

TECHNISCHE UNIVERSITÄT DRESDEN

ZENTRUM FÜR INFORMATIONSDIENSTE

UND HOCHLEISTUNGSRECHNEN

PROF. DR. WOLFGANG E. NAGEL

Belegarbeit Computational Science and Engineering  
Verteilte GPGPU-Berechnungen mit Spark

Maximilian Knespel

Hochschullehrer: Prof. Dr. Wolfgang E. Nagel

Betreuer: Dipl.-Inf. Nico Hoffmann

Dresden, 10. August 2016

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>2</b>
<b>2</b>	<b>Apache Spark</b>	<b>3</b>
2.1	MapReduce . . . . .	4
2.2	Architektur . . . . .	5
2.3	Konfiguration von Spark auf einem Slurm-Cluster . . . . .	6
<b>3</b>	<b>Rootbeer</b>	<b>7</b>
<b>4</b>	<b>Implementation</b>	<b>9</b>
4.1	Monte-Carlo-Algorithmen . . . . .	9
4.2	Berechnung von Pi . . . . .	9
4.3	Implementation des Kernels . . . . .	11
4.4	Rootbeer-Kernelaufruf . . . . .	13
4.5	Kombination von Rootbeer mit Spark . . . . .	15
4.6	Kompilierung . . . . .	17
<b>5</b>	<b>Leistungsanalyse</b>	<b>19</b>
5.1	Testsystem . . . . .	19
5.2	Profiling . . . . .	20
5.3	Implementationsunterschiede . . . . .	22
5.4	Spark mit Rootbeer . . . . .	23
<b>6</b>	<b>Zusammenfassung</b>	<b>26</b>
	<b>Literaturverzeichnis</b>	<b>27</b>
<b>A</b>	<b>Standardabweichung des Mittelwertes</b>	<b>30</b>

# 1 Einführung

Im Rahmen dieser Belegarbeit soll ein Ansatz entwickelt werden, der es ermöglicht mit Java oder Scala auf heterogenen Clustersystemen mit Grafikkarten zu rechnen. Es wurde sich für eine Kombination aus Spark für die Kommunikation im Cluster und Rootbeer für die Grafikkartenprogrammierung entschieden.

Spark vereinfacht die ausfallsichere Programmierung von Clustern mittels des Map-Reduce-Programmiermodells und stellt zahlreiche Bibliotheken z.B. für Graphenalgorithmen und statistische Analysen zur Verfügung.

Für Rootbeer wurde sich entschieden, weil es das Schreiben von CUDA-Kernels aus Java heraus erlaubt. Die so geschriebenen Kernel-Funktionen können dann sowohl auf Grafikkarten als auch auf dem Host ausgeführt werden.

Zuerst werden in den Kapiteln 2 und 3 die benutzten Frameworks genauer vorgestellt, in Kapitel 4 wird auf die eigene Implementierung eingegangen und in Kapitel 5 werden Benchmarks, die mit dieser Implementierung angefertigt wurden, ausgewertet.

## 2 Apache Spark

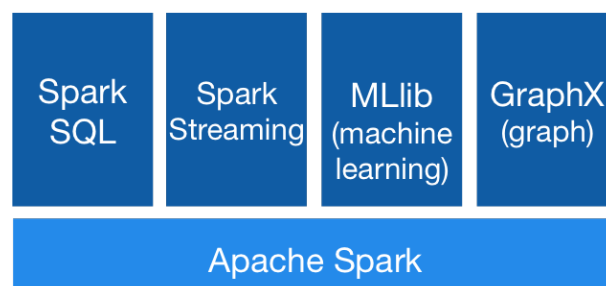
Spark ist ein Programmierframework für Datenanalyse auf Clustern, was vor allem im Zusammenhang mit "Big Data" an Beliebtheit gewonnen hat. Es vereinigt hierbei ausfallgehärtete Funktionalitäten von Batchverarbeitungssystemen bzw. Cluster-Management wie z.B. SLURM, Kommunikation zwischen Prozessen, wie z.B. OpenMPI und OpenMP sie zur Verfügung stellen, und zusätzliche problemspezifische Programmierbibliotheken. Spark stellt Programmierschnittstellen für Java, Scala und Python zur Verfügung. Für Scala und Python existieren interaktive Eingabeaufforderungen mit der z.B. interaktive Echtzeitanalysen auf Clustern mit Spark möglich sind.

Apache Spark oder auch Spark Core stellt die Grundfunktionalität für verteiltes Rechnen bereit. Das inkludiert die erwähnte Ausfallsicherheit, Management von Knoten über ein Web-Interface und Prozess-Scheduling[21]. Die Programmierschnittstelle hierfür sind der Spark-Kontext und die Resilient Distributed Datasets (RDD).

Beispielhaft für den Spark-Stack seien hier die Bibliotheken MLlib[3] und GraphX erwähnt.

MLlib, kurz für Machine Learning Library, umfasst Funktionen wie lineare Regression, den K-Means-Algorithmus, Latent Dirichlet Allocation, Hauptkomponentenanalyse, das für Maschinennlernen benötigte stochastische Gradientenverfahren, u.v.m.

GraphX erweitert Spark-RDDs zu Kanten- und Knoten-RDDs die als Tupel einen Graph beschreiben [5]. Auf diesen abgeleiteten Datentypen sind übliche RDD- und Mengenoperationen wie `subgraph`, `diff`, u.a. für das Erstellen und Modifizieren von Graphen möglich. Ein solcher Graph kann z.B. Webseitenrelationen, jeder Link ist eine Kante im Graph, jede Domain ein Knoten, darstellen. Auf diesen Graphen sind Algorithmen wie PageRank oder das Auszählen von Dreiecken in Graphen ausführbar.



**Abbildung 2.1:** Zusammensetzung des Spark-Frameworks[4]

## 2.1 MapReduce

Spark basiert auf dem MapReduce-Paradigma, welches 2004 in einem Paper der Google-Mitarbeiter Dean und Ghemawat[13, 14] in Anlehnung an die aus funktionalen Programmiersprachen bekannten Map- und Reduce-Befehle eingeführt wurden. Viele der benötigten Algorithmen wie Häufigkeitsanalyse oder Webseitengraphen hatten zuvor ihren eigenen Programmcode für die Kommunikation und Ausfallsicherheit im Cluster und folgten von der Struktur her einem simplen Schema: zuerst werden Eingabedaten datenparallel verarbeitet und anschließend werden die Zwischenergebnisse zu Endergebnissen reduziert. Diese einfach gestrickten Algorithmen mussten aber trotzdem auf beträchtlichen Datenmengen auf Knoten hoher Ausfallwahrscheinlichkeit laufen. Diese Funktionalität, also die Kommunikation im Cluster und die Ausfallsicherheit, wurde so in eine MapReduce-Bibliothek ausgelagert.

MapReduce bezeichnet hierbei sowohl das Programmierparadigma als auch die Bibliothek, die diese ausfallsicher und parallelisiert zur Verfügung stellt. Dafür definiert der Nutzer eine Map-Funktion die aus einer Liste jedes Schlüssel-Wert-Paar  $(k, v)$  auf eine Liste aus neuen Schlüssel-Wert-Paaren abbildet:

$$\mathbf{Map} : (k, v) \mapsto [(l_1, x_1), \dots, (l_{r_k}, x_{r_k})] \quad (2.1)$$

$k$  und  $l$  sind hierbei die Schlüssel und  $v$  und  $x$  die dazugehörigen Werte. Für  $r_k = 1$  erhält man den Grenzfall, dass jeder Schlüssel-Wert-Tupel auf exakt einen neuen Schlüssel-Wert-Tupel abgebildet wird.

Da es per Definition keine Abhängigkeiten zu anderen Schlüssel-Wert-Paaren für die Berechnung der Map-Funktion gibt, kann jedes Datum parallel berechnet werden. Das MapReduce-Framework stellt außerdem Funktionen zum Einlesen von Daten zur Verfügung und verteilt diese automatisch an die Knoten des Clusters, wo diese dann parallel verarbeitet werden.

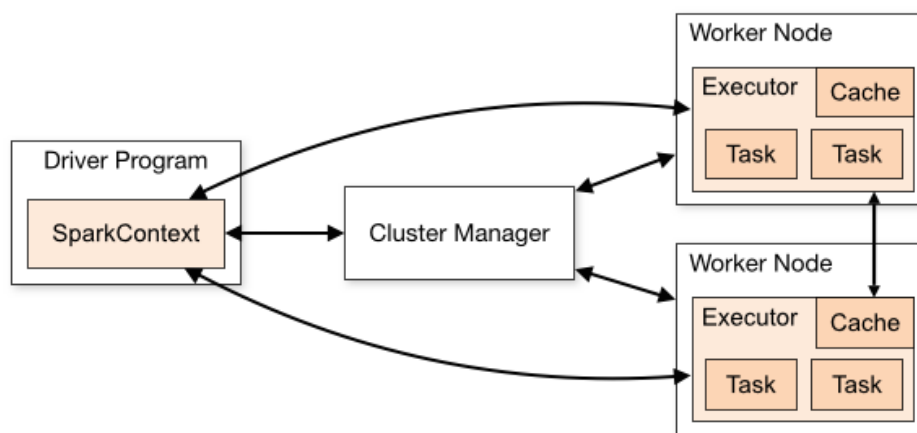
In einem impliziten Shuffle-Schritt werden alle Paare mit gleichem Schlüssel lokal gruppiert, d.h. es werden alle Paare z.B. mit Schlüssel  $k_1 = \text{'Dresden'}$  auf dem gleichen Knoten im Cluster gesammelt. Dieser Schritt kann also sehr kommunikationsaufwendig sein. Der Nutzer kann nur über die Art der Schlüssel auf diesen Prozess Einfluss nehmen. Man erhält als Ergebnis des Shuffle-Schritts einen Tupel aus einem Schlüssel und einer Liste an dazugehörigen Werten.

$$\mathbf{Shuffle} : [(k_1, x_1), \dots, (k_n, x_n)] \mapsto \left[ \left( l_1, [y_1^1, \dots, y_{r_1}^1] \right), \dots, \left( l_m, [y_1^m, \dots, y_{r_m}^m] \right) \right] \quad (2.2)$$

Im nachfolgenden Reduce Schritt werden die Tupel aus Schlüssel und Werteliste mit einer vom Nutzer definierten Reduce-Funktion abgebildet auf eine neue Werteliste. Im einfachsten Fall kann die neue Werteliste nur ein Element enthalten, z.B. die Summe oder den Mittelwert der alten Werte.

$$\mathbf{Reduce} : (l, [y_1, \dots, y_{s_l}]) \mapsto [w_1, \dots, w_{m_l}] \quad (2.3)$$

Auch die Reduce-Operation kann also bezüglich einzigartiger Schlüssel datenparallel ausgeführt werden. Anfangs war dieses Paradigma nur für einfache Beispiele wie Wortfrequenzanalysen gedacht, aber mittlerweile wurden auch komplexere Algorithmen wie z.B. Matrixmultiplikation[33] oder das Problem des Finden einer maximalen Überdeckung[9] ins MapReduce-Schema portiert.



**Abbildung 2.2:** Master/Slave-Struktur eines Spark-Clusters[4]

Eine lange Zeit beliebte Implementation basierend auf dem MapReduce-Paper[13] ist Apache Hadoop[2]. Hadoop besteht aus einem verteilten Dateisystem, dem Hadoop Distributed File System (HDFS), und einer Bibliothek namens MapReduce, die Funktionen für das Rechnen auf den verteilten Daten anbietet.

Apache Spark[4] ist eine alternative Implementation des MapReduce-Modells. Es versucht viele der Probleme von Hadoop zu beheben, bietet jedoch neben dem Zugriff vom lokalen Dateisystem auch den Zugriff von HDFS und weiteren verteilten Dateisystemen an.

Einer der Hauptvorteile von Spark ist der Geschwindigkeitsgewinn bei der iterativen Anwendung von Map und Reduce durch die Möglichkeit die Daten auch im Arbeitsspeicher, nicht nur auf der Festplatte, der Knoten zwischenspeichern.

## 2.2 Architektur

Für kleinere Tests kann Spark local auf einem Computer bzw. Knoten ausgeführt werden:

```
spark-shell --master local[*]
```

In diesem Beispielaufwurf der interaktiven Spark-Shell werden so viele Threads genutzt wie es logische Kerne gibt.

Für das Ausführen auf einem Cluster ist jedoch das getrennte Starten von einem Spark Driver (Master) und mindestens einem Executor (Slave, Worker) notwendig, vgl. Abb. 2.2. Der Spark-Driver führt das geschriebene Spark-Programm aus und verteilt u.a. die für die Map-Funktion benötigten Daten an die Executoren, welche die zugewiesenen Berechnungen dann durchführen.

Jeder Executor wird in einer eigenen Java Virtual Machine (JVM), also einem eigenen Prozess ausgeführt. Normalerweise, und so auch hier, wird für jeden Knoten ein Executor-Prozess gestartet, welcher mit der in der `SPARK_WORKER_CORES`-Umgebungsvariable definierten Anzahl an Threads arbeitet. Für den hier vorgestellten Benchmark wird `$SPARK_WORKER_CORES` identisch der Anzahl an Grafikkarten auf dem Knoten gewählt, sodass jeder Thread mit genau einer Grafikkarte arbeitet.

Ein RDD kann z.B. mit `parallelize` erstellt werden [6].



```
val distData = sc.parallelize( 1 to 10, 5 )
```

In diesem Beispiel wurden die Zahlen 1 bis 10 in ein RDD geladen. Das zweite Argument gibt die Anzahl an Partitionen an, die das RDD nutzen soll. Hier werden die Daten also in 5 Partitionen, in älteren Versionen von Spark auch Slices genannt, aufgeteilt. Eine Partition ist wie ein Task zu verstehen, der von einem der Executoren ausgeführt wird. Die Anzahl an Partitionen gibt damit eine Obergrenze für die Parallelisierbarkeit vor. Um Gebrauch von einer optimalen Lastverteilung zu machen sollte man eine Größenordnung mehr Partitionen haben, als die Sparkinstanz logische Kerne bzw. Threads zur Verfügung hat.

## 2.3 Konfiguration von Spark auf einem Slurm-Cluster

Viele Cluster stellen schon einen Task-Scheduler für Multinutzerumgebungen, wie z.B. PBS (Portable Batch System) oder SLURM (Simple Linux Utility for Resource Management), zur Verfügung, um Rechenzeit auf dem Cluster möglichst effizient und gerecht zu verteilen. Außerdem ermöglichen sie das verteilte Starten von Programmen, z.B. jene die mit MPI programmiert wurden. Der für die Benchmarks genutzte Cluster, siehe Kapitel 5.1, arbeitet mit SLURM.

Um Spark nutzen zu können, müssen zuerst Master- und Slave-Knoten gestartet werden. Damit alle gleichzeitig gestartet werden, sollten sie innerhalb desselben Jobs initiiert werden. Dies z.B. mit SLURMs `--multi-prog` Option realisiert werden. Diese erwartet als Argument einen Pfad zu einer Konfigurationsdatei, in der für jeden Rang ein möglicherweise anderes auszuführendes Programm angegeben ist.

Alternativ kann man auch anhand von der Umgebungsvariable `SLURM_PROCID` im Skript entweder einen Master-Knoten oder einen Slave-Knoten starten. Diese Möglichkeit wurde aufgrund der Übersichtlichkeit, weil damit alle Funktionalitäten in einem Skript vereint werden können, gewählt, siehe `startSpark.sh`  und `common/start_spark_slurm.sh`  [22].

Auf dem Master-Knoten wird der Spark Driver mit

```
1 "$SPARK_ROOT/bin/spark-class" org.apache.spark.deploy.master.Master \
2   --ip $(hostname) --port 7077 --webui-port 8080 &
```

gestartet. Alle anderen Knoten starten einen Executor-Prozess mit:

```
1 "$SPARK_ROOT/bin/spark-class" org.apache.spark.deploy.worker.Worker \
2   spark://$(scontrol show hostname $SLURM_NODELIST | head -n 1):7077
```

Hierbei wird vorausgesetzt, dass der Masterknoten, also jener für den `$SLURM_PROCID=0` ist, der erste Knoten in `$SLURM_NODELIST` ist. Dies wurde per assert vom Masterknoten aus auch geprüft und ist bei keinem der rund 50 Versuche fehlgeschlagen.

Wenn Spark gestartet ist, kann sich z.B. mit einer aktiven Eingabeaufforderung an den Master verbunden werden:

```
1 export MASTER_ADDRESS=spark://$MASTER_IP:7077
2 spark-shell --master=$MASTER_ADDRESS
```

Die Umgebungsvariable `MASTER_ADDRESS` wird automatisch vom `startSpark.sh`-Skript [22] gesetzt.

## 3 Rootbeer

Rootbeer[32] ist ein von Philip C. Pratt-Szeliga entwickeltes Programm welches das Schreiben von CUDA-Kerneln in Java ermöglicht. Zum aktuellen Zeitpunkt, August 2016, hat Rootbeer leider noch Beta-Status und wurde seit ca. einem Jahr nicht weiterentwickelt[29].

Der Rootbeerquellcode enthält Gerüste, um den Rootbeerkernel nicht nur nach CUDA, sondern auch nach OpenCL zu übersetzen, aber diese Funktionalität ist noch nicht fertiggestellt.

Andere Ansätze für Grafikkartenprogrammierung innerhalb von Java bzw. Scala stellen simple, teilweise direkt von den Headern der OpenCL-Spezifikation kompilierte, oder aber auch komplexere objektorientierte Java-Schnittstellen für OpenCL und CUDA zur Verfügung, so z.B. JogAmp JOCL[20], JOCL[19], JavaCL[7] und jCUDA[18] bzw. ScalaCL[8] und Firepile[28] für Scala. Diese erfordern jedoch die Kenntnis von OpenCL bzw. CUDA und der Nutzer muss sich selbstständig um Serialisierung und Host-GPU-Transfers kümmern. ScalaCL und Firepile sind außerdem auch noch in der Entwicklung.

Desweiteren gibt es fertige Bibliotheken, die auf Grafikkarten portierte Funktionen zur Verfügung stellen, welche jedoch nur eine beschränkte Einsetzbarkeit haben.

Eine Alternative zu Rootbeer stellt das ursprünglich von AMD entwickelte Aparapi[1], welches seit 2011 Open Source ist, dar. Wie in Rootbeer ist es möglich in Java Kernel zu schreiben, die von Aparapi in OpenCL übersetzt werden. Jedoch werden nur eindimensionale Felder unterstützt, im Gegensatz zu Rootbeer welches auch komplexe Objekte serialisieren kann.

Für die Nutzung von Rootbeer ist das Java SE Development Kit 7 und das NVIDIA CUDA Toolkit notwendig. Rootbeer funktioniert noch nicht mit JDK 8 und unterstützt offiziell nur JDK 6 [30], aber in den hier durchgeführten Beispielen gab es keine Probleme mit JDK 7.

Zuerst muss der Nutzer das `org.trifort.rootbeer.runtime.Kernel`-Interface implementieren und die kompilierten Klassen als JAR-Archive nochmals an den Rootbeer-Compiler geben, siehe Kapitel 4.3 und Kapitel 4.6. Es ist zu beachten, dass die an Rootbeer übergebene Datei auf `.jar` enden muss, insbesondere führen Dateinamen wie `gpu.jar.tmp` zu einer Fehlermeldung.

Der Rootbeer-Compiler nutzt dann Soot[17, 23], um den Java Bytecode der Kernel-Implementierung in Jimple zu übersetzen. Jimple ist eine vereinfachte Zwischendarstellung von Java-Bytecode in Drei-Address-Code mit nur 15 verschiedenen Befehlen. Java-Bytecode hingegen hat ca. 200 verschiedene Befehle. Der Jimple-Code wird dann analysiert und in CUDA-Quellcode übersetzt, welcher dann mit dem NVIDIA-Compiler kompiliert wird. All das geschieht automatisch, aber die Zwischenschritte kann man zur Fehlersuche unter Linux in `$HOME/.rootbeer/` einsehen. Die so erstellte cubin-Datei wird zusammen mit `Rootbeer.jar` dem JAR-Archiv des Nutzers hinzugefügt.

Die zweite große Vereinfachung, die Rootbeer zur Verfügung stellt, ist die Automatisierung des



Datentransfers zwischen GPU und CPU. Das besondere hierbei ist, dass Rootbeer die Nutzung von beliebigen, also insbesondere auch nicht-primitiven Datentypen erlaubt. Diese Datentypen serialisiert Rootbeer automatisch und unter Nutzung aller CPU-Kerne und transferiert sie danach auf die Grafikkarte.

Diese zwei Vereinfachungen obig machen die erste Nutzung von Rootbeer verglichen zu anderen Lösungen sehr einfach, sodass Rootbeer insbesondere für das Erstellen von Prototypen günstig ist. In Kontrast dazu ist es jedoch auch möglich sehr nah an der Grafikkarte zu programmieren. Dafür kann man mit Rootbeer manuell die Kernel-Konfiguration angeben, mehrere GPUs ansprechen, shared memory nutzen und auch über die `RootbeerGpu`-Klasse CUDA-Befehle wie `syncthreads` benutzen.

## 4 Implementation

### 4.1 Monte-Carlo-Algorithmen

Monte-Carlo-Algorithmen sind Algorithmen, die mit Hilfe von (Pseudo-)Zufallszahlen das gesuchte Ergebnis statistisch approximieren. Dafür werden Stichproben aus statistischen Verteilungen durch z.B. physikalisch begründete Abbildungen transformiert und jene Ergebnisse statistisch ausgewertet. Diese Art von Verfahren eignet sich z.B. zur Berechnung von sehr hochdimensionalen Integralen, die mit üblichen Newton-Cotes-Formeln nicht praktikabel wären. Eine andere Anwendung ist die Analyse von durch kosmischer Strahlung ausgelösten Teilchenschauern mit Hilfe von Markov-Ketten[26].

Monte-Carlo-Algorithmen sind als statistische Stichprobenverfahren schon länger bekannt, wurden aber erst mit dem Aufkommen der ersten Computer, z.B. dem ENIAC um 1947-1949, praktikabel[25]. Der Name, nach der Spielbank "Monte-Carlo", wurde von N. Metropolis vorgeschlagen und hielt sich seitdem. Der Vorschlag zu dieser Art von Algorithmus kam von John von Neumann, als man mit dem ENIAC thermonukleare Reaktionen simulieren wollte. Aber Fermi wird nachgesagt schon Jahre zuvor statistische Stichprobenverfahren in schlaflosen Nächten händisch angewandt zu haben und mit den überraschend genauen Resultaten seine Kollegen in Staunen versetzt zu haben.

Monte-Carlo-Verfahren sind inhärent leicht zu parallelisieren, da eine Operation, z.B. die Simulation, mehrere Tausend oder Milliarden Mal ausgeführt wird, jeweils mit unabhängigen Zufallseingaben. Eine Schwierigkeit besteht jedoch darin den Pseudozufallszahlengenerator (pseudo-random number generator - PRNG) korrekt zu parallelisieren. Das heißt vor allem muss man unabhängige Startwerte finden und an die parallelen Prozesse verteilen.

### 4.2 Berechnung von Pi

Um Pi zu berechnen wird Pi als Integral über eine Kreisfläche dargestellt. Dieses beschränkte Integrale lässt sich nun durch Monte-Carlo-Verfahren approximieren.

$$\pi = \int_{\mathbb{R}} dx \int_{\mathbb{R}} dy \underbrace{\begin{cases} 1 & |x^2 + y^2| \leq 1 \\ 0 & \text{sonst} \end{cases}}_{=: f(x,y)} \quad (4.1)$$

Da es programmatisch einfacher ist Zufallszahlen aus dem Intervall  $[0, 1]$  anstatt  $[-1, 1]$  zu ziehen,

wird das Integral über den Einheitskreis in ein Integral über einen Viertelkreis geändert:

$$\pi = 4 \int_0^\infty dx \int_0^\infty dy \begin{cases} 1 & |x^2 + y^2| \leq 1 \\ 0 & \text{sonst} \end{cases} \quad (4.2)$$

Das Integral aus Gl. 4.2 wird nun approximiert:

$$\mu_N = \langle f(\vec{x}_i) \rangle := \frac{1}{N} \sum_{i=1}^N f(\vec{x}_i), \quad \vec{x}_i \text{ uniform zufallsverteilt aus } \Omega := [0, 1] \times [0, 1] \quad (4.3)$$

Im Allgemeinen ist  $f$  eine beliebige Funktion, aber für die Berechnung von  $\pi$  ist  $f$  die Einheitskugel in 2D, vgl. Gl. 4.2. Gemäß dem Gesetz der großen Zahlen ist dann  $\lim_{N \rightarrow \infty} \mu_N = \pi$ . Für den algorithmischen Ablauf siehe Algorithmus 1

**Eingabe** : Anzahl an Zufallsziehungen  $N$

**Ausgabe** : Approximation von  $\pi$

sum  $\leftarrow$  0

**für**  $i \leftarrow 1$  **bis**  $N$  **tue**

$x \leftarrow \text{UniformRandom}(0,1)$

$y \leftarrow \text{UniformRandom}(0,1)$

**wenn**  $x^2 + y^2 < 1$  **dann**

        sum  $\leftarrow$  sum + 1

**Ende**

**Ende**

**Algorithmus 1** : Berechnung von  $\pi$  mittels Stichproben

Der Vollständigkeit halber seien kurz ein paar Worte zu den Rändern erwähnt; das betrifft die Zufallszahlen die entweder aus einem rechteckigen oder abgeschlossenen Intervall  $[0, 1]$  stammen können, d.h. der Vergleich schließt die Gleichheit mit ein oder nicht.

Aus der Integraltheorie ist klar, dass die Ränder ein Nullmaß haben und damit keine Rolle spielen. Aber für diskrete Verfahren könnte dies zu einer zusätzlichen systematischen Fehlerquelle führen, die das Fehlerskalierverhalten möglicherweise beeinträchtigt.

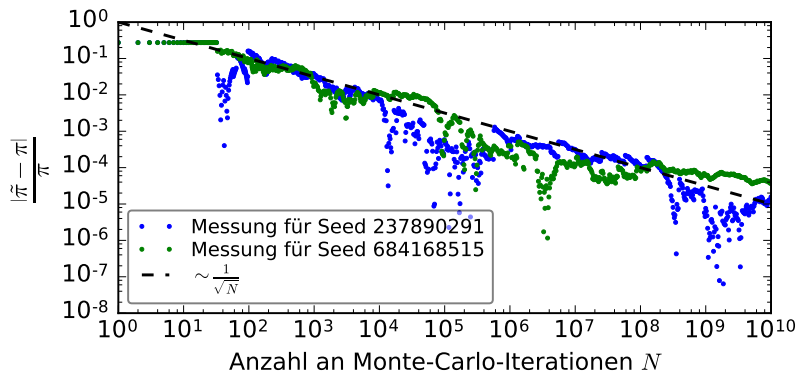
Am Beispiel von nur vier Zuständen für Zufallszahlen für den rechteckigen Fall, also  $x, y \in \{0, 0.25, 0.5, 0.75\}$ , sei dies einmal durchdacht. Damit ergibt sich

$$x^2 + y^2 = \{0, 0.0625, 0.125, 0.25, 0.3125, 0.5, 0.5625, 0.625, 0.8125, 1.125\} \quad (4.4)$$

Hier macht es aufgrund der begrenzten Anzahl an Zuständen, unter denen die 1.0 ohnehin nicht auftritt, keinen Unterschied ob man  $<$  oder  $\leq$  vergleicht, man erhielte  $\pi$  zu 3.6. Hinzu kommt aber, dass Zustände auf den Grenzen  $x = 0$  und  $y = 0$  liegen, sodass die Grenzen vierfach gezählt werden, wenn die Resultate des Viertelkreis mit vier multipliziert werden.

Man hat also ohnehin immer einen Diskretisierungsfehler von  $\mathcal{O}(\Delta x)$  wobei  $\Delta x$  die Diskretisierungslänge zwischen zwei Zuständen ist. Angemerkt sei, dass sich dies für Gleitkommazahlen komplizierter gestaltet.

Abschließend sei angemerkt, dass Monte-Carlo-Methoden dafür gedacht sind einen praktisch unerschöpflichen Raum stichprobenartig auszutesten, sodass Diskretisierungs- und Randfehler



**Abbildung 4.1:** Relativer Fehler auf die per Monte-Carlo-Integration berechnete Approximation für Pi für zwei verschiedene PRNG-Seeds.

ohnehin als vernachlässigbar angenommen werden. Wenn man merkt, dass es zu Diskretisierungsfehler wie obig an den Rändern kommt, oder man gar die Anzahl aller möglichen Zustände an Zufallszahlen erschöpft hat und sich die Approximation damit nicht mehr verbessern kann, sollte man über ein anderes Verfahren nachdenken oder den Zufallsgenerator anpassen und z.B. mit 128-Bit statt 32-Bit betreiben. Auch die maximale Periodenlänge von Pseudozufallsgeneratoren spielt hier eine Rolle!

Da die Monte-Carlo-Pi-Integration einer Mittelwertbildung entspricht, vgl. Gl.4.3, ist die statistische Unsicherheit durch die Standardabweichung des Mittelwerts  $\sigma_{\mu_N}$  gegeben.

$$\sigma_{\mu_N} = \frac{\sigma}{\sqrt{N}} \quad (4.5)$$

wobei  $\sigma$  die Standardabweichung der Stichprobe ist, vgl. Anhang A. Wenn  $f_i$  in einem beschränkten Intervall liegt, dann ist auch die Standardabweichung der Stichproben  $f_i$  beschränkt, sodass die Standardabweichung auf den Mittelwert  $\propto \frac{1}{\sqrt{N}}$  abnimmt, siehe Abb. 4.1.

### 4.3 Implementation des Kernels

In Listing 4.1 ist die Implementation der Pi-Berechnung zu sehen. Die Klasse implementiert das Rootbeer-Kernel-Interface, sodass sie von Rootbeer analysiert und auf die GPU portiert werden kann. Die `gpuMethod`-Methode ist jedoch auch normal vom Host aufrufbar.

Die erste Hälfte des Codes ist nur für das Abspeichern von Argumenten über den Konstruktor zuständig, da `gpuMethod` keine Argumente nehmen darf, um von Rootbeer erkannt zu werden. Jedes `MonteCarloPiKernel`-Objekt muss mit einem anderen Seed initialisiert werden, bevor `gpuMethod` aufgerufen wird, sonst rechnen zwei Threads mit denselben Zufallszahlenreihen und verfälschen damit die Ergebnisse.

Als Pseudozufallszahlengenerator wird ein simpler linearer Kongruenzgenerator mit 950706376 als Faktor genommen, vgl. [15, 16]

Auf `java.util.Random` wurde verzichtet, weil es in ersten Tests zu langsameren GPU-Berechnungen als auf dem Host führte. Es ist zu vermuten und in weiteren Tests zu beweisen, dass Zugriffe

auf externe Bibliotheken von Rootbeer nicht in CUDA-Code umgesetzt werden können, sodass die Argumente an den Host gesendet werden, die Funktion auf dem Host aufgerufen wird und die Ergebnisse wieder an die Grafikkarte gesendet werden.

```

1 import org.trifort.rootbeer.runtime.Kernel;
2 import org.trifort.rootbeer.runtime.RootbeerGpu;
3
4 public class MonteCarloPiKernel implements Kernel
5 {
6     private long[] mnHits;          // speichert Ergebnisse pro Kernelthread
7     private long   mRandomSeed;    // die Zufallssaat dieses Kernelthreads
8     private long   mnDiceRolls;    // Anzahl an Iterationen die zu tun sind
9
10    public MonteCarloPiKernel
11    (
12        long[] rnHits      ,
13        long   rRandomSeed,
14        long   rnDiceRolls
15    )
16    {
17        mnHits      = rnHits;
18        mRandomSeed = rRandomSeed;
19        mnDiceRolls = rnDiceRolls;
20    }
21    public void gpuMethod()
22    {
23        final int  randMax  = 0x7FFFFFFF;
24        final long randMagic = 950706376;
25        int dRandomSeed = Math.abs( (int) mRandomSeed );
26        final int dnDiceRolls = (int) mnDiceRolls;
27        long nHits = 0;
28        for ( long i = 0; i < dnDiceRolls; ++i )
29        {
30            dRandomSeed = (int)( (randMagic*dRandomSeed) % randMax );
31            float x = (float) dRandomSeed / randMax;
32            dRandomSeed = (int)( (randMagic*dRandomSeed) % randMax );
33            float y = (float) dRandomSeed / randMax;
34            if ( x*x + y*y < 1.0 )
35                nHits += 1;
36        }
37        mnHits[ RootbeerGpu.getThreadId() ] = nHits;
38    }
39 }

```

**Listing 4.1:** Implementation der Monte-Carlo-Integration für Pi

Weiterhin wird auch aus Performancegründen `mnDiceRolls` in eine lokale Variable zwischengespeichert. Dies führte auch zu einem signifikanten Geschwindigkeitsgewinn, vermutlich weil die zusätzliche Indirektion (Pointer Chasing) über die manuellen Speicherverwaltung von Rootbeer vermieden wird. Aus demselben Grund wird auch nicht direkt in `mnHits` der Zähler erhöht, sondern eine temporäre lokale Variable `nHits` genutzt. Letzteres reduzierte im Test eine Laufzeit von 2.25 s auf 0.5 s.

Auch sollte man darauf achten, wenn möglich Fließkommazahlen einfacher statt doppelter Genauigkeit zu nutzen, da Grafikkarten häufig mehr Berechnungseinheiten für einfache Genauigkeit besitzen.

Abschließend kann man sagen, dass ein erster Rootbeer-Kernel-Prototyp schnell geschrieben ist und teilweise durch Kopieren und Einfügen von normalen Java-Code möglich ist. Aber damit der Kernel auch wirklich schneller auf der Grafikkarte als auf dem Host ist, ist häufig genaues Wissen darüber wie Grafikkarten und auch wie Rootbeer arbeitet nötig.

## 4.4 Rootbeer-Kernelaufruf

Um einen Rootbeer-Kernel auf einer Grafikkarte auszuführen, muss zuerst ein Rootbeerkontext angelegt werden. Der Konstruktor des Rootbeerkontexts bereitet die Berechnungen vor, indem es die Rootbeerprogrammbibliotheken aus dem JAR-Archiv in ein temporäres Verzeichnis entpackt. In der originalen Version von Rootbeer war dies `$HOME/.rootbeer/`. Dies führt jedoch zu Problemen falls wie auf dem Testsystem, vgl. Kapitel 5.1, das Verzeichnis über alle Knoten gemeinsam verfügbar ist, oder aber auch schon falls Rootbeer thread- oder prozessparallel ausgeführt wird. Im eigenen Fork [31] wird dies mit Commit `6196bfd` gelöst, indem die benötigten Programmbibliotheken in ein Unterverzeichnis entpackt werden, welches sich aus Hostname, Prozess-ID, Thread-ID und Nanosekundenzeit zusammensetzt.

Das Rootbeer-Objekt `mRootbeerContext` stellt die Methoden `getDevices`, `createDefaultContext`, `getThreadConfig` und `run` zur Verfügung.

Die `run`-Methode nimmt eine Liste von Kernel-Objekten als Argument und führt diese auf der Grafikkarte aus. In Multi-GPU-Umgebungen wird die Grafikkarte jedoch automatisch gewählt, sodass diese Funktion vermieden werden sollte, stattdessen werden Teile des Quellcodes kopiert und abgeändert, um manuell eine der verfügbaren Grafikkarten auszuwählen, siehe Listing 4.3. Mit `getDevices` erhält man eine Liste von `GpuDevice`-Objekten die alle Informationen über die jeweilige Grafikkarte enthalten, so z.B. den maximalen Grafikkartenspeicher oder die Taktfrequenz. Außerdem können diese Objekte benutzt werden um Rootbeer-Kernel auf der ausgewählten Grafikkarte in Multi-GPU-Umgebungen auszuführen. In Listing 4.2 werden die Grafikkarteninformationen benötigt um die Seeds und die Anzahl an Iterationen pro Kernelthread zu berechnen.

Der angelegte Rootbeerfork [31] reduziert mit Commit `e48b9ae` auf Knoten mit sehr vielen Grafikkarten die Initialisierungszeit von `getDevices` um ca. 1s pro verfügbarer Grafikkarte, indem es die Aufrufe zu `cuCtxCreate` und `cuCtxDestroy` für jede Grafikkarte einspart, vgl. Abb. 5.2. Der einzige Nachteil ist, dass der freie Speicherplatz auf der Grafikkarte nicht mehr in `GpuDevice` verfügbar ist.

Als nächstes werden die Rootbeerkernelobjekte mit jeweiligem Seed initialisiert. Es werden so viele Kernelobjekte erstellt wie die Grafikkarte Threads parallel ausführen kann, um Pipelining voll auszunutzen. Die Tesla K20X z.B. hat 2688 CUDA-Kerne in 14 Shared-Multiprozessoren

(SMX), also 192 CUDA-Kerne pro SMX, vgl. Tabelle 5.2 auf Seite 20. Jeder SMX kann aber 2048 Threads parallel ausführen, um Latenzen durch Pipelining zu verbergen. In diesem Fall würden also 28672 Kernelobjekte erstellt werden.

Commit 9610c99 im eigenen Fork behebt diesbezüglich ein Problem, wo `getThreadConfig` nicht optimale Konfigurationen liefert. Und zwar wurden immer so viele Threads genutzt wie ein SMX parallel ausführen konnte. Das heißt je mehr SMX eine Grafikkarte hat, desto mehr Rechenleistung blieb ungenutzt, weil z.B. bei obigen Beispiel nur 2048 Threads anstatt der 28672 möglichen Threads gestartet wurden und damit Zugriffslatenzen nicht ausreichend überdeckt werden konnten.

Falls z.B. eine hohe Primzahl als Anzahl an Kernelobjekten benutzt wird, die also nicht in Blocks und Threads zerlegbar ist, dann kann man die Anzahl an Kernelthreads aufrunden und Rootbeer nimmt achtet darauf die überflüssigen Threads zu ignorieren.

```

1 class MonteCarloPi( iGpuToUse : Array[Int] = null )
2 {
3     private val mRootbeerContext = new Rootbeer()
4     private val mAvailableDevices = mRootbeerContext.getDevices()
5
6     def calc( nDiceRolls : Long, rSeed0 : Long, rSeed1 : Long ) : Double =
7     {
8         val lnWorkPerKernel = distributor.distribute(
9             lnWorkPerGpu(iGpu),
10            lnKernelsPerGpu(iGpu)
11        )
12        val tasks = lnWorkPerKernel.zipWithIndex.map( x => {
13            ...
14            new MonteCarloPiKernel(
15                lnHits(iGpu)      ,
16                kernelSeed        ,
17                nWorkPerKernel    /* iterations to do */
18            )
19        } )
20        runState = runOnDevice( mAvailableDevices.get( iGpuToUse ), tasks )
21        ...
22        runState.take()
23    }
24 }

```

**Listing 4.2:** Initialisierung der Rootbeerkernelobjekte, vgl. auch `multiNode/multiGpu/scala/MonteCarloPi.scala` [22]

Die Funktion `runOnDevice`, siehe Listing 4.3, führt eine Liste aus Kernelobjekten auf der ausgewählten Grafikkarte aus. Dafür wird auf dem ausgewählten `GpuDevice` ein Beschleunigerkontext, also z.B. ein CUDA-Kontext, angelegt. Das Argument an `createContext` gibt den voraussichtlich benötigten Grafikkartenspeicher an. Für Rootbeerkernel die dynamisch Speicher, z.B. durch einen `new`-Operator, allozieren, ist diese Angabe Pflicht. Leider funktioniert aber die automatische Speicherberechnung auch von Kernels ohne dynamische Allokationen nicht, sodass man in jedem Fall den benötigten Speicher angeben muss. Man benötigt jedoch tiefe Einblicke in die Funktionsweise von Rootbeer, um das abschätzen zu können, weshalb man einfach so viel

Speicher wie möglich als Reserve anfordern sollte.

Als nächstes werden die Anzahl für, die aus CUDA bekannten, Blöcke und Threads, die der Kernel umfassen soll, anhand der Anzahl an übergebenen Kernelobjekten festgelegt.

Mit der `buildState`-Funktion werden die Thread-Konfiguration zwischengespeichert, benötigte Serialisierungsspeicher auf Grafikkarte und Host angelegt und die CUDA-Binärdateien der vorkompilierten Kernel geladen und an den CUDA-Kontext gesendet.

Letztendlich werden mit `runAsync` alle benötigten Daten vom Host an die Grafikkarte übertragen und der Kernel in einem eigenen Hostthread gestartet. Zurückgegeben wird ein `GpuFuture`-Objekt mit einer `take`-Methode, mit der auf die Vollendung des Threads gewartet werden kann, siehe Listing 4.2. Dies ermöglicht das Nutzen mehrere Grafikkarten parallel aus einem Thread oder einem Prozess heraus, ohne dass sich der Rootbeernutzer sich selbst um Multithreading kümmern muss.

```

1  def runOnDevice(
2      device : GpuDevice,
3      work   : List[Kernel]
4  ) : Tuple2[ Context, GpuFuture ] =
5  {
6      val context = device.createContext( 128*1024*1024 )
7      val threadsPerBlock = 256;
8      val thread_config = new ThreadConfig(
9          threadsPerBlock, /* threadCountX */
10         1,                /* threadCountY */
11         1,                /* threadCountZ */
12         ( work.size + threadsPerBlock - 1 ) / threadsPerBlock, /*
13         blockCountX */
14         1,                /* blockCountY */
15         work.size         /* numThreads */
16     );
17     context.setThreadConfig( thread_config )
18     context.setKernel( work.get(0) )
19     context.setUsingHandles( true )
20     context.buildState()
21     val runWaitEvent = context.runAsync( work )
22     return ( context, runWaitEvent )
23 }
```

**Listing 4.3:** Ausführen der Rootbeerkernels auf einer ausgewählten Grafikkarte, vgl. auch `multiNode/multiGpu/scala/MonteCarloPi.scala` [22]

## 4.5 Kombination von Rootbeer mit Spark

Die Idee ist einfach, man nutzt die Map-Funktion eines RDD, um z.B. eine Liste an Seeds auf die Ergebnisse der Rootbeerkernel abzubilden, indem man die Logik aus Listing 4.2 kopiert. Dabei stellen sich jedoch mehrere Probleme:

Wenn je eine Spark-Partition auf eine Grafikkarte abgebildet wird und die genutzten Knoten mehrere Grafikkarten besitzen, dann muss Rootbeer threadsicher garantieren. Dies wurde erst mit Commit `aa3a0bc` und dem schon erwähnten Extraktionspfadproblem in Commit `6196bfd` im



Fork [31] gelöst. Rootbeer war nicht thread-safe, da es an vielen Stellen nicht benötigte Singletons und statische Variablen verwendet. Der Compiler selbst ist immer noch nicht thread-safe, nur die Laufzeitklassen wurden bisher angepasst.

Bei Knoten mit mehreren Grafikkarten muss darauf geachtet werden, dass nicht zwei Partitionen auf demselben Host derselben Grafikkarte zugeteilt werden, womit die Kernel nacheinander ausgeführt werden würden.

Im ersten Schritt werden hierfür so viele Partitionen wie logische Kerne über alle Knoten verfügbar sind bzw. so viele wie Grafikkarten benötigt werden, gestartet. Jede Partition sammelt Informationen darüber auf welchem Host er ist, wie viele Grafikkarten der Host besitzt und wie viel Leistung die jeweiligen Grafikkarten haben. Letztere Angabe wird benutzt um die Arbeit gleichmäßig proportional der Peak-Flops zu verteilen. Hierbei bezeichnet in Listing 4.4 `sc` den Sparkkontext.

Wichtig ist, dass diese Map auch wirklich ausgeführt wird und nicht nur lazy evaluiert wird. Außerdem muss darauf geachtet werden, dass bei einer erneuten Ausführung, z.B. für die spätere Verteilung der Grafikkarten, die Partitionen wieder auf demselben Host ausgeführt werden, damit die Zuordnungen der Grafikkarten stimmen. Dafür wird das RDD mit `cache` in den Arbeitsspeicher auf den jeweiligen Knoten gecacht und somit an den jeweiligen Host gebunden.

```

1 val cluster = sc.
2   parallelize( (0 until nPartitions).zipWithIndex ).
3   partitionBy( new ExactPartitioner( nPartitions, nPartitions ) ).
4   map( x => {
5       val devices = (new Rootbeer()).getDevices
6       val totalPeakFlops = devices.toList.map( x => {
7           x.getMultiProcessorCount.toDouble *
8           x.getMaxThreadsPerMultiprocessor.toDouble *
9           x.getClockRateHz.toDouble
10      } ).sum
11      /* return */
12      ( /* key */ InetAddress.getLocalHost.getHostName,
13        /* val */ ( devices.size, totalPeakFlops ) )
14    } ).
15    cache /* ! */

```

**Listing 4.4:** Ausschnitt aus `getClusterGpuConfiguration`, `vgl. multiNode/multiGpu/scala/TestMonteCarloPi.scala`

Nachdem Seeds und Grafikkarten aufgeteilt wurden, kann das gecachte RDD erneut auf die Ergebnisse der Monte-Carlo-Pi-Integration gemappt werden, siehe Listing 4.5.

```

1 val piTripels = cluster.zipWithIndex.map( x => {
2     val host          = x._1._1
3     val nGpusAvailable = x._1._2._1
4     val iRank         = x._2.toInt
5     val seedStart     = rankToSeedMapping(iRank)._1
6     val seedEnd       = rankToSeedMapping(iRank)._2
7     val iGpuToUse     = rankToGpuMapping(iRank)
8     val nGpusToUse    = rankToNGpusTouse(host)
9     val hostname      = InetAddress.getLocalHost.getHostName
10

```

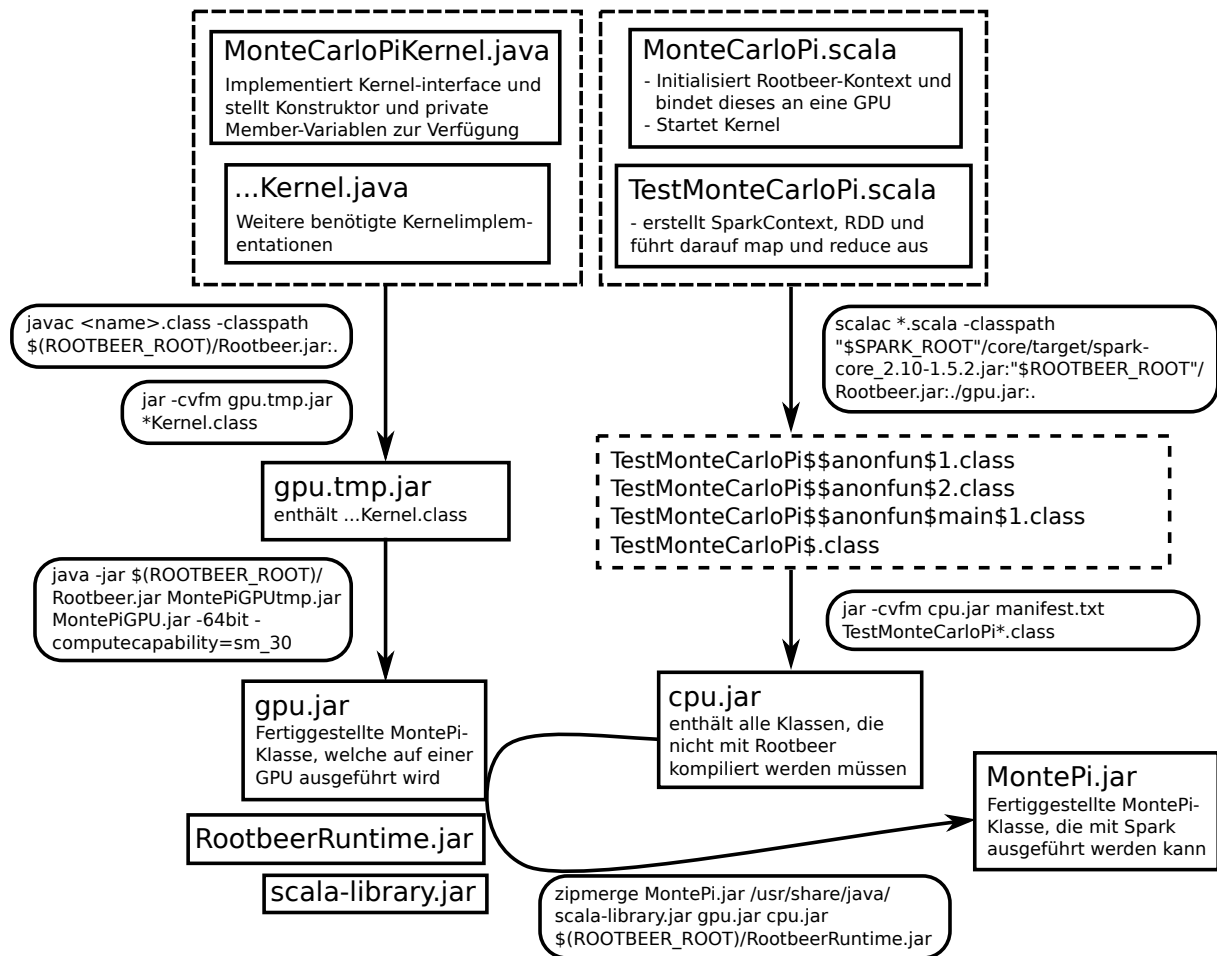
```
11     var pi = -1.0
12     if ( iGpuToUse < nGpusToUse )
13     {
14         var piCalculator = new MonteCarloPi(
15             Array( iGpuToUse ) /* Array of GPU IDs to use */ )
16         pi = piCalculator.calc( nRollsPerPartition( iRank ), seedStart, seedEnd
17             )
18     }
19     /* return */
20     ( pi, ( hostname, iRank, iGpuToUse ) )
21 } ).filter( _._1 != -1.0 ).collect
```

**Listing 4.5:** Start der Berechnung über Spark und Rootbeer, vgl. `TestMonteCarloPi.scala` [22]

## 4.6 Kompilierung

Die Kompilation wird aus Performancegründen und weil Rootbeer in manchen Fällen Problemen mit JAR-Archiven hatte, die Scala-Klassen beinhalteten, getrennt in die Kompilation der Rootbeerkernel und die Kompilation des Hostprogrammcodes, siehe Abb. 4.2.

Alle Rootbeerkernel werden zu einem JAR-Archiv gepackt und mit Rootbeer analysiert bzw. kompiliert.



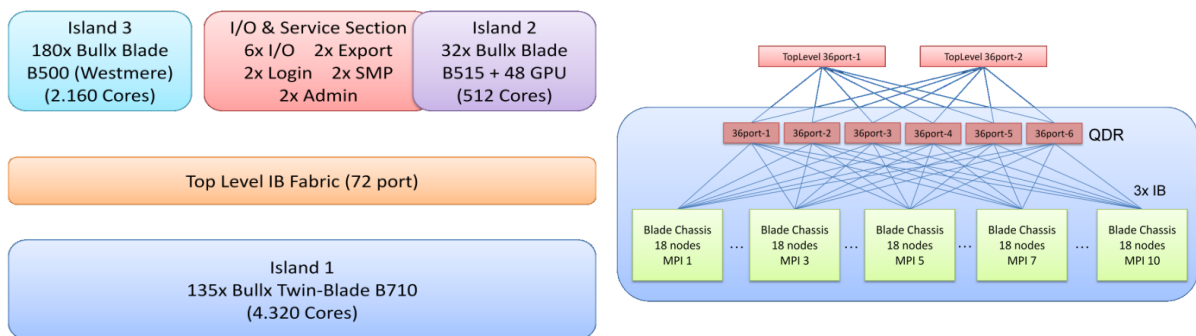
**Abbildung 4.2:** Kompilationsschema mit Kommandozeilenbefehlen und Zwischenstatussen.

Für das fertige JAR-Archiv ist es nicht nötig die Spark-Runtime reinzupacken, da das Archiv über Spark ausgeführt wird und somit alle Klassen vom Spark-Executor zur Verfügung gestellt werden. Jedoch müssen die Bibliotheken für Scala und für Rootbeer mit eingepackt werden. In Abb. 4.2 wird `RootbeerRuntime.jar` anstatt `Rootbeer.jar` gepackt. Dies ist eine Optimierung, die im Fork [31] mit Commit `6c7cba4` eingeführt wurde. `RootbeerRuntime.jar` enthält nur die für die Runtime benötigten Klassen und Abhängigkeiten und ist ca. 400 kB groß, während `Rootbeer.jar` auch den Rootbeercompiler und dessen Abhängigkeiten, also insbesondere auch die Soot-Bibliothek enthält und damit auf 14 MB anwächst.

## 5 Leistungsanalyse

### 5.1 Testsystem

Für die Skalierungstests wurde Taurus, einer der Hochleistungsrechner der TU-Dresden, benutzt. Der Bau der ersten Phase von Taurus war 2013 abgeschlossen[38]. Zum Zeitpunkt der Nutzung (2015/2016) waren alle Knoten von Phase 1, vgl. Abb. 5.1 links, in die 2015 fertiggestellt[35] Phase 2, siehe Abb. 5.1 rechts, integriert [37]. Gerechnet wurde je nach Verfügbarkeit auf Phase 1 oder 2 von Taurus, vgl. Tabelle 5.1.



**Abbildung 5.1: Links:** Übersicht Taurus Phase 1. **Rechts:** Schema der Topologie von Insel 2, auf der ausschließlich gerechnet wurde. Die Bilder wurden übernommen aus Ref.[38]

	Phase 1	Phase 2
Knoten	44	64
Hostnamen	taurusi2[001-044]	taurusi2[045-108]
Prozessor	2x Intel Xeon CPU E5-2450 (8 Kerne) zu je 2.10 GHz, MultiThreading deaktiviert, AVX, 2x 268.8 GSPFLOPS [11]	2x Intel(R) Xeon(R) CPU E5-2680 v3 (12 Kerne) zu je 2.50 GHz, MultiThreading deaktiviert, AVX2 insbesondere FMA3[10], 2x 480 GSPFLOPS[27]
GPU	2x NVIDIA Tesla K20X	4x NVIDIA Tesla K80
Arbeitsspeicher	32 GiB	64 GiB
Festplatte	128 GiB SSD	128 GiB SSD

**Tabelle 5.1:** Zusammensetzung Insel 2 von Taurus[36]

	<b>K20X</b>	<b>K80</b>
Chip	GK110	GK210
Takt	0.732 GHz	0.560 GHz
CUDA-Kerne	2688	4992
Speicher	6 GiB, GDDR5 384 Bit Busbreite	2 × 12 GiB, GDDR5 384 Bit Busbreite
Bandbreite	250 GB s <sup>-1</sup>	2 × 240 GB s <sup>-1</sup>
Theoretische Spitzenleistung	3935 GSPFLOPS	5591 GSPFLOPS
	1312 GDPFLOPS	1864 GDPFLOPS

**Tabelle 5.2:** Spezifikationen der Kepler-Grafikkarten von Taurus[12, 34]

Zu den Spitzenleistungen in Tabelle 5.2 sei angemerkt, dass jeder CUDA-Kern eine Operation einfacher Fließkommagenauigkeit oder eine 32-Bit Integeroperation berechnen kann und dass auf drei CUDA-Kerne eine Doppelpräzisionseinheit kommt, wodurch sich die DFLOPS berechnen. Die K80 hat außerdem einen Boost-Modus mit 0.875 GHz, also einer Leistungssteigerung von 1.56.

Für z.B. die Tesla K20X beträgt die Maximalleistung in der Berechnung von Fließkommazahlen einfacher Genauigkeit (Single Precision Floating Point Operations Per Second - SPFLOPS) im Verhältnis einer Multiplikation zu einer Addition:

$$0.732 \text{ GHz} \cdot 2688 \text{ CUDA-Kerne} \cdot 1 \frac{\text{FMA-Einheit}}{\text{CUDA-Kern}} \cdot 2 \frac{\text{SPFLO}}{\text{FMA-Einheit}} = 3935 \text{ GSPFLOPS} \quad (5.1)$$

Für den z.B. den Xeon E5-2680 v3 Prozessor beträgt die Maximalleistung:

$$2.50 \text{ GHz} \cdot 12 \text{ Kerne} \left( 1 \frac{\text{AVX ADD Einheit}}{\text{Kern}} + 1 \frac{\text{AVX MUL Einheit}}{\text{Kern}} \right) \cdot 8 \frac{\text{SPFLO}}{\text{AVX Einheit}} \quad (5.2)$$

$$= 480 \text{ GSPFLOPS} \quad (5.3)$$

## 5.2 Profiling

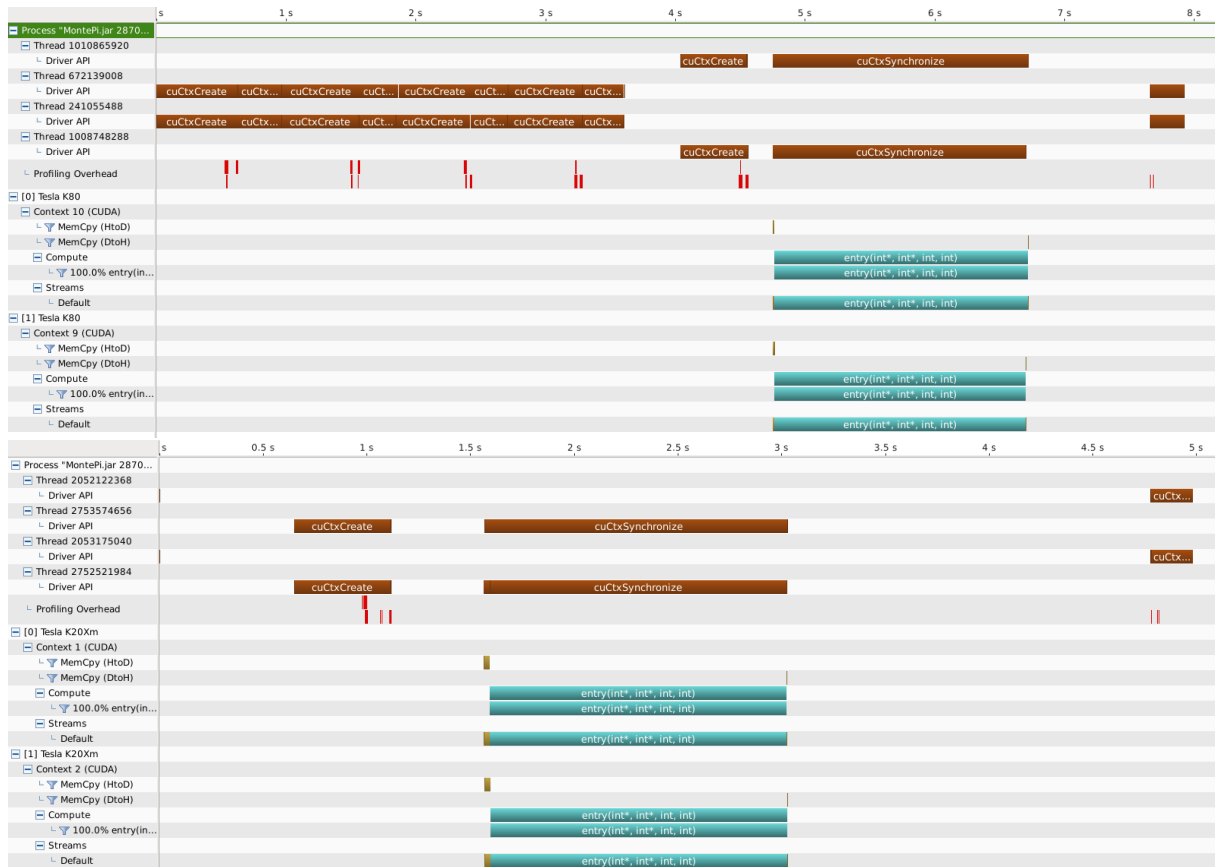
Profiling mit dem NVIDIA Visual Profiler ist auch mit Rootbeer möglich. Dafür muss als ausführbare zu analysierende Datei jedoch `java` mit den Argumenten zu der auszuführenden JAR-Datei angegeben werden:

```

1 sbatch --time=00:30:00 --nodes=1 --partition=gpu2 --cpus-per-task=4 --gres='gpu
  :4' <<EOF
2 #!/bin/bash
3 nvprof --analysis-metrics --metrics all \
4 -o $HOME/scaromare/MontePi/profilingDataMultiGpuScala.nvp%p \
5 $(which java) -jar $HOME/scaromare/MontePi/singleNode/multiGpu/scala/MontePi.jar
  $((2*14351234510)) 2
6 EOF
```

**Listing 5.1:** Erstellen der Profiling-Daten

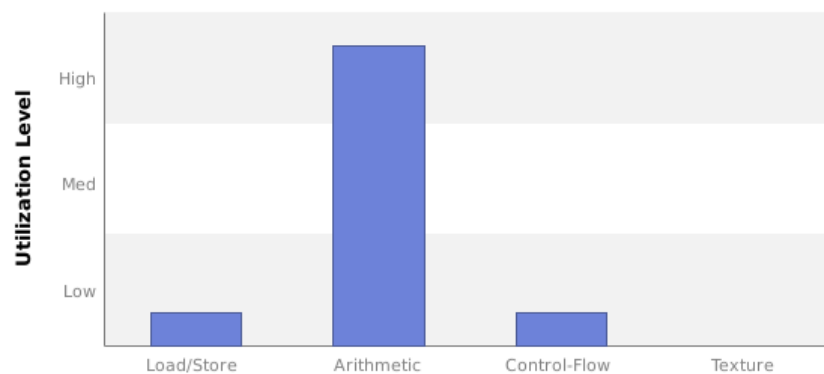
Die so erzeugten Zeitmessungen in `profilingDataMultiGpuScala.nvp` können mit `nvvp` ausgewertet und visualisiert werden.



**Abbildung 5.2:** Profiling der CUDA-API-Aufrufe. **Oben:** Vor dem `cuCtxCreate`-Fix im Rootbeerfork ausgeführt auf einem Tesla K80-Knoten. **Unten:** Nach dem Fix ausgeführt auf einem Tesla K20X-Knoten.

In Abb. 5.2 ist ein Programmmlauf mit zwei Grafikkarten zu sehen. In der originalen Rootbeer-version wurde für jede Grafikkarte erst ein `cuCtxCreate` aufgerufen, welches sehr zeitaufwändig ist, dann der freie Grafikkartenspeicher der Grafikkarte abgerufen und zuletzt der Kontext wieder mit `cuCtxDestroy` gelöscht, siehe auch Seite 13 und Abb. 5.2 oben. Dies wurde im eigenen Fork behoben, sodass nur noch ein Aufruf zu `cuCtxCreate` für die Grafikkarte, die auch wirklich benutzt wird, notwendig ist, siehe Abb. 5.2 unten. Die Gesamtlaufzeit im Beispiel wurde so von ca. 8 s auf 5 s reduziert.

Die Auswertung der Utilisierung in Abb. 5.3 zeigt, dass der Algorithmus die arithmetischen Einheiten fast voll auslastet. Es ist also noch Spielraum für Algorithmen die weniger rechenlastig sind und mehr Speicherzugriffe haben. Im Allgemeinen wird es sich aber nicht lohnen Algorithmen zu parallelisieren, die gleich viel Speicherzugriffe wie Berechnungen haben. Gut geeignet sind Algorithmen, bei denen die Anzahl an Berechnungen mit dem Quadrat der Anzahl an Speicherzugriffen oder noch schneller steigt.



**Abbildung 5.3:** Auswertung der Kernel-Utilisierung mit dem NVIDIA Visual Profiler des mit Rootbeer beschleunigten Monte-Carlo-Algorithmus.

### 5.3 Implementationsunterschiede

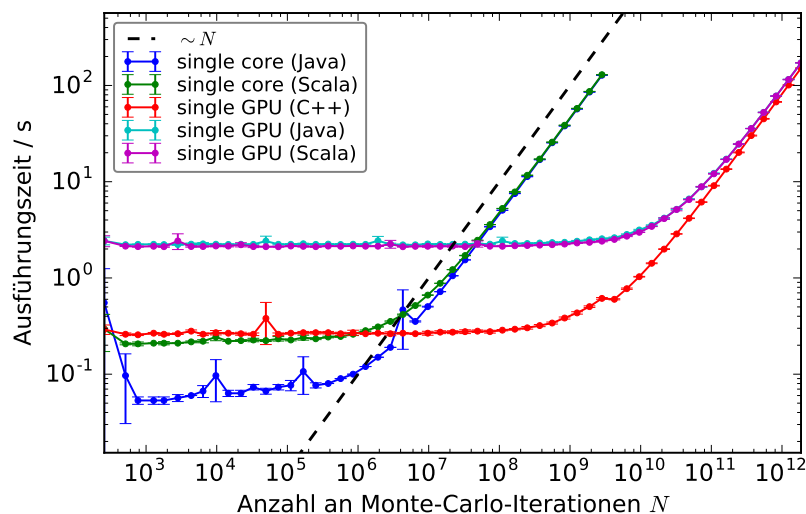
In Abb.5.4 wurde die Ausführungszeit von Monte-Carlo-Simulationen implementiert mit verschiedenen Programmiersprachen über die Anzahl an Monte-Carlo-Iterationen gemessen. Die Zeit wurde mit dem Linux `time`-Befehl gemessen. Dieser Befehl gibt real-, user- und sys-Zeit aus; die real-Zeit wird für den Benchmark genutzt.

Jede Implementation hat eine gewisse Initialisierungszeit. Bei der C++-Version beträgt diese jedoch nur knapp 70 ms, während die reine Java-Version schon ca. 220 ms benötigt. Die Nutzung von Scala erhöht dies auf 250 ms und die Nutzung von Rootbeer führt eine weitere Initialisierungszeit von 2 s ein, sodass für wenig Iterationen die Rootbeerversion bis zu 20x langsamer sind als die Hostprozessorversionen. Die Rootbeerinitialisierungszeit beinhaltet das Entpacken der benötigten Binärdateien, das Erstellen des CUDA-Kontextes und den Transfer der Daten zwischen GPU und Host. Die hier nicht gemessene Spark-Implementierung benötigte in Einzeltests noch mehr Zeit für die Initialisierung.

Erst für 10 Milliarden Iterationen beginnt die Initialisierungszeit der Rootbeer-Implementierungen im Vergleich zur Rechenzeit vernachlässigbar zu werden, sodass ab da die Lastenskalierung ein lineares Verhalten annimmt. Bei der C++-Version ist dies indes schon bei ca. 100 Millionen Iterationen der Fall.

Weiterhin ist aus dem Plot abzulesen, dass für große Lasten wie z.B. für drei Milliarden Iterationen die Versionen, die von Grafikarten Gebrauch machen, um einen Faktor 50 (Scala) bis 210 (C++) schneller sind als die Implementierungen, die auf einem Prozessor-Kern ausgeführt werden. Die theoretische Maximalleistung eines Kerns auf dem Intel Xeon E5-2680v3-Knoten beträgt 40 GSPFLOPS. Das heißt man würde einen Speedup von ca. 140 GSPFLOPS erwarten. Dass der Speedup höher ist liegt wahrscheinlich daran, dass Java keinen oder nur schlecht optimierten Gebrauch von AVX für die Monte-Carlo-Berechnung macht.

Bei ungefähr 2 s Ausführungszeit sind GPU- und CPU-Version gleich schnell. Das heißt schon ab einer Problemgröße, die mehrere Sekunden rechnen würde, lohnt sich eine Portierung auf Grafikkarten. Wenn auf dem Host AVX und alle Kerne benutzt werden, wie in der Messung hier nicht der Fall war, dann muss das Problem jedoch noch größer sein. Aber eine solche Portierung auf multithreaded AVX wäre womöglich noch komplizierter als das Schreiben eines



**Abbildung 5.4:** Benötigte Ausführungszeit der Monte-Carlo Pi-Berechnung in Abhängigkeit von der Anzahl an Iterationen. Getestet auf Taurusknoten `taurusi2095` und `taurusi2105`, also auf Tesla K80-Knoten, siehe Kapitel 5.1

Rootbeer-Kernels.

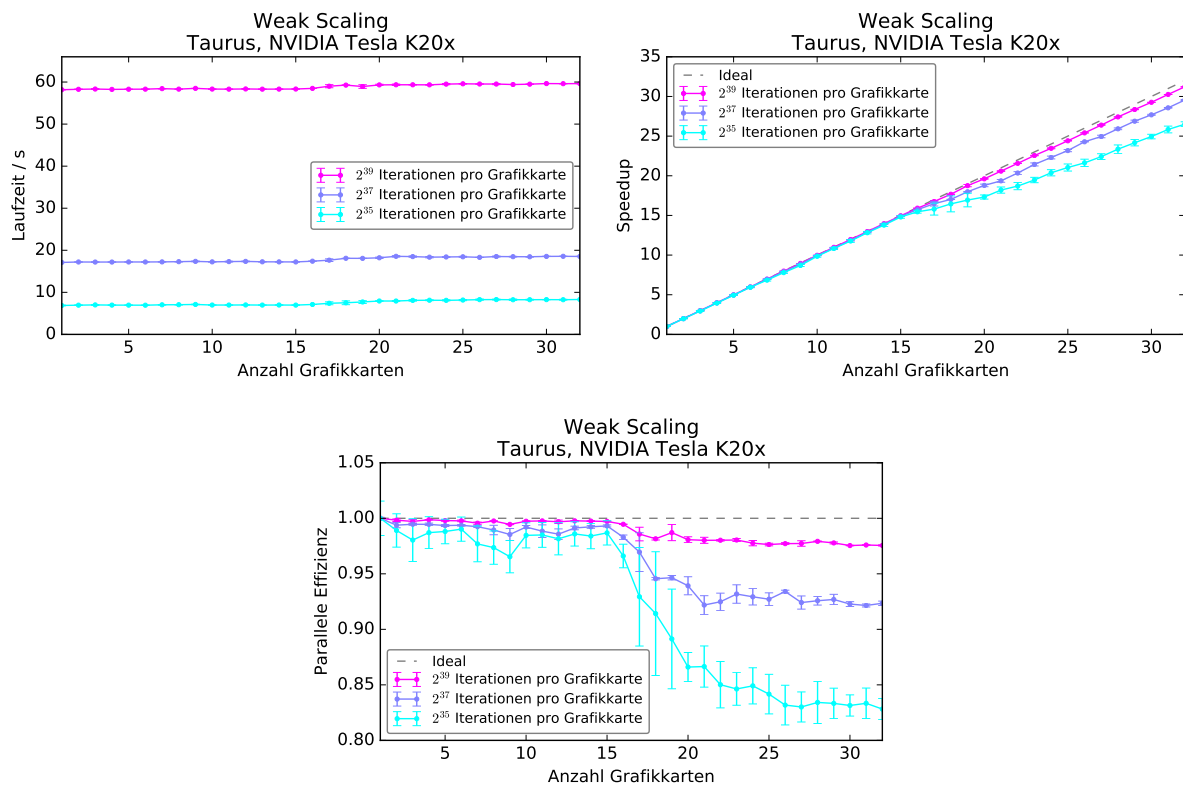
Wenn die Probleme aber genügend groß sind, dann rentieren sich Grafikkartenbeschleuniger sehr. Bei ungefähr vergleichbaren Prozessoren zu Grafikkarten und beiderseits sehr gut optimierten Programmen ist, wie in Tabelle 5.2 an den theoretischen Peak-FLOPS zu sehen, ein 10-facher Geschwindigkeitsgewinn realistisch. Dies gilt jedoch nur für einfache Genauigkeit. Für Fließkommaberechnungen doppelter Genauigkeit sind Grafikkarten je nach Modell ungeeignet.

## 5.4 Spark mit Rootbeer

In Abbildung 5.5 ist die Laufzeit über die Anzahl an Grafikkarten dargestellt. jeder Knoten hat zwei Tesla K20X-Grafikkarten. Die Anzahl an Iterationen wurde proportional zu der Anzahl an Grafikkarten erhöht, sodass jede Grafikkarte immer gleich viel Arbeit hat. Man erwartet also, dass die Ausführungszeit unabhängig von der Anzahl an Grafikkarten konstant bleibt, wie es auch der Fall ist.

Ab 16,17,18, in einem Einzelfall auch 19 Grafikkarten steigen jedoch die Laufzeiten plötzlich um ca. 1 s an, sodass der Speedup sich vom idealen Speedup entfernt, das heißt die parallele Effizienz nimmt ab. Die genaue Ursache hierfür wurde nicht gefunden. Mögliche Ursachen könnte Spark, Straggler, das Verbindungsnetzwerk oder vielleicht auch Rootbeer sein. Interessant wäre auch eine Wiederholung des Tests auf einem K80-System und jeweils mit mehr als 32 Grafikkarten, um eine weitere Erhöhung der Laufzeit bei Vielfachen von 16 Grafikkarten auszuschließen. Aus Rechenzeitgründen war dies jedoch zum aktuellen Zeitpunkt nicht mehr möglich.

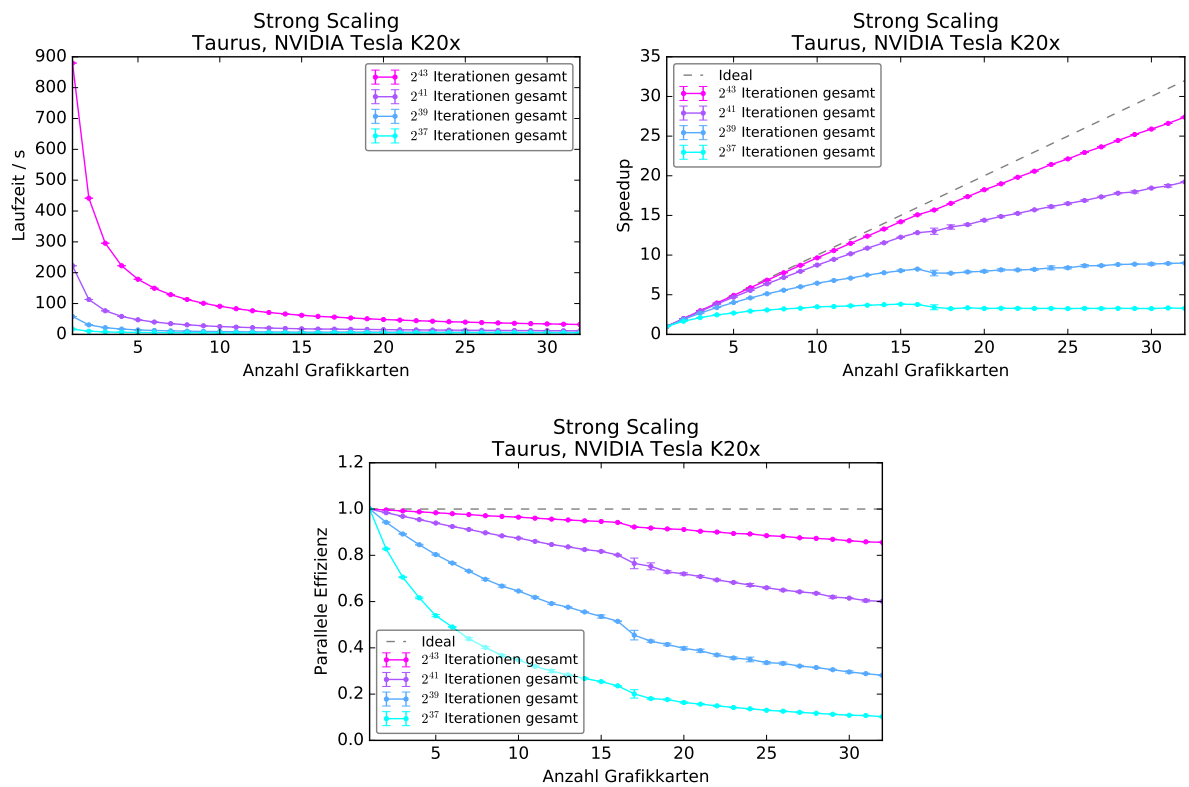




**Abbildung 5.5:** Benötigte Ausführungszeit der Monte-Carlo Pi-Berechnung in Abhängigkeit von der Anzahl an Grafikkarten gemessen auf Tesla K20X-Knoten. **Links:** Absolute Ausführungszeiten der Spark-Map-Funktion. **Rechts:** Speedup bezüglich der Laufzeit mit einer Grafikkarte. **Unten:** Parallele Effizienz

Weiterhin wurden die Benchmarks wiederholt für vier fixierte Totalproblemgrößen, sodass die Arbeit pro Grafikkarte mit Erhöhung der Parallelisierung abnimmt. Wie man in Abb. 5.6 sieht nimmt also mit Erhöhung der Grafikkarten die Laufzeit so lange proportional ab, bis der Großteil der benötigten Zeit nur noch aus der seriellen Initialisierungszeit besteht und sich somit der wirklich erreichte Speedup sättigt, vergleiche auch mit dem Amdahlschen Gesetz.

Für  $2^{37} = 137 \cdot 10^9$  Gesamtiterationen ist der Speedup bei ungefähr vier gesättigt. Es ist also nicht sinnvoll dieses Problem mit signifikant mehr als vier Grafikkarten zu parallelisieren. Wie aus Abb. 5.4 entnehmbar würde die CPU-Version ungefähr 500s benötigen.



**Abbildung 5.6:** Benötigte Ausführungszeit der Monte-Carlo Pi-Berechnung in Abhängigkeit von der Anzahl an Grafikkarten gemessen auf Tesla K20X-Knoten. **Links:** Absolute Ausführungszeiten der Spark-Map-Funktion. **Rechts:** Speedup bezüglich der Laufzeit mit einer Grafikkarte. **Unten:** Parallele Effizienz

## 6 Zusammenfassung

Im Rahmen dieser Belegarbeit wurde erfolgreich ein simpler rechenlastiger Monte-Carlo-Algorithmus mit Spark und Rootbeer auf einem Grafikkartencluster parallelisiert und gebenchmarkt. Die Benchmarkresultaten zeigen, dass es sich schon lohnen kann Probleme, die mehrere Sekunden auf einem x86-Prozessor benötigen würden, auf Grafikkarten zu parallelisieren. Die gemessenen Geschwindigkeitsgewinne sind für den Algorithmus vergleichbar mit den theoretischen Speedups, welche sich aus den theoretischen Peak-Flops berechnen. Dafür war jedoch zeitintensive Einarbeitung in den Rootbeer Quellcode nötig, um mehrere Bugs zu lösen. Ein angepasster Fork von Rootbeer befindet sich in [31] und die Beispiel Quellcodes zu Spark in Verbindung mit Rootbeer befinden sich in [22].

Ausgehend von dieser Arbeit wäre es interessant einen komplexeren Algorithmus mit mehr Speicherzugriffen und mehr Kommunikation innerhalb von Spark zu testen. Falls zwischen iterativen Rootbeerkernelaufrufen Daten auf der Grafikkarte wiederbenutzt werden können, dann wäre es sinnvoll Rootbeer so zu erweitern, dass man Speicher in der Grafikkarte behalten und für den nächsten Kernelaufruf wiederverwenden kann. Eine interessante Anwendung wäre z.B. das maschinelle Lernen auf einem Grafikkartencluster.

Weiterhin könnte man Straggler besser ausgleichen wenn man wie in Spark gewohnt ungefähr vier mal so viele Partitionen, d.h. Teilaufgaben, schedulen könnte als es Grafikkarten zur Verfügung gibt. Mit der hier vorliegenden Version ist dies nicht einfach möglich, da die Verteilung der Partitionen an die Grafikkarten statisch vor der Ausführung geschieht. Das Problem hierbei ist jedoch, dass mit CUDA nicht dynamisch festgestellt werden kann, welche Grafikkarten in Benutzung sind und welche nicht.

## Literaturverzeichnis

- [1] Aparapi - An open source API for expressing data parallel workloads in Java.  
<http://developer.amd.com/tools-and-sdks/rocm-zone/aparapi/>  
<https://github.com/aparapi/aparapi> [Online; accessed 2016-08-10],
- [2] APACHE: Apache Hadoop. <http://hadoop.apache.org/> [Online; accessed 2016-08-08],
- [3] APACHE: Apache MLlib. <http://spark.apache.org/mllib/> [Online; accessed 2016-08-08],
- [4] APACHE: Apache Spark. <http://spark.apache.org/> [Online; accessed 2016-08-08],
- [5] APACHE: GraphX Programming Guide. <http://spark.apache.org/docs/latest/graphx-programming-guide.html#graph-operators> [Online; accessed 2016-08-08],
- [6] APACHE: Spark Programming Guide. <http://spark.apache.org/docs/latest/programming-guide.html> [Online; accessed 2016-08-09],
- [7] CHAFIK, Olivier: JavaCL. <https://github.com/nativelibs4java/JavaCL> [Online; accessed 2016-08-10],
- [8] CHAFIK, Olivier: ScalaCL. <https://github.com/nativelibs4java/ScalaCL> [Online; accessed 2016-08-10],
- [9] CHIERICHETTI, Flavio ; KUMAR, Ravi ; TOMKINS, Andrew: Max-cover in Map-reduce. In: Proceedings of the 19th International Conference on World Wide Web. New York, NY, USA : ACM, 2010 (WWW '10). – ISBN 978-1-60558-799-8, 231–240
- [10] CORPORATION, Intel: Intel Xeon E5-2680v3 Prozessor Spezifikationen. [http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2\\_50-GHz?wapkw=e5-2680v3](http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz?wapkw=e5-2680v3) [Online; accessed 2016-05-08],
- [11] CORPORATION, Intel: Intel® Xeon® Processor E5-2400 Series. [http://www.intel.com/content/dam/support/us/en/documents/processors/xeon/sb/xeon\\_E5-2400.pdf](http://www.intel.com/content/dam/support/us/en/documents/processors/xeon/sb/xeon_E5-2400.pdf) [Online; accessed 2016-08-10],
- [12] CORPORATION, NVIDIA: Whitepaper - NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110/210. <http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf> [Online; accessed 2016-05-07], 2014

- [13] DEAN, Jeffrey ; GHEMAWAT, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI 2004, 2004, 137–150
- [14] DEAN, Jeffrey ; GHEMAWAT, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: Commun. ACM 51 (2008), Januar, Nr. 1, 107–113. <http://dx.doi.org/10.1145/1327452.1327492>. – DOI 10.1145/1327452.1327492. – ISSN 0001–0782
- [15] FISHMAN, George S. ; MOORE, Louis R.: A statistical evaluation of multiplicative congruential random number generators with modulus 231—1. In: Journal of the American Statistical Association 77 (1982), Nr. 377, S. 129–136
- [16] FISHMAN, George S. ; MOORE, Louis R. III: An Exhaustive Analysis of Multiplicative Congruential Random Number Generators with Modulus  $2^{31}-1$ . In: SIAM Journal on Scientific and Statistical Computing 7 (1986), Nr. 1, S. 24–45
- [17] GROUP, Sable R. ; GROUP, Secure Software E.: Soot - A framework for analyzing and transforming Java and Android Applications. <https://sable.github.io/soot/> [Online; accessed 2016-05-09],
- [18] HUTTER, Marco: Java bindings for CUDA. <http://www.jcuda.org/> [Online; accessed 2016-08-10],
- [19] HUTTER, Marco: Java bindings for OpenCL. <http://www.jocl.org/> [Online; accessed 2016-08-08],
- [20] JOGAMP: Java OpenCL. <http://jogamp.org/jocl/www/> [Online; accessed 2016-08-10],
- [21] KARAU, Holden ; KONWINSKI, Andy ; WENDELL, Patrick ; ZAHARIA, Matei: Learning Spark: Lightning-Fast Big Data Analytics. 1st. O'Reilly Media, Inc., 2015. – ISBN 1449358624, 9781449358624
- [22] KNESPEL, Maximilian: Codeverzeichnis zum Benchmark von Rootbeer über das Spark-Scala-Interface. <https://github.com/mxmlnkn/scaromare> [Online; accessed 2016-08-09], 2016
- [23] LAM, Patrick ; BODDEN, Eric ; LHOTÁK, Ondrej ; HENDREN, Laurie: The Soot framework for Java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop (CETUS 2011), 2011
- [24] MARTI, Othmar: Mittlerer Fehler des Mittelwertes - Vorlesungsskript. [wwwex.physik.uni-ulm.de/lehre/fehlerrechnung/node15.html](http://wwwex.physik.uni-ulm.de/lehre/fehlerrechnung/node15.html) [Online; accessed 2016-05-07],
- [25] METROPOLIS, Nicholas: The beginning of the Monte Carlo Method. In: Los Alamos Science 15 (1987), Nr. 584, 125–130. <http://jackman.stanford.edu/mcmc/metropolis1.pdf>
- [26] METROPOLIS, Nicholas ; ULAM, Stanislaw: The Monte Carlo Method. In: Journal of the American statistical association 44 (1949), Nr. 247, S. 335–341

- [27] MICROWAY: Detailed Specifications of the Intel Xeon E5-2600v3 Haswell-EP Processors. <https://www.microway.com/knowledge-center-articles/detailed-specifications-intel-xeon-e5-2600v3-haswell-ep-processors/> [Online; accessed 2016-08-10],
- [28] NYSTROM, Nathaniel ; WHITE, Derek ; DAS, Kishen: ScalaCL - OpenCL GPU Programming with Scala. <https://github.com/firepile/firepile> [Online; accessed 2016-08-10],
- [29] PRATT-SZELIGA: Rootbeer GPU Compiler - Java GPU Programming. <https://github.com/pcpratts/rootbeer1> [Online; accessed 2016-05-09], 2012
- [30] PRATT-SZELIGA: Issues with JRE 1.8. <https://github.com/pcpratts/rootbeer1/issues/175#issuecomment-61431951> [Online; accessed 2016-05-08], 2014
- [31] PRATT-SZELIGA, Maximilian K.: Rootbeer GPU Compiler - Java GPU Programming. <https://github.com/pcpratts/rootbeer1> [Online; accessed 2016-05-09],
- [32] PRATT-SZELIGA, Philip C. ; FAWCETT, James W. ; WELCH, Roy D.: Rootbeer: Seamlessly using gpus from java. In: High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESSE), 2012 IEEE 14th International Conference on IEEE, 2012, S. 375–380
- [33] SEO, S. ; YOON, E. J. ; KIM, J. ; JIN, S. ; KIM, J. S. ; MAENG, S.: HAMA: An Efficient Matrix Computation with the MapReduce Framework. In: Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, 2010, S. 721–726
- [34] SMITH, Ryan: NVIDIA Launches Tesla K20 & K20X: GK110 Arrives At Last. <http://www.anandtech.com/show/6446/nvidia-launches-tesla-k20-k20x-gk110-arrives-at-last> [Online; accessed 2016-05-08], 11 2012
- [35] STILLER, Andreas: Neuer Supercomputer an der TU-Dresden wird am Mittwoch eingeweiht. <http://www.heise.de/newsticker/meldung/Neuer-Supercomputer-an-der-TU-Dresden-wird-am-Mittwoch-eingeweiht-2639880.html> [Online; accessed 2016-05-08], 05 2015
- [36] ULF MARKWARDT, Guido J. u. a.: TU Dresden Internetauftritt - Hochleistungsrechner. <https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/SystemTaurus> [Online; accessed 2016-05-08], 2013-2016
- [37] WEB-TEAM, HPC: TU Dresden Internetauftritt - Zentrale Komponenten. <https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/SystemTaurus> [Online; accessed 2016-05-08], 2013-2016
- [38] WENDER, Jan: HRSK-II Nutzerschulung. <https://doc.zih.tu-dresden.de/hpc-wiki/pub/Compendium/SystemTaurus/HRSK-II-Nutzerschulung.pdf> [Online; accessed 2016-05-08], 5 2014

## A Standardabweichung des Mittelwertes

Dieses Kapitel richtet sich leicht nach Ref.[24]. Sei  $f \equiv (f_i)$  eine Folge von  $N$  Stichproben aus einer Zufallsverteilung mit einem Mittelwert  $\mu$  und  $\mu_N$  der empirische Mittelwert dieser Folge. Die Differenz  $\Delta_N := \mu - \mu_N$  wird nun abgeschätzt mit der Standardabweichung einer Folge von empirischen Mittelwerten  $(\mu_{N,k})$  die alle mit (sehr wahrscheinlich) verschiedenen Folgen bzw. Vektoren  $(f_i)$  gebildet seien.

Sei nun  $\langle \cdot \rangle_N : \mathbb{S} \times \mathbb{R}^N \rightarrow \mathbb{R}^N$ ,  $\langle (f_i)_k \rangle_N := \frac{1}{N} \sum_{i=0}^N f_{ik}$  der empirische Mittelwert und  $E(\cdot) : \mathbb{S} \rightarrow$

$\mathbb{R}$ ,  $E(x_k) := \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{k=0}^N f_k$  der Erwartungswert über eine unendlich Folge aus einem beliebigen statistischen Werten für begrenzte Folgen  $(f_i)$ . Hierbei ist  $\mathbb{S}$  der Raum der Folgen. Aus der Definition der beiden Mittelwerte wird klar, dass man die Summen und damit die Bildung der Mittelwerte vertauschen kann, sofern ein Grenzwert existiert. Dies wird in Gleichung A.3 angewandt.

$$\sigma_{\mu_N}^2 := E((\mu_N - \mu)^2) := E((\langle f_{ik} \rangle_N - \mu)^2) = E(\langle f_{ik} - \mu \rangle_N^2) \quad (\text{A.1})$$

$$= E\left(\left(\frac{1}{N} \sum_{i=0}^N (f_{ik} - \mu)\right) \left(\frac{1}{N} \sum_{i=0}^N (f_{ik} - \mu)\right)\right) = E\left(\frac{1}{N^2} \sum_{i=0}^N \sum_{j=0}^N (f_{ik} - \mu)(f_{jk} - \mu)\right) \quad (\text{A.2})$$

$$= \frac{1}{N} E\left(\frac{1}{N} \sum_{i=0}^N (f_{ik} - \mu)^2\right) + E\left(\frac{1}{N} \sum_{i=0}^N (f_{ik} - \mu) \frac{1}{N} \sum_{j=0, j \neq i}^N (f_{jk} - \mu)\right) \quad (\text{A.3})$$

$$= \frac{1}{N} E(\sigma_k) + \frac{1}{N} \sum_{i=0}^N \frac{1}{N} \sum_{j=0, j \neq i}^N \left( \underbrace{E(f_{ik})}_{=\mu} - \mu \right) \left( \underbrace{E(f_{jk})}_{=\mu} - \mu \right) = \frac{\sigma}{N} \quad (\text{A.4})$$

Man beachte, dass der Schritt in Gl.A.3-A.4 nur möglich ist, wenn  $f_i$  unabhängig von  $f_j$  ist, was hier der Fall ist, da der einzige abhängige Fall für  $i = j$  aus der SUMme rausgezogen würde, sodass  $E(ab) = E(a)E(b)$  anwendbar ist.

Man beachte, dass der zweite Summand nur durch die Mittelung über mehrere komplett verschiedene Versuchsreihen Null wird. Betrachtet man jedoch nur eine Versuchsreihe, dann hat der zweite Summand auch ein Skalierverhalten in Abhängigkeit zu  $N$ . Da aber das Vorzeichen wechseln kann, muss man den Betrag betrachten:

$$\frac{1}{N} \sum_{i=0}^N (f_i - \mu) \frac{1}{N} \sum_{j=0, j \neq i}^N (f_j - \mu) = \frac{1}{N} \sum_{i=0}^N \frac{1}{N} \sum_{j=0, j \neq i}^N (f_i f_j - \mu(f_i + f_j) + \mu^2) \quad (\text{A.5})$$

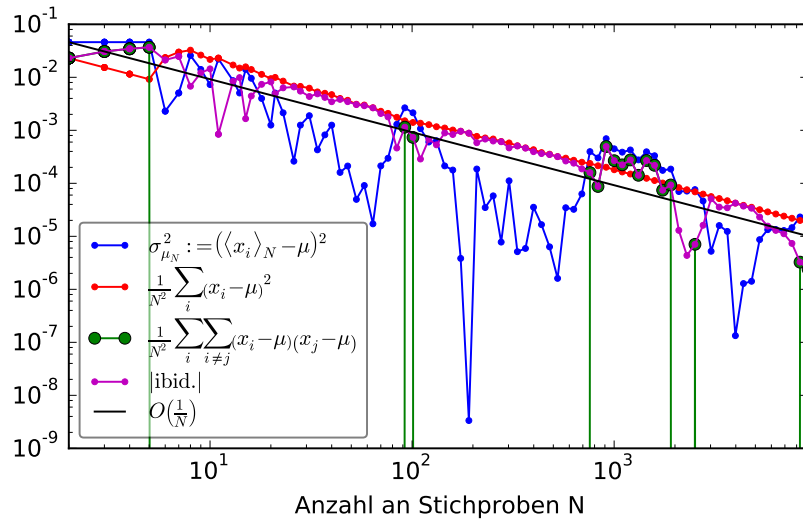
$$\approx \mu_N^2 - 2\mu\mu_N + \mu^2 \stackrel{\mu_N \approx \mu - \sigma_{\mu_N}}{\approx} \mu^2 - 2\sigma_{\mu_N}\mu + \sigma_{\mu_N}^2 - 2\mu^2 + 2\mu\sigma_{\mu_N} + \mu^2 = \sigma_{\mu_N}^2 = \frac{\sigma}{N} \quad (\text{A.6})$$

Schritt A.6 ist stark skizzenhaft und nicht mathematisch korrekt ausgeführt, wird aber gestützt durch empirische Auswertungen, vgl. Abb.A.1. In der Abbildung sieht man, dass sowohl der

erste Summand als auch der zweite invers proportional zu  $N$  skaliert. Ein wichtiger Unterschied ist jedoch, dass der erste Summand immer positiv ist, während das Vorzeichen des zweiten Summanden oszilliert, wodurch er über die Mittelung mit  $E(\cdot)$  gegen Null geht.

Interessant zu bemerken ist auch, dass der Graph der Standardvarianz des Mittelwertes  $\sigma_{\mu_N}^2$  aufgetragen über die Anzahl an einbezogener Stichproben einer Zufallsbewegung ähnelt, anstatt stochastisch zu streuen. Dies wäre nicht der Fall, würde man für alle  $N$  komplett neue Stichproben ziehen.

Weiterhin fällt auf, dass beide Summanden einer sehr glatten Geraden mit wenig Streuung folgen, während dies für  $\sigma_{\mu_N}^2$  nicht der Fall ist. Dies zeigt, dass es durchaus zu einer Fehlerrückmeldung durch den vorgeschlagenen zweiten Summanden kommt. Dies beeinträchtigt jedoch nicht die Fehlerskalierung mit  $\mathcal{O}\left(\frac{1}{N}\right)$ .



**Abbildung A.1:** Darstellung der Standardvarianz des Mittelwertes  $\sigma_{\mu_N}^2$  und der beiden in der Herleitung A.3 auftretenden Summanden über die Anzahl einbezogener Stichproben. Hierbei ist zu beachten, dass beim Vergleich von den Werten für die Stichprobenanzahl von  $N_1$  und  $N_2$  die ersten  $\min(N_1, N_2)$  Stichproben identisch sind.