1 TODO

- Double and No float

2 Grid and Blocksize

3 Solve

4 Final Thoughts

- Improvements
- view
- Multidevice support
- Grid and Blocksize

5 Anhang

- projectstructure (unittests ect, python prototypes)
- cublas check

6 Introcution

This is an documentation and project review. blabla

6.1 ABSNF

The abs-normal form is a representation of piecewise linear (PL) functions. Any PL function can be transformed to obtain this form. The process is described in []. The advantage and applications of the abs normal is not subject of this document but can be found in [] and [].

Definition. ABS-Normal-Form
For a PL function $f : \mathbb{R}^n \to \mathbb{R}^m$ the abs-normal representation takes the following form:

$$\begin{pmatrix} \Delta z \\ \Delta y \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix} + \begin{pmatrix} Z & L \\ J & Y \end{pmatrix} \times \begin{pmatrix} \Delta x \\ |\Delta z| \end{pmatrix} \tag{1}$$

where:

$$n, m, s \in \mathbb{R}, \Delta x \in \mathbb{R}^n, \Delta z \in \mathbb{R}^s, \Delta y \in \mathbb{R}^m, a \in \mathbb{R}^s, b \in \mathbb{R}^m, Z \in \mathbb{R}^{s \times n}, L \in \mathbb{R}^{s \times s}, Y \in \mathbb{R}^{m \times s}, J \in \mathbb{R}^{m \times n}$$

and

$$|\Delta z|$$

is the element-wise absolute vector of Δz. L is a lower triangular matrix.

Note that this representation is almost a linear system of equations. The non-linear parts of the function are all captured in $|\Delta z|$.

6.2 Problems

Given a PL function in abs-normal-form. The following tasks were given:

1. Evaluate the function in abs-normal form
   - Geg: $a, b, Z, L, J, Y, \Delta x$
   - Ges: $\Delta z, \Delta y$
2. Calculate the gradient of a function in abs-normal form
   - Geg: $a, b, Z, L, J, Y, \Delta z$
   - Ges: Gradient γ, Γ
3. Solve the system of equations
   - Geg: $a, b, Z, L, J, Y, \Delta y$
   - Ges: $\Delta x, \Delta Z$

Task was to solve each problem by using parallel computing with CUDA C++ to boost performance. The specific questions were

1. How can the problems be implemented with CUDA C++?
2. Is there a benifit of using parallel computing?
3. How does the implementation compare to a serial implementation?
4. Performance?

Each problem is content of one of the following sections, where we show the implementation and answer the given questions. [chapet 2] [chapter 3] [chapter 4].

Additionally chapter [...] we describe our approach of choosing the right parameters for CUDA as well as show our kernnels.

Last but not least we discuss our solution and give some final thoughts and possible improvements and further questions.

6.3 Additional Information

6.3.1 Used Software Libraries

All the plots, prototypes and a serial implementation was done in Python 3.6, where numerics were done using numpy library. For the CUDA C++ implementation, we used the following libraries:

- cuBLAS (cuda Basic Linear Algebra Subprograms)
  - Matrix Vector operations
  - Matrix Matrix operations
- cuSOLVER
  - Matrix factorization
  - Triangular solve
- C++ STL

6.3.2 Devices

We tested our code on the following devices, which were also used for benchmarking. The results can be found in chapter ...

- NVIDIA Tesla P100
  - Global Memory
  - MPU
  - Warps / MPU
  - Threads / WARP
  - MaxThreads / Block
  - DOUBLE PRECISSION
  - SINGLE PRECISSION
- NVIDIA Geforce GTX 780
  - Global Memory
  - MPU
  - Warps / MPU
  - Threads / WARP
  - MaxThreads / Block
  - MaxFlops DOUBLE PRECISSION
  - SINGLE PRECISSION
- Intel Core i5-2500 CPU, 16GB RAM

TABLE

6.3.3 Notation and Symbols

In the rest of the document we use the following symbols and notation:

- m, n, s denote the dimensions of the datastructures of the abs-normal form ()
- $\Delta x, \Delta z, \Delta y, a, b, Z, L, J, Y$ denote the datastructures with given dimensions in ( ).
- $|\circ|$ is the absolute value of $\circ$ if $\circ$ is a scalar and the elementwise absolute vector if $\circ$ is a vector.
- Tesla
- GTX
- i-5

6.3.4 Project structure

The code of the implementation as well as its corresponding unit-tests can be found ... Plots, Serial implementation, Performance tests, raw data of performance ....

7 Evaluation of the ABSNF

7.1 Problem Specification

ABS-Normal Form:

$$\begin{pmatrix}\Delta z\\ \Delta y\end{pmatrix} = \begin{pmatrix}a\\ b\end{pmatrix} + \begin{pmatrix}Z & L\\ J & Y\end{pmatrix} \circ \begin{pmatrix}\Delta x\\ |\Delta z|\end{pmatrix}$$

Given is a PL function in abs-nf. The evaluation of this function means calculating the vectors Δy and Δz:

$$\Delta y = b + (J \times \Delta x) + (Y \times |\Delta z|) \tag{2}$$
$$\Delta z = a + (Z \times \Delta x) + (L \times |\Delta z|) \tag{3}$$

where the following structures are given:

$$a, b, Z, L, J, Y, m, n, s, \Delta x$$

In (3) Δz depends on the element-wise absolute function of its own and therefore it cannot be calculated with a simple matrix vector dot product. Since the matrix L is lower triangular, the vector Δz can be iteratively calculated, by taking the row-wise dot product of L and $|\Delta z|$.

To illustrate the process, consider:

$$k = a + Z \times \Delta x$$

Now we calculate Δz element wise:

$$\begin{aligned}
\Delta z_1 &= \underbrace{L_1 \times |\Delta z|}_{=0} + k_1 = k_1\\
\Delta z_2 &= L_2 \times |\Delta z| + k_2\\
&= L_{2,1} \times |\Delta z_1| + k_2\\
\Delta z_3 &= L_3 \times |\Delta z| + k_3\\
&= L_{3,1} \times |\Delta z_1| + L_{3,2} \times |\Delta z_2| + k_3\\
\Delta z_4 &= L_4 \times |\Delta z| + k_4\\
&= L_{4,1} \times |\Delta z_1| + L_{4,2} \times |\Delta z_2| + L_{4,3} \times |\Delta z_3| + k_4\\
&\ldots.
\end{aligned}$$

7.2 Implementation

Our implementation is highly focused on speed and demands the device to hold all the required data structures in global memory simultaneously.
Given this premise, the calculation of (2) and (3) is a series of dot products and therefore is highly parallelizeable. For this we relied mainly on CUBLAS routines. A simplified version of our implementation can be found in fig. (**??**).

7.2.1 Performance Experiments

For measuring performance and the subsequent analysis, we simplified the process by equalizing the dimensions of the data-structures:

$$m = n = s$$

```
template <typename T>
void eval(T *a, T *b,
T *Z, T *L,
T *J, T *Y,
T *dx,
int m, int n, int s,
T *dz, T *dy,
T *abs_dz)
{
// dz = a
cudaMemcpy(dz, a, ., cudaMemcpyDeviceToDevice));
// dz = Z * dx + dx
cublasDgemv(.,Z, ., dx, . dz, .)
// dz[i] = L[i]_i * |dz|_i
for(int i=0; i<s; i++)
{
cublasDgemv( . ,&L[i * s], . ,abs_dz, . , &dz[i],.);
abs <<<1,1>>>(&dz[i], &abs_dz[i], 1);
}
// dy = b
cudaMemcpy(dy, b, ., cudaMemcpyDeviceToDevice);
// dy = dy + J*dx
cublasDgemv(.,J, ., dx, ., dy, .));
// dy = dy + Y * |dz|
cublasDgemv(., Y, ., abs_dz, ., dy, .));
}
```

Figure 1: Simplified Implementation of the evaluation function

7.2.2 Single Execution

In this experiment, we executed the serial python- as well as the parallel CUDA implementations for different dimensions of s and measured the runtime (fig. 2). Since the results were not as expected in favor of the CUDA implementation, we also dismantled the runtime on the parallel devices and measured the "data-upload-time" and the "execution-only-time" separately (fig. 3).

7.2.3 Multiple Executions

Since the scenario of evaluating the function just once is not quite realistic, we did a second experiment, where we uploaded data to the devices and executed the evaluation routine a 1000 times. Here we only measured the pure execution time without the data-transfer, since the upload time should be marginal with a high enough number of executions. The results can be found in fig. 4.

7.2.4 Analysis

Here we can clearly see, that the transfer time, which is the time that it takes to upload the required data structures onto the device is extremely significant and takes a disproportional high share of the total time.

The results of the experiment are heavy in favor of the serial implementation.
For data, that completely fit into the global memory of the device, we couldn't get any performance gains through the cuda implementation on given devices. (fig 2 and fig. 4). The simple numpy version even outperforms the cuda version on the tesla P100.

If the data structures get that big such that they don't fit into the global memory of the device, we can expect the performance to be even worse, since the data-transfer time, takes a huge part of the overall runtime. figure 3.

We therefore come to the conclusion, that the considerable effort of implementing a parallel version of the eval function is not worth the effort.

- Memory Complexity $O(s^2)$

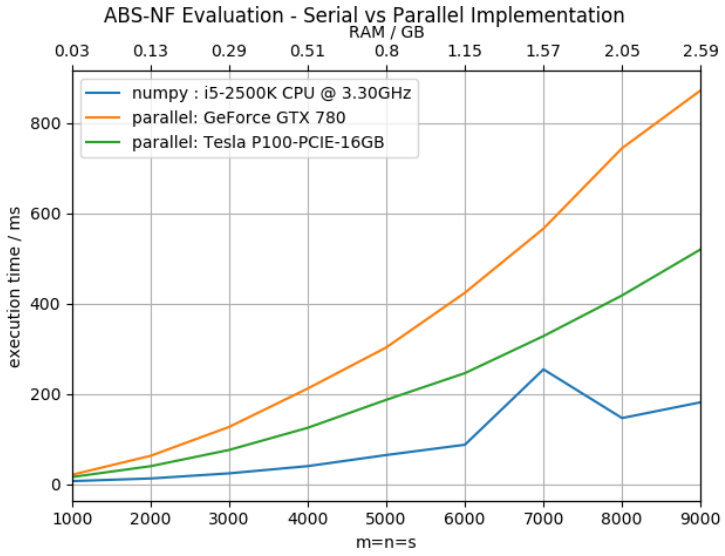

Figure 2: Single execution of the evaluation function on different devices.

- Complexity $O(s^2)$
- mempry is bottle neck
- no performance gain expected

what can be implemented differently? What can be improved?

7.2.5 Notes

- Double Precision on GTX is nuts

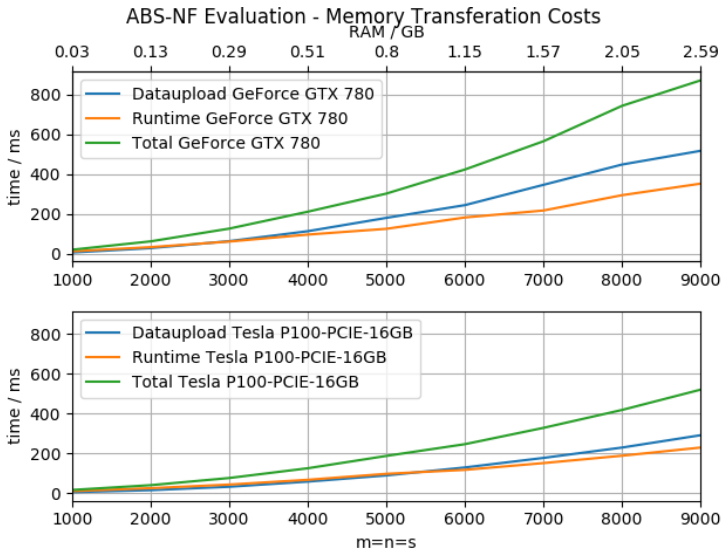

Figure 3: Data-transfer and execution time of the parallel implementation

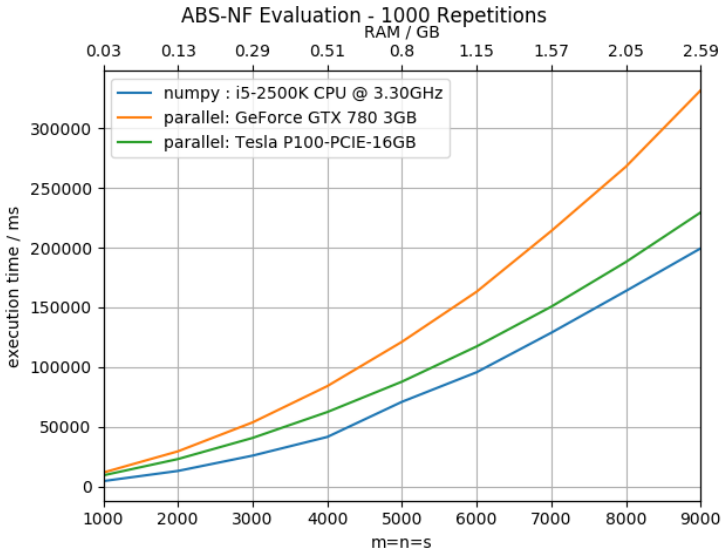

Figure 4: Multiple executions of the evaluate function on different devices

8 Gradient

8.1 Problem Specification

ABS-Normal Form:

$$\begin{pmatrix}\Delta z\\ \Delta y\end{pmatrix} = \begin{pmatrix}a\\ b\end{pmatrix} + \begin{pmatrix}Z & L\\ J & Y\end{pmatrix} \circ \begin{pmatrix}\Delta x\\ |\Delta z|\end{pmatrix}$$

The problem here is to calculate the gradient of a PL function in abs-normal form. Given the following structures:

$$a, b, Z, L, J, Y, m, n, s, \Delta z$$

the gradient can be obtained in the following way:

$$\Sigma = diag(sign(\Delta z))$$

$$\begin{aligned}\Delta z &= a + Z\Delta x + L\Sigma\Delta z\\ &= (I - L\Sigma)^{-1}(a + Z\Delta x)\\ \Delta y &= b + J\Delta x + Y|\Delta z|\\ &= b + J\Delta x + Y\Sigma\big((I - L\Sigma)^{-1}(a + Z\Delta x)\big)\\ &= b + Y\Sigma(I - L\Sigma)^{-1}a + \Big(J + Y\Sigma(I - L\Sigma)^{-1}Z\Big)\Delta x\end{aligned}$$

$$\gamma = b + Y\Sigma(I - L\Sigma)^{-1}a \tag{4}$$

$$\Gamma = J + Y\Sigma(I - L\Sigma)^{-1}Z \tag{5}$$

The gradient can now e calculated as:

$$\Delta f(\Delta x) = \gamma + \Gamma\Delta x$$

8.2 Implementation

For the implementation we heavily relied on CUBLAS routines. A simplified version of the core-function can be found in fig. **??**. The attentive reader might note that gridsize and blocksize are fixed for all the kernels in this function. We discuss our approach of choosing these parameters in section (..).

8.3 Performance Experiments

8.3.1 Single Execution

In this experiment we executed the gradient implementation on the GTX as well as on the Tesla and measured the data-upload time as well as the execution time of the function (fig. 6).

8.3.2 Multiple Executions

In this experiment we measured the runtime of the serial version as well as the parallel version for a 100 executions of the gradient funciton. Data-transfer on and from the device was not included. The results can be found in fig. 7. Since the graphs are not quite detailed, we took some of the results into table (1).

Additionally we profiled a 100 executions of the gradient function with nvprof (table. 2).

```
template <typename T>
void gradient(T *a, T *b,
T *Z, T *L,
T *J, T *Y,
T *dz,
T *Tss, T *I, T *K,
int m, int n, int s,
int gridsize, int blocksize,
T *gamma, T *Gamma)
//  d_Tss = diag(1) - L * diag(sign(dz))
initTss <<<gridsize, blocksize >>>(d_Tss,d_L, d_dz, s, s*s);
//  d_I = diag(1)
initIdentity <<<gridsize, blocksize >>> (d_I, s);
//  d_I = d_Tss * X
getTriangularInverse(handle, d_Tss, d_I, s);
//  d_I = d_I * diag(sign(dz))
multWithDz <<<gridsize, blocksize >>>(d_I, d_dz, s);
//  d_K = d_Y * d_I
cublasDgemm(.,d_Y,.,d_I,d_K,));
//  d_gamma = d_b
//  d_Gamma = J
cudaMemcpy(d_gamma, d_b,.);
cudaMemcpy(d_Gamma, d_J,.);
//  d_gamma = d_gamma + K*a
cublasDgemv(.,d_K,., d_a,., d_gamma,.);
//  d_Gamma = d_Gamma + K*Z
cublasDgemm(.,d_K,d_Z,d_Gamma,m));
}
```

Figure 5: Simplified Implementation of the gradient function

$s = m = s$	numpy (ms)	GTX (ms)	Tesla (ms)
1000	20008	3372	282
2000	144644	23122	1223
4000	1155222	173604	7947

Table 1: This table shows some data

Time(%)	Time	Calls	Avg	Min	Max	Name
47.67%	11.8009s	900	13.112ms	358.57us	90.166ms	dgemm_sm_heavy_ldg_nn
32.04%	7.93153s	100	79.315ms	77.982ms	93.386ms	dgemm_sm_heavy_ldg_nt
5.77%	1.42887s	900	1.5876ms	95.107us	10.248ms	dgemm_sm35_ldg_nn_128x8x64x16x16
4.24%	1.05012s	800	1.3126ms	397.74us	1.7257ms	void kernel_trsm_l_mul32
3.37%	835.25ms	200	4.1763ms	142.89us	9.6000ms	dgemm_sm35_ldg_nn_64x8x128x8x32
3.32%	820.79ms	100	8.2079ms	6.1753ms	9.5876ms	dgemm_sm35_ldg_nt_128x8x64x16x16
3.26%	806.34ms	100	8.0634ms	6.1730ms	8.6610ms	dgemm_sm35_ldg_nt_64x8x128x8x32
0.12%	29.230ms	200	146.15us	1.8560us	294.63us	[CUDA memcpy DtoD]
0.09%	22.342ms	8	2.7927ms	1.1850us	5.7200ms	[CUDA memcpy HtoD]
0.06%	15.472ms	100	154.72us	152.81us	165.99us	void gemv2N_kernel_val
0.06%	13.796ms	200	68.981us	18.880us	136.90us	void trsm_left_kernel
0.00%	1.1163ms	100	11.162us	10.848us	14.016us	void absnf::initTss
0.00%	1.0580ms	100	10.579us	10.304us	12.320us	void absnf::multWithDz
0.00%	557.88us	100	5.5780us	5.4400us	7.3600us	void absnf::initIdentity

Table 2: Results of nvprof on the gradient function with 100 executions

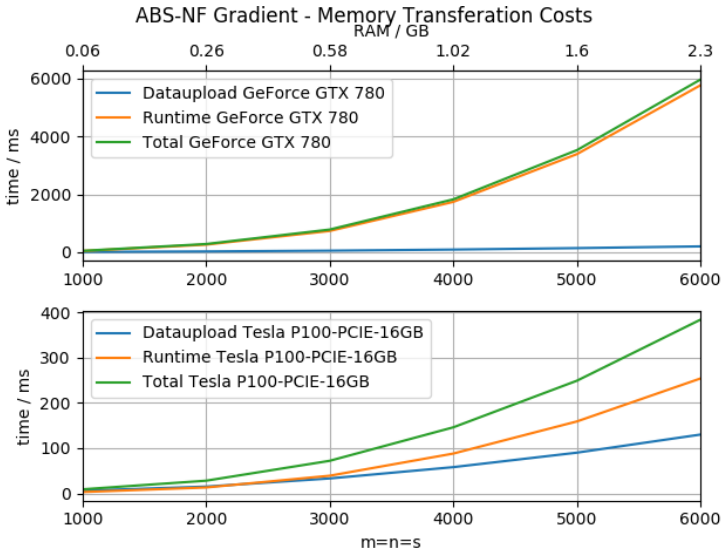

Figure 6: Multiple executions of the evaluate function on different devices

8.4 Analysis

In contrast to the evaluate function, the parallel cuda version of the gradient function was magnitudes faster than the serial numpy version fig. (7). It is important to note, that the numpy version is not optimized but rather a quick and dirty implementation to establish a baseline and should therefore not be considered as fastest serial approach.
We can also see that the Tesla device here operates much faster than the GTX which can be deduced to the better double precision support.

In figure 6 a much better data-transfer to runtime ratio for both the GTX, as well as the Tesla are perceptible. This is important, since a batched version for data structures of size bigger than the global gpu memory may be worthwhile.

The profile in table 2 shows, that the majority of the runtime goes into the calculation of matrix-matrix products. The handcrafted kernels ”initTss”, ”multWithDz” and ”initIdentity” do not preponderate, as initially feared.

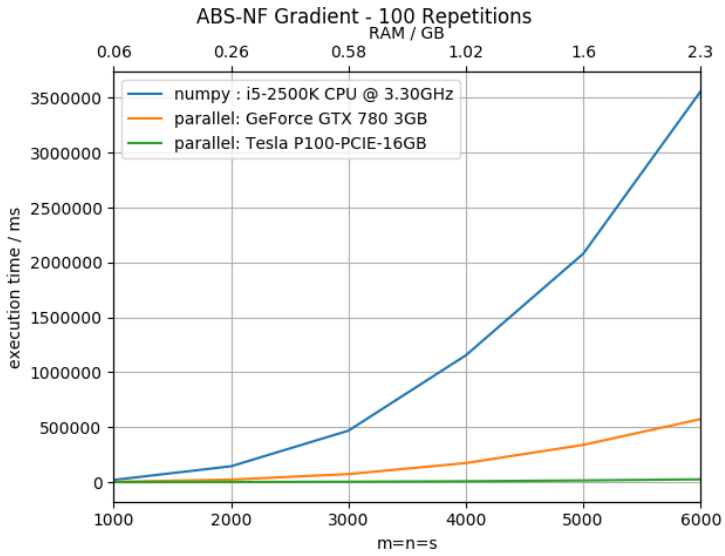

Figure 7: Multiple executions of the gradient function on different devices

9 Gridsize and Blocksize

All of our self written kernels have to operate on matrices and vectors. Usually the single entries of the structures do not depend on the others and can be independently and therefore in parallel calculated.

EXAMPLE + MORE EXAMPLE LISTS

The questions while implementing this operation were:

1. How to step through the data-structures in order to minimize cache-misses?
2. How to choose the grid-size?
3. How to choose the block-size?

Obviously the most efficient and preferment answer to this question is to decide problem specific and device specific. Unfortunately this is also the most time consuming approach.
In our case we tried to find a generic solution, that performs well enough to not deteriorate the overall performance.

The basic idea is:

- Chose and fix blocksize and gridsize depending on the device properties
- Start kernel with given blocksize and gridsize
- Each thread can be responsible for multiple tasks and chooses its next task after the current one is done or terminates

The basic algorithm for kernels of this type is listed in **??**

```
template <typename T>
void _global_ row_wise_traversal(T *matrix, int s)
{
  int i = blockIdx.x;
  int j = threadIdx.x;
  int global_id = blockIdx.x * blockDim.x + threadIdx.x;
  int id = i*s + j;
  int size = s*s;
  while(id < size && i < s)
  {
    matrix[id] = doSomething();
    j += blockDim.x;
    if (j>=s)
    {
      j = j % s;
      i = i + gridDim.x;
    }
    id = i*s+j;
  }
}
```

Here each block is assigned a row of the matrix. Threads operate on these rows and calculate autonomously their next task.

9.1 Choosing the right blocksize and gridsize

Ideally we want to achieve the following objectives:

1. Utilize all the MPUs to capacity
2. Minimizing cache misses

We can minimize cache misses by assigning threads of the same block and and the same warp adjacent matrix entries. This is automatically achieved by assigning matrix rows to blocks. To ensure that each Core of the MPU runs on maximal capacity, we start at least as many threads as a MPU can execute in parallel. This obviously finds its limit with the constant max_threads_per_block.

TABLE WHITEPAPERg

9.2 Performance

GRAPHICS

GRAPHICS 2

9.3 Notes

Our approach obviously lacks. Device specfications are not taken in consideration. SP, DP.

10 Solve

ABS-Normal Form:

$$\begin{pmatrix}\Delta z\\ \Delta y\end{pmatrix} = \begin{pmatrix}a\\ b\end{pmatrix} + \begin{pmatrix}Z & L\\ J & Y\end{pmatrix} \times \begin{pmatrix}\Delta x\\ |\Delta z|\end{pmatrix}$$

The last function, that had to be implemented was a solver for PL function in abs-normal form. Given:

$$a, b, Z, L, J, Y, m, n, \Delta y$$

We want to calculate:

$$\Delta x, \Delta z$$

10.1 Deducing a solution

First and foremost, we assume:

$$\Delta y = 0$$

It this is not the case, we can replace b with b':

$$b' = b - \Delta y$$

Now we can shift the equation system:

$$\begin{aligned}
\Delta y &= b + J\Delta x + Y|\Delta z| \\
0 &= b + J\Delta x + Y|\Delta z| \\
-b - Y|\Delta z| &= J\Delta x \\
b + Y|\Delta z| &= J\Delta x(-1) \\
J^{-1}(b + Y|\Delta z|) &= -\Delta x \\
\Delta x = -J^{-1}(b + Y|\Delta z|)&
\end{aligned}$$

$$\begin{aligned}
\Delta z &= a + Z\Delta x + L|\Delta z| \\
&= a + Z\Big(- J^{-1}(b + Y|\Delta z|)\Big) + L|\Delta z| \\
&= a + Z\Big(- J^{-1}b - J^{-1}Y|\Delta z|\Big) + L|\Delta z| \\
&= a - ZJ^{-1}b - ZJ^{-1}Y|\Delta z| + L|\Delta z| \\
&= a - ZJ^{-1}b - (ZJ^{-1}Y - L)|\Delta z|
\end{aligned}$$

$$\begin{aligned}
\Delta z &= a - ZJ^{-1}b - (ZJ^{-1}Y - L)|\Delta z| \\
&= c + S|\Delta z| \\
c &= a - ZJ^{-1}b \\
S &= (L - ZJ^{-1}Y)
\end{aligned}$$

10.2 Implementation

$$\begin{aligned}
c &= a - ZJ^{-1}b \\
S &= (L - ZJ^{-1}Y)
\end{aligned}$$

Need the inverse of J

- Calculate cublas LU
- Cusolve QR
- LU -¿ Inverse

$$c = a - Z * solve(J * X = b)$$
$$S = L - Z * solve(J * X = Y)$$

11 Operations

Operation	Function
Matrix - Vector Product	cublas
Matrix - Vector Vector Product	cublas
Vector Vector Addition	cuutils