Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Lösen riesiger linearer Gleichungssysteme mit einer Multi-GPU beschleunigten CG-Implementierung

Paul Arlt

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat. Dirk Pflüger
Betreuer/in:	M.Sc. Marcel Breyer
Beginn am:	1. Dezember 2022
Beendet am:	1. Juni 2023

Kurzfassung

Das Lösen von linearen Gleichungssystemen (LGS) ist ein zentrales Problem, dass es bei wissenschaftlichen Simulationen zu berechnen gilt. Zum Lösen von LGS mit positiv, symmetrischer Koeffizientenmatrix wird häufig das Verfahren der konjugierten Gradienten (engl. Conjugate Gradient (CG)) verwendet. Moderne Systeme, auf denen wissenschaftliche Simulationen durchgeführt werden, machen häufig sich die überlegene Rechenleistung von Grafikkarten zunutze. So sollte das Conjugate Gradient (CG)-Verfahren ebenfalls auf Grafikkarten gerechnet werden.

Ein weiterer Trend der heutigen Zeit ist, dass wissenschaftlicher Simulationen immer größer werden, sodass die zu lösenden LGS ebenfalls an Größe zunehmen. So reicht für große LGS der Speicher von Grafikkarten nicht mehr aus, um alle relevanten Daten zum Lösen des LGS im Speicher zu halten. Die daraus resultierenden regelmäßigen Kopiervorgänge von Daten auf die Grafikkarte verlangsamen die Berechnung erheblich. Mehrere Grafikkarten, mit ihrem kombinierten Speicher können diesem Problem entgegenwirken.

In dieser Arbeit wird das CG-Verfahren für mehrere Grafikkarten implementiert. Die Berechnungen des CG-Verfahrens werden in der Implementierung auf alle Grafikkarten verteilt. So kann die kombinierten Rechenleistung, zusätzlich zur erhöhten Speicherkapazität genutzt werden kann.

Dadurch ist das Rechnen auf mehreren Grafikkarten erheblich schneller als das Rechnen auf nur einer Grafikkarte. Im Extremfall ist die Laufzeit beim Rechnen auf vier NVIDIA A100-SXM4 über 100-mal geringer als die Laufzeit beim rechnen auf einer NVIDIA A100-SXM4. Beim Rechnen auf einer NVIDIA GTX 1080 Ti ist die Laufzeit im Extremfall über 50-mal größer, als beim Rechnen auf acht NVIDIA GTX 1080 Ti.

Inhaltsverzeichnis

1 Einleitung **17**

2 Verwandte Arbeiten **19**

3 Verfahren der konjugierten Gradienten **21**
3.1 Die Vorgehensweise des CG-Verfahrens 22
3.2 Der Algorithmus zur Vorgehensweise 25

4 CUDA **27**
4.1 CUDA-Grundlagen 27
4.1.1 Kernels 27
4.1.2 Hardwareumsetzung 29
4.1.3 Aufruf von Kernels 29
4.1.4 Speicherhierarchie 31
4.2 Asynchrone Operationen 32
4.3 Registrierter Speicher 34
4.4 Multiple Grafikkarten 34

5 Implementierung **37**
5.1 Allgemeine CG-Struktur 37
5.2 Sequenzielle Implementierung 39
5.3 Mehrkernprozessor Implementierung 40
5.4 CPU-Implementierung mit Speicherbegrenzung 41
5.5 GPU-Implementierung 42
5.5.1 Speichereinteilung des VRAM 43
5.5.2 Matrix-Vektor-Multiplikation 44
5.5.3 Vektor-Vektor-Operationen 48
5.6 Multi-GPU-Implementierung 51
5.6.1 Matrix-Vektor-Multiplikation 51
5.6.2 Vektor-Vektor-Operationen 53
5.7 Generierung der Datensätze 53

6 Ergebnisse **57**
6.1 Verwendete Hardware 57
6.2 Verwendete Compiler 58

6.3 Versuchsaufbau 59
6.3.1 Ermittlung der Blockgrößen der sequenziellen CPU-Implementierung 59
6.3.2 Ermittlung der Thread-Block-Größen der GPU-Implementierungen . . 61
6.4 Laufzeiten einer Iteration 62
6.4.1 CPU-Implementierungen 62
6.4.2 Einzel-GPU-Implementierung 65
6.4.3 Multi-GPU-Implementierung 66
6.4.4 Erreichte Leistung 68

7 Fazit und Ausblick 71

Literaturverzeichnis 73

Abbildungsverzeichnis

4.1 Hierarchische Struktur von Threads in CUDA Quelle: CUDA Programming Guide [NVI23], Kapitel 2.2 28
4.2 Überlagerung von Berechnungen mittels CUDA-Streams Quelle: Selbst erstellt 34

5.1 Speichereinteilung des VRAM einer Grafikkarte 43

6.1 Laufzeit einer Iteration der sequenziellen CPU-Implementierung in Abhängigkeit der Blockgröße, für verschiedene LGS-Ordnungen gerechnet mit dual-Socket Intel Xeon Gold 5120 Die optimale Blockgröße für große LGS beträgt vier 60
6.2 Laufzeit einer Iteration der GPU-Implementierung, in Abhängigkeit der Thread-Block-Größe, für unterschiedliche LGS-Größen Gerechnet mit einer NVIDIA GTX 1080 Ti 11GB Zu große Thread-Block-Größen hemmen die Ausführungsgeschwindigkeit des Kernels 62
6.3 Laufzeit einer Iteration der GPU-Implementierung, in Abhängigkeit der Thread-Block-Größe, für ein LGS mit Ordnung 32768 gerechnet mit einer NVIDIA GTX 1080 Ti 11GB Die optimale Thread-Block-Größe für die Kernels beträgt 256 63
6.4 Laufzeit einer Iteration der CPU-Implementierungen, in Abhängigkeit der Ordnung der LGS gerechnet mit dual-Socket Intel Xeon Gold 5120 64
6.5 Laufzeit einer Iteration der Einzel-GPU-Implementierungen, in Abhängigkeit der Ordnung der LGS blau wurde mit einer NVIDIA GTX 1080 Ti 11GB gerechnet gelb wurde mit einer NVIDIA A100-SXM4 40GB gerechnet 65
6.6 Laufzeit einer Iteration der Multi-GPU-Implementierungen, in Abhängigkeit der Ordnung der LGS blau wurde mit acht NVIDIA GTX 1080 Ti 11GB gerechnet gelb wurde mit vier NVIDIA A100-SXM4 40GB gerechnet 67

Tabellenverzeichnis

6.1 Spezifikation eines Intel Xeon Gold 5120 Prozessors Quelle: [Int] 57
6.2 Spezifikation einer NVIDIA GTX 1080 Ti Quellen: [NVIb] [NVIc] 58
6.3 Spezifikation eines AMD EPYC 7763 Prozessors Quelle: [Adv] 58
6.4 Spezifikation einer NVIDIA A100-SXM4 Quelle: [NVIa] 59
6.5 Durchschnittlich gemessene Laufzeiten der Multi-GPU-Implementierung, für LGS massiver Größe 68

Verzeichnis der Listings

4.1 Beispiel eines CUDA-Kernels zum Aufsummieren eines Arrays 29
4.2 Beispiel eines CUDA-Programms zum Aufsummieren eines Arrays 30
4.3 Beispiel eines CUDA-Programms mit Asynchronen Funktionen, Speicherregistrierung auf dem Host und freier Wahl des Devices 33

5.1 Auszug aus der sequenziellen CG-Implementierung: Explizite Berechnung des Residuums 39
5.2 Auszug aus der Mehrkernprozessor Implementierung: Explizite Berechnung des Residuums 40
5.3 Auszug aus der CPU-Implementierung mit Speicherbegrenzung: Explizite Berechnung des Residuums 41
5.4 Auszug aus der CPU-Implementierung mit Speicherbegrenzung: Herleitung der Blockgrößen 42
5.5 Auszug aus der GPU-Implementierung: Host-Funktion zum Berechnen des Matrix-Vektor-Produkts 45
5.6 Auszug aus der GPU-Implementierung: Kernel zum Berechnen des Matrix-Vektor-Produkts 47
5.7 Auszug aus der GPU-Implementierung: Host-Funktion zum Berechnen des Skalarprodukts 49
5.8 Auszug aus der GPU-Implementierung: Kernel zum Berechnen des Skalarprodukts 50
5.9 Auszug aus der GPU-Implementierung: Kernel zum Berechnen der Vektoraddition 51
5.10 Auszug aus der Multi-GPU-Implementierung: Host-Funktion zum Berechnen des Matrix-Vektor-Produkts 52
5.11 Auszug aus der Multi-GPU-Implementierung: Schleife zum Berechnen der Teilrechnungen 54

Verzeichnis der Algorithmen

3.1 Algorithmus des CG-Verfahrens 26
5.1 Allgemeine Struktur der CG-Implementierungen 38

Abkürzungsverzeichnis

API Programmierschnittstelle. 27

CG Conjugate Gradient. 3, 17, 19, 21, 57

CUDA Compute Unified Device Architecture. 27, 58

FIFO first in - first out. 32

GPU Graphics Processing Unit. 17, 19, 29, 37, 57

LGS Lineares Gleichungssystem. 17, 19, 21, 37, 60

RAM Hauptspeicher. 34, 37, 57

SM Streaming Multiprozessor. 29, 61

VRAM Grafikspeicher. 19, 31, 37, 61

1 Einleitung

Ein zentrales Problem bei wissenschaftlichen Simulationen ist das Lösen von Lineares Gleichungssystem (LGS). Zum Lösen von LGS, deren Koeffizientenmatrix A sowohl symmetrisch als auch positiv definit ist, kann das Verfahren der konjugierten Gradienten (engl. Conjugate Gradient (CG)) [She+94] verwendet werden. Im CG-Verfahren sind mehrfach Vektor-Vektor- und Matrix-Vektor-Operationen vertreten, für deren Berechnung sich Grafikkarten (engl. Graphics Processing Units (GPUs)) besonders eignen.

Wenn ein LGS auf einer GPU vom CG-Verfahren gelöst werden soll, dann müssen sich die Daten, für die Berechnung im Speicher der GPU befinden. Das schließt den Vektor b und insbesondere die Matrix A des LGS $Ax = b$ ein. In der heutigen Zeit nimmt die Größe von wissenschaftlichen Simulationen jedoch rapide zu, was vor allem an der Menge der verfügbaren Daten liegt, die von verteilten Systemen, smart Devices o.ä. erfasst werden. Deshalb steigt auch die Größe der LGS an, die es in den entsprechenden Simulationen zu lösen gilt. Infolge dessen reicht der Speicher einer GPU nicht mehr aus, um alle Daten für die Berechnung des CG-Algorithmus zeitgleich im Speicher zu halten. Deshalb müssen beim Lösen von zu großen LGS, mehrfach neue Daten auf die GPU kopiert werden, was die Laufzeit der Berechnung negativ beeinflusst.

Eine offensichtliche Lösung zum Mindern dieses Problems ist es, mehrere GPUs zu verwenden. Die kombinierte Speicherkapazität der GPUs ermöglicht es, mehr Daten auf den GPUs verteilt zu hinterlegen, sodass weniger Daten regelmäßig kopiert werden müssen. Zudem ist die Rechenleistung von mehreren GPUs um ein vielfaches größer als die einer einfachen GPUs, was die Rechengeschwindigkeit zusätzlich erhöht.

Ziel dieser Arbeit ist es deshalb, den CG-Algorithmus für mehrere Grafikkarten implementiert, sodass die Probleme, die beim Rechnen auf einzelnen GPU auftreten, zu minimieren. Die Programmierplatform CUDA [NVI23] wurde verwendet, um Berechnungen auf den GPUs auszuführen. Die Matrix-Vektor-Multiplikation ist die mit Abstand aufwändigste Rechenoperation, die bei der Ausführung des CG-Verfahrens zu berechnen ist. Deshalb wurde ein besonderer Fokus auf die effiziente Ausführung der Matrix-Vektor-Multiplikation gelegt.

Im folgenden Kapitel 2 werden verwandte Arbeiten vorgestellt, in denen ebenfalls das CG-Verfahren auf GPUs implementiert wird. Danach wird in Kapitel 3 die Theorie hinter dem CG-Verfahren vorgestellt. In Kapitel 4 wird die Programmierplatform CUDA vorgestellt, die im Rahmen dieser Arbeit verwendet wurde, um auf Grafikkarten zu rechnen. Das CG-Verfahren wurde für mehrere Umgebungen implementiert, um sich der Multi-GPU-Implementierung

anzunähern. Die einzelnen Implementierungen werden in Kapitel 5 vorgestellt. In Kapitel 6 werden die Ergebnisse vorgestellt, die im Rahmen dieser Arbeit erzielt wurden. Zuletzt wird in Kapitel 7 ein Fazit geschlossen und es wird ein Ausblick auf potentielle Erweiterungen der Implementierung gegeben.

2 Verwandte Arbeiten

Ursprünglich wurde das Verfahren der konjugierten Gradienten (engl. Conjugate Gradient (CG)) 1952 von Hestenes und Stiefel [HS+52] als eine Methode zum Lösen linearer Gleichungssysteme (LGS) vorgestellt. Als effizientes Verfahren zum Lösen von LGS ist das CG-Verfahren in mehreren CPU-Programmbibliotheken vertreten, wie z.B. in NumPy [Oli−] oder EIGEN [GJ+10].

Zahlreiche Veröffentlichungen befassen sich mit der effizienten Implementierung des CG-Verfahrens auf mehreren Graphics Processing Unit (GPU)s. Im Folgenden werden einige dieser Veröffentlichungen vorgestellt.

Cevahir et al. stellen 2009 in ihrer Arbeit [CNM09] eine Implementierung des CG-Verfahrens vor, das mehrere GPUs zum Rechnen verwendet. In ihrer Arbeit liegt der Schwerpunkt darauf mit GPUs, die nur einfache Präzision unterstützen, doppelte Präzision zu erzielen. In einer späteren Arbeit [CNM10] vertiefen Cevahir et al. ihre Forschung, indem sie die Berechnung auf mehrere Knoten verteilen, die ihrerseits mehrere GPUs zum Rechnen verwenden. Der Schwerpunkt dieser Arbeit liegt auf der Koordinierung der Knoten und deren Kommunikation untereinander.

2010 veröffentlichten Ament et al. [AKWS10] eine Implementierung des vorkonditionierten CG-Verfahrens auf mehreren GPUs. Die Implementierung überlagert die Kopiervorgänge zwischen den einzelnen GPUs mit Rechenvorgängen, um ein hohes Maß an Parallelität zu erzielen. Die Technik der überlagerten Kopiervorgänge wird in dieser Arbeit adaptiert.

In den Arbeiten von Li et al. [LL14], Müller et al. [MGSS13] und Labutin et al. [LS13] wird das CG-Verfahren jeweils mit einem Vorkonditionierer auf ein bestimmtes Problem spezialisiert. Dabei ist der jeweilige Vorkonditionerer der Schwerpunkt der Arbeit.

In allen hier vorgestellten Arbeiten werden für die Matrix des Lineares Gleichungssystem (LGS) Speicherformate verwendet, die für dünnbesetzte Matrizen geeignet sind. Zudem gehen sie davon aus, dass das gesammte LGS in den Grafikspeicher (VRAM) der GPUs kopiert werden kann. Diese Annahmen grenzen sie von dieser Arbeit ab.

3 Verfahren der konjugierten Gradienten

Das Verfahren der konjugierten Gradienten (engl. Conjugate Gradient (CG)) ist ein iteratives Verfahren zum Lösen eines Lineares Gleichungssystem (LGS) der Form $Ax = b$. Dabei ist das CG-Verfahren nur auf solche LGS anwendbar, bei denen die Koeffizientenmatrix A sowohl symmetrisch als auch positiv definit (d.h. alle Eigenwerte von A sind positiv) ist. Erstmals vorgestellt wurde das CG-Verfahren von Hestenes und Stiefel [HS+52] als eine Verfeinerung des Conjugate Direction Verfahrens. Eine modernere Publikation ist von Shewchuk et al. [She+94], in der das CG-Verfahren ausführlich hergeleitet, beschrieben und analysiert wird. Die Informationen dieses Kapitels bezüglich des CG-Verfahrens sind aus Shewchuk et al. [She+94] entnommen.

Bei exaktem Rechnen benötigt das CG-Verfahren maximal n Iterationen, wobei n der Ordnung des LGS entspricht, ähnlich wie direkte Löser. Tatsächlich benötigt das CG-Verfahren jedoch nur $m < n$ Iterationen, um eine gut approximierte Lösung zu finden. Dies ist dem Konvergenzverhalten CG-Verfahrens geschuldet. Frühe Iterationen nähern die Lösung schneller an, als spätere Iterationen. Wenn das CG-Verfahren nach $m < n$ Schritten terminiert, dann würden die letzten $n - m$ Iterationen die Lösung in so geringem Maße verbessern, dass diese Iterationen vernachlässigt werden können.

Bei iterativen Verfahren ist der aktuelle Fehler einer Iteration beschrieben durch $e_i = x_i - x$, wobei x_i die Lösung der aktuellen Iteration ist und x die tatsächliche Lösung. Für ein iteratives Verfahren kann die Konvergenz des Fehlers beschrieben werden durch $||e_i|| \leq c(f(\kappa(A)))^i ||e_0||$. Hier ist c eine Konstante, $|| \cdot ||$ die Betragsnorm und $f(\kappa(A))$ eine inhärente Funktion des Verfahrens, die von der Konditionszahl $\kappa(A) = \frac{\lambda_{max}}{\lambda_{min}}$ der Matrix A abhängt. Je kleiner $f(\kappa(A))$ ist, desto schneller konvergiert der Fehler gegen 0, ist $f(\kappa(A)) >= 1$, so konvergiert ein Verfahren nicht.

Für das CG-Verfahren gilt

$$f_{CG}(\kappa(A)) = \frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} = 1 - \frac{2}{\sqrt{\kappa(A)} + 1}$$

[She+94]. Damit konvergiert das CG-Verfahren schneller als andere iterative Verfahren, wie z.B. das Jacobi-Verfahren mit

$$f_{Jacobi}(\kappa(A)) = \frac{\kappa(A) - 1}{\kappa(A) + 1} = 2 - \frac{1}{\kappa(A)}$$

[Saa03].

Da das CG-Verfahren für kleineres $\kappa(A)$ schneller konvergiert, wird das es häufig in Kombination mit Vorkonditionierern verwendet. Das vorkonditionierte CG-Verfahren ist jedoch nicht Teil dieser Arbeit und wird an dieser Stelle nur der Vollständigkeit halber erwähnt.

Im Folgenden wird zunächst die Idee des CG-Verfahren erklärt, sowie die mathematischen Zusammenhänge bei der Umsetzung dieser Idee. Danach wird der Algorithmus erläutert, der das Vorgehen des CG-Verfahren umsetzt.

3.1 Die Vorgehensweise des CG-Verfahrens

Das CG-Verfahren löst das LGS $Ax = b$ durch das Lösen eines äquivalenten Problems. Um die Lösung des LGS zu ermittelt, minimiert das CG-Verfahren die quadratischen Funktion $F(x) = \frac{1}{2}x^TAx - x^Tb + c$. Da A positiv definit ist, gilt $x^TAx > 0$ und somit, dass das Minimum x_F von F eindeutig ist. Der Vektor x_F, der das Minimum von F ist, löst ebenso das LGS $Ax = b$. Dies wird ersichtlich durch Ableitung von F nach x:

$$\frac{\partial}{\partial x}F(x) = \frac{1}{2}\frac{\partial}{\partial x}(x^TAx) - \frac{\partial}{\partial x}(x^Tb) + \frac{\partial}{\partial x}(c) = \frac{1}{2}(Ax + A^Tx) - b \tag{3.1}$$

Da A symmetrisch ist, gilt $\frac{1}{2}(Ax + A^Tx) - b = Ax - b$, somit entspricht das Minimum der Funktion F der Lösung x des LGS $Ax = b$. Für diese Herleitung sind sowohl die Symmetrie als auch die positive Definitheit der Matrix A erforderlich.

Um das Minimum von F zu finden, beginnt das CG-Verfahren damit, mit einem initialen Suchpunkt x_0. Die Idee des CG-Verfahrens ist es, den initialen Fehler $e_0 = x_0 - x$ nach und nach aufzulösen, sodass nach m Iterationen gilt $e_m \approx 0$ und damit $x_m = x + e_m \approx x$.

Die Anzahl der vom CG-Verfahren benötigten Iteration hängt abgesehen vom LGS selbst zusätzlich noch von der Anzahl der linearen Komponenten ab, die notwendig sind, um den initialen Fehler e_0 auszugleichen. Wenn für viele der linearen Komponenten gilt $\alpha_i d_i \approx 0$, dann ist der zugehörige initiale Suchpunkt x_0 gut gewählt. Da der initale Fehler jedoch auch von der tatsächlichen Lösung x abhängt, die unbekannt ist, muss der initiale Suchpunkt x_0 im Normalfall willkürlich gewählt werden. Dazu betrachtet das CG-Verfahren den initialen Fehler als Linearkombination $e_0 = \sum_{i=0}^{n-1} \delta_i d_i$. In jeder Iteration entfernt das CG-Verfahren eine Komponente dieser Linearkombination durch das Voranschreiten des Suchpunktes $x_{i+1} = x_i + \alpha_i d_i$, mit Schrittweite $\alpha_i = -\delta_i$ und Suchrichtung d_i. Nach ausreichend Iterationen gilt $\sum_{i=0}^{m-1} \alpha_i d_i \approx -e_0$ und der Fehler wurde annähernd aufgelöst.

Nehmen wir für einen Moment an, dass die Suchrichtung d_i paarweise orthogonal sind, d.h. $d_i^T d_j = 0$ für $i \neq j$. In diesem Fall kann die zugehörige Schrittweite α_i bestimmt werden, da die zugehörige Richtung komplett aus dem folgenden Fehler e_{i+1} entfernt sein muss. Mathematisch ausgedrückt muss der verbleibende Fehler e_{i+1} nach dem Addieren der Linearkomponente $\alpha_i d_i$

orthogonal zum aktuellen Vektor d_i sein. Für dieses Vorgehen muss jedoch der Fehler $e_{i+1} = x_{i+1} - x$ und damit die tatsächliche Lösung bekannt sein. Der CG-Algorithmus verwendet aus diesem Grund nicht paarweise orthogonale, sondern A-orthogonale Suchrichtungen, d.h. $d_i^T A d_j = 0$ für $i \neq j$. In diesem Fall muss der nachfolgende Fehler e_{i+1} nur noch A-orthogonal zum aktuellen Vektor d_i sein, d.h. $d_i^T A e_{i+1} = 0$. Dabei wird ausgenutzt, dass $Ae_i = Ax_i - Ax = Ax_i - b$ gilt, was unabhängig von der tatsächlichen Lösung berechnet werden kann. Im Folgenden ist r_i das Residuum eines Suchpunkts x_i mit $r_i = b - Ax_i = -Ae_i$.

$$\begin{aligned} d_i^T A e_{i+1} &= 0 \\ d_i^T A(x_{i+1} - x) &= 0 \\ d_i^T (A(x_i + \alpha_i d_i - b) &= 0 \\ d_i^T (Ax_i - b) + \alpha_i d_i^T A d_i &= 0 \\ -d_i^T r_i + \alpha_i d_i^T A d_i &= 0 \\ \alpha_i &= \frac{d_i^T r_i}{d_i^T A d_i} \end{aligned} \tag{3.2}$$

Später in diesem Abschnitt wird noch gezeigt, dass für eine Suchrichtung gilt $d_{i+1} = r_{i+1} + \beta_{i+1} d_i$, mit einem Parameter β_i. Damit kann folgende Gleichung gezeigt werden:

$$\begin{aligned} r_{i+1}^T d_{i+1} &= r_{i+1}^T (r_{i+1} + \beta_{i+1} d_i) \\ &= r_{i+1}^T r_{i+1} + \beta_{i+1} r_{i+1}^T d_i \\ r_{i+1}^T d_{i+1} &= r_{i+1}^T r_{i+1} + 0 \end{aligned} \tag{3.3}$$

Diese Gleichung in die obige Formel für α_i eingesetzt ergibt die Formel für α_i, wie im CG-Verfahren verwendet:

$$\alpha_i = \frac{d_i^T r_i}{d_i^T A d_i} = \frac{r_i^T r_i}{d_i^T A d_i} \tag{3.4}$$

Offen steht noch, wie das CG-Verfahren die A-orthogonalen Suchrichtungen konstruiert. Die erste Suchrichtung d_0 wird mit dem Residuum $r_i = b - Ax_i$ des initialen Suchpunkts gleichgesetzt, also $d_0 = r_0 = b - Ax_0$. Alle folgenden Suchrichtungen werden durch $d_{i+1} = r_{i+1} + \beta_{i+1} d_i$ konstruiert.

Die Definition des Parameters β_i leitet sich her vom Gram-Schmidtschen Orthogonalisierungsverfahrens [Sch07]. Das Gram-Schmidt-Verfahren konstruiert aus einer Menge von N linear unabhängigen Vektoren $\{u_0, ..., u_{N-1}\}$ eine Menge von N orthogonalen Vektoren $\{o_0, ..., o_{N-1}\}$. Um einen neuen Vektor o_k zu konstruieren, werden alle Anteile in Richtung der o_i mit $i < k$ von u_k entfernt:

$$o_k = u_k - \sum_{i=0}^{k-1} \frac{o_i^T u_k}{o_i^T o_i} o_i \tag{3.5}$$

Da die u_i linear unabhängig sind, bleibt ein Vektor o_i übrig, der orthogonal zu den o_j mit $j < i$ ist.

Für das CG-Verfahren ist die A-orthogonalität der Suchrichtungen d_i notwendig, deshalb wird Gleichung (3.5) dahingehend abgewandelt:

$$d_k = u_k - \sum_{i=0}^{k-1} \frac{d_i^T A u_k}{d_i^T A d_i} d_i \tag{3.6}$$

Die Formel für die Suchrichtungen, die im CG-Verfahren verwendet wird, lässt sich aus Gleichung (3.6) herleiten. Dazu sind jedoch noch zwei Gleichungen notwendig, die nicht explizit in Shewchuk et al. [She+94] vorkommen, aber wie hier hergeleitet werden können:

$$\begin{aligned} x_{i+1} &= x_i + \alpha_i d_i \\ d_i &= \frac{x_{i+1} - x_i}{\alpha_i} \\ d_i &= \frac{(x + e_{i+1}) - (x + e_i)}{\alpha_i} \\ d_i &= \frac{e_{i+1} - e_i}{\alpha_i} \end{aligned} \tag{3.7}$$

Gleichung (3.7) wird sogleich in Gleichung (3.8) verwendet:

$$\begin{aligned} d_i^T A r_k &= (\frac{e_{i+1} - e_i}{\alpha_i})^T A r_k \\ &= \frac{1}{\alpha_i}(e_{i+1}^T A r_k - e_i^T A r_k) \\ &= \frac{1}{\alpha_i}(r_{i+1}^T r_k - r_i^T r_k) \\ d_i^T A r_k &= \frac{d_i^T A d_i}{r_i^T r_i}(r_{i+1}^T r_k - r_i^T r_k) \end{aligned} \tag{3.8}$$

Gleichung (3.6) kann nun vereinfacht werden. Als Menge von linear unabhängigen Vektoren $\{u_0, ..., u_{N-1}\}$ verwendet das CG-Verfahren gerade die Residuen $\{r_0, ..., r_{N-1}\}$.

$$\begin{aligned} d_k &= r_k - \sum_{i=0}^{k-1} \frac{d_i^T A r_k}{d_i^T A d_i} d_i \\ d_k &= r_k - \sum_{i=0}^{k-1} \frac{d_i^T A d_i}{r_i^T r_i} \frac{r_k^T r_{i+1} - r_k^T r_i}{d_i^T A d_i} d_i \\ d_k &= r_k - \sum_{i=0}^{k-1} \frac{r_k^T r_{i+1} - r_k^T r_i}{r_i^T r_i} d_i \\ d_k &= r_k - \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}} d_{k-1} \end{aligned} \tag{3.9}$$

Letzter Gleichungsschritt ist möglich, da die Residuen r_i paarweise orthogonal sind. Um das zu sehen, ist es notwendig sich klar zu machen, dass eine Schrittweite α_i immer so gewählt ist, dass der Fehler nach dem Schritt e_{i+1} A-orthogonal zur aktuellen Suchrichtung d_i ist, d.h. $d_i^T A e_{i+1} = 0$. Mit $r_{i+1} = Ae_{i+1}$, folgt daraus, dass das Residuum nach dem Schritt orthogonal zur Suchrichtung ist: $d_i^T r_{i+1} = 0$. Da d_i eine Linearkombination der r_j mit $j \leq i$ ist, ist r_{i+1} ebenfalls orthogonal zu den r_j mit $j \leq i$.

Es wurde gezeigt, dass jede Suchrichtung des CG-Verfahrens eine Linearkombination aller vergangenen Residuen ist. Da das Residuum eines Suchpunktes $r_i = x_i - x$ zum Minimum der Funktion F hinzeigt, ist das die Richtung des stärksten Gefälles und damit gerade der negative Gradient $grad\ F(x_i)$. Somit ist eine Suchrichtung ebenfalls eine Linearkombination oder Konjugation der Gradienten. Hierher stammt der Name des CG-Verfahrens.

3.2 Der Algorithmus zur Vorgehensweise

Mit den geklärten Zusammenhängen kann nun der Algorithmus zusammengesetzt werden, der das CG-Verfahren implementiert. In Algorithmus 3.1 steht dieser in Pseudocode, abgewandelt aus Shewchuk et al. [She+94]. Von Zeile zwei bis Zeile fünf wird der Algorithmus initialisiert. In Zeile sechs wird die Schleife begonnen, deren Rumpf sich von Zeile sieben bis einschließlich Zeile 16 erstreckt. Die restlichen Zeilen rahmen den Algorithmus ein und haben keine weitere Bedeutung.

In Zeile zwei wird zunächst der initiale Suchpunkt x_0 gewählt, hier als Nullvektor. Anschließend wird in Zeile drei das zugehörige Residuum r_0 berechnet und der erste Suchvektor d_0 wird in Zeile vier mit dem Residuum gleichgesetzt. Zeile fünf initialisiert den Schleifenzähler für die kommende Schleife. Die Hauptschleife wird in Zeile sechs begonnen und auf i_{max} Iterationen beschränkt. Zusätzlich endet die Schleife, wenn eine relative Toleranz ϵ unterschritten wurde. In Zeile sieben wird die Schrittweite α_i zur bereits berechnen Suchrichtung d_i bestimmt. Mit der Suchrichtung und der Schrittweite feststehend, wird in Zeile acht der neue Suchpunkt x_{i+1} berechnet. Anschließend wird das zum neuen Suchpunkt x_{i+1} gehörende Residuum r_{i+1} berechnet. In Zeile zehn wird dies, wie in den meisten Fällen implizit durchgeführt. In regelmäßigen Abständen (hier, jeder 50. Schleifendurchlauf) wir das Residuum in Zeile zwölf explizit berechnet. In Zeile 14 wird der Parameter β_{i+1} berechnet, mit dem in Zeile 15 die neue Suchrichtung d_{i+1} konstruiert wird. Abschließend wird in Zeile 16 noch der Schleifenzähler inkrementiert.

Bei exaktem Rechnen ist das implizite und das explizite Berechnen des Residuums äquivalent:

$$
\begin{aligned}
r_{i+1} &= b - Ax_{i+1} \\
r_{i+1} &= -Ae_{i+1} \\
&= -A(e_i - \alpha_i d_i) \\
&= r_i - \alpha_i A d_i
\end{aligned}
\tag{3.10}
$$

Algorithmus 3.1 Algorithmus des CG-Verfahrens

1: **procedure**
2: $x_0 \leftarrow (0, ..., 0)^T$ // willkürliche Wahl von x_0
3: $r_0 \leftarrow b - Ax_0$
4: $d_0 \leftarrow r_0$
5: $i \leftarrow 0$
6: **while** $i < i_{max}$**and**$||r_i|| > \epsilon||r_0||$ **do** // Abbruch nach maximal i_{max} Iterationen
7: $\alpha_i \leftarrow \frac{r_i^T r_i}{d_i^T A d_i}$
8: $x_{i+1} \leftarrow x_i + \alpha_i d_i$
9: **if** i is multiple of 50 **then**
10: $r_{i+1} \leftarrow b - Ax_{i+1}$ // berechne r_{i+1} explizit
11: **else**
12: $r_{i+1} \leftarrow r_i - \alpha_i A d_i$ // berechne r_{i+1} implizit
13: **end if**
14: $\beta_{i+1} \leftarrow \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$
15: $d_{i+1} \leftarrow r_{i+1} + \beta_{i+1} d_i$
16: $i \leftarrow i + 1$
17: **end while**
18: **end procedure**

Das Residuum implizit zu berechnen, ermöglicht es, das Matrix-Vektor-Produkt Ad_i, das in Zeile sieben berechnet werden muss, wiederzuverwendet. So muss in jedem Schleifendurchlauf nur eine Matrix-Vektor-Multiplikation durchgeführt werden.

Der Nachteil der impliziten Berechnungsweise ist, dass der aktuelle Suchpunkt x_{i+1} nicht explizit in der Berechnung des Residuum r_{i+1} enthalten ist. Das bedeutet, dass in der Gegenwart von Rundungsfehlern das explizit berechnete Residuum $r_{i+1} = b - Ax_{i+1}$ und das implizite berechnete Residuum $r_{i+1} = r_i - \alpha_i A d_i$ voneinander abweichen. In diesem Fall muss das Residuum regelmäßig explizit berechnet werden, um ein zu starkes Abweichen zu verhindern.

Neben der bereits erwähnten Matrix-Vektor-Multiplikation in Zeile sieben werden noch mehrere Vektor-Vektor Operationen und skalare Operationen im Schleifenrumpf durchgeführt. Damit hat eine Ausfühung der Schleife eine asymptotische Laufzeitkomplexität von $\mathcal{O}(n^2)$. Wie bereits angemerkt benötigt, benötigt dass das Verfahren $m < n$ Iterationen, um eine Lösung zu approximieren. Damit gilt für die asymptotische Laufzeitkomplexität des oben gezeigten Algorithmus $\mathcal{O}(n^3)$.

4 CUDA

In diesem Kapitel wird Compute Unified Device Architecture (CUDA) vorgestellt, die Programmierplatform und Programmierschnittstelle (API), die im Rahmen dieser Arbeit verwendet wurde, um auf Grafikkarten zu rechnen. Es werden die Aspekte von CUDA beleuchtet, die notwendig sind, um die Implementierungen aus Kapitel 5 zu verstehen. Die Informationen dieses Kapitels stammen aus dem CUDA Programming Guide [NVI23]. Die einzelnen Funktionsaufrufe sind im CUDA API Reference Guide zu finden [NVI23].

Im Folgenden werden zunächst Grundlagen von CUDA erklärt. Anschließend werden asynchrone Operationen in CUDA vorgestellt. Zuletzt wird darauf eingegangen, wie multiple Grafikkarten genutzt werden können.

4.1 CUDA-Grundlagen

CUDA erweitert eine Programmiersprache (hier C++) mit Konstrukten und Funktionen, um heterogenes Programmieren zu ermöglichen. Durch die von CUDA angebotenen Erweiterungen, können spezielle Funktionen, genannt Kernels definiert werden, die auf der Grafikkarte aufgerufen werden. Die ausführende Grafikkarte wird in CUDA $Device$ genannt, die Maschine, auf der ein CUDA-Programm gestartet wird, wird $Host$ genannt.

4.1.1 Kernels

Ein CUDA-Kernel ist eine Funktion, die von mehreren Threads nebenläufig auf der Grafikkarte ausgeführt wird. Die Ausführung eines Kernels erfolgt auf einem Gitter (engl. Grid), über das definiert wird, von wie vielen Threads ein Kernel ausgeführt wird. Ein Gitter beschreibt eine Anordnung von Thread-Blöcken und innerhalb von jedem Block eine Anordnung von Threads. Threads im selben Thread-Block teilen sich eine Speicheradresse auf dem Grafikprozessor und können über diesen effizient Daten austauschen. In komplexeren Kernels arbeiten die Threads in einem Thread-Block als Einheit zusammen.

Die Anordnung von Threads in Thread-Blöcken und von Thread-Blöcken im Gitter kann in einer, zwei oder in drei Dimensionen angegeben werden. In Abbildung 4.1 ist ein Beispiel eines Gitters mit Thread-Blöcken, die in zwei Dimensionen definiert sind. Im Beispiel ist zu sehen, wie Threads innerhalb ihres Thread-Blocks und Thread-Blöcke innerhalb des Gitters

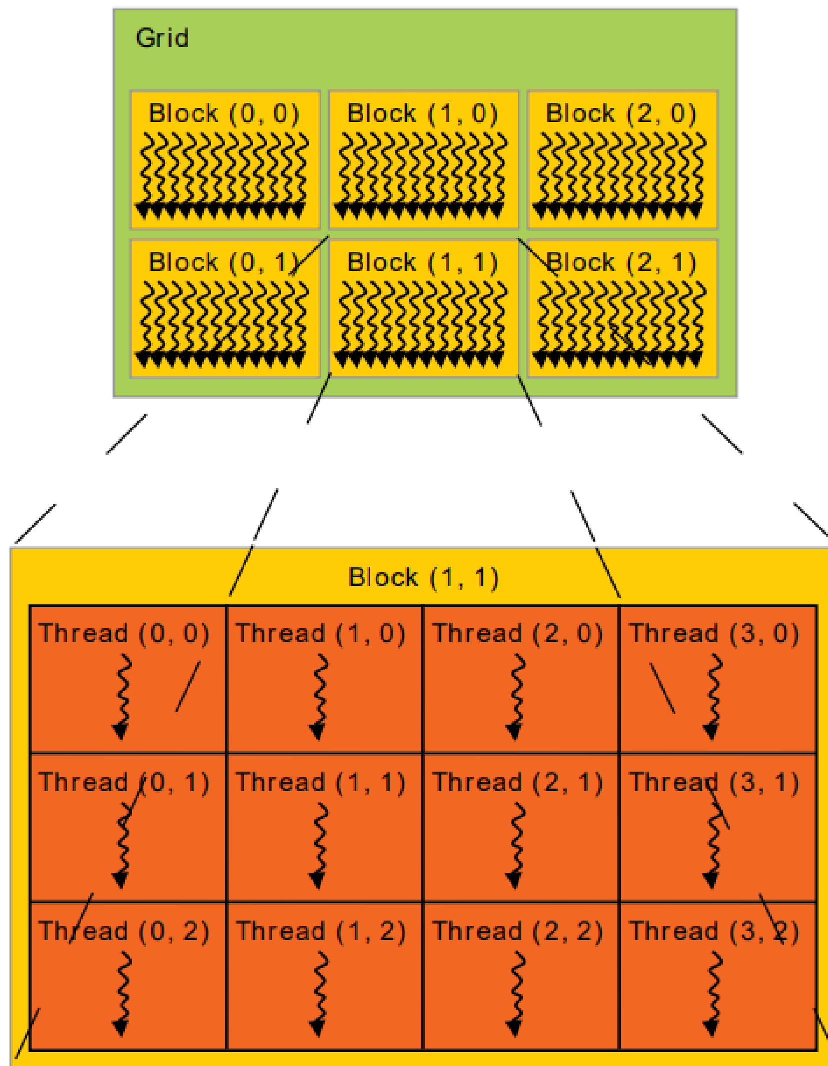

Abbildung 4.1: Hierarchische Struktur von Threads in CUDA
Quelle: CUDA Programming Guide [NVI23], Kapitel 2.2

nummeriert werden. Mit dieser Nummerierung hat jeder Thread einen globalen Index, anhand dessen er seine Berechnung ausrichtet.

In Listing 4.1 ist ein Beispiel eines Kernels zu sehen, der ein Array aufsummiert. Der Präfix `__global__` bei der Funktionsdeklaration gibt an, dass es sich bei dieser Funktion um einen Kernel handelt, der vom Host aufgerufen werden kann. Die Variable `tid` definiert den globalen Index des ausführenden Threads. Falls der Index des ausführenden Threads innerhalb der Länge des aufzusummierenden Arrays ist, so addiert der Thread den Wert an diese Stelle. Die Funktion `atomicAdd` wird dazu verwendet, wechselseitigen Ausschluss beim Addieren auf die Variable `d_sum` zu garantieren.

Listing 4.1 Beispiel eines CUDA-Kernels zum Aufsummieren eines Arrays

```
1 __global__ sum_array_kernel(double* d_arr, double* d_sum, size_t array_size) {
2     unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
3     if (tid < array_size)
4         atomicAdd(d_sum, d_arr[tid]);
5 }
```

4.1.2 Hardwareumsetzung

In diesem Kapitel wird auf die Architektur von CUDA-fähigen Grafikkarten eingegangen und wie Kernels darauf ausgeführt werden. Damit eine NVIDIA-Grafikkarte CUDA-fähig ist, muss der Grafikprozessor (engl. Graphics Processing Unit (GPU)) der Grafikkarte einer einheitlichen Architektur folgen. Wie diese Architektur im Detail implementiert ist, hängt von der Mikroarchitektur der jeweiligen Grafikkarte ab. Mikroarchitekturen werden in dieser Arbeit nicht beleuchtet.

Die GPU einer Grafikkarte, ist unterteilt in eine Reihe von Streaming Multiprozessoren (SMs). Ein SM ist seinerseits unterteilt in mehrere CUDA-Kerne. Wenn ein Kernel vom Host aufgerufen wird, so werden die Thread-Blöcke des zugehörigen Gitters auf die SMs verteilt. 32 Threads eines Thread-Blocks werden zu einer Gruppen zusammengefasst, die $Warp$ genannt werden. In jedem Berechnungsschritt eines SMs wird ein Berechnungsschritt eines gesammten Warps parellel von den CUDA-Kernen des SMs ausgeführt. Die SMs rechnen ihrerseits parallel, wodurch ein hohes Maß an Parallelität erzielt wird.

4.1.3 Aufruf von Kernels

In Listing 4.2 ist eine Funktion zu sehen, die den Kernel aus Abschnitt 4.1.1 aufruft, sodass dieser ein Array aufsummiert. Der Funktion wird eine Referenz auf das aufzusummierende Array und die Größe des Arrays übergeben, sowie einen Zeiger auf die Adresse im Speicher, an die die Summe geschrieben werden soll.

Noch bevor Daten zur Berechnung auf das Device geladen werden können, muss dort Speicher allokiert werden. In Listing 4.2 werden in Zeile drei und vier Zeiger deklariert, die auf das Device zeigen werden. Die Funktion `cudaMalloc` allokiert in Zeile sieben und acht Speicher auf dem Device und schreibt die Adresse des allokierten Speichers in den als Referenz übergebenen Zeiger.

Die Speicherstelle, auf den der Zeiger `d_sum` zeigt, wird in Zeile elf auf 0 gesetzt, damit dort das Array aufsummiert werden kann. In der darauffolgenden Zeile zwölf wird das Array auf das Device kopiert.

Um den Kernel auszuführen wird in den Zeile 15 und 16 das Gitter vorbereitet. Ein Thread-Block enthält 256 Threads, die in einer Reihe, d.h. in einer Dimension angeordnet sind. Die

Listing 4.2 Beispiel eines CUDA-Programms zum Aufsummieren eines Arrays

```
void sum_array(double* h_arr, double* h_sum, size_t array_size) {
    // Deklarierung der Zeiger auf das Device
    double* d_sum;
    double* d_arr;

    // Speicherallokierung auf dem Device
    cudaMalloc(&d_sum, sizeof(double));
    cudaMalloc(&d_arr, array_size * sizeof(double));

    // Initialisieren von Daten auf dem Device
    cudaMemset(d_sum, 0, sizeof(double));
    cudaMemcpy(d_arr, h_arr, array_size * sizeof(double), cudaMemcpyHostToDevice);

    // Kernel-Aufruf
    dim3 block_dim(256);
    dim3 grid_dim((array_size - 1) / 256 + 1);
    sum_array_kernel<<<grid_dim, block_dim>>>(d_arr, d_sum, array_size);

    // Kopieren des Ergebnisses zum Host
    cudaMemcpy(h_sum, d_sum, sizeof(double), cudaMemcpyDeviceToHost);

    // Speicherfreigabe
    cudaFree(d_sum);
    cudaFree(d_arr);
}
```

Größe des Gitters wird mittels einer ganzzahligen Division bestimmt, bei der das Ergebnis aufgerundet wird. Dadurch überdecken die Thread-Blöcke das Array vollständig. Aus diesem Grund wird innerhalb des Kernels geprüft, ob sich der Index mit dem auf das Array zugegriffen wird noch innerhalb der Grenzen des Arrays befindet. Der Kernel-Aufruf erfolgt in Zeile 17.

Nach der Berechnung des Kernels wird das Ergebnis in Zeile 20 in die Adresse von `h_sum` kopiert. Zuletzt wird in den Zeilen 23 und 24 der allokierte Speicher auf dem Device wieder freigegeben.

Kopieren einer Teilmatrix

Beim Kopieren von Teilmatrizen ist der Speicher, den es zu kopieren gilt, häufig nicht linear im Speicher angeordnet. Für diesen Fall gibt es in CUDA eine spezialisierte Funktion.

```
cudaError_t cudaMemcpy2D ( void* dst, size_t dpitch, const void* src,
    size_t spitch, size_t width, size_t height, cudaMemcpyKind kind )
```

`cudaMemcpy2D` kopiert `height` Speichersegmente der Größe `width` in Bytes. Die Adresse des ersten Segments ist `src`, die Adresse der folgenden zu kopierenden Segmente ist jeweils um

`spitch` Bytes verschoben. Analog wird das erste Segment an die Adresse `dst` kopiert, die Zieladresse der folgenden Segmente ist jeweils um `dpitch` Bytes verschoben. Damit entspricht `cudaMemcpy2D` dem mehrfachen Aufrufen von `cudaMemcpy`, wenn dabei die Ziel- und Quelladressen einheitlich verschoben sind.

4.1.4 Speicherhierarchie

Das Speichermodell in CUDA stellt unterschiedliche Arten von Speichern zur Verfügung, die einen festen Platz auf der Hardware der Grafikkarte einnehmen. Die gängigsten Speicherstellen und ihre Hierarchie werden hier vorgestellt.

Global memory ist der allgemeinste Speicher in der Hierarchie. Jeder Thread in einem Grid kann darauf zugreifen. `Global memory` befindet sich im Hauptspeicher der Grafikkarte, im Grafikspeicher (VRAM), weshalb Lese- und Schreibzugriffe darauf vergleichsweise langsam erfolgen (typischerweise hunderte von Zyklen).

Lese- und Schreibzugriffe auf `global memory` werden immer im L2-Cache der Grafikkarte zwischengespeichert und werden deshalb in Speicherbänke von 128 Bytes eingeteilt.

Shared memory ist ein Speicherbereich, auf den von allen Threads in einem Thread-Block zugegriffen werden kann. Damit Daten in diesem Bereich gespeichert werden, muss eine Variable mit dem Schlüsselwort `__shared__` annotiert werden. `Shared memory` befindet sich im L1-Cache des SMs, dem der Thread-Block zugeordnet wurde. Damit sind Lese- und Schreibzugriffe auf `shared memory` um ein vielfaches schneller als Zugriffe auf `global memory`.

Ist der Platzbedarf der annotierten Variable zur Zeit der Kompilierung bekannt, so wird der `shared memory` statisch im L1-Cache eines SMs allokiert. Wenn der Platzbedarf nicht bekannt ist, so muss die Größe beim Kernel-Aufruf (siehe Abschnitt 4.1.1) als Parameter übergeben werden, um dynamisch allokiert zu werden.

Register sind die kleinsten und schnellsten Speicherbereiche, die einer GPU zur Verfügung stehen. Die Register befinden sich, ähnlich wie `shared memory` direkt auf einem SM. Wenn ein Thread-Block einem SM zugeordnet wird, wird jedem Thread dieses Thread-Blocks eine Menge von Registern zugeteilt. Bevor eine Berechnung für einen Thread auf einem CUDA-Kern ausgeführt werden kann, müssen die Daten für diese Berechnung in ein Register des Threads geladen werden.

Variablen und Arrays, die in einem Kernel definiert werden, werden nach Möglichkeit in Registern gehalten. Wenn kein Register mehr zum Speichern von Variablen zu Verfügung steht (engl. register spilling), so werden die Daten in `local memory` ausgelagert.

Local memory ist, ähnlich wie ein Register ein Speicherbereich, der einem bestimmten Thread zur Verfügung steht. Anders als der Name es vermuten lässt, befindet sich `local memory` im VRAM, genau wie `global memory`. Zugriffe auf `local memory` sind also ähnlich langsam (typischerweise hunderte von Zyklen). `Local memory` wird immer dann verwendet, wenn die Register nicht ausreichend Speicher bieten oder wenn ein Thread-lokales Array eine dynamische Größe besitzt.

4.2 Asynchrone Operationen

Um eine einfache Berechnung auf der Grafikkarte durchzuführen müssen Daten auf das Device kopiert werden, eine Berechnung muss ausgeführt werden und das Ergebnis muss zurück auf den Host kopiert werden. In CUDA können diese drei Operationen unabhängig voneinander ausgeführt werden. Das heißt, zeitgleich zur Ausführung eines Kernels können Daten für andere Berechnungen vom und zum Device kopiert werden.

Dazu ist es notwendig, Asynchrone Operationen in Kombination mit CUDA-Streams zu verwenden. Abbildung 4.2 zeigt ein Beispiel, in dem drei Berechnungen überlagert werden können, um deren Ausführung zu beschleunigen. Grüne- und blaue Blöcke repräsentieren jeweils Kopieroperationen von Host zu Device und Device zu Host. Violette Blöcke repräsentieren Kernel aufrufe.

Ein CUDA-Stream ist eine Befehlswarteschlange für ein Device. Die Operationen in einem CUDA-Stream werden nach dem first in - first out (FIFO) Prinzip abgearbeitet. Wenn mehr als eine Warteschlange auf einem Device existieren, so können die Operationen der unterschiedlichen CUDA-Stream parallel ausgeführt werden, solange sie nicht auf den selben Speicher zugreifen und genügend Ressourcen verfügbar sind.

Listing 4.3 zeigt eine abgewandelte Funktion des Kernel-Aufrufs aus Abschnitt 4.1.3. Im hiesigen Beispiel wird in Zeile zehn ein CUDA-Stream deklariert und anschließend in Zeile elf erstellt. Die Funktionsaufrufe aus Abschnitt 4.1.3 finden sich hier in asynchronen Varianten wieder. Von Zeile 14 bis Zeile 31 werden die Funktionsaufrufe aus Abschnitt 4.1.3 in asynchronen Varianten ausgeführt.

Die asynchronen Funktionsaufrufe haben gemein, dass sie mit dem Suffix `Async` enden. Zusätzlich muss einem asynchronen Aufruf immer ein CUDA-Stream übergeben werden, sodass die Operation in diesen eingereiht werden können.

Der CUDA-Stream wird in Zeile 33 erst dann zerstört, wenn er leer ist, d.h. wenn alle Operationen ausgeführt sind. Um zu gewährleisten, dass die Summe des Arrays auf den Host kopiert wurde, wird dieser in Zeile 35 blockiert, bis alle Operationen auf dem Device fertig ausgeführt sind. Die nicht erwähnten Zeilen des Beispiels werden in den folgenden Abschnitten erklärt.

Listing 4.3 Beispiel eines CUDA-Programms mit Asynchronen Funktionen, Speicherregistrierung auf dem Host und freier Wahl des Devices

```
void sum_array(double* h_arr, double* h_sum, size_t array_size, size_t device_id) {
    cudaSetDevice(device_id);          // Auswahl des Devices

    double* d_sum;                     // Deklarierung der Zeiger auf das Device
    double* d_arr;

    // Registrieren von Speicher auf dem Host
    cudaHostRegister(h_arr, array_size, cudaHostRegisterDefault);

    cudaStream_t stream;               // Derklarieren und
    cudaStreamCreate(&stream);         // Erstellen eines CUDA-Streams

    // Asynchrone Speicherallokierung auf dem Device
    cudaMallocAsync(&d_sum, sizeof(double), stream);
    cudaMallocAsync(&d_arr, array_size * sizeof(double), stream);

    // Asynchrones Initialisieren von Daten auf dem Device
    cudaMemsetAsync(d_sum, 0, sizeof(double), stream);
    cudaMemcpyAsync(d_arr, h_arr, array_size * sizeof(double), cudaMemcpyHostToDevice,
            stream);

    dim3 block_dim(256);               // Definieren des Gitters
    dim3 grid_dim((array_size - 1) / 256 + 1);

    // Asynchroner Kernel-Aufruf
    sum_array_kernel<<<grid_dim, block_dim, 0, stream>>>(d_arr, d_sum, array_size);

    // Asynchrones Kopieren der Ergebnisse zum Host
    cudaMemcpyAsync(h_sum, d_sum, sizeof(double), cudaMemcpyDeviceToHost, stream);

    cudaFreeAsync(d_arr, stream);      // Asynchrone Speicherfreigabe
    cudaFreeAsync(d_sum, stream);

    cudaStreamDestroy(stream);         // Zerstörden der CUDA-Streams

    cudaDeviceSynchronize();           // Warten auf das Device

    cudaHostUnregister(h_arr);         // De-registrieren des Speichers auf dem Host
}
```

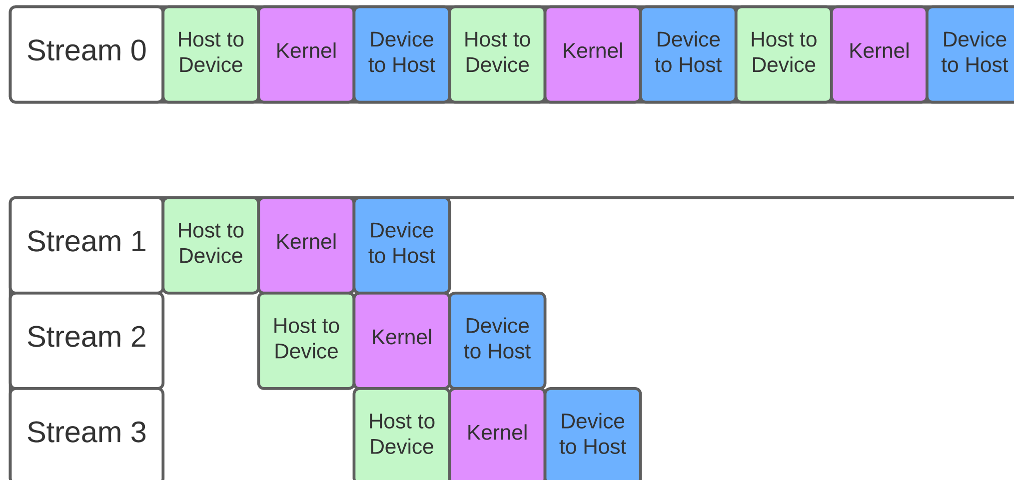

Abbildung 4.2: Überlagerung von Berechnungen mittels CUDA-Streams
Quelle: Selbst erstellt

4.3 Registrierter Speicher

Wenn Daten vom Host zum Device kopiert werden, werden diese zunächst in den Hauptspeicher (RAM) des Hosts geladen und dort in CUDA registriert. Erst nachdem die Daten registriert sind, können diese auf das Device kopiert werden. Wenn Daten an einer Speicherstelle mehrfach vom Host auf das Device kopiert werden, so ist es sinnvoll den Speicher manuell in CUDA zu registrieren. Das hat zur Folge, dass die Daten an der registrierten Speicherstelle unverzüglich auf das Device kopiert werden können.

In Listing 4.3 wird in Zeile sieben der Speicherbereich des zu kopierenden Arrays auf dem Host registriert. Nachdem das Array auf das Device kopiert wurde, wird der registrierte Speicher in Zeile 23 wieder de-registriert.

4.4 Multiple Grafikkarten

CUDA ermöglicht es, mehrere Devices gleichzeitig zu koordinieren. Die dazu notwendige Funktion wird in Listing 4.3, in Zeile zwei verwendet. Durch den Aufruf der Funktion `cudaSetDevice` werden alle nachfolgenden Operationen im Kontext des Device mit Index `device_id` ausgeführt.

Jedes Device verfügt über einen eigenen VRAM, sowie eigene SMs. Dadurch tragen mehrere Devices dazu bei, den Grad der parallelen Ausführung zusätzlich zu erhöhen. Noch wichtiger im Kontext dieser Arbeit ist jedoch, dass mehrere Devices gemeinsam über einen größeren VRAM verfügen, als ein einzelnes Device.

Wenn Funktionen oder Kernels im Kontext eines Device aufgerufen werden, muss darauf geachtet werden, dass die referenzierten Speicheradressen und CUDA-Streams ebenfalls zum Kontext des Devices gehören. Ansonsten würde ein Device dazu aufgefordert werden auf einer Speicheradresse zu operieren, auf den es keinen Zugriff hat.

5 Implementierung

Im Rahmen dieser Arbeit wurde der CG-Algorithmus in fünf Versionen implementiert. Diese Versionen sind:

1. Eine sequenzielle Implementierung des CG-Algorithmus
2. Eine Implementierung für Mehrkernprozessoren
3. Eine Implementierung, bei der jede Berechnung maximal einen vorgegebenen Speicherplatz in Anspruch nimmt
4. Eine Implementierung für eine Graphics Processing Unit (GPU)
5. Eine Implementierung für mehrere GPUs

Die Einschränkung des Speicherplatzes, die für die dritte Implementierung gilt, gilt auf ähnliche Weise für die vierte und fünfte Implementierung. Der Grafikspeicher (VRAM), der einer GPU zur Verfügung steht ist für gewöhnlich deutlich kleiner als der Hauptspeicher (RAM), der einer CPU zur Verfügung steht. Deshalb sind Implementierungen vier und fünf an den VRAM der Grafikkarte gebunden, auf der gerechnet wird.

Für die Implementierungen wurden C++ Templates verwendet, sodass sowohl mit einfacher, als auch mit doppelter Genauigkeit gerechnet werden kann. Der Datentyp `real_type` ersetzt in den Templates die Datentypen `float` und `double`.

Im Folgenden wird zunächst die Struktur des CG-Algorithmus gezeigt, die die fünf CG-Implementierungen folgen. Danach wird jede der Implementierungen in einem eigenen Abschnitt vorgestellt. Zuletzt wird kurz darauf eingegangen, wie Lineare Gleichungssysteme (LGS) zum Testen und sammeln der Laufzeiten generiert wurden.

5.1 Allgemeine CG-Struktur

In Algorithmus 5.1 ist die allgemeine Struktur des CG-Algorithmus zu sehen, an der sich jede der Implementierungen orientiert.

Algorithmus 5.1 basiert auf der CG-Variante von Shewchuk et al. aus [She+94], die bereits in Kapitel 3 vorgestellt wurde. Die für die Implementierung verwendete, angepasste Variante unterscheidet sich jedoch in Bezug auf den Toleranzparameter ϵ.

Algorithmus 5.1 Allgemeine Struktur der CG-Implementierungen

```
1: procedure CG-ALGORITHMUS(A, b, x, ϵ, i_max)
2:     x ← (0, ..., 0)^T
3:     r ← b                                   // Eigentlich r ← b − Ax, aber Ax = 0
4:     r_r ← r^T r                             // Berechne r^T r immer mit r
5:     d ← r
6:     i ← 0
7:     while i < i_max do                      // Abbruch nach maximal i_max Iterationen
8:         A_d ← Ad                            // Berechnen von Ad zum Beginn der Schleife
9:         α ← r_r / (d^T A_d)
10:        x ← x + αd
11:        β ← r_r                             // Speichern des alten Wertes von r_r
12:        if i % 32 ≠ 0 then
13:            r ← r − αA_d                    // implizites Berechnen von r
14:            r_r ← r^T r
15:        end if
16:        if i % 32 = 0 or √r_r < ϵ then
17:            r ← b − Ax                      // explizites Berechnen von r
18:            r_r ← r^T r
19:            if √r_r < ϵ then
20:                break while
21:            end if
22:        end if
23:        β ← r_r / β                         // β enthält hier den alten Wert von r_r
24:        d ← r + βd
25:        i ← i + 1
26:    end while
27: end procedure
```

In der Beispielimplementierung aus Shewchuk et al. [She+94] wurde sich für eine relativen Toleranzparameter ϵ entschieden, der mit der Norm des initialen Residuums $||r_0||$ multipliziert wird. Da das initiale Residuum r_0 vom initialen Fehler e_0 und damit vom willkürlich gewählten initialen Suchpunkt x_0 abhängt, ist die resultierende Toleranz ebenso willkürlich. Aus diesem Grund wird in dieser Arbeit ein absoluter Toleranzparameter verwendet.

Ein weiterer Unterschied ist, dass Algorithmus 5.1 nicht direkt terminiert, wenn das implizite Residuum die Toleranz ϵ unterschreitet. In diesem Fall wird, anders als in Algorithmus 3.1, zunächst das explizite Residuum berechnet und danach erneut geprüft, ob die Toleranz ϵ unterschritten wurde. Wie in Zeile 19 in Algorithmus 5.1 zu sehen, terminiert der Algortihmus erst, nachdem diese Bedingung erfüllt ist.

In Algorithmus 5.1 sind skalare Operationen, Vektor-Vektor-Operationen, sowie Matrix-Vektor-Multiplikationen vertreten. Die skalaren Operationen der Zeilen 6, 7, 11, 12, 16, 19, 23 und

Listing 5.1 Auszug aus der sequenziellen CG-Implementierung:
Explizite Berechnung des Residuums

```
// calculate actual residual (r = b - Ax)
std::memset(residual_r.data(), 0, order * sizeof(real_type));
for (size_t ii = 0; ii < order; ii += MATRIX_BLOCKING_SIZE) {
    for (size_t jj = 0; jj < order; jj += MATRIX_BLOCKING_SIZE) {
        for (size_t i = 0; i < MATRIX_BLOCKING_SIZE && ii + i < order; i++) {
            real_type Ax_i = 0.0;
            for (size_t j = 0; j < MATRIX_BLOCKING_SIZE && jj + j < order; j++) {
                Ax_i += matrix_A[(ii + i) * order + jj + j] * search_point_x[jj + j];
            }
            residual_r[ii + i] += Ax_i;
        }
    }
}
product_r_r = 0.0;
for (size_t i = 0; i < order; i++) {
    residual_r[i] = vector_b[i] - residual_r[i];
    product_r_r += residual_r[i] * residual_r[i];
}
```

25 werden in allen Implementierungen direkt auf der CPU ausgeführt. Die einzelnen Implementierungen unterscheiden sich in ihrer Umsetzung der Matrix-Vektor-Multiplikationen in Zeilen acht und 17, sowie der Vektor-Vektor-Operationen der übrigen Zeilen.

5.2 Sequenzielle Implementierung

Die erste Implementierung des CG-Verfahrens ist sequenziell. Die Matrix-Vektor-Multiplikationen und Vektor-Vektor-Operationen aus Algorithmus 5.1 werden auf einem Einzelkernprozessor (engl. Single-Core CPU) ausgeführt, zusammen mit den skalaren Operationen.

Listing 5.1 zeigt einen Auszug aus der sequenziellen CG-Implementierung, in dem das Residuum mit der Formel $r = b - Ax$ explizit berechnet wird. Von Zeile zwei bis 13 wird das Matrix-Vektor-Produkt Ax ausgeführt und im Vektor `residual_r` gespeichert. Anschließend wird von Zeile 14 bis 18 das Residuum $r = b - Ax$ und dessen quadratische Norm $r^T r$ berechnet.

In der sequenziellen Implementierung werden die Matrix-Vektor-Multiplikation block-weise ausgeführt. In Listing 5.1 wird initial, in Zeile zwei der Vektor `residual_r` auf 0 gesetzt. Die beiden äußeren Schleifen in Zeile drei und vier iterieren über die Teilmatrizen der Matrix `matrix_A`. Im Inneren wird von Zeile fünf bis Zeile elf eine Matrix-Vektor-Multiplikation auf den Teilmatrizen ausgeführt und zum Ergebnis addiert.

Unterschiedliche Vektor-Vektor-Operationen werden in der sequenziellen Implementierung nach Möglichkeit in eine Schleife zusammengefasst. In Listing 5.1 wird in von Zeile 15 bis

Listing 5.2 Auszug aus der Mehrkernprozessor Implementierung:
Explizite Berechnung des Residuums

```
1  product_r_r = 0.0;
2  // calculate actual residual (r = b - Ax)
3  #pragma omp parallel for reduction(+ : product_r_r)
4  for (size_t i = 0; i < order; i++) {
5      real_type Ax_i = 0.0;
6      for (size_t j = 0; j < order; j++) {
7          Ax_i += matrix_A[i * order + j] * search_point_x[j];
8      }
9      residual_r[i] = vector_b[i] - Ax_i;
10     product_r_r += residual_r[i] * residual_r[i];
11 }
```

Zeile 18, in der selben Schleife, in der das Residuum berechnet wird, dessen quadratische Norm $r^T r$ berechnet.

5.3 Mehrkernprozessor Implementierung

In diesem Abschnitt wird die Implementierung für Mehrkernprozessoren vorgestellt. Um die zusätzlichen Kerne solcher Prozessoren auszureizen, verwendet diese CG-Implementierung mehrere Threads. OpenMP [Ope15] wird verwendet, um Threads zu erzeugen und zu koordinieren.

Für eine Schleife wurde die Compileranweisung `#pragma omp parallel for` verwendet, um für diese Schleife mehrere Threads zu starten und die Berechnung innerhalb der Schleife auf diese Threads aufzuteilen.

Falls in der parallel ausgeführten Schleife eine Summe berechnet wird, z.B. beim Skalarprodukt zweier Vektoren, so wird die Klausel `reduction (+ : var)` hinzugefügt, um die Teilergebnisse der einzelnen Threads effizient in der Variable `var` zu summieren.

In dieser Implementierung konnten die Matrix-Vektor-Multiplikationen nicht durch blockweises Berechnen beschleunigt werden. Die Matrix-Vektor-Multiplikationen werden Zeilenweise auf die Threads unterteilt, sodass jeder Thread eine Menge von Skalarprodukten berechnet. Zudem, da kein Cache effizientes Berechnen mehr praktiziert wird, werden alle Arten von Schleifen zusammengefasst, wenn dies möglich ist. In Listing 5.2 wird das Residuum explizit berechnet. Hier werden Schleifen für die Berechnung des Matrix-Vektor-Produkts Ax und des Residuums $r = b - Ax$ zusammengefasst.

Listing 5.3 Auszug aus der CPU-Implementierung mit Speicherbegrenzung: Explizite Berechnung des Residuums

```
1  product_r_r = 0.0;
2  // calculate actual residual (r = b - Ax)
3  for (size_t ii = 0; ii < order; ii += matrix_block_size) {
4      const size_t block_size_i = std::min(order - ii, matrix_block_size);
5
6      std::memset(&residual_r[ii], 0, block_size_i * sizeof(real_type));
7      for (size_t jj = 0; jj < order; jj += matrix_block_size) {
8          #pragma omp parallel for
9          for (size_t i = 0; i < block_size_i; i++) {
10             real_type Ax_i = 0.0;
11             for (size_t j = 0; j < matrix_block_size && jj + j < order; j++) {
12                 Ax_i += matrix_A[(ii + i) * order + jj + j] * search_point_x[jj + j];
13             }
14             residual_r[ii + i] += Ax_i;
15         }
16     }
17 }
18 for (size_t ii = 0; ii < order; ii += space_for_rt / 2) {
19     const size_t block_size = std::min(order - ii, space_for_rt / 2);
20
21     #pragma omp parallel for reduction(+ : product_r_r)
22     for (size_t i = 0; i < block_size; i++) {
23         residual_r[ii + i] = vector_b[ii + i] - residual_r[ii + i];
24         product_r_r += residual_r[ii + i] * residual_r[ii + i];
25     }
26 }
```

5.4 CPU-Implementierung mit Speicherbegrenzung

In dieser Implementierung ist darauf zu achten, dass jede parallele Berechnung maximal einen vorgegebenen Speicherplatz in Anspruch nimmt. Um das zu bewerkstelligen werden Matrix-Vektor-Multiplikationen und Vektor-Vektor-Operationen in Blöcken berechnet.

Listing 5.3 zeigt einen Auszug aus der Implementierung, in dem das Residuum des aktuellen Suchpunktes explizit berechnet wird. Von Zeile drei bis 17 wird das Produkt Ax berechnet. Die beiden äußeren Schleifen in Zeile drei und sechs iterieren über die Teilmatrizen der Matrix `matrix_A`. Für jede Teilmatrix wird von Zeile acht bis 15 eine Matrix-Vektor-Multiplikationen parallel ausgefüht.

Mit dem Produkt Ax wird von Zeile 18 bis 26 das Residuum $r = b - Ax$ und das Produkt $||r||^2$ berechnet. Hier wird die parallele Berechnung in den Zeilen 21 bis 25 ebenfalls mehrfach ausgefüht.

Die Größen, die zum Unterteilen der Schleifen verwendet werden, sind `matrix_block_size` und `space_for_rt`. In Listing 5.4 ist zu sehen, wie die Größen zum Unterteilen der Schleifen

Listing 5.4 Auszug aus der CPU-Implementierung mit Speicherbegrenzung: Herleitung der Blockgrößen

```
1 const size_t space_for_rt = l3_cache_size / sizeof(real_type);
2 const size_t cache_lines_matrix = (sqrt(space_for_rt) - 1) / l3_cache_line_size;
3 const size_t matrix_block_size = cache_lines_matrix * l3_cache_line_size;
```

berechnet werden. Der Speicherplatz, den die parallelen Berechnungen nicht überschreiten dürfen ist der L3-Cache des Prozessors.

Die Größe `space_for_rt` sagt aus, wie viele Variablen vom Typ `real_type` in den L3-Cache passen. Daran kann direkt die Blockgröße beim Rechnen mit Vektoren hergeleitet werden. In Listing 5.3, in Zeile 21 bis 25, sind zwei Vektoren Bestandteil der parallelen Berechnung. Deshalb wird in Zeile 19 die Blockgröße so gewählt, dass zwei Subvektoren den L3-Cache nicht überschreiten. Hier wird darauf spekuliert, dass die Anzahl der Cache-Zeilen im L3-Cache durch zwei teilbar ist, sodass der Cache auf die beiden Subvektoren aufgeteilt werden kann.

Die Größe `matrix_block_size` entspricht der Quadratwurzel der Elemente, die im Cache Platz finden, minus eins, abgerundet auf das größte Vielfache einer L3-Cache-Zeilengröße.

Die Idee dahinter ist, dass für eine Matrix-Vektor-Multiplikation n^2 Elemente für die Matrix, sowie $2n$ Elemente für den Vektor Platz im Speicher benötigen. Die Elemente einer Teilmatrix sind im RAM nicht komplett linear angeordnet, sondern nur zeilenweise linear. Da eine Cache-Zeile nur linear angeordneten Speicher zwischenspeichern kann, muss der Parameter `matrix_block_size` abschließend auf das größte Vielfache einer L3-Cache-Zeilengröße abgerundet werden.

5.5 GPU-Implementierung

Bei der GPU-Implementierung des CG-Verfahrens werden alle Vektor-Vektor- und Matrix-Vektor-Operationen auf der GPU berechnet. Für große LGS reicht der VRAM einer Grafikkarte nicht aus, um die Daten für alle Berechnungen zeitgleich zu halten. So benötigt z.B. allein die Matrix eines LGS der Ordnung 75000 über 41GB Speicherplatz. Die im Rahmen dieser Arbeit verwendete Server-Grafikkarte NVIDIA A100-SXM4 [NVIa] besitzt einen VRAM von 40GB und kann damit die Matrix nicht im Speicher halten. Aus diesem Grund müssen während der Ausführung von Algorithmus 5.1 mehrfach Daten zum und vom Device kopiert werden.

Vor Beginn der Ausführung muss Speicher auf dem Device allokiert werden, sodass die Daten kopiert werden können. Zusätzlich, um das Kopieren der Daten effizienter zu gestalten, werden sämtliche zu kopierenden Daten im Speicher des Hosts registriert (siehe Abschnitt 4.3).

Im Folgenden wird zunächst gezeigt, wie der VRAM auf der Grafikkarte eingeteilt wird. Anschließend wird im Detail erklärt, wie die Matrix-Vektor-Multiplikation auf dem Device

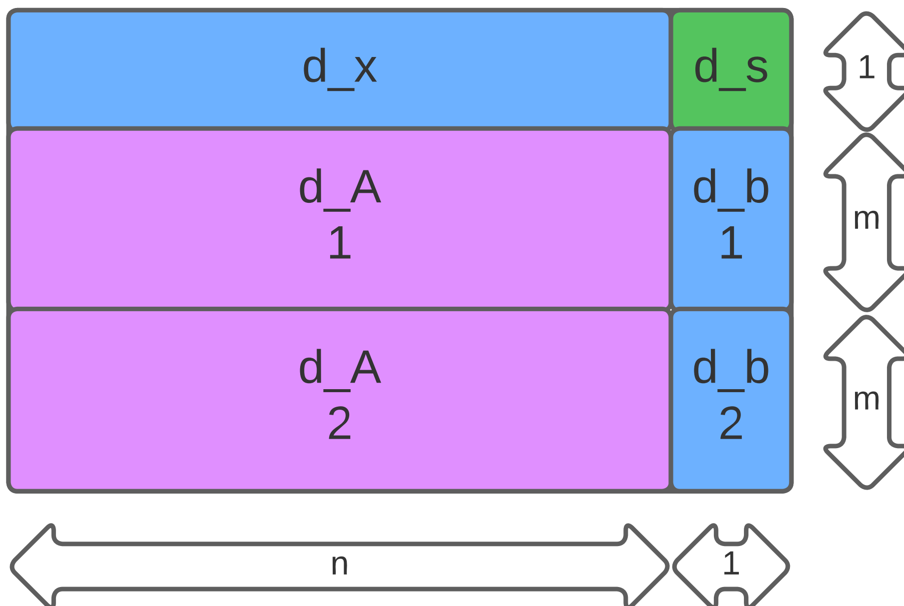

Abbildung 5.1: Speichereinteilung des VRAM einer Grafikkarte

ausgeführt wird. Zuletzt wird darauf eingegangen, wie Vektor-Vektor-Operationen auf dem Device ausgeführt werden.

5.5.1 Speichereinteilung des VRAM

In Abbildung 5.1 ist zu sehen, wie der VRAM der Grafikkarte eingeteilt wird. Der grüne Block `d_s` repräsentiert einen Speicherbereich für einen einzelnen Wert, ein Skalar. Die Speicherbereiche, die von den blauen Blöcken repräsentiert werden, sind für Vektoren unterschiedlicher Länge vorgesehen. `d_x` bietet dabei Platz für einen Vektor, dessen Länge n der Ordnung des zu lösenden LGS entspricht. In den Speicherbereichen `d_b` 1 und 2 finden jeweils Vektoren der Länge m Platz. Die violetten Blöcke `d_A` 1 und 2 repräsentieren jeweils eine Speicherbereiche für eine $(m$ x $n)$-Matrix.

Anhand von Abbildung 5.1 ist leicht zu erkennen, dass folgende Ungleichung für den VRAM der Grafikkarte gelten muss:

$$VRAM \geq (2m + 1)(n + 1)e$$

Hier ist die Konstante e ist der Speicherbedarf eines Elements. Für doppelte Präzision gilt $e = 8$ Bytes. In der GPU-Implementierung wurde die Größe m wie folgt bestimmt:

$$mb = \frac{(0.95 \cdot VRAM) - (n + 1)e}{2(n + 1)}$$

$$m = 128 \cdot \left\lfloor \frac{mb}{128} \right\rfloor \cdot \frac{1}{e}$$

In der Berechnung werden nur 95% des VRAMs der Grafikkarte verwendet. 5% des VRAM werden nicht allokiert, sodass darin der Code der Kernels geladen werden kann. Würde der Speicher vollständig allokiert werden, so könnten im Anschluss keine Kernels ausgeführt werden [NVI23]. Um auf den Wert m zu schließen wird der Wert mb auf das größte vielfache von 128 abgerundet.

5.5.2 Matrix-Vektor-Multiplikation

Die Berechnung des Matrix-Vektor-Produkts $b = Ax$ erfolgt über mehrere Teilrechnungen $b_i = A_i x$. Hier wird A_i, eine $(m$ x $n)$-Teilmatrix mit dem vollständigen Eingabevektor x multipliziert. Ergebnis der Teilrechnung ist ein Teil des Ergebnisvektors b_i mit Länge m.

Für eine Teilrechnung wird zunächst die $(m$ x $n)$-Teilmatrix in einen der Speicherbereiche `d_A` 1 oder 2 kopiert. Anschließend wird der entsprechende Speicherbereich des Ergebnisvektors `d_b` 1 oder 2 auf 0 gesetzt. Nun wird der Kernel aufgerufen, der die Matrix-Vektor-Multiplikation ausführt. Im letzten Schritt wird das Ergebnis auf den Host kopiert.

Listing 5.5 zeigt die Host-Funktion, der die Matrix-Vektor-Multiplikation koordiniert. Im Folgenden wird erklärt, welche Optimierungen auf Host-Seite verwendet werden und wie sich diese auf den Code des Hosts auswirken.

Überlagerung der Teilrechnungen Die zeitaufwändigste Operation einer Teilrechnung ist das Kopieren der $(m$ x $n)$-Teilmatrix auf das Device. Für eine optimale Auslastung der Hardware werden die einzelnen Teilrechnungen so überlagert, dass ununterbrochen neue Teilmatrizen auf das Device kopiert werden können. Während mit der Teilmatrix A_i in einem der Speicherbereiche gerechnet wird, wird die nächste Teilmatrix A_{i+1} auf das Device kopiert. Aus diesem Grund sind die Speicherbereiche für Teilmatrizen und Teilergebnisse doppelt im VRAM vorhanden. Analog zu den Speicherstellen werden zwei CUDA-Streams für das Device auf dem Host erstellt.

Listing 5.5 Auszug aus der GPU-Implementierung:
Host-Funktion zum Berechnen des Matrix-Vektor-Produkts

```
void gpu_cg_re<real_type>::matrix_vector_product(const real_type* matrix,
        const real_type* vector_in, real_type* vector_out) {
    const dim3 thread_block_dim(THREAD_BLOCK_SIZE, 1);

    // which submatrices are precopied and at which memory-space
    size_t stream_index = copied_submatrix_stream;
    size_t chunk_shift = copied_submatrix_index;
    cudaMemcpy(d_x, vector_in, order * sizeof(real_type), cudaMemcpyHostToDevice);
    for (size_t chunk_num = 0; chunk_num < num_chunks; chunk_num++) {
        const size_t chunk_offset = ((chunk_num + chunk_shift) % num_chunks) * chunk_size;
        const size_t this_chunk_size = std::min(order - chunk_offset, chunk_size);
        const dim3 grid_dim((order - 1) / THREAD_BLOCK_SIZE + 1,
                (this_chunk_size - 1) / THREAD_BLOCK_SIZE + 1);

        // first (NUM_CUDA_STREAMS) submatrices are precopied
        if (chunk_num >= NUM_CUDA_STREAMS) {
            cudaMemcpy2DAsync(d_A[stream_index], chunk_size * sizeof(real_type),
                    &matrix[chunk_offset], order * sizeof(real_type),
                    this_chunk_size * sizeof(real_type), order,
                    cudaMemcpyHostToDevice, streams[stream_index]);
            copied_submatrix_stream = (copied_submatrix_stream + 1) % NUM_CUDA_STREAMS;
            copied_submatrix_index = (copied_submatrix_index + 1) % num_chunks;
        }
        cudaMemsetAsync(d_b[stream_index], 0, this_chunk_size * sizeof(real_type),
                streams[stream_index]);
        d_matrix_vector_product_col<<<grid_dim, thread_block_dim,
                0, streams[stream_index]>>>(d_A[stream_index], d_x, d_b[stream_index],
                chunk_size, order, this_chunk_size);
        cudaMemcpyAsync(&vector_out[chunk_offset], d_b[stream_index],
                this_chunk_size * sizeof(real_type), cudaMemcpyDeviceToHost,
                streams[stream_index]);

        stream_index = (stream_index + 1) % NUM_CUDA_STREAMS;
    }

    cudaDeviceSynchronize();
}
```

In Listing 5.5 entspricht jeder Schleifendurchlauf einer Teilrechnung. Alle Operationen der Teilrechnung werden asynchron ausgeführt und in den selben CUDA-Stream eingereiht. Der selbe Index, der zum Anwählen des CUDA-Streams verwendet wird, wird auch verwendet, um die Speicherstellen `d_A` und `d_b` auf dem Device zu referenzieren.

Am Ende der Schleife, in Zeile 29 wird der `stream_index` angepasst. Dadurch wird die nächste Teilrechnung in den anderen CUDA-Stream eingereiht wird und verwendet die anderen Speicherstellen `d_A` und `d_b` auf dem Device. So kommt jedem CUDA-Streams ein eigener Speicherbereich zum Verwalten zu.

Kopieren von transponierten Teilmatrizen Beim Rechnen mit Matrizen auf der GPU ist es für effizientere Speicherzugriffe sinnvoll, dass eine Matrix spaltenweise im Speicher des Devices hinterlegt ist. Um eine Teilmatrix A_i spaltenweise vom Host auf das Device zu kopieren, wird die Symmetrie der Matrix A ausgenutzt.

Anstatt eine $(m$ x $n)$-Teilmatrix A_i zu kopieren, wird ihr transponiertes A_i' auf das Device kopiert. Da A_i' zeilenweise im Speicher hinterlegt ist, entspricht das der spaltenweisen Darstellung von A_i.

Das Kopieren der Teilmatrix wird in Listing 5.5 von Zeile 16 bis 19 in Auftrag gegeben. Der Funktion `cudaMemcpy2dAsync` wird hier die transponierte $(n$ x $m)$-Teilmatrix A_i' referenziert.

Wiederverwendung der kopierten Teilmatrizen Die in Algorithmus 5.1 vertretenen Matrix-Vektor-Multiplikationen erfolgen allesamt mit der selben Matrix A. Nachdem eine Matrix-Vektor-Multiplikation vollständig auf dem Device ausgeführt wurde, verbleiben die zuletzt kopierten Teilmatrizen noch dort. Die nächste Matrix-Vektor-Multiplikation wird mit den noch auf dem Device befindlichen Teilmatrizen begonnen, sodass initial keine Teilmatrix kopiert werden muss.

Vor Ausführung der ersten Matrix-Vektor-Multiplikation müssen dazu die ersten beiden Teilmatrizen auf das Device kopiert werden. In Listing 5.5 befinden sich zwei Teilmatrizen A_k und A_{k+1} bereits auf dem Device. Der kleinere Index k der beiden Teilmatrizen wird in der Variable `copied_submatrix_index` hinterlegt. Die Variable `copied_submatrix_stream` gibt an, an welchem Speicherplatz sich Teilmatrix mit dem kleineren Index befindet.

In Zeile fünf wird initial der CUDA-Stream ausgewählt, in dessen Speicherbereich sich die Teilmatrix mit Index k befindet. Der Index der folgenden Teilberechnungen wird um den Wert von k verschoben. In Zeile neun wird der verschobene Offset entsprechend berechnet.

Das Kopieren einer neuen Teilmatrix in Zeile 16 bis 19 wird für die ersten beiden Schleifendurchläufe ausgesetzt, da sich zwei Teilmatrizen bereits auf dem Device befinden. Wird jedoch eine neue Teilmatrix auf das Device kopiert, so werden die Variablen `copied_submatrix_index` und `copied_submatrix_stream` dementsprechend angepasst.

Listing 5.6 Auszug aus der GPU-Implementierung:
Kernel zum Berechnen des Matrix-Vektor-Produkts

```
1  __global__ void d_matrix_vector_product(
2          const real_type* d_A, const real_type* d_x, real_type* d_b,
3          const kernel_index_type pitch,
4          const kernel_index_type width,
5          const kernel_index_type height) {
6      const kernel_index_type block_offset_x = blockIdx.x * THREAD_BLOCK_SIZE;
7      const kernel_index_type block_offset_y = blockIdx.y * THREAD_BLOCK_SIZE;
8
9      __shared__ real_type local_x[THREAD_BLOCK_SIZE];
10     if (block_offset_x + threadIdx.x < width) {     // boundary check
11         local_x[threadIdx.x] = d_x[block_offset_x + threadIdx.x];
12     }
13     __syncthreads();
14
15     if (block_offset_y + threadIdx.x < height) {    // boundary check
16         real_type b_i = 0.0;
17         for (kernel_index_type i = 0; i < THREAD_BLOCK_SIZE; i++) {
18             if (block_offset_x + i < width) {       // boundary check
19                 b_i += d_A[(block_offset_x + i) * pitch + block_offset_y + threadIdx.x] *
20                         local_x[i];
21             }
22         }
23         atomicAdd(&d_b[block_offset_y + threadIdx.x], b_i);
24     }
25 }
```

Kernel der Matrix-Vektor-Multiplikation

In Listing 5.6 ist der Kernel zu sehen, der in der GPU-Implementierung zum Berechnen des Matrix-Vektor-Produkts verwendet wird. Seine Funktionsweise wird im Folgenden erklärt.

Dem Kernel wird die Speicheradresse der Matrix A, des Eingabevektors x und des Ergebnisvektors b mit den Zeigern `d_A`, `d_x` und `d_b` übergeben. Die Matrix A ist dabei spaltenweise im Speicher hinterlegt.

Der Übergabeparameter `pitch` gibt den Abstand zweier aufeinanderfolgender Spalten der Matrix A an. Mithilfe von `pitch` werden die Elemente in A adressiert. Die Parameter `width` und `height` geben die Anzahl der zu multiplizierenden Spalten und Zeilen der Matrix an. Mit ihnen wird mehrfach geprüft, ob sich ein Index noch innerhalb der Grenzen des zulässigen Speichers befindet (engl. boundary checking). Die Grenzprüfungen werden im Text nicht erneut erwähnt, sind aber im Code kommentiert.

Innerhalb eines Thread-Blocks führen t linear angeordnete Threads gemeinsam eine Matrix-Vektor-Multiplikation auf einer $(t \text{ x } t)$-Teilmatrix aus. Auf die Größe t hat ein Thread Zugriff

über die Konstante THREAD_BLOCK_SIZE. Die Zuständigkeit des Thread-Blocks wird in den Zeilen sechs und sieben bestimmt.

Caching Für die Ausführung eines Thread-Blocks wird der relevante Teil des Eingabevektors x in shared memory zwischengespeichert. Da die Latenz von shared memory um ein vielfaches geringer ist, als die Latenz von global memory, kann dieses Vorgehen als Caching beschrieben werden.

Ein shared memory Array wird in Zeile neun deklariert, in das in Zeile elf der für den Thread-Block relevante Teil des Eingabevektors x gespeichert wird. Anschließend werden die Threads eines Thread-Blocks in Zeile 13 mit der Anweisung __syncthreads() synchronisiert. So wird garantiert, dass das shared memory Array vollständig befüllt wurde, bevor mit dessen Einträgen gerechnet wird.

Verschmolzener Speicherzugriff (Coalesced Memory Access) Von Zeile 15 bis 24 erfolgt die eigentliche Matrix-Vektor-Multiplikation. Die for-Schleifen von Zeile 17 bis 22 summiert die Produkte des zwischengespeicherten Eingabevektors und einer Teilzeile der Matrix in der lokalen Variable b_i auf. In Zeile 23 wird diese Variable mit der Funktion atomicAdd atomar zum Ergebnisvektor addiert.

Die Indizierung der Matrix in Zeile 19 erfolgt so, dass die Threads in einem Warp im Thread-Block auf Einträge zugreifen, die linear im Speicher angeordnet sind. Mit dieser Indizierung werden mehrere Anfragen auf den global memory zu einem einzelnen Speicherzugriff verschmolzen (coalesced memory access). Zusätzlich wurde der Speicher bei der Allokierung auf 128 Bytes ausgerichtet, sodass bei doppelter Präzision ein Warp nur zwei Speicherzugriffe benötigt.

5.5.3 Vektor-Vektor-Operationen

Die in Algorithmus 5.1 vertretenen Vektor-Vektor-Operationen sind das Skalarprodukt und die Vektoraddition. Im Fall der Vektoraddition werden die Vektoren zuvor mit jeweils einem Skalar multipliziert.

Eine Zusätzliche Host-Funktion wurde geschrieben, die einen Vektor quadriert. In dieser wird nur ein Vektor auf das Device kopiert. Dann wird der Kernel des Skalarprodukts verwendet, um der Vektor mit sich selbst zu multiplizieren.

Die in einer Vektor-Vektor-Operation vertretenen Vektoren werden in $\lceil \frac{n}{m} \rceil$ Teilvektoren der Länge m unterteilt und paarweise auf das Device kopiert. Dafür werden die Speicherplätze d_b 1 und 2, sowie d_x verwendet. Speicherplatz d_x wird intern in zwei Speicherplätze der Länge m unterteilt, um als Gegenstück für die Speicherplätze d_b 1 und 2 zu dienen.

Listing 5.7 Auszug aus der GPU-Implementierung:
Host-Funktion zum Berechnen des Skalarprodukts

```
void gpu_cg<real_type>::dot_product
        (const real_type* vector_1, const real_type* vector_2, real_type* sum) {
    size_t stream_index = 0;
    cudaMemsetAsync(d_sum, 0, sizeof(real_type), streams[stream_index]);
    for (size_t chunk_offset = 0; chunk_offset < order; chunk_offset += chunk_size) {
        const size_t this_chunk_size = std::min(order - chunk_offset, chunk_size);
        const dim3 grid_dim((this_chunk_size - 1) / THREAD_BLOCK_SIZE + 1);

        cudaMemcpyAsync(&d_x[stream_index * chunk_size], &vector_1[chunk_offset],
                this_chunk_size * sizeof(real_type),
                cudaMemcpyHostToDevice, streams[stream_index]);
        cudaMemcpyAsync(d_b[stream_index], &vector_2[chunk_offset],
                this_chunk_size * sizeof(real_type),
                cudaMemcpyHostToDevice, streams[stream_index]);
        d_dot_product<<<grid_dim, THREAD_BLOCK_SIZE, 0, streams[stream_index]>>>
                (&d_x[stream_index * chunk_size], d_b[stream_index], d_sum,
                this_chunk_size);

        stream_index = (stream_index + 1) % NUM_CUDA_STREAMS;
    }

    cudaMemcpy(sum, d_sum, sizeof(real_type), cudaMemcpyDeviceToHost);
}
```

Die Skalare, die in der Vektoraddition verwendet werden, werden als Werte übergeben und müssen nicht explizit an eine Speicherstelle im VRAM der Grafikkarte kopiert werden. Bei der Vektoraddition wird der zweite Vektor vom Ergebnis überschrieben, sodass kein zusätzlicher Speicherplatz für einen Vektor benötigt wird. Das Ergebnis des Skalarprodukts wird im Speicherplatz `d_s` aufsummiert.

In Listing 5.7 ist die Host-Funktion zu sehen, die die Berechnung des Skalarprodukts koordiniert.

In Zeile vier wird die Speicherplatz `d_sum`, in der der das Erbenis des Skalarprodukts aufsummiert werden soll auf 0 gesetzt.

In jedem Schleifendurchlauf der folgenden `for`-Schleife wird eine Teilrechnung des Skalarprodukts mit zwei Vektoren der Länge m berechnet. Dazu werden die Kopieroperationen und der Kernel-Aufruf asynchron in den selben `cuda`-Stream eingereiht. Der selbe Index, der zum Anwählen des CUDA-Streams verwendet wird, wird auch verwendet, um die Speicherstellen `d_A` und `d_b` auf dem Device zu referenzieren. So kommt jedem CUDA-Streams ein eigener Speicherbereich zum Verwalten zu.

Listing 5.8 Auszug aus der GPU-Implementierung:
Kernel zum Berechnen des Skalarprodukts

```
__global__ void d_dot_product(const real_type* d_x, const real_type* d_b,
        real_type* d_sum, const kernel_index_type this_chunk_size) {
    kernel_index_type i = blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ real_type reduction_array[THREAD_BLOCK_SIZE];
    real_type temp = 0.0;
    if (i < this_chunk_size) {
        temp = d_x[i] * d_b[i];
    }
    reduction_array[threadIdx.x] = temp;
    __syncthreads();

    // reduce b_i
    for (kernel_index_type s = THREAD_BLOCK_SIZE / 2; s > 0; s >>= 1) {
        if (threadIdx.x < s) {
            reduction_array[threadIdx.x] += reduction_array[threadIdx.x + s];
        }
        __syncthreads();
    }
    if (threadIdx.x == 0) {
        atomicAdd(d_sum, reduction_array[0]);
    }
}
```

Vor Beginn des nächsten Schleifendurchlaufs wird in Zeile 19 der `stream_index` angepasst. Die Anweisung `cudaMemcpy` wird synchron ausgeführt. Das Ergebnis wird erst dann auf den Host kopiert, wenn alle CUDA-Streams leer sind und alle Berechnungen beendet wurden.

Die Host-Funktion für die Vektoraddition funktioniert analog. Unterschied ist, dass ein anderer Kernel aufgerufen wird, der anstatt dem Zeiger `d_sum` die Skalare `alpha` und `beta` entgegennimmt und die Vektoraddition ausführt. Zusätzlich wird nach jedem Kernel-Aufruf der zweite, vom Ergebniss überschriebene Vektor auf das Device zurückkopiert. Die Anweisung `cudaMemcpy` nach der Schleife wird durch `cudaDeviceSynchronize` ersetzt, die den Host und das Device explizit synchronisieren.

Vektor-Vektor Kernel

In Listing 5.8 ist der Kernel zu sehen, der das Skalarprodukt auf dem Device berechnet. Erneut arbeiten t linear angeordnete Threads in einem Thread-Block zusammen.

In Zeile drei wird damit begonnen, den globalen Thread-Index zu bestimmen und in der Variable `i` zu speichern. Die Threads in einem Thread-Block teilen sich ein `shared memory` Array, das in Zeile fünf deklariert wird. Das Array wird von Zeile sechs bis zehn befüllt. Wenn

Listing 5.9 Auszug aus der GPU-Implementierung:
Kernel zum Berechnen der Vektoraddition

```
__global__ void d_vector_add(const real_type* d_a, real_type* d_b,
        const real_type alpha, const real_type beta,
        const kernel_index_type this_chunk_size) {
    kernel_index_type i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < this_chunk_size)
        d_b[i] = alpha * d_a[i] + beta * d_b[i];
}
```

der globale Index des Threads innerhalb der Grenzen der referenzierten Vektoren ist, so werden die Elemente der beiden Vektoren an der entsprechenden Stelle multipliziert und in das Array eingefügt. Ist der globale Index des Threads außerhalb der Grenzen, wird 0 eingefügt.

Von Zeile 14 bis Zeile 19 werden die Einträge des `shared memory` Arrays mittels Reduktion aufsummiert. Abschließend addiert ein Thread des Thread-Blocks das Teilergebnis des Thread-Blocks zum Gesamtergebnis `d_sum`.

In Listing 5.9 ist der Kernel der Vektoraddition zu sehen. In ihm führt ein Thread mit globalem Index i die Berechnung $b[i] = \alpha \cdot a[i] + \beta \cdot b[i]$ aus, wenn sein Index in den Grenzen der Vektorn ist.

5.6 Multi-GPU-Implementierung

In der Multi-GPU-Implementierung werden die Vektor-Vektor- und Matrix-Vektor-Operationen von mehreren GPUs verteilt ausgeführt. In dieser Arbeit wird von identischen GPUs ausgegangen. Die Vektor-Vektor- und Matrix-Vektor-Operationen werden so unterteilt, dass diese gleichmäßig auf die Grafikkarten aufgeteilt werden können.

Die Kernels aus der GPU-Implementierung werden hier wiederverwendet. Ebenso wird die Speichereinteilung der einzelnen Devices beibehalten.

Unterschied ist, dass auf dem Host für jedes Device dessen Speicherplätze und CUDA-Streams gespeichert und koordiniert werden müssen.

5.6.1 Matrix-Vektor-Multiplikation

Die Host-Funktion, die in der Multi-GPU-Implementierung zur Ausführung der Matrix-Vektor-Multiplikation verwendet wird ist in Listing 5.10 und Listing 5.11 zu sehen.

In Listing 5.10 wird der vollständige Vektor x in der Schleife von Zeile sechs bis Zeile elf asynchron auf jedes Device kopiert. Dazu wird der erste CUDA-Stream jedes Devices verwendet.

Listing 5.10 Auszug aus der Multi-GPU-Implementierung:
Host-Funktion zum Berechnen des Matrix-Vektor-Produkts

```
1  void mgpu_cg_re<real_type>::matrix_vector_product
2          (const real_type* matrix, const real_type* vector_in, real_type* vector_out) {
3      const dim3 thread_block_dim(THREAD_BLOCK_SIZE, 1);
4
5      // copy x to all devices
6      for (size_t device_index = 0; device_index < num_devices; device_index++) {
7          cudaSetDevice(device_index);
8
9          cudaMemcpyAsync(d_x[device_index], vector_in, order * sizeof(real_type),
10                 cudaMemcpyHostToDevice, streams[device_index * NUM_CUDA_STREAMS]);
11     }
12     for (size_t device_index = 0; device_index < num_devices; device_index++) {
13         cudaSetDevice(device_index);
14
15         cudaDeviceSynchronize();
16     }
17
18     // Schleife zum Berechnen der Teilrechnungen, vgl. Listing 5.11
19     [...]
20
21     for (size_t device_index = 0; device_index < num_devices; device_index++) {
22         cudaSetDevice(device_index);
23
24         cudaDeviceSynchronize();
25     }
26 }
```

In der darauffolgenden Schleife wird darauf gewartet, dass der Kopiervorgang auf jedem Device abgeschlossen ist.

Dann wird die Schleife aus Listing 5.11 ausgeführt, die die Teilrechnungen auf den Devices verteilt durchführt. Zuletzt wartet der Host darauf, dass die Operationen aller Devices vollständig ausgeführt wurden.

Die Multi-GPU-Implementierung verwendet die selben Optimierungen für die Matrix-Vektor-Multiplikation wie die GPU-Implementierung Abschnitt 5.5.2. Diese finden sich ebenso in der Schleife in Listing 5.11 wieder.

Die äußere Schleife in Listing 5.11 iteriert über die Anzahl an Teilrechnungen (chunks), die von einem der Device ausgeführt werden soll. Durch vorkopierte Matrizen entsteht auch hier eine Indexverschiebung, der zu berechnenden Teilmatrizen. Diese wird mit der Variable `general_chunk_offset` in den Zeilen neun und zehn berücksichtigt. Die innere Schleife iteriert über alle Devices, die dem Host zur Verfügung stehen.

In Zeile 13 wird das aktuelle Device angewählt, sodass die nachfolgenden Operationen im Kontext dieses Devices erfolgen. Die in den Operationen referenzierten Speicherplätze und CUDA-Streams gehören ebenfalls zum Kontext des angewählten Devices.

In Zeile 15 wird der Anfang der aktuellen Teilmatrix berechnet. Nach einer Grenzüberprüfung in Zeile 17, wird die Matrix-Vektor-Multiplikation in Auftrag gegeben. Dabei werden die Speicherplätze und CUDA-Streams mit der Variable `global_stream_index` angewählt. Die Größe der aktuellen Teilmatrix wird in den Zeilen 19 und 20 bestimmt.

Erneut wird das Kopieren von neuen Teilmatrix von Zeile 26 bis 29 für die ersten beiden äußeren Schleifendurchläufe ausgesetzt, da sich zwei Teilmatrizen bereits auf den Devices befinden. Werden jedoch neue Teilmatrizen auf die Devices kopiert, so werden die Variablen `copied_submatrix_index` und `copied_submatrix_stream` einmal für alle Devices angepasst.

5.6.2 Vektor-Vektor-Operationen

Die Vektoren der Vektor-Vektor-Operationen werden ebenfalls auf mehrere Devices verteilt. Ähnlich wie in Listing 5.11 wird in einer äußeren Schleife über die Anzahl der Teilrechnungen iteriert, die von einem Device ausgeführt werden soll. In einer inneren Schleife werden dann zunächst ein Device angewählt und anschließend werden die Operationen in einen CUDA-Stream des Devices eingereiht.

Am Ende der Berechnung wartet der Host explizit mit der Anweisung `cudaDeviceSynchronize` auf jedes der Devices. Im Fall des Skalarprodukts wird noch das Teilergebnis aller Devices auf den Host kopiert und dort aufsummiert.

5.7 Generierung der Datensätze

Zum Generieren von LGS mit symmetrisch, positiv definiter Koeffizientenmatrix wurde ein Python-Skript geschrieben. Eingabeparameter des Skripts sind die Ordnung o des zu generierenden LGS sowie zwei Werte min und max, die einen Wertebereich für den Vektor b festlegen.

Das Python-Skript verwendet die Funktion `make_spd _matrix` aus der `scikit`-Bibliothek [PVG+11], um eine symmetrisch, positiv definite Matrix A der spezifizierten Ordnung zu erzeugen. Anschließend wird ein Vektor b der Länge o mit Zufallszahlen im spezifizierten Wertebereich generiert.

Die Funktion `make_spd _matrix` benötigte jedoch bereits mehrere Stunden zum Generieren von Matrizen der Ordnung $o = 2^{14}$. Für die eine Ordnung von $o = 2^{15}$ warf die Funktion eine Fehlermeldung und brach die Ausführung ab. Die Maschine, auf der das Skript ausgeführt

Listing 5.11 Auszug aus der Multi-GPU-Implementierung:
Schleife zum Berechnen der Teilrechnungen

```
// which submatrices are precopied and at which memory-space
size_t stream_index = copied_submatrix_stream;
size_t chunk_shift = copied_submatrix_index;
for (size_t chunk_num = 0; chunk_num < chunks_per_device; chunk_num++) {
    const size_t general_chunk_offset =
            ((chunk_num + chunk_shift) % chunks_per_device) * chunk_size;

    for (size_t device_index = 0; device_index < num_devices; device_index++) {
        cudaSetDevice(device_index);

        const size_t chunk_offset =
                general_chunk_offset + device_index * rows_per_device;

        if (chunk_offset < order) {
            const size_t global_stream_index =
                    device_index * NUM_CUDA_STREAMS + stream_index;
            const size_t device_limit =
                    std::min((device_index + 1) * rows_per_device, order);
            const size_t this_chunk_size =
                    std::min(device_limit - chunk_offset, chunk_size);
            const dim3 grid_dim((order - 1) / THREAD_BLOCK_SIZE + 1,
                    (this_chunk_size - 1) / THREAD_BLOCK_SIZE + 1);

            // first (NUM_CUDA_STREAMS) submatrices are precopied
            if (chunk_num >= NUM_CUDA_STREAMS) {
                cudaMemcpy2DAsync(d_A[global_stream_index],
                        chunk_size * sizeof(real_type), &matrix[chunk_offset],
                        order * sizeof(real_type),
                        this_chunk_size * sizeof(real_type), order,
                        cudaMemcpyHostToDevice, streams[global_stream_index]);
                if (device_index == 0) {
                    copied_submatrix_stream =
                            (copied_submatrix_stream + 1) % NUM_CUDA_STREAMS;
                    copied_submatrix_index =
                            (copied_submatrix_index + 1) % chunks_per_device;
                }
            }
            cudaMemsetAsync(d_b[global_stream_index], 0,
                    this_chunk_size * sizeof(real_type),
                    streams[global_stream_index]);
            d_matrix_vector_product_col<<<grid_dim, thread_block_dim, 0,
                    streams[global_stream_index]>>>(d_A[global_stream_index],
                    d_x[device_index], d_b[global_stream_index],
                    chunk_size, order, this_chunk_size);
            cudaMemcpyAsync(&vector_out[chunk_offset], d_b[global_stream_index],
                    this_chunk_size * sizeof(real_type),
                    cudaMemcpyDeviceToHost, streams[global_stream_index]);
        }
    }
    stream_index = (stream_index + 1) % NUM_CUDA_STREAMS;
}
```

wurde, verfügt über einen RAM von 755GB. Der Speicherbedarf einer Matrix mit Ornung 2^{15} entspricht nur 8GB.

Deshalb wurden zum Generieren noch größeren LGS, eine C++ Funktion geschrieben. Übergabeparameter der Funktion sind die Ordnung o des zu generierenden LGS und zwei Werte min und max. Die C++ Funktion generiert zwei Vektoren a und b mit zufälligen Werten im Bereich $[min, max)$. Der Vektor a wird dann in die Haupt-diagonale der Matrix A eingesetzt, sodass diese, zusammen mit dem Vektor b ein zulässiges LGS bildet, dass in einer CG-Iteration gelöst werden kann.

Die so generierten LGS bestehen aus einer Diagonalmatrix und einem Vektor, die beide mit Zufälligen Zahlen gefüllt sind. Ein so generiertes LGS wird in dieser Arbeit verwendet, um die Laufzeit einer Iteration der CG-Implementierungen zu messen. Für diesen Anwendungsfall ist der Inhalt der so generierten LGS unbedeutend.

6 Ergebnisse

In diesem Kapitel wird vorgestellt, welche Laufzeiten von den Conjugate Gradient (CG)-Implementierungen für eine Iteration des CG-Algorithmus aus Algorithmus 5.1 benötigen.

Im Folgenden wird zunächst auf die verwendete Hardware und die verwendeten Compiler eingegangen. Dann wird der Versuchsaufbau erklärt. In der selben Sektion wird noch gezeigt, wie die optimalen Blockgrößen in einigen der Implementierungen ermittelt wurden. Danach werden die gemessenen Laufzeiten der Implementierungen bei einer Iteration des CG-Algorithmus aus Algorithmus 5.1 vorgestellt und erklärt. Zuletzt wird die erreichte Leistung der Multi-GPU-Implementierung vorgestellt.

6.1 Verwendete Hardware

Zur Messung der Laufzeiten wurden zwei Maschinen verwendet. Die erste Maschine besitzt einen Hauptspeicher (RAM) von 755 GB. Sie besitzt zwei Intel Xeon Gold 5120 [Int] Prozessoren und acht NVIDIA GTX 1080 Ti 11GB [NVIb]. Spezifikationen der verbauten Recheneinheiten sind jeweils in Tabelle 6.1 und Tabelle 6.2 aufgelistet.

Codename	Skylake
Anzahl Kerne	14
Anzahl Threads	28
Grundtaktfrequenz	2,20 GHz
Max. Taktfrequenz	3,20 GHz
L3 Cache	19,25 MB

Tabelle 6.1: Spezifikation eines Intel Xeon Gold 5120 Prozessors
Quelle: [Int]

Die NVIDIA GTX 1080 Ti sind Verbraucher-Grafikkarten, was sich in den Spezifikationen äußert. Die FLOPS für das Rechnen mit einfacher Präzision sind 32-mal höher, als die FLOPS beim Rechnen mit doppelter Präzision. Aus diesem Grund wurde für die Graphics Processing Unit (GPU)-Laufzeiten eine zweite Maschine mit Server-Grafikkarten hinzugezogen. Die zweiten Maschine besitzt zwei AMD EPYC 7763 [Adv] Prozessor und vier NVIDIA A100-SXM4 [NVIa], sowie 1 TB RAM. In Tabelle 6.3 und Tabelle 6.4 sind Spezifikationen der verbauten Prozessoren und Grafikkarten aufgelistet.

Mikroarchitektur	Pascal
Anzahl CUDA-Kerne	3584
SMs	28
Grundtaktfrequenz	1480 MHz
Max. Taktfrequenz	1582 MHz
Single-Precision	11,34 TFLOPS
Double-Precision	354,4 GFLOPS
Speicherbandbreite	484 GB/s
VRAM	11 GB
CUDA-Fähigkeit	6.1

Tabelle 6.2: Spezifikation einer NVIDIA GTX 1080 Ti
Quellen: [NVIb] [NVIc]

Anzahl Kerne	64
Anzahl Threads	128
Taktfrequenz	2,45 GHz
Max. Taktfrequenz	3,5 GHz
L3 Cache	256 MB

Tabelle 6.3: Spezifikation eines AMD EPYC 7763 Prozessors
Quelle: [Adv]

6.2 Verwendete Compiler

Der in Compute Unified Device Architecture (CUDA) geschriebene Code wurde mit dem NVIDIA CUDA Compiler (NVCC), Version 12.0.1 kompiliert. Die in C++ geschriebenen Teile aller Implementierungen wurden mit dem GNU Compiler Collection (GCC), Version 9.4.0 kompiliert. Zum kompilieren mit GCC wurden die folgenden Compiler-Flags gesetzt:

-O3 teilt dem Compiler mit, welche Freiheiten dieser beim Optimieren des zu kompilierenden Codes hat. Die Stufe 3 entspricht viel Freiheit und einer aggressiven Optimierung des Codes.

-ffast-math fasst eine Reihe von Compiler-Flags zusammen, die Sicherheitsvorkehrungen wie IEEE-Standards vernachlässigen. Dadurch kann schneller mit Gleitkommazahlen gerechnet werden, aber die Genauigkeit der Rechnungen kann sinken.

-march=native weist den Compiler an, den Code für die spezifische Mikroarchitektur des lokalen Prozessors zu kompilieren. Code, der mit **-march=native** kompiliert wurde ist auf

Mikroarchitektur	Ampere
SMs	108
Anzahl CUDA-Kerne	6912
Max. Taktfrequenz	1410 MHz
Single-Precision	19,5 TLFOPS
Double-Precision	9,7 TLFOPS
Speicherbandbreite	1555 GB/s
VRAM	40 GB
CUDA-Fähigkeit	8.0

Tabelle 6.4: Spezifikation einer NVIDIA A100-SXM4
Quelle: [NVIa]

einen spezifischen Prozessor angepasst und auf Maschinen mit anderen Prozessoren nicht lauffähig.

Beim Kompilieren mit NVCC wurden erneut die Flags **-O3** und **-ffast-math** in Form von **–use_fast_math** verwendet.

Damit NVCC den Device-Code kompilieren kann, muss die CUDA-Faähigkeit der Grafikkarte spezifiziert werden. Diese Information wurde indirekt mit der CMAKE-Variable `CMAKE_CUDA_ARCHITECTURES` an NVCC übergeben.

6.3 Versuchsaufbau

Um die Performanz der einzelnen CG-Implementierungen zu vergleichen wurde die Zeit gemessen, die diese zum Berechnen einer Iteration aus Algorithmus 5.1 benötigen. Zum Rechnen wurden LGS verwendet, die vom C++ Programm, beschrieben in Abschnitt 5.7, generiert wurden.

Für das Sammeln der Laufzeiten wurde ein Szenarion fünf-mal hintereinander ausgeführt. Die Laufzeiten der fünf Durchläufe wurden anschließend gemittelt. Zum Messen der Laufzeiten wurde mit doppelter Präzision gerechnet.

6.3.1 Ermittlung der Blockgrößen der sequenziellen CPU-Implementierung

In der sequenziellen CPU-Implementierung werden die Matrix-Vektor-Multiplikationen blockweise ausgeführt. Die Daten für eine Teilrechnung sollten dabei vollständig in den L1-Cache

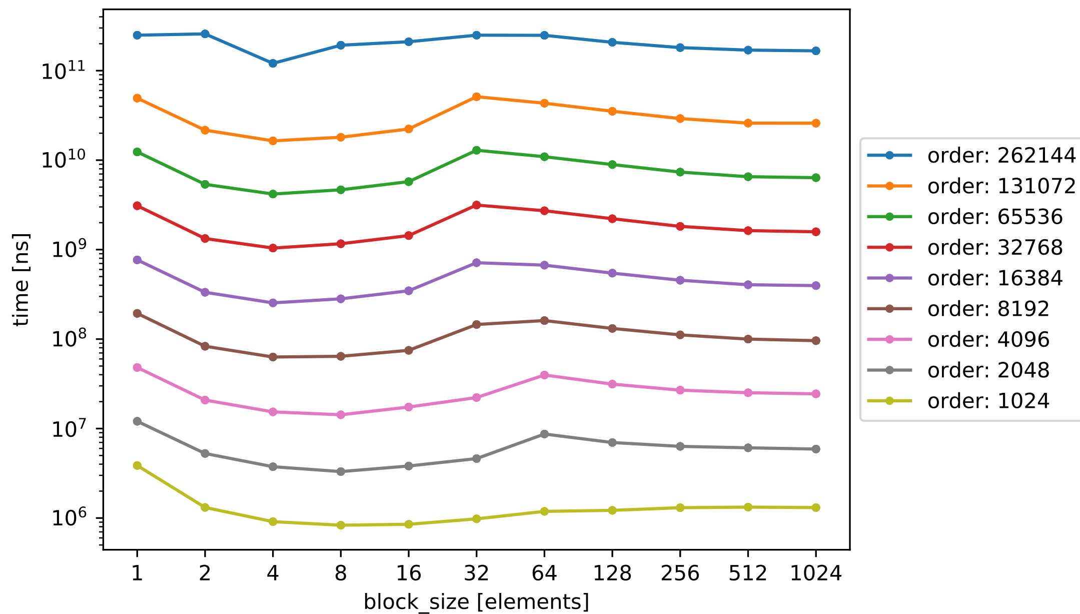

Abbildung 6.1: Laufzeit einer Iteration der sequenziellen CPU-Implementierung in Abhängigkeit der Blockgröße, für verschiedene LGS-Ordnungen
gerechnet mit dual-Socket Intel Xeon Gold 5120
Die optimale Blockgröße für große LGS beträgt vier

geladen werden können, sodass diese effizient ausgeführt werden kann. So kann die Geschwindigkeit der Gesamtberechnung beschleunigt werden.

In Abbildung 6.1 sind die Auswirkung der Blockgröße auf die Laufzeit der sequenziellen CPU-Implementierung zu sehen. Auf der x-Achse des Graphen sind die verwendeten Blockgrößen in Elementen aufgetragen. Die y-Achse spiegelt die Laufzeit einer Iteration des CG-Algorithmus in Nanosekunden wieder. Eingabedaten dieser Messung waren Lineares Gleichungssystem (LGS) mit einer Ordnung zwischen 1024 und 262144.

Das Verhältnis von Blockgröße zu Laufzeit verfolgt für alle LGS-Ordnungen einen ähnlichen Trend. Blöcke der Größe eins und zwei machen aus logische Sicht wenig Sinn, da die Größe einer L1-Cache-Zeile des Intel Xeon Gold 5120 Prozessors, auf dem gerechnet wird, vier Elemente doppelter Präszision fassen kann.

Für kleine LGS steigt die Laufzeit für eine Blockgröße ab acht Elementen wieder an. Bei größeren LGS, ab einer Ordnung von 16384 ist die minimale Laufzeit bei einer Blockgröße von vier Elementen erreicht. Aufgrund dieser Erkenntnis wurde sich für die sequenziellen CPU-Implementierung für eine Blockgröße von vier Elemente entschieden.

Für die parallelen CPU-Implementierungen wurde versucht, die Matrix-Vektor-Multiplikation ebenfalls mittels block-weisem Rechnen zu beschleunigen, dies gelang jedoch nicht. Deshalb berechnet jeder Thread in den parallelen Implementierungen eine Anzahl von Skalarprodukten, anstatt einer Anzahl von Teilmatrix-Vektor-Produkten.

6.3.2 Ermittlung der Thread-Block-Größen der GPU-Implementierungen

Beim Rechnen mit CUDA arbeiten jeweils t Threads in einem Thread-Block zusammen. Die Anzahl t der Threads in einem Thread-Block kann dabei die Ausführungsgeschwindigkeit eines Kernels beeinflussen. Um zu testen, welchen Einfluss die Größe der Thread-Blöcke auf die verwendeten Kernel haben, wurde die GPU-Implementierung mit verschiedenen Block-größen auf der NVIDIA GTX 1080 Ti ausgeführt.

Die Auswirkung der Thread-Block-Größen auf die Laufzeit des Algorithmus ist in Abbildung 6.2 zu sehen. Auf der x-Achse sind die Thread-Block-Größen aufgetragen. Die Thread-Block-Größen sind im Bereich von 32, der Größe eins Warps in CUDA und 1024, der maximalen Thread-Block-Größe, die von CUDA zugelassenen wird [NVI23]. Auf der y-Achse sind die entsprechenden Laufzeiten in Nanosekunden zu sehen.

Bei kleinen LGS lässt sich erkennen, dass große Thread-Block-Größen zu erhöhten Laufzeiten führen. Das liegt daran, das die resultierende Anzahl der Thread-Blöcke im Gitter so gering ist, dass nicht alle 28 Streaming Multiprozessoren (SMs) der NVIDIA GTX 1080 Ti ausgelastet werden.

Die durchschnittliche Laufzeit eines LGS beträgt 1.810.249,2ns für eine Ordnung von 1024, 1.867.858ns für eine Ordnung von 2048 und 1.924.286,8ns für ein LGS mit Ordnung 4096. Die Laufzeiten sind annähernd gleich, da die maximale Anzahl der Thread-Blöcke in einem Gitter immer unter der maximalen Anzahl der SM der GPU ist. In diesen Fällen werden die zusätzlichen Daten echt parallel verarbeitet, sodass die Kernel-Ausführung die selbe Zeit in Anspruch nehmen.

Für das LGS mit Ordnung bis $32768 = 2^{15}$ beträgt die Größe der darin enthaltenen Matrix A $2^{15} \cdot 2^{15} \cdot 2^3$ Bytes oder 8 GB. Da die gesammte Matrix in den Grafikspeicher (VRAM) der NVIDIA GTX 1080 Ti 11GB passt, müssen für eine Matrix-Vektor-Multiplikation der GPU-Implementierung keine neuen Teilmatrizen auf das Device kopiert werden. Für das LGS mit Ordnung bis $65536 = 2^{16}$ passt die zugehörige Matrix A nicht mehr in den VRAM des Devices. Damit ist der große Abstand der Laufzeiten zwischen den LGS der Ordnung 32768 und 65536 zu erklären.

Die in Kontext dieses Abschnitts aussagekräftigsten Laufzeiten sind die des LGS mit Ordnung 32768. Da für diese Größe noch nicht regelmäßig Teilmatrizen auf das Device kopiert werden müssen, fällt die Ausführungsgeschwindigkeit der Kernel am stärksten ins Gewicht. Abbildung 6.3 zeigt die entsprechende Stelle vergrößert an.

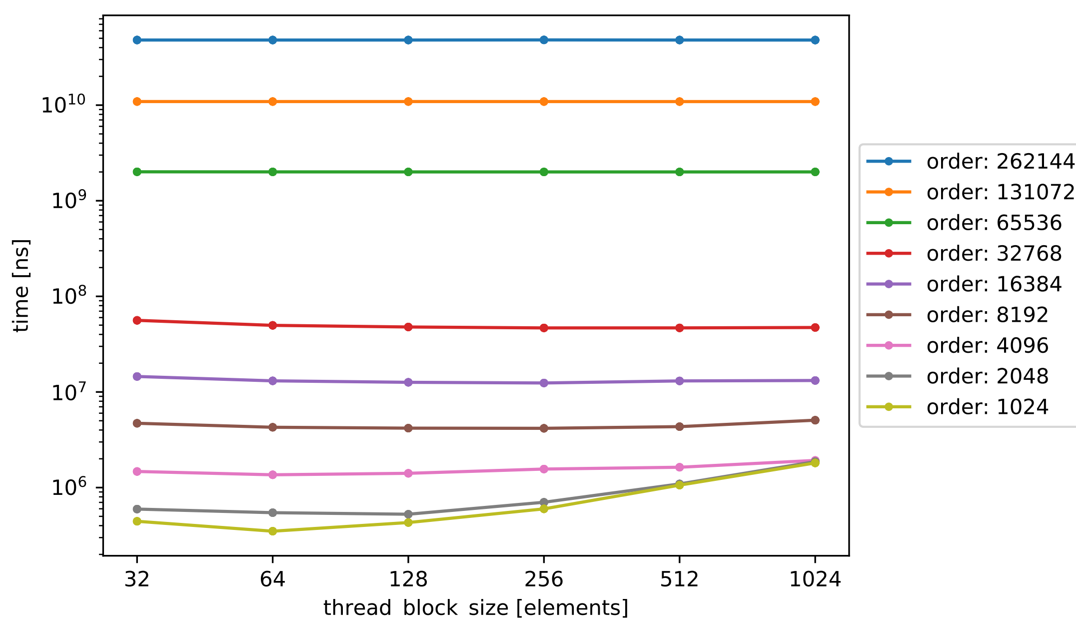

Abbildung 6.2: Laufzeit einer Iteration der GPU-Implementierung, in Abhängigkeit der Thread-Block-Größe, für unterschiedliche LGS-Größen
Gerechnet mit einer NVIDIA GTX 1080 Ti 11GB
Zu große Thread-Block-Größen hemmen die Ausführungsgeschwindigkeit des Kernels

Für eine Thread-Block-Größe von 32 ist die Ausführungszeit am höchsten. Für größere Thread-Block-Größen sind die Laufzeiten mertklich geringer, wobei sich das Minimum bei 256 befindet. Aufgrund der Daten wurde sich für die GPU Implementierungen für eine Thread-Block-Größe von 256 entschieden.

6.4 Laufzeiten einer Iteration

Die Laufzeiten, die in diesem Kapitel vorgestellt werden, sind die Zeit, die eine CG-Implementierung für eine Iteration benötigt. In der ausgeführten Iteration wird das Residuum auf implizite Weise, sodass nur eine Matrix-Vektor-Multiplikation ausgeführt wird.

6.4.1 CPU-Implementierungen

In Abbildung 6.4 sind die erzielten Laufzeiten der CPU-Implementierungen für unterschiedlich große LGS zu sehen. Die x-Achse zeigt die Ordnung der verwendeten LGS. Auf der y-Achse

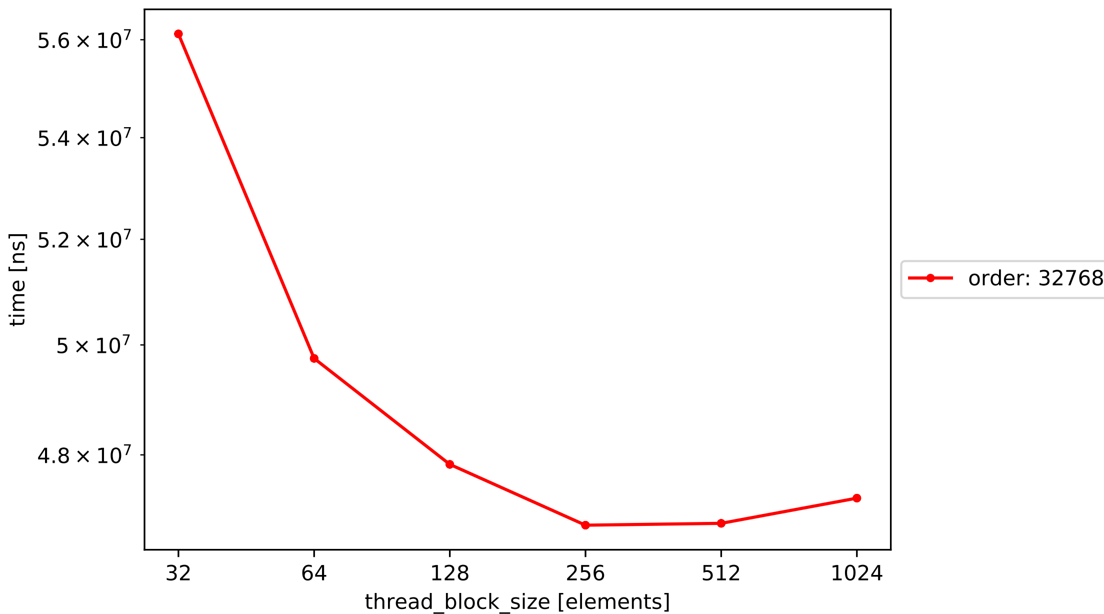

Abbildung 6.3: Laufzeit einer Iteration der GPU-Implementierung, in Abhängigkeit der Thread-Block-Größe, für ein LGS mit Ordnung 32768
gerechnet mit einer NVIDIA GTX 1080 Ti 11GB
Die optimale Thread-Block-Größe für die Kernels beträgt 256

sind die gemessenen Laufzeiten in Nanosekunden zu sehen. Die blaue Linie repräsentiert die sequenzielle Implementierung. Die parallelen Implementierung mit und ohne Speicherbegrenzung werden jeweils von der gelben (eingeschränkt) und grünen (unbeschränkte) Linie repräsentiert.

Die sequenzielle Implementierung ist die langsamste der CPU-Implementierungen, hat jedoch den konstantesten Zuwachs der Laufzeit, bei wachsender LGS-Ordnung. Dies liegt daran, dass die Matrix-Vektor-Multiplikation in der sequenziellen Implementierung block-weise berechnet wird. Die parallelen Implementierungen sind um ein vielfaches schneller, wobei die Implementierung, die mit begrenztem Speicher auskommen muss etwas langsamer ist, als die unbeschränkte parallele Implementierung.

Bei einer LGS-Ordnung von 4096, benötigt die sequenzielle Implementierung im Schnitt 29.029.058ns für eine Iteration des CG-Algorithmus. Für die selbe Berechnung benötigt die parallele Implementierung ohne Speichereinschränkung 2.425.354ns, sie ist also zwölf-mal schneller als die sequenzielle Implementierung. Die parallele Implementierungen mit Speicherbeschränkung ist in geringerem Maße schneller, sie benötigt für die Berechnung 4.708.676ns, was einem sechstel der sequenziellen Zeit entspricht.

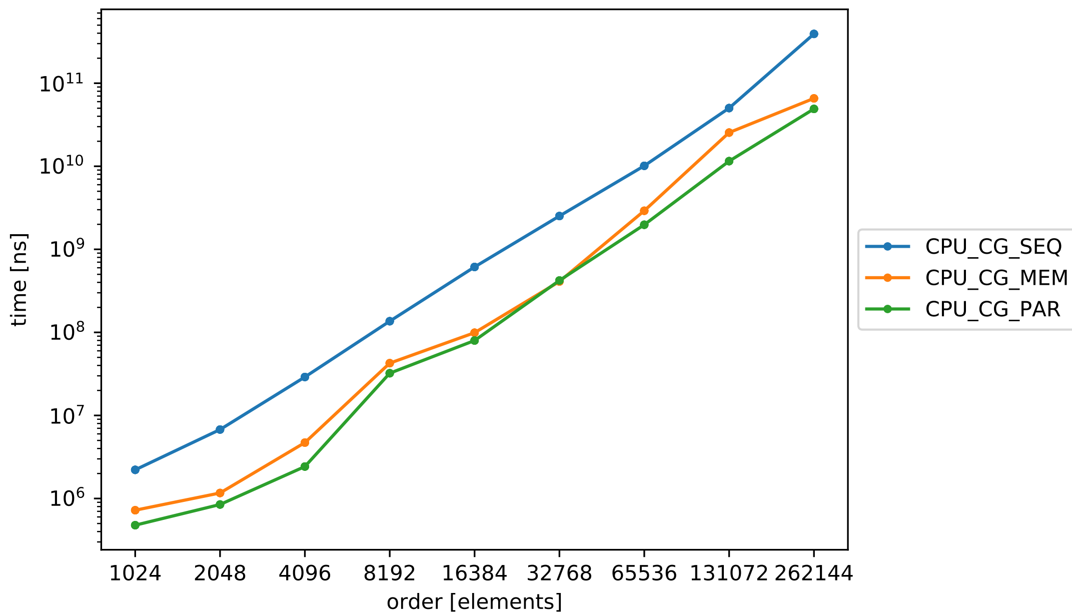

Abbildung 6.4: Laufzeit einer Iteration der CPU-Implementierungen, in Abhängigkeit der Ordnung der LGS
gerechnet mit dual-Socket Intel Xeon Gold 5120

Für eine LGS-Ordnung von 8192 gibt es einen überproportionalen Zuwachs der Laufzeit bei den parallelen Implementierungen. Hier beträgt die Laufzeit der parallele Implementierung ohne Speichereinschränkung 32.235.489ns, die Laufzeit der parallelen Implementierungen mit Speicherbeschränkung beträgt 42.609.088ns. Damit sind die parallelen Implementierungen nur noch $4,2$ bzw. $3,2$-mal schneller, als die sequenzielle Implementierung, deren Laufzeit 136.314.177ns beträgt.

Das liegt daran, dass ein Vektor der Länge 4096 mit doppelter Präzision 32768 Bytes groß ist und damit gerade noch in den 32KB großen L1-Cache eines Prozessorkerns des Intel Xeon Gold 5120 passt. Bei der Matrix-Vektor-Multiplikation auf einem Kern kann also der gesamte Eingabevektor x im L1-Cache gehalten werden. Bei einer Vektorgröße von 8192 passt der Eingabevektor x nicht mehr in den L1-Cache, was den überproportionalen Zuwachs der Laufzeit erklärt.

Den parallelen Implementierungen stehen $2 \cdot 14$ Kerne zum Berechnen zur Verfügung. Trotzdem sind sie nur maximal zwölf-mal schneller als die sequenzielle Implementierung. Dieser Umstand und die Tatsache, dass kein Cache-effizientes Rechnen umgesetzt werden konnte, spricht gegen die hier vorgestellten parallelen CPU-Implementierungen.

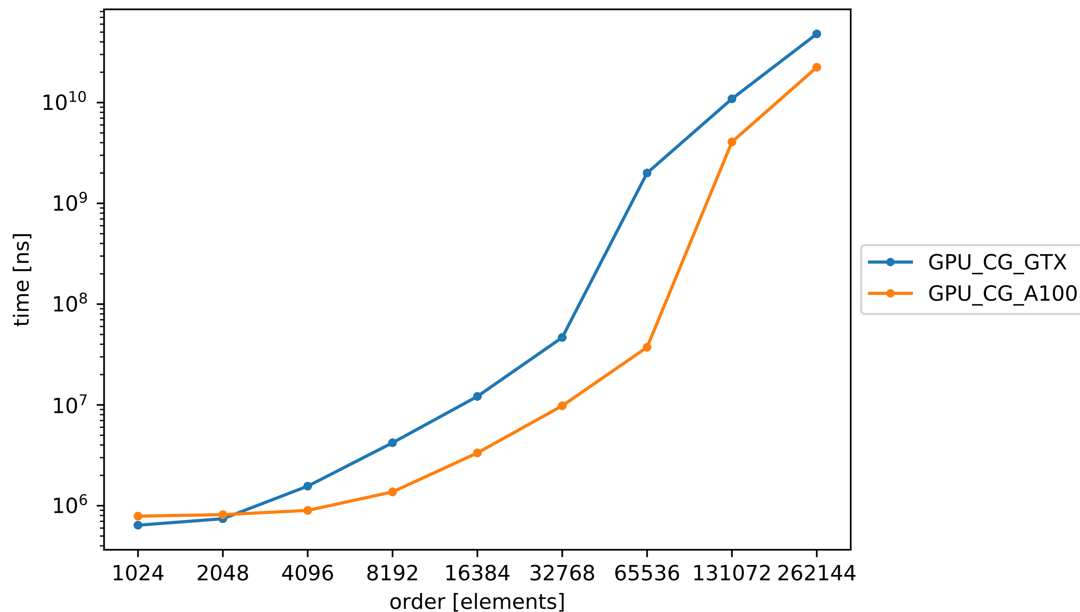

Abbildung 6.5: Laufzeit einer Iteration der Einzel-GPU-Implementierungen, in Abhängigkeit der Ordnung der LGS
blau wurde mit einer NVIDIA GTX 1080 Ti 11GB gerechnet
gelb wurde mit einer NVIDIA A100-SXM4 40GB gerechnet

6.4.2 Einzel-GPU-Implementierung

Für die Einzel-GPU-Implementierungen wurden die Laufzeiten auf zwei unterschiedlichen Grafikkarten gemessen. Die gemessenen Laufzeiten sind in Abbildung 6.5 zu sehen. Dabei zeigt die x-Achse die Ordnung der verwendeten LGS. Die Zeit, die zum Berechnen einer CG-Iteration benötigt wurde ist auf der y-Achse zu sehen. Die beiden Linien zeigen jeweils die Laufzeiten beim Rechnen auf einer NVIDIA GTX 1080 Ti (blau) mit 11GB VRAM und beim Rechnen auf einer NVIDIA A100-SXM4 (gelb) mit 40GB VRAM.

Für LGS mit einer Ordnung von 2048 und weniger fällt auf, dass die NVIDIA GTX 1080 Ti eine bessere Laufzeit aufweist, als die NVIDIA A100-SXM4. Bei einem LGS mit Ordnung 1024 wird der Kernel der Matrix-Vektor-Multiplikation auf einem Gitter mit 16 Thread-Blöcken gestartet. Diese Anzahl an Thread-Blöcken bleibt hinter der Anzahl der SM beider Grafikkarten zurück, sodass diese nicht ihr volles Rechenpotenzial nutzen können.

Bei einer LGS-Ordnung von 2048 werden für eine Matrix-Vektor-Multiplikation 64 Thread-Blöcke an die SM der Grafikkarten aufgeteilt. Auch hier sind die SM der NVIDIA A100-SXM4 nicht vollständig belegt, weshalb sich die Laufzeit von 1024 auf 2048 nur von 787.883,4ns auf

815.190,6ns erhöht. Die Laufzeit der NVIDIA GTX 1080 Ti steigen im selben Interval etwas stärker an, von 640.821,6ns auf 742.734,8ns.

Für alle größeren LGS-Ordnungen sind die Laufzeiten der NVIDIA A100-SXM4 deutlich geringer als die der NVIDIA GTX 1080 Ti. Der zweitgrößte Unterschied ist bei einer LGS-Ordnung von 32768 vorzufinden. Für ein LGS diese Ordnung beträgt die Laufzeit beim Rechnen mit der NVIDIA A100-SXM4 im Schnitt 9.801.637,2ns. Die NVIDIA GTX 1080 Ti benötigt für die selbe Rechnung durchschnittlich 46.709.204,2ns, was der 4,7-fachen Laufzeit der NVIDIA A100-SXM4 entspricht.

Der größte Unterschied ist bei einer LGS-Ordnung von 65536. Hier reicht der VRAM der NVIDIA GTX 1080 Ti nicht mehr aus, um die vollständige Matrix des LGS im Speicher zu halten, sodass während der Berechnung Teilmatrizen auf das Device kopiert werden müssen. Dies verlangsamt die Berechnung auf der NVIDIA GTX 1080 Ti erheblich, sodass die durchschnittliche Laufzeit auf 2.002.037.053ns ansteigt. Für diese LGS-Ordnung ist die NVIDIA A100-SXM4 mit einer durchschnittlichen Laufzeit von 37.341.613,6ns über 50-mal schneller.

Für eine LGS-Ordnung von 131072 schließt sich diese Kluft der Laufzeiten wieder. Hier reicht der VRAM der NVIDIA A100-SXM4 ebenfalls nicht mehr aus, um die vollständige Matrix des LGS im Speicher zu halten. Als Konsequenz müssen Teilmatrizen während der Berechnung kopiert werden. So ergibt sich auch hier ein Sprung der Laufzeiten. Die durchschnittliche Laufzeit der NVIDIA A100-SXM4 beträgt für eine LGS-Ordnung von 131072 4.068.300.894,8ns. Die Laufzeit der NVIDIA GTX 1080 Ti beträgt 10.898.605.782,2ns was dem 2,5-fache der Laufzeit der NVIDIA A100-SXM4 entspricht. Für eine Ordnung von 262144 sind die Laufzeiten der Grafikkarten noch näher beisammen. Die NVIDIA A100-SXM4 benötigt im Schnitt 22408328965,4ns, für die NVIDIA GTX 1080 Ti sind es 48.017.166.116,4ns. Der Laufzeitunterschied beträgt nur noch das 2,15-fache.

Die Speicherbandbreite der NVIDIA A100-SXM4 beträgt jedoch das 3,2-fache der Speicherbandbreite der NVIDIA GTX 1080 Ti. Die Rechenleistung für eine Matrix-Vektor-Multiplikation sollte also mehr als das dreifache sein. Ab einer Ordnung von 131072 müssen jedoch regelmäßig neue Teilmatrizen auf das Device kopiert werden. Hier ist die Berechnung weniger durch die Rechenleistung der Grafikkarte begrenzt als durch die Übertragungsrate vom Host zum Device. Die NVIDIA A100-SXM4 ist über PCIe 4.0 mit dem Host verbunden. Diese hat eine doppelt so große Übertragungsrate wie die PCIe 3.0 Anbindung der NVIDIA GTX 1080 Ti [SIG]. Deshalb rücken die gemessenen Laufzeiten für LGS mit großer Ordnung näher zusammen.

6.4.3 Multi-GPU-Implementierung

In Abbildung 6.6 sind die gemessenen Laufzeiten der Multi-GPU-Implementierung zu sehen. Wie bei der Einzel-GPU-Implementierung wurden die Laufzeiten auf zwei unterschiedlichen Maschinen gemessen, die jeweils Zugriff auf mehrere GPUs haben. Die x-Achse zeigt die Ordnung der verwendeten LGS. Auf der y-Achse sind die gemessenen Laufzeiten in Nanosekunden zu sehen. In der Abbildung zeigt die blaue Linie die gemessenen Laufzeiten beim Rechnen mit

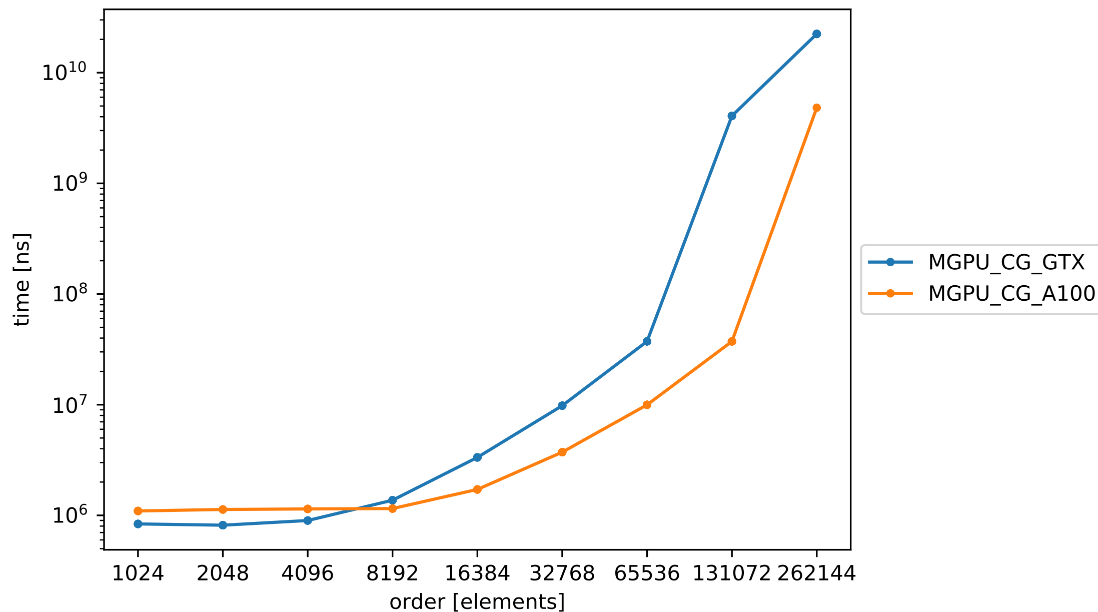

Abbildung 6.6: Laufzeit einer Iteration der Multi-GPU-Implementierungen, in Abhängigkeit der Ordnung der LGS
blau wurde mit acht NVIDIA GTX 1080 Ti 11GB gerechnet
gelb wurde mit vier NVIDIA A100-SXM4 40GB gerechnet

acht NVIDIA GTX 1080 Ti, mit jeweils 11GB VRAM. Die gelbe Linie zeigt die Laufzeiten beim Rechnen mit vier NVIDIA A100-SXM4, mit jeweils 40GB VRAM.

Erneut fällt auf, dass die NVIDIA GTX 1080 Ti für kleine Datenmengen besser abschneidet, als die NVIDIA A100-SXM4. Bei einem LGS mit Ordnung 2048 müssen in Summe 64 Thread-Blöcke zum Berechnen des Kernel der Matrix-Vektor-Multiplikation gestartet werden. Diese sind jedoch verteilt auf mehrere Grafikkarten. Im Fall der NVIDIA GTX 1080 Ti wird der Kernel auf jeder der acht Grafikkarten auf einem Gitter mit jeweils acht Thread-Blöcken gestartet. Hier bleibt die Anzahl der Thread-Blöcke erneut hinter der Anzahl der SM zurück, weshalb die NVIDIA GTX 1080 Ti nicht vollständig ausgelastet werden.

Das selbe Phänomen tritt für die NVIDIA A100-SXM4 bis zu einer LGS-Ordnung von 8192 auf. Erst ab einer LGS-Ordnung von 4096 für die NVIDIA GTX 1080 Ti und 16384 für die NVIDIA A100-SXM4 steigt mit der Ordnung der LGS auch die Laufzeit merklich.

Bei einer LGS-Ordnung von 65536 beträgt die Laufzeit beim Rechnen mit die NVIDIA A100-SXM4 im Schnitt 9.976.281 ns. Die NVIDIA GTX 1080 Ti benötigen für die selbe Rechnung durchschnittlich 37.341.613,6 ns, was der vierfachen Laufzeit der NVIDIA A100-SXM4 entspricht.

Die Matrix des LGS mit Ordnung 131072 überschreitet den kombinierten VRAM der acht NVIDIA GTX 1080 Ti, sodass bei der zugehörigen Berechnung mehrfach Teilmatrizen auf das Device kopiert werden müssen. Für diese LGS-Ordnung beträgt die gemessene Laufzeit der NVIDIA GTX 1080 Ti 4.068.300.894,8ns. Die Laufzeit, die von den NVIDIA A100-SXM4 erreicht wurde beträgt 37.300.665,6ns. Sie ist damit 110-mal geringer, als die Laufzeit der NVIDIA GTX 1080 Ti.

Die gemessenen Laufzeiten nähern sich nach diesem Extremfall wieder auf das 5,5-fache an. Es ist anzunehmen, dass die Laufzeiten, wie bei der Einzel-GPU-Implementierung, für noch größere LGS-Ordnungen näher zusammenrücken.

6.4.4 Erreichte Leistung

Um die erreichte Leistung der Multi-GPU-Implementierung zu quantifizieren, werden die erreichten FLOPS manuell berechnet.

Für die gemessenen Laufzeiten wird eine Iteration aus Algorithmus 5.1 mit impliziter Berechnung des Residuums ausgeführt. Dabei werden ein Matrix-Vektor-Produkt, zwei Skalarprodukte, drei Vektoradditionen, drei Skalar-Vektor-Produkte und eine skalare Division berechnet. Das ergibt in Summe eine FLOP-Anzahl von:

$$2N^2 - N + 2(2N - 1) + 3N + 3N + 1 = 2N^2 + 9N - 1$$

Der Aufwand zum Berechnen einer Operation ist aus der Arbeit von Hunger et al. [Hun05] entnommen.

LGS Ordnung	65536	131072	262144
acht NVIDIA GTX 1080 Ti	37.341.613,6ns	4.068.300.894,8ns	22.408.328.965,4ns
vier NVIDIA A100-SXM4	9.976.281ns	37.300.665,6ns	4.823.752.491,2ns
Benötigte FLOP	4.295.557.119	17.181.048.831	68.721.836.031
NVIDIA GTX 1080 Ti	115 GFLOPS	4,2 GFLOPS	3 GFLOPS
NVIDIA A100-SXM4	430,5 GFLOPS	460,6 GFLOPS	14 GFLOPS

Tabelle 6.5: Durchschnittlich gemessene Laufzeiten der Multi-GPU-Implementierung, für LGS massiver Größe

In Tabelle 6.5 sind die gemessenen Laufzeiten, sowie die errechneten FLOPS zu sehen. Im Sinne dieser Arbeit wurden nur die Laufzeiten jener LGS berücksichtigt, die nicht vollständig in den VRAM einer NVIDIA GTX 1080 Ti passen.

Die erreichten FLOPS für die LGS-Ordnung, in denen keine zusätzlichen Teilmatrizen auf das Device kopiert werden müssen, betragen 115 GFLOPS für die acht NVIDIA GTX 1080 Ti. Das

bleibt verständlicherweise weit hinter den kombinierten $8 \cdot 384,4$ GFLOPS der Grafikkarten zurück. Das liegt daran, dass die auf der GPU gerechneten Operationen hauptsächlich durch die Speicherbandbreite der GPU begrenzt sind. Zudem werden zwischen den Berechnungen Daten zum und vom Device kopiert, Kernel werden gestartet und die Geräte werden synchronisiert. All das führt dazu, dass die erreichten 115 GFLOPS ca. 25-mal geringer sind als, die theoretisch von den GPU erreichbare Rechenleistung. Das selbe gilt für die Berechnung auf den vier NVIDIA A100-SXM4.

Die erreichten FLOPS sind noch geringer, wenn zusätzlich noch das Kopieren von Teilmatrizen in die Rechnung mit einfließt. Beim Rechnen auf den acht NVIDIA GTX 1080 Ti sinkt die Anzahl der FLOPS von 115 GFLOPS auf 4,2 GFLOPS. Der Leistungseinbruch bei den vier NVIDIA A100-SXM4 steht in ähnlichem Verhältnis. Hier sinkt die Anzahl der erreichten FLOPS von 460,6 GFLOPS auf 14 GFLOPS. Für die NVIDIA GTX 1080 Ti sinkt die Rechenleistung um das 27-fache, für die NVIDIA A100-SXM4 um das 32-fache.

Bei einer LGS-Ordnung von 65536 erfordert das Rechnen auf einer einzelnen NVIDIA GTX 1080 Ti, dass während des Rechnens Teilmatrizen auf das Device kopiert werden. Das führt für diese LGS-Größe zu einer EinzelGPU-Laufzeit von 2.002.037.053ns. Das entspricht der 53-fachen Laufzeit der Multi-GPU-Implementierung (vgl. Tabelle 6.5). Die Einzel-GPU-Laufzeit einer NVIDIA A100-SXM4 beträgt für ein LGS mit Ordnung 131072 im Schnitt 4.068.300.894,8ns, was dem 109-fachen der Laufzeit der Multi-GPU-Implementierung entspricht (vgl. Tabelle 6.5).

7 Fazit und Ausblick

In dieser Arbeit wurde eine effiziente Implementierung des CG-Verfahrens vorgestellt, das sich mehrere Grafikkarten zunutze macht, um massive LGS zu lösen. Es wurden mehrere Implementierungen des CG-Verfahrens für unterschiedliche Umgebungen gezeigt und deren Laufzeiten wurden vorgestellt.

Die Multi-GPU-Implementierung erreicht maximal 115 GFLOPS beim Rechnen auf acht NVIDIA GTX 1080 Ti und 460,6 GFLOPS beim Rechnen auf vier NVIDIA A100-SXM4, was weit hinter den theoretischen kombinierten FLOPS der Grafikkarten zurückliegt. Das ist vor allem der Probemstellung geschuldet. Es müssen mehrfach Vektoren zum und vom Device kopiert werden, die Maschinen müssen synchronisiert werden und es müssen Kernel auf den Devices gestartet werden. All das verlangsamt die Ausführung einer Iteration.

Beim Vergleich der Fälle, die die Nutzung von mehreren Grafikkarten motivieren sollen, ist die Multi-GPU-Implementierung 53-mal schneller als die Einzel-GPU-Implementierung unter Verwendung von acht NVIDIA GTX 1080 Ti. Unter verwendung von view NVIDIA A100-SXM4 ist die Multi-GPU-Implementierung sogar 109-mal schneller als die Einzel-GPU-Implementierung.

Ausblick

Bei der Multi-GPU-Implementierung wurden mehrere Techniken zum Optimieren Datenübertragung und einige Optimierungen des Kernels angewandt. Eine grundlegende Optimierung des Kernels wurde jedoch nicht umgesetzt, das Padding. Beim Padding werden die Daten auf dem Device so ausgestopft, dass die Grenzüberprüfungen im Kernel überflüssig werden. Dadurch wird der Kernel weniger rechenintensiv, was seine Laufzeit beschleunigen kann. Aufgrund der ohnehin übermäßigen Kopiervorgänge auf das Device und der geringen Rechenbelastung der GPU, wurde dieser Optimierung jedoch eine geringe Priorität eingeräumt, sodass sie letztendlich nicht umgesetzt wurde.

Eine weiter Möglichkeit zur Beschleunigung der Berechung ist, dass alle Daten, die für die Berechnungen verwendet werden, mit Ausnahme der Matrizen des LGS dauerhaft auf dem Device hinterlegt werden. Hier entstehen neue Komplikationen, da die Devices untereinander die Ergebnisse der aufgeteilten Matrix-Vektor-Multiplikation kommunizieren müssen. Die Gesamtanzahl der Kopiervorgänge zwischen dem Host und den Devices sollte jedoch sinken, was die Laufzeit zusätzlich beschleunigen kann.

Literaturverzeichnis

[Adv] Advanced Micro Devices Incorporated. *AMD EPYC™ 7763*. URL: https://www.amd.com/en/products/cpu/amd-epyc-7763 (zitiert auf S. 57, 58).

[AKWS10] M. Ament, G. Knittel, D. Weiskopf, W. Straßer. „A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform“. In: Feb. 2010, S. 583–592. DOI: 10.1109/PDP.2010.51 (zitiert auf S. 19).

[CNM09] A. Cevahir, A. Nukada, S. Matsuoka. „Fast Conjugate Gradients with Multiple GPUs.“ In: *ICCS (1)* 5544 (2009), S. 893–903 (zitiert auf S. 19).

[CNM10] A. Cevahir, A. Nukada, S. Matsuoka. „High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning“. In: *Computer Science-Research and Development* 25 (2010), S. 83–91 (zitiert auf S. 19).

[GJ+10] G. Guennebaud, B. Jacob et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010 (zitiert auf S. 19).

[HS+52] M. R. Hestenes, E. Stiefel et al. „Methods of conjugate gradients for solving linear systems“. In: *Journal of research of the National Bureau of Standards* 49.6 (1952), S. 409–436 (zitiert auf S. 19, 21).

[Hun05] R. Hunger. *Floating point operations in matrix-vector calculus*. Bd. 2019. Munich University of Technology, Inst. for Circuit Theory und Signal …, 2005 (zitiert auf S. 68).

[Int] Intel Coprporation. *Intel® Xeon® Gold 5120 Prozessor*. URL: https://ark.intel.com/content/www/de/de/ark/products/120474/intel-xeon-gold-5120-processor-19-25m-cache-2-20-ghz.html (zitiert auf S. 57).

[LL14] X. Li, F. Li. „GPU-based power flow analysis with Chebyshev preconditioner and conjugate gradient method“. In: *Electric Power Systems Research* 116 (2014), S. 87–93 (zitiert auf S. 19).

[LS13] I. B. Labutin, I. V. Surodina. „Algorithm for Sparse Approximate Inverse Preconditioners in the Conjugate Gradient Method.“ In: *Reliab. Comput.* 19.1 (2013), S. 120–126 (zitiert auf S. 19).

[MGSS13] E. Müller, X. Guo, R. Scheichl, S. Shi. „Matrix-free GPU implementation of a preconditioned conjugate gradient solver for anisotropic elliptic PDEs“. In: *Computing and Visualization in Science* 16 (2013), S. 41–58 (zitiert auf S. 19).

[NVIa] NVIDIA Coprporation. *NVIDIA A100 Tensor Core GPU Architecture*. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf (zitiert auf S. 42, 57, 59).

[NVIb] NVIDIA Coprporation. *NVIDIA GeForce GTX 1080 Ti*. URL: https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1080-ti/specifications/ (zitiert auf S. 57, 58).

[NVIc] NVIDIA Coprporation. *Whitepaper NVIDIA GeForce GTX 1080*. URL: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf (zitiert auf S. 58).

[NVI23] NVIDIA Corporation. *CUDA C++ Programming Guide*. Version Release 12.1. 2023. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (zitiert auf S. 17, 27, 28, 44, 61).

[Oli–] T. Oliphant. *NumPy: A guide to NumPy*. USA: Trelgol Publishing. 2006–. URL: http://www.numpy.org/ (zitiert auf S. 19).

[Ope15] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.5*. Mai 2015. URL: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (zitiert auf S. 40).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. „Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830 (zitiert auf S. 53).

[Saa03] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003 (zitiert auf S. 22).

[Sch07] E. Schmidt. „Zur Theorie der linearen und nichtlinearen Integralgleichungen". In: *Mathematische Annalen* 63.4 (1907), S. 433–476 (zitiert auf S. 23).

[She+94] J. R. Shewchuk et al. *An introduction to the conjugate gradient method without the agonizing pain*. 1994 (zitiert auf S. 17, 21, 24, 25, 37, 38).

[SIG] P. SIG. *PCI Express Base Specification*. URL: https://pcisig.com/specifications/pciexpress (zitiert auf S. 66).

Alle URLs wurden zuletzt am 17. 03. 2008 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Sindelfingen, den 02.06.2023, P. Arlt

Ort, Datum, Unterschrift