1 TODO

- Double and No float

2 Gliederung

- ABSNF
- Problemstellung
- Project Structure (where is what)
- Devices
- Eval
- Gradient
- Solve
- Grid and Blocksize
- improvements
  - solve algorithmus
  - sparsity
  - multidevice support
  - memorymangaer

2.0.1 Kapitel

- Problemstellung
- Implementierung
- Performance
- Improvements and bottlenecks

3 Introduction

- Einführung ABSNF
- Aufgabenstellung
- Wichtige Fragen
- Was habe ich gemacht
- Used Libraries
- Devices
- Notation and Symbols
- Description of the data structures
- elementwise abs

4 Evalutation of the ABSNF

- Problem
- Implementations
- Performance
  - Single Core Float and Double
  - GTX Float and Double
  - Tesla Float and Double
- Review and Notes

5 Gradient of the ABSNF

6 Grid and Blocksize

7 Solve

8 Final Thoughts

- Improvements
- view
- Multidevice support
- Grid and Blocksize

9 Anhang

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

10 Introcution

10.1 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 $|\Delta z|$.

10.2 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.

10.3 Additional Information

10.3.1 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

10.3.2 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

10.3.3 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.

10.3.4 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 ....

11 Evaluation of the ABSNF

11.1 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 $|\Delta z|$.

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

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

11.2 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 parallelize-able. For this we relied mainly on CUBLAS routines. The implementation is available on [...] with several interfaces in [...]..

```
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, .));
}
```

11.2.1 Performance Experiments

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

$$m = n = s$$

11.2.2 Single Execution

In this experiment, we executed the serial python implementation as well as the parallel cuda implementation for different dimensions of s and measured the total runtime of the program. The results can be seen in figure 1. 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-time" separately. Those results can be found in figure 2. Here we can clearly see, that the transfer time, which is the time that it takes to upload the required datastaructures onto the device is extremely significant and takes a disproportional high share of the total time.

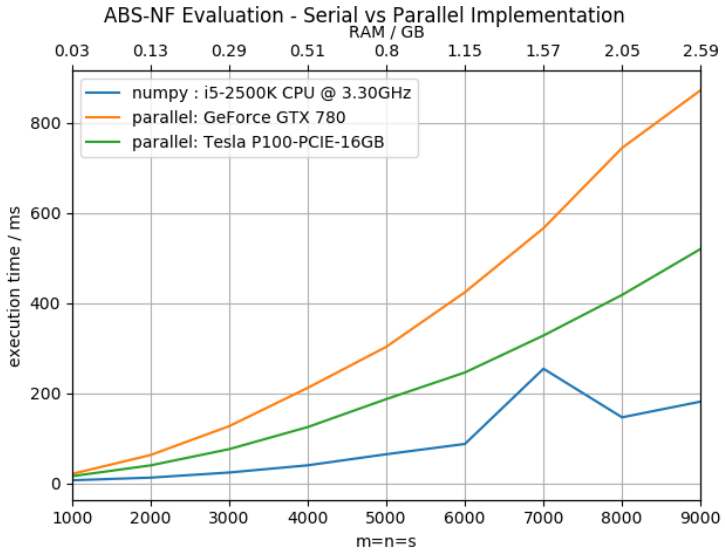

Figure 1: Single execution of the evaluation function on different devices.

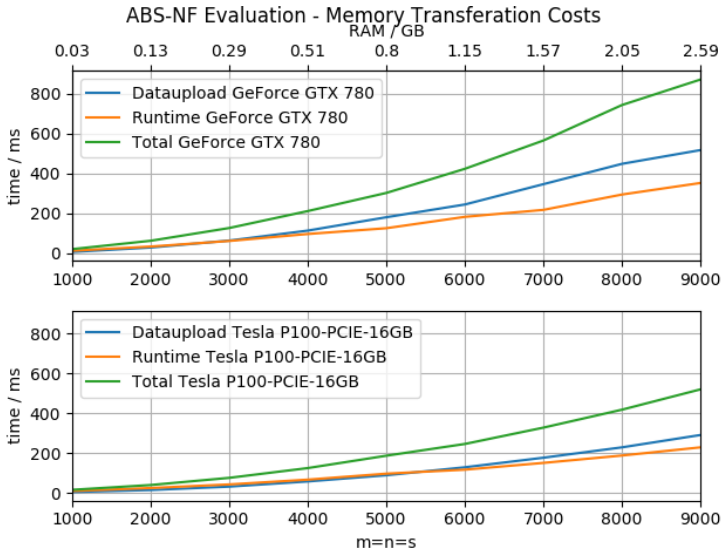

Figure 2: Datatransfer and execution time of the parallel implementation

11.2.3 Multiple Executions

Since the scenario of a single execution of the evaluate function is not quite realistic, we crafted a second experiment, where we uploaded the data to the devices and executed the code 1000 times. Here we only measured the pure execution time without dataupload, which should be marginal with a high enough number of executions. The results can be found in figure 3.

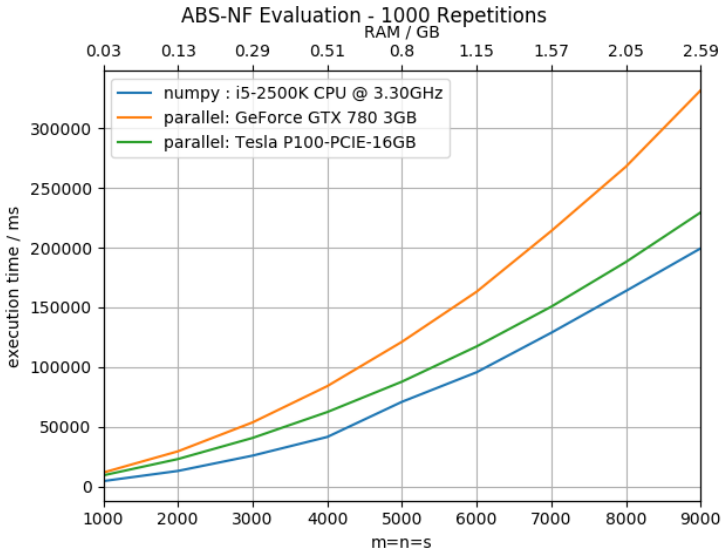

Figure 3: Multiple executions of the evaluate function on different devices

11.2.4 Analysis

The results of the experiment are heavy in favor of the serial implementation. For small data structures, 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 1 and fig. 3. If the data structures get big enoguh 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 2.

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

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

what can be implemented differently? What can be improved?

11.2.5 Notes

- Double Precision on GTX is nuts

12 Gradient

12.1 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:

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

The problem task is to implement a function that calculates γ as well as Γ.

12.2 Implementation

```
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,.);
```

```
24 //  d_gamma = d_gamma + K*a
25 cublasDgemv(.,d_K,., d_a,., d_gamma,.);
26 //  d_Gamma = d_Gamma + K*Z
27 cublasDgemm(.,d_K,d_Z,d_Gamma,m));
28 }
```

12.3 Performance

12.4 Analysis

12.5 Notes

13 Operations

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