

CS601: Software Development for Scientific Computing

Autumn 2024

Week8: Motifs- Sparse Matrix Computation,
Intermediate C++

Matrix Data and Efficiency

- Sparse Matrices
 - E.g. banded matrices
 - Diagonal
 - Tridiagonal etc.
 - Symmetric Matrices
- Admit optimizations w.r.t.* →
- Storage
 - Computation

Sparse Matrices - Motivation

- Matrix Multiplication with Upper Triangular Matrices
($C=C+AB$)

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

A
 B
 $A*B$

The result, $A*B$, is also upper triangular.

The non-zero elements appear to be like the result of *inner-product*

Sparse Matrices - Motivation

- $C=C+AB$ when A, B, C are upper triangular, pseudocode:
 for $i=$
 for $j=$
 for $k=$
 $C[i][j] = C[i][j] + A[i][k]*B[k][j]$

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

A
 B
 $A*B$

Sparse Matrices - Motivation

- $C=C+AB$ when A, B, C are upper triangular, pseudocode:
for $i=1$ to N
 for $j=i$ to N
 for $k=i$ to j
 $C[i][j] = C[i][j] + A[i][k]*B[k][j]$
- Cost = $\sum_{i=1}^N \sum_{j=i}^N 2(j-i+1)$ flops (why 2?)
- Using $\sum_{i=1}^N i \approx \frac{n^2}{2}$ and $\sum_{i=1}^N i^2 \approx \frac{n^3}{3}$
- $\sum_{i=1}^N \sum_{j=i}^N 2(j-i+1) \approx \frac{n^3}{3}$, 1/3rd the number of flops required for dense matrix-matrix multiplication

Sparse Matrices

- Have lots of zeros (a *large* fraction)

X	X	0	0	X	0	0	0	X
0	X	0	0	X	0	X	0	0
0	X	X	X	0	X	0	0	X
X	0	0	X	0	0	X	0	0
0	X	0	X	X	0	0	0	X
0	X	X	0	0	0	X	X	X

- Representation
 - Many formats available
 - Compressed Sparse Row (CSR)

Implementation: Three arrays:

```
double *val;  
int *ind;  
int *rowstart;
```

Sparse Matrices - Example

- Using Arrays

A

a_{11}	a_{12}	0	0	a_{15}	0	0	0	a_{19}
0	a_{22}	0	0	a_{25}	0	a_{27}	0	0
0	a_{32}	a_{33}	a_{34}	0	a_{36}	0	0	a_{39}
a_{41}	0	0	a_{44}	0	0	a_{47}	0	0
0	a_{52}	0	a_{54}	a_{55}	0	0	0	a_{59}
0	a_{62}	a_{63}	0	0	0	a_{67}	a_{68}	a_{69}

```
double *val; //size= NNZ
int *ind; //size=NNZ
int *rowstart; //size=M=Number of rows
```

val:

a_{11}	a_{12}	a_{15}	a_{19}	a_{22}	a_{25}	a_{27}	a_{32}	a_{33}	a_{34}	a_{36}	a_{39}	a_{41}	a_{44}	a_{47}	a_{52}	a_{54}	a_{55}	a_{59}	a_{62}	a_{63}	a_{67}	a_{68}	a_{69}
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

ind:

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

rowstart:

0	4	7	12	15	19	
---	---	---	----	----	----	--

Nikhil Hegde

Gaxpy with Sparse Matrices: $y=y+Ax$

- Using arrays

```
for i=0 to numRows
```

```
    for j=rowstart[i] to rowstart[i+1]-1
```

```
        y[i] = y[i] + val[j]*x[ind[j]]
```

- Does the above code reuse y , x , and val ? (we want our code to reuse as much data elements as possible while they are in fast memory):
 - y ? Yes. Read and written in close succession.
 - x ? Possible. Depends on how data is scattered in val .
 - val ? Good spatial locality here. Less likely for a sparse matrix in general.

Gaxpy with Sparse Matrices: $y=y+Ax$

- Optimization strategies:

```
for i=0 to numRows
```

```
    for j=rowstart[i] to rowstart[i+1]-1
```

```
        y[i] = y[i] + val[j]*x[ind[j]]
```

- Unroll the j loop // we need to know the number of non-zeros per row
- Eliminate ind[i] and thereby the indirect access to elements of x. Indirect access is not good because we cannot predict the pattern of data access in x. //We need to know the column numbers
- Reuse elements of x //The elements of a should be e.g. located closely

These optimizations will not work for $y=y+Ax$ pseudocode in general. When you know the data pattern and metadata info as mentioned above, you can reorder computations (scheduling optimization), reorganize data for better locality.

Banded Matrices

- Special case of sparse matrices, characterized by two numbers:

- Lower bandwidth p , and upper bandwidth q
- E.g. $p=1$, $q=2$

for a 8×5 matrix

(x represents non-zero element)

x	x	x	0	0
x	x	x	x	0
0	x	x	x	x
0	0	x	x	x
0	0	0	x	x
0	0	0	0	x
0	0	0	0	0
0	0	0	0	0

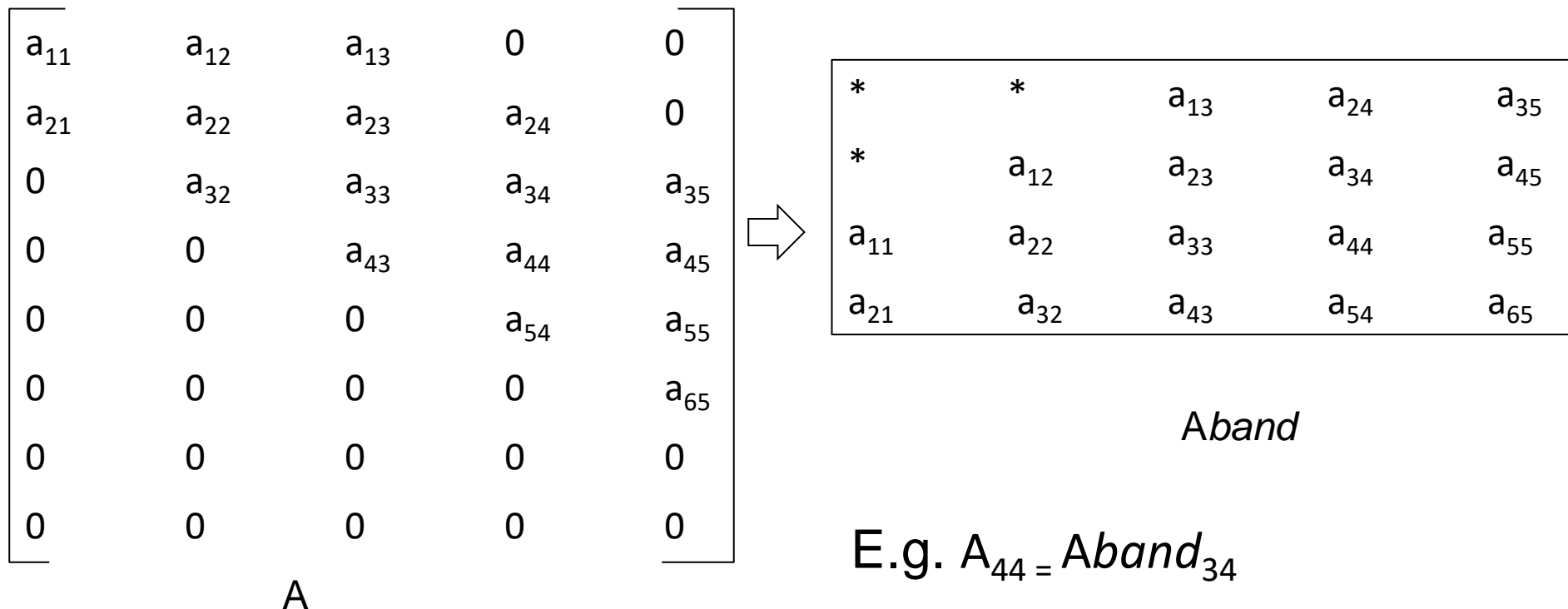
Exercise: When is $a_{ij} = 0$?

(Write the constraints in terms of i, j, p, q)

- $a_{ij} = 0$ if $i > j+p$
- $a_{ij} = 0$ if $j > i+q$

Banded Matrices - Representation

- Optimizing storage (specific to banded matrices)



Exercise: $A_{ij} = Aband(i-j+q+1, j)$

Gaxpy with Banded Matrices: $y = y + A \text{band } x$

- $A = A_{\text{band}}$: optimizing computation and storage

```
for j=1 to n
    alpha1=max(1, j-q)
    alpha2=min(n, j+p)
    beta1=max(1, q+2-j)
    for i=alpha1 to alpha2
         $y[i] = y[i] + A_{\text{band}}(\text{beta1}+i-\text{alpha1}, j) * x[j]$ 
```

- Cost? $2n(p+q+1)$ time! Much lesser than $2N^2$ time required for regular $y = y + Ax$ (assuming p and q are much smaller than n)

References and Const

- C++ reference variables: closely related to pointers
 - Directly name another object of the *same* type.
 - A reference is defined using the & (address of operator) symbol. Furthermore, unlike in pointer definitions, a reference must be defined/initialized with the object that it names (*cannot be changed later*).

References

```
int n=10;
int &re=n; //re must be initialized
int* ptr; //ptr need not be initialized here
ptr=&n //ptr now initialized (now pointing to n)
int x=20;
ptr=&x; //ptr now pointing to x
re=x; //doesn't do what you think. Re still points to
n but the value of n is changed..
printf("%p %p\n",&re, &n); // re and n are naming the
same box in memory. Hence, they have the same address.
```

const

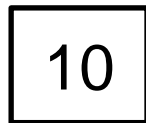
- A type qualifier
- The type is a constant (cannot be modified).
- const is the keyword
- Example:

```
const int x=10; //equivalent to: int const x=10;  
//x is a constant integer. Hence, cannot be  
//modified.
```

In what memory segment does x gets stored?

Const Properties

- Needs to be initialized at the time of definition
- Can't modify after definition
- `const int x=10;`
`x=20;` //compiler would throw an error
- `int const x=10;`
`x=10;` //can't even assign the same value
- `int const y;` //uninitialized const variable y. Useless.



← Can't alter the content of this box

x

Const Example1 (error)

*/*ptrCX is a pointer to a constant integer. So, can't modify what ptrCX points to.*/*

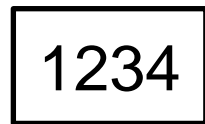
const int ptrCX; //or equivalently:*

int const ptrCX;*

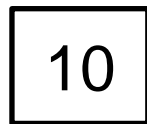
int const x=10;

ptrCX = &x;

**ptrCX = 20; //Error*



ptrCX



x

Addr: 1234

← Can't alter the content of this box
using ptrCX or x

Const Example2 (error)

```
/*cptrX is a constant pointer to an integer. So, can't  
point to anything else after initialized.*/  
int x=10, y=20;  
int *const cptrX=&x;  
cptrX = &y; //Error
```

Can't alter the
content of this box

1234

cptrX

10

x

Addr: 1234

20

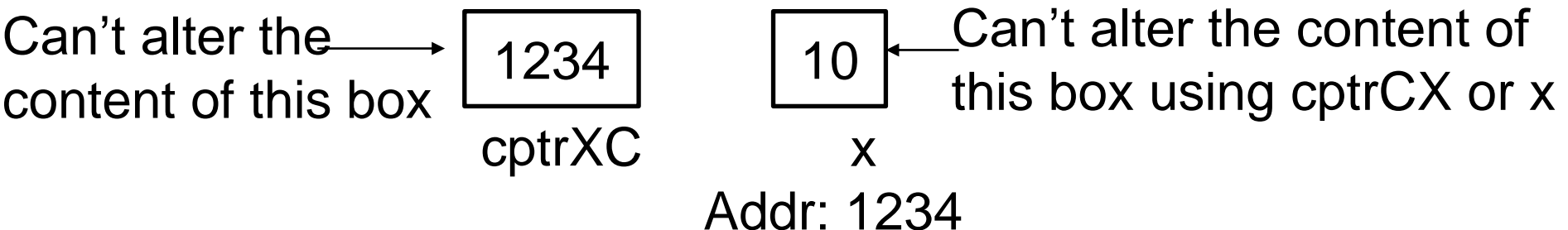
y

Addr: 5678

Const Example3 (error)

*/*cptrXC is a constant pointer to a constant integer. So, can't point to anything else after initialized. Also, can't modify what cptrXC points to.*/*

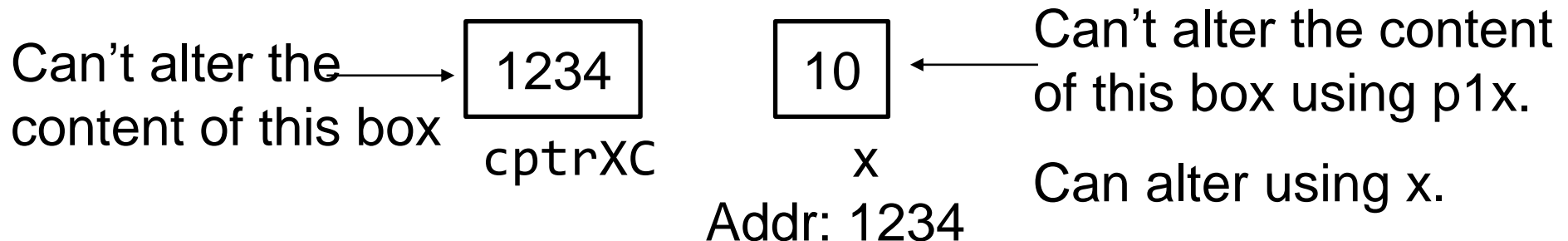
```
const int x=10, y=20;  
const int *const cptrXC=&x;  
int const *const cptrXC2=&x; //equivalent to prev. defn.  
cptrXC = &y; //Error  
*cptrXC = 40; //Error
```



Const Example4 (no warning, no error)

`/* cptrXC is a constant pointer to a constant integer. So, can't point to anything else after initialized. Also, can't modify what cptrXC points to (via cptrXC). However, the memory location it points to can be altered (through some other means e.g. using x)*/`

```
int x=10;
const int *const cptrXC=&x;
```



Const Example5 (warning)

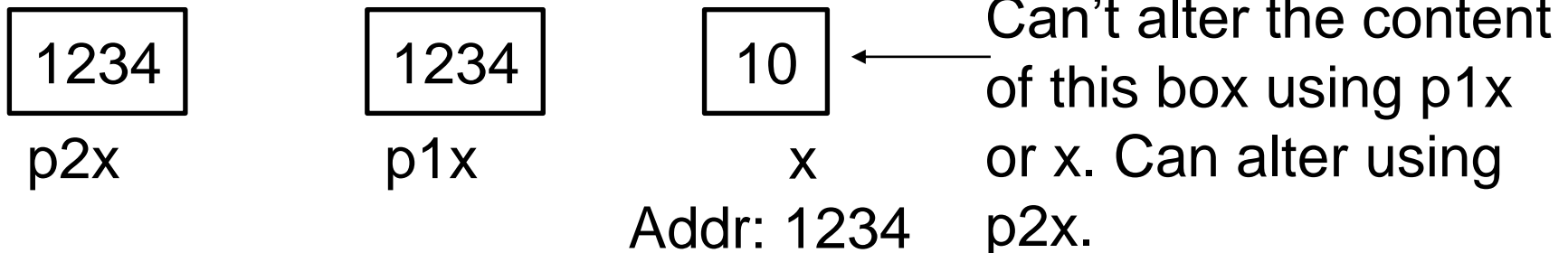
`/*p2x is a pointer to an integer.` So, we can use p2x to alter the contents of the memory location that it points to. However, the memory location contains read-only data - cannot be altered. `*/`

```
const int x=10;
```

```
const int *p1x=&x;
```

```
int *p2x=&x; //warning
```

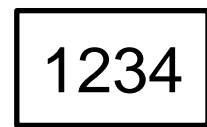
```
*p2x = 20; //goes through. Might crash depending on memory location accessed
```



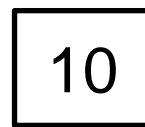
Const Example6 (no warning, no error)

*/*p1x is a pointer to a constant integer.* So, we can't use p1x to alter the content of the memory location that it points to. However, the memory location it points to can be altered (through some other means e.g. using x)*/

```
int x=10;  
const int *p1x=&x;
```



p1x



x

Addr: 1234

← Can't alter the content of this box using p1x.

Can alter using x.

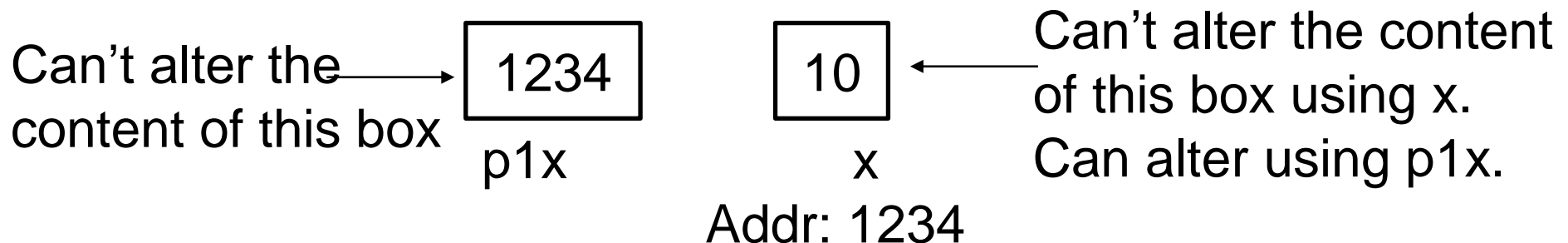
Const Example7 (warning)

*/*p1x is a constant pointer to an integer.* So, we can use p1x to alter the contents of the memory location that it points to (and we can't let p1x point to something else other than x). However, the memory location contains read-only data - cannot be altered. */

```
const int x=10;
```

```
int *const p1x=&x; //warning
```

```
*p1x = 20; //goes through. Might crash depending on memory location accessed
```



Const and References - Summary

- Allow for compiler optimizations
 - pass-by-reference: allows for passing large objects to a function call
- Tell us immediately (by looking at the interface) that a parameter is read-only