ABS-Normal Form

Eine Implementierung mit CUDA

Matthias Mitterreiter

Seminar Parallel Computing - FSU Jena
Prof. Martin Bücker, Dr. Torsten Bosse, Dipl-Inf. Ralf Seidler

June 30, 2017

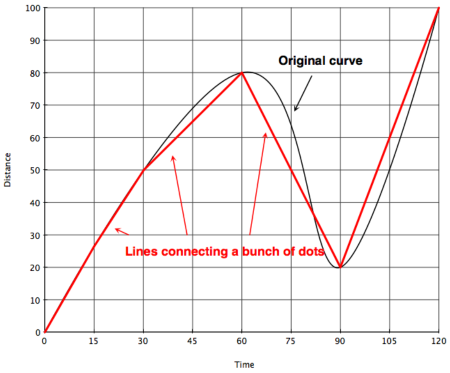
Einführung ABS-NF

Ausgangspunkt:

- Picewise smooth functions
- Picewise linear function

Ausgangspunkt:

- Picewise smooth functions
- Picewise linear function



ABS-Normal Form:

- Repräsentierung für Picewise linear functions (PL)
- Wird zur Approximierung von picewise smooth functions verwendet

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

Vorgehen:

1. (Theorem) Jede PL kann mithilfe von *min* und *max* ausgedrückt werden

Vorgehehen:

1. (Theorem) Jede PL kann mithilfe von *min* und *max* ausgedrückt werden
2. (Theorem) Jeder *min max* Ausdruck kann mit *abs* repäsentiert werden

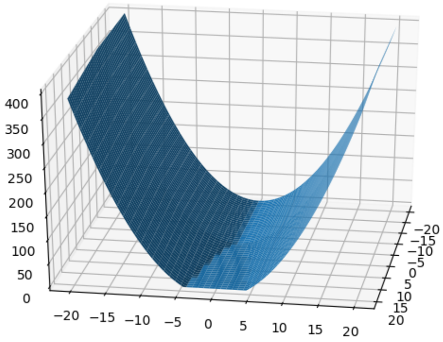
Das Ergebniss ist die ABS-NF einer PL Funktion.

Vorgehen:

1. (Theorem) Jede PL kann mithilfe von *min* und *max* ausgedrückt werden
2. (Theorem) Jeder *min max* Ausdruck kann mit *abs* repäsentiert werden
3. Picewise linearization wird durch algorithmisches differenzieren erreicht.

Das Ergebniss ist die ABS-NF einer PL Funktion.

$$F(x_1, x_2) = (x_2^2 - x_1^+)^+ \qquad F : \mathbb{R}^2 \to \mathbb{R}$$
$$(i)^+ = \max(0, i)$$



Nach der Transformation:

$$\left(\begin{array}{c}\Delta Z_1\\ \Delta Z_2\\ \hline \Delta Y\end{array}\right) = \left(\begin{array}{c} w_1\\ w_7 - \frac{1}{2}|w_1|\\ \hline \frac{1}{4}|w_1| - \frac{1}{2}|w_7|\end{array}\right) + \left(\begin{array}{cccc} 1 & 0 & 0 & 0\\ \frac{1}{2} & 0 & \frac{1}{2} & 0\\ \hline -\frac{1}{4} & w_2 & -\frac{1}{4} & \frac{1}{2}\end{array}\right) \times \left(\begin{array}{c}\Delta x_1\\ \Delta x_2\\ \hline |\Delta Z_1|\\ |\Delta Z_2|\end{array}\right)$$

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

Aufgaben

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

Aufgaben:

- Evaluate abs normal form:
  - Geg: $a, b, Z, L, J, Y, \Delta x$
  - Ges: $\Delta z, \Delta y$

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

Aufgaben:

- Evaluate abs normal form:
  - Geg: $a, b, Z, L, J, Y, \Delta x$
  - Ges: $\Delta z, \Delta y$
- Calculate Gradient
  - Geg: $a, b, Z, L, J, Y, \Delta Z$
  - Ges: Gradient γ, Γ

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

Aufgaben:

- Evaluate abs normal form:
  - Geg: $a, b, Z, L, J, Y, \Delta x$
  - Ges: $\Delta z, \Delta y$
- Calculate Gradient
  - Geg: $a, b, Z, L, J, Y, \Delta Z$
  - Ges: Gradient γ, Γ
- Solve abs-normal form system
  - Geg: $a, b, Z, L, J, Y, \Delta y$
  - Ges: $\Delta x, \Delta Z$

Programmiersprachen

- Python 3.5: Prototyping und Serial Performance benchmarks
- Cuda C++: Implementierung der parallelen ABSNF Aufgaben

Annahmen:

1. Global memory der GPU ist groß genug um alle benötigten Datenstrukturen zeitgleich zu halten
2. Daten werden vektorisiert übergeben
3. Sofern möglich mappe alle Problem auf existierende Librariers

Benutzte Libraries:

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

```
#include <cublas_v2.h>
#include <cusolverDn.h>
```

```
nvcc -std=c++11 x.cu -lcublas -lcusolver -o x
```

Evaluate

ABS-Normal Form:

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

Gegeben:

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

Gesucht:

$$\Delta z, \Delta y$$

ABS-Normal Form:

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

Gegeben:

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

Gesucht:

$$\Delta z, \Delta y$$

$$\Delta y = b + (J \times \Delta x) + (Y \times |\Delta z|)$$
$$\Delta z = a + (J \times \Delta x) + (L \times |\Delta z|)$$

ABS-Normal Form:

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

Gegeben:

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

Gesucht:

$$\Delta z, \Delta y$$

$$\Delta y = b + (J \times \Delta x) + (Y \times |\Delta z|)$$
$$\Delta z = a + (J \times \Delta x) + (L \times |\Delta z|)$$

Problem:

$$\Delta z = a + (J \times \Delta x) + (L \times |\Delta z|)$$

$$\begin{pmatrix} \Delta z_1 \\ \Delta z_2 \\ \Delta z_3 \\ \Delta z_4 \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ L_{2,1} & 0 & 0 & 0 \\ L_{3,1} & L_{3,2} & 0 & 0 \\ L_{4,1} & L_{4,2} & L_{4,3} & 0 \end{pmatrix} \times \begin{pmatrix} |\Delta z_1| \\ |\Delta z_2| \\ |\Delta z_3| \\ |\Delta z_4| \end{pmatrix}$$

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

$$\Delta z_1 = \underbrace{L_1 \times |\Delta z|}_{=0} + k_1 = k_1$$

$$
\begin{pmatrix} \Delta z_1 \\ \Delta z_2 \\ \Delta z_3 \\ \Delta z_4 \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ L_{2,1} & 0 & 0 & 0 \\ L_{3,1} & L_{3,2} & 0 & 0 \\ L_{4,1} & L_{4,2} & L_{4,3} & 0 \end{pmatrix} \times \begin{pmatrix} |\Delta z_1| \\ |\Delta z_2| \\ |\Delta z_3| \\ |\Delta z_4| \end{pmatrix}
$$

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

$$
\boxed{\Delta z_1} = \underbrace{L_1 \times |\Delta z|}_{=0} + k_1 = k_1
$$

$$
\begin{aligned}
\boxed{\Delta z_2} &= L_2 \times |\Delta z| + k_2 \\
&= L_{2,1} \times \boxed{|\Delta z_1|} + k_2
\end{aligned}
$$

$$\begin{pmatrix} \Delta z_1 \\ \Delta z_2 \\ \Delta z_3 \\ \Delta z_4 \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ L_{2,1} & 0 & 0 & 0 \\ L_{3,1} & L_{3,2} & 0 & 0 \\ L_{4,1} & L_{4,2} & L_{4,3} & 0 \end{pmatrix} \times \begin{pmatrix} |\Delta z_1| \\ |\Delta z_2| \\ |\Delta z_3| \\ |\Delta z_4| \end{pmatrix}$$

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

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

$$\begin{pmatrix} \Delta z_1 \\ \Delta z_2 \\ \Delta z_3 \\ \Delta z_4 \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ L_{2,1} & 0 & 0 & 0 \\ L_{3,1} & L_{3,2} & 0 & 0 \\ L_{4,1} & L_{4,2} & L_{4,3} & 0 \end{pmatrix} \times \begin{pmatrix} |\Delta z_1| \\ |\Delta z_2| \\ |\Delta z_3| \\ |\Delta z_4| \end{pmatrix}$$

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

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

Aufgabe 1) Evaluate

Implementierung

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

Speicherkomplexität:

a	b	Z	L	J	Y	Δy	Δx	Δz	$\|\Delta z\|$
s	m	$s*n$	$s*s$	$m*n$	$m*s$	m	n	s	s

$$(s^2 + (3+m+n)*s + (m+2)m + n) * sizeof(T)$$

Speicherkomplexität:

a	b	Z	L	J	Y	Δy	Δx	Δz	$\lvert\Delta z\rvert$
s	m	$s*n$	$s*s$	$m*n$	$m*s$	m	n	s	s

$$(s^2 + (3 + m + n) * s + (m + 2)m + n) * sizeof(T)$$

Seien

- $m = 1000, n = 1000, s = 1000$
- Datatype: $double \approx 8 bytes$
- 32.048.000 Bytes $\approx 0.032048 GB$

Speicherkomplexität:

a	b	Z	L	J	Y	Δy	Δx	Δz	$\lvert\Delta z\rvert$
s	m	$s*n$	$s*s$	$m*n$	$m*s$	m	n	s	s

$$(s^2 + (3 + m + n) * s + (m + 2)m + n) * sizeof(T)$$

Seien

- $m = 1000, n = 1000, s = 1000$
- Datatype: $double \approx 8 bytes$
- 32.048.000 Bytes $\approx 0.032048 GB$

Seien

- $m = 1000, n = 1000, s = 100.000$
- Datatype: $double \approx 8 bytes$
- 81.610.424.000 Bytes $\approx 81.610 GB$

Komplexität:

Funktion	Komplexität Seriell	Komplexität Parallel
$cudaMemcpy(dz, a)$	s	s/p
$cublasDgemv(Z, dx, dz)$	$s * n$	$(s * n)/p$
$cublasDgemv(L, \|dz\|)$	$s * s$	$(s * s)/p$
$cublasMemcpy(dy, b)$	m	m/p
$cublasDgemv(J, dx, dy)$	$m * n$	$(m * n)/p$
$cublasDgemv(Y, \|dz\|, dy)$	$m * s$	$(m * s)/p$

Komplexität:

Funktion	Komplexität Seriell	Komplexität Parallel
$cudaMemcpy(dz, a)$	s	s/p
$cublasDgemv(Z, dx, dz)$	$s * n$	$(s * n)/p$
$cublasDgemv(L, \|dz\|)$	$s * s$	$(s * s)/p$
$cublasMemcpy(dy, b)$	m	m/p
$cublasDgemv(J, dx, dy)$	$m * n$	$(m * n)/p$
$cublasDgemv(Y, \|dz\|, dy)$	$m * s$	$(m * s)/p$

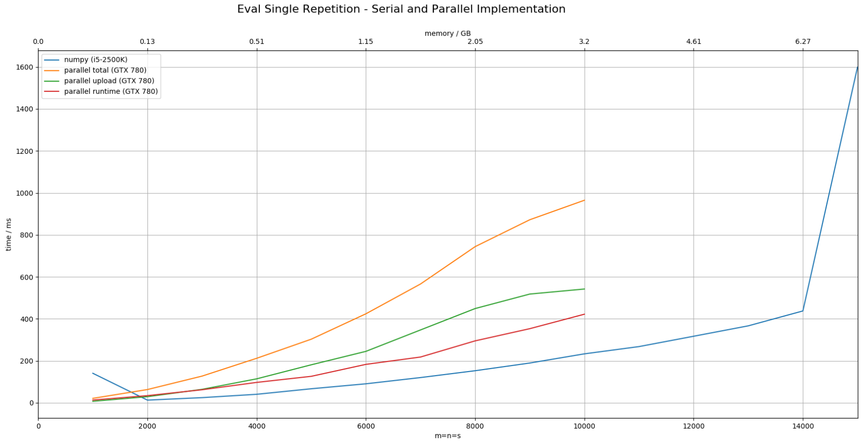
Der Rechenaufwand steigt im selben Maße wie der Speicheraufwand !

Komplexität:

Funktion	Komplexität Seriell	Komplexität Parallel
$cudaMemcpy(dz, a)$	s	s/p
$cublasDgemv(Z, dx, dz)$	$s * n$	$(s * n)/p$
$cublasDgemv(L, \|dz\|)$	$s * s$	$(s * s)/p$
$cublasMemcpy(dy, b)$	m	m/p
$cublasDgemv(J, dx, dy)$	$m * n$	$(m * n)/p$
$cublasDgemv(Y, \|dz\|, dy)$	$m * s$	$(m * s)/p$

Der Rechenaufwand steigt im selben Maße wie der Speicheraufwand !

Vermutung, parallelisieren bringt hier nicht viel!



Eval Single Repetition - Serial and Parallel Implementation
memory / GB
0.0
0.13
0.51
1.15
2.05
3.2
4.61
6.27
numpy (i5-2500K)
parallel total (GTX 780)
parallel upload (GTX 780)
parallel runtime (GTX 780)
time / ms
1600
1400
1200
1000
800
600
400
200
0
0
2000
4000
6000
8000
10000
12000
14000
m=n=s

Eval 1000 Repetitions - Serial and Parallel Implementation
memory / GB
0.0
0.03
0.13
0.29
0.51
0.8
1.15
1.57
2.05
2.59
numpy : i5-2500K CPU @ 3.30GHz
parallel: GeForce GTX 780
time / ms
300000
250000
200000
150000
100000
50000
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
m=n=s

Gradient

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

Gegeben:

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

Gesucht:

$$\gamma, \Gamma$$

Wobei:

$$\gamma = b + Y\Sigma(I - L\Sigma)^{-1} a$$
$$\Gamma = J + Y\Sigma(I - L\Sigma)^{-1} Z$$

$$\Sigma = Diag(Sign(\Delta z))$$

Brauchen:

$$\Sigma(I - L\Sigma)^{-1}$$

$$\Sigma = Diag(Sign(\Delta z))$$

Fallstricken:

Brauchen:

$$\Sigma(I - L\Sigma)^{-1}$$

$$\Sigma = Diag(Sign(\Delta z))$$

Fallstricken:

- Sparse Matrix Σ

Brauchen:

$$\Sigma(I - L\Sigma)^{-1}$$

$$\Sigma = Diag(Sign(\Delta z))$$

Fallstricken:

- Sparse Matrix Σ
- Inverse $(I - L\Sigma)^{-1}$

Sei:

$$\Delta z = [-3, 0, 4, -1]$$

Dann gilt für $I - L\Sigma$:

$$I - \begin{pmatrix} 0 & 0 & 0 & 0 \\ L_{2,1} & 0 & 0 & 0 \\ L_{3,1} & L_{3,2} & 0 & 0 \\ L_{4,1} & L_{4,2} & L_{4,3} & 0 \end{pmatrix} \times \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -L_{2,1} & 1 & 0 & 0 \\ -L_{3,1} & 0 & 1 & 0 \\ -L_{4,1} & 0 & 0 & 1 \end{pmatrix}$$

Sei:

$$\Delta z = [-3, 0, 4, -1]$$

Dann gilt für $I - L\Sigma$:

$$I - \begin{pmatrix} 0 & 0 & 0 & 0 \\ L_{2,1} & 0 & 0 & 0 \\ L_{3,1} & L_{3,2} & 0 & 0 \\ L_{4,1} & L_{4,2} & L_{4,3} & 0 \end{pmatrix} \times \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -L_{2,1} & 1 & 0 & 0 \\ -L_{3,1} & 0 & 1 & 0 \\ -L_{4,1} & 0 & 0 & 1 \end{pmatrix}$$

das entspricht folgenden Operationen:

- Hinzufügen einer Hauptdiagonalen
- Skalieren der Spalten von L mit den Vorzeichen von Δz

Sei:

$$\Delta z = [-3, 0, 4, -1]$$

Dann gilt für $I - L\Sigma$:

$$I - \begin{pmatrix} 0 & 0 & 0 & 0 \\ L_{2,1} & 0 & 0 & 0 \\ L_{3,1} & L_{3,2} & 0 & 0 \\ L_{4,1} & L_{4,2} & L_{4,3} & 0 \end{pmatrix} \times \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -L_{2,1} & 1 & 0 & 0 \\ -L_{3,1} & 0 & 1 & 0 \\ -L_{4,1} & 0 & 0 & 1 \end{pmatrix}$$

das entspricht folgenden Operationen:

- Hinzufügen einer Hauptdiagonalen
- Skalieren der Spalten von L mit den Vorzeichen von Δz

Besser dieses als Operation zu implementieren. Das Auflösen der unteren Dreiecksmatrix $(I - L\Sigma)^{-1}$ übernimmt CUBLAS.

```
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) // room for improvement, operations can be merged
  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));
}
```

Speicherkomplexität:

a	b	Z	L	J	Y	Δz	γ	Γ	Tss	I	K
s	m	$s*n$	$s*s$	$m*n$	$m*s$	s	m	$m*n$	$s*s$	$s*s$	$m*s$

Bei $m = n = s$:

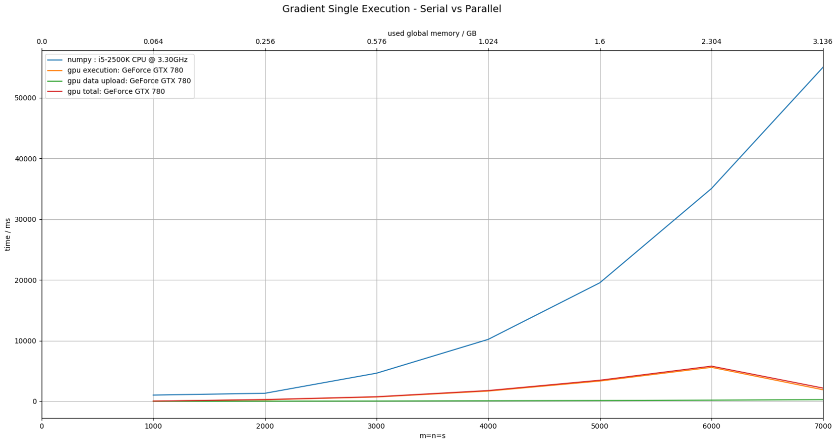
$$8s^2 + 4s \times sizeof(type)$$

- $m = n = s = 1000$: 0.064 GB
- $m = n = s = 5000$: 1.6 GB
- $m = n = s = 10.000$: 6.40 GB

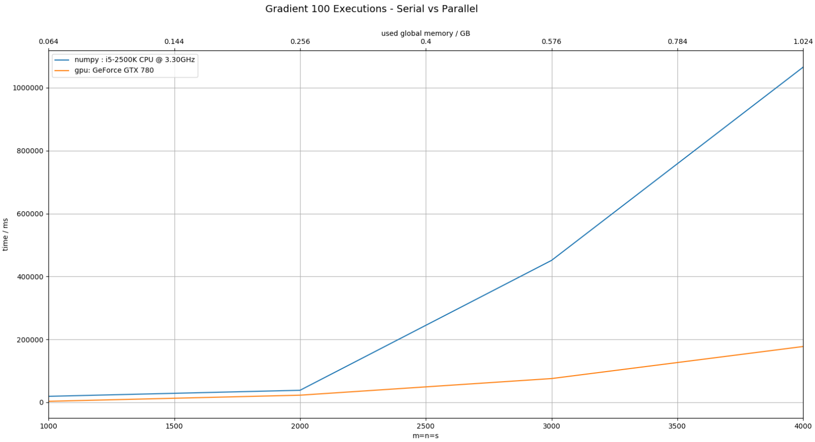
Komplexität ($m = n = s$):

Funktion	Komplexität Seriell
$initTss()$	s^2
$initIdentity()$	s^2
$getTriangularInverse()$	s^2 (backsubstitution)
$multWithDz()$	s^2
$cublasDgemm()$	s^3
$cublasDgemv()$	s^2
$cudaMemcpy()$	s

Lässt sich alles gut parallelisieren.



Gradient Single Execution - Serial vs Parallel
used global memory / GB
0.0
0.064
0.256
0.576
1.024
1.6
2.304
3.136
numpy : i5-2500K CPU @ 3.30GHz
gpu execution: GeForce GTX 780
gpu data upload: GeForce GTX 780
gpu total: GeForce GTX 780
time / ms
0
10000
20000
30000
40000
50000
0
1000
2000
3000
4000
5000
6000
7000
m=n=s



Gradient 100 Executions - Serial vs Parallel
used global memory / GB
0.064
0.144
0.256
0.4
0.576
0.784
1.024
numpy : i5-2500K CPU @ 3.30GHz
gpu: GeForce GTX 780
1000000
800000
600000
400000
200000
0
time / ms
1000
1500
2000
2500
3000
3500
4000
m=n=s

Blocksize und Gridsize?

Wie sollen gridsize and blocksize gewählt werden?

Wie sollen gridsize and blocksize gewählt werden?

- Generischen Ansatz

Wie sollen gridsize and blocksize gewählt werden?

- Generischen Ansatz
- Starte mit gewisser blocksize and gridsize in abh. von device spec.

Wie sollen gridsize and blocksize gewählt werden?

- Generischen Ansatz
- Starte mit gewisser blocksize and gridsize in abh. von device spec.
- threads berechnen, welche aufgaben sie abarbeiten sollen

Wie sollen gridsize and blocksize gewählt werden?

- Generischen Ansatz
- Starte mit gewisser blocksize and gridsize in abh. von device spec.
- threads berechnen, welche aufgaben sie abarbeiten sollen
- über die optimalen parameter kann optimiert werden.

Zu Implementierende Operation:

$$A = A \times Diag(Sign(dz))$$

Zu Implementierende Operation:

$$A = A \times Diag(Sign(dz))$$

Beispiel:

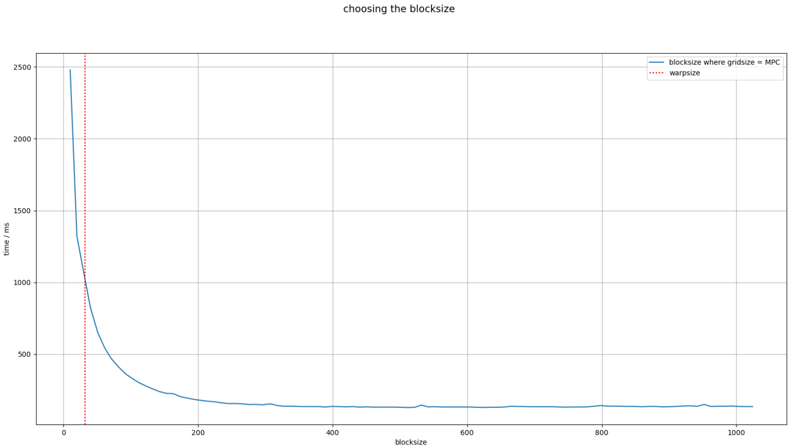
$$A \in \mathbb{R}^{3\times 3}, dz \in \mathbb{R}^2$$

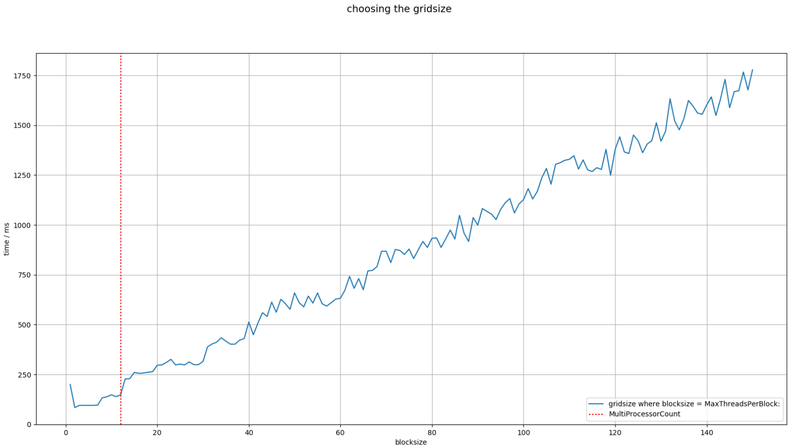
$$dz = (-j, 0, k)$$

$$\begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix} \times \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -a & 0 & g \\ -b & 0 & h \\ -c & 0 & i \end{pmatrix}$$

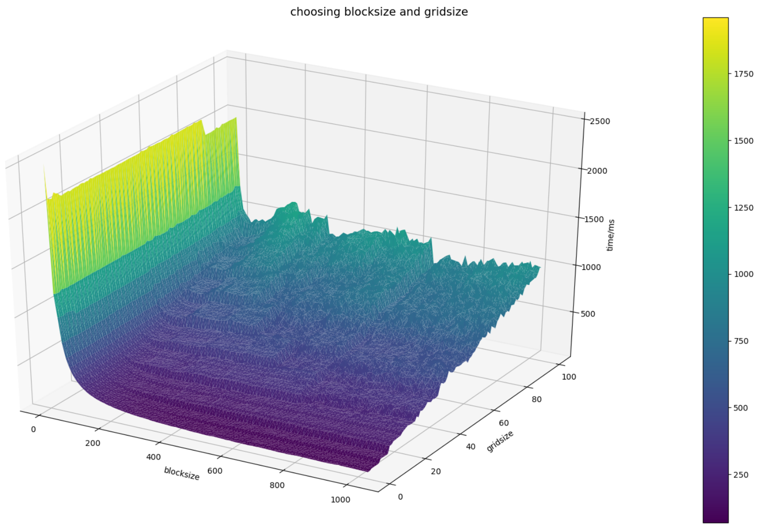
ANIMATION

```
template <typename T>
void __global__ multWithDz(T *A, T *dz, int s){
  int i = threadIdx.x;
  int j = blockIdx.x;
  int id = i*s + j;
  while(id < s*s && j < s){
    if(i<s){
      if(A[id] != T(0))
        A[id] = A[id] * cuutils::sign(&dz[j]);
      i+=blockDim.x;
    }
    else{
      i = i%s;
      j = j + gridDim.x;
    }
    id = i*s + j;
  }
}
```


choosing the blocksize
blocksize where gridsize = MPC
warpsize
2500
2000
1500
1000
500
time / ms
0
200
400
600
800
1000
blocksize



choosing the gridsize
time / ms
blocksize
gridsize where blocksize = MaxThreadsPerBlock:
MultiProcessorCount
0
250
500
750
1000
1250
1500
1750
0
20
40
60
80
100
120
140



choosing blocksize and gridsize
blocksize
0
200
400
600
800
1000
gridsize
0
20
40
60
80
100
time/ms
500
1000
1500
2000
2500
250
500
750
1000
1250
1500
1750

Solve

Gegben:

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

Gesucht:

$$\Delta x, \Delta z$$

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

Wollen Lösen:

$$\Delta z = a + Z\Delta x + L|\Delta z|$$
$$\Delta y = b + J\Delta x + Y|\Delta z|$$

Ausgangslage:

$$\Delta y = 0$$

Andernfalls

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

Wollen Lösen:

$$\Delta z = a + Z\Delta x + L|\Delta z|$$
$$\Delta y = b + J\Delta x + Y|\Delta z|$$

Ausgangslage:

$$\Delta y = 0$$

Andernfalls

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

Umstellen nach Δx:

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

Wollen Lösen:

$$\Delta z = a + Z\Delta x + L|\Delta z|$$
$$\Delta y = b + J\Delta x + Y|\Delta z|$$

Haben:

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

Wollen Lösen:

$$\Delta z = a + Z\Delta x + L|\Delta z|$$
$$\Delta y = b + J\Delta x + Y|\Delta z|$$

Haben:

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

Einsetzen in

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

Wollen Lösen:

$$\Delta z = a + Z\Delta x + L|\Delta z|$$
$$\Delta y = b + J\Delta x + Y|\Delta z|$$

Haben:

$$\Delta x = -J^{-1}(b + Y|\Delta z|)$$
$$\Delta z = a - ZJ^{-1}b - (ZJ^{-1}Y - L)|\Delta z|$$

Modularisieren:

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

$$c = a - ZJ^{-1}b$$
$$S = L - ZJ^{-1}Y$$

Problem:

$$\Delta z = c + S|\Delta z|$$

Problem:

$$\Delta z = c + S|\Delta z|$$

Lösung mithilfe Fixpunktiteration:

1. Generalized (Pseudo) Newton
2. Block-Seidel Algorithmus
3. Modulus Iteration Algorithmus

Konvergieren unter gewissen Konvergenzkrieterien (linear / endlich).

Modulus Iteration Algorithmus:

$\Delta z = \mathrm{Init}()$
$c = a - ZJ^{-1}b$
$S = L - ZJ^{-1}Y$
while not converged:
    $\Delta_z = c + S|\Delta z|$
$\Delta x = -J^{-1}(b + Y|\Delta z|)$

Notes:

- Problem ist die Berechnung von c und S
- J nicht singulär.

Für

$$S = L - ZJ^{-1}Y$$

QR - Zerlegung:

$$\begin{aligned} J^{-1}Y &= X \\ Y &= JX \\ Y &= QRX \\ QY &= RX \end{aligned}$$

Benutze triangular solve!
Berechne:

$$S = L - ZX$$

Komplexität ($m = n = s$) bei k Iterationen

Funktion	Komplexität Seriell
cusolverDnDgeqrf	s^3 (QR)
cusolverDnDormqr	s^2 (QRxB)
cublasDgemm()	s^3
cublasDgemv()	$s^2 * k$
cudaMemcpy()	$s * k$

Lässt sich alles gut parallel ausführen !!!

Final Thoughts

Was ist da nach der ersten Implementation?

Was ist da nach der ersten Implementation?

```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                          25            285            486           7451
C/C++ Header                     5             82            209           1256
C++                              7             41             19            334
-------------------------------------------------------------------------------
SUM:                            37            408            714           9041
-------------------------------------------------------------------------------
```

Final Thoughts

Was ist da?

Was ist da nach der ersten Implementation?

```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                          25            285            486           7451
C/C++ Header                     5             82            209           1256
C++                              7             41             19            334
-------------------------------------------------------------------------------
SUM:                            37            408            714           9041
-------------------------------------------------------------------------------
```

Dabei:

- Working prototype in CUDA C++ und Python
- Unittests
- Coole Plot Generatoren

Was fehlt:

Was fehlt:

- Refactoring

Was fehlt:

- Refactoring
- Funktionierende Implementierung des Gen. Newton solvers

Was fehlt:

- Refactoring
- Funktionierende Implementierung des Gen. Newton solvers
- Useability

Was fehlt:

- Refactoring
- Funktionierende Implementierung des Gen. Newton solvers
- Useability
- Anwendung

Was fehlt:

- Refactoring
- Funktionierende Implementierung des Gen. Newton solvers
- Useability
- Anwendung
- Numerische Checks der Ergebnisse bei größeren Daten

Was fehlt:

- Refactoring
- Funktionierende Implementierung des Gen. Newton solvers
- Useability
- Anwendung
- Numerische Checks der Ergebnisse bei größeren Daten
- Speichermanager

Was fehlt:

- Refactoring
- Funktionierende Implementierung des Gen. Newton solvers
- Useability
- Anwendung
- Numerische Checks der Ergebnisse bei größeren Daten
- Speichermanager
- Spezielle Wahl für Gridsize und Blocksize

Was fehlt:

- Refactoring
- Funktionierende Implementierung des Gen. Newton solvers
- Useability
- Anwendung
- Numerische Checks der Ergebnisse bei größeren Daten
- Speichermanager
- Spezielle Wahl für Gridsize und Blocksize
- Multidevice Support

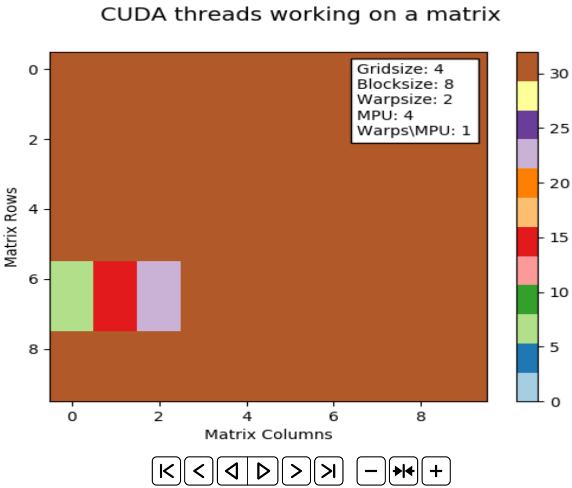
Was fehlt:

- Refactoring
- Funktionierende Implementierung des Gen. Newton solvers
- Useability
- Anwendung
- Numerische Checks der Ergebnisse bei größeren Daten
- Speichermanager
- Spezielle Wahl für Gridsize und Blocksize
- Multidevice Support
- Sparsity

Was fehlt:

- Refactoring
- Funktionierende Implementierung des Gen. Newton solvers
- Useability
- Anwendung
- Numerische Checks der Ergebnisse bei größeren Daten
- Speichermanager
- Spezielle Wahl für Gridsize und Blocksize
- Multidevice Support
- Sparsity
- Math Tuning

- Archiv Torsten Bosse
- Linear Algebra and its Applications - Griewank
- Cuda DOC
- $https://castingoutnines.wordpress.com/2010/01/12/piecewise-linear-calculus-part-2-getting-to-smoothness/$


CUDA threads working on a matrix
Gridsize: 4
Blocksize: 8
Warpsize: 2
MPU: 4
Warps\MPU: 1
Matrix Rows
Matrix Columns
0
2
4
6
8
0
5
10
15
20
25
30