Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2011-01

Bachelorarbeit

im Studiengang "Angewandte Informatik"

CUDA-basierte Implementierung polynomieller Vorkonditionierer für Krylov-Methoden

Christoph Pfaffenbach

am Institut für

Numerische und Angewandte Mathematik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

21. November 2011

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel. +49 (5 51) 39-1 44 14

Fax +49 (5 51) 39-1 44 15

Email office@informatik.uni-goettingen.de

WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 21. November 2011

Bachelorarbeit

CUDA-basierte Implementierung polynomieller Vorkonditionierer für Krylov-Methoden

Christoph Pfaffenbach

21. November 2011

Betreut durch Prof. Dr. G. Lube und Dipl.-Math. S. Kramer
Institut für Numerische und Angewandte Mathematik
Georg-August-Universität Göttingen

Inhaltsverzeichnis

Danksagung

An dieser Stelle möchte ich mich ganz herzlich für die wissenschaftliche Betreuung bei Herrn Dipl.-Math. Stephan Kramer bedanken. Ich bedanke mich ebenfalls für die Betreuung und die Erstellung des Erstgutachtens bei Herrn Prof. Dr. Gert Lube. Mein Dank gilt auch dem Herrn Dr. Jochen Schulz für die Erstellung des Zweitgutachtens. Für die Erstellung und Bereitstellung des Testmaterials möchte ich mich zudem bei Markus Rösler und Ralf Gritzki von der Technischen Universität Dresden bedanken.

1. Einleitung

In den folgenden Abschnitten werden die Motivation zur Arbeit und die Ziele vorgestellt. Dabei wird dem Leser ein Einblick in die Vorgehensweise bei der Erstellung gegeben werden, um eine Grundlage für die weiteren Kapitel zu schaffen.

1.1. Motivation

Die Motivation bezüglich dieser Arbeit bestand zum großen Teil darin, die BLANC-Bibliothek weiter zu vervollständigen. BLANC (*Basic Linear Algebra and Numerical Computing in C*), siehe [16], ist eine in der prozeduralen Programmiersprache C in den 90er Jahren im Institut für Numerische und Angewandte Mathematik geschriebene Bibliothek. Diese bietet alle nötigen Datenstrukturen zum Lösen dünn besetzter Gleichungssysteme, wie sie im Rahmen von Strömungssimulationen auftreten. Im einzelnen sind dies Matrizen, Vektoren, Ein- & Ausgaberoutinen, eine Reihe von Krylov-Lösern, insbesondere CG, CGS, TFQMR, GMRES, QMRCGSTAB und BICGSTAB. Zudem stehen noch verschiedene Vorkonditionierer (SOR, SSOR, ILU, JACOBI) zur Verfügung.
BLANC wird seit mehr als 15 Jahren erfolgreich im Rahmen von Raumluftströmungssimulationen mittels des Finite-Elemente-Programmpakets PNS (*Parallel Navier-Stockes*) eingesetzt. PNS ist Teil einer größeren Infrastruktur zur energetischen Gebäudesimulation, so dass der Bedarf entstand, diese mit minimalem Aufwand auf moderne, parallel arbeitende Hardwarearchitektur zu portieren. Die Wahl der Parallelisierungstechnik fiel dabei auf CUDA (*Compute Unified Device Architecture*). Diese Technik stellt eine Erweiterung der Programmiersprache C um einige Schlüsselwörter dar, so dass auf der Grafikkarte auszuführende Teile eines in C geschriebenen Programms mit wenig Aufwand portiert werden können.

1.2. Ziel der Arbeit

Das Ziel dieser Arbeit ist es, einen parallelen Vorkonditionierer für große lineare Gleichungssysteme zu implementieren. Bei der Technik handelt es sich um die *polynomielle Vorkonditionierung*. Diese Routine wird in die bereits genutzte BLANC-Bibliothek integriert und soll mittels CUDA parallelisiert werden. Diese Arbeit setzt auf einem zuvor absolvierten forschungsbezogenen Praktikum auf, in dem es darum ging, BLANC mit CUDA zu reimplementieren. Es reicht aus, die Kernfunktionen zu ersetzen, da ein großer Teil der Löser auf diese Routinen zurückgreifen. Um die BLANC-Bibliothek noch effizienter zu machen, waren die oben genannten Vorkonditionierungen zu integrieren, da die Matrix durch die Vorkonditionierung bessere Eigenschaften besitzt, die sich im Anschluss beim iterativen Lösen als Vorteil erweisen können.
Eine weitere Technik, die von mir behandelt wird, ist das Vorkonditionieren mittels einer dünn besetzten approximierten Inversen. Diese Thematik wird allerdings nicht weiter im Rahmen dieser Bachelor-Arbeit, sondern gesondert in einer geplanten Publikation unter dem Titel „Parallel Preconditioning Strategies for Decoupled Indoor Air Flow Simulation“ von S. Kramer [1], G. Lube [1], R. Gritzki [2], M. Rösler [2] und mir [1] vorgestellt.

[1] Institut für Numerische und Angewandte Mathematik an der Georg-August-Universität Göttingen, Deutschland.
[2] Institut für Energietechnik an der Technischen Universität Dresden, Deutschland.

2. Grundlagen

Ein Vorkonditionierer ist eine implizite oder explizite Modifikation eines linearen Gleichungssystems. Mittels dieser Modifikation wird es „einfacher“, das Gleichungssystem durch eine iterative Methode zu lösen, zum Beispiel durch Reskalierung aller Zeilen eines linearen Gleichungssystems, damit alle Diagonaleinträge 1 sind. Das Problem wird so umgeformt, dass die Lösung erhalten bleibt, sich jedoch für das gewählte numerische Lösungsverfahren günstigere Eigenschaften, wie bessere Kondition oder schnellere Konvergenz, ergeben. Bei der Vorkondtionierung unterscheidet man Linksvorkonditionierung, bei der das Gleichungssystem $Ax = b$ von links mit einer regulären Matrix multipliziert wird, das heißt $MAx = Mb$, und Rechtsvorkonditionierung, bei der das Gleichungssystem $AMy = b$ mit $y = M^{-1}x$ gelöst wird. Ein Vorkonditionierer sollte die Inverse von A mit geringstmöglichem Aufwand bestmöglich approximieren. Dieses vorkonditionierte System kann anschließend durch ein Krylov Unterraum-Verfahren gelöst werden und braucht in der Regel weniger Schritte bis zur Konvergenz als das nicht vorkonditionierte Verfahren.

2.1. Krylov-Unterraum Methoden

Sei

$$Ax = b, \qquad A \in \mathbb{R}^{n \times n}, \qquad x, b \in R^n \tag{2.1}$$

ein lineares Gleichungssystem mit einer Startlösung x_0 und einem Residuum $r_0 = b - Ax_0$. Eine Krylov-Unterraum Methode spannt einen Unterraum

$$\mathcal{K}_m(A, r_0) = span\left\{r_0, Ar_0, A^2r_0, \ldots, A^{m-1}r_0\right\} \tag{2.2}$$

mit der Dimension m auf. Fast alle Verfahren finden eine bessere Näherungslösung

$$x_m \in x_0 + \mathcal{K}_m \tag{2.3}$$

mit der Bedingung, dass der Vektor $b - Ax_m$ orthogonal zu allen anderen Vektoren steht, eine sogenannte Projektion. Diese Bedingung wird *Galerkin-Petrov* Bedingung genannt. Die verschiedenen Krylov-Unterraum Methoden unterscheiden sich in der Wahl des Unterraums $\mathcal{L}_m$ und Ausnutzung von verschiedenen Eigenschaften der Matrix A, welche das Verfahren beschleunigen, aber wiederum die Anwendbarkeit einschränken.

2.2. Polynomielle Vorkonditionierung

Eine polynomiell vorkonditionierte Matrix M ist durch

$$M^{-1} = s(A) \tag{2.4}$$

definiert, bei der s ein Polynom

$$s(A) = \sum_{i=0}^{n} c_i A^i = c_n A^n + c_{n-1} A^{n-1} + \ldots + c_2 A^2 + c_1 A + c_0, \qquad n \geq 0 \tag{2.5}$$

von niedrigem Grad ist. Auf diese Weise wird das ursprüngliche durch das vorkonditionierte System

$$s(A)Ax = s(A)b \tag{2.6}$$

ersetzt. Dieser Ansatz wird durch die gute Performance von Matrix-Vektor Operationen auf der GPU motiviert, da die Matrizen $s(A)$ oder $As(A)$ nicht explizit aufgestellt werden müssen, sondern $As(A)v$ mit einem beliebigen Vektor v als eine Reihe von Matrix-Vektor Produkten berechnet werden kann. Bei der polynomiellen Vorkonditionierung gibt es verschiedene Ansätze, welche Art von Polynomen man wählen sollte. In dieser Arbeit werden allerdings nur Faber-Polynomen verwendet.

Die generelle Aufgabe eines polynomiellen Vorkonditionierers ist die vage Abschätzung eines Gebiets $D \in \mathbb{C}$, so dass es das Spektrum $\sigma(A)$ der Matrix A umschließt. Generell gibt es zwei Möglichkeiten. Zum einen kann man ein Polygon konstruieren auf Grund-

lage der Schwarz-Christoffel-Abbildung, siehe [18], oder D ist das Ergebnis konformer Abbildungen auf eine Ellipse, siehe [8]. Auf die eine oder andere Art kommt man mit Faber-Polynomen zu einem Vorkonditionierer, der sich mit dem Zugang über Chebychev-Polynome deckt, wenn D Moebius-äquivalent zu einer Ebene ist. Vorläufige Ergebnisse über die Brauchbarkeit von polynomiellen Vorkonditionierern um einen CUDA-basierten Vorkonditionierer für symmetrische Matrizen zu implementieren, findet man in [10].

In diesem Abschnitt wird der Zugang über [8] näher erläutert und zudem ein Ausblick auf die CUDA-basierte Implementierung der verallgemeinerten Arnoldi-Bratwurst-Faber (ABF) Methode gemacht. Aber es werden auch andere Gebiete, die konformäquivalent zu einer Ellipse sind, ausprobiert. Der Hauptvorteil von ABF-ähnlichen Methoden ist, dass der Rand des Gebietes analytisch beschrieben ist, im Gegensatz zum Schwarz-Christoffel Ansatz, bei dem der Rand numerisch bestimmt werden muss, in dem eine Integral-Gleichung gelöst wird. Dies kann, wie man in der Literatur sieht, ein teurer Prozess sein, der anfällig für Rundungsfehler ist.

Faber-Polynome

Die Grundlage des polynomiellen Vorkonditionierers ist, dass bei linearen Gleichungssystemen $Ax = b$ mit Anfangslösung x_0 und Anfangsresiduum

$$r_0 = b - Ax_0$$

das Residuum im n.ten Schritt gegeben ist durch

$$r_n = p_n(A)r_0 = [I - Aq_n(A)]\, r_0.$$

Dabei ist $q_n(z)$ das Iterationspolynom mit dem Grad $n-1$ in Abhängigkeit von der gewählten Krylov-Methode. Der Fehler in der n.ten Iteration ist

$$e_n = A^{-1}b - x_n = p_n(A)e_0.$$

Folglich ist $\|p_n(A)\| < 1$ eine notwendige Konvergenzbedingung. Nach der Diagonalisierung ist dies äquivalent zur Bedingung

$$\|p_n(z)\| < 1, \forall z \in \sigma(A),$$

wobei $\sigma(A)$ das Spektrum von A ist. Das Spektrum einer Matrix ist wie folgt definiert

Definition 2.1. *Die Menge der $k \leq n$ paarweise verschiedenen Eigenwerte*

$$\sigma(A) := \{\lambda_1, \ldots, \lambda_k\} = \{z \in \mathbb{C} : det(A - zI) = 0\} \tag{2.7}$$

bezeichnet man als Spektrum von A.

Um ein minimierendes Polynom zu finden, ist der allgemeine Ansatz eine kompakte Menge $D \in \mathbb{C}$ zu finden, die $\sigma(A)$ enthält. Durch $p_n(0) = 1$ muss man den Nullpunkt aus der Menge D ausschließen. Dieses Minimierungsproblem wird durch die Faber-Polynome gelöst, die vollständig durch die Form von D beschrieben sind.

Konstruktion der äußeren Abbildungs-Funktion

Wenn $D \subset \mathbb{C}$ kompakt, $0 \notin D$ und das Komplement D^c einfach zusammenhängend in $\overline{C} := \mathbb{C} \cup \{\infty\}$ ist, dann garantiert der Riemann'sche Abbildungssatz die Existenz einer konformen Abbildung $f : S^c \to D^c$, der sogenannten Exterior Mapping Function (EMF) von D. S ist der Einheitskreis und S^c das Komplement. Wichtige Eigenschaften von f sind

$$f(\infty) = \infty \tag{2.8}$$

$$f'(\infty) > 0 \tag{2.9}$$

$$f(S^c) \ni 0 \tag{2.10}$$

und, dass $f(S^c)$ einfach zusammenhängend ist.
Der transfinite Durchmesser von D ist definiert als $f'(\infty) =: t$.

Die EMF und ihre Inverse kann durch eine Laurent-Reihe entwickelt werden.

$$f(w) = t\left(w + a_0 + \sum_{k=0}^{\infty} \frac{a_k}{w^k}\right) \tag{2.11}$$

$$f^{-1}(z) = \frac{z}{t} + b_0 + \sum_{k=0}^{\infty} \frac{b_k}{z^k}. \tag{2.12}$$

Dann sind die Faber-Polynome $F_n(z; D)$ für ein gegebenes Gebiet D definiert durch $\left[f^{-1}(z)\right]^n, n \geq 0$. Sei die Jordan'sche Kurve

$$J_R := \{f(w) \ : \ |w| = R > 1\} \tag{2.13}$$

gegeben, die den Rand der offenen Menge J bildet. Das n.te Faber-Polynom für D ist implizit definiert durch die Entwicklung

$$\frac{f'(w)}{f(w) - z} =: \sum_{0}^{\infty} \frac{F_n(z)}{w^{n+1}}, \quad |w| > R, \quad z \in J. \tag{2.14}$$

Die Reihe beginnt mit $F_0(z) = 1$ und für $n \geq 1$ wird diese fortgesetzt mit

$$F_n(z) = \frac{z}{t} F_{n-1}(z) - \sum_{j=0}^{n-2} b_j F_{n-1-j}(z) - n b_{n-1}. \tag{2.15}$$

$F_n(z)$ ist vom Grad n mit dem Hauptterm $(z/t)^n$. Der asymptotische Konvergenzfaktor von D ist gegeben durch

$$R(D) = \frac{1}{|f^{-1}(0)|}, \tag{2.16}$$

falls $0 \notin D$ und D^c einfach zusammenhängend ist. Dies garantiert auch die Konvergenz, zum Beispiel $0 < R(D) < 1$. Falls D eine Ellipse ist, dann decken sich die Faber-Polynome mit den Chebychev-Polynomen und können durch eine Drei-Term-Rekursion berechnet werden. Die Idee ist, $F_n(z; D)$ so zu konstruieren unter Berücksichtigung der Tatsache, dass D Abbild einer Ellipse E ist. Die entsprechende EMF wird aus der EMF einer Ellipse konstruiert durch Verknüpfung mit einer analytisch bekannten konformen Abbildung

$g : E \to D$. Somit reduziert sich das Problem auf die Bestimmung der Funktion g, die den Rand der Ellipse auf den gesuchten Rand der Einschlussmenge abbildet.

Drei-Term-Rekursion

Die allgemeine Rekursion aus Gleichung (2.15) verlangt die Abspeicherung aller vorher berechneten Polynome. Dies führt allerdings zu einem unwirtschaftlichen Vorkonditionierer. Für eine kurze Rekursion, wie bei Ellipsen, muss die EMF eine rationale Funktion der Form

$$f(w) \;=\; \frac{P(w)}{Q(w)} \;:=\; \frac{w^2 + \mu_1 w + \mu_0}{\nu_1 w + \nu_0} \tag{2.17}$$

sein. Ihre Inverse und ihre Ableitung sind definiert durch:

$$f^{-1}(z) \;=\; -\frac{\mu_1 - z\nu_1}{2} \pm \sqrt{\left(\frac{\mu_1 - z\nu_1}{2}\right)^2 - \mu_0 + z\nu_0}\,, \tag{2.18}$$

$$f'(w) \;=\; \frac{2 + \frac{\mu_1}{w}}{\nu_1 + \frac{\nu_0}{w}} - \nu_1 \frac{w^2 + \mu_1 w + \mu_0}{\nu_1^2 w^2 + 2\nu_1\nu_0 w + \nu_0^2}\,. \tag{2.19}$$

Asymptotischer Konvergenzfaktor

Dies erlaubt die Berechnung des asymptotischen Konvergenzfaktors $R(D)$ und des transfiniten Durchmessers $t(D)$ der Einschlussmenge D mittels

$$R(D) \;=\; \frac{1}{|f^{-1}(0)|} \;=\; \frac{1}{\left|-\frac{\mu_1}{2} - \frac{1}{2}\sqrt{\mu_1^2 - 4\mu_0}\right|}\,, \tag{2.20}$$

$$t(D) \;=\; f'(\infty) \;=\; \frac{1}{\nu_1}\,. \tag{2.21}$$

Aufgrund der Definition des transfiniten Durchmessers muss der Koeffizient ν_1 positiv und reell sein.

Das Fehlerpolynom

Das Fehlerpolynom wird durch ein normalisiertes Faber-Polynom $\tilde{F}_n(z)$ approximiert

$$r_n = \tilde{F}_n(A)r_0 = \frac{1}{\varrho_n}F_n(A)r_0\,, \tag{2.22}$$

bei dem die Normalisierungskonstante durch

$$\varrho_n := F_n(0) \tag{2.23}$$

gegeben ist. Diese Berechnung wird vereinfacht durch die Benutzung der verschobenen Faber-Polynome $\hat{F}_n(z)$, die in Relation zu den original Faber-Polynomen durch

$$\hat{F}_n(z) - F_n(z) = \left(\frac{-\nu_0}{\nu_1}\right)^n =: S^n\,, \quad \nu_0 \neq 0\,, \tag{2.24}$$

$$\hat{F}_n(z) - F_n(z) = 1 =: S^n\,, \quad \nu_0 = 0 \tag{2.25}$$

stehen. Das verschobene Faber-Polynom kann durch die Nullstellen von $w_1(z)$ und $w_2(z)$ des Hilfspolynoms $P(w) - zQ(w)$ ausgedrückt werden mittels

$$\hat{F}_n(z) = w_1(z)^n + w_2(z)^n, \quad n \geq 1. \tag{2.26}$$

Wie in [11, Thm. 3.1] für Gebiete, die Moebius-äquivalent zu einer Ellipse sind, gezeigt wird, können Faber-Polynome durch eine Drei-Term-Rekursion über die Funktionen $w_1(z)$ und $w_2(z)$ berechnet werden. Mittels der Funktionen

$$\begin{aligned} 2W(z) &:= w_1(z) + w_2(z) = \nu_1 z - \mu_1\,, \\ V(z) &:= w_1(z)w_2(z) = \mu_0 - \nu_0 z\,, \end{aligned}$$

ist die Rekursion dann gegeben durch

$$\begin{aligned}\hat{F}_0(z) &= 2\\ \hat{F}_1(z) &= 2W(z)\\ \hat{F}_n(z) &= 2W(z)\hat{F}_{n-1}(z) - V(z)\hat{F}_{n-2}(z)\,, \quad n \geq 2\,. \end{aligned} \tag{2.27}$$

Die Zusammenfassung der Terme entsprechend der Abhängigkeiten der Koeffizienten von z und die Definition des Hilfspolynoms

$$v_n(z) := \nu_1\hat{F}_{n-1}(z) + \nu_0\hat{F}_{n-2}(z) \tag{2.28}$$

führt zur endgültigen Form des n.ten Polynoms

$$\hat{F}_n(z) = zv_n(z) - \mu_1\hat{F}_{n-1}(z) - \mu_0\hat{F}_{n-2}(z)\,. \tag{2.29}$$

Von dieser Rekursion kann man sofort auf eine Rekursion des Normalisierungsfaktors schließen. Durch Auswertung von der Gleichung (2.24) bei $E = 0$ folgt mit den Startwerten

$$\begin{aligned}\varrho_0 &= 2\\ \varrho_1 &= -\mu_1 - \left(\frac{-\nu_0}{\nu_1}\right)\end{aligned}$$

schließlich

$$\hat{F}_n(0) = -\mu_1\hat{F}_{n-1}(0) - \mu_0\hat{F}_{n-2}(0)\,, \quad n \geq 2\,, \tag{2.30}$$

$$\varrho_n = \hat{F}_n(0) - \left(\frac{-\nu_0}{\nu_1}\right)^n\,, \quad n \geq 2\,. \tag{2.31}$$

Iterationspolynome

Die nichtverschobenen Faber-Polynome werden angewendet, um das zu minimierende Fehlerpolynom zu berechnen. Zur Bestimmung der Lösung x_n braucht man das Iterationspolynom

$$q_n(z) = \frac{1 - p_n(z)}{z}. \tag{2.32}$$

Man erhält

$$\begin{aligned} \varrho_n z q_n(z) &= \varrho_n - F_n(z) \\ &= \hat{F}_n(0) - S^n - (\hat{F}_n(z) - S^n) \\ &= \hat{F}_n(0) - \hat{F}_n(z) \end{aligned} \tag{2.33}$$

und dass $\hat{F}_n(0)$ der konstante Term in der Monomentwicklung von $\hat{F}_n(z)$ ist. Die Iterationspolynome müssen ebenso rekursiv berechnet werden mittels

$$\begin{aligned} \varrho_n q_n(z) &= \frac{\hat{F}_n(0) - \hat{F}_n(z)}{z} \\ &= -\frac{z v_n(z)}{z} - \mu_1 \frac{\hat{F}_{n-1}(z) - \hat{F}_{n-1}(0)}{z} \\ &\quad -\mu_0 \frac{\hat{F}_{n-2}(z) - \hat{F}_{n-2}(0)}{z}. \end{aligned} \tag{2.34}$$

Definiert man $\hat{G}_n(z) := \varrho_n q_n(z)$, erhält man

$$\hat{G}_1(z) = -\nu_1 \tag{2.35}$$

$$\hat{G}_n(z) = v_n(z) - \mu_1 \hat{G}_{n-1}(z) - \mu_0 \hat{G}_{n-2}(z). \tag{2.36}$$

Somit ist die Lösung in der n.ten Iteration gegeben durch

$$x_n = x_0 + q_n(z) r_0. \tag{2.37}$$

2.3. Polynomielle Vorkonditionierung mit ABF-Polynomen

Die Idee der Vorkonditionierung mit ABF-Polynomen ist

$$M^{-1} = s_k(A)$$

zu nutzen, wobei $s_k(A)$ das k.te Iterationspolynom der ABF-Methode ist. Bevor diese konstruiert werden können, braucht man eine ungefähre Schätzung von $\sigma(A)$, um die Einschlussmenge $s\Omega_\epsilon \in B(\lambda_m, \Phi)$ richtig zu bestimmen. Nachdem die Parameter berechnet wurden, können die Polynome durch die gegebene Drei-Term-Rekursion konstruiert werden. Der nachfolgende Algorithmus 1 veranschaulicht die Benutzung des ABF-Methode, die in einen GMRES Löser integriert wurde.

Algorithmus 1 Polynomiell Vorkonditionierter GMRES [11, S. 72]

1: Start oder Neustart:
2: Berechnung des aktuellen Residuums: $r = b - Ax$
3: Adaptiver GMRES Schritt:
4: Berechne m_1 Schritte von GMRES um $Ad = r$ zu lösen
5: Aktualisiere x durch $x = x + d$
6: Berechung der Eigenwerte, die durch die Hessenberg-Matrix abgeschätzt werden
7: Parameter für den ABF polynomiellen Vorkonditionierer :
8: Berechne $\sigma(H)$
9: Berechne eine Einschlussmenge $s\Omega_\epsilon \in B(\lambda_m, \phi)$ für $\sigma(H)$
10: Berechne $s_k(z)$, das k.te Iterationspolynom der ABF Methode
11: Polynomielle Iteration:
12: Berechnung des aktuellen Residuums $r = b - Ax$
13: Berechne m_2 Schritte von GMRES angewandt auf $s_k(A)Ad = s_k(A)r$
14: Aktualisiere x durch $x = x + d$
15: Test auf Konvergenz
16: Falls die Lösung konvergiert, dann Stopp, sonst gehe zu Zeile 1

Die Hessenberg-Matrix H wird automatisch generiert während eines adaptiven GMRES-Schritts. Der Grad des Polynoms, welches der Vorkonditionierung dient, ist typischerweise relativ klein im Vergleich zur Matrix A.

2.4. Parallele Programmierung mit CUDA

Beim parallelen Programmieren mittels CUDA [15] besteht das Programm aus einem sequentiellen Programmteil, auch *Host* genannt, das parallele Programmfragmente, bekannt als *Kernel*, auf der Grafikkarte ausführt. Das Gerät, auf dem die parallelen Programmeinheiten ausgeführt werden, nennt man umgangssprachlich auch *Device*. Ein *Kernel* ist eine *SPMD* (*Single Program Multiple Data*) Einheit, bei der möglichst viele *Threads* parallel ausgeführt werden. Jeder *Thread* durchläuft dabei das gleiche sequentielle Programm. Der Programmierer organisiert die *Threads* in einem Gitter von *Thread Blöcken*. Die *Threads* aus einem *Thread Block* können untereinander kooperieren und jeder Block kann auf den *Shared Memory*, der privat ist in jedem Block, zugreifen, siehe Abbildung 2.1.

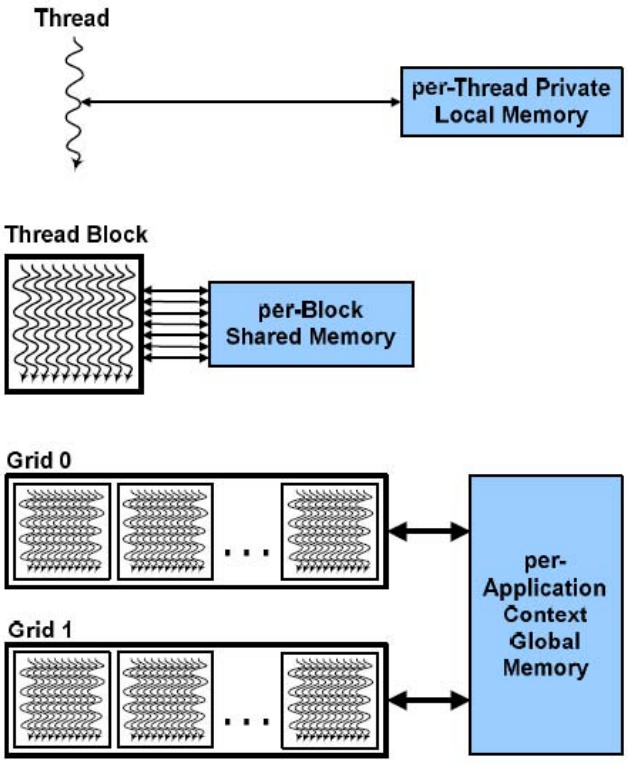


Abbildung 2.1.: CUDA Hierarchie von Threads, Blöcken und Gittern, mit zugehörigem pro-Thread privat, pro-Block shared, und pro-Programm globalem Speicher [13]

Eine heutige Grafikkarte von NVIDIA besteht aus einer Reihe von Multiprozessoren, von denen bis zu 1024 unabhängige Threads unterstützt werden. Jeder Multiprozessor besteht aus 32 Kernen, 32768 32-bit Registern und 64KB Shared Memory, siehe Abbildung 2.2. Eine Grafikkarte wie die *NVIDIA Tesla C2070* besitzt 14 Multiprozessoren, somit könnten ungefähr 500.000 Threads verwaltet werden. Um diese immense Menge von Threads effizient zu verwalten, besitzt die GPU eine SIMT (*Single Instruction Multiple Threads*) Architektur, in der die Threads eines Blocks in Gruppen von je 32, auch *Warp* genannt, ausgeführt werden. Ein Warp führt eine einzelne Operation zu einem Zeitpunkt in allen Threads aus. Den Threads eines Warps ist es allerdings erlaubt, ihrem eigenen Verzweigungspfad zu folgen, da alle Unterschiede im Verzweigungspfad automatisch von der Hardware gehandhabt werden. Es ist jedoch effizienter, wenn alle Threads aus einem Warp dem selben Pfad folgen.

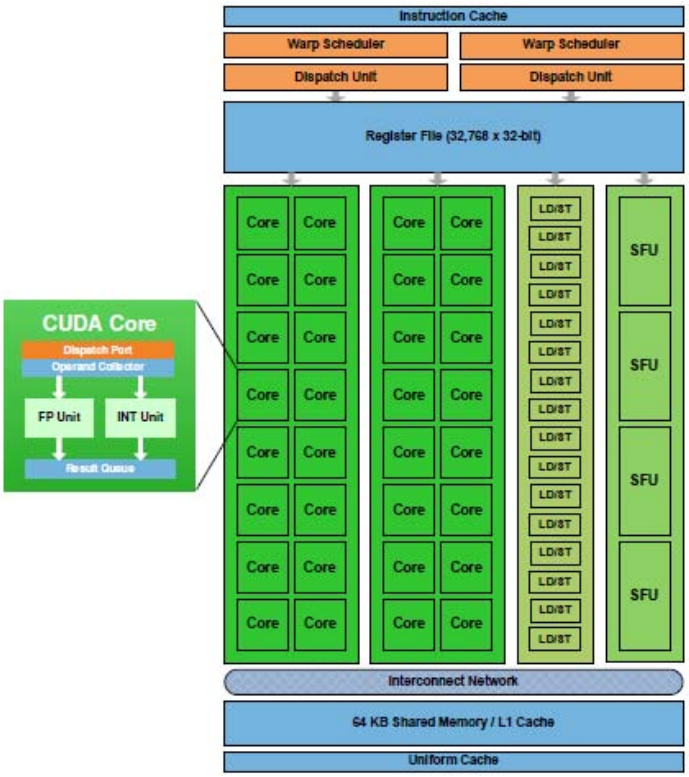


Abbildung 2.2.: Fermi Multiprozessor [13]

3. Laufzeit- und Effizienzanalyse der CUDA-Implementierung

Durch den höheren Aufwand bei der Parallelisierung mittels CUDA wird in diesem Abschnitt geprüft, ob die GPU- schneller als die CPU-Implementierung ist. Als objektives Maß wird die theoretisch mögliche Speichertransferrate und die maximale Rechenleistung herangezogen. Um die tatsächliche Speichertransferrate zu berechnen, muss man die Anzahl der Elemente mit der Anzahl der Speicherzugriffe und dem Datentyp der Elemente multiplizieren und durch die Zeit in Sekunden teilen. Das Ergebnis wird anschließend durch 1024^3 geteilt, was zu folgender Formel führt:

$$\frac{\#Elemente \cdot \#Speicherzugriffe \cdot Datentyp}{\text{Zeit in Sek.} \cdot 1024^3} GB/s. \tag{3.1}$$

Für die tatsächliche Berechnung der $GFlops/s$ wird die oben aufgeführte Formel leicht abgewandelt. Statt der Multiplikation der Speicherzugriffe mit Größe des Datentyps und der Anzahl der Elemente werden nun die Anzahl der Elemente mit der Anzahl der Rechenoperationen multipliziert:

$$\frac{\#Elemente \cdot \#Rechenoperationen}{\text{Zeit in Sek.} \cdot 1024^3} GFlops/s. \tag{3.2}$$

Der Zugriff auf den globalen Speicher sollte minimiert werden, da circa 400-600 Taktzyklen gebraucht werden, was ungefähr $0.25\mu sek.$ bis $0.4\mu sek.$ entspricht [15].
Rechenoperationen, wie zum Beispiel die Addition und Multiplikation, brauchen hingegen in der Regel einen Taktzyklus und werden von den 32 CUDA-Kernen pro Multiprozessor übernommen. Spezialoperationen, wie das Ziehen der Wurzel, die Berechnung eines Logarithmus, benötigen wiederum circa 8 Taktzyklen, weil diese Berechnungen in einer

Special Funtion Unit getätigt werden, die es nur einmal pro Multiprozessor gibt. Für die Bestimmung der Zeit, die für die Ausführung des Kernels benötigt wird, wurde sowohl das cudaEvent, was ein Timer-Objekt der CUDA-Bibliothek ist, als auch der NVIDIA Visual Profiler genutzt. Die Berechnungen wurden auf folgendem System durchgeführt:

CPU	Intel®Xeon®X5650 @ 2.67GHz, 96GB RAM
Speicherbandbreite	25,6 GB/s
GPU	2x NVIDIA Tesla C2070 @ 1.5GHz, 6GB VRAM
GPU-Interface	PCI-Express 2.0 16x
Speicherbandbreite (ECC off)	144 GB/s
Speicherbandbreite (ECC on)	120 GB/s
max. Geschwindigkeit, doppelte Genauigkeit	515GFlops/s
max. Geschwindigkeit, einfache Genauigkeit	1030GFlops/s
CUDA Toolkit	4.0

Tabelle 3.1.: Hardware System

Im Folgenden werden die wichtigsten Kernel beschrieben und ihre Performance analysiert.

3.1. Implementierung elementarer Operationen

Zuweisung eines Vektors

```
1 template<typename T>
2 __global__ void __asgn(T *values, const T value, const int N )
3 {
4     int threadId = blockIdx.x * blockDim.x + threadIdx.x ;
5     if(threadId < N)
6         values[threadId] = value;
7 }
```

Listing 3.1: Zuweisung, $\vec{a} = \alpha$

Die Zuweisung eines Wertes zu einem Vektor ist eine triviale Operation, vergleiche Listing 3.1. Es werden lediglich #*Threads* Rechenoperation für die Bestimmung des Threadindizes, rot markiert, allerdings N Zugriffe auf den globalen Speicher, blau markiert, gebraucht, siehe Listing 3.1. Statt zwei Rechenoperationen handelt es sich um eine *Fused Multiply ADd* Operation bei der eine Multiplikation mit einer Addition gekoppelt wird und so weniger

Zeit verbraucht, als wenn diese getrennt durchgeführt werden [13, vgl. Seite 14]. Bei der Variablen N handelt sich um die Anzahl der Elemente multipliziert mit der Dimension des Vektors, in diesem Fall $80621 \cdot 4 = 322484$ Einträge. Somit ergeben sich mittels der oben genannten Formeln (3.1) (3.2) ein Speichertransfer von circa $117.52 GB/s$ und $13.1 GFlops/s$ beim Rechendurchsatz.

Addition eines Vektors

```
1  template <typename T>
2  __global__ void __vec_add(T* A, const T* B, const T* C, const int N)
3  {
4      int threadId = threadIdx.x + blockDim.x * blockIdx.x;
5      int gridSize = gridDim.x * blockDim.x;
6      for(int i = threadId; i<N; i += gridSize)
7          A[i] = B[i] + C[i];
8  }
```

Listing 3.2: Addition, $\vec{a} = \vec{b} + \vec{c}$

Bei der Addition, siehe Listing 3.2, werden in jedem Thread unter anderem einmal der Threadindex und zusätzlich zwei Rechenoperationen in der Schleife, die N-Durchläufe iteriert, benötigt. So entstehen insgesamt $\#Threads + 2 \cdot N$ Rechenoperationen. Der Speichertransfer liegt bei $3 \cdot N$, da aus zwei Vektoren gelesen und in einen geschrieben wird. Die Anzahl der Threads liegt bei dieser Berechnung bei $Blocksize \cdot N_{blocks} = 256 \cdot 1260 = 322816$. Als Speicher- bzw. Rechendurchsatz ergeben sich $97.51 GB/s$ und $15.61 GFlops/s$.

Skalierte Addition eines Vektors

```
1  template <typename T>
2  __global__
3  void __vec_add_scaled(T* A,
4                        const T* B, const T* C,
5                        const T a, const int N)
6  {
7      int threadId = threadIdx.x + blockDim.x * blockIdx.x;
8      int gridSize = gridDim.x * blockDim.x;
9
10     for(int i = threadId; i<N; i += gridSize)
11         A[i] = B[i] + a * C[i];
12
13 }
```

Listing 3.3: skalierte Addition I, $\vec{a} = \vec{b} + \alpha \cdot \vec{c}$

Die erste Variante der skalierten Addition, siehe Listing 3.3, unterscheidet sich bei den Rechen- und Speicheroperationen nicht, da die zusätzliche Rechenoperation innerhalb der Schleife als FMAD beschrieben werden kann. Somit ergeben sich beim Rechen- und Speicherdurchsatz $15.21 GFlops/s$ und $95.05 GB/s$, die ähnlich sind zur obigen Operation.
In der zweiten Version der skalierten Addition, vergleiche Listing 3.4, gibt es nur kleine Unterschiede, die bei der Anzahl der Rechenoperationen zu bemerken sind.
Dieser Kernel schließt $\#Threads + 3 \cdot N$ Rechenoperationen und $3 \cdot N$ Speicheroperationen ein. Allerdings ergeben sich durch die zusätzlichen Rechenoperationen nur geringe Unterschiede bei der Anzahl der Floating point Operations pro Sekunde, die bei $18.77 GFlops/s$ liegt. Die Anzahl der Speicherzugriffe gegenüber der obigen Operation ändert sich nicht und beträgt $93.85 GB/s$.

```
1  template <typename T>
2  __global__
3  void __vec_add_scaled(T* A,
4                        const T* B, const T* C,
5                        const T alpha, const T beta, const int N)
6  {
7      int threadId = threadIdx.x + blockDim.x * blockIdx.x;
8      int gridSize = gridDim.x * blockDim.x;
9
10     for(int i = threadId; i<N; i += gridSize)
11         A[i] = alpha * B[i] + beta * C[i];
12
13 }
```

Listing 3.4: skalierte Addition II, $\vec{a} = \alpha \cdot \vec{b} + \beta \cdot \vec{c}$

```
1  template <typename T>
2  __global__
3  void __vec_add_scaled(T* A,
4                        const T* B, const T* C, const T* D,
5                        const T alpha, const T beta, const int N)
6  {
7      int threadId = threadIdx.x + blockDim.x * blockIdx.x;
8      int gridSize = gridDim.x * blockDim.x;
9
10     for(int i = threadId; i<N; i += gridSize)
11         A[i] = B[i] + alpha * C[i] + beta * D[i];
12 }
```

Listing 3.5: skalierte Addition III, $\vec{a} = \vec{b} + \alpha \cdot \vec{c} + \beta \cdot \vec{d}$

Die dritte Version der skalierten Addition, siehe Listing 3.5, weist wiederum nur geringe Unterschiede zur ersten Version auf. Gegenüber dieser ist die Anzahl der Rechenoperationen innerhalb der Schleife doppelt so groß und liegt bei $4 \cdot N$. Da jeder Thread am Anfang

seiner Ausführung den Threadindex berechnen muss, entstehen somit, abhängig von der Anzahl der Threads, zusätzliche Rechenoperationen. Die Anzahl der Rechenoperationen liegt aus den genannten Gründen daher bei $\#Threads + 4 \cdot N$. Die Anzahl der Speicherzugriffe kann durch $4 \cdot N$ abgeschätzt werden, da aus drei Vektoren gelesen und in einen Vektor das Ergebnis geschrieben wird. Der Speichertransfer und der Rechendurchsatz für diesen Kernel liegen somit bei $93.50 GB/s$ bzw. $17.10 GFlops/s$. Genutzt werden alle drei Varianten der skalierten Addition, da es sinnvoll ist, möglichst viele Programmteile zu parallelisieren.

Skalierte Zuweisung eines Vektors

```
1 template <typename T>
2 __global__ void __vec_equ(T* A,const  T* B, const T a, const int N)
3 {
4     int threadId = threadIdx.x + blockDim.x * blockIdx.x;
5     int gridSize = gridDim.x * blockDim.x;
6
7     for(int i = threadId; i<N; i += gridSize)
8         A[i] = a * B[i];
9 }
```

Listing 3.6: skalierte Zuweisung, $\vec{a} = \alpha \cdot \vec{b}$

Die skalierte Zuweisung eines Vektors, siehe Listing 3.6, ähnelt dem Kernel, der die Zuweisung behandelt, vergleiche Listing 3.1. Statt der reinen Zuweisung wird nun jeder Wert aus dem Vektor $\vec{b}$ gelesen und mit einem Faktor α skaliert und dem Ergebnisvektor $\vec{a}$ zugewiesen. Daher entstehen $\#Threads + 2 \cdot N$ Rechenoperationen, womit ein Durchsatz von $24.03 GFlops/s$ erreicht wird. Die Anzahl der Speicheroperationen beläuft sich auf $2 \cdot N$, daher ist der Speichertransfer äquivalent zum Kernel, der die Zuweisung abarbeitet, und beträgt $102.10 GB/s$.

Berechnung des Betrags eines Vektors

Die Berechnung des Betrags, siehe Listing 3.7, in jedem Eintrag des Vektors benötigt $\#Threads$, da von jedem Thread der Threadindex berechnet wird. Die Speicheroperationen beschränken sich auf $2 \cdot N$, weil zwei Vektoren genutzt werden. Somit resultieren $11.55 GFlops/s$ als Rechendurchsatz und $98.18 GB/s$ als Speichertransfer.

```
template <typename T>
__global__ void __vec_add(T* A, const T* B, const int n_entries)
{
    int threadId = blockIdx.x * blockDim.x + threadIdx.x ;

    if(threadId < n_entries)
    {
        if(B[threadId] < 0 )
            A[threadId] = B[threadId] * (-1);
        else
            A[threadId] = B[threadId];

    }
}
```

Listing 3.7: Absolutwert, $\vec{a} = |\vec{b}|$

3.2. Implementierung von Matrix-Vektor Produkten

Im weiteren Verlauf diesen Kapitels werden die Begriffe *skalares-* und *mehrdimensionales* Matrix-Vektor Produkt verwendet. Jedes Matrixelement beinhaltet im mehrdimensionalen Fall eine dicht besetzte Matrix mit $n \times n$ Einträgen. Ein mehrdimensionaler Vektor hat somit, analog zur Matrix, pro Vektoreintrag einen Vektor der Länge n. Beim mehrdimensionalen Matrix-Vektor Produkt wird eine mehrdimensionale Matrix mit einem mehrdimensionalen Vektor multipliziert. Zur Verdeutlichung dient Abbildung 3.1.

$$y = \begin{pmatrix} \begin{pmatrix} 1 & 1 \\ -1.23 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{pmatrix} 2 & -1 \\ -3 & 1 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 1 \\ -1.23 & 1 \end{pmatrix} & \begin{pmatrix} 3 & 3 \\ 3 & 3 \end{pmatrix} \end{pmatrix} \cdot \begin{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ \begin{pmatrix} 1 \\ 1 \end{pmatrix} \end{pmatrix}$$

Abbildung 3.1.: mehrdimensionales Matrix-Vektor Produkt

Eine skalare Matrix beinhaltet pro Matrixelement nur einen Eintrag, ebenso ein Vektor. Diese Art des Matrix-Vektor Produkts gleicht dem klassischen Matrix-Vektor Produkt, siehe Abbildung 3.2.

Jede Matrixzeile wird mit einem Eintrag aus dem Vektor multipliziert, aufsummiert und im Ergebnisvektor $\vec{y}$ gespeichert.

$$y = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 2 \\ 0 & 1 & 3 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Abbildung 3.2.: skalares Matrix-Vektor Produkt

Skalares Matrix-Vektor Produkt

```
1  __global__ void __spmv_dim_1(
2          cublanc_Real* const* const m_values,
3          size_t* const* const m_colIdxs,
4          const size_t* const m_rowlens,
5          const cublanc_Real* const x_values,
6          cublanc_Real* y_values,
7          const size_t eltnum)
8  {
9      // Berechnung der aktuellen Zeile.
10     size_t row = blockDim.x * blockIdx.x + threadIdx.x;
11
12     if(row >= eltnum) return;
13
14     cublanc_Real* m_values_row = m_values[row];
15     const size_t* colidxs_row = m_colIdxs[row];
16     const int rowlen = m_rowlens[row];
17     cublanc_Real buffer = 0.0;
18
19     // inneres Produkt
20     for(int j = 0; j < rowlen ; j++)
21         buffer +=  m_values_row[ j ] * x_values[ colidxs_row[j] ] ;
22
23     y_values[row] = buffer;
24 }
```

Listing 3.8: skalares Matrix-Vektor Produkt, $\vec{y} = M \cdot \vec{x}$

Das skalare Matrix-Vektor Produkt, siehe Listing 3.8, wurde so programmiert, dass jeder Thread eine Matrixzeile aus der Matrix M bearbeitet. Jeder Thread berechnet in der zehnten Zeile des obigen Quellcodes seinen Threadindex und testet anschließend ab, ob die ThreadID innerhalb der Anzahl der Matrixzeilen liegt. Im weiteren Verlauf benötigt jeder Thread die Werte aus einer Matrixzeile, deren Länge und die dazugehörigen Spaltenindizes. Nun wird das Produkt der entsprechenden Werte aus dem Vektor $\vec{x}$ mit den Werten aus der Matrix M in eine temporäre Variable geschrieben, fortwährend aufsummiert und anschließend in den Ergebnisvektor $\vec{y}$ gespeichert. Durch diese Prozedur ergeben sich

$eltnum + 2 \cdot NNZ$ Rechen- und $4 \cdot eltnum + 3 \cdot NNZ$ Speicheroperationen, wobei $eltnum$ der Anzahl der Zeilen der Matrix bzw. des Vektors und NNZ der Anzahl der Nichtnull-Einträge der Matrix entspricht. In der Berechnung des Rechendurchsatzes und Speichertransfers sind $eltnum = 80621$ und $NNZ = 1142209$. Daraus resultieren $0.65 GFlops/s$ und $3.83 GB/s$.

Mehrdimensionale Matrix-Vektor Produkte

```
__global__ void __spmv_dim_2(
        cublanc_Real* const* const m_values,
        size_t* const* const m_colIdxs,
        const size_t* const m_rowlens,
        const cublanc_Real* const x_values,
        cublanc_Real* y_values,
        const size_t eltnum,
        const size_t eltdim)
{
    // Berechnung der aktuellen Zeile.
    size_t row = (blockDim.x * blockIdx.x + threadIdx.x) / eltdim;
    //Berechnung der lokalen Position
    size_t eltrow = (threadIdx.x + eltdim -1 ) % eltdim;

    if(row >= eltnum ) return;

    size_t j = 0;
    size_t k = 0;
    size_t l = 0;
    size_t r = 0;
    size_t s = row * eltdim;
    size_t p = 0;
    size_t q = 0;
    cublanc_Real buffer;

    cublanc_Real* m_values_row = m_values[row];
    size_t* colidxs_row = m_colIdxs[row];
    size_t rowlen = m_rowlens[row];

    // Schleife ueber die Elemente einer Zeile
    for( j = 0, r = 0; j < rowlen; j++) {
        r = j * eltdim * eltdim + eltdim * eltrow;
        p = colidxs_row[j] * eltdim;
        q = p + eltdim;
        k = s + eltrow;
        buffer = 0.0;

        // inneres Produkt
        for(l = p; l < q; l++, r++)
            buffer += m_values_row[r] * x_values[l];

        y_values[k] += buffer;
    }
}
```

Listing 3.9: Mehrdimensionales Matrix-Vektor Produkt, $\vec{y} = M \cdot \vec{x}$

Dieses mehrdimensionale Matrix-Vektor Produkt, vergleiche Listing 3.9, unterscheidet sich in der Vorgehensweise zum Vorherigen Matrix-Vektor Produkt nur leicht. Jeder Thread berechnet wieder seine eigene ThreadID und lädt die Werte einer Matrixzeile, die Spaltenindizes und die Länge einer Matrixzeile. Der Unterschied besteht darin, dass jeder Thread $eltdim^2$ Einträge jedes Matrixelements lädt, mit den entsprechenden Werten des Vektors $\vec{x}$ multipliziert, in einer temporäre Variablen aufsummiert und am Ende der Berechnung in den Zielvektor $\vec{y}$ speichert. Auf diese Art und Weise enstehen $5 \cdot eltnum + NNZ \cdot (6 + 3 \cdot eltdim^2)$ Rechen- und $eltnum \cdot (3 + eltdim) + NNZ \cdot (1 + eltdim^2)$ Speicheroperationen, wobei $eltnum = 80621$, $eltdim = 4$ und $NNZ = 1142209$ ist. Zudem ist die Anzahl der Spezialoperationen $2 \cdot eltnum$, pink gekennzeichnet. Durch dieses Verfahren wird eine größere Speichertransferrate von $8.91 GB/s$ und ein größerer Rechendurchsatz von $1.94 GFlops/s$ erreicht. Das nun kommende Matrix-Vektor Produkt, vergleiche Listing 3.10, wurde weiter optimiert. Gegenüber dem vorherigen mehrdimensionalen Matrix-Vektor Produkt wird jetzt der SharedMemory der Grafikkarte genutzt. In diesen kleinen, aber sehr schnellen Speicher werden sowohl der Quell- als auch der Ergebnisvektor geladen. Der SharedMemory bei der genutzten Tesla C2070 beträgt pro Multiprozessor 64KB. Eine weitere Änderung ist, dass pro Matrixzeile ein Halbwarp eingesetzt wird, der aus 16Threads besteht. Die weitere Funktionsweise unterscheidet sich zum Obigen sonst nicht. Allerdings ergeben sich durch diese Abänderungen dennoch große Unterschiede bei der Speichertransferrate, die nun $49.32 GB/s$ beträgt, und beim Rechendurchsatz, der bei $11.64 GFlops/s$ liegt.

```
1  __global__ void __spmv_eltdim_4_warp(
2          cublanc_Real* const* const m_values,
3          size_t* const* const m_colIdxs,
4          const size_t* const m_rowlens,
5          const cublanc_Real* const x_values,
6          cublanc_Real* y_values,
7          const size_t eltnum,
8          const size_t eltdim)
9  {
10
11     // Ein thread-Block kann maximal 16 Elemente bearbeiten
12     // und jeder Halbwarp bearbeitet eine Zeile.
13     __shared__ cublanc_Real y_ps_shm[256 /*BLOCKSIZE_spmv_eltdim_4_warp*/];
14     __shared__ cublanc_Real x_shm[64 /*half-warp size * eltdim*/ ];
15
16     y_ps_shm[threadIdx.x] = 0.;
17     if (threadIdx.x < 64)
18         x_shm[threadIdx.x] = 0.;
19
20     __syncthreads();
21
22     // 16 threads beackern eine Matrixzeile aus 4x4-Matrizen
```

```
23      const size_t n_entries_per_element_row = 4;
24      const size_t n_entries_per_element = 16;
25
26      size_t n_rows_per_block = 16;
27      size_t local_row = threadIdx.x / 16;
28
29      size_t entry_idx = threadIdx.x % 16
30      // der Eintraege eines Matrixelements von 0-15
31
32      size_t x_component = entry_idx % 4;
33      size_t local_entry_row   = entry_idx / 4;
34
35      // Blockzeile
36      size_t row = blockIdx.x * 16 + local_row;
37
38      // Globaler Zeilenindex fuer Eintrag in y
39      size_t entry_row = n_entries_per_element_row * row + local_entry_row;
40
41      // schauen, ob die Zeile wirklich existiert
42      if(row >= eltnum ) return;
43
44      size_t* colidxs_row = m_colIdxs[row];
45      cublanc_Real* A_i   = m_values[row];
46
47      size_t rowlen = m_rowlens[row];
48
49      int y_ps_pos = threadIdx.x;
50      volatile cublanc_Real * y_ps = y_ps_shm +y_ps_pos ;
51
52      // Schleife ueber die Elemente einer Zeile
53      for(int j = 0; j < rowlen; j++)
54      {
55          // Globaler Spaltenindex
56          size_t c = colidxs_row[j];
57
58          // Globaler Index eines Eintrags im Quellvector
59          size_t c_entry = n_entries_per_element_row * c + x_component;
60
61          // Lokale Zeile
62          size_t c_x_shm = local_row * n_entries_per_element_row + x_component;
63
64          // 1. Lade die Quellvektoreintraege
65          if (entry_idx < n_entries_per_element_row)
66              x_shm[c_x_shm] = x_values[c_entry];
67
68          // 2. Lade Elementeintraege
69          cublanc_Real A_ij = A_i[j * n_entries_per_element + entry_idx];
70
71          // 3. BIlde Partialsumme
72          (*y_ps) += A_ij * x_shm[c_x_shm];
73      }
74
75      // Summiere die Partialsummen. Annnahme: eltdim == 4 !!!
76      if (x_component < 2)
77          y_ps_shm[y_ps_pos] += y_ps_shm[y_ps_pos + 2];
78
79      if (x_component == 0)
80          y_ps_shm[y_ps_pos] += y_ps_shm[y_ps_pos + 1];
81
82      // Schreibe Ergebnis weg
83      if (x_component == 0)
```

```
84          y_values[entry_row] = (*y_ps);
85
86
87  }
```

Listing 3.10: Matrix-Vektor Produkt speziell für die Dimension 4, $\vec{y} = M \cdot \vec{x}$

Die Auslagerung der Vektoroperationen auf die Grafikkarte ist generell ein guter Ansatz. Die GPU erreicht eine Auslastung bei der Speichertransferrate von 80 $\sim$ 98%. Die Auslastung betreffend der Rechenleistung liegt allerdings sehr gering bei nur 2 $\sim$ 5%. Das Potential der Grafikkarte ist somit noch nicht vollständig erschöpft. Dennoch sind die Vektoroperationen generell schneller auf der Grafikkarte, als wenn diese vom Prozessor ausgeführt werden würden. Bei den Matrix-Vektor Produkten ist eine Portierung mittels CUDA auch als sinnvoll zu erachten. Sie sind zwar nicht so leistungsstark wie die Vektoroperationen, aber immer noch circa 2.5 $\sim$ 55x schneller als die gleiche Ausführung durch die CPU. Die Benutzung der Matrix-Vektor Produkte sollte jedoch generell minimiert werden, da sie im Verhältnis zu verschiedenen Vektoroperationen viel Zeit in Anspruch nehmen, siehe Tabelle 3.2.

Kernel	Zeit in $\mu sek.$
__asgn	23
__add_scaled_I	79
__add_scaled_II	80
__add_scaled_III	106
__equ	50
__abs	52
__scale	50
__add	77
__spmv_dim_1	3723
__spmv_dim_2	4601
__spmv_dim_4	2950

Tabelle 3.2.: Tabelle mit benötigten Zeiten von jedem Kernel

3.3. Zusammenfassung

Das folgende Diagramm 3.3 fasst die in diesem Kapitel genannten Fakten noch einmal bildlich zusammen. In blau wurde die Speichertransferrate und in rot die Rechenleisung dargestellt. Zudem wurden zwei Linien eingefügt, die die maximale Speichertransferrate der GPU als auch der CPU darstellt.

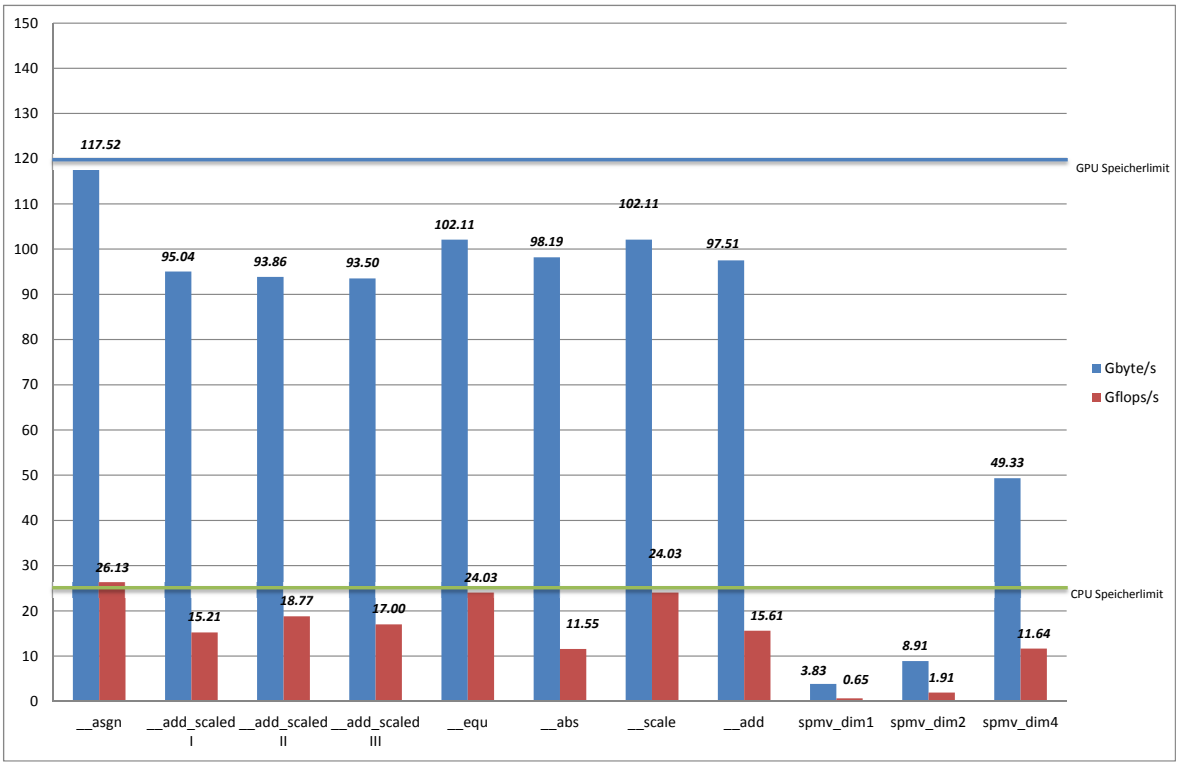


Abbildung 3.3.: Übersicht über die Speichertransferraten und Rechenoperationen der Kernel.

4. Fallstudie zur Simulation von Raumluftströmungen

4.1. Untersuchtes Problem

In diesem Kapitel werden die gesammelten Ergebnisse vorgestellt und detailliert erläutert. Die Testmatrizen wurden zur Simulation von Raumluftströmungen von R. Gritzki vom Institut für Energietechnik der Technischen Universität Dresden zur Verfügung gestellt. Ziel war es, die Auswirkungen auf den Luftaustausch durch dezentrale Klimaanlagen an den Fenstern zu messen. Zur Diskretisierung wurde ein tetraedisches Gitter mit 80621 Knoten gewählt, die zu 644968 Freiheitsgraden insgesamt führt. In Abbildung 4.1 erkennt man ein typisches Vektorfeld einer Strömungssimulation.

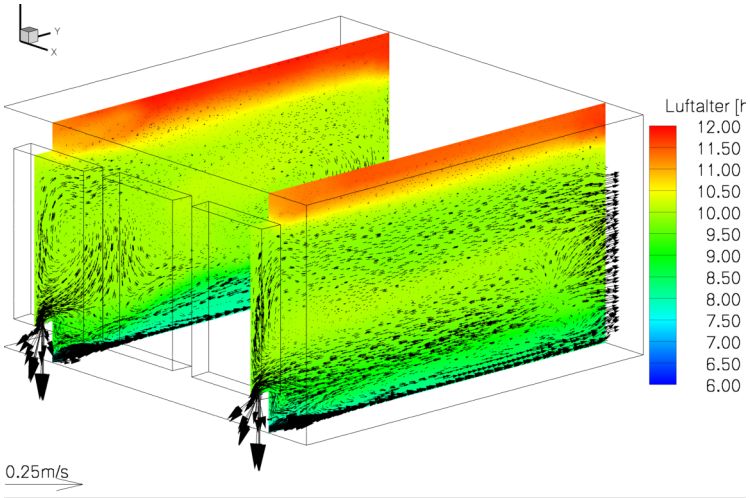


Abbildung 4.1.: Visualisierung des Luftalters.

Wie man in Abbildung 4.1 erkennt, findet an den Fenstern und im Bodenbereich ein großer Luftaustausch statt. Dies wird durch die grünliche Färbung dargestellt. Die rote Färbung im Bereich der Decke visualisiert wiederum einen geringeren Luftaustausch. Nun stellt sich die Frage, welche Strömungen sich mit der Zeit entwickeln.

$$\partial_t u - \nabla \cdot (2\nu_\epsilon S(u)) + (u \cdot \nabla)u + \nabla p = -\beta\theta g, \qquad u = \text{Strömungsfeld} \tag{4.1}$$

$$\nabla \cdot u = 0, \tag{4.2}$$

$$\partial_t \theta + (u \cdot \nabla)\theta - \nabla \cdot (a_\epsilon \nabla\theta) = c_p^{-1}\dot{q}^V, \qquad \theta = \text{Temperatur} \tag{4.3}$$

inkompressible Navier-Stokes-Gleichungen und Fourier-Modell

$$\partial_t k - \nabla \cdot (\nu_k \nabla k) + (u \cdot \nabla)k = P_k + G - \epsilon, \qquad k = \text{Turbulenz} \tag{4.4}$$

$$\partial_t \epsilon - \nabla \cdot (\nu_\epsilon \nabla\epsilon) + (u \cdot \nabla)\epsilon + C_2\frac{\epsilon^2}{k} = C_1\frac{\epsilon}{k}(P_k + G), \qquad \epsilon = \text{Dissipation} \tag{4.5}$$

$$\tag{4.6}$$

k-ϵ-Gleichung

$$\partial_t \tau_p + u \cdot \nabla\tau_p - \nabla \cdot (a_{\tau,\epsilon}\nabla\tau_p) = 1, \qquad \tau = \text{Luftalter} \tag{4.7}$$

Luftalter-Gleichung

Um die Strömung zu berechnen, wird zuerst eine Zeitdiskretisierung mittels des BDF(1) oder BDF(2) (*Backward Differentiation Formulas*) Schemas durchgeführt, um anschließend eine Linearisierung und Entkopplung der Gleichungen vorzunehmen. Der nächste Schritt beinhaltet das Aufstellen von möglichen Randbedingungen. Man erhält als Ergebnis ein System von entkoppelten ADR (*Advektions-Diffusions-Reaktions*)-Gleichungen und das Oseen-Problem in jedem Zeitschritt.

$$-\nabla \cdot (2\nu \mathbb{S}(u)) + (a \cdot \nabla)u + cu + \nabla p = f \qquad \text{in } \Omega \tag{4.8}$$
$$\nabla \cdot u = 0 \qquad \text{in } \Omega \tag{4.9}$$
$$\sigma(u,p)n = \tau_n n \qquad \text{bei } \Gamma_- \cup \Gamma_+ \tag{4.10}$$
$$(\mathbb{I} - n \otimes n)\sigma(u,p)n = \tau_t \qquad u \cdot n = 0 \qquad \text{bei } \Gamma_0 \tag{4.11}$$

Oseen-Gleichungen

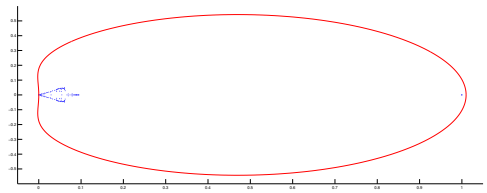
Die linearisierten Gleichungen für das Fourier-Modell und der k-ϵ-Gleichung sind auch ADR-Gleichungen mit variabler Viskosität, die folgende Form besitzen:

$$-\nabla \cdot (\nu \nabla v) + (a \cdot \nabla)u + cu = f \qquad \text{in } \Omega \text{ oder } \Omega \setminus \Omega_\delta \tag{4.12}$$
$$u = g \qquad \text{bei } \tilde{\Gamma}_D \tag{4.13}$$
$$\nu \nabla u \cdot n = h \qquad \text{bei } \tilde{\Gamma}_N \tag{4.14}$$

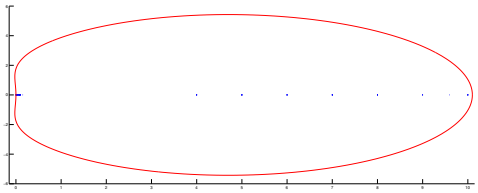
Fourier-Modell und k-ϵ-Gleichung

Im nächsten Schritt führt man eine P1 Finite Elemente Methode bei der Ortsableitung durch. Allerdings muss eine Stabilisierung durch SUPG (*Streamline Upwind Petrov-Galerkin*) oder PSPG (*Pressure Stabilization Petrov-Galerkin*) vorgenommen werden [7], da sonst die inf-sup Bedingung verletzt wird.

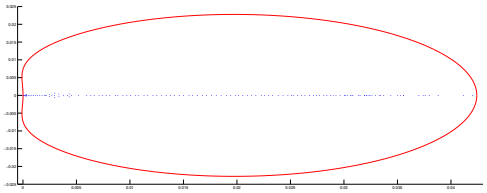
Die entstehenden großen schwach besetzen linearen Gleichungssysteme sind nichtsymmetrisch. Um eine Einschätzung zu erhalten, wie die approximierten Spektren der oben genannten Gleichungen verlaufen, werden jene an dieser Stelle, in blau zu erkennen, dargestellt. Zudem sind die gewählten Einschlussmengen der jeweiligen Gleichung in rot visualisiert. Die Spektren der k-ϵ- und der Luftalter-Gleichung sind rein reell (siehe Diagramme 4.2b und 4.2d), wohin gegen die Oseen- und Temperatur-Gleichung vereinzelt Eigenwerte besitzen die einen imaginären Teil haben (siehe Diagramme 4.2a und 4.2c).



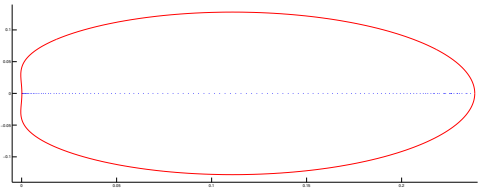
(a) Oseen-Gleichung



(b) k-ϵ-Gleichung



(c) Temperatur-Gleichung



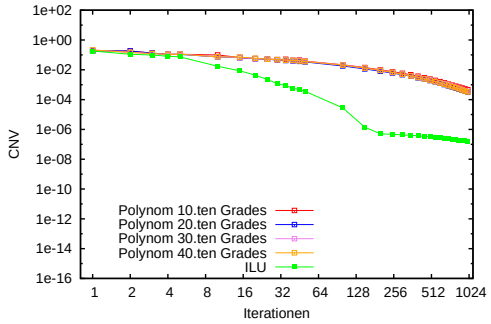
(d) Luftalter-Gleichung

Abbildung 4.2.: Spektren und Einschlussmengen

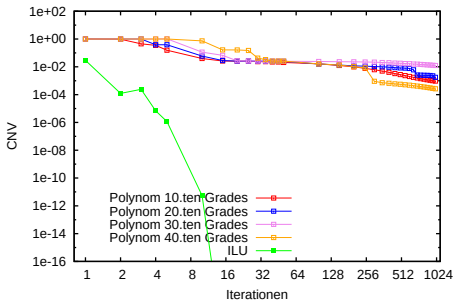
Die Einschlussmengen um die Spektren fallen relativ groß aus. Es wurden mehrere Einschlussmengen durchgetestet, doch nur diese führten zu einer akzeptablen Konvergenz. Falls die Einschlussmenge kleiner gewählt wurde, führte dies zu einer schlechteren Konvergenz oder sogar zur Divergenz.

4.2. Ergebnisse mit polynomialer Vorkonditionierung von GMRES

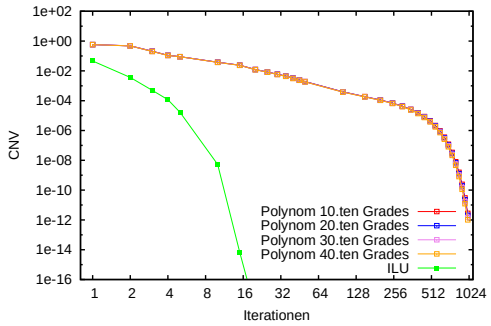
Im ersten Versuch wurde eine GMRES-Implementierung (*Generalized Minimal RESidual*) als Löser eingesetzt. Als Maßstab wurde die Abbruchgenauigkeit in logarithmischer Skalierung gewählt und gegen die Anzahl der Iterationen aufgetragen. Um einen Vergleich ziehen zu können, wurde eine ILU (*incomplete LU*) Zerlegung als typischer Vorkonditionierer einer CPU-Implementierung gewählt und in grün in den folgenden Diagrammen eingezeichnet.



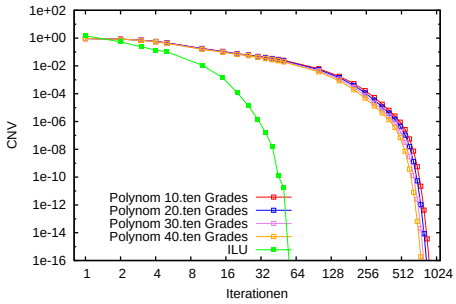
(a) Oseen-Gleichung



(b) k-ϵ-Gleichung



(c) Temperatur-Gleichung

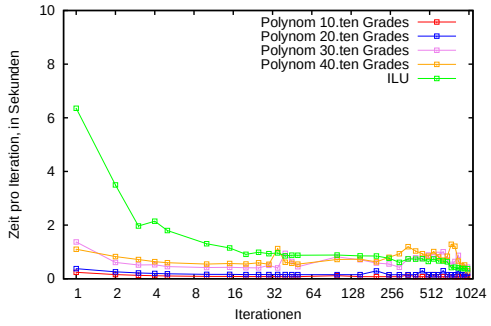


(d) Luftalter-Gleichung

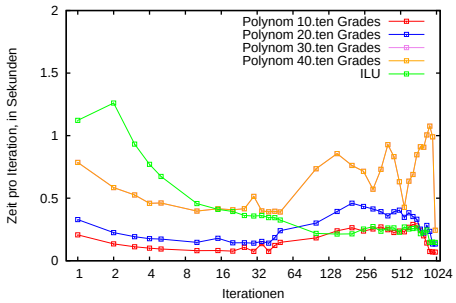
Abbildung 4.3.: Konvergenzdiagramm des GMRES-Lösers mit ILU und Polynomen unterschiedlichen Grades als Vorkonditionierer

In den Diagrammen ist zu sehen, dass beim polynomiellen Vorkonditionierer eine höhere Anzahl von Iterationen benötigt werden, bis man eine festgelegte Abbruchgenauigkeit

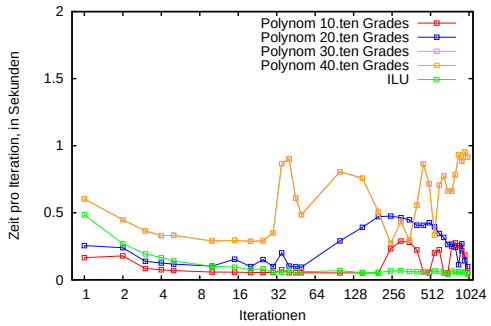
erreicht. Es wird eine maximale Abbruchgenauigkeit von 10^{-4} (siehe Diagramm 4.3a und 4.3b) erlangt. Bei den Gleichungen bezüglich der Temperatur beziehungsweise des Luftalters besteht eine maximale Abbruchgenauigkeit von 10^{-12} und 10^{-16}. Somit schneidet der polynomielle Vorkonditionierer im Vergleich zur unvollständigen LU Zerlegung relativ schlecht ab. Um die gleiche Abbruchgenauigkeit zu erreichen, benötigt man je nach Wahl der Gleichung unter Umständen 64 mal mehr Iterationen (siehe Diagramm 4.3a, 4.3c und 4.3d). Nur im Fall der Luftalter-Gleichung (siehe Diagramm 4.3d) werden 16 mal mehr Iterationen benötigt.
Im Weiteren wurde die Zeit pro Iteration gegen die Anzahl der Iterationen dargestellt.



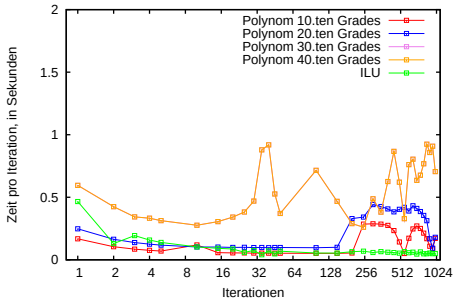
(a) Oseen-Gleichung



(b) k-ϵ-Gleichung



(c) Temperatur-Gleichung



(d) Luftalter-Gleichung

Abbildung 4.4.: Zeit pro Iteration des GMRES-Lösers mit ILU und Polynomen unterschiedlichen Grades als Vorkonditionierer

In diesem Vergleich schneidet der polynomielle Vorkonditionierer teilweise besser ab. Bei der Oseen-Gleichung (siehe Diagramm 4.4a) benötigt der ILU Vorkonditionierer im Bereich von 1 bis 256 Iterationen mehr Zeit. Leider unterliegt der polynomielle Vorkonditionierer zeitlich bei der Temperatur- bzw. Luftalter-Gleichung (siehe Diagramm 4.4c und 4.4d). Wie man im Diagramm 4.4b bei der k-ϵ-Gleichung erkennt, ist im Bereich von 16 bis 64 Iterationen der polynomielle Vorkonditionierer mit verschiedenen Polynomgraden schneller.

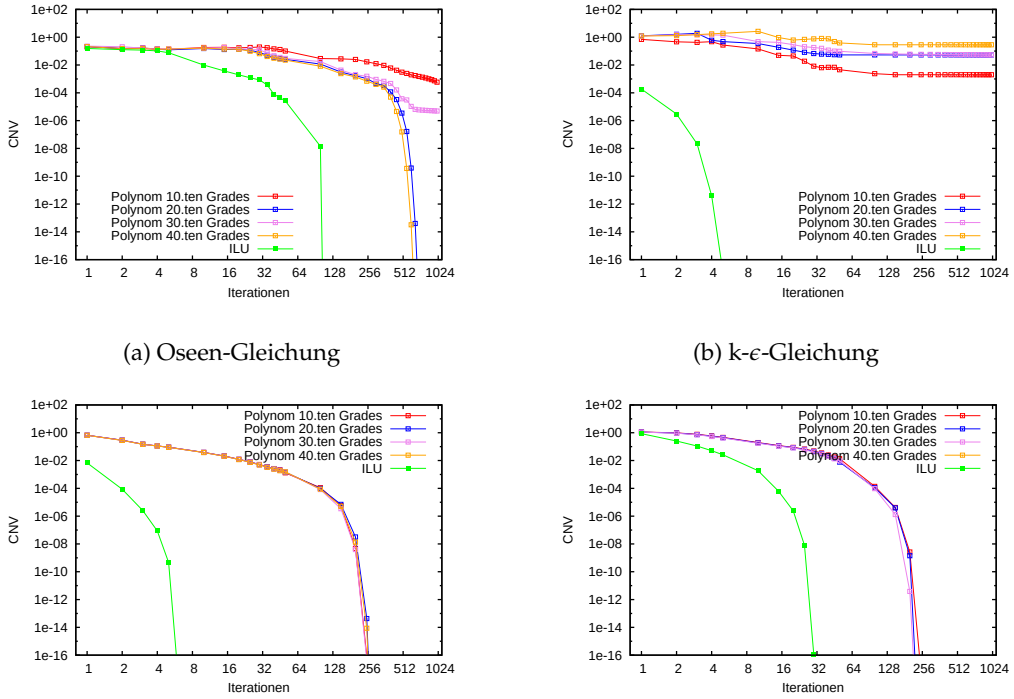
	polynomiell		ILU	
Gleichung	Anzahl der Iterationen	Zeit, in Sek.	Anzahl der Iterationen	Zeit, in Sek.
Oseen	700	$\sim$586	35	$\sim$33
k-ϵ	300	$\sim$208	2	$\sim$2
Temperatur	100	$\sim$112	3	0.5
Luftalter	150	$\sim$157	20	1.6

Tabelle 4.1.: Erreichen der Abbruchgenauigkeit von 10^{-4}

Wenn man nun eine bestimmte Abbruchgenauigkeit festlegt und vergleicht, wie viele Iterationen sowohl der polynomielle Vorkonditionierer als auch der ILU hierfür benötigen, stellt man fest, dass der polynomielle Vorkonditionierer suboptimal ist. Um eine Abbruchgenauigkeit von 10^{-4} zu erreichen, beansprucht der polynomielle- circa 700 Iterationen, was ungefähr 590 Sekunden dauert, wohin gegen der ILU Vorkonditionierer mit 35 Iterationen auskommt, die in 34 Sekunden ablaufen. Dies ist nur ein Beispiel anhand der Oseen-Gleichung. Bei den anderen Gleichungen zeichnet sich dennoch ein gleicher Trend ab, siehe Tabelle 4.1.

4.3. Ergebnisse mit polynomialer Vorkonditionierung von QMRCGSTAB

Als weiterer Löser wurde QMRCGSTAB (*Quasi-Minimal Residual Conjugate Gradient STABilized*) in Kombination mit ILU und polynomieller Vorkonditionierung genutzt. In diesem Fall erkennt man, dass bis auf kleine Ausnahmen (siehe Diagramm 4.5b), der polynomielle Vorkonditionierer schneller das Abbruchkriterium nach einer gewissen Anzahl von Iterationen erreicht als der ILU-Vorkonditionierer (siehe Diagramm 4.5a, 4.5c und 4.5d). Um die



(a) Oseen-Gleichung

(b) k-ϵ-Gleichung

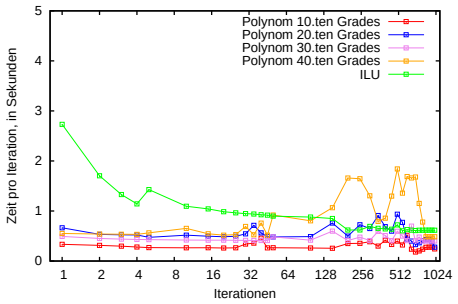
(c) Temperatur-Gleichung

(d) Luftalter-Gleichung

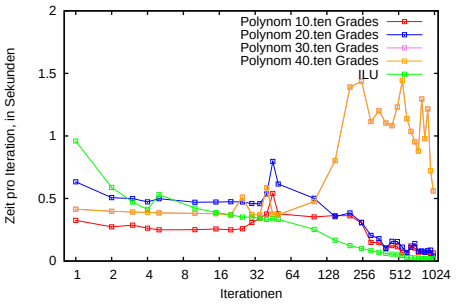
Abbildung 4.5.: Konvergenzdiagramm des QMRCGSTAB-Lösers mit ILU und Polynomen unterschiedlichen Grades als Vorkonditionierer

gleiche Abbruchgenauigkeit von 10^{-4} zu erreichen, benötigt man bei der Oseen-Gleichung circa fünf mal mehr Iterationen (siehe Diagramm 4.5a). Bei der Gleichung für das Luftalter

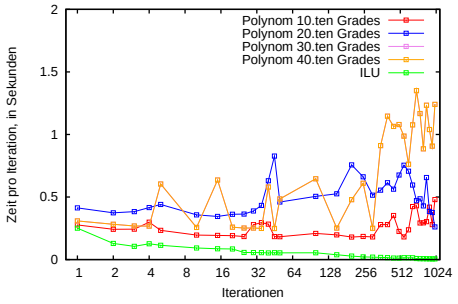
(vergleiche Diagramm 4.5d) sind ungefähr acht mal mehr Iterationen nötig. Das Verhältnis ist bei der Gleichung für die Temperatur mit eins zu 43 relativ schlecht (siehe Diagramm 4.5c). Ein Vergleich bei der k-ϵ-Gleichung ist nicht möglich, da die geforderte Abbruchgenauigkeit niemals erreicht wird (vergleiche Diagramm 4.5b). In den vier nun folgenden Diagrammen wird wieder die Zeit pro Iteration gegen die Anzahl der Iteration dargestellt. Diese ähneln sich im Kurvenverlauf, verglichen mit jenen der GMRES Implementierung (siehe Diagramme 4.4a bis 4.4d) sehr.



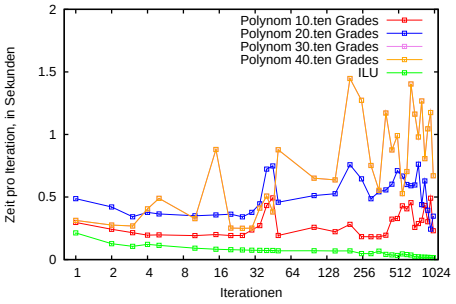
(a) Oseen-Gleichung



(b) k-ϵ-Gleichung



(c) Temperatur-Gleichung



(d) Luftalter-Gleichung

Abbildung 4.6.: Zeit pro Iteration des QMRCGSTAB-Lösers mit ILU und Polynomen unterschiedlichen Grades als Vorkonditionierer

Betreffend der Oseen-Gleichung hat der polynomielle Vorkonditionierer bis zu 128 Iterationen ein besseres Verhältnis als die ILU-Implementierung (siehe Abbildung 4.6a). Bei einer größeren Anzahl von Iterationen verbessert sich das Verhältnis beim ILU Vorkonditio-

nierer. Wenn man die Diagramme für die Gleichungen der Temperatur und des Luftalters betrachtet, stellt man jedoch fest, dass der ILU Vorkonditionierer von Anfang an besser ist als der polynomielle Vorkonditionierer (siehe Diagramme 4.6c und 4.6d). Bei der Betrachtung der Abbildung, die die k-ϵ-Gleichung repräsentiert (siehe Diagramm 4.6b), erkennt man wiederum, dass der polynomielle Vorkonditionierer im Bereich von eins bis 32 Iterationen eine geringere Zeit pro Iteration aufweist, als der ILU Vorkonditionierer.

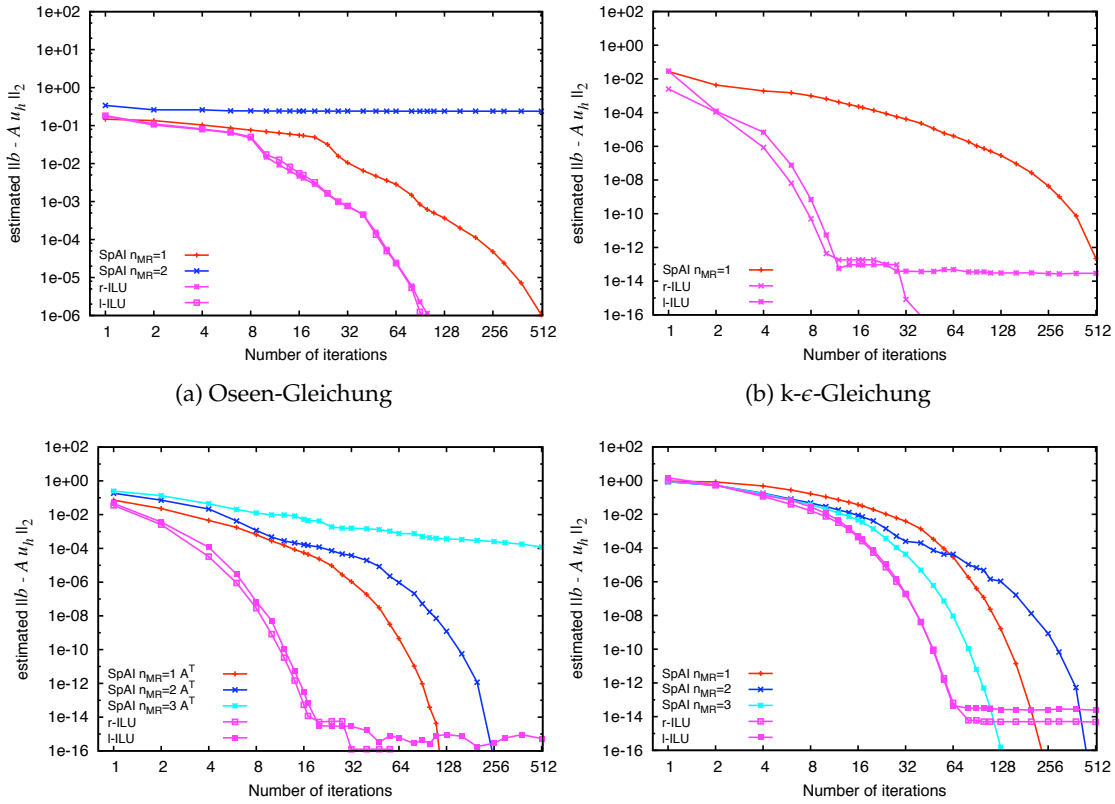
	polynomiell		ILU	
Gleichung	Anzahl der Iterationen	Zeit, in Sek.	Anzahl der Iterationen	Zeit, in Sek.
Oseen	250	$\sim$360	30	$\sim$28
k-ϵ	-	-	1	1
Temperatur	100	$\sim$66	2	0.25
Luftalter	150	$\sim$157	20	1.69

Tabelle 4.2.: Erreichen der Abbruchgenauigkeit von 10^{-4}

Wenn man nun eine bestimmte Abbruchgenauigkeit von 10^{-4} vorgibt und den polynomiellen- mit dem ILU Vorkonditionierer vergleicht, ergibt sich bei der Oseen-Gleichung folgendes. Im polynomiellen Fall benötigt man ungefähr 250 Iterationen, die circa 360 Sekunden benötigen. ILU beansprucht jedoch nur 30 Iterationen, welche in ungefähr 28 Sekunden durchlaufen sind. Für die k-ϵ-Gleichung konnte kein Vergleich getroffen werden, da der polynomielle Vorkonditionierer mit zunehmender Anzahl der Iterationen niemals die Abbruchgenauigkeit von 10^{-4} erreicht (siehe Diagramm 4.5b). Jedoch ist davon auszugehen, dass der polynomielle Vorkonditionierer die Vorgaben des ILU Vorkonditionierers nicht unterbieten wird. Dieser benötigt bei der k-ϵ-Gleichung nur eine Iteration, die in ungefähr einer Sekunde berechnet wurde. Für die Temperatur- bzw. Luftalter-Gleichung ergibt sich leider der gleiche Trend wie bei der Oseen-Gleichung, siehe Tabelle 4.2.

4.4. Ergebnisse mit einer dünn besetzten approximierten Inversen als Vorkondtionierer

In diesem Unterabschnitt werden kurz die Ergebnisse über die Vorkonditionierung mittels einer dünn besetzen approximierten Inversen vorgestellt. Diese Vorkonditionierungstechnik erreicht schneller eine vorgegebene Abbruchgenauigkeit als die polynomiale Vorkonditionierer.



(a) Oseen-Gleichung

(b) k-ϵ-Gleichung

(c) Temperatur-Gleichung

(d) Luftalter-Gleichung

Abbildung 4.7.: Konvergenzdiagramm des GMRES-Lösers mit ILU und einer dünn besetzen approximierten Inversen als Vorkonditionierer

Es werden im Vergleich zum polynomiellen Vorkonditionierer weniger Iterationen benötigt bis man eine Genauigkeit von 10^{-6} erreicht. In der Regel reicht ein MR-Schritt in-

nerhalb der Berechnung der dünn besetzen approximierten Inversen aus, um eine gute Konvergenz sicherzustellen. Wenn man jedoch die Anzahl der MR-Schritte erhöht, kann es unter Umständen zu einer Verschlechterung der Konvergenz führen (siehe Diagramme 4.7a und 4.7c). Im Vergleich zur Vorkonditionierung mittels ILU stellt man allerdings fest, dass auch das Vorkonditionieren durch eine dünn besetzte approximierte Inverse nicht zufriedenstellend ist. Im Fall der Oseen- und Luftalter-Gleichung (siehe Diagramm 4.7a und 4.7d) benötigt man circa vier bis fünf, beziehungsweise bei der Temperatur-Gleichung (siehe Diagramm 4.7c) acht mal mehr Iterationen. Wenn man jedoch die k-ϵ-Gleichung betrachtet, muss man feststellen, dass ungefähr 50 mehr Iterationen gebraucht werden um die Abbruchgenauigkeit von 10^{-14} zu erreichen.

	SpAI		ILU	
Gleichung	Anzahl der Iterationen	Zeit, in Sek.	Anzahl der Iterationen	Zeit, in Sek.
Oseen	512	$\sim$10	100	$\sim$60
k-ϵ	512	$\sim$5	11	$\sim$2
Temperatur	100	$\sim$1	16	$\sim$0.8
Luftalter	110	$\sim$1.1	64	$\sim$3.2

Tabelle 4.3.: Erreichen der Abbruchgenauigkeit von 10^{-14}

Wenn man nun eine bestimmte Abbruchgenauigkeit von 10^{-14} festlegt und den ILU Vorkonditionierer und das Vorkonditionieren durch eine dünn besetze approximierte Inverse (*SpAI*) vergleicht, ergibt sich bei der Oseen-Gleichung ein Beschleunigungsfaktor von sechs. Es werden zwar 512 Iterationen benötigt, allerdings werden diese in circa 10 Sekunden abgearbeitet. Für die anderen Gleichungen ergibt sich jedoch ein geringerer Beschleunigungsfaktor von nur eins bis drei, siehe Tabelle 4.3.

5. Zusammenfassung und Ausblick

5.1. Zusammenfassung

Diese Arbeit beschäftigt sich mit der Implementierung eines polynomiellen Vorkonditionierers zur Lösung großer linearer Gleichungssysteme basierend auf der CUDA-Architektur. Diese Aufgabe wurde erfolgreich umgesetzt, jedoch lassen die Ergebnisse zur Zeit nur den Schluss zu, den polynomiellen Vorkonditionierer weiter zu optimieren. Zum Zeitpunkt der Abgabe sind noch nicht alle Möglichkeiten bei der Wahl der Einschlussmenge getestet. Die Nutzung der dünn besetzten approximierten Inversen als Vorkonditionierungstechnik stellt eine bessere Alternative dar. Betreffend der Oseen-Gleichung erreicht man einen Beschleunigungsfaktor von bis zu sechs gegenüber der ILU Vorkonditionierung auf einer CPU. Der Beschleunigungsfaktoren bei der k-ϵ-, Temperatur- und Luftalter-Gleichung fallen zwar geringer aus, jedoch bleibt die Ausführung der dünn besetzten approximierten Inversen auf der Grafikkarte schneller als die Berechnung der unvollständigen LU Zerlegung auf dem Prozessor.

5.2. Ausblick

Bei Weiterführung des Projekts müsste man beim polynomiellen Vorkonditionierer die Bestimmung der Einschlussmenge und der approximierten Spektren der Matrix automatisieren. Eventuell findet man auch Einschlussmengen, die das Spektrum besser umschließen und weniger Iterationen bis zur gewünschten Abbruchgenauigkeit benötigen. Zudem sollten möglichst viele Operationen zu einem parallelen Programmfragment zusammengefügen werden, um einen weiteren Geschwindigkeitsgewinn vermutlich zu erhalten.

Abbildungsverzeichnis

Literaturverzeichnis

[1] T. F. Chan, E. Gallopoulos, V. Simoncini, T. Szeto, and C. H. Tong. A quasi-minimal residual variant of the bi-cgstab algorithm for nonsymmetric systems. *SIAM J. Sci. Comput.*, 15:338–347, 1994.

[2] R. Döffinger. Physikalische Hochleistungssimulationen auf GPU-Systemen. 2009.

[3] Martin H. Gutknecht and Stefan Röllin. The Chebyshev iteration revisited, 2001.

[4] Bao jiang Zhong. A product hybrid gmres algorithm for nonsymmetric linear systems. *Journal of Computational Mathematics*, 23(1):83, 2005.

[5] Wayne Joubert. On the convergence behavior of the restarted gmres algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 1:427–447, 1992.

[6] Wayne Joubert. A robust gmres-based adaptive polynomial preconditioning algorithm for nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 15:427–439, 1994.

[7] Tobias Knopp, Gert Lube, Ralf Gritzki, and Markus Rösler. A near-wall strategy for buoyancy-affected turbulent flows using stabilized fem with applications to indoor air flow simulation. *Computer Methods in Applied Mechanics and Engineering*, 194(36-38):3797 – 3816, 2005.

[8] Tino Koch and Jörg Liesen. The conformal 'bratwurst' maps and associated Faber polynomials. *Numerische Mathematik*, 86(1):173–191, 2000.

[9] S. Kramer, G. Lube, R. Gritzki, M. Rösler, and C. Pfaffenbach. Parallel preconditioning strategies for decoupled indoor air flow simulation. *NAM-Preprint*, 2011.

[10] Ruipeng Li and Yousef Saad. Gpu-accelerated preconditioned iterative linear solvers. *Minnesota Supercomputer Institute, University of Minnesota*, 2010.

[11] Jorg Liesen. Construction and analysis of polynomial iterative methods for non-hermitian systems of linear equations. *Universität Bielefeld*, 1998.

[12] QT Nokia. Website, 2011. Homepage: `http://qt.nokia.com`.

[13] NVIDIA Corporation. *Fermi Compute Architecture Whitepaper*.

[14] NVIDIA Corporation. The cuda compiler driver nvcc, 2007.

[15] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010. Version 3.2.

[16] A. Priesnitz. Untersuchung iterativer Lösungsverfahren am Beispiel diskretisierter Konvektions-Diffusions-Reaktions-Gleichungen. 1996.

[17] Yousef Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. Society for Industrial and Applied Mathematics, 2 edition, 2003.

[18] Gerhard Starke and Richard S. Varga. A hybrid arnoldi-farber iterative method for nonsymmetric systems of linear equations, 1993.

[19] Charles H. Tong. A family of quasi-minimal residual methods for nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 15:89–105, 1994.

A. Quellcode des polynomiellen Vorkonditionierers

Auf den nun folgenden Seiten wird der Programmteil aufgeführt, der den polynomiellen Vorkonditionierer implementiert. Die Dokumentation wurde mittels Doxygen erzeugt.

Funktion: precond_pol_exec

Diese Funktion wendet den polynomiellen Vorkonditionierer an.

Parameters:

m : Matrix, die vorkonditioniert werden soll
x : Vektor, der später die Lösung enthält
b : rechte Seite
omega : Für diesen Vorkonditionierer unwichtig

Stage 0:

$$\rho_{n-2} = 2$$

```
rho_n_2 = 2;
```

$$S = -\nu_0/\nu_1$$

```
S = -nu_0/nu_1;
```

$$\vec{r}_0 = \vec{b} - A\vec{x}_0$$

```
cublanc_vmv_resid( &r_0, b, m, &x_0 );
```

$$f_{n-2} = 2$$

```
f_n_2 = 2;
```

$$\vec{F}_{n-2} = 2\vec{r}_0$$

```
cublanc_rv_mul( &F_n_2, 2.0, &r_0);
```

$$\vec{G}_{n-2} = \vec{0}$$

```
cublanc_v_entries_set( &G_n_2, 0.0);
```

Stage 1:

$$f_{n-1} = -\mu_1$$

```
f_n_1     = -mu_1;
```

$$\varrho_{n-1} = f_{n-1} - S$$

```
rho_n_1   = f_n_1 - S;
```

$$\vec{F}_{n-1} = \mu_1 \vec{r}_0$$

```
cublanc_rv_mul( &F_n_1, mu_1, &r_0);
```

$$\vec{tmp} = A\vec{r}_0$$

```
    cublanc_mv_mul( &tmp, m, &r_0);
```

$$\vec{F}_{n-1} = -\mu_1 \cdot \vec{r}_0 + \nu_1 \cdot \vec{tmp}$$

```
cublanc_rvrv_addmul(&F_n_1, -mu_1, &r_0, nu_1, &tmp);
```

$$\vec{G}_{n-1} = -\nu_1 \vec{r}_0$$

```
cublanc_rv_mul(&G_n_1, -nu_1/rho_n_1, &r_0);
```

$$\vec{r}_n = \frac{1}{\varrho_{n-1}} \cdot \vec{F}_{n-1} - \frac{S}{\varrho_{n-1}} \cdot \vec{r}_0$$

```
cublanc_rvrv_addmul(&r_n, 1/rho_n_1, &F_n_1, -S/rho_n_1, &r_0);
```

Rekursion:

```
    for( size_t n = 2; n < n_iter; n++)
    {
```

$$f_n = -\mu_1 f_{n-1} - \mu_0 f_{n-2}$$

```
        f_n   = -mu_1 * f_n_1 - mu_0 * f_n_2;
```

$$\varrho_n = f_n - S^n$$

```
        rho_n = f_n - pow( S, n);
```

$$\vec{v}_n = \nu_1 \cdot \vec{F}_{n-1} + \nu_0 \cdot \vec{F}_{n-2}$$

```
        cublanc_rvrv_addmul(&v_n, nu_1, &F_n_1, nu_0, &F_n_2);
```

$$\vec{F}_n = M \cdot \vec{v}_n - \mu_1 \cdot \vec{F}_{n-1} - \mu_0 \cdot \vec{F}_{n-2}$$

```
        cublanc_mv_mul( &F_n, m, &v_n, -mu_1, &F_n_1, -mu_0, &F_n_2 );
```

$$\vec{v}_n = -\vec{v}_n$$

```
        cublanc_rv_mul(&v_n, -1, &v_n);
```

$$\vec{G}_n = \vec{v}_n - \mu_1 \cdot \varrho_{n-1} \cdot \vec{G}_{n-1} - \mu_0 \cdot \varrho_{n-2} \cdot \vec{G}_{n-2}$$

```
        cublanc_vrvrv_addmul(&G_n, &v_n, -mu_1*rho_n_1, &G_n_1, -mu_0*rho_n_2, &G_n_2 );
```

$$\vec{G}_n = \frac{1}{\varrho_n} \cdot \vec{G}_n$$

```
        cublanc_rv_mul(&G_n, 1/rho_n, &G_n);
```

$$\vec{r}_n = \frac{1}{\varrho_n} \cdot \vec{F}_n - \frac{S^n}{\varrho_n} \cdot \vec{r}_0$$

```
        cublanc_rvrv_addmul(&r_n, 1/rho_n, &F_n, -pow(S,n)/rho_n, &r_0);
```

$$\begin{aligned} \vec{F}_{n-2} &= \vec{F}_{n-1} \\ \vec{G}_{n-2} &= \vec{G}_{n-1} \end{aligned}$$

```
        cublanc_vv_asgn(&F_n_2, &F_n_1);
        cublanc_vv_asgn(&G_n_2, &G_n_1);
```

$$\begin{aligned} f_{n-2} &= f_{n-1} \\ \varrho_{n-2} &= \varrho_{n-1} \end{aligned}$$

```
        f_n_2   = f_n_1;
        rho_n_2 = rho_n_1;
```

$$\begin{aligned} \vec{F}_{n-1} &= \vec{F}_n \\ \vec{G}_{n-1} &= \vec{G}_n \end{aligned}$$

```
        cublanc_vv_asgn(&F_n_1, &F_n);
        cublanc_vv_asgn(&G_n_1, &G_n);
```

$$\begin{aligned} f_{n-1} &= f_n \\ \varrho_{n-1} &= \varrho_n \end{aligned}$$

```
        f_n_1   = f_n;
        rho_n_1 = rho_n;

    }
```

$$\vec{x} \quad = \quad \vec{x}_0 + (1/\varrho_n)\vec{G}_n$$

```
cublanc_vrv_addmul( x, &x_0, 1, &G_n);
```

B. Quellcode des Beispielsprogramms

Auf den nun folgenden Seiten wird die Dokumentation des Beispielprogramms aufgelistet. Es zeigt exemplarisch das Einlesen einer Matrix, dessen Besetzungsmusters, der rechten Seite und der original Lösung. Anschließend werden diese Daten in den Grafikkartenspeicher kopiert und das dünn besetze Gleichungssystem sowohl durch den Prozessor als auch durch die Grafikkarte mittels verschiedener iterativer Löser in Kombination mit Vorkonditionierern gelöst. Das Programm ist ebenfalls in Doxygen verfasst.

blanc++ Reference Manual for Version 1.0.0

The step-10 tutorial program

Table of contents

Introduction

Dies ist ein Beispielprogramm aus einer Serie von Programmen, die sich damit beschäftigen BLANC CUDA-fähig zu machen. In diesem Programm werden zwei Vorkonditionierer auf ihre Performance und ihre Genauigkeit getestet. Es handelt sich zum einen um den *Sparse Approximate Inverse Preconditioner* und zum anderen um den *Poylnomial Preconditioner* , die auf Grundlage von BLANC-Befehlen mittels CUDA parallelisiert wurden. Die beiden Vorkonditionierer werden im GMRES Löser eingesetzt, da dieser sich durch die Berechnung der Eigenwerte als optimal erwiesen hat.

The commented program

```
#ifndef CUDADriver_STEP_10_H
#define CUDADriver_STEP_10_H

#include <QStringList>
#include <QString>
```

eigene Header

```
#include <blanc.h>
#include <cublanc.h>
```

STL-Header

```
#include <string.h>
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <vector>
```

deal.II-Header

```
#include <base/parameter_handler.h>
#include <base/convergence_table.h>
```

Jedes Beispielprogramm kommt in einen separaten namespace, damit man die verschiedenen Treiberklassen ggf. später in komplexeren Projekten kombinieren kann.

```
namespace step10 {
```

Klasse: CublancPrecondTest

Diese Klasse steuert die host-device-Kommunikation, also Datentransfer und wann welcher Kernel aufgerufen wird. Die Dokumentation der member-Funktionen steht bei und in der Implementierung.

```
class CublancPrecondTest {

public:

    CublancPrecondTest(std::string prm_filename);

    void run();

    void bench_test_gpu( );
    void bench_test_cpu( );

    void calc_l2_norm();

    void read_data();

    void setup_and_assemble_cpu();
    void setup_gpu();
    void destruct_all();

    void declare(::ParameterHandler & prm);
    void get(::ParameterHandler & prm);

    void write_data(cublanc_Vector *sol, char *filename);
    void write_data(blanc_Vector *sol, char *filename);

private:
    blanc_Vector    __x_prec_h;
    blanc_Vector    __x_none_h;
    blanc_Vector    __x_h_gpu;
    blanc_Vector    __rhs_h, __source_rhs_h;
    blanc_Vector    __u_h, __source_u_h ;

    cublanc_Vector  __x_prec_d;
    cublanc_Vector  __x_none_d;
    cublanc_Vector  __rhs_d;
    cublanc_Vector  __u_d;
```

```
    cublanc_Matrix  __B_d;
    cublanc_Mpat    __B_pattern_d;

    blanc_Vector    __u_ralf_h;

    blanc_Matrix    __A_h, __source_A_h   ;
    blanc_Mpat      __A_pattern_h, __source_A_pattern_h;
    blanc_Melt      __A_unit_h;
    blanc_Melt      __A_diag_h;
    blanc_Melt      __A_off_diag_h;



    std::string m_file, mp_file, rs_file, u_file;

    size_t __eltdim;
    size_t __eltnum;
    size_t __no_of_eig, __no_of_n_iter;

    std::map<std::string, blanc_Solver_id> blancSolverNameToID;
    std::map<std::string, cublanc_Solver_id> cuSolverNameToID;

    std::map<std::string, cublanc_Precond_id> cuPrecondNameToID;


    ::ConvergenceTable _results_cnv;
    ::ConvergenceTable _results_distances;
    ::ConvergenceTable _results_times;

    std::map<std::string, ::ConvergenceTable> _table_of_results;

    double __dx;
    double __tol;

    QStringList __iterats_list;
    size_t __iterats;

    bool __print_eigenvalues;
    bool __calc_eigenvalues;

    cublanc_Real __mu_0, __mu_1, __nu_0, __nu_1;
    bool __internal;

    QStringList __solver_list;
    QString __cuPrecond_id;
    std::vector<std::string> __tokens_solver;
    std::string __act_solver;

    std::string __prm_filename;
};

} // namespace step10 END

#endif // CUDADriver_STEP_10_H



#ifndef CUDA_DRIVER_STEP_10_HH
#define CUDA_DRIVER_STEP_10_HH
#include "cuda_driver_step-10.h"
```

Deklarationen der kernel-wrapper-Funktionen.

```
#include <kernels/cuda_kernels.h>
```

CUDA-Header include <cutil_inline.h>

eigene Header

```
#include <cublanc.h>
#include <blanc.h>
```

STL-Header

```
#include <ctype.h>
#include <time.h>
#include <vector>
#include <fstream>
```

QT-Header

```
#include <QTime>
#include <QVector>
#include <QString>
```

Konstruktor: CublancPrecondTest

Der Konstruktor der Treiber-Klasse allokiert den Speicher auf der GPU und kopiert die Daten vom host auf die GPU.

```
step10::CublancPrecondTest::CublancPrecondTest(std::string prm_filename)
    : __prm_filename(prm_filename)

{
```

Verlinkung der Solvernamen mit den entsprechenden ID's, die CuBlanc bzw. Blanc vorgibt. So reicht es aus, den Namen des Solvers später zu schreiben.

```
    cuSolverNameToID["CG"]          = cublanc_SOLVER_CG;
    cuSolverNameToID["CGS"]         = cublanc_SOLVER_CGS;
    cuSolverNameToID["TFQMR"]       = cublanc_SOLVER_TFQMR;
    cuSolverNameToID["GMRES"]       = cublanc_SOLVER_GMRESM ;
    cuSolverNameToID["BICGSTAB"]    = cublanc_SOLVER_BICGSTAB;
    cuSolverNameToID["QMRCGSTAB"]   = cublanc_SOLVER_QMRCGSTAB;
    cuSolverNameToID["GMRES_TEST"]  = cublanc_SOLVER_GMRESM_TEST;
    cuSolverNameToID["POL"]  = cublanc_SOLVER_POL;
```

Weil die Solver-ID's für Blanc und CuBlanc unterschiedlich sind, brauch man dieses Mapping zweimal

```
    blancSolverNameToID["CG"]          = blanc_SOLVER_CG;
    blancSolverNameToID["CGS"]         = blanc_SOLVER_CGS;
    blancSolverNameToID["TFQMR"]       = blanc_SOLVER_TFQMR;
    blancSolverNameToID["GMRES"]       = blanc_SOLVER_GMRESM ;
    blancSolverNameToID["BICGSTAB"]    = blanc_SOLVER_BICGSTAB;
    blancSolverNameToID["QMRCGSTAB"]   = blanc_SOLVER_QMRCGSTAB;
    blancSolverNameToID["GMRES_TEST"]  = blanc_SOLVER_GMRESM_TEST;
```

Verlinkung der Precondnamen mit den entsprechenden ID's, die CuBlanc vorgibt

```
    cuPrecondNameToID["NONE"]           = cublanc_PRECOND_NONE;
    cuPrecondNameToID["AINV"]           = cublanc_PRECOND_AINV;
    cuPrecondNameToID["POL"]           = cublanc_PRECOND_POL;

    _table_of_results["cnv"]      = _results_cnv;
```

```
    _table_of_results["distance"] = _results_distances;
    _table_of_results["times"]    = _results_times;

}
```

Funktion: run

Diese Funktion steuert den Aufruf des Kernels. Wenn eine Klasse mehrere Kernel ansteuern soll, können natürlich auch mehrere run()-Funktion angelegt werden. Sinnigerweise sollten diese so benannt sein, dass man erkennen kann, welcher Kernel von dieser Funktion aufgerufen wird.

```
void step10::CublancPrecondTest::run()
{
    cublanc_Init();

    ::ParameterHandler prm_handler;

    declare(prm_handler);

    prm_handler.read_input (__prm_filename);
    get(prm_handler);

    cublanc_precond_set_parameter(__mu_0, __mu_1, __nu_0, __nu_1);

    QVector<QString> tokens_solver  = QVector<QString>::fromList(__solver_list);
    QVector<QString> tokens_iterats = QVector<QString>::fromList(__iterats_list);


    prm_handler.print_parameters(std::cout , ::ParameterHandler::Text );

    if(__print_eigenvalues){
        printf("Erstellung eines Matlab-Skripts fÃ¼r die Hessenbergmatrix\n");
        FILE* ostream1 = blanc_file_open("output/matlab_skript_gpu.txt","w");
        fprintf(ostream1,"figure;\n");
        fprintf(ostream1,"clear all;\n");
        fclose(ostream1);

        FILE* ostream2 = blanc_file_open("output/matlab_skript_cpu.txt","w");
        fprintf(ostream2,"figure;\n");
        fprintf(ostream2,"clear all;\n");
        fclose(ostream2);

        cublanc_solver_print_eigenvalues_on();
        blanc_solver_print_eigenvalues_on();
    }

    if(__calc_eigenvalues){
        printf("Erstellung eines Matlab-Skripts fÃ¼r die Eigenwerte\n");
        FILE* ostream_eig = blanc_file_open("output/matlab_skript_gpu_plot.txt","w");
        fprintf(ostream_eig,"figure;\n");
        fprintf(ostream_eig,"clear all;\n");
        fclose(ostream_eig);

        ostream_eig = blanc_file_open("output/matlab_skript_cpu_plot.txt","w");
        fprintf(ostream_eig,"figure;\n");
        fprintf(ostream_eig,"clear all;\n");
        fclose(ostream_eig);

        cublanc_solver_calc_eigenvalues_on();
        blanc_solver_calc_eigenvalues_on();
    }


    if(__internal)
            setup_and_assemble_cpu();
    else
```

```
        read_data();

    for(int i = 0; i < tokens_solver.size(); i++)
    {

        std::cout  << tokens_solver.at(i).toStdString() << std::endl;
        __act_solver = tokens_solver.at(i).toStdString();

        for(unsigned int j = 0; j < tokens_iterats.size(); j++)
        {
            __iterats = tokens_iterats.at(j).toInt();

            std::cout << "No of iteration \t" << __iterats << std::endl;

            bench_test_gpu();
            bench_test_cpu();

            if(__print_eigenvalues){
                FILE* ostream3 = blanc_file_open("output/matlab_skript_cpu.txt","a");
                fprintf(ostream3,"legend('%d');\n",j+1);
                fprintf(ostream3,"M(%d) = getframe;\n",j+1);
                fclose(ostream3);
            }
        }

        * Zwei Leerzeilen
        std::cout << std::endl << std::endl;
```

Tabelle mit den Ergebnissen

```
        std::ofstream ostream_tex_cnv("output/Tabelle_CNV.tex");
        _table_of_results["cnv"].write_tex( ostream_tex_cnv, true );
        ostream_tex_cnv.close();

        std::ofstream ostream_tex_dis("output/Tabelle_Distance.tex");
        _table_of_results["distance"].write_tex( ostream_tex_dis, true );
        ostream_tex_dis.close();

        std::ofstream ostream_tex_times("output/Tabelle_Times.tex");
        _table_of_results["times"].write_tex( ostream_tex_times, true );
        ostream_tex_times.close();


    }

    blanc_v_destr(&__rhs_h);
    blanc_v_destr(&__u_h);
    blanc_m_destr(&__A_h);
    blanc_mp_destr(&__A_pattern_h);
```

Berechnet die L2-Norm von Vektor U.

```
    calc_l2_norm();

    cublanc_Shutdown();
}
```

Funktion: read_data

Diese Funktion liest die Beispieldaten aus den Dateien ein.

```
void step10::CublancPrecondTest::read_data()
{
```

rechte Seite-Vektor für GPU und CPU konstruieren

```
#ifdef DEBUG
        std::cout << "Read:\t right side" << std::endl;
#endif

    blanc_v_read(&__rhs_h, __eltnum, __eltdim, rs_file.c_str());
```

orig Lsg-Vektor für GPU und CPU konstruieren

```
#ifdef DEBUG
      std::cout << "Read:\t original solution" << std::endl;
#endif

    blanc_v_read(&__u_h, __eltnum, __eltdim, u_file.c_str());
```

Pattern konstruieren und einlesen

```
#ifdef DEBUG
      std::cout << "Read:\t matrix pattern" << std::endl;
#endif

    blanc_mp_read(&__A_pattern_h, __eltnum, mp_file.c_str() );
```

Matrix konstruieren und einlesen

```
#ifdef DEBUG
        std::cout << "Read:\t matrix" << std::endl;
#endif

    blanc_m_read(&__A_h, &__A_pattern_h, __eltdim, m_file.c_str());


}
```

Funktion: bench_test_gpu

Diese Funktion rechnet die Beispieldaten auf der GPU durch. Es werden die Daten eingelesen und verarbeitet. Anschließend das Gleichungssystem $y = A \cdot x$ gelöst und der Lösungsvektor von der GPU zur CPU zurückkopiert und in eine Datei ausgegeben.

```
void step10::CublancPrecondTest::bench_test_gpu()
{

    _table_of_results["cnv"].add_value("N iter",int(__iterats));
    _table_of_results["distance"].add_value("N iter",int(__iterats));
    _table_of_results["times"].add_value("N iter",int(__iterats));


    cublanc_v_init(&__rhs_d);
    cublanc_v_init(&__u_d);
    cublanc_v_init(&__x_prec_d);
    cublanc_v_init(&__x_none_d);
```

Vektor für die Differenz zwischen orig.- und berechneter Lsg.

```
    cublanc_Vector  diff_d            = cublanc_V_INIT;

    cublanc_v_constr  ( &diff_d, __eltnum, __eltdim, NULL );
    cublanc_v_entries_set(&diff_d,0.0);
```

Lsg. Vektor für die GPU

```
    cublanc_v_constr  ( &__x_prec_d, __eltnum, __eltdim, NULL );
    cublanc_v_entries_set(&__x_prec_d,0.0);

    cublanc_v_constr  ( &__x_none_d, __eltnum, __eltdim, NULL );
    cublanc_v_entries_set(&__x_none_d,0.0);

    cublanc_v_constr  ( &__rhs_d, __eltnum, __eltdim, NULL );
    cublanc_v_constr  ( &__u_d, __eltnum, __eltdim, NULL );
```

Device Matrix-Pattern initialisieren und konstruieren

```
    cublanc_mp_init(&__B_pattern_d);
    cublanc_mp_constr(&__B_pattern_d, __eltnum, NULL);

    cublanc_mpmp_asgn(&__B_pattern_d, &__A_pattern_h);
```

Device Matrix initialisieren und konstruieren

```
    cublanc_m_init(&__B_d);
    cublanc_m_constr(&__B_d, &__B_pattern_d, __eltdim, NULL);
```

Konditionen setzen

```
    cublanc_Cc cc = cublanc_CC_INIT;
    cublanc_cc_constr ( &cc, NULL );
```

Fehlertoleranz setzen

```
    cublanc_cc_tol_set(&cc,__tol);

    cublanc_vv_asgn(&__rhs_d, &__rhs_h);
    cublanc_vv_asgn(&__u_d,&__u_h);

    cublanc_mm_asgn(&__B_d,&__A_h);
```

Zeitmessung für den Löser starten

```
    QTime solver_time;
    solver_time.start();
```

ist nur wichtig, wenn die Eigenwerte berechnet werden wollen

```
    if(__print_eigenvalues || __calc_eigenvalues) cublanc_solver_restart_set(__iterats);

    cublanc_precond_no_of_eigenvalues_set( __no_of_eig );
    cublanc_precond_no_of_niteration_set(__no_of_n_iter);
```

Vorkonditionierter Löser

```
    cublanc_solver ( cuSolverNameToID[__act_solver],
```

```
                    &__B_d,
                    &__x_prec_d,
                    &__rhs_d,
                    &__iterats,
                    &cc,
                    cuPrecondNameToID[__cuPrecond_id.toStdString()],
                    NULL,
                    NULL );

    int solver_t = solver_time.elapsed();

    QString text;
    text = "u_{GPU_{"+__cuPrecond_id+"}} = A^{-1}b /sec";

    _table_of_results["times"].add_value( text.toStdString() , double(0.001*(solver_t)));
    _table_of_results["times"].set_precision(text.toStdString(), 8);

    text = "CNV_{"+__cuPrecond_id+"} GPU";
    _table_of_results["cnv"].add_value(text.toStdString(), double(cublanc_vc_r_get(&cc.cnv)));
    _table_of_results["cnv"].set_scientific(text.toStdString(),true);
```

Zeitmessung für den Löser starten

```
    solver_time;
    solver_time.start();
```

Unvorkonditionierter Löser

```
    cublanc_solver ( cuSolverNameToID[__act_solver],
                    &__B_d,
                    &__x_none_d,
                    &__rhs_d,
                    &__iterats,
                    &cc,
                    cublanc_PRECOND_NONE,
                    NULL,
                    NULL );



    solver_t = solver_time.elapsed();

    _table_of_results["times"].add_value("u_{GPU_{None}} = A^{-1}b /sec", double(0.001*(solver_t)));
    _table_of_results["times"].set_precision("u_{GPU_{None}} = A^{-1}b /sec", 8);

    _table_of_results["cnv"].add_value("CNV_{None} GPU", double(cublanc_vc_r_get(&cc.cnv)));
    _table_of_results["cnv"].set_scientific("CNV_{None} GPU",true);


    cublanc_cc_destr ( &cc );

    cublanc_Real scalar = 0.0;
```

Differenz zwischen `__x_prec_d` bzw. `__x_none_d` und `__u_d` berechnen

```
    cublanc_vv_sub(&diff_d, &__x_prec_d, &__u_d);
    scalar = cublanc_v_n2(&diff_d);


    text = " \| u_{GPU_{"+ __cuPrecond_id +"}} - u_{orig} \|_{2} ";
    _table_of_results["distance"].add_value(text.toStdString(), double(scalar));
    _table_of_results["distance"].set_precision(text.toStdString(), true);

    cublanc_vv_sub(&diff_d, &__x_none_d, &__u_d);
    scalar = cublanc_v_n2(&diff_d);
```



```
    _table_of_results["distance"].add_value(" \| u_{GPU_{None}} - u_{orig} \|_{2} ", double(scalar));
    _table_of_results["distance"].set_precision(" \| u_{GPU_{None}} - u_{orig} \|_{2} ", true);
```

GPU-Vektor wieder zurückkopieren für spätere Berechnung...

```
    blanc_v_init(&__x_h_gpu);
    blanc_v_constr  ( &__x_h_gpu, __eltnum, __eltdim, NULL );
    cublanc_vv_asgn(&__x_h_gpu,&__x_prec_d);
```

Destruktoren

```
    cublanc_m_destr(&__B_d);
    cublanc_mp_destr(&__B_pattern_d);

    cublanc_v_destr( &__rhs_d );
    cublanc_v_destr( &__x_prec_d );
    cublanc_v_destr( &__x_none_d );
    cublanc_v_destr(&diff_d);
    cublanc_v_destr(&__u_d);
}
```

Funktion: bench_test_cpu

Diese Funktion rechnet die Beispieldaten auf der CPU durch. Es werden die Daten eingelesen und verarbeitet. Anschließend das Gleichungssystem $y = A \cdot x$ gelöst und der Löseungsvektor von der GPU zur CPU zurückkopiert und in eine Datei ausgegeben.

```
void step10::CublancPrecondTest::bench_test_cpu()
{
```

Vektor für die Differenz zwischen orig.- und berechneter Lsg.

```
    blanc_Vector  x_prec_h            = blanc_V_INIT;
    blanc_v_constr  ( &x_prec_h, __eltnum, __eltdim, NULL );
    blanc_v_entries_set(&x_prec_h,0.0);

    blanc_Vector  x_none_h            = blanc_V_INIT;
    blanc_v_constr  ( &x_none_h, __eltnum, __eltdim, NULL );
    blanc_v_entries_set(&x_none_h,0.0);
```

Vektor für die Differenz zwischen orig.- und berechneter Lsg.

```
    blanc_Vector  diff_h            = blanc_V_INIT;
    blanc_v_constr  ( &diff_h, __eltnum, __eltdim, NULL );
    blanc_v_entries_set(&diff_h,0.0);
```

Konditionen setzen

```
    blanc_Cc cc = blanc_CC_INIT;
    blanc_cc_constr ( &cc, NULL );
```

Fehlertoleranz setzen

```
    blanc_cc_tol_set(&cc,__tol);
```

Zeitmessung für den Löser starten

```
    QTime solver_time;
    solver_time.start();
```

muss später wieder ausgeschaltet werden, nur wichtig, wenn die Eigenwerte ausgegeben werden sollen!!!

```
    if(__print_eigenvalues || __calc_eigenvalues)  blanc_solver_restart_set(__iterats);
```

Vorkonditionierter Löser

```
    blanc_solver ( blancSolverNameToID[__act_solver],
                   &__A_h,
                   &x_prec_h,
                   &__rhs_h,
                   &__iterats,
                   &cc,
                   blanc_PRECOND_ILU,
                   NULL,
                   NULL );


    int solver = solver_time.elapsed();

    _table_of_results["times"].add_value("u_{CPU_{ILU}} = A^{-1}b /sec", double(0.001*(solver)));
    _table_of_results["times"].set_precision("u_{CPU_{ILU}} = A^{-1}b /sec", 8);

    _table_of_results["cnv"].add_value("CNV_{ILU} CPU", double(blanc_vc_r_get(&cc.cnv)));
    _table_of_results["cnv"].set_scientific("CNV_{ILU} CPU",true);
```

Zeitmessung für den Löser starten

```
    solver_time;
    solver_time.start();
```

Unvorkonditionierter Löser

```
    blanc_solver ( blancSolverNameToID[__act_solver],
                   &__A_h,
                   &x_none_h,
                   &__rhs_h,
                   &__iterats,
                   &cc,
                   blanc_PRECOND_NONE,
                   NULL,
                   NULL );

    solver = solver_time.elapsed();

    _table_of_results["times"].add_value("u_{CPU_{None}} = A^{-1}b /sec", double(0.001*(solver)));
    _table_of_results["times"].set_precision("u_{CPU_{None}} = A^{-1}b /sec", 8);

    _table_of_results["cnv"].add_value("CNV_{None} CPU", double(blanc_vc_r_get(&cc.cnv)));
    _table_of_results["cnv"].set_scientific("CNV_{None} CPU",true);


    blanc_cc_destr ( &cc );

    blanc_Real scalar = 0.0;
```

Differenz zwischen `x_prec_h` bzw. `x_none_h` und `__u_h` berechnen

```
    blanc_vv_sub(&diff_h, &x_prec_h, &__u_h);
    scalar = blanc_v_n2(&diff_h);

    _table_of_results["distance"].add_value(" \| u_{CPU_{ILU}} - u_{orig} \|_{2} ", double(scalar));
    _table_of_results["distance"].set_precision(" \| u_{CPU_{ILU}} - u_{orig} \|_{2} ", true);

    blanc_vv_sub(&diff_h, &x_none_h, &__u_h);
    scalar = blanc_v_n2(&diff_h);

    _table_of_results["distance"].add_value(" \| u_{CPU_{None}} - u_{orig} \|_{2} ", double(scalar));
    _table_of_results["distance"].set_precision(" \| u_{CPU_{None}} - u_{orig} \|_{2} ", true);



    scalar = 0.0;
    blanc_vv_sub(&diff_h, &x_prec_h, &__x_h_gpu);
    scalar = blanc_v_n2(&diff_h);


    blanc_v_destr(&x_prec_h);
    blanc_v_destr(&x_none_h);
    blanc_v_destr(&diff_h);



}
```

Funktion: write_data

Diese Funktion schreibt die Werte eines cublanc_Vektors in eine Datei.

Parameters:

vector : Zeiger auf eine cublanc_Vector, dessen Werte in eine Datei ausgegeben werden sollen.

filename : Zeiger auf ein char, der den Dateinamen beinhaltet.

```
void step10::CublancPrecondTest::write_data(cublanc_Vector *vector, char *filename)
{
    FILE* ostream = blanc_file_open(filename,"we");
    cublanc_v_print(vector,ostream);
    fclose(ostream);
}
```

Funktion: write_data

Diese Funktion schreibt die Werte eines blanc_Vektors in eine Datei.

Parameters:

vector : Zeiger auf eine blanc_Vector, dessen Werte in eine Datei ausgegeben werden sollen.

filename : Zeiger auf ein char, der den Dateinamen beinhaltet.

```
void step10::CublancPrecondTest::write_data(blanc_Vector *vector,char *filename)
{

    FILE* ostream = blanc_file_open(filename,"we");
    blanc_v_print(vector,ostream);
    fclose(ostream);
}
```

Funktion: declare

Diese Funktion bringt dem Parameterhandler bei, welche Parameter erwartet werden.

```
void step10::CublancPrecondTest::declare(::ParameterHandler & prm)
{

    prm.enter_subsection("Dimensions of test problems.");

    prm.declare_entry("iterations","1|2|4|8|16|32|64|128|256|512",::Patterns::Anything(),"this is the number o

    prm.declare_entry("eltnum","80621",::Patterns::Integer(1,1000000),"this is die number of elements") ;
    prm.declare_entry("eltdim","4",::Patterns::Integer(1,4),"this is the dimension of an entry");

    prm.declare_entry("ToeplitzMatrix?",
                      "false",
                      ::Patterns::Bool(),
                      "if you want to run this program with an modified toeplitz matrix, you have to set t

    prm.leave_subsection();

    prm.enter_subsection("Paths of the test problem");

    prm.declare_entry("Matrix",
                      "../../TestExamples/Matrix.e0.dat",
                      ::Patterns::Anything(),
                      "Filename of test matrix. May contains relativ paths");

    prm.declare_entry("MatrixPattern",
                      "../../TestExamples/MPAT0.dat",
                      ::Patterns::Anything(),
                      "Filename of test matrix pattern. May contains relativ paths");

    prm.declare_entry("RightSide",
                      "../../TestExamples/RS_VEC.e0.dat",
                      ::Patterns::Anything(),
                      "Filename of test right side. May contains relativ paths");

    prm.declare_entry("U",
                      "../../TestExamples/U_VEC.e0.dat",
                      ::Patterns::Anything(),
                      "Filename of test u. May contains relativ paths");

    prm.leave_subsection();

    prm.enter_subsection("Operations");

    prm.declare_entry("Solver-IDs",
                      "CG|CGS|TFQMR|GMRES|BICGSTAB|QMRCGSTAB|GMRES_TEST",
                      ::Patterns::Anything(),
                      "If you want to run several solver on the GPU, you have to set the IDs of the solver
                      " The possible Solvers are: CG, CGS, TFQMR, GMRESM, BICGSTAB, QMRCGSTAB."
                      " You can additionaly choose GMRES_TEST as solver. This solver calculates the eigenv
                      " The delimiter is | between two solvers.");

    prm.declare_entry("cuPrecond-IDs",
                      "AINV",
                      ::Patterns::Anything(),
                      "You can choose between AINV and POL as parallel preconditioner."
                      "This preconditioner are for the GPU");


    prm.declare_entry("Fehlertoleranz",
                      "1e-6",
                      ::Patterns::Double(1e-14,1e+14),
                      "");

    prm.declare_entry("NoOfEigenvalues",
                      "10",
                      ::Patterns::Integer(0,1000000),
                      "Number of calculated Eigenvalues in the GMRES routine");

    prm.declare_entry("nIteration",
                      "10",
                      ::Patterns::Integer(0,1000000),
                      "Number of Iteration in the polynomial preconditioner");

    prm.declare_entry("CalcEigenvalues",
                      "true",
                      ::Patterns::Bool(),
                      "If you want to calculate all eigenvalues of the hessenbergmatrix through a lapack-r
                      " default: true");

    prm.declare_entry("PrintEigenvalues",
                      "false",
                      ::Patterns::Bool(),
                      "If you want to calculate all eigenvalues through matlab, you have to set this optio
                      " default: false");



    prm.leave_subsection();

    prm.enter_subsection("Parameter of the polynomial preconditioner");

    prm.declare_entry("mu0",
                      "1.0",
                      ::Patterns::Double(-1e+14,1e+14),
                      "");

    prm.declare_entry("mu1",
                      "1.0",
                      ::Patterns::Double(-1e+14,1e+14),
                      "");

    prm.declare_entry("nu0",
                      "1.0",
                      ::Patterns::Double(-1e+14,1e+14),
                      "");

    prm.declare_entry("nu1",
                      "1.0",
                      ::Patterns::Double(-1e+14,1e+14),
                      "");

    prm.leave_subsection();
}
```

Funktion: get

Diese Funktion liest die Parameter in das Objekt.

```
void step10::CublancPrecondTest::get(::ParameterHandler & prm)
{
    prm.enter_subsection("Dimensions of test problems.");

    __iterats_list =  QString(prm.get ("iterations").c_str()).split("|") ;

    __eltnum = prm.get_integer ("eltnum") ;
    __eltdim = prm.get_integer ("eltdim") ;

    __internal = prm.get_bool("ToeplitzMatrix?");

    prm.leave_subsection();

    prm.enter_subsection("Paths of the test problem");

    m_file   = prm.get("Matrix");
    mp_file  = prm.get("MatrixPattern");
    rs_file  = prm.get("RightSide");
```

```
    u_file   = prm.get("U");

    prm.leave_subsection();

    prm.enter_subsection("Operations");

    __solver_list    = QString(prm.get("Solver-IDs").c_str()).split("|");
    __cuPrecond_id   = QString(prm.get("cuPrecond-IDs").c_str());
    __tol            = prm.get_double("Fehlertoleranz");
    __no_of_eig      = prm.get_integer("NoOfEigenvalues");
    __no_of_n_iter      = prm.get_integer("nIteration");
    __calc_eigenvalues      = prm.get_bool("CalcEigenvalues");
    __print_eigenvalues     = prm.get_bool("PrintEigenvalues");

    prm.leave_subsection();

    prm.enter_subsection("Parameter of the polynomial preconditioner");
    __mu_0 = prm.get_double("mu0");

    __mu_1 = prm.get_double("mu1");

    __nu_0 = prm.get_double("nu0");

    __nu_1 = prm.get_double("nu1");

    prm.leave_subsection();
}
```

Funktion: setup_and_assemble_cpu

Diese Funktion legt alle nötigen Vektoren und Matrizen, die auf der CPU sind, an.

```
void step10::CublancPrecondTest::setup_and_assemble_cpu()
{


    blanc_v_init(&__rhs_h);
    blanc_v_constr(&__rhs_h, __eltnum, __eltdim, NULL);
    blanc_v_entries_set(&__rhs_h, 1.0);

    blanc_v_init(&__u_h);
    blanc_v_constr(&__u_h, __eltnum, __eltdim, NULL);
    blanc_v_entries_set(&__u_h, 0.0);

    blanc_mp_init(&__A_pattern_h);
    blanc_m_init(&__A_h);

    blanc_m_constr_grcar(&__A_h, &__A_pattern_h, __eltnum, __eltdim);
}
```

Funktion: setup_and_assemble_gpu

Diese Funktion legt alle nötigen Vektoren und Matrizen, die auf der GPU sind, an.

```
void step10::CublancPrecondTest::setup_gpu()
{
```

Lösungsvektor für die verschiedene Löser

```
    cublanc_v_init(&__x_prec_d);
    cublanc_v_constr  ( &__x_prec_d, __eltnum, __eltdim, NULL );

    cublanc_v_init(&__x_none_d);
    cublanc_v_constr  ( &__x_none_d, __eltnum, __eltdim, NULL );
```

rechte-Seite-Vektor für die verschiedene Löser

```
    cublanc_v_init(&__rhs_d);
    cublanc_v_constr  ( &__rhs_d, __eltnum, __eltdim, NULL );
```

Device Matrix initialisieren und konstruieren

```
    cublanc_mp_init(&__B_pattern_d);
    cublanc_mp_constr(&__B_pattern_d, __eltnum, NULL);
```

Matrix-Pattern auf die Grafikkarte übertragen

```
    cublanc_mpmp_asgn( &__B_pattern_d, &__A_pattern_h);
```

Device Matrix initialisieren und konstruieren

```
    cublanc_m_init(&__B_d);
    cublanc_m_constr(&__B_d, &__B_pattern_d, __eltdim, NULL);
}
```

Funktion: destruct_all

Diese Funktion zerstört alle gebrauchten Matrizen und Vektoren.

```
void step10::CublancPrecondTest::destruct_all()
{
    blanc_me_destr (&__A_diag_h);
    blanc_me_destr (&__A_off_diag_h);

    blanc_m_destr  ( &__A_h );
    cublanc_m_destr( &__B_d);

    blanc_mp_destr ( &__A_pattern_h );
    cublanc_mp_destr ( &__B_pattern_d );

    cublanc_v_destr  ( &__x_prec_d );
    cublanc_v_destr  ( &__x_none_d );

    blanc_v_destr(&__rhs_h);
    cublanc_v_destr  ( &__rhs_d );
}
```

Funktion: calc_l2_norm

Diese Funktion berechnet die L2-Norm von u_{Ralf}.

```
void step10::CublancPrecondTest::calc_l2_norm()
{
    blanc_v_init(&__u_ralf_h);
    blanc_v_read(&__u_ralf_h, __eltnum, __eltdim, u_file.c_str() );
    blanc_Real u_l2_norm = blanc_v_n2(&__u_ralf_h);
    FILE* ostream = blanc_file_open("u_l2_norm.txt","w");
```

```
    fprintf(ostream, "norm_u_ralf = %f", u_l2_norm);
    fclose(ostream);
}

#endif // CUDA_DRIVER_STEP_10_HH
```

Funktion: solver_gmresm_test_exec

Diese Funktion beinhaltet den modifizierten GMRES-Löser. Es werden zusätzlich neben dem GMRES-Algorithmus die Eigenwerte aus der Hessenberg-Matrix berechnet.

Parameters:

m	: Matrix
x	: Lösungsvektor
b	: Vektor der rechten Seite
iterats	: Anzahl der Iterationen die der Löser rechnen soll
cc	: Condition Control
omega	: unwichtig für den GMRES-Löser
precond_id	: ID des Vorkonditionierers der angewendet werden soll, falls kein Vorkondtionierer benutzt werden soll muss dort 'blanc_precond_NONE' stehen.
pi	:unwichtig für den GMRES-Löser
h	: Hessenbergmatrix
y	:
c	: Variable für die Givens-Rotation
s	: Variable für die Givens-Rotation
v	:
temp	: Hilfsvektor, der nur benutzt wird, wenn vorkonditioniert wird
residual	: Residuum

```
static
        void
        solver_gmresm_test_exec ( cublanc_Matrix*  const m,
                                  cublanc_Vector*  const x,
                                  const cublanc_Vector*  const b,
                                  unsigned long* const iterats,
                                  cublanc_Cc*    const cc,
                                  const cublanc_Vcont* const omega,
                                  cublanc_Precond_id precond_id,
                                  size_t (* pi) (size_t),
                                  cublanc_Real**       const h,
                                  cublanc_Real*        const y,
                                  cublanc_Real*        const c,
                                  cublanc_Real*        const s,
                                  cublanc_Vector*      const v,
                                  cublanc_Vector*      const temp,
                                  cublanc_Vcont*       const residual ) {
```

Algorithmus:
Start or Restart:
Compute current residual vector: $r = b - Ax$
Adaptive GMRES step:
Run m_1 steps of GMRES for solving $Ad = r$
Update x by $x = x + d$
Get eigenvalues estimates from the eigenvalues of the Hessenberg matrix
Compute new polynomial:
Refine H from previous hull H and new eigenvalues estimates
Get new best polynomial s_k

Polynomial Iteration:
Compute current residual vector $r = b - Ax$
Run m_2 steps of GMRES applied to $s_k(A)Ad = s_k(A)r$
Update x by $x = x + d$
Test for convergence
If solution converged then stop, else GoTo 1

```
    size_t i          = 0;
    size_t j          = 0;
    size_t k          = 0;

    cublanc_Real         buffer     = 0.0;

    cublanc_Boolean  flag = cublanc_B_TRUE;
    cublanc_Boolean  scalres =  (cublanc_Boolean) ( !vecres
                                                    && (precond_id == cublanc_PRECOND_NONE)
                                                    && (cublanc_cc_type_get ( cc ) == cublanc_VT_REAL)
                                                    && (cublanc_cc_norm_get ( cc ) == cublanc_N2) );

    unsigned long maxiterats = *iterats;
    *iterats = 0;
```

$v_0 = b - M \cdot x$

```
    cublanc_vmv_resid ( &v[0], b, m, x );
    if ( cc ) {
        if ( scalres ) {
            cublanc_vc_r_set  ( residual, cublanc_v_n2 ( &v[0] ) );
        } if
        else {
            cublanc_vc_v_set ( residual, &v[0] );
        } else
        flag = cublanc_cc_check ( cc,
                                  residual,
                                  m,
                                  x,
                                  b,
                                  *iterats,
                                  restart );
        if ( cublanc_error_get() != cublanc_ES_OK ) {
            cublanc_error_set ( cublanc_ES_STARTCC );
            cublanc_error_printf ( "cublanc_solver",
                                   6 );
            return;
        } if
    } if

    while (   flag
              && (*iterats < maxiterats) ) {

        if ( precond_id != cublanc_PRECOND_NONE ) {
```

$temp = v_0$

```
            cublanc_vv_asgn ( temp, &v[0] );
```

$prec(M) \cdot v_0 = temp$

```
            cublanc_precond ( precond_id, m, &v[0], temp, omega, pi );
            if ( cublanc_error_get() != cublanc_ES_OK ) {
                cublanc_error_set ( cublanc_ES_PRECOND );
                cublanc_error_printf ( "cublanc_solver" );
                return;
```

```
        } if
    } if
```

$y_0 = \|v_0\|_2$

```
    y[0] = cublanc_v_n2 ( &v[0] );
    if ( cublanc_r_is_zero ( y[0] ) ) {
        cublanc_error_set ( cublanc_ES_ALGO );
        cublanc_error_printf ( "cublanc_solver" );
        cublanc_error_set ( cublanc_ES_OK );
        return;
    } if
```

$v_0 = \frac{1}{y_0} \cdot v_0$

```
    cublanc_rv_mul ( &v[0], 1.0/y[0], &v[0] );

    i = 0;
    while (    flag
               && (i < restart)
               && (*iterats < maxiterats) ) {

        if ( precond_id != cublanc_PRECOND_NONE ) {
```

$temp = M \cdot v_i$

```
            cublanc_mv_mul ( temp, m, &v[i] );
```

$prec(M) \cdot v_{i+1} = temp$

```
            cublanc_precond ( precond_id, m, &v[i+1], temp, omega, pi );
            if ( cublanc_error_get() != cublanc_ES_OK ) {
                cublanc_error_set ( cublanc_ES_PRECOND );
                cublanc_error_printf ( "cublanc_solver" );
                return;
            } if
        }       if
        else {
```

$v_{i+1} = M \cdot v_i$

```
            cublanc_mv_mul ( &v[i+1], m, &v[i] );

        }else
        for ( j=0 ; j<=i ; j++ ) {
```

$H_{j,i} = (v_{i+1}, v_j)$

```
            h[j][i] = cublanc_vv_mul ( &v[i+1], &v[j] );
```

$v_{i+1} = v_{i+1} - H_{j,i} \cdot v_j$

```
            cublanc_vrv_addmul ( &v[i+1], &v[i+1], -h[j][i], &v[j] );

        } j
```


$H_{i+1,i} = \|v_{i+1}\|_2$

```
            h[i+1][i] = cublanc_v_n2 ( &v[i+1] );

            if ( cublanc_r_is_zero ( h[i+1][i] ) ) {
                cublanc_error_set ( cublanc_ES_ALGO );
                cublanc_error_printf ( "cublanc_solver" );
                cublanc_error_set ( cublanc_ES_OK );
                return;
            } if
```

$v_{i+1} = \frac{1}{H_{i+1,i}} \cdot v_{i+1}$

```
            cublanc_rv_mul ( &v[i+1], 1.0/h[i+1][i], &v[i+1] );
```

Givens-Rotation
Beim Givens-Verfahren werden Ähnlichkeitstransformationen durch ebene Drehungen so ausgeführt, dass die Nichtdiagonalelemente unterhalb (oder oberhalb) der Hauptdiagonalen zeilenweise eleminiert werden. Durch die Drehung $G(\phi, i, j)$ wird das Matrixelement an der Stelle $(i-1, j)$ eleminiert.(Computer-Numerik 2, Seite 378-379).

See also:
http://en.wikipedia.org/wiki/Givens_rotation
http://www.netlib.org/lapack/lawnspdf/lawn150.pdf

```
            for ( j=0 ; j<i ; j++ ) {
                buffer = h[j][i];
                h[j][i] = c[j] * h[j][i] + s[j] * h[j+1][i];
                h[j+1][i] = - s[j] * buffer + c[j] * h[j+1][i];
            } j
```

$buffer = \sqrt{H_{i,i}^2 + H_{i+1,i}^2}$

```
            buffer = cublanc_r_sqrt ( h[i][i] * h[i][i] + h[i+1][i] * h[i+1][i] );
            if ( cublanc_r_is_zero ( buffer ) ) {
                cublanc_error_set ( cublanc_ES_ALGO );
                cublanc_error_printf ( "cublanc_solver" );
                cublanc_error_set ( cublanc_ES_OK );
                return;
            } if
```

$c_i = \frac{H_{i,i}}{buffer}$
$s_i = \frac{H_{i+1,i}}{buffer}$
$H_{i,i} = buffer$
$H_{i+1,i} = 0$
$y_{i+1} = -s_i \cdot y_i$
$y_i = y_i \cdot c_i$

```
            c[i] = h[i][i] / buffer;
            s[i] = h[i+1][i] / buffer;
            h[i][i] = buffer;
            h[i+1][i] = 0.0;
            y[i+1] = - s[i] * y[i];
            y[i] *= c[i];
            i++;
```

```
            (*iterats)++;

            if (    cc
                    && scalres
                    && i < restart
                    && *iterats < maxiterats ) {
                cublanc_vc_r_set  ( residual, cublanc_r_abs (y[i]) );
                flag = cublanc_cc_check ( cc,
                                          residual,
                                          m,
                                          x,
                                          b,
                                          *iterats,
                                          restart );
            } if
        } while

        for ( j=i ; j-->0 ; ) {
            for ( k=i ; --k>j ; ) {
```

$y_j = y_j - H_{j,k} \cdot y_k$

```
                y[j] -= h[j][k] * y[k];
            }       k
```

$y_j = \frac{1}{H_{j,j}} \cdot y_j$

```
            y[j] /= h[j][j];
```

$x = x + y_j \cdot v_j$

```
            cublanc_vrv_addmul ( x, x, y[j], &v[j] );
        } j
```

$v_0 = b - M \cdot x$

```
        cublanc_vmv_resid ( &v[0], b, m, x );

        if (    cc
                && flag ) {
            if ( scalres ) {
                cublanc_vc_r_set  ( residual, cublanc_r_abs (y[i]) );
            }       if
            flag = cublanc_cc_check ( cc,
                                      residual,
                                      m,
                                      x,
                                      b,
                                      *iterats,
                                      restart );
        } if
    } while

    return;
} solver_gmresm_test_exec
```

STL-Header

```
#include <iostream>
#include <vector>
```

QT-Header

```
#include "QTime"
```

Treiber für den GPU-Teil

```
#include "cuda_driver_step-10.h"
```

deal.II-Komponenten include <lac/vector.h>

cublas-Wrapper-Klassen. Binden alle sonstigen benötigten header-Dateien ein.

```
#include "cuda_driver_step-10.hh"

#define USE_DEAL_II
#undef USE_DEAL_II

namespace step10 {
```

Klasse: TestDriver

Diese Klasse steuert die Simulation des eigentlichen physikalischen Problems, welches man untersuchen will.

```
class TestDriver {

public:

    void run(std::string filename);

};


}
```

Funktion: run

Die eigentliche Ausführung der Simulation ist wie so oft in einer run()-Funktion versteckt.

```
void step10::TestDriver::run(std::string filename)
{

    step10::CublancPrecondTest testcase(filename);

    testcase.run();

    std::cout << "Fertig." << std::endl;
}
```

Funktion: main

Hier würden die Benutzereingaben verarbeitet und an die Simulation uebergeben.

```
int main(int argc, char *argv[])
{
    std::string prm_filename( argv[0] );
    prm_filename += ".prm";
```

```
    int n_CUDA_devices;

    cudaGetDeviceCount(&n_CUDA_devices);
    std::cout << "available CUDA devices : " << n_CUDA_devices << std::endl;

    using namespace step10;
    int DevNo = 1;

*    for(int DevNo = 0; DevNo < n_CUDA_devices; DevNo++)
    {
        cudaSetDevice(DevNo);
        cublanc_gpu_info(DevNo);

        TestDriver machma;
        machma.run(prm_filename);
    }

}
```

The plain program

(If you are looking at a locally installed blanc++ version, then the program can be found at *.. /.. /step-by-step / /step-10 /step-cu.cc* . Otherwise, this is only the path on some remote server.)

```
#ifndef CUDADriver_STEP_10_H
#define CUDADriver_STEP_10_H

#include <QStringList>
#include <QString>

#include <blanc.h>
#include <cublanc.h>

#include <string.h>
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <vector>

#include <base/parameter_handler.h>
#include <base/convergence_table.h>

namespace step10 {
```

```
class CublancPrecondTest {

public:

    CublancPrecondTest(std::string prm_filename);

    void run();

    void bench_test_gpu( );
    void bench_test_cpu( );

    void calc_l2_norm();

    void read_data();

    void setup_and_assemble_cpu();
    void setup_gpu();
    void destruct_all();
```

```
    void declare(::ParameterHandler & prm);
    void get(::ParameterHandler & prm);

    void write_data(cublanc_Vector *sol, char *filename);
    void write_data(blanc_Vector *sol, char *filename);

private:
    blanc_Vector    __x_prec_h;
    blanc_Vector    __x_none_h;
    blanc_Vector    __x_h_gpu;
    blanc_Vector    __rhs_h, __source_rhs_h;
    blanc_Vector    __u_h, __source_u_h ;

    cublanc_Vector  __x_prec_d;
    cublanc_Vector  __x_none_d;
    cublanc_Vector  __rhs_d;
    cublanc_Vector  __u_d;

    cublanc_Matrix  __B_d;
    cublanc_Mpat    __B_pattern_d;

    blanc_Vector    __u_ralf_h;

    blanc_Matrix    __A_h, __source_A_h   ;
    blanc_Mpat      __A_pattern_h, __source_A_pattern_h;
    blanc_Melt      __A_unit_h;
    blanc_Melt      __A_diag_h;
    blanc_Melt      __A_off_diag_h;



    std::string m_file, mp_file, rs_file, u_file;

    size_t __eltdim;
    size_t __eltnum;
    size_t __no_of_eig, __no_of_n_iter;

    std::map<std::string, blanc_Solver_id> blancSolverNameToID;
    std::map<std::string, cublanc_Solver_id> cuSolverNameToID;

    std::map<std::string, cublanc_Precond_id> cuPrecondNameToID;


    ::ConvergenceTable _results_cnv;
    ::ConvergenceTable _results_distances;
    ::ConvergenceTable _results_times;

    std::map<std::string, ::ConvergenceTable> _table_of_results;

    double __dx;
    double __tol;

    QStringList __iterats_list;
    size_t __iterats;

    bool __print_eigenvalues;
    bool __calc_eigenvalues;

    cublanc_Real __mu_0, __mu_1, __nu_0, __nu_1;
    bool __internal;

    QStringList __solver_list;
    QString __cuPrecond_id;
    std::vector<std::string> __tokens_solver;
    std::string __act_solver;

    std::string __prm_filename;
};

} // namespace step10 END
```

```
#endif // CUDADriver_STEP_10_H



#ifndef CUDA_DRIVER_STEP_10_HH
#define CUDA_DRIVER_STEP_10_HH
#include "cuda_driver_step-10.h"

#include <kernels/cuda_kernels.h>


#include <cublanc.h>
#include <blanc.h>

#include <ctype.h>
#include <time.h>
#include <vector>
#include <fstream>

#include <QTime>
#include <QVector>
#include <QString>
```

```
step10::CublancPrecondTest::CublancPrecondTest(std::string prm_filename)
    : __prm_filename(prm_filename)

{

    cuSolverNameToID["CG"]          = cublanc_SOLVER_CG;
    cuSolverNameToID["CGS"]         = cublanc_SOLVER_CGS;
    cuSolverNameToID["TFQMR"]       = cublanc_SOLVER_TFQMR;
    cuSolverNameToID["GMRES"]       = cublanc_SOLVER_GMRESM ;
    cuSolverNameToID["BICGSTAB"]    = cublanc_SOLVER_BICGSTAB;
    cuSolverNameToID["QMRCGSTAB"]   = cublanc_SOLVER_QMRCGSTAB;
    cuSolverNameToID["GMRES_TEST"]  = cublanc_SOLVER_GMRESM_TEST;
    cuSolverNameToID["POL"]  = cublanc_SOLVER_POL;

    blancSolverNameToID["CG"]          = blanc_SOLVER_CG;
    blancSolverNameToID["CGS"]         = blanc_SOLVER_CGS;
    blancSolverNameToID["TFQMR"]       = blanc_SOLVER_TFQMR;
    blancSolverNameToID["GMRES"]       = blanc_SOLVER_GMRESM ;
    blancSolverNameToID["BICGSTAB"]    = blanc_SOLVER_BICGSTAB;
    blancSolverNameToID["QMRCGSTAB"]   = blanc_SOLVER_QMRCGSTAB;
    blancSolverNameToID["GMRES_TEST"]  = blanc_SOLVER_GMRESM_TEST;

    cuPrecondNameToID["NONE"]           = cublanc_PRECOND_NONE;
    cuPrecondNameToID["AINV"]           = cublanc_PRECOND_AINV;
    cuPrecondNameToID["POL"]           = cublanc_PRECOND_POL;

    _table_of_results["cnv"]      = _results_cnv;
    _table_of_results["distance"] = _results_distances;
    _table_of_results["times"]    = _results_times;

}
```

```
void step10::CublancPrecondTest::run()
{
    cublanc_Init();

    ::ParameterHandler prm_handler;

    declare(prm_handler);

    prm_handler.read_input (__prm_filename);
    get(prm_handler);

    cublanc_precond_set_parameter(__mu_0, __mu_1, __nu_0, __nu_1);
```

```
QVector<QString> tokens_solver  = QVector<QString>::fromList(__solver_list);
QVector<QString> tokens_iterats = QVector<QString>::fromList(__iterats_list);


prm_handler.print_parameters(std::cout , ::ParameterHandler::Text );

if(__print_eigenvalues){
    printf("Erstellung eines Matlab-Skripts fÃ¼r die Hessenbergmatrix\n");
    FILE* ostream1 = blanc_file_open("output/matlab_skript_gpu.txt","w");
    fprintf(ostream1,"figure;\n");
    fprintf(ostream1,"clear all;\n");
    fclose(ostream1);

    FILE* ostream2 = blanc_file_open("output/matlab_skript_cpu.txt","w");
    fprintf(ostream2,"figure;\n");
    fprintf(ostream2,"clear all;\n");
    fclose(ostream2);

    cublanc_solver_print_eigenvalues_on();
    blanc_solver_print_eigenvalues_on();
}

if(__calc_eigenvalues){
    printf("Erstellung eines Matlab-Skripts fÃ¼r die Eigenwerte\n");
    FILE* ostream_eig = blanc_file_open("output/matlab_skript_gpu_plot.txt","w");
    fprintf(ostream_eig,"figure;\n");
    fprintf(ostream_eig,"clear all;\n");
    fclose(ostream_eig);

    ostream_eig = blanc_file_open("output/matlab_skript_cpu_plot.txt","w");
    fprintf(ostream_eig,"figure;\n");
    fprintf(ostream_eig,"clear all;\n");
    fclose(ostream_eig);

    cublanc_solver_calc_eigenvalues_on();
    blanc_solver_calc_eigenvalues_on();
}


if(__internal)
        setup_and_assemble_cpu();
else
    read_data();

for(int i = 0; i < tokens_solver.size(); i++)
{

    std::cout  << tokens_solver.at(i).toStdString() << std::endl;
    __act_solver = tokens_solver.at(i).toStdString();

    for(unsigned int j = 0; j < tokens_iterats.size(); j++)
    {
        __iterats = tokens_iterats.at(j).toInt();

        std::cout << "No of iteration \t" << __iterats << std::endl;

        bench_test_gpu();
        bench_test_cpu();

        if(__print_eigenvalues){
            FILE* ostream3 = blanc_file_open("output/matlab_skript_cpu.txt","a");
            fprintf(ostream3,"legend('%d');\n",j+1);
            fprintf(ostream3,"M(%d) = getframe;\n",j+1);
            fclose(ostream3);
        }
    }

    * Zwei Leerzeilen
    std::cout << std::endl << std::endl;

    std::ofstream ostream_tex_cnv("output/Tabelle_CNV.tex");
```

```
        _table_of_results["cnv"].write_tex( ostream_tex_cnv, true );
        ostream_tex_cnv.close();

        std::ofstream ostream_tex_dis("output/Tabelle_Distance.tex");
        _table_of_results["distance"].write_tex( ostream_tex_dis, true );
        ostream_tex_dis.close();

        std::ofstream ostream_tex_times("output/Tabelle_Times.tex");
        _table_of_results["times"].write_tex( ostream_tex_times, true );
        ostream_tex_times.close();


    }

    blanc_v_destr(&__rhs_h);
    blanc_v_destr(&__u_h);
    blanc_m_destr(&__A_h);
    blanc_mp_destr(&__A_pattern_h);

    calc_l2_norm();

    cublanc_Shutdown();
}
```

```
void step10::CublancPrecondTest::read_data()
{


#ifdef DEBUG
        std::cout << "Read:\t right side" << std::endl;
#endif

    blanc_v_read(&__rhs_h, __eltnum, __eltdim, rs_file.c_str());

#ifdef DEBUG
       std::cout << "Read:\t original solution" << std::endl;
#endif

    blanc_v_read(&__u_h, __eltnum, __eltdim, u_file.c_str());

#ifdef DEBUG
      std::cout << "Read:\t matrix pattern" << std::endl;
#endif

    blanc_mp_read(&__A_pattern_h, __eltnum, mp_file.c_str() );

#ifdef DEBUG
        std::cout << "Read:\t matrix" << std::endl;
#endif

    blanc_m_read(&__A_h, &__A_pattern_h, __eltdim, m_file.c_str());


}
```

```
void step10::CublancPrecondTest::bench_test_gpu()
{

    _table_of_results["cnv"].add_value("N iter",int(__iterats));
    _table_of_results["distance"].add_value("N iter",int(__iterats));
    _table_of_results["times"].add_value("N iter",int(__iterats));


    cublanc_v_init(&__rhs_d);
    cublanc_v_init(&__u_d);
    cublanc_v_init(&__x_prec_d);
    cublanc_v_init(&__x_none_d);
```

```
    cublanc_Vector  diff_d            = cublanc_V_INIT;

    cublanc_v_constr  ( &diff_d, __eltnum, __eltdim, NULL );
    cublanc_v_entries_set(&diff_d,0.0);

    cublanc_v_constr  ( &__x_prec_d, __eltnum, __eltdim, NULL );
    cublanc_v_entries_set(&__x_prec_d,0.0);

    cublanc_v_constr  ( &__x_none_d, __eltnum, __eltdim, NULL );
    cublanc_v_entries_set(&__x_none_d,0.0);

    cublanc_v_constr  ( &__rhs_d, __eltnum, __eltdim, NULL );
    cublanc_v_constr  ( &__u_d, __eltnum, __eltdim, NULL );

    cublanc_mp_init(&__B_pattern_d);
    cublanc_mp_constr(&__B_pattern_d, __eltnum, NULL);

    cublanc_mpmp_asgn(&__B_pattern_d, &__A_pattern_h);

    cublanc_m_init(&__B_d);
    cublanc_m_constr(&__B_d, &__B_pattern_d, __eltdim, NULL);


    cublanc_Cc cc = cublanc_CC_INIT;
    cublanc_cc_constr ( &cc, NULL );

    cublanc_cc_tol_set(&cc,__tol);

    cublanc_vv_asgn(&__rhs_d, &__rhs_h);
    cublanc_vv_asgn(&__u_d,&__u_h);

    cublanc_mm_asgn(&__B_d,&__A_h);


    QTime solver_time;
    solver_time.start();

    if(__print_eigenvalues || __calc_eigenvalues) cublanc_solver_restart_set(__iterats);

    cublanc_precond_no_of_eigenvalues_set( __no_of_eig );
    cublanc_precond_no_of_niteration_set(__no_of_n_iter);

    cublanc_solver ( cuSolverNameToID[__act_solver],
                     &__B_d,
                     &__x_prec_d,
                     &__rhs_d,
                     &__iterats,
                     &cc,
                     cuPrecondNameToID[__cuPrecond_id.toStdString()],
                     NULL,
                     NULL );

    int solver_t = solver_time.elapsed();

    QString text;
    text = "u_{GPU_{"+__cuPrecond_id+"}} = A^{-1}b /sec";

    _table_of_results["times"].add_value( text.toStdString() , double(0.001*(solver_t)));
    _table_of_results["times"].set_precision(text.toStdString(), 8);

    text = "CNV_{"+__cuPrecond_id+"} GPU";
    _table_of_results["cnv"].add_value(text.toStdString(), double(cublanc_vc_r_get(&cc.cnv)));
    _table_of_results["cnv"].set_scientific(text.toStdString(),true);


    solver_time;
    solver_time.start();

    cublanc_solver ( cuSolverNameToID[__act_solver],
                     &__B_d,
                     &__x_none_d,
```

```
                    &__rhs_d,
                    &__iterats,
                    &cc,
                    cublanc_PRECOND_NONE,
                    NULL,
                    NULL );



    solver_t = solver_time.elapsed();

    _table_of_results["times"].add_value("u_{GPU_{None}} = A^{-1}b /sec", double(0.001*(solver_t)));
    _table_of_results["times"].set_precision("u_{GPU_{None}} = A^{-1}b /sec", 8);

    _table_of_results["cnv"].add_value("CNV_{None} GPU", double(cublanc_vc_r_get(&cc.cnv)));
    _table_of_results["cnv"].set_scientific("CNV_{None} GPU",true);


    cublanc_cc_destr ( &cc );

    cublanc_Real scalar = 0.0;

    cublanc_vv_sub(&diff_d, &__x_prec_d, &__u_d);
    scalar = cublanc_v_n2(&diff_d);


    text = " \| u_{GPU_{"+ __cuPrecond_id +"}} - u_{orig} \|_{2} ";
    _table_of_results["distance"].add_value(text.toStdString(), double(scalar));
    _table_of_results["distance"].set_precision(text.toStdString(), true);

    cublanc_vv_sub(&diff_d, &__x_none_d, &__u_d);
    scalar = cublanc_v_n2(&diff_d);

    _table_of_results["distance"].add_value(" \| u_{GPU_{None}} - u_{orig} \|_{2} ", double(scalar));
    _table_of_results["distance"].set_precision(" \| u_{GPU_{None}} - u_{orig} \|_{2} ", true);


    blanc_v_init(&__x_h_gpu);
    blanc_v_constr  ( &__x_h_gpu, __eltnum, __eltdim, NULL );
    cublanc_vv_asgn(&__x_h_gpu,&__x_prec_d);


    cublanc_m_destr(&__B_d);
    cublanc_mp_destr(&__B_pattern_d);

    cublanc_v_destr( &__rhs_d );
    cublanc_v_destr( &__x_prec_d );
    cublanc_v_destr( &__x_none_d );
    cublanc_v_destr(&diff_d);
    cublanc_v_destr(&__u_d);
}
```

```
void step10::CublancPrecondTest::bench_test_cpu()
{

    blanc_Vector  x_prec_h            = blanc_V_INIT;
    blanc_v_constr  ( &x_prec_h, __eltnum, __eltdim, NULL );
    blanc_v_entries_set(&x_prec_h,0.0);

    blanc_Vector  x_none_h            = blanc_V_INIT;
    blanc_v_constr  ( &x_none_h, __eltnum, __eltdim, NULL );
    blanc_v_entries_set(&x_none_h,0.0);

    blanc_Vector  diff_h            = blanc_V_INIT;
    blanc_v_constr  ( &diff_h, __eltnum, __eltdim, NULL );
    blanc_v_entries_set(&diff_h,0.0);


    blanc_Cc cc = blanc_CC_INIT;
```

```
    blanc_cc_constr ( &cc, NULL );

    blanc_cc_tol_set(&cc,__tol);

    QTime solver_time;
    solver_time.start();

    if(__print_eigenvalues || __calc_eigenvalues)  blanc_solver_restart_set(__iterats);

    blanc_solver ( blancSolverNameToID[__act_solver],
                   &__A_h,
                   &x_prec_h,
                   &__rhs_h,
                   &__iterats,
                   &cc,
                   blanc_PRECOND_ILU,
                   NULL,
                   NULL );


    int solver = solver_time.elapsed();

    _table_of_results["times"].add_value("u_{CPU_{ILU}} = A^{-1}b /sec", double(0.001*(solver)));
    _table_of_results["times"].set_precision("u_{CPU_{ILU}} = A^{-1}b /sec", 8);

    _table_of_results["cnv"].add_value("CNV_{ILU} CPU", double(blanc_vc_r_get(&cc.cnv)));
    _table_of_results["cnv"].set_scientific("CNV_{ILU} CPU",true);


    solver_time;
    solver_time.start();

    blanc_solver ( blancSolverNameToID[__act_solver],
                   &__A_h,
                   &x_none_h,
                   &__rhs_h,
                   &__iterats,
                   &cc,
                   blanc_PRECOND_NONE,
                   NULL,
                   NULL );

    solver = solver_time.elapsed();

    _table_of_results["times"].add_value("u_{CPU_{None}} = A^{-1}b /sec", double(0.001*(solver)));
    _table_of_results["times"].set_precision("u_{CPU_{None}} = A^{-1}b /sec", 8);

    _table_of_results["cnv"].add_value("CNV_{None} CPU", double(blanc_vc_r_get(&cc.cnv)));
    _table_of_results["cnv"].set_scientific("CNV_{None} CPU",true);


    blanc_cc_destr ( &cc );

    blanc_Real scalar = 0.0;

    blanc_vv_sub(&diff_h, &x_prec_h, &__u_h);
    scalar = blanc_v_n2(&diff_h);

    _table_of_results["distance"].add_value(" \| u_{CPU_{ILU}} - u_{orig} \|_{2} ", double(scalar));
    _table_of_results["distance"].set_precision(" \| u_{CPU_{ILU}} - u_{orig} \|_{2} ", true);

    blanc_vv_sub(&diff_h, &x_none_h, &__u_h);
    scalar = blanc_v_n2(&diff_h);

    _table_of_results["distance"].add_value(" \| u_{CPU_{None}} - u_{orig} \|_{2} ", double(scalar));
    _table_of_results["distance"].set_precision(" \| u_{CPU_{None}} - u_{orig} \|_{2} ", true);



    scalar = 0.0;
    blanc_vv_sub(&diff_h, &x_prec_h, &__x_h_gpu);
```

```
    scalar = blanc_v_n2(&diff_h);


    blanc_v_destr(&x_prec_h);
    blanc_v_destr(&x_none_h);
    blanc_v_destr(&diff_h);



}
```

```
void step10::CublancPrecondTest::write_data(cublanc_Vector *vector, char *filename)
{
    FILE* ostream = blanc_file_open(filename,"we");
    cublanc_v_print(vector,ostream);
    fclose(ostream);
}
```

```
void step10::CublancPrecondTest::write_data(blanc_Vector *vector,char *filename)
{

    FILE* ostream = blanc_file_open(filename,"we");
    blanc_v_print(vector,ostream);
    fclose(ostream);
}
```

```
void step10::CublancPrecondTest::declare(::ParameterHandler & prm)
{

    prm.enter_subsection("Dimensions of test problems.");

    prm.declare_entry("iterations","1|2|4|8|16|32|64|128|256|512",::Patterns::Anything(),"this is the number o

    prm.declare_entry("eltnum","80621",::Patterns::Integer(1,1000000),"this is die number of elements") ;
    prm.declare_entry("eltdim","4",::Patterns::Integer(1,4),"this is the dimension of an entry");

    prm.declare_entry("ToeplitzMatrix?",
                      "false",
                      ::Patterns::Bool(),
                      "if you want to run this program with an modified toeplitz matrix, you have to set t

    prm.leave_subsection();

    prm.enter_subsection("Paths of the test problem");

    prm.declare_entry("Matrix",
                      "../../TestExamples/Matrix.e0.dat",
                      ::Patterns::Anything(),
                      "Filename of test matrix. May contains relativ paths");

    prm.declare_entry("MatrixPattern",
                      "../../TestExamples/MPAT0.dat",
                      ::Patterns::Anything(),
                      "Filename of test matrix pattern. May contains relativ paths");

    prm.declare_entry("RightSide",
                      "../../TestExamples/RS_VEC.e0.dat",
                      ::Patterns::Anything(),
                      "Filename of test right side. May contains relativ paths");

    prm.declare_entry("U",
                      "../../TestExamples/U_VEC.e0.dat",
                      ::Patterns::Anything(),
                      "Filename of test u. May contains relativ paths");

    prm.leave_subsection();
```

```
    prm.enter_subsection("Operations");

    prm.declare_entry("Solver-IDs",
                      "CG|CGS|TFQMR|GMRES|BICGSTAB|QMRCGSTAB|GMRES_TEST",
                      ::Patterns::Anything(),
                      "If you want to run several solver on the GPU, you have to set the IDs of the solver
                      " The possible Solvers are: CG, CGS, TFQMR, GMRESM, BICGSTAB, QMRCGSTAB."
                      " You can additionaly choose GMRES_TEST as solver. This solver calculates the eigenv
                      " The delimiter is | between two solvers.");

    prm.declare_entry("cuPrecond-IDs",
                      "AINV",
                      ::Patterns::Anything(),
                      "You can choose between AINV and POL as parallel preconditioner."
                      "This preconditioner are for the GPU");


    prm.declare_entry("Fehlertoleranz",
                      "1e-6",
                      ::Patterns::Double(1e-14,1e+14),
                      "");

    prm.declare_entry("NoOfEigenvalues",
                      "10",
                      ::Patterns::Integer(0,1000000),
                      "Number of calculated Eigenvalues in the GMRES routine");

    prm.declare_entry("nIteration",
                      "10",
                      ::Patterns::Integer(0,1000000),
                      "Number of Iteration in the polynomial preconditioner");

    prm.declare_entry("CalcEigenvalues",
                      "true",
                      ::Patterns::Bool(),
                      "If you want to calculate all eigenvalues of the hessenbergmatrix through a lapack
                      " default: true");

    prm.declare_entry("PrintEigenvalues",
                      "false",
                      ::Patterns::Bool(),
                      "If you want to calculate all eigenvalues through matlab, you have to set this opt
                      " default: false");



    prm.leave_subsection();

    prm.enter_subsection("Parameter of the polynomial preconditioner");

    prm.declare_entry("mu0",
                      "1.0",
                      ::Patterns::Double(-1e+14,1e+14),
                      "");

    prm.declare_entry("mu1",
                      "1.0",
                      ::Patterns::Double(-1e+14,1e+14),
                      "");

    prm.declare_entry("nu0",
                      "1.0",
                      ::Patterns::Double(-1e+14,1e+14),
                      "");

    prm.declare_entry("nu1",
                      "1.0",
                      ::Patterns::Double(-1e+14,1e+14),
                      "");

    prm.leave_subsection();
```

```
}
```

```
void step10::CublancPrecondTest::get(::ParameterHandler & prm)
{
    prm.enter_subsection("Dimensions of test problems.");

    __iterats_list =  QString(prm.get ("iterations").c_str()).split("|") ;

    __eltnum = prm.get_integer ("eltnum") ;
    __eltdim = prm.get_integer ("eltdim") ;

    __internal = prm.get_bool("ToeplitzMatrix?");

    prm.leave_subsection();

    prm.enter_subsection("Paths of the test problem");

    m_file   = prm.get("Matrix");
    mp_file  = prm.get("MatrixPattern");
    rs_file  = prm.get("RightSide");
    u_file   = prm.get("U");

    prm.leave_subsection();

    prm.enter_subsection("Operations");

    __solver_list    = QString(prm.get("Solver-IDs").c_str()).split("|");
    __cuPrecond_id   = QString(prm.get("cuPrecond-IDs").c_str());
    __tol            = prm.get_double("Fehlertoleranz");
    __no_of_eig      = prm.get_integer("NoOfEigenvalues");
    __no_of_n_iter      = prm.get_integer("nIteration");
    __calc_eigenvalues      = prm.get_bool("CalcEigenvalues");
    __print_eigenvalues     = prm.get_bool("PrintEigenvalues");

    prm.leave_subsection();

    prm.enter_subsection("Parameter of the polynomial preconditioner");
    __mu_0 = prm.get_double("mu0");

    __mu_1 = prm.get_double("mu1");

    __nu_0 = prm.get_double("nu0");

    __nu_1 = prm.get_double("nu1");

    prm.leave_subsection();
}
```

```
void step10::CublancPrecondTest::setup_and_assemble_cpu()
{


    blanc_v_init(&__rhs_h);
    blanc_v_constr(&__rhs_h, __eltnum, __eltdim, NULL);
    blanc_v_entries_set(&__rhs_h, 1.0);

    blanc_v_init(&__u_h);
    blanc_v_constr(&__u_h, __eltnum, __eltdim, NULL);
    blanc_v_entries_set(&__u_h, 0.0);

    blanc_mp_init(&__A_pattern_h);
    blanc_m_init(&__A_h);

    blanc_m_constr_grcar(&__A_h, &__A_pattern_h, __eltnum, __eltdim);
}
```

```
void step10::CublancPrecondTest::setup_gpu()
{
    cublanc_v_init(&__x_prec_d);
    cublanc_v_constr  ( &__x_prec_d, __eltnum, __eltdim, NULL );

    cublanc_v_init(&__x_none_d);
    cublanc_v_constr  ( &__x_none_d, __eltnum, __eltdim, NULL );

    cublanc_v_init(&__rhs_d);
    cublanc_v_constr  ( &__rhs_d, __eltnum, __eltdim, NULL );

    cublanc_mp_init(&__B_pattern_d);
    cublanc_mp_constr(&__B_pattern_d, __eltnum, NULL);

    cublanc_mpmp_asgn( &__B_pattern_d, &__A_pattern_h);

    cublanc_m_init(&__B_d);
    cublanc_m_constr(&__B_d, &__B_pattern_d, __eltdim, NULL);
}
```

```
void step10::CublancPrecondTest::destruct_all()
{
    blanc_me_destr (&__A_diag_h);
    blanc_me_destr (&__A_off_diag_h);

    blanc_m_destr  ( &__A_h );
    cublanc_m_destr( &__B_d);

    blanc_mp_destr ( &__A_pattern_h );
    cublanc_mp_destr ( &__B_pattern_d );

    cublanc_v_destr  ( &__x_prec_d );
    cublanc_v_destr  ( &__x_none_d );

    blanc_v_destr(&__rhs_h);
    cublanc_v_destr  ( &__rhs_d );
}
```

```
void step10::CublancPrecondTest::calc_l2_norm()
{
    blanc_v_init(&__u_ralf_h);
    blanc_v_read(&__u_ralf_h, __eltnum, __eltdim, u_file.c_str() );
    blanc_Real u_l2_norm = blanc_v_n2(&__u_ralf_h);
    FILE* ostream = blanc_file_open("u_l2_norm.txt","w");
    fprintf(ostream, "norm_u_ralf = %f", u_l2_norm);
    fclose(ostream);
}

#endif // CUDA_DRIVER_STEP_10_HH
```

```
static
        void
        solver_gmresm_test_exec ( cublanc_Matrix*  const m,
                                  cublanc_Vector*  const x,
                                  const cublanc_Vector*  const b,
                                  unsigned long* const iterats,
                                  cublanc_Cc*    const cc,
                                  const cublanc_Vcont* const omega,
                                  cublanc_Precond_id precond_id,
                                  size_t (* pi) (size_t),
                                  cublanc_Real**       const h,
                                  cublanc_Real*        const y,
                                  cublanc_Real*        const c,
                                  cublanc_Real*        const s,
                                  cublanc_Vector*      const v,
                                  cublanc_Vector*      const temp,
                                  cublanc_Vcont*       const residual ) {
```

```
size_t i          = 0;
size_t j          = 0;
size_t k          = 0;

cublanc_Real         buffer     = 0.0;

cublanc_Boolean  flag = cublanc_B_TRUE;
cublanc_Boolean  scalres =  (cublanc_Boolean) ( !vecres
                                              && (precond_id == cublanc_PRECOND_NONE)
                                              && (cublanc_cc_type_get ( cc ) == cublanc_VT_REAL)
                                              && (cublanc_cc_norm_get ( cc ) == cublanc_N2) );

unsigned long maxiterats = *iterats;
*iterats = 0;

cublanc_vmv_resid ( &v[0], b, m, x );
if ( cc ) {
    if ( scalres ) {
        cublanc_vc_r_set  ( residual, cublanc_v_n2 ( &v[0] ) );
    } if
    else {
        cublanc_vc_v_set ( residual, &v[0] );
    } else
    flag = cublanc_cc_check ( cc,
                              residual,
                              m,
                              x,
                              b,
                              *iterats,
                              restart );
    if ( cublanc_error_get() != cublanc_ES_OK ) {
        cublanc_error_set ( cublanc_ES_STARTCC );
        cublanc_error_printf ( "cublanc_solver",
                               6 );
        return;
    } if
} if

while (   flag
          && (*iterats < maxiterats) ) {

    if ( precond_id != cublanc_PRECOND_NONE ) {
        cublanc_vv_asgn ( temp, &v[0] );

        cublanc_precond ( precond_id, m, &v[0], temp, omega, pi );
        if ( cublanc_error_get() != cublanc_ES_OK ) {
            cublanc_error_set ( cublanc_ES_PRECOND );
            cublanc_error_printf ( "cublanc_solver" );
            return;
        } if
    } if

    y[0] = cublanc_v_n2 ( &v[0] );
    if ( cublanc_r_is_zero ( y[0] ) ) {
        cublanc_error_set ( cublanc_ES_ALGO );
        cublanc_error_printf ( "cublanc_solver" );
        cublanc_error_set ( cublanc_ES_OK );
        return;
    } if

    cublanc_rv_mul ( &v[0], 1.0/y[0], &v[0] );

    i = 0;
    while (    flag
               && (i < restart)
               && (*iterats < maxiterats) ) {

        if ( precond_id != cublanc_PRECOND_NONE ) {
            cublanc_mv_mul ( temp, m, &v[i] );
            cublanc_precond ( precond_id, m, &v[i+1], temp, omega, pi );
            if ( cublanc_error_get() != cublanc_ES_OK ) {
                cublanc_error_set ( cublanc_ES_PRECOND );
                cublanc_error_printf ( "cublanc_solver" );
                return;
            } if
        }  if
        else {

            cublanc_mv_mul ( &v[i+1], m, &v[i] );

        }else
        for ( j=0 ; j<=i ; j++ ) {

            h[j][i] = cublanc_vv_mul ( &v[i+1], &v[j] );

            cublanc_vrv_addmul ( &v[i+1], &v[i+1], -h[j][i], &v[j] );

        } j

        h[i+1][i] = cublanc_v_n2 ( &v[i+1] );

        if ( cublanc_r_is_zero ( h[i+1][i] ) ) {
            cublanc_error_set ( cublanc_ES_ALGO );
            cublanc_error_printf ( "cublanc_solver" );
            cublanc_error_set ( cublanc_ES_OK );
            return;
        } if

        cublanc_rv_mul ( &v[i+1], 1.0/h[i+1][i], &v[i+1] );


        for ( j=0 ; j<i ; j++ ) {
            buffer = h[j][i];
            h[j][i] = c[j] * h[j][i] + s[j] * h[j+1][i];
            h[j+1][i] = - s[j] * buffer + c[j] * h[j+1][i];
        } j

        buffer = cublanc_r_sqrt ( h[i][i] * h[i][i] + h[i+1][i] * h[i+1][i] );
        if ( cublanc_r_is_zero ( buffer ) ) {
            cublanc_error_set ( cublanc_ES_ALGO );
            cublanc_error_printf ( "cublanc_solver" );
            cublanc_error_set ( cublanc_ES_OK );
            return;
        } if


        c[i] = h[i][i] / buffer;
        s[i] = h[i+1][i] / buffer;
        h[i][i] = buffer;
        h[i+1][i] = 0.0;
        y[i+1] = - s[i] * y[i];
        y[i] *= c[i];
        i++;
        (*iterats)++;

        if (    cc
                && scalres
                && i < restart
                && *iterats < maxiterats ) {
            cublanc_vc_r_set  ( residual, cublanc_r_abs (y[i]) );
            flag = cublanc_cc_check ( cc,
                                      residual,
                                      m,
                                      x,
                                      b,
                                      *iterats,
                                      restart );
        } if
    } while
```

```
        for ( j=i ; j-->0 ; ) {
            for ( k=i ; --k>j ; ) {
                y[j] -= h[j][k] * y[k];
            }  k

            y[j] /= h[j][j];

            cublanc_vrv_addmul ( x, x, y[j], &v[j] );
        } j

        cublanc_vmv_resid ( &v[0], b, m, x );

        if (    cc
                && flag ) {
            if ( scalres ) {
                cublanc_vc_r_set  ( residual, cublanc_r_abs (y[i]) );
            }  if
            flag = cublanc_cc_check ( cc,
                                      residual,
                                      m,
                                      x,
                                      b,
                                      *iterats,
                                      restart );
        } if
    } while

    return;
} solver_gmresm_test_exec



#include <iostream>
#include <vector>

#include "QTime"

#include "cuda_driver_step-10.h"


#include "cuda_driver_step-10.hh"

#define USE_DEAL_II
#undef USE_DEAL_II

namespace step10 {
```

```
class TestDriver {

public:

    void run(std::string filename);

};


}
```

```
void step10::TestDriver::run(std::string filename)
{

    step10::CublancPrecondTest testcase(filename);

    testcase.run();

    std::cout << "Fertig." << std::endl;
}
```

```
int main(int argc, char *argv[])
{
    std::string prm_filename( argv[0] );
    prm_filename += ".prm";

    int n_CUDA_devices;

    cudaGetDeviceCount(&n_CUDA_devices);
    std::cout << "available CUDA devices : " << n_CUDA_devices << std::endl;

    using namespace step10;
    int DevNo = 1;

*    for(int DevNo = 0; DevNo < n_CUDA_devices; DevNo++)
    {
        cudaSetDevice(DevNo);
        cublanc_gpu_info(DevNo);

        TestDriver machma;
        machma.run(prm_filename);
    }

}
```