Project #3:
Matrix-free LSQR

Presented by Yuval Freund and Dominik Marino

Table of content

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~~ is really big

What we have done

- cuBlas implementation
  - Only using cuBlas subroutines for the lsqr implementation
- cuSparse implementation
  - Using cuSparse subroutines for the lsqr implementation
  - If there is no subroutine implemented → using own kernels
    - Norm kernel
    - Addition and subtraction vector kernel
    - Multiply scalar with a vector
- Kernel implementation
  - Norm kernel
  - Addition and subtraction kernel
  - Multiply scalar to vector
  - Multiply sparse matrix with vector
- Matlab and Python code for Testing and creating sparse matrices and vectors

Project - Read data and compile

- Read Data
  - First number has to be the row size
  - Second number has to be the column size
  - Then values for the matrix or vector
- How to start the program?

Modes:
- ➢ 0: cuBlas
- ➢ 1: Kernels
- ➢ 2: cuSparse

```
:~/PMPP_Project/build$ ./lsqr ../Data/Matrix5000x5000p50.txt ../Data/Vector5000x1p50.txt 5000 0
```

Mode

Directory

Path of matrix and vector

Maximum number of iteration

Example

dominik — ssh dmarino@gccg201.igd.fraunhofer.de — 111×29

```
dmarino@gccg201:~/PMPP_Project/build$ ./lsqr ../Data/Matrix5000x5000p50.txt ../Data/Vector5000x1p50.txt 5000 0
```

https://github.com/marinodominik/LSQR_Project

Sparse GPU Kernels for Deep Learning

Trevor Gale
Stanford University
United States of America
tgale@cs.stanford.edu

Matei Zaharia
Stanford University
United States of America
matei@cs.stanford.edu

Cliff Young
Google Brain
United States of America
cliffy@google.com

Erich Elsen
DeepMind
United Kingdom
eriche@google.com

***Abstract*—Scientific workloads have traditionally exploited high levels of sparsity to accelerate computation and reduce memory requirements. While deep neural networks can be made sparse, achieving practical speedups on GPUs is difficult because these applications have relatively moderate levels of sparsity that are not sufficient for existing sparse kernels to outperform their dense counterparts. In this work, we study sparse matrices from deep learning applications and identify favorable properties that can be exploited to accelerate computation. Based on these insights, we develop high-performance GPU kernels for two sparse matrix operations widely applicable in neural networks: sparse matrix–dense matrix multiplication and sampled dense–dense matrix multiplication. Our kernels reach 27% of single-precision peak on Nvidia V100 GPUs. Using our kernels, we demonstrate sparse Transformer and MobileNet models that achieve 1.2–2.1× speedups and up to 12.8× memory savings without sacrificing accuracy.**

***Index Terms*—Neural networks, sparse matrices, graphics processing units**

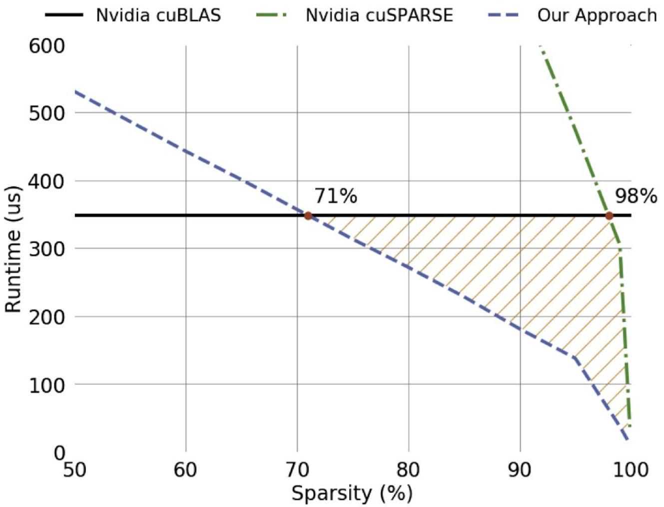
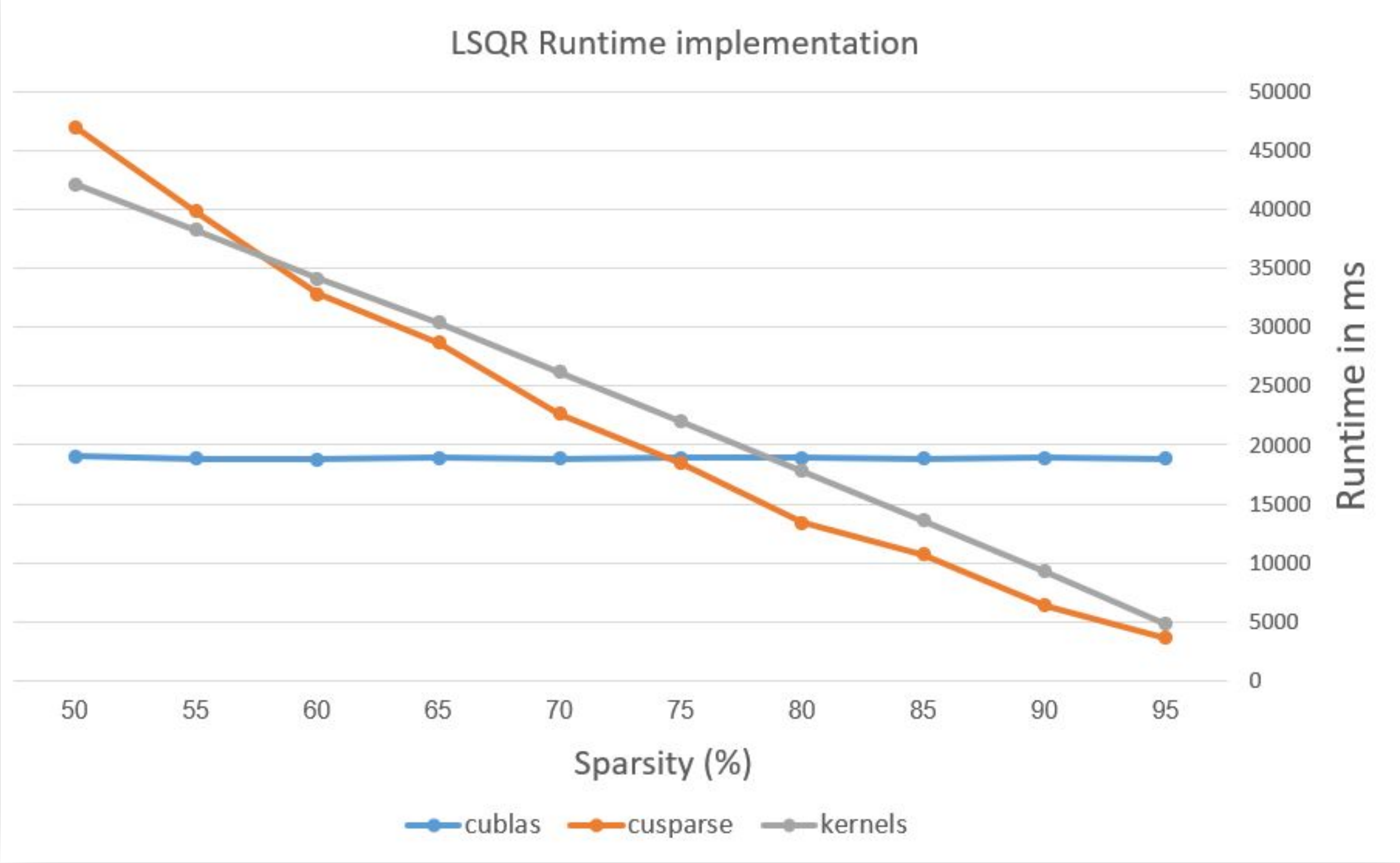


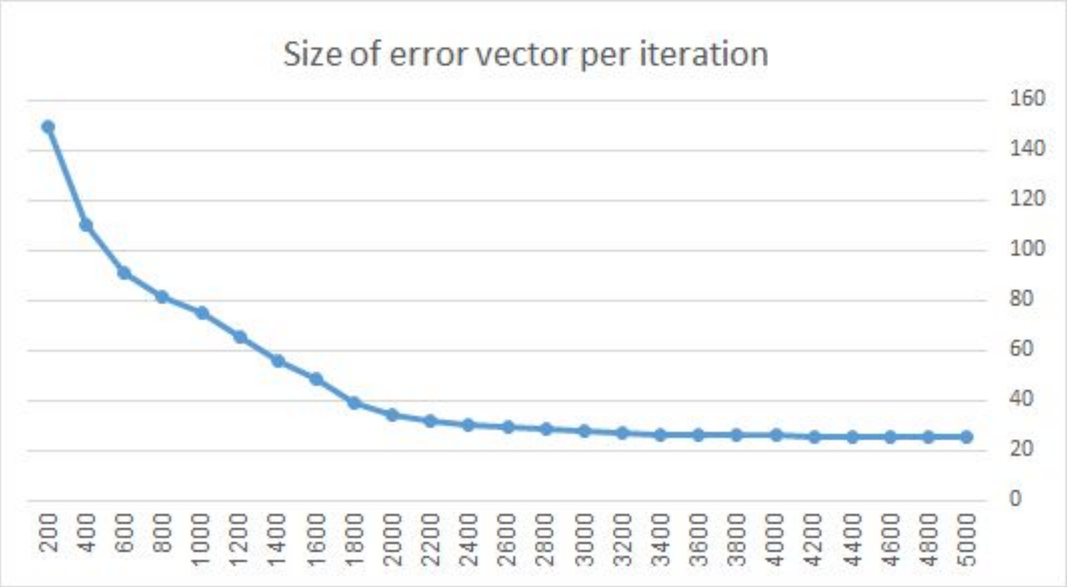
Fig. 1. **Sparse matrix–matrix multiplication runtime for a weight-sparse**

https://cs.stanford.edu/~matei/papers/2020/sc_sparse_gpu.pdf


LSQR Runtime implementation
50000
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
Runtime in ms
50
55
60
65
70
75
80
85
90
95
Sparsity (%)
cublas
cusparse
kernels

Convergence of the algorithm

(square matrix, 5000*5000, sparsity: 50%, values are integers in range 0-100)-> |Ax-b|



Troubles

- “Voodoo” Errors in kernels - when using more than one block of threads, variables had “left over” values in shared memory
- Errors with Linking/using old libraries of cublas/cusparse
- Univertable matrices might cause weird error values
- The algorithm is working for about 150 iterations and then goes completely wrong
- Problems with cudaMallocPitch
- Problems with transposing the matrix

```
if(tid < size){
    sdata[tid] = in_data[i];        //load global data in sh_memory
}else{
    sdata[tid] = 0;
}
__syncthreads();
```

What we have learned

- cuBlas und cuSparse are old-fashioned and hard to work with
- Checking cuda errors after every operation
- Cuda error codes are partial and confusing
- Shared memory needs to be initialized to 0
- Memory needs to be freed as soon as possible
- It’s better to avoid allocating memory and copying it
- Using sparse libraries is worth it at about 80% sparsity

```
cusparseStatus_t
cusparseCsr2cscEx2_bufferSize(cusparseHandle_t     handle,
                              int                  m,
                              int                  n,
                              int                  nnz,
                              const void*          csrVal,
                              const int*           csrRowPtr,
                              const int*           csrColInd,
                              void*                cscVal,
                              int*                 cscColPtr,
                              int*                 cscRowInd,
                              cudaDataType         valType,
                              cusparseAction_t     copyValues,
                              cusparseIndexBase_t  idxBase,
                              cusparseCsr2CscAlg_t alg,
                              size_t*              bufferSize)
```

```
cusparseStatus_t
cusparseCsr2cscEx2(cusparseHandle_t     handle,
                   int                  m,
                   int                  n,
                   int                  nnz,
                   const void*          csrVal,
                   const int*           csrRowPtr,
                   const int*           csrColInd,
                   void*                cscVal,
                   int*                 cscColPtr,
                   int*                 cscRowInd,
                   cudaDataType         valType,
                   cusparseAction_t     copyValues,
                   cusparseIndexBase_t  idxBase,
                   cusparseCsr2CscAlg_t alg,
                   void*                buffer)
```

Possible improvements

- Matrix multiplication with shared memory, warps to unroll
- Working on a better stopping criteria (taking into account improvement slow-down)
- Detect if given matrix ***A*** is invertible
- More general use of functions, that allow for example for transpose operation

```
__global__ void matrix_vector_multiplication(const int n_rows, const double *elements,
                                             const int *rowPtr, const int *colIdx, const double *x, double *result) {
    unsigned int row = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < n_rows) {
        const int row_start = rowPtr[row];
        const int row_end = rowPtr[row + 1];

        double sum = 0.0;
        for (int idx = row_start; idx < row_end; idx++) {
            int col = colIdx[idx];
            sum += elements[idx] * x[col];
        }
        result[row] = sum;
    }
    __syncthreads();
}
```

Do you have any questions?

https://www.pinterest.de/pin/396457573431396817/