1 Gliederung

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

1.0.1 Kapitel

- Problemstellung
- Implementierung
- Performance
- Improvements and bottlenecks

2 Introduction

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

3 Evalutation of the ABSNF

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

4 Gradient of the ABSNF

5 Grid and Blocksize

6 Solve

7 Final Thoughts

- Improvements
- view
- Multidevice support
- Grid and Blocksize

8 Anhang

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

9 Introcution

9.1 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 $|\Delta z|$.

9.2 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.

9.3 Additional Information

9.3.1 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

9.3.2 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

9.3.3 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.

9.3.4 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 ....

10 Evaluation of the ABSNF

10.1 Problem Specification

ABS-Normal Form:

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

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 straightforward. Since the matrix L is lower triangular, the vector Δz can be iteratively calculated, by taking the row-wise dotproduct of L and $|\Delta z|$.

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

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

10.2 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 [...].

10.2.1 Performance

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

$$m = n = s$$

10.2.2 Single Execution

code

10.2.3 Multiple Executions

code

- one iteration with memory
- multiple executions

10.2.4 Analysis

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

10.2.5 Notes

- Double Precision on GTX is nuts

11 Gradient

11.1 Problem Specification

ABS-Normal Form:

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

11.2 Implementation

11.3 Performance

11.4 Analysis

11.5 Notes

12 Operations

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