Programmierung mit CUDA Numerische Tests Fazit

<日本<br />
<日本</p>

# Lösung der zweidimensionalen Wirbeltransportgleichung auf NVIDIA Grafikkarten - Bachelorarbeit -

Numerische Methoden

### Manuel Baumann, Pavel Buran

Seminar des Fachgebiets Optimierung bei partiellen Differentialgleichungen

13. Januar 2011

<日本<br />
<日本</p>

# Gliederung

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

 Einleitung
 Physikalische Grundlagen
 Numerische Methoden
 Programmierung mit CUDA
 Numerische Tests
 Fazit

• Die Rechenleistung von Grafikkarten (GPUs) hat in den letzten zehn Jahren stark zugenommen:



- Die Programmierumgebung CUDA ermöglicht parallele Programmierung auf NVIDIA Grafikkarten.
- CUDA-Funktionen können direkt aus einem *C/C++* Code gestartet werden.

(D) (A) (A) (A)

Einleitung	Physikalische Grundlagen	Numerische Methoden	Programmierung mit CUDA	Numerische Tests	Fazit
	0000000			0 <b>0</b> 0000000	

Navier-Stokes-Gleichung

### Die Navier-Stokes-Gleichung (NSG)

Für inkompressible Newton-Fluide wird das Strömungsverhalten durch die NSG beschrieben:

$$rac{\partial ec{c}}{\partial t} + ec{c} \cdot \textit{ grad } ec{c} = ec{f} - rac{1}{
ho} \textit{ grad } p + 
u \Delta ec{c}$$

Hierbei bezeichnet:

- $\vec{c} = [u, v, w]^T \in \mathbb{R}^{2,3}$  die Geschwindigkeit des Fluids,
- $p \in \mathbb{R}$  den Druck,
- $\rho \in \mathbb{R}$  die Dichte und  $\nu \in \mathbb{R}$  die kinemat. Viskosität des Fluids,
- $\vec{f} \in \mathbb{R}^{2,3}$  die Summe der konservativen Kräfte.

Die NSG kann für inkompressible, ebene Strömungen in die so genannten **Wirbeltransportgleichung** überführt werden.

(周) (三) (三)

### Definition

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

$$\vec{\omega} := \mathit{rot}\left(\vec{c}
ight)$$

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

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



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

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

### Definition

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

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

(D) (A) (A)

Einleitung	Physikalische Grundlagen	Numerische Methoden	Programmierung mit CUDA	Numerische Tests	Fazit
	000000			0 <b>0</b> 0000000	

Herleitung der Wirbeltransportgleichung

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

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

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

・ 同 ト ・ ヨ ト ・ ヨ ト

э

Herleitung der Wirbeltransportgleichung

### Betrachte nun die NSG komponentenweise:

$$\frac{\partial u}{\partial t} + u \cdot \frac{\partial u}{\partial x_1} + v \cdot \frac{\partial u}{\partial x_2} = -\frac{\partial}{\partial x_1} (U + \frac{p}{\rho}) + \nu (\underbrace{\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}}_{=\Delta u}) \quad (\mathsf{I})$$

$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{u} \cdot \frac{\partial \mathbf{v}}{\partial x_1} + \mathbf{v} \cdot \frac{\partial \mathbf{v}}{\partial x_2} = -\frac{\partial}{\partial x_2} (U + \frac{p}{\rho}) + \nu (\underbrace{\frac{\partial^2 \mathbf{v}}{\partial x_1^2} + \frac{\partial^2 \mathbf{v}}{\partial x_2^2}}_{=\Delta \mathbf{v}}) \quad (\mathsf{II})$$

イロト イロト イヨト イヨト 三日

Einleitung

Physikalische Grundlagen

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

Herleitung der Wirbeltransportgleichung

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

$$\frac{\partial}{\partial t}\left(\underbrace{\frac{\partial v}{\partial x_1} - \frac{\partial u}{\partial x_2}}_{=\omega}\right) + \underbrace{u \cdot \frac{\partial^2 v}{\partial x_1^2} - u \cdot \frac{\partial^2 u}{\partial x_1 \partial x_2}}_{=u \cdot \frac{\partial \omega}{\partial x_1}} + \underbrace{v \cdot \frac{\partial^2 v}{\partial x_1 \partial x_2} - v \cdot \frac{\partial^2 u}{\partial x_2^2}}_{=v \cdot \frac{\partial \omega}{\partial x_2}} + \underbrace{\frac{\partial u}{\partial x_1} \cdot \frac{\partial v}{\partial x_1} + \frac{\partial v}{\partial x_1} \cdot \frac{\partial v}{\partial x_2} - \frac{\partial u}{\partial x_2} \cdot \frac{\partial u}{\partial x_1} - \frac{\partial v}{\partial x_2} \cdot \frac{\partial u}{\partial x_2}}_{=0 \text{ (folgt aus der Definition von }\Psi)}$$

$$= \underbrace{-\frac{\partial^2}{\partial x_1 \partial x_2}(U + \frac{p}{\rho}) + \frac{\partial^2}{\partial x_2 \partial x_1}(U + \frac{p}{\rho})}_{=0 \text{ (Satz von Schwarz)}} + \underbrace{v \cdot \frac{\partial^2 u}{\partial x_1 \partial x_2}}_{=v \Delta \omega \text{ (Linearität)}}$$

Man erhält somit die Wirbeltransportgleichung:

$$\frac{\partial \omega}{\partial t} + \vec{c} \cdot \operatorname{grad} \omega = \nu \Delta \omega \tag{2a}$$

イロト イヨト イヨト イヨト

E

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

(D) (A) (A) (A) (A)

Boussinesq-Annahme und Wärmeleitungsgleichung

### Die Boussinesq-Annahme

Die bisher als konstant angenommene Dichte  $\rho_0$  wird um einen temperaturabhängigen Term  $\delta\rho$  erweitert, so dass  $\rho = \rho_0 + \delta\rho$  gilt. Die **Boussinesq-Approximation** besagt, dass für Trägheits- und Reibungskräfte weiterhin angenommen werden kann, dass der temperaturinduzierte Dichteunterschied vernachlässigbar ist:

$$\rho = \rho_0 + \delta \rho \approx \rho_0 \quad \Leftrightarrow \quad \delta \rho \ll 1$$

Somit erweitert sich die Wirbeltransportgleichung zu:

$$\frac{\partial \omega}{\partial t} + \vec{c} \cdot \operatorname{grad} \omega = \nu \Delta \omega + \underbrace{\beta g}_{=:C} \frac{\partial T}{\partial x_1}, \quad (2b)$$

wobei als äußere Kraft  $\vec{f} = [0, -g]^T$  angenommen wird.

Boussinesq-Annahme und Wärmeleitungsgleichung

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

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

Hierbei wurde in der Konstanten a

- die Wärmeleitfähigkeit  $\lambda$ ,
- die Dichte  $\rho$  des Fluids,
- sowie die Wärmekapazität  $c_W$  des Fluids

berücksichtigt.

(D) (A) (A)

э

Programmierung mit CUDA Numerische Tests Fazit

臣

### Zusammenfassung

Es ist folgendes System von partiellen Differentialgleichungen zu lösen:

$$-\Delta \Psi = \omega$$
(1)  

$$\vec{c} = [u, v]^T = \left[\frac{\partial \Psi}{\partial x_2}, -\frac{\partial \Psi}{\partial x_1}\right]^T$$

$$\frac{\partial \omega}{\partial t} + \vec{c} \cdot \operatorname{grad} \omega = \nu \Delta \omega + C \frac{\partial T}{\partial x_1}$$
(2b)  

$$\frac{\partial T}{\partial t} + \vec{c} \cdot \operatorname{grad} T = a \Delta T$$
(3)

Dieses System wird mit der **Finiten-Volumen-Methode (FVM)** im Ort und einer impliziten Euler-Integration in der Zeit diskretisiert. Ortsdiskretisierung

Als **Diskretisierungsgebiet** wird ein rechteckiges Gebiet  $\Omega \subset \mathbb{R}^2$  gewählt, das in drei versetzte Gitter mit Schrittweite *h* äquidistant diskretisiert wird.



- Hauptgitter mit Mittelwerten für  $\zeta = \{\omega, \Psi, T\}$  innerhalb eines Kontrollvolumens
- Gitter 1 mit Geschwindigkeitswerten im Norden und Süden eines Kontrollvolumens (rot)
- Gitter 2 mit Geschwindigkeitswerten im Osten und Westen eines Kontrollvolumens (lila)

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

#### Poissongleichung

Anwenden der FVM auf die Poissongleichung:

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

Betrachte einzelnes finites Volumen (=Kontrollvolumen):

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

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.

Manuel Baumann, Pavel Buran

Lösung der Wirbeltransportgleichung mit CUDA

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

#### Poissongleichung

$$-\int_{\Gamma_{i,j}} \frac{\partial \Psi}{\partial \vec{n}} \, d\Gamma_{i,j} = -\int_{\Gamma_{N}} \frac{\partial \Psi}{\partial x_{2}} \, dx_{1} - \int_{\Gamma_{O}} \frac{\partial \Psi}{\partial x_{1}} \, dx_{2} + \int_{\Gamma_{S}} \frac{\partial \Psi}{\partial x_{2}} \, dx_{1} + \int_{\Gamma_{W}} \frac{\partial \Psi}{\partial x_{1}} \, dx_{2}$$
$$\approx -\frac{\Psi_{i,j+1} - \Psi_{i,j}}{h} h - \frac{\Psi_{i+1,j} - \Psi_{i,j}}{h} h + \frac{\Psi_{i,j} - \Psi_{i,j-1}}{h} h + \frac{\Psi_{i,j} - \Psi_{i-1,j}}{h} h$$
$$= -\Psi_{i,j+1} - \Psi_{i+1,j} + 4\Psi_{i,j} - \Psi_{i,j-1} - \Psi_{i-1,j}$$

$$\int_{\Omega_{i,j}} \omega \ d\Omega_{i,j} \approx h^2 \omega_{i,j}$$

Dies kann bei lexikographischer Anordnung in ein LGS geschrieben werden:  $A_h \Psi_h = \omega_h + b_h^{\Psi}$ , mit Systemmatrix

$$A_{h} := h^{-2} \begin{bmatrix} T_{A} & -E_{N_{1}} & & \\ -E_{N_{1}} & T_{A} & \ddots & \\ & \ddots & \ddots & -E_{N_{1}} \\ & & -E_{N_{1}} & T_{A} \end{bmatrix}, \ T_{A} := \begin{bmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{bmatrix}$$

und Randwertevektor

$$b_{h}^{\Psi} = h^{-2} [\Psi_{0,1} + \Psi_{1,0}, \Psi_{2,0}, ..., \Psi_{N_{1}-1,0}, \Psi_{N_{1},0} + \Psi_{N_{1}+1,1}, \Psi_{0,2}, ..., \Psi_{N_{1}+1,2}, ...]^{T}$$

(ロ) (部) (E) (E) (E)

Transportgleichung

Wirbeltransportgleichung und Wärmeleitungsgleichung lassen sich verallgemeinert schreiben als:

$$rac{\partial \zeta}{\partial t} = k_1 \Delta \zeta - ec{c} \cdot \textit{grad} \ \zeta + k_2 rac{\partial T}{\partial x_1},$$

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:

$$\sum_{i \in I, j \in J} \int_{\Omega_{i,j}} \frac{\partial \zeta}{\partial t} d\Omega_{i,j} = \sum_{i \in I, j \in J} \left( \underbrace{k_1 \int_{\Omega_{i,j}} \Delta \zeta d\Omega_{i,j}}_{\text{Laplace-Term}} - \underbrace{\int_{\Omega_{i,j}} \vec{c} \cdot \operatorname{grad} \zeta d\Omega_{i,j}}_{\text{Temperatur-Term}} + \underbrace{k_2 \int_{\Omega_{i,j}} \frac{\partial T}{\partial x_1} d\Omega_{i,j}}_{\text{Temperatur-Term}} \right)$$
$$\approx B_h \zeta_h - C_h \zeta_h + D_h T_h + b_h$$

#### **Transportgleichung**

### Laplace-Term: Der Laplace-Term kann analog zur Poissongleichung diskretisiert werden:

$$k_1 \int_{\Omega_{i,j}} \Delta \zeta d\Omega_{i,j} = k_1 \int_{\partial \Omega_{i,j}} \operatorname{grad} \zeta \cdot \vec{n} \, d\partial \Omega_{i,j} \approx k_1 (\zeta^N + \zeta^O - 4\zeta^P + \zeta^S + \zeta^W),$$

so dass sich für die Matrix  $B_h := -k_1 h^2 A_h$  ergibt.

### Temperatur-Term:

Für den Temperatur-Term muss eine erste Ableitung diskretisiert werden. Dies geschieht via:

$$k_2 \int_{\Omega_{i,j}} \frac{\partial T}{\partial x_1} d\Omega_{i,j} = k_2 \int_{\mathcal{S}}^{\mathcal{N}} \int_{\mathcal{W}}^{\mathcal{O}} \frac{\partial T}{\partial x_1} dx_1 dx_2 = \frac{k_2}{2} \int_{\mathcal{S}}^{\mathcal{N}} (T^{\mathcal{O}} - T^{\mathcal{W}}) dx_2 \approx \frac{k_2 h}{2} (T^{\mathcal{O}} - T^{\mathcal{W}})$$

und führt in Matrixschreibweise auf:

$$D_{h} := \frac{k_{2}h}{2} \begin{bmatrix} T_{D} & & \\ & T_{D} & \\ & & \ddots & \\ & & & T_{D} \end{bmatrix}, \qquad T_{D} := \begin{bmatrix} 0 & -1 & & \\ 1 & 0 & \ddots & \\ & \ddots & \ddots & -1 \\ & & 1 & 0 \end{bmatrix}$$

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

(D) (A) (A) (A) (A)

2

#### Transportgleichung

### Konvektionsterm:

$$\begin{split} &\int_{\Omega_{i,j}} \vec{c} \cdot \operatorname{grad} \zeta d\Omega_{i,j} = \int_{\Omega_{i,j}} u \frac{\partial \zeta}{\partial x_1} + v \frac{\partial \zeta}{\partial x_2} d\Omega_{i,j} \\ &= \int_{\mathcal{S}}^{\mathcal{N}} \int_{\mathcal{W}}^{\mathcal{O}} u \frac{\partial \zeta}{\partial x_1} dx_1 dx_2 + \int_{\mathcal{W}}^{\mathcal{O}} \int_{\mathcal{S}}^{\mathcal{N}} v \frac{\partial \zeta}{\partial x_2} dx_2 dx_1 \\ &\approx \int_{\mathcal{S}}^{\mathcal{N}} \frac{h}{2} \left( u^O \frac{\zeta^O - \zeta^P}{h} + u^W \frac{\zeta^P - \zeta^W}{h} \right) dx_2 + \int_{\mathcal{W}}^{\mathcal{O}} \frac{h}{2} \left( v^N \frac{\zeta^N - \zeta^P}{h} + v^S \frac{\zeta^P - \zeta^S}{h} \right) dx_1 \\ &\approx \frac{h}{2} \left( u^O \zeta^O + (u^W - u^O) \zeta^P - u^W \zeta^W \right) + \frac{h}{2} \left( v^N \zeta^N + (v^S - v^N) \zeta^P - v^S \zeta^S \right) \end{split}$$

Diese erste Diskretisierung war instabil. Daher wurde unter Einführung von [x, y] := max(x, y) ein *Upwind*-Verfahren implementiert:

$$\begin{split} \int_{\Omega_{i,j}} \vec{c} \operatorname{grad} \zeta d\Omega_{i,j} &\approx h(-\llbracket -v^N, 0 \rrbracket \zeta^N - \llbracket -u^O, 0 \rrbracket \zeta^O + \llbracket v^S, 0 \rrbracket \zeta^S - \llbracket u^W, 0 \rrbracket \zeta^W \\ &- (\llbracket u^W, 0 \rrbracket + \llbracket -u^O, 0 \rrbracket + \llbracket v^S, 0 \rrbracket + \llbracket -v^N, 0 \rrbracket) \zeta^P) \end{split}$$

・ 同 ト ・ ヨ ト ・ ヨ ト

#### Zeitintegration

Nachdem die Transportgleichung im Ort durch die Matrizen  $B_h$ ,  $C_h$  und  $D_h$  diskretisiert wurde, erfolgt die Zeitintegration durch eine Variation des impliziten Eulerverfahren:

$$\int_{\Omega_{i,j}} \frac{\partial \zeta}{\partial t} d\Omega_{i,j} \approx h^2 \frac{\zeta_h^{i+1} - \zeta_h^i}{\delta t} = B_h \zeta_h^{i+1} - C_h^i \zeta_h^i + D_h T_h^i + b_h^i,$$

### mit Zeitschrittweite $\delta t$ .

Durch elementare Umformungen führt dies zu einem LGS:

$$\underbrace{(E_{n} - \frac{\delta t}{h^{2}} B_{h})}_{Systemmatrix} \zeta_{h}^{i+1} = \underbrace{(E_{n} - \frac{\delta t}{h^{2}} C_{h}^{i})\zeta_{h}^{i} + \frac{\delta t}{h^{2}} D_{h} T_{h}^{i} + \frac{\delta t}{h^{2}} b_{h}^{i}}_{Rechte Seite}$$

Als Richtwert für den Zeitschritt, gilt die Courant-Zahl  $Co := \frac{c_{max} \cdot \delta t}{h} < 1.$ 

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

イロト イポト イヨト イヨト

æ

Rand- und Anfangsbedingungen

### 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  $\Psi_h, \omega_h, T_h$  auf  $\partial \Omega$
- Anfangswerte  $T_h^0, w_h^0$  im Innern von  $\Omega$

Rand- und Anfangsbedingungen

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  $\Omega$

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:

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

Programmierung mit CUDA Numerische Tests Fazit

Rand- und Anfangsbedingungen

Berechnung von Randwerten für die Stromfunktion  $\Psi_h$  durch Integration der Gleichungen:

$$u = \frac{\partial \Psi}{\partial x_2} \qquad \mathbf{v} = -\frac{\partial \Psi}{\partial x_1} \qquad \rightarrow \qquad \Psi = \int_{\Gamma_4} u \, dy \qquad = \int_{\Gamma_4} u \, dy \qquad = \int_{\Gamma_2} u \, dy$$

Berechnung von Randwerten für die Wirbelstärke  $\omega_h$  nach:

 $\rightarrow$ 

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



Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

#### Aufbau des Solvers



Manuel Baumann, Pavel Buran Lösung

Lösung der Wirbeltransportgleichung mit CUDA

(D) (A) (A) (A)

#### Was ist CUDA ?

- Die Rechenleistung von GPUs ist deutlich höher als von CPUs, da sie **parallel** arbeiten.
- Besonders interessant fürs wissenschaftliche Rechnen bisher nur für Grafikberechnungen nutzbar.

![](_page_23_Figure_6.jpeg)

- CUDA ist eine 2006 entwickelte Softwareumgebung zur Programmierung von NVIDIA Grafikkarten
- C/C++ Spracherweiterung

Numerische Methoden

#### Schematischer Aufbau

![](_page_24_Figure_4.jpeg)

### Eigenschaften der Grafikkarte:

CUDA Device Query	
There are 1 CUDA devices.	
CUDA Device #0	
Major revision number:	1
Minor revision number:	3
Name:	Tesla C1060
Total global memory:	4294770688
Total shared memory per block:	16384
Total registers per block:	16384
Warp size:	32
Clock rate:	1296MHz
Memory Bandwidth:	102 GB/sec
Total constant memory:	65536
Number of multiprocessors:	30

(D) (A) (A) (A)

э

- Auf einem SM werden 32 Threads (1 Warp) parallel ausgeführt.
- Dies geschieht nach dem SIMD-Prinzip, daher sollten Programmverzweigungen vermieden werden (Divergency).
- Rechenleistung:

$$\begin{split} P_{single} &= 3Flops \cdot 1296 \cdot 10^{6} \text{Hz} \cdot 240 \approx 933 \text{GFlops/s} \\ P_{double} &= 2Flops \cdot 1296 \cdot 10^{6} \text{Hz} \cdot 30 \approx 78.8 \text{GFlops/s} \end{split}$$

Numerische Methoden

#### Das CUDA-Programmiermodell

![](_page_25_Figure_4.jpeg)

- Serielle Anteil des Programms wird auf dem Host (CPU) ausgeführt.
- Parallele Anteil des Programms wird auf das **Device** (GPU) ausgelagert.
- Wechsel vom Host auf das Device erfolgt über Funktionen, welche als Kernel bezeichnet werden.
- Auf jedem Thread wird eine Instanz des Kernels ausgeführt.

э

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

#### Das CUDA-Programmiermodell

![](_page_26_Figure_4.jpeg)

- Threads sind in Blöcke (**Blocks**) gruppiert, welche wiederum in Gitter (**Grids**) gruppiert sind.
- Die Threadindizierung erfolgt jeweils dreidimensional.
- Threads innerhalb eines Blocks werden auf einem Multiprozesser ausgeführt, können daher synchronisiert werden und über den Shared-Memory kommunizieren.
- Blöcke werden unabhängig voneinander abgearbeitet.

Einleitung	Physikalische	Grundlagen

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

#### Das CUDA-Programmiermodell

![](_page_27_Figure_4.jpeg)

- Arbeitsspeicher (Host-Memory): Schlechteste Bandbreite, daher Kopiervorgänge vermeiden.
- Globaler Speicher (Global-Memory): Bandbreite gering im Vergleich zur Rechenleistung. Hier sollte Coalescing angestrebt werden sowie Zugriffe minimiert werden.
- Register: Schnellster Speicher.
- Gemeinsamer Speicher (Shared-Memory): Kommunikation innerhalb eines Blocks. Gleiche Geschwindigkeit wie Register, sofern keine Bankkonflikte auftreten.

Texturen:

Ähnlich wie Global Memory, lesender Zugriff gecacht. 

#### 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 v. *d erg:
  cudaMalloc((void **)&d x, dim*sizeof(double));
  cudaMalloc((void **)&dy, 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;
                                                  <ロ> (四) (四) (三) (三) (三)
```

Optimierungen am Beispiel

### Erinnerung:

In dem Solver müssen in jedem Zeitschritt drei LGS der Form  $Ax = b, A \in \mathbb{R}^{N,N}$ , gelöst werden.

Hierfür soll das CG-Verfahren verwendet werden:

- gehört zur Klasse der Krylov-Unterraum-Verfahren
- iteratives Lösungsverfahren
- anwendbar bei symmetrisch, positiv definiter Systemmatrix A
- besonders geeignet für dünnbesetzte Matrix A
- konvergiert nach spätestens *N* Iterationen in exakter Arithmetik

Im Folgenden soll eine Parallelisierung des CG-Verfahrens präsentiert werden.

э

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

Optimierungen am Beispiel

### **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 Ap_{j-1})

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

r_j = r_{j-1} - \gamma_{j-1}Ap_{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)

æ

Einleitung	Physikalische Grundlagen	Numerische Methoden	Programmierung mit CUDA	Numerische Tests	Fazit
Optimierun	gen am Beispiel				
Skala	rorodukt				

### 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 Ergebisse der Blöcke.

![](_page_31_Figure_6.jpeg)

Optimierungen am Beispiel

### **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){</pre>
    if (\text{threadIdx} \cdot 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];
}
```

<ロ> (四) (四) (三) (三) (三)

![](_page_33_Picture_0.jpeg)

### Verbesserung: Mit Coalescing

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

![](_page_33_Figure_4.jpeg)

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

![](_page_34_Figure_0.jpeg)

Programmierung mit CUDA Numerische Tests Fazit

э

Optimierungen am Beispiel

### Verbesserung: Ohne Divergency

![](_page_34_Figure_4.jpeg)

Abbildung: Links mit Divergency, rechts ohne

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

Optimierungen am Beispiel

### Verbesserung: Ohne Bankkonflikte

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

![](_page_35_Figure_5.jpeg)

Abbildung: Links mit Bankkonflikten, rechts ohne

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

э

#### Optimierungen am Beispiel

### 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

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

#### Optimierungen am Beispiel

# Zeitmessungen: Skalarprodukt

![](_page_37_Figure_5.jpeg)

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

$$P_{single} = \frac{102GByte/s \cdot 2Flops}{2 \cdot 4Byte} = 25.5GFlops/s \quad P_{double} = \frac{102GByte/s \cdot 2Flops}{2 \cdot 8Byte} = 12.75GFlops/s$$
Manuel Baumann, Pavel Buran
Lösung der Wirbeltransportgleichung mit CUDA

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

イロト イポト イヨト イヨト

э

Optimierungen am Beispiel

# Matrix-Vektor Multiplikation

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

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

Generelle Idee:

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

Programmierung mit CUDA Numerische Tests Fazit

<ロ> (四) (四) (三) (三) (三)

Optimierungen am Beispiel

### Erste Implementierung:

```
global void k spmv dia1 (Real∗y,
                                            //Ergebnisvektor
                                            //data-Matrix
                            Real* data,
                            int *offsets.
                                            //Offsetvektor
                            Real *x.
                                            //Vektor
                            int dim r , //#Zeilen der Sparsematrix
                            int dim <sup>-</sup> 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 \ge 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) Einleitung Physikalische Grundlagen Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit 

(ロ) (部) (E) (E) (E)

Optimierungen am Beispiel

### 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)</pre>
    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

Einleitung Physikalische Grundlagen Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit 

#### Optimierungen am Beispiel

### 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:
          shared int shared offsets[];
  extern
  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 + ro w];
      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;
}
                                                 ・ロト ・部ト ・ヨト ・ヨト … ヨ
```

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

э

#### Optimierungen am Beispiel

# Zeitmessungen: Matrix-Vektor Multiplikation

![](_page_42_Figure_5.jpeg)

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

$$\begin{split} P_{single} &= \frac{102GByte/s \cdot 5 \cdot dim \cdot 2Flops}{(5+5+1) \cdot dim \cdot 4Byte} = 23.2GFlops/s\\ P_{double} &= \frac{102GByte/s \cdot 5 \cdot dim \cdot 2Flops}{(5+5+1) \cdot dim \cdot 8Byte} = 11.6GFlops/s \end{split}$$

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

イロト イポト イヨト イヨト

Zeitmessungen

# 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 Arbeisspeicher lädt und das Ergbnis 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.

Programmierung mit CUDA Numerische Tests Fazit

#### Zeitmessungen

Gitter	LGS	Iter.	CG_GP	U_DIA1	CG_GP	U_DIA2	CG_CP	U_DIA
64 × 64	Ah	225	22.52 <i>ms</i>	0.77 <i>GF</i>	22.06 <i>ms</i>	0.79 <i>GF</i>	156 <i>ms</i>	0.11 <i>GF</i>
64  imes 64	Bh	15	1.63 <i>ms</i>	0.74 <i>GF</i>	1.50 <i>ms</i>	0.80 <i>GF</i>	10.9 <i>ms</i>	0.11 <i>GF</i>
64  imes 64	Bh	1	0.29 <i>ms</i>	0.45 <i>GF</i>	0.16 <i>ms</i>	0.81 <i>GF</i>	0.80 <i>ms</i>	0.16 <i>GF</i>
256  imes 256	Ah	932	151.5 <i>ms</i>	7.94 <i>GF</i>	149.9 <i>ms</i>	8.03 <i>GF</i>	12.57 <i>s</i>	0.10 <i>GF</i>
256 imes256	Bh	54	10.34 <i>ms</i>	6.82 <i>GF</i>	8.78 <i>ms</i>	8.04 <i>GF</i>	631 <i>ms</i>	0.11 <i>GF</i>
256 imes256	Bh	1	1.70 <i>ms</i>	1.30 <i>GF</i>	0.26 <i>ms</i>	8.29 <i>GF</i>	13.29 <i>ms</i>	0.17 <i>GF</i>
1024  imes 1024	Ah	3786	4.68 <i>s</i>	16.9 <i>GF</i>	4.65 <i>s</i>	17.0 <i>GF</i>	11.4 <i>min</i>	0.12 <i>GF</i>
1024  imes 1024	Bh	215	0.28 <i>s</i>	16.3 <i>GF</i>	0.27 <i>s</i>	17.0 <i>GF</i>	40.8 <i>s</i>	0.11 <i>GF</i>
1024  imes 1024	Bh	2	13.6 <i>ms</i>	4.16 <i>GF</i>	3.24 <i>ms</i>	17.4 <i>GF</i>	0.4 <i>s</i>	0.13 <i>GF</i>

Gitter	LGS	Iter.	CG_GPI	J_DIA1	CG_GP	U_DIA2	CG_CP	U_DIA
64 × 64	Ah	202	22.75 <i>ms</i>	0.69 <i>GF</i>	22.96 <i>ms</i>	0.68 <i>GF</i>	137 <i>ms</i>	0.11 <i>GF</i>
64  imes 64	B	15	1.97 <i>ms</i>	0.61 <i>GF</i>	1.68 <i>ms</i>	0.72 <i>GF</i>	10.8 <i>ms</i>	0.11 <i>GF</i>
64  imes 64	Bh	1	0.36 <i>ms</i>	0.36 <i>GF</i>	0.18 <i>ms</i>	0.74 <i>GF</i>	1.31 <i>ms</i>	0.10 <i>GF</i>
256  imes 256	Ah	801	182.4 <i>ms</i>	5.67 <i>GF</i>	180.5 <i>ms</i>	5.73 <i>GF</i>	9011 <i>ms</i>	0.11 <i>GF</i>
256 imes256	B	54	14.46 <i>ms</i>	4.88 <i>GF</i>	12.21 <i>ms</i>	5.78 <i>GF</i>	624 <i>ms</i>	0.11 <i>GF</i>
256 imes256	Bh	1	2.48 <i>ms</i>	0.88 <i>GF</i>	0.36 <i>ms</i>	6.03 <i>GF</i>	22.35 <i>ms</i>	0.10 <i>GF</i>
1024 imes1024	Ah	3564	7.45 <i>s</i>	9.99 <i>GF</i>	7.39 <i>s</i>	10.08 <i>GF</i>	10.1 <i>min</i>	0.12 <i>GF</i>
1024  imes 1024	Bh	215	0.47 <i>s</i>	9.67 <i>GF</i>	0.45 <i>s</i>	10.04 <i>GF</i>	40.66 <i>s</i>	0.11 <i>GF</i>
1024  imes 1024	Bh	2	21.8 <i>ms</i>	2.59 <i>GF</i>	5.36 <i>ms</i>	10.52 <i>GF</i>	0.57 <i>s</i>	0.11 <i>GF</i>

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

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

æ

#### Zeitmessungen

# SpeedUp: Zeitschritt des Solvers

Gitter	Version	Gesamt	LGS lösen	Abspeichern	Rest
64  imes 64	GPU 1	53.35 <i>ms</i>	26.06 <i>ms</i>	26.70 <i>ms</i>	0.58 <i>ms</i>
64  imes 64	GPU <sup>2</sup>	52.89 <i>ms</i>	25.41 <i>ms</i>	27.19 <i>ms</i>	0.30 <i>ms</i>
64  imes 64	CPU	204.6 <i>ms</i>	176.7 <i>ms</i>	27.40 <i>ms</i>	0.50 <i>ms</i>
256  imes 256	GPU 1	0.53 <i>s</i>	0.17 <i>s</i>	0.34 <i>s</i>	6.72 <i>ms</i>
256  imes 256	GPU <sup>2</sup>	0.52 <i>s</i>	0.17 <i>s</i>	0.35 <i>s</i>	0.85 <i>ms</i>
256 imes256	CPU <sup>-</sup>	13.02 <i>s</i>	12.67 <i>s</i>	0.34 <i>s</i>	6.83 <i>ms</i>
1024 imes1024	GPU 1	12.00 <i>s</i>	5.27 <i>s</i>	6.62 <i>s</i>	0.11s
1024 imes1024	GPU <sup>2</sup>	11.74 <i>s</i>	5.20 <i>s</i>	6.53 <i>s</i>	10.35 <i>ms</i>
1024 $ imes$ 1024	CPU <sup>-</sup>	9.13 <i>min</i>	9.03 <i>min</i>	5.50 <i>s</i>	0.13 <i>s</i>

Gitter	Version	Gesamt	LGS lösen	Abspeichern	Rest
64  imes 64	GPU 1	52.84 <i>ms</i>	24.98 <i>ms</i>	27.36 <i>ms</i>	0.5 <i>ms</i>
64 imes 64	GPU <sup>2</sup>	51.68 <i>ms</i>	23.97 <i>ms</i>	27.41 <i>ms</i>	0.31 <i>ms</i>
64 imes 64	CPU <sup>-</sup>	178.4 <i>ms</i>	150.1 <i>ms</i>	27.76 <i>ms</i>	0.5 <i>ms</i>
256 imes256	GPU 1	0.58 <i>s</i>	0.21 <i>s</i>	0.37 <i>s</i>	7.1 <i>ms</i>
256 imes256	GPU <sup>2</sup>	0.56 <i>s</i>	0.20 <i>s</i>	0.35 <i>s</i>	0.93 <i>ms</i>
256 imes256	CPU <sup>-</sup>	9.90 <i>s</i>	9.56 <i>s</i>	0.34 <i>s</i>	7.06 <i>ms</i>
1024 imes1024	GPU 1	16.02 <i>s</i>	7.63 <i>s</i>	8.27 <i>s</i>	0.13 <i>s</i>
1024 $ imes$ 1024	GPU <sup>2</sup>	15.67 <i>s</i>	7.41 <i>s</i>	8.25 <i>s</i>	12.3 <i>ms</i>
1024 $ imes$ 1024	CPU <sup>-</sup>	9.15 <i>min</i>	9.07 <i>min</i>	5.45 <i>s</i>	0.13 <i>s</i>

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

Zur Validierung der Implementierung werden Numerische Tests durchgeführt.

Hierzu werden die verwendeten Stoffparameter auf die Werte von Wasser bei einem Druck von p = 1bar und einer Umgebungstemperatur von  $T_0 = 20$  °C gesetzt:

Parameter	Wert
kinematische Viskosität	$\nu = 1000 \ m^2 s^{-1}$
Wärmeleitfähigkeit	$\lambda = 0.597 \; Wm^{-1}K^{-1}$
Dichte	$ ho=$ 1000 kg $m^{-3}$
Wärmekapazität	$c_w = 4187 \; J \; kg^{-1} \; K^{-1}$
therm. Ausdehnungskoeffizient	$eta = 0.21 \cdot 10^{-3} \; K^{-1}$

Einleitung	Physikalische Grundlagen	Numerische Methoden	Programmierung mit CUDA	Numerische Tests	Fazit
				0000000	
D.I. Carl					

### Versuchsaufbau:

![](_page_47_Figure_2.jpeg)

![](_page_47_Picture_3.jpeg)

Abbildung: Driven Cavity Versuch mit Re = 1

・ 同 ト ・ ヨ ト ・ ヨ ト

臣

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

Einleitung	Physikalische Grundlagen	Numerische Methoden	Programmierung mit CUDA	Numerische Tests	Fazit
			000000000000000000000000000000000000000	0000000	

#### **Driven Cavity**

![](_page_48_Figure_2.jpeg)

![](_page_48_Figure_3.jpeg)

Abbildung: Driven Cavity Versuch mit Re = 100

Abbildung: Driven Cavity Versuch mit Re = 3.333

イロン イヨン イヨン イヨン

æ

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

Ebene Poisseuille-Strömung

### Versuchsaufbau:

![](_page_49_Figure_5.jpeg)

![](_page_49_Figure_6.jpeg)

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

æ

・ロト ・聞ト ・ヨト ・ヨト

Abbildung: Geschwindigkeitsprofil bei der Kanalströmung

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

・ロト ・ 日 ・ ・ ヨ ・ ・ ヨ ・

æ

#### Ebene Poisseuille-Strömung

![](_page_50_Figure_4.jpeg)

Abbildung: Temperaturverteilung bei einer Kanalströmung zu verschiedenen Zeitpunkten

Einleitung	Physikalische Grundlagen	Numerische Methoden	Programmierung mit CUDA	Numerische Tests	Fazit
	00000000	00000000000	000000000000000000000000000000000000000	000000000	
Mischung z	weier Fluide				

### Versuchsaufbau:

![](_page_51_Figure_3.jpeg)

![](_page_51_Figure_4.jpeg)

Abbildung: Geschwindigkeitsverteilung bei der Mischung zweier Strömungen

A (2) > (

E

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

#### Mischung zweier Fluide

![](_page_52_Figure_4.jpeg)

### Abbildung: Temperaturverteilung zu verschiedenen Zeitpunkten

Manuel Baumann, Pavel Buran Lösung der Wirbeltransportgleichung mit CUDA

Einleitung	Physikalische Grundlagen	Numerische Methoden	Programmierung mit CUDA	Numerische Tests ○○○○○○●○	Fazit
Diffusion					

### Versuchsaufbau:

![](_page_53_Figure_2.jpeg)

![](_page_53_Figure_3.jpeg)

Abbildung: Geschwindigkeitsfeld durch Diffusion

 Bei diesem Versuch soll die Wärmeleitung die Wärmekonvektion überwiegen.

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

臣

#### Diffusion

![](_page_54_Figure_4.jpeg)

# Abbildung: Temperaturverteilung durch Diffusion

- 4 同 6 - 4 目 6 - 4 目 6

# 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.

Numerische Methoden

Programmierung mit CUDA Numerische Tests Fazit

3

# Vielen Dank an Prof. Bärwolff für die Betreuung und Danke für Ihre Aufmerksamkeit!

Gibt es Fragen / Anmerkungen ?