
Project #3:
Matrix-free LSQR

Table of content

Introduction of LSQR

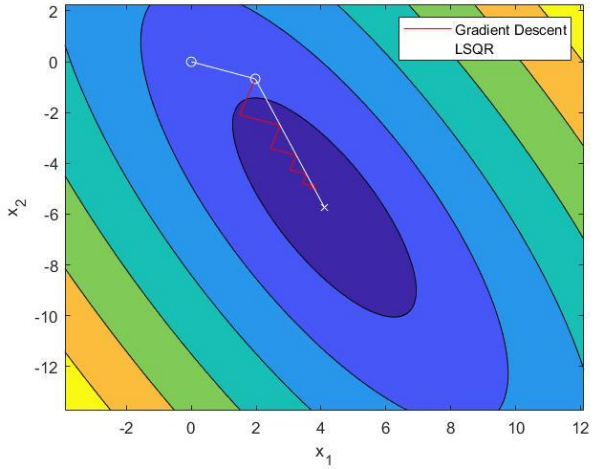
- LSQR is an **iterative method** that finds a solution x to the following problems:

$$\begin{aligned} &Unsymmetric\ equatios: && solve\ Ax = b \\ &Linear\ least\ squares: && minimize \|Ax - b\|_2 \\ &Damped\ least\ squares: && minimize \left\| \begin{bmatrix} A \\ \lambda I \end{bmatrix} x - \begin{bmatrix} b \\ 0 \end{bmatrix} \right\|_2 \end{aligned}$$

- where ***A*** is a *MxN*-Matrix, ***b*** is an *M*-Vector, ***λ*** is a scalar
- and potentially very **large** and **sparse**!

Concept of LSQR

- LSQR uses gradient information to iteratively locate the minimum of ||Ax-b||
  - LSQR analytically eq. to conjugate gradients
- More efficient then gradient descent but still without hessian matrix
- Since ||Ax-b|| is convex, if a minimum is found it is a global minimum



Matrix-free LSQR

- parallelize LSQR by Paige and Saunders
- most computationally expensive operations:

$$Av \quad \text{and} \quad A^T u$$

- **problem**: A is too big to fit into global memory
- need to be able to…

  … represent A in a compact format
  … parallelize matrix vector multiplication (using that format)

1. Initialize:
   $\beta_1 u_1 = b, a_1 v_1 = A^T u_1, w_1 = v_1,$
   $x_0 = 0, \overline{\phi}_1 = \beta_1, \overline{\rho} = \alpha_1$
2. Iterate over $i = 1, \ldots, i_{\max}$
3. Bidiagonalization of $\alpha_{i+1}\beta_{i+1}$:
   $\beta_{i+1} u_{i+1} = A v_i - \alpha_i u_i$
   $\alpha_{i+1} v_{i+1} = A^T u_{i+1} - \beta_{i+1} v_i$
4. Compute Orthogonal transformation variables
   (all scalar and dependent on $\alpha_{i+1}, \beta_{i+1}$):
   $\rho_i, c_i, s_i, \theta_{i+1}, \overline{\rho}_{i+1}, \phi_i, \overline{\phi}_{i+1}$
5. Update x and gradient equivalent w:
   $x_i = x_{i-1} + \left(\frac{\phi_i}{\rho_i}\right) w_i$
   $w_{i+1} = v_{i+1} - \left(\frac{\theta_{i+1}}{\rho_i}\right) w_i$

CUDA cuSPARSE

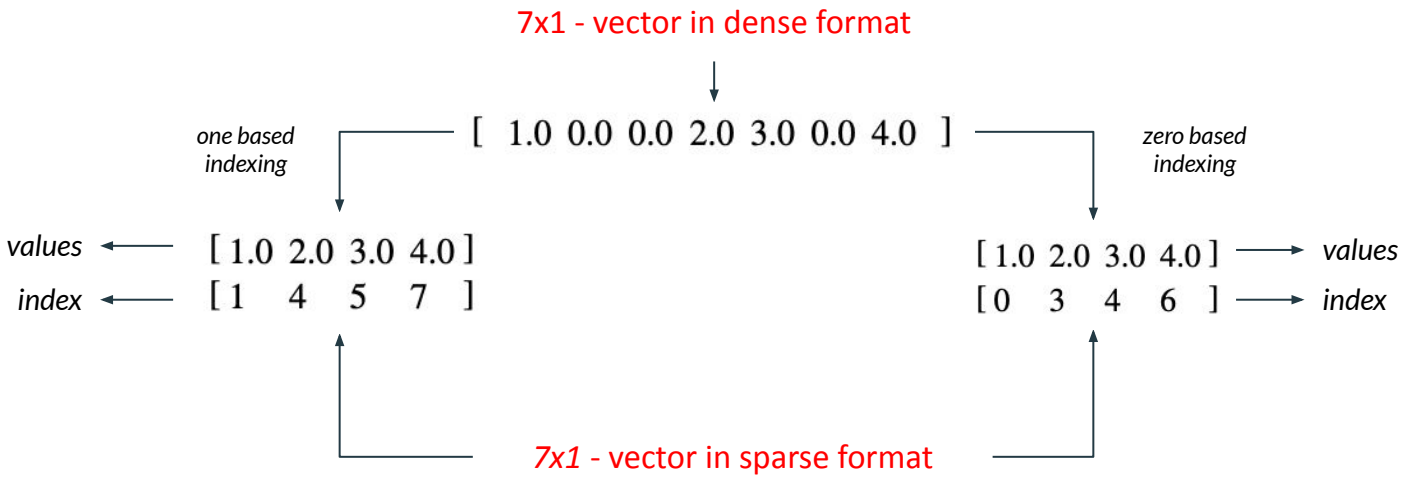
- is a matrix library in CUDA
- it contains basic linear algebra subroutines for handling sparse matrices

Library categories:

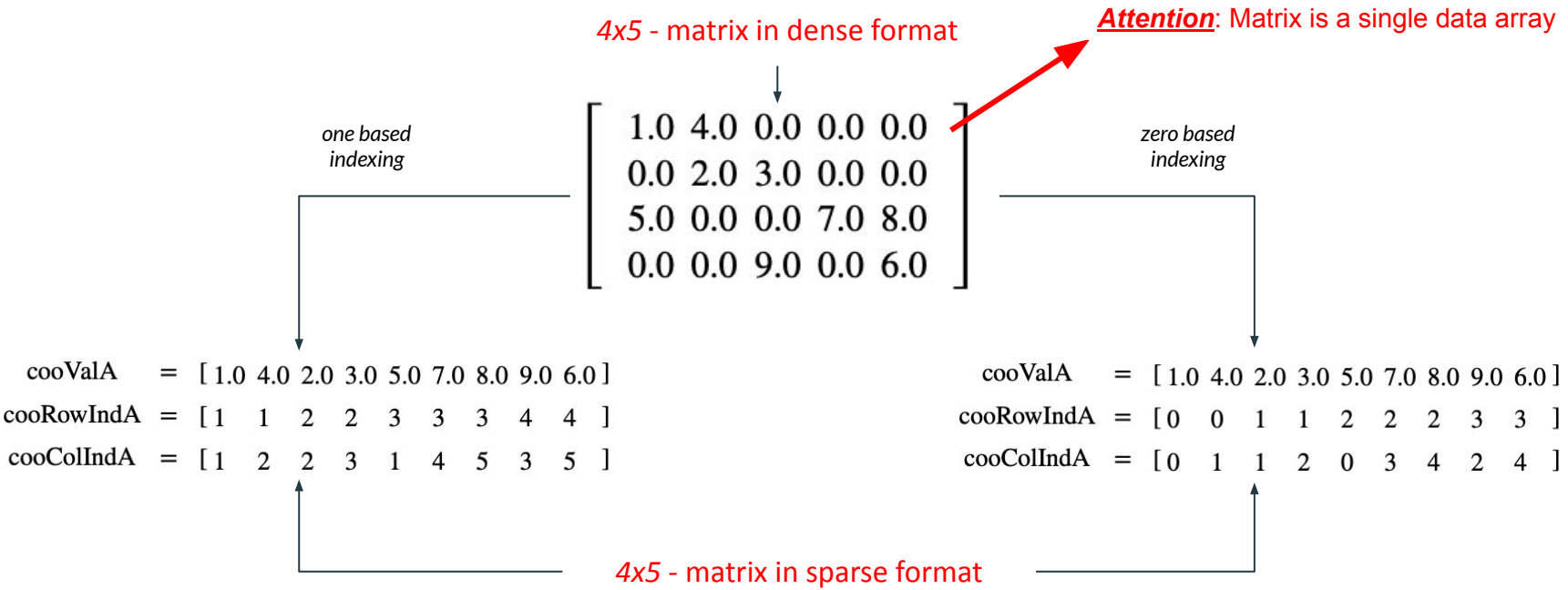
1. operations between two vectors
2. operations between a matrix and a vector
3. operations between a matrix and a set of vectors
4. operations that allow conversation between different matrix formats (dense/sparse)

Source: https://docs.nvidia.com/cuda/cusparse/index.html

Sparse and Dense for a vector



Sparse and Dense for a matrix



Working Plan

Phase 0: Understanding of LSQR method

Phase 1: Programm and test of the LSQR method with a small matrix

Phase 2: Try and modify LSQR method for a matrix ***A*** that can not fully stored in the memory

Group approach: Dominik & Yuval

1. Load a representation of the matrix ***A*** to the GPU memory
2. Design a GPU based implementation of the given algorithm
    a. Utilizing different memory hierarchies when possible (shared memory, constant..)
    b. Decide what parts of the algorithm can be efficiently parallelized
        i. vectors norms
        ii. vector/matrix operators
3. Use the appropriate cuSPARSE library functions in order to reduce needed operations
4. Design test methods for measuring CPU & GPU execution time and memory usage.

Group approach: Evgeny & Matheus

- Use LSQR algorithm as base
- Write interface for arbitrary sized vector-matrix multiplications (using cuSPARSE)
- Parallelize computation of vectors u, v (step 3) and x, w (step 5)
    - Synchronize always after step 3
- Possibly all vectors can be computed in parallel (then values of u and v may be discarded)
- Possibly step 4 can be computed in parallel (but without much gain)
    - In this case also synchronize after step 4

1. Initialize:
$$\beta_1 u_1 = b, \alpha_1 v_1 = A^T u_1, w_1 = v_1,$$
$$x_0 = 0, \overline{\phi}_1 = \beta_1, \overline{\rho} = \alpha_1$$
2. Iterate over $i = 1, \dots, i_{\max}$
3. Bidiagonalization of $\alpha_{i+1}\beta_{i+1}$:
$$\beta_{i+1} u_{i+1} = A v_i - \alpha_i u_i$$
$$\alpha_{i+1} v_{i+1} = A^T u_{i+1} - \beta_{i+1} v_i$$
4. Compute Orthogonal transformation variables
(all scalar and dependent on $\alpha_{i+1}, \beta_{i+1}$):
$$\rho_i, c_i, s_i, \theta_{i+1}, \overline{\rho}_{i+1}, \phi_i, \overline{\phi}_{i+1}$$
5. Update x and gradient equivalent w:
$$x_i = x_{i-1} + \left(\frac{\phi_i}{\rho_i}\right) w_i$$
$$w_{i+1} = v_{i+1} - \left(\frac{\theta_{i+1}}{\rho_i}\right) w_i$$

Group approach: Christian & Moritz

- Make sure that the matrix A is in a format that can be stored on a GPU
  - already in a compressed format
  - or our implementation will compress it (csr, coo, ...)
- Find the most computation heavy calculation steps of the algorithm
  - This includes the matrix x vector operations
  - And maybe the norms if the dimension of the vectors are large enough
- Use cuSPARSE to gain a significant performance increase for matrix x vector
- Move most of the code to the GPU to decrease communication overhead
- Test the implementation against the CPU code
  - measure performance increase
  - verify correctness of the solution

Group approach: Felix & Ruxandra

- the input matrix and auxiliary vectors should be represented in a **compressed format** and **loaded on the GPU** using **cuSPARSE**;
- analyzing the sequential algorithm, the most computationally intensive operations are matrix and vector multiplications so we must concentrate on making them as efficient as possible;
- considering the **matrix size** in memory and the **data dependencies** we must analyze:
  - how long can we keep the matrix on the GPU (upload/download)
  - synchronization techniques that must be used to avoid inaccuracy (data dependencies between iterations or in the same iteration);
- analyze the speedup in comparison with the CPU version;

Do you have any questions?