PracticeSession05

May 1, 2025

Numerical Linear Algebra for Computational Science and Information Engineering

Sparse Data Structures and Basic Linear Algebra Subprograms

Nicolas Venkovic (nicolas.venkovic@tum.edu)

[1]:
```julia
using LinearAlgebra, Plots, Printf, Latexify, LaTeXStrings, BenchmarkTools
```

Exercise #1: Julia's built-in CSC sparse matrix format

[2]:
```julia
using MatrixMarket: mmread
using SparseArrays
```

The built-in sparse matrix format of Julia is CSC:

[3]:
```julia
A_csc = mmread("../../../matrix-market/cage3.mtx")
```

[3]:
```
5×5 SparseMatrixCSC{Float64, Int64} with 19 stored entries:
 0.666667   0.366556  0.300111  0.366556  0.300111
 0.100037   0.533407            0.200074
 0.122185             0.577704            0.244371
 0.0500184  0.100037            0.283315  0.183278
 0.0610927            0.122185  0.150055  0.272241
```

[4]:
```julia
dump(SparseMatrixCSC)
```

```
UnionAll
  var: TypeVar
    name: Symbol Tv
    lb: Union{}
    ub: Any
  body: UnionAll
    var: TypeVar
      name: Symbol Ti
      lb: Union{}
      ub: Integer <: Real
    body: SparseMatrixCSC{Tv, Ti<:Integer} <:
SparseArrays.AbstractSparseMatrixCSC{Tv, Ti<:Integer}
```

```
m::Int64
n::Int64
colptr::Vector{Ti}
rowval::Vector{Ti}
nzval::Vector{Tv}
```

[5]:
```
A_csc.m, A_csc.n
```

[5]:
```
(5, 5)
```

[6]:
```
A_csc.nzval
```

[6]:
```
19-element Vector{Float64}:
 0.6666666666666667
 0.100036889486116
 0.122185332736106
 0.050018444743058
 0.0610926663680531
 0.366555998208319
 0.533407112305565
 0.100036889486116
 0.300110668458348
 0.577703998805546
 0.122185332736106
 0.366555998208319
 0.200073778972232
 0.283314888590275
 0.150055334229174
 0.300110668458348
 0.244370665472212
 0.183277999104159
 0.27224066696528
```

[7]:
```
A_csc.rowval, A_csc.colptr
```

[7]:
```
([1, 2, 3, 4, 5, 1, 2, 4, 1, 3, 5, 1, 2, 4, 5, 1, 3, 4, 5], [1, 6, 9, 12, 16,
20])
```

In CSC, the non-zero values are stored by column.

For the CSC format, the SpMV kernel has a column-wise inner-most loop which iterates over row indices of a given colum, and the outer loop iterates over columns. An implementation of the SpMV kernel is given by

Exercise #2: SpMV kernel for the CSC format

[10]:
```
function dcscmv(A::SparseMatrixCSC, x::Vector{Float64})
    y = zeros(A.m)
    for j in 1:A.n
        for i in A.colptr[j]:A.colptr[j+1]-1
            y[A.rowval[i]] += A.nzval[i] * x[j]
        end
    end
    return y
end
```

[10]:
```
dcscmv (generic function with 1 method)
```

[11]:
```
x = rand(A_csc.n);
@btime A_csc * x
```

```
  30.767 ns (2 allocations: 96 bytes)
```

[11]:
```
5-element Vector{Float64}:
 0.9604101914079568
 0.24553763592463487
 0.6198029953005734
 0.3279669573558319
 0.34004500105563085
```

[12]:
```
@btime dcscmv(A_csc,x)
```

```
  35.037 ns (2 allocations: 96 bytes)
```

[12]:
```
5-element Vector{Float64}:
 0.9604101914079568
 0.24553763592463487
 0.6198029953005734
 0.3279669573558319
 0.34004500105563085
```

Exercise #3: Convert CSC matrix to COO format

[13]:
```
mutable struct SparseMatrixCOO
    m::Int # Number of rows
    n::Int # Number of columns
    rowval::Vector{Int} # Starting index for each row
    colval::Vector{Int} # Column indices
    nzval::Vector{Float64} # Matrix entries
end
```

[14]:
```
function csc_to_coo(A::SparseMatrixCSC)
    m, n = A.m, A.n
```

```julia
    nzval = A.nzval
    rowval = A.rowval
    colval = similar(A.rowval)
    j = 0
    for jj in 1:n
        j += 1
        for k in A.colptr[jj]:A.colptr[jj+1]-1
            colval[k] = j
        end
    end
    return SparseMatrixCOO(m, n, rowval, colval, nzval)
end
```

[14]: csc_to_coo (generic function with 1 method)

[15]:
```julia
A_coo = @btime csc_to_coo(A_csc);
```

  30.814 ns (3 allocations: 256 bytes)

[16]:
```julia
A_coo.m, A_coo.n
```

[16]: (5, 5)

[17]:
```julia
A_coo.nzval
```

[17]:
```
19-element Vector{Float64}:
 0.6666666666666667
 0.100036889486116
 0.122185332736106
 0.050018444743058
 0.0610926663680531
 0.366555998208319
 0.533407112305565
 0.100036889486116
 0.300110668458348
 0.577703998805546
 0.122185332736106
 0.366555998208319
 0.200073778972232
 0.283314888590275
 0.150055334229174
 0.300110668458348
 0.244370665472212
 0.183277999104159
 0.27224066696528
```

[18]:
```julia
A_coo.rowval, A_coo.colval
```

[18]: ([1, 2, 3, 4, 5, 1, 2, 4, 1, 3, 5, 1, 2, 4, 5, 1, 3, 4, 5], [1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5])

Exercise #4: SpMV kernel for the COO format

In the COO format, the non-zero elements are not stored in any particular order.

The corresponing SpMV kernel is written simply by browsing through the non-zero elements in the order they are given.

[19]:
```julia
function dcoomv(A::SparseMatrixCOO, x::Vector{Float64})
    y = zeros(A.m)
    for k in 1:length(A.nzval)
        y[A.rowval[k]] += A.nzval[k] * x[A.colval[k]]
    end
    return y
end
```

[19]: dcoomv (generic function with 1 method)

[20]:
```julia
@btime dcoomv(A_coo, x)
```

```
27.097 ns (2 allocations: 96 bytes)
```

[20]:
```
5-element Vector{Float64}:
 0.9604101914079568
 0.24553763592463487
 0.6198029953005734
 0.3279669573558319
 0.34004500105563085
```

One built-in way to define a `SparseMatrixCSC` is through the elements of a COO datastructure:

Exercise #5: Convert COO matrix to CSR format

[23]:
```julia
mutable struct SparseMatrixCSR
    m::Int # Number of rows
    n::Int # Number of columns
    rowptr::Vector{Int} # Starting index for each row
    colval::Vector{Int} # Column indices
    nzval::Vector{Float64} # Matrix entries
end
```

[24]:
```julia
function coo_to_csr(A::SparseMatrixCOO)
    nnz = length(A.nzval)

    coo_tuples = [(A.rowval[i], A.colval[i], A.nzval[i]) for i in 1:nnz]
    sort!(coo_tuples, by = x -> (x[1], x[2]))
```

```julia
    colval = [t[2] for t in coo_tuples]
    nzval = [t[3] for t in coo_tuples]

    rowptr = zeros(Int, A.n+1)
    # Count number of non-zero values per row
    for i in A.rowval
        rowptr[i+1] += 1
    end

    # Prefix sum to get rowptr
    rowptr[1] = 1
    for i in 1:A.n
        rowptr[i+1] += rowptr[i]
    end

    return SparseMatrixCSR(A.m, A.n, rowptr, colval, nzval)
end
```

[24]:
```
coo_to_csr (generic function with 1 method)
```

[25]:
```julia
A_csr = @btime coo_to_csr(A_coo)
```

```
  154.489 ns (11 allocations: 1.59 KiB)
```

[25]:
```
SparseMatrixCSR(5, 5, [1, 6, 9, 12, 16, 20], [1, 2, 3, 4, 5, 1, 2, 4, 1, 3, 5,
1, 2, 4, 5, 1, 3, 4, 5], [0.6666666666666667, 0.366555998208319,
0.300110668458348, 0.366555998208319, 0.300110668458348, 0.100036889486116,
0.533407112305565, 0.200073778972232, 0.122185332736106, 0.577703998805546,
0.244370665472212, 0.050018444743058, 0.100036889486116, 0.283314888590275,
0.183277999104159, 0.0610926663680531, 0.122185332736106, 0.150055334229174,
0.27224066696528])
```

[26]:
```julia
A_csr.m, A_csr.n
```

[26]:
```
(5, 5)
```

[27]:
```julia
A_csr.nzval
```

[27]:
```
19-element Vector{Float64}:
 0.6666666666666667
 0.366555998208319
 0.300110668458348
 0.366555998208319
 0.300110668458348
 0.100036889486116
 0.533407112305565
 0.200073778972232
 0.122185332736106
```

```
 0.577703998805546
 0.244370665472212
 0.050018444743058
 0.100036889486116
 0.283314888590275
 0.183277999104159
 0.0610926663680531
 0.122185332736106
 0.150055334229174
 0.27224066696528
```

[28]:
```julia
A_csr.colval, A_csr.rowptr
```

[28]:
```
([1, 2, 3, 4, 5, 1, 2, 4, 1, 3, 5, 1, 2, 4, 5, 1, 3, 4, 5], [1, 6, 9, 12, 16,
20])
```

Exercise #6: Convert COO matrix to CSR format using the sparse function

[29]:
```julia
function coo_to_csr2(A::SparseMatrixCOO)
    A_csc = sparse(A.colval, A.rowval, A.nzval, A.m, A.n)
    return SparseMatrixCSR(A.m, A.n, A_csc.colptr, A_csc.rowval, A_csc.nzval)
end
```

[29]:
```
coo_to_csr2 (generic function with 1 method)
```

[30]:
```julia
A_csr2 = @btime coo_to_csr2(A_coo)
```

```
  215.712 ns (16 allocations: 1.22 KiB)
```

[30]:
```
SparseMatrixCSR(5, 5, [1, 6, 9, 12, 16, 20], [1, 2, 3, 4, 5, 1, 2, 4, 1, 3, 5,
1, 2, 4, 5, 1, 3, 4, 5], [0.6666666666666667, 0.366555998208319,
0.300110668458348, 0.366555998208319, 0.300110668458348, 0.100036889486116,
0.533407112305565, 0.200073778972232, 0.122185332736106, 0.577703998805546,
0.244370665472212, 0.050018444743058, 0.100036889486116, 0.283314888590275,
0.183277999104159, 0.0610926663680531, 0.122185332736106, 0.150055334229174,
0.27224066696528])
```

[31]:
```julia
A_csr2.m, A_csr2.n
```

[31]:
```
(5, 5)
```

[32]:
```julia
A_csr2.nzval
```

[32]:
```
19-element Vector{Float64}:
 0.6666666666666667
 0.366555998208319
 0.300110668458348
 0.366555998208319
```

```
0.300110668458348
0.100036889486116
0.533407112305565
0.200073778972232
0.122185332736106
0.577703998805546
0.244370665472212
0.050018444743058
0.100036889486116
0.283314888590275
0.183277999104159
0.0610926663680531
0.122185332736106
0.150055334229174
0.27224066696528
```

[33]:
```
A_csr2.colval, A_csr2.rowptr
```

[33]:
```
([1, 2, 3, 4, 5, 1, 2, 4, 1, 3, 5, 1, 2, 4, 5, 1, 3, 4, 5], [1, 6, 9, 12, 16,
20])
```

Exercise #7: SpMV kernel for the CSR format

In CSR, the non-zero values are stored by row.

For the CSR format, the more efficient SpMV kernel has a row-wise inner-most loop which iterates over column indices of a given row, and the outer loop iterates over rows. An implementation of the SpMV kernel is given by

[34]:
```
function dcsrmv(A::SparseMatrixCSR, x::Vector{Float64})
    y = zeros(A.m)
    for i in 1:A.m
        for j in A.rowptr[i]:A.rowptr[i+1]-1
            y[i] += A.nzval[j] * x[A.colval[j]]
        end
    end
    return y
end
```

[34]:
```
dcsrmv (generic function with 1 method)
```

[35]:
```
@btime dcsrmv(A_csr, x)
```

```
  28.958 ns (2 allocations: 96 bytes)
```

[35]:
```
5-element Vector{Float64}:
 0.9604101914079568
 0.24553763592463487
 0.6198029953005734
```

```
0.3279669573558319
0.34004500105563085
```

Exercise #8: Convert COO matrix to ELL format

[36]:
```
mutable struct SparseMatrixELL
    m::Int # Number of rows
    n::Int # Number of columns
    rownnz::Int # Maximum number of non-zero values per row
    colval::Vector{Int} # Column indices
    nzval::Vector{Float64} # Matrix entries
end
```

[37]:
```
function coo_to_ell(A::SparseMatrixCOO)
    # Count non-zeros per row
    nnz_per_row = zeros(Int, A.n)
    for r in A.rowval
        nnz_per_row[r] += 1
    end

    # Find maximum number of non-zeros in any row
    rownnz = maximum(nnz_per_row)

    # Create ELL format arrays
    ell_size = A.n * rownnz
    nzval = zeros(eltype(A.nzval), ell_size)
    colval = fill(-1, ell_size)  # -1 indicates padding

    # Track how many elements we've processed for each row
    row_counts = zeros(Int, A.n)


    # Fill ELL arrays in column-major order
    for i in 1:length(A.nzval)
        r = A.rowval[i]
        c = A.colval[i]
        v = A.nzval[i]

        # Calculate position in column-major ELL format
        pos = row_counts[r] * A.n + r
        nzval[pos] = v
        colval[pos] = c
        row_counts[r] += 1
    end

    return SparseMatrixELL(A.m, A.n, rownnz, colval, nzval)
end
```

[37]: coo_to_ell (generic function with 1 method)

[38]:
```
A_ell = @btime coo_to_ell(A_coo)
```

```
94.120 ns (9 allocations: 752 bytes)
```

[38]:
```
SparseMatrixELL(5, 5, 5, [1, 1, 1, 1, 1, 2, 2, 3, 2, 3  …  4, -1, -1, 5, 5, 5,
-1, -1, -1, -1], [0.6666666666666667, 0.100036889486116, 0.122185332736106,
0.050018444743058, 0.0610926663680531, 0.366555998208319, 0.533407112305565,
0.577703998805546, 0.100036889486116, 0.122185332736106  …  0.366555998208319,
0.0, 0.0, 0.183277999104159, 0.27224066696528, 0.300110668458348, 0.0, 0.0, 0.0,
0.0])
```

[39]:
```
A_ell.m, A_ell.n
```

[39]: (5, 5)

[40]:
```
A_ell.rownnz
```

[40]: 5

[41]:
```
A_ell.nzval
```

[41]:
```
25-element Vector{Float64}:
 0.666666666666667
 0.100036889486116
 0.122185332736106
 0.050018444743058
 0.0610926663680531
 0.366555998208319
 0.533407112305565
 0.577703998805546
 0.100036889486116
 0.122185332736106
 0.300110668458348
 0.200073778972232
 0.244370665472212
 0.283314888590275
 0.150055334229174
 0.366555998208319
 0.0
 0.0
 0.183277999104159
 0.27224066696528
 0.300110668458348
 0.0
 0.0
 0.0
```

```
0.0
```

[42]:
```julia
A_ell.colval
```

[42]:
```
25-element Vector{Int64}:
  1
  1
  1
  1
  1
  2
  2
  3
  2
  3
  3
  4
  5
  4
  4
  4
 -1
 -1
  5
  5
  5
 -1
 -1
 -1
 -1
```

Exercise #9: SpMV kernel for the ELL format

Remember to left-align each row in order to store by column the non-zero values of the sparse matrix, i.e.,

For the ELL format, the SpMV kernel works as follows on the left-justified non-zero values:

[43]:
```julia
function dellmv(A::SparseMatrixELL, x::Vector{Float64})
  y = zeros(A.m)
  for i in 1:A.m
    for j in 1:A.rownnz
      k = A.colval[(j - 1) * A.m + i]
      if k > 0
        y[i] += A.nzval[(j - 1) * A.m + i] * x[k]
      end
    end
  end
```

```
  return y
end
```

[43]: dellmv (generic function with 1 method)

[44]:
```
@btime dellmv(A_ell, x)
```

```
  30.180 ns (2 allocations: 96 bytes)
```

[44]:
```
5-element Vector{Float64}:
 0.9604101914079568
 0.24553763592463487
 0.6198029953005734
 0.3279669573558319
 0.34004500105563085
```