Inhalt

...

Definition

Smooth funcion
A smooth function $f(x)$ is continuous and has a continuous derivative

$f(x) = x^2 \; f'(x) = 2x$

Definition

Picewise smooth function
A picewise smooth function $f(x)$ is made up of finitly many smooth functions, separated by jump discontiuities.

$$f(x) = \begin{cases} 1 & x < -1 \\ x^2 + 1 & -1 <= x < 2 \\ x & 2 <= x \end{cases}$$

Wir betrachten ausschließlich continuous picewise smooth functions.

Picewise smooth funcitons können durch picewise linear functions (PL) approximiert werden.

BILD

Betrachten ausschließlich continuous PL

ABS-Normal Form:

Repräsentierung für Picewise linear functions (PL)

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

Vorgehen:

- Any picewise linear scalar function has a so called min-max repräsentation
- All min max expressions can be expressed in terms of abs functions
- Picewise linearization wird erreicht durch algorithmic differentiation

$$F(x_1, x_2) = (x_2^2 - x_1^+)^+$$
$$(i)^+ = \max(0, i)$$

BILD

ADOL C, Beispiel

- Evaluate abs normal form
- Calculate Gradient
- Solve abs-normal form system

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

- 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 paralleln ABSNF Aufgaben

Annahmen:

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

Programmiersprachen

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

Annahmen:

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

- utils.hpp test_utils.hpp
- absnf.h test_absnf.cu
- cuutils.h test_cuutils.cu
- tabsnf.h test_tabsnf.h
- make
- absnf.py

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
```

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 $z = f(|\Delta z|)$

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

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

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

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

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

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

Aufgabe 1) Evaluate

Implementierung

```
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	$\lvert\Delta z\rvert$
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 $\approx 81.610 GB$

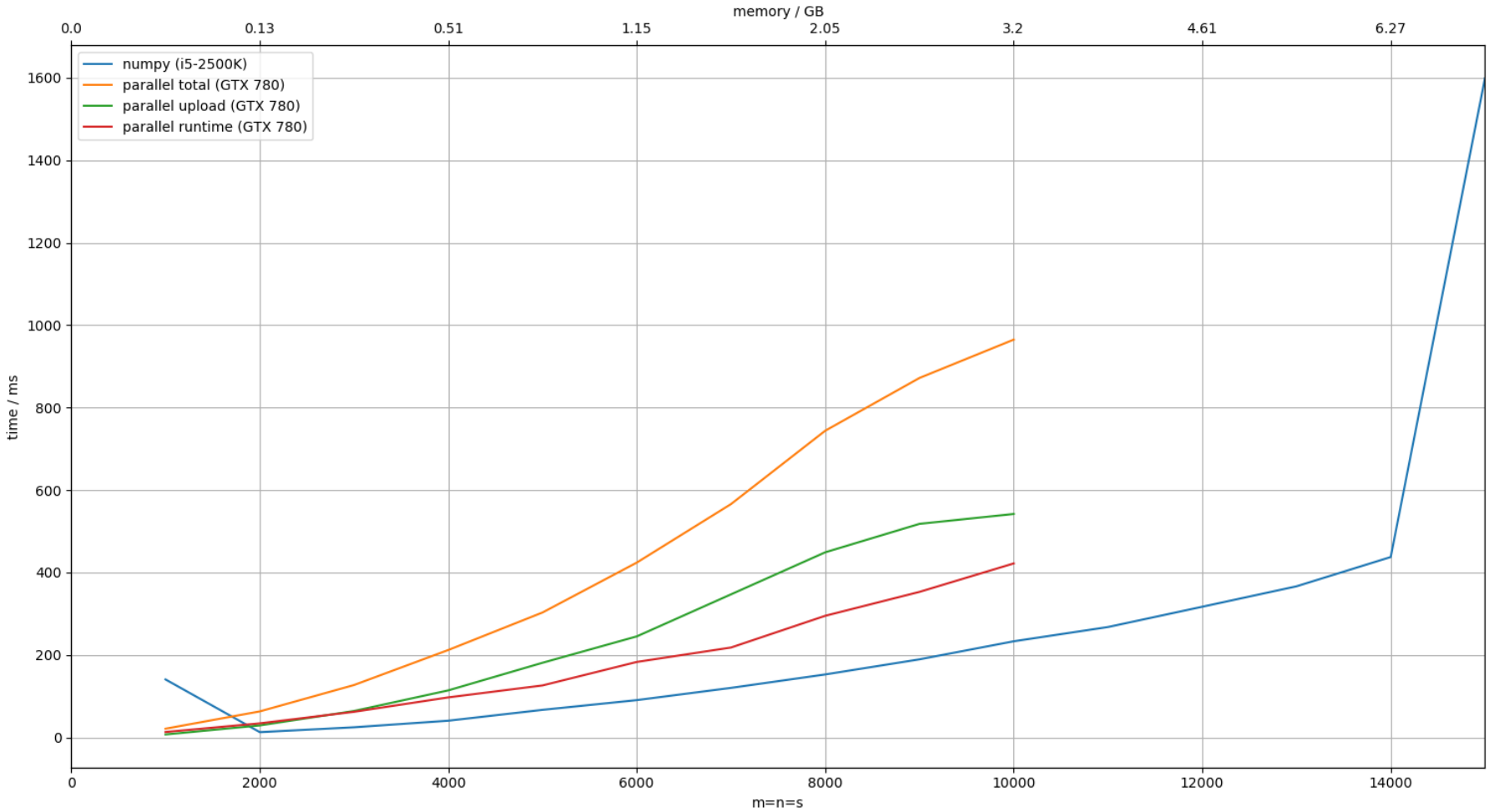
Komplexität:

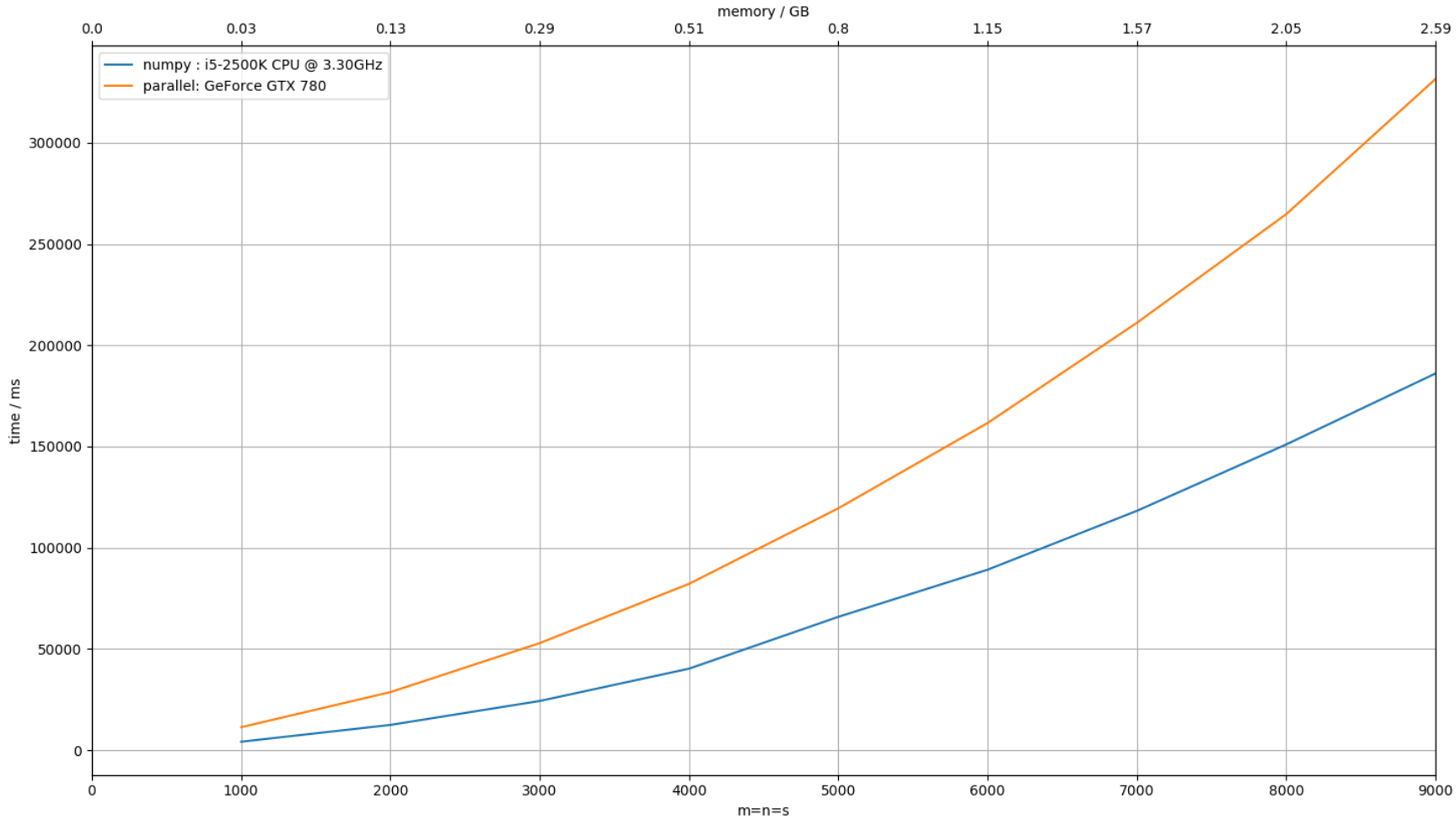
Funktion	Komplexität Seriell	Komplexität Parallel
$cudaMemcpy(dz, a)$	s	s/p
$cublasDgemv(Z, dx, dz)$	$s * n$	$(s * n)/p$
$cublasDgemv(L, \|dz\|)$	$s * s$	$(s * s)/p$
$cublasMemcpy(dy, b)$	m	m/p
$cublasDgemv(J, dx, dy)$	$m * n$	$(m * n)/p$
$cublasDgemv(Y, \|dz\|, dy)$	$m * s$	$(m * s)/p$

Der Rechenaufwand steigt im selben Maße wie der Speicheraufwand !

Vermutung, parallelisieren bringt hier nicht viel!

Flaschenhals: Speicher, Speicheroperationen


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
0
200
400
600
800
1000
1200
1400
1600
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