Roofline Model, Sparse Matrix Computation

Lecture 9

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4700/

The Roofline Model

Roofline Model

- Performance: locality, locality, locality
- Because computation is really really fast
- And moving data is expensive
- Limits on performance: Bandwidth, Latency, Peak Compute

$$\text{Performance} = \frac{\text{GFlop}}{\text{second}} = \min \begin{cases} \text{CPU Peak} \\ \text{Bandwidth} \end{cases}$$

Williams, Samuel, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." *Communications of the ACM* 52.4 (2009): 65-76.

Roofline Model

- Performance: locality, locality, locality
- Because computation is really really fast
- And moving data is expensive
- Limits on performance: Bandwidth, Latency, Peak Compute

$$\text{Performance} = \frac{\text{GFlop}}{\text{second}} = \min \begin{cases} \text{CPU Peak} & \frac{\text{GFlop}}{\text{second}} \\ \text{Bandwidth} & \frac{\text{Gbytes}}{\text{second}} \end{cases}$$

Roofline Model

- Performance: locality, locality, locality
- Because computation is really really fast
- And moving data is expensive
- Limits on performance: Bandwidth, Latency, Peak Compute

$$\text{Performance} = \frac{\text{GFlop}}{\text{second}} = \min \begin{cases} \text{CPU Peak } \frac{\text{GFlop}}{\text{second}} \\ \text{Bandwidth } \frac{\text{Gbytes}}{\text{second}} \times \frac{\text{GFlop}}{\text{Gbyte}} \end{cases}$$

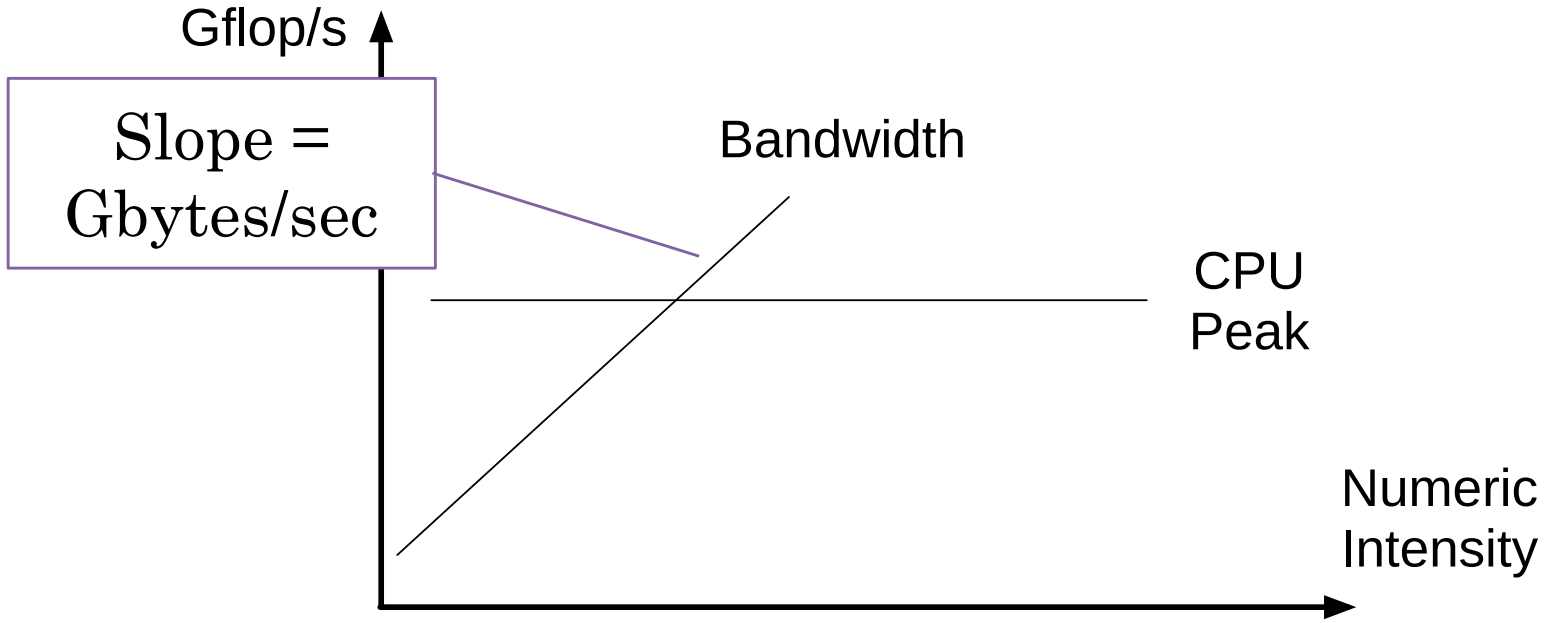
Roofline Model

- Performance: locality, locality, locality
- Because computation is really really fast
- And moving data is expensive
- Limits on performance: Bandwidth, Latency, Peak Compute

$$\text{Performance} = \frac{\text{GFlop}}{\text{second}} = \min \begin{cases} \text{CPU Peak } \frac{\text{GFlop}}{\text{second}} \\ \text{Bandwidth } \frac{\text{Gbytes}}{\text{second}} \times \text{Numerical Intensity } \frac{\text{GFlop}}{\text{Gbyte}} \end{cases}$$

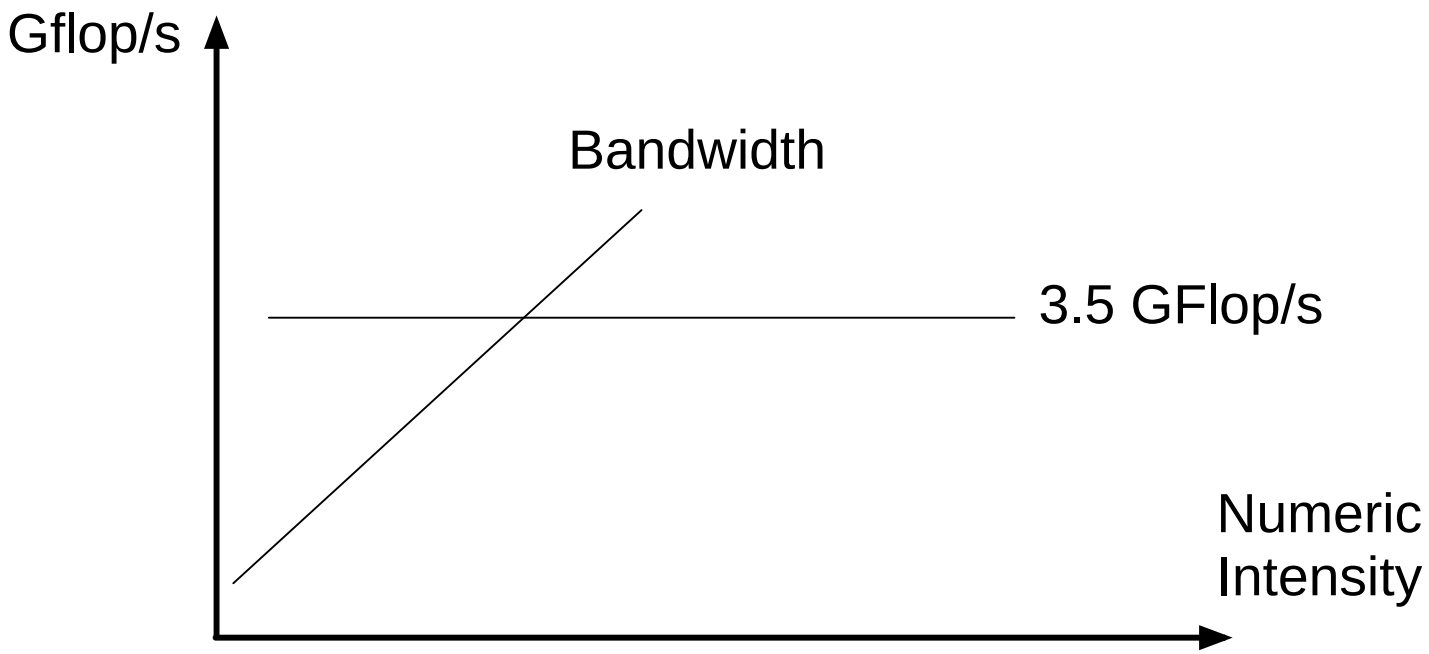
Roofline Model

$$\text{Performance} = \frac{\text{GFlop}}{\text{second}} = \min \begin{cases} \text{CPU Peak } \frac{\text{GFlop}}{\text{second}} \\ \text{Bandwidth } \frac{\text{Gbytes}}{\text{second}} \times \text{Numerical Intensity } \frac{\text{GFlop}}{\text{Gbyte}} \end{cases}$$



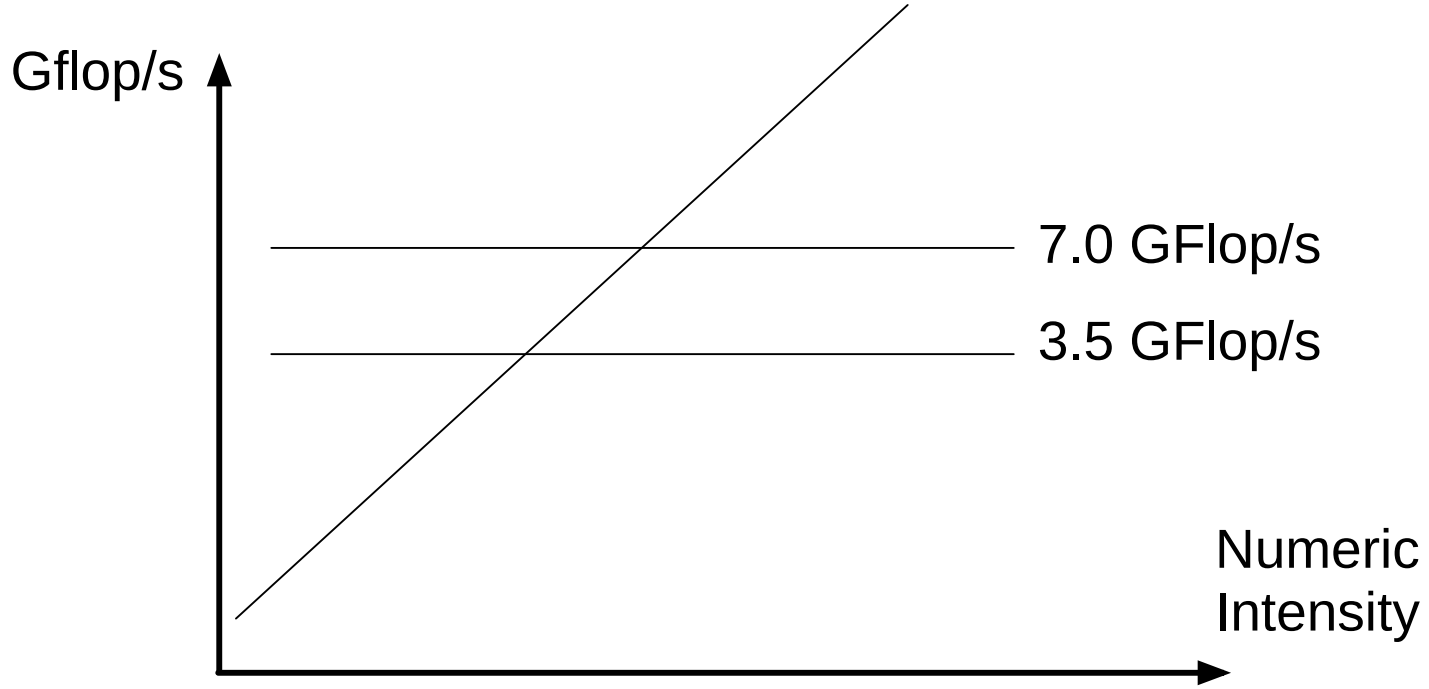
Roofline Model

- Single scalar core



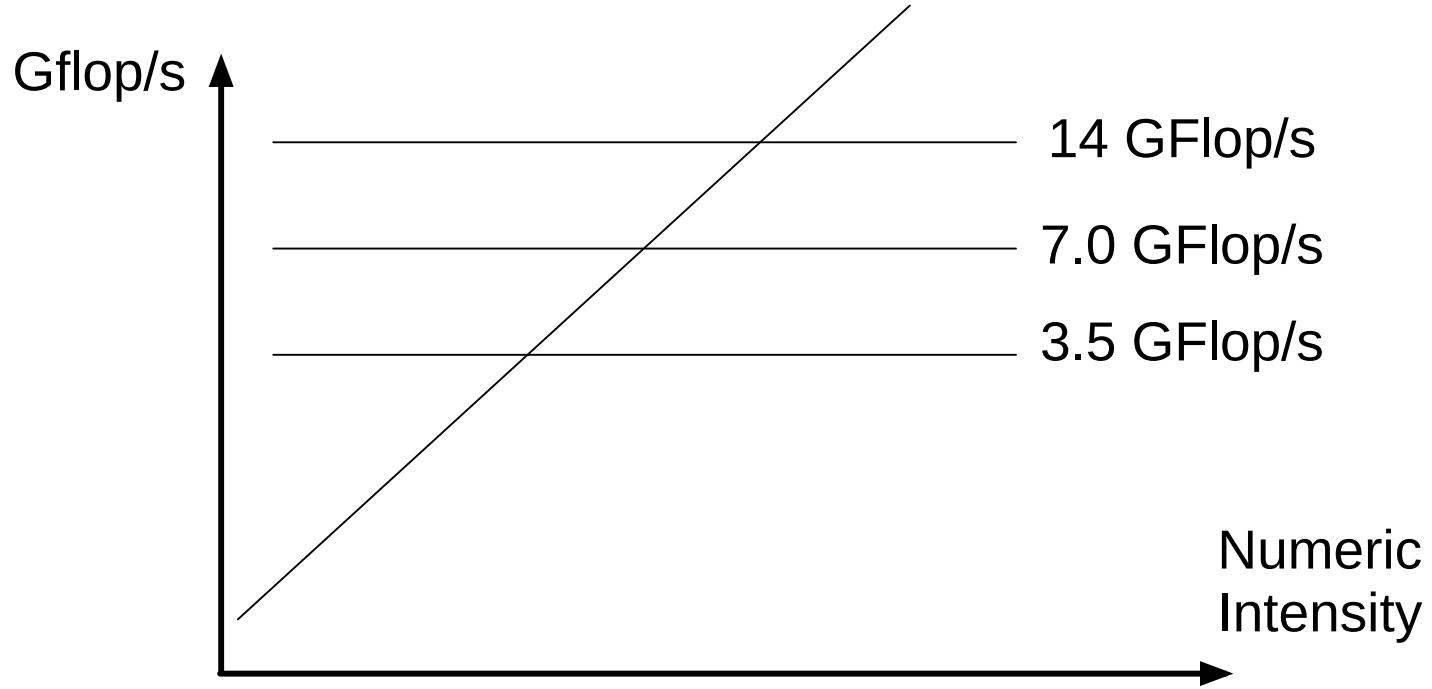
Roofline Model

- SSE single core (vectorize using 128 bit registers)



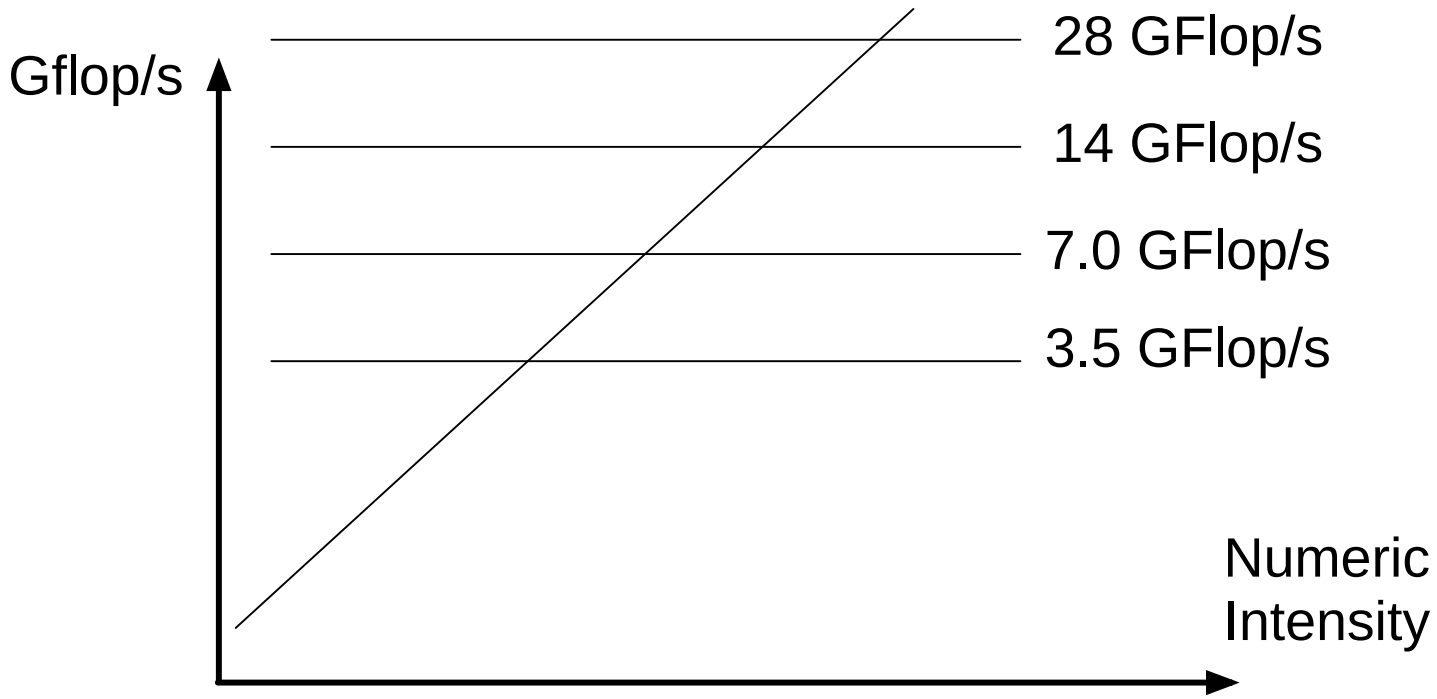
Roofline Model

- AVX single core (vectorize using 256 bit registers)



Roofline Model

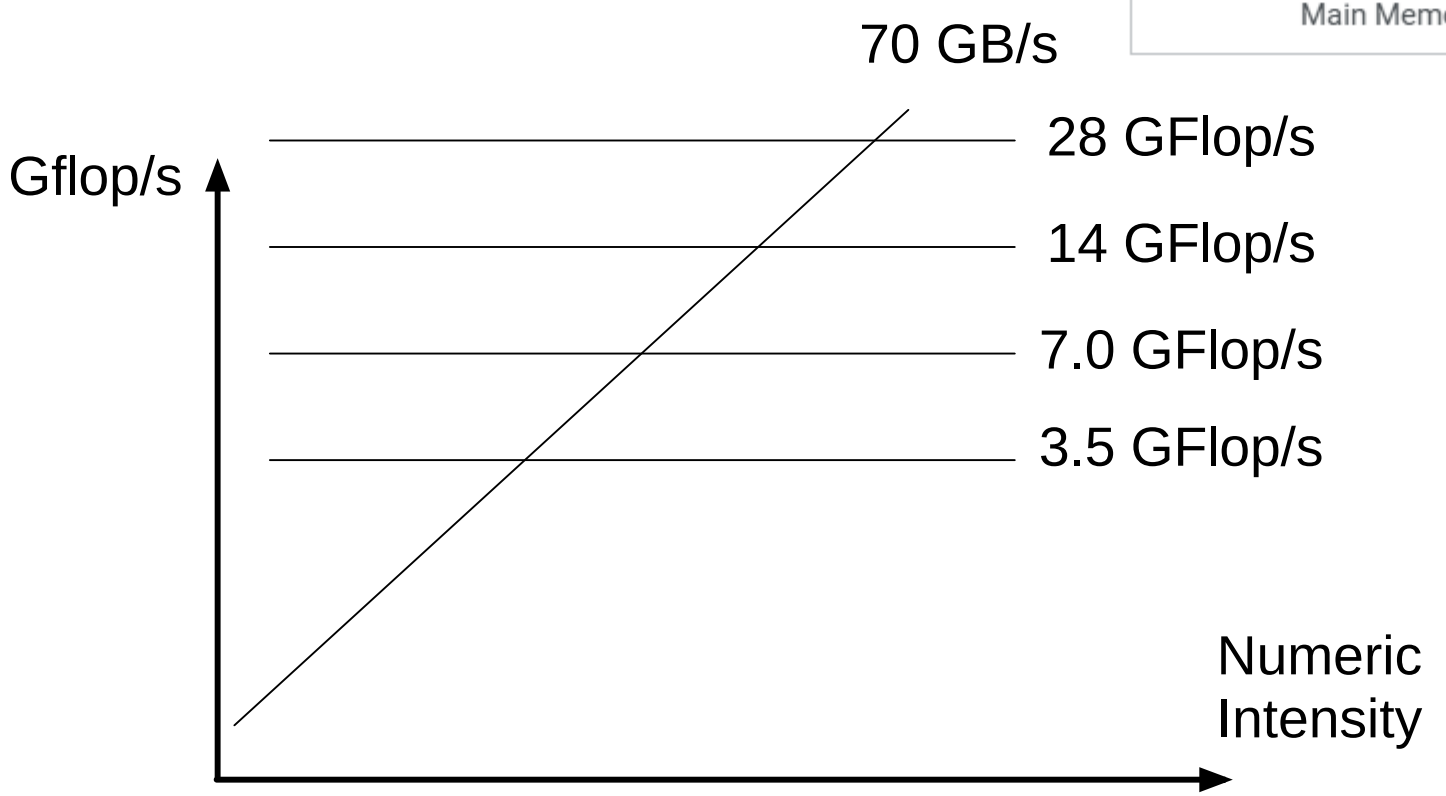
- AVX single core, fused multiply-add (FMA)



Roofline Model

- L2 Cache

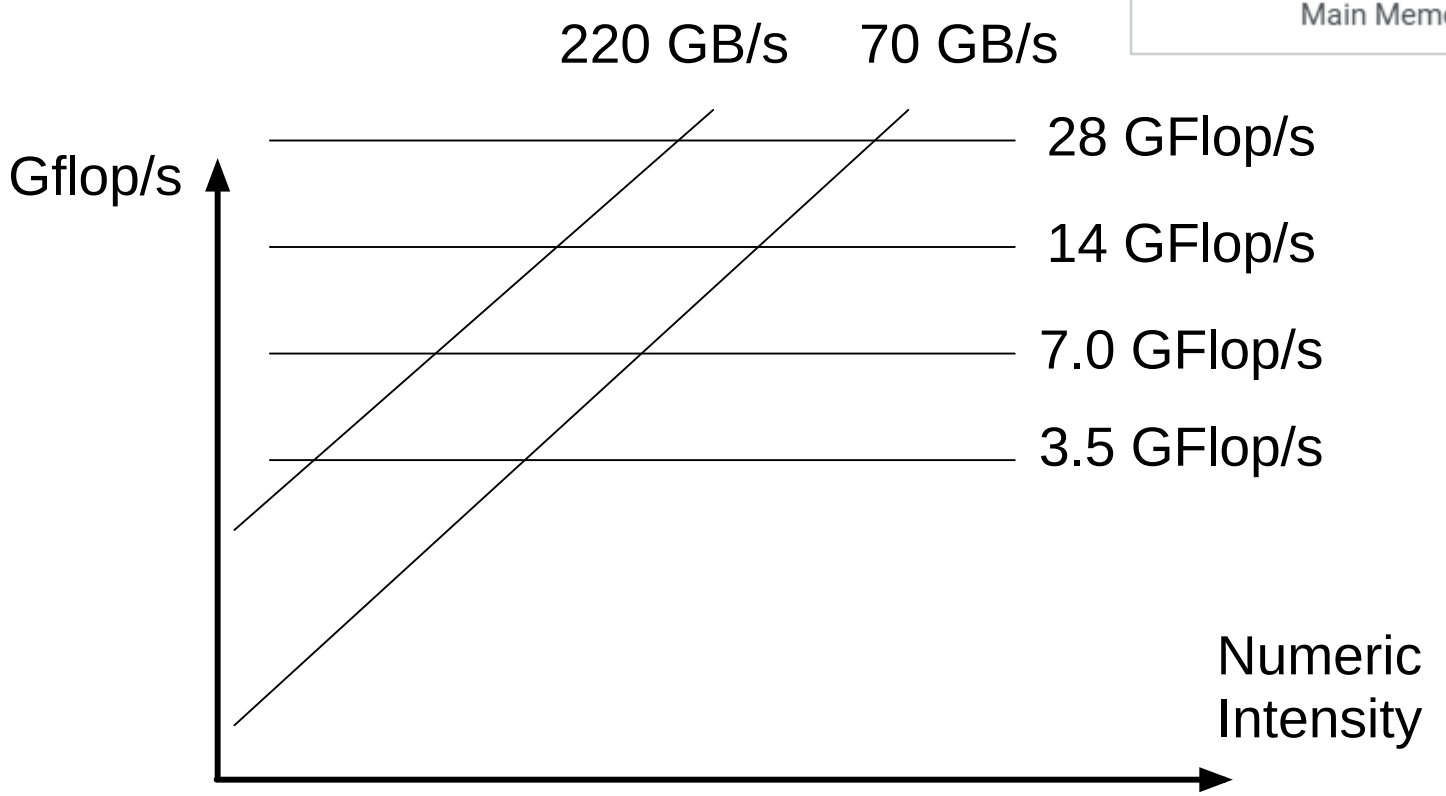
Memory Location	CPU Cycles
Register	1
L1 Cache	~4
L2 Cache	~14
L3 Cache	~75
Main Memory	~200



Roofline Model

- L1 Cache

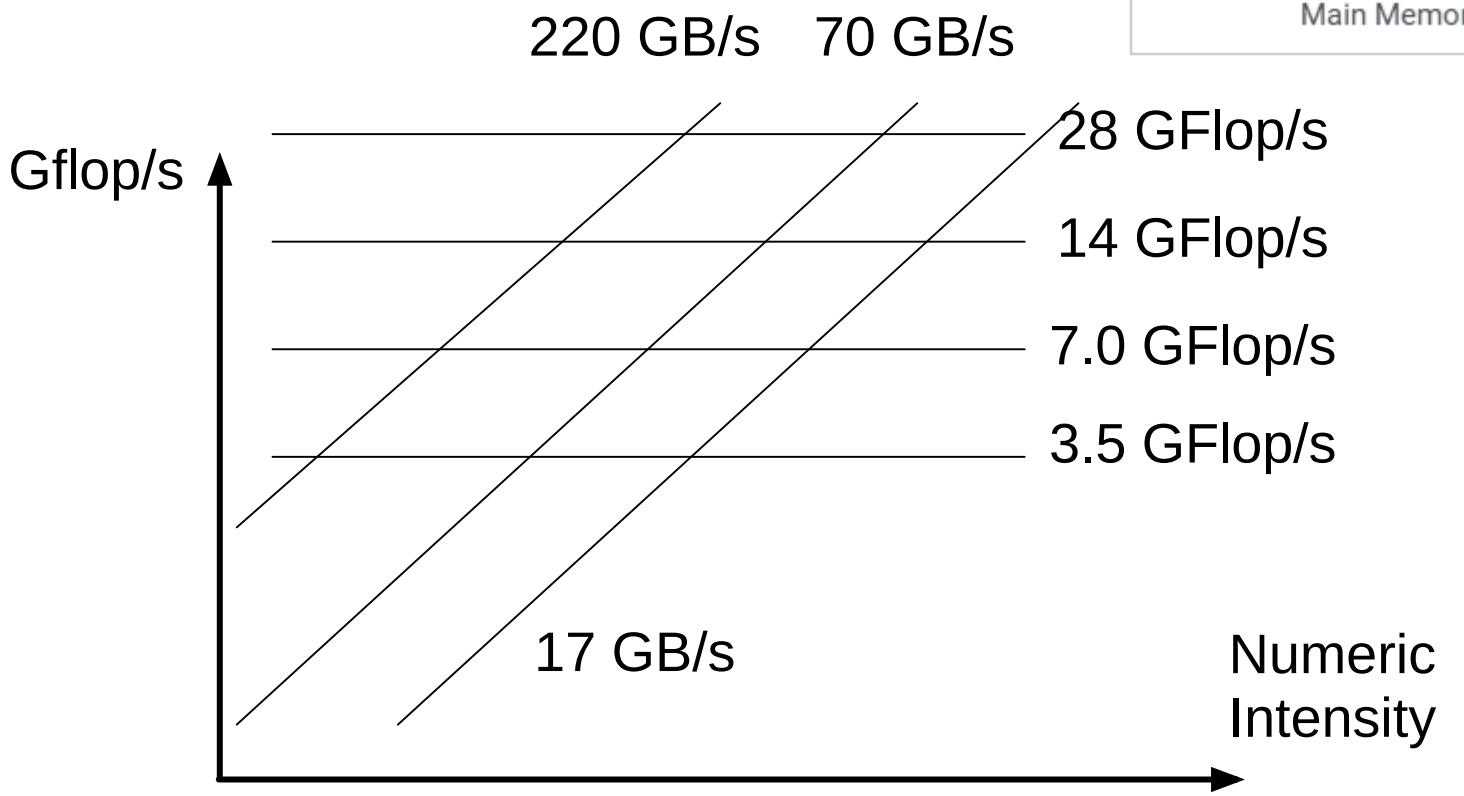
Memory Location	CPU Cycles
Register	1
L1 Cache	~4
L2 Cache	~14
L3 Cache	~75
Main Memory	~200



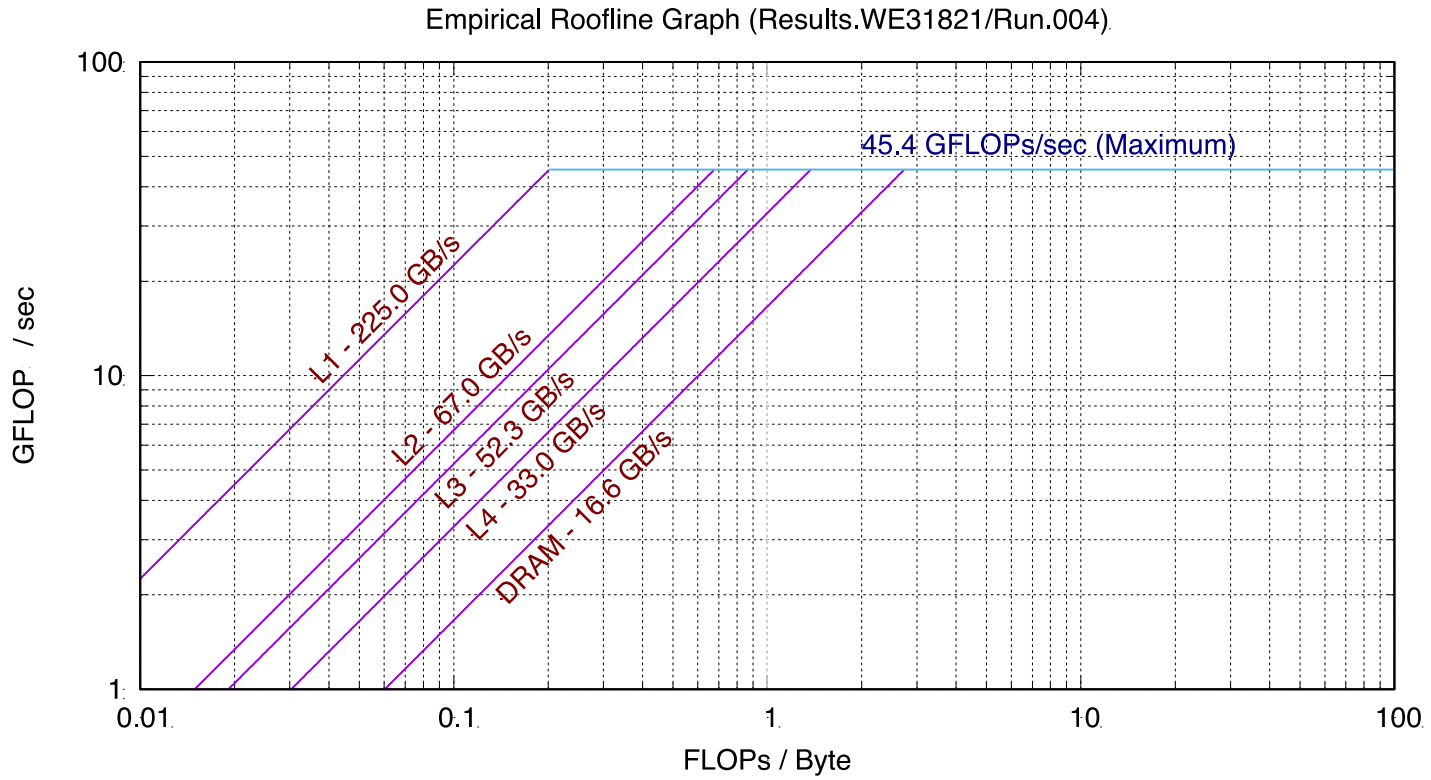
Roofline Model

- DRAM (main memory)

Memory Location	CPU Cycles
Register	1
L1 Cache	~4
L2 Cache	~14
L3 Cache	~75
Main Memory	~200



Measured


Empirical Roofline Graph (Results.WE31821/Run.004)
100
10
1
GFLOP / sec
0.01
0.1
1
10
100
FLOPs / Byte
45.4 GFLOPs/sec (Maximum)
L1 - 225.0 GB/s
L2 - 67.0 GB/s
L3 - 52.3 GB/s
L4 - 33.0 GB/s
DRAM - 16.6 GB/s

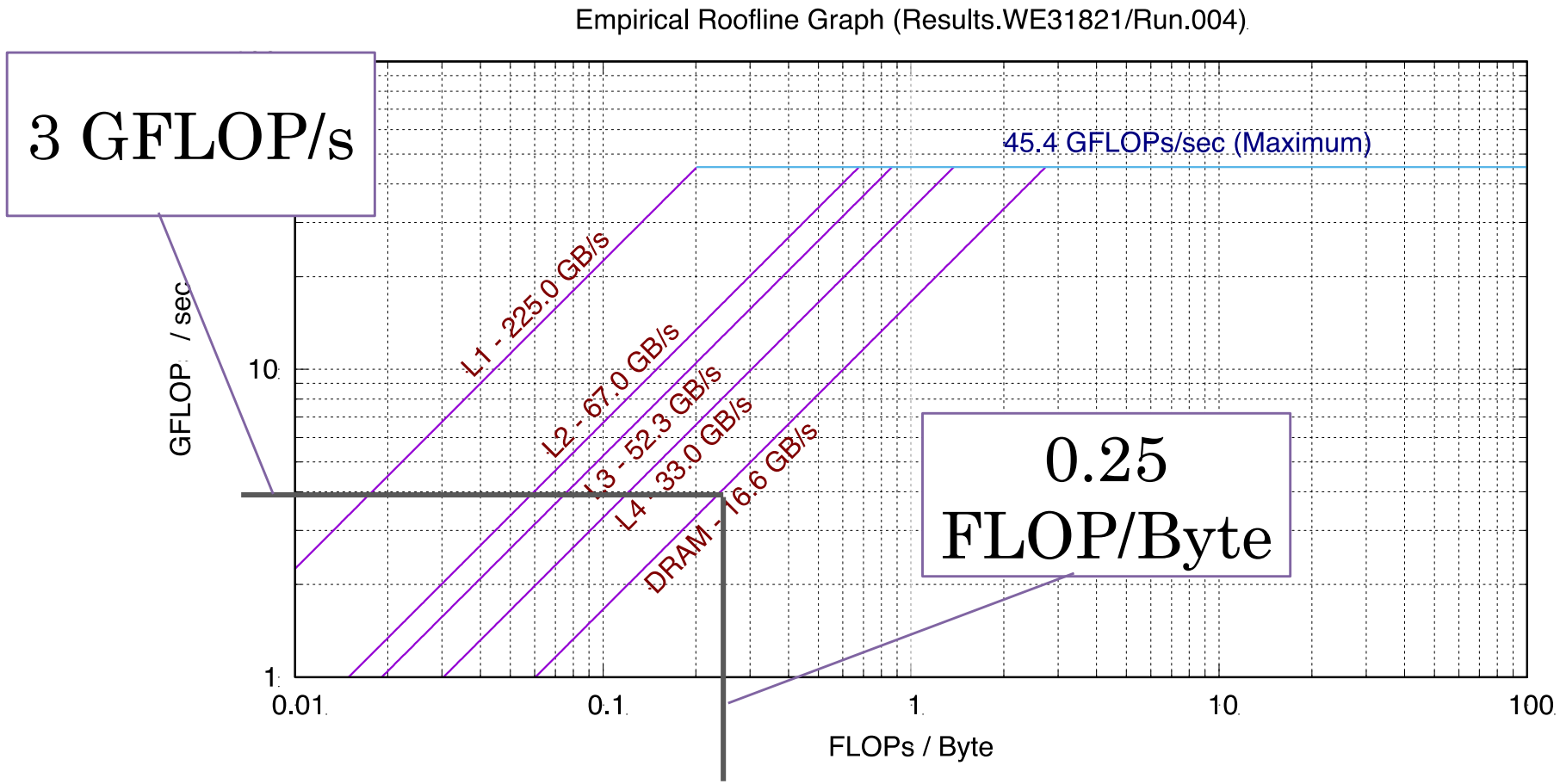
Numerical Intensity

```
void matvec_dense(Matrix const& A, Vector const& x, Vector& y)
{
    for (size_t i = 0; i != A.num_rows(); ++i)
        for (size_t j = 0; j != A.num_cols(); ++j)
            y(i) += A(i, k) * x(j);
}
```

- Inner loop:
  - Two FLOP (one add, one multiplication)
  - Three doubles moved in/out of memory = 24 bytes
- Overall:
  - $2N^2$ FLOP
  - $N^2 + 2N$ doubles = $8(N^2 + 2N)$ bytes

$$\frac{1}{4}\ \frac{FLOP}{Byte}$$

Measured


Empirical Roofline Graph (Results.WE31821/Run.004)
3 GFLOP/s
45.4 GFLOPs/sec (Maximum)
L1 - 225.0 GB/s
L2 - 67.0 GB/s
L3 - 52.3 GB/s
L4 - 33.0 GB/s
DRAM - 16.6 GB/s
0.25 FLOP/Byte
GFLOP / sec
10
1
0.01
0.1
1
10
100
FLOPs / Byte

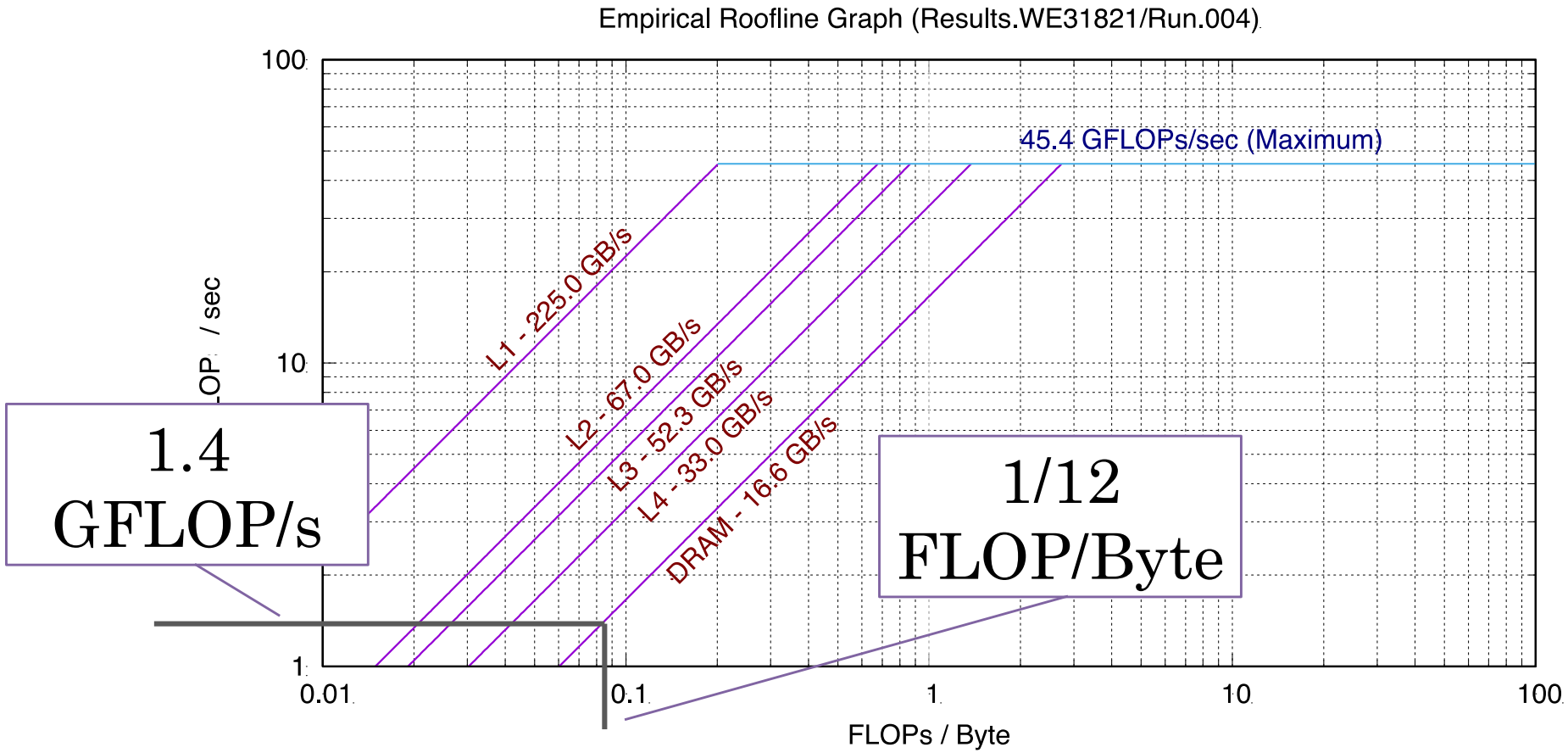
No Reuse: Memory-bound

```
void matmat_dense(Matrix const& A, Matrix const& B, Matrix& C)
{
    for (size_t i = 0; i != A.num_rows(); ++i)
        for (size_t j = 0; j != B.num_cols(); ++j)
            for (size_t k = 0; k != A.num_cols(); ++k)
                C(i, j) = A(i, k) * B(k, j);
}
```

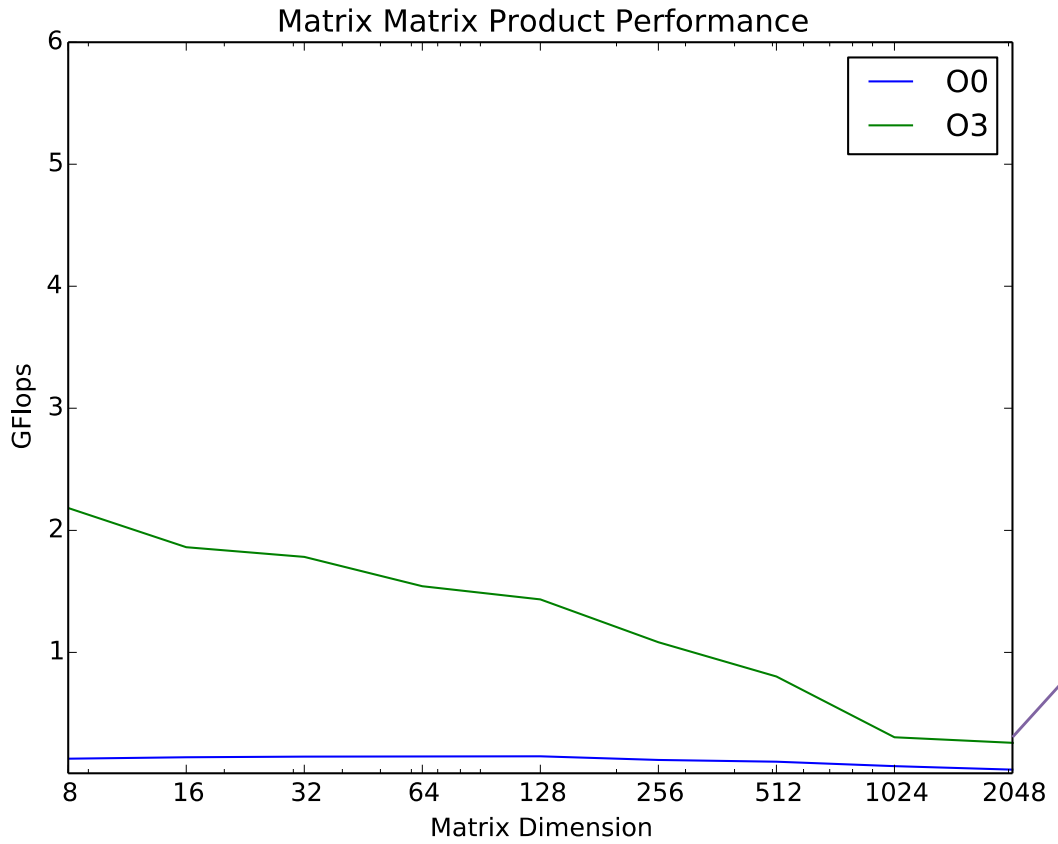
- Inner loop:
  - Two FLOP (one add, one multiplication)
  - Three doubles moved in/out of memory = 24 bytes
- Overall:
  - $2N^3$ FLOP
  - $3N^3$ doubles = $24N^3$ bytes

$$\frac{1}{12}\ \frac{FLOP}{Byte}$$

No Reuse: Memory-bound



No Reuse: Memory-bound



0.4 GFLOP/s

Full reuse: Compute-bound

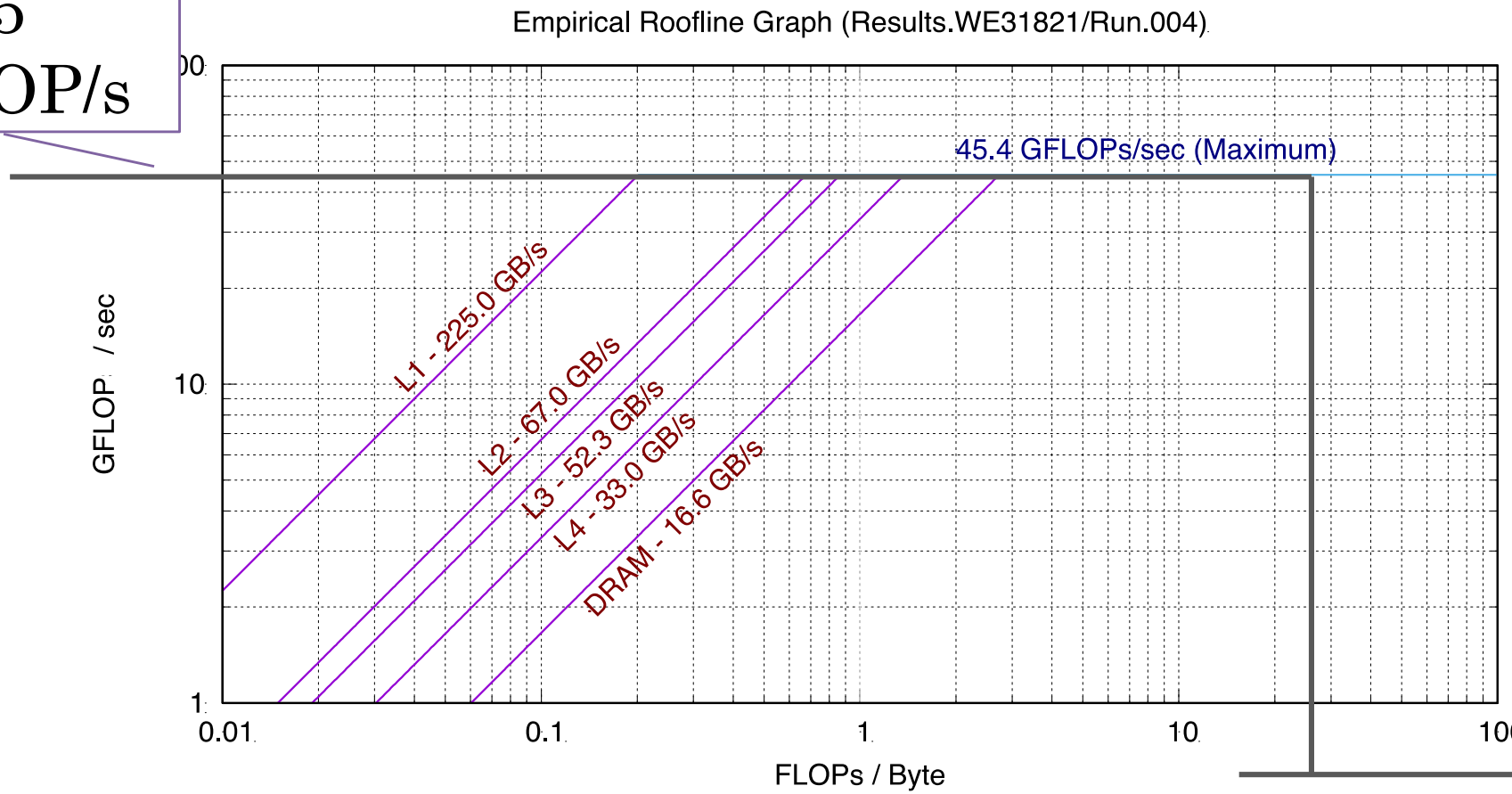
```
void matmat_dense(Matrix const& A, Matrix const& B, Matrix& C)
{
    for (size_t i = 0; i != A.num_rows(); ++i)
        for (size_t j = 0; j != B.num_cols(); ++j)
            for (size_t k = 0; k != A.num_cols(); ++k)
                C(i, j) = A(i, k) * B(k, j);
}
```

- Inner loop:
  - Two FLOP (one add, one multiplication)
  - Three doubles moved in/out of memory = 24 bytes
- Overall:
  - $2N^3$ FLOP
  - Full reuse: $3N^2$ doubles = $24N^2$ bytes

$$\frac{N}{12}\ \frac{FLOP}{Byte}$$

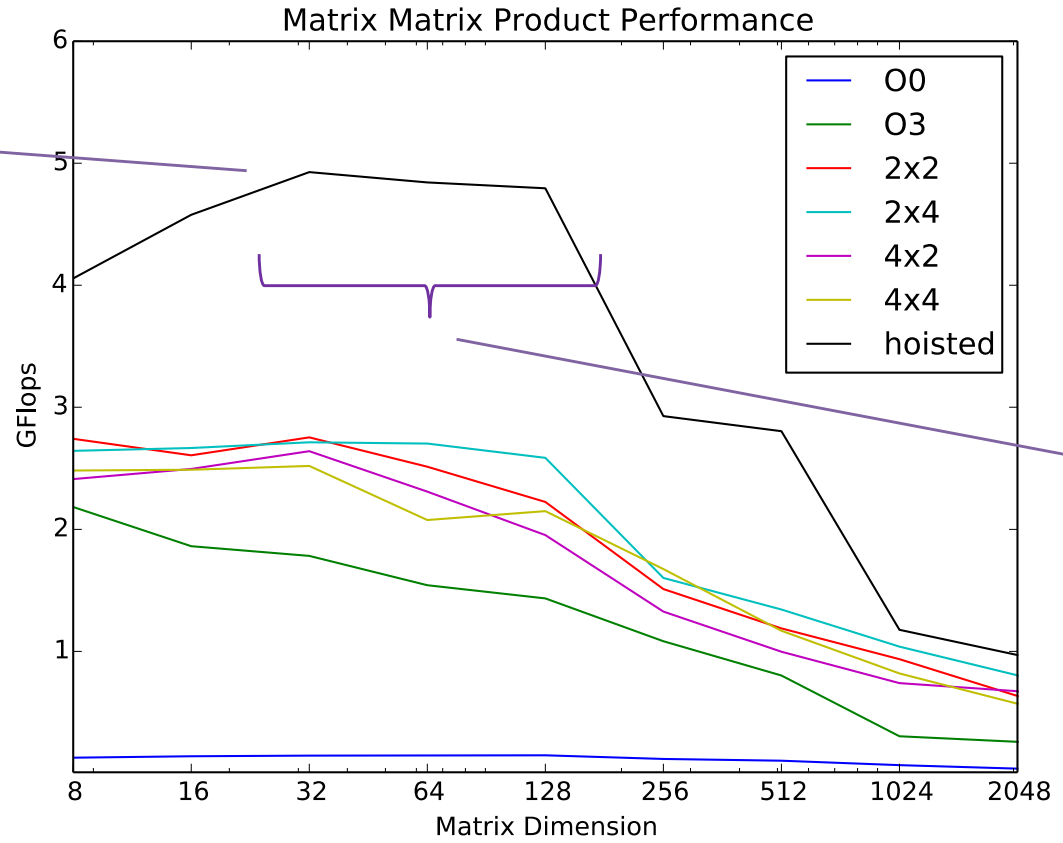
Full reuse: Compute-bound

45 GFLOP/s



Hoisting / Unrolling etc.: L2

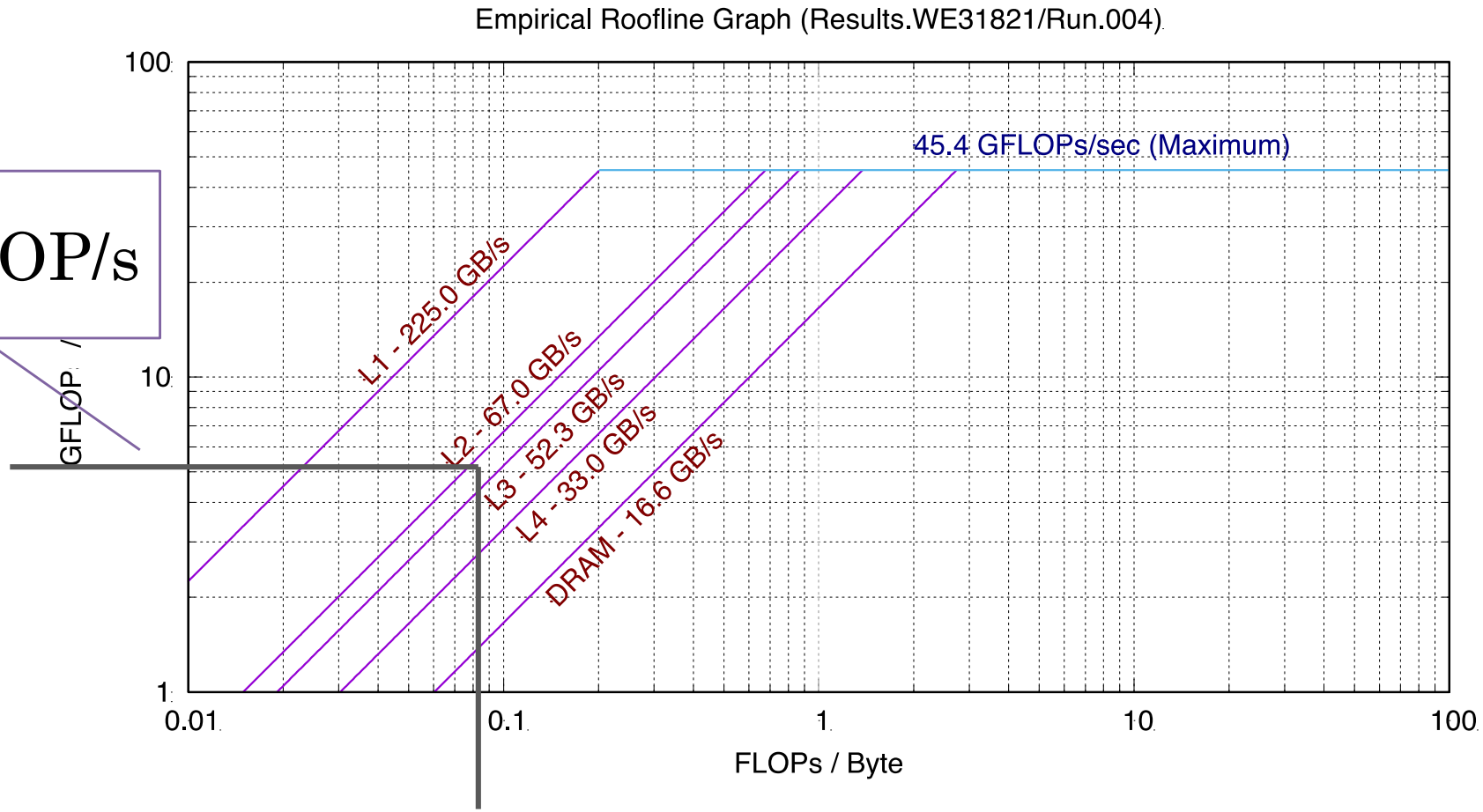
5 GFLOP/s



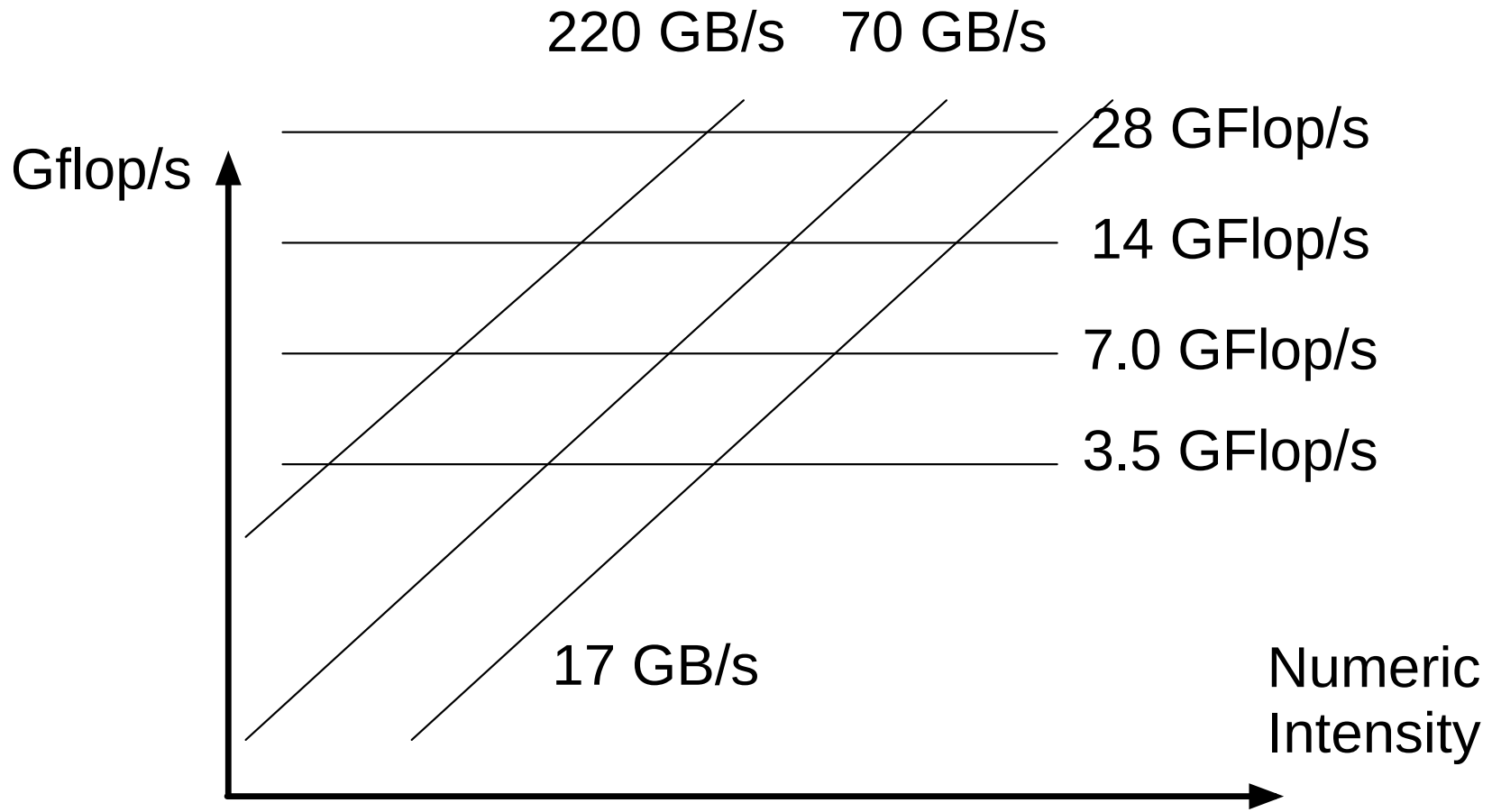
L2 zone

Hoisting / Unrolling etc.: L2

5 GFLOP/s



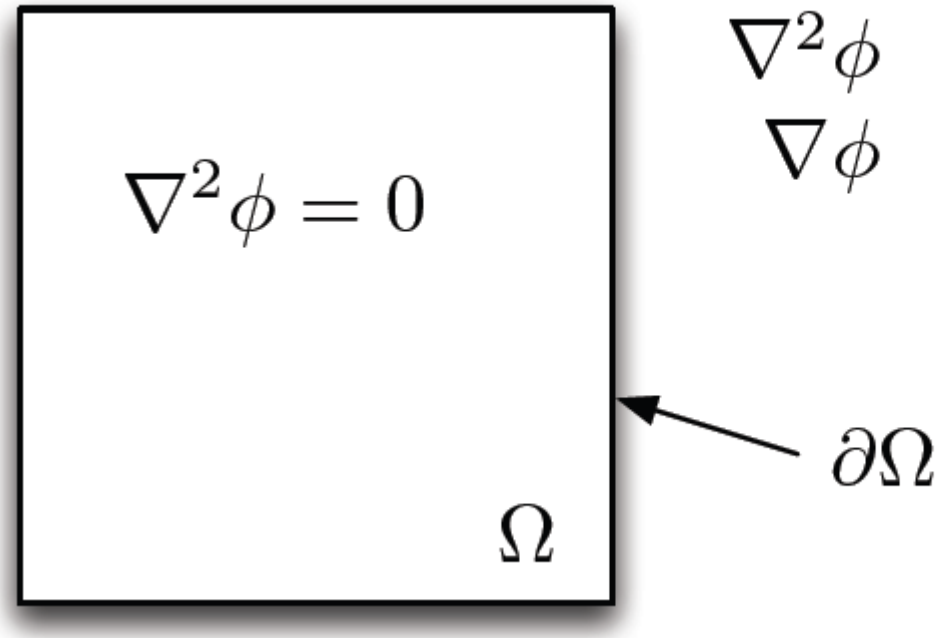
Summary (Roofline Model)



Sparse Matrices

In Practice

- Many scientific applications are based on solving systems of partial differential equations that model physical phenomena
- Laplace's equation on unit square is prototypical PDE



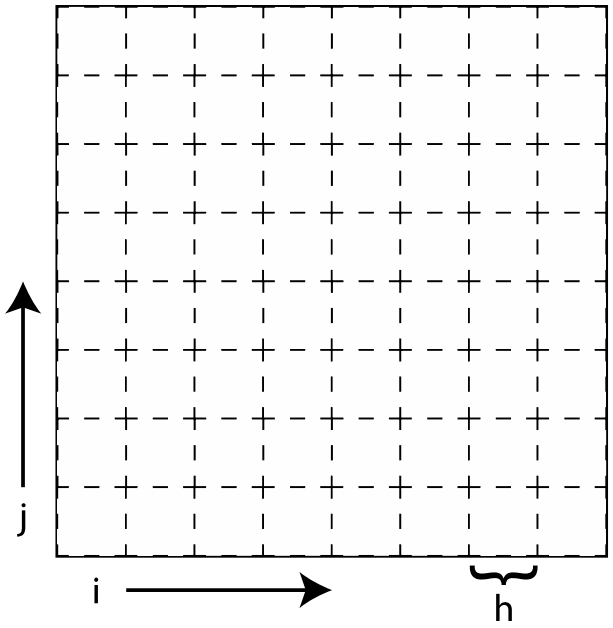
$$\begin{aligned} \nabla^2\phi &= 0 \quad \text{on } \Omega \\ \nabla\phi &= f \quad \text{on } \partial\Omega \end{aligned}$$

Discretized

- Finite difference approximation to derivative

$$\begin{aligned}\nabla\phi &= \frac{\partial\phi}{\partial x} + \frac{\partial\phi}{\partial y} \\ \nabla^2\phi &= \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2}\end{aligned}$$

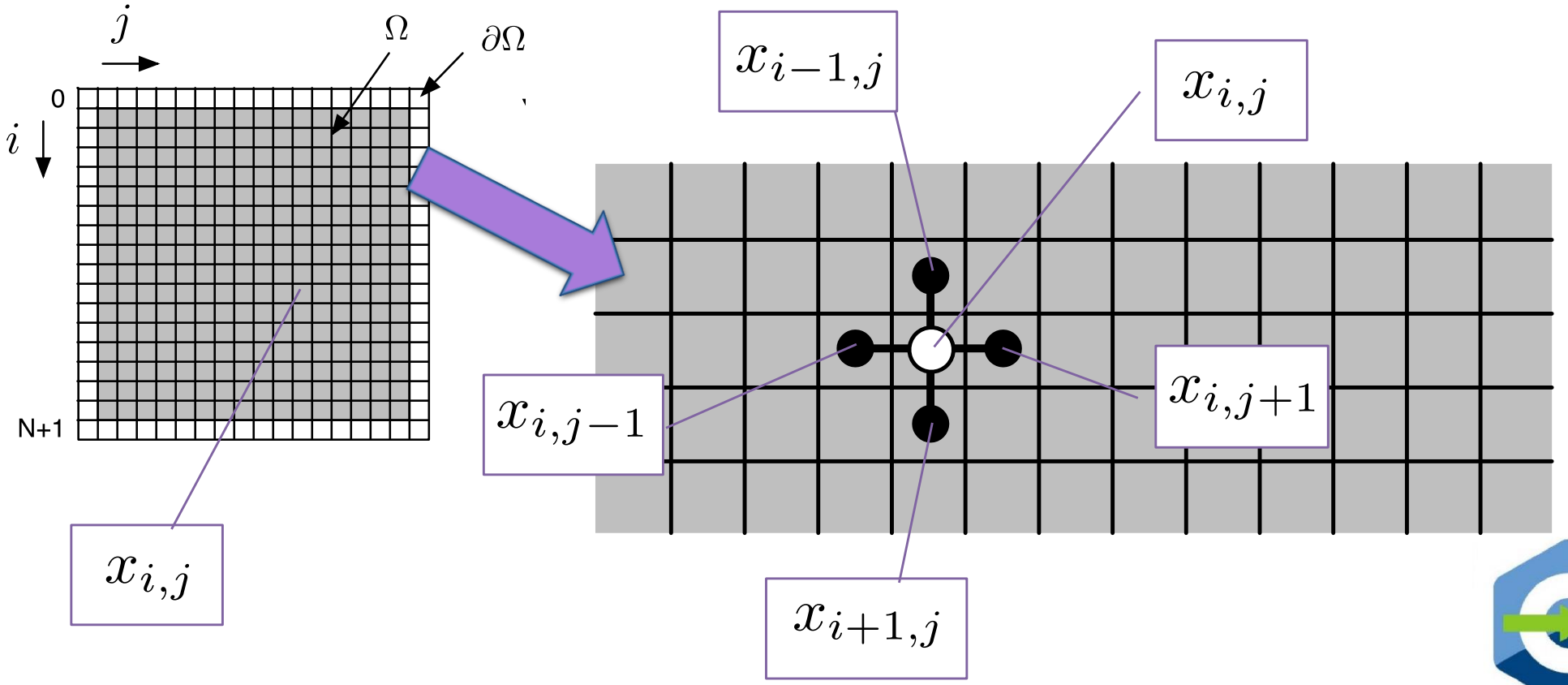
$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t}(t_0) &\approx \frac{x(t_0+h)-x(t_0)}{h} \\ \frac{\mathrm{d}^2x}{\mathrm{d}t^2}(t_0) &\approx \frac{\frac{\mathrm{d}x}{\mathrm{d}t}(t_0+h)-\frac{\mathrm{d}x}{\mathrm{d}t}(t_0)}{h} \\ &= \frac{x(t_0+h+h)-x(t_0+h)-x(t_0+h)+x(t_0)}{h^2} \\ &= \frac{x(t_0+2h)-2x(t_0+h)+x(t_0)}{h^2} \\ &= \frac{x(t_0+h)-2x(t_0)+x(t_0-h)}{h^2}\end{aligned}$$



$$\frac{\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j-1} + \phi_{i,j+1} - 4\phi_{i,k}}{h^2} = 0$$

Laplace's Equation on a Regular Grid

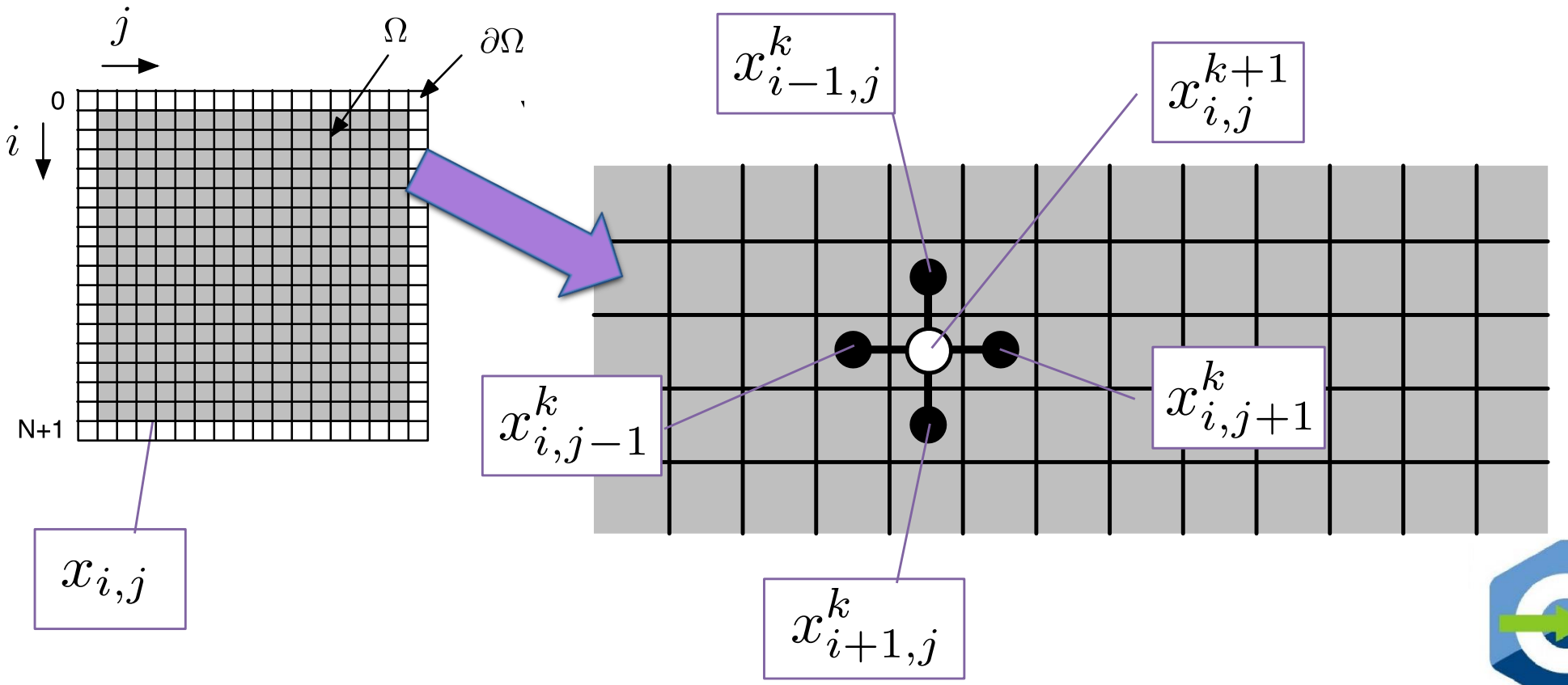
$$x_{i,j} = (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})/4$$



Iterating for a Solution

Timesteps

$$x_{i,j}^{k+1} = (x_{i-1,j}^{k} + x_{i+1,j}^{k} + x_{i,j-1}^{k} + x_{i,j+1}^{k})/4$$



Iterating for a Solution

- Only need to use two arrays: old and new
  - Array x: average of approximation at iteration k
  - Array y: average of approximation at iteration k + 1

```
while (!converged())
{
    for (size_t i = 1; i != N + 1; ++i)
        for (size_t j = 1; j != N + 1; ++j)
            y(i, j) = (
                    x(i - 1, j    ) +
                    x(i + 1, j    ) +
                    x(i,     j - 1) +
                    x(i,     j + 1)
                ) / 4.0;
    swap(x, y);
}
```

- At end of each outer iteration: new becomes old (and v.v.)

Matrix Formulation

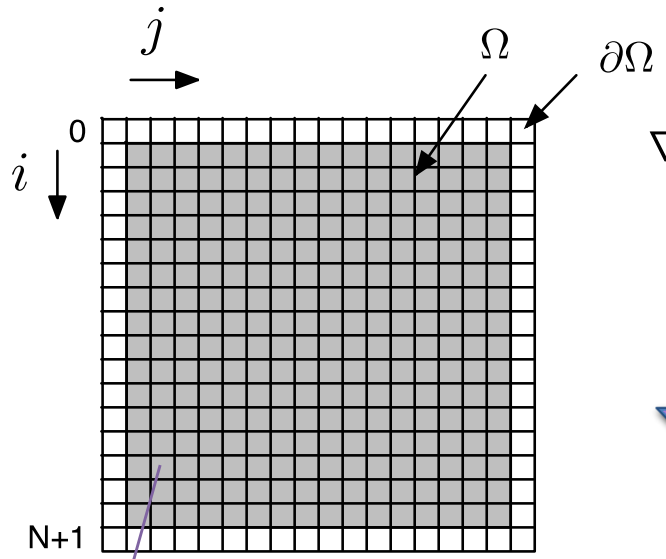
- Lexicographically order unknowns (note some will be boundary values), get system of linear equations, one for each grid point:

$$\frac{x_{i+1} + x_{i-1} + x_{i+N} + x_{i-N} - 4x_i}{h^2} = 0 \qquad i \in [0, N^2)$$

- Formulate as a matrix problem:
- Laplacian matrix
  - Diagonal matrix
  - Many zero elements

$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & \cdots & -1 & & 0 \\ -1 & \ddots & \ddots & \ddots & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ -1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & \ddots & \ddots & -1 \\ 0 & & -1 & \cdots & -1 & 4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \\ \\ \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ \\ \\ \end{bmatrix}$$

Laplace's Equation on a Regular Grid



$$\begin{aligned} \nabla^2\phi &= 0 \quad \text{on } \Omega \\ \phi &= f \quad \text{on } \partial\Omega \end{aligned}$$

$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & \cdots & -1 & & 0 \\ -1 & \ddots & \ddots & \ddots & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ -1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & \ddots & \ddots & -1 \\ 0 & & -1 & \cdots & -1 & 4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ \end{bmatrix}$$

Discretization

$$x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1} - 4x_{i,j} = 0$$

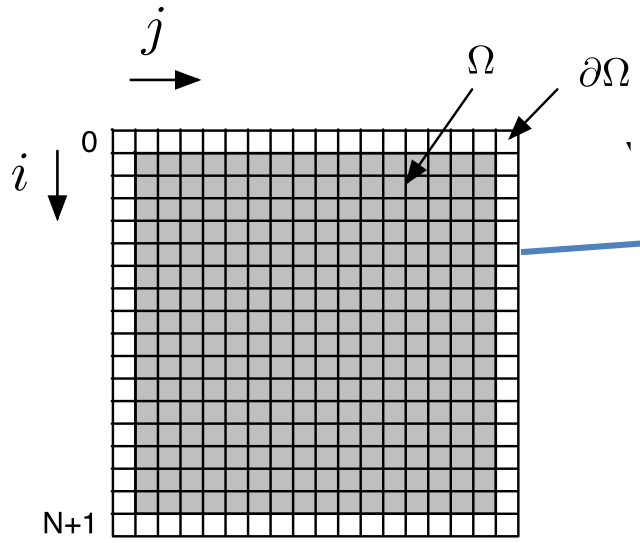
$$x_{i,j} = (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})/4$$

$x_{i,j}$

The value of each point on the grid

The average of its neighbors in 4 directions

Laplace's Equation on a Regular Grid



$$x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1} - 4x_{i,j} = 0$$

$$x_{i,j} = (x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1})/4$$

- Why isn't 0 the solution?
- The boundary is non-zero
- Non-zeros in here due to boundary

$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & \cdots & -1 & & \\ -1 & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ -1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & \ddots & \ddots & -1 \\ 0 & & -1 & \cdots & -1 & 4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \\ \\ \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ \\ \\ \end{bmatrix}$$

Linear System Solution

Linear System Solution

- Matrix-matrix product is the essential kernel operation

```
void multiply(Matrix const& A, Matrix const& B, Matrix& C)
{
    for (size_t i = 0; i != A.num_rows(); ++i)
        for (size_t j = 0; j != B.num_cols(); ++j)
            for (size_t k = 0; k != A.num_cols(); ++k)
                C(i, j) = A(i, k) * B(k, j);
}
```

- What happens with the Laplacian matrix?
  - Multiplying and adding zero to zero (partially)
- Work Smarter! Don't multiply and add zero to zero!

Solution?

- Let's Avoid zeros!

```
void multiply(Matrix const& A, Matrix const& B, Matrix& C)
{
    for (size_t i = 0; i != A.num_rows(); ++i)
        for (size_t j = 0; j != B.num_cols(); ++j)
            for (size_t k = 0; k != A.num_cols(); ++k)
                if (A(i, k) != 0.0 && B(k, j) != 0.0)
                    C(i, j) = A(i, k) * B(k, j);
}
```

- But we still touch every element
  - And that's what expensive!
- We need to avoid zeros, but without looking to see if there is a zero

Solution: Sparse Matrices

- In order to avoid zeros, just don't store them!
  - A zero is a null op after all
- Use data structures and algorithms accordingly
- Sparse matrix techniques

$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & \cdots & -1 & & 0 \\ -1 & \ddots & \ddots & \ddots & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ -1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & \ddots & \ddots & -1 \\ 0 & & -1 & \cdots & -1 & 4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \\ \\ \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ \\ \\ \end{bmatrix}$$

Solving Sparse Systems

- Work only with non-zeros
- Direct methods
  - Perform LU factorization on sparse matrix
  - Create non-zeros during elimination process
  - Pre-order (using heuristics) to minimize the amount of fill
  - Very sequential
  - Fill can be quite significant
    - Expensive as space needs to be created!
- Iterative methods
  - Successively create better approximations to x
  - Relaxation methods (e.g., Jacobi) – very very very slow to converge
  - Krylov subspace methods (e.g., conjugate gradient)
    - Good preconditioning often required

Aside: Conjugate Gradient Algorithm

Initial $r^{(0)}=b-Ax^{(0)}$
For $i=1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = r^{(i-1)\top} z^{(i-1)}$
  If $i=1$
    $p^{(1)} = z^{(0)}$
  Else
    $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  Endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1} / p^{(i)\top} q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  Check convergence
end

```
mult(A, scaled(x, -1.0), b, r);
while (! iter.finished(r)) {
  solve(M, r, z);
  rho = dot_conj(r, z);

  if ( iter.first() )
    copy(z, p);
  else {
    beta = rho / rho_1;
    add(z, scaled(p, beta), p);
  }
  mult(A, p, q);
  alpha = rho / dot_conj(p, q);
  add(x, scaled(p, alpha), x);
  add(r, scaled(q, -alpha), r);
  rho_1 = rho;
  ++iter;
}
```

Sparse Storage

- A matrix is map from two indices to a value

$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & \cdots & -1 & & 0 \\ -1 & \ddots & \ddots & \ddots & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ -1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & \ddots & \ddots & -1 \\ 0 & & -1 & \cdots & -1 & 4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \\ \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ \\ \end{bmatrix}$$

- So if we want to store just elements that are not zero (the "non-zeros")
- We need to store the two indices and the value

Dense Storage

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

- Dense storage:
  - All matrix elements are kept at the location corresponding to their indices

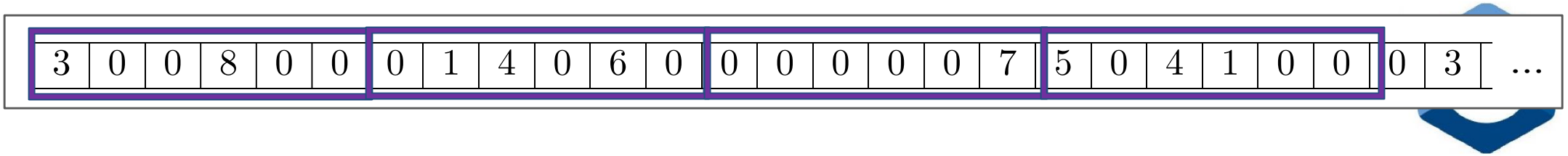
3	0	0	8	0	0	0	1	4	0	6	0	0	0	0	0	0	7	5	0	4	1	0	0	0	3	…

Dense Storage

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

```
void multiply(
    Matrix const& A, Vector const& x, Vector& y)
{
    for (size_t i = 0; i != A.num_rows(); ++i)
        for (size_t j = 0; j != A.num_cols(); ++j)
            y(i) = A(i, k) * x(j);
}
```

- We go through all possible valid indices
  - And thus all values of A
  - Zeros and non-zeros
- With dense storage, we loop through all possible indices and look up corresponding value



Sparse Storage

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

```
void multiply(
    Matrix const& A, Vector const& x, Vector& y)
{
    for (size_t i = 0; i != A.num_rows(); ++i)
        for (size_t j = 0; j != A.num_cols(); ++j)
            y(i) = A(i, k) * x(j);
}
```

- Store only the non-zeros
- Goal: Loop over all indices for non-zero entries
  - But what is non-zero is a property of matrix
  - Algorithm can't know it
  - So we need to store indices also

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

Sparse Storage

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

```
void multiply(
    Matrix const& A, Vector const& x, Vector& y)
{
    for (size_t i = 0; i != A.num_rows(); ++i)
        for (size_t j = 0; j != A.num_cols(); ++j)
            y(i) = A(i, k) * x(j);
}
```

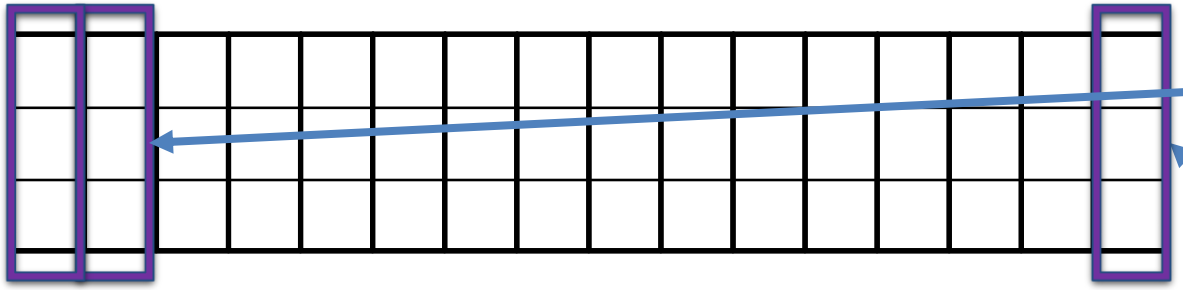
- Goal: Loop over all indices for non-zero entries
- Does order of elements matter?

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

0	0	1	1	1	2	3	3	3	4	4	5

0	3	1	2	4	5	0	2	3	1	4	5

Coordinate Storage (Array of Structs)

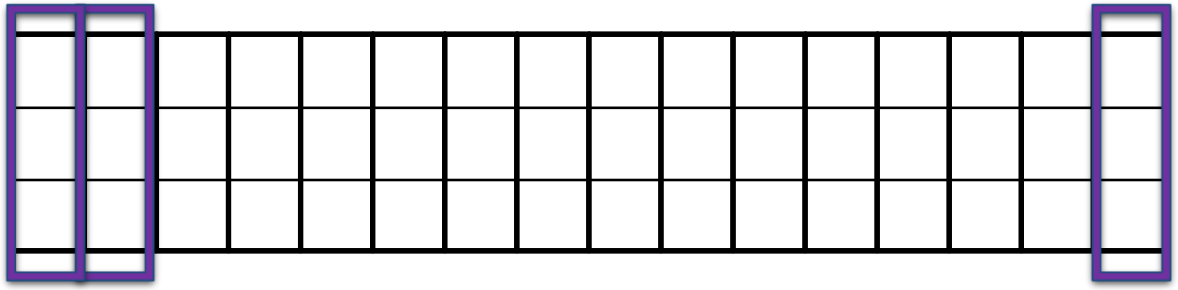


- Each element has two indices and a value stored

$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & \cdots & -1 & & \\ -1 & \ddots & \ddots & \ddots & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ -1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & \ddots & \ddots & -1 \\ & & -1 & \cdots & -1 & 4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \\ \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ \\ \end{bmatrix}$$

- Single array with 3-element structs
  - Struct contains two indices and a value

Coordinate Storage (Array of Structs)



$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & \cdots & -1 & & \\ -1 & \ddots & \ddots & \ddots & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ -1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & \ddots & \ddots & -1 \\ & & -1 & \cdots & -1 & 4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \\ \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ \\ \end{bmatrix}$$

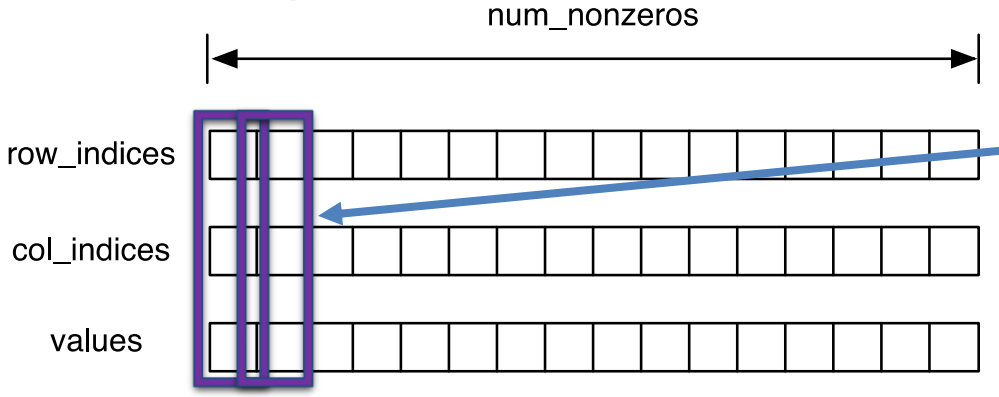
- Each element has two indices and a value stored

```
struct Element {
private:
  int row, col;
  double value;
};
```

- Single array with 3-element structs
  - Struct contains two indices and a value

```
struct COOMatrix {
private:
  std::vector<Element> arrayData;
};
```

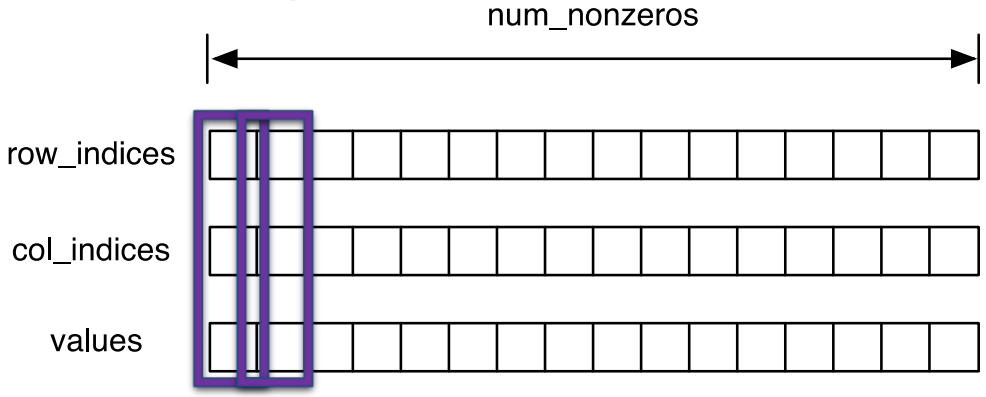
Coordinate Storage (Struct of Arrays)



- Each element has two indices and a value stored
- One array has row indices
- One array has column indices
- One array has element values

$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & \cdots & -1 & & \\ -1 & \ddots & \ddots & \ddots & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ -1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & \ddots & \ddots & -1 \\ & & -1 & \cdots & -1 & 4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \\ \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ \\ \end{bmatrix}$$

Coordinate Storage (Struct of Arrays)



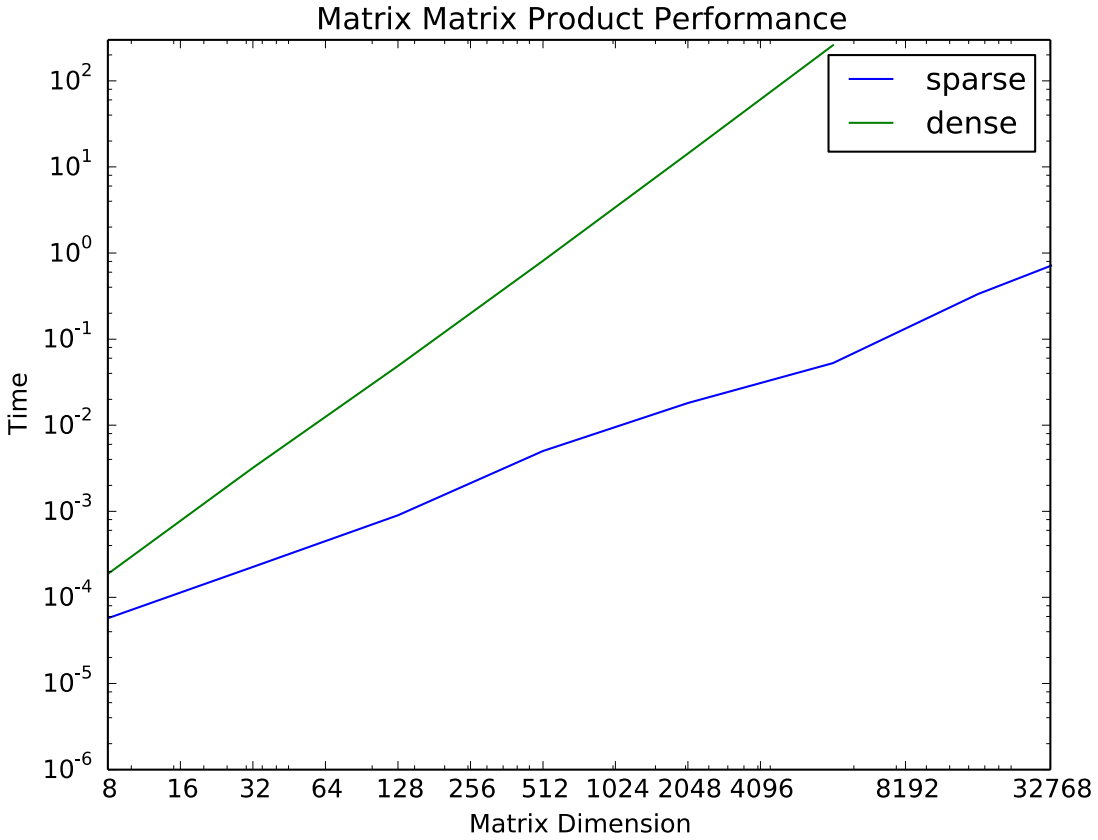
$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & \cdots & -1 & & \\ -1 & \ddots & \ddots & \ddots & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots & -1 \\ -1 & \ddots & \ddots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & \ddots & \ddots & -1 \\ & & -1 & \cdots & -1 & 4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \\ \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ \\ \end{bmatrix}$$

- One array has row indices
- One array has column indices
- One array has element values

```
struct COOMatrix {
private:
  std::vector<size_t> row_indices_;
  std::vector<size_t> col_indices_;
  std::vector<double> values_;
};
```

- Conventional Wisdom: Struct of Arrays is faster

Performance Comparison



What's the Catch?

- In the Matrix type we:
  - Provide indices, compute element in constant time
  - Return a reference, so we can modify the element

```
class Matrix {
public:
    Matrix(size_t M, size_t N)
      : num_rows_(M), num_cols_(N), storage_(num_rows_ * num_cols_)
    {}

    double& operator()(size_t i, size_t j) { return storage_[i * num_cols_ + j];}
    double const& operator()(size_t i, size_t j) const { return storage_[i * num_cols_ + j]; }

    size_t num_rows() const { return num_rows_; }
    size_t num_cols() const { return num_cols_; }

private:
    size_t num_rows_, num_cols_;
    std::vector<double> storage_;
};
```

Uh…

- How do we get to a value in constant time

```
class COOMatrix {
public:
    COOMatrix(size_t M, size_t N)
      : num_rows_(M), num_cols_(N)
    {}

    size_t num_rows() const { return num_rows_; }
    size_t num_cols() const { return num_cols_; }

private:
    size_t num_rows_, num_cols_;
    std::vector<size_t> row_indices_;
    std::vector<size_t> col_indices_;
    std::vector<double> storage_;
};
```

- We can't!

Next Problem…

- `Matrix`: nice external function using operator()():

```
void matvec(Matrix const& A, Vector const& x, Vector& y)
{
    for (size_t i = 0; i != A.num_rows(); ++i)
        for (size_t j = 0; j != A.num_cols(); ++j)
            y(i) = A(i, k) * x(j);
}
```

- `COOMatrix`: no operator()(), no external function:

```
void matvec(COOMatrix const& A, Vector const& x, Vector& y)
{
    for (size_t i = 0; i != A.num_rows(); ++i)
        for (size_t j = 0; j != A.num_cols(); ++j)
            // ???
}
```

Dense Matvec

```
void matvec(Matrix const& A, Vector const& x, Vector& y)
{
    for (size_t i = 0; i != A.num_rows(); ++i)
        for (size_t j = 0; j != A.num_cols(); ++j)
            y(i) = A(i, k) * x(j);
}
```

- We see that we need three items to properly access the data:
  - `y(i)`: here, `i` is the row index
  - `A(i, k)`: this represents the value
  - `x(j)`: here, `j` is the column index
- We have all these things in the coordinate format!

Coordinate Matrix Matvec

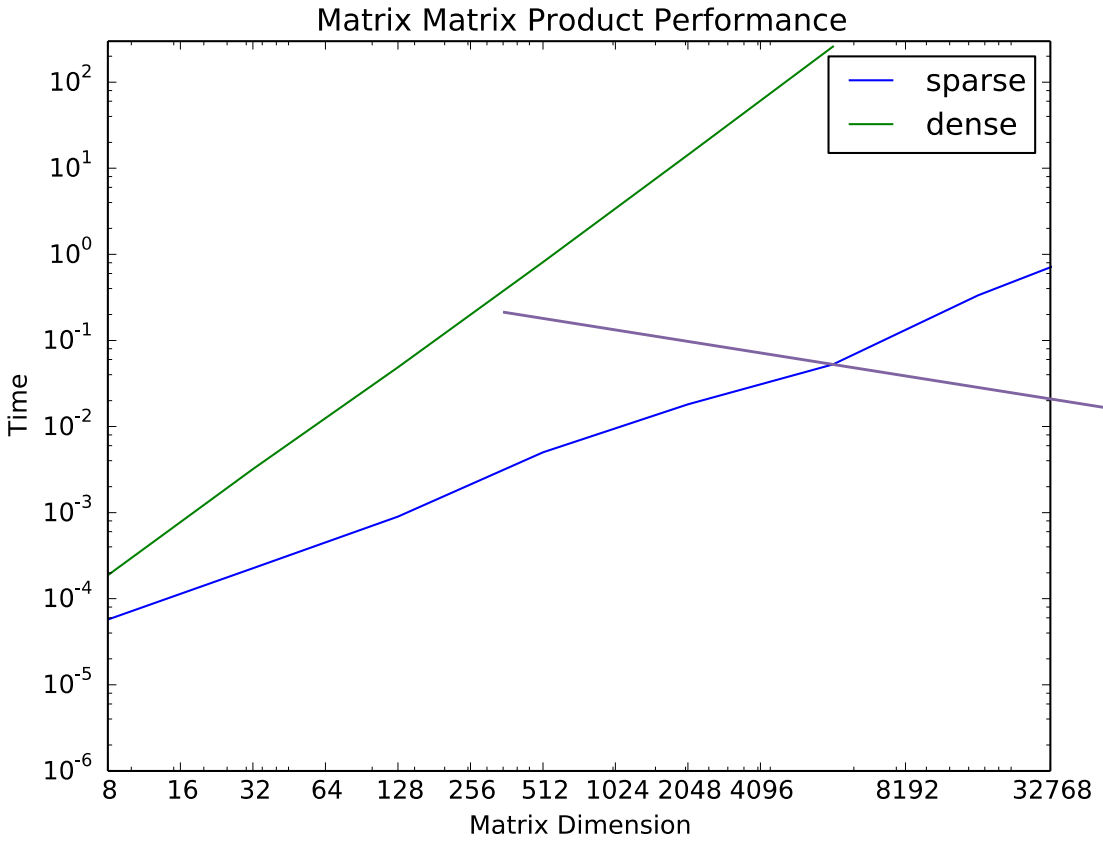
```
class COOMatrix {
public:
    COOMatrix(size_t M, size_t N)
      : num_rows_(M), num_cols_(N)
    {}


    size_t num_rows() const { return num_rows_; }
    size_t num_cols() const { return num_cols_; }

    void matvec(Vector const& x, Vector& y) const
    {
        for (size_t k = 0; k < storage_.size(); ++k)
            y(row_indices_[k]) += storage_[k] * x(col_indices_[k]);
    }

private:
    _rows_, nu
    r<size_t>
    r<size_t>
    r<double>
```

Index into y with row index

Multiply by corresponding value

Index into x with column index

Performance Comparison



$O(N^3)$

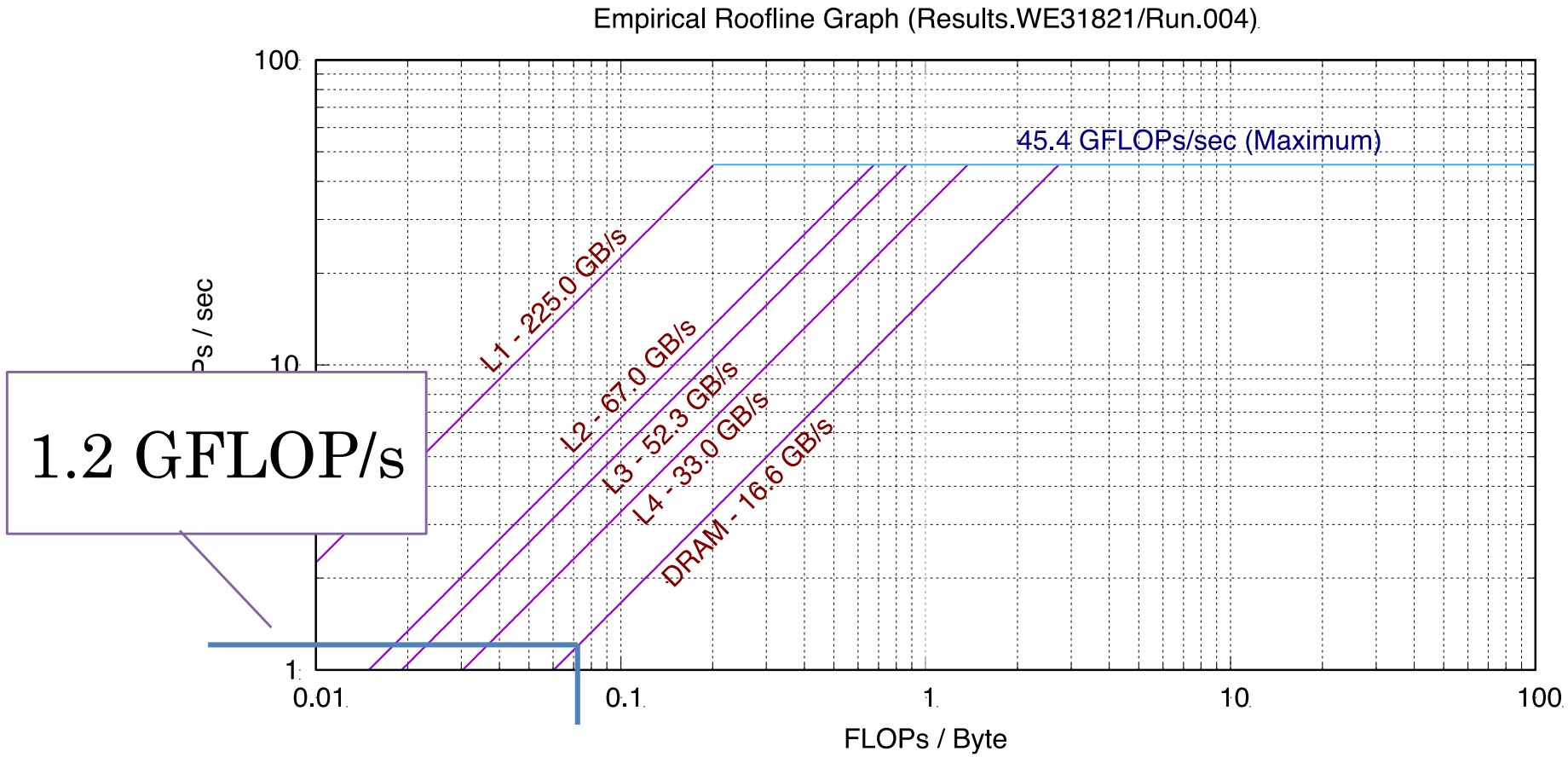
10x problem size = 1000x run time

Numerical Intensity

```
void matvec(Vector const& x, Vector& y) const {
    for (size_t k = 0; k < storage_.size(); ++k)
        y(row_indices_[k]) += storage_[k] * x(col_indices_[k]);
}
```

- Three doubles + 2 ints = 32 bytes?
- Two FLOP
- 1/6 FLOP/Byte

Measured


Empirical Roofline Graph (Results.WE31821/Run.004)
100
10
1
0.01
0.1
1
10
100
FLOPs / Byte
45.4 GFLOPs/sec (Maximum)
L1 - 225.0 GB/s
L2 - 67.0 GB/s
L3 - 52.3 GB/s
L4 - 33.0 GB/s
DRAM - 16.6 GB/s
1.2 GFLOP/s

Coordinate Storage

```
class COOMatrix {
public:
    COOMatrix(size_t M, size_t N) : num_rows_(M), num_cols_(N) {}

    void matvec(Vector const& x, Vector& y) const {
        for (size_t k = 0; k < storage_.size(); ++k)
            y(row_indices_[k]) += storage_[k] * x(col_indices_[k]);
    }

private:
    size_t num_rows_, num_cols_;
    std::vector<size_t> row_indices_;
    std::vector<size_t> col_indices_;
    std::vector<double> storage_;
};
```

- How do we initialize `storage_`?
- How do we create a sparse matrix to begin with?

Filling a Sparse Matrix

- Matrix is filled with something after being created
  - Starts off with zeros only (constructor)

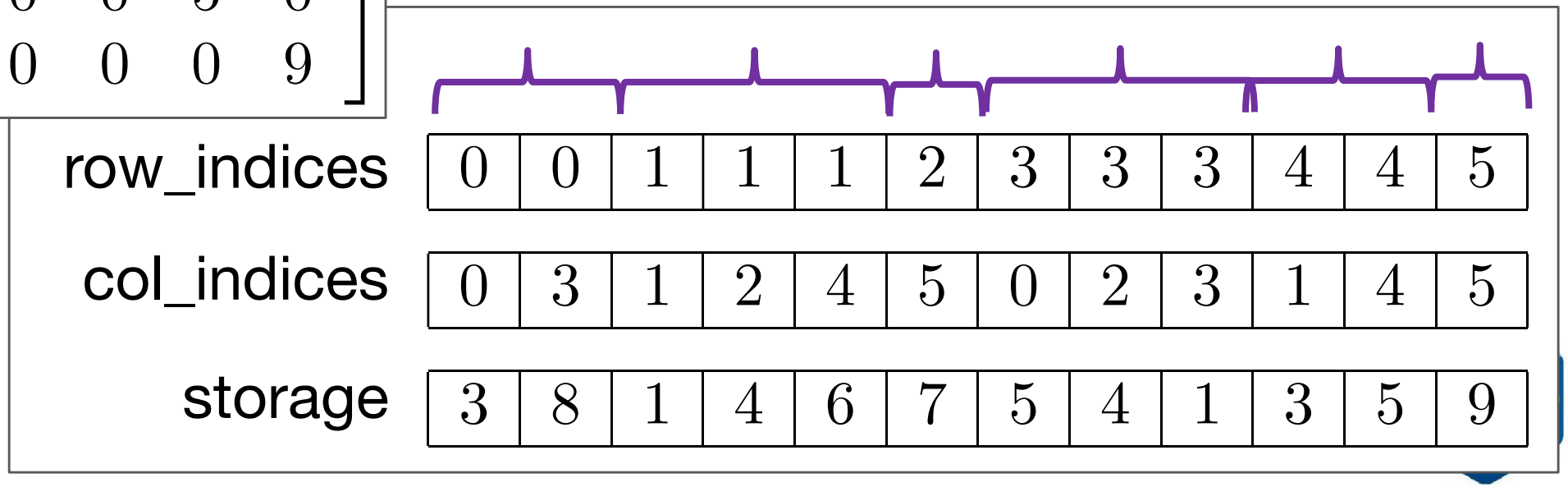
```
class COOMatrix {
public:
    COOMatrix(size_t M, size_t N) : num_rows_(M), num_cols_(N) {}

    void push_back(size_t i, size_t j, double val)
    {
        row_indices_.push_back(i);
        col_indices_.push_back(j);
        storage_.push_back(val);
    }

private:
    size_t num_rows_, num_cols_;
    std::vector<size_t> row_indices_;
    std::vector<size_t> col_indices_;
    std::vector<double> storage_;
};
```

- Can also append elements later (no ordering required)

Compressed Sparse Storage

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

- Each array stores same number of elements (number of non-zeros - nnz)
- But we can sort the elements by either row index or column index
- Values repeat



Compressed Sparse Storage

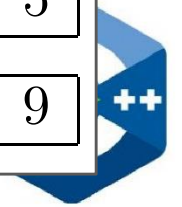
$$\begin{bmatrix} 3 & 0 & 0 & 8 & 0 & 0 \\ 0 & 1 & 4 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 \\ 5 & 0 & 4 & 1 & 0 & 0 \\ 0 & 3 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 9 \end{bmatrix}$$

- Unordered elements:

row_indices	4	0	3	0	1	1	3	1	2	3	5	4
col_indices	4	0	2	3	1	2	3	4	5	0	5	1
storage	5	3	4	8	1	4	1	6	7	5	9	3

- Elements ordered by row:
  - Note all arrays get reordered
  - Values stay together with their indices

row_indices	0	0	1	1	1	2	3	3	3	4	4	5
col_indices	0	3	1	2	4	5	0	2	3	1	4	5
storage	3	8	1	4	6	7	5	4	1	3	5	9



Run Length Encoding of Row Indices

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

row_indices	0	1	2	3	4	5						
run_length	2	3	1	3	2	1						
col_indices	0	3	1	2	4	5	0	2	3	1	4	5
storage	3	8	1	4	6	7	5	4	1	3	5	9

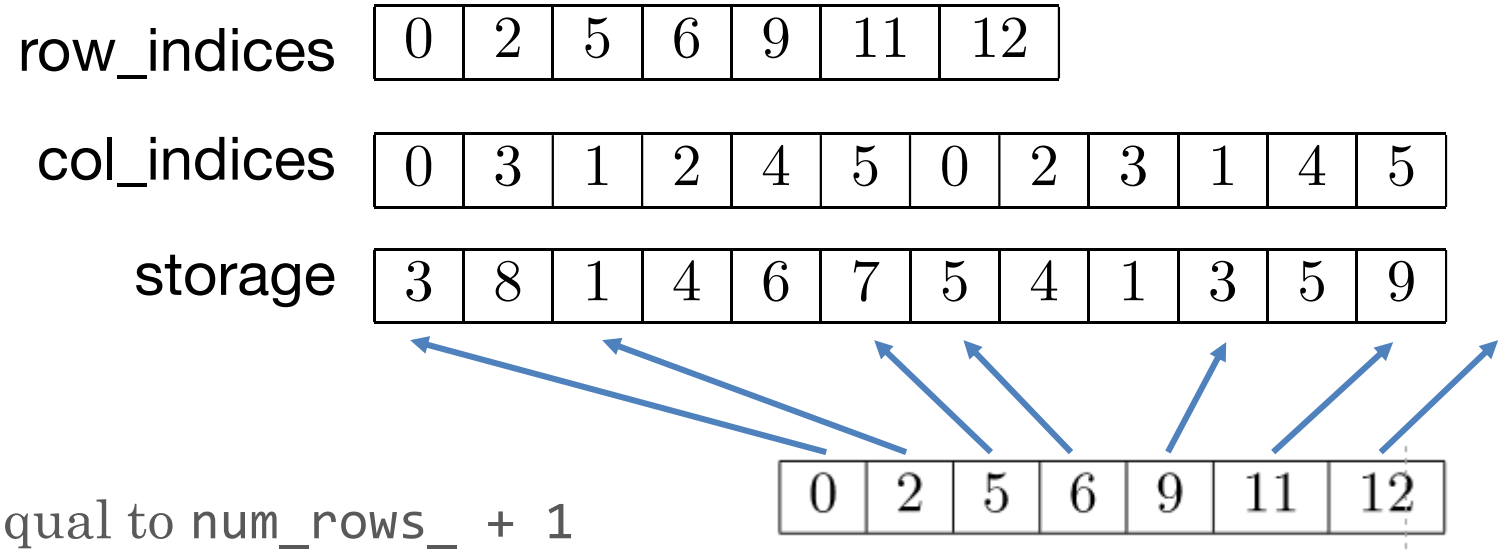
Do we need this?

```
size_t row_ptr = 0;      // keeps running total
for (size_t i = 0; i < num_rows_; ++i) {
    for(size_t j = row_ptr; j < row_ptr + run_length[i]; ++j)
        y[row_indices_[i]] += storage_[j] * x[col_indices_[j]];
    row_ptr += run_length[i];
}
```

Compressed Sparse Row (CSR) Storage

- Store running total instead of computing it:

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



- `row_indices` size is equal to `num_rows_ + 1`
  - Last entry (12) points to one past the end
- `row_indices` are indices to first element in each row

CSR Implementation

```
class CSRMatrix {
public:
  CSRMatrix(size_t M, size_t N)
    : is_open(false), num_rows_(M), num_cols_(N)
    , row_indices_(num_rows_ + 1, 0)      // note: initial value
  {}

  // Matrix size accessors
  size_t num_rows() const { return num_rows_; }
  size_t num_cols() const { return num_cols_; }

  // Useful info for sparse matrix
  size_t num_nonzeros() const { return storage_.size(); }

private:
  bool                is_open;
  size_t              num_rows_, num_cols_;
  std::vector<size_t> row_indices_, col_indices_;
  std::vector<double> storage_;
};
```

CSR Implementation (Matrix Vector Multiply)

```
class CSRMatrix {
public:
  CSRMatrix(size_t M, size_t N)
    : is_open(false), num_rows_(M), num_cols_(N)
    , row_indices_(num_rows_ + 1, 0)                // note: initial value
  {}

  void matvec(Vector const& x, Vector& y) const {
      for (size_t i = 0; i < num_rows_; ++i) {      // for each row
          // for each element in that row
          for (size_t j = row_indices_[i]; j < row_indices_[i + 1]; ++j)
              y(i) += storage_[j] * x(col_indices_[j]);
      }
  }
```

Index into y with row index

Multiply by corresponding value

Index into x with column index

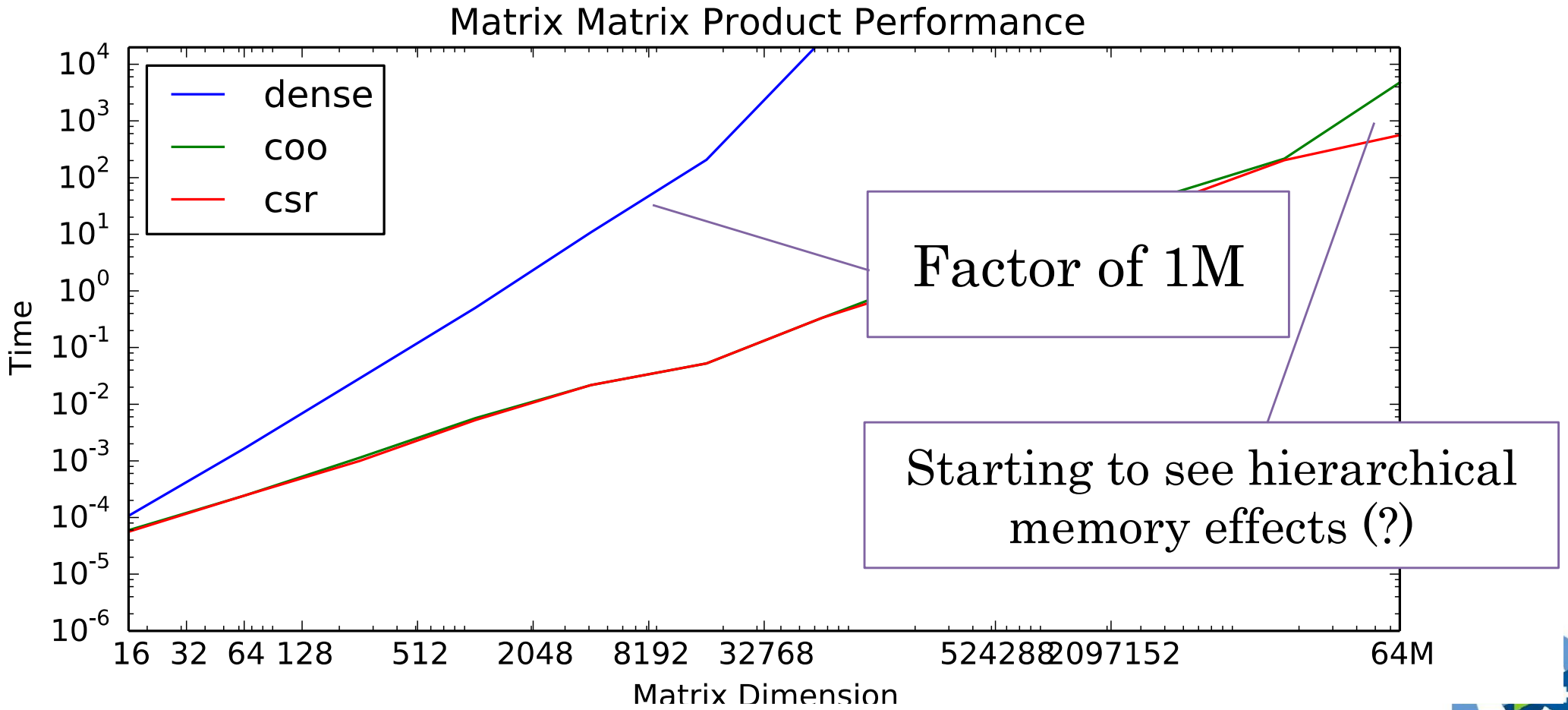
Building a CSR Matrix

```
class CSRMatrix {
public:
  CSRMatrix(size_t M, size_t N)
    : is_open(false), num_rows_(M), num_cols_(N), row_indices_(num_rows_ + 1, 0) {}

  void open_for_push_back() { is_open = true; }

  void close_for_push_back() {
    is_open = false;
    for (size_t i = 0; i < num_rows_; ++i) row_indices_[i + 1] += row_indices_[i];
    for (size_t i = num_rows_; i > 0; --i) row_indices_[i] = row_indices_[i - 1];
    row_indices_[0] = 0;
  }

  void push_back(size_t i, size_t j, double value) {
    ++row_indices_[i];
    col_indices_.push_back(j);
    storage_.push_back(value);
  }
};
```

Building a CSR Matrix

- Start pushing by calling:
  - `open_for_push_back()`
- Rows ***must*** be added in order and contiguously using `push_back()`
- When done pushing, accumulate run lengths to offsets:
  - `close_for_push_back()`

Performance



LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP