

ABS-Normal Form

Eine Implementierung mit CUDA



Matthias Mitterreiter

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

July 9, 2017

1. ABS-NF Einführung
2. Aufgabenbeschreibung
3. Evaluate
4. Gradient
5. Blocksize and Gridsize
6. Solve
7. Final Thoughts

Einführung ABS-NF

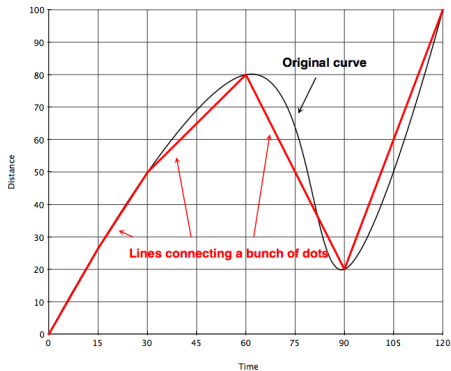
-
1. **Einführung**
 2. Aufgaben
 3. Evaluate
 4. Gradient
 5. Blocksize und Gridsize
 6. Solve
 7. Final Thoughts

Ausgangspunkt:

- Picewise smooth functions
- Picewise linear function

Ausgangspunkt:

- Piecewise smooth functions
- Piecewise linear function



ABS-Normal Form:

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

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

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

Erzeugung der ABS-NF:

- ④ (Theorem) Jede PL kann mithilfe von *min* und *max* ausgedrückt werden.

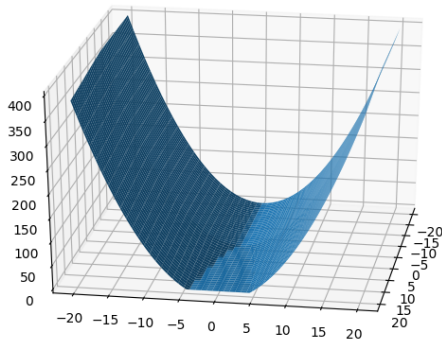
Erzeugung der ABS-NF:

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

Erzeugung der ABS-NF:

- 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.

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



Nach der Transformation:

$$\begin{pmatrix} \Delta Z_1 \\ \Delta Z_2 \\ \Delta Y \end{pmatrix} = \begin{pmatrix} w_1 \\ w_7 - \frac{1}{2}|w_1| \\ \frac{1}{4}|w_1| - \frac{1}{2}|w_7| \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ -\frac{1}{4} & w_2 & -\frac{1}{4} & \frac{1}{2} \end{pmatrix} \times \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ |\Delta Z_1| \\ |\Delta Z_2| \end{pmatrix}$$

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

Aufgaben

-
1. Einführung
 2. **Aufgaben**
 3. Evaluate
 4. Gradient
 5. Blocksize und Gridsize
 6. Solve
 7. Final Thoughts

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

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
 - 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 ABS-NF 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 Libraries

Benutzte Libraries:

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

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

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

Evaluate

-
1. Einführung
 2. Aufgaben
 3. **Evaluate**
 4. Gradient
 5. Blocksize und Gridsize
 6. Solve
 7. Final Thoughts

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:

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

Aufgabe 1) Evaluate

Löse $z = f(|\Delta z|)$



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

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

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

Aufgabe 1) Evaluate

Löse $z = f(|\Delta z|)$



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

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

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

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

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

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

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

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

$$\begin{aligned} \Delta z_3 &= L_3 \times |\Delta z| + k_3 \\ &= L_{3,1} \times |\Delta z_1| + L_{3,2} \times |\Delta z_2| + k_3 \end{aligned}$$

Aufgabe 1) Evaluate

Löse $z = f(|\Delta z|)$



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

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

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

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

$$\begin{aligned} \Delta z_3 &= L_3 \times |\Delta z| + k_3 \\ &= L_{3,1} \times |\Delta z_1| + L_{3,2} \times |\Delta z_2| + k_3 \end{aligned}$$

$$\begin{aligned} \Delta z_4 &= L_4 \times |\Delta z| + k_4 \\ &= L_{4,1} \times |\Delta z_1| + L_{4,2} \times |\Delta z_2| + L_{4,3} \times |\Delta z_3| + k_4 \end{aligned}$$

```
1  template <typename T>
2  void eval(T *a, T *b,
3           T *Z, T *L,
4           T *J, T *Y,
5           T *dx,
6           int m, int n, int s,
7           T *dz, T *dy,
8           T *abs_dz)
9  {
10     // dz = a
11     cudaMemcpy(dz, a, ., cudaMemcpyDeviceToDevice));
12     // dz = Z * dx + dx
13     cublasDgemv(.,Z, ., dx, . dz, .)
14     // dz[i] = L[i]_i * |dz|_i
15     for(int i=0; i<s; i++)
16     {
17         cublasDgemv(.,&L[i * s], .,abs_dz, ., &dz[i],.);
18         abs <<<1,1>>>(&dz[i], &abs_dz[i], 1);
19     }
20     // dy = b
21     cudaMemcpy(dy, b, ., cudaMemcpyDeviceToDevice);
22     // dy = dy + J*dx
23     cublasDgemv(.,J, ., dx, ., dy, .));
24     // dy = dy + Y * |dz|
25     cublasDgemv(., Y, ., abs_dz, ., dy, .));
26 }
```

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

$$(s^2 + (3 + m + n) * s + (m + 2)m + n) * \text{sizeof}(T)$$

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

$$(s^2 + (3 + m + n) * s + (m + 2)m + n) * \text{sizeof}(T)$$

Seien

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

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

$$(s^2 + (3 + m + n) * s + (m + 2)m + n) * \text{sizeof}(T)$$

Seien

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

Seien

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

Komplexität:

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

Komplexität:

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

Der Rechenaufwand steigt im selben Maße wie der Speicheraufwand !

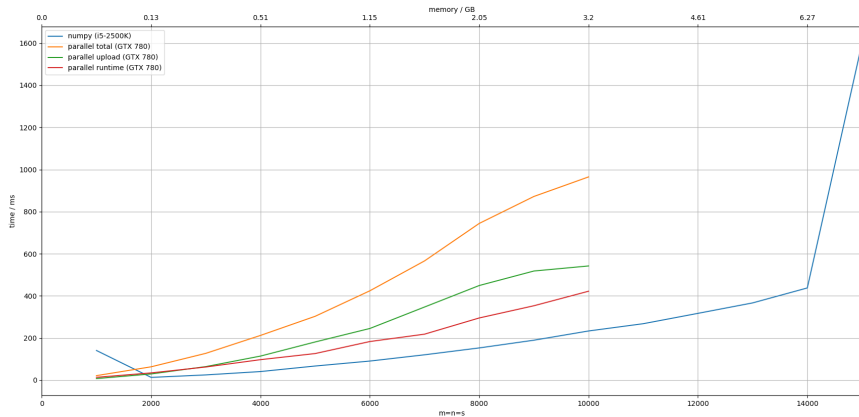
Komplexität:

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

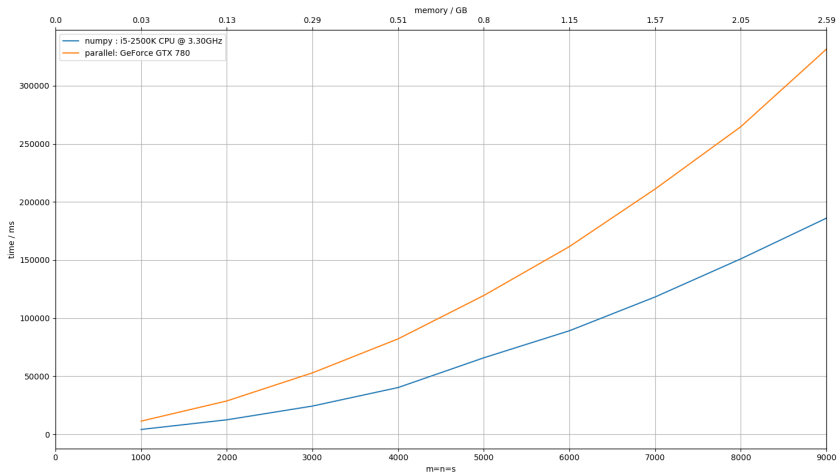
Der Rechenaufwand steigt im selben Maße wie der Speicheraufwand !

Vermutung, parallelisieren bringt hier nicht viel!

Eval Single Repetition - Serial and Parallel Implementation



Eval 1000 Repetitions - Serial and Parallel Implementation



Gradient

-
1. Einführung
 2. Aufgaben
 3. Evaluate
 4. **Gradient**
 5. Blocksize und Gridsize
 6. Solve
 7. Final Thoughts

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

Gegeben:

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

Gesucht:

$$\gamma, \Gamma$$

Wobei:

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

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

$$\Sigma = \text{Diag}(\text{Sign}(\Delta z))$$

Brauchen:

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

$$\Sigma = \text{Diag}(\text{Sign}(\Delta z))$$

Fallstricken:

Brauchen:

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

$$\Sigma = \text{Diag}(\text{Sign}(\Delta z))$$

Fallstricken:

- Sparse Matrix Σ

Brauchen:

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

$$\Sigma = \text{Diag}(\text{Sign}(\Delta z))$$

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 den 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 den folgenden Operationen:

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

Kann als lineare Operation implementiert werden. Das Auflösen der unteren Dreiecksmatrix $(I - L\Sigma)^{-1}$ übernimmt CUBLAS.

```
1  template <typename T>
2  void gradient(T *a, T *b,
3              T *Z, T *L,
4              T *J, T *Y,
5              T *dz,
6              T *Tss, T *I, T *K,
7              int m, int n, int s,
8              int gridsize, int blocksize,
9              T *gamma, T *Gamma)
10 // d_Tss = diag(1) - L * diag(sign(dz))
11 initTss <<<gridsize, blocksize >>>(d_Tss,d_L, d_dz, s, s*s);
12 // d_I = diag(1) // room for improvement, operations can be merged
13 initIdentity <<<gridsize, blocksize >>> (d_I, s);
14 // d_I = d_Tss * X
15 getTriangularInverse(handle, d_Tss, d_I, s);
16 // d_I = d_I * diag(sign(dz))
17 multWithDz <<<gridsize, blocksize >>>(d_I, d_dz, s);
18 // d_K = d_Y * d_I
19 cublasDgemm(.,d_Y,.,d_I,d_K,);
20 // d_gamma = d_b
21 // d_Gamma = J
22 cudaMemcpy(d_gamma, d_b,.);
23 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 }
```

Speicherkomplexität:

a	b	Z	L	J	Y	Δz	γ	Γ	T_{ss}	I	K
s	m	$s * n$	$s * s$	$m * n$	$m * s$	s	m	$m * n$	$s * s$	$s * s$	$m * s$

Bei $m = n = s$:

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

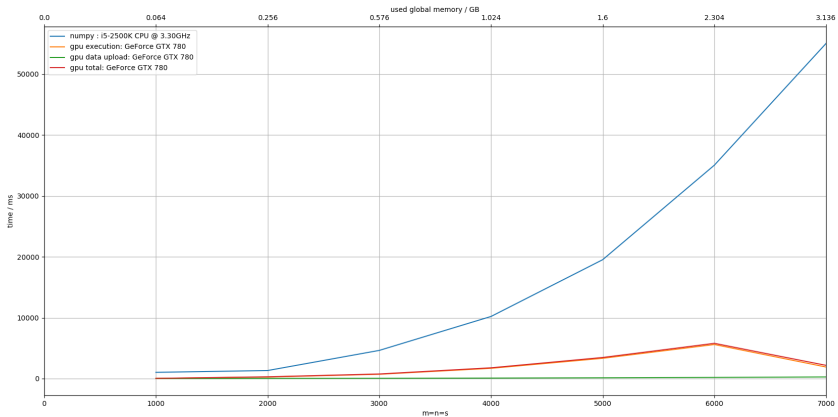
- $m = n = s = 1000$: 0.064 GB
- $m = n = s = 5000$: 1.6 GB
- $m = n = s = 10.000$: 6.40 GB

Komplexität ($m = n = s$):

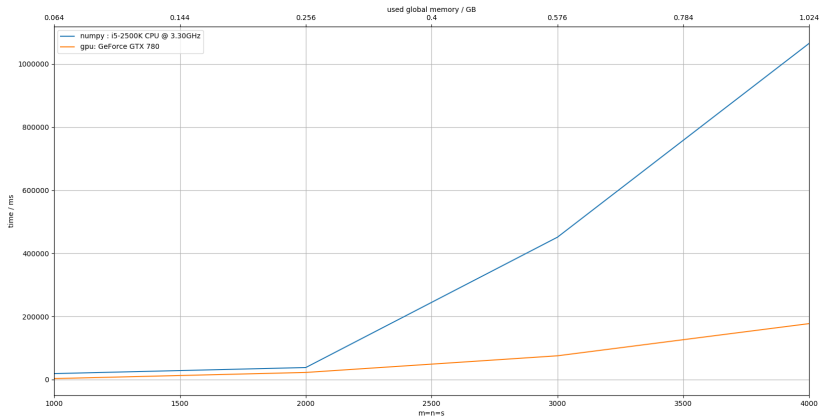
Funktion	Komplexität Seriell
<i>initTss()</i>	s^2
<i>initIdentity()</i>	s^2
<i>getTriangularInverse()</i>	s^2 (backsubstitution)
<i>multWithDz()</i>	s^2
<i>cublasDgemm()</i>	s^3
<i>cublasDgemv()</i>	s^2
<i>cudaMemcpy()</i>	s

Lässt sich alles gut parallelisieren.

Gradient Single Execution - Serial vs Parallel



Gradient 100 Executions - Serial vs Parallel



Blocksize und Gridsize?

-
1. Einführung
 2. Aufgaben
 3. Evaluate
 4. Gradient
 5. Blocksize und Gridsize
 6. Solve
 7. Final Thoughts

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 blocksize and gridsize in abh. von device spec.

Wie sollen gridsize and blocksize gewählt werden?

- Generischen Ansatz
- Starte mit 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 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:

$$A = A \times \text{Diag}(\text{Sign}(dz))$$

Zu Implementierende Operation:

$$A = A \times \text{Diag}(\text{Sign}(dz))$$

Beispiel:

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

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

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

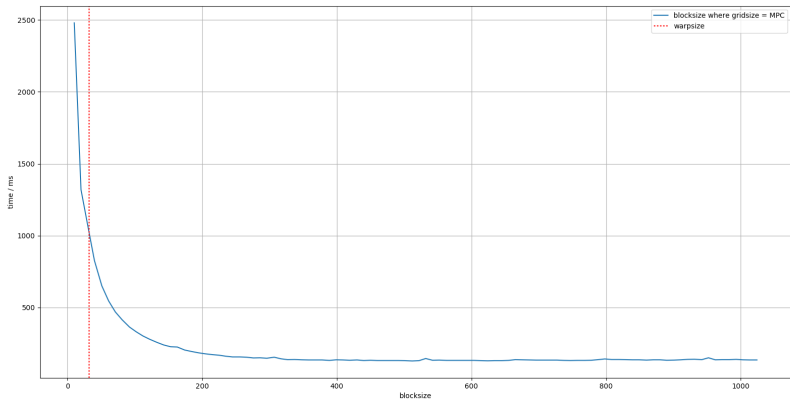
Choosing Gridsize and Blocksize

Beispiel

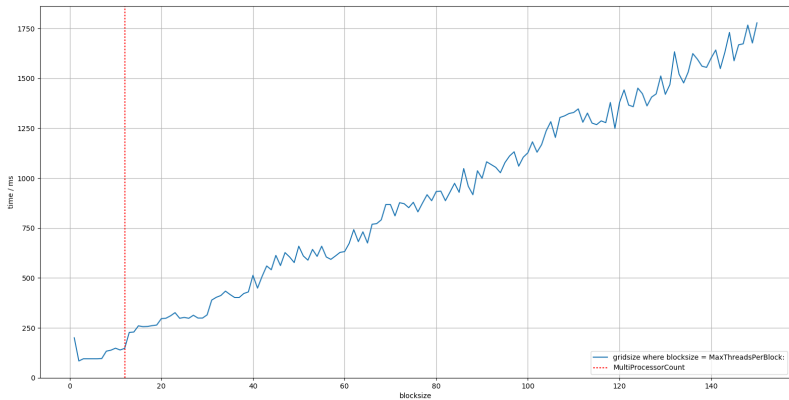



```
1  template <typename T>
2  void __global__ multWithDz(T *A, T *dz, int s){
3      int i = threadIdx.x;
4      int j = blockIdx.x;
5      int id = i*s + j;
6      while(id < s*s && j < s){
7          if(i<s){
8              if(A[id] != T(0))
9                  A[id] = A[id] * cuutils::sign(&dz[j]);
10             i+=blockDim.x;
11         }
12         else{
13             i = i%s;
14             j = j + gridDim.x;
15         }
16         id = i*s + j;
17     }
18 }
```

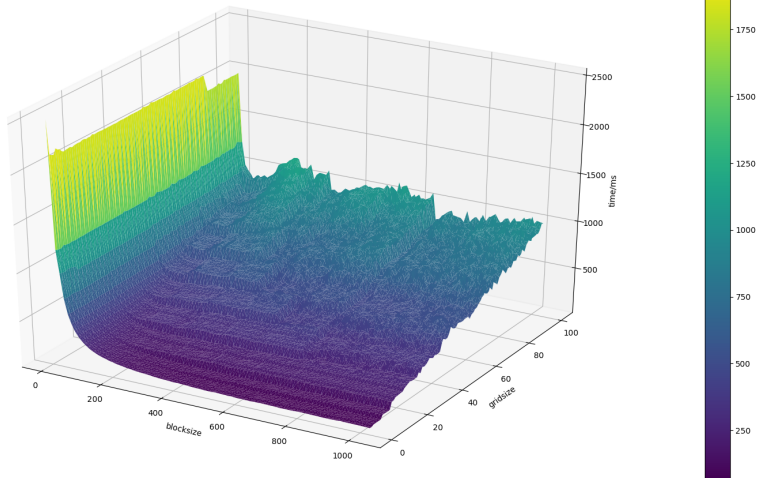
choosing the blocksize



choosing the gridsize



choosing blocksize and gridsize



Solve

-
1. Einführung
 2. Aufgaben
 3. Evaluate
 4. Gradient
 5. Blocksize und Gridsize
 6. **Solve**
 7. Final Thoughts

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 :

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

$$0 = b + J\Delta x + Y|\Delta z|$$

$$-b - Y|\Delta z| = J\Delta x$$

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

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

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

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

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

Wollen Lösen:

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

Haben:

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

Modularisieren:

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

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

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 Konvergenzkriterien (linear / endlich).

Modulus Iteration Algorithmus:

```
1   $\Delta z = \text{Init}()$   
2   $c = a - ZJ^{-1}b$   
3   $S = L - ZJ^{-1}Y$   
4  while not converged:  
5       $\Delta z = c + S|\Delta z|$   
6   $\Delta x = -J^{-1}(b + Y|\Delta z|)$ 
```

Notes:

Modulus Iteration Algorithmus:

```
1   $\Delta z = \text{Init}()$   
2   $c = a - ZJ^{-1}b$   
3   $S = L - ZJ^{-1}Y$   
4  while not converged:  
5       $\Delta z = c + S|\Delta z|$   
6   $\Delta x = -J^{-1}(b + Y|\Delta z|)$ 
```

Notes:

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

Für

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

QR - Zerlegung:

Für

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

QR - Zerlegung:

$$J^{-1}Y = X$$

$$Y = JX$$

$$Y = QRX$$

$$QY = RX$$

Berechne:

$$S = L - ZX$$

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

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

Final Thoughts

-
1. Einführung
 2. Aufgaben
 3. Evaluate
 4. Gradient
 5. Blocksize und Gridsize
 6. Solve
 7. **Final Thoughts**

Was ist da nach der ersten Implementierung?

Was ist da nach der ersten Implementierung?

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

Was ist da nach der ersten Implementierung?

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/>*

SPEICHER-ANIMATION