Parallelisierung von Genetischen Algorithmen für Anwendungen der Finanzwirtschaft

Bachelorarbeit

im Studiengang BSc Informatik

Fernuniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Parallelität und VLSI

Amanjit Gill
Matrikel-Nr.: 7314361

Betreuer: Prof. Dr. Jörg Keller,
Patrick Eitschberger, MSc

Berlin, Dezember 2013

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problemdarstellung

Portfoliotheorie bezeichnet die Analyse des Investitionsverhalten am Kapitalmarkt, insbesondere am Aktienmarkt. Die Portfoliotheorie hat seit ihrer Einführung im Jahre 1952 einen großen Einfluss auf die Finanzwirtschaft ausgeübt. Sie hat sich sowohl im Bereich der Kapitalmarkttheorie, als auch in der Praxis etabliert. Sie wird somit bei der Planung und Durchführung von konkreten Investitionen im Kapitalmarkt als wichtiges Hilfsmittel verwendet [Mar91]. Ihr Erfinder Harry Markowitz wurde für seine Errungenschaften 1990 mit dem Wirtschaftnobelpreis ausgezeichnet[1].

Als Ausgangslage sollen die Entscheidungsmöglichkeiten eines Investors im Aktienmarkt betrachtet werden. Der Grundgedanke dabei ist die Frage, in welchem Verhältnis Risiko und Rendite stehen. Wenn sich ein Investor ausschließlich an der Rendite orientiert, müsste er auch ausschließlich in eine Aktie mit dem höchsten Erwartungswert[2] investieren. Dies bedeutet jedoch auch ein sehr großes Risiko, da der Aktienmarkt bekannterweise starken Schwankungen unterlegen ist.

Eine Betrachtungsweise des Investitionsverhaltens, welche nur die Rendite miteinbezieht, ist somit unzureichend, da das Risiko nicht berücksichtigt wird.

In der Realität wird sich ein Investor für ein Portfolio entscheiden, also für ein Menge von Aktien. Anschaulich wird dies durch die Redewendung 'Don't put all your eggs into one basket' wiedergegeben - Diversifikation als eine Investitionsstrategie.

[1] Zusammen mit Merton H. Miller und William F. Sharpe, *"for their pioneering work in the theory of financial economics"* [Nob].

[2] Dies setzt vorraus, daß sich dieser auch tatsächlich hinreichend genau abschätzen lässt.

Es ist die Errungenschaft von Harry Markowitz, als erster ein konkretes theoretisches Modell zu dieser Fragestellung erschafft zu haben - basierend auf gewissen Annahmen, die mehr oder weniger realistisch sind. Es ist mit diesem Modell möglich, sogenannte *'optimale Portfolios'* zu bestimmen, die in gewisser Weise ideal sind. Bei der Bestimmung dieser Portfolios gilt es ein mathematisches Optimierungsproblem zu lösen, welches mit Hilfe von modernen Rechnern gelöst werden kann. Um in der Praxis als Werkzeug für die Kundenbetreuung dienen zu können, muss diese Berechnung möglichst schnell erfolgen, idealerweise im Sekundenbereich.

Um mehr realistische Modelle abzubilden, sind im Laufe der Zeit Anpassungen an diesem klassischen Modell vorgenommen worden. Diese - praktisch gesehen - weitaus interessanteren Modelle bringen jedoch den großen Nachteil mit sich, dass i.d.R. keine geschlossenen Lösungsformeln mehr existieren, um optimale Portfolios zu berechnen.

Evolutionäre Algorithmen sind ein etablierter Ansatz, derartige Modelle am Rechner - trotz des Fehlens von geschlossenen Lösungsformen - hinreichend genau zu berechnen. Diese Ausarbeitung konzentriert sich dabei auf sog. genetische Algorithmen (abgekürzt: GA). Der erwähnte Aspekt der Laufzeitoptimierung spielt hier wieder eine sehr große Rolle, zumal evolutionäre Algorithmen oftmals großen Rechenaufwand benötigen[3].

Parallelisierung bezeichnet ein Programmierparadigma aus der Informatik, bei der (als ein Ziel) die nebenläufige Ausführung von Programmen zu einer Erhöhung der absoluten Ausführungsgeschwindigkeit führen soll. In dieser Ausarbeitung soll die Geschwindigkeit von GAs durch Parallelisierung[4], genauer einer Implementierung auf GPUs (Grafikkarten) erhöht werden. Diese ermöglichen die Ausführung von massiv daten-parallelen Programmen. Dabei wird das Programmmodell CUDA verwendet. Aktuelle Publikationen berichten über einen massiven Geschwindigkeitszuwachs bei der Verwendung von GPUs im Vergleich zu herkömmlichen Prozessoren ([PJS10], [SNK], [Ois+11]).

Ein Ziel dieser Ausarbeitung ist die komplette Neuentwicklung eines PGA Frameworks[5]) auf der CUDA Plattform und das Lösen der Fragestellung mit dieser. Als Vergleich soll eine serielle GA-Variante dienen, welche auf einer CPU läuft und mit Hilfe des etablierten GA Frameworks GALib [Wal08] in C++ implementiert wird. Ziel ist es, die Ausführungsgeschwindigkeit und Genauigkeit des seriellen GAs mit

[3]Vor allem auch im Vergleich zu geschlossenen Lösungsformen.

[4]Diese 'Parallelen GAs' werden gemeinhin als PGAs abgekürzt.

[5]Engl: Erweiterbare Programmierumgebung.

dem PGA zu vergleichen und zu klären, ob die Verwendung von CUDA-basierten PGAs für die gestellte Fragestellung einen Vorteil bringen[6].

1.2 Aufbau

Im Kapitel 2 wird auf die elementaren Grundlagen dieser Arbeit eingegangen, sowohl aus promblemspezifischer als auch aus fachlicher und technischer Sicht. Zunächst werden im Kapitel 2.1 Optimale Portfolios und die zugrundeliegenden Annahmen erläutert. Genetische Algorithmen werden im Kapitel 2.2 eingeführt. Im Anschluss daran werden im Kapitel 2.3 Erweiterungen am Markowitz-Modell vorgestellt, gefolgt von einer Beschreibung von Parallelisierungskonzepten (Kapitel 2.4) und dem CUDA Programmiermodell (Kapitel 2.4.2).

Das Kapitel 3 befasst sich mit der Umsetzung. Es müssen Entscheidungen bzgl. der GAs an sich, als auch der Parallelisierungskonzepte basierend auf der CUDA Architektur, getroffen werden. Auf grundsätzliche Entscheidungen wird in Kapitel 3.1 eingegangen. Wie man optimale Portfolios mit dem seriellen GA berechnen kann wird in Kapitel 3.2 erläutert. Anschließend wird die PGA Implementierung in CUDA vorgestellt (Kapitel 3.4).

Die Ergebnisse für beide PGAs werden im Kapitel 4 präsentiert. Dabei werden allgemeine Testfunktionen, die resultierenden Referenzkurven des klassischen Markowitz Modells und der Speedup[7] präsentiert.

Am Ende werden die Ergebnisse bewertet (Kapitel 5). Einer Betrachtung und Bewertung der Testergebnisse (Kapitel 5.1) schließt sich eine Analyse der umgesetzten PGA-Konzepte an.

Zu guter Letzt erfolgt ein Fazit und eine abschließende Bewertung von GAs und PGAs im Rahmen dieser Aufgabenstellung (Kapitel 6).

[6] Eine ursprüngliche Hauptmotivation dieser Ausarbeitung war es, eines der erweiterten Modelle neben dem klassischen Markowitz Modell als Berechnungsgrundlage zu verwenden. Aus zeitlichen Gründen kann nur auf die grundsätzliche Lösbarkeit mit dem präsentierten Ansatz eingegangen werden.

[7] Engl., Messung der Beschleunigung.

Kapitel 2

Grundlagen

2.1 Portfolio Optimierung

Zentrales Thema des Markowitz Modells sind Investitionen am Kapitalmarkt, insbesondere dem Wertpapiermarkt ([Mar91]). Es wird als *'Ein-Perioden-Modell'* bezeichnet. Dies bedeutet, dass die Investition zu einem Zeitpunkt t getätigt wird, und an dieser bis zum Zeitpunkt $t + 1$(z.B. einem Jahr) festgehalten wird. Dann wird der Ertrag begutachtet. Die folgenden Ausführungen und Definitionen orientieren sich an [Lue13].

Angenommen, ein Investor möchte einen Betrag X_0 investieren. X_1 sei der Betrag, den der Investor zum Zeitpunkt $t + 1$ besitzt. Der **Gesamtertrag** R sowie die **Rendite**[1] r lassen sich somit folgendermaßen definieren:

$$R = \frac{X_1}{X_0}, \quad r = \frac{X_1 - X_0}{X_0}, \quad R = 1 + r; \tag{2.1}$$

Angenommen, dieser Investor möchte den Betrag in ein Portfolio mit n Aktien investieren. Dann lässt sich die anteilige Investition (Gewicht) in eine Aktie i ($i = 1, 2, ...n$) als ω_i bezeichnen, und es gilt:

$$\sum_{i=1}^{n} \omega_i = 1 \tag{2.2}$$

In einem Modell ohne Leerkäufe gilt $\omega_i > 0$, in einem Modell mit Leerkäufen könnten einige ω_i negativ sein. Dabei unterstellt Markowitz, dass die einzelnen anteiligen Investitionen in den Wertpapieren beliebig teilbar sind.

[1] Im englischen: 'Rate of return', wird oftmals in Prozent angegeben

Wenn R_i der Gesamtertrag für eine Aktie i darstellt, so lässt sich der **Portfoliogesamtertrag** R_p sowie die **Portfoliorendite** r_p als

$$R_p = \sum_{i=1}^{n} \omega_i R_i, \quad r_p = \sum_{i=1}^{n} \omega_i r_i \tag{2.3}$$

darstellen. Selbstverständlich sind r_i und R_i zum Investitionszeitpunkt t unbekannt. Markowitz modelliert als Konsequenz die Rendite r_i einer Aktie als eine normalverteilte Zufallsvariable. Die Gesamtrendite r_p eines Portfolios wird dementsprechend als mehrdimensionale Normalverteilung modelliert.

Somit lässt sich der Erwartungswert μ eines Portfolios im Sinne des Erwartungswertes der mehrdimensionalen Normalverteilung bestimmen:

$$\mu = E(R_p) = \sum_{i=1}^{n} \omega_i E(R_i) \tag{2.4}$$

Der Mittelwert $\overline{x}$ einer Zufallsvariable X nähert sich bei beliebig hoher Wiederholung eines Experimentes dem Erwartungswert[2] Somit lässt sich der Erwartungswert nährungsweise durch die Ermittlung von $\overline{x}$ aus historischen Daten ermitteln.

2.1.1 Risiko

Als **Risiko** legt Markowitz die Varianz σ^2 der Rendite fest, welche wohlgemerkt als Streuungsmaß sowohl von Ausschlägen in 'positiver' als auch 'negativer' Form beeinflusst wird; Ein ungewöhnlich hoher Kursgewinn wird also als erhöhtes Risiko interpretiert.

Risikominimierung soll somit durch die Minimierung der Streuung erreicht werden. Die Standardabweichung σ (Wurzel der Varianz) wird in der Finanzwirtschaft in diesem Zusammenhang oftmals als *Volatilität* bezeichnet.

Wenn man mehr als eine Zufallsvariable betrachtet (wie hier bei einem Portfolio), dann gewinnt der Begriff der **Kovarianz** σ_{ij} an Bedeutung. Sie ist als Verallgemeinerung der Varianz ein Maß für die Gleichläufigkeit zweier Zufallsvariablen i und j.

$$\begin{aligned} \sigma_{\mathrm{ij}} &= E[(i - E(i)) \cdot (j - E(j))] \\ \sigma_{\mathrm{ij}} &= \sigma_{ji} \quad \textit{(Symmetrie)} \end{aligned}$$

[2]Sog. Gesetz der großen Zahlen, siehe [Wik13]

Hierbei steht E für den jeweiligen Erwartungswert. Man kann über den Wert der Kovarianz eine Aussage über die Korrelation der beiden Zufallsvariablen treffen

$$\begin{aligned}\sigma_{\text{ij}} &= 0 \Rightarrow \text{Beide Variablen sind unkorreliert}\\ \sigma_{\text{ij}} &> 0 \Rightarrow \text{Beide Variablen sind positiv korreliert}\\ \sigma_{\text{ij}} &< 0 \Rightarrow \text{Beide Variablen sind negativ korreliert}\end{aligned}$$

Eine normierte Version der Kovarianz ist der **Korrelationskoeffizient** $\rho_{A,B} := corr(A, B)$:

$$\rho_{i,j} = corr(i, j) := \frac{\sigma_{ij}}{\sigma_i \sigma_j}, \quad -1 \leq \rho_{i,j} \leq 1$$

Für eine n-dimensionale Normalverteilung X ist die Kovarianzmatrix Σ_X der entscheidende Verteilungsparameter. Diese ist symmetrisch und quadratisch und enthält die Varianzen aller Komponenten sowie die Kovarianzen aller möglichen Komponenten-Paare:

$$\Sigma_X = \begin{pmatrix} \sigma_1^2 & \cdots & \sigma_{1n} \\ \vdots & \ddots & \vdots \\ \sigma_{n1} & \cdots & \sigma_n^2 \end{pmatrix}$$

Übertragen auf das Portfolioproblem ist die Varianz der Portfoliorendite r_p der entscheidende Ausdruck. Diese gilt es zu minimieren:

$$\begin{aligned}\sigma^2 &= \sum_{i,j=1}^{n} \omega_i \omega_j \sigma ij \\ &= \omega^\top \Sigma_X \omega \quad \textit{(Matrixschreibweise)}\end{aligned}$$

Markowitz unterstellt dem Investor, risikoscheu zu sein - wenn ein Portfolio mit der gleichen Rendite, aber geringerem Risiko existiert, so wird sich ein Investor *stets* für das risikoärmere Portfolio entscheiden. Ein **effizientes Portfolio** liegt dann vor, wenn es kein anderes Portfolio gibt, welches für die gleiche Rendite μ ein geringeres Risiko σ_r^2 aufweist, bzw bei gegebenem Risiko σ_r^2 eine höhere Rendite μ ermöglicht (siehe dazu [Lue13]).

2.1.2 Diversifikation

In der Einleitung wurde bereits die Redewendung 'Don't put all your eggs into one basket' erwähnt. Im Falle von unkorrelierten Aktien kann dieser Diversifikationseffekt nachgewiesen werden.

Wie in [Lue13] demonstriert wird, kann man sich ein Portfolio vorstellen, welches aus n Aktien besteht, deren (erwartete) Rendite alle jeweils einen Durchschnitt von

	Szenario 1	Szenario 2	Szenario 3	Szenario 4
Eintrittswahrscheinlichkeit	0,25	0,25	0,25	0,25
Rendite der Aktie A	28%	12%	2%	2%
Rendite der Aktie B	-6%	2%	10%	18%

Tabelle 2.1: Beispielsszenario mit zwei Aktien

m und eine Varianz von σ^2 aufweisen. Jede Aktie i hat den gleichen Anteil am Gesamt-Portfolio, d.h. es gilt $\omega_i = 1/n$. Die gesamte Portfoliorendite beträgt dann

$$r = \frac{1}{n} \sum_{i=1}^{n} r_i$$

und deren Varianz

$$\sigma_r^2 = \frac{1}{n^2} \sum_{i=1}^{n} \sigma^2 = \frac{\sigma^2}{n}$$

Für $n \to \inf$ geht deren Grenzwert gegen 0. Luenberger weist darauf hin, dass die einheitliche Definition von m dieses konkrete Beispiel einschränkt. Dennoch erkennt man, dass durch Diversifikation im Falle von unkorrelierten Aktien eine Risikoreduktion[3] erreicht werden kann.

2.1.3 Portfolio aus zwei Aktien

Im Folgenden soll der Zusammenhang anhand eines Beispiels und einer grafischen Darstellung erläutert werden. Ein Portfolio soll aus zwei Aktien A und B bestehen, und die optimale Zusammensetzung/Gewichtung soll bestimmt werden. Für A und B gibt es historische Daten bzgl. des Erwartungswertes der Rendite, mit denen diese über den Mittelwert abgeschätzt werden können (Tabelle 2.1). Die Varianzen und Kovarianzen lassen sich nun einfach bestimmen (Tabelle 2.2).

Es werden 11 verschiedene Portfolios betrachtet, und jeweils die Portfoliorendite und das Portfoliorisiko errechnet (2.3). Die Gewichtung von A und B (ω_A und ω_B) wird für mehrere Szenarien angepasst, angefangen von einem Portfolio, welches zu 0% aus A und zu 100% aus B besteht - bis zu einem Portfolio, welches zu 100% aus A und zu 0% aus B besteht. Es ist nun möglich, die Varianz gegenüber dem Erwartungswert grafisch darzustellen, siehe Abbildung 2.1.

[3] im Sinne der Varianzreduktion.

Eigenschaft	Wert
Erwartete Rendite von A, r_A	11%
Erwartete Rendite von B, r_B	6%
Varianz von A, σ_A^2	114,14
Varianz von B, σ_B^2	80,80
Kovarianz von A und B, σ_{AB}	-88.8 (d.h. negativ korreliert)

Tabelle 2.2: Eigenschaften des Szenarios

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
ω_A	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
ω_B	1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1	0
μ_P	80.81	50.60	27.84	12.54	4.69	4.29	11.35	25.87	47.84	77.26	114.14
r_P	6.0	6.5	7.0	7.5	8.0	8.5	9.0	9.5	10.0	10.5	11.0

Tabelle 2.3: Portfolios aus zwei Aktien

Wenn man den oberen Schenkel der Kurve mit dem unteren Schenkel vergleicht, so ist erkennbar, das es Portfolios gibt, welche für das gleiche Risiko eine bessere Rendite versprechen. Man sagt, alle Portfolios auf dem oberen Schenkels **dominieren** alle Portfolios auf dem unteren Teil des Schenkels [Lue13]. Das Portfolio mit dem geringsten Risiko befindet sich am weitesten links, am Scheitelpunkt der Kurve.

Ein rationaler Investor würde sich immer für Portfolios auf dem oberen Schenkel entscheiden, und sich dann ausgehend von seinen Risikopräferenzen für ein Portfolio entsprechend weiter links oder rechts entscheiden. Dieser Bereich wird von Markowitz als **"efficient frontier"** bezeichnet [Mar52].

2.1.4 Portfolios mit mehr als zwei Aktien

Sobald man jedoch mehr als zwei Aktien berücksichtigt, ergibt sich aus der Menge der validen n-Kombinationen eine Fläche, siehe Abbildung 2.2.

Das *"efficient frontier"* wäre auch in diesem Falle entsprechend der obere Rand der Fläche, die Bestimmung dieser Fläche wird durch das Lösen eines Optimierungsproblems ermöglicht.

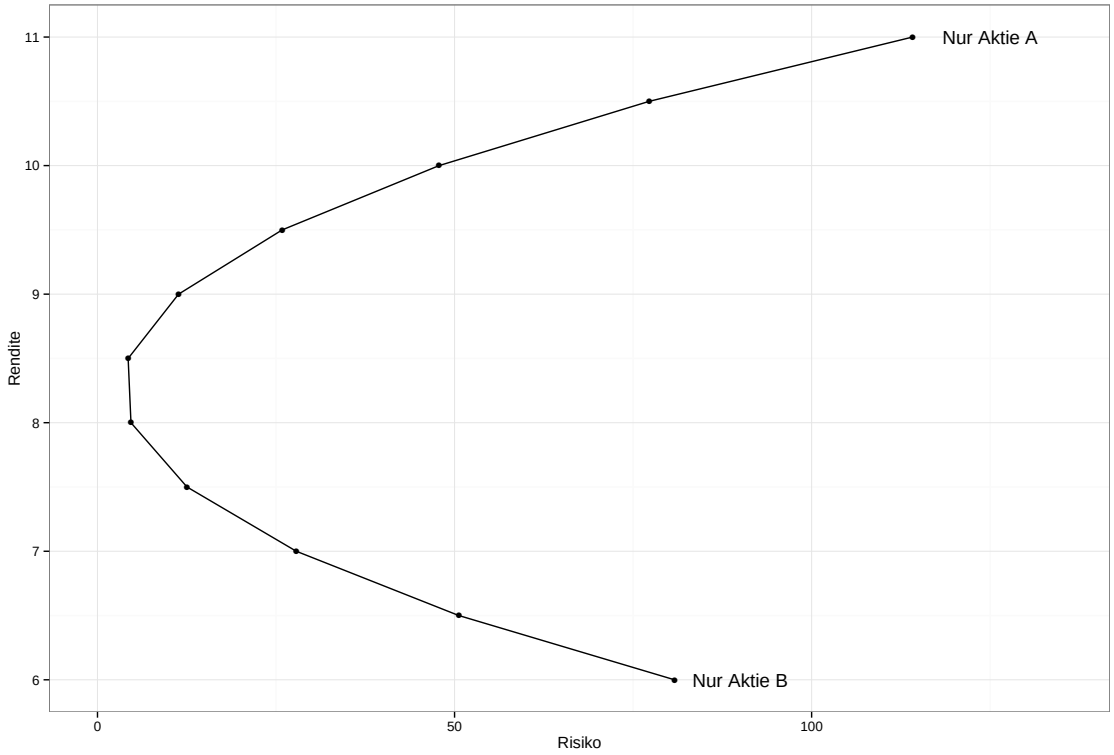


Abbildung 2.1: Portfolios aus zwei Aktien

2.1.5 Optimierungsproblem

Hier folgt nun die Formulierung des Optimierungsproblems für das klassische Markowitz Modell.

$$\min \sigma^2 \Leftrightarrow \min \omega^\top \Sigma \omega \qquad \textit{Minimierungsproblem}$$

$$R_p = \sum_{i=1}^{n} \omega_i \overline{r}_i = \overline{r} \qquad \textit{Nebenbedingung 1}$$

$$\sum_{i=1}^{n} \omega_i = 1 \qquad \textit{Nebenbedingung 2}$$

Dabei wird eine gewünschte Zielrendite $\overline{r}$ als *Nebenbedingung 1* fixiert.*Nebenbedingung 2* beinhaltet die Aufsummierung der Portfoliogewichte zu 1. Um die Effizienzkurve zu erhalten, kann das Problem für verschiedene Zielrenditen gelöst werden.

Eine erweiterte Formulierung des Optimierungsproblem ist möglich. Da man zu einem gegebenen Minimalrisiko gleichzeitig die Rendite maximieren möchte, kann man auch die Risiko Toleranz durch einen Faktor $q \in [0, \infty)$ modellieren, und im Grunde den Ausdruck $Portfoliorisiko - q \cdot Portfoliorendite$ optimieren, in Matrixform:

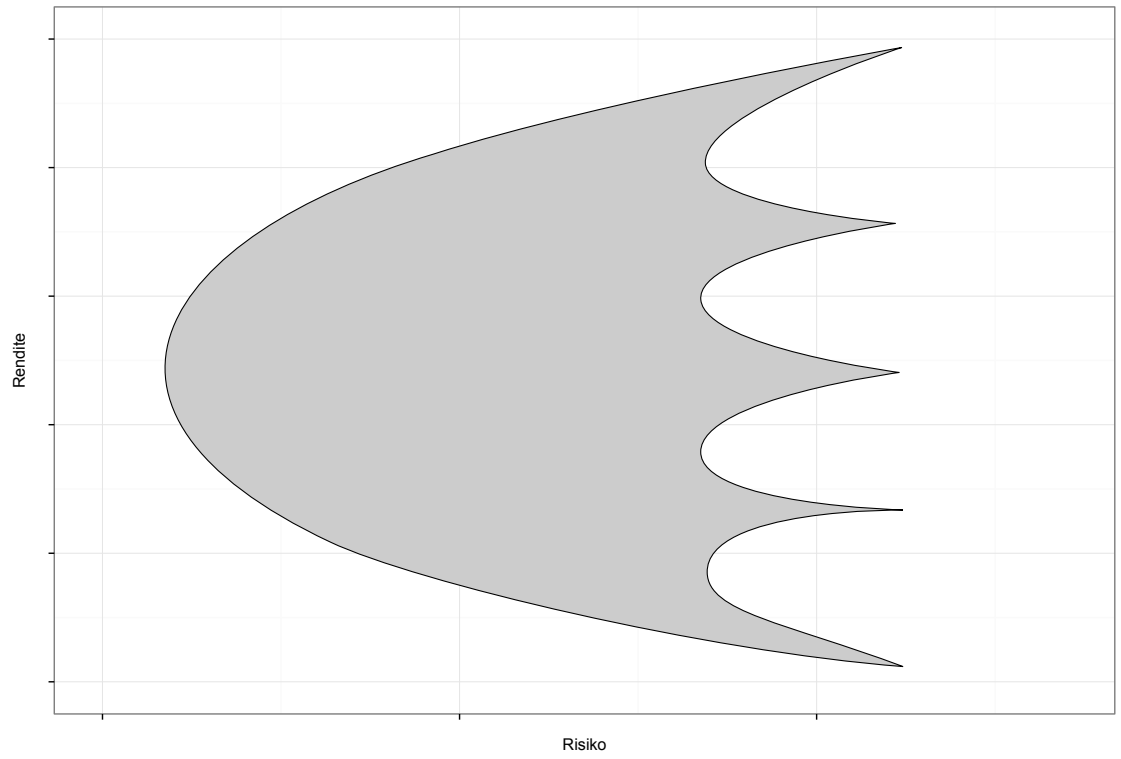


Abbildung 2.2: Das "Feasible Set"für Portfolios mit mehr als zwei Aktien, idealisierte Darstellung nach [Lue13]

.

$$\min \omega^T \Sigma \omega - q \cdot R_p^T \omega \qquad \textit{Kombiniertes Minimierungsproblem}$$

$$\sum_{i=1}^{n} \omega_i = 1 \qquad \textit{Nebenbedingung}$$

Diese gängige Vereinfachung des Optimierungsproblems wurde zum ersten Mal in [SAT93] beschrieben.

2.1.6 Lösungsmöglichkeit als quadratisches Optimierungsproblem

Ein quadratisches Optimierungsproblem ist ein Problem der Form

$$\min x^T Q x + c^T x \qquad \textit{QP Minimierungsproblem}$$

$$Ax = b \qquad \textit{QP Nebenbedingung 1}$$

$$x \geq 0 \qquad \textit{QP Nebenbedingung 2}$$

Das Markowitz-Modell lässt sich somit in dieser Form lösen, für $x = \omega$, $Q = \Sigma$ und $c = R_p$ sowie der Modellierung von *Nebenbedingung* als $Ax = b$. Dieser Ansatz ist wichtig, da dadurch eine exakte Lösung des klassischen Markowitz Modells möglich ist[4], welcher als Benchmark (bzgl. Exaktheit) für die GA Implementierung verwendet werden kann.

2.1.7 Kritik am klassischen Modell

Das Markowitz-Modell ist ein theoretisches Modell - somit wurden Annahmen getroffen, die natürlich nur begrenzt mit der Realität übereinstimmen:

- Renditen sind in der Realität nicht normal-verteilt.
- Alle Investoren handeln nicht ausschließlich rational und risiko-avers. Oftmals gibt es persönliche Entscheidungen und Präferenzen, die eine Rolle spielen.
- Auswirkungen der Investitionen auf den Kurs werden nicht berücksichtigt.
- Ein reines Ein-Perioden Modell ist relativ unflexibel.
- Es werden keine Transaktionskosten berücksichtigt.
- Das Verfahren ist relativ rechenintensiv.
- Die jeweiligen Aktienanteile sind in der Praxis nicht beliebig teilbar, oftmals müssen diese ganzzahlig sein
- Der Effekt der Diversifikation im Markowitz Modell kann aufgrund der genannten falschen Annahmen überbewertet werden, siehe [BSZ11].

2.2 Genetische Algorithmen

2.2.1 Evolutionäre Algorithmen und Metaheuristiken

Ein Evolutionärer Algorithmus (EA) ist ein Optimierungsverfahren, welches sich stark an der Evolution orientiert[5]. Sie gehören der Klasse der Heuristiken an

[4] Z.b. mit dem Statistikprogramm R und dem Modul `quadprog` [Mat].

[5] Man bezeichnet sich deshalb auch als natur-analoge Optimierungsverfahren.

und sind somit Suchverfahren. Suchverfahren finden oftmals keine exakte Lösung für ein Optimierungsproblem - sie sind jedoch so konzeptioniert, daß sie oftmals hinreichend gute[6] Lösungen finden. Da man sie für verschiedenste Probleme verwenden kann, bezeichnet man EAs zusätzlich auch als **Metaheuristiken**. Sean Luke fasst Metaheuristiken wie folgt griffig als Lösungsverfahren für *'I know it when I see it problems'* zusammen [Luk13]:

> 'Metaheuristics are applied to I know it when I see it problems [..]
> you don't know beforehand what the optimal solution looks like [..]
> you don't know how to go about finding it in a principled way [..]
> but if you're given a candidate solution to your problem, you can test it
> and assess how good it is' .

Es ist bei diesen Problemen also im vorhinein nicht klar, wie man gute Kandidaten finden kann. Gute Kandidaten werden jedoch als solche erkannt. Bekannte Metaheuristiken sind Simulated Annealing, Tabu Search, Genetische Algorithmen und Particle Swarm Optimization. Eine große Herausforderung bei der Suche nach Lösungen ist es dabei, nicht in einem lokalem Optimum "steckenzubleiben" ([Luk13], [Mar05]).

EAs versuchen sich dabei bei der Suche nach Kandidaten an den Prinzipien der Evolution zu orientieren.

Diese werden bei EAs wie folgt umgesetzt (Tabelle 2.4): Ein *Individuum* repräsentiert einen Lösungskandidaten, dessen Erbgut (*Gene*) werden als Datenstruktur repräsentiert. EAs können mit einer Menge von Kandidaten gleichzeitig arbeiten, diese wird als *Population* bezeichnet. Kandidaten können entsprechend ihrer *Fitness* bewertet werden, dieser Vorgang wird als *Selektion* bezeichnet. Gene können entsprechend ihres natürlichen Vorbildes mutiert werden (*Mutation*). Das Prinzip der Fortpflanzung wird durch Selektion und *Crossover*[7] umgesetzt.

Für weitere Informationen sei auf Standardliteratur, z.B. [Luk13] oder [Lip06] verwiesen. Die genannten Konzepte, welche Aktionen auf Genen durchführen, werden oftmals als sogenannte *Genetische Operatoren* bezeichnet.

[6] Oder sogar gute bis sehr gute.

[7] Aus zwei Eltern werden dabei i.d.R. zwei Kinder gebildet, es gibt jedoch weitere, flexiblere Modelle.

Natürliches Vorbild	Repräsentation in EAs
Individuum	Eine Lösung für das Problem
Gen	Datenstruktur, welche die Lösung repräsentiert (z.B. binäre Zeichenkette, Float-Array)
Population	Eine Menge von Individuen
Fitness	Die Qualität einer Lösung (Individuums)
Selektion	Auswahl von Individuen anhand ihrer Fitness
Mutation	Mutation des Genoms
Crossover	Rekombination des Genmaterials von mind. zwei Individuen zu neuem Genmaterial (Fortpflanzung)

Tabelle 2.4: Konzepte der Natur und die Repräsentation in EAs, nach [Luk13]

2.2.2 Der GA Algorithmus

Genetische Algorithmen sind in den 1970er Jahren von John Holland erfunden worden [Hol75]. Das Grundkonzept der evolutionären Algorithmen an sich war bereits seit Mitte der 1960er Jahre bekannt (siehe [Rec73]). Zunächst einmal stellt sich die Frage der Codierung der Gen-Information. Pro Individuum werden dabei n Gene-Loci unterstützt, klassischerweise handelt es sich dabei einen Binärstring, oder ein Float bzw. Integer Array[8]. Grundsätzlich sind jedoch beliebige Datenstrukturen, z.B. auch Baumstrukturen möglich, je nach GA Framework ist dies mehr oder weniger einfach möglich [9]. Gleichzeitig sind gültige Min. und Max. Werte für die jeweiligen Genausprägungen zu definieren.

Der allgemeine Ablauf eines genetischen Algorithmus ist in Abbildung 2.3 dargestellt. Anbei eine Beschreibung der einzelnen Schritte:

Populationserstellung

Hier wird eine Population von Individuen erstellt. Das Erbgut jedes Individuum wird dabei zufällig erzeugt.

Evaluation

Bei der Evaluation wird für jedes Individuum die Fitnessfunktion aufgerufen, um die Fitness zu ermitteln. Die Fitness wird meist als numerischer Wert oder

[8] Aus dem engl: "*String*" = Zeichenkette; "*Float*" = Gleitkommazahl; "*Integer*" = Ganzzahl.

[9] [Wal08] unterstützt z.B. benutzerdefinierte Genom-Typen; Es werden reichhaltige Basisklassen bereitgestellt.

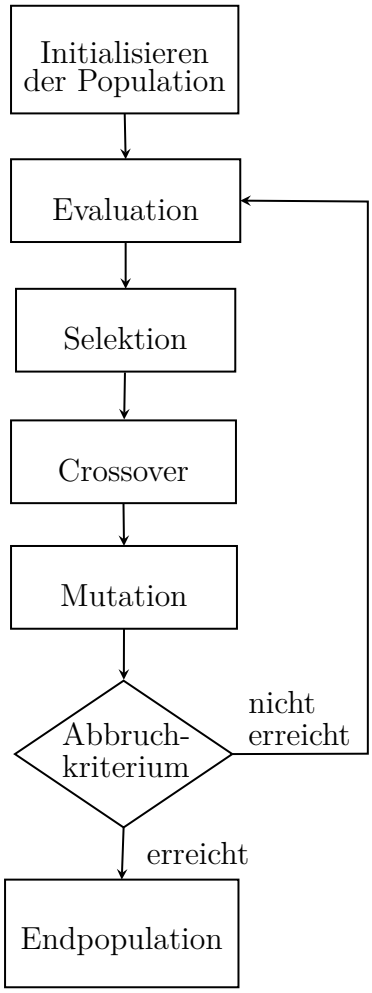


Abbildung 2.3: Allgemeiner Ablauf eines GAs

als Float Wert gespeichert. Es gibt gewisse Anforderungen an diese Funktion: Sie sollte sich wie eine Metrik verhalten, so dass die Individuen einer Population bzgl. ihrer Fitness vergleichbar sind.

Selektion

Um das Prinzip der natürlichen Selektion nachzuahmen, sollen aus der Menge der vorhandenen Individuen Eltern ausgesucht werden, die sich untereinander paaren können. Dabei spielt meist die Fitness der Individuen die entscheidende Rolle. Oft wird jedoch auch beim direkten Vergleich von Kandidaten mit einer gewissen Wahrscheinlichkeit der Verlierer ausgewählt, um die genetische Diversität zu erhalten. Am Ende dieses Schrittes ist eine Menge von Eltern bestimmt worden, die sich fortpflanzen dürfen.

Um geeignete Kandidaten zu bestimmen, sind verschiedene Verfahren gebräuchlich, u.A:

- **Roulette Wheel Selection**: Hier wird jedem Kandidat eine bestimmte Fläche auf einem virtuellen Roulette-Rad eingeräumt, proportional zu seiner Fitness. Anschließend wird das Roulette-Rad gedreht und ein Gewinner bestimmt. So steigt die Wahrscheinlichkeit, das sich fittere Kandidaten durchsetzen.
- **Stochastic Universal Sampling**: Dieses Verfahren ähnelt der Roulette Wheel Selection, hier wird die Aufteilung der Fläche analog durchgeführt. Anschließend werden n Individuen auf einmal selektiert. Wenn der Maximalwert der Fitness s beträgt, wird zufällig eine Position zwischen 0 und s/n ausgewählt, und das Element an dieser Position ausgewählt. Anschließend wird n Mal die Position um s/n Schritte inkrementiert. Hier kann sichergestellt werden, dass fitte Individuen mindestens einmal ausgewählt werden (siehe dazu [Luk13]).
- **Tournament Selection**: hier werden zufällig k Kandidaten ausgewählt, die jeweils gegeneinander antreten. Der Kandidat mit der größten Fitness gewinnt. Für $k = 1$ wäre das somit ein zufällige Auswahl. Oftmals wird $k = 2$ verwendet, so dass stets Pärchen verglichen werden.

Crossover (Rekombination)

Hier soll das Erbgut (Genom) von zwei Eltern[10] so kombiniert werden, das ein neues Genom entsteht (Nachwuchs im Sinne der Fortpflanzung). Das Genmaterial wird dabei nach verschiedenen Strategien rekombiniert. Im folgenden werden zwei Eltern betrachtet, und eines davon als $Vater$, das andere als $Mutter$ bezeichnet. Die beiden Nachwüchse werden als $Kind_1$ und $Kind_2$ bezeichnet.

- **1-Punkt-Crossover**: Es wird zufällig eine Trennstelle definiert, die für beide Eltern-Genome gilt. $Kind_1$ erhält die Gene vom $Vater$ bis zur Trennstelle, den Rest von der $Mutter$; $Kind_2$ erhält die Gene der $Mutter$ bis zur Trennstelle, den Rest vom $Vater$.
- **2-Punkt-Crossover**: Hier werden zufällig zwei Trennstellen definiert, an der für die Kinder jeweils vom $Vater$ auf die $Mutter$ umgeschaltet wird.
- **Uniform-Crossover**: Ausgehend von einem vorher (meist zufällig) gewähltem Mischverhältnis $n = [0, 1]$ wird die Verteilung der Genomquelle auf $Vater$ oder $Mutter$ Genom gesetzt. Ein Bitmuster mit entsprechender

[10] Es gibt auch Verfahren, die mit einem oder mehr als zwei Ursprungsgenomen arbeiten.

Länge und zu n passender Verteilung wird vorher erstellt, wobei 0 den $Vater$, und 1 die $Mutter$ codiert.

- **Arithmetic-Crossover**: Die ist bei einer Genom-Encodierung als reele Zahl möglich. Hier wird für jedes Genom x eine Linearkombination des entsprechenden Genoms von $Vater$ und $Mutter$ durchgeführt, um das Genom in $Kind_1$ und $Kind_2$ zu setzen. Dabei wird pro Genom x vorher ein zufällig Gewichtsfaktor $a = [0, 1]$ bestimmt.

$$Kind_1[x] = a \cdot Vater[x] + (1 - a) \cdot Mutter[x] \tag{2.5}$$

$$Kind_2[x] = (1 - a) \cdot Vater[x] + a \cdot Mutter[x] \tag{2.6}$$

Mutation

Mutation orientiert sich an der Biologischen Mutation auf Genomebene. Dabei werden Genome einer Genomsequenz zufällig geändert. Oftmals wird die Mutation nur auf die beim Crossover-Operator entstandenen neuen Kinder ausgeführt. Zu unterscheiden sind u.A.:

- **Bitstring Mutation**: Bei einer binären Repräsentation werden zufällig Bits invertiert.
- **Flip Bit Mutation**: Bei einer binären Repräsentation wird das komplette Genom invertiert.
- **Boundary Mutation**: Innerhalb einer optionalen Unter- und Obergrenze wird das komplette Genom zufällig neu gesetzt. Verwendbar bei einer Repräsentation durch reele und ganze Zahlen.
- **Uniform Mutation**: Hier wird für eine gewähltes Gen der Wert durch eine Zufallszahl innerhalb des gültigen Wertebereichs ersetzt. Verwendbar bei einer Repräsentation durch reele und ganze Zahlen.
- **Non-uniform Mutation**: Hier wird die Mutationsrate mit jeder Generation schrittweise bis 0 herabgesetzt. Verwendbar bei einer Repräsentation durch reele und ganze Zahlen.
- **Gaussian Mutation**: Hier wird eine normal-verteilte Zufallszahl zum ausgewählten Gen hinzu addiert (und gegebenenfalls an den Wertebereich angepasst). Verwendbar bei einer Repräsentation durch reele und ganze Zahlen.
- **Swap Mutation**: Hier werden zwei zufällig ausgewählte Genome vertauscht.

Abbruchkriterium

Das Verfahren springt zurück zum Punkt Evaluation, bis das vorher zu definierende Abbruchkriterium erfüllt ist. Jeder Durchlauf der kompletten Prozedur wird als **Generation** bezeichnet. Am Ende jeder Generation wird das beste Ergebnis i.d.R. zentral vorgehalten. Abbruchkriterien können sein:

- **Anzahl der Iterationen**: Das Verfahren wird nach einer gewissen Anzahl von Generationen abgebrochen.
- **Der durchschnittliche Fitnesswert in der Population**: Dieser Fitnesswert wird mit dem besten Fitnesswert in der jetzigen Population verglichen, und abgebrochen, sobald der Unterschied einen Schwellenwert unterschreitet.
- **Die Standardabweichung der Fitness in der Population**: Hier wird abgebrochen, sobald die Standardabweichung unter einem vordefinierten Schwellenwert sinkt.

Nach dem Terminieren des Verfahrens steht neben der Endpopulation außerdem das beste gefundene Ergebnisse bereit, welches ausgegeben werden kann.

Der Begriff **Elitismus** bedeutet, dass der fitteste Kandidat einer Population stets in die nächste Generation übernommen wird[11]. Als Standardliteratur zum Thema GA sei auf Goldberg [Gol89] verwiesen.

2.2.3 Erweiterte Populationsmodelle

Hier soll darauf eingegangen werden, welche gängigen Strategien bzgl. der Populationsersetzung existieren.

SimpleGA

Dieses von Goldberg vorgestellte Verfahren [Gol89] ersetzt die komplette Population durch Kinder.

Steady State GA

Hier werden nicht alle Individuen durch Nachwuchs ersetzt. Ein Großteil der Individuen (oft die fittesten) werden jedoch in die neue Generation übernommen. Oftmals werden iterativ Kinder erstellt und anhand ihrer Fitness jeweils entschieden, ob diese in die Population übernommen werden oder nicht.

[11] Oftmals wird der schlechteste Kandidat einer Population durch ihn ersetzt.

Island Modell

Hier gibt es mehrere Populationen, welche sich auf eigenen *Inseln* unabhängig voneinander entwickeln und nur begrenzt miteinander interagieren. Hier muss zunächst eine Topologie (z.B. Netz oder Ring) festgelegt werden, um zu entscheiden, welche Population mit welchem Nachbar kommunizieren kann. Ein **Migrations-Operator** übernimmt dann den Prozess des Austauschens von Individuen. Oft werden dabei die besten Individuen verschickt und die schlechtesten eigenen Individuen vom Nachbar übernommen. Dieses Verfahren spielt bei der Parallelisierung eine Rolle, da sich dieses Modell gut auf Multi-Prozessoren umsetzen lässt (Darauf wird im Kapitel 2.4.3 eingegangen).

2.3 Erweiterungen des Markowitz-Modells

Hier wird kurz auf die Erweiterungen des Markowitz Modells eingegangen, mit einem Schwerpunkt auf Publikationen, die das Problem mit Metaheuristiken gelöst haben.

Maringer ([Mar05], Kapitel 3) beschreibt eingehend Erweiterungen bzgl. Transaktionskosten und Ganzzahligkeit. Dabei wird eine Lösung mit Simulated Annealing vorgestellt. GAs werden im Rahmen der Vorstellung von EAs präsentiert.

Chang [Cha+00] erweitert das klassische Markowitz Modell um Kardinalitätsbeschränkungen und Ganzzahligkeit und liefert u.A. eine Lösungsmöglichkeit im Rahmen eines stark angepassten GA.

Baule [Bau08] liefert eine effiziente und präzise Lösung eines Modells, das Transaktionskosten berücksichtigt. Das Verfahren zeichnet sich gegenüber einer Lösung mit Metaheuristiken durch eine weitaus niedrigere Laufzeit und einer hohen Präzision aus.

Eine aktuelle Übersicht zur Konzeption von erweiterten Modellen und der Lösung mit Metaheuristiken bietet [SD06].

2.4 Parallelisierungskonzepte

2.4.1 Einführung

Parallelisierung bedeutet die parallele Ausführung von Programmen auf mehreren Ausführungseinheiten mit dem Ziel, eine Aufgabe gemeinsam besser zu lösen. Das Hauptziel ist dabei oftmals eine Verringerung der Ausführungszeit.

Der Leistungszuwachs neuer Prozessoren ist mit der Zeit gesunken [Sut05]. Als Konsequenz auf diesen Trend wurde die Anzahl der Ausführungseinheiten erhöht[12] und ein Programmierschwerpunkt auf die parallele Programmierung gelegt.

Bei der Charakterisierung von parallelen Hardware-Architekturen ist die sog. 'Flynnsche Klassifikation' hilfreich [Fly72]:

- SISD: *Single instruction, single data*: Klassische Computer mit einem Prozessor, keine Parallelverarbeitung.
- SIMD: *Single instruction, multiple data*: Vektorrechner (Cray), aber auch moderne Streaminginstruktionen von CPUs (SSE).
- MISD: *Multiple instruction, single data*: Praktisch nicht relevant.
- MIMD: *Multiple instruction, multiple data*: z.B. eng gekoppelte Mehrprozessorsysteme, oder lose gekoppelte Rechnerverbände.

MIMD-Systeme sind aus praktischen Gesichtspunkten heutzutage die wichtigste Rechnerklasse. Das wichtigste Programmiermodell für MIMD ist dabei SPMD: *Single program, multiple data*. Dabei wird ein Programm mehrmals mit jeweils verschiedenen Daten ausgeführt. Bei vielen Problemen ist es ausreichend, die Daten sinnvoll zu partitionieren und auf die vorhandene Rechenkapazität aufzuteilen.

Anhand der Speicherstruktur lassen sich diese Systeme wie folgt klassifizieren: *Shared Memory* und *Message Passing* [Kum+02]. Shared Memory bedeutet den Zugriff auf einen gemeinsamen Speicherbereich, während Message Passing die Kommunikation von Recheneinheiten über Nachrichten bedeutet.

[12] Heutzutage sind neue PCs üblicherweise bereits mit mehreren Rechenkernen ausgestattet.

Morderne PC Systeme sind Shared-Memory Systeme, die relevanten Programmier-APIs dafür sind Thread-APIs wie PThreads, OpenMP, Windows Threads, etc. Die wichtigste API für Message Passing ist MPI.

Oftmals werden Kombinationen verwendet, beispielsweise sind heutzutage Superrechner aus Rechnerknoten aufgebaut, welche über Message Passing kommunizieren. Die einzelnen Rechenknoten bestehen dabei aus Shared-Memory Multiprozessoren. Die modernsten Rechenknoten enthalten heutzutage zusätzlich eine weitere Art von Koprozessoren, die GPUs.

2.4.2 GPUs und CUDA

Eine GPU[13] ist ein Koprozessor für die Grafikdarstellung. Zum Ende der 1990er Jahre wurden diese immer leistungsfähiger und haben auch die Teile der 3D Bildgenerierung übernommen, die bisher der Prozessor übernommen hatte, nämlich rechenintensive Gleitkomma-Operationen auf Vektoren und Matrizen.

Somit stellte sich die Frage, wie man diese Leistungsfähigkeit auch für andere Anwendungen als der Grafikdarstellung verwenden kann (sog. GPGPU-Programmierung[14]. Die Verwendung der GPU war bisher nur über Umwege möglich (Zugriff über 3D APIs wie OpenGL, DirectX, dabei Verwendung von Texturen und Grafikspeicher), und auch die Genauigkeit dieser Operationen war begrenzt.

An der Universität Stanford wurde die allgemeine Frage, wie man ANSI C zum Zwecke der GPGPU Programmierung möglichst minimal erweitern könnte, in Form der "*Brook Streaming Language for GPUs*" [Buc+04] angegangen. Ian Buck, der maßgeblich an diesem Projekt beteiligt war, hat anschließend als Mitarbeiter bei NVIDIA CUDA erfunden.

2.4.2.1 Daten- und Taskparallelität

Algorithmen wie die der 3D-Bildgenerierung zeichnen sich durch massive 'Datenparallelität aus'. Daten-Parallelität wird im Gegensatz zu Task-Parallelität in [HS86] wie folgt definiert:

[13]Engl: graphical processing unit.

[14]engl: General purpose gpu programming, übersetzt: Allzweck-Berechnung auf Grafikprozessoreinheit.

> 'We call these algorithms data parallel algorithms because their parallelism comes from simultaneous operations across large sets of data, rather than from multiple threads of control'.

Beide Ansätze arbeiten somit mit mehreren Ausführungseinheiten und meist mit einem partitionierten Datenset. Der entscheidende Unterschied ist jedoch, dass der daten-parallele Zugriff simultan die *gleichen* Instruktionen auf jeweils verschiedene Daten durchführt, während der task-parallele Zugriff nicht gleichzeitig die selbe Operation ausführen *muss*.

2.4.2.2 Hardwarearchitektur

Es ist wichtig, kurz auf die Hardwareeigenschaften einzugehen, da diese (noch) notwendig sind, um GPUs effizient programmieren zu können. In [HP11] werden moderne GPUs als eine Menge von *'multithreaded SIMD processors'* bezeichnet und wie folgt zusammengefasst:

> "[..] we can see that GPUs are really just multithreaded SIMD processors, although they have more processors, more lanes per processor, and more multithreading hardware than do traditional multicore computers."

Dies bedeutet, dass die Architektur sowohl klassische MIMD als auch SIMD Eigenschaften aufweist:

MIMD Eigenschaft : Eine GPU besteht aus mehreren multithreaded SIMD Prozessoren, diese führen ihre Instruktionen jeweils unabhängig von anderen SIMD Prozessoren aus.

SIMD Eigenschaft : Ein einzelner dieser SIMD Prozessoren führt stets parallel die gleiche Instruktion mit mehreren Threads aus[15].

NVIDIA als Erfinder von CUDA bezeichnet multithreaded SIMD Prozessoren als *"Streaming Multiprocessors"*[NVI13], im folgenden mit SM abgekürzt. Seit der Einführung der G80 Architektur (Geforce 8) wurde diese Architektur beibehalten und

[15] Bei CUDA: sog. 'Instruktions-Warp'

ist als grundsätzliches Prinzip in CUDA verankert. Da CUDA als Programmierumgebung gewählt wurde, ist es im folgenden unumgänglich, gewisse NVIDIA-spezifische Eigenheiten zu erwähnen. GPUs enthalten i.d.R. 8 bis 20 SMs.

Ein einzelner dieser SMs besteht aus einer Menge von *"Stream Processors"* (SP), die man sich als ALUs vorstellen kann, deren primäre Aufgabe somit Rechenoperationen sind[16]. Somit erklärt sich auch die weitaus größere rohe Floating-Point Rechenleistung (GFLOPs) von GPUs im Vergleich zu handelsüblichen CPUs. Ein erklärtes Ziel ist es natürlich, diese entsprechend auszulasten. Das Verhältnis von arithmetischen Operationen zu Speicher Operationen wird als "Arithmetic Intensity" [WWP09] bezeichnet, und sollte bei GPUs entsprechend hoch sein.

Eine moderne, supraskalare x86-CPU besteht aus mehreren Kernen und verfügt über eine umfangreiche Cache-Hierarchie. Sie arbeitet mit *"out-of-order-execution"* und spekulativer Ausführung. Das Hauptziel liegt hierbei in der Maximierung der Ausführungsgeschwindigkeit eines Instruktionsstroms. CPUs sind auf **Latenz** optimiert.

Ein GPU arbeitet wiederum massiv datenparallel mit vielen (skalaren) ALUs und tausenden von Threads. Sie arbeiten heutzutage meist in-order und verfügen oftmals über keine Cache-Hierarchien. Sie sind auf **Durchsatz** optimiert.

Aus diesem Grunde gilt eine Faustregel, das GPUs eher für daten-parallele Verarbeitung, und CPUs eher für task-parallele Verarbeitung geeignet sind.

Cray, einer der Pioniere des Supercomputings, soll sich zu Lebzeiten kritisch über den Architekturansatz von vielen, kleinen, parallel arbeitenden Rechenkernen im Vergleich zu leistungsfähigen, wenigen Prozessoren geäußert haben:

> "If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"

Zusammenfassend lässt sich sagen, dass sich ideale Anwendungen für GPUs folgendermaßen auszeichnen sollten[17]:

1. **Massive Daten-Parallelität** *(Data Parallelism)*

[16] Der Begriff *"Programmierbarer Stream Prozessor"* war in der Forschung schon seit einiger Zeit ein Begriff [Kap+03], auch dort lag der Schwerpunkt auf daten-parallele Verarbeitung von medialen Datenströmen. Man kann die modernen GPUs somit als modernste Vertreter dieser Art betrachten.

[17] Siehe dazu `http://en.wikipedia.org/wiki/Stream_processing`

2. **Hohe Daten-Lokalität** *(Data Locality)*

3. **Hohe Rechen-Intensität** *(Compute/Arithmetic Intensity)*

2.4.2.3 CUDA Programmiermodell

CUDA[18] ermöglicht es, parallele C Programme für die Ausführung auf Grafikkarten zu schreiben. CUDA erweitert die C/C++ Sprache dabei minimal um gewisse Schlüsselworte/Keywords. Mit dem CUDA SDK wird ein Compiler (`nvcc`) mitgeliefert, der es ermöglicht Programme zu schreiben, die zum einen klassischen Code auf einer CPU ausführen und zum anderen Teile enthalten, die auf der Grafikkarte (parallel) ausgeführt werden[19]. Die Ausführung in diesem Kapitel orientiert sich an [SK10], [NVI13] und [Nic+08].

Der Ansatz ist, das Problem in Einzelteile zu zerlegen, die unabhängig voneinander parallel gelöst werden können und anschließend diese Einzelteile in weitere Untereinheiten zu zerlegen, die kooperativ parallel gelöst werden können [Nic+08].

GPUs werden meist auf Grafikkarten verbaut und verfügen über einen eigenen Speicher. Die GPU wird in CUDA als `device` bezeichnet. Sie sind meist mit einem schnellen Port[20] mit dem sogenannten `host` (also dem PC) verbunden. Der `host` und das `device` können sich den Adressraum somit i.d.R. nicht teilen.

Die wichtigste Funktionalität in CUDA sind die sog. Kernel Funktionen, welche durch das Schlüsselwort `__kernel` gekennzeichnet werden. Die Kernel Funktion wird parallel von mehreren leichtgewichtigen Threads parallel ausgeführt[21]. Die GPU führt zu einem Zeitpunkt nur eine Kernel Funktion aus. Aufgrund der immensen Anzahl von möglichen Threads wird Rekursion nicht auf jedem Gerät unterstützt. Function Pointer sind ebenso nicht möglich. Kernel können keine anderen Kernel aufrufen[22], sondern nur sog. Device-Funktionen, welche durch das Schlüsselwort `__device` gekennzeichnet werden.

[18] Engl: "Compute unified device architecture".

[19] Dabei verfügt das CUDA SDK über die notwendigen Laufzeitbibliotheken und interagiert mit der bestehenden Compiler und Linker-Chain.

[20] z.B. PCI Express.

[21] Diese sind sehr viel leichtgewichtiger als herkömmliche CPU Threads und teilen eigentlich nur noch ihren Namen mit ihnen.

[22] Die allerneuste GPU Generation unterstützt dies mittlerweile.

Ein weiteres Kernprinzip von CUDA sind die sog. Blöcke. Dies sind eine Menge von Threads, welche gemeinsam ausgeführt werden. Der Entwickler kann die Anzahl der Blöcke und die Anzahl der Threads pro Block bestimmen.

Der Anwender kann somit das Maß der Parallelität beim Aufruf eines Kernels bestimmen, wie in Listing 2.1 gezeigt wird. Der Anwender kann in diesem Falle `BLOCK_NUM` · `THREAD_NUM` Threads ausführen. Jeder Block enthält dabei `THREAD_NUM` Threads.

Listing 2.1: Einfacher Kernel und Kernel Aufruf

```
// Einfache Kernelfunktion
__global__ void simpleKernell(int myParam) {
  int absoluteThreadIdx = blockIdx.x * threadIdx.x;
    __shared__ float buffer[256];
}

// ... host code ...
bool result = <<<BLOCK_NUM, THREAD_NUM>>> simpleKernel(666);
```

Der Zugriff auf den aktuellen Block und Thread Index innerhalb des Kernels erfolgt über die reservierten Schlüsselwörter `blockIdx` und `threadIdx`. Es ist auch möglich für beide Parameter mehr als eine Dimension anzugeben, um Topologien aufzubauen.

Es ist wichtig zu wissen, das die Threads eines Blocks immer komplett auf einem SM ausgeführt werden. Diese Unterteilung ermöglicht es der GPU, Blöcke je nach Bedarf SMs zuzuordnen und auszuführen. Dieser Ansatz skaliert auch bei verschiedenen GPUs mit einer unterschiedlichen Anzahl von SMs. Dabei wird stets ein sog. Warp ausgeführt, das ist eine Ausführungseinheit von Threads. Alle Threads innerhalb eines Warps führen gleichzeitig die *gleiche* Instruktion auf ihren Daten aus (SIMD Eigenschaft). Ein Warp ist dabei meist ein Vielfaches von 32.

Dieses Wissen ist für die korrekte Ausführung von Programmcode nicht relevant, jedoch für die Entwicklung von hochperformanten Programmcode. Im Falle eines bedingten Programmsprungs, der nur von einem Thread eines Warps genommen wird, bleiben sämtliche anderen Threads stehen, bis der bedingte Codepfad komplett abgearbeitet wurde.

Es gilt die Faustregel, das eine GPU tausende von sinnvoll arbeitenden Threads benötigt, um effizient zu arbeiten [SK10]. Im Falle eines langsamen Speicherzugriffs

Speichertyp	Eigenschaft	Zugriffsmöglichkeit	Lebenszyklus
Register	Schnellste Speicherform	Thread	Thread
Shared Memory	So schnell wie Register	Threads im gleichen Block	Block
Global Memory	100-150x langsamer	`host` und `device`	Applikation
Local Memory	Wie Global Memory	Thread	Thread

Tabelle 2.5: GPU Speicher (Lesen/Schreiben) und Verwendungszweck, nach [NVI13]

kann ein SM einen anderen Warp (eines anderen Blocks) ausführen, so dass die Zugriffslatenzen beim Speicher dadurch teilweise verdeckt werden können.

Jeder SM verfügt über sehr schnelles On-Chip-Memory, das sogenannte "*shared memory*" (Schlüsselwort `__shared`), welches im Moment in der Größenordnung von 48kb bis 64kb angelegt ist, und nur für die Threads eines Blocks sichtbar ist. Threads aus anderen Blöcken können nicht auf das Shared Memory eines anderen Blocks zugreifen. Auf Shared Memory kann i.d.R. wahlweise innerhalb von 4 Taktzyklen zugegriffen werden, während der GPU Hauptspeicher eine Latenz von 400 Taktzyklen aufweist.

Tabelle 2.5 zeigt einen Überblick über die verschiedenen GPU Speicher mit Lese und Schreibeigenschaft sowie ihre Anwendungsgebiete. Zusätzlich existieren *"Texture-Memory"* und *"Const-Memory"*, auf die nur lesend zugegriffen werden können [SK10].

CUDA verwendet weiterhin herkömmliche C-Pointer, der Anwender muss jedoch strikt darauf achten, diese im richtigen Kontext zu dereferenzieren (GPU Speicheradressen nur in Kernel und Device Funktionen). Für die Allokation existieren CUDA Versionen von `malloc`, `memcpy` und `free` (`cudaMalloc`, `cudaMemcpy` und `cudaFree`). `cudaMemcpy` zeichnet sich durch die Fähigkeit aus, zwischen dem `host` und `device` Speicher transferieren zu können.

Aufgrund der massiven Multithreading-Eigenschaft von CUDA stellt sich die Frage der Synchronisation von Threads. Threads innerhalb eines Blocks lassen sich durch einen Aufruf von `__syncthreads()` synchronisieren[23]. Bei unsachgemäßer Verwendung sind jedoch auch Deadlocks möglich [SK10]. Atomare Datenzugriffe ("*read-modify-write*") sind über die sog. *"Atomics"* möglich [SK10].

Als weitere Eigenschaft lassen sich C++ Klassen mit Einschränkungen verwenden, was die Strukturierung des Codes erleichtert [NVI13].

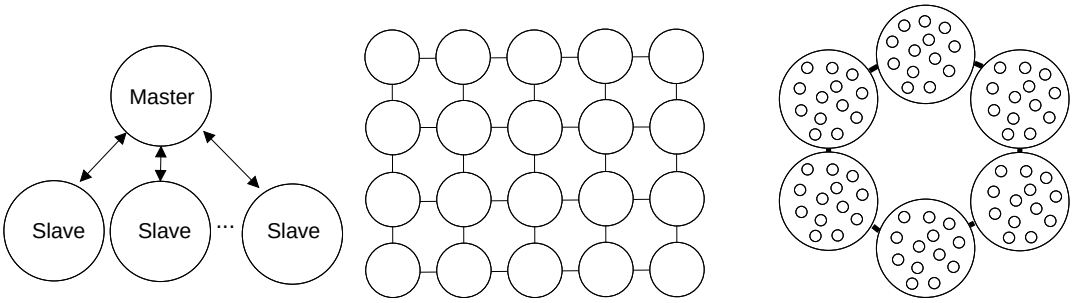
[23] Sog. "*Block Barrier*"

CUDA liefert mit dem SDK umfangreiche Beispiele aus dem naturwissenschaftlichen Bereich und allgemeine nützliche Implementierungen (z.B. Sortiernetzwerke), jedoch auch APIs (Zufallsgeneratoren).

Das 'CUDA Programming Guide' [NVI13] ist das wichtigste Nachschlagewerk für die CUDA Programmierung. Eine gute Einführung stellt [SK10] dar. CUDA 6 wird in Zukunft eine Vereinheitlichung des Speicherzugriffs ermöglichen, welches moderne, integrierte CPU und GPU Lösungen[24] auf Hardwareebene bereits bieten und dem Entwickler die Programmierung vereinfachen. Mittlerweile gibt es mit OpenCL [Gro10] einen weiteren Standard, der mit CUDA grob vergleichbar ist.

2.4.3 Parallelisierung von Genetischen Algorithmen

Eine parallele Implementierung eines GAs wird als PGA bezeichnet. Von einem theoretischen Standpunkt aus gesehen sollten sich GAs relativ gut parallelisieren lassen - Es gibt genügend Operationen, die absolut unabhängig voneinander auf disjunkten Datensets operieren. Ein Datenset könnte beispielsweise nur aus einem Individuum bestehen, und die genetischen Operationen "Fitness", "Mutation" und bedingt "Crossover" könnten unabhängig voneinander parallel ausgeführt werden.



(a) Master-Slave PGA (b) Feingranulierte PGA (c) Grobgranulierte PGA

Abbildung 2.4: Verschiedene PGA Modelle, nach [Paz98]

Bei der Parallelisierung wird in der bestehenden Literatur zwischen verschiedenen Modellen unterschieden. Eine gute Orientierung ist die Taxonomie, die in [Paz98] vorgestellt wird. Dabei wird primär untersucht, wie die Arbeit auf mehrere Ausführungseinheiten verteilt werden kann und wie die Daten entsprechend partitioniert werden können.

[24] Z.b. neuste Konsolengeneration PS4 und XBox One.

Ausführungseinheiten könnten mehrere Cores bei einem Multicore System sein, mehrere Rechner in einem Computer Cluster oder SMs auf einer GPU. Die Kommunikation zwischen Ausführungseinheiten kann dabei über *Shared Memory* oder *Message Passing* erfolgen[25]. Eine schematische Übersicht ist in Abbildung 2.4 dargestellt.

Master-Slave PGA

- Es existiert eine einzige, globale Population.
- Die Hauptschleife des GA läuft auf einer Ausführungseinheit (*Master*), und Operationen auf Individuen werden bei Bedarf parallel von weiteren Ausführungseinheiten durchgeführt (*Slaves*).
- Alle Genetischen Operatoren, die auf jeweils disjunkten Datensätze operieren, können so parallelisiert werden.
- Meist wird die Fitnessfunktion parallelisiert, vor allem wenn diese einen hohen Rechenbedarf benötigt. Dies kann bei Anwendungen im Engineering-Bereich der Fall sein.
- Die Latenz der Kommunikation zwischen *Master* und *Slaves* muss bedacht werden, insbesondere bei Message-Passing Systemen [26].

Feingranulierte PGA

- Es existiert eine einzige, globale Population.
- Die Population wird entlang der Ausführungseinheiten aufgeteilt (idealerweise ein Individuum pro Ausführungseinheit).
- Selektion und Vermehrung werden auf eine kleine Umgebung beschränkt. Die resultiert in einer kontrollierten, begrenzten Interaktion zwischen Individuen.

Grobgranulierte PGA

- Hier handelt sich somit nicht nur um einen Parallelisierungsansatz eines bestehenden GA Modells, sondern um eine neue Variante, denn es existieren nun mehrere Populationen, die **Inseln**, welche sich parallel weiterentwickeln.

[25] Kombinationen sind auch möglich.

[26] z.B. bei einem Rechen-Cluster, welcher über MPI kommuniziert.

- Zwischen den Inseln herrscht ein kontrollierter Austausch (von fitten Individuen), die Populationen entwickeln sich ansonsten unabhängig voneinander.
- Durch diesen Ansatz soll verhindert werden, dass sich der gesamte Algorithmus nicht mehr weiterentwickelt, weil er in einem lokalen Optimum "feststeckt".

Eine etwas detaillierte Aufstellung, die auch auf Hybridvarianten der vorgestellten Modelle eingeht, findet sich in [NP99]. Eine aktuelle Übersicht über das Themengebiet findet sich in [LA11].

Kapitel 3

Umsetzung

Die zentrale Herausforderung bei der Umsetzung ist es zunächst, das grundsätzliche Modell effizient mit einem GA umzusetzen, und dann geeignet zu parallelisieren. Dazu mussten initial ein paar **grundsätzliche Entscheidungen** bzgl. der Umgebung und der Technologien getroffen werden.

3.1 Grundsätzliche Entscheidungen

Es wurde entschieden, das Open-Source Statistikprogramm R([R C13]) zu verwenden, um Referenzdaten zu erstellen und um das Problem mit einem quadratischen Solver zu lösen. Hintergrund dieser Entscheidung ist die Robustheit der Software und die Verfügbarkeit von Modulen (`quadprog` für das Lösen des Markowitz-Models basierend auf [Mat]). R bietet weiterhin ein Modul zum bequemen Einlesen vom Aktienkursen (`stockportfolio`), sowie statistische Grundfunktionalitäten wie die Berechnung von Kovarianzen und Erwartungswerte. Weiterhin ist ein Statistikprogramm wie R natürlich robuster bzgl. Programmierfehlern als C oder C++. R verfügt über Module für GAs (Module `GA` und `genalg`), mit denen erste Versuche für GA Implementierungen getätigt werden können.

C++ wurde als Implementierungssprache für die serielle GA genommen, da diese den Standard im HPC und Financial Computing Bereich darstellt, und die mit Abstand performanteste und ressourcensparendste Hochsprache ist[1]. Die Anbindung von GPUs aus anderen Hochsprachen entspricht auch nicht dem Funktionsumfang, den C/C++ bietet.

[1] Sie entsprach auch den Präferenzen des Autors.

GALib [Wal08] wurde als Framework für die serielle GA verwendet, da diese in verschiedensten Publikationen als Referenz verwendet wird (u.A. [PJS10], [SNK]) und einen modularen, objektorientierten Ansatz ermöglicht.

Eine mögliche Alternative zu CUDA wäre OpenCL [Gro10] gewesen, welche jedoch als reine Spezifikation immer noch stark implementationsabhängig ist und nicht so gute Tools anbietet wie CUDA. Grundsätzlich ist die Portierung von CUDA auf OpenCL technische möglich und bedarf meist keinerlei größeren konzeptionellen Änderungen.

Die weitere Umsetzung lässt sich grob in drei Teile unterteilen:

Referenzdaten und Referenzmodell
: Bereitstellung der Referenzdaten und Lösen des Problems mit einem 'Quadratic Solver' im Statistikprogramm R. Diese Arbeit basierte zum großen Teil auf [Mat]. Zu den Arbeiten gehörte zunächst das Einlesen der Aktienkurse, Berechnung der zu erwartenden Rendite und der Covarianzmatrix sowie der Export als .CSV Datei. Anschließend folgte der Aufruf des "Quadratic Solvers" und Plotten der Effizienzkurve. Nach der Bereitstellung der anderen Implementierungen erfolgte das Einlesen deren Ergebnisse zum Plotten.

Serielle GA
: Konzeption der seriellen GA Variante zur Portfolio Optimierung nach dem Markowitz Modell. Umsetzung in C++ mit Hilfe der Biblothek GALib[Wal08].

Master-Slave GA
: Implementierung einer Kernel Funktion in CUDA zur Fitnessberechnung einer kompletten Population und Einbindung in das GALib Framework.

Parallele GA(PGA)
: Entwurf der Parallelisierungskonzepte für eine CUDA PGA und Implementierung derselben.

Benchmarks
: Konzeption und Ausführung von Tests, um die Qualität und Geschwindigkeit der PGA und Insel-PGA zu bewerten.

3.1.1 Hardware- und Softwareumgebung

Als Hardwareumgebung wird ein handelsüblicher PC verwendet, bestückt mit einem Core2Quad Q6600 @ 2.4GHz, 4GB RAM, und einer NVIDIA Geforce GTX560ti Grafikkarte mit 1GB RAM.

Die GTX 560ti verfügt über 8 SMs mit jeweils 48 SPs, also insgesamt 384 sog. CUDA Kerne. Jeder SM verfügt über 48kb Shared Memory.

Als Softwareumgebung wurde Ubuntu Linux 10.04LTS (64bit) verwendet , sowie das CUDA SDK 5.5. Als C++ IDE wurde Qt Creator verwendet. Für Matrix und Vector Operationen wurde bei der seriellen GA Variante die C++ Biblothek `libeigen` in der Version 3 verwendet.

3.2 Serieller GA

Bei der Auswahl des GA Modells wird auf Goldbergs SimpleGA [Gol89] zurückgegriffen, welcher auch in GALib verwendet werden kann. SimpleGA bietet sich an, da es keine allgemeingültige Auswahl für den optimalen GA zur Lösung eines Problems gibt, und dieses einfache Modell somit als sinnvoller Ausgangspunkt dienen kann [LLM07].

Für die Optimierung des klassischen Markowitz Problems wird ein Float-Array als Repräsentation des Portfolios gewählt. Vor der Fitness-Berechnung eines Individuums müssen diese Werte nur noch normalisiert werden, so dass die Summe 1 ergibt (Siehe dazu 2.1.5).

Eine Übersicht zur Modellierung des Markowitz-Optimierungsproblems mit Metaheuristiken findet sich in [SD06].

Tatsächlich ist die Definition der *idealen* Konfiguration eines EAs für ein bestimmtes Problem keine einfache Aufgabe, und weiterhin ein Thema aktueller Forschungsbemühungen [LLM07]. Beim Einsatz in der Praxis ist ein individuelle Anpassung nach dem Trial-and-Error Prinzip die Regel.

Als wichtigste Parameter neben der Modellauswahl sind die Parameter Populationsgröße, Crossover-Wahrscheinlichkeit und Mutations-Wahrscheinlichkeit zu nennen.

Die Umsetzung dieser Modellierung mit dem C++ Framework GALib stellt keine größeren Herausforderungen dar. Es wird die Klasse `SimpleGA` verwendet. Im

Folgenden werden alle relevanten Eigenschaften der seriellen GA Implementierung aufgeschlüsselt:

- **Populationsgröße**: Die Populationsgröße wurde initial auf 200 festgelegt, varierte jedoch im Laufe der Benchmarks und Experimente.
- **Selektions Operator**: "Roulette-Wheel'"-Selektion.
- **Crossover Operator**: 1-Punkt Crossover.
- **Crossover Wahrscheinlichkeit**: 0,9.
- **Mutations Operator**: Swap-Mutation
- **Mutations Wahrscheinlichkeit**: 0,01.
- **Terminierung**: Nach 250 Iterationen. (Standard)
- **Repräsentation**: Float Array (Klasse `GA1DArrayGenome<float>`).

Die Covarianz Matrix und die Erwartungswerte der Renditen werden als .CSV bereitgestellt und zur Laufzeit in Matrix/Vektor Repräsentationen von `libeigen` geladen.

Die Fitness Funktion (Listing 3.2) optimiert den Matrixausdruck des Optimierungsproblems. Dabei werden die Float-Werte auf den gültigen Definitionsbereich normalisiert (Listing 3.1).

Listing 3.1: Normalisierungsfunktion für seriellen GA

```
void setPortfolioWeights(MatrixXf& outputVector,
      GA1DArrayGenome<float>& genome) {
  outputVector.resize(d.stockNames.size(), 1);
  float sumWeights = 0;
  for (size_t x=0; x< (size_t) genome.size(); x++) {
    sumWeights += genome.gene(x);
  }
  for (size_t x=0; x< (size_t) genome.size(); x++) {
    outputVector(x, 0) = genome.gene(x) / sumWeights;
  }
}
```

Listing 3.2: Markowitz Fitnessfunktion für seriellen GA

```
float marko_objective(GAGenome& genome) {
  GA1DArrayGenome<float>& g = (GA1DArrayGenome<float>&) genome;
  MatrixXf omega;
  setPortfolioWeights(omega, g);
  MatrixXf val = (omega.transpose() * d.covarianceMatrix * omega)
     - (q * d.stockColMeansMatrix) * omega );
  return -val(0,0);
}
```

3.3 Master-Slave PGA

In dieser Variante wird die Hauptschleife der SimpleGA (GALib) auf der CPU ausgeführt. Im Rahmen dieser Hauptschleife wird ein CUDA Kernel für die Fitnessberechnung aufgerufen. In diesem Falle gibt es nur einen Slave, die GPU.

Der Kernel implementiert die Fitnessfunktion für das klassische Markowitz-Modell. GALib bietet eine Überladung für die Evaluation (Fitnessberechnung) der kompletten Population an, so dass zunächst eine mit `extern "C"` Bindung klassifizierte Schnittstelle aufgerufen kann. Die Implementierung dieser Schnittstelle (deren Übersetzungseinheit wurde mit nvcc übersetzt) ruft dann den eigentlichen Kernel auf. Dabei wird die Population vorher in ein Float Array kopiert und mit cudaMemCpy auf das Device transferiert. Nach der Fitnessberechnung werden die errechneten Fitnesswerte (Float-Array) zurück auf den Host kopiert (Abbildung 3.1).

Der Kernel selber implementiert die Fitnessfunktion. Dabei wird ein Thread pro Individuum verwendet.

Es gibt eine fixe Zuordnung von Threads pro Block[2] (64), und somit werden $\frac{populationSize+ThreadsPerBlock+1}{ThreadsPerBlock}$ Blöcke gestartet.

Innerhalb des Kernels wird das Individuum aus dem Hauptspeicher in lokale Variablen (d.h. Register) geladen, die Normalisierungsfunktion durchgeführt und anschließend die Fitness berechnet. Nach der Berechnung erfolgt das Speichern des Ergebnis in den GPU Hauptspeicher und der Kernel terminiert.

[2]Empirisch ermittelt unter Zuhilfenahme von [SK10]

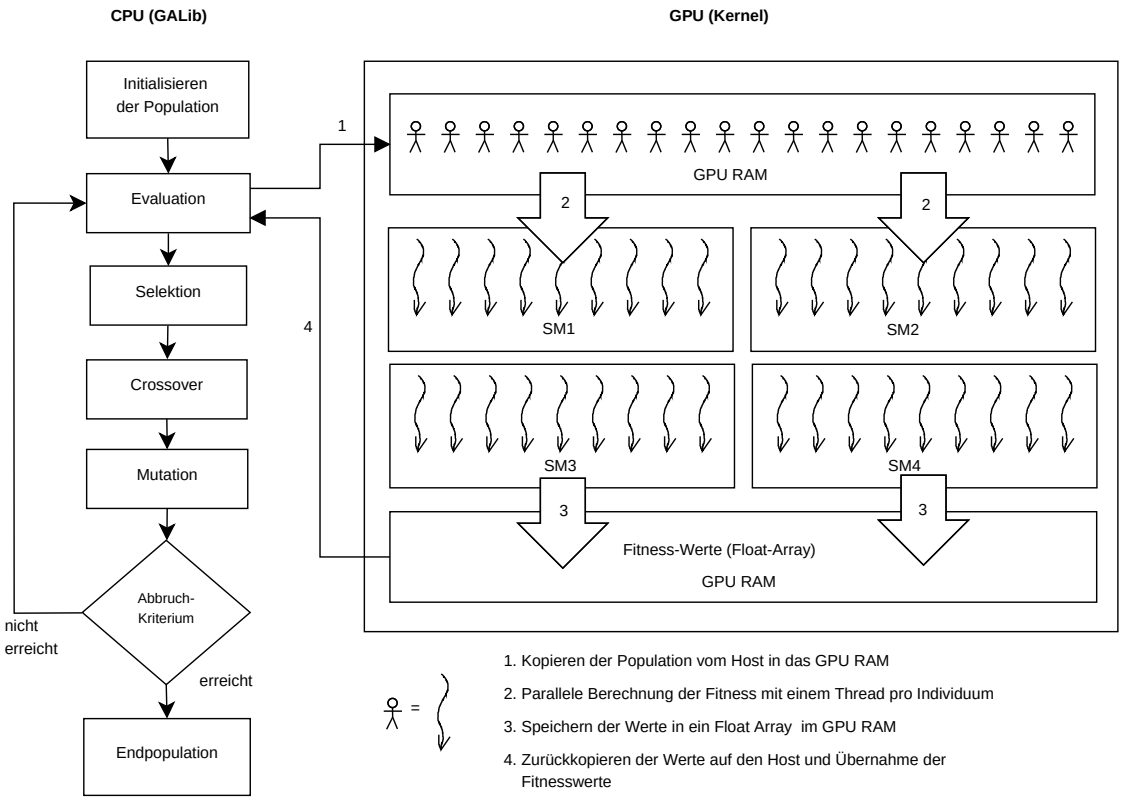


Abbildung 3.1: Master-Slave PGA

Die Matrixoperationen innerhalb des Kernels wurden dabei von Hand implementiert. Hauptgrund waren das Fehlen einer Biblothek, die von Device-Code aufgerufen werden kann[3]. Diese Operationen wurden nicht parallelisiert.

Da es Threads geben kann, die nichts zu tun haben, wird geprüft, ob die absolute threadId (`threadIdx.x` · `blockIdx.x`) größer als die Populationsgröße ist; In dem Falle wird der Kernel sofort beendet.

3.4 PGA

3.4.1 Bestehende Ansätze (GPU)

Hier folgt zunächst kurz eine Zusammenfassung der bestehenden Lösungsansätze für die Umsetzung von PGAs mit GPUs und CUDA.

In [PJS10] wird ein CUDA PGA mit mehreren Inseln realisiert, welche über einen unidirektionalen Ring kommunizieren. Die Populationen werden im schnellen “Shared

[3]NVIDIA cuBLAS ist prinzipiell für Host-Code konzeptioniert. Die Device-Funktionalität ist nur für die allerneusten GPUs verfügbar (seit Fermi)

Memory" gespeichert. Dabei wird pro Individuum parallelisiert. Die Implementierung benötigt nur einen Kernel.

[Ois+11] beschreibt einen CUDA PGA, welcher auf Genomebene parallelisiert. Die Funktionalität wird hier mit mehreren Kerneln realisiert, der Kontrollfluss wird von der CPU gesteuert. Dabei werden sowohl "Shared Memory" als auch "Global Memory" verwendet. Ein ähnlicher Ansatz wird in [Kol] beschrieben.

Der Versuch der Konzeption eines universellen CUDA PGA Frameworks wird in [SNK] unternommen. Dabei wird GALib [Wal08] als Vorbild genommen. Hier wird sowohl auf Individuums- als auch auf Genom Ebene parallelisiert. Es werden ebenso mehrere Kernel verwendet, und die Kontrolle obliegt der CPU. Für die Speicherung wird primär "Global Memory" verwendet.

In [LMT10] werden drei verschiedene Insel PGAs mit CUDA realisiert, welche verschiedene Speicherungs- und Parallelisierungsstragien repräsentieren.

Der Großteil der Publikationen verwenden Testfunktionen (siehe Kapitel 4.1), um ihre Implementierungen zu testen und zu bewerten.

3.4.2 Konzeption

Die hier beschriebene CUDA PGA orientiert sich sehr stark an den Ansatz aus [PJS10]. Die Ausnutzung von Shared Memory in Kombination mit einer einheitlichen, rein GPU-basierten Implementierung erschien im Kontext der Portfolio Optimierung am erfolgversprechendsten.

Es wurde versucht, so viel wie möglich im Ablauf der PGA zu parallelisieren. Es wird nur ein Kernel verwendet.

Ein Thread wird wieder auf eine Individuum abgebildet. Es werden Inseln von Populationen verwendet (also eine Grobgranulierte PGA). Jede Population befindet sich komplett im `Shared Memory` eines SM, dadurch soll die Zugriffsgeschwindigkeit erhöht werden. [4] Die Wahl der implementierten genetischen Operatoren wurde von [PJS10] übernommen.

[4] Dies bedeutet auch, das nur ein Bruchteil der Rechenkapazität der GPU für die Operationen auf eine Population zur Verfügung steht.Die Entscheidung ist ein Trade-off zwischen roher Rechengeschwindigkeit und der Latenz des Datenzugriffs.

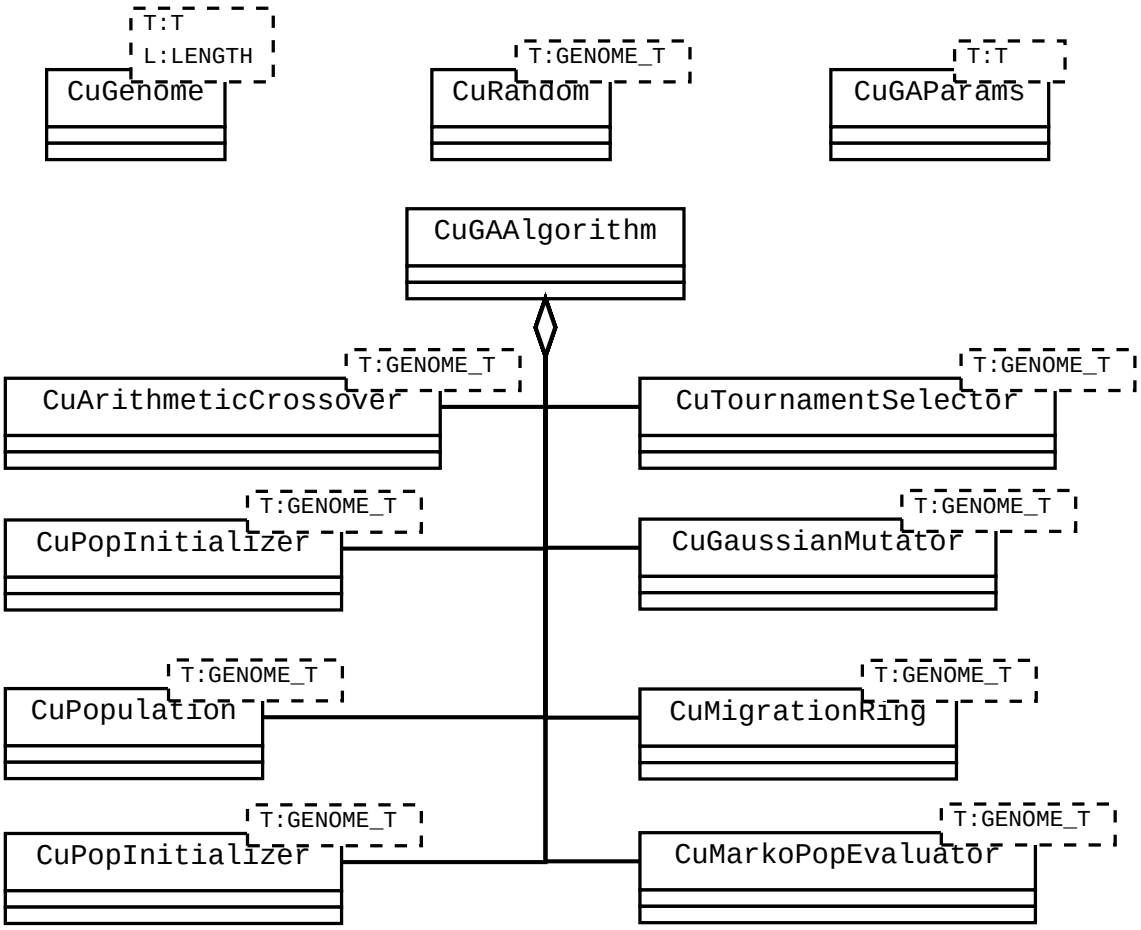


Abbildung 3.2: Klassendiagram der CUDA PGA Implementierung

Eine Übersicht über die zentralen Klassen liefert Abbildung 3.2. Es werden generische C++ Klassen (Templates) verwendet, die jedoch abgesehen von der Konfiguration und des Populations-Buffers keinen Zustand miteinander teilen. Klassen werden primär als Strukturierungshilfe des Codes verwendet.

"Shared Memory" wird für die Populationsspeicherung und für Thread-Synchronisation verwendet (Reduktion zur Ermittlung des besten Individuums).

Der zentrale Algorithmus wird in der Klasse `CuGAAlgorithm` gekapselt. Diese enthält als Membervariablen die entsprechenden Operatoren, sowie die Population [5].

Die elementare Datenstruktur ist der generische Datentyp `CuGenome` (C++ Template), welcher über den Typ und die Länge parametrisiert wird und prinzipiell ein Array eines Types mit einer bestimmten Länge und den Fitnesswert kapselt.

Die Konfiguration der PGA und das Setzen von Parametern und Buffer-Adressen für Eingabe und Ausgabe erfolgt über die generische Klasse `CuGAPArams`. Die Signatur des Kernels übernimmt diese als einzigen Parameter.

Da häufig auf einen Zufallsgenerator zugegriffen wird, muss jeder Thread über einen eigen Zustand verfügen und mit einem eigenen Start-Seed initialisiert werden. Somit verfügt jeder Thread über eine eigene Instanz einer Klasse `CuRandom`, welche diesen Zustand kapselt[6].

3.4.3 Ablauf

Anbei die Beschreibung des GA-Algorithmus. Nach jedem hier dargestellten Schritt wurde `__syncthreads()` verwendet, um die Konsistenz der Daten zu sichern.

Populationserstellung

Jeder Thread liest sein Individuum aus dem Speicher und belegt das Genom mit zufälligen Float Werten innerhalb des definierten Gültigkeitsbereiches. Anschließend ist die komplette Population im Shared Memory initialisiert.

[5] Es ist grundsätzlich möglich, über statischen Polymorphismus und dem CRTP Pattern Basisklassen für die Operatoren zu verwenden - dies wurde in der aktuellen Implementierung aus Zeitgründen nicht durchgeführt.

[6] Es wird intern die beim CUDA-SDK verfügbare API *cuRAND* verwendet.

Beginn einer Generation, Fitnessberechnung

Die Fitnessberechnung der Population wird ebenso parallel ausgeführt. Jeder Thread speichert die Fitness im entsprechendem Genom Objekt.

Speichern des besten Ergebnisses

Um das beste Ergebniss zu sichern, wird dieses durch eine Reduktionsoperation ermittelt und für jede Generation im `CuPopulation` Objekt gespeichert. Dazu wird ein temporärer Speicherbereich (*Shared Memory*) benötigt

Selektions-Operator

Es wird parallel auf allen Individuen eine "Tournament Selection" mit k=2 durchgeführt. Jeder Thread sucht sich einen zufälligen Partner und führt ein "Tournament" durch. Der Sieger wird mit einer Wahrscheinlichkeit von 0.7 selektiert, ansonsten der Verlierer. Sämtliche Gewinner werden in einem eigenen, temporären Speicherbereich vorgehalten (*Shared Memory*).

Crossover-Operator

Die Crossover Operation wird mit der Hälfte der vorhandenen Threads durchgeführt. Jeder Thread selektiert einen Partner aus dem temporären Speicherbereich und führt einen Arithmetic Crossover Operator aus, aus denen zwei Kinder entstehen, welche in der Originalpopulation gespeichert werden. Damit sind alle ursprünglichen Individuen der alten Population ersetzt worden.

Mutations-Operator

Es wird eine Gaussian Mutation durchgeführt (Erwartungswert: 0, Standardabweichung: 1), wobei die Mutationswahrscheinlichkeit bei 0.2 liegt.

Ende einer Generation

Wenn Elitismus aktiviert ist, wird das beste Individuum wieder an eine zufällige Position in der Population kopiert. Wenn die Migration aktiviert wurde, wird diese durchgeführt (siehe Kapitel 3.4.4). Der Algorithmus terminiert nach einer bestimmten Anzahl von Runden

Der Kernel kopiert im Anschluss das beste Ergebnis in den GPU Speicher.

3.4.4 Migrationsoperator

Für den PGA ist es über Konfiguration möglich, den Kernel auf mehreren SMs mit jeweils eigenen Populationen zu starten. Ein Austausch von besten Individuen

zu Nachbarn erfolgt im Rahmen der sog. Migrationsrunden. Diese wird nach eine bestimmten Anzahl von Runden durchgeführt (Migrationsinterval).

Da eine Menge von Inseln existiert, stellt sich die Frage der Topologie. Inspiriert von [PJS10] wird die Topologie zwischen den Inseln als unidirektionaler Ring realisiert.

Eine Migration des besten Ergebnisses erfolgt dabei nach einer fest vorgelegten Reihenfolge, jede Insel kennt seinen Vorgänger und seinen Nachfolger. Der Nachfolger erhält das beste Ergebniss einer Insel, das beste Ergebnis eines Vorgängers wird in die eigene Population übernommen.

Da sich die Inseln jedoch unabhängig voneinander entwickeln, stellt sich die Frage der Synchronisation.

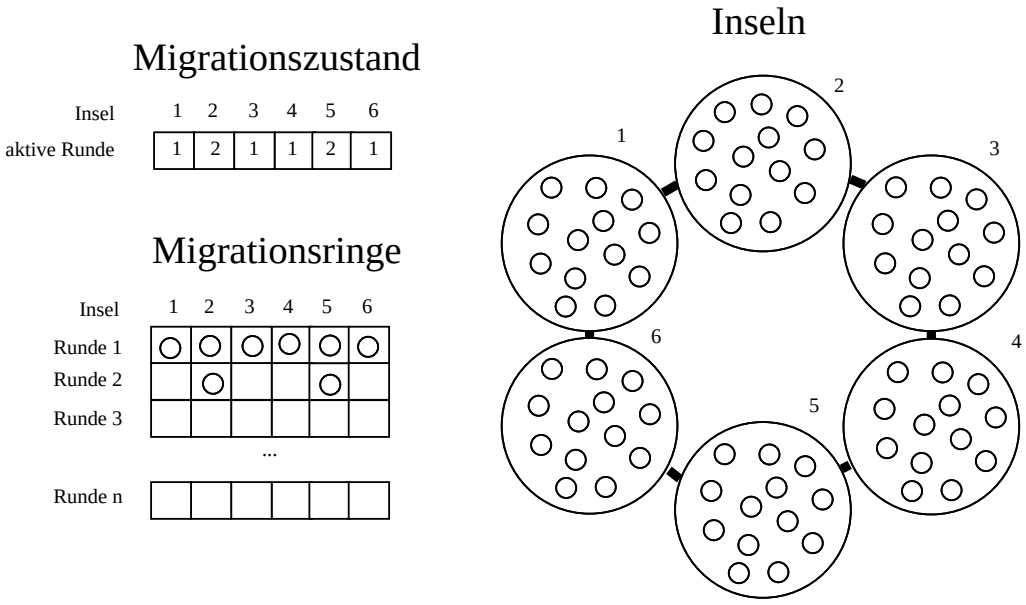


Abbildung 3.3: Migrationsring der CUDA Implementierung

Hier wird eine Menge von Migrationsringen verwendet, welche jeweils das beste Ergebnis eines Ringes einer Runde enthält (Siehe Abbildung 4.4 für eine beispielhafte Konfiguration mit 6 Inseln). Da die Ringe von allen SMs auf der GPU erreichbar sein müssen, werden diese im Hauptspeicher der GPU vorgehalten. Die Migrationsringe für den hier beschriebenen Anwendungsfall verbrauchen i.d.R. sehr wenig RAM[7].

Der Zugriff auf die Ringe erfolgt unsynchronisiert. Damit es aufgrund der Asynchronität keine Synchronisationsprobleme gibt, speichert jede Insel seine zuletzt aktiv

[7]Konzeptionell ist es problemlos möglich, mehrere Individuen pro Runde und Ring vorzuhalten, in dieser Implementierung wurde jedoch nur das beste Individuum gespeichert.

durchgeführte Migrationsrunde atomar[8] in einem Datenbereich Migrationszustand. Alle SMs können so atomar erkennen, welche Migrationsringe problemlos lesbar sind. Da i.d.R. nicht in jeder Generation migriert wird, sollte sich der Overhead durch die Verwendung von *"Atomics"* in Grenzen halten.

3.4.5 Übersicht über PGA Implementierungen

Hier abschließend eine Übersicht über die implementierten PGAs und ihrer Ansätze, das Optimierungsproblem (siehe Kapitel 2.1.5) zu lösen:

Insel-PGA : Der CUDA PGA, wie sie in Kapitel 3.4 beschrieben wird. Diese löst mit mehreren Inseln das Optimierungsproblem für ein q. Jede Insel führt somit einen GA aus.

PGA : Der CUDA PGA, wie sie in Kapitel 3.4 beschrieben ist, jedoch mit deaktivierter Inselfunktionalität. Statt dessen ist dieser auf das gleichzeitige Lösen mehrerer Probleme für das Markowitz-Problem optimiert[9]. Der PGA erhält einen Wertebereich für q und eine Menge von Intervallen, die berechnetet werden müssen. Für jedes Problem q wird parallel ein Block (also ein GA) ausgeführt.

Master-Slave PGA : GALib und ein CUDA Kernel, welcher nur die Fitnessfunktion (für ein q) löst. Der GA läuft nur auf der CPU.

[8] Mit Hilfe von CUDA "Atomics" [SK10]

[9] Optimiert bedeutet hier die Konfiguration, also vor allem die Anzahl der Blöcke und Threads sowie die Funktionalität zum Kopieren von mehreren q-Ergebnissen pro Kernelaufruf. Es handelt sich um den exakt gleichen Code wie die Insel-PGA und um keine Codereplikation.

Kapitel 4

Ergebnisse

Zum Zwecke des Vergleichs der umgesetzten PGA Implementierungen werden verschiedene Tests und Benchmarks durchgeführt. Dabei werden stets der Serielle GA mit dem PGA und dem Insel-PGA verglichen. Bei den Benchmarks wird auch der Master-Slave PGA miteinbezogen. Anbei eine Übersicht über die Struktur:

Testfunktionen für PGA
: Spezielle Testfunktionen werden zum Überprüfen der PGA Funktionalität gelöst und mit den Ergebnissen des Seriellen GA verglichen.

Effizienzkurven
: Hier wird das Markowitz Problems mit drei GA Implementierungen gelöst. Die Effizienzkurve wird dargestellt. Die Genauigkeit der Lösung wird mit Hilfe einer Referenzkurve geprüft.

Benchmarks
: Hier wird die PGA-Implementierung und die Master-Slave PGA Implementierung bzgl der Ausführungsgeschwindigkeit geprüft.

Die Systemumgebung enspricht der in Kapitel 3.1.1 beschriebenen. Für die Compilerseinstellungen (`gcc`, `nvcc`) wurden die Einstellungen aus Listing 4.1 verwendet[1].

Bei der Berechnung von Portfolios wird ein Portfolio mit 6 Aktien verwendet. Die Repräsentation benötigte 28 Bytes Speicher (jeweils 4 Bytes für eine Aktie und nochmals 4 Bytes für die Fitness).

[1] GALib wurde mit identischen Einstellungen übersetzt (statische Biblothek)

Listing 4.1: Einstellungen für gcc und nvcc

```
// Compilerflags für gcc (64bit)
-O2 -msse2 -ffast-math

// Compilerflags für nvcc (64bit)
-m64 -O3 -arch=sm_20 --compiler-options -fno-strict-aliasing
-use_fast_math --ptxas-options=-v -G
```

4.1 Testfunktionen

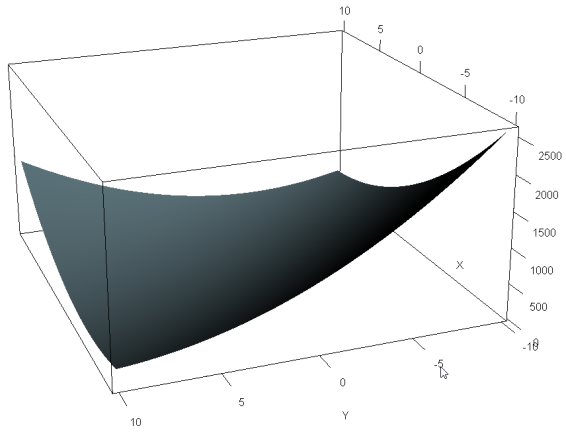


Abbildung 4.1: Booth Funktion

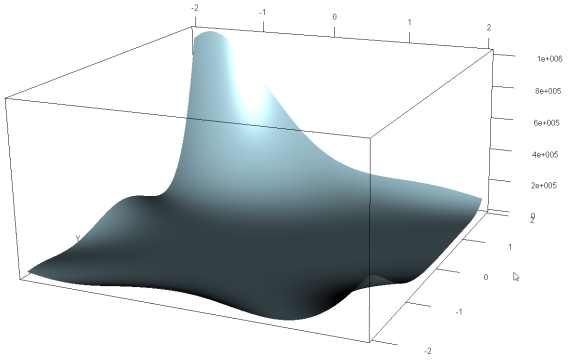


Abbildung 4.2: Goldstein-Price Funktion

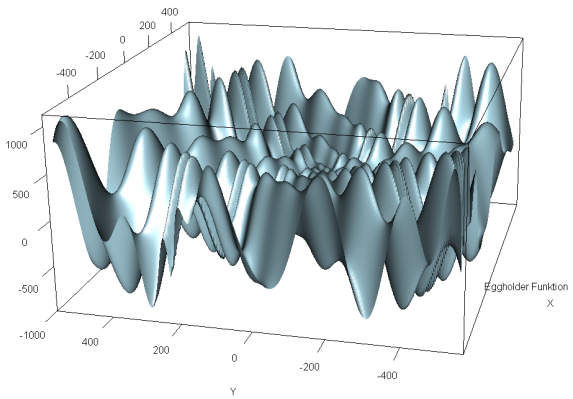


Abbildung 4.3: Eggholder Funktion

Um die Leistungsfähigkeit von Optimierungsalgorithmen zu überprüfen, existieren eine Menge von Testfunktionen [Bäc96], oft in der Form $f(x, y) \rightarrow z$, die sich gut darstellen lassen. Die Herausforderung für den Algorithmus ist meist die Minimierung der Funktion[2]. Hier werden drei Testfunktionen verwendet, mit deren Hilfe die

[2]Maximierung wäre die Minimierung von $-1 \cdot f(x, y)$

GA	Booth Fkt.	Goldstein-Price Fkt.	Eggholder Fkt.
Serieller GA	Ø: 0.0468868	Ø: 3.68986	Ø: -924.195
	m : 0.0352223	m : 3.26871	m : -928.959
PGA	Ø: 3.36976e-07	Ø: 3.00002	Ø: -882.114
	m : 1.80256e-07	m :3	m : -886.124
Insel PGA	Ø: 0.00316892	Ø: 2.99999	Ø: -915.161
	m : 2.00664e-07	m : 2.99999	m : -926.326

Tabelle 4.1: Durchschnitt und Median m der Fitnesswerte für drei Testfunktionen

Nützlichkeit der PGA und Insel PGA Implementierung grundsätzlich geprüft werden kann:

Booth Funktion (Abbildung 4.1)

$f(x,y) = (x+2y-7)^2 + (2x+y-5)^2$

Minimum: $f(1,3) = 0$, für $-10 \leq x, y \leq 10$

Goldstein-Price Funktion (Abbildung 4.2)

$$f(x,y) = \left(1 + (x+y+1)^2 \left(19 - 14x + 3x^2 - 14y + 6xy + 3y^2\right)\right) \cdot \left(30 + (2x-3y)^2 \left(18 - 32x + 12x^2 + 48y - 36xy + 27y^2\right)\right)$$

Minimum: $f(0,-1) = 3$, für $-2 \leq x, y \leq 2$

Eggholder Funktion (Abbildung 4.3)

$f(x,y) = -(y+47)\sin\left(\sqrt{\left|y + \frac{x}{2} + 47\right|}\right) - x\sin\left(\sqrt{|x-(y+47)|}\right)$.

Minimum: $f(512, 404.2319) = -959.6407$, für $-512 \leq x, y \leq 512$

Die Ergebnisse für die GA Implementierungen (Serieller GA, PGA und Insel PGA) finden sich in Tabelle 4.1. Dabei wurden bei einer Populationsgröße von 512 jeweils 250 Generationen berechnet. Der Insel PGA wurde mit 16 Inseln gestartet. Die gezeigten Ergebnisse sind der Mittelwert aus 100 Durchläufen, wobei als Lagemaß zusätzlich der Median angegeben ist.

4.2 Effizienzkurven

Zur Bewertung der Funktionalität wird die Effizienzkurve für das klassische Markowitz problem mit allen drei GA Implementierungen (Serieller GA, PGA, Insel PGA) errechnet und dargestellt (Siehe Abbildungen 4.4 - 4.6).

Dabei wird die Effizienzkurve mit 41 verschiedenen Werten für den Risikopräferenzfaktor q berechnet (Siehe Kapitel 2.1.5). Es werden jeweils 250 Generationen berechnet.

Dieses Experiment wird für jeweils drei Populationsgrößen (10,20,200) durchgeführt. Die beiden kleineren Populationsgrößen wurden gewählt, da bei der Verwendung des PGA und Insel PGA das *Shared Memory* eine rare Ressource darstellt, und bei derartigen Problemen oft nur mit kleinen Populationsgrößen gearbeitet werden kann.

Die dargestellte Rendite ist die zu erwartende *tägliche* Rendite.

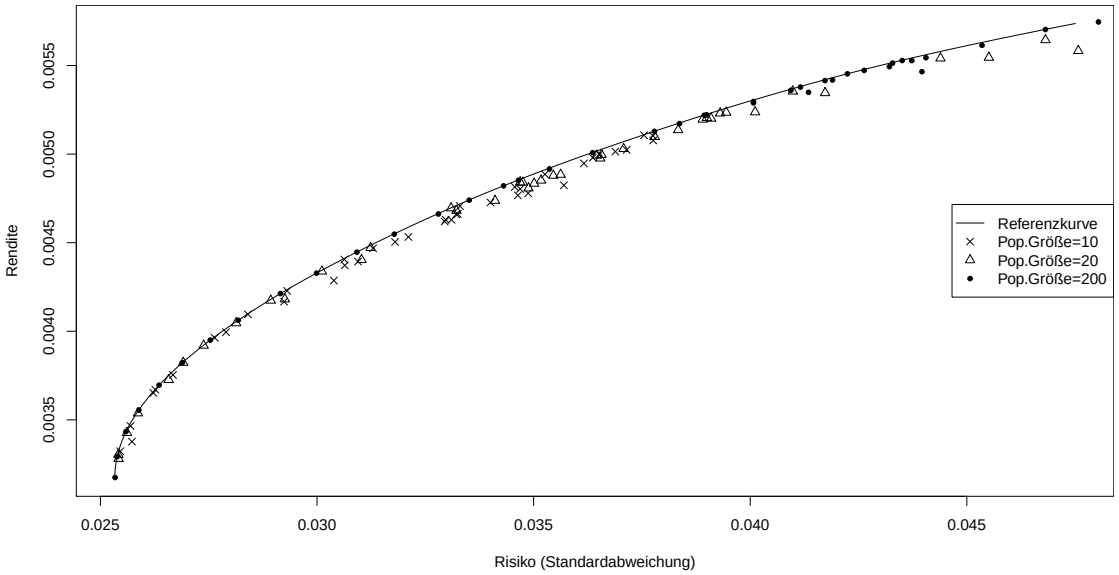


Abbildung 4.4: Effizienzkurve des seriellen GA

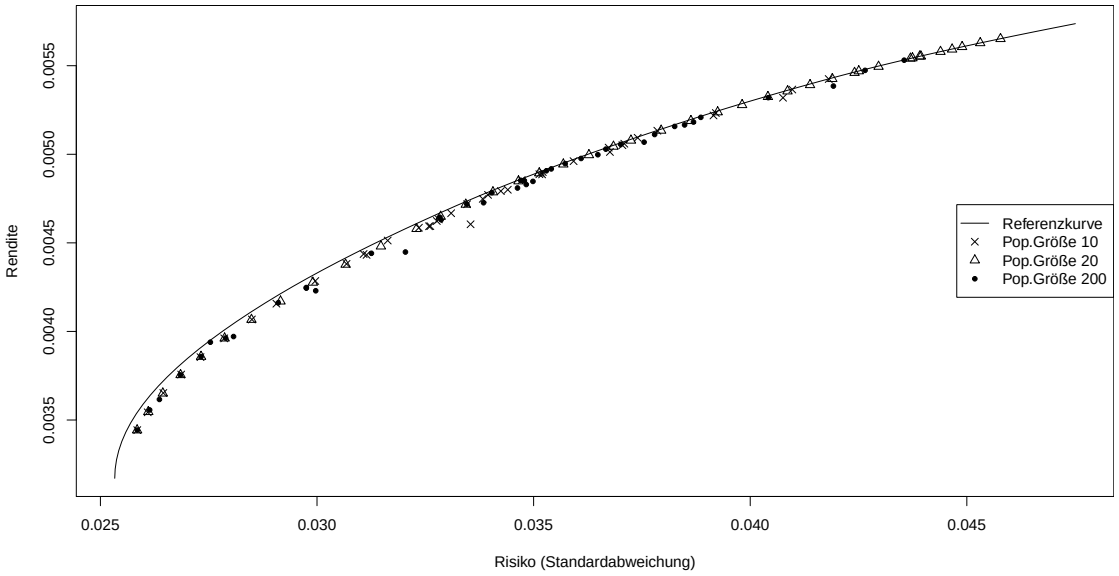


Abbildung 4.5: Effizienzkurve des CUDA PGA

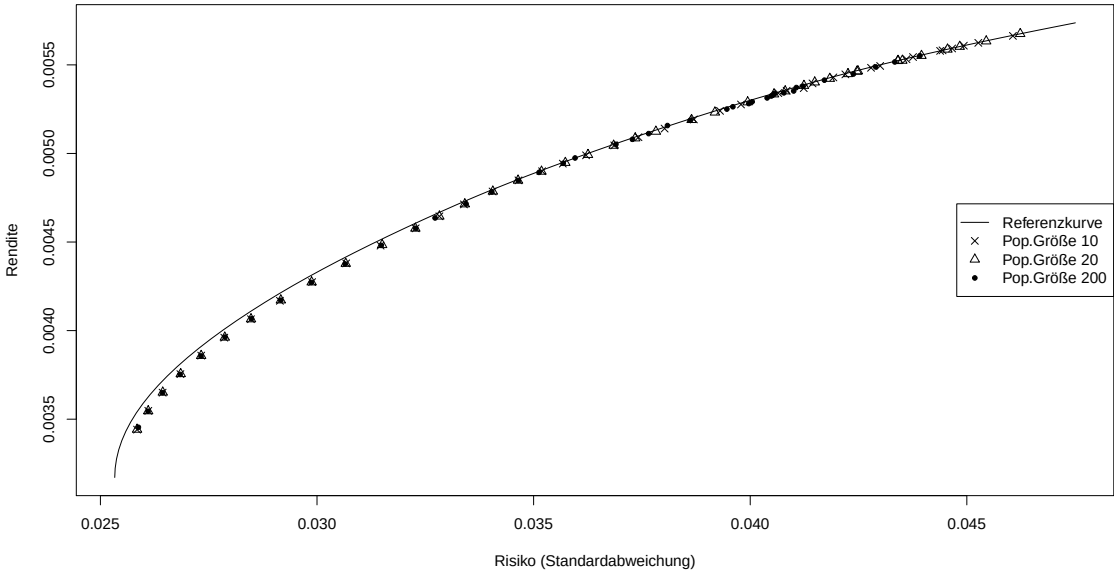


Abbildung 4.6: Effizienzkurve des CUDA Insel PGA

4.2.1 Genauigkeit

Ein Mass für die Abweichung einer Datenreihe von einer Referenzkurve ist die Summe der Fehlerquadrate, die wie folgt definiert ist:

$$S = \sum (f(x) - y)^2$$

Je kleiner S, desto genauer gleicht sich die Datenreihe dem Verlauf der Referenzkurve an. Die Tabelle 4.2 zeigt S für die drei vorgestellten GA Implementierungen mit jeweils drei Populationsgrößen, nach S aufsteigend sortiert.

Implementierung	Pop.größe	S
Serieller GA	200	1.4277e-08
CUDA Insel PGA	20	3.988e-08
CUDA PGA	10	4.0230e-08
CUDA PGA	20	4.0230e-08
CUDA Insel PGA	10	4.0413e-08
CUDA Insel PGA	200	4.1142e-08
CUDA PGA	200	8.9963e-08
Serieller GA	20	9.2303e-08
Serieller GA	10	1.2433e-07

Tabelle 4.2: Summe der Fehlerquadrate für GA Implementierungen, aufsteigend sortiert

4.3 Benchmarks

4.3.1 PGA

Das Ziel der Benchmark-Messungen ist ein Vergleich der Laufzeiten des PGA mit dem seriellen GA für eine identische Aufgabenstellung. Dabei ist die Insel-Funktionalität des PGA ausgeschaltet.

Zu diesem Zweck wird das klassiche Markowitz Optimierungsproblem für verschiedene Werte der Risikotoleranz q gelöst (Siehe Kapitel 2.1.5). Es werden n Einzelaufgaben im Bereich $0 \leq q \leq 4$ erstellt. Die Populationsgröße wird pro Experiment dynamisch angepasst.

- Der Kernel des PGA wird mit n Blöcken aufgerufen - jeder Block berechnet für q einen eigenen Teil der Kurve. Das Gesamtergebnis wird nach Beendigung des Kernels komplett in den host-Speicher zurück kopiert und ausgegeben/gesichert.
- Der serielle GA wird für jedes q seriell in einer Schleife aufgerufen (n Aufrufe).

Im Experiment wurde $n = 800$ gesetzt, was einen sehr genauen Kurvenverlauf ermöglicht und gleichzeitig eine ausreichend hohe Rechenherausforderung darstellt (800 komplette GA Durchläufe mit jeweils 250 Generationen).

Die Tabelle 4.3 zeigt die gemessenen Zeiten in Sekunden. Dabei wird der Durchschnitt von 5 Durchläufen dargestellt. Die Zeit umfasst die Gesamtzeit der Problemlösung, inklusive Programmstart, einlesen, ggf. Kernelaufrufen und Daten-Kopieren (sog. *"wall time"*). Gemessen wurde mit dem `time` Kommando aus der Linux Kommandozeile.

4.3.2 Master-Slave PGA

In diesem Experiment wurde das klassiche Markowitz-Problem mit $0 \leq q \leq 4.0$ und 41 Schritten berechnet. Die Hauptschleife läuft weiterhin im GALib Framework, die Fitnessevaluation für die komplette Population wird auf der GPU durchgeführt. Die Anzahl der Generationen ist auf 250 festgelegt. Die Anzahl der Populationen wird pro Experiment variiert. Es werden jeweils 10291 Populationsevaluationen während eines Experiments durchgeführt.

Der GPU Kernel wird in Blöcken von 64 Threads aufgerufen, wobei jeder Thread die Fitness eines Individuum berechnet.

Die Tabelle 4.5 zeigt die Laufzeiten ("wall time") im Vergleich zur reinen CPU-Implementierung mit GALib, sowie die absolute Zeit in ms, die für die Fitnessberechnung verbraucht wurde.

Populationsgröße	CPU Laufzeit (s)	PGA Laufzeit (s)	Speedup
20	9,04	2,11	4,28
50	23,126	2,9	7,97
100	46,49	3,76	12,36
150	79,22	4,18	18,95
200	90,54	5,88	15,40
250	131,40	5,95	22,08
500	264,80	12,6	21,02

Tabelle 4.3: Speedup für klassisches Markowitz Problem

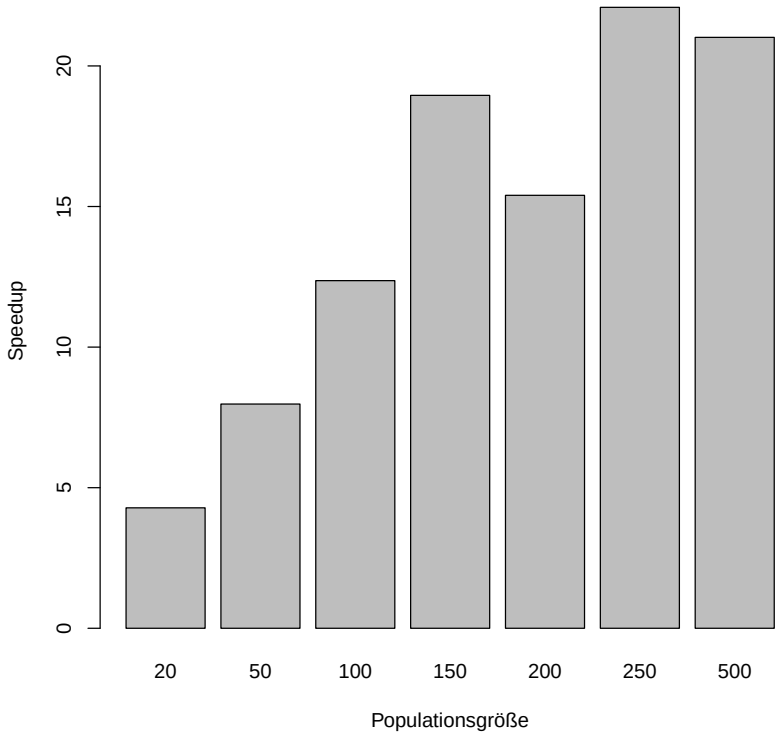


Tabelle 4.4: Speedup der PGA im Vergleich zum seriellen GA

Pop.größe	GPU Laufzeit (s)	GPU Fitness (ms)	CPU Laufzeit(s)	CPU Fitness (ms)
50	5,46	4601	1,02	488
100	6,13	4626	2,14	969
200	7,98	4693	5,03	1946
500	31,48	8182	19,99	4820
1000	113,18	8480	72,89	9691
2000	285,42	6795	292,02	19566

Tabelle 4.5: Laufzeiten für Master-Slave PGA

Kapitel 5

Bewertung

5.1 Bewertung der Ergebnisse

5.1.1 Testfunktionen

Die Optimas der Testfunktionen werden von dem PGA und dem Insel-PGA gut approximiert. Dabei wird - mit Ausnahme der "*Èggholder*" Funktion - ein besseres Ergebnis als mit dem seriellen GA erreicht. Dabei ist der Median als robustes Lagemaß ein guter Indikator.

Auch zeigt sich, dass sich das Ergebnis des PGA durch die Verwendung des Insel-PGA meist verbessern lässt. Dies ist vor allem bei der "*Eggholder*" Funktion offensichtlich, welche viele lokale Minimas aufweist, in der ein EA "steckenbleiben" kann.

5.1.2 Effizienzkurven

Alle GA Implementierungen approximieren die Referenzkurve mit guter Genauigkeit.

Gemessen an der Metrik "Summe der Fehlerquadrate" stellt der serielle GA mit einer Populationsgröße von 200 das beste Ergebnis dar. Die PGA und Insel-PGA ermöglichen auch bei niedriger Populationsgröße eine glatte Kurvenapproximierung. Der serielle GA zeigt bei den Größen 10 und 20 hingegen deutliche Abweichungen von der Referenzkurve.

Somit würden sich der PGA und Insel-PGA auch für Probleme mit großem Speicherbedarf pro Individuum anbieten, zumal dort aufgrund der Verwendung der limitierten Ressource “Shared Memory” die Populationsgröße nicht beliebig groß sein darf.

Gleichwohl leiden die Insel-PGA und PGA trotz der visuell wahrnehmbaren Glätte an einem Genauigkeitsverlust, welches bei kleinen Risikowerten zu sichtbaren Abweichungen von der Referenzkurve führt (linke untere Seite der Kurven).

Hier ist zu prüfen, ob die von Hand geschriebenen Matrix Operationen Probleme bzgl der Genauigkeit aufweisen[1]. Im Moment ist davon auszugehen. Dieser konstante Fehler führt dann auch bei steigenden Populationsgrößen zu einem Ansteigen der Summe der Fehlerquadrate, was das “‘schlechtere” Abschneiden des PGA und Insel-PGA bei der maximalen Populationsgröße in Tabelle 4.2 erklärt.

Abschließend ist festzustellen, dass die hier präsentierten Implementierungen (PGA und Insel-PGA) qualitativ vergleichbare Ergebnisse zum etablierten Framework GALib liefern und für kleinere Populationsgrößen sogar geeigneter sein könnten.

5.1.3 Benchmarks

5.1.3.1 PGA

Die PGA ermöglicht einen sehr guten Speedup (bis zu 22) und ermöglicht stets interaktive Antwortzeiten der Anwendung. Im Gegensatz dazu benötigte der serielle GA teilweise Minuten.

Die PGA Implementierung zeichnet sich durch eine hohe Anzahl von Blöcken aus, welche die GPU selbstständig und optimal auf die SMs verteilen konnte.

Dabei war eine Optimierung der Verwendung von *Shared Memory* der Hauptgrund dafür, dass im Laufe der Entwicklung der maximale Speedup von 8 auf 22 erhöht wurden konnte. Darauf wird im nächsten Kapitel 5.2 detailliert eingegangen.

Zusammenfassend ist zu sagen, dass sich die PGA als Lösung für das in der Einleitung erwähnte Szenario (Beratung im individuellem Kundengespräch) anbietet, da interaktive Antwortzeiten gewährleistet werden können.

[1] Zwischenergebnisse, Floating Point Genauigkeit, Verwendung des `nvcc` Parameters $-use\_fast\_math$

5.1.3.2 Master-Slave PGA

Die Master-Slave PGA hat keinen nennenswerten Vorteil im Vergleich zur CPU Lösung mit GALib gezeigt, sie war i.d.R. langsamer. Erst ab einer Populationsgröße von 2000 ist die GPU Implementierung etwas "schneller", ohne jedoch eine signifikante Auswirkung auf die Gesamtlaufzeit zu bewirken.

Die Fitness-Berechnung mit GALib benötigte für eine Testpopulation von 200 Individuen bereits weniger als 1 msec. Die Fitnessfunkton war also insgesamt nicht rechenintensiv genug. Dies ist auch auf die gute Performance von libeigen([GJ+10] zurückzuführen, und der Tatsache, dass der C++ Compiler den kompletten Code analysieren und entsprechende Funktionen inlinen konnte. Der minimale Zeitaufwand für den Aufruf und Start eines CUDA Kernels lag bei 30ms.

Der Master-Slave PGA Ansatz würde also nur bei einer sehr rechenintensiven Fitnessfunktion Vorteile bieten. Entscheidend ist also der Anteil der Laufzeit der Fitnessfunktion an der Gesamtlaufzeit einer Generation innerhalb der GA.

5.2 PGA Konzepte

5.2.1 Shared Memory Speicherkonzept

Es ist für die optimale Ausnutzung der Rechenkapazität von GPUs unumgänglich, ausreichend viele Blöcke zur Verfügung zu stellen, damit die GPU in Falle gerade "blockierter" Blöcke[2] andere Blöcke auf dem gleichen SM ausführen kann, um die Hardware optimal auszunützen ([NVI13], [Har07]).

Dies hat sich auch in der Implementierung der PGA mit deaktivierten Insel-Modus gezeigt, bei der gleichzeitig hunderte Optimierungsprobleme (=Blöcke) zur Ausführung bereitgestellt wurden. Jedes Optimierungsproblem repräsentierte dabei einen kompletten GA zur Berechnung eines Kurvenpunktes der Effizienzkurve.

Harris bemerkt zu dieser Problematik in [Har07]:

> "Keep resource usage low enough to support multiple active thread blocks per multiprocessor"

[2]Der Grund dafür ist meist ein GPU Hauptspeicher-Zugriff.

Eine dieser entscheidenden aktiven Ressourcen[3] ist dabei das “Shared Memory”, welches inspiriert durch [PJS10] ein zentrales Konzept in dem hier realisierten PGA darstellt. Die Möglichkeit der GPU, einen weiteren Block pro SM zuzuweisen ist somit abhängig davon, wie viel “Shared Memory” noch zu Verfügung steht.

Der Verbrauch von “Shared Memory” innerhalb des PGA ist dabei mindestens “Größe des Genoms eines Individuums multipliziert mit der Größe der Population”[4]. Dieser Verbrauch ist also problem-abhängig. Dies widerspricht der Anforderung an ein generell einsetzbares GA Framework.

Wenn beispielsweise nur 48kb “Shared Memory” pro SM verfügbar sind, die Lösung des Problems mit der PGA jedoch einen Speicherbedarf von mehr als 24kb ausweist, so wird die GPU niemals mehr als einen Block pro SM ausführen und somit die Hardware nicht optimal auslasten. Gleichzeitig gibt es sicherlich Probleme, die selbst mit 48kb nicht zurechtkommen würden. Dieser Aspekt spielt bei [PJS10] keine große Rolle, da ein Individuum dort aus 2 Floats bestand. Bei realen Anwendungen wie hier der Portfolio Optimierung spielt das aber eine große Rolle.

5.2.2 Insel PGA

Bei einer GPU mit n SMs ist die maximale, absolute Rechenleistung einer Insel in der Insel-PGA $1/n$. Dies kann schon im Vergleich zu einer handelsüblichen CPU zu wenig sein.

Weiterhin gibt es ein rein konzeptionelles Problem mit der Insel-PGA. Sie verlässt sich darauf, dass die beteiligten Inseln (=Blöcke) zu einem bestimmten Zeitpunkt auch tatsächlich *aktiv* sind - d.h. einem SM zugeordnet und in Ausführung sind. Ansonsten könnten sich diese auch nicht aktiv an den Migrationsrunden beteiligen. Das ist jedoch eine Laufzeit-Entscheidung der CUDA Implementierung und abhängig davon, wieviele physikalische SMs existieren, und ob die Blöcke tatsächlich gleichzeitig ausgeführt werden können.

So war es bei einem PGA “Shared Memory” Bedarf von 15kb[5] möglich, insgesamt 16 *gleichzeitig* aktive Blöcke (2 aktive Blöcke auf 8 SMs) in der Insel-PGA zu verwenden,

[3] Neben den Registern, d.h. lokalen Variablen

[4] Plus anderen Elementen, z.b. der Fitness pro Individuum und weiteren Shared Memory Pufferbereichen, die für Threadkommunikation und gemeinsame Threadoperationen (z.B. Reduktion) verwendet werden.

[5] Wohlgemerkt eine problemabhängige Größe.

17 jedoch nicht. Erst nach dem “Beenden” einer der 16 Inseln wurde die 17te Insel zugeordnet und gestartet[6]. Somit war der unidirektionale Ring unterbrochen. Dies ist also ein konzeptionelles Problem. Man kann zwar davon ausgehen, dass genau so viele Blöcke gleichzeitig gestartet und ausgeführt werden können, wie physikalische SMs existieren, doch dies führt zu einer weitaus schlechteren Ausnutzung der Ressourcen.

Zusammengefasst widerspricht es dem Konzept von CUDA, innerhalb eines Kernels Abhängigkeiten zwischen den Blöcken zu konstruieren und sich darauf zu verlassen, das gewisse Blöcke zu einem gewissen Zeitpunkt aktiv sind[7]. Wer dies forciert, schränkt die Fähigkeit der GPU ein, die Ressourcen optimal zu verwenden. Der korrekte Ansatz wäre es, mehrere Kernel zu verwenden.

Der Hauptvorteil bei der Verwendung der Insel-PGA ist also in der Verbesserung der Ergebnisse durch die Migration zu sehen. Diese liefert auch dann gute Ergebnisse, wenn die Anzahl der aktiven Inseln (Blöcke) “nur” der Anzahl der SMs entspricht.

[6]Sie lief jedoch bald ganz alleine, da alle anderen 16 Inseln relativ zeitgleich terminierten

[7]Siehe dazu die Beschreibung von CUDA in Kapitel 2.4.2.3

Kapitel 6

Fazit und Ausblick

Die durch diese Arbeit erlangten Erkenntnisgewinne lassen sich zwei Bereiche unterteilen. Zum einen die Erkenntnisse zur Umsetzungsfähigkeit der Portfolio Optimierung mit GAs im Allgemeinen und zum anderen die Erkenntnisgewinne bei der Implementierung von PGAs auf CUDA-basierten GPU Architekturen in diesem Aufgabenkontext .

6.1 Portfolio Optimierung mit GAs

Diese Ausarbeitung hat die grundsätzlichen Anforderungen die Portfolio Optimierung dargelegt und eine performante Umsetzung mit GAs präsentiert.

Dabei zeigte sich ein großer Vorteil von GAs: Sie sind für dieses Lösungsproblem ohne größere Konzeptanpassungen einsetzbar. Somit wären die in Kapitel 2.3 vorgestellten erweiterten Modelle exzellente Kandidaten für eine erweiterte Analyse mit GAs und PGAs.

Eine der Herausforderungen dabei wäre die Konfiguration des GAs (GA Operatoren) für das Optimierungsproblem das erweiterten Modells. Interessant dabei wäre, ob eine reine Implementierung von Operatoren ausreicht, oder ob der GA an sich angepasst werden muss. Es existieren je nach Verfahren mehrere Optimierungsziele (sog. *"Multi-objective optimization"*), welche u.U. Anpassungen für den GA nachziehen.

6.2 CUDA-basierende PGAs

Bei der PGA mit CUDA zeigt sich, dass die Umsetzung von realen Problemstellungen wie der Portfolio Optimierung einen erheblichen Aufwand an Planung und Konzeption miteinbezieht. Die GPU Architektur bietet eigene Konzepte in Hinblick auf Parallelisierung, Programmiermodell und Speichernutzung (Speicher-Hierarchien), die es optimal zu Nutzen gilt.

Einen universellen GA Framework-Ansatz für die PGA auf CUDA vergleichbar zu GALib [Wal08] gibt es im Moment noch nicht.

Im Moment ist eine iterative, experimentelle Vorgehensweise und *"Micro-Benchmarking"* für das Erreichen der maximalen Performance unumgänglich.

Gleichwohl entwickelt sich die Technik und die Forschung in diesem Bereich in einem rasanten Tempo. Insgesamt - das hat der erreichte Speedup von über 20 gezeigt - ist es möglich, gute Ergebnisse mit dieser Architektur zu erzielen.

Als Ausblick wären folgende Dinge zu untersuchen:

Parallelisierung auf Gen-Ebene : Da die eigentliche Fitnessfunktion nur von einem relativ langsamen GPU Thread ausgeführt wird, dürfte es ab der Größenordnung von 20 oder 30 Aktien pro Portfolio sinnvoll sein, auf Genom Ebene zu parallelisieren. Dieser Ansatz eines universellen GA Frameworks wird in [Kol], [Ois+11] und [SNK] angegangen.

Verwendung von GPU Hauptspeicher : Da die neusten GPUs auch mit L1 und L2 Cache ausgestattet sind und ein universelles GA Framework auch mit größeren Problemen zurechtkommen muss, gilt es die stärkere Verwendung von Hauptspeicher zu untersuchen. Shared Memory sollte nur als Zwischenspeicher verwendet werden oder für die Kommunikation der Threads innerhalb eines Blocks. Ansätze in der Forschung gibt es bereits ([Kol], [SNK]). Dabei ist auch die Verwendung von mehreren Kerneln und die Steuerung über die CPU zu evaluieren.

Verbesserung der Verwendung von Shared Memory : Es ist zu prüfen, ob sog. *"Shared-Memory Bank Conflicts"* [Har07] die Performance verschlechtern. Es ist weiterhin der Gebrauch von unnötigen Shared Memory durch die Verwendung von Registern zu minimieren.

OpenMP Portierung : Grundsätzlich wäre auch eine Implementierung mit OpenMP auf Multi-Core CPUs denkbar. Der Großteil des bestehenden Programmcodes ist valides C++. Diese Portierung würde auch eine bessere Vergleichbarkeit der CPU und GPU Implementierung ermöglichen, da beide identische Abläufe und identische Operatoren verwenden würden[1].

[1] Gleichwohl müsste man die Anzahl der Threads und die Parallelisierungskonzepte an die Multi-Core Architektur anpassen.

Literatur

[Bau08] R. Baule. „Optimal portfolio selection for the small investor considering risk and transaction costs“. In: *OR Spectrum* 32.1 (2008), S. 61–76.

[BSZ11] N. Bissantz, V. Steinorth und D. Ziggel. „Stabilität von Diversifikationseffekten im Markowitz-Modell“. In: *AStA Wirtschafts- und Sozialstatistisches Archiv* 5.2 (2011), S. 145–157.

[Buc+04] I. Buck u. a. „Brook for GPUs: stream computing on graphics hardware“. In: *ACM SIGGRAPH 2004 Papers*. SIGGRAPH '04. Los Angeles, California: ACM, 2004, S. 777–786.

[Bäc96] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford, UK: Oxford University Press, 1996. ISBN: 0-19-509971-0.

[Cha+00] T.-J. Chang u. a. „Heuristics for cardinality constrained portfolio optimisation“. In: *Comput. Oper. Res.* 27.13 (Nov. 2000), S. 1271–1302. ISSN: 0305-0548.

[CPG99] E. Cantú-Paz und D. E. Goldberg. „On the Scalability of Parallel Genetic Algorithms.“ In: *Evolutionary Computation* 7.4 (1999), S. 429–449.

[Fly72] M. Flynn. „Some Computer Organizations and Their Effectiveness“. In: *Computers, IEEE Transactions on* C-21.9 (1972), S. 948–960. ISSN: 0018-9340.

[GJ+10] G. Guennebaud, B. Jacob u. a. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0201157675.

[Gro10] K. Group. *The OpenCL Specification*. Sep. 2010.

[Har07] M. Harris. „Optimizing CUDA“. In: *Supercomputing 2007 Tutorial, Reno, NV, USA* (2007).

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems*. second edition, 1992. Ann Arbor, MI: University of Michigan Press, 1975.

[HP11] J. L. Hennessy und D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728.

[HS86] W. D. Hillis und G. L. Steele Jr. „Data Parallel Algorithms“. In: *Commun. ACM* 29.12 (Dez. 1986), S. 1170–1183. ISSN: 0001-0782.

[Kap+03] U. J. Kapasi u. a. „Programmable Stream Processors“. In: *Computer* 36.8 (Aug. 2003), S. 54–62. ISSN: 0018-9162.

[Kol] M. Kolomycki. *Use NVIDIA CUDA technology to create genetic algorithms with extensive population*. Student's Conference (STC), 2013.

[Kum+02] V. Kumar u. a. *Introduction to Parallel Computing*. Benjamin/Cummings, 3. Jan. 2002. ISBN: 0-8053-3170-0.

[LA11] G. Luque und E. Alba. *Parallel Genetic Algorithms: Theory and Real World Applications*. Studies in Computational Intelligence. Springer, 2011. ISBN: 9783642220838.

[Lip06] W.-M. Lippe. *Soft-Computing*. Berlin [u.a.]: Springer, 2006. ISBN: 978-3-540-20972-0.

[LLM07] F. G. Lobo, C. F. Lima und Z. Michalewicz. *Parameter Setting in Evolutionary Algorithms*. 1st. Springer Publishing Company, Incorporated, 2007. ISBN: 3540694315, 9783540694311.

[LMT10] T. V. Luong, N. Melab und E.-G. Talbi. „GPU-based Island Model for Evolutionary Algorithms“. In: *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. GECCO '10. Portland, Oregon, USA: ACM, 2010, S. 1089–1096. ISBN: 978-1-4503-0072-8.

[Lue13] D. Luenberger. *Investment Science*. Oxford University Press, Incorporated, 2013. ISBN: 9780199740086.

[Luk13] S. Luke. *Essentials of Metaheuristics*. second. Lulu, 2013. URL: `http://cs.gmu.edu/~sean/book/metaheuristics/`.

[Mar05] D. Maringer. *Portfolio Management with Heuristic Optimization*. Advances in Computational Management Science. Springer, 2005. ISBN: 9780387258522.

[Mar52] H. Markowitz. „Portfolio Selection“. In: *The Journal of Finance* 7.1 (März 1952), S. 77–91. ISSN: 00221082.

[Mar91] H. M. Markowitz. „Foundations of Portfolio Theory“. In: *Journal of Finance* 46.2 (Juni 1991), S. 469–77.

[Mat] A. Matuszak. *Modern Portfolio Theory*. `http://economistatlarge.com/portfolio-theory`. Stand: 20.11.2013.

[Nic+08] J. Nickolls u. a. „Scalable Parallel Programming with CUDA“. In: *Queue* 6.2 (März 2008), S. 40–53. ISSN: 1542-7730.

[Nob] Nobelprize.org. *The Prize in Economics 1990 - Press Release*. `http://www.nobelprize.org/nobel_prizes/economic-sciences/laureates/1990/press.html`. Stand: 10.11.2013.

[NP99] M. Nowostawski und R. Poli. „Parallel Genetic Algorithm Taxonomy“. In: *Proceedings of the Third International*. IEEE, 1999, S. 88–92.

[Ois+11] M. Oiso u. a. „Implementing Genetic Algorithms to CUDA Environment using data parallelization“. In: *Technical Gazette* 18.4 (2011), S. 511–517.

[Paz98] E. C. Paz. „A Survey of Parallel Genetic Algorithms“. In: *Calculateurs Paralleles, Reseaux et Systems Repartis* 10.2 (1998), S. 141–171.

[PJS10] P. Pospichal, J. Jaros und J. Schwarz. „Parallel Genetic Algorithm on the CUDA Architecture“. In: *Applications of Evolutionary Computation*. Hrsg. von C. Chio u. a. Bd. 6024. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, S. 442–451. ISBN: 978-3-642-12238-5.

[Rec73] I. Rechenberg. *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Problemata 15. Stuttgart-Bad Cannstatt: Frommann-Holzboog, 1973.

[SAT93] A. L. S. Arnone und Tettamanzi. „A genetic approach to portfolio selection“. In: *Journal on Neural and Mass-Parallel Computing and Information Systems* 3 (1993), S. 597–604.

[SD06] P. Skolpadungket und K. P. Dahal. „A survey on portfolio optimisation with metaheuristics“. In: *SKIMA 2006*. 2006.

[SK10] J. Sanders und E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Addison-Wesley Professional, 2010. ISBN: 0131387685, 9780131387683.

[SNK] R. Shah, P Narayanan und K. Kothapalli. *GPU-Accelerated Genetic Algorithms*. Indian Institute of Information Technology (IIIT-A).

[Sut05] H. Sutter. „The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software“. In: *Dr. Dobb's Journal* 30.3 (2005), S. 202–210.

[Wal08] M. Wall. *GAlib, A C++ Library of Genetic Algorithm Components*. http://lancet.mit.edu/ga/. 2008.

[Wik13] Wikipedia. *Das Gesetz der großen Zahlen*. [Online; Zugriff vom 2.12.2013]. 2013. URL: `http://de.wikipedia.org/wiki/Gesetz_der_gro%C3%9Fen_Zahlen`.

[WWP09] S. Williams, A. Waterman und D. Patterson. „Roofline: An Insightful Visual Performance Model for Multicore Architectures“. In: *Commun. ACM* 52.4 (Apr. 2009), S. 65–76. ISSN: 0001-0782.

[NVI13] NVIDIA. *NVIDIA CUDA Programming Guide 5.5*. 2013.

[R C13] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2013. URL: `http://www.R-project.org/`.

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfaßt habe.

Amanjit Gill

Berlin, den 12. 12 2013