

Gliederung

- 1 Einleitung: Rechenleistung von Grafikkarten
- 2 Strömungsmechanische Grundlagen
- 3 Numerische Methoden
- 4 Parallele Programmierung mit CUDA
 - Schematischer Aufbau
 - Das CUDA-Programmiermodell
 - Optimierungen am Beispiel
 - Zeitmessungen
- 5 Numerische Tests
- 6 Fazit

Wirbelstärke ω

Definition

Die **Wirbelstärke** $\vec{\omega}$ einer Strömung ist definiert als Rotation der Geschwindigkeit \vec{c} :

$$\vec{\omega} := \text{rot}(\vec{c})$$

Unter der Annahme einer **ebenen Strömung** ergibt sich die Wirbelstärke zu einer skalarwertigen Größe:

$$\text{rot} \vec{c} = \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} \end{pmatrix} \times \begin{pmatrix} u(x_1, x_2, \phi) \\ v(x_1, x_2, \phi) \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \frac{\partial v}{\partial x_1} - \frac{\partial u}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \omega(x_1, x_2, t) \end{pmatrix}$$

$$\Rightarrow \omega = \frac{\partial v}{\partial x_1} - \frac{\partial u}{\partial x_2}$$

Stromfunktion Ψ

Betrachtet man ebene Strömungen für inkompressible Fluide, so vereinfacht sich die **Kontinuitätsgleichung**:

$$\frac{\partial \rho}{\partial t} + \operatorname{div}(\rho \vec{c}) = 0 \quad \Rightarrow \quad \frac{\partial u}{\partial x_1} + \frac{\partial v}{\partial x_2} = 0$$

Definition

Auf Grund der folgenden Definition für die **Stromfunktion** $\Psi(x_1, x_2)$ wird die Kontinuitätsgleichung implizit erfüllt:

$$u := \frac{\partial \Psi}{\partial x_2} \qquad v := -\frac{\partial \Psi}{\partial x_1}$$

Setze die Definition der Stromfunktion (*) in die Gleichung für die Wirbelstärke ein:

$$\omega = \frac{\partial v}{\partial x_1} - \frac{\partial u}{\partial x_2} \stackrel{(*)}{=} -\frac{\partial}{\partial x_1} \left(\frac{\partial \Psi}{\partial x_1} \right) - \frac{\partial}{\partial x_2} \left(\frac{\partial \Psi}{\partial x_2} \right) = -\Delta \Psi \quad (1)$$

Diese Gleichung $-\Delta \Psi = \omega$ wird als **Poissongleichung** bezeichnet und stellt den ersten Teil des Differentialgleichungssystems dar.

Herleitung der Wirbeltransportgleichung

Aus $\frac{\partial}{\partial x_1}(II) - \frac{\partial}{\partial x_2}(I)$ folgt:

$$\begin{aligned} & \frac{\partial}{\partial t} \underbrace{\left(\frac{\partial v}{\partial x_1} - \frac{\partial u}{\partial x_2} \right)}_{=\omega} + \underbrace{u \cdot \frac{\partial^2 v}{\partial x_1^2} - u \cdot \frac{\partial^2 u}{\partial x_1 \partial x_2}}_{=u \cdot \frac{\partial \omega}{\partial x_1}} + \underbrace{v \cdot \frac{\partial^2 v}{\partial x_1 \partial x_2} - v \cdot \frac{\partial^2 u}{\partial x_2^2}}_{=v \cdot \frac{\partial \omega}{\partial x_2}} \\ & + \underbrace{\frac{\partial u}{\partial x_1} \cdot \frac{\partial v}{\partial x_1} + \frac{\partial v}{\partial x_1} \cdot \frac{\partial v}{\partial x_2} - \frac{\partial u}{\partial x_2} \cdot \frac{\partial u}{\partial x_1} - \frac{\partial v}{\partial x_2} \cdot \frac{\partial u}{\partial x_2}}_{=0 \text{ (folgt aus der Definition von } \Psi \text{)}} \\ & = \underbrace{-\frac{\partial^2}{\partial x_1 \partial x_2} \left(U + \frac{p}{\rho} \right) + \frac{\partial^2}{\partial x_2 \partial x_1} \left(U + \frac{p}{\rho} \right)}_{=0 \text{ (Satz von Schwarz)}} + \underbrace{\nu \left(\frac{\partial}{\partial x_1} \Delta v - \frac{\partial}{\partial x_2} \Delta u \right)}_{=\nu \Delta \omega \text{ (Linearität)}} \end{aligned}$$

Man erhält somit die **Wirbeltransportgleichung**:

$$\frac{\partial \omega}{\partial t} + \vec{c} \cdot \text{grad } \omega = \nu \Delta \omega \quad (2a)$$

Die **Wärmeleitungsgleichung für bewegte Fluide** hat die selbe mathematische Struktur wie die Wirbeltransportgleichung:

$$\frac{\partial T}{\partial t} + \vec{c} \cdot \text{grad } T = \frac{\lambda}{\underbrace{\rho c_W}_{=:a}} \Delta T \quad (3)$$

Hierbei wurde in der Konstanten a

- die Wärmeleitfähigkeit λ ,
- die Dichte ρ des Fluids,
- sowie die Wärmekapazität c_W des Fluids

berücksichtigt.

Anwenden der FVM auf die Poissongleichung:

$$\begin{aligned}
 & - \int_{\Omega} \Delta \Psi d\Omega = \int_{\Omega} \omega d\Omega \\
 \Rightarrow & \sum_{i \in I, j \in J} - \int_{\Omega_{i,j}} \Delta \Psi d\Omega_{i,j} = \sum_{i \in I, j \in J} \int_{\Omega_{i,j}} \omega d\Omega_{i,j}
 \end{aligned}$$

Betrachte einzelnes finites Volumen (=Kontrollvolumen):

$$\begin{aligned}
 & - \int_{\Omega_{i,j}} \Delta \Psi d\Omega_{i,j} = \int_{\Omega_{i,j}} \omega d\Omega_{i,j} \\
 \Leftrightarrow & - \int_{\Omega_{i,j}} \operatorname{div} \operatorname{grad} \Psi d\Omega_{i,j} = \int_{\Omega_{i,j}} \omega d\Omega_{i,j} \\
 \Leftrightarrow & - \int_{\partial\Omega_{i,j}} \operatorname{grad} \Psi \cdot \vec{n} d\partial\Omega_{i,j} = \int_{\Omega_{i,j}} \omega d\Omega_{i,j} \\
 \Leftrightarrow & - \int_{\partial\Omega_{i,j}} \frac{\partial \Psi}{\partial \vec{n}} d\partial\Omega_{i,j} = \int_{\Omega_{i,j}} \omega d\Omega_{i,j},
 \end{aligned}$$

wobei $\partial\Omega_{i,j} = \Gamma_{i,j} = \Gamma_{i,j}^N \cup \Gamma_{i,j}^O \cup \Gamma_{i,j}^S \cup \Gamma_{i,j}^W$ für rechteckige Kontrollvolumen.

Wirbeltransportgleichung und Wärmeleitungsgleichung lassen sich verallgemeinert schreiben als:

$$\frac{\partial \zeta}{\partial t} = k_1 \Delta \zeta - \vec{c} \cdot \text{grad} \zeta + k_2 \frac{\partial T}{\partial x_1},$$

mit $\zeta := \{\omega, T\}$, $k_1 := \begin{cases} \nu & \text{für } \zeta = \omega \\ a & \text{für } \zeta = T \end{cases}$ sowie $k_2 := \begin{cases} C & \text{für } \zeta = \omega \\ 0 & \text{für } \zeta = T \end{cases}$.

Anwenden der FVM ergibt:

$$\sum_{i \in I, j \in J} \int_{\Omega_{i,j}} \frac{\partial \zeta}{\partial t} d\Omega_{i,j} = \sum_{i \in I, j \in J} \left(\underbrace{k_1 \int_{\Omega_{i,j}} \Delta \zeta d\Omega_{i,j}}_{\text{Laplace-Term}} - \overbrace{\int_{\Omega_{i,j}} \vec{c} \cdot \text{grad} \zeta d\Omega_{i,j}}^{\text{Konvektionsterm}} + \underbrace{k_2 \int_{\Omega_{i,j}} \frac{\partial T}{\partial x_1} d\Omega_{i,j}}_{\text{Temperatur-Term}} \right)$$

$$\approx B_h \zeta_h - C_h \zeta_h + D_h T_h + b_h$$

Zusammenfassung

Das System aus partiellen Differentialgleichungen liegt nun in vollständig diskretisierter Form vor:

$$A_h \Psi_h = \underbrace{\omega_h + b_h^\Psi}_{=: \tilde{\omega}_h}$$

$$\underbrace{(E_n + \delta t \cdot \nu \cdot A_h)}_{=: B_h^\omega} \omega_h = \underbrace{(E_n - \frac{\delta t}{h^2} \cdot C_h) \omega_h + \frac{\delta t}{h^2} \cdot D_h T_h + \frac{\delta t}{h^2} b_h^\omega}_{=: rhs^\omega}$$

$$\underbrace{(E_n + \delta t \cdot a \cdot A_h)}_{=: B_h^T} T_h = \underbrace{(E_n - \frac{\delta t}{h^2} \cdot C_h) T_h + \frac{\delta t}{h^2} b_h^T}_{=: rhs^T}$$

Eindeutig lösbar durch Vorgabe von Anfangswerten und Randbedingungen:

- Randbedingungen für Ψ_h, ω_h, T_h auf $\partial\Omega$
- Anfangswerte T_h^0, ω_h^0 im Innern von Ω

In einem physikalischen System sind in der Regel bekannt:

- Randbedingungen für T_h, \vec{c}_h auf $\partial\Omega$
- Anfangswerte für T_h, \vec{c}_h im Innern von Ω

Zum Einbau von Randbedingungen werden diese als explizite Werte benötigt (Dirichlet-Randbedingungen).

Wärmeisolierte Wände werden aber durch Neumann-Randbedingungen modelliert. Diese können aber leicht in Dirichlet-Randbedingungen umgeschrieben werden:

$$\frac{\partial T}{\partial \vec{n}} \approx \frac{T_{i,1} - T_{i,0}}{h} \stackrel{!}{=} 0 \quad \iff \quad T_{i,0} = T_{i,1} \quad \text{auf } \Gamma_3$$

Rand- und Anfangsbedingungen

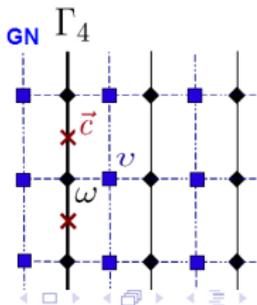
Berechnung von Randwerten für die Stromfunktion Ψ_h durch Integration der Gleichungen:

$$u = \frac{\partial \Psi}{\partial x_2} \quad v = -\frac{\partial \Psi}{\partial x_1} \quad \rightarrow$$

$$\Psi = \int_{\Gamma_4} u \, dy \quad \rightarrow \quad \begin{array}{c} \Psi = -\int_{\Gamma_1} v \, dx \\ + \quad \xrightarrow{\hspace{2cm}} \quad = \\ \uparrow \quad \square \quad \Omega \quad \uparrow \\ \Psi = \int_{\Gamma_2} u \, dy \\ 0 \quad \xrightarrow{\hspace{2cm}} \quad + \\ \Psi = -\int_{\Gamma_3} v \, dx \end{array}$$

Berechnung von Randwerten für die Wirbelstärke ω_h nach:

$$\omega = \frac{\partial v}{\partial x_1} - \frac{\partial u}{\partial x_2} \quad \rightarrow$$



Ein einfaches Beispiel: SAXPY-Operation

```

__global__ void k_saxpy(double *erg, double *x, double *y, double alpha,
    int dim){
    int index=blockIdx.x*blockDim.x+threadIdx.x; //globale Thread-ID
    if(index<dim)
        erg[index]=alpha*x[index]+y[index];
}
int main (int argc, char * const argv []){
    int dim=100000;
    //Anlegen der Vektoren auf dem Host
    double* x=(double*)malloc(sizeof(double)*dim);
    double* y=(double*)malloc(sizeof(double)*dim);
    double* erg=(double*)malloc(sizeof(double)*dim);
    double alpha=5.0;
    //Anlegen der Vektoren auf dem Device
    double *d_x, *d_y, *d_erg;
    cudaMalloc((void**)&d_x, dim*sizeof(double));
    cudaMalloc((void**)&d_y, dim*sizeof(double));
    cudaMalloc((void**)&d_erg, dim*sizeof(double));
    //Vektoren x und y mit Werten belegen
    //Vektoren auf Device kopieren
    cudaMemcpy(d_x, x, dim*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, dim*sizeof(double), cudaMemcpyHostToDevice);
    //Kernelaufruf
    dim3 dimBlock(128);
    dim3 dimGrid(dim/dimBlock.x+1);
    k_saxpy<<<dimGrid, dimBlock>>>(d_erg, d_x, d_y, alpha, dim);
    //Ergebnis auf Host kopieren
    cudaMemcpy(erg, d_erg, dim*sizeof(double), cudaMemcpyDeviceToHost);
    //Speicherfreigabe
    cudaFree(d_x); cudaFree(d_y); cudaFree(d_erg);
    free(x); free(y); free(erg);
    return 0;
}

```


Algorithm 1 CG-Verfahren zur Lösung von $Ax = b$ (Pseudocode)

Input: $A \in \mathbb{R}^{N,N}$, $b, x_0 \in \mathbb{R}^N$, $MAXIT \in \mathbb{N}$, $tol \in \mathbb{R}_+$

Output: Lösung $x \in \mathbb{R}^N$

$$r_0 = b - Ax_0$$

$$p_0 = r_0$$

for $j = 1$ to $MAXIT$ **do**

$$\gamma_{j-1} = (r_{j-1}^T r_{j-1}) / (p_{j-1}^T A p_{j-1})$$

$$x_j = x_{j-1} + \gamma_{j-1} p_{j-1}$$

$$r_j = r_{j-1} - \gamma_{j-1} A p_{j-1}$$

if $\|r_j\|_2 / \|r_0\|_2 < tol$ **then**

return x_j

end if

$$\beta_j = r_j^T r_j / r_{j-1}^T r_{j-1}$$

$$p_j = r_j + \beta_j p_{j-1}$$

end for

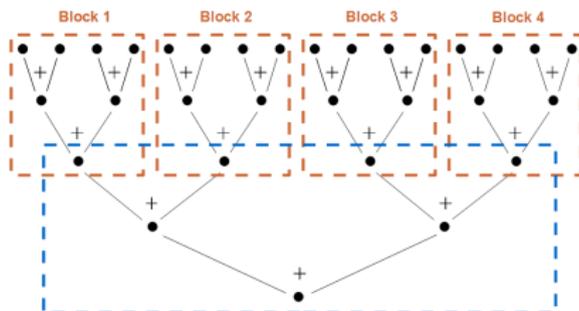
Aufwändige Operationen sind hierbei:

- SAXPY-Operation
- Skalarprodukt bzw. Norm (blau)
- Matrix-Vektor Multiplikation (rot)

Skalarprodukt

Generelle Idee:

- Das Skalarprodukt zwischen zwei Vektoren, kann in Teilsummen unabhängig voneinander berechnet werden.
- Jeder Thread berechnet n Produkte und summiert diese auf.
- Anschließend werden alle Teilsummen innerhalb des Blockes summiert.
- Ein zweiter Kernel summiert wiederum die Ergebnisse der Blöcke.



Erster Implementierung:

```

__global__ void k_skp1(Real *c, Real *a, Real *b, int dim){
    //Anzahl der von jedem Thread zu berechnenden Produkte
    int n=dim/(blockDim.x*gridDim.x+1)+1;
    extern __shared__ Real shared_summands[];
    Real sum=0.0;

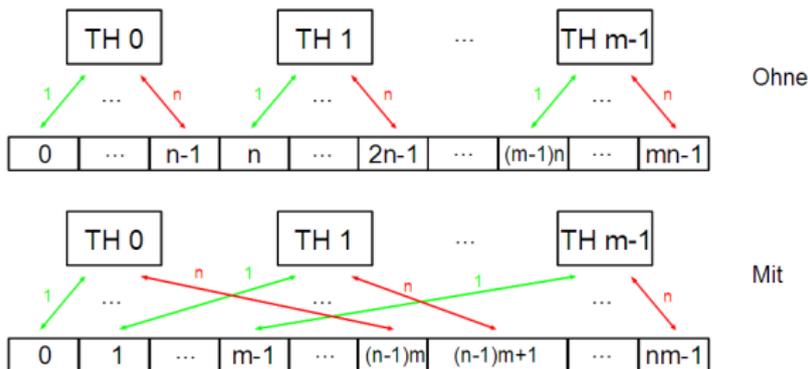
    //Berechnung der Produkte und Aufsummation innerhalb des Thread
    int index=(blockIdx.x*blockDim.x+threadIdx.x)*n; //Startindex
    for(int i=0;i<n;i++){
        if((index+i)<dim)
            sum+=a[index+i]*b[index+i];
    }
    //Abspeicherung der Teilsumme im Shared-Memory
    shared_summands[threadIdx.x]=sum;

    //Berechnung der Teilsumme des gesamten Blockes
    __syncthreads();
    for(int s=1; s<blockDim.x; s*=2){
        if(threadIdx.x%(2*s) == 0){
            shared_summands[threadIdx.x] += shared_summands[threadIdx.x+s];
        }
        __syncthreads();
    }
    //Abspeicherung der Teilsumme des gesamten Blocks im Global-Memory
    if(threadIdx.x==0)
        c[blockIdx.x] = shared_summands[0];
}

```

Verbesserung: Mit Coalescing

- Auf Global Memory wird segmentweise zugegriffen.
- Liegen die Elemente eines Warps innerhalb eines Segments, so kann der Zugriff zusammengefasst werden.



Bei diesem Beispiel gibt es m Threads, die jeweils auf n Elemente zugreifen.

Verbesserung: Ohne Divergency

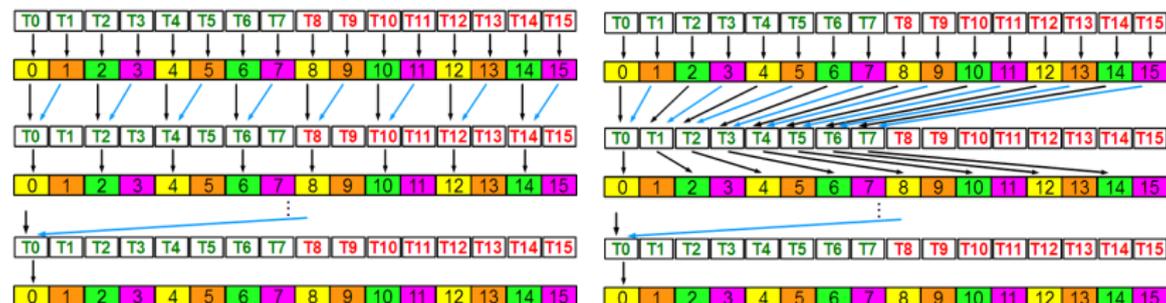


Abbildung: Links mit Divergency, rechts ohne

Vereinfachende Annahme: Ein Warp besteht aus 8 Threads und der Shared-Memory besteht aus 4 Speicherbänken.

Verbesserung: Ohne Bankkonflikte

- Shared-Memory ist in 16 Bänken organisiert
- Elemente liegen in jeweils nach einander folgenden Bänken

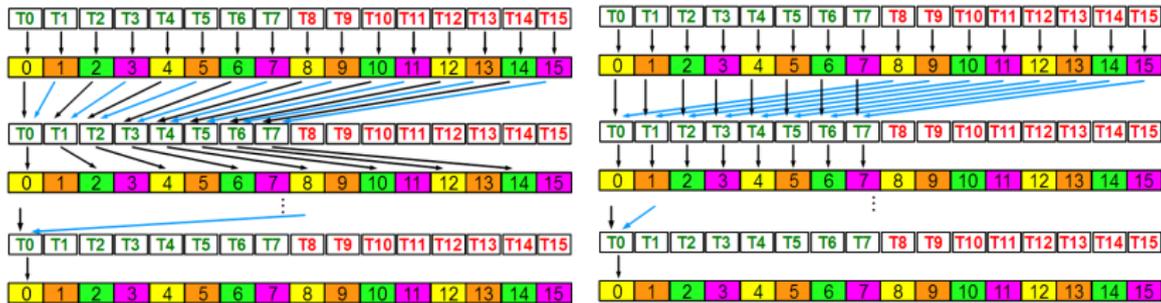


Abbildung: Links mit Bankkonflikten, rechts ohne

Vereinfachende Annahme: Auch hier besteht ein Warp aus 8 Threads und der Shared-Memory aus 4 Speicherbänken.

Weitere Verbesserungen:

- Reduzierung von if-Abfragen
- Abrollen der for-Schleife, welche die Teilsumme innerhalb des Blockes bildet
- Extra-Funktion zur Berechnung der Norm, bei der der Zugriff auf den Global-Memory halbiert wurde

Zeitmessungen: Skalarprodukt

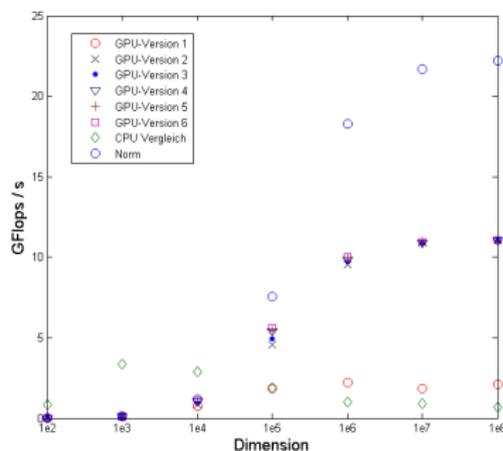
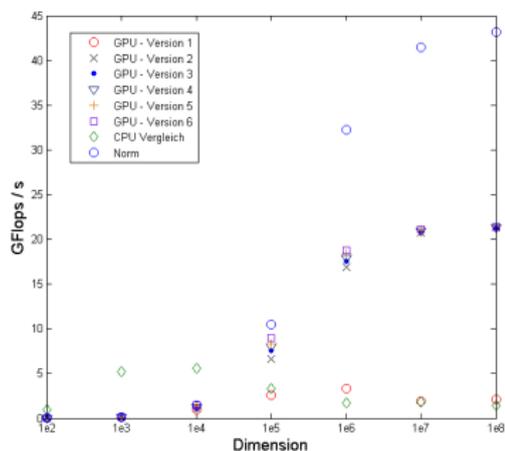


Abbildung: Vergleich der Performance der verschiedenen Skalarprodukt Implementierungen für einfache (links) und doppelte (rechts) Genauigkeit.

$$P_{\text{single}} = \frac{102 \text{ GByte/s} \cdot 2 \text{ Flops}}{2 \cdot 4 \text{ Byte}} = 25.5 \text{ GFlops/s}$$

$$P_{\text{double}} = \frac{102 \text{ GByte/s} \cdot 2 \text{ Flops}}{2 \cdot 8 \text{ Byte}} = 12.75 \text{ GFlops/s}$$

Matrix-Vektor Multiplikation

Zum Speichern von Bandmatrizen, eignet sich das DIA-Format:

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \rightarrow \text{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix}, \text{ offset} = [-2 \quad 0 \quad 1]$$

Generelle Idee:

- Diagonalen werden in einer Matrix data gespeichert.
- Der offset-Vektor enthält den Abstand der Diagonalen von der Hauptdiagonalen.

Erste Implementierung:

```

__global__ void k_spmv_dia1 (Real* y,           //Ergebnisvektor
                           Real* data,       //data-Matrix
                           int *offsets,     //Offsetvektor
                           Real *x,         //Vektor
                           int dim_r,       //#Zeilen der Sparsematrix
                           int dim_c,       //#Spalten der Sparsematrix
                           int dim_offsets, //#Diagonalen
                           int stride){
//Berechnung der Zeile
int row = blockDim.x * blockIdx.x + threadIdx.x;
if(row < dim_r){
    Real dot = 0.0;
    for ( int n = 0; n < dim_offsets ; n++){
        //Berechnung der Spalte
        int col = row + offsets [n];
        if( col >= 0 && col < dim_c)
            dot += data[stride * n + row] * x[col];
    }
    y[row] = dot;
}
}

```

- Jeder Thread berechnet ein Element des Ergebnisvektors (Zeile der Matrix mal Vektor)

Verbesserung: Nutzung von Shared-Memory

```

__global__ void k_spmv_dia2(Real* y, Real* data, int *offsets, Real *x,
                           int dim_r, int dim_c, int dim_offsets, int
                           stride){
    int row = blockDim.x * blockIdx.x + threadIdx.x;

    extern __shared__ int shared_offsets[];
    if (threadIdx.x < dim_offsets)
        shared_offsets[threadIdx.x] = offsets[threadIdx.x];
    __syncthreads();

    if (row < dim_r){
        Real dot = 0.0;
        for (int n = 0; n < dim_offsets; n++){
            int col = row + shared_offsets[n];
            Real val = data[dim_r * n + row];
            if (col >= 0 && col < dim_c)
                dot += val * x[col];
        }
        y[row] = dot;
    }
}

```

- Reduzierung des Zugriffs auf den Global-Memory durch Speichern des offset-Vektor im Shared-Memory

Verbesserung: Nutzung von Texturen

```

__global__ void k_spmv_dia3(Real* y,Real* data,int *offsets,Real *x,
                          int dim_r,int dim_c ,int dim_offsets,int
                          stride){
    int row = blockDim.x * blockIdx.x + threadIdx.x;
    Real d;
    extern __shared__ int shared_offsets[];
    if (threadIdx.x < dim_offsets)
        shared_offsets[threadIdx.x] = offsets[threadIdx.x];
    syncthreads();
    if (row < dim_r){
        Real dot = 0.0;
        for ( int n = 0; n < dim_offsets ; n ++){
            int col = row + shared_offsets[n];
            Real val = data[dim_r * n + row];
            if ( col >= 0 && col < dim_c){

                #if PRAEZISSION == 2
                    int2 temp = tex1Dfetch(Tex, col);
                    d = __hiloint2double(temp.y,temp.x);
                #else
                    d = tex1Dfetch(Tex, col);
                #endif
                dot += val * d;

            }
        }
        y[row] = dot;
    }
}

```

Zeitmessungen: Matrix-Vektor Multiplikation

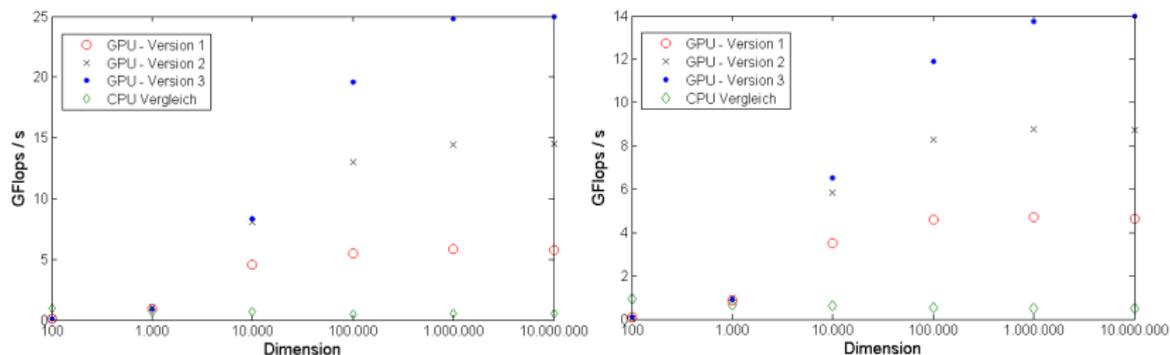


Abbildung: Vergleich der Performance der verschiedenen Matrix-Vektor Implementierungen für einfache (links) und doppelte (rechts) Genauigkeit.

$$P_{\text{single}} = \frac{102\text{GByte/s} \cdot 5 \cdot \text{dim} \cdot 2\text{Flops}}{(5 + 5 + 1) \cdot \text{dim} \cdot 4\text{Byte}} = 23.2\text{GFlops/s}$$

$$P_{\text{double}} = \frac{102\text{GByte/s} \cdot 5 \cdot \text{dim} \cdot 2\text{Flops}}{(5 + 5 + 1) \cdot \text{dim} \cdot 8\text{Byte}} = 11.6\text{GFlops/s}$$

SpeedUp: CG-Verfahren

Es wurden drei Versionen des CG-Verfahrens implementiert:

- Die erste Version `CG_CPU_DIA` ist eine sequentielle Version auf der CPU.
- Die zweite Version `CG_GPU_DIA1` ist eine parallele Version die auf der GPU läuft, die das LGS vom Arbeitsspeicher lädt und das Ergebnis wieder im Arbeitsspeicher abspeichert
- Die dritte Version `CG_GPU_DIA2` lädt das LGS im Vergleich zur zweiten Version direkt aus dem Global-Memory und speichert dort auch das Ergebnis.

Zeitmessungen

Gitter	LGS	Iter.	CG GPU DIA1	CG GPU DIA2	CG CPU DIA
64 × 64	$A_{h,h}$	225	22.52ms 0.77GF	22.06ms 0.79GF	156ms 0.11GF
64 × 64	$B_{h,h}^T$	15	1.63ms 0.74GF	1.50ms 0.80GF	10.9ms 0.11GF
64 × 64	B_h	1	0.29ms 0.45GF	0.16ms 0.81GF	0.80ms 0.16GF
256 × 256	$A_{h,h}$	932	151.5ms 7.94GF	149.9ms 8.03GF	12.57s 0.10GF
256 × 256	$B_{h,h}^T$	54	10.34ms 6.82GF	8.78ms 8.04GF	631ms 0.11GF
256 × 256	B_h	1	1.70ms 1.30GF	0.26ms 8.29GF	13.29ms 0.17GF
1024 × 1024	$A_{h,h}$	3786	4.68s 16.9GF	4.65s 17.0GF	11.4min 0.12GF
1024 × 1024	$B_{h,h}^T$	215	0.28s 16.3GF	0.27s 17.0GF	40.8s 0.11GF
1024 × 1024	B_h	2	13.6ms 4.16GF	3.24ms 17.4GF	0.4s 0.13GF

Gitter	LGS	Iter.	CG GPU DIA1	CG GPU DIA2	CG CPU DIA
64 × 64	$A_{h,h}$	202	22.75ms 0.69GF	22.96ms 0.68GF	137ms 0.11GF
64 × 64	$B_{h,h}^T$	15	1.97ms 0.61GF	1.68ms 0.72GF	10.8ms 0.11GF
64 × 64	B_h	1	0.36ms 0.36GF	0.18ms 0.74GF	1.31ms 0.10GF
256 × 256	$A_{h,h}$	801	182.4ms 5.67GF	180.5ms 5.73GF	901ms 0.11GF
256 × 256	$B_{h,h}^T$	54	14.46ms 4.88GF	12.21ms 5.78GF	624ms 0.11GF
256 × 256	B_h	1	2.48ms 0.88GF	0.36ms 6.03GF	22.35ms 0.10GF
1024 × 1024	$A_{h,h}$	3564	7.45s 9.99GF	7.39s 10.08GF	10.1min 0.12GF
1024 × 1024	$B_{h,h}^T$	215	0.47s 9.67GF	0.45s 10.04GF	40.66s 0.11GF
1024 × 1024	B_h	2	21.8ms 2.59GF	5.36ms 10.52GF	0.57s 0.11GF

Tabelle: Zeitmessungen CG-Verfahren in einfacher (oben) und doppelter (unten) Genauigkeit

SpeedUp: Zeitschritt des Solvers

Gitter	Version	Gesamt	LGS lösen	Abspeichern	Rest
64 × 64	GPU_1	53.35ms	26.06ms	26.70ms	0.58ms
64 × 64	GPU_2	52.89ms	25.41ms	27.19ms	0.30ms
64 × 64	CPU	204.6ms	176.7ms	27.40ms	0.50ms
256 × 256	GPU_1	0.53s	0.17s	0.34s	6.72ms
256 × 256	GPU_2	0.52s	0.17s	0.35s	0.85ms
256 × 256	CPU	13.02s	12.67s	0.34s	6.83ms
1024 × 1024	GPU_1	12.00s	5.27s	6.62s	0.11s
1024 × 1024	GPU_2	11.74s	5.20s	6.53s	10.35ms
1024 × 1024	CPU	9.13min	9.03min	5.50s	0.13s

Gitter	Version	Gesamt	LGS lösen	Abspeichern	Rest
64 × 64	GPU_1	52.84ms	24.98ms	27.36ms	0.5ms
64 × 64	GPU_2	51.68ms	23.97ms	27.41ms	0.31ms
64 × 64	CPU	178.4ms	150.1ms	27.76ms	0.5ms
256 × 256	GPU_1	0.58s	0.21s	0.37s	7.1ms
256 × 256	GPU_2	0.56s	0.20s	0.35s	0.93ms
256 × 256	CPU	9.90s	9.56s	0.34s	7.06ms
1024 × 1024	GPU_1	16.02s	7.63s	8.27s	0.13s
1024 × 1024	GPU_2	15.67s	7.41s	8.25s	12.3ms
1024 × 1024	CPU	9.15min	9.07min	5.45s	0.13s

Tabelle: Zeitmessungen eines Zeitschritts bei einfacher (oben) und doppelter (unten) Genauigkeit

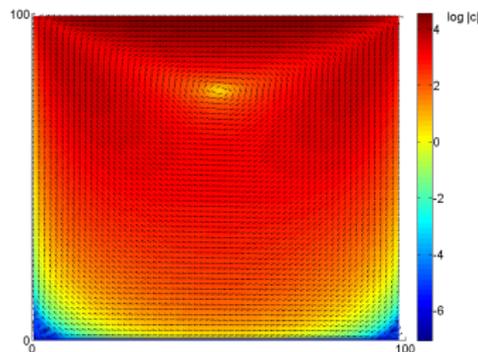
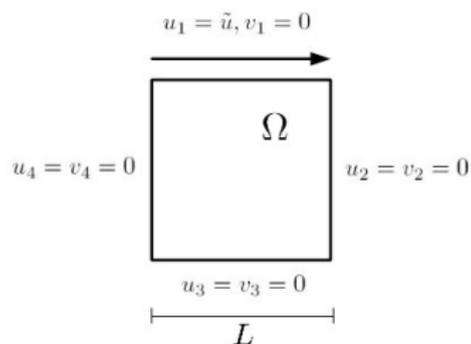
Versuchsaufbau:

Abbildung: Driven Cavity Versuch mit $Re = 1$

Der Versuch wird für verschiedene Reynoldszahlen $Re := \frac{\tilde{u} \cdot L}{\nu}$ durchgeführt.

Driven Cavity

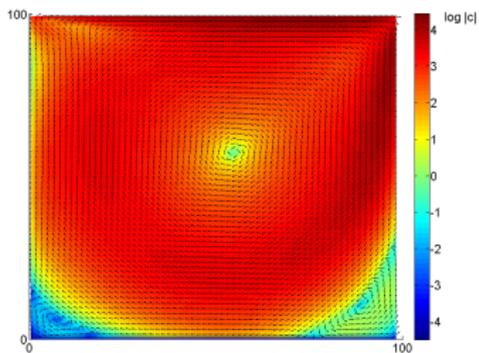


Abbildung: Driven Cavity Versuch mit $Re = 100$

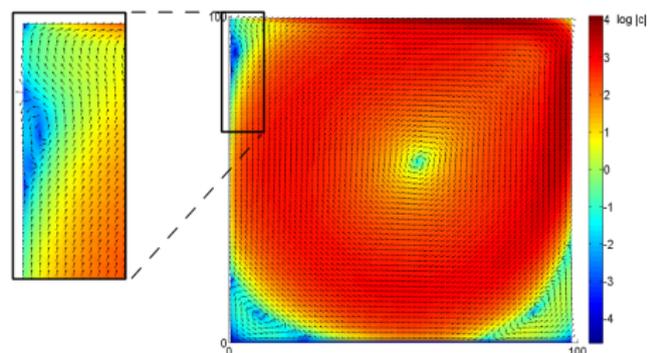
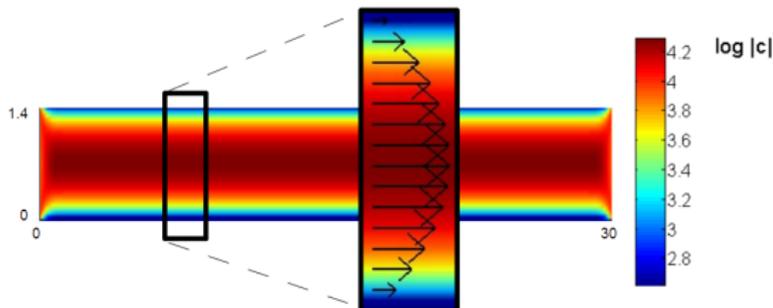
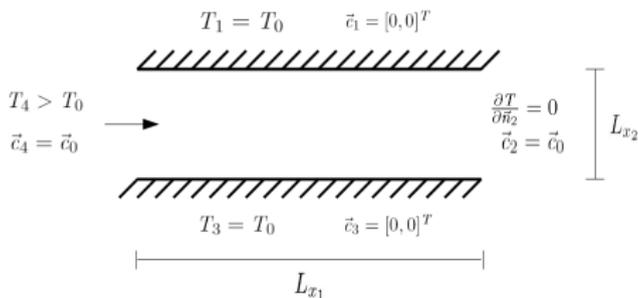


Abbildung: Driven Cavity Versuch mit $Re = 3.333$

Versuchsaufbau:

$$u(x_2) = 6u_0 \frac{x_2}{h} \left[1 - \frac{x_2}{h} \right]$$

Abbildung: Geschwindigkeitsprofil bei der Kanalströmung

Ebene Poiseuille-Strömung

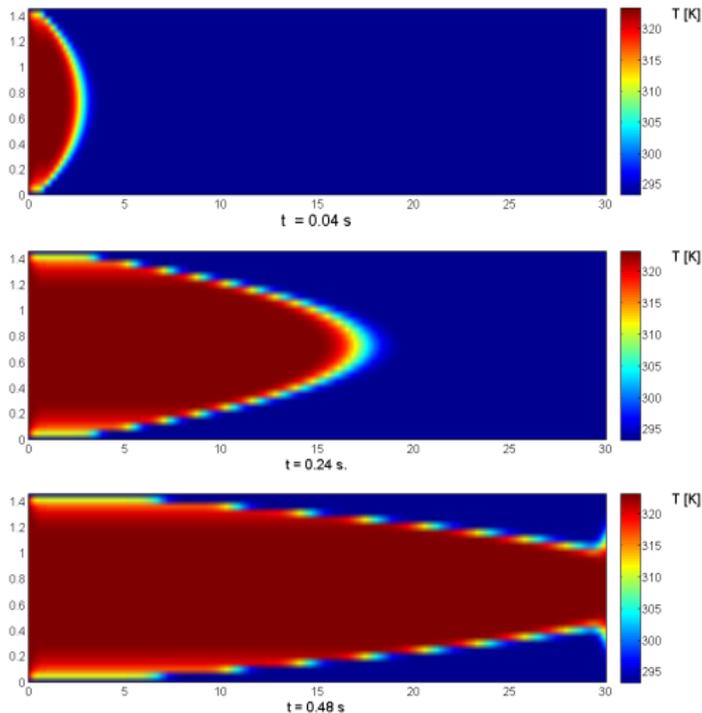


Abbildung: Temperaturverteilung bei einer Kanalströmung zu verschiedenen Zeitpunkten

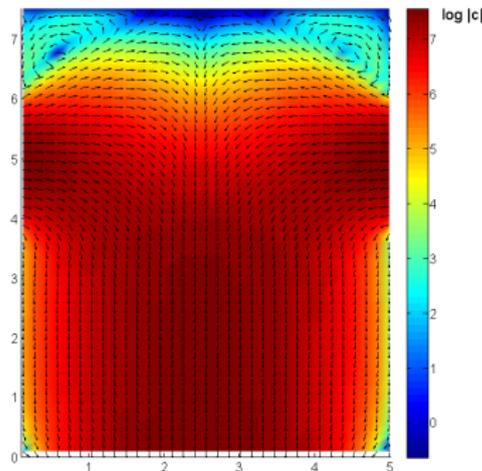
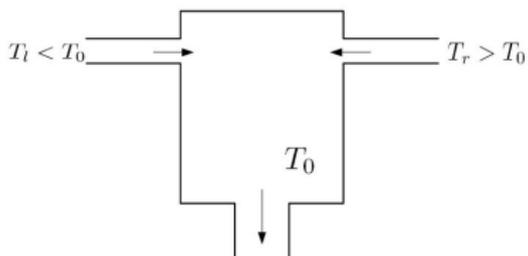
Versuchsaufbau:

Abbildung: Geschwindigkeitsverteilung bei der Mischung zweier Strömungen

Mischung zweier Fluide

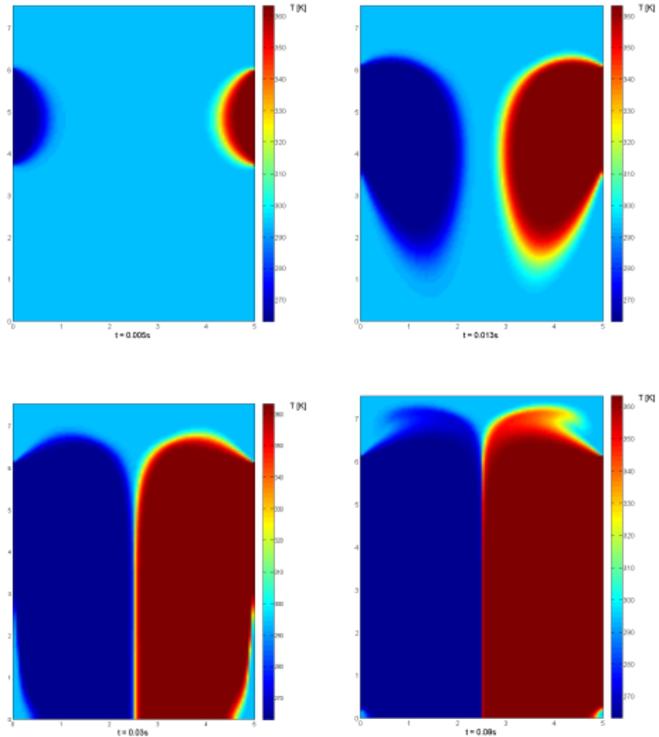


Abbildung: Temperaturverteilung zu verschiedenen Zeitpunkten

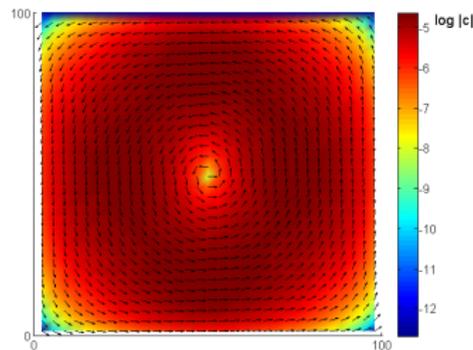
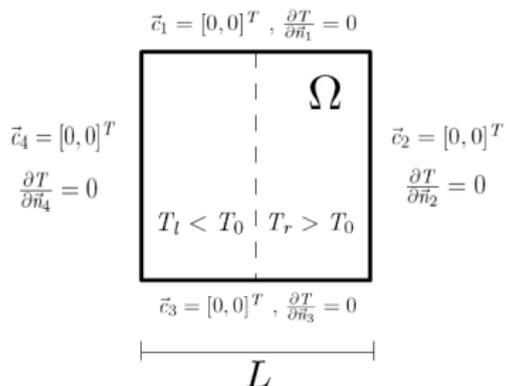
Versuchsaufbau:

Abbildung: Geschwindigkeitsfeld durch Diffusion

- Bei diesem Versuch soll die Wärmeleitung die Wärmekonvektion überwiegen.



Diffusion

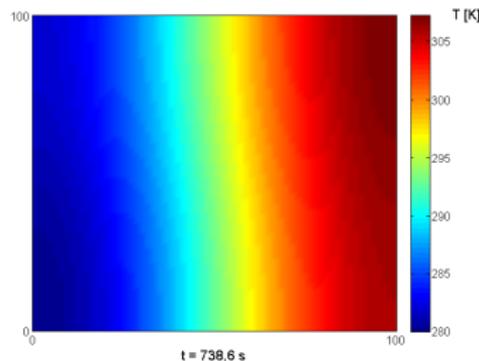
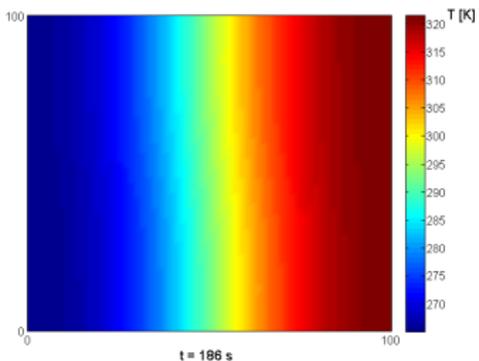
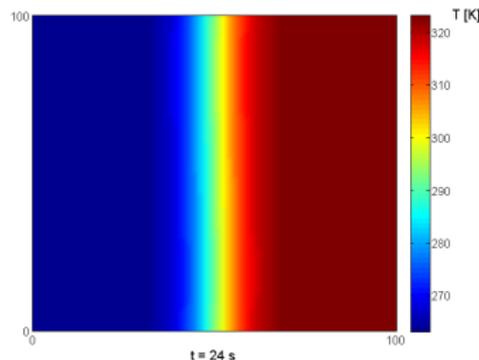
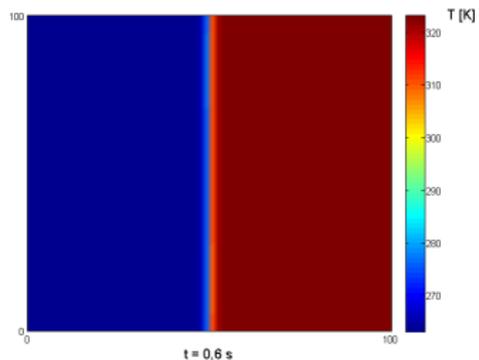


Abbildung: Temperaturverteilung durch Diffusion

Zusammenfassung und Ausblick

- Entwicklung eines **2d-Strömungslösers** mit guten Ergebnissen.
- Bei numerischen Simulationen stellt das Lösen von linearen Gleichungssystemen typischerweise den Hauptaufwand des Algorithmus dar: Hierfür wurde in CUDA ein parallelisiertes CG-Verfahren mit bis zu **35-facher Beschleunigung** implementiert.
- Die Angabe der Rechenleistung von GPUs sind theoretische Maximalwerte und nur bei speziellen Problemen erreichbar.
- Als Flaschenhals stellte sich dabei vor allem die **Speicheranbindung** heraus.
- Die neu entwickelten FERMI Karten stellen eine weitere, wesentliche Verbesserung der Performance dar.

