Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences
Fachbereich Informatik
Computer Science Department

Fraunhofer
SCAI

Abschlussarbeit

im Masterstudiengang Informatik

Entwicklung und Implementierung von Partitionierungsstrategien für dünn besetzte Matrizen auf hybriden Systemen mit verteiltem Speicher

von Lauretta Schubert

Erstbetreuer:	**Prof. Dr. Rudolf Berrendorf**
Zweitbetreuer:	**Prof. Dr. Peter Becker**
Eingereicht am:	**25. Februar 2012**

Eidesstattliche Erklärung

Ich versichere an Eides statt, die von mir vorgelegte Arbeit selbstständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als solche kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Lauretta Schubert

Ahornweg 47b, 58566 Kierspe

Zusammenfassung

Die Matrix-Vektor-Multiplikation (SpMV) für dünn besetzte Matrizen stellt für weitreichende wissenschaftliche Anwendungen eine der Kernoperationen des High-Performance-Computing-Bereichs dar. Für die verteilte Berechnung mit immer beliebter werdenden hybriden Rechenclustern kommt dabei die Frage nach einer geeigneten Partitionierungsstrategie für die Verteilung von Daten und Berechnung auf. Diese Arbeit beschäftigt sich damit welchen Einfluss die Struktur der Matrix und die unterschiedlichen Prozessortypen auf die Leistung der SpMV haben und schlägt ein Modell vor, um für diese eine lastbalancierte Verteilung zu erreichen. Wesentliche Bestandteile sind dabei die Laufzeitvorhersage für aktuelle CPUs und GPUs basierend auf einem abgewandelten Roofline-Modell sowie die bewährte Methode der Graph-Partitionierung. Es wird gezeigt, dass für die Laufzeit unter idealen Voraussetzungen eine gute Vorhersage gemacht werden kann und mit der Graph-Partitionierung die Kommunikation für die verteilte Berechnung mit asynchroner Kommunikation hinreichend minimiert wird. Weiterhin wird erläutert, warum für hybride Rechencluster die Abbildung der Rechenlast auf die Anzahl der Matrix-Einträge i. d. R. nicht ausreichend ist um ein balanciertes Gesamtsystem zu erhalten. Diese Problemstellung betreffend werden erste Verbesserungsvorschläge für die Modellierung gegeben.

Abstract

Sparse-Matrix-Vector-Product (SpMV) is among the predominant mathematical operation of High-Performance-Computing. The beginning emergence of hybrid clusters on established computing ressources imposes new questions on partition strategies for distribution of computation and data. This work deals with the impact of the given matrix structure and different processor types on the performance of SpMV and suggests a model for reaching a balanced distribution. The essential component is a prediction of runtime of actual CPUs and GPUs based on the roofline model and the widely used graph partitioning. It will be shown, that under ideal conditions the runtime can be predicted well and that graph partitioning minimizes communication sufficiently for the distributed calculation with asynchronous communication. Further, it will be illustrated, that modelling the calculation load with the number of matrix elements is generally not adequate for reaching an balanced system for hybrid clusters. Concerning this problem first improvements for the model will be given.

Inhaltsverzeichnis

Abkürzungsverzeichnis

BLAS	Basic Linear Algebra Subroutines
Cell B.E.	Cell Broadband Engine
CPU	Central Processing Unit
CSR	Compressed Sparse Row
DPU	Double Precision Unit
ELL	Ellpack
FLOP	Floating Point Operation
FLOP/s	Floating Point Operations per Second
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GPGPU	General Purpose Computation on GPUs
GPU	Graphic Processing Unit
HPC	High-Performance-Computing
KL	Kernighan-Lin
LAMA	Library of Accelerated Math Applications
LAPACK	Linear Algabra Package
MAD	Multiply-Add-Operation
MPI	Message Passing Interface
OI	Operational Intensity
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PCI	Peripheral Component Interconnect

QPL	Quadratic Path Length
RCB	Rekursive Koordinaten Bisektion
RGB	Rekursive Graph-Bisektion
RIB	Rekursive Inertial-Bisektion
SFU	Special Function Unit
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprozessor
SP	Streaming Processor
SpMV	Sparse Matrix-Vektor-Multiplikation
SSE	Streaming SIMD Extension

Symbolverzeichnis

A	Matrix der SpMV
$*$	beliebiger nicht-null-Eintrag
CAS	Größe eines coalesced memory access in Byte (coalesced access size)
$\mathcal{E}$	Kantenmenge eines Graphen
ϵ	Inbalance-Faktor bei der Graph-Partitionierung
$\mathcal{G}$	Graph mit Knotenmenge $\mathcal{V}$ und Kantenmenge $\mathcal{E}$
$\mathcal{H}$	Menge der Hyperkanten in einem Hypergraph
i	Zeilenindex
j	Spaltenindex
m	y-Dimension der Matrix A
n	x-Dimension der Matrix A
$\mathcal{P}$	Partitionsvektor anhand dessen eine Zuordnung von den Knoten eines Graphen zu den Partitionen bzw. Prozessoren gemacht wird
p	Anzahl der Partitionen für die Verteilung bzw. Anzahl der Prozessoren im Ziel-System
RL	Rechenleistung eines Prozessors
$\mathcal{V}$	Knotenmenge eines Graphen
x	Quell-Vektor der SpMV
y	Ziel-Vektor der SpMV

1. Einleitung

Berechnungen für heutige Simulationsanwendungen bedürfen der Lösung großer linearer Gleichungssysteme mit oftmals dünn besetzten Matrizen. Die Struktur der Matrix wird innerhalb der verwendeten Algorithmen bei Matrix-Vektor- und Matrix-Matrix-Operationen ausgenutzt, indem ausschließlich nicht-null-Einträge gespeichert werden und in die Berechnung einfließen. Trotzdem sind die Matrizen oft von solch einem Ausmaß, dass sie nicht auf einem einzelnen Prozessor effizient berechnet werden können. Zum einen können die Daten oft nicht im Arbeitsspeicher des Prozessors gehalten werden, zum anderen dauert die Berechnung einfach zu lang, um aufwendige Berechnungen mit tausenden Iterationen oder auch Echtzeit-Simulationen durchführen zu können.

Entscheidend für die Gesamtleistung eines verteilten Systems ist die Partitionierungsstrategie, die festlegt, wie viele und welche Daten auf welchem Prozessor vorliegen. Insbesondere die Aufgabe, die einzelnen Matrix-Einträge auf die Partitionen aufzuteilen, ist für dünn besetzte Matrizen nicht leicht, da eine unregelmäßige Struktur der Einträge bei einfachen Verteilungsmethoden dem Erreichen eines lastbalancierten Zustands im Wege steht und viele Einträge zwischen den Prozessoren kommuniziert werden müssen.
Stattdessen versucht man mittels Graph-Partitionierung eine gute Lastbalance bei einem kleinen Kommunikationsvolumen zu erreichen. Dafür existieren bereits zahlreiche Ansätze das Kommunikationsvolumen der Anwendung auf den Graphen abzubilden [HK99a, VB05, cA99], um dieses mit der Balancierung der Einträge zu minimieren. Ebenso gibt es verschiedene Software-Bibliotheken wie Metis[Kar] und Scotch[Pel] zur Lösung des Graph-Partitionierungs-Problem.

Die Frage, wie viele Daten jeder Prozessor erhält, stellte sich erst mit der Entwicklung immer heterogenerer Rechenclustern, da sich die Leistung der einzelnen Prozessoren stark voneinander unterscheiden kann und eine gleichmäßige Verteilung der Daten dann i. d. R. keine lastbalancierte Lösung erzielt. Mit der Hinzunahme von Rechenbeschleunigern in solchen Systemen wird ein hybrides System aufgebaut, dass eine noch größere Heterogenität aufweist. Dadurch, dass sie die CPUs als Co-Prozessor unterstützen und nicht als eigenständige Prozessoren auftreten, wird dem System eine weitere Hierachiestufe in der Kommunikation hinzugefügt. Zusätzlich kommt durch die hardwarespezifischen Unterschiede der System-Komponenten zu den generellen aufgabenspezifischen Hürden ein weiterer Faktor für die Bewertung der Hardware dazu.

Dem Problem der Heterogenität der Prozessoren begegnet man in den Graph-Partitionierungs-Bibliotheken dadurch, dass Partitionen mit entsprechenden Gewichten erstellt werden.

Die Angabe dieser ist aber meist dem Anwender überlassen. Solange es sich um einfache heterogene Cluster handelt, reicht eine relative Aussage zu der Leistungsfähigkeit verschiedener Prozessoren aus, um eine geeignete Abschätzung für die Gewichtung zu liefern, da die Algorithmen sich auf gleiche Art auf den Central Processing Units (CPUs) verhalten werden. Handelt es sich aber um ein hybrides System, gilt dies bereits nicht mehr. Aufgrund der hardwarespezifischen Unterschiede ist sowohl die Durchführung der Berechnung als auch der Zugriff auf die Daten verschieden, sodass sich von Aufgabe zu Aufgabe das Verhältnis zwischen CPUs und Rechenbeschleunigern verändern kann. Grund dafür ist u. A., dass viele dieser Rechenbeschleuniger das Single Instruction Multiple Data (SIMD)-Prinzip verfolgen, um ihre hohe Leistung überhaupt erreichen zu können, dies aber für verschiedene Berechnungen unterschiedlich gut umgesetzt wird.

Um den gewachsenen Anforderungen bei der Partitionierungen für dünn besetzte Matrizen großer Gleichungssysteme für hybride System gerecht zu werden, gilt es dabei die starke Heterogenität des Clusters beachten. In dieser Arbeit soll für Kombinationen aus Multicore-CPUs und Graphic Processing Units (GPUs), als ein Beispiel für solche Rechenbeschleuniger, ein geeigneter Ansatz zur Lastbalancierung gefunden werden. Wesentlicher Bestandteil wird dazu die Modellierung der Leistungsfähigkeit der Prozessoren sein, sodass anhand dieser eine adäquate Gewichtung für die Graph-Partitionierung vorgenommen werden kann. Der Ansatz beschränkt sich dabei darauf die Sparse Matrix-Vektor-Multiplikation (SpMV) zu optimieren. Das Modell soll dazu in der Library of Accelerated Math Applications (LAMA) integriert werden.

Der Rest dieser Arbeit strukturiert sich wie folgt:

In Kapitel 2 wird die Matrix-Partitionierung dünn besetzter Matrizen und der hierfür weitläufig eingesetzte Ansatz der Graph-Partitionierung behandelt. In Kapitel 3 werden die wesentlichen hardwarespezifischen Unterschiede von aktuellen CPUs und GPUs erläutert, um nach zusätzlichen Implementierungs-Details von LAMA (Kapitel 4) daraus in Kapitel 5 ein Modell zur Leistungsbewertung ableiten zu können. Weiterhin setzt sich Kapitel 5 noch mit der dynamischen Balancierung auseinander. Kapitel 6 erklärt, wie sich das Modell in LAMA einfügt. Zum Schluss (Kapitel 7) folgt eine ausführliche Auswertung der einzelnen Strategiebausteine sowie eine abschließende Bewertung des Gesamtansatzes. Kapitel 8 fasst die Ergebnisse der Arbeit noch einmal zusammen und stellt weitere Verbesserungen in Aussicht.

2. Matrix-Partitionierung

Die Partionierung von Matrizen wurde mit dem Aufkommen von Parallelrechnern eine wichtige Aufgabe zur Durchführung von großen Rechnungen aus dem Bereich der linearen Algebra. Die Notwendigkeit hoch paralleler Cluster wird aus den wachsenden Datenmengen, mit denen Programme und Bibliotheken umgehen müssen, deutlich. Eine Aufteilung der Berechnung schafft nicht nur Parallelität in der Ausführung sondern auch eine Aufteilung der Daten, sodass größere gleichzeitig benötigte Datenmengen im Arbeitsspeicher des Systems vorliegen können. Dies macht die Berechnung für manche Aufgaben erst möglich, da die Daten nicht in Teilen geladen und verarbeitet werden müssen.

Die Entwicklung heutiger Rechencluster geht zudem in die Richtung immer heterogenere Prozessoren zu verbinden, was zur Folge hat, dass die Partitionierung von Matrizen um einen weiteren Komplexitätsfaktor erweitert wird. Die Last eines zugewiesenen Teils sollte für eine gute Lastbalance der Leistung des Prozessors angemessen sein. Außerdem gilt es für homogene wie heterogene Systeme möglichst wenig Datenabhängigkeiten über Prozessorgrenzen hinweg zu verteilen, weil dadurch Kommunikation zwischen diesen notwendig wird. Erschwerend kommt hinzu, dass die Matrizen dünn besetzt sind. Dies begünstigt nicht die Qualität von Partitionierungen, die ohne spezielle Berücksichtigung der Struktur eine Aufteilung vornehmen. Die Intention der Partitionierung lässt sich demnach als Lastbalancierung zwischen den beteiligten Prozessoren bei gleichzeitiger Minimierung zu kommunizierender Datenmengen formulieren.

Diese Arbeit wird sich mit der Partitionierung dünn besetzter Matrizen auf ein hybrides Rechencluster auseinandersetzen. Das Ziel der Partitionierung ist es für die gegebene Matrix und die vorliegende Hardware einen lastbalancierten Zustand zu erreichen. Das Problem bei dieser Aufgabe wird vor allem die unregelmäßige Matrixstruktur sowie das stark heterogene Zielsystem aus CPUs und GPUs sein.

In diesem ersten Kapitel soll die grundlegende Problemstellung, die hinter der Matrix-Partionierung steht, umrissen werden. Dazu wird sich zuerst mit dem Thema der Lastbalancierung im Allgemeinen, danach für die Partitionierung von Matrizen auseinandergesetzt. Um dem Faktor der Datenabhängigkeiten beizukommen wird die Graph-Partitionierung im Hinblick auf die Matrix-Partionierung näher beleuchtet. Abschließend werden gängige Bibliotheken zur Graph-Partionierung vorgestellt.

2.1. Lastbalancierung

Ein Ziel der Matrix-Partionierung ist die Lastbalance. Als Last versteht man in diesem Kontext die Menge an Arbeit, die einem Prozessor zugewiesen wird. Die Arbeit in der verteilten Berechnung von Matrix-Vektor- und Matrix-Matrix-Operationen besteht dabei zum einem in den durchzuführenden Rechenoperationen sowie in der Kommunikation von Daten auf andere Prozessoren. Die Last wird für beide Arbeiten über die Anzahl der beteiligten Datenelemente beschrieben. Unter einem lastbalancierten System versteht man eines, in dem jeder Prozessor gleichermaßen ausgelastet ist. Für homogene Systeme bedeutet dies, dass jedem Prozessor die gleiche Last (in Form von Datenelementen) zugewiesen werden soll. Da alle Prozessoren dieselbe Leistung haben, sollten sie in etwa gleichermaßen ausgelastet sein und die Berechnung zeitgleich abschließen. Für heterogene Systeme gilt es eine Abschätzung der verhältnismaßigen Leistungsfähigkeit zu finden, sodass auch hier alle Prozessoren die Ausführung zur gleichen Zeit beenden.
Die Leistungsfähigkeit eines Prozessors wird i. d. R. in Floating Point Operations per Second (FLOP/s) gemessen. Die maximale Leistung, die sich aus Taktrate und maximal ausführbaren Operationen pro Takt ergibt, wird aber meist nicht erreicht, da der Prozessor aufgrund der Speicheranbindung oft garnicht so schnell mit Daten bedient wird und diese wegschreiben kann, wie sie verarbeitet werden. Als Maßstab für die tatsächliche Leistungsfähigkeit eines Supercomputers gilt deshalb sein Abschneiden im LINPACK-Benchmark [Don88]. Letztenendes wird die Leistung für jeden Algorithmus aufgrund von verschiedenen Zugriffsmustern unterschiedlich ausfallen und durch eine Vielzahl an Variablen, wie z. B. den Cache-Größen und deren Zugriffszeiten, beeinflusst. Somit kommt es auf eine genaue Kenntniss des Algorithmus, prozessorspezifischer Kennzahlen, sowie Implementierungsdetails des Prozessors an, um die Leistung eines Prozessors für einen Algorithmus vorherzusagen. Weil dies oftmals schwierig ist statisch vorherzusagen wird dem Partitionierungs-Problem oft über dynamische oder heuristische Balancierungsalgorithmen beigekommen.
Lastbalancierungsalgorithmen unterteilen sich in erster Linie in *statische* und *dynamische*. Streng genommen partitionieren statische Algorithmen zur Compile-Zeit anhand von a priori Wissen über das System und der zu lösenden Aufgabe, während dynamische Algorithmen die Partionierung zur Laufzeit vornehmen und dabei aktuelle Werte in die Bewertung einfließen lassen können. Damit hat die statische Methode den Vorteil keinen Laufzeit-Overhead zu erzeugen. Sie ist dann reizvoll, wenn Berechnungs- und Kommunikationszeiten gut vorausgesagt werden können. Manchmal ist sie auch die einzige Lösung, wenn Aufgaben zu groß sind, um effizient auf andere Prozessoren verschoben werden zu können [XL97]. Da Wissen über die zu lösende Aufgabe für die Aufteilung nötig ist und dieses sich oft erst zur Laufzeit festlegt, ist aber auch eine weiter gefasste Definition möglich, die die Analyse der Aufgabe zur Laufzeit einmalig für statische Algorithmen zulässt. Dagegen arbeiten dynamische Algorithmen mit Hilfe von Laufzeitinformationen wie Berechnungs-, Kommunikations- und Wartezeiten. Sie erkennen Lastungleichgewichte

und versuchen diese iterativ durch das Verschieben von Last auszugleichen. Wichtigstes Instrument der dynamischen Lastbalancierung ist das Monitoring[1] des Systems, um mit den gesammelten Daten eine Konfiguration mit verbessertem Zustand vorherzusagen.
Ein *heuristischer* Ansatz zur Lastbalancierung ist einer, der nicht versucht die genaue Last vorherzusagen, sondern aufgrund von Vereinfachungen eine möglichst gute Näherungslösung zu finden. Sie werden häufig verwendet, wenn keine exakte Lösung nötig ist oder die Berechnung dieser im Vergleich zum resultierenden Vorteil zu aufwendig ist.
Weiter unterscheidet man Lastbalancierungsalgorithmen in:

- kooperative und nicht-kooperative
- adaptive und nicht-adaptive
- periodische und zeitgesteuerte
- Single- und Multilevel-Algorithmen

Unter der *Kooperativität* eines Algorithmus versteht man die Eigenschaft, dass Entscheidungen bezüglich der Lastverteilung aufgrund des Gesamtsystems entschieden werden bzw. aufgrund einer einzelnen Komponente. Im Zusammenhang mit dynamischen Algorithmen unterscheidet man zwischen periodischen Algorithmen, die beispielsweise alle zehn Iterationen einer Berechnung die Last analysieren und ausgleichen und solchen, die über ein Ereignis dazu angestoßen werden. Mit *Adaption* ist die Anpassungsfähigkeit an sich ändernde Einflüsse gemeint. Wichtig ist dies beispielsweise für Gitterlöser, die auf verschiedenen Gittern arbeiten auf denen neue Annahmen getroffen werden müssen. *Multilevel*-Algorithmen lösen im Gegensatz zu *Singlelevel*-Algorithmen die Partionierungsaufgabe nicht direkt, sondern iterativ auf verschiedenen Level, indem sie die Aufgabe schrittweise vergröbern, auf dem untersten Level lösen und die Lösung schrittweise auf die höheren Level übertragen.

Welche Art von Lastbalancierung gewählt werden sollte, hängt stark von der Aufgabe und äußeren Einflüssen ab. Kann bereits mit statischen Verfahren eine gute Lastbalance erzielt werden, ist sie einer rein dynamischen Balancierung vorzuziehen, da dynamische Verfahren einen größeren Overhead für das Monitoring und das wiederholte Verschieben der Last bedeuten. Jedoch ist die Last in vielen Fällen vorab schwer einzuschätzen, was oft in ungünstigen Aufteilungen und unbalancierten Zuständen endet, die nur durch dynamische Verfahren optimiert werden können. Oft ist daher eine Kombination aus beiden Methoden eine gute und „kostengünstige“ Wahl.

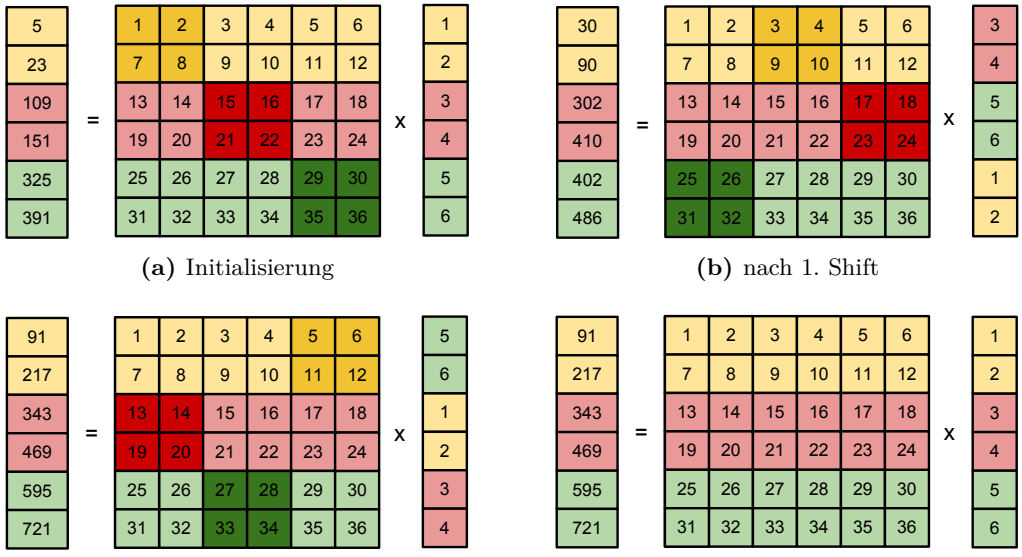
Bei der Partitionierung dünn besetzter Matrizen ist die Struktur der Matrix ausschlaggebend für die Qualität der anzugebenden Verteilung. Deswegen muss sie für die Partionierung analysiert werden. Dafür wird die Matrix üblicherweise auf einen Graphen abgebildet sowie das Problem der Matrix-Partionierung auf das Problem der Graph-Partionierung. Bestehende Algorithmen, die das Graph-Partitionierungs-Problem in akzeptabler Laufzeit lösen, sind von heuristischer Natur und bieten neben statischen Vorhersagen auch Möglichkeiten der dynamischen Anwendung zur Verbesserung bestehender Partitionen.

[1] Monitoring ist die systematische Beobachtung von Prozessen mit dem Ziel, diese optimieren. Dazu werden verschiedenste Daten erfasst und analysiert. Für die Lastbalancierung wären das beispielsweise einzelne Ausführzeiten und resultierende Ungleichgewichte, die zu Handlungsbedarf aufrufen.

Nachfolgend soll an die Partitionierung von Matrizen anhand der Matrix-Vektor-Multiplikation ($y = A \cdot b$) herangeführt werden. Sie gehört zu den Basis-Operationen der linearen Algebra und bleibt auch in den weiteren Betrachtungen der fokussierte Untersuchungsgegenstand. Zu Beginn wird von einer dicht besetzten Matrix ausgegangen, um den Unterschied von zeilen- zur spaltenweisen Partitionierung zu erklären. Danach wird das Modell auf dünn besetzte Matrizen erweitert. An dieser Stelle wird lediglich die Struktur der Matrix berücksichtigt, nicht aber die des Zielsystems. Vielmehr ist dieses der Einfachheit halber als homogen angenommen und erwartet daher für gleichgroße Partitionen die gleiche Last. Der Schritt zu heterogenen Systemen wird erst in Kapitel 5 mit der Modell-Entwicklung gegangen.

2.2. Partitionierung dicht besetzter Matrizen

Soll eine Matrix der Größe $m \times n$ für eine Matrix-Vektor-Multiplikation über ein System mit p Prozessoren verteilt werden, so wird man zu der Lösung kommen, die Matrix entweder zeilen- oder spaltenweise und ebenso den Vektor x über die p Prozessoren aufzuteilen. Wir gehen davon aus, dass eine komplette Speicherung des Vektors x aufgrund der Größe nicht möglich ist.



(a) Initialisierung **(b)** nach 1. Shift

(c) nach 2. Shift **(d)** nach 3. Shift

Abbildung 2.1.: Stadien bei der Matrix-Vektor-Multiplikation einer zeilenweise partitionierten dicht besetzen 6×6 - Matrix für 3 Prozessoren; in (a)-(c) werden die Matrixteile hervorgehoben, mit denen in diesem Stadium gerechnet wird; (d) zeigt die Ausgangslage des Quell-Vektors x sowie das Resultat

Bei der zeilenweise Partitionierung (s. Abbildung 2.1) wird der Vektor dann ähnlich der Matrix-Zeilen auf die Prozessoren aufgeteilt. Dadurch kann ein Teil der Berechnung durchgeführt werden und als Teilergebnis gespeichert werden. Schrittweise werden die Vektorteile solange an den nächsten Nachbarn geschickt und die neue Teilberechnung zu dem Ergebnis addiert, bis der Vektor wieder auf dem ersten Prozessor angekommen ist. Bei

einer gleichmäßigen Aufteilung der Zeilen $\frac{n}{p}$ hat man eine optimale Lastbalance erreicht. Die Anzahl der Kommunikationen pro Prozessor liegt bei p Kommunikationsschritten, in denen jeweils der vorliegende Vektorteil der Länge $\frac{n}{p}$ verschickt wird. Wird der Ausgangsvektor zwischengespeichert (zusätzlicher Speicherbedarf) ist der letzte Kommunikationsschritt überflüssig.

Teilt man stattdessen die Matrix spaltenweise über die Prozessoren auf (s. Abbildung 2.2) und den Vektor x in entsprechend gleichgroße Teile, so kann auf jedem Prozessor für jede Zeile das Teilergebnis über die lokalen Spalten gebildet werden. Diese Teilergebnisse müssen dann zu dem Prozessor, auf dem das Ergebnis vorliegen soll, geschickt und aufaddiert werden. Insgesamt verschickt jeder Prozessor $p-1$ Teilergebnisse der Länge $\frac{n}{p}$. Der Vektor y muss aber repliziert vorliegen.

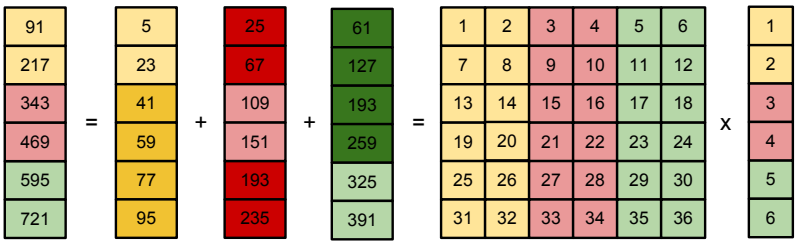
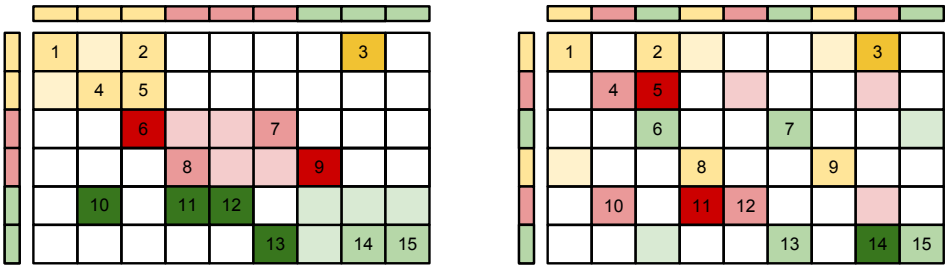


Abbildung 2.2.: Spaltenweise Partitionierung einer dicht besetzten 6×6-Matrix für 3 Prozessoren bei der Matrix-Vektor-Multiplikation; in den Teilergebnissen sind jeweils die Teile hervorgehoben, die zur Berechnung des Gesamtergebnis kommuniziert werden müssen

Mit der Zwischenspeicherung des Ausgangsvektor für die zeilenweise Partitionierung ist das Kommunikationsvolumen für beide Methoden dasselbe, allerdings teilt es sich für die spaltenweise Partitionierung in mehrere Nachrichten auf. Die Pufferung des Vektors benötigt zusätzlicher Speicherplatz für $\frac{n}{p}$ Elemente. Für die spaltenweise Partitionierung wird dagegen ein vollständiger Vektor (n Elemente) gebraucht. Eine Entscheidung zwischen zeilen- und spaltenweiser Partionierung ist daher aufgrund der Anzahl der zu verschickenden Nachrichten und der Menge an zusätzlich nötigen Speicherplatzes abzuwägen.

2.3. Partitionierung dünn besetzter Matrizen

Bei der Partitionierung dünn besetzter Matrizen ist das Finden einer lastbalancierten Verteilung wegen der unregelmäßigen Anzahl nicht-null Einträge pro Zeile (und Spalte) wesentlich komplizierter. An dem Beispiel in Abbildung 2.3 sieht man bereits die Bedeutung der Zuordnung der Zeilen zu den Prozessoren. Im Fall (a), der blockweisen Partitionierung, werden den drei Prozessoren vier, fünf und sechs Matrix-Einträge zugeordnet, während im Fall (b), der zyklischen Partitionierung, ein Lastgleichgewicht von fünf Einträgen vorherrscht.



(a) Blockweise Partitionierung **(b)** Zyklische Partitionierung

Abbildung 2.3.: Bedeutung der Zuordnung der dünn besetzten Zeilen und Spalten zu den Prozessoren bzgl. des Lastausgleichs bei einer SpMV für zwei einfache Partitionierungsmethoden; für die Berechnung müssen die Vektor-Einträge, die den hervorgehobenen Matrix-Einträgen zuzuordnen sind, kommuniziert werden

Außer dem Lastausgleich für die Berechnung sollte das Kommunikationsvolumen beachtet werden (s. hervorgehobene Einträge in Abbildung 2.3). Für jede Berechnung innerhalb einer Zeile i werden nur die Einträge j des Vektors x benötigt für die der Eintrag a_{ij} der Matrix A nicht null ist. Somit sollten für eine Zeile auch nur diese Einträge ausgetauscht und nicht, wie im voll besetzten Fall, der komplette Vektor durchgereicht werden. Das Kommunikationsvolumen und die Anzahl der zu verschickenden Nachrichten hängt damit maßgeblich von der Struktur der Matrix ab. Dabei bleibt der generelle Ansatz zur zeilen- oder spaltenweise Aufteilung sowie die daraus resultierenden Kommunikationsmuster bestehen. Für eine geblockte Aufteilung bei der SpMV ($y = A \cdot b$) ergeben sich daraus folgende Phasen, wie sie Vastenhouw und Bisseling in [VB05] darlegen:

1. Jeder Prozessor sendet seine Komponenten b_j zu den Prozessoren, die mit einem nicht-null-Eintrag a_{ij} in der Spalte j zu rechnen haben.
2. Jeder Prozessor berechnet das Produkt $a_{ij} \cdot b_j$ und addiert das Ergebnis zum dem lokalen Ergebnis der Zeile i. Dies führt zu einer Reihe Teilergebnisse y_{is} mehrerer Prozesse s, wobei $0 \leq s \leq p$.
3. Jeder Prozessor sendet sein Teilergebnis y_{is} zu dem Prozessor, der y_i berechnet.
4. Jeder Prozessor addiert die empfangenen Teilergebnisse für seine zuständigen Komponenten y_i.

Die zwei Kommunikationsphasen (1. + 3.) vor den Berechnungsphasen (2. + 4.) rühren von der Abhängigkeit zwischen den Komponenten b_j des Quell-Vektors und der nicht-null-Einträge a_{ij} sowie der Abhängigkeit der Einträge a_{ij} und der Komponenten y_i des Ziel-Vektors her. Wie man bereits im dicht besetzten Fall gesehen hat entfallen bei einer rein zeilenweise Aufteilung die beiden letzten Phasen, hingegen bei einer rein spaltenweisen Aufteilung die ersten beiden, weil alle Daten bereits auf dem zuständigen Prozessor vorhanden sind. Bei einer geblockten Aufteilung (Aufteilung nach Zeilen und Spalten) bedarf es i. d. R. entsprechend aller vier beschriebenen Phasen.
Wie die geblockte Aufteilung für die Beispiel-Matrix aus Abbildung 2.3 aussehen könnte,

zeigt Abbildung 2.4. Wieder hat jede Partition die gleiche Anzahl nicht-null-Einträge erhalten, es müssen jedoch nur noch drei Einträge aus dem Quell-Vektor x kommuniziert werden. Ebenso entfällt in diesem speziellen Fall die 2. Kommunikationsphase, da jede Berechnung für eine Zeile auf dem Prozessor durchgeführt wurde, der den Ziel-Vektor y speichert.

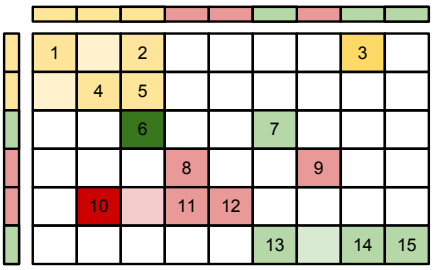


Abbildung 2.4.: Geblockte Partitionierung; unabhängige Zuordnung von Zeilen und Spalten zu den Prozessoren; für die hervorgehobenen Matrix-Einträge müssen die entsprechenden Vektor-Einträge vor der Berechnung kommuniziert werden (1.); in diesem Fall tritt keine Kommunikation von Teilergebnissen (3.) auf

2.4. Graph-Partitionierung

In diesem Kapitel soll der Übergang zwischen Matrix- und Graph-Partitionierung näher erläutert werden. Dazu wird zu Beginn die Graph-Notation eingeführt, danach Modelle zur Darstellung dünn besetzter Matrizen als Graph beschrieben, bevor zum Schluss das allgemeine Graph-Partitionierungs-Problem formuliert wird und die Ideen gängiger Partitionierungs-Algorithmen vorgestellt werden.

2.4.1. Beschreibung eines Graphen

Ein Graph $\mathcal{G}(\mathcal{V},\mathcal{E})$ wird durch eine Menge von Knoten $\mathcal{V}$ sowie einer zugehörigen Menge Kanten $\mathcal{E}$, die jeweils zwei Knoten aus $\mathcal{V}$ verbinden, beschrieben. Formal lässt sich die Knotenmenge $\mathcal{V}$ durch $\mathcal{V} = \{v_i | i = 1, \ldots, n\}$ mit $n \in \mathbb{N}$ für die Anzahl der beinhalteten Knoten ausdrücken ($|\mathcal{V}| = n$). Für die Kantenmenge $\mathcal{E}$ gilt: $\mathcal{E} = \{e_{ij} = (i,j) | \exists$ Kante zwischen v_i und $v_j\}$. Sowohl Knoten als auch Kanten eines Graphen können über Gewichte verfügen ($\mathcal{W}_\mathcal{V} = \{w_v(v_i) \in \mathbb{N} | v_i \in \mathcal{V}\}$, $\mathcal{W}_\mathcal{E} = \{w_e(e_{ij}) \in \mathbb{N} | e_{ij} \in \mathcal{E}\}$). Man spricht dann von einem gewichteten Graphen. Bei einem ungewichteten Graphen geht man davon aus, dass alle Gewichte *eins* sind. Weiterhin unterscheidet man bei einem Graphen zwischen gerichteten und ungerichteten Graphen. In einem gericheten Graphen besteht bei der Kante e_{ij} eine Verbindung (bspw. Kommunikation) von v_i nach v_j aber nicht explizit auch umgekehrt. Bei einem ungerichteten Graphen werden dagegen durch eine Kante e_{ij} beide Richtungen ausgedrückt. [Saa03]

2.4.2. Graphenmodelle für dünn besetzte Matrizen

Mit Hilfe von Graphen können Abhängigkeiten (Kanten $\mathcal{E}$) zwischen einzelnen Objekten (Knoten $\mathcal{V}$) beschrieben werden. Für die Partitionierung soll der Rechenaufwand und nötige Kommunikation zwischen den Partitionen modelliert werden. Der Rechenaufwand entspricht dabei der Anzahl der zugeteilten Einträge, die Kommunikation beschreibt sich über die Abhängigkeiten zum Quell- und Ziel-Vektor (x und y). Kommunikation ist immer dann von Nöten, wenn für die einzelne Berechnung $y_i += a_{ij} \cdot x_j$ die Einträge y_i, a_{ij} und x_j nicht auf demselben Prozessor vorliegen. Zur Modellierung dieser Abhängigkeiten gibt es mittlerweile verschiedenste Modelle, die versuchen so einfach wie möglich verschiedene Matrixstrukturen zu repräsentieren. Je einfacher - soll heißen mit so wenigen Knoten und Kanten wie möglich - eine Matrix modelliert werden kann, desto geringer ist die Laufzeit für die Graph-Partitionierung. Daneben unterscheiden sich die Modelle darin wie genau das tatsächliche Kommunikationsvolumen abgebildet wird.

Modell des ungerichteten Graphen Dieses Modell ist das wahrscheinlich einfachste und wird daher gern in diesem Kontext angeführt [Bom07]. Es setzt eine symmetrische (und damit auch quadratische) Matrix voraus. Dabei ist symmetrisch in diesem Fall nicht im mathematischen Sinne gemeint, wobei die Koeffizienten der Matrix spiegelsymmetrisch gegenüber der Hauptdiagonalen angeordnet sind ($a_{ij} = a_{ji}$), sondern als eine strukturelle Symmetrie. Darunter versteht man die Symmetrie der Platzierung von nicht-null Einträgen: $\forall a_{ij}$ gilt, wenn $a_{ij} \neq 0$ dann ist auch $a_{ji} \neq 0$.
Für die Partitionierung der Matrix kann diese daher rein auf ihre Struktur reduziert werden, indem ihre nicht-null-Einträge als einheitlicher nicht-null-Eintrag ($*$) betrachtet werden. Durch diese Annahme ist die Matrix auch wieder im mathematischen Sinn symmetrisch. Dies bedeutet, dass eine solche Matrix in diesem Sinne identisch zu ihrer Transponierten ist ($A = A^T$) und damit die Rollen von Zeilen und Spalten austauschbar sind. Dies führt zur ersten Vereinfachung des Modells: Zeile und Spalte i werden zusammen über einen Knoten i modelliert. Zwischen zwei Knoten i und j existiert eine Kante e, wenn a_{ij} ein nicht-null-Eintrag ($a_{ij} \neq 0$) ist. Dies beschreibt die Abhängigkeit von a_{ij} zu x_j bei der Einzelberechnung $y_i += a_{ij} \cdot x_j$. Die Abhängigkeit zu y_i wird nicht mit modelliert, da man von einer zeilenweisen Partitionierung ausgeht, bei der die Zeile i der Matrix A und die Vektoren x und y auf ein und demselben Prozessor vorliegen. Somit kann es keine Kommunikation bzgl. y geben. Aus dem gleichen Grund muss die Berechnung von $y_i += a_{ii} \cdot x_i$ nicht modelliert werden. Damit stimmt die Beschreibung des Graphen im Compressed Sparse Row (CSR)-Format mit der Beschreibung einer CSR-Matrix bis auf die Diagonal-Einträge überein[2]. Das Gewicht eines Knoten wird zudem über die Anzahl der Einträge pro Zeile beschrieben und definiert damit die Last, die einer Partition mit der Zuteilung der Zeile i zukommt. Eine Veranschaulichung des Modells für eine symmetrische Matrix und ihre Graphdarstellung befindet sich in Abbildung 2.5.

[2]Eine Erläuterung zum CSR-Format für die Speicherung dünn besetzter Matrizen befindet sich in 4.2.3.

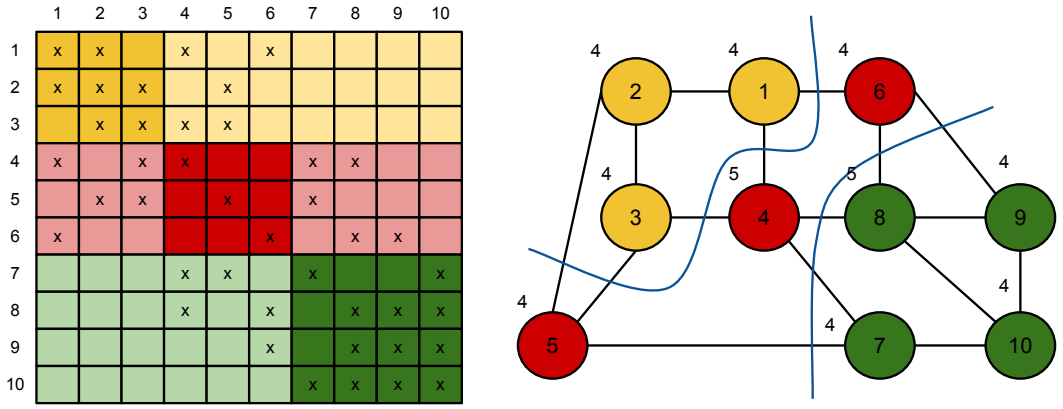


Abbildung 2.5.: Symmetrische 10×10-Matrix und deren Graphdarstellung blockweise aufgeteilt in 3 Partitionen. Die blauen Linien zeigen den Schnitt zwischen den einzelnen Partitionen. Der damit beschriebene Kantenschnitt besteht aus 10 Kanten, während das aufkommende Kommunikationsvolumen bei 20 Elementen in 4 Nachrichten liegt. Die Knotengewichte sind an den Knoten angegeben.

Bipartite Graph-Modell Das bipartite Graph-Modell [HK99b] versucht die Nachteile des Modells des ungerichteten Graphen zu beheben. Bei der Entwicklung des Modells stand im Vordergrund, dass auch unsymmetrische oder nicht quadratische Matrizen behandelt werden können und das Kommunikationsvolumen besser beschrieben wird.

Ein bipartiter Graph, $\mathcal{G} = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E})$, wird über zwei disjunkte Knotenmengen $\mathcal{V}_1$ und $\mathcal{V}_2$ und die Kantenmenge $\mathcal{E} \subset \mathcal{V}_1 \times \mathcal{V}_2$ beschrieben. Damit bestehen keine Kanten innerhalb einer Knotenmenge, sondern nur zwischen den beiden. $\mathcal{V}_1$ wird auf die Menge der Zeilen abgebildet und $\mathcal{V}_2$ auf die Menge der Spalten; es besteht genau dann eine Kante zwischen zwei Knoten v_i und v_j, wenn in der Matrix an der Position i, j ($a_{i,j}$) ein nicht-null-Eintrag vorhanden ist. Ein Beispiel findet sich in Abbildung 2.6.

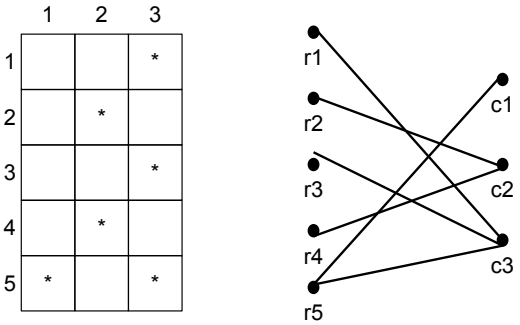


Abbildung 2.6.: Rechteckige Matrix und deren Darstellung als bipartiter Graph; man beachte, dass nur Kanten zwischen den Mengen r und c bestehen aber nicht untereinander

Hypergraph-Modell Das Hypergraph-Modell wurde von Çatalyürek und Aykanat [cA99] entwickelt, um das Kommunikationsvolumen besser als im Standart-Graphmodell zu beschreiben. Da die Anzahl der geschnittenen Kanten im Graphmodell nicht der Anzahl der zu kommunizierenden Elementen entspricht (z. B. bestehen in Abbildung 2.5 zwei Kanten von Knoten 4 (Partition 2) in Partition 1) wird das Kommunikationsvolumen durch den

Kantenschnitt oft überschätzt. Die zu kommunizierenden Elemente werden durch die Anzahl der verschiedenen Partitionen, zu denen ein Knoten eine Kante besitzt, beschrieben. Hendrickson und Kolda bezeichnen dies in [HK99a] als *Grenzschnitt*, definiert durch $\sum_i b_i$, wobei b_i die Anzahl der Partitionen zu denen ein Knoten Kanten besitzt darstellt.

Ein Hypergraph $\mathcal{G}(\mathcal{V}, \mathcal{H})$ zeichnet sich über seine Knoten-($\mathcal{V}$) und Hyperkantenmengen($\mathcal{H}$) aus. Eine Hyperkante kann jede Anzahl an Knoten miteinander verbinden. Die Kante wird über die Menge an Knoten, die sie verbindet, beschrieben. Damit ist ein allgemeiner Graph ein spezieller Hypergraph, bei dem eine Kante nur genau zwei Knoten miteinander verbindet. In Abbildung 2.7 befindet sich die Hypergraph-Darstellung der rechteckigen Matrix aus Abbildung 2.6 in zwei Darstellungsformen. Die zweite Form zeigt die Ähnlichkeit des Hypergraph-Modells zum bipartiten Graph-Modell (ohne Umsortierung der Knoten und Hyperkanten zeigt sich dasselbe Bild der Datenabhängigkeiten).

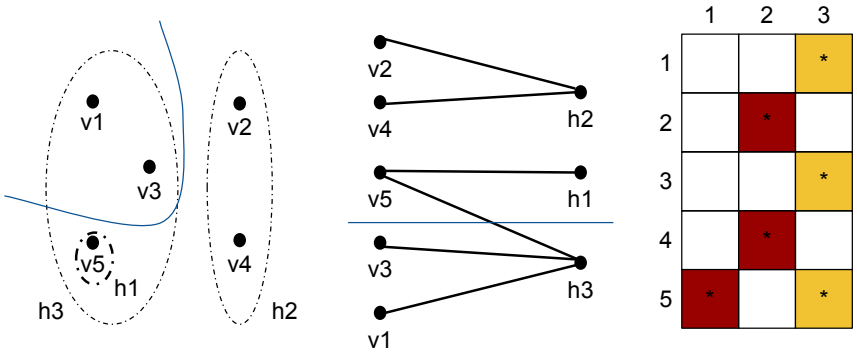


Abbildung 2.7.: Zwei Hypergraph-Darstellungen der Matrix aus Abb. 2.6 sowie deren Partitionierung (die blauen Linien definieren den Schnitt der Hypergraph-Partitionierung); links: die Darstellung zeigt den Hypergraphen als Mengen; mittig: die Darstellung zeigt den Hypergraphen als verbundene Knoten; rechts: die Matrix eingefärbt entsprechend ihrer Partitionierung; der Grenzschnitt liegt bei 1, was der Kommunikation in der letzten Zeile entspricht

Indem sowohl Zeilen als auch Spalten einer Matrix auf einzelne Knoten im Modell abgebildet werden, ist es möglich unsymmetrische Matrizen als Graph darzustellen. Dadurch ist es aber auch sehr wahrscheinlich, dass bei einer Partitionierung des Hypergraphen die Verteilung der Zeilen und Spalten unterschiedlich ausfällt, was bedeutet, dass einzelne Matrix-Einträge und nicht ganze Zeilen bzw. Spalten einer Partition zugeordnet werden.

2.4.3. Das Problem der Graph-Partitionierung

Das Problem einen Graphen in p Partitionen aufzuteilen beschreibt nichts anderes als die Aufteilung der Knoten aus $\mathcal{V}$ in p disjunkte Teilmengen. Das heißt die Vereinigung aller Teilmengen muss der Menge $\mathcal{V}$ entsprechen ($\bigcup_{s\in\mathbb{N}} \mathcal{V}_s = \mathcal{V}$), während der paarweise Schnitt der Mengen leer ist ($\mathcal{V}_s \cap \mathcal{V}_t = \varnothing,\ \forall s \neq t$). Die Partitionierung wird oft durch einen Partitionierungsvektor $\mathcal{P}$ ausgedrückt, der für jeden Knoten $v \in \mathcal{V}$ eine Ganzzahl zwischen 1 und p entsprechend der zugeordneten Partition enthält.

Für eine optimale Partitionierung muss diese zum einem balanciert sein, zum anderem müssen die Kommunikationskosten zwischen den Partitionen minimiert sein. Balanciert ist eine Partitionierung, wenn das Gewicht der Knoten über die Partitionen gleichmäßig verteilt ist:

$$\sum_{v(i)\in\mathcal{V}_j} w_V(v_i) = \sum_{v(i)\in\mathcal{V}_k} w_V(v_i),\ \forall j \neq k,\ j,k \in 1,\ldots,p \tag{2.1}$$

Oft wird aber eine geringe Inbalance zwischen den Partitionen in Kauf genommen, da dies in den Algorithmen häufig qualitativ bessere Partitionierungen erzeugt. Beschrieben wird die Inbalance über den Faktor ϵ, so dass $\max_s |\mathcal{V}_f| \leq (1+\epsilon)\frac{|\mathcal{V}|}{p}$ für ungewichtete Graphen. Bei gewichteten Graphen gilt dies entsprechend für die Summe der Gewichte. Für die Minimierung der Kommunikationskosten können verschiedene Optimierungsmetriken eingesetzt werden.

Kantenschnitt Die am häufigsten verwendete Metrik ist die des Kantenschnittes. Der Kantenschnitt ist die Summe der geschnittenen Kanten, wobei eine Kante e_{ij} als geschnitten gilt, wenn die Knoten v_i und v_j unterschiedlichen Partitionen zugeordnet sind. Die zu erzielende Bedingung, die demnach an die Partitionierung gestellt wird, ist:

$$\sum_{\substack{e_{i,j}\ \in\ \mathcal{E}\ \text{mit}\ v_i\ \in\ \mathcal{V}_s,\ v_j\ \in\ \mathcal{V}_t \\ \text{und}\ s\ \neq\ t}} w_e(e_{i,j})\ \text{minimal}\ \forall\ \text{Partitionen}\ \mathcal{V}_s \tag{2.2}$$

Auf der Annahme basierend, dass das Maß des Kantenschnittes dem Kommunikationsvolumen der verteilten Anwendung entspricht, versuchen die meisten Algorithmen dieses Maß zu minimieren, mit der Absicht auch die Kommunikationszeit zu reduzieren. Dabei ist dies eine „Täuschung“[Hen00]. Hendrickson nennt das Kommunikationsvolumen allenfalls einen schwachen Einflussfaktor für die aufkommenden Kommunikationskosten, denn diese müssen nicht proportional zu der Anzahl der geschnittenen Kanten sein. Grund dafür ist, dass mehrere geschnittene Kanten für denselben Datenaustausch stehen können (vgl. in Abbildung 2.5 Knoten 4, der jeweils zwei Kanten zu den anderen Partitionen besitzt). Die Anzahl der Nachrichten und damit die Auswirkungen von Latenz- und eigentlicher Übertragungszeit finden dabei keine Beachtung in dem Modell (vgl. in Abbildung 2.5, dass ausgehend von Partition 1(gelb) nur Kanten in Partition 2(rot) gehen, während Partition 2 Kanten in beide anderen Partitionen besitzt). Außerdem spielt das Netzwerk selbst im Modell keine Rolle.

Grenzschnitt Alternativ versuchen deshalb neue Ansätze das Kommunikationsvolumen realistischer zu modellieren. Wie bereits erwähnt gibt der Grenzschnitt, also die Anzahl der verschiedenen Partitionen zu denen ein Knoten adjazent ist, eine genaue Aussage darüber, wie viele Elemente kommuniziert werden. Dies kann im Standard-Graphmodell gezählt werden oder, wie im Hypergraph-Modell von Çatalyürek und Aykanat [cA99], durch die Abbildung von Zeilen und Spalten (von Einträgen) auf Hyperknoten über den Kantenschnitt eines Hypergraphen bestimmt werden.

Für den Beispiel-Graphen aus Abbildung 2.5 (rechts) findet sich eine optimale Partitionierung bzgl. des Kanten- und Grenzschnittes wie sie in Abbildung 2.8 (rechts) gezeigt wird. Der Kantenschnitt hat sich dabei von 10 auf 6 reduziert; das Kommunikationsvolumen (Anzahl der Elemente in den heller gefärbten Bereichen der Matrix) hat sich gleichmaßen von 20 auf 12 verringert. Zur Übersicht würde die Matrix entsprechend der Partitionierung umsortiert.

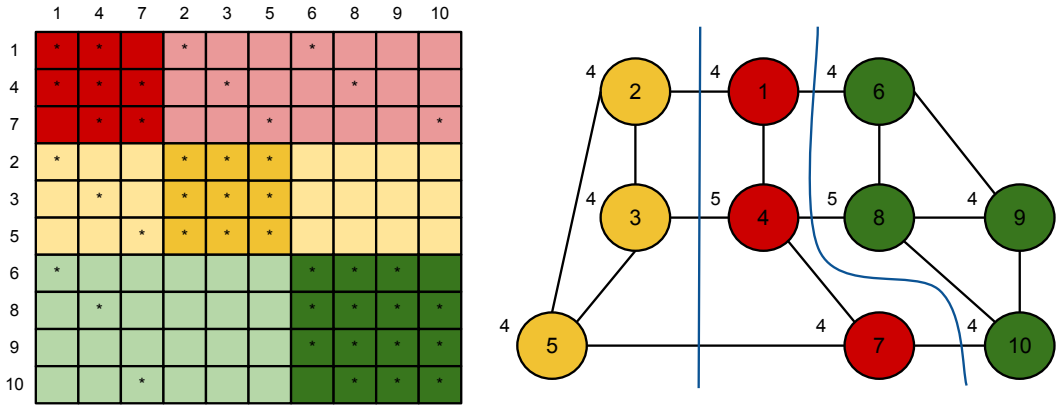


Abbildung 2.8.: Symmetrische 10×10-Matrix und deren Graphdarstellung. Diese Partitionierung minimiert den Kantenschnitt (s. blaue Linien) gegenüber der Partitionierung aus 2.5. Dieser besteht aus 6 Kanten, während das aufkommende Kommunikationsvolumen bei 12 Elementen in 4 Nachrichten liegt.

Strukturoptimierung

Die Abbildung veranschaulicht, dass das Graph-Partitionierungs-Problem bzgl. der Matrix-Partitionierung vergleichbar mit dem Problem ist, die Zeilen (und Spalten) der Matrix so zu vertauschen, dass möglichst wenige Elemente kommuniziert werden müssen. Für die Kommunikation optimal wäre eine blockdiagonale Form der Matrix, sodass alle Einträge innerhalb des lokalen Teils der Partition (dunkel gefärbt) liegen. Diese Idee verfolgt der Ansatz der Bandbreitenreduzierung.

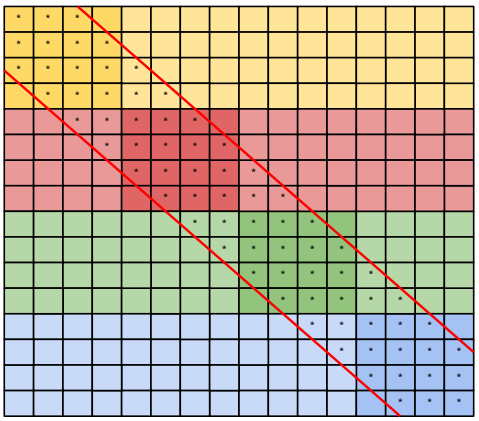


Abbildung 2.9.: Matrix mit optimaler Bandstruktur (5 Diagonalen); der lokale Teil der Partition ist quasi voll besetzt und nur einzelen Einträge ragen darüber hinaus; jeder Prozessor (mit Ausnahme des ersten und letzten) hat dieselbe Anzahl außerhalb des lokalen Teils

Ziel bei der Bandbreitenreduzierung ist es die Breite des Bandes entlang der Hauptdiagonalen, in dem Einträge auftreten, durch entsprechende Vertauschungen zu minimieren. Mit der Annahme, dass dieses Band symmetrisch zur Hauptdiagonalen liegt und die Breite kleiner als die Partitionsgröße ist, liegt ein wesentlicher Teil der Einträge im lokalen Teil der Partition während nur wenige Einträge in den Dreiecken zwischen zwei aufeinanderfolgenden Blöcken liegen. Optimalerweise liegen nach dem Vertauschen der Zeilen und Spalten keine Einträge weit von der Diagonale entfernt. Damit ist nicht nur die Anzahl der zu kommunizierenden Elemente reduziert sondern auch die Anzahl der Nachrichten. Diese liegt, unter den gegebenen Annahmen, bei zwei (s. Abbildung 2.9)(eine Nachricht für den jeweils linken und rechten Nachbarn, mit Ausnahme von der ersten und letzten Partition). Bekannte Algorithmen zur Bandbreitenreduzierung sind der Cuthill-McKnee-Algorithmus [CM69] und der von Gibbs, Poole und Stockmeyer [GPS76]. Ähnlich ist aber auch der Ansatz von George [Geo73] über die nested dissection, die in den meisten Bibliotheken zur Strukturoptimierung von dünn besetzten Matrizen verwendet wird, da er Multilevelfähig ist. Im folgenden wird deswegen nur noch der Graph-Partitionierungsansatz betrachtet wird. Dazu eine Einführung in dessen grundlegenden Ideen.

2.4.4. Graph-Partitionierungs-Algorithmen

Algorithmen zur Graph-Partitionierung unterteilt man in erster Linie in solche, die eine globale und solche, die eine lokale Sicht auf den Graphen haben. Ein globaler Algorithmus findet daher eine Partitionierung, die insgesamt ein optimales Ergebnis erzielt. Ein lokaler Algorithmus arbeit nur auf einem Teilproblem. Im Fall der Graph-Partitionierung arbeitet dieser meist rekursiv und teilt den Graphen in zwei Teilgraphen auf (Bisektion). Dabei wird in jedem Iterationsschritt des Algorithmus eine optimale Teilung der Knotenmenge vorgenommen, jedoch muss dadurch keine global optimale Lösung gefunden werden. Da sie ein größeres Gesamtproblem lösen müssen, benötigen globale Algorithmen wesentlich mehr Zeit als lokale Verfahren. Um sowohl das Gesamtproblem für eine gute Lösung im Blick zu halten als auch der Laufzeit wegen auf kleineren Problemen zu arbeiten wurden Multilevel-Verfahren entwickelt. Sie vergröbern den Graphen schrittweise, ohne dabei seine grundsätzliche Struktur zu verändern, um auf dem gröbsten Level ein direktes Lösungsverfahren anzuwenden und die Lösung rückwärts auf die feineren Level zu übertragen.
Das Partitionierungs-Problem wird allgemein so formuliert, dass eine Partitionierung in p Partitionen gesucht wird. Praktisch wird aber oft nach einer Partitionierung gesucht bei der p eine Potenz von 2 ist ($p = 2^k$ mit $k = 1, 2, \ldots$), weil die Systeme, für die die Partitionierung vorgenommen wird, meist aus 2^k Prozessoren bestehen. Aufgrund dessen und der Einfachheit des Ansatzes halber sind viele Algorithmen rekursiv, wobei sie in jedem Schritt eine einfache Bisektion vornehmen.
Außerdem machen sich manche Algorithmen evtl. vorhandene geometrische Informationen über den Graphen zunutze. Dies resultiert oft in qualitativ besseren Partitionierungen. Geometrische Informationen sind aber nicht bei allen Problemstellungen einfach zu ermitteln, zudem sind die Algorithmen komplexer und haben damit auch eine längere Laufzeit.

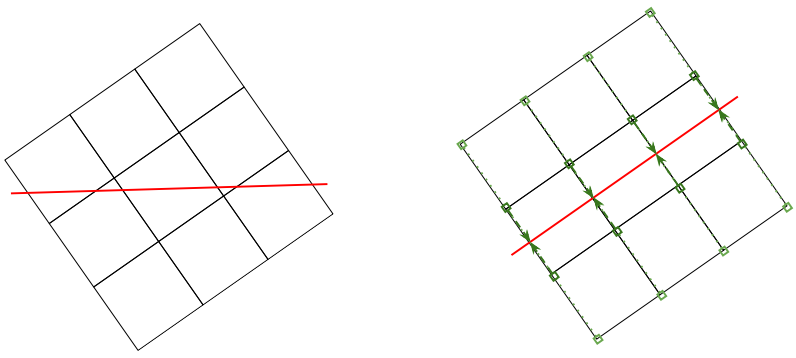
Die Sortierung der im Folgenden vorgestellten Algorithmen versucht die Komplexität dieser schrittweise zu steigern und sollen auf diese Weise das Verständnis verbessern.

Rekursive Koordinaten Bisektion

Die Rekursive Koordinaten Bisektion (RCB) gehört zu den einfachsten Partitionierungs-Algorithmen. Sie ist Teil der geometrischen Algorithmen und erzeugt lediglich lokal optimale Partitionen. Dabei wird orthogonal zu einer der Koordinatenachsen eine Hyperebene gewählt, die die Knoten so aufteilt, dass zwischen den Partitionen eine Lastbalance besteht. Von Rekursionsschritt zu Rekursionsschritt wird die zugrundeliegende Koordinatenachse gewechselt, um zu vermeiden, dass lange Grenzen entstehen und um benachbarte Knoten in einer Partition zu halten.

Rekursive Inertial-Bisektion

Die Rekursive Inertial-Bisektion (RIB) [DPHL95] verbessert den Ansatz der RCB und gehört damit auch zu den lokalen Algorithmen, die geometrische Informationen berücksichtigen. Bei der Inertial-Bisektion wird die Grenze zwischen den Partitionen über die Methode der kleinsten Quadrate gewählt. Die dahinter stehende Idee ist, dass eine Trennung orthogonal zu einer der Koordinatenachsen im Allgemeinen nicht optimal sein kann. Betrachtet man das einfache Beispiel eines recheckigen Gitters, so lässt es sich mit Hilfe von RCB optimal partitionieren, solange das Gitter dieselbe Ausrichtung wie die Achsen hat. Ist das Gitter aber gegenüber den Achsen gedreht (kein Vielfaches von 90°) ist die Partitionierung nicht optimal. Mit RIB sucht man eine Trennung, sodass die Abstände der Knoten zu dieser minimal ist. Dadurch wird auch im gedrehten Falle die optimale Partitionierung gefunden. Dies wird in Abbildung 2.10 illustriert.



(a) Bisektion durch RCB
Trennung parallel zur x-Achse
(6 geschnittene Kanten)

(b) Bisektion durch RIB
Trennung durch kleinste Quadrate
(4 geschnittene Kanten)

Abbildung 2.10.: Vergleich zwischen RCB und RIB bei einem gedrehten 4×4 Gitter

Rekursive Graph Bisektion

Die Rekursive Graph-Bisektion (RGB) [Wil91] zählt zu den graphbasierten lokalen Algorithmen der Graph-Partitionierung. Oft wird dieser Algorithmus auch Breitensuche genannt, weil er grundlegend auf einer solchen aufbaut. Der Algorithmus bestimmt über die Breitensuche die zwei Knoten, die am weitesten voneinander entfernt sind. Dazu wird die Breitensuche auf einen beliebigen initialen Knoten (Wurzel) angewendet und der von ihm am weitesten entfernten Knoten bestimmt. Dieser wird nun zur neuen Wurzel und die Suche wird solange fortgesetzt, bis sich die Entfernung zwischen den Knotenpaaren nicht mehr erhöht. Diese Heuristik konvergiert in der Praxis oft innerhalb weniger Iterationen und gibt damit eine gute Abschätzung des realen Durchmessers des Graphen. Ausgehend von der so bestimmten Wurzel wird die erste Hälfte der Knoten der ersten Partition, der Rest der zweiten Partition zugeordnet.

Kernighan-Lin

Der Kernighan-Lin (KL)-Algorithmus [KL70] gehört zu einem der ältesten Algorithmen zur Graph-Partitionierung. Dabei bestimmt er keine Partitionierung aufgrund von geometrischen oder graphentheoretischen Überlegungen von Anfang an, sondern er startet mit einer gegebenen oder zufällig gewählten lastbalancierten Partitionierung, die er durch Austauch von Knoten zwischen den Partitionen bzgl. des Kantenschnittes verbessert. Auch der KL behandelt dabei nur die Teilung in zwei Partitionen und muss für eine k-Partitionierung rekursiv angewandt werden.
Ausgangspunkt für die Frage, welche Knoten ausgetauscht werden, ist der maximale Gewinn, der durch den Austauch erzielt wird. Für zwei disjunkte Mengen $\mathcal{A}$ und $\mathcal{B}$, die initialen lastbalancierten Partitionen, werden für jedes Knotenpaar $(a, b),\ a \in \mathcal{A},\ b \in \mathcal{B}$ die internen(I) und exteren(E) Kommunikationskosten betrachtet. Die Differenz von externen zu internen Kosten für einen Knoten a ist dabei $D_a = E_a - I_a$. Ein potentieller Austausch der Knoten a und b zwischen den Partitionen reduziert dabei die Gesamtkommunikationskosten $T = T_{old} - T_{new}$ und erzielt damit einen Gewinn von $D_a + D_b - 2 \cdot c_{a,b}$, wobei $c_{a,b}$ die Kosten der Kanten zwischen a und b beschreibt.
Der Algorithmus berechnet für alle Knotenpaare den Gewinn und tauscht die zwei Knoten des Paares mit dem maximalen Gewinn aus. Im Folgenden werden diese Knoten aus der weiteren Betrachtung ausgeschlossen und es wird für die restlichen Knoten gleichermaßen fortgefahren bis kein Gewinn mehr zu erzielen ist.
Der Algorithmus hat eine Komplexität von $O(n^2 log(n))$ und ist damit relativ aufwendig. Fidduccia und Mattheyses [FM82] haben daher einen verbesserten Algorithmus vorgeschlagen, der als KL-FM bezeichnet wird und seitdem vorzugsweise angewendet wird. Er vertauscht nicht zwei Knoten miteinander, sondern verschiebt einen Knoten in die jeweils andere Partition, wenn dies einen maximalen Gewinn verspricht. Dazu wird eine bestimmte Abweichung in der Last (Anzahl der Knoten pro Partition) zugelassen.

Multilevel-Verfahren

Multilevel-Verfahren bestehen grundlegend aus drei Phasen (s. Abbildung 2.11): der Vergröberung (coarsening), der Partitionierung (partitioning) und der Verfeinerung (uncoasening). Dabei wird der zugrundeliegende Graph G durch das Zusammenfassen von Knoten (schrittweise) in seiner Größe reduziert ($G \leftarrow G^i$) und auf dem untersten Level dieser vereinfachte Graph partitioniert. Die Partitionierung wird darauf in der letzten Phase auf den Ursprungsgraphen übertragen und mit einem Verfahren, wie dem KL-FM, verfeinert. Dieses Verfahren kann mit verschiedensten Partitionierungs-Algorithmen und Verfeinerungsmethoden angewendet werden und erzielt dabei oft nicht nur bessere Ausführzeiten sondern auch ähnlich gute Lösungen.

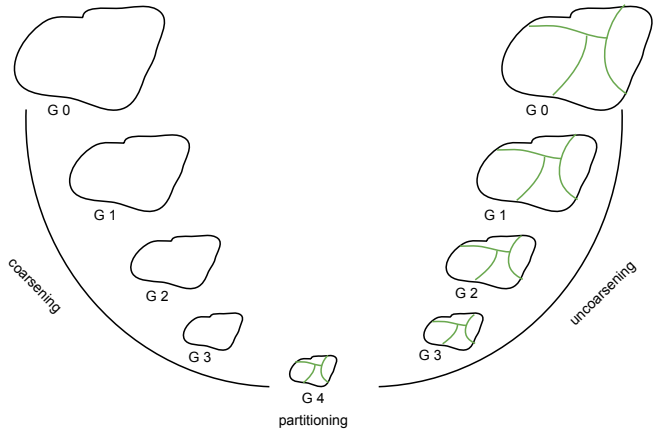


Abbildung 2.11.: Methode der Multilevel-Verfahren (nach [Kar])

Wichtig ist, dass bei der Vergröberung des Graphen die wesentlichen Abhängigkeiten (Gewichte zwischen den Knoten) bestehen bleiben, damit diese bei der Partitionierung berücksichtigt werden können. Dazu werden die Gewichte der Einzelknoten beim Zusammenfassen zu einem Superknoten in aufaddierter Form übernommen. So wird garantiert, dass der Kantenschnitt auf dem gröberen Graphen dem Kantenschnitt auf dem feinerenm entspricht.

Für genaue Beschreibungen der aufgeführten Algorithmen sowie die Ergänzung um weitere, wie die der helpful-Sets, der Tabu Suche, des Simulated Annealing oder Genetic Algorithm, sei auf [Per98] und [Els97] verwiesen.

2.4.5. Bibliotheken zur Partitionierung von Graphen und Hypergraphen

Das Problem der Graph-Partitionierung ist ein NP-schwieriges Problem, weswegen es keinen deterministischen Algorithmus gibt, der das Problem in polynomieller Zeit optimal löst [HB01]. Deshalb versucht man über Approximationen oder Heuristiken eine gute, oft aber nicht optimale, Lösung zu finden. Wie die Vielzahl an Ansätzen vermuten lässt gibt es nicht die eine Methode, mit der man die beste (nicht optimale) Lösung findet. Je mehr man bereits über das dahinterliegende Problem und die Struktur der Matrix/des Graphen

weiß, desto besser lässt sich eine passende Methode auswählen. Dabei ist auch abzuwägen, wie wichtig die Quälität der Lösung ist oder ob auch Näherungslösungen ausreichen. Ein wichtiger Faktor ist in dieser Beziehung auch der Kostenanteil zur Berechnung der Partitionierung gegenüber der Gesamtberechnung, sowie der Aufwand für die Datenumverteilung, um die Partitionierung zu erreichen. Angesichts der Laufzeitoptimierung müssen sich diese Kosten über die Gesamtlaufzeit amortisieren. Zur Auswahl hat man zahlreiche Bibliotheken, die verschiedene Algorithmen mit unterschiedlichen Optimierungsmetriken anbieten. Im Folgenden soll ein kurzer Überblick der etabliertesten Bibliotheken geschafft werden.

Metis

Metis [Kar] gehört zu den bekanntesten Bibliotheken zur Graph-Partitionierung und kommt mit ParMetis und hMetis als ganze Familie zur seriellen und parallen Graph-Partitionierung sowie zur Hypergraph-Partitionierung von der University of Minnesota. Das Projekt ist aus der Doktorarbeit von George Karypsis entstanden und wurde seitdem weiterentwickelt. Sie ist frei verfügbar und kann kostenlos zu Lehr- und Forschungszwecken genutzt werden.
Zur Graph-Partitionierung werden Multilevel-Verfahren verwendet. In der seriellen Version (Metis) stehen Multiconstraint-Verfahren der rekursiven Bisektion und k-fachen Partitionierung zur Verfügung. Für die k-Partitionierung kann man zudem zwischen Algorithmen zur Minimierung des Kantenschnitts oder des Kommunikationsvolumen auswählen. Unter Multiconstraint-Verfahren versteht man, dass für die Gewichte der Partitionen mehrere Bedingungen angegeben werden können. Dies ist für die Partitionierung von Multilevel-Verfahren nötig, da auf den verschiedenen Level unterschiedliche Anforderungen auftreten können. Mit den Multiconstraint-Verfahren wird versucht, im Mittel die beste Partitionierung zu finden.
In der parallelen Version (ParMetis) werden drei verschiedene Algorithmen angeboten: eine Multilevel- und eine koordinatenbasierte-Multiconstraint k-fach Partitionierung, sowie koordinatenbasierte Space-filling curves. Mit hMetis werden zur Hypergraph-Partitionierung ebenso Multilevel Algorithmen genutzt.

Scotch

Scotch [Pel] ist ein Projekt aus der staatlichen französischen Forschungseinrichtung INRIA. Es verfolgt mit den rekursiven dualen Bipartitionierungs-Algorithmen in der sequentiellen und parallelen Version (PT-Scotch) einen anderen Ansatz als das zuvor beschriebene Metis, erzeugt dabei aber qualitativ vergleichbare Partitionierungen bei höherer Laufzeit [Gup]. Insgesamt stellt Scotch sich als direkte Konkurrenz zu Metis dar, denn auch Scotch ist frei verfüg- und nutzbar und wird ebenso gepflegt. Für den leichten Umstieg und einen einfachen direkten Vergleich bietet es sogar Routinen mit dem Interface von Metis an. Für die Berücksichtigung heterogener Zielsysteme liest es eine Datei ein, die die zugrundeliegende Architektur beschreibt.

Jostle

Mit Jostle [jos] reiht sich eine weitere Bibliothek zur Graph-Partitionierung ein. Schirmherr des Projektes ist Chris Walshaw. Auch sie ist in serieller und paralleler Form verfügbar. Das letzte Update der Bibliothek liegt aber schon im Jahre 2002 zurück. Grund dafür wird sein, dass die Bibliothek danach als Kern in NetWorks [net] kommerziell veröffentlich wurde.
Basis stellen auch hier Multilevel-Verfahren dar. Zur Partitionierung wird RGB und zur Verfeinerung eine k-fach Variante des KL-Algorithmus verwendet. Außerdem stellt es die Möglichkeit zur Verfügung, heterogene Netzwerkverbindungen bei der Partitionierung zu berücksichtigen anstatt lediglich den Kantenschnitt zu betrachten.

PaToH

Partitioning Tools for Hypergraphs (PaToH) [cA11] ist ein Hypergraph-Partitionierungstool, dass aus der Doktorarbeit von Ümit Çatalyürek entstanden ist. Die Bibliothek ist als Binary für gängige Architekturen frei verfügbar und wird regelmäßig gewartet. Zudem steht ein Interface zu MatLab bereit. Algorithmisch setzt PaToH, wie hMetis, auf Multilevel-Verfahren und kann sich nicht nur qualitativ mit diesem messen [cA99], sondern ist dabei auch dreimal schneller[VB05].

Zoltan

Zoltan [BDF+99] ist ein Teil des Trilinos-Projektes der Sandia National Laboratories. Trilinos ist ein Framework zur Lösung großer wissenschaftlicher Anwendungen mit dem Fokus auf Multi-Physik. Es soll das Design, die Entwicklung, Integration und die beständige Unterstützung mathematischer Bibliotheken erleichtern [HBH+05]. Zoltan stellt darin ein Unterprojekt zur Organisation von Daten in verteilten Systemen dar. Es bietet die Möglichkeit zur dynamischen Lastbalancierung, Daten-Migration, Graphfärbung oder Matrixsortierung. Damit liefert es für die Partitionierung von dünn besetzten Matrizen die Möglichkeit der Graph- und Hypergraph-Partitionierung. Neben eigenen Partitionierung-Algorithmen bringt es Schnittstellen zu den bereits erwähnten Bibliotheken ParMetis, PT-Scotch und PaToH sowie Anbindungen an geometrisch-basierte Algorithmen mit. Optionen ermöglichen einen einfachen Wechsel zwischen den verschiedenen Möglichkeiten.
Zoltan ist unabhängig von den anderen Trilinos-Teilen nutzbar und frei verfügbar. Entsprechend der Design-Ziele wird es fortwährend aktualisiert und die Benutzerschnittstelle einfach gehalten. Allerdings ist der Funktionsumfang sehr groß und im default-Modus werden die zoltaneigenen Routinen benutzt, sodass der Nutzer für eine effiziente Berechnung selbst reichlich Kenntnis mitbringen muss, um aus den angebotenen Routinen eine geeignete auszuwählen.

Durch die Partitionierung von Matrizen soll die Leistung der Berechnung mit der Aufteilung auf ein paralleles System erhöht werden. Für sehr große Aufgaben wird erst durch ein Aufteilen eine (effiziente) Berechnung ermöglicht, da ansonsten die Daten nicht im Speicher gehalten werden können. Um bei der parallelen Berechnung das System bestmöglich auszunutzen sollte die Aufteilung balanciert sein, wodurch die Frage aufkommt wie die Matrix zu verteilen ist. Denn für dünn besetzte Matrizen gilt es dazu die Struktur der Matrix zu beachten, da nicht jede mögliche Zuordnung der Zeilen zu den Partitionen ein Lastgleichgewicht erzeugt. Mit der Graph-Partitionierung hat man nicht nur eine Heuristik um eine Zuordnung zu finden, die die Gewichung erfüllt, sondern gleichermaßen die Abhängigkeiten zwischen den Partitionen minimiert. Der daraus gewonnene Vorteil ist, dass weniger zwischen den Partitionen kommuniziert werden muss. Das Ergebnis der Partitionierung ist anschauungshalber eine Permutation der Zeilen und Spalten, sodass die Partitionen einen zusammenhängenden Block formen (vgl. Abbildung 2.8), bei der eine Großzahl der Matrix-Einträge in den Diagonal-Block „permutiert“ wurden. Für jeden Eintrag außerhalb dessen besteht eine Abhängigkeit die kommunizierent werden muss. Abgesehen davon, dass durch die angestrebte blockdiagonale Form die Kommunikation minimiert wird, hat dies ebenfalls den Effekt, dass die Einträge enger beeinander liegen, was in der weiteren Betrachtung einen verbesserten Zugriff für die SpMV bdeutet.

Zur Beschreibung einer dünn besetzten Matrix gibt es verschiedene Modelle, die symmetrische und nicht symmetrische Matrizen abdecken und in unterschiedlichen Abschätzungen des aufkommenden Kommunikationsvolumen resultieren. Zudem gibt es verschiedene Algorithmen die in mehreren Bibliotheken umgesetzt wurden. Für diese Arbeit wurde das allgemeine Modell des ungerichteten Graphen benutzt und sich damit (vorerst) auf symmetrische Matrizen beschränkt. Als Bibliothek wurde Metis mit dem multi-level basierten Graph-Bisektions-Verfahren und Kantenschnitt-Metrik eingesetzt.

3. Hybride Systeme

Die Entwicklung aktueller Hochleistungsrechencluster bewegt sich schon seit einiger Zeit in Richtung heterogenerer Systeme. Die Nutzung von Grid-Systemen, die unterschiedlichste CPUs in verschiedenen, über den Globus verteilten, Rechenverbänden zu einem System verbinden, gehört zum Standard-Repertoire großer wissenschaftlicher Berechnungen. Spätestens mit dem Einzug der General Purpose Computation on GPUs (GPGPU) fanden auch spezialisierte Prozessoren ihren Platz als Rechenbeschleuniger in Rechencluster. Dabei werden die Rechenbeschleuniger den CPUs als Co-Prozessor für rechenaufwändige Programmteile zur Seite gestellt. Derzeitige Rechenbeschleuniger sind GPUs, Field Programmable Gate Arrays (FPGAs) und die Cell Broadband Engine (Cell B.E.). Neu dazu kommt der Many Integrated Core (MIC) von Intel.

Ein solches hybrides System besteht dabei schon aus zwei Hierachiestufen. Auf der ersten Stufe stehen die CPUs, die sich in erster Linie um die Logik des Programmes kümmern, auf der zweiten Stufe stehen die Rechenbeschleuniger (compute devices), die die eigentliche Berechnung übernehmen. Dabei ist ein Rechenbeschleuniger mit einer CPU verbunden von der er verwaltet wird und seine Daten bezieht. Diese Verbindung besteht i. d. R. über den PCI-Express-Bus zwischen dem Device und dem Chipsatz der CPU. Die CPU ist wiederum zum Austausch gemeinsamer Daten an ein Netzwerk mit anderen Prozessoren angeschlossen. Das dazu verwendete Modell ist meist das des Message Passing, in dem Prozessoren über den Austausch von Nachrichten kommunizieren. Eine direkte Kommmunikation zwischen verteilten Devices ist dagegen nicht möglich, sondern muss immer über die zugehörigen CPUs laufen.

Eine verteilte hybride Anwendung setzt sich daher daraus zusammen, dass die CPUs sich über die Verteilung der Rechenlast einigen, die Daten entsprechend verteilen und danach jede CPU die Daten auf ihr(e) Device(s) lädt und dort die Berechnung startet. Am Ende der Berechnung muss die Lösung vom Device zurück auf die CPU kopiert werden, um sie zwischen den Prozessoren kommunizieren zu können.

In größeren Rechenverbänden oder Grid-Systemen bauen sich weitere Hierachiestufen zwischen einzelnen Domänen meist baumartig auf. Damit erfolgt die Kommunikation zwischen CPUs verschiedener Domänen i. d. R. über dedizierte Knoten, während die CPUs innerhalb einer Domäne direkt miteinander kommunizieren können. Durch dieses Modell kommt neben der Heterogenität der Prozessoren auch eine hohe Heterogenität im Kommunikationsnetzwerk dazu. Der grundlegende Aufbau wird in Abbildung 3.1 skizziert.

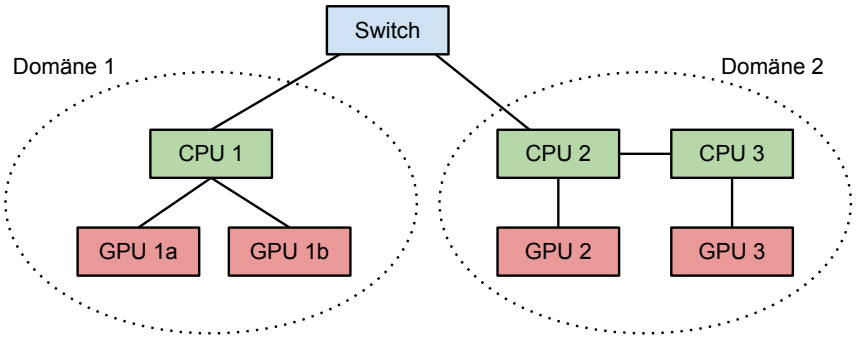


Abbildung 3.1.: Aufbau eines heterogenen Systems

Im Supercomputing Sektor setzte sich im Jahre 2008 der Roadrunner als cell-basiertes Hybridsystem auf Platz 1 der Top500 [top]. In den folgenden Jahren kamen weitere, hauptsächlich GPU-basierte, Systeme hinzu, die in den vordersten Reihen mitspielen. Aktuell (Stand:Februar 2012) befinden sich mit dem Tianhe-1A (Platz 2), Nebulae (Platz 4) und TSUBAME 2.0 (Platz 5) drei GPU-basierte Systeme unter den ersten zehn Plätzen. Diese Verbindung erfreut sich großer Beliebtheit, da die Entwicklung von Grafikkarten aus dem Konsumentenbereich (Computerspiele, Bildverarbeitung) angetrieben wird und somit immer schneller größere Leistungssteigerungen als aktuelle CPUs erzielt [KH10]. Der Roadrunner belegt aktuell immer noch Platz 10.
Der Cell-Prozessor ist jedoch mit der GPU-Entwicklung verdrängt worden, weil er mit seinen 16 SIMD-Kernen nicht mit einigen Hundert Streaming-Kernen der Grafikkarten mithalten kann. Zudem hat sich die SIMD-konforme Programmierung des Cells nur für wenige Anwendungsfelder durchgesetzt. Die GPU-Programmierung dagegen hat sich in einem breiten Feld etabliert. Nvidia bietet zu ihrer Compute Unified Device Architecture (CUDA) entsprechende Spracherweiterungen für C und Fortran an. AMD/ATI setzt neben der Windows-Schnittstelle DirectX auf die Open Computing Language (OpenCL). Diese wird entsprechend der Intention zusätzlich von CUDA und dem Cell unterstützt und bietet somit eine portable Programmiermöglichkeit für Rechenbeschleuniger. Auch FPGAs bedürfen hardwarespezifischer Programmierung oder besser gesagt einer Konfiguration der internen Schaltungen. Der MIC dagegen lässt sich als einfache x86-Architektur mit dem OpenMP-Modell handhaben. Daneben existieren aber auch Modelle mit OpenCL[opea] und Intel Cilk Plus[Int].

3.1. Hardwarekomponenten eines hybriden Systems

Für die verschiedenen Prozessortypen gehen neben den oben genannten unterschiedlichen Programmiermodellen auch voneinander abweichende Ausführ- und Speichermodelle einher. Je nach Anwendung ist eine Hardware besser oder schlechter für eine effiziente Berechnung geeignet. Für diese Arbeit soll sich auf ein hybrides System aus CPUs und GPUs beschränkt werden. Um die wesentlichen Einflüsse der Hardware auf die Leistung der durchzuführenden SpMV-Berechnung erklären zu können, wird eine detailierte Betrach-

tung der Hardware nötig, denn eine Bewertung auf Basis der maximalen Rechenleistung, die Hardware-Hersteller angeben, gibt kein genaues Bild über die tatsächlich zu erwartende Leistung. In den nächsten Abschnitten wird deshalb der Aufbau der Prozessoren vorgestellt und in leistungsrelevante Details eingeführt. Als Letztes wird die Netzwerkverbindung der Systemkomponenten behandelt.

3.1.1. Central Processing Unit (CPU)

Die CPU ist die zentrale Verarbeitungseinheit eines Rechners und ist für sämliche Aufgaben des Betriebssystems und laufender Programme verantwortlich. Aus diesem Grund ist die Architektur einer CPU auf die effiziente Ausführung sequentiellen Codes spezialisiert. Dies bedeutet, dass sie gut mit Sprüngen im Kontrollfluss des Programmes sowie in den Speicherzugriffen umgehen können muss, um zwischen den Aufgaben verschiedener (Anwendungs-)Prozesse wechseln zu können. Zudem verfügt die CPU über eine hierarchische Cache-Struktur, damit in rechenintensiven Aufgaben die räumliche und zeitliche Lokalität der Daten ausgenutzt wird und nicht für jedes Datum aufwendig auf den Arbeitsspeicher zugegriffen werden muss.
Wie in Abbildung 3.2 gezeigt wird, besteht der Chip der CPU aus der Kontrolllogik für das Laden und Dekodieren von Befehlen, den Recheneinheiten (Arithmetic Logic Units (ALU) für die Verarbeitung unterschiedlichster Berechnungen sowie der Cache-Hierachie aus L1-, L2- und meist auch L3-Cache. Der Arbeitsspeicher ist außerhalb des Chips angeordnet.

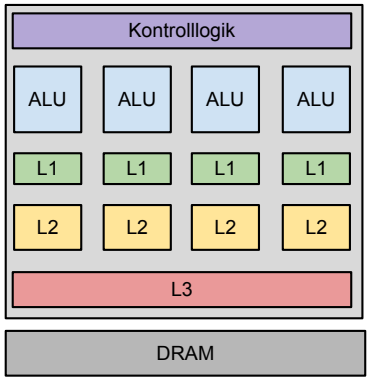


Abbildung 3.2.: Aufbau einer CPU

Die Kontrolllogik und Caches nehmen schon einen großen Anteil der Chipfläche ein, sodass insgesamt verhältnismäßig wenig Platz für die Recheneinheiten übrig bleibt. Diese verfügen bei einer CPU sowohl über Register für einfach genaue (single precision) als auch doppelt genaue (double precision) Rechnungen. Verarbeitet werden von ihnen zum einen einfache Fließkommaberechnungen sowie komplexere Befehle (z. B. das Ziehen einer Wurzel oder die Berechnung des Sinus). Lediglich die Dauer für die Berechnung der komplexen Befehle kann variieren[1]. Für wissenschaftliche Berechnungen ist die Anzahl der Fließkom-

[1]Die Dauer einer Befehls hängt vom jeweiligen Befehlssatz ab. Man unterscheidet dabei zwischen sogenannten CISC- und RISC-Architekturen, die in einem Maschinenbefehl eine komplexe Operation

maberechnungen (i. A. in doppelter Genauigkeit) pro Takt relevant. Meist kann ein Kern eine Multiply-Add-Operation (MAD)-Operationen pro Takt ausführen. Mit zusätzlicher Streaming SIMD Extension (SSE)-Unterstützung können es auch vier MAD sein.

Die Caches dienen als Zwischenspeicher von Daten, die voraussichtlich wiederverwendet werden. Dabei werden benötigte Daten vor ihrer Nutzung aus dem Arbeitsspeicher in den L1-Cache geladen und beim nächsten Zugriff direkt von dort bezogen. Das Laden von Daten aus dem Cache verringert die Latenzzeiten signifikant gegenüber dem Beziehen aus dem Arbeitsspeicher. Dabei wird immer eine ganze Cache-Zeile geladen, da neben der zeitlichen Wiederverwendung der Daten auch eine räumliche Lokalität in den Datenzugriffen eines Programmes wahrscheinlich ist. Die Größe eines Caches ist jedoch begrenzt, sodass Daten von dort verdrängt und in der nächst höheren Hierachieebene gehalten werden. Jeder Cache-Miss bedeutet einen Zugriff auf die nächst höhere Hierachie bis zu einem Zugriff auf den Arbeitsspeicher. Man unterscheidet bei Cache-Misses drei Klassen: Compulsary, Conflict und Capacity (3C's der Cache Misses [WWP08]):

Compulsary Ein compulsary miss tritt bei der erstmaligen Verwendung eines Speicherdatum auf. Dies bezeichnet man auch als cold start.

Conflict Man spricht von einem conflict miss, wenn ein Datum erneut in den Cache geladen werden muss, weil an der gleichen Cache-Adresse eine andere Zeile geladen wurde.

Capacity Ein capacity miss bezeichnet ein erneutes Laden, dass auftritt, weil nicht alle Daten für die Berechnung in den Cache passen.

Der L1- und L2-Cache ist i. d. R. einem Kern zugeordnet, während sich alle den L3-Cache teilen. Ferner spaltet sich der L1-Cache in je einen Teil für Daten und für Instruktionen auf. Ziel der Caches ist es, durch das Zwischenspeichern der Daten einen schnelleren Zugriff auf diese zu ermöglichen, um den Prozessor schnell genug mit Daten versorgen zu können. Anderenfalls wird ein Programm nicht durch die Leistung des Prozessors, sondern durch die Speicherbandbreite begrenzt. Die resultierende Leistung wird allerdings stark vom Zugriffsmuster auf die Daten sowie Cache-Größen, Verdrängungs- und Prefetchstrategien beeinflusst, sodass sie i. d. R. nur schwer kalkuliert werden kann.

Für die parallele Ausführung auf CPUs stehen die Modelle des Open Multi-Processing (OpenMP) und Message Passing Interface (MPI) zur Verfügung. Dabei wird OpenMP für die Parallelisierung auf einem Multicore-Prozessor der MPI-Parallelisierung vorgezogen, da die parallelen Threads gegenüber den Prozessen leichtgewichtiger und damit günstiger in der Erstellung sind. Dabei werden parallele Programmteile durch Pragmas für den Compiler gekennzeichnet und von diesem auf die Threads verteilt. Eine Parallelisierung über Prozessorgrenzen hinweg kann nur über MPI vorgenommen werden. Dabei tauschen die parallelen Prozessoren über Nachrichten Daten aus. Grundsätzliche Vorraussetzung

durchführen, wobei jeder Maschinenbefehl unterschiedlich lang dauern kann, bzw. für eine Operation mehrere gleich lang andauernde Maschinenbefehle benötigen.

für eine effiziente Parallelisierung ist eine generelle Datenunabhängigkeit zwischen den parallelen Teilen, sodass die Prozesse autark rechnen können und nicht von der Ausführung anderer gebremst werden.

3.1.2. Graphic Processing Unit (GPU)

Die eigentliche Aufgabe einer Grafikkarte ist die Berechnung der Grafikausgabe. Dafür ist ihr Aufbau darauf ausgelegt möglichst effizient dieselbe Operation auf große Datenströme anzuwenden. Im Zuge der GPGPU-Bewegung kam sie allerdings auch für die Berechnung großer wissenschaftlicher Anwendungen ins Blickfeld. Erfüllen diese ebenfalls die Anforderung von vielen unabhängigen Operationen auf sequentiellen Daten, so lässt sich die Leistung der GPU auch für Berechnungen aus dem High-Performance-Computing (HPC)-Bereich nutzen. Zu Anfang mussten für die Nutzung der GPU als compute device große Anstrengungen in der Programmierung unternommen und die Berechnung auf die Shader des Grafikrenderings gemappt werden. So bestehen derzeit mit CUDA, DirectX und OpenCL intuitivere Programmiermodelle für die GPU. Außerdem gingen die Grafikkarten-Hersteller auf die Anforderungen von IEEE-konformen doppelt genauen Berechnung ein und entwickelten entsprechende Karten, die diese unterstützen. So sind neueste Karten, wie die der Nvidia Tesla-Serie, allein auf den HPC-Bereich fokusiert, da sie nicht mehr über einen Grafikausgang verfügen.

Aufbau einer GPU

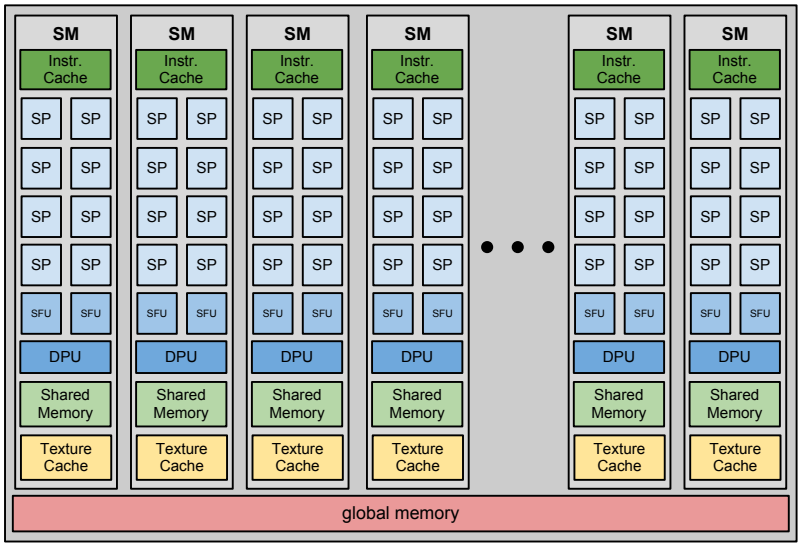


Abbildung 3.3.: Aufbau einer GPU

Eine GPU besteht aus einer Vielzahl einfacher Rechenkerne, sogenannte Streaming Processors (SPs), die gitterartig auf dem Chip angeordnet sind. Eine Menge SPs formt einen sogenannten Streaming Multiprozessor (SM). Außerdem verfügt die GPU über ein hierarisches Speichermodell. Der grundlegende Aufbau ist in Abbildung 3.3 illustriert.

Jeder SP verfügt über lokalen Speicher sowie eine eigene Floating Point Unit (FPU), die ausschließlich simple, einfache genaue Fließkommaberechnungen umsetzen können. Die Kontrolllogik und der Instruktions-Cache werden von den SPs innerhalb eines SMs geteilt, sodass die SPs ihre Berechnungen simultan durchführen. Zudem teilen sie sich einen Cache-Speicher für Daten und einen für Texturen. Auf diesen Speicher kann lediglich von den SPs des zugehörigen SM zugegriffen werden; zwischen den SMs können keine Daten geteilt werden. Die Berechnung von komplexen Befehlen, wie beispielsweise der Berechnung einer Wurzel, wird von den Special Function Units (SFUs), von denen mehrere zu einem SM gehören, übernommen. Ebenso gibt es nur ein begrenzte Anzahl Double Precision Unit (DPU) pro SM bzw. doppelt genaue Berechnungen werden von zwei SPs umgesetzt, sodass die Leistung für doppelt genau Berechnungen hinter der von einfach genauen bleibt.
Der globale Speicher der GPU, aus dem die Blöcke ihre Daten beziehen, verfügt zur schnellen Datenversorgung der SPs über eine wesentlich höhere Bandbreite als der Arbeitsspeicher einer CPU. Dafür ist die Latenzzeit des globalen Speichers um einiges höher. Dieser Nachteil wird allergings durch das Ausführmodell der CUDA-Architektur wieder aufgefangen. Zusätzlich müssen für eine volle Ausnutzung der Speicherbandbreite zum globalen Speicher Bedingungen an den Zugriff gestellt werden (Details dazu folgen in Absatz 3.1.2).
Der shared memory erlaubt einen schnelleren Zugriff auf Daten innerhalb eines SMs. Die Daten müssen dafür aber speziell von dem Kernel dazu instrumentiert werden. Dies macht den shared memory zu einer Art Cache, der vom Programmierer verwaltet wird. Der Texture Cache ist vergleichbar mit dem L1-Cache einer CPU und ist für den effizienten Zugriff auf Shader-Texturen gedacht. Die Daten werden dafür in zwei Dimensionen gecached. Allerdings sind die Texturen sehr klein, sodass er nur wenige Kilobyte umfasst. Da der Zugriff auf den Texture Cache nicht den Bedingungen des globalen Speichers unterliegt, verringert er bei Daten mit zweidimensionaler Lokalität die Speicherzugriffszeit erheblich.

CUDA-Ausführmodell

CUDA sieht ein Modell der thread-parallelen Ausführung, die es als Single Instruction Multiple Thread (SIMT) bezeichnet, vor. Dabei operieren eine große Anzahl Threads (wesentlich mehr als vorhandene SPs) parallel auf den SPs der GPU. Die Threads werden in einem ein- bis dreidimensionenalen Gitter organisiert. Ein Gitter aus Threads ist wiederum in Blöcke eingeteilt[2], die ihre Threads wiederum in einem zweidimensionenalen Array anordnen. Einem Thread wird innerhalb des Programmes über seine Position im Grid sein Index und darüber sein Teil der Rechenarbeit zugewiesen.
Die Thread-Organisation ist anhand eines zweidimensionenalen Gitters aus zwei mal drei Thread-Blöcken mit je neun Threads pro Block in Abbildung 3.4 skizziert.

[2]Ein Block kann je nach compute capability maximal 512 bzw. 1024 Threads beinhalten.

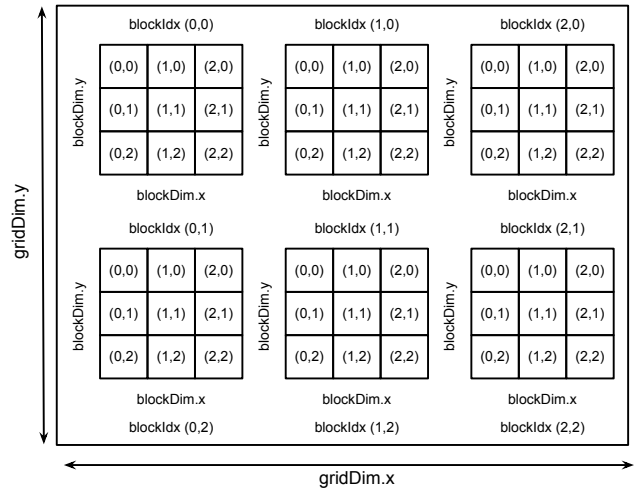


Abbildung 3.4.: Thread-Organisation von CUDA am Beispiel eines zweidimensionenalen Gitters

CUDA-Threads werden sehr schnell erzeugt und können ebenso schnell auf den SPs gewechselt werden, sodass Threads, die auf Daten warten, durch andere ausgetauscht werden können. Durch den schnellen Wechsel von aktiven und auf Daten wartenden Threads wird die hohe Latenzzeit beim Zugriff auf den Speicher versteckt, wenn eine entsprechend hohe Datenparallelität gegeben ist. Nur so kann die Recheneinheit voll ausgelastet werden.
Von der CUDA-Runtime wird die Verteilung der Thread-Blöcke (maximal 8 Blöcke auf einmal) auf die SMs vorgenommen. Durch die Organisation der Threads in Blöcken wird ein einfaches Modell gebildet, anhand dessen für beliebige CUDA-Karten eine skalierbare Möglichkeit der Threadverteilung vorliegt. Die Threads eines Blockes werden in sogenannte Warps eingeteilt. Die Anzahl an Threads pro Warps liegt derzeit für alle GPUs bei 32 und wird als *warpsize* bezeichnet. Ein Warp ist dabei die minimale Anzahl an Threads die ein SM bearbeitet.

Speicherzugriffe innerhalb von CUDA

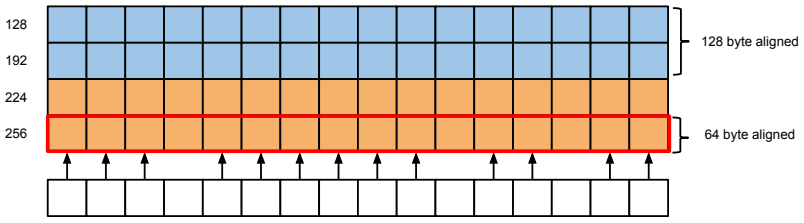
Bei der Vorstellung der GPU sind bereits die verschiedenen Speicherklassen des lokalen, shared und globalen Speichers sowie des Texture Cache gefallen. Im lokalen Speicher hält ein SP private Variablen, die innerhalb des Kernels benötigt werden. Im wesentlichen handelt es sich hierbei um Schleifenzähler und temporäre Daten, wodurch der lokale Speicher mit einzelnen Registern der CPU zu vergleichen ist. Der shared memory ist ein vom Programmierer selbst festzulegender Speicherbereich mit schnelleren Zugriffszeiten als auf den globalen Speicher. Somit kann eine Anwendung deutlich beschleunigt werden, wenn häufig benötigte Daten im shared memory vorliegen. Allerdings ist die Größe des Speichers begrenzt und die Daten nur innerhalb eines SM verfügbar, sodass er nur verwendet werden kann, wenn die Daten nicht global von allen Threads benötigt werden. Der Texture Cache ist noch kleiner, da er beim Grafikrendering nur für die Zwischenspeicherung von Texturen verwendet wird. Aufgrund dessen ist er ausschließlich read-only, hat aber den großen Vorteil, dass ein schneller und uneingeschränkter Zugriff ermöglicht wird. Die Einschränkungen werden im Folgenden für den globalen Speicher erläutert.

Da beim Grafikrendering eine Transformation auf eine Vielzahl an Pixel angewandt wird, benötigen die einem Warp zugeordneten SPs einer SM aufeinanderfolgende Pixel-Daten. Dieses Zugriffsmuster wurde daher bei der Architektur für die Erzielung einer hohen Speicherbandbreite ausgenutzt. Mit einem einzelnen Zugriff auf den globalen Speicher werden je nach compute capability der Grafikkarte 64 bzw 128 Byte am Stück gelesen oder geschrieben. Für eine volle Auslastung der Speicherbandbreite müssen daher die kompletten Daten auch für die Berechnung verwendet werden. Dies wird als coalesced (verschmolzener) Speicherzugriff bezeichnet und Nvidia bezeichnet die Einhaltung dessen als die „einzig wichtige Leistungsberücksichtigung“ bei der CUDA-Programmierung [Nvi10]. Denn es wird nur dann einmal ein einzelner Datenblock für den Warp geladen, wenn der Zugriff im Programm gewisse Bedingungen erfüllt.
Für Karten mit compute capability $1.x$ liegt ein coalesced memory access nur dann vor, wenn die Bedingungen innerhalb eines Half-Warp erfüllt sind und beziehen sich auf 64 Byte. Bei compute capability $2.x$ müssen sie innerhalb eines Warp erfüllt sein und beziehen sich auf 128 Byte. Dies ist nochmal in Tabelle 3.1 zusammengestellt.

compute capability	$1.x$	$2.x$
coalesced Zugriff innerhalb	Half-Warp 16 Threads	Warp 32 Threads
coalesced Zugriff in Byte	64	128

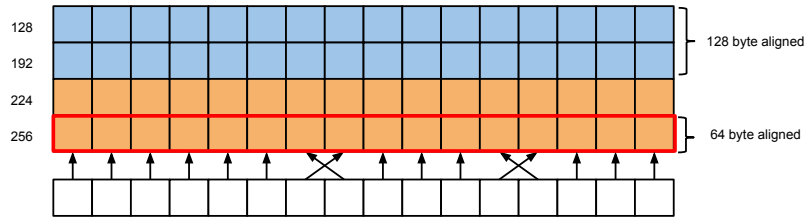
Tabelle 3.1.: Angaben zum coalesced memory access nach spezieller compute capability

Ein optimaler coalesced memory access liegt vor, wenn (Half-)Warpsize Threads auf ebensoviele Daten zugreifen, wobei der n-te Thread auf das n-te Datum zugreift. Zudem muss die Adresse des ersten Datums aligned sein. Im Detail heißt dies, dass der erste Thread auf eine Adresse, die auf der Grenze eines Vielfachen von 64 bzw. 128 beginnt ($n = 64 \cdot k$ bzw. $128 \cdot k, k \in \mathbb{Z}$), zugreift und die nächsten 16 bzw 32 Threads die darauffolgenden Elemente adressieren ($n + 1$ bis $n + 15$ bzw. $n + 31$). Sind einzelne Threads an dem Zugriff nicht beteiligt wird Bandbreite zwar nicht effektiv genutzt, es bleibt aber bei einem einzigen Zugriff auf den globalen Speicher. Liegt dagegen kein streng sequentielles Zugriffsmuster vor, kann für jedes einzelne Datenelement das Laden eines ganzen Blockes hervorrufen, was ein Vielfaches der Bandbreite benötigt. Dies wird näher an den dargestellten Zugriffen in Abbildung 3.5 erläutert.

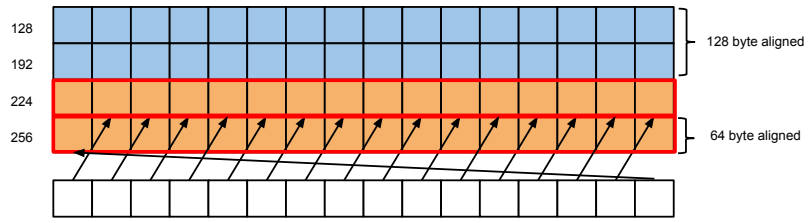


(a) sequentieller Zugriff

Abbildung 3.5.: (un)coalesced Speicherzugriffe in CUDA



(b) überkreuzender Zugriff



(c) mis-aligned Zugriff

Abbildung 3.5.: (un)coalesced Speicherzugriffe in CUDA (Beispiel für compute capability 1.x - Halfwarpsize); (a) sequentieller Zugriff: immer coalesced; resultiert in einem einzelnen Speicherzugriff (roter Block) für alle 13 teilnehmenden Threads innerhalb des Half-Warps; (b) überkreuzender Zugriff: uncoalesced bis compute capability 1.2 (16 Zugriffe), ab 1.3 coalesced (1 Zugriff); (c) mis-aligned Zugriff: uncoalesced bis compute capability 1.2 (16 Zugriffe), ab 1.3 coalesced (2 Zugriffe)

Bei Abweichungen von einem streng sequentiellen Zugriffsmuster unterscheiden sich, abgesehen von der Größe des Datenblockes, noch einmal die Bedingungen an den Zugriff. Für einen von den Threads überkreuzenden Zugriff wird bis compute capability 1.2 jeweils ein einzelner Zugriff für jedes Datum nötig; ab compute capability 1.3 ruft dies, abgesehen vom ersten Laden, keine weitere Zugriffe hervor. Ähnlich verhält es sich für einen verschobenen (mis-aligned) Zugriff auf die Daten. Bis 1.2 störte dies den coalesced Zugriff, sodass für jedes Datum ein Block geladen werden musste; mit 1.3 reduziert sich die Anzahl der Ladevorgänge auf die Anzahl der Blöcke, auf die zugegriffen werden muss.

3.1.3. Netzwerkverbindungen

Zu einem verteilten System gehören nicht nur die einzelnen Rechenkomponenten, sondern auch immer die Verbindungen zwischen diesen. Ein solches System beginnt bei zwei verbundenen Prozessoren und endet bei einem Grid-System aus verteilten Komponenten auf dem gesamten Globus. Dabei ist das Netzwerk sicherlich mindestens so heterogen wie seine Rechenkomponenten. Hierachien im Netzwerk bauen sich von den Verbindungen innerhalb von symmetrischer Multiprozessoren (SMPs) über Verbindungen zwischen verschiedenen Systemen an einem Standort (Domänen), bis hin zu weltweiten Grid-Systemen auf. Das Einbringen von Rechenbeschleunigern wie GPUs in verteilte Systeme fügt dabei eine weitere Hierachiestufe hinzu, weil die Daten aus dem Speicher der CPU auf die GPU kopiert werden müssen. Bei der Verbindung zwischen CPU und GPU handelt es sich i. d. R. um eine Peripheral Component Interconnect (PCI)-Express-Schnittstelle.
Relevante Größen für die Beurteilung des Kommunikationsnetzwerkes sind für jedes Prozessorpaar Latenzzeit und Bandbreite. Die Latenzzeit beschreibt den Kommunikations-

aufbau, also den Zeitraum vom Starten der Kommunikation bis zum tatsächlichen Senden des ersten Bytes. Die Bandbreite gibt an wie viele Daten pro Sekunde (Byte/s) übertragen werden können. Die resultierende Kommunikationszeit ist damit folglich in der Theorie für eine Verbindung so anzugeben:

$$t_{comm} = \text{Latenzzeit} + \text{Bandbreite} \cdot \text{Menge an Daten (in Byte)} \tag{3.1}$$

Oft wird die Latenzzeit für die Bewertung der Kommmunikation nicht mit einbezogen, da sie bei der Übertragung großer Datenmengen nicht ins Gewicht fällt. Für kleine Übertragungen kann sie aber durchaus großen Einfluss nehmen. Liegt keine Vollvermaschung vor, wird die Leistung zusätztlich von der allgemeinen Netzwerklast und Kongestion in einzelenen Knotenpunkten beeinflusst [WC01], da das Netzwerk geteilt werden muss und mehrere Nachrichten über eine Leitung oder über gleiche Knoten geroutet werden.

Insgesamt stellt das Verbindungsnetzwerk ein komplexes Gebilde dar, das von Hardwaregrößen, aber auch zeitlichen Zusammenkünften und damit von den Rechenkomponenten und Kommunikationsmustern, wer mit wem zu kommunizieren hat, abhängt. Je größer und heterogener das Netzwerk, desto schwieriger ist die Gesamtbetrachtung, ob und wenn ja wieviele und welche Daten auf einen anderen Prozessor ausgelagert werden. Letztlich muss es sich für die Gesamtausführung lohnen, die Daten an einen anderen Prozessor zu schicken und die Ergebnisse wieder von dort zu beziehen. Die Berechnung auf einem Rechenbeschleuniger lohnt sich daher aufgrund der doppelten Kommunikation erst bei großem Rechenaufwand. Für die im Folgenden betrachtete SpMV-Berechnung auf CPUs und GPUs sind alle Matrix- und Vektor-Daten (A, x) an eine neue Partition abzugeben und die Ergebnis-Daten (y) wieder einzusammeln. Ist der Partition eine GPU zugeordnet muss die Kommmunikation erst zwischen den Knoten über das Verbindungsnetzwerk und danach über den PCI-Express-Bus erfolgen.

3.2. Begegnung der Heterogenität

Spätestens mit dem Aufkommen des Grid-Computing wurde die Berücksichtigung heterogener Systemkomponenten für die Lastbalancierung im Allgemeinen und die Graph-Partitionierung als speziellen Anwendungsfall wichtig. Während bis dahin die Last lediglich gleichmäßig aufgeteilt werden musste, galt es dann die Last in Relation zur jeweiligen Leistung des Prozessors und/oder Verbindungsnetzwerkes zu setzen, um die Gesamtausführung zu balancieren. Dadurch kamen drei neue Problemstellungen hinzu:

1. Wie stellt man verschiedene Prozessoren in Relation?
2. Wie stellt man verschiedene Verbindungsnetzwerke in Relation?
3. Wie setzt man Rechen- und Kommunikationsleistung zusammen, sodass die Gesamtleistung dadurch repräsentiert wird?

Bei der Bewertung der Rechenleistung der Prozessoren genügt für die Balancierung eine relative Angabe, wie sich die unterschiedlichen Prozessoren zueinander verhalten, da sich Seiteneffekte durch Caches oder Schleifenoverheads im gleichen Verhältnis auf die resultierende Laufzeit auswirken. Somit stellt sich eine Gewichtung anhand der maximalen Rechenleistung als ausreichend dar. Ebenso kann für das Verbindungsnetzwerk mit Formel 3.1 ein repräsentativer Wert gefunden werden.
Während sich die ersten beiden Fragen theoretisch ohne Beachtung der durchzuführenden Operation beantworten lassen, kann die letzte nicht ungeachtet des Verhältnisses von Berechnungs- und Kommunikationsaufkommen geklärt werden. Wie damit in Bezug auf die Graph-Partitionierung und der Last-Balancierung im Allgemeinen umgegangen wird, soll in den nächsten Abschnitten vorgestellt werden.

3.2.1. Begegnung der Heterogenität in der Graph-Partitionierung

Auf Seiten der Graph-Partitionierung stellen sich mit der Heterogenität der Systeme zwei Probleme dar: zum einem müssen verschieden große Partitionen erstellt werden, zum anderen gilt es das Verhältnis von Berechnungs- zu Kommmunikationsaufwand zu berücksichtigen. Durch den meist rekursiven Ansatz der Graph-Partitionierungsalgorithmen begegnet man der Anforderung der verschieden großen Partitionen darüber erst kleinere Partitionen zu erstellen und sie entsprechend der Gewichte günstig zusammenzufassen. Somit kann durch die Graph-Partitionierung trotz Heterogenität eine Aufteilung gefunden werden, die die unregelmäßige Struktur des Problems (hier: der Matrix) ausgleicht und dabei die Kommmunikation minimiert.

Für heterogene Netzwerke schlagen Walshaw und Cross in [WC01] eine andere Optimierungsmetrik vor, als in 2.4.3 vorgestellt wurde, um den unterschiedlichen Netzwerkverbindungen Genüge zu tun und mit der Rechenleistung zu verbinden. Intergriert wurde dies in ihrem Graph-Partitionierer JOSTLE [jos]. Walshaw und Cross verfolgen auf Grundlage des Kantenschnittes den Ansatz, Netzwerkgrößen bei der Bewertung des Schnittes einzubeziehen, um so im Gegensatz zum einfachen Kanten- oder Grenzschnitt nicht das Kommmunikationsvolumen, sondern die Kommmunikationszeit zu minimieren. Dazu bauen sie einen vollständigen Kommmunikationsgraph des Systems auf, in dem nicht direkt verbundene Knoten durch Aufsummieren der Kommunikationskosten des kürzesten Weges veranschlagt werden. Dabei hat sich für sie herausgestellt, dass der mehrstufige Kommunikationsweg am besten durch die Quadratic Path Length (QPL) modelliert wird. Jede Kommmunikation wird dadurch aufgrund ihrer verursachenden Kosten gewichtet (Kantenschnitt $\cdot$ Kosten im Kommmunikationsgraph). Somit wird beispielsweise eine größere Datenmenge für einen nah gelegen (schnell erreichbaren) Prozessor weniger Daten zu einem weit entfernten (nur langsam erreichbaren) vorgezogen.

Moulitsas und Karypsis beschreiben in [MK08] einen zweiphasigen Weg für die Berücksichtigung heterogener Architekturen. Bezeichnet wird dies als *predictor-corrector*-Ansatz, in dem zuerst nur anhand der Rechen- und Speicherressourcen eine Partitionierung be-

stimmt wird, um in der zweiten Phase das Kommunikationsnetzwerk mit einzubeziehen. Dabei werden die Kosten für Berechnung und Kommunikation nach der ersten Aufteilung bestimmt und iterativ korrigiert, bis das Optimum einer Zielfunktion erreicht wird. In der Korrektur-Phase werden ähnliche Algorithmen, wie bei der in Metis angebotenen Repartitionierung, verwendet. Die Zielfunktionen sind das maximale Kommunikationsvolumen bzw. die Gesamtausführungzeit. Damit werden in den Tests für verschiedene heterogene Systeme und unterschiedliche Graphen 10%-25% schnellere Partitionen als mit der normalen Metis-Partitionierung erreicht, die den Kantenschnitt minimiert. Dabei hat sich herausgestellt, dass das Ignorieren des Kantenschnittes in der zweiten Phase eine bessere Berücksichtigung des tatsächlichen Kommunikationsvolumens erzielt.
Neben den beiden vorgestellten Ansätzen für heterogene Systeme findet man weitere Methoden in den Bibliotheken MiniMax [KDB02] und PaGrid [HAB03].

3.2.2. Begegnung der Heterogenität in der Lastbalancierung

Bei der Lastbalancierung auf heterogenen Systemen steht das Finden angemessener Partitionsgrößen im Vordergrund. Jedem Prozessor soll ein so großer Anteil zugewiesen werden, dass alle Prozessoren in etwas zur gleichen Zeit die Berechnungen beenden. Kann man eine gute Abschätzung über die Leistung der Systemkomponenten geben, lassen sich die Größen für diese entsprechend gewichten. Moulitsas und Karypsis gehen bereits mit dem predictor-corrector-Ansatz dynamisch vor. Eine Alternative dazu stellt die Bibliothek DRUM (Dynamic Resource Utilazation Model) [Fai05] dar. Sie gehört zu dem bereits vorgestellten Projekt Zoltan [BDF+99] und kann dort von verschiedenen Graph-Partitionierern zur dynamischen Lastbalance eingesetzt werden, indem das System angesprochen wird, um neue Gewichte zur Partitionierung zu erhalten.
Die Bibliothek DRUM bietet ein Monitoring von Rechen-, Kommunikations- und Speicherfähigkeiten und fasst diese zu einer Leistungskennzahl zusammen. Das heterogene Zielsystem wird in Form einer Baum-Struktur repräsentiert, in der jedem Knoten Rechen- und Kommunikationsleistung zugeordnet wird. Dabei werden nicht nur Prozessoren, sondern auch Netzwerkknoten wie Router und Switches abgebildet, wodurch auch komplexe Grid-Systeme sehr genau beschrieben werden können. Das Monitoring beobachtet die Beanspruchung jedes Knotens. In die Bewertung der Rechenleistung fließen zudem der Anteil nicht genutzter Zeit sowie statische verfügbare Benchmark-Ergebnisse ein. Die Kommmunikationsleistung wird als verfügbare Bandbreite aus ein- und ausgehenden Daten-Paketen berechnet. Gewichtet ergeben beide Werte einen einzelnen Leistungswert für den Knoten. Zusammengefasst wird dem Zielprozessoren ein Leistungswert zugeordnet, anhand dessen eine neue Partitionierung vorgenommen werden kann, wobei dies nicht auf die Graph-Partitionierung beschränkt ist, sondern für jede Art der Lastbalancierung angewandt werden kann.

3.3. Bedeutung hybrider Systeme

In Bezug auf Rechenbeschleuniger in Clustern kann man nicht mehr nur von heterogenen, sondern muss von hybriden Systemen sprechen, denn die Prozessoren unterschieden sich zum Teil signifikant und bringen damit unterschiedliche leistungsbeeinflussende Faktoren mit. Damit wird erneut Frage 1 aufgeworfen: Wie stellt man verschiedene Prozessoren in Relation? Dies lässt sich nun nicht mehr mit der maximalen Rechenleistung beantworten. Vielmehr müssen die unterschiedlichen Leistungsaspekte vergleichbar gemacht und zusammengefasst werden. Ziel ist es auch für hybride Systeme eine Gewichtung zu finden, sodass ein lastbalancierter Zustand erreicht wird. Neben statischen Gewichtungen wird dabei auch ein dynamisches Nachkorrigieren nötig sein.

In diesem Zusammenhang besteht für den hybriden Supercomputer Roadrunner (bestehend aus dual-core Opteron CPUs und PowerXCell 8i Prozessen) ein Balancierungs-Verfahren für die SpMV in iterativen Lösern [SK09], das zum Abschluss als erster hybrider Balancierungsansatz vorgestellt werden soll:
Zu Beginn wird eine initiale Aufteilung anhand der maximalen Rechenleistung vorgenommen und diese aus den erzielten Leistungen in einer zweiten Gewichtung für die konkrete Aufgabe korrigiert. Damit wird ein laufzeitbasierter Ansatz gewählt. Im Anschluss wird diese Gewichtung in einem *trial-and-error*-Verfahren nachkorrigiert. Dieser letzte Schritt wird vorgenommen, um die zusätzliche Übertragungszeit zwischen dem Opteron und des Cells zwischen den Iterationen (Austausch von x, zurückschreiben des Ergebnisses y) zu berücksichtigen. Der Overhead, der für das Aufrechterhalten von effizienten DMA-Transfers nötig ist, wird zwar erwähnt, aber innerhalb des Ansatzes unter der Annahme, dass diese Zeit für die Gesamtlaufzeit bei genügend großer Iterationszahl vernachlässigbar klein ist, ignoriert. Die Ergebnisse zeigen, dass dieses Vorgehen innerhalb weniger Iterationen konvergiert.

4. Matrix-Partitionierung in LAMA

In LAMA soll eine Strategie zur Matrix-Partitionierung hinzugefügt werden, sodass die verteilte SpMV-Berechnung effizient erfolgt. Um die Randbedingungen zu klären wird in diesem Kapitel die Bibliothek vorgestellt. Auf eine Einordnung von LAMA in das Umfeld alternativer Projekte und einer Erklärung des Aufbaus der Bibliothek folgen Erläuterungen zu relevanten Implementierungs-Details. Abschließend wird sich mit der Qualität einer Partitionierung und den Einflussfaktoren darauf auseinandergesetzt.

4.1. Die Library of Accelerated Math Applications (LAMA)

Das am Fraunhofer Institut SCAI entwickelte Framework LAMA stellt Basisoperationen der linearen Algebra zur Verfügung. Im Fokus liegt dabei der Umgang mit großen, dünn besetzten Matrizen und Vektoren sowie die einfache Integration verschiedener Zielarchitekturen in verteilten Systemen. Der derzeitige Schwerpunkt liegt bei Grafikkarten, die entweder über CUDA oder OpenCL unterstützt werden. Bei der Entwicklung des Systems standen neben schnellen Rechenkernel und hoher Parallelität die leichte Erweiterbarkeit sowie einfache Benutzung im Vordergrund.

Innerhalb einzelner Compute-Backends werden die Operationen für die jeweilige Zielarchitektur optimiert. Für die parallele Berechnung auf Systemen mit verteiltem Speicher werden durch das Framework die Daten und die nötige Kommunikation während der Berechnung organisiert. Nach außen hin werden die Operationen in natürlicher mathematischer Text-Buch-Syntax bereitgestellt, wobei Multi-Precision und verschiedene Sparse-Matrix-Formate unterstützt werden. Zusätzlich werden auf den Basis-Operationen aufbauend einzelne iterative Löser angeboten.

Die Bibliothek ermöglicht somit das Schreiben leicht verständlichen Quellcodes, der ohne hardwarespezifische Programmierung auf verschiedenster Hardware ausgeführt werden kann. Zudem wird die Parallelisierung auf Thread- und Prozessebene übernommen sowie eine parallele hybride Berechnung unterstützt. Mit LAMA geschriebene Algorithmen können infolgedessen flexibel auf verschiedenen Systemen eingesetzt werden wobei alle zur Verfügung stehenden Systemressourcen ausgenutzt werden.

4.1.1. Ähnliche Arbeiten

LAMA platziert sich damit zwischen verschiedenen Basic Linear Algebra Subroutines (BLAS)- und Linear Algabra Package (LAPACK)-Bibliotheken im HPC-Bereich. Daneben

gibt es verschiedene Alternativen mit ähnlichem Ansätzen, aber unterschiedlichen Schwerpunkten. Eine Auswahl ähnlicher Projekte soll an dieser Stelle kurz vorgestellt werden.

HONEI - Hardware oriented numerics, efficiently implemented HONEI ist eine Open-Source-Sammlung von Bibliotheken mit denen eine Abstraktion von hardwareeigener Programmierung für numerische Berechnungen geschaffen wird [DGM+09]. Es werden CPUs, GPUs sowie die Cell B. E. unterstützt. Zum einem soll die effiziente Implementierung von hardwarespezifischen Kerneln ermöglicht werden, weil alle architekurabhängigen Infrastrukturen durch die Laufzeitumgebung bereitgestellt werden. Zum anderem soll HONEI genutzt werden um, auf den bereitgestellten Operationen aufbauend, Anwendungen zu programmieren, die flexibel über verschiedene Hardwaretypen ausgetauscht werden können und dabei automatisch effizient auf optimierte low-level Operationen zugreifen.

MTL4 - Matrix Template Library 4 Mit der MTL4 [mtl] wird die verteilte Berechnung von Basis-Operationen sowie verschiedenen Lösern der linearen Algebra auf verteilten CPU-Systemen ermöglich. Durch die einfache mathematische Schreibweise in C++ soll eine schnelle und problemfreie Programmierung geschaffen werden. Für die Verteilung der Daten werden verschiedene Methoden wie blockweise und zyklische angeboten; zudem besteht die Möglichkeit der Partitionierung mit Metis. Eine Ausweitung auf zusätzliche Unterstützung von GPUs ist in der Entwicklung.

ViennaCL ViennaCL [vie] ist ein OpenSource Projekt der TU Wien. Es bietet für Grafikkarten high-level Routinen in C++ für gängige Operationen der linearen Algebra an. Dazu wird die hardwareunabhängige Programmierung mit OpenCL genutzt. Unterstützt werden damit sowohl Nvidia als auch AMD Grafikkarten. Grundsätzlich sollte durch Open Computing Language (OpenCL) auch eine Ausführung auf dem Cell-Prozessor möglich sein. Auf den eigenen Routinen aufbauend werden iterative Löser angeboten, die präkonditioniert werden können. Als zusätzliche Features ist die Anbindung an ublas[ubl], eigen[eig] und die MTL4 zur alternativen Berechnung zu nennen sowie die Schnittstelle zu MatLab.

PetSc - Portable, Extensible Toolkit for Scientific Computation PetSc [BBB+10] bietet als Toolkit für wissenschaftliche Berechnungen, angefangen bei grundlegenden Operationen der linearen Algebra, alles über gängige Löser mit verschiedenen Präkonditionierern, bis hin zu Krylov-Unterraum-Methoden. Dabei ist eine MPI-basierte Parallelisierung sowie die Unterstützung von Nvidia GPUs (auch für Multi-GPU-Systeme) gegeben. Für die Partitionierung der Daten werden Anbindung an die Graph-Partitionierer Chaco, ParMetis und PT-Scotch angeboten. Wie die Gewichtung für die Partitionierung auszusehen hat muss jedoch vom Anwender selbst angebeben werden.

Von dieser Auswahl an Bibliotheken stellt PetSc die größte Bandbreite an Möglichkeiten bereit. So wird eine Vielzahl von Operationen angeboten, die verteilt auf CPU-GPU-basierten Systemen berechnet werden können. Die anderen Projekte bieten in den meis-

ten Fällen weniger Funktionen. Einzige Einschränkung bei PetSc ist, dass lediglich Nvidia GPUs und keine anderen Grafikkarten oder Rechenbeschleuniger unterstützt werden. Eine weite Unterstützung bieten dagegen HONEI und ViennaCL, dafür aber keine Möglichkeit zu verteilten Berechnung. Dies ist wiederum mit der MTL4 möglich, bei welcher derzeit noch jeglich GPU-Unterstützung fehlt. Zudem können mit PetSc für die verteilte Berechnung Verteilungen mit drei verschiedenen Graph-Partitionierern berechnet werden.
Mit LAMA wird ähnlich zu MTL4 und PetSc die Möglichkeit der verteilten Berechnung geschaffen. LAMA unterstützt aber neben Grafikkarten auch andere Rechenbeschleuniger dadurch, dass neben Compute Unified Device Architecture (CUDA) für nVidia Grafikkarten auch OpenCL genutzt wird. Für die Partitionierung soll in dieser Arbeit eine Anbindung an Metis [Kar] geschaffen werden. Zusätzlich zu dem, was PetSc mit der Graph-Partitionierung bietet, soll in LAMA die Gewichtung für die Partitionierung nicht vom Benutzer vorgegeben werden müssen, sondern von der Bibliothek gefunden werden.

4.1.2. Aufbau von LAMA

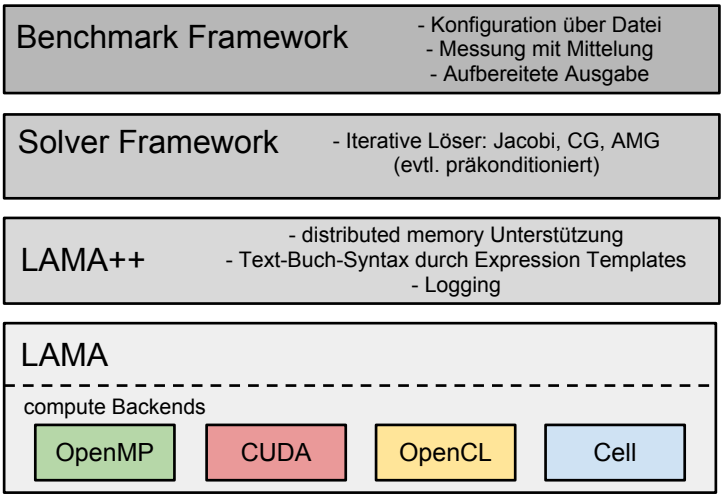


Abbildung 4.1.: Aufbau von LAMA

Die Bibliothek LAMA (s. Abbildung 4.1) setzt sich aus mehreren Schichten zusammen. Die Basis bildet die C-Schicht (LAMA) mit den integrierten Backends. In dieser Schicht wird die Verwaltungsarbeit gekapselt und die hardware-spezifische Programmierung[1] verborgen, während die Schnittstellen nach außen für die verschiedenen Zielsysteme identisch sind. Für Operationen auf dicht besetzten Matrizen werden die Aufrufe an entsprechend hochoptimierte BLAS-Bibliotheken, wie die MKL oder GOTO für CPU oder CUBLAS für CUDA, weitergereicht. Für verschiedene Sparse-Matrix-Formate und die jeweiligen Zielsysteme sind frameworkeigene Rechenkernel implementiert. Derzeit werden das CSR-,

[1] Als hardware spezifische Programmierung zählt für CPUs die OpenMP-Parallelisierung, für Rechenbeschleuniger im Allgemeinen die OpenCL-Programmierung, für CUDA-fähige Geräte im Speziellen die CUDA-Programmierung nach dem CUDA-Ausführmodell und für den Cell die SIMDsierung und Nutzung von Cell-Intrinsics.

COO-, ELL-, JDS- und DIA-Format unterstützt. Für eine Übersicht einer Großzahl existierender Formate sowie deren Darstellung sei auf [spa] verwiesen; die zwei im weiteren Verlauf relevanten Formate, CSR und ELL, werden in 4.2.3 näher erläutert.
Auf der C++-Schicht (LAMA++) wird durch die Einführung von Matrixklassen für dicht besetzte Matrizen, die verschiedenen Sparse-Matrix-Formate sowie von einem zugehörigen Satz aus Expression Templates die Verwendung der natürlichen, mathematischen Syntax geschaffen. Die mathematischen Ausdrücke werden durch die Expression Templates aufgelöst und rufen die entsprechenden Routinen in der ersten Schicht auf. Zusätzlich werden die Daten innerhalb der Matrixklassen für die verteilte Berechnung organisiert (Details dazu folgen in Absatz 4.2.2) und die Kommunikation vorbereitet.
Diese beiden ersten Schichten bilden das Grundgerüst des Frameworks. Darauf aufbauend befinden sich auf der dritten Schicht das Solver- und Benchmark-Framework. Das Solver-Framework stellt fertig implementierte und speziell optimierte Löser für lineare Gleichungssysteme zur Verfügung. Zum derzeitigen Repertoire von Lösern gehören Jacobi-, Conjugate Gradient (CG) und Algebraic Multigrid (AMG)-Verfahren. CG- und AMG-Verfahren können zusätzlich präkonditioniert werden.
Das Benchmark-Framework eignet sich zum einfachen Messen der Gesamtperformance der Operationen und Löser. Ein Benchmark kann verschieden konfiguriert auf den verschiedenen Zielsystemen ausgeführt werden. Die Ergebnisse erhält man in aufgearbeiteter Form.

4.2. Verteilte Berechnung in LAMA

Für die verteilte Berechnung von Matrix-Vektor- und Matrix-Matrix-Operationen sind zum einem die Daten, zum anderen die Einzeloperationen über das System zu verteilen. In der Regel sind einem Prozessor die Einzeloperationen zugeordnet, die zu seinen zugewiesenen Zieldaten gehören. Für die SpMV heißt dies, dass ein Prozessor der y_i speichert die Summe $\sum_{j=0}^{n} a_{ij} \cdot x_j$ berechnet. Dies wiederum bedeutet, dass sämliche Einträge in der Zeile i mit dem zugehörigen Eintrag im Vektor x multipliziert und die Ergebnisse aufsummiert werden. Je nachdem wie die Einträge von Matrix und Vektor auf den Prozessoren vorliegen, müssen die Vektor-Einträge kommuniziert werden.

4.2.1. Verteilungsmethoden

In Abschnitt 2.3 wurden bereits die möglichen Methoden zur Verteilung der Matrix-Einträge vorgestellt. In LAMA wird die zeilenweise Partitionierung genutzt. Zum einem ist dies die intuitivste Zuordnung und bringt dabei gegenüber der spaltenweisen Partitionierung keine Nachteile. Zum anderen wird im Gegensatz zur geblockten Partitionierung nur eine und nicht zwei Kommunikationsphasen eingeführt. Dies hat den Vorteil, dass die Berechnung in einem längerem Interval ohne Unterbrechung laufen kann. Welchen Vorteil dies birgt wird in Absatz 4.2.4 genauer erläutert.

Die Zuordnung von einer Zeile zu einem Prozessor/einer Partition kann in zahlreichen Kombinationen erfolgen. LAMA stellt dafür die folgenden vier Verteilungen bereit:

Blockweise Verteilung Mit der blockweisen Verteilung wird jedem Prozessor ein zusammenhängender Zeilenblock der Länge $\lfloor \frac{\text{Anzahl Matrixzeilen}}{\text{Anzahl Prozessoren}} \rfloor$ (dem letzten Prozessor den zusätzlichen Rest, welcher durch Rundung verloren gegangen ist) zugewiesen. Der Unterschied von den ersten Partitionen zu der letzten kann daher maximal $p - 1$ betragen.

Zyklische Verteilung Bei der zyklischen Verteilung wird eine Blockgröße (*chunksize*) vorgegeben, nach der die Matrix in Zeilenblöcke eingeteilt wird. Diese Blöcke werden zyklisch an die Prozessoren verteilt. Bei p Prozessoren erhält der erste Prozessor somit Block $1, p + 1, 2p + 1$, usw. Ähnlich zur Blockverteilung erhalten hier alle Prozessoren in etwa die gleiche Anzahl an Zeilen (sofern die Blockgröße um einiges kleiner als die Anzahl Matrixzeilen ist). Die Abweichung zwischen den Partition kann daher maximal *chunksize* groß sein.

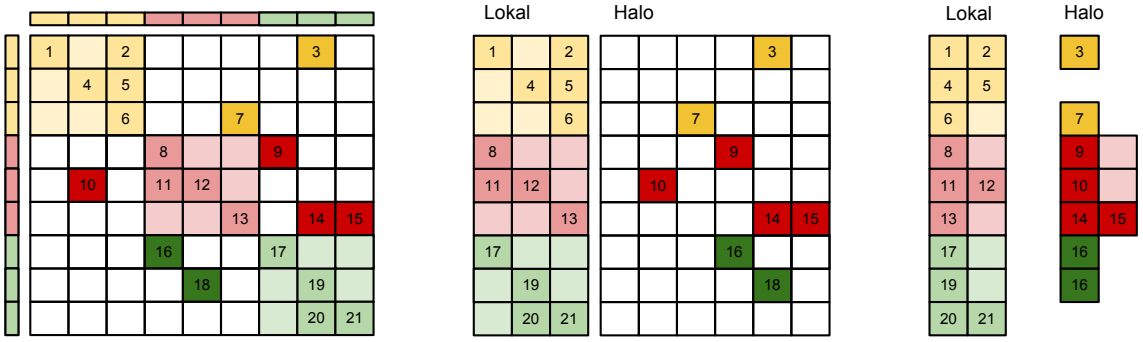
Allgemeine Block-Verteilung Der allgemeinen Block-Verteilung werden die Größen für die einzelnen Partitionen übergeben. Damit kann jeder Prozessor beliebig viele Zeilen erhalten. Einzige Vorraussetzung ist, dass insgesamt die Summe der Blockgrößen der Anzahl der Matrixzeilen entsprechen muss. Die Blöcke werden zusammenhängend verteilt.

Allgemeine Verteilung Durch die Übergabe eines Partitionsvektors wird mit dieser Verteilung jeder Zeile ein Prozessor zugeordnet. Damit ist hier die höchste Flexibilität gegeben. Den Prozessoren können unterschiedlich viele Zeilen zugeordnet werden, wobei die Zeilen beliebig gewählt werden können und damit nicht zusammenhängend sein müssen.

Von Matrix zu Matrix und abhängig von dem dahinterliegenden Problem kann die eine oder andere Verteilung sinnvoller sein. Für Matrizen, bei denen die Anzahl der Einträge pro Zeile über alle Zeilen ungefähr gleich groß ist, erhält man mit zusammenhängenden Zeilenblöcken Verteilungen mit vorhersagbar vielen Einträgen pro Partition, was die erste Vorraussetzung für eine gute Lastbalance ist. Variiert die Anzahl der Einträge pro Zeile dagegen stark wird eine schlecht balancierte Verteilung erstellt. Eine zyklische Verteilung mit ausreichend kleiner chunksize kann dies i. d. R. gut ausgleichen. Mit den allgemeinen Verteilungen kann eine Gewichtung angegeben werden und damit erstmals für unterschiedlich leistungsstarke Prozessoren verschieden große Partitionen erstellt werden. Bei der allgemeinen Block-Verteilung besteht wieder das Problem mit der schlechten Balancierung bei einer unregelmäßigen Anzahl Einträge pro Zeile. Mit der allgemeinen Verteilung kann diese berücksichtigt werden. Dies bedingt allerdings eine genaue Analyse der Matrixstruktur. Hier wird die zu entwickelnde Strategie ansetzen.

4.2.2. Organisation der Daten

Für einen einzelnen Prozessor liegen die ihm zugeteilten Zeilen unabhängig von der Verteilungsmethode in zusammenhängender Form vor und stellen sich ihm als vollständige Matrix/vollständigen Vektor dar. Zusätzlich hat eine Matrix neben der Verteilung der Zeilen, die die Zeilen den Prozessoren zuweist, auch eine spaltenweise Verteilung. Denn auch der Vektor bei der SpMV hat eine zeilenweise Verteilung. Über die Spaltenverteilung wird damit beschrieben welche Vektor-Einträge für die Operation direkt vorliegen und welche vorerst kommuniziert werden müssen. Der Teil der Matrix, für den die zugehörigen Einträge zur Berechnung lokal vorliegen, wird im folgenden als lokaler Anteil, der Teil, für den Einträge kommuniziert werden müssen, als Halo-Anteil bezeichnet.
In LAMA werden beide Teile getrennt voneinander als eigenständige Matrizen gespeichert. Für die Matrix des Halo-Anteils wird lediglich noch ein zusätzlicher Index-Vektor über die gefüllten Zeilen gespeichert. Da der Halo-Anteil der Matrix idealerweise wenig gefüllt sein sollte, werden nicht in jeder Zeile Daten vorliegen. Somit muss nicht über alle Zeilen der Matrix iteriert werden, sondern nur über diejenigen, in denen Einträge vorhanden sind. Der gesamte Zusammenhang der Zeilen- und Spaltenverteilung wird in Abbildung 4.2 gezeigt.



(a) Verteilung der Matrix

(b) Aufteilung in lokalen und Halo-Anteil

(c) Speicherung in LAMA

Abbildung 4.2.: Darstellung der verteilten Matrix in LAMA; (a) vollständige Matrix auf 3 Partitionen verteilt, die gefärbten Verteilungen links neben und über der Matrix entsprechen der zeilen- und spaltenweisen Verteilung der Matrix sowie der Verteilung der Vektoren in der SpMV, dunkel hervorgehobene Einträge müssen für die Berechnung vorerst kommuniziert werden; (b) Sicht auf die Matrix als lokalen und Halo-Anteil; (c) tatsächlich gespeicherte Daten in LAMA im ELL-Format

Die Daten werden in frameworkeigenen Datenstrukturen (`LAMA-Arrays`) gespeichert. Diese wiederum in `Kontexten` verwaltet, sodass für die verteilte hybride Speicherung in den verschiedenen Speicherbereichen der Zielarchitekturen (CPU, GPU, Cell) keine Inkonsistenzen entstehen können. Dazu besteht die Möglichkeit vom mehrfachen Lese-, aber nur einfachen Schreibzugriff auf die Daten. Für die Kommunikation werden mit der Speicherung der Matrix `Kommmunikationspläne` aufgebaut, welche die ein- und ausgehenden Datenabhängigkeiten organisieren. Dadurch steht zur Zeit der Berechnung einer Operation fest welche Daten an wen verschickt werden müssen und von wem welche Daten empfangen werden.

4.2.3. verwendete Sparse-Matrix-Formate

Die Speicherung innerhalb des lokalen und Halo-Anteils erfolgt entsprechend dem verwendeten Sparse-Matrix-Format. Für die spätere Leistungsbetrachtung von CPU und GPU wird der Zugriff auf die Daten aufgrund von unterschiedlichem Zugriffsverhaltens relevant. Deshalb werden nachfolgend die zwei verwendeten Formate CSR und Ellpack (ELL) erläutert. Ihre Darstellung wird anhand der folgenden Beispielmatrix M veranschaulicht.

$$M = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 0 \\ 3 & 4 & 0 & 5 & 0 & 0 \\ 6 & 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 10 & 11 & 0 & 0 \\ 0 & 0 & 0 & 0 & 12 & 0 \\ 0 & 13 & 0 & 0 & 0 & 14 \end{pmatrix}$$

Abbildung 4.3.: Beispielmatrix M

CSR

CSR steht für Compressed Sparse Row und ist das wohl verbreiteste Speicherformat für dünn besetzte Matrizen, da es keine Annahmen über die Struktur der nicht-null-Einträge macht und dabei keinen zusätzlichen fill-in[2] hat.
Die nicht-null Werte werden zeilenweise in einem Vektor *data* abgelegt. Parallel dazu wird in dem Vektor *ja* zu jedem Wert die Spaltenposition gespeichert. Durch die zeilenweise Speicherung ist ein effizienter Zugriff für die SpMV gegeben. Ein dritter Vektor *ia* referenziert den Index des Elements, an dem eine neue Zeile beginnt. Der Speicheraufwand dieses Formates beträgt $m \cdot IndexSize + nnz \cdot (IndexSize + DataSize)$. Dabei steht nnz für die Anzahl der nicht-null-Einträge und m für die Anzahl der Zeilen.
Für die Matrix M sieht dies wie folgt aus:

data	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ja	1	4	1	2	4	1	3	4	6	3	4	5	2	6
ia	1	3	6	10	12	13	15	-	-	-	-	-	-	-

Tabelle 4.1.: CSR-Darstellung der Beispielmatrix M

Ellpack (ELL)

Das Ellpack-Format ist vorzugsweise für dünn besetzte Matrizen geeignet, bei der die Anzahl der nicht-null-Einträge pro Zeile nicht stark variiert, da es eine strenge Form hat und zur Erhaltung dieser explizit Nullen speichert (fill-in). Bei der Form handelt es sich um eine rechteckige (gestauchte) Matrix (D), der Dimension m (Anzahl Zeilen) $\times$ max(Einträge pro Zeile). Zeilenweise werden die Einträge der Ursprungsmatrix darin abgelegt und evtl.

[2] Der fill-in bezeichnet innerhalb einer Sparse-Matrix Nullen, die zur Formerhaltung des Speicherformates beitragen.

am Ende der Zeile mit Nullen aufgefüllt. Daneben existiert eine entsprechende Matrix (J), die die Spaltenindizes verwaltet. Diese beiden Matrizen werden als spaltenweise Arrays (*data* und *ja*) im Speicher abgelegt[3]. In LAMA wird zusätzlich ein weiteres Array (*ia*) mit der Anzahl der Einträge pro Zeile geführt, um die Null-Einträge bei der Berechnung zu überspringen. Für die Beispielmatrix M ergeben sich demnach folgende Matrizen und Vektoren.

$$D = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 5 & 0 \\ 6 & 7 & 8 & 9 \\ 10 & 11 & 0 & 0 \\ 12 & 0 & 0 & 0 \\ 13 & 14 & 0 & 0 \end{pmatrix} \qquad J = \begin{pmatrix} 1 & 4 & 0 & 0 \\ 1 & 2 & 4 & 0 \\ 1 & 3 & 4 & 5 \\ 3 & 4 & 0 & 0 \\ 5 & 0 & 0 & 0 \\ 2 & 6 & 0 & 0 \end{pmatrix}$$

Abbildung 4.4.: Matrix D und J für die Beispielmatrix M

data	1	3	6	10	12	13	2	4	7	11	0	14	0	5	8	0	0	0	0	0	9	0	0	0
ja	1	1	1	3	5	2	4	2	3	4	0	6	0	4	4	0	0	0	0	0	6	0	0	0
ia	2	3	4	2	1	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Tabelle 4.2.: ELL-Darstellung der Beispielmatrix M

Der Speicheraufwand ($IndexSize + m \cdot max$(Einträge pro Zeile)$\cdot (IndexSize + DataSize)$ $[+m \cdot IndexSize]$) hängt stark von der Struktur der Matrix ab. Besitzt eine einzelne Zeile wesentlich mehr nicht-null-Einträge, als andere Zeilen, vergrößert sich der Speicheraufwand immens. Der Vorteil dieses Formates ist es, dass durch die Einführung der recheckigen Struktur eine spaltenweise Speicherung ermöglicht wird, die für die GPU einen coalesced memory access bewirkt.

4.2.4. Abarbeitungsmodell der SpMV

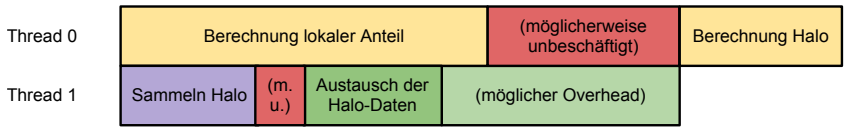
Die Intention der getrennten Speicherung von lokalen und Halo-Anteil der Matrix ist es, die nötige Kommunikation für den Halo-Anteil der Matrix gesammelt und nicht auf Anforderung durchführen zu können. Damit fällt für die Übertragung der Halo-Einträge nur einmal Latenzzeit an. Viel wichtiger ist aber die Auswirkung auf die Abarbeitung der Einzeloperationen. Mit der Aufteilung in einzelne Matrizen kann jede (Halo-Matrix nach der Kommunikation) ohne Unterbrechung durch Kommunikation abgearbeitet und damit die höchst mögliche Leistung des Prozessors erzielt werden. Zudem sind heutige CPUs multithreadingfähig und GPUs verarbeiten mehrere Streams, sodass Kommunikation und Berechnung asynchron ausgeführt werden können. Idealerweise ist die Kommunikationszeit kürzer als die Berechnungszeit des lokalen Teils und wird somit vollständig verdeckt,

[3] Generell ist genauso eine zeilenweise Speicherung der beiden Matrizen möglich. Die spaltenweise Speicherung bringt für die Berechung auf GPUs aber Laufzeitvorteile, da der Zugriff auf die Daten uncoalesced ist.

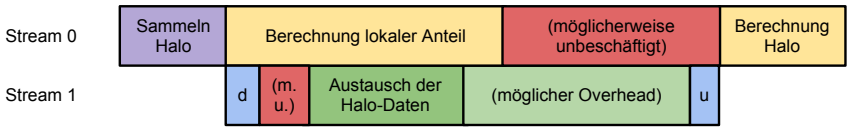
sodass die Gesamtlaufzeit nicht durch die Kommunikation beeinflusst wird. Somit sollte dieselbe Leistung erzielt werden, wie wenn alle Daten des Vektors x auf dem Prozessor vorliegen. Allerdings entsteht die zusätzliche Aufgabe, die Daten für die Kommunikation zusammenzusammeln, sodass nur die relevanten Daten verschickt werden. Weitere Einflussfaktoren auf die Berechnungszeit werden in Absatz 4.3.1 näher erläutert. Den wesentlichen Unterschied zwischen der synchronen und asynchronen Ausführung für CPU und GPU wird in Abbildung 4.5 dargestellt.

(a) Synchrones Abarbeitungsmodell CPU

(b) Synchrones Abarbeitungsmodell GPU



(c) Asynchrones Abarbeitungsmodell CPU



(d) Asynchrones Abarbeitungsmodell GPU

Abbildung 4.5.: Abarbeitungsmodelle für die verteilte SpMV-Berechnung in LAMA; m.u.: möglicherweise unbeschäftigt; d: download von GPU auf CPU; u: upload von CPU auf GPU

Im synchronen Fall werden die einzelnen Aufgaben auf CPU und GPU in gleicher Weise ausgeführt, die GPU sollte dabei aber i. d. R. in der Berechnung und dem Sammeln der benötigten Halo-Daten für die Kommunikation schneller sein. Die Kommunikationszeit hängt vom Netzwerk und der Menge der zu übertragenden Daten ab. Außerdem spielt die Synchronität zwischen den parallelen Prozessen eine Rolle. Nach dem MPI-Kommmunikations-Modell müssen beide Kommunikationspartner gleichzeitig empfangs- bzw. sendebereit sein. Dadurch kann es für einen Prozess nötig sein auf die Übertragung warten müssen. Dies führt zu längeren Kommunikationszeiten, was den weiteren Verlauf beeinflussen kann. Unabhängig von Rechen- und Kommunikationleistung ist mitzunehmen, dass alle Operationen nacheinander ausgeführt werden und ein Prozessor durch die Kommunikation möglicherweise eine Zeit lang unbeschäft sein kann, weil auf einen anderen gewartet werden muss.
Bei der asynchronen Ausführung unterscheiden sich CPU und GPU. Der prinzipielle Ansatz für beide Zielsysteme ist es, Berechnung und Kommunikation in verschiedenen Threads bzw. Streams zu verarbeiten, um die Kommunikation hinter der Berechnung zu verstecken. Da für den lokalen Teil der Matrix alle nötigen Daten von Beginn an vorhan-

den sind, kann diese Berechnung unabhängig von der Kommunikation ausgeführt werden. Dazu parallel soll die Kommunikation für den Halo-Teil der Matrix erfolgen. Die Berechnung für den Halo-Teil der Matrix kann erst nach einer abgeschlossenen Kommunikation beginnen. Der Unterschied für CPU und GPU liegt darin, dass das Sammeln der benötigten Halo-Daten auf der GPU nicht parallel zur lokalen Berechnung ausgeführt werden kann, weil auf denselben Daten operiert wird. Somit muss diese Operation vor der lokalen Berechnung erfolgen, damit die Kommunikation parallel zu dieser stattfinden kann. Essentieller Unterschied zur synchronen Ausführung ist, dass noch einmal zusätzliche unbeschäftigte Intervalle eingeführt werden, wenn die Kommunikation länger als die lokale Berechnung dauert. Genau diese gilt es zu vermeiden.

4.2.5. Parallele Verarbeitung

Nicht nur bei der Verteilung der Daten über die Prozessoren, sondern auch bei der Parallelisierung innerhalb eines Prozessors, erfolgt die Aufteilung der Arbeit auf Basis der Zeilen. Dabei geht es vornehmlich um eine gleichmäßige Aufteilung der Rechenlast für eine vollständige Auslastung des Prozessores. Datenabhängigkeiten, wie bei der Prozessorübergreifenden Parallelisierung, müssen nicht beachtet werden, da alle Threads auf denselben Speicher zugreifen können.
Für die CPU findet die Parallelisierung über das OpenMP-Modell statt, indem die Schleife über die Zeilen parallelisiert wird. Für das Scheduling der Iterationen stehen die Strategien `static`, `cyclic` und `dynamic` zur Auswahl. Mit den Strategien kann die Last auch bei unterschiedlich langen Zeilen mit Kenntniss über die Verteilung der Zeilenlängen balanciert werden.
Mit dem CUDA-Modell werden die Zeilen einzelnen Threads zugeordnet, die wiederum in Blöcke zusammengefasst werden. Die Blöcke werden dann von den SMs verarbeitet. Die Zuordnung der Zeilen zu den Threads erfolgt kontinuierlich über die Position des Threads innerhalb des Gitters. Die Reihenfolge der Verarbeitung liegt bei der CUDA-Runtime. Eine Balance zwischen den Blöcken wird durch das Scheduling erzielt. Innerhalb eines Blockes muss die Balance aber gegeben sein oder einzelne Threads müssen auf die Beendigung aller Threads des Blockes warten. Deswegen sind stark variierende Zeilenlängen innerhalb des Blockes ungünstig für die Gesamtleistung.

4.3. Diskussion der Qualität der Matrix-Partitionierung

Die Aufteilung einer dünn besetzten Matrix kann auf vielfältige Art und Weise erfolgen. Zum einem unterscheiden sich zeilen-, spalten- oder blockweise Partitionierung der Einträge. Daneben gilt es die Größe der einzelnen Partitionen festzulegen und sie entsprechend auf die Verteilung zu übertragen. Die Implementierung in LAMA gibt eine zeilenweise Partitionierung vor. Entscheidungskriterium dafür war, dass nur eine Kommunikationsphase während der Berechnung nötig ist und diese mit asynchroner Kommunikation, die

für aktuelle Systeme i. d. R. möglich ist, hinter der lokalen Berechnung versteckt werden kann. Hendrickson und Kolda [HK99b] bestätigen, dass eindimensionale Partitionierungen die Komplexität des Lastbalancierungs-Problems auf heterogenen Systemen reduzieren.

Die Aufgabe des zu entwickelnden Modells wird es sein, die Größe bzw. Gewichte der Partitionen anzuberaumen, sodass die Last einer Partition der Leistung des zugehörigen Prozessores gerecht wird. Im Modell des ungerichteten Graphen wird die Last der kleinsten aufzuteilenden Einheit als Knotengewicht angenommen. Bei der zeilenweisen Partitionierung ist diese Einheit eine Zeile und die Last ist dabei die Anzahl der Einträge pro Zeile. Mit der Graph-Partitionierung sollen die Partitionen, wie sie von den Gewichten vorgegeben sind, erstellt werden, sodass dabei möglichst wenig Kommunikation auftritt. Das heißt, dass die Kommunikationszeit eines Prozessors kleiner als die lokale Berechnung sein sollte. Wie aus Formel 3.1 bekannt, setzt sich die Kommunikationszeit aus Latenzzeit und Bandbreite zusammen, damit ist neben der Menge der Daten auch die Anzahl der Nachrichten wichtig. Insgesamt sind es folgende Kriterien, die mit der Graph-Partitionierung optimiert werden:

die Anzahl der Einträge in einer Partition Da die Anzahl der Einträge pro Zeile in einer dünn besetzten Matrix je nach Struktur geringfügig bis stark variiert, kann bei einfachen Partitionierungsmethoden, die sich an der Zeilenanzahl orientieren, keine genaue Lastbalance bezüglich der Einträge und damit des tatsächlichen Rechenaufwandes erfolgen. Erst mit einer Berücksichtigung der genauen Anzahl Einträge pro Zeile, wie bei der Graph-Partitionierung, kann die Aufteilung entsprechend vorgenommen werden. Dabei behält sich auch diese vor, eine geringe Inbalance zugunsten des Kantenschnittes zu erlauben.

die Menge der ein- und ausgehenden Daten Ebenfalls abhängig von der Struktur der Matrix verteilen sich die Einträge über den lokalen und Halo-Anteil einer Partition. Demnach sind die Abhängigkeiten zwischen den Zeilen der Partitionen entscheidend für die Größe des Halos und damit der zu kommunizierenden Daten. Die Graph-Partitionierung setzt mit der Metrik des Kantenschnittes an dieser Stelle an und minimiert die Anzahl der Einträge, die kommuniziert werden müssen. Mit der blockweisen und zyklischen Partitionierung kann dies nicht erreicht werden, da die genaue Struktur der Matrix i. d. R. keine Beachtung findet. Für Matrizen mit bekannten geometrischem Hintergrund können sie gute Partitionierungen liefern, wenn die Schnitte aufgrund der Geometrie die Abhängigkeiten minimieren. Da sich in diesen Matrizen aber ein wiederholendes Muster befindet, müsste dies mit den Positionen der Schnitte und damit mit der Gewichtung der Partitionen vereinbar sein.

die Anzahl der ein-/ausgehenden Nachrichten Neben der Menge der zu kommunizierenden Daten ist auch die Anzahl der Nachrichten, auf die sie aufgeteilt sind, bedeutsam, da die Latenzzeit für eine Nachricht relevant werden kann. Mit der Minimierung der Anzahl der Elemente wird i. d. R. auch die Anzahl der Nachrichten minimiert. Für

ein System, in dem der Latenzzeit eine größere Bedeutung als der Bandbreite zugetragen wird, kann es aber auch vorteilhaft sein diese Größe über die Metrik des Grenzschnittes zu minimieren.

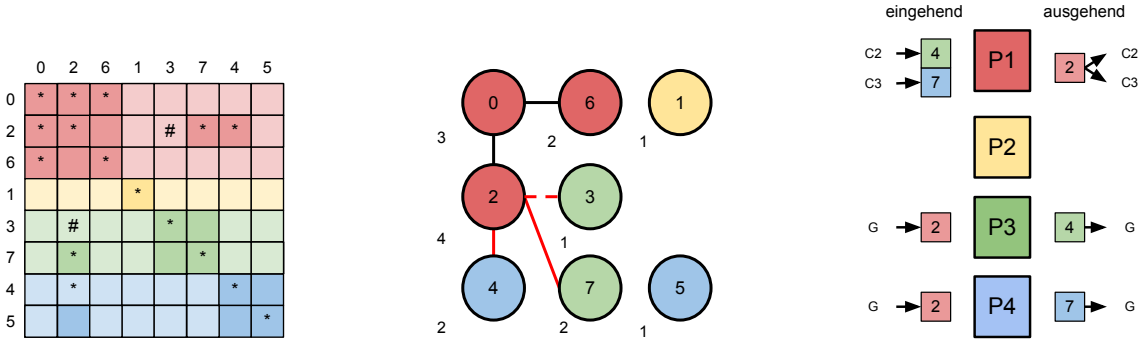
Es stellt sich aber die Frage, ob diese Ziele die Anzahl der Einträge entsprechend der Gewichte zu verteilen und die Datenmenge anhand des Kantenschnittes zu minimieren, zielführend für die Aufteilung (auf einem hybriden System) ist.

4.3.1. Lastbalancierung

Durch die Graph-Partitionierung erfolg die Aufteilung der Zeilen so, dass die Anzahl der Einträge in den Partitionen (innerhalb des Rahmens des Inbalance-Faktors) den Gewichten entsprechen. Unter der Annahme, dass die Berechnung für jeden Eintrag dieselben Kosten verursacht bzw. dieselbe Zeit braucht, ist die Gesamtlast gemäß der Leistung der Prozessoren gleichmäßig aufgeteilt. Für die Berechnung spielt aber der Zugriff auf die Daten ebenso eine Rolle, sodass neben der reinen Anzahl der Einträge auch deren Lokalität für ein gutes Cache-Verhalten wichtig ist. Es können andere Hits und Misses auf die Caches auftreten und sich dadurch maßgeblich auf die Zugriffszeiten auswirken. Zudem unterscheidet sich das Verhalten für CPUs und GPUs, da der Texture Cache Speicherblöcke hält und keine Zeilen (vlg Abbildung 5.3).
Zusätzlich dazu kann für die GPU Leistung verloren gehen, weil sie unterschiedlich lange Zeilen zugewiesen bekommen hat. Denn müssen die Threads innerhalb eines Blockes aufeinander warten, anstatt dass andere Berechnungen durchgeführt werden. Viele kurze bzw. lange Zeilen bedeutet auch einen Unterschied in der Anzahl der Zeilen die ein Prozessor zugewiesen bekommen hat. Für die CPU kostet das Durchlaufen der Zeilen pro Zeile einen konstanten Overhead (Schleifenoverhead [LE08]), sodass weniger Zeilen weniger Overhead bedeutet. Auf der GPU besteht dabei eher die Gefahr, dass dadurch zu wenig Parallelität vorhanden ist um durch Tauschen der Threads die Latenzzeiten auf den Speicher zu verdecken.
In jedem der erwähnten Fälle liegt nach der Verteilung eine balancierte vor, weil die Last, so wie sie definiert wurde gleichmäßig aufgeteilt wurde, jedoch können die Laufzeiten etwas anderes zeigen.

4.3.2. Kommunikationsoverhead

Mit der Metrik des Kantenschnittes soll das Kommmunikationsvolumen für die Berechnung minimiert werden. Diese Metrik wird aber im Zusammenhang mit dem Modell des ungerichteten Graphen von vielen Autoren [cA99, VB05, TK06] kritisiert, weil diese Metrik nicht das exakte Kommmunikationsvolumen abbildet. Dies soll stückweise an Abbildung 4.6 erklärt werden.



(a) Partitionierte Matrix **(b)** Partitionierter Graph **(c)** Kommunikationsplan

Abbildung 4.6.: Kommunikationsplan für eine partitionierte Matrix; die zwei geschnittenen Kanten im Graphen sind rot hervorgehoben; kommuniziert werden müssen aber 3 verschiedene Elemente in 4 Nachrichten; mit dem zusätzlichen Eintrag (#) wird eine Kante mehr geschnitten, es muss aber nicht mehr kommuniziert werden

Für die gegebene Matrix werden bei der gewichteten Aufteilung auf vier Partitionen zwei Kanten (durchgängige rote Linie) geschnitten. Damit trifft das Modell die Aussage, dass das Kommmunikationsvolumen vier ist, weil jede Abhängigkeit die geschnitten wurde zum jeweils anderen Prozessor kommuniziert werden muss. Dies stimmt in diesem Fall, da Prozessor 1 zwei Nachrichten verschicken muss, Prozessor 3 und 4 jeweils eine Nachricht. Kommt ein zusätzlicher Eintrag (#) mit der Abhängigkeit zwischen Knoten 2 und 3 (gestrichelte (rote) Linie) dazu, erhöht sich der Kantenschnitt auf drei und das prognostizierte Kommunikationsvolumen dementsprechend auf 6. Am der tatsächlich zu kommunizierenden Datenmenge ändert sich aber nichts, da die Abhängigkeit des Eintrages 2 zu Prozessor 3 nur einmal kommuniziert werden muss. Jede Vielfachheit eines Eintrages innerhalb einer Partition wird demnach zuviel gezählt.
Mit dem Grenzschnitt oder dem bipartiten bzw. Hypergraph-Modell würde dieser Fehler nicht gemacht werden. Jedoch fällt der Fehler i. d. R. vergleichweise klein aus, da durch die dünn besetzte Struktur der Fall der doppelten Einträge in einer Spalte selten vorkommt. Hendrickson und Kolda [HK99a] bestätigen, dass der Kantenschnitt gerade für Probleme, die aus Differenzialgleichungen hervorgehen, nicht schlecht abschneidet, weil der Radius in dem die Anzahl der Nachrichten liegen kann eher klein ist. Somit kann davon ausgegangen werden, das die Minimierung des Kommmunikationsvolumens für eine versteckte asynchrone Kommunikation zumindest für die verwendeten Poisson-Matrizen ausreicht.

5. Partitionierungsstrategie für hybride Systeme

Bei der Aufgabe, eine dünn besetzte Matrix über ein hybrides System zu verteilen, gibt es diverse Faktoren, die berücksichtigt werden müssen. Zum einem beeinflusst die Matrixstruktur die Qualität einer Partitionierung und die Größe des Kommunikationsvolumen. Zum anderen muss die Größe der Partitionen den Prozessorleistungen angemessen sein.

In Kapitel 2 wurde in das Problem der Matrix-Partitionierung für dünn besetzte Matrizen eingeführt. Es wurde erläutert, dass durch die Graph-Partitionierung eine Verteilung bestimmt werden kann, die die unregelmäßige Struktur der Matrix berücksichtigt, so dass Lastbalance bzgl. der Berechnung und eine Minimierung der Kommmunikation erzielt wird. Damit wird die Matrixstruktur für die Qualität der Partitionierung und für die Minimierung des Kommmunikationsaufwandes einbezogen. Für homogene Systeme sollte mit einer gleichmäßigen Partitionierung so ein lastbalancierte Konstellation erreicht werden.

Bei heterogenen Systemen kommt hinzu, dass die verschiedenen Prozessoren unterschiedliche Leistungen aufweisen und die Größen der Partitionen dementsprechend angepasst sein müssen. Solange es sich bei den Prozessoren nur um CPUs handelte, die denselben grundsätzlichen Aufbau haben, ließen sich die Systemkomponenten auf Basis ihrer Maximalleistung beurteilen und vergleichen. Mit der Hinzunahme von GPUs in ein System gilt es zudem noch eine Bewertungsgrundlage zu finden, die die Leistung von CPUs und GPUs in Relation zueinander stellt. Dabei muss bei der Leistungsbewertung genauer auf die Hardwaregegebenheiten geachtet werden. Insbesondere die Leistung einer GPU ist von vielen Faktoren abhängig. Je nach Operation (und Implementierung) kann die Leistung der GPU verschieden gut umsetzt werden. Hinzu kommt, dass von der reinen Rechenleistung die Zeit für das Starten der Berechnung auf der GPU sowie für zusätzliche Kommmunikation zwischen GPU und CPU abgezogen werden muss. Dies muss im Gesamtvergleich zur CPU berücksichtigt werden. Somit lässt sich kein allgemeines Modell für alle Matrix-Vektor- und Matrix-Matrix-Operationen oder alle Rechenbeschleuniger aufstellen. Das Modell sollte aber entsprechend darauf übertrag- und erweiterbar sein können. Außerdem muss es alle Bewertungsfaktoren in einem einzelnen Wert repräsentieren, da die Graph-Partitionierung, wie sie in derzeitigen Bibliotheken angeboten wird, nur einzelne Gewichte für die Partitionen berücksichtigt. Wie später genauer erläutert wird, bietet sich dazu am besten die Laufzeit. Es gilt also ein Modell zur Laufzeitvorhersage zu entwickeln.
Die Vorhersage einer genauen Laufzeit ist durch vielfältige Einflussfaktoren, deren Details teilweise unbekannt sind (bspw. die Verdrängungsstrategien eines Caches), nicht exakt.

Doch mit einer annähernd guten Angabe des Verhältnis der Leistung verschiedener Prozessoren sollte eine initiale Gewichtung der Partitionen gefunden werden. In einer anschließenden dynamischen Balancierung kann der durch die erwähnten Ungenauigkeiten entstandene Fehler korrigiert werden. Im Folgenden soll zuerst noch einmal die Problemstellung zusammengefasst werden, bevor die einzelnen Teile des Modells vorgestellt und in einem Gesamtmodell zusammengeführt werden.

5.1. Problemstellung

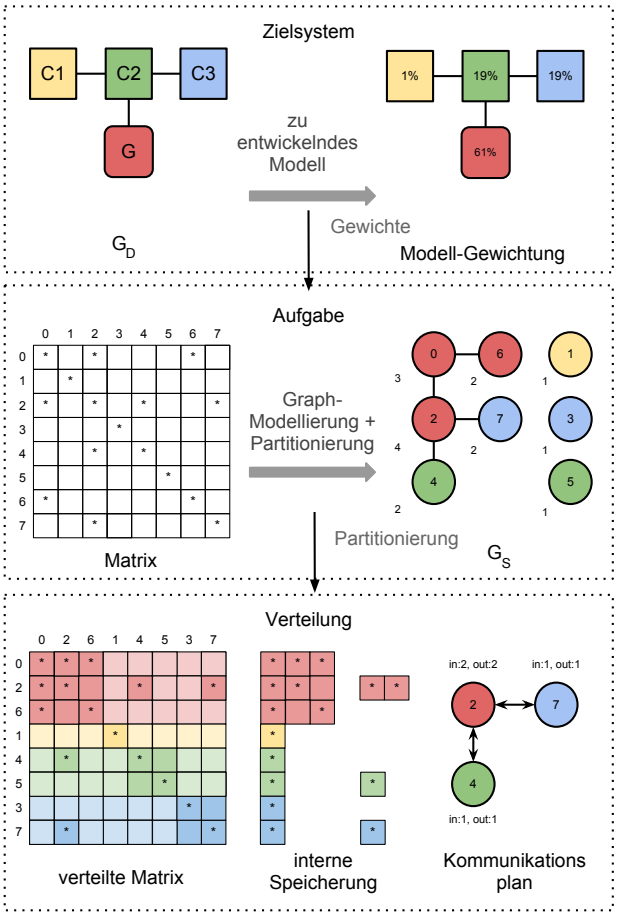


Abbildung 5.1.: Prozess der Lastverteilung für hybride Systeme; Zielsystem: Archtitektur des Zielsystems (G = GPU, C = CPU) sowie dessen Gewichtung nach dem Modell; Aufgabe: zu verteilende Matrix sowie deren partitionierte Graphdarstellung; Verteilung: resultierende Verteilung der Matrix sowie deren interne Darstellung in LAMA (GPU im ELL-Format (Padding), CPUs im CSR-Format) und Kommunikationsmuster

Die Aufgabe der Lastverteilung besteht darin, angemessen große Teilaufgaben an die Prozessoren des Zielsystems zu verteilen. Ziel dabei ist es, die einzelnen Ausführzeiten zu balancieren und bei der Aufteilung wenig zusätzliche Last (Overhead) zu erzeugen. Unter der Verteilung versteht man eine Zuordnung von der Berechnung (den Matrix-Einträgen) zu den Prozessoren. Wie die Matrix bei der SpMV-Berechnung auf einen Graphen $\mathcal{G}_S(w_{v_S}, w_{e_S})$ abgebildet wird ist aus Sektion 2.4.2 bekannt. Dabei wird eine zei-

lenweise Partitionierung der Matrix verwendet. Die Knotengewichte w_{v_S} entsprechen der Anzahl der Matrix-Einträge in der Zeile und die Kantengewichte w_{e_S} werden als *eins* angenommen, weil zu jedem Element genau eine Abhängigkeit vorliegt.
Die Architektur des Zielsystems kann ebenso als Graph $\mathcal{G}_D(w_{v_D}, w_{e_D})$ beschrieben werden. Jeder Knoten vertritt einen Prozessor, jede Kante eine Verbindung im Netzwerk. Die Gewichte geben Auskunft über die Rechen-(w_{v_D}) und Kommmunikationsleistung(w_{e_D}) der jeweiligen Prozessoren.
Es ist ein Modell zur Gewichtung des Zielsystems zu entwickeln, sodass mit der berechneten Gewichtung mit Hilfe der Graph-Partitionierung eine Verteilung der Matrix gefunden werden kann, die eine balancierte Berechnung mit wenig Kommunikation aufweist.
Das Gesamt-Modell setzt sich aus der Bewertung der Rechenleistung, der Overhead-Betrachtung für die GPU sowie der Übertragung von Rechenleistung zu Rechenzeit zusammen. Die Bewertung der Rechenleistung beinhaltet eine Abstraktionen der Hardware sowie eine detailierte Betrachtung der Implementierung der durchzuführenden Operation und ist der Kern des Modells. Dem folgt eine Beschreibung der Faktoren, die einen Overhead erzeugen. Zum Schluss wird die Rechenleistung der einzelenen Prozessoren auf die jeweiligen Laufzeiten übertragen, wozu das algorithmische Vorgehen zur Balancierung erklärt wird.

5.2. Bewertung der Rechenleistung

Die Rechenleistung eines Prozessors wird meist in FLOP/s angegeben. Diese Größe beschreibt die Anzahl der Fließkommaberechnung, die pro Sekunde ausgeführt werden können. Bestimmt wird sie durch die Taktrate, die Anzahl der Prozessoren und der Floating Point Operations (FLOPs), die pro Takt durchgeführt werden. Daneben existieren Maßzahlen wie Instruktionen pro Sekunde oder Frames pro Sekunde. Sie sind aber für Berechnungen wenig aussagekräftig. Eine Angabe von Instruktionen pro Sekunde ist über die Architektur und dem damit verbundenen Befehlssatz zu relativieren[1] Für Grafikprozessoren gilt die Angabe der Frames pro Sekunde i. A. als aussagekräftige Kenngröße, allerdings ist diese Angabe bei der Verwendung von Grafikprozessoren als Rechenbeschleuniger ungeeignet, da für Berechnungen neben dem Grafikrendering keine volle Auslastung der Streaming-Prozessoren zu erreichen ist, wodurch sie irrelevant wird. Grund dafür ist, dass die Berechnung nur selten die Anforderungen erfüllen, welche von der GPU für die Stream-Verarbeitung gestellt werden: viele gleichartige, sequentielle und unabhängige Operationen.

[1] Prozessor-Architekturen unterscheidet man in Reduced Instruction Set Computer (RISC)- oder Complex Instruction Set Computer (CISC)-Architekturen. Maschinenbefehle einer RISC-Architektur werden in einem einzelnen Prozessortakt ausgeführt, jedoch setzen sich komplexe Operationen aus mehrern Maschinenbefehlen zusammen. CISC-Architekturen kodieren komplexe Operationen in einzelenen Maschinenbefehle, die aber unterschiedlich viele Prozessortakte dauern können. Damit hängt die eigentliche Leistung von den Instruktionen pro Sekunde und den durchzuführenden Operationen ab.

Rechenleistung einer CPU

Heutige CPUs sind i. d. R. Multicores mit 2 bis 16 Kernen. Pro Takt kann jeder Kern bspw. eine MAD (2 Fließkommaberechnungen) ausgeführen. Zusätzlich haben heutige CPUs Streaming SIMD Extension (SSE)-Unterstützung. Dadurch können auf SIMD-konformen Daten vektorisierte Operationen durchgeführt werden. Um SSE überhaupt nutzen zu können muss das Programm jedoch vektorisierbar sein. Die Anzahl der gleichzeitig berechneten Daten wird vom Datentyp bestimmt, da ein SSE-Register in der aktuellen Version immer 128 Bit umfasst. Dies ergibt bis zu vier MAD in doppelter Genauigkeit. Für nicht vektorisierte Operationen leisten aktuelle CPUs in einfacher, wie in doppelter Genauigkeit, i. d. R. die gleiche Anzahl an FLOP. Allgemein lässt sich die Rechenleistung einer CPU wie folgt formulieren:

$$\mathrm{RL}_{\mathrm{CPU}} = \text{Taktrate} \cdot \#\text{Prozessoren} \cdot \frac{\#\text{FLOP}}{\text{Takt}} \cdot \text{Vielfachheit Vektorisierung} \tag{5.1}$$

Rechenleistung einer GPU

Jeder SP einer GPU kann ähnlich einem CPU-Kern eine MAD ausführen, jedoch nur in einfacher Genauigkeit, weil dies für das derzeitige Grafikrendering ausreicht. Die Anzahl der doppelt genauen Operationen hängt von der Generation der Grafikkarte ab. Zu Beginn unterstützten Grafikkarten doppelt genaue Berechnungen überhaupt nicht, spätere sind mit wenigen DPUs pro SM ausgestattet. Beispielsweise für eine Tesla C1060, die pro SM nur eine DPU besitzt, kann bei doppelter Genauigkeit nur $\frac{1}{8}$ der einfach genauen Leistung erreicht werden. Bei neueren Karten aus Nvidia's Fermi-Serie liegt dieser Faktor bei $\frac{1}{2}$, da pro SM 16 doppelt genaue MAD ausgeführt werden können[2] [Nvi09]. Bei der Angabe der Rechenleistung ist daher die Genauigkeit der Operation zu beachten. Außerdem verfügen SMs über zusätzliche SFUs, die je nach Operation zur Leistung betragen können. Insgesamt ergibt sich eine Bewertung wie folgt:

$$\mathrm{RL}_{\mathrm{GPU}} = \begin{cases} \text{Taktrate} \cdot \left[\#\text{SMs} \cdot \left(\frac{\#\text{SPs}}{SM} \cdot \frac{\#\text{FLOP}}{\text{Takt}} + \frac{SFU}{SM} \cdot \frac{\#\ \text{FLOP}}{\text{Takt}}\right)\right] & (\textit{single precision}) \\ \text{Taktrate} \cdot \left[\#\text{SM} \cdot \left(\frac{\#\ \text{DPU}}{SM} \cdot \frac{\#\text{FLOP}}{\text{Takt}}\right)\right] & (\textit{double precision}) \end{cases} \tag{5.2}$$

Flaschenhals Speicherbandbreite und alternative Abschätzung der Rechenleistung

Für die SpMV-Berechnung, wie für viele andere Berechnungen, gibt die maximale Rechenleistung keine annähernd genaue Schätzung der real zu erwartenden Leistung ab, da das Verhältnis von Rechenintensivität zur Speicherbelastung zu gering ist. Schon lange stellt die Speicherbandbreite den Flaschenhals für solche Berechnungen da, weil die für

[2] Genau gesagt berechnen 2 SPs eine doppelt genaue MAD, sodass eine SM mit 32 SPs insgesamt 16 Ergebnisse produziert.

die Berechnung notwendigen Daten nicht schnell genug vom Arbeitsspeicher in die Register geladen werden können [MV99]. Diese Algorithmen bezeichnet man auch als *memory bound* (den umgekehrten Fall als *compute bound*). Die zu erwartende Rechenleistung wird deshalb wesentlich besser durch die Speicherbandbreite charakterisiert. Die Speicherbandbreite setzt sich dabei (sofern nicht angegeben) folgendermaßen zusammen:

$$\text{Speicherbandbreite} = \text{Speichergeschwindigkeit} \cdot \text{Anzahl Kanäle} \cdot \frac{\text{Byte}}{\text{Kanal}} \tag{5.3}$$

Praktisch wird jedoch nur ein Teil der theoretischen Bandbreite erreicht, da ein optimaler Zugriff auf die Daten sowie ein optimales Prefetching vorrausgesetzt sein müssten, um eine volle Auslastung des Busses zu erzielen. Zur praktischen Bestimmung eines Referenzwertes für den durchschnittlichen Durchsatz hat sich für CPUs der STREAM-Benchmark [McC07] in vielen Fällen [KD10, WWP08] bewährt. Er misst die Zeit unterschiedlicher Operationen (angeboten werden vier verschiedene Operation: copy, scale, sum, triad) und berechnet daraus real zu erreichende Bandbreitenwerte. Die Operationen weisen unterschiedliche Verhältnisse von Datenzugriffe (lesend und schreibend) zu Rechenopertionen auf und geben damit verschiedene Referenzwerte für unterschiedliche Zugriffsmuster. Für die GPU wird der BandwidthTest aus dem CUDA SDK [cud] genutzt. Mit diesem wird sowohl die Bandbreite für das Kopieren innerhalb der GPU als auch von und zur GPU über den PCI-Express-Bus gemessen (vergleichbar mit copy Operation des STREAM-Benchmarks). Dabei kann zudem unterschieden werden, ob die zu kopierenden Daten in pageable oder page-locked Speicherbereichen allokiert wurden[3].

Das **Roofline Modell** [WWP08] verbindet Rechenleistung und Speicherbandbreite in einem Modell (s. Gleichung 5.4) und versucht weitere Einflüsse auf die Leistung eines Programmes einzubringen. Mit der Operational Intensity (OI) wird im allgemeinen Modell das Verhältnis von der Anzahl der Operationen zu den dafür nötigen Speicherzugriffen beschrieben und ist daher in $\frac{FLOP}{byte}$ angegeben. Sie gibt an wie effizient auf die Daten eines Programmes zugegriffen werden kann. Das Minimum beider Werte macht eine Abschätzung über die zu erreichende Leistung.

$$\text{erreichbare Flop/s} = min \begin{cases} \text{Rechenleistung} \\ \text{Speicherbandbreite} \times \text{Operational Intensity} \end{cases} \tag{5.4}$$

Das Modell soll vereinfacht eine Abschätzung darüber geben, welche Leistung ein Programm erzielen kann, in welcher Reihenfolge nächste Schritte zur Verbesserung dessen vorgenommen werden sollten und wann mit der Optimierung aufgehört werden kann, weil dem Problem technische Grenzen gesetzt sind. Der Einfluss der OI auf die Leistung wird

[3] In LAMA wird immer page-locked (pinned) Speicher verwendet, da dieser benötigt wird, um asynchrone Kommunikation zu nutzen. Dieser wird im Vergleich zu pageable Speicher schneller kopiert, da keine Seitenfehler beim Zugriff auftreten können. Allerdings sollte der page-locked Speicher nicht überansprucht werden, da die Verwendung dieser seltenen Ressource die Gesamtleistung des Systems einschränken kann [Nvi10].

am besten an einem Diagramm wie in Abbildung 5.2 deutlich, welches anhand weniger Kenngrößen für eine Architektur erstellt werden kann.

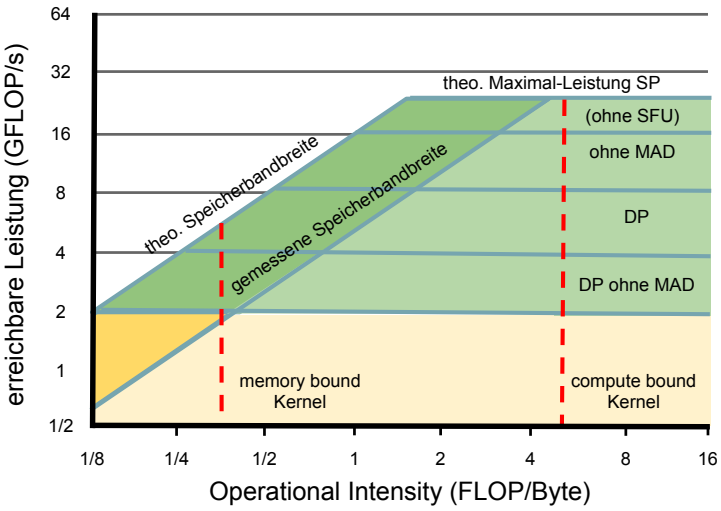


Abbildung 5.2.: Beispiel des Roofline-Modells mit wesentlichen Einflussfaktoren für die SpMV-Berechnung auf CPU und GPU; diagonale Rooflines: theoretische Speicherbandbreite nach maximaler Ausnutzung der Hardware, gemessene Speicherbandbreite (beispielsweise durch den STREAM-Benchmark); horizontale Rooflines: theoretisch maximale mögliche Rechenleistung bei voller Auslastung der gegebenen Recheneinheiten

Je nach OI wird zuerst eine diagonale oder horizontale Roofline erreicht (memory bzw. compute bound). Einzelne Rooflines werden durch unterschiedliche begrenzende Faktoren beschrieben. In [WWP08] werden beispielsweise als Gründe für eine eingeschränkte Bandbreitenausnutzung fehlendes softwareseitiges Prefetching oder ungünstige Lokalität von Daten in Mehrsockelsystemen angeführt. Als Ursache für geringere Rechenleistung werden eine schlechte Floating-Point-Balance[4] oder fehlende Vektorisierung genannt. Außerdem wird erläutert, dass das Modell ebenso auf andere Maßzahlen angewandt werden kann. Wenn z. B. alle Daten für eine Berechnung in den L2-Cache passen, wird die Leistung durch die Bandbreite zum L2-Cache und nicht zum Arbeitsspeicher begrenzt. Genauso ist für Grafikkarten eine Bewertung in Frames pro Sekunde anstatt FLOP/s möglich.
Das Roofline-Modell soll veranschaulichen, wie sich verschiedene Faktoren auf die Leistung einer Operation auswirken. Wichtigster Schritt ist dabei sich zu vergegenwärtigen, welche Faktoren auf die jeweilige Operation Einfluss nehmen. Die SpMV-Berechnung ($y_i = \sum_{j=0}^{n} a_{ij} \cdot x_j$) besteht pro Matrix-Eintrag ($a_{ij} \neq 0$) aus lediglich einer Multiplikation und Addition. Dafür ist der Zugriff auf die Daten der Matrix und der zwei Vektoren nötig. Eine Floating-Point-Balance ist damit gegeben, es kann sogar die MAD eingesetzt werden kann. Die SFU der GPU kann dies hingegen nicht, da sie nicht für einfache Multiplikation und Addition (von der CUDA Runtime) verwendet wird. Entscheidend ist demnach an

[4] Als Floating-Point-Balance wird in [WWP08] die Ausgeglichenheit zwischen Floating-Point-Multiplikationen und -Additionen verstanden, da Prozessoren i. d. R. gleich viele Register für diese beiden Berechnungen besitzen.

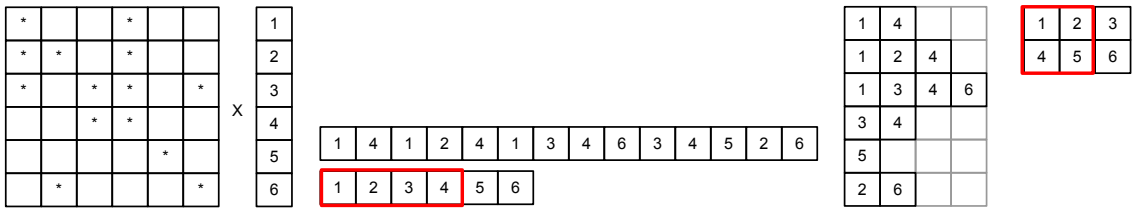
dieser Stelle ausschließlich die Genauigkeit der Berechnung. I. A. kann man für spätere Anwendungsfälle von doppelter Genauigkeit ausgehen.
Trotzdem ist die horizontale Roofline bei der SpMV für die meisten Systeme nebensächlich, da die Berechnung zuerst von der Speicherbandbreite begrenzt wird. Damit rückt die Bestimmung der OI in den Vordergrund. Dieser Wert ist für jede Kombination aus durchzuführender Operation und ausführender Hardware neu zu bestimmen. Nach William et. al.[WWP08] ist der sicherste Weg zur Bestimmung dieses Wertes das Messen durch Performance Counter. In der Praxis soll aber meist auch die händische Berechnung ausreichen.
Die Anzahl der Operationen ist einfach angegeben: sie beträgt pro Matrix-Eintrag zwei (eine Multiplikation und eine Addition). Die Anzahl der Zugriffe kann dagegen nicht so einfach und exakt vorhergesagt werden. Vielmehr müssen dabei die Charakteristika der Hardware abstrahiert werden, weil nur die Zugriffe auf den Arbeitsspeicher bzw. globalen Speicher gezählt werden dürfen. Auf der CPU wird die Gesamtzahl der Zugriffe auf den Arbeitsspeicher durch dazwischenliegende Caches verringert. Auf der GPU reduziert ein coalesced Zugriff (s. Abbschnitt 3.1.2) sowie der Textur-Cache die Anzahl der Zugriffe auf den globalen Speicher. Für die Bestimmung der OI der SpMV-Berechnung auf CPUs und GPUs wird im folgenden Abschnitt die Implementierung in LAMA genauer betrachtet.

SpMV-Implementierung in LAMA

Die Speicherung und Verarbeitung bei der SpMV-Berechnung wird durch das jeweilige Speicherformat beschrieben. Für einen effizienten Zugriff auf die Daten wurde für CPUs das CSR- und für GPUs das ELL-Format verwendet. Eine detailierte Darstellung dieser Formate mit Beispiel liegt in Absatz 4.2.3 vor.
Durch die zeilenweise Speicherung des CSR-Formates und gleichzeitiger zeilenweiser Verarbeitung ist ein sequentieller Zugriff auf die Matrix-Einträge möglich. Dies ist optimal für den Cache, da die räumliche Lokalität der Daten bestmöglich ausgenutzt wird. Für die GPU bietet sich eine spaltenweise Speicherung wie im ELL-Format an, da die einzelnen SPs innerhalb eines Blockes coalesced auf den Speicher zugreifen können.
Auf jeden Matrix-Eintrag a_{ij} wird genau einmal zugegriffen. Außerdem erfolgt der Zugriff aufgrund der dünn besetzten Speicherung indirekt über zusätzliche Arrays, die dazu linear durchlaufen werden. Je nach Matrixformat kann die Anzahl der indirekten Indizierung unterschiedlich hoch sein. Für das CSR- und ELL-Format hat man in LAMA aber jeweils zwei Indexarrays (ia und ja)[5]. Auf ja wird ebenso wie auf das *data*-Array sequentiell zugegriffen. Der Zugriff auf ia erfolgt nur zweimal (CSR) bzw. einmal (ELL) pro Zeile. Damit profitiert der Zugriff auf ja auch von caching und coalescedness. Die Daten von ia werden dagegen vor ihrer Wiederverwendung verdrängt worden sein (Capacity Miss).
Auch auf jeden Eintrag des Ergebnis-Vektors y wird einmal pro Zeile (schreibend) zugegriffen. Da aber zwischenzeitlich auf viele andere Daten zugegriffen wurde, wird die

[5] In LAMA wird für das ELL-Format zusätzlich das ia-Array gespeichert, um die Zeilen schneller durchlaufen zu können.

Cache-Zeile auf einer CPU i. d. R. vor dem erneuten Zugriff verdrängt worden sein. Zudem kommt es auf die write-Policy an, ob der Cache überhaupt beansprucht wird. Für die GPU erfolgt der Schreibzugriff auf den Ziel-Vektor y von allen Threads eines Blockes gleichzeitig und damit coalesced.
Der Zugriff auf den Quell-Vektor x erfolgt abhängig von der Matrixstruktur rein zufällig. Im schlimmsten Fall muss jeder Wert aus dem Speicher gelesen werden. Auf der GPU wird der Vektor zudem im Texture Cache gehalten. Für diesen muss kein coalesced Zugriff vorliegen, allerdings teilen sich alle Threads eines Blockes den Texture Cache, sodass sowohl zwischen den Threads Konflikte auftreten können als auch innerhalb der Zugriffe eines Threads. Eine Lokalität der Daten muss daher in zwei Dimensionen (zwischen den Threads (spaltenweise) und innerhalb einer Zeile (zeilenweise)) vorliegen. Ist diese Lokalität nicht gegeben, so ist auch hier ein häufiger Zugriff auf den globalen Speicher zum Nachladen nötig. Abbildung 5.3 veranschaulicht den Unterschied des Cacheverhaltens auf der CPU und GPU.



(a) SpMV **(b)** CPU **(c)** GPU

Abbildung 5.3.: Zugriffe bei der SpMV für CPU (CSR-Format) und GPU (ELL-Format) auf den Cache bzw. Texture Cache; bei einer Cache-Größe von vier Elementen würde zu Beginn der rote umrahmte Block des Vektors geladen; für die CPU tritt der erste Cache-Miss für den 9. Matrix-Eintrag (Index 6); für die GPU liegt mit einem spaltenweisen Durchlaufen schon beim 4. Eintrag (Index 3) ein Cache-Miss vor, man sieht aber auch das mit der spaltenweise Speicherung die Lokalität zwischen den Zeilen (die erste drei Einträge) für den Cache zuträglich sein kann

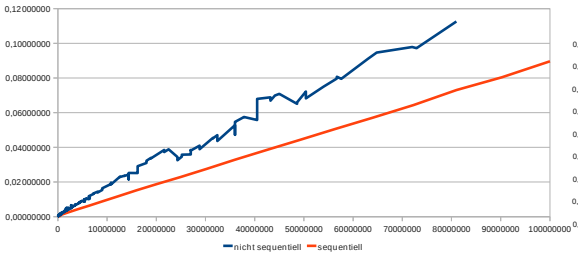
Bei einer Matrix mit nnz nicht-null-Einträgen in einer Zeile muss auf diese nnz Matrix-Einträge (lesend), sowie auf nnz Vektor-Einträge aus x (lesend), und auf den einen Vektor-Eintrag der Zeile in y (schreibend) zugegriffen werden. Hinzu kommen zwei (CSR) bzw. ein (ELL) Zugriff auf ia und nnz Zugriffe auf ja zur korrekten Indizierung. Tabelle 5.1 fasst alle Informationen zu den Zugriffen auf die Arrays zusammen.

	ia	ja	$data$	x	y
Zugriffe	2 (1)	nnz	nnz	1	nnz
Typ	Index	Index	Daten	Daten	Daten
Art	lesend	lesend	lesend	lesend	schreibend
CPU	2. gecached	gecached	gecached	nicht gecached	zufällig gecached
GPU	coalesced	coalesced	coalesced	coalesced	zufällig gecached, uncoalesced

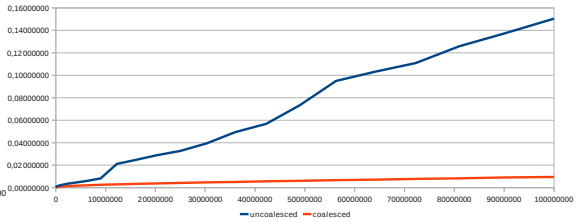
Tabelle 5.1.: Details zu den Datenzugriffen für CPU und GPU für die verschiedenen Strukturen; Anzahl der Zugriffe pro Zeile, Datentyp des Vektors, Zugriffsart, cached?, coalesced?

Messungen des allgemeinen Leistungsverhalten

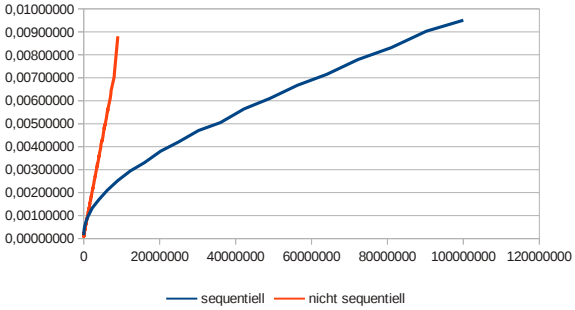
an den Benchmark-Ergebnissen aus Abbildung 5.4 sollen die Annahmen, dass das Zugriffsmuster auf den Quell-Vektor x sowie für die GPU ein coalesced Zugriff auf die Daten Einfluss auf die Laufzeit nehmen, gezeigt werden. Für einen sequentiellen Zugriff wurden dicht besetzte Matrizen benutzt. Für diese ist der Zugriff auf x zwangsläufig sequentiell und damit genauso gut für den Cache, wie der Zugriff auf ja und $data$. Nicht-sequentieller Zugriff liegt bei den verwendeten Poisson-Matrizen vor. Die Poisson-Matrizen besitzen ein regelmäßiges Zugriffsmuster auf den Vektor x, die räumliche Lokalität besteht jedoch zwischen den Zeilen. Wegen des Offsets von einer Diagonalen der Matrix zur nächsten besteht aber kaum Lokalität innerhalb einer Zeile. Um einen coalesced gegen einen uncoalesced Zugriff zu bewerten wurden Messungen für deas CSR- und ELL-Format gemacht.



(a) CPU: sequentiell vs. nicht sequentiell



(b) GPU - sequentiell: coalesced vs. uncoalesced



(c) GPU - coalesced: sequentiell vs. nicht sequentiell

Abbildung 5.4.: Leistungsverhalten bei unterschiedlichen Zugriffsmustern; (a) CPU: Vergleich für sequentielles Zugriffsverhalten auf x bei dicht besetzten Matrizen gegenüber keinem sequentiellen Zugriffsverhalten bei 2D-9Punkte-Poisson-Matrizen; (b) GPU: Vergleich von coalesced (ELL) gegenüber uncoalesced (CSR) Zugriff bei sequentiellen Zugriffsverhalten auf x; (c) GPU: Vergleich von sequentiellem (dicht besetzte Matrix) gegenüber nicht sequentiellem (2D-9Punkte-Poisson-Matrix) Zugriffsverhalten auf x für coalesced (ELL) Zugriff auf die Matrix; Messung erfolgte auf dem Testsystem F1Boensch (s. Sektion 7.1)

Den Ausführungen des vorangegangen Abschnittes ist zu entnehmen, dass die Leistung einer CPU von dem Zugriffsverhalten auf den Vektor x bedingt wird, da der Zugriff nicht, wie für die anderen Zugriffe, implizit aus dem Cache bedient wird. In Abbildung 5.4a wird dieser Einfluss dargestellt. Somit verhält sich die Laufzeit für einen sequentiellen Zugriff mit Skalierung der Matrixgröße linear. Gegenübergestellt werden 2D-9Punkte-Poisson-Matrizen[6] mit derselben Anzahl nicht-null-Einträge. Da mit der geringen Anzahl Einträge

[6]Für den Aufbau der Poisson-Matrizen s. A.1

pro Zeile die Anzahl der Zugriffe auf ia höher ist, ist der Verlauf generell höher aber er variiert auch, da für die CPU kaum gecachter Zugriff möglich ist. Weiterhin sieht man deutlich, dass dieser mit wachsender Matrixgröße immer stärkeren Einfluss nimmt.
Für die GPU wurden zwei Einflussfaktoren auf die Leistung angeführt. Der Ausschlaggebendste ist, ob der Zugriff auf den Speicher coalesced oder uncoalesced erfolgt, da nur mit coalesced Zugriff die volle Speicherbandbreite ausgenutzt werden kann. Anderenfalls kann effektiv nur $\frac{1}{coa}$ der Bandbreite genutzt werden. In Abbildung 5.4b wird dieser Einfluss für dicht besetzte Matrizen ($\rightarrow$ mit sequentiellem Zugriff) verglichen. Der coalesced Zugriff kommt durch das ELL-Format zum Tragen, der uncoalesced Zugriff wurde durch Nutzung des CSR-Format veranlasst. Aufgrund des sequentiellen Zugriffes auf y ist für beide das Laufzeitverhalten linear zur Matrixgröße, der Unterschied von coalesced zu uncoalesced Zugriff wächst aber mit der Matrixgröße.
Übertragt man diesen Effekt des uncoalesced Zugriff für das Nachladen der Daten aus x in den Textur-Cache bei ansonsten coalesced Zugriff, ergibt sich ein Verhalten wie in Abbildung 5.4c. Für sehr kleine Matrixgrößen passt noch der gesamte Vektor in den Textur-Cache, sodass nie nachgeladen werden muss. Sobald dies nicht mehr möglich ist muss für jeden Thread-Block (mehrmals) nachgeladen werden. Bei dicht besetzten Matrizen nimmt mit wachsender Zeilenlänge[7] die Anzahl der Nachladeoperationen linear zur Zeilenlänge zu. Für die Poisson-Matrizen ist ein Nachladen wesentlich öfter nötig, da kaum räumliche Lokalität besteht. Dadurch steigt die Laufzeit wesentlich stärker.

Anwendung auf die Operational Intensity

Das unterschiedliche Laufzeitverhalten für CPU und GPU aufgrund des Zugriffsverhalten soll für das zu entwickelnde Modell in der Operational Intensity ausgedrückt werden. Mit der so prognostizierten Leistung kann in weiteren Schritten ein Modell zur Laufzeitvorhersage entworfen werden. Dazu gilt es die Zugriffe auf den Arbeitsspeicher bzw. globalen Speicher zu veranschlagen. Alle nötigen Details sind in Tabelle 5.1 zusammengestellt.
Der Cache bringt für die sequentiellen Zugriffe von ia, ja, $data$ den Vorteil, dass nur jeder n-te Zugriff auf den Arbeitsspeicher geht. n ist dabei die Anzahl der Elemente einer Cacheline. Bei den Zugriffen, die auf den Arbeitsspeicher erfolgen, handelt es sich um compusary misses, die in den L1-Cache geladen werden müssen. Da kein wiederholter Zugriff vorliegt, tragen L2- und L3-Cache nichts dazu bei, dass Daten nicht aus dem Arbeitsspeicher geladen werden müssen. Zugriffe auf x wiederholen sich dagegen und können nach einer Verdrängung noch im L2- oder L3-Cache vorliegen. Dies soll aber für das Modell ignoriert werden, da zur Bewertung genaue Informationen über Verdrängungsstrategien und die Struktur der Matrix vorliegen müssen. Stattdessen wird der worst case betrachtet, also dass alle Zugriffe nicht gecached sind.
Für die GPU muss die Anzahl der Zugriffe pro Warp betrachtet werden, da innerhalb dessen coalesced Zugriffe erfolgen. Ein coalesced Zugriff lädt eine feste, hardwareabhängige

[7] Für dicht besetzte Matrizen mit m Elemente (auf der Abszisse abgetragen) beträgt die Zeilenlänge $\sqrt{m}$.

Menge *coalescedAccessSize* (CAS) an Daten auf einmal. Diese Größe ist von der compute capability der Grafikkarte abhängig. Derzeit sind dies 64 oder 128 Byte (compute capability ≤ 1.3 bzw. > 1.3). Mit einem Zugriff werden $coa = \frac{CAS}{DataSize}$ Elemente geladen. Für einen Warp sind demnach $\lceil \frac{warpsize}{coa} \rceil$ Zugriffe nötig. Die warpsize beträgt bei allen bisher erschienenen CUDA-Karten 32. Dies kann sich aber für nächste Generationen ändern.
Schlussendlich kann alles zu folgender Definition der OI zusammengefasst werden:

$$\mathrm{OI}_{\mathrm{CPU}} = \frac{2 * nnz}{1 \cdot (\underbrace{I}_{ia} + \underbrace{D}_{y}) + \underbrace{\left\lceil \frac{nnz}{n_I} \right\rceil}_{ja} \cdot I + \underbrace{\left\lceil \frac{nnz}{n_D} \right\rceil}_{data} \cdot D + \underbrace{nnz}_{x} \cdot D} \tag{5.5}$$

$$\mathrm{OI}_{\mathrm{GPU}} = \frac{2 * \frac{nnz}{rows} * warpsize}{\left(\underbrace{\left\lceil \frac{warpsize}{coa_I} \right\rceil}_{ia+ja} CAS + \underbrace{\left\lceil \frac{warpsize}{coa_D} \right\rceil}_{y+data} \cdot CAS \right) \cdot (\frac{nnz}{rows} + 1) + \underbrace{\frac{nnz}{rows}}_{x} \cdot CAS} \tag{5.6}$$

mit:

$$\begin{aligned}
I &= IndexSize \\
D &= DataSize \\
n_I &= \frac{CachelineSize}{I} \\
n_D &= \frac{CachelineSize}{D} \\
CAS &= \begin{cases} 64 \text{ Byte} & \text{compute capability} \leq 1.3 \\ 128 \text{ Byte} & \text{compute capability} > 1.3 \end{cases} \\
coa_I &= \frac{CAS}{I} \\
coa_D &= \frac{CAS}{D}
\end{aligned}$$

5.3. Overhead-Betrachtung

Generell wäre eine Gewichtung der Partitionen anhand der Leistung der Prozessoren möglich. Für die GPU kommt aber für die Berechnung noch verschiedener Overhead hinzu. Da dieser am besten in Zeit beschrieben wird bietet, sich eine Umrechnung der Leistung in Zeit an. Overhead wird produziert durch:

Kernel-Launch Vom Starten eines Kernels bis zur ersten Berechnung vergeht eine Zeit. Dies bezeichnet man als *kernel launch*. Die Zeit variiert von Grafikkarte zu Grafikkarte. Angegeben werden kann sie, indem die Laufzeit eines leeren Kernels gemessen wird. Für die SpMV-Berechnung wird ein Kernel für die Berechnung des lokalen Anteils, sowie für die Berechnung des Halo-Teils (sofern ein Halo existiert) gestartet.

Kommunikationszeit Da die GPU keinen Zugriff auf den Arbeitsspeicher der CPU hat, müssen die Daten explizit auf die GPU kopiert werden. Dabei unterscheiden wir:

initiale Daten Die Daten für die Matrix und die Vektoren müssen vor der Berechnung auf die GPU kopiert werden. Dies dauert ein Vielfaches einer einfachen SpMV-Berechnung, sodass sich die Berechnung der SpMV auf einer GPU erst für wiederholte Berechnung mit denselben Daten, wie z. B. bei iterativen Lösern, lohnt. Daher kann dieser Aufwand als initale Kosten betrachtet werden und wird an dieser Stelle nicht in die Laufzeitvorhersage einbezogen, sonder muss gesondert für die Gesamtlaufzeit eines iterativen Lösers betrachtet werden. Dabei spielen Größe der Matrix und das allgemeine Kovergenzverhalten eine Rolle.

Halo Daten Die Halo-Einträges des Vektors x werden in jedem Fall für jede einzelene SpMV-Berechnung benötigt. Die Kommunikationsmuster für den Halo-Austausch wurden in Absatz 4.2.4 beschrieben. Entsprechend geht der Kommunikationsoverhead je nach synchroner oder asychoner Kommunikation unterschiedlich in die Gesamtlaufzeit ein. Für eine Aussage über die Zeit der Halo-Kommmunikation kann nur mit der Größe des aufkommenden Kommmunikationsvolumen gemacht werden. dies kann apriori aber nicht abgegeben werden, das dies von der Matrixstruktur und der speziellen Verteilung abhängt. I. A. sollte diese Größe mit der Graph-Partitionierung ausreichend minimiert worden sein, sodass sie keinen großen Einfluss nimmt.

Vorbereitung der Kommmunikation (Sammeln der Daten) Damit nicht alle lokalen Vektor-Daten an alle Prozessoren verschickt werden, werden die Einträge gesammelt, die ein Prozessor tatsächlich benötigt. Informationen darüber, welche Daten an wen gehen, werden mit dem Aufsetzen der Matrix evaluiert und in der Struktur `Kommunikationsplan` gespeichert. Somit muss dies nicht für jede Berechnung herausgefunden werden und der Aufwand liegt beim Zusammenkopieren von Daten. Für die wiederholte SpMV-Berechnung, wie sie hier evaluiert wird, erfolgt dieser Schritt auf der CPU, da sich der Quell-Vektor seit der Initialisierung nicht verändert hat. Für eine spätere Verwendung in iterativen Lösern erfolgt dieser Schritt hingegen auf der GPU und kann nicht asynchron zur Berechnung erfolgen. Somit muss dies gesondert betrachtet werden. Zudem kommt ein weiterer Kernel Launch hinzu. dies ist aber für die folgende Analyse der reinen SpMV nicht relevant.

Der Overhead für den Kernel Launch ist, gegenüber den anderen, unabhängig von dem zugewiesenen Anteil der Matrix. Er wird damit in jedem Fall mit der korrekten Zeit veranschlagt, muss jedoch ins richtige Verhältnis zur Größe der Matrix gestellt werden. Die Kommunikation sowie die Vorbereitung ist abhängig von der Datenmenge.

5.4. Von der Rechenleistung zur Laufzeit

Für eine Übersetzung der Leistung in Laufzeit wird lediglich die Anzahl der Operationen benötigt. Für eine Laufzeitvorhersage der Berechnung der gesamten Matrix-Vektor-Multiplikation ergibt sich demnach mit der machbaren Overhead-Betrachtung:

$$t_{compALL} = \frac{\text{Operationen}}{\text{Leistung}} + \text{Overhead} = \frac{2 \cdot \text{Matrix-Einträge}}{wComp} + \begin{cases} 0 & \text{für CPUs} \\ \text{kernel Launch} & \text{für GPUs} \end{cases} \tag{5.7}$$

Für die verteilte Berechnung werden allerdings die lokalen Operationen bzw. Matrix-Einträge benötigt. Die Laufzeit soll daher iterativ bei einer Balancierung zwischen den Prozessoren angenähert werden, indem die Anzahl der Operationen aus vorangegangenen Laufzeitvorhersagen berechnet wird. Inital erfolgt eine Laufzeitvorhersage für jeden Prozessor und die gesamte Matrix nach Gleichung 5.7. Damit kann eine erste Gewichtung ($oldWeight$) berechnen werden. Iterativ wird sich einer Gewichtung angenähert, bei der die prognostizierten Laufzeiten balanciert sind.
Die Laufzeit von Prozessor p in Iteration i der Balancierung berechnet sich wie folgt, wobei der Faktor zwei für den kernel launch davon komm, dass mit der Partitionierung auch ein Kernel für die Halo-Berechnung gestartet werden muss:

$$t_{compPART_i}(p) = \frac{2 \cdot \text{Matrix-Einträge}}{wComp} \cdot oldweight_i(p) + \begin{cases} 0 & \text{für CPUs} \\ 2 \cdot \text{kernel Launch} & \text{für GPUs} \end{cases} \tag{5.8}$$

Der Fehler der bei dieser Bewertung gemacht wird, besteht aus der fehlenden Berücksichtigung des Caches bei der Leistungsbewertung und der Kommunikation der Halo-Daten. Des Weiteren nimmt die Gewichtung keine Rücksicht darauf, ob die Menge an Daten die einer Partition zugewiesen werden in den jeweiligen Speicher passt. Dies ist insbesondere für die GPU relevant, da der globale Speicher i. d. R. kleiner als der Arbeitsspeicher der CPU ist, dabei aber einiges leistungsstärker. Schlussendlich kann die Gewichtung auch keine Aussage darüber machen, ob das Problem für die verfügbare Anzahl Prozessoren nicht zu klein ist, um darauf zu skalieren. Somit erfolgt keine Begrenzung der Anzahl der Prozesse.

5.5. Dynamische Lastbalancierung

Die verteilte Berechnung für die SpMV lohnt sich erst bei wiederholter Ausführung, wie in iterativen Lösern, da das Verteilen der Daten bereits Zeit beansprucht. Sind GPUs beteiligt zählt dieses Argument doppelt, weil die Daten zusätzlich auf diese geladen werden müssen. Iterative Löser, wie das Jacobi-Verfahren, das auf einer SpMV-Berechnung aufsetzt, werden sehr häufig gebraucht und müssen oft hunderte oder tausende Iterationen lang rechnen. Somit bringen auch kleinere Laufzeitgewinne für eine einzelne Iteration

große Vorteile für die Gesamtberechnung. Es macht daher durchaus Sinn in den ersten Iterationen Zeit für das Suchen einer guten Verteilung zu investieren. Der große Vorteil der dynamischen Lastbalancierung ist, dass zur Laufzeit durch Monitoring neue Informationen in die Bewertung eingehen können. Aus verschiedenen Werten können dabei unterschiedliche Verfahren zur dynamischen Balancierung abgeleitet werden.
Zum einem kann aus vorhergesagter und real erreichter Laufzeit ein Maß über den Fehler des Modells gegeben werden. Geht man davon aus, dass dieser für die gegebene Matrix und den jeweiligen Prozessor konstant ist, kann mit einer einzelnen Korrektur der Gewichte ein balancierter Zustand erreicht werden. In der Praxis sollte bei einer schrittweisen Näherung zur idealen Gewichtung der Fehler auch bei weiteren Einflüssen kleiner werden und so nach wenigen Iterationen eine Lösung gefunden haben. Alternativ kann anhand der real zu erzielten Zeiten und der dabei vorliegenden Gewichtung eine neue Bewertung der Prozessoren vorgenommen werden. Diesen Ansatz verfolgen Lee und Eigenmann in [LE08] für blockweise Verteilungen. Die Bewertung basiert dabei auf der normalisierten Laufzeit pro Zeile (NRET). Dieser Wert bezieht neben der spezifischen Leistung eines Prozessors auch weitere Einflüsse wie den Schleifenoverhead oder das Teilen von Speicherbandbreite auf Multicore-CPUs und Kommunikation mit ein.
Neben der Frage wie die Gewichtung zu verbessern ist, stellt sich auch die Frage wie diese auf die neue Verteilung abzubilden ist. Eine Variante ist es eine neue Verteilung über Graph-Partitionierung berechnen zu lassen, eine andere entsprechend der neuen Gewichte einzelne Zeilen an Nachbarn abzugeben bzw. von ihnen anzunehmen. Dabei bietet die Neuberechnung den Vorteil, dass das Kommunikationsvolumen zwischen den Partitionen berücksichtigt wird. Allerdings bedeutet dies auch, dass eine vollkommen veränderte Partitionierung das Ergebnis sein kann und viel Kommunikation für die Umverteilung der Daten nötig ist. Werden nur Zeilen, die in einer Partition zu viel sind, in eine andere verschoben, wird die Kommunikation auf das Lastungleichgewicht begrenzt. Allerdings kann nicht ohne zusätzlichen Aufwand dabei das Kommmunikationsvolumen für die Berechnung beachtet werden.
Im Rahmen dieser Arbeit kann dieser Aspekt der dynamischen Lastbalancierung nicht genauer untersucht werden. Es soll nur soweit erwähnt werden, dass für eine effiziente Balancierung folgende Aspekte beachtet werden müssen:

Der Aufwand für die Berechnung der neuen Verteilung: Bei der Neuberechnung über die Graph-Partitionierung ist dieser abhängig von der Größe des Graphen und dem verwendeten Algorithmus. Zu einem Verschieben von Zeilen gehört eine Heuristik die zum einem die Partitionen bestimmt an die eine Partition mit zu viel Last ihre Zeilen abgeben kann und einer Auswahl von Zeilen die am besten abgegeben werden. Dabei ist die Anzahl der beteiligten Prozessoren und der zu verschiebenden Zeilen relevant.

Der Kommunikationsaufwand, der für die Umverteilung der Daten nötig ist: Je nachdem wie sehr sich die Gewichtung und die Verteilung der Zeilen auf die Prozessoren geändert hat, müssen mehr oder weniger Zeilen an andere Prozessoren verschickt werden. Um

dies Aufwand möglichst klein zu halten, bieten einige Graph-Partitionierungs-Bibliotheken auch Funktionen für die Verfeinerung bestehender Partionen an (Repartitioning) und behalten dabei sowohl das Kommmunikationsvolumen für die Anwendung (resultierend aus der neuen Verteilung) als auch das Kommunikationsvolumen, das für die Umverteilung nötig ist, im Blick. Der Aufwand dessen liegt aber sicher in der Höhe der Neuberechnung einer Verteilung.

Es ist abzuwägen, ob sich eine Neuberechnung und eine anschließend wahrscheinlich aufwendige Umverteilung lohnt. Dies wird voraussichtlich von der relativen Änderung der Gewichtung und der Größe und Struktur der Matrix abhängen. Bei einer grundlegenden Änderung der Gewichtung wird eine Neuberechnung sinnvoll sein, um den Kommmunikationsoverhead nicht unnötig in die Höhe zu treiben. Bei kleineren Änderungen wird ein Verschieben von Zeilen ungeachtet dessen performanter sein, weil der zusätzlich eingeführte Overhead so gering ist, dass er nicht ins Gewicht fällt (insbesondere bei asynchroner Kommmunikation).
Trotz der Leistungsschmälerung durch eine ineffiziente Neuberechnung und Umverteilung soll eine dynamische Balancierung durch Korrektur der Gewichte und eine Neuberechnung der Verteilung vorgenommen werden, um damit den, in der statischen Gewichtung gemachten, Fehler zu korrigieren. Deswegen soll ersteinmal der Ansatz verfolgt werden die Gewichte anhand des Fehlers bei der Laufzeitvorhersage zu korrigieren, um zu einem balancierten Ergebnis zu kommen. Eine absolute Balance ist aber aufgrund von Schwankungen in der Laufzeit nicht möglich, sodass eine gewisse Toleranz erlaubt sein muss. Die Inbalance ist die Abweichung der einzelnen Zeiten der Prozessoren voneinander. Da es das Optimierungsziel ist, die maximale Abweichung zu minimieren, wird diese Metrik wie folgt definiert:

$$\text{Inbalance} = \frac{max(\text{time}_p) - min(\text{time}_p)}{avg(\text{time}_p)},\ p \in P \tag{5.9}$$

Erweiterung: Einbeziehung Kommunikationsleistung

Für die Laufzeitvorhersage wurde ausschließlich die Rechenleistung (und der dafür nötige Kernel Launch) betrachtet. Die Kommmunikation der Halo-Daten sowie das dafür nötige Einsammeln der relevanten Daten fließt aber wie in Abbildung 4.5 gezeigt, ebenfalls in die Laufzeit ein. Für die GPU kommt zur Kommmunikation über die Prozessoren mittels MPI auch noch Up- und Download der Daten hinzu. Die Zeiten für diese Aufgaben werden durch die Menge der zu kommunizierenden Daten festgelegt. Mit dem Wissen aus vorangegangen Iterationen kann bei der dynamischen Balancierung darüber eine annähernde Schätzung über das Kommmunikationsvolumen gemacht werden. Überträgt man ein- und ausgehende Datenmengen auf die neue Gewichtung können zusätzlich auch die folgenden

Zeiten abgeschätzt werden:

$$\begin{aligned}
upload_p &= \text{Bandbreite}_{\text{Upload}} \cdot \#\text{Daten}_{\text{eingehend}} \cdot \text{Größe(Datentyp)} \\
download_p &= \text{Bandbreite}_{\text{Download}} \cdot \#\text{Daten}_{\text{ausgehend}} \cdot \text{Größe(Datentyp)} \\
MPIcomm_{pq} &= \text{Bandbreite}_{pq} \cdot (\#\text{Daten}_{\text{eingehend}} + \#\text{Daten}_{\text{ausgehend}}) \cdot \text{Größe(Datentyp)} \\
gather_p &= \text{Speicherbandbreite}_p \cdot [\#\text{Daten}_{\text{ausgehend}} \cdot (2 \cdot \text{Größe(Datentyp)} \\
&\quad + \text{Größe(Indextyp)})]
\end{aligned}$$

Entsprechend gilt es synchrone Kommmunikation folgende Gesamtzeiten zu balancieren:

$$t_{totalPART_i}(p) = t_{compPART_i}(p) + \begin{cases} \sum\limits_{q=0,\ q\neq p} MPIcomm_{pq} + gather_p & \text{für CPUs} \\ \sum\limits_{q=0,\ q\neq p} MPIcomm_{pq} + gather_p + upload_p & \text{für GPUs} \end{cases} \tag{5.10}$$

Ein- und ausgehende Datenmengen können sich dabei unterscheiden. Welchen Zusammenhang diese Mengen mit der Matrixstruktur aufweisen wurde bereits in Absatz 4.3.2 erläutert. Die Datenmenge kann sich zwar mit einer neuen Verteilung ändern, bei einer hohen Dünnbesetztheit der Matrix ist aber davon auszugehen, dass dies nicht im hohen Maße passiert, da dafür viele Zeilen im Halo in derselben Spalte einen Eintrag haben müssten. Bestünde diese Abhängigkeit hätte die Graph-Partitionierung dies aber anders aufgeteilt. Somit sollte mit der dynamischen Balancierung auch eine gute Vorhersage der Gesamtlaufzeit möglich sein. Eine Balancierung anhand dessen spielt besonders für Systeme auf denen synchrone Kommmunikation nötig ist oder solche mit langen Übertragungszeiten, die nicht mehr durch die Berechnung verdeckt werden können, eine Rolle.

Erweiterung: für iterative Löser

Für iterative Löser ändert sich der Quell-Vektor x mit jeder Iteration. Für eine einfache wiederholte Berechnung der SpMV, wie sie bisher betrachtet wurde, waren die zu versendenden Daten in aktueller Form von der Initialisierung noch auf der CPU vorhanden und diese konnte das Sammeln der Daten übernehmen. Für die GPU brauchten nur die empfangenen Daten hochkopiert werden. Mit einem verändertem Vektor müssen die Daten schon auf der GPU gesammelt werden, um möglichst wenig Daten runterkopieren zu müssen. Dies bedeutet zudem einen zusätzlichen Kernel Launch. Damit ergibt sich für die GPU für iterative Löser noch folgende Sonderbetrachtung:

$$t_{totalPART_i}(p) = (5.10) + \begin{cases} \text{Kernel Launch} & \text{async} \\ download_p + \text{Kernel Launch} & \text{sync} \end{cases} \tag{5.11}$$

6. Implementierung

Dieses Kapitel beschäftigt sich mit den Implementierungsdetails dieser Arbeit. Wesentlicher Bestandteil ist die Integration der Partitionierungsstrategie in das Framework. Dazu werden kurz die grundsätzlichen Design-Ziele und die angestrebte Funktionalität dargelegt sowie der daraus abgeleitete Aufbau und angebotene Schnittstellen erläutert. Es folgen einzelne Aspekte, die in den Kern der C++-Schicht hinzugefügt werden mussten, um die Verteilung von Daten sowie das Analysieren von Zeiten zu ermöglichen.

6.1. Design-Ziele und deren Umsetzung in LAMA

Übergeordnetes Ziel in LAMA ist die Benutzerfreundlichkeit. Dem Nutzer soll die Formulierung seiner Algorithmen leicht fallen und es soll sich dabei nicht um Detailfragen kümmern müssen. So wird in den Compute Backends auf der C-Schicht sämliche spezielle Programmierung der verschiedenen Hardwaretypen gekapselt und von der C++-Schicht so verwaltet, dass er sich nicht darum kümmern muss wie das Verteilen der Daten geschehen muss oder wie die Prozessoren während der Berechnungs miteinander kommunizieren müssen.
Bei der Umsetzung stellen Wiederverwendbarkeit und leichte Erweiterbarkeit wesentliche Design-Ziele von LAMA dar. Auf der zweiten Schicht des Frameworks wurden deshalb vielfältige Konzepte der Programmiersprache C++ umgesetzt. Neben grundsätzlicher Objekt-Orientierung wird mit (teilweise mehrstufigen) Vererbungen und Templatisierung generische Schnittstellen-Entwicklung ermöglicht. Bestehende Konzepte lassen sich somit prinzipielle über das Ableiten von Basis-Klassen erweitern.
Ein Beispiel dafür ist die Umsetzung der Text-Buch-Syntax[1] über *Expression Templates*, die sämliche Matrix-Formate in einfacher und doppelter Genauigkeit mit einmal abdecken und somit Multi-Precision-Fähigkeit erzielen. Ein neues Sparse-Matrix-Format kann so einfach durch das Ableiten der Basis-Klasse `SparseMatrix`, dem Hinzufügen zugehöriger Interfaces zu den Compute Backends sowie der Implementierung in diesen ergänzt werden und um die verteilte Berechnung mit diesem Format kümmert sich der Rest des Frameworks automatisch.
Ergänzt wird dies durch das Entwurfsmuster der *Factory*, wie es in [GHJV95] vorgestellt wird. Mit den verschiedenen (gleichzeitig) einzusetzenden compute locations sowie mehreren möglichen Bibliotheken für die Kommunikation zwischen verteilten Speicherbereichen

[1] Die Text-Buch-Syntax bezeichnet die Programmierung von Algorithmen der linearen Algebra in Form von einfachen Matrix-Vektor-Operationen, wie z. B. `Vector y = A * x;`.

existieren verschiedene Implementierungen eines anzuwendenden Konzeptes, z. B. die Berechnung mit CUDA oder die Kommunikation über MPI. Aber nicht jede Variante steht auf einem System zur Verfügung, da bspw. keine CUDA-Karten vorliegen oder keine MPI-Bibliothek. Mit dem Factory-Muster registrieren sich zur Laufzeit die verfügbaren Klassen bei der Factory, die sie verwaltet und auf Anfrage zurückliefert. Somit können nur registrierte Klassen benutzt werden. Zudem gibt es für diese Konzepte immer eine Default-Lösung (→ Einfachste: unverteilte sequentielle CPU-Berechnung), die in jedem Fall vorhanden ist. Dadurch kann ein mit LAMA geschriebener Programm auf jeder vorliegenden Hardware ohne Änderung am eigentlichen Algorithmus ausgeführt werden.

6.2. Funktion der Partitionierungsstrategie

Bis jetzt stehen für die verteilte Berechnung mit LAMA die in Absatz 4.2.1 vorgestellten Verteilungen zur Auswahl. Wie bereits erläutert sind mit den vier möglichen Verteilungen alle Möglichkeiten für eine zeilenweise Partitionierung offen. Allerdings muss der Nutzer bei der Wahl der Verteilung selbst entscheiden, was für seine Zwecke angebracht ist.
Eine blockweise und zyklische Verteilung macht für Problemstellungen mit strenger Struktur Sinn, ein homogenes System ist aber dann Vorraussetzung, da jeder Partition gleich viele Zeilen zugeordnet werden. Mit der allgemeinen Block-Verteilung ist eine Gewichtung, wie sie für ein heterogenes System nötig ist, möglich. Die Matrix-Struktur kann so allerdings nur innerhalb der Block-Struktur berücksichtigt werden. Für die allgemeine Verteilung wird eine genaue Zuordnung jeder Zeile zu einer Partition benötigt. Dazu ist eine genau Analyse der Matrix nötig.
Die Graph-Partitionierung ist ein gängiges Werkzeug zur Berücksichtigung der Struktur dünn besetzter Matrizen und wird häufig zur Konditionierung (Strukturoptimierung, Balancierung und Kommunikationsminimierung) verwendet. Deswegen soll dem Nutzer diese Funktionalität innerhalb von LAMA zur Verfügung stehen. Neben der reinen Funktionalität der Graph-Partitionierung ist es mit der Partitionierungsstrategie das Ziel dem Nutzer die Frage nach einer effizienten Aufteilung abzunehmen.
Die Integration der Graph-Partitionierung soll mit den generellen Design-Zielen von LAMA konform gehen und sich damit neben der eben erläuterten benutzerfreundlichen Schnittstelle auch in den Punkten Wiederverwendbarkeit und leichte Erweiterbarkeit einfügen. So sollte das Design ein einfaches Hinzufügen verschiedener Partitionierungsansätze in Form von mehreren Graph-Partitionierungs-Bibliotheken ermöglichen.

6.2.1. Schnittstellen für die Partitionierung

Für die Generierung von Verteilungen müssen dem Nutzer zum einem Funktionen zur statischen Bestimmung und zum anderen zur dynamischen Balancierung zur Verfügung stehen. Bei der initialen Generierung mit einer statischen Methode sollte er die Möglichkeit haben selbst Gewichte für die Partitionierung anzugeben oder diese vom Framework

berechen zu lassen. Für die dynamische Partitionierung kann eine Neuberechnung der Gewichte und Partitionierung oder aber ein Verschieben von Zeilen zwischen benachtbarten Prozessoren sinnvoll sein. Insgesamt leiten sich die folgenden Schnittstellen ab:

statisch

`giveWithWeights` Anhand vom Nutzer festgelegter Gewichte wird mit der Graph-Partitionierung eine Verteilung für eine Matrix berechnet.

`getDistribution` Mit Hilfe des vorgeschlagenen Gewichtungs-Modells wird selbständig eine Verteilung berechnet. Die nötigen Informationen zum System sollen dafür vom Benutzer in einer Konfigurations-Datei hinterlegt werden.

dynamisch

`recalc` Auf Basis der Ausführzeiten der letzten Iteration wird eine neue Gewichtung veranschlagt, die in eine neue Verteilung umgesetzt wird.

`shift` Ebenso wird eine neue Gewichtung anhand der zuvor erfolgten Ausführzeiten berechnet. Die neue Verteilung wird aber druch Verschieben von Zeilen zwischen den bestehenden Partitionen erreicht.

Grundsätzlich stehen dem Nutzer damit nicht die zahlreichen Möglichkeiten offen, die ihm eine Graph-Partitionierungs-Bibliothek bietet, wie z. B. die Auswahl aus verschiedenen Algorithmen oder Optimierungsmetriken. Dies steht aber auch nicht im Vordergrund, sondern vielmehr die einfach Benutzung der bestehenden Strategie. Der erfahrene Nutzer von solchen Bibliotheken kann diese leicht selbst in sein Programm einbinden und sich daraus eine Verteilung erstellen. Deswegen das schmale Interface.
Mit den zwei verschiedenen dynamischen Routinen soll der Nutzer selbst anhand der Anzahl der Iterationen entscheiden können, was sich für ihn lohnt, wil sowohl eine Neuberechnung als auch das Verschieben von Zeilen seine Vor- und Nachteile hat. Eine Neuberechnung der Verteilung minimiert wieder den Kantenschnitt und damit das Kommunikationsvolumen, allerdings kann sich damit viel an der Verteilung ändern, wodurch viele Daten umverteilt werden müssen. Beim Verschieben von Zeilen bleibt die grundsätzliche Verteilung erhalten, aber das Kommunikationsvolumen wird nicht beachtet. Dadurch muss vergleichsweise wenig zur Umverteilung kommuniziert werden, dafür aber aller Voraussicht nach in der (iterativen) Berechnung. Eine Neuberechnung der Verteilung macht daher bei einer großen Veränderung der Gewichtung Sinn, während sich für ein feines Ausbalancieren der Laufzeit eher das Vertauschen der Zeilen bezahlbar macht.

6.2.2. Klassen-Struktur

Mit der Schnittstellendefinition ist festgelegt welche Funktionalität eine Partitionierungsstrategie nach außen anbieten muss. Innerhalb der Klasse beziehen die ersten drei Funktionen die Gewichte und sprechen dann die jeweilige Bibliothek an. Die *shift*-Routine arbeitet für alle Strategien auf gleiche Art und Weise. Daraus ergibt sich ein grundsätzlich hierarischer Aufbau für Partitionierungsstrategien: die `Distribution Strategy` stellt eine

abstrakte Basis-Klasse dar, konkretisiert wird sie von spezialisierten Verteilungsstrategien. Die `Distribution Strategy` implementiert die Berechnung der Gewichte über das Modell oder die vorherigen Laufzeiten sowie die dynamische Balancierung des Shiftens, während die spezifische Ansteuerung der Bibliothek in den spezialisierten Klassen geschieht. Dabei müssen nur die abstrakten Schnittstellen-Funktionen implementiert werden, sodass es sich bei den Implementierungsaufwand um Dinge handelt, die mit der neuen Bibliothek in Kontakt stehen. Allgemeinen Funktionen wie das Gewichtungsmodell oder die dynamische Balancierung des Shiftens wird über die Basisklasse bereitgestellt. Insgesamt ergibt sich folgendes Klassenschema für die derzeit einzig angebundene Metis-Bibliothek:

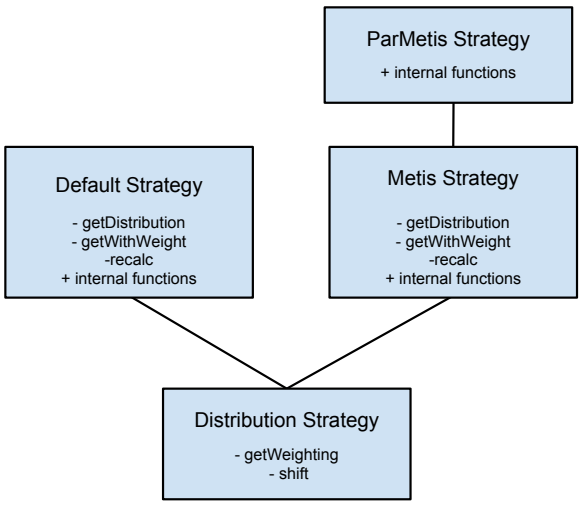


Abbildung 6.1.: Bestehende Klassenstrukur der Verteilungsstrategien

6.2.3. Organisation der Verteilungsstrategien

Eine Verteilungsstrategie kann nur existieren, wenn die jeweilige Bibliothek auf dem System vorhanden ist. Das nicht-Vorhandensein darf jedoch nicht zur Compile-Zeit der Bibliothek zu Fehlern führen. Dazu muss eine lose Kopplung zwischen den aufrufenden Funktionen und den Strategien vorliegen. Dies betreffend wird das erwähnte Factory-Muster wieder aufgegriffen. Existierende Strategien melden sich bei einer Registry an, die diese verwaltet und dem Nutzer über statische Funktionen zur Verfügung stellt oder zur Laufzeit des Programmes einen Fehler liefert. Sollte auf einem System keine Bibliothek zur Verfügung stehen, gibt es eine Default-Strategie, die verwendet werden kann. In diesem Fall wird auf die allgemeine Block-Verteilung zurückgegriffen.

6.2.4. Integration der Partitionierungsstrategie in LAMA

Im Folgenden sollen kurz Aspekte, die bei der Implementierung der Strategie wichtig waren, angeschnitten werden. Wesentliche Komponenten waren neben der bereits erwähnten Klassen-Struktur und Organisation der Verteilungsstrategien, die Graph-Modellierung, das Berechnen der Gewichtung sowie die Zeitmessung innerhalb der SpMV-Routine für die Evaluierung und Analyse bei der dynamischen Balancierung.

Modellierung des Graphen

Für die Graph-Partitionierung wird eine Modellierung der Matrix als Graph benötigt. Dazu wird derzeit, dass Modell des ungerichteten Graphen verwendet, da es sich besonders leicht durch Wiederverwendung bestehender Konzepte in LAMA integieren ließ.
Die Darstellung als Adjazenzliste des Graphen ist vergleichbar mit der Speicherung der Matrix im CSR-Format. Da die Liste Knotenweise gespeichert wird, werden die Elemente damit genauso durchlaufen wie bei der zeilenweise Speicherung der Indizes in ja der Matrix. Einziger Unterschied ist, dass im Graphen das Diagonalelement nicht abgebildet wird, da es explizit im lokalen Teil der Partition dieser Zeile liegt.
Somit muss für den Aufbau des Graphen aus einer Matrix diese in eine CSR-Matrix konvertiert und die Diagonal-Einträge aus dem bestehenden Indexarray ja gelöscht werden. Das Konvertieren zwischen den Sparse-Matrix-Formaten gehört ohnehin zu den Aufgaben des Frameworks, da Matrizen i. d. R. im CSR-Format eingelesen werden.

Gewichtungsmodell

Für die Angabe einer Gewichtung nach dem vorgeschlagenen Modell, müssen die Laufzeiten für die einzelnen Partitionen vorhergesagt werden. Aufbauend auf dem Roofline-Modell wird dazu neben den maximalen Leistungsdaten der Prozessoren auch die OI für die Berechnung auf der jeweiligen compute location benötigt. Im Detail müssen für die Berechnung der Gewichtung die folgenden Daten vorliegen:

- matrixbezogen:
  - **Anzahl nicht-null-Einträge der Matrix (nnz)** Nur über die Anzahl der nicht-null-Einträge kann die Leistungsbewertung des Roofline-Modells in die Laufzeit für die spezielle Matrix umgerechnet werden. Dies ist wichtig, um Einflussfaktoren auf die Leistung für die spezielle Matrix einzubeziehen.
  - **compute location** Die compute location legt fest auf welchem Prozessortyp die Berechnung ausgeführt werden soll und damit welches Bewertungsschema angesetzt werden muss.
  - **communication type** Innerhalb des Bewertungsschemas muss für die dynamische Balancierung entschieden werden, ob die Kommunikation mit in die Laufzeitvorhersage eingehen muss oder nicht.
  - **einfache oder doppelte Genauigkeit** Abhängig von der Datengröße passen unterschiedlich viele Einträge in eine Cacheline bzw. können gleichzeitig mit einem coalesced memory access geladen werden. Deswegen sind diese Größen wichtig für das Zählen der Speicherzugriffe.
- hardwarebezogen:
  - **maximale Rechenleistung** Die maximale Rechenleistung stellt eine von zwei maximalen Leistungsschranken (des Roofline-Modells) dar. Sie kann nur von compute bound Algorithmen erreicht werden, die die volle Anzahl an Recheneinheiten ausnutzt. Festgelegt wird sie neben der Anzahl dieser von der Taktfrequenz des Prozessors.

maximale Speicherbandbreite Die maximale Speicherbandbreite kann zum einen theoretisch angegeben werden, aber auch durch Benchmarks, wie STREAM [McC07], praktisch bestimmt werden.

Cacheline size bzw. coalesced access size Dabei handelt es sich um feste Hardwaregrößen. Die Cacheline Size für den Level 1 Cache einer CPU findet man in `/sys/devices/system/cpu/cpu0/cache/size`, die coalesced access size wird über die compute capabilty der Grafikkarte bestimmt.

Kernel Launch Zeit Die Kernel Launch Zeit muss individuell für jeden Prozessor bestimmt werden. Dazu muss ein leerer CUDA-Kernel gestoppt werden.

Die matrixbezogenen Daten können direkt auf der Matrix abgefragt werden. Die hardwarebezogenen Informationen sollen hingegen vom Nutzer in einer Datei angegeben werden und vom Programm in einer Struktur verwaltet werden, die die Hardware des verteilten Systems repräsentiert, wie es auch in DRUM [Fai05] gemacht wird. Wichtig ist, dass die Bestimmung der hardwareseitigen Daten nicht während des Programmlaufes (wiederholt) durchgeführt werden muss, da ansonsten die Bewertung der Komponenten zu lang braucht.

6.2.5. Zeitmessung

Für die Evaluierung der Balance und des Kommunikationsoverheads, aber auch für Analysezwecke in der dynamischen Balancierung ist eine dedizierte Laufzeitbetrachtung innerhalb der SpMV-Berechnung nötig. Da diese nach außen über ein Expression Template zur Verfügung steht ist dort nur ein Messen der Gesamtberechnung möglich und es können ebensowenig Laufzeitinformationen von der Routine zurückgegeben werden. Deswegen müssen die Zeiten innerhalb der Routine gemessen und außen über statische Funktionen zugreifbar gemacht werden. Dazu werden die Zeiten in einer statischen Liste, implementiert als *Singleton*[GHJV95], über einen Schlüssel gespeichert und können über diesen von außen wieder ausgelesen werden. Bei dem Schlüssel handelt es sich um den Methodennamen der Unterroutinen, die einzeln gemessen werden sollten:

- die Berechnung des lokalen Anteils der Matrix
- das Einsammeln der Daten für die Halo-Kommunikation
- die Halo-Kommunikation
- die Berechnung des Halo-Anteils der Matrix
- die gesamte SpMV

Die Zeitmessung erfolgte für die CPU über `MPI_Wtime()`. Für die GPU wurden die Routinen der CUDA-API verwendet. Da die Berechnung innerhalb eines Kernels asynchron gestartet werden kann, muss die Zeitmessung der Kernel über Events erfolgen.

6.3. Umverteilung zwischen den Prozessoren

Um in der dynamischen Balancierung eine direkte Umverteilung einer bestehenden Matrix vorzunehmen, muss ein Kommunikationsmuster aufgebaut werden nach dem die Zeilen zwischen den Prozessoren danach ausgetauscht werden können. Dazu muss jeder Prozessor jedem anderem mitteilen wie viele Zeilen er nach der neuen Verteilung an ihn senden möchte. Erst danach können sie die globale Position der Zeile, sowie die zugehörigen Daten kommunizieren. Für diesen Austausch muss entsprechend ein einheitliches Protokoll zugrunde liegen. Man spricht dabei auch von der Serialisierung von Daten, da die Strukturen durch den Austausch von Nachrichten nur in Form eines Datenstromes erfolgen kann. Außerdem muss eine einheitliche Darstellung der Daten gegeben sein, denn kann die Matrix auf den verschiedenen Prozessoren in unterschiedlichen Speicherformaten vorliegen. Dafür wurde ein vereinfachtes CSR-Format gewählt, dass sich anstatt der Indizierung die Anzahl der Elemente pro Zeile in *ia* speichert. Zudem kommt hinzu, dass bei einer bereits verteilten Matrix, die sich in einen lokalen und Halo-Anteil aufteilt, wieder zusammengesetzt werden musss. Die Umverteilung der Daten setzt sich damit aus folgenden Schritten zusammen:

- Aufbau des Kommunikationsmusters
- Zusammensetzen des lokalen und Halo-Anteils
  (mit Umrechnung von lokalen Indizes zu globalen)
- Konvertierung in das einheitliche Speicherformat
  mit gleichzeitigem Serialisieren der Daten
- Versenden der Daten
- Deserialisierung der empfangenen Daten
  und Konvertierung in das eigens verwendete Speicherformat
- Aufteilung in lokale und Halo-Matrix

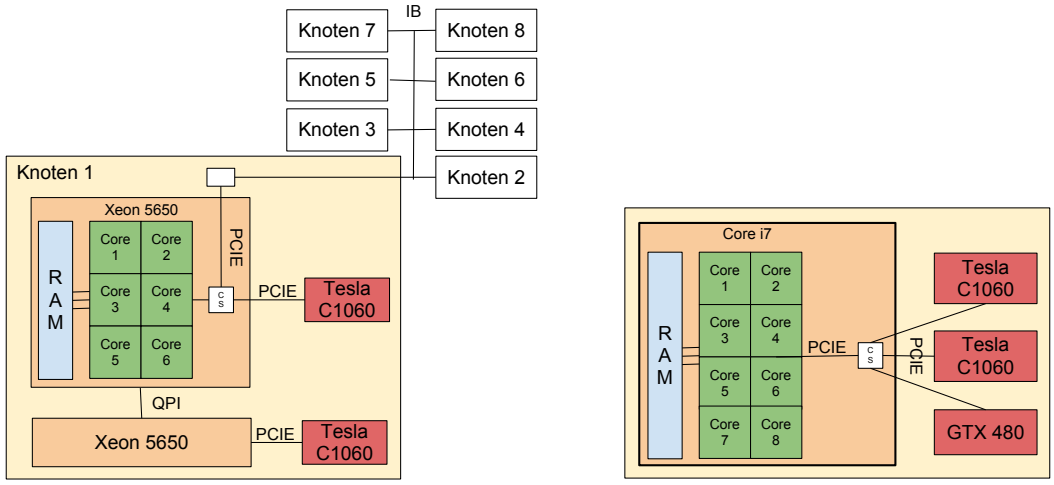
Auch hier konnte eine hohe Wiederverwendung erzielt werden. Zum einem konnte nach dem Aufbau des Kommunikationsmusters in Form des `Kommunikationsplans` der Austausch über bestehende Routinen für den Austausch während der Berechnung verwendet werden. Zum anderen wurde wieder die Konvertierung der Formate genutzt.

7. Auswertung

In diesem Kapitel soll die entwickelte Partitionierungsstrategie evaluiert werden. Dazu werden zu Beginn die zur Verfügung stehenden Test-Systeme sowie die Testmatrizen vorgestellt. Danach werden schrittweise die Einzelkomponenten des Modells, bestehend aus der Graph-Partitionierung, der Laufzeitvorhersage und der daraus abgeleiteten Gewichtung analysiert, bevor das Gesamtmodell bewertet wird. Zum Schluss wird beurteilt, wann eine solche Partitionierung sinnvoll einsetzbar ist.

7.1. Test-Systeme

Bei der Entwicklung des Modells stand der hybride Charakter von Rechensystemen im Vordergrund. Eingeschlossen sind darin vor allem Systeme mit verschiedenen Prozessoren unterschiedlicher Prozessortypen. Eine zusätzliche Heterogenität im Verbindungsnetzwerk zwischen den Einzelkomponten des Systems wurde für die einfache Verwendungen der frei verfügbaren Graph-Partitionierungsbibliotheken ausgeschlossen. Für die Evaluierung der Partitionierungsstrategie werden daher Systeme benötigt, die eine möglichst hohe Heterogenität in der Auswahl der Prozessoren aufweisen aber alle über dasselbe Netzwerk verbunden sind. Einzige Ausnahme ist die Verbindung zu den GPUs, die gesondert im Modell einbezogen wurde, da diese Verbindung zum wesentlichen Charakter hybrider Systeme gehört.



(a) Bull-Cluster **(b)** F1Boensch

Abbildung 7.1.: Schemata der Test-Systeme

Für die Messungen standen zwei Systeme des Fraunhofer Instituts SCAI zur Verfügung: das Bull-Cluster und der Rechner F1Boensch. Das Cluster soll insbesondere für die Aufteilung auf mehrere Partitionen genutzt werden, der F1Boensch hingegen um das Modell bei einer höheren Heterogenität zu testen. Schematische Skizzen des Aufbaus der Systeme sind in Abbildung 7.1 zu finden.

Im Bull-Cluster findet man pro Knoten zwei Intel Xeon 5650 Prozessoren, an die jeweils eine Tesla C1060 GPU angebunden ist. Um die Netzwerk-Heterogenität auszuschließen wurde jeweils nur ein Prozessor des Zwei-Sockel-Systems genutzt, weil zwischen den Prozessoren eine QuickPath Interconnect (QPI)-Verbindung vorliegt, während die Knoten untereinander mit Infiniband verbunden sind. Im F1Boensch befindet sich ein Intel Core i7 Prozessor, der mit einer GTX 480 und zwei Tesla C1060 GPUs ausgestattet ist. Die GPUs sind jeweils über den PCI-Express-Bus angeschlossen. Die wesentlichen Kennzahlen der Prozessoren sind in den Tabellen 7.1 (CPUs) und 7.2 (GPUs) angegeben, detailierte Informationen sind in A.2 zusammengestellt.

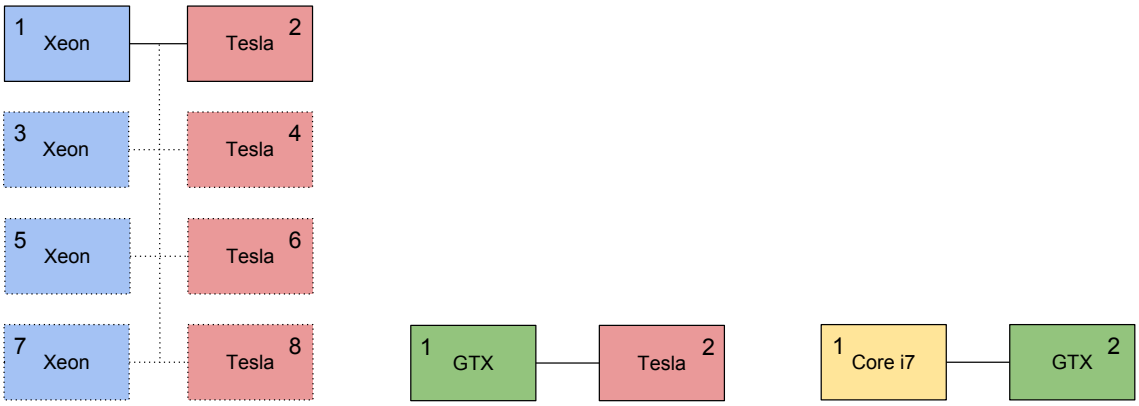
	Intel Xeon 5650	Intel Core i7
Kerne	6	8
Taktung	2.67 GHz	2.67 GHz
Bandbreite	32 GB/s	21 GB/s

Tabelle 7.1.: CPU-Kennzahlen

	Tesla C1060	GTX 480
Kerne	240	480
Taktung	1.3 GHz	1.4 GHz
Bandbreite	102.4 GB/s	177.4 GB/s

Tabelle 7.2.: GPU-Kennzahlen

Für die Tests wurden verschiedene Konfigurationen auf den beiden Systemen angenommen. Zum einen auf dem Bull mit verschieden vielen Prozessoren, zum anderen unterschiedliche Kombinationen der Prozessoren auf dem F1Boensch. Abbildung 7.2 zeigt die in den Tests verwendeten Konfigurationen im Überblick.



(a) Bull: 2 - 8 Prozessoren **(b)** F1Boensch: Konfig. 1 **(c)** F1Boensch: Konfig. 2

Abbildung 7.2.: Test-Konfigurationen

Für die SpMV-Berechnung ist im Modell, wie bereits erwähnt, die Speicherbandbreite wichtig. Repräsentative Werte für die real zu erreichende Leistung (gegenüber der Peak-

Performance aus den Tabellen oben) wurden auf den CPUs mit dem Stream-Benchmark (TRIAD-Operation) und auf der GPU mit dem Bandwidth-Test von Nvidia bestimmt. Die Ergebnisse sind in den Tabellen 7.3 (CPU) und 7.4 (GPU) zusammengetragen.

Intel Xeon 5650 (6 Threads)	Intel Core i7 (8 Threads)
15.29 GB/s	9.05 GB/s

Tabelle 7.3.: Speicherbandbreite für Intel Xeon 5650 und Intel Core i7 anhand des Stream-Benchmark (TRIAD-Operation)

	Tesla C1060	GTX 480
Upload	5.72 GB/s	3.03 GB/s
Download	5.31 GB/s	3.20 GB/s
Device to Device	73.67 GB/s	117.90 GB/s

Tabelle 7.4.: Speicherbandbreiten für Tesla C1060 und GTX 480 anhand des BandwidthTests; Upload: Speicherbandbreite von Host (CPU) zum Device (GPU); Download: Speicherbandbreite vom Device zum Host; Device to Device: Speicherbandbreite auf den GPU-Speicher; die Messung erfolgt für pinned memory

7.2. Test-Matrizen

Bei dünn besetzten Matrizen hat die Struktur der Matrix einen entscheidenden Einfluss auf das Laufzeitverhalten bei der SpMV-Berechnung. Eine bedeutende Rolle spielt die Verteilung der Einträge über die Zeile, da sie das Zugriffsmuster auf den Quell-Vektor beschreibt. Optimal ist eine hohe Lokalität der Daten, damit die Vektor-Einträge aus dem Cache wiederverwendet werden können. Außerdem ist die Anzahl der Einträge pro Zeile relevant. Insbesondere ist die Streuung dieses Wertes bezeichnend. Variiert die Anzahl der Einträge pro Zeile stark, wird mit herkömmlichen Partitionierungsmethoden kein optimales Gleichgewicht bzgl. der Anzahl der Einträge erzielt. Zudem steigt die Wahrscheinlichkeit, dass durch die unterschiedliche Anzahl von Einträgen pro Zeile die Wiederverwendung der Einträge im Vektor x geringer ist. Für die GPU hat es den zusätzlichen Nachteil, dass die Berechnung innerhalb eines CUDA-Blockes nicht balanciert ist.

Um die unterschiedliche Anzahl der Einträge pro Zeile für die Partitionierung zu beachten, wird die Graph-Partitionierung verwendet. Damit erfolgt ebenso eine Strukturoptimierung der Matrix, da die Einträge in den Diagonal-Block permutiert werden. Innerhalb dieses Blockes ist aber keine besondere Struktur gegeben, sodass nicht explizit eine Verbesserung für den Cache erzielt wird. Dadurch wird auch nicht garantiert, dass in jeder Zeile der Blockes in etwa gleich viele Einträge vorkommen, sodass eine volle Ausnutzung der CUDA-Threads gegeben wäre. U. A. deshalb nimmt die Struktur der Matrix einen enormen Einfluss auf die Berechnungzeit ein.

Für die Tests sollten zum einen Matrizen verwendet werden, die den Einschränkungen des Modells so wenig wie möglich zum Opfer fallen. Dazu wurden Block-Matrizen[1] künstlich erzeugt. Diese bestehen aus diagonal angeordneten dicht besetzten Blöcken. Damit ist eine konstante Anzahl von Einträgen pro Zeile sowie eine optimale Lokalität der Daten geschaffen. Die Größe der Blöcke ist im Vergleich zur Matrix-Größe relativ gering. Damit ist zum einem dem dünnbesetzten Charakter genüge getan, zum anderen ist die Ausdehnung der Datenabhängigkeit klein, sodass möglichst gut balancierte Partitionen erstellt werden können ohne die Blöcke aufteilen zu müssen.

Daneben stellen Poisson-Matrizen[2], die sich aus der Diskretisierung von Differenzialgleichungen ergeben (für eine Strukturherleitung s. Anhang A.1), eine zweite Klasse Matrizen dar. Sie bestehen aus der Hauptdiagonalen und mehreren Nebendiagonalen, wobei zwischen den Diagonalen unterschiedlich große Abstände bestehen. Die Diagonalstruktur fördert grundsätzlich die räumliche Lokalität der Daten, wird aber von den Abständen gestört, sodass häufiger Cache-Misses auftreten werden.

Außerdem werden reale Matrizen, mit grundlegend unstrukturierter Form, aus möglichen Anwendungsfeldern getestet. Sie wurden aus dem Matrix Market [BPR+97] sowie der Sparse-Matrix-Collection der University of Florida [Dav94] entnommen. Die Matrizen kommen aus vielfältigen Bereichen wie Optimierungsproblemen, Strukturproblemen oder stellen ungerichtete Graphen dar. Die Auswahl der Matrizen erfolgte rein zufällig. Einzige Gemeinsamkeit unter ihnen ist, dass sie alle struktur-symmetrisch sind, um mit dem Modell der ungerichteten Graphen (s. Absatz 2.4.2) beschrieben werden zu können und dass sie eine gewisse Größe haben, damit sich die verteilte Berechnung lohnen kann.

Name		Zeilen	Einträge	Füllgrad	$\varnothing \frac{\text{Einträge}}{\text{Zeile}}$
künstliche Matrizen					
(a)	Block_30000_70	30.000	2.100.000	2,33e-03	70
	Block_1000000_50	1.000.000	50.000.000	5,00e-05	50
	Block_1500000_45	1.500.000	67.500.000	3,00e-05	45
(b)	Poisson_3D27P_100_100_100	1.000.000	26.463.592	2,64e-05	27
	Poisson_3D7P_150_150_150	3.375.000	23.490.000	2,06e-06	7
reale Matrizen					
(c)	bcsstk30	28.924	1.036.208	2,44e-05	35
(d)	Geo_1438	1.437.960	63.156.690	3,05e-05	43
(e)	ldoor	952.203	42.493.817	5,13e-05	49
(f)	nd3k	9.000	3.279.690	4,04e-02	364
(g)	nlpkkt160	8.345.600	229.518.112	3,29e-06	28
(h)	pattern	19.242	9.323.432	2,51e-02	485
(i)	thread	29.736	4.470.048	5,06e-03	150

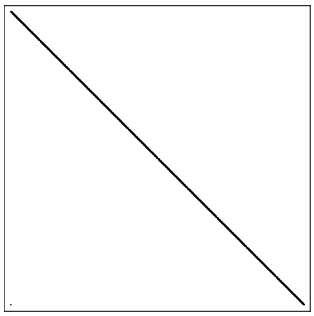
Tabelle 7.6.: Details über Größe und Füllgrad der Test-Matrizen

[1] Die Namensgebung der Block-Matrizen folgt dem Schema: Block_*#Zeilen_Blockgröße*.
[2] Die Namensgebung der Poisson-Matrizen folgt dem Schema:
Poisson_*Dimension*D*#Punkte*P_*Ausdehnung Dim1[_Dim2_Dim3]*.

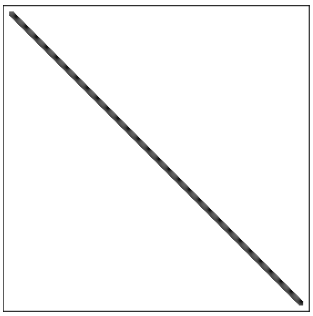
Einen Überblick der, für die Tests verwendeten, Matrizen verschafft Tabelle 7.6, die die wichtigsten Details der Matrizen protokolliert sowie Abbildung 7.3 mit Struktur-Plots, die ein Bild über die Verteilung der Einträge innerhalb dieser Matrix geben. Einen detailierten Blick „in“ die Struktur der Block- und Poisson-Matrizen zeigt Abbildung 7.4.

7.3. Partitionierungsziel und Einflussfaktoren

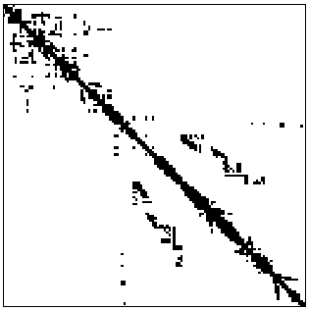
Das Ziel der Partitionierung ist eine lastbalancierte Ausführung bei minimalen Kommunikationsoverhead, um eine optimale Laufzeit zu erzielen. Idealerweise sollte so für die Leistung der verteilten Berechnung die Summe der Einzelleistungen der Prozessoren erreicht werden. Ein balanciertes System ermöglicht den größtmöglichen Gewinn, da keine Leistung in unnötiger Wartezeit verloren geht. Je weniger Overhead durch die Kommunikation eingeführt wird desto mehr bleibt vom Leistungsgewinn bestehen.
Damit sich die Aufteilung der Rechenlast über die Prozessoren lohnt, muss jeder gleichermaßen beschäftigt sein und der Overhead kleiner als die Laufzeitverbesserung durch die Arbeitsteilung sein. Dafür muss jeder Prozessor angemessen zu seiner Leistung Zeilen der Matrix zugeteilt bekommen, wobei möglichst wenige (verschiedene) Einträge in seinem Halo-Anteil liegen sollten, um wenig Kommunikation hervorzurufen. Für eine Ausführung mit asynchroner Kommunikation muss die Kommunikationszeit lediglich geringer als die Berechnungszeit für den lokalen Anteil der Matrix sein, um so keinen Einfluss auf die Gesamtlaufzeit zu nehmen. Bei synchroner Kommunikation sollte die Kommunikationszeit lediglich so gering wie möglich sein.
Eine geringere effektive Laufzeit erhält man bei einer Aufteilung über eine möglichst große Anzahl Prozessoren. Das Skalierungsverhalten hängt aber von der Größe und Struktur der Matrix ab, da mit zunehmender Prozessoranzahl auch mehr kommuniziert werden muss. Zudem wird der lokale Anteil der Matrix immer kleiner und damit auch die Berechnungszeit hinter der die Kommunikation versteckt werden kann.
Einfluss auf die Balance der Berechnung nimmt die Bewertung der Rechenleistung der Systemkomponenten, die Partitionierungsmethode sowie nicht berücksichtigte Einflüsse wie die Matrixstruktur. Bei einer falschen Bewertung werden den Prozessoren unangemessene Rechenlasten zugewiesen. Aber auch bei einer korrekten Gewichtung der Partitionen kann eine schlechte Partitionierungsmethode die Last ungünstig verteilen, weil sie die Gewichte nicht korrekt auf die Partitionen überträgt. Durch die Verteilung wird außerdem die Struktur der Matrix verändert, sodass die Leistung entgegen der Erwartung schlechter ausfallen kann als im unverteilten Zustand.
Die Kommunikation wird von der Kommunikationleistung der Prozessorpaare sowie vom Kommunikationsaufwand beeinflusst. Da die Leistung für das Modell als homogen angenommen und in den Testkonfigurationen entsprechen umgesetzt wurde, sollte die Kommunikation lediglich vom effektiven Aufwand bestimmt werden. Dieser wird maßgeblich durch die Menge der zu kommunizierenden Daten und durch die Anzahl der Kommunikationspartner festgelegt, welche wiederum von der Partitionierungsmethode abhängen.



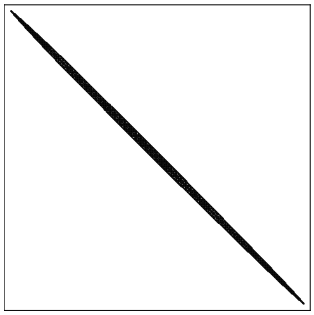
(a) Block_30000_70



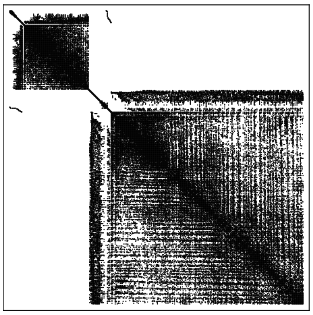
(b) Poisson_3D27P_100_-100_100



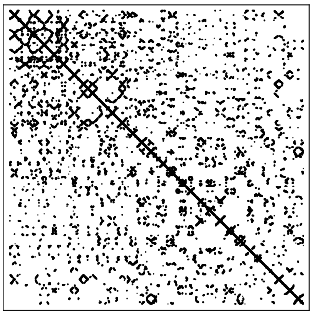
(c) bcsstk30



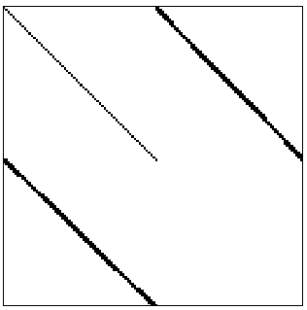
(d) Geo_1438



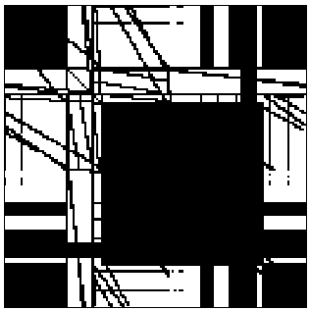
(e) ldoor



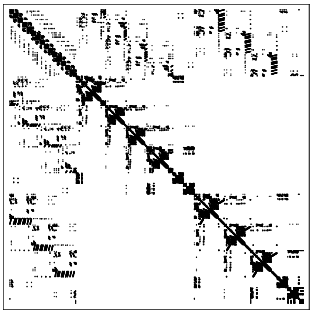
(f) nd3k



(g) nlpkkt160

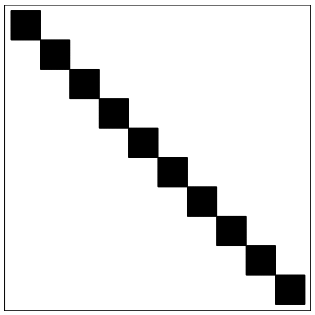


(h) pattern1

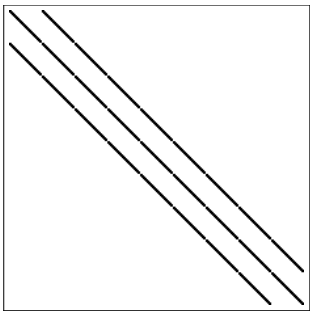


(i) thread

Abbildung 7.3.: Test-Matrizen



(a) Block_30000_70, Zoom 700x700



(b) Poisson_3D27P_100_-100_100, Zoom 900x900

Abbildung 7.4.: Zoom in die künstlichen Test-Matrizen

Abgesehen von der Menge der Daten und der Aufteilung auf die verschiedenen Nachrichten kann sich auch das Kommunikationsmuster (die Zuordnung wer mit wem Daten austauschen muss) auf die Zeit für die Kommunikation auswirken. Am besten ist daher auch eine Balance in der Kommunikationlast für bestehende Prozessorpaare, da dann jede Kommunikation in etwa gleich lang dauert und keine Wartezeit auf den nächsten Kommunikationspartner entsteht[3]. Dies wird allerdings auch bei der Graph-Partitionierung aktuell nicht beachtet.

7.4. Auswertung der Einzelkomponenten

Der gesamtheitliche Ansatz zur Partitionierung von dünn besetzten Matrizen für hybride Systeme setzt sich aus dem Finden einer passenden Gewichtung und geeigneter Schnitte zusammen. Gemeinsames Ziel dabei ist eine gute Lastbalance zwischen den Partitionen und eine Minimierung des Overheads. Das Gewichtungs-Modell bestimmt dazu eine Lastaufteilung, die der Leistung der Systemkomponenten gerecht werden soll; die Graph-Partitionierung stellt ein Werkzeug dar, dass versucht die Aufteilung für die jeweilige Matrix möglichst gleichmäßig herzustellen und dabei wenig Overhead einzuführen. Nachfolgend sollen die Bausteine einzeln und dann kombiniert analysiert werden. Begonnen wird mit dem bestehenden Ansatz der Graph-Partionierung.

7.4.1. Graph-Partitionierung im Vergleich zu herkömmlichen Partitionierungen

Die Idee der herkömmlichen Partitionierungsmethoden (blockweise und zyklisch) kommt aus Überlegungen des Schedulings von Aufgaben, wie man es von der Schleifenparallelisierung z. B. mittels OpenMP [opeb] kennt. Dabei soll Lastbalance für die Aufteilung einer Menge an (verschieden) großen Aufgaben auf Prozessoren erzielt werden. Eine blockweise Aufteilung der Aufgaben ist einfach und erreicht für viele, im Mittel gleich große, Probleme eine zufriedenstellende Lösung. Die zyklische Verteilung ist vorteilhaft um Größenunterschiede zwischen den Aufgaben auszugleichen, da sie ein feineres Verteilungsmuster hat. Daneben gibt es dynamische Verfahren, die zur Laufzeit die Lasten zuteilen.
Für die zeilenweise Partitionierung einer dünn besetzten Matrix wird die Anzahl der Einträge in einer Zeile als Aufgabengröße angenommen. Diese Last gilt es zu verteilen. Allerdings reicht eine Lastbalance allein nicht aus, auch der Kommunikationsoverhead muss betrachtet werden (vgl. 4.3). Mit der Graph-Partitionierung werden auch die Abhängigkeiten der Zeilen untereinander einbezogen, sodass jener Overhead berücksichtigt wird. Die gleiche Überlegung kann bei einer blockweisen oder zyklischen Verteilung nicht einfließen. Insgesamt setzt sich die Qualität einer Verteilung aus der Anzahl der Einträge in einer Partition, der Menge der zu kommunizierenden Daten und der Anzahl der ein- und aus-

[3] Vorausgesetzt für die Kommunikation wird das Message Passing Modell eingesetzt. MPI ist bis jetzt das einzig unterstützte Kommunikationsmodell in LAMA für Systeme mit verteiltem Speicher. Daneben entsteht ein Modell mit dem Global address space Programming Interface (GPI)

gehenden Nachrichten zusammen. Die Qualität einer Partitionierung kann somit anhand dieser Größen gemessen werden. Ein Vergleich bzgl. der Berücksichtigung dieser Ziele zeigen die Tabellen 7.7 bis 7.12 anhand von drei Test-Matrizen. Die Tabellen stellen eine blockweise (b) und zyklische[4] (z) Partitionierung derjenigen gegenüber, die mit der Graph-Partitionierungsbibliothek Metis [Kar] erstellt (m)[5] wurde. Bei allen drei Partitionierungsmethoden ist für die verschiedenen Aufteilungen eine gleichmäßige Gewichtung ($\frac{1}{p}$) verwendet worden, um eine leichtere Bewertung der absoluten Größen zu schaffen.

Betrachtet man zuerst die Anzahl der Einträge pro Partition (Tabelle 7.7 - 7.9) so fällt auf, dass für strukturierte Matrizen wie der Block- und Poisson-Matrix, unabhängig von der Partionierungsmethode und der Anzahl der Partitionen, eine gleichmäßige Partionierung erreicht wird. Für eine weniger strukturierte Matrix wie *ldoor* hingegen trifft dies mit steigender Anzahl von Partitionen nicht mehr zu. Die blockweise Methode hängt vollständig von der Struktur der Matrix ab, sodass das Ergebnis beliebig schlechte Partionierungen sein können. Die zyklische Partionierung gleicht dagegen die Struktur der drei Matrizen ideal aus. Dies ist durch die Struktur der Matrizen und die chunksize bedingt[6]. In dieser Betrachtung ist die chunksize *eins*, was in den meisten Fällen eine gute Balance verspricht[7]. Die gute Balance geht in diesem Fall aber auf die Kosten der Kommunikation, wie man schon an der Anzahl der Einträge im Halo sieht. So fallen für die Matrizen mit in etwa gleich vielen Einträgen pro Zeile und eine kleinen chunksize ganze $\frac{n-1}{n}\%$ der Einträge in den Halo. Die mit Metis erzeugte Partionierung erreicht, ähnlich der zyklischen, aber gänzlich unabhängig von der Matrixstruktur ein gutes Gleichgewicht der Einträge in den Partitionen, während die Größe des Halos wesentlich verringert wird, was der Kommunikation zu Gute kommt. Für die ideale Block-Matrix fällt bei der Metis-Partionierung auf, dass im Algorithmus ein kleines Ungleichgewicht in der Verteilung (im Vergleich zu den beiden anderen Methoden) in Kauf genommen wurde, um einen optimalen Kantenschnitt mit dem Resultat eines leeren Halo-Anteils und keiner Kommunikation zu erreichen.
Dass mit der Metis-Partitionierung weniger kommuniziert werden muss, lässt sich schon an der Aufteilung der Einträge zwischen lokalem und Halo-Anteil erkennen, da weniger potentiell auszutauschende Elemente im Halo auftreten. Das Verhältnis zwischen den einzelnen Partitionierungsmethoden, dass für die Anzahl der Einträge im Halo-Anteil auftritt, bildet sich natürlich nicht eins zu eins auf die Menge an ein- und ausgehender Daten ab, weil nicht für jeden Matrix-Eintrag ein Vektor-Eintrag kommuniziert werden muss (für

[4] Die zyklischen Verteilungen in den Testläufen wurde immer mit einer chunksize von eins durchgeführt. Die chunksize ist dabei die zusammenhängende Blockgröße für eine Partition. Mit einer chunksize von eins wird eine Matrix damit in höchsten Maße über die Partitionen verteilt.

[5] Alle mit Metis erstellten Partitionen wurden durch die rekursive Bisektion mit der Metrik des Kantenschnittes erzeugt. Damit wurde der Anzahl der Einträge im Halo gegenüber der Anzahl der ein- und ausgehenden Nachrichten eine höhere Bedeutung zugetragen, da die Latenzzeit gegenüber der Bandbreite bei den auftretenden Nachrichtengrößen und den gegebenen Systemen vernachlässigbar klein ist. Zudem ist es auch für die Berechnung besser, weniger Einträge im Halo zu haben.

[6] Die zyklische Verteilung gleicht immer dann die Struktur der Matrix aus, wenn diese nicht dasselbe Muster wie die Verteilung aufweist.

[7] Generell sollte die chunksize wesentlich kleiner als die Anzahl der zu verteilenden Zeilen sein und kein Muster der Matrix treffen.

Matrix	P	Einträge in der Partition			Einträge im Halo		
		b	z	m	b	z	m
Block_1500000_45	0	33750000	33750000	33750675	450	16866659	0
	1	33750000	33750000	33749325	900	16866674	0
Poisson_3D27P_-100_100_100	0	13231796	13231796	13231790	88804	8791596	93915
	1	13231796	13231796	13231802	88804	8791596	93915
ldoor	0	23982209	23266643	23261195	2658479	11619075	25688
	1	22540266	23255832	23261280	2658479	11619075	25688

Tabelle 7.7.: Vergleich der Qualität zwischen den Methoden für 2 Partitionen bzgl. der Aufteilung

Matrix	P	Einträge in der Partition			Einträge im Halo		
		b	z	m	b	z	m
Block_1500000_45	0	16875000	16875000	16876350	450	12649992	0
	1	16875000	16875000	16874325	900	12650000	0
	2	16875000	16875000	16875000	450	12650004	0
	3	16875000	16875000	16874325	450	12650004	0
Poisson_3D27P_-100_100_100	0	6571496	6571496	6615898	88804	4351396	93804
	1	6660300	6660300	6615892	177608	4440200	91091
	2	6660300	6660300	6615898	177608	4440200	96149
	3	6571496	6571496	6615904	88804	4351396	91164
ldoor	0	11700622	11632618	11630551	314733	8706215	22356
	1	12281587	11627004	11630644	2973212	8703599	26558
	2	11734786	11634025	11630654	3171165	8706878	22693
	3	10805480	11628828	11630626	2696182	8704066	30153

Tabelle 7.8.: Vergleich der Qualität zwischen den Methoden für 4 Partitionen bzgl. der Aufteilung

Matrix	P	Einträge in der Partition			Einträge im Halo		
		b	z	m	b	z	m
Block_1500000_45	0	8437500	8437500	8438175	450	7374996	0
	1	8437500	8437500	8438175	900	7375000	0
	2	8437500	8437500	8436150	450	7375000	0
	3	8437500	8437500	8438175	450	7375000	0
	4	8437500	8437500	8436150	900	7375000	0
	5	8437500	8437500	8438175	450	7375000	0
	6	8437500	8437500	8436825	450	7375002	0
	7	8437500	8437500	8438175	900	7375002	0
Poisson_3D27P_-100_100_100	0	3241346	3285748	3307943	89102	2913248	66749
	1	3330150	3330150	3307955	177906	2957650	74783
	2	3330150	3330150	3307940	177906	2957650	66856
	3	3330150	3285748	3307952	177906	2913248	69517
	4	3330150	3285748	3307952	177906	2913248	72892
	5	3330150	3330150	3307958	177906	2957650	67736
	6	3330150	3330150	3307946	177906	2957650	67221
	7	3241346	3285748	3307946	89102	2913248	67206
ldoor	0	6096884	5817929	5815284	1104410	5077589	20970
	1	5603801	5812708	5815295	1178353	5073898	27487
	2	6204775	5816709	5815314	2277009	5077007	18569
	3	6076861	5815353	5815302	1865015	5075692	26605
	4	5952996	5814689	5815271	2312384	5075422	17248
	5	5781825	5814296	5815320	2516095	5075077	32046
	6	5602954	5817316	5815355	2078114	5077019	20880
	7	5202379	5813475	5815334	1859648	5074412	24261

Tabelle 7.9.: Vergleich der Qualität zwischen den Methoden für 8 Partitionen bzgl. der Aufteilung

Matrix	P	eing. N. b	eing. N. z	eing. N. m	ausg. N. b	ausg. N. z	ausg. N. m	eing. Daten b	eing. Daten z	eing. Daten m	ausg. Daten b	ausg. Daten z	ausg. Daten m
Block_1500000_45	0	1	1	0	1	1	0	120	6000000	0	480	6000000	0
	1	1	1	0	1	1	0	480	6000000	0	120	6000000	0
Poisson_3D27P_-100_100_100	0	1	1	1	1	1	1	6144	115632	2232	9088	115648	2432
	1	1	1	1	1	1	1	9088	115648	2432	6144	115632	2232
ldoor	0	1	1	1	1	1	1	1813920	3808808	14704	644536	3808816	14752
	1	1	1	1	1	1	1	644536	3808816	14752	1813920	3808808	14704

Tabelle 7.10.: Vergleich der Qualität zwischen den Methoden für 2 Partitionen bzgl. des Kommunikationsaufwands

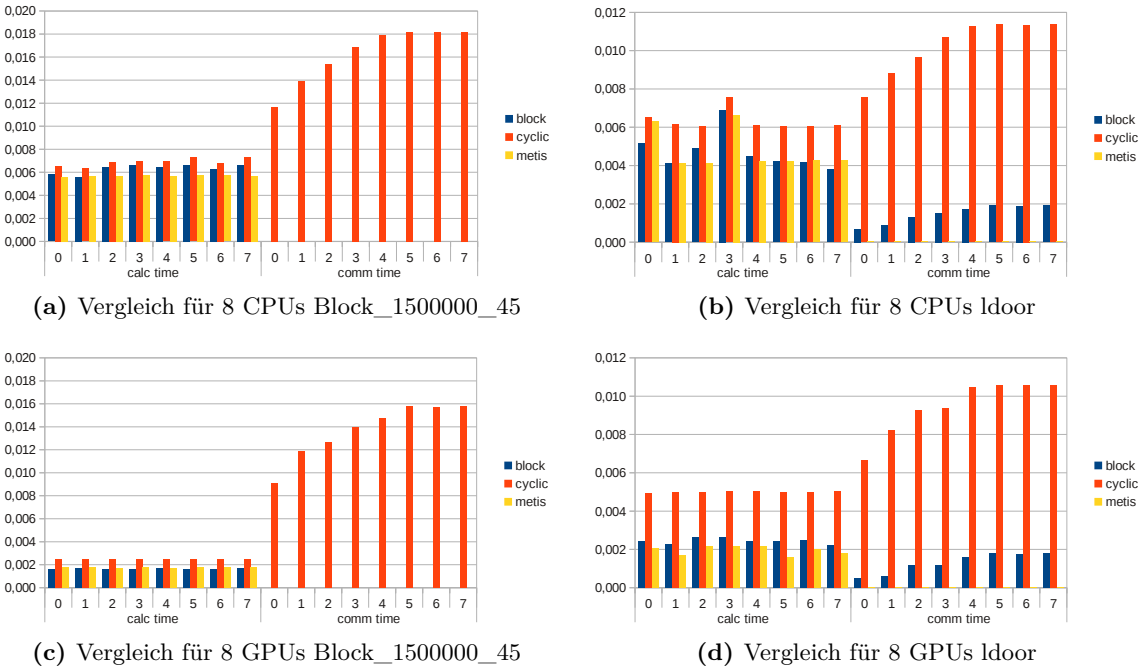
Matrix	P	eing. N. b	eing. N. z	eing. N. m	ausg. N. b	ausg. N. z	ausg. N. m	eing. Daten b	eing. Daten z	eing. Daten m	ausg. Daten b	ausg. Daten z	ausg. Daten m
Block_1500000_45	0	1	3	0	2	3	0	240	9000000	0	360	9000000	0
	1	2	3	0	2	3	0	240	9000000	0	480	9000000	0
	2	1	3	0	1	3	0	240	9000000	0	120	9000000	0
	3	1	3	0	0	3	0	240	9000000	0	0	9000000	0
Poisson_3D27P_-100_100_100	0	1	2	3	1	2	3	80000	3920000	95280	80000	3920000	95432
	1	2	2	3	2	2	3	160000	4000000	92664	160000	4000000	92568
	2	2	2	3	2	2	3	160000	4000000	99672	160000	4000000	99928
	3	1	2	3	1	2	3	80000	3920000	95296	80000	3920000	94984
ldoor	0	1	3	2	1	3	2	170952	5713216	12688	150152	5713224	13072
	1	3	3	3	3	3	3	1964072	5713216	15456	932728	5713224	15120
	2	2	3	2	2	3	2	1369464	5713216	12944	1510696	5713224	13152
	3	2	3	3	2	3	3	806616	5713224	17488	1717528	5713200	17232

Tabelle 7.11.: Vergleich der Qualität zwischen den Methoden für 4 Partitionen bzgl. des Kommunikationsaufwands

Matrix	P	eing. N. b	eing. N. z	eing. N. m	ausg. N. b	ausg. N. z	ausg. N. m	eing. Daten b	eing. Daten z	eing. Daten m	ausg. Daten b	ausg. Daten z	ausg. Daten m
Block_1500000_45	0	1	7	0	2	7	0	120	10500000	0	480	10500000	0
	1	2	7	0	2	7	0	480	10500000	0	240	10500000	0
	2	1	7	0	1	7	0	120	10500000	0	240	10500000	0
	3	1	7	0	1	7	0	120	10500000	0	240	10500000	0
	4	2	7	0	2	7	0	480	10500000	0	240	10500000	0
	5	1	7	0	1	7	0	120	10500000	0	240	10500000	0
	6	1	7	0	1	7	0	120	10500000	0	240	10500000	0
	7	2	7	0	1	7	0	480	10500000	0	120	10500000	0
Poisson_3D27P_-100_100_100	0	1	5	4	1	5	4	80800	4920000	70152	80800	4920000	69464
	1	2	5	7	2	5	7	160800	5000000	76360	160800	5000000	77648
	2	2	5	4	2	5	4	160800	5000000	65968	160800	5000000	65280
	3	2	5	6	2	5	6	160800	4920000	70864	160800	4920000	71168
	4	2	5	7	2	5	7	160800	4920000	75024	160800	4920000	76024
	5	2	5	5	2	5	5	160800	5000000	69360	160800	5000000	69192
	6	2	5	4	2	5	4	160800	5000000	69800	160800	5000000	69112
	7	1	5	5	1	5	5	80800	4920000	69136	80800	4920000	68776
ldoor	0	2	7	2	2	7	2	699792	6665416	11632	331336	6665448	12448
	1	2	7	4	2	7	4	359032	6665416	15840	665432	6665448	15912
	2	7	7	2	7	7	2	1803664	6665416	11264	797648	6665448	11304
	3	5	7	3	5	7	3	976096	6665424	16144	880856	6665392	15776
	4	5	7	2	5	7	2	1161568	6665424	9688	1197080	6665392	10192
	5	5	7	6	5	7	6	1221712	6665424	18816	1357872	6665392	18200
	6	5	7	4	5	7	4	919984	6665368	12104	1292936	6665400	12088
	7	5	7	3	5	7	3	694512	6665424	14216	1313200	6665392	13784

Tabelle 7.12.: Vergleich der Qualität zwischen den Methoden für 8 Partitionen bzgl. des Kommunikationsaufwands

das tatsächliche Kommunikationsaufkommen s. 4.3.2), aber die zu kommunizierende Datenmenge ist bei der Metis-Partitionierung immernoch mit Abstand die geringste. Zudem fällt auf, dass für die Metis-Partionierung ein besseres Gleichgewicht im Kommunikationsaufwand zwischen den Partionen gegeben ist. Dies hat den zusätzlichen Vorteil, dass die Prozessoren gleichermaßen mit Kommunikation beschäftigt sind.
Bei der Anzahl der ein- und ausgehenden Nachrichten profitiert die blockweise Partionierung i. d. R. von der Nachbarschaftsstruktur der Einträge und erreicht für die Block- und Poisson-Matrix die kleinste Anzahl Nachrichten. Für *ldoor* fällt erst für eine höhere Anzahl Partitionen auf, dass keine gute Nachbarschaftsstruktur für eine blockweise Partitionierung vorliegt. Die zyklische Verteilung erreicht dadurch, dass sie benachbarte Einträge aufteilt, für *Block_1500000_45* und *ldoor* immer die maximale Nachrichtenanzahl. Nur für *Poisson_3D27P_100_100_100* ist die Anzahl geringer, da diese Matrix ebenfalls ein zyklisch wiederkehrendes Muster aufweist innerhalb dessen weniger Kommunikation nötig ist. Dies ist aber die Ausnahme. Die Metis-Partitionierung erzielt für die Block-Matrix einen optimalen Schnitt, wodurch keine Kommunikation auftritt, für die anderen beiden Matrizen fällt die Nachrichtenanzahl beliebig aus. Mit der Metrik des Kantenschnittes wird die Anzahl nur indirekt minimiert, unterschreitet aber weder eine Grenze noch liegt ein Muster für die Anzahl der Nachrichten wie beispielsweise bei der blockweisen Partionierung vor.
Insgesamt erfüllt die Metis-Partionierung die Qualitätsansprüche an eine Partitionierungsmethode im Vergleich zur blockweisen und zyklischen am besten, da sie die Anzahl der Nachrichten grundsätzlich reduziert, das Kommunikationsvolumen aber sichtlich stärker minimiert und zugleich auch balanciert. Gerade für die unstrukturierte Matrix *ldoor* erzielt sie deutliche Verbesserungen zu den beiden anderen Methoden.
Dass sich die Qualität der Partitionierung auch in der Ausführzeit wiederspiegelt wird in Abbildung 7.5 gezeigt. Diese stellt für Block_1500000_45 und *ldoor* den Vergleich zwischen blockweiser, zyklischer und Metis-Partitionierung für ein homogenes System aus acht CPUs (6 Threads) bzw. GPUs des Bull-Clusters dar. Entsprechende Abbildungen für die Aufteilung auf zwei bzw. vier Partitionen befinden sich in A.3.1.
Die *Block_1500000_45* zeigt für jede der Partionierungsmethoden ein Gleichgewicht in der Berechnungszeit zwischen den Prozessoren. Im Vergleich zwischen der blockweisen und zyklischen Verteilung weist die zyklische aber ein höheres Laufzeitverhalten auf, weil nur jede 8.Zeile zur selben Partion gehört und damit das Caching innerhalb einer Partionen nicht so optimal ist, wie im Fall, dass alle Zielen eines Blockes in der Partition liegen. Die Metis-Partitionierung hat geringfügig andere Schnitte berechnet als die blockweise Partionierung, um komplette Diagonal-Blöcke in eine Partionen zu legen, sodass keine Kommunikation nötig ist, erreicht damit aber ebenso eine balancierte Ausführung wie die blockweise, die qualitativ mit dieser zu vergleichen ist. Bei der Kommunikation wird direkt der Nachteil der zyklischen Partionierungsmethode ersichtlich. Für die CPU übersteigt die Kommunikationszeit nach der Aufteilung auf vier Prozessoren die Berechnungszeit, für die GPU schon bereits nach der Aufteilung auf zwei Prozessoren. Dabei handelt es sich zudem



(a) Vergleich für 8 CPUs Block_1500000_45

(b) Vergleich für 8 CPUs ldoor

(c) Vergleich für 8 GPUs Block_1500000_45

(d) Vergleich für 8 GPUs ldoor

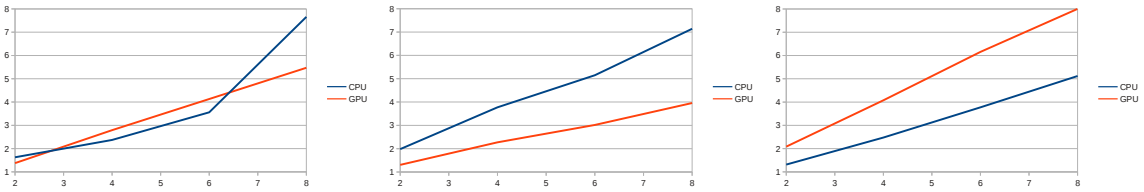
Abbildung 7.5.: Zeitlicher Vergleich der Qualität zwischen den Methoden für 8 Partitionen eines homogenen Systems aus CPUs (a) bzw. GPUs (b) für die Matrizen *Block_1500000_45* und *ldoor*

um die Gesamtberechnungszeit. Um die Kommunikation bei asynchroner Ausführung zu verstecken muss sie kleiner der Berechnung des lokalen Anteils sein. Dieser ist wiederum aufgrund des großen Halo-Anteils kleiner als bei den anderen Partitionierungsmethoden, sodass dies für die CPU schon bei vier Partitionen nicht mehr, für die GPU nicht mal bei zwei Partitionen, möglich ist.
Für *ldoor* sieht das Bild mit dem Unterschied, dass für die CPU 2 Partitionen die Balancierung der Berechnungszeit stören, ähnlich aus. Zum Teil bildet sich die ungleichmäßige Größe der Partitionen auf die Berechnungszeit (für die CPU sichtbar) ab. Auf der GPU wirken sich die Unterschiede nicht so stark aus. Die zwei hervorstechenden Partionen sind dabei aber nicht die absolut größten Partitionen, was darauf schließen lässt, dass in diesen ein äußerst schlechtes Zugriffsmuster vorliegt. Dies führt sich bei den Aufteilungen auf zwei und vier Partionen (s. Anhang) fort, wo ebenfalls eine Partition die Balance auch für die Metis-Partionierung mit guter Balance stört. Zudem kommt bei *ldoor* für alle Partionierungsmethoden Kommunikation dazu. Bei der Metis-Partionierung fällt diese so gering aus, dass sie problemlos hinter der lokalen Berechnung versteckt werden kann, für die blockweise-Partitionierung funktioniert diese für die GPU-Konfiguration nur bis sechs Prozessoren, für die zyklische nur für zwei.

Skalierungsverhalten der Graph-Partitionierung für homogene Systeme

Wie eben gezeigt kann nur die Partionierung mit Metis für beliebige Matrizen dem Anspruch gerecht werden die Kommunikation zu minimieren, weshalb nur sie für die verteilte

SpMV in Frage kommt. Mit den beiden anderen Methoden ist i. d. R. keine Verbesserung in der Gesamtlaufzeit zu erzielen, da sich der mögliche Laufzeitgewinn durch die Kommunikation erübrigt. An der steigenden Anzahl von Einträgen im Halo sowie der zu kommunizierenden Elemente für die Partitionierung von zwei auf vier und acht Prozessoren sieht man aber bereits, dass der Overhead für die Kommunikation auch mit der Metis-Partitionierung steigt. Zudem kommt der schwer berechenbare Faktor der Struktur der (verteilten) Matrix hinzu, der das Skalierungsverhalten der Ausführungzeit beschränkt. Wie sich dies auf die drei Beispiel-Matrizen auswirkt, ist in Abbildung 7.6 dargestellt.



(a) Skalierungsverhalten für Block_1000000_50

(b) Skalierungsverhalten für Poisson_3D27P_100_100_100

(c) Skalierungsverhalten für ldoor

Abbildung 7.6.: Skalierungsverhalten für homogene Systeme aus CPUs bzw. GPUs

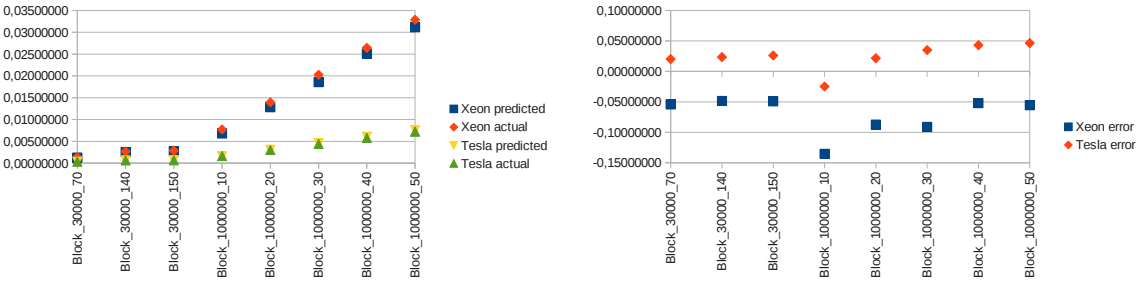
7.4.2. Gewichtungs-Modell

Mit Hilfe des Modells aus Sektion 5.1 soll die Leistung hybrider Systemkomponenten vergleichbar bewertet werden. Ein Vergleich anhand der maximalen Rechenleistung ist für verschiedene Prozessortypen wenig tragbar, da die SpMV memory bound ist. Ein alleiniger Vergleich über die Speicherbandbreite vertritt zwar den memory bound Charakter der Berechnung, für GPUs muss daneben aber auch berücksichtigt werden ob Zugriffe coalesced erfolgen und welcher zusätzlicher Overhead im Vergleich zur CPU auftritt. Das vorgeschlagene Modell bezieht diese Einflüsse mit ein und versucht damit eine Laufzeitvorhersage zu machen.
In den nächsten Teilen dieses Abschnittes wird evaluiert wie sich die Laufzeitvorhersage des Modells bewährt und wie sich die daraus resultierende Gewichtung insgesamt gegenüber anderen Gewichtungen behauptet. Außerdem wird beurteilt wie effizient die dynamische Korrektur anhand der Ausführzeit für synchrone und asynchrone Kommunikation arbeitet und wie gut das Verfahren mit steigender Prozessorzahl skaliert.

Laufzeitvorhersage

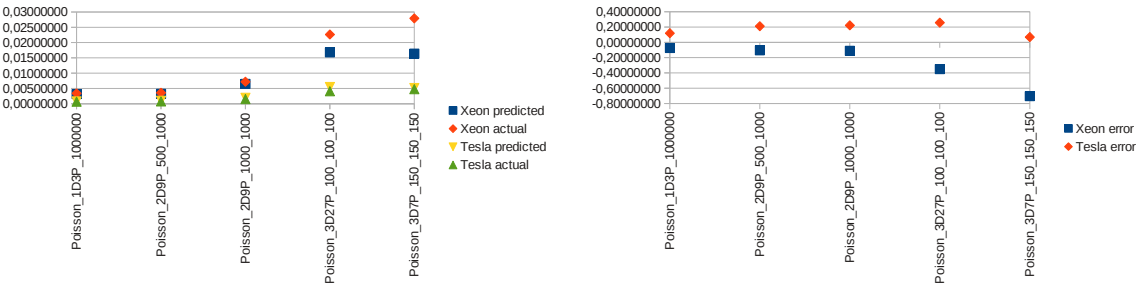
Die statische (initiale) Gewichtung baut gänzlich auf der Laufzeitvorhersage für die Prozessoren auf. Anhand dieser wird für die gegebene Matrix die Gewichtung berechnet. Schlägt die Vorhersage fehl, wird die Gewichtung auch kein gutes Ergebnis liefern können, weil sie von falschen Vorraussetzung ausgeht. In den nächsten Abbildungen (7.7 - 7.9) sind für verschiedene Block-, Poisson- und beliebige Test-Matrizen die vom Modell vorhergesagten (*predicted*) Laufzeiten gegenüber der gemessenen (*actual*) Laufzeiten für die

Berechnung der kompletten Matrix-Vektor-Multiplikation und der dabei gemachte Fehler dargestellt. Bei dem hier gezeigten Vergleich handelt es sich um Vorhersage und Messung für das Bull-Cluster (Intel Xeon und Tesla C1060). Entsprechende Abbildungen für den F1Boensch (Core i7 und GTX 480) sind im Anhang in Abschnitt A.3.2 zusammengestellt. Die hier gezeigten Tendenzen und die folgenden Interpretationen finden sich dort aber ebenso wieder.



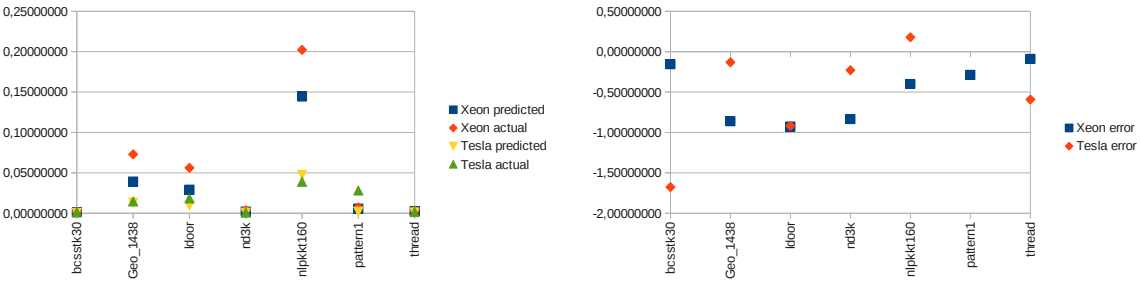
(a) Laufzeitvorhersage **(b)** Fehler

Abbildung 7.7.: Laufzeitvorhersage für Block-Matrizen auf dem Bull



(a) Laufzeitvorhersage **(b)** Fehler

Abbildung 7.8.: Laufzeitvorhersage für Poisson-Matrizen auf dem Bull



(a) Laufzeitvorhersage **(b)** Fehler

Abbildung 7.9.: Laufzeitvorhersage für reale Matrizen auf dem Bull

Bei den Block-Matrizen handelt es sich um die ideal erzeugten Matrizen mit sequentiellem Zugriffsmuster auf den Quell-Vektor x. Ein irregulärer Zugriff auf x, der im Modell nicht berücksichtigt wurde, kann hier also keine Probleme bereiten. Sowohl für den Xeon als

auch für die Tesla-Karte wird die Laufzeit gut vorhergesagt. Für den Xeon liegt der gemachte Fehler bei unter 10% (mit einer Ausnahme), für die Tesla-Karte unter 5%. Die Zeit für den Xeon ist dabei immer höher als vorhergesagt. Dies lässt sich dadurch erklären, dass das Modell lediglich die Daten-Zugriffe einbezieht, aber keinen zusätzlichen Overhead für die CPU, wie z.B. für die Ausführung der Schleife über die Zeilen berücksichtigt. So sieht man, dass der Fehler für die Block-Matrizen mit 1000000 Zeilen und 10 - 50 Einträgen pro Zeile mit wachsender Anzahl Einträge pro Zeile schrumpf, weil eben dieser Overhead weniger ins Gewicht fällt. Für die Tesla-Karte wird mit einer Ausnahme immer eine höhere Laufzeit veranschlagt. Dies begründet sich darin, dass die 2-dimensionale Lokalität der Daten aus dem Quell-Vektor x gegeben ist. Mit steigender Anzahl Elemente pro Zeile (s. Block_1000000_(10-50)) schrumpft dieser Vorteil aber aufgrund der begrenzten Größe des Texture Caches, sodass der Fehler wächst.
Die Poisson-Matrizen weisen eine sehr regelmäßige Struktur auf, sodass auch hier der Texture Cache der GPU Vorteile bringt. Von der 1D- zur 3D-Poisson-Matrix verstärkt sich dieser Vorteil bei Matrizen, deren Zeilenanzahl konstant bleibt, die Anzahl der Elemente und der Offset zwischen den Nebendiagonalen aber steigt, sodass der Fehler durch die worst case Abschätzung im Modell wächst. Für den Xeon hingegen zeigt sich, dass der Fehler mit steigender Anzahl Elemente, die für den CPU-Zugriff eine geringe Lokalität (nur 1-dimensional: zwischen den Zeilen) aufweisen, wächst.
Für die realen Test-Matrizen ist der Fehler wesentlich höher, da die Struktur der Matrizen gegenüber den künstlichen Matrizen beim Zugriff auf x i. d. R. kein gutes Caching für CPU und GPU ermöglicht, was nicht durch das Modell abgebildet wird. Für die Matrix *Geo_1438*, die einen diagonalen Charakter aufweist, ist für die Tesla-Karte eine relativ gute 2-dimensionale Lokalität gegeben, sodass der Fehler mit 13% verhältnismäßig klein ist. Gleiches gilt für *nlpkkt160*, die aus drei Diagonalen besteht. Besonders interessant ist das Verhalten der Matrix *ldoor*, für die der Fehler für beide Prozessoren in etwa gleich ist (92%). Trotz des hohen Fehlers sollte an dieser Stelle die Modell-Gewichtung gut sein, da es für diese nur darum geht, wie die Prozessoren sich im Vergleich zueinander verhalten.

An dieser Stelle wird deutlich, welchen Einfluss die Struktur der Matrix auf die Laufzeit hat. Für diese Art der Matrizen ist hier deshalb eine dynamische Balancierung zum Erreichen einer guten Balance unerlässlich. Für Matrizen mit einer erkennbarer Struktur und wiederkehrenden Mustern, wie sie bei Matrizen wie den Poisson-Matrizen mit Diagonalstruktur vorliegen, kann ein Korrekturfaktor für ungünstiges Caching eingeführt werden. An dieser Stelle werden Überlegungen, wie sie bei der Blockung von Algorithmen für die Cacheoptimierung bereits gemacht wurden, [KGK09] wichtig, um abhängig von der Größe des Cache und der Zusammenhangskomponenten in der Matrix eine gute Abschätzung über Wiederverwendung machen zu können.

Balancierung des Gewichtungs-Modell

Wie in Absatz 4.2.4 gezeigt, kann bei einer genügend kleinen Kommunikationszeit mit asynchron zur Berechnung erfolgter Kommunikation, diese vollständig hinter der Berechnung versteckt werden und die Gesamtlaufzeit damit auf die Berechnungszeit reduziert werden. Durch den Vergleich zwischen den Partitionierungsmethoden (vgl. Absatz 7.4.1) ist bekannt, dass dies durch die Graph-Partionierung hinreichend erreicht werden kann. Somit hängt eine gute Gesamtlaufzeit für die verteilte Berechnung im Wesentlichen von der Balance in der Berechnungszeit zwischen den Prozessoren ab. Diese sollte bei einer adäquaten Gewichtung erreicht werden. Im folgenden Vergleich soll die Balance des vorgeschlagenen Modells anderen Gewichtungsansätzen gegenübergestellt werden. Bei den Ansätzen handelt es sich um:

practical Die praktische Gewichtung stellt einen Vergleichswert für das Modell dar. Sie wird aus den Laufzeiten der Komplettberechnung gewonnen. Da die Laufzeiten die Struktur der Matrix am besten berücksichtigen und damit in die Gewichtung einbringen, sollte dies eine gute Balancierungen erzielen. Allerdings ist dieser Ansatz i. d. R. wegen der Größe der zu verteilenden Matrizen nicht möglich, da sie nicht vollstängig in den Arbeitsspeicher geladen werden können.

theoretical Die theoretische Gewichtung setzt die Gewichte aufgrund der maximalen Rechenleistung der Prozessoren an. Diese Art der Gewichtung ist leicht für den Nutzer apriori anzugeben. Allerdings macht sie nur für compute bound Algorithmen Sinn. Für die Bewertung der SpMV wird hier für die vorliegende Hardware die Leistung für doppelt genaue Berechnungen angenommen. Für GPUs bedeutet dies, dass die theoretische Leistung der vorhandenen DPUs gewertet wird. Eine Optimierung durch Vektorisierung (SSE) für CPUs wird nicht berücksichtigt.

bandwidth Geht man von einem memory bound Algorithmus aus, ist diese Gewichtung angebrachter und wäre für den Nutzer ebenso einfach möglich wie die theoretische. Dabei kann entweder von der theoretischen Maximalleistung oder praktisch bestimmten Benchmark-Ergebnissen ausgegangen werden. Für die Bewertung hier wurde die Gewichtung auf Grundlage von praktischen Werten aus STREAM- und BandwidthTest-Benchmarks (s. Tabelle 7.3 und 7.4) berechnet.

model Die Modell-Gewichtung kalkuliert grobe Gewichtungsvorschläge anhand einer Laufzeitvorhersage für die Komplettberechnung und korrigiert diese iterativ nach, indem neue Vorhersagen für den vorher veranschlagten Anteil der Berechnung gemacht werden. Die Iteration bricht ab, wenn die prognostizierten Laufzeiten für die Prozessoren weniger als 3% voneinander abweichen.

corrected Die korrigierte Modell-Gewichtung arbeitet genauso wie die unkorrigierte, bezieht aber den Fehler, der bei der Vorhersage für die Komplettberechnung gemacht

wurde für die Laufzeitvorhersage mit ein, um den Fehler, der bei der Laufzeitvorhersage gemacht wurde bei dieser Bewertung auszuklammern.

dynamic x Die dynamische Methode bezieht, ausgehend von der (unkorrigierten) Modellgewichtung, den Fehler der Laufzeitvorhersage in die Modellbewertung mit ein und verbessert damit nachfolgende Gewichtungen und sollte somit für unstrukturierte Matrizen schnell zu einer Balancierung führen. Gibt es keine dynamischen Werte, wurde das Gleichgewicht, das bei einem maximalen Inbalancefaktor von 3% angenommen wird, bereits mit der initialen Modell-Gewichtung gefunden. Maximal werden jedoch 20 Iterationen gemacht.

Abbildung 7.10 zeigt für ausgewählte Matrizen den Vergleich der Ansätze für die Konfiguration auf dem Bull mit zwei Prozessoren. Entsprechende Vergleiche für die Konfigurationen auf dem F1Boensch befinden sich in den Abbildungen A.9 und A.10 im Anhang. In den Diagrammen der Abbildungen sind die Berechnungszeiten für die verschiedenen Gewichtungsmodelle sowie als Refenzenbalken die Komplettberechnung auf einem Prozessor abgetragen. Die passende Tabelle hält die zugehörigen Gewichte und die im Detail erzielte Leistung fest.

Für die gewählten Matrizen wird, wie erwartet, nur für *ldoor* mit der Modell-Gewichtung ein ideal balancierter Zustand erreicht. Trotz falscher Laufzeitvorhersage war für beide Prozessoren derselbe Fehler gemacht worden, sodass dies für die Berechnung der Gewichtung keine Bedeutung hatte. Für die anderen Matrizen war der Fehler für die Matrizen nicht nur unterschiedlich, sondern addiert sich, weil der Xeon in seiner Leistung grundsätzlich unterschätzt und die Tesla-Karte überschätzt wurde. Wird der Fehler der Laufzeitvorhersage für die Modell-Gewichtung mit einbezogen, so erzielt man auch für *pattern1* ein ideales Gleichgewicht. Insgesamt addiert sich so für die beiden Matrizen die Leistung der Prozessoren. Der theoretische und bandbreitenbasierte Gewichtungsansatz führt hingegen in beiden Fällen zu keinem balancierten Zustand. Für *ldoor* können sie die Leistung geringfügig steigern, für *pattern1* bleibt die Leistung hinter der höchsten Einzelleistung.
Bei den restlichen Matrizen wird mit den statischen Gewichtungen kein oder nur ein nicht sehr gut balancierter Zustand erzielt. Selbst die praktische Gewichtung, die mit den meisten Vorkenntnissen eine Gewichtung berechnet, balanciert die Berechnungszeit nicht gut. Für die strukturierten Matrizen *Block_30000_70*, *Block_1000000_50* und *Poisson_3D7P_150_150_150* wird die Gesamtleistung nicht einmal verbessert. Nur für *bcsstk30* wird dies erreicht, was aber daran liegt, dass die Leistung beider Prozessoren für diese Matrix nah beeinander liegen und eine vergleichsweise große Verbesserung durch die Aufteilung zu erwarten war, wovon nur ein Teil erreicht wurde. Bei den anderen Matrizen ist die Tesla-Karte um einiges leistungsstärker als der Xeon-Prozessor, sodass die Leistungssteigerung im Verhältnis eher klein zu erwarten war.
Erst mit der dynamischen Balancierung wurden mit Ausnahme von *Block_1000000_50*, bessere Leistungen erzielt. Insgesamt „lernt“ das Modell über den bei der Vorhersage gemachten Fehler und kommt innerhalb weniger Iterationen zu einem guten Ergebnis. Al-

Block_30000_70

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	3,0626
100% 1	-	100,000	12,4260
practical	19,5446	80,4554	7,8417
theoretical	29,1167	70,8833	9,9573
bandwidth	17,1882	82,8118	7,8009
model	20,9816	79,0184	12,4703
corrected	19,8041	80,1959	8,3300
dynamic 4	39,4021	60,5979	14,3052

Block_1000000_50

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	3,04388
100% 1	-	100,000	13,8596
practical	17,9963	82,0037	12,1501
theoretical	29,1167	70,8833	10,5476
bandwidth	17,1882	82,8118	13,1296
model	19,5666	80,4334	12,3481
corrected	18,0075	81,9925	12,9561
dynamic 1	23,3664	76,6336	12,9860

Poisson_3D7P_150_150_150

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	1,68245
100% 1	-	100,000	9,85525
practical	14,5676	85,4324	8,9387
theoretical	29,1167	70,8833	5,1528
bandwidth	17,1882	82,8118	9,2183
model	23,9296	76,0704	7,2417
corrected	14,6422	85,3578	9,9185
dynamic 11	17,5608	82,4392	10,4141
dynamic 18	16,6949	83,3051	9,0447

bcsstk30

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	2,7970
100% 1	-	100,000	3,4513
practical	44,8394	55,1606	4,2922
theoretical	29,1167	70,8833	4,0960
bandwidth	17,1882	82,8118	3,2717
model	27,5268	72,4732	3,9943
corrected	45,2526	54,7474	4,7292
dynamic 5	58,2212	41,7788	5,7081

ldoor

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	1,6554
100% 1	-	100,000	5,0675
practical	24,6184	75,3816	6,7592
theoretical	29,1167	70,8833	5,7620
bandwidth	17,1882	82,8118	5,2913
model	24,7908	75,2092	6,7446
corrected	24,6352	75,3648	6,9613
dynamic	-	-	-

pattern1

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	2,5645
100% 1	-	100,000	0,6634
practical	79,4782	20,5218	3,1815
theoretical	29,1167	70,8833	1,1608
bandwidth	17,1882	82,8118	0,8846
model	25,6866	74,3134	1,3870
corrected	79,4199	20,5801	3,1999
dynamic 4	70,3949	29,6051	3,1057
dynamic 20	73,7577	26,2423	1,6976

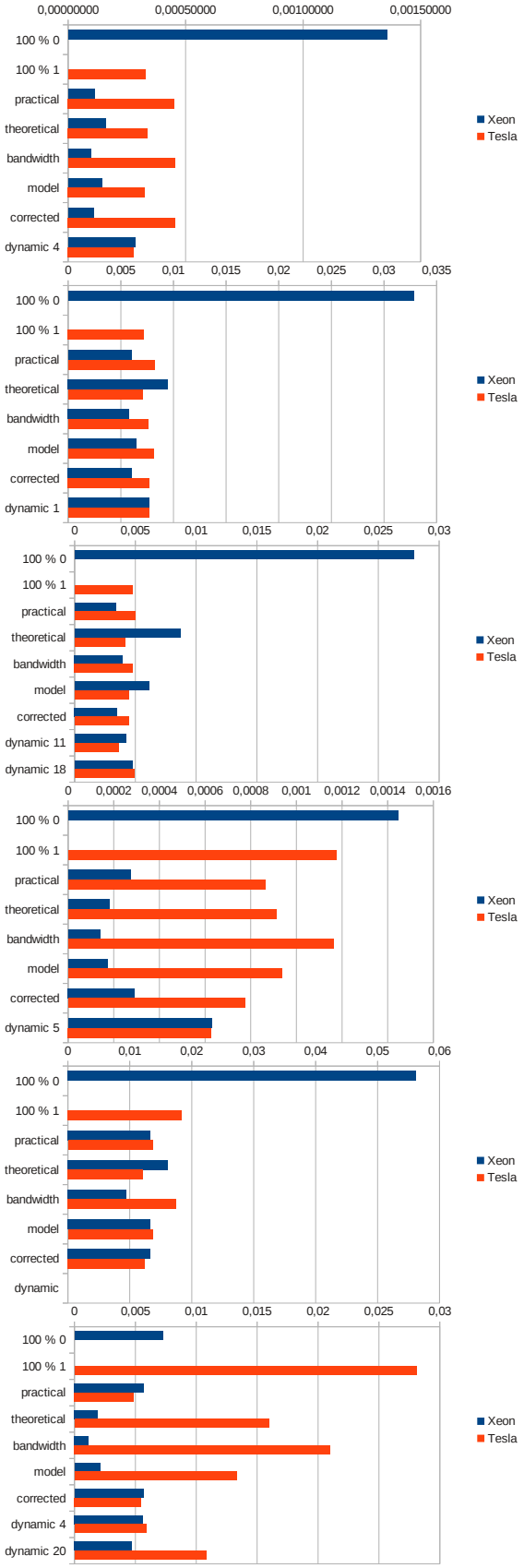
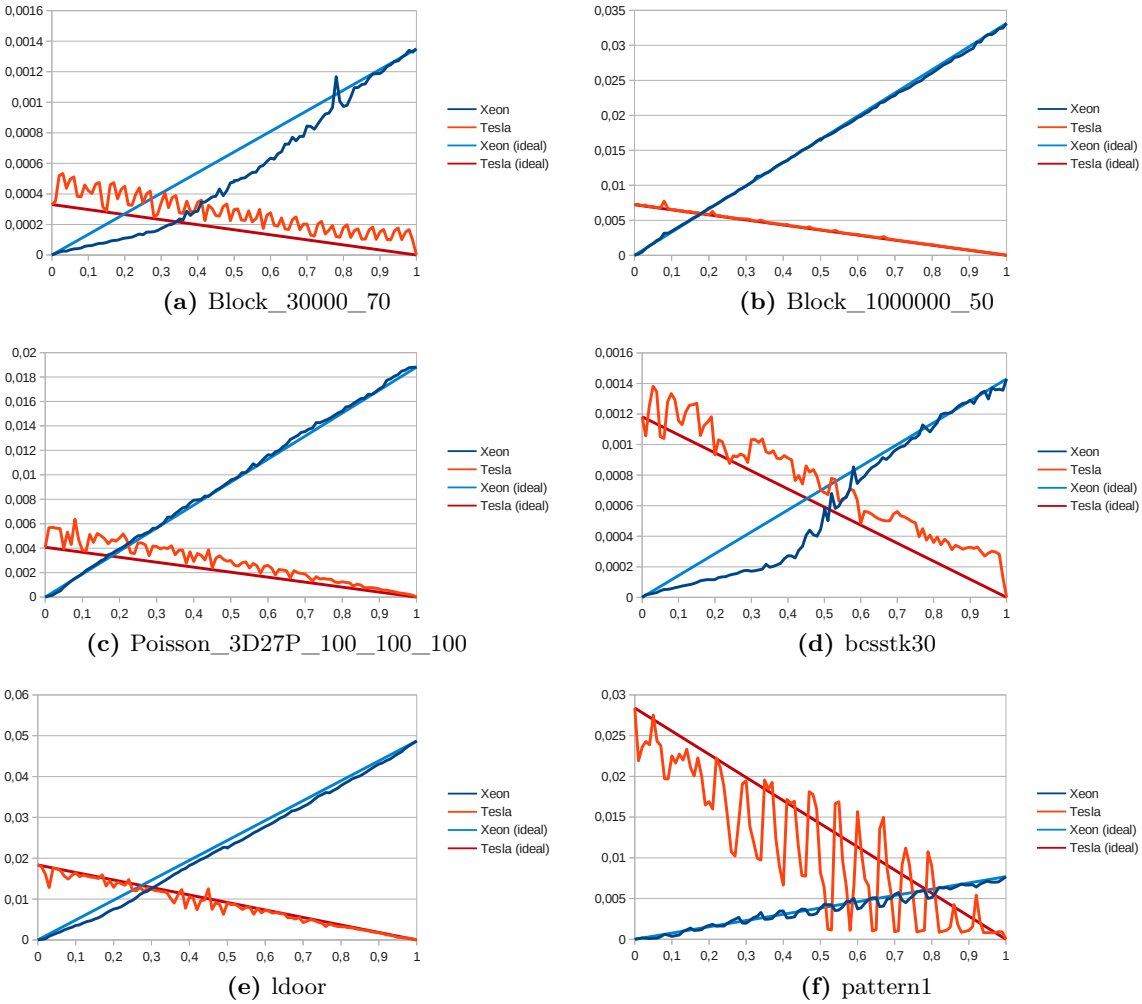


Abbildung 7.10.: Balancierungsbetrachtung für die Gewichtungsansätze: Balance zwischen einem Xeon-Prozessor und einer Tesla-Karte auf dem Bull

lerdings stoppte die Methode nicht immer im Optimum, weil keine Balance innerhalb der 3% Toleranz vorlag. Dies und dass für *Block_1000000_50* keine Verbesserung bei einem balancierten Zustand eingetreten ist, zeigt, dass die Strategie bei der Lastverteilung fehlerhaft ist.

Zusammengefasst kann man sagen, dass sich bei korrekter Laufzeitvorhersage (oder gleichem Fehler für alle Prozessoren) mit der Modell-Gewichtung ein balancierter Zustand erreichen lässt und diese dabei besser als eine Gewichtung über die theoretische Maximalleistung oder Speicherbandbreite ist. Dies liegt daran, dass das Modell bei der Laufzeitvorhersage die Matrix zumindest in Form der Anzahl der Elemente pro Zeile bzw. Warp für die Leistung berücksichtigt und nicht allein auf Basis der Hardware eine Gewichtung angibt. Allerdings bezieht es noch keine weiteren Informationen zur Matrix ein. Mit den Vorschlägen aus 7.4.2 könnte die Laufzeitvorhersage präzisiert werden, was auch die Balance für die Modellgewichtung für alle Matrizen ohne Korrektur verbessert. Lernt das Modell jedoch in der dynamischen Gewichtung über seine Unzulänglichkeit, die unregelmäßige Struktur der Matrix zu berücksichtigen, so gelangt es i. d. R. in einen balancierten Zustand. Trotz allem resultiert dieser nicht immer in einem optimalen Gesamtergebnis, was auf einen grundsätzlichen Fehler in der Strategie hindeutet.

Wie man an den variierenden Leistungen der beiden Prozessoren für die verschiedenen Matrizen sieht, legt die Struktur der Matrix und nicht die Hardware die Leistung des Systems fest. Den Extremfall stellt die Matrix *pattern1* dar, für die die prinzipiell leistungsstärkere Tesla-Karte wesentlich länger rechnet als der Xeon-Prozessor. Vergleicht man *Block_1000000_50* und *ldoor*, die sich im Wesentlichen nur in der Verteilung der Elemente unterscheiden, sieht man, dass die Leistung für *ldoor* weit hinter der für *Block_1000000_50* bleibt. Damit nicht genug, der Leistungsabfall von *Block_1000000_50* nach *ldoor* fällt für CPU und GPU in unterschiedlichem Maße aus. Die Struktur hat damit einen wesentlichen Einfluss auf die tatsächliche Last die der Partition zukommt. Um zu verdeutlichen, dass die Gewichtung mit der derzeitigen Lastbewertung anhand der Anzahl der Matrix-Einträge keine eindeutige Aussage über die tatsächliche Rechenlast macht und damit sowohl für das Modell als auch für eine anschließende dynamische Balancierung Probleme bereitet ist in Abbildung 7.11 gezeigt, wie sich für die Test-Matrizen die Berechnungszeit bei verschiedenen Gewichten (in 1%-Schritten durchlaufen) mit der Metis-Partitionierung verhält. Die Diagramme zeigen, dass sich der reale Verlauf für die verschiedenen Gewichtungen in fast allen Fällen weitgehend anders verhält als der ideal zu erwartende Verlauf. Aufgrund der ersten Messungen (s. Abbildung 5.4) konnte man für einen sequentiellen und coalesced Zugriff von einem streng linearen Verlauf bei einer Variation der Einträge ausgehen. Der Fehler, der durch das nicht Beachten des nicht sequentiellen Zugriffes auf den Vektor x gemacht wird, wirkt sich aber so stark auf die tatsächliche Last aus, dass jegliche Änderung in der Verteilung die Laufzeit ändert. Neben den verschiedenen Gewichtungen ist der Suchraum über die Verteilungen noch wesentlich größer. So besteht zu jeder Gewichtung eine Menge an Zeilenkombinationen, die die gewünschte Gewichtung ergeben und für jede



(a) Block_30000_70

(b) Block_1000000_50

(c) Poisson_3D27P_100_100_100

(d) bcsstk30

(e) ldoor

(f) pattern1

Abbildung 7.11.: reales und ideal angenommenes Laufzeitverhalten für alle Gewichte zwischen einem Xeon-Prozessor und einer Tesla-Karte mit der Metis-Verteilung auf dem Bull (Ordinate = prozentualer Anteil des Xeon-Prozessors)

Kombination nochmal beliebige Permutationen der Zeilen für die lokale Matrix. Alle resultierenden Verteilungen weisen ein mitunter anderes Zugriffsmuster auf. Dementsprechend gibt es auch noch andere Laufzeiten für eine Gewichtung, neben den im Diagramm vorliegenden. Mit der Metis-Partitionierung wurde lediglich eine Verteilung mit minimierten Kommunikationsvolumen ausgewählt.

Für die Matrizen *Block_1000000_50* und *ldoor* ist für CPU und GPU ein gutes lineares Verhalten ohne große Schwankungen gegeben. Der Schnittpunkt bei diesen Matrizen ist damit für reale und ideal Werte in derselben Größenordnung. Kombiniert mit einer guten Laufzeitvorhersage für diese Matrizen werden deshalb gute Ergebnisse erzielt. Bei *Block_1000000_50* wurde hier nur deshalb kein Optimum erzielt, weil der Gewinn nur sehr gering sein konnte und noch Schwankungen aufgrund der Auswahl der Verteilung dazu kommen. Bei den anderen Matrizen sieht man, dass sich die optimale Gewichtung nicht eindeutig im Schnittpunkt der beiden Leistungskurven befindet. Da die Kurven zum

Teil große Schwankungen aufweisen, kann ein Minimum beider Verläufe auch (kurz) davor oder dahinter auftreten. Betrachtet man weiter die idealen Verläufe und den oft entfernten Schnittpunkt dieser vom realen Schnittpunkt wird deutlich, woran jede Gewichtung scheitert.

Die Balancierung anhand der vom Modell vorgeschlagenen Gewichtung liegt abhängig vom Fehler der Laufzeitvorhersage im Rahmen der Gewichte, die mit einer praktischen Bestimmung einher gehen. Mit einer verbesserten Laufzeitvorhersage könnte die Struktur der Matrix bei der Gewichtung besser berücksichtigt werden, sodass der dabei gemachte Fehler kleiner wird. Das weitaus größere Problem ist jedoch, dass die Leistung bei gleichbleibender Gewichtung aber geänderter Struktur zum Teil große Schwankungen im Laufzeitverhalten bewirkt. Die macht sowohl die Vorhersage einer optimalen Gewichtung schwierig als auch eine dynamische Korrektur der Gewichte, da keine eindeutige Richtung festgelegt ist, in welcher das Optimum liegt und keine Bedingung gegeben ist, ob man es erreicht hat. Ein optimaler Zustand kann demnach auch ohne Balance der Ausführungzeiten vorliegen.
Grund dafür ist die bis jetzt angenommene Definition der Last einer Zeile über die Anzahl der Einträge. Verschiedene Zeilen mit gleicher Anzahl Einträge verhalten sich je nach Verteilung der Einträge über die Zeile aufgrund von Cache-Effekten grundsätzlich verschieden, sodass dies in die Bewertung der Last eingehen müsste. Für die Balancierung von CPUs untereinander gleicht sich dieser Fehler über die Partitionen wieder aus, da er für alle Prozessoren unter denselben Bedingungen auftritt und sich gleichermaßen auswirkt. GPUs hingegen haben andere Anforderungen an die Lokalität der Daten im Texture Cache, sodass dies hier nicht mehr zutrifft.
Das Ziel für eine verbesserte Strategie wäre es, die Last entsprechend für CPUs und GPUs unterschiedlich zu definieren. Die Last der Berechnung wird allerdings in sämtlichen bestehenden Modellen über die Anzahl der Einträge definiert und nur daran bei der Graph-Partitionierung bemessen. Stattdessen sollte sie noch mit der Rechenleistung des Prozessors gewichtet und damit ein zusätzliches Mapping von einer Partition zu einem Prozessor vorgenommen werden. Einen vergleichbaren Ansatz haben Walshaw und Cross für die Bewertung der Kommunikationslast für heterogene Netzwerke verfolgt [WC01] (vgl. auch 3.2.1). Dabei werden die Kosten der Kommunikation über die Anzahl der Elemente multipliziert mit der Leistung des Netzwerkes an der Stelle des Schnittes ($|(p,q)|$) bewertet, sodass sich der Kantenschnitt nicht wie in Formel 2.2 berechnet, sondern über

$$\sum_{\substack{e_{i,j} \in \mathcal{E} \text{ mit } v_i \in \mathcal{V}_s,\ v_j \in \mathcal{V}_t \\ \text{und } s \neq t,\ p,q \in \mathcal{P}}} w_e(e_{i,j}) * |(p,q)| \text{ minimal } \forall \text{ Partitionen } \mathcal{V} \tag{7.1}$$

Nötig war dafür ein Kommunikationsgraph des Zielsystems, um für das Mapping von den Prozessoren zu den Partitionen die Leistung des Netzwerkes anzugeben. Für die Balancie-

rung der Last sollte vergleichbar die Funktion 2.1 mit

$$\mathcal{W}_{\mathcal{V}} = \{\pi(p, w_v(v_i)) \cdot \mu(p) | v_i \in \mathcal{V} \;\; p \in \mathcal{P}\} \tag{7.2}$$

und $\pi(p, w_v(v_i))$ der Laufzeitvorhersage für eine Zeile i balanciert werden, um die Knoten im Grahen nach einem prozessorspezifischen Gewicht zu verteilen.

Weiter sollte überlegt werden, ob die Aufteilung nicht blockweise vorgenommen werden kann, um das zweidimensionale Cache-Verhalten des Texture Cache besser auszunutzen und durch die Blockung auch auf der CPU ein vergleichbares Caching zu haben. Die Unregelmäßigkeiten im Zugriff könnten dadurch ausgeglichen werden, dass auf CPU und GPU derselbe Fehler bzgl. des Cache-Zugriffes gemacht wird. Zudem kann man durch die Blockung eine allgemeine Verbesserung der Leistung erwarten. Das Blocked-CSR-Format gehört schon lange für die Berechnung auf der CPU zu den gängigen Formaten eben weil es eine performanter ist. Yang et. al. zeigt in [YPS11], dass eine Blockung der SpMV mit gpuspezifischen Blockgrößen auch eine deutliche Leistungssteigerung für Single-GPU- und Multi-GPU-Systeme bringt. Einziger Nachteil der blockweisen Partitionierung ist, dass zwei Kommunikationsphasen nötig sind, die wesentlich schwieriger mit asynchroner Kommunikation hinter Berechnung zu verstecken sind, da die am Stück durchgeführte Berechnung des lokalen Matrix-Anteils schneller berechnet wird und die zweite Phase erst nach der Berechnung aller Blöcke erfolgen kann.

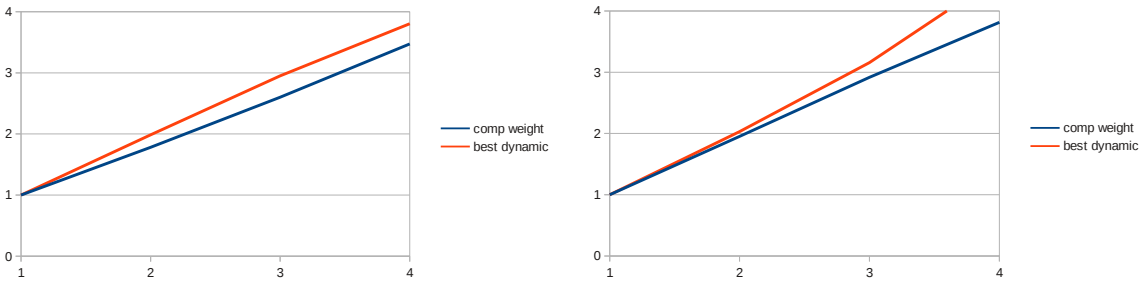
dynamische Skalierung des Gewichtungs-Modells

Nach dem derzeitigen Verhalten wie in Abbildung 7.11 kann die dynamische Balancierung mit einer Veränderung der Gewichte nicht in einem verbesserten und fast idealen Zustand enden, weil die Leistung unregelmäßig steigt und fällt. I. A. zeigt sich, dass die erste Iteration durch die Korrektur der Laufzeitvorhersage eine Verbesserung erzielt, die anschließenden Verfeinerungsschritte dann aber zwischen verschiedenen Zuständen hin- und herspringen, je nachdem wie sehr die reale Laufzeit von der (korrigiert) idealen abweicht. Für *Block_1000000_50* liegen diese dicht beisammen, deswegen wurde bereits in der ersten Iteration nach der Modellgewichtung ein balancierter Zustand erreicht. Bei *pattern1* hingegen kann der Fehler von Iteration zu Iteration mal in die eine Richtung und mal in die andere abweichen, sodass sich die Gewichtung immer stark ändert anstatt auf das Minimum zu zulaufen. Zudem kann die Suche nicht im Optimum stoppen, da dieses nicht in einem balancierten Zustand vorliegt.

Skalierungsverhalten des Gewichtungsmodells

Für eine abschließende Bewertung des Gewichtungsmodells gilt es dieses für mehr als zwei Prozessoren zu betrachten, um es auf seine Skalierbarkeit zu überprüfen. Dazu findet man in Abbildung 7.12 für die zwei Matrizen *Block_1000000_50* und *ldoor*, die sowohl in der Laufzeitvorhersage als auch beim Verlauf über die verschiedenen Gewichtungen

gut abschnitten, das Skalierungsverhalten von zwei bis acht Prozessoren des Bull-Clusters (aufgetragen sind eins bis vier Kombinationen aus einem Xeon-Prozessor und einer Tesla-Karte). Darauf folgt eine Analyse des Balancierungsverhalten über acht Prozessoren.



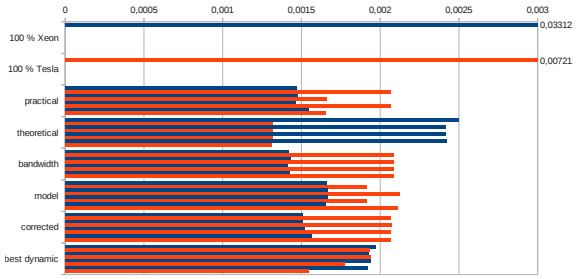
(a) Skalierungsverhalten für Block_1000000_50 **(b)** Skalierungsverhalten für ldoor

Abbildung 7.12.: Skalierungsverhalten für homogene Systeme aus CPUs bzw. GPUs

Für beide Matrizen wird mit der dynamischen Balancierung über die vier Konfigurationen lineare Skalierung erreicht, die einfache Modellgewichtung liegt dabei nur knapp darunter. Bei *ldoor* wird mit der dynamischen Balancierung sogar superlinearer Speedup erzielt, da die größere Blockung das Caching innerhalb der Partitionen verbessert. Abhängig von der Anzahl der durchzuführenden Iterationen des späteren iterativen Lösers dürfte sich aber die dynamische Korrektur, die innerhalb der angesetzten 20 Iterationen ein ideales Ergebnis erreicht, auszahlen. Dazu fehlen an dieser Stelle noch genauere Untersuchungen des Overheads durch die Graph-Partitionierung und für das Umverteilen der Daten in jeder Iteration. Mit dem erwähnten Verschieben von Zeilen anstatt einer Neuberechnung der Verteilung (vgl. Abschnitt 5.5) können aber beide Zeiten sicherlich noch einmal reduziert werden, wobei dies der Qualität der Verteilung nicht schaden sollte.
Die Abbildung 7.13 zeigt, dass eine Bewertung der Prozessoren auf Basis der theoretischen Maximalleistung zu keiner guten Balance führt, sondern dem Xeon mehr Leistung zuspricht als er in real aufgrund der memory bound Charakteristik der SpMV umsetzen kann. Bei der Bewertung aufgrund der Speicherbandbreite der Prozessoren tritt der umgekehrte Fall ein: die Leistung der Tesla-Karte wird überschätzt, weil zusätzliche Beanspruchung durch den Kernel Launch nicht berücksichtigt wird. Zudem wird die Speicherbandbreite absolut bewertet und nicht im Verhältnis zu restlichen Hardware. Das Einbeziehen von hardwarespezifischen Kenngrößen (coalesced access size) und des Kernel Launches bringt noch keine ideale aber ein wesentliche bessere Balance für die Modellgewichtung gegenüber der Bandbreitengewichtung. Dieser Faktor kann dann bei der dynamischen Balancierung ausgeglichen werden.

Block_1000000_50

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	3,04412
100% 1	-	100,000	13,8500
practical	4,5276	20,5995	47,7920
theoretical	7,2792	17,7208	38,3112
bandwidth	4,2971	20,7029	47,4834
model	4,9320	20,0508	46,7946
corrected	4,5695	20,5438	47,7555
best dynamic	5,2024	18,2535	51,4933



ldoor

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	1,78519
100% 1	-	100,000	5,83339
practical	5,8666	19,1702	28,3294
theoretical	7,27917	17,7208	22,8623
bandwidth	4,2971	20,7029	26,2839
model	6,2852	18,7176	26,0893
corrected	5,9224	19,1162	27,4988
best dynamic	5,1862	19,8855	31,2693

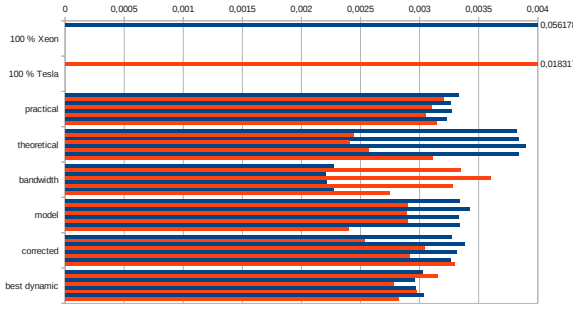


Abbildung 7.13.: Balancierungsbetrachtung für die Gewichtungsansätze: Balance zwischen vier Xeon-Prozessoren und vier Tesla-Karten auf dem Bull

7.5. Gesamtvergleich

Tabelle 7.13 und Abbildung 7.14 zeigen abschließend einen gesamtheitlichen Vergleich der verschiedenen Partitionierungsmethoden und die Referenzen der Komplettberechnung für die Aufteilung auf acht Prozessoren. Zu den Methoden gehören die gleichmäßigen Block- und Cyclic-Verteilungen, eine gleichmäßige Verteilung mit Metis-Partitionierung (even) sowie die verschiedenen Gewichtungsmodelle, die ebenfalls mit der Metis-Partitionierung umgesetzt wurden. Gleiche Gewichtungen mit der allgemeinen Block-Verteilungen waren i. d. R. schlechter, da sie die Zusammenhangskomponenten der Matrix aufgeteilt haben, wodurch eine schlechtere Lokalität der Daten und damit ein schlechterer Zugriff vorlag. Deswegen wurde auf diesen Vergleich verzichtet.

Von allen Gewichtungen schneiden die gleichmäßigen Verteilungen für die großen Matrizen immer am schlechtesten ab. Für kleinere und dabei auch dichter besetzte Matrizen haben die gewichteten Verteilungen keine großen Vorteile, da zu wenig Parallelität vorliegt. Dagegen weisen sie für große Matrizen einen deutlichen Leistungsgewinn gegenüber den gleichmäßigen auf. Von den drei gleichmäßigen Verteilungen schneidet in jedem Fall die Metis-Partitionierung am besten ab, weil damit Zusammenhangskomponenten der Matrix beachtet und möglichst in eine Partition gelegt werden, wodurch mehr Lokalität für die Daten geschaffen wird. Somit macht der Aufwand der Graph-Partitionierung[8] je länger mit der jeweiligen Matrix gerechnet wird Sinn. Die zyklische Verteilung erreicht für die Testfälle die

[8]Der Aufwand der Graph-Partitionierung liegt für die SpMV abhängig von der Anzahl der Zeilen in der Größenordnung von 10 Iterationen der Gesamtberechnung.

Matrix	Xeon	Tesla	∑	block	cyclic	even	theo	band	prac	model	corr.	dyn.	+/-%
Block_30000_70	3,02	12,35	61,49	14,70	5,61	16,20	15,49	16,07	14,01	13,61	13,76	17,33	-71,82
Poisson_3D27P_100_100_100	2,77	13,00	63,09	19,52	5,44	20,40	33,71	34,09	33,76	33,98	33,94	38,24	-39,39
bcsstk30	2,89	3,43	25,26	8,41	4,61	9,14	8,40	7,95	9,91	8,95	9,31	12,03	-52,37
Geo_1438	1,73	8,59	41,28	13,10	5,13	13,72	23,13	30,81	30,87	26,72	30,93	32,19	-22,01
ldoor	1,66	5,08	26,93	10,30	5,08	13,76	23,51	25,30	27,48	26,81	27,07	28,54	5,98
nd3k	1,79	7,72	38,05	7,15	8,86	6,73	6,51	6,49	6,53	6,72	6,51	8,48	-77,72
nlpkkt160	2,26	11,79	56,23	10,37	5,21	12,94	22,26	37,51	37,23	23,95		28,98	-33,29
pattern1	2,62	0,67	13,15	2,43	2,93	2,69	2,98	2,89	9,37	3,00	10,33	10,31	-21,57
thread	3,01	6,01	36,09	13,13	9,43	13,47	12,58	12,15	12,98	12,11	12,07	14,97	-58,52

Tabelle 7.13.: Tabellarischer Gesamtvergleich der Partitionierungsmethoden über 8 Prozessore (4 Xeon Prozessoren, 4 Tesla-Karten) für die Test-Matrizen

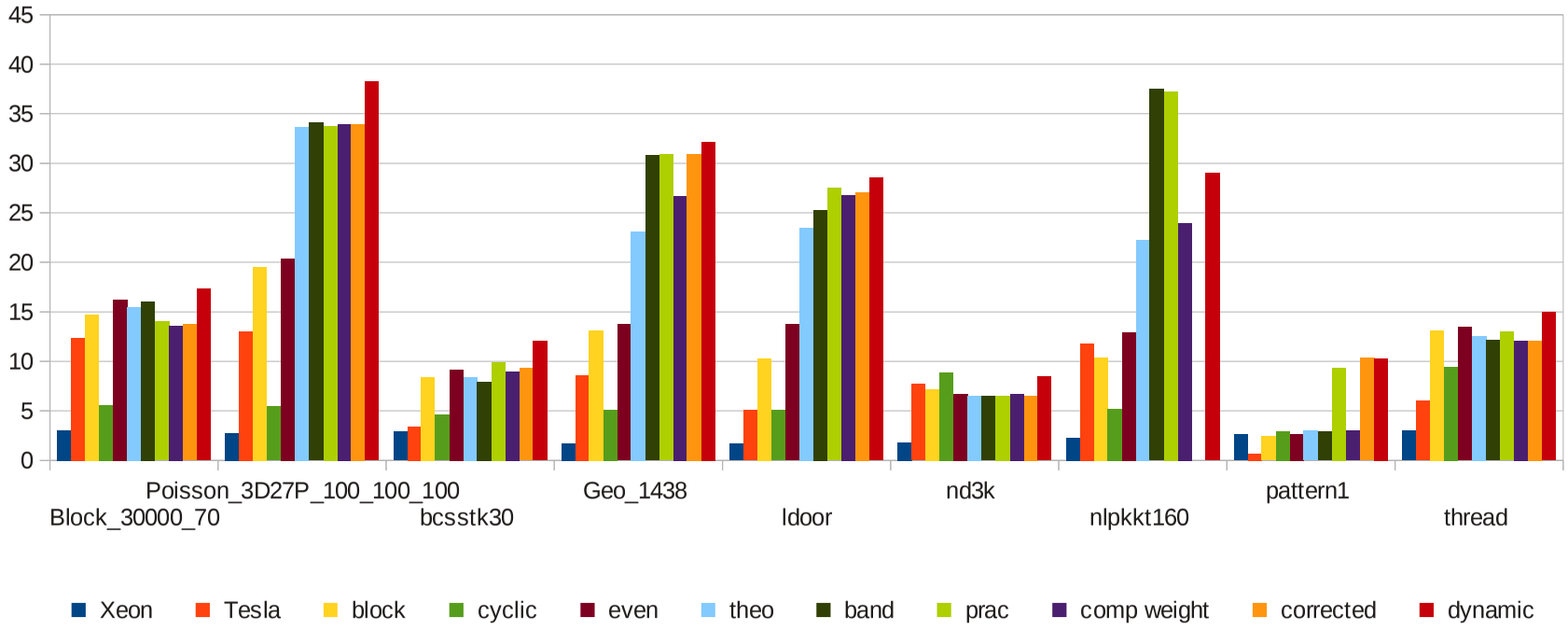


Abbildung 7.14.: Visueller Gesamtvergleich der Partitionierungsmethoden über 8 Prozessoren (4 Xeon Prozessoren, 4 Tesla-Karten) für die Test-Matrizen

schlechteste Leistung. Einzig für die zufällig verteilte *nd3k* ist sie besser als die anderen Methoden. Zwischen den verschiedenen Gewichtungsansätzen sind die Unterschiede i. d. R. nicht so stark wie zu den gleichmäßigen Verteilungen. Trotzdem lassen sich Tendenzen erkennen. So fällt die Gewichtung anhand der maximalen Rechenleistung (theo) meist am schlechtesten aus, weil die SpMV memory bound ist. Die Gewichtung auf Basis der Speicherbandbreite (band) berücksichtigt diesen Faktor und erzielt damit i. d. R. bessere Ergebnisse. Mit der Modell-Gewichtung werden zusätzlich das unterschiedliche Zugriffsverhalten auf die Matrixdaten sowie der Overhead des Kernel Launches berücksichtigt, was noch einmal eine bessere Gewichtung liefert. Damit liegt man meist in Bereichen, die ein praktische Gewichtung auch erzielt, nur dass diese für große Matrizen so nicht möglich wäre. Mit der dynamischen Balancierung lassen sich strukturbedingte Ungleichgewichte balancieren und die Leistung noch steigern. Insgesamt wurde mit der Aufteilung auf acht Prozessoren nicht die Summe ($\sum$) aus vier Xeon-Prozessoren und vier Tesla-Karten erzielt, sondern für große Matrizen nur etwa 60%-80% davon (+/-%); bei kleineren Matrizen fällt dieser Faktor noch deutlich geringer aus, weil für die einzelnen Prozessoren, insbesondere die GPUs, zu wenig zu berechnen ist. An der Matrix *ldoor* sieht man aber, dass die Aufteilung nicht nur so gut sein, dass die addierte Einzelleistung eingestellt wird, sondern das auch noch ein zusätzlicher Laufzeitgewinn möglich ist.

Die Wahl einer geeigneten Gewichtung für das System bei der SpMV ist schwierig, da viele Faktoren auf die resultierende Laufzeit Einfluss nehmen. Wie bereits erläutert wurde, setzt das verwendete Modell zur Repräsentation der Matrix als Graphen voraus, dass die Last für jeden Matrix-Eintrag dieselbe ist. Aufgrund der räumlichen Lokalität von Einträgen und verschiedenen Caches in den Prozessoren trifft dies aber nicht nur nicht zu, da für räumlich lokale Daten beim Zugriff auf den Quell-Vektor x nur auf den Cache und nicht auf den Arbeitsspeicher bzw. globalen Speicher zugegriffen werden muss, sondern wirkt sich für CPU und GPU auch unterschiedlich aus. Grund dafür ist die unterschiedliche Größe und Struktur der Caches im Vergleich zur sonstigen Rechenleistung. Neben allgemeinen Überlegungen bzgl. des Speicherzugriffes auf GPUs im Vergleich zu CPUs spielt das Caching für Matrizen, die keine regelmäßige Verteilung der Einträge haben, eine wichtige Rolle. Die Struktur der Matrix bereitet damit nicht nur bei der Verteilung der Zeilen für das Balancieren der Einträge pro Partition und das Minimieren der Kommunikation Probleme, sondern auch für ein lineares Verhalten der Ausführzeiten bei einer Variation der Gewichte. Entweder bezieht man die strukturbedingte Abweichung bei der Laufzeitvorhersage mit ein, indem man die Last auch abhängig von der Lokalität der Daten definiert oder man verbessert die Struktur, sodass der Fehler kleiner ist. Die Abhängigkeit von der Lokalität der Daten anzunähern (s. Verbesserungsvorschlag für die Laufzeitvorhersage in Absatz 7.4.2) wird schwierig für die Lastdefinition umzusetzten sein. Dies müsste wie in Formel 7.2 skizziert auf der Ebene der Graph-Partitionierung gelöst werden, da man ansonsten zu wenig Informationen zu den Beziehungen zwischen den Einträgen zur Verfügung hat. Alternativ kann die Struktur für die Zugriffe verbessert werden. Dies kann aber nur über

eine zweidimensionale Partionierung geschehen, sodass eine höhere Lokalität der Daten geschaffen wird und das daraus resultierende Cache-Verhalten sich für CPU und GPU nicht so stark unterscheidet. An *ldoor* sieht man, wenn die Voraussetzungen gegeben sind, dass die Laufzeitvorhersage für beide gleich gut ist und sich dies über alle Verteilungen der Matrix so verhält, eine höhere Leistung erzielt werden kann, weil die Partitionierung wie eine Blockung des Algorithmus arbeitet.

8. Zusammenfassung und Ausblick

In dieser Arbeit wurde eine Partitionierungsstrategie entwickelt, um dünn besetzte Matrizen für die SpMV auf ein hybrides System aus Multicore-CPUs und GPUs mit verteiltem Speicher aufzuteilen. Die Aufgabe besteht darin eine geeignete Bewertung der System-Komponenten zu finden, um ihnen einen angemessenen Anteil der Gesamtlast zuzuweisen sowie die Daten, trotz der unregelmäßigen Struktur der Matrix, mit einer zeilenweisen Partitionierung entsprechend der resultierten Gewichtung aufzuteilen. Gleichzeitig muss vermieden werden viele Abhängigkeiten zwischen den Partitionen zu verletzen, um zusätzliche Kommunikation auszuschließen. Das Ziel der Strategie ist es eine lastbalancierte Konfiguration zu finden, die möglichst wenig kommunizieren muss.

Der Aufgabe wurde begegnet, indem mit einem, auf CPU und GPU angepassten Roofline-Modell [WWP08] die Rechenleistung der Hardware für die SpMV und daraus die Laufzeit für eine Matrix vorhergesagt wird. Anhand der Laufzeitprognose kann eine Gewichtung mit balancierten Laufzeitvorhersagen gebildet werden über die mit Hilfe der Graph-Partitionierung eine Verteilung bestimmt wird, die die Last gemäß der Gewichtung verteilt und dabei möglichst wenig Abhängigkeiten zwischen den Partitionen schneidet.
Für die Abbildung der Matrix auf einen Graphen wurde das Standard-Modell der ungerichteten Kanten angewendet. Ferner wurde für die Graph-Partitionierung die häufig eingesetzte Bibliothek Metis [Kar] genutzt.

Es konnte gezeigt werden, dass mit dem vorgeschlagenen Modell das grundsätzlich unterschiedliche Ausführ- und Speicherzugriffsverhalten bei idealen Matrizen für CPU und GPU berücksichtigt werden kann und damit Laufzeiten vorhergesagt wurden, die nur um 10% bzw 5% von der realen Zeit abweichen. Für nicht ideale Matrizen müssten nach weiterer Analyse Korrekturfaktoren, die das unterschiedliche Cache-Verhalten der Prozessoren[1] auf Basis der Matrix-Struktur annähern, hinzugefügt werden.

Daneben steht mit der Graph-Partitionierung und dem Graph-Modell der ungerichteten Kanten eine ausreichende Optimierung des Kommunikationsvolumens zur Verfügung, um die Kommunikation während der SpMV hinter der Berechnung des lokalen Anteils der Matrix zu verstecken. Damit lässt sich in idealen Fällen mit der Aufteilung der Berechnung die addierte Leistung der Einzel-Komponenten des Systems erzielen. Durch zusätzliche Verbesserung der Matrix-Struktur sogar mehr.

Analysen zeigten aber, dass mit einem Verändern der Last, wie sie mit dem Standard-Modell des ungerichteten Graphen definiert wird, i. d. R. keine lineare Auswirkung auf

[1] begründet auf der Ein- bzw. Zweidimensionalität des Caches

die Ausführzeiten genommen wird. Dies liegt nur für einen sequentiellen Zugriff auf den Quell-Vektor x vor. Bei unregelmäßigen Zugriffen definiert sich die Last einer Partition nicht allein über die Anzahl der Einträge, sondern vielmehr über die Lokalität der Einträge und dem daraus resultierenden Cacheverhalten.
Grundsätzlich kann davon ausgegangen werden, dass die Eigenschaft von guter oder schlechter Lokalität in jeder Partition der Verteilung vorliegt, allerdings wirkt sich dies bei einem hybriden System für verschiedene Prozessortypen unterschiedlich aus. Da in den Cache der CPU Zeilen geladen werden, in den Texture Cache der GPU aber Blöcke, treten in unterschiedlichen Fällen Hits und Misses auf, was sich in einer unterschiedlichen Bewältigung der Last wiederspiegelt.

Demnach gehört es zu der Aufgabe der Partitionierungsstrategie auch das Mapping von Partition zu Prozessor zu berücksichtigen. Dazu müsste die Last für die aufzuteilenden Einheiten (in LAMA: Zeilen) entsprechend des zugewiesenen Prozessors schon als Gewicht bei der Graph-Partitionierung angegeben werden. Vorstellbar ist ein Ansatz wie in [WC01] für die Berücksichtigung heterogener Netzwerke. Indem mit dem System-Graphen die spezifische Rechenleistung eingebracht wird, könnte eine Gewichtung der Einheiten, die das Mapping berücksichtigt, angesetzt werden. Dem Problem ist aber mit den derzeitigen Graph-Partitionierungsalgorithmen und -Bibliotheken nicht beizukommen.
Alternativ könnte die Berechnung für CPU und GPU dahingehend optimiert werden, dass beide denselben Einflüssen der Matrixstruktur ausgesetzt werden, wodurch die Last für den bisherigen Ansatz konstant bleibt. Anbieten würde sich eine zweidimensionale Partitionierung, die eine Blockung des Algorithmus vornimmt. Dadurch wird die Lokalität der Daten begrenzt und ein optimales Caching, wie es die derzeitige Laufzeitvorhersage voraussetzt, gefördert. Dazu ist dann ein anderes Graphen-Modell nötig.

Trotz der Beeinflussung durch die Matrixstruktur auf das tatsächliche Laufzeitverhalten einer Partition erzielt der Ansatz der gewichteten Partitionierung über Graph-Partitionierung einen Vorteil gegenüber herkömmlichen Methoden. So konnte gezeigt werden, dass das Kommunikationsvolumen im Vergleich zu einer geblockten Aufteilung deutlich reduziert werden konnte, sodass die Kommunikation bei asynchroner Durchführung keine Auswirkungen auf die Gesamtlaufzeit hat. Außerdem ist eine grundsätzlich bandbreitenbasierte Gewichtung für memory bound Algorithmen wie die SpMV i. d. R. besser als Bewertungen über die maximale Rechenleistung. Dabei zeigt ein laufzeitkorrigiertes Modell oft, dass das zusätzliche Einbeziehen der Matrixgröße, des Zugriffsverhaltens und des extra Kernel Launches für die GPU nochmal bessere Ergebnisse zur alleinigen Bandbreiten-Gewichtung.

Insgesamt ist mit der Aufteilung ein Gewinn zu erzielen, der mit weiteren Verbesserungen am Modell noch optimiert werden kann. An erster Stelle sollte die Laufzeitvorhersage dahingehend verbessert werden, dass der Fehler, der durch die Caches auftritt, reduziert wird. Dafür bietet sich eine genaue Analyse der Cache-Misses mit PAPI [pap] an. Insbesondere die Zweidimensionalität des Texture Caches sollte näher in Verbindung zur Struktur der Matrix untersucht werden und in diesem Kontext, ob es sich lohnt eine zweidimensionale

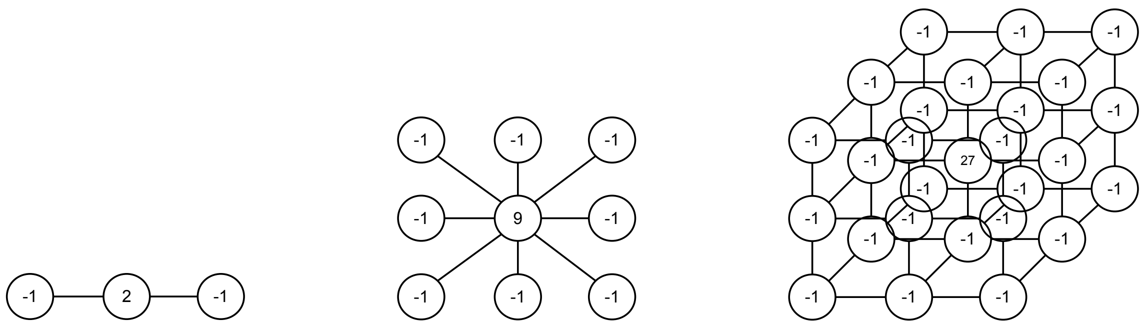
Partitionierung vorzunehmen. Eine solche Aufteilung sollte für die derzeitige Gewichtung Konstanz in die Laufzeiten bei variierenden Partitionsgrößen bringen, was eine bessere Vorhersage möglich macht. Unter diesen Voraussetzungen kann dann auch in Richtung einer geeigneten dynamischen Balancierung weiter entwickelt werden.

Wenn auch mit dem vorgeschlagenen Modell nur in einzelnen Fällen eine optimale Verteilung gefunden werden konnte, so haben sich die Einzelkomponenten (Laufzeitvorhersage und Graph-Partitionierung zur Kommunikationsminimierung) als durchaus tragfähig für die Lastbalancierung gezeigt. Auch bei nicht optimaler Balancierung konnten Verbesserungen in der Laufzeit erzielt werden. Innerhalb von iterativen Lösern kann dieser Gewinn einen deutlichen Vorteil bringen. Dabei wurden die besten Ergebnisse innerhalb weniger Iterationen der (nicht-optimalen) dynamischen Balancierung gefunden, sodass sich der Ansatz schon für schnell konvergierende Verfahren lohnen wird. Zudem sollte es mit den gemachten Verbesserungsvorschlägen zum einem möglich sein die Laufzeitvorhersage zumindest für Matrizen mit vorhersagbarer Strukur, wie den Poisson-Matrizen, zu verbessern, als auch die ausstehenden Unterschiede von CPU- und GPU-Verhalten für die Partitionierung zu berücksichtigen, um so zu besseren Ergebnissen zu gelangen.

A. Anhang

A.1. Poisson-Matrizen

Poisson-Matrizen ergeben sich aus der Diskretisierung von Differenzialgleichungen (DGL). Partielle DGLs können von einer oder mehreren Variablen, Funktionen und ihrer Ableitungen abhängen. Sie beschreiben in gewissen Sinne das Änderungsverhalten der Größen zueinander. Die Diskretisierung einer, in der Physik weit verbreiten partiellen DGL 2.Ordnung, wird über sogenannte Poisson-Sterne dargestellt. Diese können eine beliebige Dimension und eine dazu passende Anzahl Bezugspunkte haben. Definiert man als Bezugspunkte lediglich direkte Nachbarn, so sind es im eindimensionalen Fall maximal 3 Punkte, im zweidimensionalen 9 Punkte und im dreidimensionalen 27 Punkte (s. Abbildug A.1). Klassisches Beispiel für eine eindimensionale Differenzialgleichung ist die Auslenkung einer eingespannten schwingenden Saite, im zwei-/ bzw. dreidimensionalen Fall ist es die Wärmeausbreitung in der Fläche bzw. im Raum.



(a) 1D-Stern **(b)** 2D-Stern **(c)** 3D-Stern

Abbildung A.1.: Poisson-Sterne

An dieser Stelle ist nur die grundsätzliche Struktur der Matrizen von Bedeutung, für eine detailierte Herleitung sei auf [Saa03] verwiesen.
Werden die Poisson-Sterne auf jeden Punkt des jeweiligen Gebietes angewendet, entstehen charakteristische Muster in der zugehörigen Matrix. Es bildet sich immer eine Hauptdiagonale und q Nebendiagonalen entsprechend der Anzahl der Bezugspunkte. Die Abstände zwischen den Nebendiagonalen leiten sich aus den geometrischen Abhängigkeiten der Bezugspunkte ab. Für den gezeigten 1D-3-Punkte-Stern sieht dies wie folgt aus:

$$M = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix}$$

Abbildung A.2.: 1D-3Punkte-Poisson-Matrix der Größe 6×6

A.2. Testsysteme im Detail

A.2.1. Kennzahlen

	Intel Xeon 5650	Intel Core i7
Kerne	6	8
Taktung	2.67 GHz	2.67 GHz
Speichergeschwindigkeit	800 MHz	1066 MHz
Speicherkanäle	3	2
Speicherbandbreite	32 GB/s	21 GB/s

Tabelle A.1.: CPU-Kennzahlen für Intel Xeon 5650 und Intel Core i7

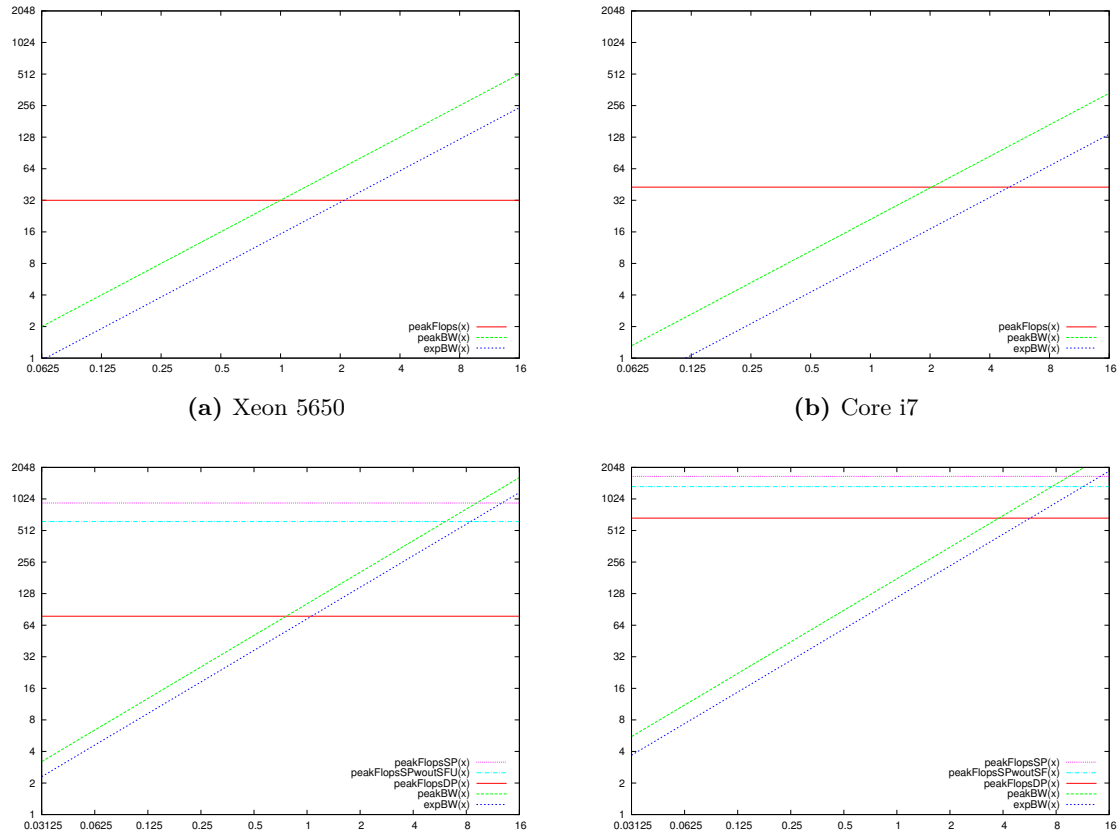
	Tesla C1060	GTX 480
compute capability	1.3	2.0
SMs	32	15
SPs/SM	8	32
$\sum$SPs	240	480
Taktung	1.3 GHz	1.4 GHz
globaler Speicher	4 GB	1.5 GB
Speichergeschwindigkeit	800 MHz	1848 MHz
Speicherbandbreite	102.4 GB/s	177.4 GB/s

Tabelle A.2.: GPU-Kennzahlen für Tesla C1060 und GTX 480

A.2.2. Roofline-Modell

	Xeon 5650	Core i7	Tesla C1060	GTX 480
max. FLOP/s SP	32	42	624 (936)	1344 (1680)
max. FLOP/s DP	32	42	278	672
max. theo. GB/s	32	21	102	177
max. exp. Gb/s	15	8	73	117

Tabelle A.3.: Maximale Leistungswerte der Prozessoren; in Klammern: mit SFU-Unterstützung



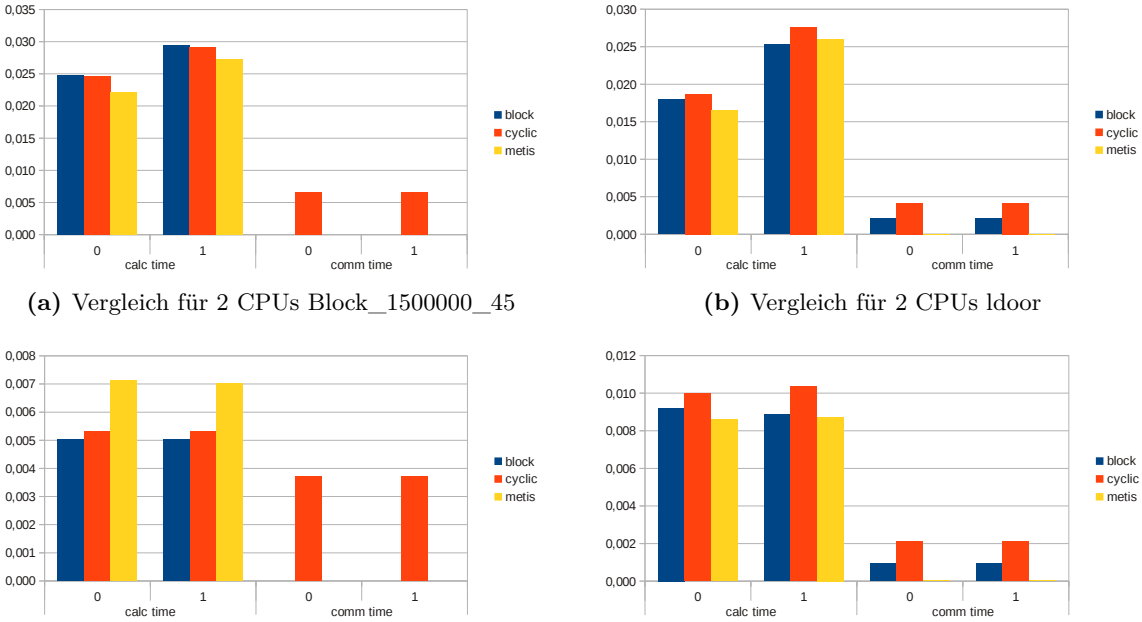
(a) Xeon 5650 **(b)** Core i7

(c) Tesla C1060 **(d)** GTX 460

Abbildung A.3.: Roofline-Modelle der Testsysteme (bei logarithmischer Skalierung)

A.3. Zusätzliche Ergebnisse

A.3.1. Qualität der Partitionierungen



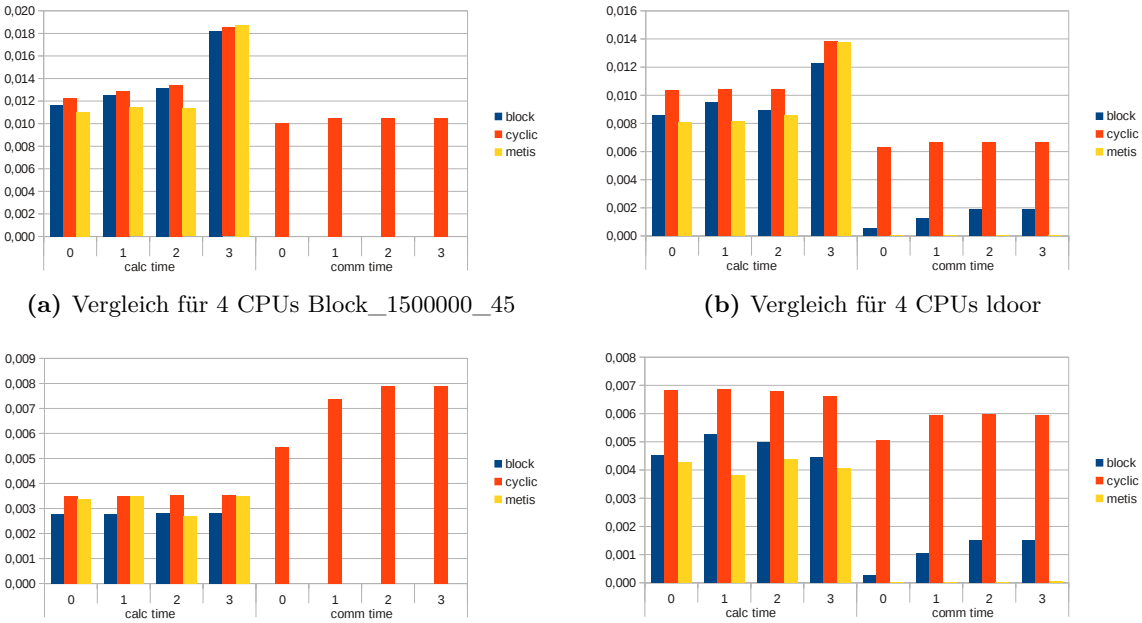
(a) Vergleich für 2 CPUs Block_1500000_45

(b) Vergleich für 2 CPUs ldoor

(c) Vergleich für 2 GPUs Block_1500000_45

(d) Vergleich für 2 GPUs ldoor

Abbildung A.4.: Zeitlicher Vergleich der Qualität zwischen den Methoden für 2 Partitionen eines homogenen Systems aus CPUs (a) bzw. GPUs (b) für die Matrizen *Block_1500000_45* und *ldoor*



(a) Vergleich für 4 CPUs Block_1500000_45

(b) Vergleich für 4 CPUs ldoor

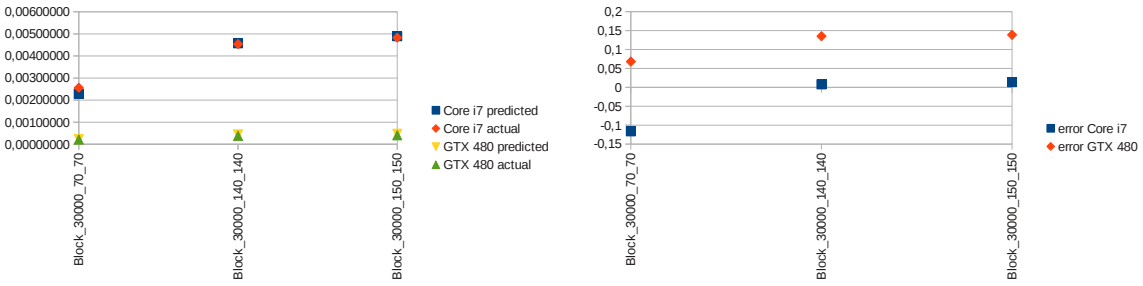
(c) Vergleich für 4 GPUs Block_1500000_45

(d) Vergleich für 4 GPUs ldoor

Abbildung A.5.: Zeitlicher Vergleich der Qualität zwischen den Methoden für 4 Partitionen eines homogenen Systems aus CPUs (a) bzw. GPUs (b) für die Matrizen *Block_1500000_45* und *ldoor*

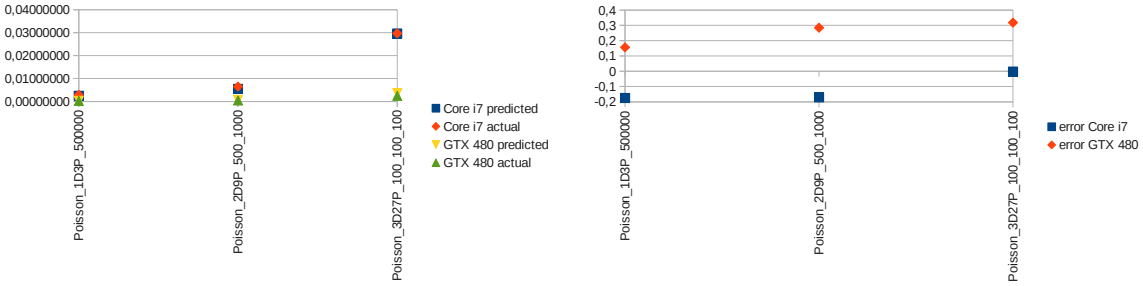
A.3.2. Laufzeitvorhersage (F1Boensch)

Eine entsprechende Laufzeitvorhersage für den F1Boensch (Core i7 und GTX 480) wie in 7.4.2 für den Bull (Xeon und Tesla C1060). Allerdings fehlen teilweise Matrizen, die auf dem Bull berechnet werden, auf dem F1Boensch aber nicht komplett in den Speicher des GTX 480 passten.



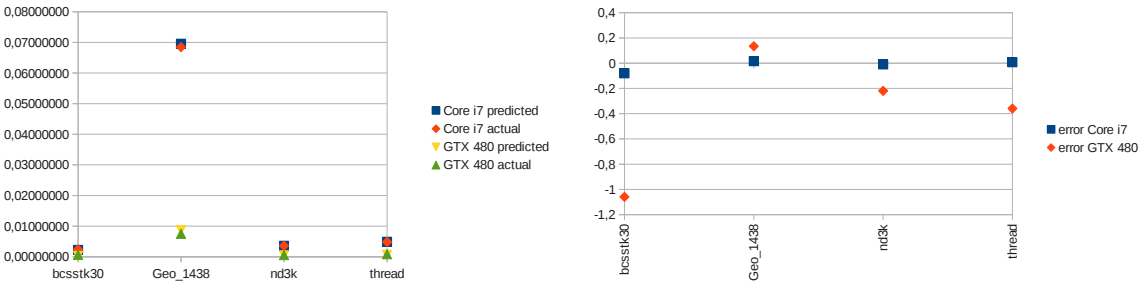
(a) Laufzeitvorhersage **(b)** Fehler

Abbildung A.6.: Laufzeitvorhersage für Block-Matrizen auf dem F1Boensch (Konfg. 2)



(a) Laufzeitvorhersage **(b)** Fehler

Abbildung A.7.: Laufzeitvorhersage für Poisson-Matrizen auf dem F1Boensch (Konfg. 2)



(a) Laufzeitvorhersage **(b)** Fehler

Abbildung A.8.: Laufzeitvorhersage für reale Matrizen auf dem F1Boensch (Konfg. 2)

A.3.3. Gewichtungs-Modell (F1Boensch)

Block_30000_70

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	19,7183
100% 1	-	100,000	12,1107
practical	61,9507	38,0493	20,3883
theoretical	89,6000	10,4000	14,6853
bandwidth	61,4369	38,5631	20,0382
model	60,4691	39,5309	13,3929
corrected	63,0935	36,9065	21,4724
dynamic	64,0195	35,9805	20,8127

Block_1000000_50

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	23,4214
100% 1	-	100,000	13,9723
practical	62,7313	37,2687	29,1886
theoretical	89,6000	10,4000	26,3657
bandwidth	61,4369	38,5631	26,1042
model	59,4234	37,2152	24,6281
corrected	62,7848	37,2152	26,8817
dynamic 4	62,8801	37,1199	36,8949

Poisson_3D7P_150_150_150

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	17,7914
100% 1	-	100,000	9,90429
practical	64,2719	35,7281	20,2832
theoretical	89,6000	10,4000	17,8143
bandwidth	61,4369	38,5631	18,6059
model	60,1234	39,8766	17,7524
corrected	64,4401	35,5599	21,1774
dynamic 1	63,0474	36,9526	21,1736
dynamic 6	66,9862	33,0138	20,9582

bcsstk30

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	6,7755
100% 1	-	100,000	3,4677
practical	66,1466	33,8534	6,7935
theoretical	89,6000	10,4000	6,2819
bandwidth	61,4369	38,5631	6,8436
model	61,4787	38,5213	6,8116
corrected	66,8552	33,1448	6,3561
dynamic	-	-	-

ldoor

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	13,6968
100% 1	-	100,000	5,2203
practical	72,4199	27,5801	18,4569
theoretical	89,6000	10,4000	15,1184
bandwidth	61,4369	38,5631	14,0492
model	59,98	27,5292	13,1598
corrected	72,4708	27,5292	18,2871
dynamic 4	71,6297	28,3703	18,9401

pattern1

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	1,3934
100% 1	-	100,000	0,6908
practical	60,2817	39,7183	1,2406
theoretical	89,6000	10,4000	1,2335
bandwidth	61,4369	38,5631	-
model	-	-	-
corrected	-	-	-
dynamic 1	-	-	-

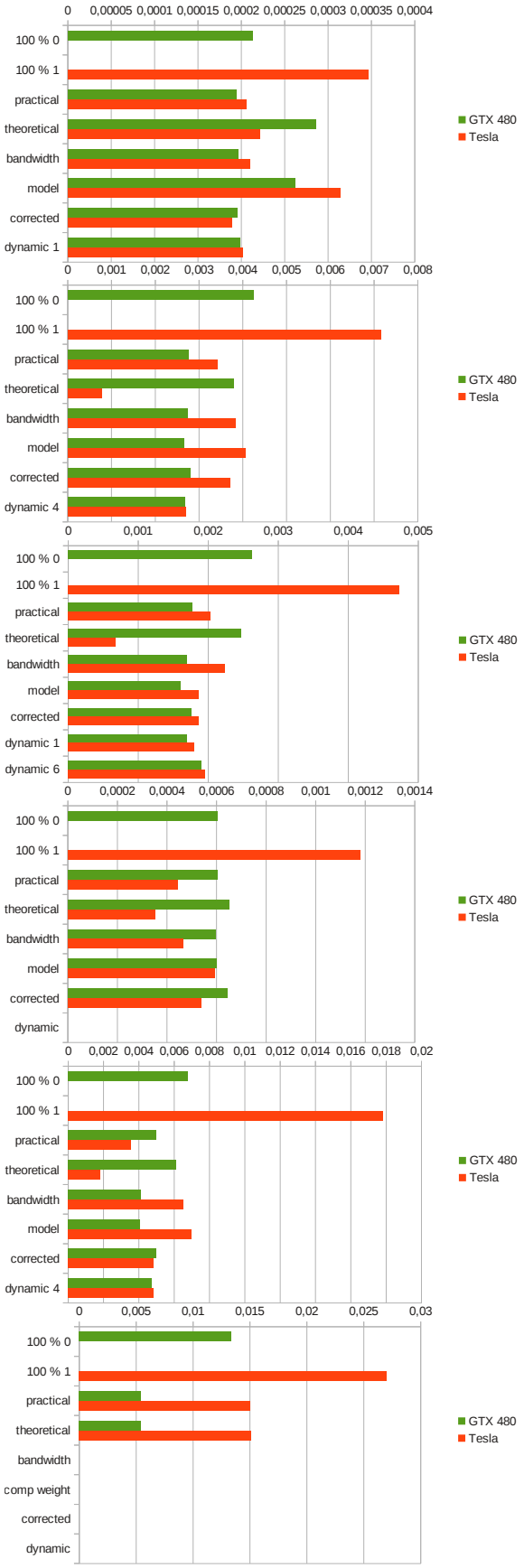


Abbildung A.9.: Balancierungsbetrachtung für die Gewichtungsansätze: Balance zwischen den einzelnen Prozessoren der Konfiguration 1 auf dem F1Boensch

Block_30000_70

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	1,6405
100% 1	-	100,000	19,6445
practical	7,7072	92,2927	19,1083
theoretical	35,3877	64,6123	4,6781
bandwidth	6,7447	93,2552	10,8025
model	9,1435	90,8564	14,3149
corrected	7,7790	92,2209	15,3509
dynamic 7	7,7010	92,2989	18,8004
dynamic 20	3,3372	9,6662	14,9786

Block_1000000_50

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	1,95712
100% 1	-	100,000	23,5051
practical	7,6736	92,3263	17,1975
theoretical	35,3877	64,6123	5,43289
bandwidth	6,7447	93,2552	23,7744
model	8,5213	91,4786	21,3977
corrected	7,6800	92,3199	24,2601
dynamic 4	7,4887	92,5112	24,2460

Poisson_3D7P_150_150_150

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	1,29029
100% 1	-	100,000	17,7968
practical	6,7423	93,2576	16,2820
theoretical	35,3877	64,6123	3,48542
bandwidth	6,7447	93,2552	16,0440
model	11,0422	88,9578	10,8554
corrected	6,7895	93,2104	16,1310
dynamic 2	6,5229	93,4770	16,7034
dynamic 3	6,9562	93,0437	16,4243

bcsstk30

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	1,69556
100% 1	-	100,000	6,76877
practical	20,0319	79,9681	6,96249
theoretical	35,3877	64,6123	5,11769
bandwidth	6,7447	93,2552	6,13477
model	12,2579	87,7421	5,76118
corrected	20,2026	79,7974	6,42002
dynamic 6	24,3535	75,6465	6,92944

ldoor

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	1,8488
100% 1	-	100,000	13,6795
practical	11,8960	88,104	14,7812
theoretical	35,3877	64,6123	4,8846
bandwidth	6,7447	93,2553	12,5057
model	10,9825	89,0175	14,9359
corrected	11,9223	88,0777	14,8029
dynamic 1	11,4098	88,5902	14,9475

pattern1

	Gewicht Xeon	Gewicht Tesla	GFlops
100% 0	100,000	-	1,9754
100% 1	-	100,000	1,4030
practical	58,5106	41,4894	2,4433
theoretical	35,3877	64,6123	1,5738
bandwidth	6,7447	93,2553	-
model	-	-	-
corrected	-	-	-
dynamic 1	-	-	-

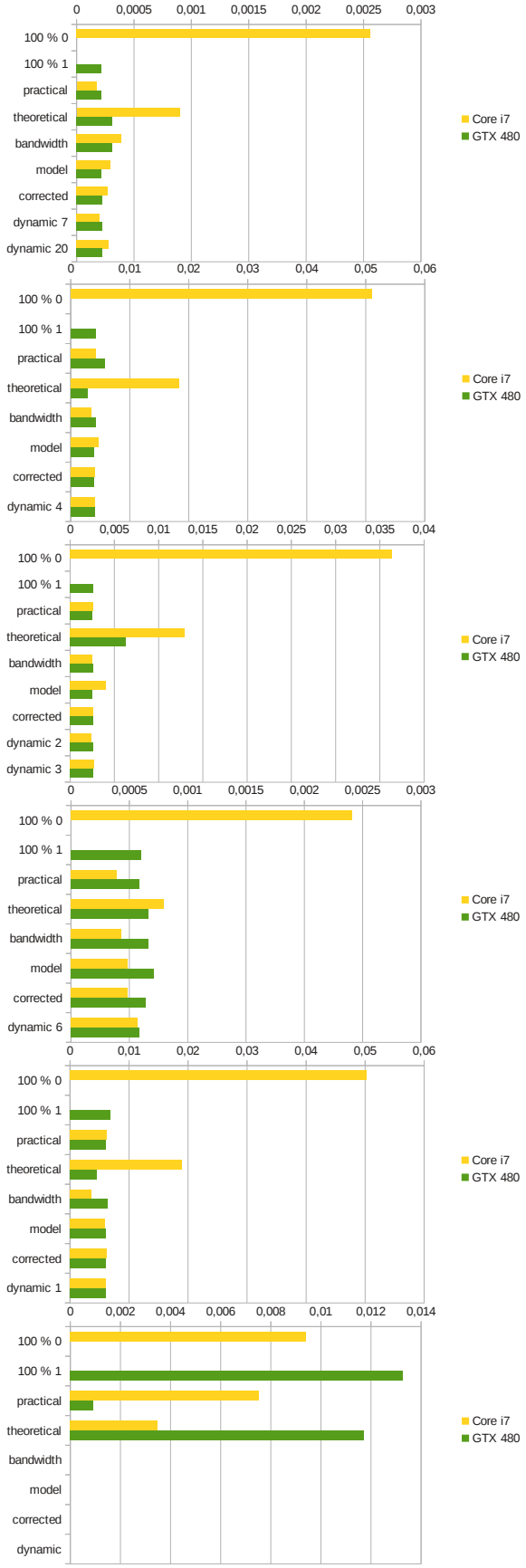


Abbildung A.10.: Balancierungsbetrachtung für die Gewichtungsansätze: Balance zwischen den einzelnen Prozessoren der Konfiguration 2 auf dem F1Boensch

Glossar

AMD

AMD (Advanced Micro Devices) ist einer der größten Chip-Hersteller. Neben Mikroprozessoren und Chipsätzen gehören Grafikchips zu den Produkten, die unter den Markennamen ATI verkauft werden.

Basic Linear Algebra Subprograms (BLAS)

Die Basic Linear Algebra Subprograms sind eine Bibliothek grundlegender Vektor- und Matrixoperationen für dicht besetzte Matrizen. Die Operationen teilen sich auf drei Ebenen auf: Vektor-Vektor-, Matrix-Vektor- und Matrix-Matrix-Operationen. Die BLAS-Schnittstelle zu diesen Operationen wird als allgemeiner Standard angesehen, den eine Vielzahl kommerzieller und frei verfügbarer Alternativen für verschiedene Programmiersprachen und unterschiedliche Hardware implementieren.

compute bound

Ein Algorithmus ist compute bound, wenn die Ausführung im Wesentlichen von der maximalen Rechenleistung des Prozessors bestimmt wird. Dies ist der Fall, wenn die Daten für die Berechnung vornehmlich im Cache vorliegen und nicht aus dem Hauptspeicher gelesen werden. Dafür muss eine hohe räumliche und zeitliche Lokalität der Daten bestehen. Im Roofline Modell werden diese Art mit einer hohen Operational Intensity beschrieben.

compute capability

Die compute capability beschreibt eine Generation von CUDA-Grafikkarten, die dieselben hardwaremäßigen Möglichkeiten und Einschränkungen teilen. So ist mit Karten einer compute capability von 1.0 bis 1.2 nur einfach genaue Floating-Point-Berechnung unterstützt worden, danach erst doppelt genaue nach IEEE 754 Standard. Seit 2.0 werden neben C-Routinen auch eine Menge C++-Klassenfunktionen unterstützt. Wichtigster Unterschied zwischen verschiedenen compute capabilities ist aber die coalesced access size und die Bedingungen die für diese Hardware an einen coalesced memory access geknüpft sind.

Compute Unified Device Architecture (CUDA)

Die Compute Unified Device Architecture ist ein von nVidia entwickeltes Programmiermodell zur einfacheren Programmierung von allgemeinen Rechenanwendungen auf Grafikprozessoren. Oft spricht man von CUDA auch als Programmiersprache,

gemeint ist dabei die C-Spracherweiterung *C for CUDA* zur Umsetzung des CUDA-Modells.

DirectX

DirectX ist Microsofts Grafik-Programmierschnittstelle für Windows Systeme vergleichbar mit OpenGL für Linux-Systeme. Sie wird haupsächlich zur Shader-Programmierung von 2D- und 3D-Grafiken genutzt.

Grid

Unter einem Grid versteht man eine Hard- und Software-Infrastruktur, die einen virtuellen Supercomputer bildet. Dabei sind verschiedenste Rechenressourcen meist in Hierachien und über weite Distanzen miteinander lose gekoppelt. Die Recheninstanzen werden von teilnehmenden Institutionen der Grid-Community über eine sogenannte Middleware bereitgestellt und konsumiert. Ein solches Grid-System besteht aus äußerst heterogenen Recheninstanzen und Netzwerkverbindungen.

Infiniband

Infiniband ist eine Industrie-Standard verschiedener Hardware-Hersteller, der I/O-Architekturen zur Verbindung von verteilten Systemen beschreibt. Dahinter verbirgt sich eine serielle Hochgeschwindigkeitsübertragungstechnologie mit zu 120 GigaBit/s. Infiniband wird vorwiegend für die Verbindung von Clustern verwendet und bringt dabei den großen Vorteil von kurzen Latenzzeiten und hohen Übertragungszeiten mit sich.

Many Integrated Core (MIC)

Der Intel MIC ist ein x86-basierter Multicore mit 512-bit breiten Single Instruction Multiple Data (SIMD)-Registern. In den ersten Chips, die ab 2012 offiziell zu kaufen sein sollen, werden über 50 dieser Kerne pro Chip verbaut sein. Als Schnittstelle bietet sich aufgrund der x86-Architektur ein einfaches OpenMP-Modell an, unterstützt werden außerdem OpenCL sowie Intel Cilk Plus.

memory bound

Ein Algorithmus wird als memory bound bezeichnet, wenn die Ausführzeit im Wesentlichen durch die Speicherzugriffszeit bestimmt wird, weil die Daten für die Berechnung nicht im Cache vorliegen sondern aus dem Hauptspeicher geladen werden müssen. Dieser Umstand wird im Roofline Modell mit einer geringen Operational Intensity charakterisiert.

Message Passing Interface (MPI)

Das Message Passing Interface ist ein Standard, der eine Menge an Operationen zum Nachrichtenaustausch auf verteilten Systemen definiert. Es beschreibt lediglich die Funktion der Operationen, nicht wie sie zu implementiert zu sein hat oder

wie das Nachrichtenprotokol auszusehen hat. Es gibt mehrere Bibliotheken die den MPI-Standard implementieren. Dazu gehören im Wesentlichen MPICH und die frei verfügbare OpenMPI-Bibliothek.

nVidia

nVidia gehört zu einen der größten Chiphersteller. Bekannt ist er vor allem für seine Grafik-Prozessoren und dem dafür entwickelten Programmiermodell CUDA.

Open Computing Language (OpenCL)

OpenCL ist eine Programmierschnittstelle zum Schreiben von plattformunabhängigen Anwendungen. Es handelt sich dabei um einen offenen Standard, der für verschiedene Prozessoren (CPUs, GPUs, DSPs oder der Cell Broadband Engine) implementiert ist.

Open Multi-Processing (OpenMP)

Das Open Multi-Processing ist eine Programmier-Schnittstelle zur compilerseitigen Parallelisierung von Programmen für Multicore-Prozessoren. Die Parallelisierung erfolgt im Gegensatz zu MPI-Programmen bei OpenMP parallelisierten Programmen auf Thread-Ebene und nicht auf Prozess-Ebene. Dabei geht es darum Schleifen sinnvoll unter den Thread aufzuteilen und den gemeinsamen Speicher sicher zu nutzen. Da heutige Parallelrechner aus verteilten Multicore-Prozessoren bestehen sind die Programme i. d. R. sowohl OpenMP- als auch MPI-parallelisiert.

QuickPath Interconnect (QPI)

QuickPath Interconnect ist der neue Verbindungsmechanismus zwischen Prozessoren untereinander und von Prozessoren zum Chipsatz bei Intel ab dem Core i7 und der Nehalem-Architektur. Er löst seinen Vorgänger den Front Side Bus ab, womit von einem Bussystem zu einem Routing-Mechanismus gewechselt wurde.

Single Instruction Multiple Data (SIMD)

SIMD ist eine von vier Klassen der flynnschen Rechnerklassifikation und bezeichnet i. A. Vektorrechner, also solche, die eine Anweisung auf mehreren daten gleichzeitig berechnen. Die Cell B. E. und auch GPUs zählen dazu.

Streaming Multiprozessor (SM)

Innerhalb der CUDA-Architektur beschreibt ein Streaming Multiprozessor einen Verbund aus SP's, die alle dieselbe Kontrolllogik teilen und somit dasselbe Programm simultan ausführen.

Streaming Prozessor (SP)

Ein Streaming Prozessor ist ein einzelner CUDA-Kern, der eine nThread paralle zu den adneren Prozessoren ausführt. Angeordnet sind mehrere SPs in einem SM.

Streaming SIMD Extension (SSE)

Die Streaming SIMD Extensions wurden von Intel als Befehlserweiterung der x86-Architektur entwickelt. Ab dem Pentium III sind Intel-Prozessoren mit 128bit breiten Registern ausgestattet, die es erlauben 4 Instruktionen mit einmal durchzuführen.

Top500

Die Top500 ist eine halbjährlich erscheinende Liste der gleichnamigen Organisation über die schnellsten Supercomputer der Welt. Das Ranking erfolgt anhand der erzielten Leistung im LINPACK. Seit 2007 gibt es daneben auch die Green500, die die Leistung im Verhältnis zu ihrem Stromverbrauch bewerten.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[BBB+10] Balay, Satish ; Brown, Jed ; Buschelman, Kris ; Eijkhout, Victor ; Gropp, William D. ; Kaushik, Dinesh ; Knepley, Matthew G. ; McInnes, Lois C. ; Smith, Barry F. ; Zhang, Hong: PETSc Users Manual / Argonne National Laboratory. 2010 (ANL-95/11 - Revision 3.1). – Forschungsbericht

[BDF+99] Boman, Erik ; Devine, Karen ; Fisk, Lee A. ; Heaphy, Robert ; Hendrickson, Bruce ; Leung, Vitus ; Vaughan, Courtenay ; Çatalyürek Ümit ; Bozdag, Doruk ; Mitchell, William: *Zoltan home page*. 1999. – `http://www.cs.sandia.gov/Zoltan`

[Bom07] Boman, Erik G.: A nested dissection approach for sparse matrix partitioning. In: *PAMM* 7 (2007), Nr. 1, S. 1010803–1010804. – ISSN 1617–7061

[BPR+97] Boisvert, Ronald F. ; Pozo, Roldan ; Remington, Karin ; Barrett, Richard F. ; Dongarra, Jack J.: Matrix Market: A Web Resource for Test Matrix Collections. In: *The Quality of Numerical Software: Assessment and Enhancement*, Chapman & Hall, 1997, S. 125–137

[cA99] Çatalyürek, Ümit V. ; Aykanat, Cevdet: Hypergraph-Partitioning Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. In: *IEEE Transactions on Parallel and Distributed Computing* 10 (1999), S. 673–693

[cA11] Çatalyürek, Ümit V. ; Aykanat, Cevdet: *PaToH: Partitioning Tool for Hypergraphs*, 2011. – `http://bmi.osu.edu/~umit/software.html`,

[CM69] Cuthill, E. ; McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: *Proceedings of the 1969 24th national conference*. New York, NY, USA : ACM, 1969 (ACM '69), S. 157–172

[cud] *CUDA C/C++ SDK CODE Samples*. `http://developer.nvidia.com/cuda-cc-sdk-code-samples`

[Dav94] Davis, Timothy A.: University of Florida Sparse Matrix Collection. In: *NA Digest* 92 (1994)

[DGM+09] Dyk, Danny van ; Geveler, Markus ; Mallach, Sven ; Ribbrock, Dirk ; Göddeke, Dominik ; Gutwenger, Carsten: HONEI: A collection of libraries for numerical computations targeting multiple processor architectures. In: *CoRR* abs/0904.4152 (2009)

[Don88] Dongarra, Jack: The LINPACK Benchmark: An Explanation. In: *Proceedings of the 1st International Conference on Supercomputing*. London, UK : Springer-Verlag, 1988. – ISBN 3–540–18991–2, S. 456–474

[DPHL95] Diniz, Pedro ; Plimpton, Steve ; Hendrickson, Bruce ; Leland, Robert: *Parallel Algorithms for Dynamically Partitioning Unstructured Grids*. 1995

[eig] *Eigen*. `http://eigen.tuxfamily.org/`

[Els97] Elsner, Ulrich: *Graph Partitioning - A Survey*. 1997

[Fai05] Faik, Jamal: *A Model for Resource-Aware Load Balancing on Heterogeneous and Non-Dedicated Clusters*. Troy, Rensselaer Polytechnic Institute, Diss., 2005

[FM82] Fiduccia, C. M. ; Mattheyses, R. M.: A linear-time heuristic for improving network partitions. In: *Proceedings of the 19th Design Automation Conference*. New York, NY, USA : ACM, 1982 (DAC '82). – ISBN 0–89791–020–6, S. 175–181

[Geo73] George, Alan: Nested Dissection of a Regular Finite Element Mesh. In: *SIAM Journal on Numerical Analysis* 10 (1973), Nr. 2, S. 345–363

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 0–201–63361–2

[GPS76] Gibbs, Norman E. ; Poole, William G. ; Stockmeyer, Paul K.: An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. In: *SIAM Journal on Numerical Analysis* 13 (1976), S. 236–250

[Gup] Gupta, Anshul: An Evaluation of Parallel Graph Partitioning and Ordering Softwares on a Massively Parallel Computer / IBM Research Division, Thomas J. Watson Research Center. – Forschungsbericht

[HAB03] Huang, Sili ; Aubanel, Eric ; Bhavsar, Virendrakumar C.: Mesh partitioners for computational grids: a comparison. In: *Proceedings of the 2003 international conference on Computational science and its applications: PartIII*. Berlin, Heidelberg : Springer-Verlag, 2003 (ICCSA'03). – ISBN 3–540–40156–3, S. 60–68

[HB01] Hu, Y. F. ; Blake, R. J.: Load balancing for unstructured mesh applications. Commack, NY, USA : Nova Science Publishers, Inc., 2001. – ISBN 1–59033–011–0, S. 117–148

[HBH+05] Heroux, Michael A. ; Bartlett, Roscoe A. ; Howle, Vicki E. ; Hoekstra, Robert J. ; Hu, Jonathan J. ; Kolda, Tamara G. ; Lehoucq, Richard B.

; Long, Kevin R. ; Pawlowski, Roger P. ; Phipps, Eric T. ; Salinger, Andrew G. ; Thornquist, Heidi K. ; Tuminaro, Ray S. ; Willenbring, James M. ; Williams, Alan ; Stanley, Kendall S.: An overview of the Trilinos project. In: *ACM Trans. Math. Softw.* 31 (2005), Nr. 3, S. 397–423

[Hen00] Hendrickson, Bruce: Load balancing fictions, falsehoods and fallacies. In: *Applied Mathematical Modelling* 25 (2000), Nr. 2, S. 99–108

[HK99a] Hendrickson, Bruce ; Kolda, Tamara G.: Graph Partitioning Models for Parallel Computing. In: *Parallel Computing* 26 (1999), S. 1519–1534

[HK99b] Hendrickson, Bruce ; Kolda, Tamara G.: Partitioning Rectangular and Structurally Unsymmetric Sparse Matrices for Parallel Processing. In: *SIAM J. Sci. Comput.* 21 (1999), December, S. 2048–2072. – ISSN 1064–8275

[Int] Intel: *Cilk Plus - A quick, easy and reliable way to improve threaded performance.* `http://software.intel.com/en-us/articles/intel-cilk-plus/`

[jos] *JOSTLE — graph partitioning software.* `http://staffweb.cms.gre.ac.uk/~wc06/jostle/`

[Kar] Karypsis, George: *Family of Graph and Hypergraph Partitioning Software.* `http://glaros.dtc.umn.edu/gkhome/views/metis`

[KD10] Krotkiewski, M. ; Dabrowski, M.: Parallel symmetric sparse matrix-vector product on scalar multi-core CPUs. In: *Parallel Comput.* 36 (2010), April, S. 181–198. – ISSN 0167–8191

[KDB02] Kumar, S. ; Das, S.K. ; Biswas, R.: Graph partitioning for parallel applications in heterogeneous Grid environments. In: *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002*, 2002, S. 66 –72

[KGK09] Karakasis, Vasileios ; Goumas, Georgios ; Koziris, Nectarios: Perfomance Models for Blocked Sparse Matrix-Vector Multiplication Kernels. In: *Proceedings of the 2009 International Conference on Parallel Processing.* Washington, DC, USA : IEEE Computer Society, 2009 (ICPP '09). – ISBN 978–0–7695–3802–0, S. 356–364

[KH10] Kirk, David B. ; Hwu, Wen-mei W.: *Programming Massively Parallel Processors: A Hands-on Approach.* 1st. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2010. – ISBN 0123814723

[KL70] Kernighan, B. W. ; Lin, S.: An Efficient Heuristic Procedure for Partitioning Graphs. In: *The Bell system technical journal* 49 (1970), Nr. 1, S. 291–307

[LE08] Lee, Seyong ; Eigenmann, Rudolf: Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems. In: Zhou, Pin (Hrsg.): *ICS*, ACM, 2008. – ISBN 978–1–60558–158–3, S. 195–204

[McC07] McCalpin, John D.: STREAM: Sustainable Memory Bandwidth in High Performance Computers / University of Virginia. Version: 1991-2007. `http://www.cs.virginia.edu/stream/`. Charlottesville, Virginia, 1991-2007. – Forschungsbericht

[MK08] Moulitsas, Irene ; Karypis, George: Architecture Aware Partitioning Algorithms. In: *Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*. Berlin, Heidelberg : Springer-Verlag, 2008 (ICA3PP '08). – ISBN 978–3–540–69500–4, S. 42–53

[mtl] *MTL4 - Matrix Template Library 4*. `http://www.simunova.com/de/mtl4`

[MV99] Mahapatra, Nihar R. ; Venkatrao, Balakrishna: The processor-memory bottleneck: problems and solutions. In: *Crossroads* 5 (1999). – ISSN 1528–4972

[net] *FocusWare NetWorks MNO*. `http://focusware.co.uk/`

[Nvi09] Nvidia: *Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009

[Nvi10] Nvidia: *CUDA C Best Practices Guide*, 2010

[opea] *OpenCL - The open standard for parallel programming of heterogeneous systems*. `http://www.khronos.org/opencl/`

[opeb] *OpenMP - The OpenMP API specification for parallel programming*. `http://openmp.org/wp/openmp-specifications/`

[pap] *PAPI - Performance Application Programming Interface*. `http://icl.cs.utk.edu/papi/index.html`

[Pel] Pellegrini, François: *Software package and libraries for sequential and parallel graph partitioning, static mapping, and sparse matrix block ordering, and sequential mesh and hypergraph partitioning*. `http://www.labri.fr/perso/pelegrin/scotch/`

[Per98] Per-Olof Fjällström: Algorithms for Graph Partitioning: A Survey. In: *Linköping Electronic Articles in Computer and Information Science* 3 (1998)

[Saa03] Saad, Y.: *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003. – ISBN 0898715342

[SK09] Sancho, José C. ; Kerbyson, Darren J.: Dynamic Load Balancing of Matrix-Vector Multiplications on Roadrunner Compute Nodes. In: *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg : Springer-Verlag, 2009 (Euro-Par '09). – ISBN 978–3–642–03868–6, S. 166–177

[spa] *SPARSKIT: a basic tool kit for sparse matrix computations*. `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/`

[TK06] Trifunovic, Aleksandar ; Knottenbelt, William J.: A General Graph Model for Representing Exact Communication Volume in Parallel Sparse Matrix-Vector Multiplication. In: *21st International Symposium on Computer and Information Sciences (ISCIS 2006)*, 2006, S. 813–824

[top] *TOP500 Supercomputing Sites*. `http://www.top500.org/`

[ubl] *Boost: Basic Linear Algebra Library*. `http://www.boost.org/doc/libs/1_48_0/libs/numeric/ublas/doc/index.htm`

[VB05] Vastenhouw, Brendan ; Bisseling, Rob H.: A Two-Dimensional Data Distribution Method For Parallel Sparse Matrix-Vector Multiplication. In: *SIAM Review* 47 (2005), S. 67–95

[vie] *ViennaCL*. `http://viennacl.sourceforge.net/`

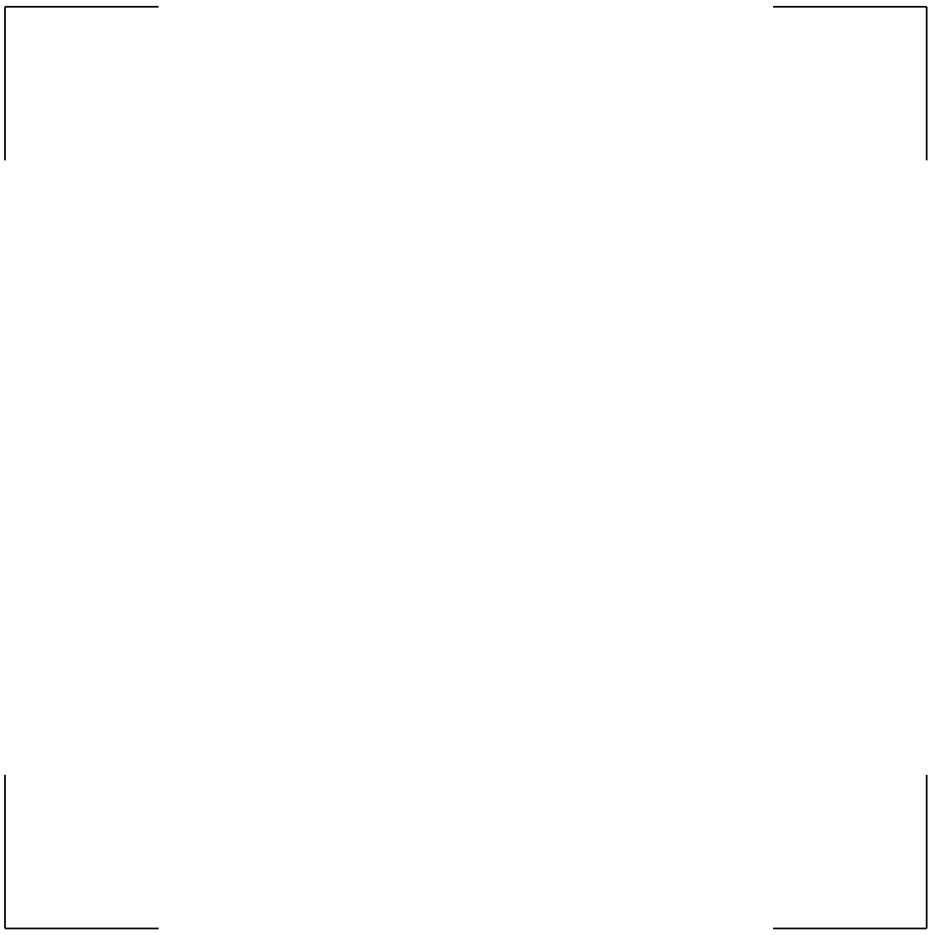
[WC01] Walshaw, C. ; Cross, M.: Multilevel Mesh Partitioning for Heterogeneous Communication Networks. In: *Future Generation Comput. Syst* 17 (2001), S. 601–623

[Wil91] Williams, Roy: Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations. In: *Concurrency* 3 (1991), S. 457–481

[WWP08] Williams, Samuel W. ; Waterman, Andrew ; Patterson, David A.: Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures / EECS Department, University of California, Berkeley. 2008 (UCB/EECS-2008-134). – Forschungsbericht

[XL97] Xu, C. ; Lau, F.C.M.: *Load balancing in parallel computers: theory and practice*. Kluwer Academic Publishers, 1997 (The Kluwer international series in engineering and computer science). – ISBN 9780792398196

[YPS11] Yang, Xintian ; Parthasarathy, Srinivasan ; Sadayappan, P.: Fast sparse matrix-vector multiplication on GPUs: implications for graph mining. In: *Proc. VLDB Endow.* 4 (2011), S. 231–242. – ISSN 2150–8097



Inhalt:

/Ergebnisse

Benchmark-Ergebnisse aus dem Auswertungs-Kapitel

/Literatur

Zitierte Literatur in PDF-Format

/PDF

Diese Arbeit

/Sourcecode_LAMA

Sourcecode des LAMA-Framework

/TestMatrizen

TestMatrizen im MatrixMarket Format