

# Aufgabe 2) Gradient

## Aufgabenbeschreibung

Gegeben:

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

Gesucht:

$$\gamma, \Gamma$$

Wobei:

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

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

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

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

Brauchen:

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

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

Fallstricken:

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

Sei:

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

Dann gilt für  $I - L\Sigma$ :

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

das entspricht folgenden Operationen:

- Hinzufügen einer Hauptdiagonalen
- Skalieren der Spalten von  $L$  mit den Vorzeichen von  $\Delta z$

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

# Aufgabe 1) Gradient

## Implementierung

```
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 }
```

# Aufgabe 2) Gradient

## Speicherkomplexität

Speicherkomplexität:

$a$	$b$	$Z$	$L$	$J$	$Y$	$\Delta z$	$\gamma$	$\Gamma$	$T_{ss}$	$I$	$K$
$s$	$m$	$s * n$	$s * s$	$m * n$	$m * s$	$s$	$m$	$m * n$	$s * s$	$s * s$	$m * s$

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

# Aufgabe 2) Gradient

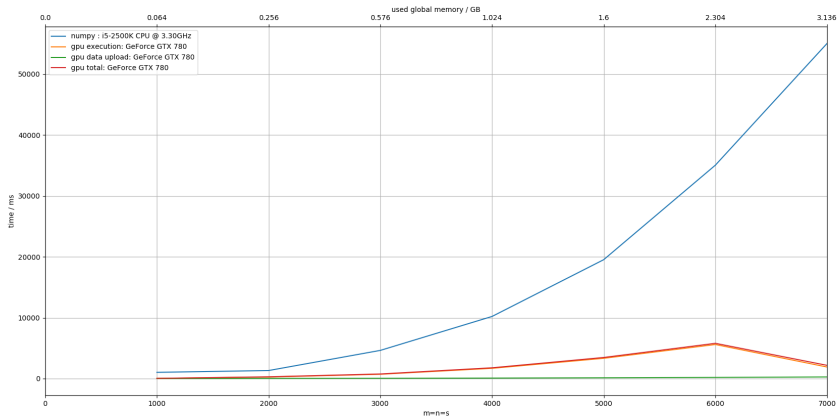
## Komplexität

Komplexität ( $m = n = s$ ):

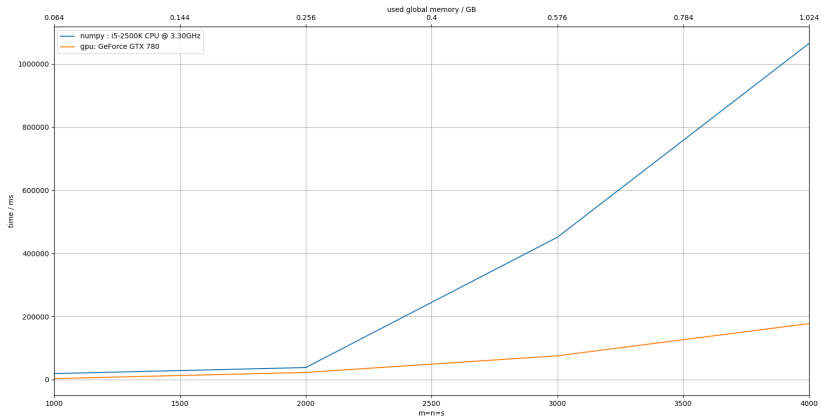
Funktion	Komplexität Seriell
<i>initTss()</i>	$s^2$
<i>initIdentity()</i>	$s^2$
<i>getTriangularInverse()</i>	$s^2$ (backsubstitution)
<i>multWithDz()</i>	$s^2$
<i>cublasDgemm()</i>	$s^3$
<i>cublasDgemv()</i>	$s^2$
<i>cudaMemcpy()</i>	$s$

Lässt sich alles gut parallel ausführen !!!

# Gradient Single Execution - Serial vs Parallel



Gradient 100 Executions - Serial vs Parallel





$$\begin{pmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \\ a & b & c & d \end{pmatrix}$$

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

$$\begin{pmatrix} \Delta z_1 \\ \vdots \\ \Delta z_s \\ \Delta y_1 \\ \vdots \\ \Delta y_m \end{pmatrix} = \begin{pmatrix} a_1 \\ \vdots \\ a_s \\ b_1 \\ \vdots \\ b_m \end{pmatrix} + \begin{pmatrix} Z_{1,1} & \dots & Z_{1,n} & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & L_x & \ddots & 0 \\ Z_{s,1} & \dots & Z_{s,n} & L_x & L_x & 0 \\ J_{1,1} & \dots & J_{1,n} & Y_{1,1} & \dots & Y_{1,s} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ J_{m,1} & \dots & J_{m,n} & Y_{m,1} & \dots & Y_{m,s} \end{pmatrix} \times \begin{pmatrix} \Delta x_1 \\ \vdots \\ \Delta x_n \\ |\Delta z_1| \\ \vdots \\ |\Delta z_s| \end{pmatrix}$$

BLOCKSIZE, GRIDSIZE OPTIMIZATION  
ROW FORMAT, COL FORMAT  
INTERFACE ???

# Choosing Gridsize and Blocksize

## Choosing Gridsize and Blocksize

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:

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

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

## Choosing Gridsize and Blocksize

- Blocksize: 2
- Gridsize: 2

# Choosing Gridsize and Blocksize

## Beispiel - Implementierung

```
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 }
```