# ABS-Normal Form
# An Implementation in CUDA C++

Matthias Mitterreiter
matthias.mitterreiter@uni-jena.de

Friedrich Schiller Universität Jena

August 2017

---

This project report of the abs-normal form implementation in CUDA C++
was created as part of the course

*Seminar Rechnerarchitektur - Grafikkarten*

supervised by

*Prof. Dr. Martin Bücker* *Dipl-Inf. Ralf Seidler* *Dr. Torsten Bosse*

---

**Abstract**

*The abs-normal form is a representation of piecewise linear functions. In this project we implemented basic routines to deal with these functions in CUDA C++. In particular we wanted to evaluate, solve and calculate the gradient of a given function in abs-normal form. We tested our implementation for correctness and made several performance benchmarks on multiple devices.*

# Contents

# 1 Introduction

TODO

- in jedem kapitel bezug auf introduction fragen nehmen
- Titlepage with graphics
- Plot of eval where mismeasured

This is an documentation and project review. blabla

## 1.1 The ABS-Normal Form

The abs-normal form is a representation of piecewise linear (PL) functions. Any PL function can be transformed and represented within this form. The process of transforming a PL function into abs-normal form is described in [2] and [3]. Applications and properties of the abs-normal form is not subject of this document but can be found in [3].

**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 $\Delta z$. $L$ 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 $|\Delta z|$.

## 1.2 Problems

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

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

## 1.3 Tasks

The aim of this project was to solve each of these problems by using parallel computing with CUDA C++ to boost performance. In particular, for each of the problems the following questions had to answered:

(I) How can the problem be solved theoretically?

(II) How can the problems be implemented with CUDA C++?

(III) Is there a benefit of using parallel computing?

(IV) How does the implementation perform on different devices?

## 1.4 Outline

In the following sections we answer each question for all of the given problems. Section 2 deals with the evaluation of functions in abs-normal form, where we show that a parallel implementation might not be beneficial. In section 3 we address the problem of calculating the gradient. Those results were quite promising. In section 4 we show one possibility of solving a function in abs-normal form.

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

In the appendix we listed all devices, that we used to benchmark our implementation B. Used software libraries are described in A and our notations as well abbrevations and symbols are documented in C.

# 2 Evaluation of a function in abs-normal form

## 2.1 Problem Specification

Given is a PL function in abs-normal form. The evaluation of it means calculating the vectors $\Delta y$ and $\Delta z$:

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

where the following structures are given:

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

In (3) $\Delta z$ depends on the element-wise absolute function of its own and therefore it cannot be calculated with a simple matrix vector dot product. The matrix $L$ is lower triangular and therefore $\Delta z$ 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 $\Delta z$ element-wise:

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

## 2.2 Implementation

Our implementation is 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. 1.

## 2.3 Performance Experiments

For our experiments and the subsequent analysis, we equalized the dimensions of given data-structures:

$$m = n = s$$

### 2.3.1 Single Execution of the evaluation function

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).

```
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 version of the CUDA C++ evaluation implementation

### 2.3.2 Multiple Executions of the evaluation function

Since the scenario of evaluating the function just once is not quite realistic, we did a second experiment, where we uploaded data onto the devices and executed the evaluation routine a 1000 times. Here we only measured the pure execution time, since the upload time should be marginal with a high enough number of executions. The results can be found in fig. 4.

## 2.4 Experiment Analysis

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 fits into the global memory of the device, we couldn't measure any performance gains through the CUDA implementation on given devices (fig 2 and fig. 4). The simplistic and serial numpy version even outperforms the CUDA version running on the tesla device.

If the data structures get that big, such that they don't fit into the global memory of the device, we can expect the parallel versions' performance to be even worse, since the data-transfer time takes a disproportional high share of the overall runtime (fig. 3).

The complexity of the evaluation function is $O(s^2)$. The memory also grows quadratic depending on the variable $s$ and therefore we get a memory complexity of $O(s^2)$, which may explain the results of the experiment.

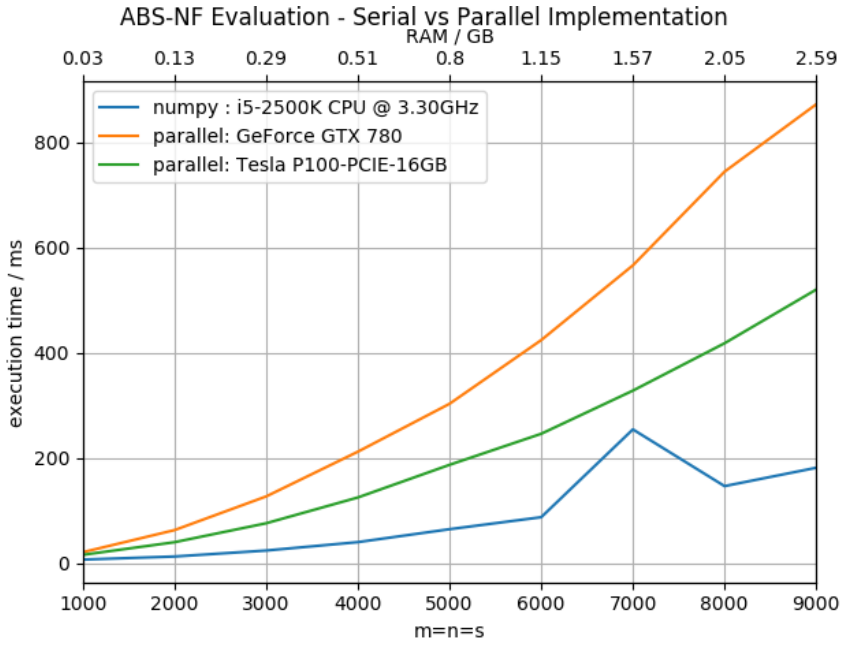


Figure 2: Single execution of the evaluation function on different devices.

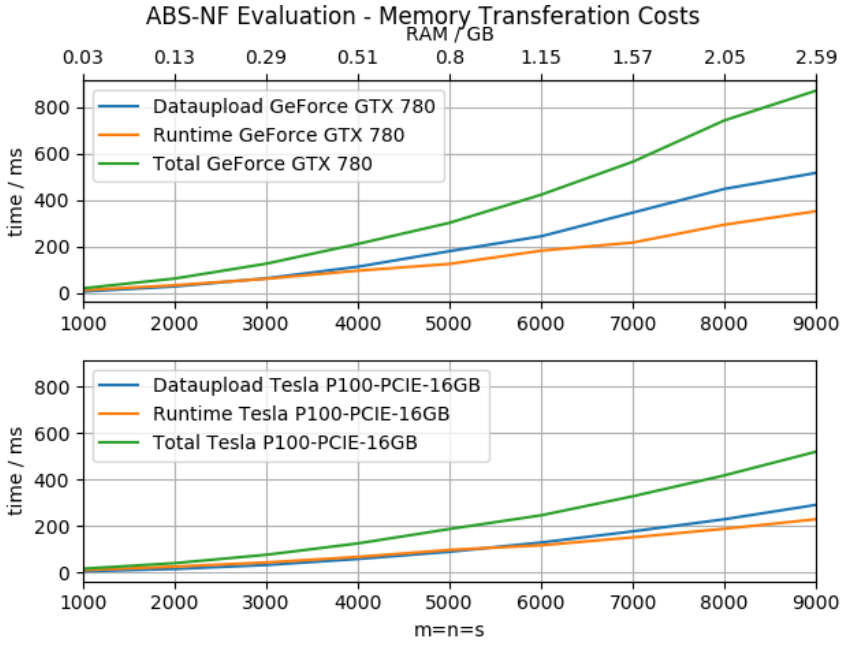


Figure 3: Data-transfer and execution time of the parallel implementation

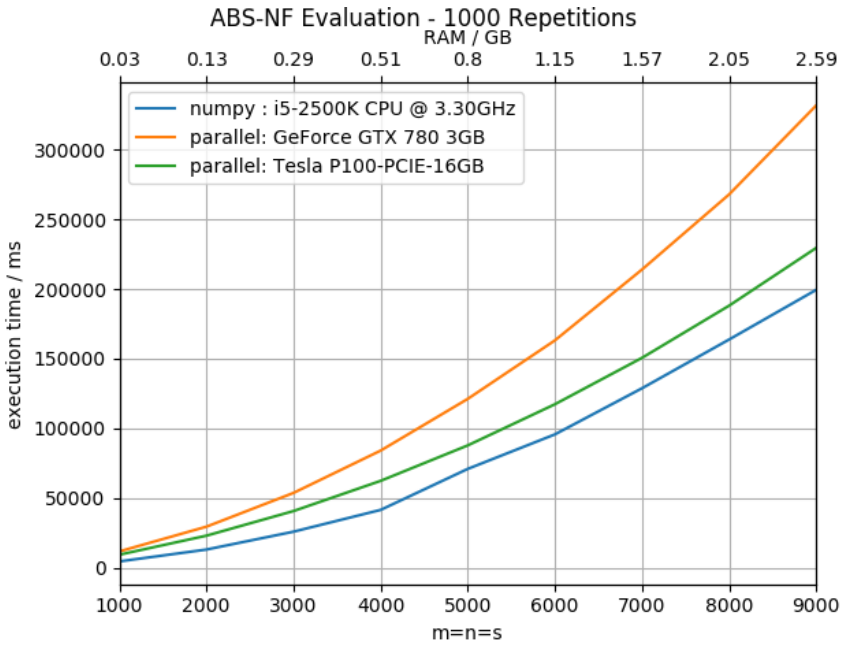


Figure 4: Multiple executions of the evaluation function on different devices

# 3 The gradient of a function in the abs-normal form

## 3.1 Problem Specification

The problem is to calculate the gradient of a PL function $f$ in abs-normal form, given the following structures:

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

Deducing a closed formula for the gradient is quite simple:

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

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

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

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

The gradient can now be calculated as:

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

## 3.2 Implementation

For the implementation we heavily relied on CUBLAS routines. A simplified version of the core-function can be found in fig. 5. Like in the implementation of the evaluate function, we require the the device to hold all the data-structures in global memory.

## 3.3 Performance Experiments

### 3.3.1 Single execution of the gradient function

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

### 3.3.2 Multiple executions of the gradient function

In this experiment we measured the runtime of the serial numpy implementation as well as the runtime of the parallel version for a 100 executions of the gradient function. 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).

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

Table 1: Runtime of executing the gradient function a 100 times of different devices

```
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

| 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

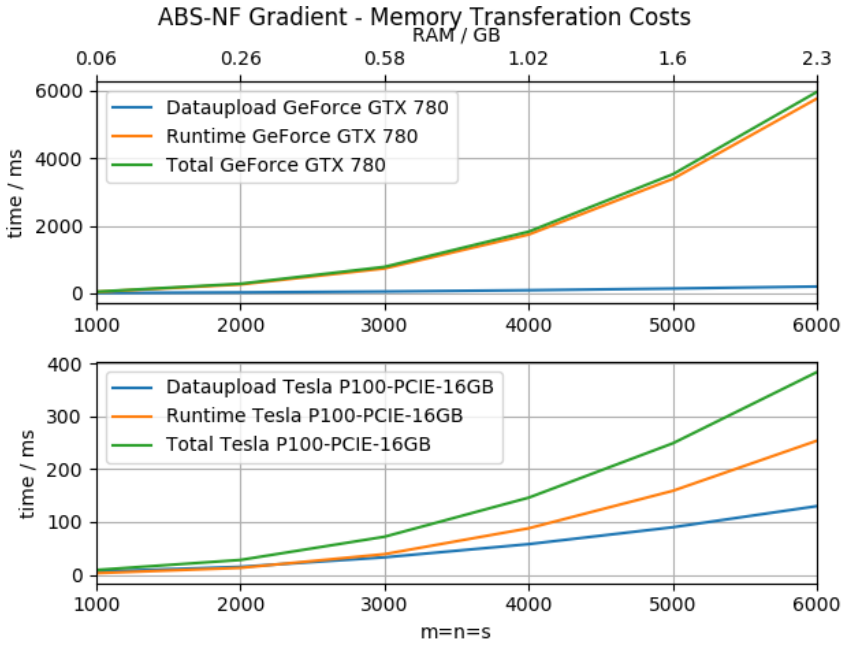


Figure 6: Single execution of the gradient function on different devices

## 3.4 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). Here 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 the 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 bigger sizes than the global device 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 self-implemented kernels "initTss", "multWithDz" and "initIdentity" do not preponderate.

We therefore conclude that the parallel version of the gradient function payed off quite nicely. Yet a optimized serial version , running on a CPU of comparable class like the Tesla should be done.

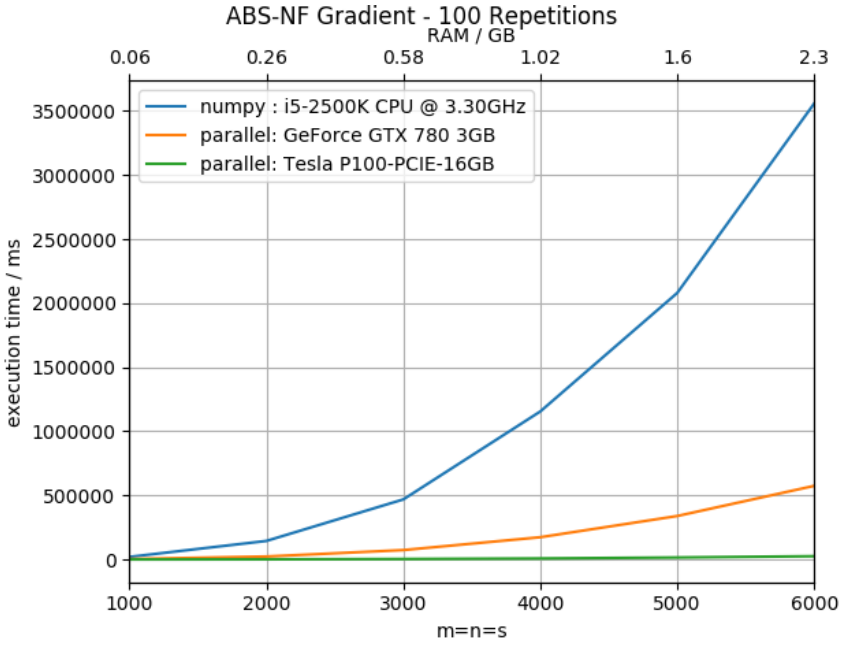


Figure 7: Multiple executions of the gradient function on different devices

# 4 Solve - The modulus iteration algorithm

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

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

We want to calculate:

$$\Delta x, \Delta z$$

## 4.1 Deducing a solution

In [3] Multiple solutions with different properties in convergence and complexity have been suggested. Here we focus on the algorithm that in [3] is called modulus iteration algorithm.
For deducing this algorithm we first and foremost assume:

$$\Delta y = 0$$

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

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

and replace $\Delta y$ with the zero-vector $O_m$. Now we can rearrange the equation system:

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

and obtain:

$$\Delta x = -J^{-1}(b + Y|\Delta z|) \tag{6}$$

For calculating $\Delta z$ we can now use (6):

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

Summarized:

$$\Delta z = c + S|\Delta z| \tag{7}$$
$$c = a - ZJ^{-1}b \tag{8}$$
$$S = L - ZJ^{-1}Y \tag{9}$$

We can use this result to construct a fix-point iteration algorithm where we recalculate $\Delta z$ in each step until convergence. In the following we focus onto the implementation of this algorithm.

## 4.2 Implementation

Implementing this algorithms means calculating (8) and (9) once and using these to repeatedly calculate (7). The key problem here is the calculation of $J^{-1}$, since this is the most expensive Operation. We decided to do a QR-decomposition of $J = QR$ and solving the linear equation system instead of calculating $J^{-1}$ directly. E.g. calculating $c$ is done in the following way:

1. $J = QR$
2. Solve $b = QRx$

$$QRx = b$$
$$x = solve(Rx = Qb)$$

3. Calculate $c$:

$$c = a - Zx$$

The calculation of $S$ follows the same pattern. After convergence $\Delta x$ can be calculated according to (6).

## 4.3 Performance Experiment

We did an experiment to benchmark the performance of our implementation. Obviously this only made sense as long as the results were correct. To verify this, we preceded as follows:

1. Randomly generate a function in abs-normal form: $a, b, Z, L, J, Y, \Delta x$ according to (1).
2. Evaluate given function to obtain: $\Delta y$ and $\Delta z$
3. Solve the system for $\Delta x$ and $\Delta z$ with the results of the previous step
4. Verify if the resulting $\Delta x$ and $\Delta z$ match the original ones.

We measured the runtime of the solve function in given context. To make sure, that each device works with the same data, we fixed the seed for the pseudo-random number generator. The results of this experiment can be found in fig 8.

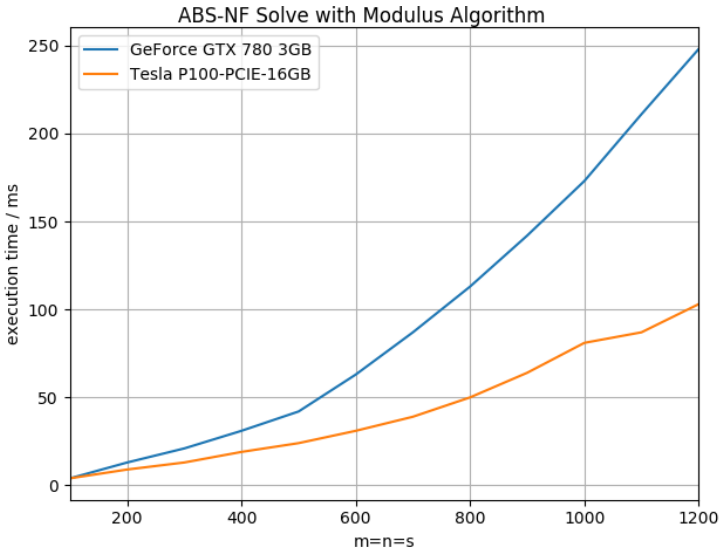


Figure 8: Runtime of the modulus solve implementation

## 4.4 Analysis and Notes

The implementation works correctly and the runtime results in fig. 8 behave as expected. We did not include the runtime of the serial version on purpose. This was for two reasons:

1. The numpy implementation calculates the inverse of $J$ directly.
2. We had to make sure, that each version operates on the exact same data, which we didn't

# 5 Conclusion

We implemented the basic operations to deal with functions in abs-normal form. Where the parallel version of the evaluation routine might not lead to additional performance, the parallel gradient and solve functions worked fast on given devices. For serious applications, the question of whether the CUDA implementation performs better than a serial version is still pending due to a lack of a competitive serial implementation. Additionally there are several more problems and questions that we didn't look into:

### 5.0.1 Sparsity

Some of the data-structures follow certain sparsity patters. Especially for operations where the global memory is a bottleneck (e.g. evaluation), we might gain additional performance by exploiting these patterns.

### 5.0.2 Precision

In our implementation we only worked with double precision. For devices like the NVIDIA GTX 780 , this has some major drawbacks, since its double precision power is only 1/24th of its FP32. Therefore a huge speed up is expected by switching from double to single precision or even mixed precision with the Tesla P100. Unfortunately this means reworking almost all of the core-routines, since libraries like cuBLAS and cuSOLVE offer different interfaces depending on the desired precision.

### 5.0.3 Kernel-tuning

In none of the implementations we could observe an loss in overall performance through the simplistic nature of our self implemented kernels. However for serious applications there is enough space to tune these kernels. Especially when also considering sparsity, an overhaul of these kernels will be necessary.

### 5.0.4 Multi device support

Our implementation works with only one device at a time. For all of our functions, its is possible to distribute tasks over multiple devices.

### 5.0.5 Memory-manager

For all of our functions, memory is a huge problem. We always required the device to hold all the data-structures in global memory, but this assumption does not scale. In this case one has to introduce an additional layer of abstraction, that deals with this problem.

# A Software Libraries

All the plots, prototypes and a serial implementations were done using Python 3.6 and the 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

# B Devices

We tested our code on the following devices, which were also used for performance benchmarks:

- NVIDIA Tesla P100 16GB RAM
- NVIDIA Geforce GTX 780, 3GB RAM
- Intel Core i5-2500 CPU, 16GB RAM

The device specification of the Tesla P100 model can be found in [1]. Also the compute capability of different NVIDIA GPU architectures is listed there.

# C Notation and Symbols

## C.1 Symbols

| Symbols | Description |
|---|---|
| $m, n, s$ | Dimensions of the data structures of a function in abs-normal (1). |
| $\Delta x$ | Vector $\Delta x \in \mathbb{R}^n$ |
| $\Delta z$ | Vector $\Delta z \in \mathbb{R}^s$ |
| $\Delta y$ | Vector $\Delta y \in \mathbb{R}^m$ |
| $a$ | Vector $a \in \mathbb{R}^s$ |
| $b$ | Vector $b \in \mathbb{R}^m$ |
| $Z$ | Matrix $Z \in \mathbb{R}^{s \times n}$ |
| $L$ | Lower triangular matrix $L \in \mathbb{R}^{s \times s}$ |
| $Y$ | $Y \in \mathbb{R}^{m \times s}$ |
| $Z$ | $Y \in \mathbb{R}^{m \times s}$, $J \in \mathbb{R}^{m \times n}$ |
| $\lvert \circ \rvert$ | the absolute value of $\circ$ if $\circ$ is a scalar and the element-wise absolute vector if $\circ$ is a vector. |

## C.2 Abbreviation

| Abbreviation | Description |
|---|---|
| Tesla | NVIDIA Tesla P100 16GB RAM |
| GTX | NVIDIA Geforce GTX 780, 3GB RAM |
| i-5 | Intel Core i5-2500 CPU, 16GB RAM |
| numpy | Serial implementation unsing python numpy |

# References

[1] N. Corporation. NVIDIA Tesla P100 whitepaper, 1999.

[2] A. Griewank. On stable piecewise linearization and generalized algorithmic differentiation. *Optimization Methods Software*, 28(6):1139–1178, Dec. 2013.

[3] A. Griewank, J. U. Bernt, M. Radons, and T. Streubel. Solving piecewise linear equations in abs-normal form. *ArXiv e-prints*, Jan. 2017.