \\ \hline 3 & 6 & 7 & 3 \\ \hline 4 & 5 & 2 & 9 \\ \hline 4 & 7 & 1 & 8 \\ \hline \end{array} \quad \text{x} \quad \begin{array}{|c|c|c|c|}\hline \boxed{5} & 2 & 5 & 7 \\ \hline 5 & 8 & 1 & 2 \\ \hline 8 & 6 & 3 & 1 \\ \hline 8 & 3 & 5 & 9 \\ \hline \end{array}$$

Matrix multiplication

$$\begin{array}{|c|c|c|c|}\hline 1 & \boxed{2} & 5 & 0 \\ \hline 3 & 6 & 7 & 3 \\ \hline 4 & 5 & 2 & 9 \\ \hline 4 & 7 & 1 & 8 \\ \hline \end{array} \quad \text{x} \quad \begin{array}{|c|c|c|c|}\hline 5 & 2 & 5 & 7 \\ \hline \boxed{5} & 8 & 1 & 2 \\ \hline 8 & 6 & 3 & 1 \\ \hline 8 & 3 & 5 & 9 \\ \hline \end{array}$$

Matrix multiplication



$$\begin{array}{|c|c|c|c|}\hline 1 & 2 & 5 & 0 \\ \hline 3 & 6 & 7 & 3 \\ \hline 4 & 5 & 2 & 9 \\ \hline 4 & 7 & 1 & 8 \\ \hline \end{array} \quad x \quad \begin{array}{|c|c|c|c|}\hline 5 & 2 & 5 & 7 \\ \hline 5 & 8 & 1 & 2 \\ \hline 8 & 6 & 3 & 1 \\ \hline 8 & 3 & 5 & 9 \\ \hline \end{array}$$

$$= \quad \begin{array}{|c|c|}\hline (1 \text{ x } 5) + (2 \text{ x } 5) + (5 \text{ x } 8) + (0 \text{ x } 8) = 55 & \ldots \\ \hline \ldots & \\ \hline \end{array}$$

Each element in the result matrix is the sum of four multiplications of elements in the argument matrices

Patterns

How does this number interact with the numbers in the other matrix?

$$\begin{array}{|c|c|c|c|}\hline \boxed{1} & 2 & 5 & 0 \\ \hline 3 & 6 & 7 & 3 \\ \hline 4 & 5 & 2 & 9 \\ \hline 4 & 7 & 1 & 8 \\ \hline \end{array} \quad \text{x} \quad \begin{array}{|c|c|c|c|}\hline 5 & 2 & 5 & 7 \\ \hline 5 & 8 & 1 & 2 \\ \hline 8 & 6 & 3 & 1 \\ \hline 8 & 3 & 5 & 9 \\ \hline \end{array}$$

Patterns

1	2	5	0
3	6	7	3
4	5	2	9
4	7	1	8

x

5	2	5	7
5	8	1	2
8	6	3	1
8	3	5	9

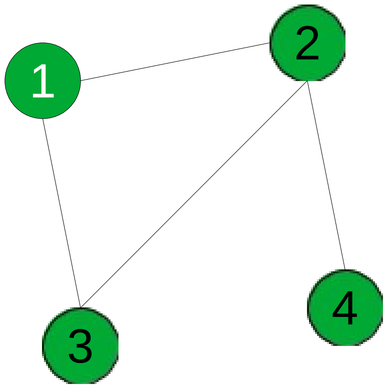
The numbers in *column* 1 are multiplied by all the numbers in *row* 1

=

(1 x 5) + ...	(1 x 2) + ...	...
...		

This pattern is independent of the numbers themselves: it's purely structural

Other sources of matrices

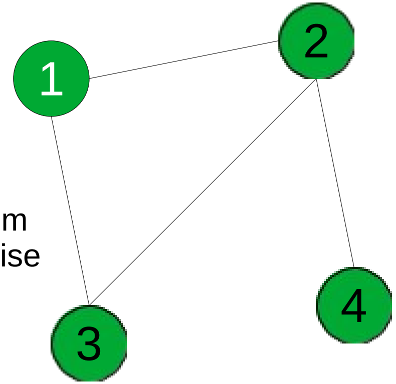


Other sources of matrices

Row/column positions correspond to node labels

$$\begin{array}{c|c|c|c|c|}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 0 \\
\hline
2 & 1 & 0 & 1 & 1 \\
\hline
3 & 1 & 1 & 0 & 0 \\
\hline
4 & 0 & 1 & 0 & 0 \\
\hline
\end{array}$$

Cell is 1 if there is an edge from node *i* to node *j*, and 0 otherwise

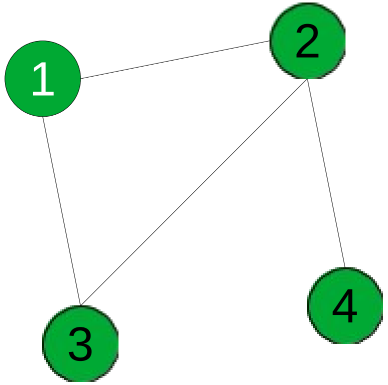


The *adjacency matrix A* corresponding to the graph

Operators

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 1 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 \end{array}$$

The adjacency matrix captures the *static* features of the graph – its structure

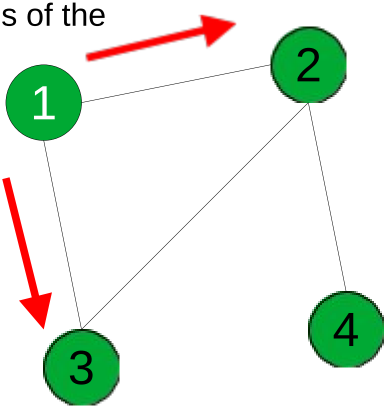


Define an *operator* that maps values at nodes to new values based on the edge structure – the *dynamics* of processes over the graph

Operators

How would information (or something else) “flow” across the edges of the graph?

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 1 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 \end{array}$$

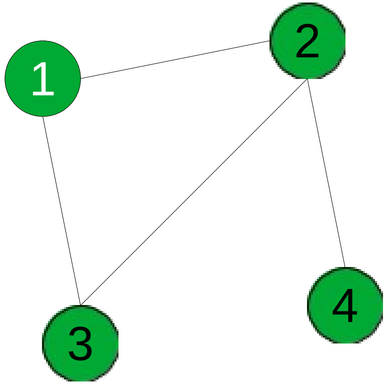


The adjacency matrix captures the *static* features of the graph – its structure

Define an *operator* that maps values at nodes to new values based on the edge structure – the *dynamics* of processes over the graph

Operators

	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	0
4	0	1	0	0



L = D - A

	1	2	3	4
1	2	-1	-1	0
2	-1	3	-1	-1
3	-1	-1	2	0
4	0	-1	0	-1

The *Laplacian matrix L* corresponding to the graph

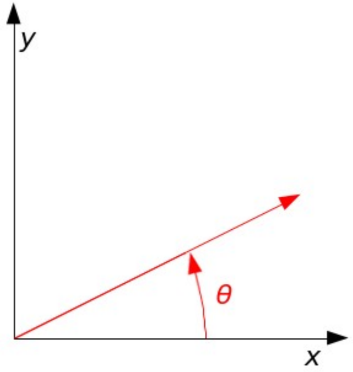
The degree of nodes on the diagonal; -1 where nodes are adjacent; 0 otherwise

In linear algebra “operator” typically (although not necessarily) means “a special matrix derived somehow that we’ll then multiply by”

Operators

$$\begin{array}{|c|c|}\hline \cos\theta & -\sin\theta \\ \hline \sin\theta & \cos\theta \\ \hline \end{array} \quad \mathrm{x} \quad \begin{array}{|c|}\hline x \\ \hline y \\ \hline \end{array}$$

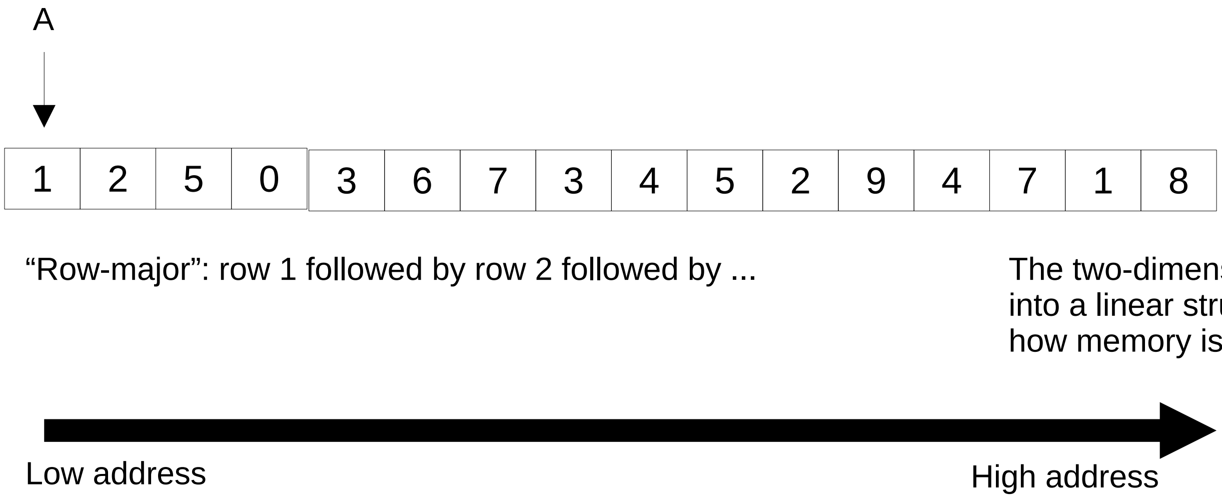
A *rotation* operator, a constant matrix that takes any point (expressed as a column vector) to that point rotated around the origin by θ



Wikipedia

Matrices are closely related to geometry, and any operator has a corresponding (not always obvious, or interesting) geometric analogue

Representing matrices

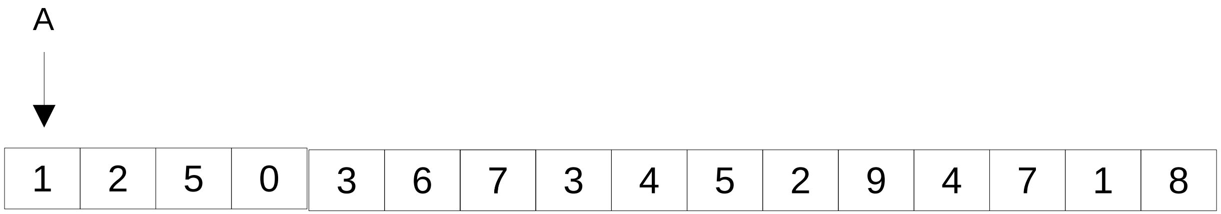


1	2	5	0	3	6	7	3	4	5	2	9	4	7	1	8

“Row-major”: row 1 followed by row 2 followed by ...

The two-dimensional structure is mapped into a linear structure, corresponding with how memory is represented

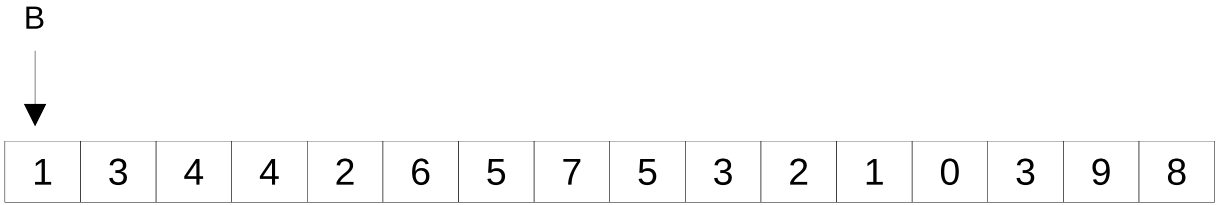
Representing matrices



A

1	2	5	0	3	6	7	3	4	5	2	9	4	7	1	8

“Row-major”: row 1 followed by row 2 followed by ...

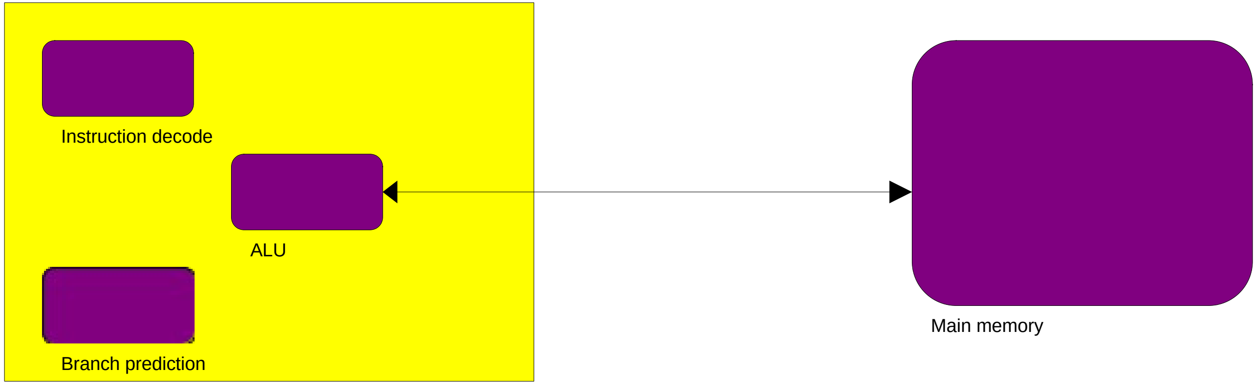


B

1	3	4	4	2	6	5	7	5	3	2	1	0	3	9	8

“Column-major”: column 1 followed by column 2 followed by ...

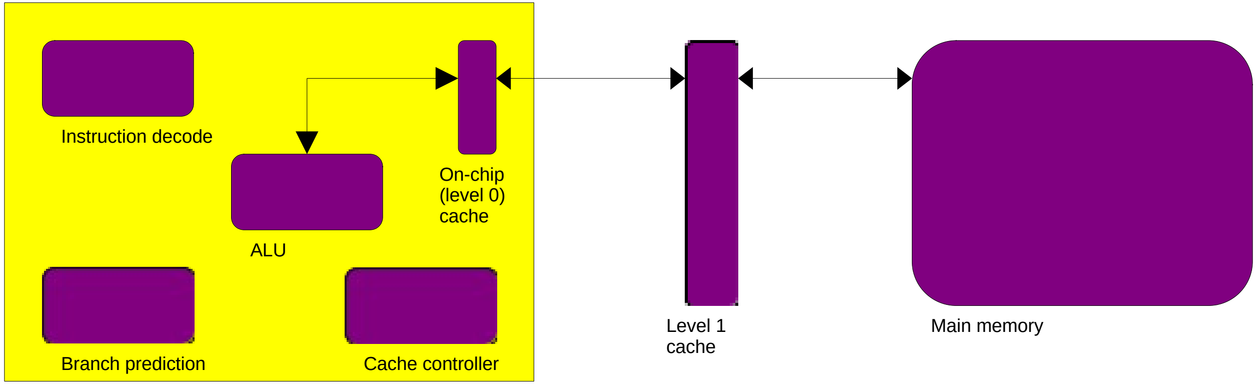
Why this matters



The “von Neumann” model of a processor core: a single CPU interacting with a single block of memory

(John von Neumann would probably be horrified to have his name associated with this, but never mind...)

Why this matters



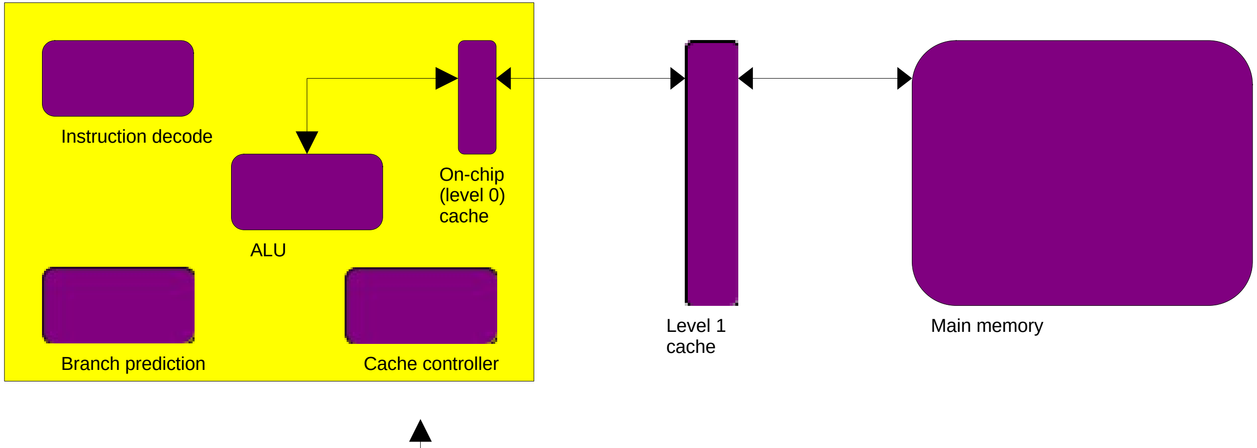
Getting performance means having data in on-chip cache ready to be accessed by the ALU

May be several levels of cache, with their own logic and/or API to move data around programmatically between the core and the main memory

Fast (but not much of it)

Slow (but lots of it)

Why this matters

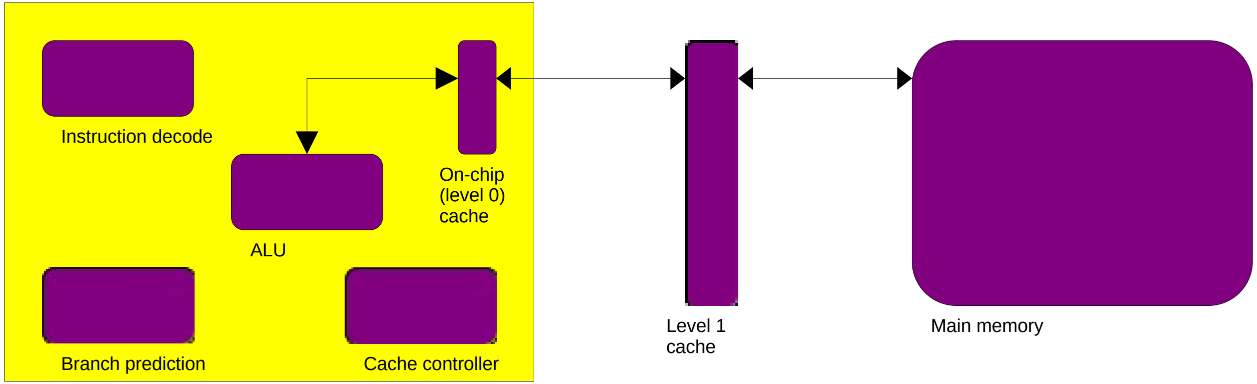


1 2 5 0 x 5 5 8 8

To do multiplication efficiently, we need to move the rows and columns into cache so we can multiply and sum them

...but we know the pattern, so the compiler can statically schedule this activity

Why this matters



1	2	5	0	x	5	5	8	8

To do multiplication efficiently, we need to move the rows and columns into cache so we can multiply and sum them

...but we know the pattern, so the compiler can statically schedule this activity

Wherever there is a *pattern* (if you can find it)...

...there is (potentially) an *optimisation* (if you can find it)

Sparsity

1	2	5	0
3	6	7	3
4	5	2	9
4	7	1	8

Our first matrices so far were *dense*: few if any of the values are zero

Sparsity

1	2	5	0
3	6	7	3
4	5	2	9
4	7	1	8

0	1	1	0
1	0	1	1
1	1	0	0
0	1	0	0

The adjacency matrix, however, is *sparse*: most of its elements are zero

Sparse multiplication

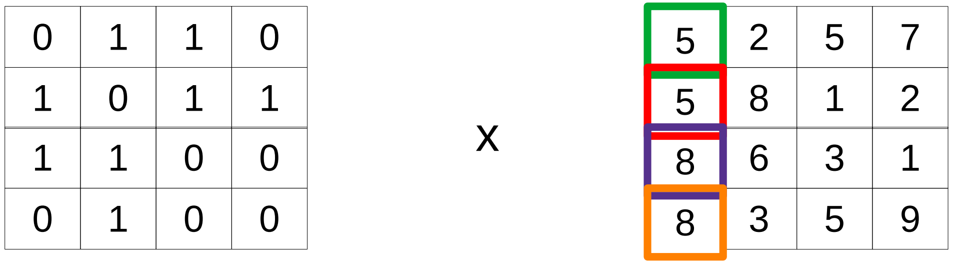
0	1	1	0
1	0	1	1
1	1	0	0
0	1	0	0

Lots of zero terms in one (or both) argument matrices

x

5	2	5	7
5	8	1	2
8	6	3	1
8	3	5	9

Sparse multiplication



0	1	1	0
1	0	1	1
1	1	0	0
0	1	0	0

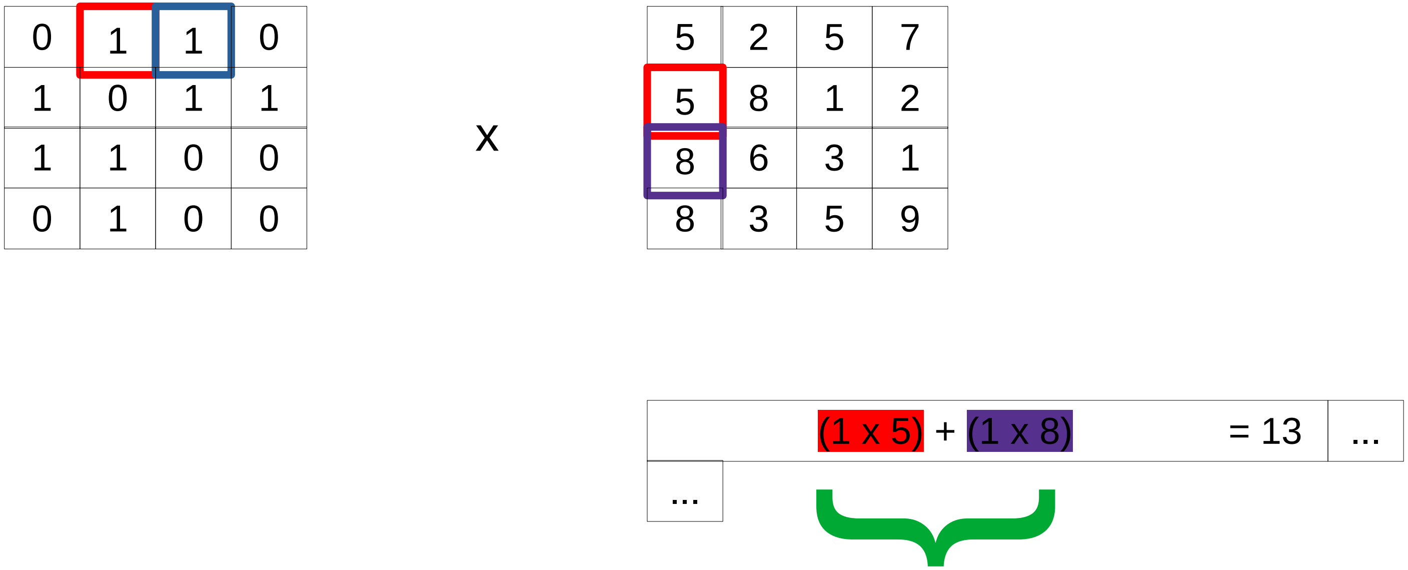
x

5	2	5	7
5	8	1	2
8	6	3	1
8	3	5	9

Several result terms are *necessarily* zero, regardless of the other matrix

(0 x 5) + (1 x 5) + (1 x 8) + (0 x 8) = 13	...
...	

Sparse multiplication



0	1	1	0
1	0	1	1
1	1	0	0
0	1	0	0

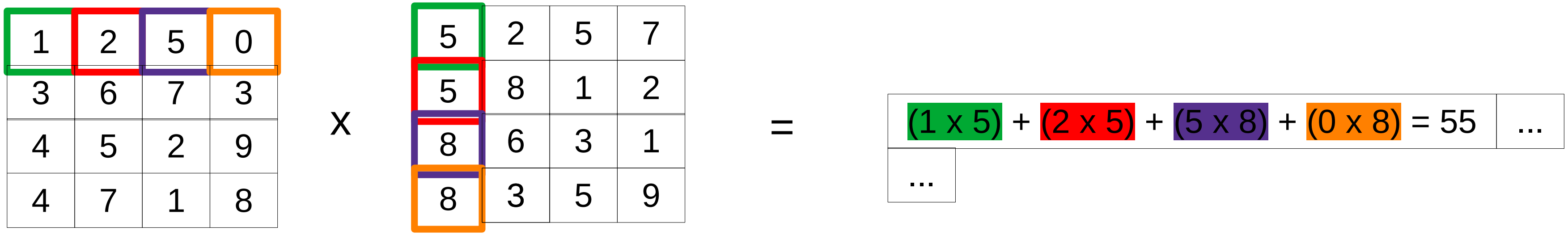
x

5	2	5	7
5	8	1	2
8	6	3	1
8	3	5	9

(1 x 5) + (1 x 8) = 13	...
...	

Only do the multiplications that
can result in a non-zero result

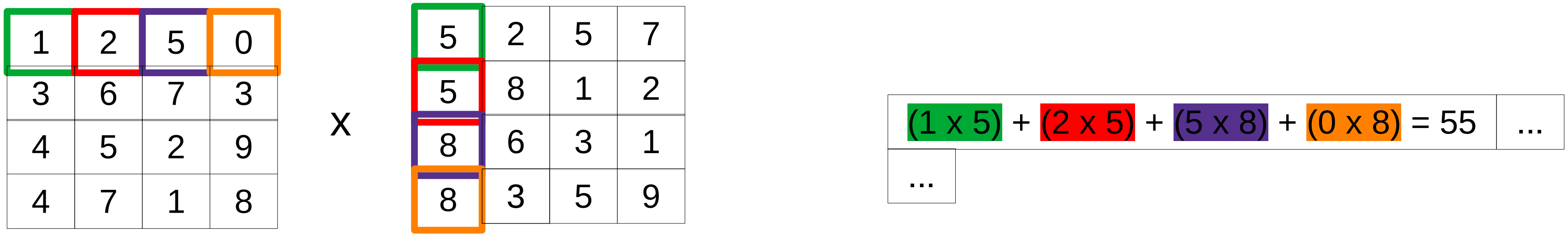
Sparsity patterns



$$\begin{pmatrix}1&2&5&0\\3&6&7&3\\4&5&2&9\\4&7&1&8\end{pmatrix} \times \begin{pmatrix}5&2&5&7\\5&8&1&2\\8&6&3&1\\8&3&5&9\end{pmatrix} = \begin{pmatrix}(1 \times 5) + (2 \times 5) + (5 \times 8) + (0 \times 8) = 55 & \dots\\ \dots & \end{pmatrix}$$

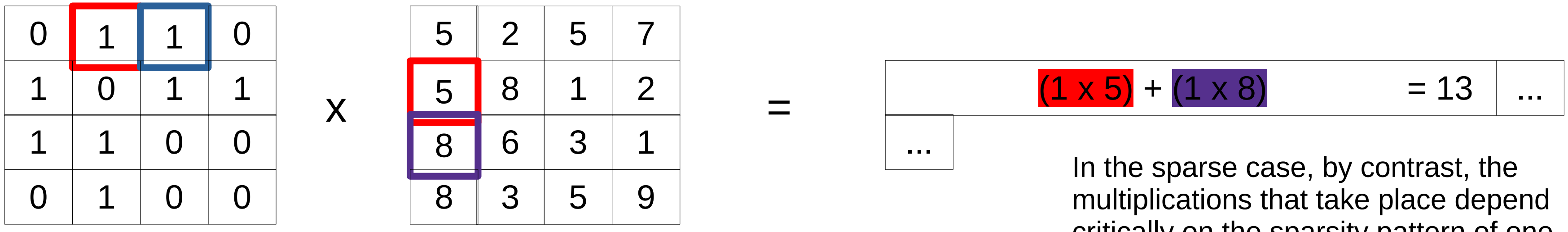
In the dense case the pattern of operations is constant and independent of the actual values in the argument matrices

Sparsity patterns



$$\begin{bmatrix} 1 & 2 & 5 & 0 \\ 3 & 6 & 7 & 3 \\ 4 & 5 & 2 & 9 \\ 4 & 7 & 1 & 8 \end{bmatrix} \times \begin{bmatrix} 5 & 2 & 5 & 7 \\ 5 & 8 & 1 & 2 \\ 8 & 6 & 3 & 1 \\ 8 & 3 & 5 & 9 \end{bmatrix}$$

$$\begin{array}{|c|c|} \hline (1 \times 5) + (2 \times 5) + (5 \times 8) + (0 \times 8) = 55 & \ldots \\ \hline \ldots & \\ \hline \end{array}$$

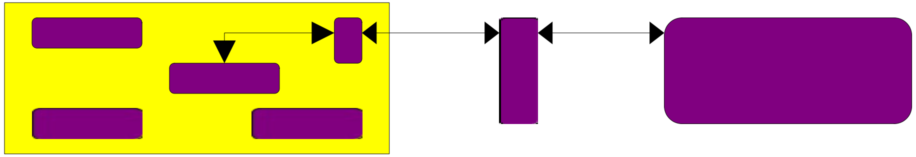


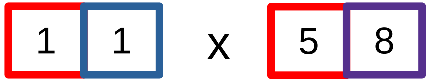
$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 5 & 2 & 5 & 7 \\ 5 & 8 & 1 & 2 \\ 8 & 6 & 3 & 1 \\ 8 & 3 & 5 & 9 \end{bmatrix} =$$

$$\begin{array}{|c|c|} \hline (1 \times 5) + (1 \times 8) = 13 & \ldots \\ \hline \ldots & \\ \hline \end{array}$$

In the sparse case, by contrast, the multiplications that take place depend critically on the sparsity pattern of one or both argument matrices

Sparsity and performance





1 1 x 5 8

The compiler only knows what to move to which cache with reference to the actual data in the argument matrices

...and we can only get performance when we have the data in cache: if it’s in main memory the core can’t run at anything approaching full speed

...and we can’t know this until run-time

Sparsity and storage

$$\begin{array}{cccccc}
 & 1 & 2 & & & \\
1 & 0 & 1 & \dots & 1 & 0 \\
2 & 1 & 0 & & 1 & 1 \\
 & \dots & & & \dots & \\
999{,}999 & 1 & 1 & \dots & 0 & 0 \\
1{,}000{,}000 & 0 & 1 & & 0 & 0
\end{array}$$

Consider the adjacency matrix for
a 1,000,000-node graph

Sparsity and storage

$$\begin{array}{c|ccccc} & 1 & 2 & & & \\ 1 & 0 & 1 & \cdots & 1 & 0 \\ 2 & 1 & 0 & & 1 & 1 \\ & \cdots & & & \cdots & \\ \text{999,999} & 1 & 1 & & 0 & 0 \\ \text{1,000,000} & 0 & 1 & \cdots & 0 & 0 \end{array}$$

If each node has on average 100 neighbours, each row has on average 100 non-zero elements and 999,900 zero elements

Zero almost everywhere

Sparse representations

1	2	5	0
3	6	7	3
4	5	2	9
4	7	1	8

Dense: two representations
(row- *versus* column-major)

Sparse representations

0	1	1	0
1	0	1	1
1	1	0	0
0	1	0	0

Sparse: 11+ representations

Sparse representations

0	1	1	0
1	0	1	1
1	1	0	0
0	1	0	0

Sparse: 11+ representations

```
(((0 1) 1) ((0 2) 1)
 ((1 0) 1) ((1 2) 1) ((1 3) 1)
 ((2 0) 1) ((2 1) 1)
 ((3 1) 1))
```

Co-ordinate list (COO)

```
(((1 1) (2 1))
 ((0 1 (2 1) (3 1))
 ((0 1) (1 1))
 ((1 1)))
```

List of lists (LoL)

Sparse representations

0	1	1	0
1	0	1	1
1	1	0	0
0	1	0	0

Sparse: 11+ representations

(((0 1) 1) ((0 2) 1)
((1 0) 1) ((1 2) 1) ((1 3) 1)
((2 0) 1) ((2 1) 1)
((3 1) 1))

Co-ordinate list (COO)

(((1 1) (2 1))
((0 1 (2 1) (3 1))
((0 1) (1 1))
((1 1)))

List of lists (LoL)

V	1	1	1	1	1	1	1	1

The non-zero values
in row-major order

CINDEX	1	2	0	2	3	0	1	1

The column position of each
non-zero element in its row

RINDEX	1	2	5	7	8

The start of each row,
as an index into V

Compressed Sparse Row (CSR)

The length of V

Sparse representations

0	1	1	0
1	0	1	1
1	1	0	0
0	1	0	0

Sparse: 11+ representations

(((0 1) 1) ((0 2) 1)
((1 0) 1) ((1 2) 1) ((1 3) 1)
((2 0) 1) ((2 1) 1)
((3 1) 1))

Co-ordinate list (COO)

(((1 1) (2 1))
((0 1 (2 1) (3 1))
((0 1) (1 1))
((1 1)))

List of lists (LoL)

V	1	1	1	1	1	1	1	1	The non-zero values in row-major order
CINDEX	1	2	0	2	3	0	1	1	The column position of each non-zero element in its row
RINDEX	1	2	5	7		8			The start of each row, as an index into V

Compressed Sparse Row (CSR)

Sparse representations

0	1	1	0
1	0	1	1
1	1	0	0
0	1	0	0

Sparse: 11+ representations

(((0 1) 1) ((0 2) 1)
((1 0) 1) ((1 2) 1) ((1 3) 1)
((2 0) 1) ((2 1) 1)
((3 1) 1))

Co-ordinate list (COO)

(((1 1) (2 1))
((0 1 (2 1) (3 1))
((0 1) (1 1))
((1 1)))

List of lists (LoL)

...and *none* of these representations make use of the two most obvious facts:

- All non-zero values are 1
- The matrix is symmetrical ($A_{ij} = A_{ji}$)

V	1	1	1	1	1	1	1	1
CINDEX	1	2	0	2	3	0	1	1
RINDEX	1	2	5	7	8			

Compressed Sparse Row (CSR)

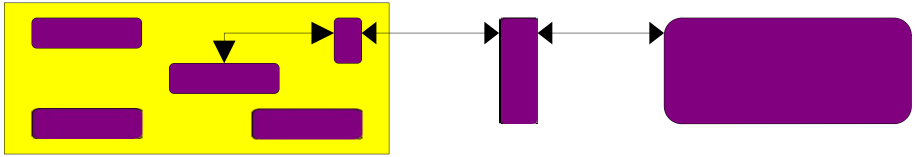
Our challenge

Make this:

V	1	1	1	1	1	1	1	1
CINDEX	1	2	0	2	3	0	1	1
RINDEX	1	2	5	7	8			

Compressed Sparse Row (CSR)

Work efficiently on this:



Our challenge

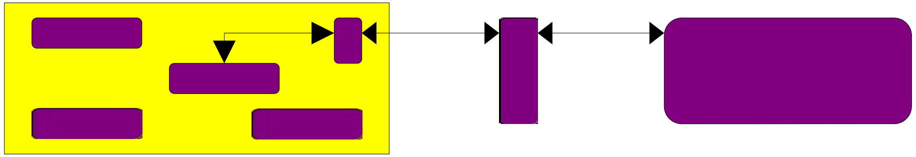
Make this:

V	1	1	1	1	1	1	1	1
CINDEX	1	2	0	2	3	0	1	1
RINDEX	1	2	5	7	8			

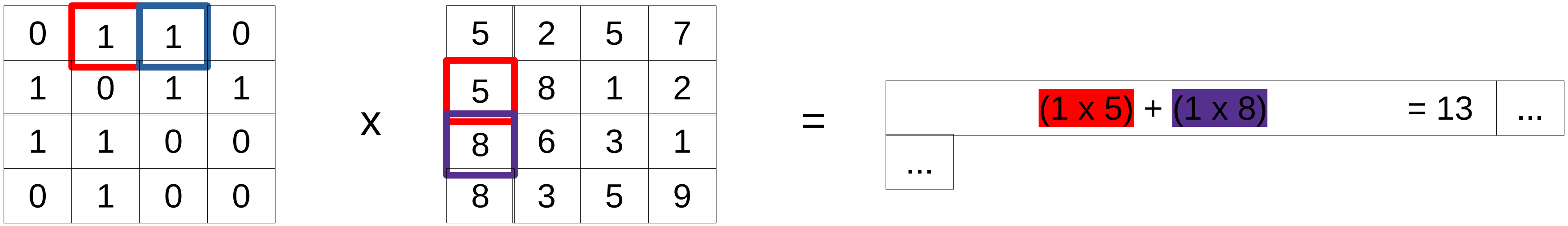
Compressed Sparse Row (CSR)

What are the patterns we observe in (sparse) matrix operations that we can use to drive efficient computation?

Work efficiently on this:



The plan



$$\begin{bmatrix}0&\boxed{1}&\boxed{1}&0\\1&0&1&1\\1&1&0&0\\0&1&0&0\end{bmatrix} \text{x} \begin{bmatrix}5&2&5&7\\\boxed{5}&8&1&2\\\boxed{8}&6&3&1\\8&3&5&9\end{bmatrix} = \begin{bmatrix}(1 \text{ x } 5) + (1 \text{ x } 8) = 13 & \dots\\ \dots & \end{bmatrix}$$

How do the different representations interact? Are there special cases?

V	1	1	1	1	1	1	1	1
CINDEX	1	2	0	2	3	0	1	1
RINDEX	1	2	5	7		8		

x

5	5	8	8	2	8	6	3	...

The plan

$$\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 0 \\
2 & 1 & 0 & 1 & 1 \\
3 & 1 & 1 & 0 & 0 \\
4 & 0 & 1 & 0 & 0
\end{array}$$

V	1	1	1	1	1	1	1	1
CINDEX	1	2	0	2	3	0	1	1
RINDEX	1	2	5	7	8			

Are there efficient ways of deriving the interesting operators under different representations?

$L = D - A$

$$\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 2 & -1 & -1 & 0 \\
2 & -1 & 3 & -1 & -1 \\
3 & -1 & -1 & 2 & 0 \\
4 & 0 & -1 & 0 & -1
\end{array}$$

V	2	-1	-1	-1	3	3	-1	-1	…
CINDEX	0	1	2	0	1	2	3	0	…
RINDEX	3	7	…						

Present state

Writing a matrix library that records the basic operations (as well as doing them)

- Gives us a stream of element-wise multiplications and additions

Implement the algorithms against (descriptions of) the different representations

- What sorts of locality of reference are there?
- Do the special operators generate characteristic patterns of data or operations?

Wherever there is a *pattern* (if you can find it)...

...there is (potentially) an *optimisation* (if you can find it)

...and this is hopefully how we find it