Performanceanalyse eines Block-Eigenwert-Lösers

Bachelor Thesis

zur Erlangung des Grades

Bachelor of Science in Angewandter Mathematik

an der Hochschule RheinMain
Wiesbaden

Autor: Florian Fritzen
florianfritzen@web.de
Matr.-nummer: 473887
Adresse: Robert-Bosch Straße 4
64293 Darmstadt
06151/9811437

Betreuer: Prof. Dr. Edeltraud Gehrig
Betreuer: Dr. Jonas Thies

Abgabedatum: 15.10.2015

I Kurzfassung

Heutige Supercomputer bestehen im Allgemeinen aus einer Vielzahl von „Knoten“ (sog. nodes), die über ein schnelles Netzwerk verbunden sind. Jeder Knoten kann wiederum aus verschiedenen Komponenten bestehen, z.B. CPUs mit mehreren Rechenkernen, Grafikkarten (GPUs) und anderen arithmetischen Beschleunigern. Im Rahmen des DFG Projekts ESSEX („Equipping Sparse Solvers for Exa-scale“) wurde in den letzten zwei Jahren ein Block-Jacobi-Davidson Löser zur Berechnung einiger Eigenwerte einer großen, dünnbesetzten Matrix entwickelt, der nachweisbar auf jeder dieser Komponenten optimale Performance erzielt. Da der Algorithmus jedoch verschiedene Grundoperationen ausführt, die auf den unterschiedlichen Komponenten eines Knoten unterschiedlich schnell sein können und somit eine Lastimbalanz entstehen kann, werden in dieser Bachelorthesis zunächst diese Operationen auf ihre Performance untersucht, um anschließend auf eine sinnvolle Verteilung auf CPU und Grafikkarte schließen zu können.

II Inhaltsverzeichnis

III Abbildungsverzeichnis

IV Abkürzungsverzeichnis

BLAS Basic Linear Algebra Subprograms
CRS Compressed Row Storage
CUDA Compute Unified Device Architecture
DLR Deutsches Zentrum für Luft- und Raumfahrt
ESSEX Equipping Sparse Solvers for Exascale
GEMM General Matrix Multiplication
GHOST General, Hybrid and Optimized Sparse Toolkit
HPC High Performance Computing
MPI Message Passing Interface
OpenMP Open Multi-Processing

1 Einleitung

1.1 Arbeitsgruppe High Performance Computing

Das Deutsche Zentrum für Luft- und Raumfahrt e.V. (DLR)[1] ist ein Forschungszentrum der Bundesrepublik Deutschland mit Hauptsitz in Köln. Mit seinen rund 8000 Mitarbeitern sind die größten Tätigkeitsfelder in der angewandten Forschung und Grundlagenforschung in den Bereichen Luft- und Raumfahrt, Verkehr, Energie und Sicherheit angesiedelt. Die Einrichtung *Simulations- und Softwaretechnik* befasst sich mit der Softwareentwicklung für verteilte und mobile Systeme, Software für eingebettete Systeme, Visualisierung und High Performance Computing (kurz: HPC). Diese DLR-Einrichtung ist an den Standorten Köln-Porz, Braunschweig und Berlin vertreten und gliedert sich in die Abteilungen *Verteilte Systeme und Komponentensoftware* und *Software für Raumfahrtsysteme und interaktive Visualisierung* auf. Die Abteilung *Verteilte Systeme und Komponentensoftware* ist in die drei Arbeitsgruppen *Verteilte Softwaresysteme, Software Engineering* und *High Performance Computing* aufgeteilt, wobei diese Thesis im Rahmen der letzteren Gruppe verfasst wurde. Die HPC-Gruppe befasst sich mit Themen wie Big Data, Algorithmen und Datenstrukturen, sowie deren Parallelisierung für moderne Rechnerarchitekturen. Eine solch moderne Rechnerarchitektur stellt zum Beispiel das Rechnercluster EMMY[2] im RRZE Erlangen dar.

1.2 Setup und Testumgebung

Die Testläufe und Benchmarks wurden auf dem EMMY-Cluster am Regionalen Rechenzentrum Erlangen durchgeführt. EMMY ist für Prozesse mit mittleren bis hohen Kommunikationsanforderungen gedacht und besteht aus 560 Rechenknoten, welche jeweils mit zwei Intel Xeon 2660v2 „Ivy Bridge“ Prozessoren (Taktrate 2.2 GHz, 64GB RAM) besetzt sind. Als Beschleuniger dienen EMMY 16 Intel Xeon Phi Koprozessoren, sowie 16 Nvidia K20 GPGPUs. Diese Rechenknoten sind durch ein schnelles InfiniBand Netzwerk miteinander verbunden. Beim Starten einer Anwendung auf EMMY wird für jeden Sockel und für die GPU ein eigener Prozess erstellt. Kommuniziert ein Sockel des einen Knotens mit einem Sockel des anderen Knotens, verläuft diese Kommunikation über das eben genannte InfiniBand Netzwerk. Kommuniziert jetzt ein Sockel

[1] http://www.dlr.de/

[2] https://www.rrze.fau.de/dienste/arbeiten-rechnen/hpc/systeme/emmy-cluster.shtml

des einen Knotens mit der GPU des selben Knoten, so geschieht dies über einen langsameren PCI Express Bus. Zusätzlich sei noch die Kommunikation zwischen zwei Sockeln des selben Knotens via shared memory erwähnt. Die unterschiedliche Kommunikationsgeschwindigkeit und die verschiedene Performance der einzelnen Komponenten kann bei parallelen Anwendungen, bei der die unterschiedlichen Threads auf alle Komponenten eines Rechenknotens verteilt werden, zu Lastimbalanzen führen.

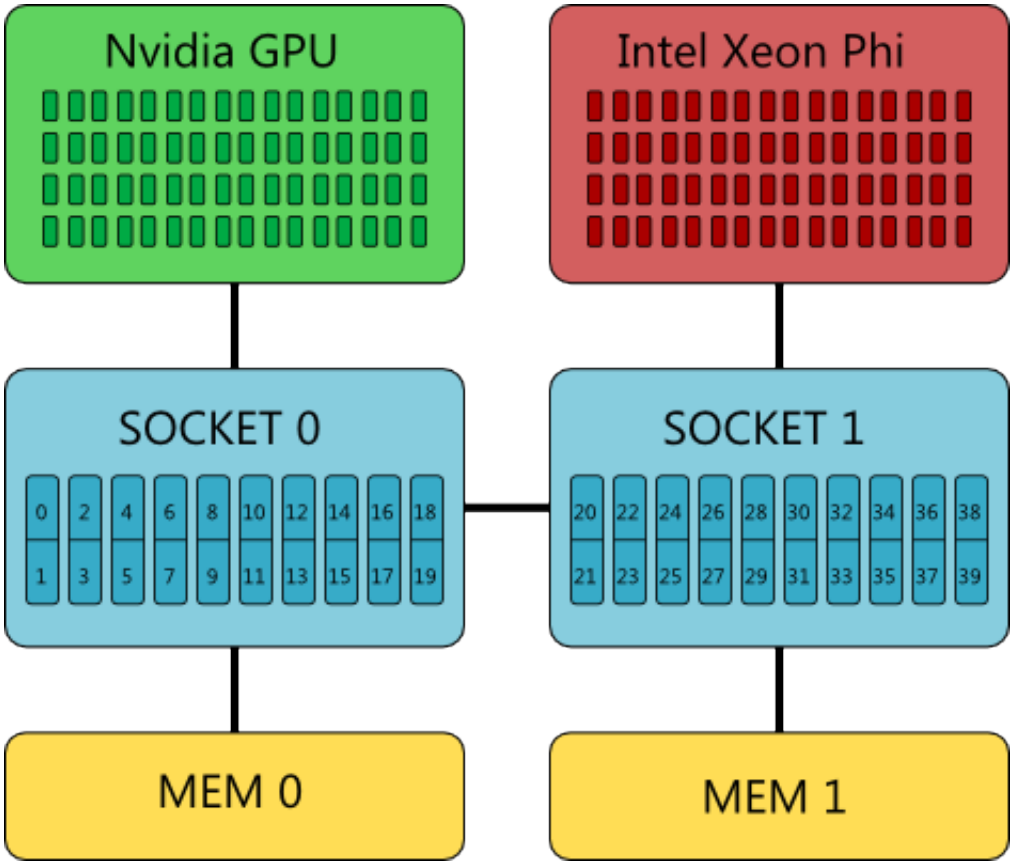


Abbildung 1: Heterogener Rechenknoten mit 2 CPU Sockets + GPU + Koprozessor

1.3 Motivation und Ziel der Bachelorarbeit

Das ESSEX-Projekt[3], welches im Rahmen des SPPEXA-Programmes[4] im Bereich Grundlagenforschung gefördert wird, untersucht rechnerische Herausforderungen bei extrem großen Eigenwertproblemen, entwickelt Programmkonzepte und numerische Methoden im Exascale-Bereich und befasst sich mit

[3]http://blogs.fau.de/essex/
[4]http://www.sppexa.de/

extremer Parallelisierung, Energieeffizienz und Stabilität dieser Konzepte.
Exascale heißt zunächst, dass man in einigen Jahren Supercomputer erwartet, die pro Sekunde 10^{18} Gleitkommaoperationen (1 Exa-Flop/s) ausführen, auf denen man Eigenwertprobleme der Größenordnung 10^{15} lösen möchte.
Möchte man sich nun mit diesen Matrizen auseinandersetzen, benötigt man entsprechende Werkzeuge. Diese Werkzeuge bietet die GHOST-Library[5], welche verschiedene Bausteine für dünnbesetzte lineare Algebra liefert. Sie unterstützt verschiedene Ebenen von parallelen Programmierstrukturen (MPI, OpenMP, CUDA), verschiedene Speichermodelle für große dünnbesetzte Matrizen (CRS, SELL-C-σ) und beinhaltet viele Rechenoperationen, die schon in Hinsicht auf Parallelität optimiert sind. Ziel dieser Bachelorarbeit ist es nun, zunächst den wichtigsten Baustein des Jacobi-Davidson Algorithmus mithilfe des GHOST-Frameworks zu implementieren, um danach dessen Performance auf CPU+GPU zu messen und zu analysieren. Anschließend wird, ausgehend von den Ergebnissen, eine möglichst optimale Verteilung der Rechenoperationen auf die unterschiedlichen Rechenkomponenten erschlossen.

1.4 Unterschied zwischen CPU und GPU

Um sich den Leistungsunterschieden zwischen CPU und GPU bewusst zu werden, muss man sich zunächst die unterschiedlichen Ansätze hinter beiden Architekturen vor Augen führen. CPUs sind seit jeher auf die Abarbeitung von seriellem Code bzw. auf die Ausführung von klassischen „single-threaded“-Programmen ausgelegt. Diese Programme besitzen oft eine hohe Datenlokalität, d.h. sie arbeiten auf einem relativ kleinen Speicherbereich, bestehen aus einer Vielzahl komplexer Instruktionen und weisen eine hohe Anzahl bedingter Verzweigungen im Programmcode auf. Aus diesem Grund wird bei der Entwicklung von CPUs nicht ausschließlich darauf geachtet, dass der Prozessor möglichst viel Rechenleistung aufzeigt, sondern darauf, dass die Abläufe, die der Prozessor zu berechnen hat, optimiert sind. Daher verfügt die CPU über eine Vielzahl verschiedener Rechenwerke und in der Regel wenige Kerne mit einer Taktfrequenz im Gigaherz-Bereich. Mit der Einführung der Multicore-Architektur für Prozessoren war es CPUs nun möglich, Aufgaben auf alle Kerne zu verteilen, so dass entweder alle Kerne zusammen eine Aufgabe abarbeiten oder jeder einzelne Kern eine separate Aufgabe parallel abarbeitet.

[5] https://bitbucket.org/essex/ghost

Im Gegensatz dazu sind Grafikprozessoren auf die Anforderungen datenparalleler Anwendungen zugeschnitten. Mit datenparallelen Anwendungen sind Programme gemeint, die dieselben Operationen auf einer großen, gleichartigen Datenmenge ausführen können. Bei einer GPU kann somit ein erheblicher Teil der Chipfläche für Recheneinheiten, sog. ALUs, verwendet werden, die bei großen Datenmengen auch effektiv genutzt werden können. Klassische CPU-Aufgaben lassen sich jedoch nicht immer ohne Weiteres auf die GPU portieren, da die Grafikprozessoren bei seriellen Programmen nicht von den vielen Recheneinheiten profitiert. Würden diese nur seriell auf einem dieser Recheneinheiten ausgeführt, könnte nur ein Bruchteil der Leistung erreicht werden, vorausgesetzt die Aufgaben beinhalten keine komplexen Befehle oder Sonderfunktionen und lassen sich überhaupt auf einer simplen ALU ausführen.

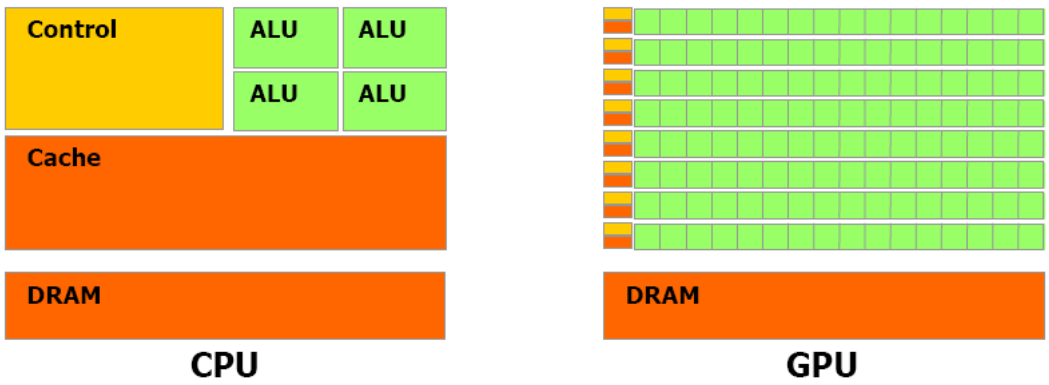


Abbildung 2: Aufbau von CPU-Core und GPU

Abbildung 2 zeigt hier den Unterschied der beiden besprochenen Architekturen. Auf der linken Seite erkennt man die großen Bereiche für Cache- und Control-Einheiten, sowie die vergleichsweise geringe Anzahl an Recheneinheiten. Vergleicht man diese Bereiche mit der GPU auf der rechten Seite, so erkennt man, dass hier Cache- und Steuereinheiten verhältnismäßig klein ausfallen und die meiste Fläche von einer Vielzahl an Recheneinheiten belegt ist. Für den Zweck, für den GPUs entworfen werden, ist allerdings auch keine Notwendigkeit für große Caches und Controller, da sich Speicherlatenzen hinter einer großen Zahl paralleler Rechenoperationen verstecken lassen.[6]

[6]https://www.matse.itc.rwth-aachen.de/dienste/public/show_document.php

2 Der Jacobi-Davidson Algorithmus

Der Jacobi-Davidson Algorithmus, so wie wir ihn kennen, wurde 1996 von Gerard Sleijpen & Henk van der Vorst[7] entwickelt und setzt sich aus zwei Algorithmen zusammen.

2.1 Der Davidson-Algorithmus

Der Davidson-Algorithmus ist ein gängiges Verfahren um einige der kleinsten oder größten Eigenwerte einer großen dünnbesetzten reellen symmetrischen Matrix zu berechnen. Am effektivsten ist es, wenn die Matrix nahezu in Diagonalgestalt ist, z.B. wenn die Matrix der Eigenvektoren ähnlich der Einheitsmatrix ist. Die Hauptverwendung findet die Davidson-Methode in der theoretischen Chemie und Quantenphysik, wo Matrizen meist stark diagonaldominant sind.
Ähnlich wie die Lanczos-Methode ist das Davidson-Verfahren eine iterative Projektionsmethode, die jedoch nicht die Vorteile von Krylov-Unterräumen benutzt, sondern das Rayleigh-Ritz-Verfahren mit nicht-Krylov-Räumen und den Suchraum auf andere Weise erweitert.

Sei $U_k := \left[u^1, \ldots, u^k\right] \in \mathbb{R}^m$ ein Satz aus Orthonormalvektoren und sei (θ, x) ein Eigenpaar des projizierten Problems

$$U_k^T A U_k x = \lambda x \tag{1}$$

und $y = U_k x$.
Davidson schlug 1975[8] vor den Suchraum U_k in die Richtung

$$t := (D_A - \theta \mathbb{1})^{-1} r \tag{2}$$

zu erweitern, wobei $r := Ay - \theta y$ das Residuum des Ritz-Paares (y, θ) ist und D_A die Diagonalelemente von A bezeichnet. u^{k+1} erhält man indem man t zu U_k orthogonalisiert (z.B. mithilfe des Gram-Schmidt-Verfahrens).
Ein wenig irritierend ist die Tatsache, dass das Verfahren für Diagonalmatrizen fehlschlägt, d.h. wenn A Diagonalgestalt hat und (θ, y) ein Ritz-Paar ist,

[7] Siehe [SV00].
[8] Siehe [Dav75].

dann ist

$$t = (D_A - \theta\mathbb{1})^{-1} r = y \in \text{Erzeugnis von } U_j$$

und U_j würde nicht erweitert werden.

Algorithmus 1 Davidson-Algorithmus-Pseudocode

1: Wähle Startvektor u^1 mit $\| u^1 \| = 1, U_1 = [u^1]$
2: **for** $j = 1, 2, \ldots$ **do**
3: $w^j = Au^j$
4: **for** $k = 1, \ldots, j-1$ **do**
5: $b_{kj} = \left(u^k\right)^T w^j$
6: $b_{jk} = \left(u^j\right)^T w^k$
7: **end for**
8: $b_{jj} = \left(u^j\right)^T w^j$
9: Berechne größten Eigenwert θ von B und den entsprechenden Eigenvektor x mit $\| x \| = 1$
10: $y = U_j x$
11: $r = Ay - \theta y$
12: $t = (D_A) - \theta\mathbb{1})^{-1} r$
13: $t = t - U_j U_j^T t$
14: $u^{j+1} = t / \| t \|$
15: $U_{j+1} = [U_j, u^{j+1}]$
16: **end for**

2.2 Das Jacobi-Verfahren

Carl Gustav Jacob Jacobi stellte in seiner bahnbrechenden Veröffentlichung 1845[9] nicht nur das Lösen eines symmetrischen Eigenwertproblems durch die aufeinanderfolgende Anwendung von Jacobi-Iterationen vor, sondern auch einen Ansatz, um angenäherte Eigenpaare durch ein Iterationsverfahren zu verbessern. Sleijpen und van der Vorst verallgemeinerten diesen Ansatz und fügten ihn anschließend zum heute bekannten Jacobi-Davidson Algorithmus zusammen.

Nehmen wir an, u_j sei eine Annäherung an den Eigenvektor x von A bezüglich des Eigenwertes λ. Jacobi schlug vor, u_j durch einen Vektor v, mit $u_j \perp v$, zu korrigieren, so dass

$$A(u_j + v) = \lambda(u_j + v), u_j \perp v. \tag{3}$$

[9] Siehe [Jac46].

Dies wurde von Sleijpen und van der Vorst als die so genannte *Jacobi orthogonal component correction* bezeichnet. Weil auch $v \perp u_j$ gilt, dürfen wir die Gleichung aufteilen in den Teil, der parallel zu u_j ist und in den Teil, der orthogonal zu u_j ist. Wenn $\| u_j \| = 1$, dann ist der Teil, der parallel zu u_j ist

$$u_j u_j^T A(u_j + v) = \lambda u_j u_j^T (u_j + v), \tag{4}$$

welches sich zu der skalaren Gleichung

$$\theta_j + u_j^T A v = \lambda \tag{5}$$

vereinfachen lässt, wobei θ_j der Rayleigh-Quotient von u_j ist. Der Teil, der orthogonal zu u_j ist, ist

$$(\mathbb{1} - u_j u_j^T) A(u_j + v) = \lambda(\mathbb{1} - u_j u_j^T)(u_j + v), \tag{6}$$

was äquivalent ist zu

$$\begin{aligned} (\mathbb{1} - u_j u_j^T)(A - \lambda\mathbb{1})v &= (\mathbb{1} - u_j u_j^T)(-Au_j + \lambda u_j) && (7)\\ &= -(\mathbb{1} - u_j u_j^T)Au_j = -(A - \theta_j\mathbb{1})u_j := -r. && (8) \end{aligned}$$

Dadurch, dass $(\mathbb{1} - u_j u_j^T)v = v$ ist, können wir die Gleichung schreiben als

$$(\mathbb{1} - u_j u_j^T)(A - \lambda\mathbb{1})(\mathbb{1} - u_j u_j^T)v = -r. \tag{9}$$

Wenn A nun symmetrisch ist, dann ist die Matrix aus der Gleichung auch symmetrisch. Unglücklicherweise kennt man den Eigenwert λ noch nicht und ersetzt ihn deshalb durch θ_j, um letztendlich die *Jacobi-Davidson Korrekturgleichung* zu bekommen:

$$(\mathbb{1} - u_j u_j^T)(A - \theta_j\mathbb{1})(\mathbb{1} - u_j u_j^T)v = -r = -(A - \theta\mathbb{1})u_j, v \perp u_j. \tag{10}$$

Diese Korrekturgleichung wird in der Regel mithilfe iterativer Lösungsverfahren, wie dem GMRES- oder dem MINRES-Verfahren gelöst[10].

[10] Siehe [Saa03].

Algorithmus 2 Jacobi-Davidson Algorithmus Pseudocode

1: Wähle Startvektor t und setze $U = [], V = []$
2: **for** $m = 1, 2, \ldots$ **do**
3: $t = t - UU^T t$
4: $u = t/ \parallel t \parallel; U = [U\ u]; v = Au; V = [V\ v]$
5: $C(1 : m-1, m) = U(:, 1 : m-1)^T V(:, m)$
6: $C(m, 1 : m-1) = U(:, m)^T V(:, 1 : m-1)$
7: $C(m : m) = U(:, m)^T V(:, m)$
8: Berechne den größten Eigenwert θ von C und den zugehörigen Eigenvektor s, so dass $\parallel s \parallel = 1$ ist
9: $y = Us$
10: $r = Ay - \theta y$
11: **if** $\parallel r \parallel \leq \epsilon$ **then**
12: $\lambda = \theta, x = y, STOP$
13: **end if**
14: Löse näherungsweise $(\mathbb{1} - yy^T)(A - \theta\mathbb{1})(\mathbb{1} - yy^T) = -r, y \perp y$
15: **end for**

In der Praxis sucht man oft mehr als ein Eigenpaar, wobei sich der obige Algorithmus dahingehend erweitern lässt. In den letzten Jahren wurde im DLR eine Block-Jacobi-Davidson Methode[11] entwickelt, bei der viele Rechenoperationen auf ganze Blöcke von Vektoren, auch Blockvektoren genannt, angewendet werden. Benutzt man ein entsprechend effektives Speichermodell für Matrizen, kann somit die Performance von den Matrix-Vektor-Operationen erhöht werden, da viele Matrixprodukte gleichzeitig berechnet werden ohne dass die Matrix mehrfach aus dem langsameren Hauptspeicher geladen werden muss.

2.3 Kernel

Im folgenden Kapitel werden die einzelnen Grundoperationen des Jacobi-Davidson Algorithmus kurz beschrieben. Ausgehend von der Darstellung

$$(\mathbb{1} - VV^T)(A - \theta\mathbb{1})x = y, \text{ mit } V \in \mathbb{R}^{nxk}, x, y \in \mathbb{R}^{nxk}, A \in \mathbb{R}^{nxn} \text{ dünnbesetzt}$$

beziehungsweise

$$(\mathbb{1} - VV^T)Ax = y,$$

weil in dieser Thesis der Shift nicht berücksichtigt wurde, da dieser nichts an der geladenen Datenmenge ändert und somit für die Performance im speicher-

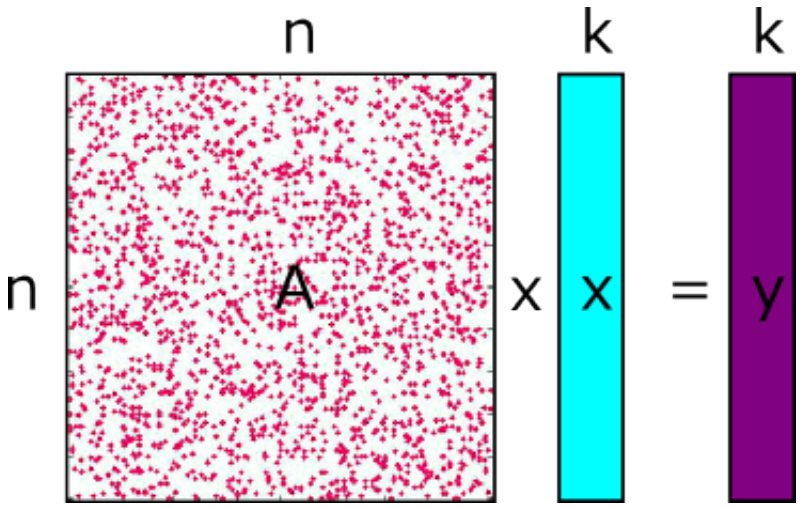
[11] Siehe [RZ14].

gebundenen Fall irrelevant ist, ergeben sich drei Matrix-Vektor-, beziehungsweise Matrix-Matrix-Multiplikationen:

- Matrix-Blockvektor-Produkt : $Ax = y$
- Matrix-Matrix-Produkt : $tmp = V^T y$
- Matrix-Matrix-Produkt : $y = y - V \cdot tmp$

2.3.1 spMMVM

Die erste Operation, das sog. spMMVM (engl.:„sparse matrix-multiple-vector multiplication“) beschreibt das Matrix-Vektor-Produkt einer dünnbesetzten Matrix mit einem Blockvektor oder Multivektor.



Wir multiplizieren also eine $n \times n$ Matrix mit einem $n \times k$ Multivektor und erhalten wieder einen $n \times k$ Blockvektor.
Ein Aufruf dieser Funktion könnte im GHOST-Framework folgendermaßen aussehen

```
ghost_spmv(y_mul,mat,x_mul,&opt);
```

wobei der Variablen &*opt* bestimmte Eigenschaften für das Matrix-Vektor Produkt mitgegeben werden.

Was die Kosten dieser Operation aus Performance-Sicht interessant macht, sind zum einen die indirekte Adressierung der Datenelemente. Auf der anderen

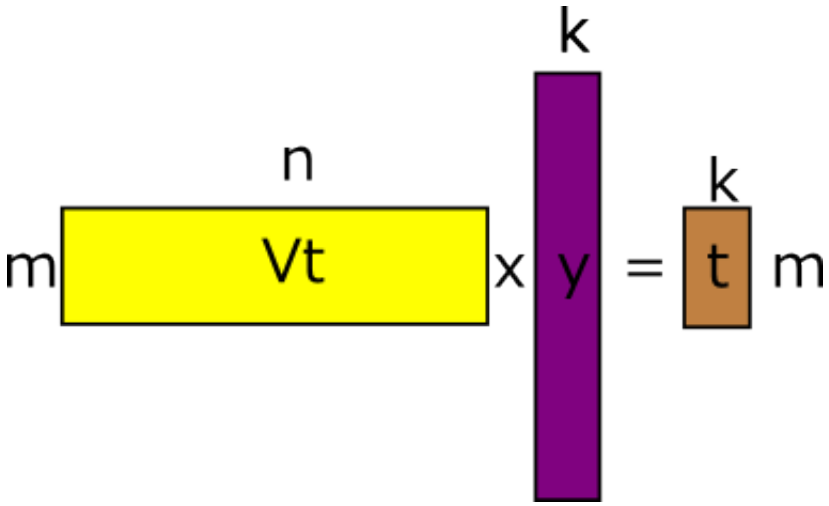
Seite liegt hier, nach dem Roofline-Modell (siehe Kapitel 3.2), eine speichergebundene Performance vor, da jedes Matrixelement höchstens k mal verwendet wird, nachdem es in den Cache geladen wurde, wobei k beim Jacobi-Davidson Algorithmus in der Regel klein ist.

2.3.2 GEMM_ALLREDUCE

Die erste Matrix-Matrix-Operation, genannt GEMM (engl. „General Matrix-Multiplication“), ist Teil eines Lineare Algebra Software Pakets (BLAS) und beschreibt zunächst eine Multiplikation einer $m \times n$ Matrix mit einer $n \times k$ Matrix, aus dem sich eine kleine vollbesetzte $m \times k$ Matrix ergibt. Die Operation berechnet

$$A = \alpha \cdot B \cdot C + \beta \cdot A,$$

wobei B und C optionalerweise transponiert werden können. BLAS dient hierbei als eine Art Interface, welches z.B. in der Intel Math Library (MKL) implementiert ist. Eine entsprechende Nvidia Implementation liefert das Software Paket cuBLAS.



Ein möglicher Aufruf, so wie er in dieser Arbeit auch implementiert wurde, sieht folgendermaßen aus

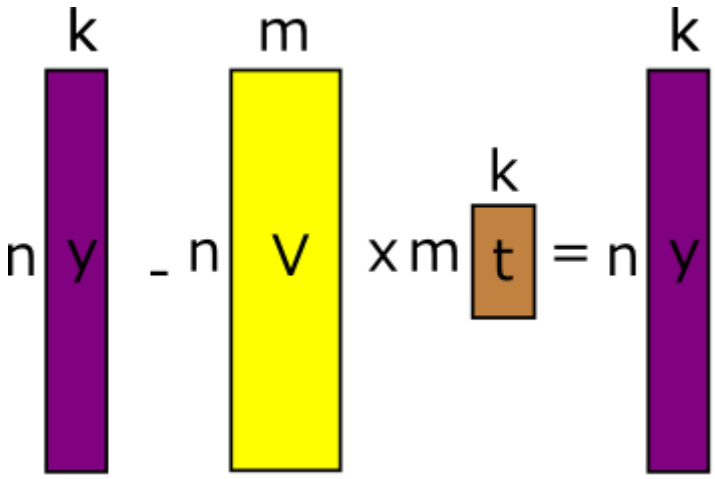
```
ghost_gemm(tmp_mul,v_mul,"T",y_mul,(char*)"N",(void*)&alpha,
(void*)&beta,GHOST_GEMM_ALL_REDUCE,GHOST_GEMM_DEFAULT);
```

Hierbei wird, wie nach dem oben angegebenen Schema $tmp = V^T y$ berechnet, wobei tmp, V^T und y Blockvektoren sind (hier gekennzeichnet mit Suffix

„_*mul*“). Die char-pointer Parameter geben an, ob der jeweilige Vektor transponiert wird („T“) oder nicht („N“). Die Referenzen zu den Parametern „&alpha“ und „&beta“ sind erst bei der kommenden Operation ausschlaggebend und werden erst dort erläutert.
Die Flag „GHOST_GEMM_ALL_REDUCE“ ist eine MPI-Flag, die das gleiche Ergebnis allen MPI-Prozessen zur Verfügung stellt und nicht nur dem Main-Thread (Prozess 0), denn dadurch, dass die Zeilen von V und y auf verschiedene MPI-Prozesse verteilt sind, müssen die lokalen Teilsummen global aufaddiert werden. Dies geschieht durch die Funktion MPI_ALLREDUCE.
Die resultierende $m \times k$ Matrix steht auf allen Prozessen zur Verfügung und benötigt daher keine weitere Kommunikation oder Synchronisation.

2.3.3 GEMM_NO_REDUCE

Die GEMM-Operation mit dem NO_REDUCE Suffix vereint nun die beiden vorherigen Teilergebnisse und stellt das Gesamtergebnis einer Iteration dar. Hierbei wird ein $n \times m$ Blockvektor zunächst mit einer kleinen $m \times k$ Matrix multipliziert und dann von einem $n \times k$ Blockvektor subtrahiert. Das Ergebnis wird in dann in einen $n \times k$ Vektor gespeichert.



Ein möglicher Aufruf, so wie er in dieser Arbeit auch implementiert wurde, sieht folgendermaßen aus:

```
ghost_gemm(y_mul,v_mul,(char*)"N",tmp_mul,(char*)"N",(void*)&malpha,
(void*)&alpha,GHOST_GEMM_NO_REDUCE,GHOST_GEMM_DEFAULT);
```

Kurz kann man den Funktionsaufruf als

$$W = alpha * V * C + beta * W$$

beschreiben. Die Parameter sind zunächst sehr ähnlich. Wir multiplizieren jedoch zwei Blockvektoren, die beide nicht transponiert sind („N“) und ziehen das Ergebnis von einem weiteren Blockvektor ab. Hierbei kommen die Parameter „&alpha“ und „&beta“ (hier: „&malpha“) zum tragen, wie man aus der Funktionsvorschrift entnehmen kann.
Die Parameter wurden also auf

```
matdt_t alpha=1., beta=0., malpha=-1.;
```

gesetzt.
matdt ist der Datentyp der Matrixeinträge und kann z.B. *single* oder *double precision*, *reell* oder *komplex* sein.

3 Performanceanalyse

In der Performanceanalyse gibt es drei Komponenten, die ausschlaggebend für die Performance einer Rechenoperation sind:

- Berechnungen
- Kommunikation innerhalb des Algorithmus
- die für die Rechenoperation benötigten Daten

Hierbei verhält sich jede Rechnerarchitektur unterschiedlich, was das Gewicht dieser Komponenten anbelangt. Zur Thematik *Performance* lässt sich für speichergebundene Operationen generell sagen: „Berechnung ist kostenlos, Kommunikation ist teuer.“
Somit versucht man die Anzahl der lokalen Operationen zu maximieren und die Kommunikation zwischen den einzelnen Komponenten (Prozessoren, Koprozessoren, Grafikkarten...) zu minimieren.

3.1 Benchmarks

Mithilfe eines Benchmarks lässt sich die so genannte Bandbreite (GB/s) eines Systems, beispielsweise einer Grafikeinheit, ermitteln. Da die Bandbreite aus

Hardware-Sicht beschreibt, wie schnell Daten über den BUS von Hauptspeicher in den Cache und wieder zurück kommen, kann man sich entweder auf die Herstellerangaben des Gerätes verlassen oder die Leistung selber messen. Um nun die Bandbreite eines Gerätes zu messen, lässt man in einer Schleife bestimmte Vektorrechenoperationen durchlaufen, misst die Zeit und kann somit die Bandbreite berechnen.

Algorithmus 3 Stream-Benchmark-Pseudocode

```
1: procedure TIMING-LOOP
2:     for each NTIMES iteration do
3:         time Copy -Operation B = A
4:         time Scale -Operation B = x · A
5:         time Add - Operation C = A + B
6:         time Triad-Operation C = A + x · B
7:     end for
8: end procedure
```

Die Operation „*Copy*“ misst hierbei zum Beispiel die Geschwindigkeit, mit der Daten geschrieben und geladen werden, wobei die Funktion „*Add*“ die Additionen pro geladenem Byte misst.

Um die Performance des Jacobi-Davidson Algorithmus zu analysieren, benötigt man die Bandbreite der Prozessoren und der Grafikkarte auf dem Rechencluster EMMY. Die Bandbreite der Prozessoren kann man aus vorherigen Arbeiten und Veröffentlichungen entnehmen, für die Grafikkarte musste jedoch ein solcher Benchmark angelegt werden. Benutzt wurde hierzu der STREAM-Benchmark von McCalpin[12]. Bei den eben genannten Benchmark-Operationen in STREAM ist die Code-Balance (bytes pro flop) sehr einfach gehalten und die Operationen sind eindeutig speichergebunden, da jedes Element nur einmal angefasst wird, sodass man genau sehen kann, wie schnell Daten geladen oder gespeichert werden.

[12]Siehe [McC95].

Die Ausgabe des Codes, sowie die resultierende Performance der NVIDIA K20M veranschaulicht die folgende Grafik.

```
hpc023@e1168:~/src/cuda-stream> ./stream
 STREAM Benchmark implementation in CUDA
 Array size (double precision) =1073.74 MB
 using 192 threads per block, 699051 blocks
0.500000
0.500000
2.500000
Function      Rate (GB/s)   Avg time(s)   Min time(s)   Max time(s)
Copy:         151.0844      0.01423345    0.01421380    0.01426196
Scale:        150.0150      0.01433725    0.01431513    0.01435900
Add:          150.5096      0.02143339    0.02140212    0.02146411
Triad:        150.5583      0.02142586    0.02139521    0.02146101
hpc023@e1168:~/src/cuda-stream>
```

3.2 Roofline Modell

Die Performance und Skalierbarkeit einer Mehrkernarchitektur kann einen Programmiereinsteiger zunächst einmal erschlagen. Um das Ganze ein wenig intuitiver und grafisch übersichtlicher zu gestalten, wurde das so genannte Roofline Modell entwickelt. Mit seiner Hilfe lassen sich realistisch die zu erwartende Performance einer Rechenoperation darstellen und gleichzeitig feststellen, ob sich ein möglicher Flaschenhals in der Performance auftut und wodurch dieser verursacht wird (Datentransfer oder Core-Execution).
Warum ist diese Erkenntnis von Bedeutung? Die Antwort liegt auf der Hand: Mit wachsenden Anforderungen an die Parallelität von Programmen, wachsen auch die Anforderungen an die parallele Performance von Algorithmen und deren Optimierung.

Als Beispiel für das Roofline Modell nehmen wir der Einfachheit halber das Skalarprodukt

$$c = x^T y \Rightarrow c = \sum_{i=1}^{n} x_i y_i$$

zweier Vektoren.

Algorithmus 4 Skalarprodukt

```
1: c=0
2: for each i = 1:n  do
3:       c = c + x(i)y(i)
4: end for
```

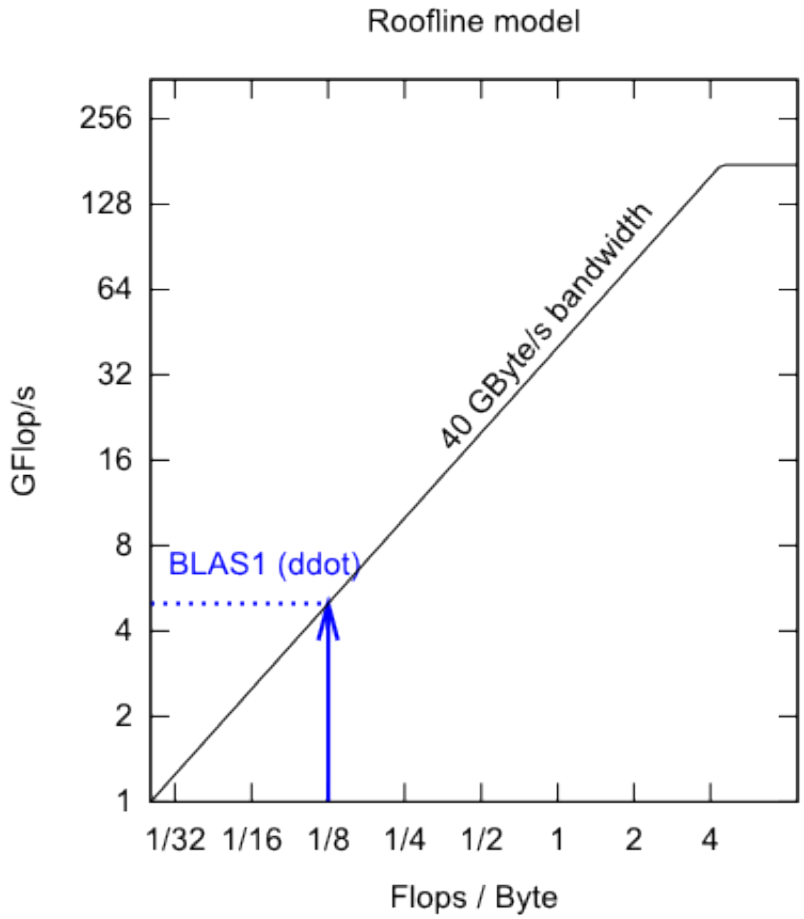
Das bedeutet, für jedes geladene x und y, führen wir eine Addition und eine Multiplikation aus. Das führt bei *double-precision* auf die folgende Code-Balance:

$$\frac{2}{8+8}\frac{\text{flops}}{\text{bytes}} = \frac{1}{8}\frac{\text{flops}}{\text{bytes}},$$

wobei mit Code-Balance das Verhältnis von arithmetischen Operationen und Datentransfers, also Laden und Speichern von Daten, beschrieben wird

$$B_c = \frac{\text{Arithmetische Operationen [flops]}}{\text{Datentransfers[Ld/St]}}.$$

Gegeben sei nun eine fiktive Rechnerumgebung mit einer Bandbreite von 40GB/s. Wenden wir nun das Roofline-Modell auf die oben besprochene Operation an, erhalten wir die folgende Darstellung:



Aus der Grafik kann man entnehmen, dass die zu erwartende Performance bei 5GFlop/s liegt und man sich in dem Bereich befindet, wo die Bandbreite die Leistung begrenzt, also: Speicherbandbreite-gebunden (engl. „bandwidth-bound“) ist. Der Bereich hinter dem „Knick“ heißt „memory-bound“ und wird durch die Taktfrequenz und die Anzahl der Prozessoren begrenzt. Ab diesem Punkt ist das Roofline Modell nicht mehr nützlich. Landet man in diesem Teil des Modells, muss man seine Operationen beispielsweise so optimieren, dass eine optimale Balance zwischen Additionen und Multiplikationen herrscht.

3.3 Bandbreite ermitteln in der Praxis

Die maximale bzw. optimale Performance eines Kernels lässt sich allgemein herleiten, indem man zunächst schaut, wieviele Bytes pro Floating-Point-Operation geladen werden. Am Beispiel des spMMVM sähe dies folgendermaßen aus:

$$\frac{\text{bytes}}{\text{flops}} = \frac{(8+4)\cdot n_{nz}\ \text{bytes} + 4\cdot n\ \text{bytes} + 2\cdot 8\ \text{bytes}\cdot n\cdot k}{2\cdot(n_{nz}+n)\cdot k}$$

Für jedes Nicht-Null-Element der Matrix n_{nz} benötigen wir in *double-precision* Arithmetik 8bytes plus 4bytes für den Matrixindex (aufgrund des Matrixformats). Zudem rechnet man mit einem Integer-Row-Pointer, der mit dem Faktor $4 \cdot n$ ebenfalls für den Matrix-Index in die Gleichung einfließt. Dazu kommen jeweils 8bytes pro Element des Vektors, mit dem die Matrix multipliziert werden muss, sowie der Zielvektor. Dadurch dass die Vektoren meist vollbesetzt sind und wir uns mit Blockvektoren beschäftigen, multiplizieren wir noch mit den Faktoren n für die Matrixgröße und k für die Blockgröße. Schließlich wird der Ausdruck durch die Anzahl der Floating-Point-Operationen geteilt. Da wir bei einem Matrix-Vektor-Produkt pro Eintrag der Matrix und pro Vektor eine Multiplikation und eine Addition haben, teilen wir unsere errechneten Bytes durch $2 \cdot$ (Nicht-Null Einträge der Matrix, Elemente des Vektors) $\cdot$ Blockgröße.

Bei den GEMM-Operationen können wir uns die Bytes pro Flop genauso herleiten.

Für die erste Operation benötigen wir 8byte pro Eintrag der Matrizen. Da die Matrizen vollbesetzt sind und wir am Ende eine kleine $m \times k$ Matrix erhalten, wobei das Speichern dieser Matrix vernachlässigbar ist, ergibt sich der Zähler

des folgenden Ausdrucks.

$$\frac{8 \cdot m \cdot n + 8 \cdot n \cdot k}{2 \cdot n \cdot m \cdot k} \frac{\text{bytes}}{\text{flops}} = \frac{8 \cdot (m + k)}{2 \cdot m \cdot k} \frac{\text{bytes}}{\text{flops}}$$

Der Nenner lässt sich dadurch erklären, dass wir pro Eintrag der Matrizen eine Addition und Multiplikation durchführen müssen.
Die zweite Operation gleicht größtenteils der ersten, der Unterschied liegt allerdings darin, dass wir am Ende keine kleine vollbesetzte Matrix erhalten, sondern einen $n \times k$ Blockvektor.
Das heißt, dass sich die Code-Balance durch

$$\frac{8 \cdot m \cdot n + 2 \cdot 8 \cdot n \cdot k}{2 \cdot n \cdot m \cdot k} \frac{\text{bytes}}{\text{flops}} = \frac{8 \cdot (m + 2 \cdot k)}{2 \cdot m \cdot k} \frac{\text{bytes}}{\text{flops}}$$

ausdrücken lässt.

In dem Modell, welches während dieser Thesis erstellt wurde, wurde mit einer Blockgröße von $k = 4$ und $m = 20$ Spalten in V gerechnet, wobei sich die Anzahl der Spalten von V im Laufe des Jacobi-Davidson Algorithmus ändert.

Aus vorherigen Arbeiten des DLRs geht hervor, dass die Bandbreite der CPUs auf dem EMMY-Rechencluster bei ungefähr 42GB/s liegt. Die reine Lese-Bandbreite ist etwas höher und liegt bei ca. 48GB/s. Das kommt vor allem den Operationen zugute, bei denen wenig Daten geschrieben werden müssen, z.B. GEMM. Der GPU-Stream-Benchmark ergab ungefähr eine Performance von 150GB/s.

Mithilfe der fogenden GHOST-Funktion

```
ghost_sparsemat_string(&matInfostr,mat);
```

lassen sich bestimmte Eigenschaften der untersuchten Matrix ausgeben:

```
------------------------------------------------------------------------
-------------------------    Sparse matrix @ rank 0 ---------------
------------------------------------------------------------------------
Data type                                :                      Double
Matrix location                          :                        Host
Total number of rows                     :                     2704156
Total number of nonzeros                 :                    35154028
Avg. nonzeros per row                    :                      13.000
Bandwidth                                :                      420077
Avg. row band                            :                  291175.919
Avg. avg. row band                       :                  147977.636
Smart row band                           :                  328625.000
Local number of rows                     :                     2704156
Local number of rows (padded)            :                     2704160
Local number of nonzeros                 :                    35154028
Full   matrix format                     :                        SELL
Local  matrix format                     :                        SELL
Local  matrix symmetry                   :                     General
Local  matrix size                 (MB) :                         405
Remote matrix format                     :                        SELL
Remote matrix size                 (MB) :                           0
Full   matrix size                 (MB) :                         405
Permuted                                 :                         Yes
Max row length (# rows)                  :                      24 (2)
Row length variance                      :                    5.739130
Row length standard deviation            :                    2.395648
Row length coefficient of variation      :                    0.184281
Chunk height (C)                         :                          32
Chunk occupancy (beta)                   :                    0.993525
Threads per row (T)                      :                           1
------------------------------------------------------------------------
```

Für die Performanceanalyse sind hier wichtig:

- Datentype *double*
- Gesamtanzahl der Zeilen n: 2704156
- Gesamtanzahl nicht-Null-Elemente nnz: 35154028
- durchsch. Anzahl nicht-Null-Elemente pro Zeile: 13

Ausgehend von den Bandbreiten und den Matrixeigenschaften der Spin24-Matrix, ergibt sich je nach Gerät für jede Operation die optimale Performance (GFlop/s):

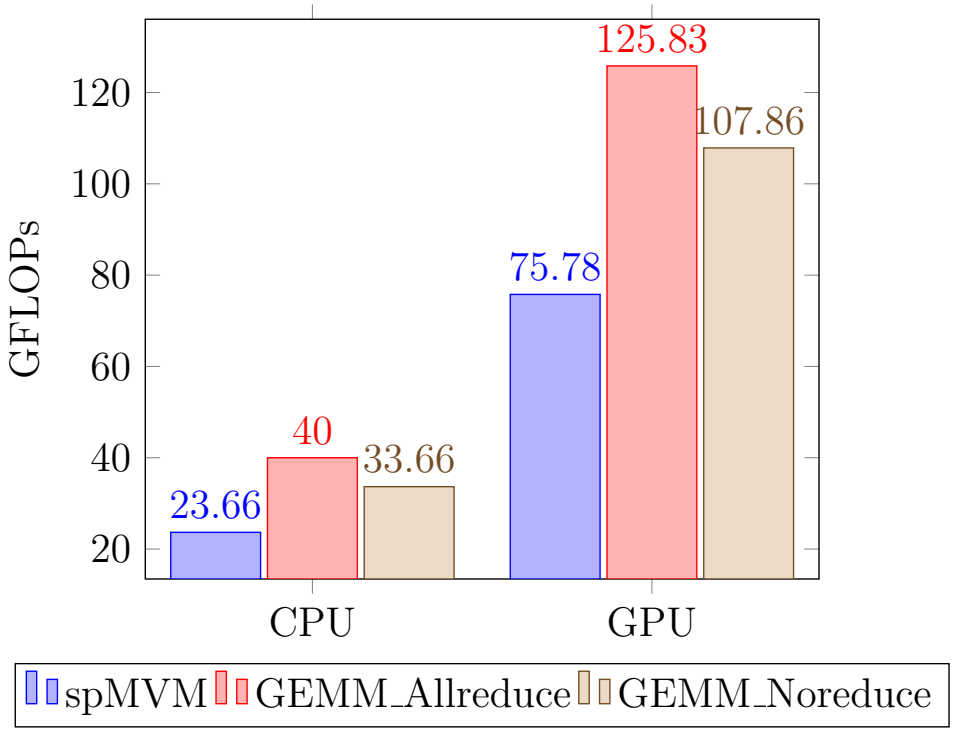


Abbildung 3: Roofline Modell: Optimale Performance

Zur Erklärung sei an dieser Stelle gesagt, dass in der Berechnung der effektiven Performance zwischen Lesebandbreite (CPU: 48GB/s) und Schreibbandbreite (CPU: 42GB/s) unterschieden werden muss, da in den Operationen unterschiedlich viele Daten geladen und geschrieben werden.
Somit werden beim spMMVM die Matrix A und der Vektor x gelesen und nur der Vektor y geschrieben. Das bedeutet in Zahlen, dass 85.8% des verursachten Traffics mit der Lesebandbreite und 14.2% mit der Schreibbandbreite berechnet werden müssen.
Beim Gemm_Allreduce werden V und y gelesen und tmp geschrieben, wobei tmp so klein ist, dass sie irrelevant für die Performance ist, sodass die gesamte Performance von der schnelleren Lesebandbreite profitiert.
Gemm_Noreduce liest V und tmp und schreibt y, sodass 85.71% der Performance über die Lesebandbreite und 14.29% über die Schreibbandbreite ermittelt werden.

In der Praxis kann man auch Tools, wie z.B. LIKWID[13] benutzen, welches direkt die erreichte Bandbreite einer Operation berechnet.

4 Ergebnisse

Führt man das geschriebene Programm mit dem folgenden Konsolenbefehl aus

```
hpc023@e1168:~/work/essex/ghost-sandbox/build-GPU/ghost testimpl> mpirun -np 1
env GHOST_TYPE=work env OMP_SCHEDULE=guided,250 ./ghost_testimpl
-m "Spin-24" -c 10 -f SELL-32-2048 -d row -t 1
```

so wird das Programm „ghost_testimp“ mit einem MPI-Prozess („-np 1“), nur auf einem Prozessor („env work“) mit der Testmatrix Spin-24 aufgerufen. Zusätzlich sollen 10 Kerne („-c 10“)und das SELL-32-2048 Matrixformat verwendet werden.
Als Ausgabe folgt nach kurzer Rechenzeit

```
spMMVM avg. time:       1.909506e-02 s
VtV    avg. time:       9.939204e-03 s
VC     avg. time:       1.222935e-02 s
-------------------------------------
total  avg. time:       4.126362e-02 s with 300 iterations.
total       time:       1.237909e+01 s.

Real Case.
spMMVM approx GFLOP/s:       1.529447e+01
VtV    approx GFLOP/s:       4.353115e+01
VC     approx GFLOP/s:       3.537923e+01
-------------------------------------
Total  approx GFLOP/s:       2.804841e+01
Everything went fine.
```

Vergleicht man nun die gemessenen Werte mit dem Roofline Modell, so stellt man fest, dass die Operationen nicht die optimale Performance erreichen.

[13]https://github.com/RRZE-HPC/likwid

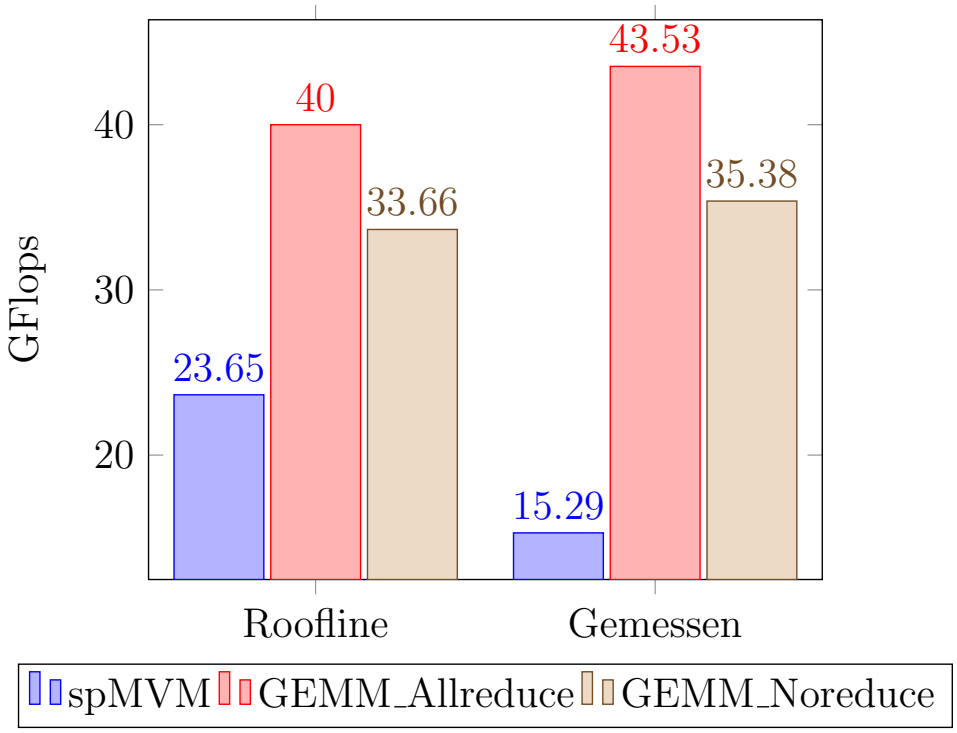


Abbildung 4: Roofline Modell verglichen mit Benchmark-Ergebnissen

Die beiden GEMM-Operationen sind optimiert für den Fall, dass B und C in etwa quadratisch sind. Wenn dem so ist, wird jedes Element der geladenen Matrizen n-mal in einer Multiplikation und Addition benutzt und es ergibt sich eine „compute bound“ Operation. Sind B und C so wie in dem betrachteten Fall „tall and skinny“, d.h. $nZeilen \gg nSpalten$, wird jedes Element nur $nSpalten$-mal benutzt und es ergibt sich ein speichergebundenes Verhalten.
Für das Matrix-Vektor-Produkt ist das Ganze ein wenig komplizierter, da auf den Vektor x in $y = A \cdot x$ unregelmäßig zugegriffen wird, weil die Matrix A dünnbesetzt ist. Dadurch funktioniert das Laden der Daten aus dem Cache nicht optimal. Es werden immer ganze Blöcke von Einträgen geladen, deren Inhalte teilweise erst später benötigt werden. Bis zum Zeitpunkt ihrer Anforderung sind die benötigten Daten vielleicht schon wieder aus dem Cache gelöscht oder überschrieben worden und der betreffende Block muss erneut geladen werden. Dies ergibt im Performance Modell einen Faktor, der in Arbeiten des DLRs in Kooperation mit dem RRZE Erlangen[14] „Ω“ genannt wird und den Overhead an geladenen Daten darstellt.

[14]Siehe [MRZ].

Vergleicht man nun die Werte, die mit einem Prozessor gemessen wurden, mit den Werten, die auf der Grafikkarte gemessen wurden, erwartet man nach dem Roofline Modell eine Beschleunigung mit Faktor 3.333.

	Roofline CPU	**Roofline GPU**	**Roofline ∅**
BW	45	150	3.333
spMVM	22.582	75.273	3.333
GEMM1	37.5	125	3.333
GEMM2	32.143	107.14	3.333

Tabelle 1: Performance von CPU und GPU(Roofline)

Zur Erklärung sei hier Folgendes gesagt: Der Stream-Benchmark schreibt und liest jeweils einen Vektor, sodass die resultierende Bandbreite in etwa $\frac{1}{2}$(Lesebandbreite + Schreibbandbreite) ist, weswegen in der Tabelle die Bandbreite der CPU auch so berechnet wurde.

In der Realität sieht dies jedoch anders aus. Führt man das Programm mit den entsprechenden Parametern erneut aus, diesmal mit dem CUDA-Setup, so ergibt sich folgendes:

```
spMMVM avg. time:       9.837143e-03 s
VtV    avg. time:       9.115945e-03 s
VC     avg. time:       1.180191e-02 s
-------------------------------------
total  avg. time:       3.075499e-02 s with 300 iterations.
total       time:       9.226498e+00 s.

Real Case.
spMMVM approx GFLOP/s:       2.968838e+01
VtV    approx GFLOP/s:       4.746243e+01
VC     approx GFLOP/s:       3.666060e+01
-------------------------------------
Total  approx GFLOP/s:       3.763222e+01
Everything went fine.
```

Die gemessenen Werte entsprechen einer Beschleunigung mit Faktor 2.02 beim spMVM und kaum nennenswerte Beschleunigung bei den GEMM Operationen mit Faktor 1.09 und 1.04. Damit erreicht das spMVM gerade einmal 41.2% der vorausgesagten Geschwindigkeit. Zu erklären ist dies dadurch, dass in der Implementation von GHOST für die GPU-GEMM-Funktion nach wie

vor cuBLAS, welches für den „compute bound“-Fall optimiert ist, aufgerufen wird, wodurch die Performance weit unter dem vorausgesagten Roofline-Modell landet.

5 Zusammenfassung

Im Laufe dieser Bachelorthesis wurde der wichtigste Baustein des Jacobi-Davidson Algorithmus im GHOST-Framework implementiert. Des weiteren wurde ein Timing-Tool eingebaut, welches die Performance auf CPU und GPU messen sollte[15].

Die Tests mit unterschiedlichen Matrixgrößen zeigten, dass die Performance auf der CPU erklärbar nahe am erstellten Roofline Modell liegt. Die Matrix-Vektor-Multiplikation funktioniert nicht optimal, weil auf die Elemente der Matrix unregelmäßig zugegriffen wird und sich ein zusätzlicher Faktor (Ω) ergibt. Die GEMM-Operationen sind anders optimiert, als wir sie eigentlich benötigen. Da in unserem Fall die beiden Matrizen, die multipliziert werden, nicht quadratisch, sondern „tall and skinny“ sind, ergibt sich ein speichergebundenes Verhalten und die tatsächliche Performance liegt über dem vorausgesagten Modell. Die geladene Datenmenge ist verhältnismäßig klein und es könnten sich noch Daten aus der vorherigen Operation im Cache befinden. Zum Beispiel berechnen wir $tmp = V^T y$ und dann $V \cdot tmp$, wobei tmp und ein Teil von V noch im Cache liegen könnte, nicht neu geladen werden müssten und somit eine höhere Performance erreicht werden könnte.

Die GPU-Tests lieferten aus Performance-Sicht ernüchternde Ergebnisse. Die gemessene Performance liegt mit knapp 41.2% unter der zu erreichenden Leistung. Grund dafür liegt in der Implementation von GHOST, da für die GPU-GEMM-Funktionen die cuBLAS-Library aufgerufen wird, die allerdings für den „compute bound“-Fall optimiert ist.

Eine sinnvolle Verteilung der Daten auf CPU und GPU sollte im Idealfall (Roofline, Tabelle 1) im Verhältnis 2:3.333 stehen, da auf einem Knoten jeweils 2 CPU Sockets und eine GPU stecken.

[15] Der erstellte Code ist im GIT-Repository https://bitbucket.org/essex/ghost-sandbox/src zu finden.

Quellenverzeichnis

[Dav75] Davidson, Ernest R.: The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. In: *Journal of Computational Physics* 17 (1975), Nr. 1, S. 87–94

[Jac46] Jacobi, Carl Gustav J.: Über ein leichtes Verfahren die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen. In: *Journal für die reine und angewandte Mathematik* 30 (1846), S. 51–94

[McC95] McCalpin, John D.: Memory Bandwidth and Machine Balance in Current High Performance Computers. In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (1995), Dezember, S. 19–25

[MRZ] Melven Röhrig-Zöllner, Moritz Kreutzer Andreas Alvermann Andreas Pieper Achim Basermann Georg Hager Gerhard Wellein Holger F. Jonas Thies T. Jonas Thies: Increasing the performance of the Jacobi-Davidson method by blocking.

[RZ14] Röhrig-Zöllner, Melven: Parallel solution of large sparse eigenproblems using a Block-Jacobi-Davidson method. (2014)

[Saa03] Saad, Yousef: *Iterative methods for sparse linear systems*. Siam, 2003

[SV00] Sleijpen, Gerard L. ; Vorst, Henk A. d.: A Jacobi–Davidson iteration method for linear eigenvalue problems. In: *SIAM Review* 42 (2000), Nr. 2, S. 267–293

Erklärung

Hiermit versichere ich, dass ich meine Abschlussarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum: .....

(Unterschrift)