Ein Framework zur automatisierten Speedup-Vorhersage und Parallelisierung Populationsbasierter Algorithmen

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades
Doktor-Ingenieur
(abgekürzt: Dr.-Ing.)
genehmigte Dissertation

von
M. Sc. Ioannis Zgeras

geboren am 13.12.1982
in Hannover

2014

1. Referent apl. Prof. Dr. Jürgen Brehm
2. Referent Prof. Dr. Michael Rohs
3. Referent Prof. Dr. Kurt Schneider
Tag der Promotion 23.10.2014

Zusammenfassung

Schlagworte: Multi-Core, GPGPU, Automatisierte Parallelisierung, Optimierung, Genetische Algorithmen, Particle Swarm Optimization, Speedup-Vorhersage, Software-Hardware-Mapping

Heutige Computersysteme sind hochgradig homogen und bestehen aus einer Vielzahl unterschiedlicher Berechnungseinheiten. Während sich zu Beginn der Parallelrechner die Parallelität aus vielen Rechenknoten ergab, sind heutige Rechenknoten für sich genommen schon hochkomplexe parallele Einheiten. Diese bestehen aus einfachen Multi-Core CPUs aber auch aus spezialisierten, hochparallelen Berechnungseinheiten, wie *General Purpose Graphics Processing Units* (GPGPUs). Ein großer Vorteil dieser Recheneinheiten sind die relativ günstigen Beschaffungskosten. Dadurch wird einer großen Anzahl an Entwicklern die Möglichkeit eröffnet, kostengünstig ihre Programme durch Parallelisierung zu beschleunigen. Das Ausnutzen dieser Parallelität erfordert von den Entwicklern jedoch ein drastisches Umdenken bei der Programmierung. Eine einfache Partitionierung des Programms auf einzelne Rechenknoten ist für eine effiziente Parallelisierung nicht ausreichend. Insbesondere GPGPUs erfordern, aufgrund ihrer hochkomplexen, mehrschichtigen Speicherarchitekturen, einen weitaus größeren Aufwand. Die Leistung der parallelen Programme hängt nicht nur von der richtigen Partitionierung, sondern auch von dem effizienten Ausnutzen der korrekten Speicherbereiche, der Art der Speicherzugriffe und dem Datentransfer ab. Weiterhin lassen sich die zu erwartende Leistungssteigerung einer Parallelisierung und die optimale Wahl der vorhandenen parallelen Recheneinheiten nicht vorher abschätzen.

Diese Arbeit beschreibt ein Framework, mit dessen Hilfe der Entwickler bei der Parallelisierung auf modernen Architekturen unterstützt wird. Es werden Verfahren entwickelt, mit denen sich das optimale Software-Hardware-Mapping und die zu erwartende Leistungssteigerung voraussagen lassen. Des Weiteren wird ein Verfahren vorgestellt, mit dem sich serielle Software teilautomatisiert in parallele Software für Multi-Core CPUs und GPGPUs transformieren lässt.

Untersuchungen in dieser Arbeit haben gezeigt, dass die Verfahren zum größten Teil zu einem optimalen Software-Hardware-Mapping führen. Ebenso wird die zu erwartende Leistungssteigerung mit einem geringen Fehler vorhergesagt. Die automatisiert generierten parallelen Programme konnten eine ähnliche Leistung erreichen, wie manuell erstellte Implementierungen.

Abstract

Keywords: Multi-Core, GPGPU, Automated Parallelization, Optimization, Genetic Algorithms, Particle Swarm Optimization, Speedup Estimation, Software Hardware Mapping

Today's computer systems are highly homogeneous and consist of a variety of different compute units. While at the start of parallel machines parallelism arose from many compute nodes, today's compute nodes are highly complex parallel units taken by itself. These nodes consist of simple multi-core CPUs but also of specialized, highly parallel compute units, such as *General purpose graphics processing units* (GPGPUs). A great advantage of these processing units are the relatively low acquisition costs. As a result, a large number of developers have the ability to accelerate their programs by parallelization. However, to exploit this parallelism requires a drastic rethinking of the developers in programming. A simple partitioning of the program on individual compute nodes is not sufficient for efficient parallelization. In particular, GPGPUs require additional effort, due to their highly complex multi-layered memory architectures. The performance of the parallel programs depends not only on the right partitioning, but also on efficient utilization of the correct memory areas, the memory access and data transfer. Furthermore, the expected increase in performance of a parallelized program and the optimal software-hardware-mapping can not be estimated in advance to actually parallelizing the program.

This work describes a framework that supports the developer in parallelizing programs on modern parallel architectures. The developed approach

predicts the optimal software-hardware-mapping and the expected increase in performance by parallelizing the program. Furthermore, a method is presented, which can - to a certain degree - automatically transform serial source code into parallel source code for multi-core CPUs and GPGPUs. Evaluation studies in this work have shown that the presented techniques lead to an optimal software-hardware-mapping in most of the time. Also, the expected performance increase is predicted with a small error. The automatically generated parallel programs achieve similar performance as manually created implementations.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Entwicklung in der Halbleiterindustrie seit dem Jahrhundertwechsel führte zu grundlegenden Veränderungen im Bereich der Rechnerarchitektur. Während bis zum damaligen Zeitpunkt der Großteil des Leistungsgewinns durch Erhöhung der Taktrate und immer kompliziertere Cache- und Pipeline-Architekturen erreicht wurde [1],[2], änderte sich dies 2001 mit der Einführung des ersten nicht eingebetteten Multi-Core Prozessors von IBM [3]. Die Entwicklung von bezahlbaren Multi-Core Prozessoren veränderte den Trend [4], indem nicht mehr nur parallel auf mehreren Maschinen mit Hilfe von *MPI* (Message Passing Interface) programmiert werden konnte, sondern auf einem einzelnen Prozessor mit einer *Shared-Memory*-Architektur (s. Abschn. 2.1.1). Diesen Trend verstärkte NVIDIA mit der Entwicklung der ersten sogenannten *General Purpose Graphics Processing Units* [5],[6] (GPGPUs) (s. Abschn. 2.1.2), Grafikkarten, mit denen ähnliche parallele Berechnungen durchführbar sind wie mit üblichen Multi-Core Prozessoren [7].

Die Entwicklung von immer komplexeren und leistungsstärkeren parallelen Architekturen erforderte ein grundlegendes Umdenken bei der Softwareentwicklung [8]. Während bei seriellen Programmen die einzelnen Befehle nacheinander von der CPU abgearbeitet werden, erfordert es bei der parallelen Programmierung ein Aufbrechen der üblichen Abarbeitungsreihenfolge zur parallelen Abarbeitung des Programms. Dabei sind prinzipiell zwei Vorgehensweisen möglich: Entweder werden einzelne unabhängigen Code Abschnitte nebenläufig berechnet (*task decomposition*) oder ein Code Abschnitt, meistens

Schleifen, parallel von mehreren Threads abgearbeitet (*data decomposition*) [9]. Letztere Methode wird auch als *Single Instruction Multiple Data* (SIMD) bezeichnet und ist die präferierte Methode bei der Parallelisierung auf GPGPUs [10]. Auf Multi-Core Prozessoren ist OpenMP [11] (s. Abschn. 2.1.1.1), eine Programmierschnittstelle für C++, die präferierte Methode, um parallele Programmierung zu vereinfachen.

Die Entscheidung des Entwicklers bezüglich einer der oben beschriebenen geeigneten Parallelisierungsmethoden ist nicht das einzige Problem bei der Parallelisierung. Das Erkennen von Datenabhängigkeiten [12] zwischen den einzelnen Code Abschnitten ist besonders in komplexen Projekten nicht trivial. Weiterhin bildet für Entwickler die Einarbeitung in neue Frameworks [13], die für die parallele Programmierung verwendet werden, und die komplexen Architekturen von parallelen Maschinen eine enorme Einstiegshürde. Insbesondere die Programmierung auf GPGPUs erfordert ein tiefes Verständnis der verschiedenen Speicher- und Threadebenen (s. Abschnitt 2.1.2 und 2.1.2.1).

Außer den konkreten Problemen bei der Implementierung von parallelen Programmen besteht weiterhin das Problem der Software-Hardware Abbildung und das Problem der korrekten Laufzeitparameterwahl. Heutige Rechnerarchitekturen sind hochgradig heterogen und bestehen aus mehreren Recheneinheiten, üblicherweise ein Multi-Core Prozessor, mindestens eine GPGPU und möglicherweise spezialisierte Hardware wie *FPGAs* (Field Programmable Gate Arrays) [14], welche alle unterschiedliche Charakteristika aufweisen. Bei der Parallelisierung des Programms muss der Entwickler entscheiden, auf welche Hardware er welche Programmteile auslagern möchte. Dabei gilt es üblicherweise die *bestmögliche Systemleistung* [1] zu erreichen. Diese Entscheidung kann nur mittels Erfahrungswerten und Heuristiken gefällt werden oder konkret, wenn das Programm tatsächlich parallelisiert und auf den einzelnen Maschinen ausgeführt wurde. Einen großen Einfluss auf die Laufzeit des parallelen Programms haben weiterhin die sogenannten Laufzeitparameter. Diese Parameter sind z.B. die Anzahl der Threads, mit denen das Programm ausgeführt wird oder die Topologie, wie die einzelnen Threads angeordnet werden (s. Ab-

[1] Im weiteren Verlauf der Arbeit wird der englische Begriff *Speedup* verwendet

schn. 2.1.2.1). Auch hier gilt, dass die optimalen Parameter meist nur durch Ausprobieren verschiedener Kombinationen ermittelt werden können.

Immer komplexere Aufgabenstellungen und größere Datenmengen machen eine Parallelisierung trotz der beschriebenen Probleme dringend notwendig. Auch zeitkritische Probleme, bei denen nur ein festgelegtes Zeitfenster für die Berechnung gegeben ist, profitieren von einer Parallelisierung. Dabei sind besonders Problemstellungen mit einer großen Datenmenge, auf der einzelne Operationen durchgeführt werden, dazu geeignet, die Probleme in kleinere Teilprobleme aufzuteilen und parallel zu berechnen. Eine Algorithmenklasse, die für die Berechnung solcher Art von Problemen besonders gut geeignet ist, ist die Klasse der *populationsbasierten Algorithmen* (PBAs) (s. Abschnitt 2.2). Dabei handelt es sich um Heuristiken, die an Analogien aus der Natur angelehnt sind, wie z.B. *Genetische Algorithmen* (GAs) [15] [16] (s. Abschnitt 2.2.1) oder *Particle Swarm Optimization* (PSO) [17] (s. Abschnitt 2.2.2). Die Gemeinsamkeit dieser Algorithmen bildet die iterative Ausführung von Operationen, wobei die Daten auf einzelne sogenannte *Individuen*, die in einer *Population* zusammengefasst sind, aufgeteilt werden. Dabei führen alle Individuen einer Population die gleichen Operationen auf unterschiedlichen Daten aus. Diese Individuen lassen sich eins zu eins auf Threads abbilden. Dadurch entspricht das Prinzip von *PBAs* dem von SIMD. Dieses ist das selbe Prinzip, welches durch die Architektur von GPGPUs unterstützt wird, was diese Algorithmenklasse im hohen Maß für eine Parallelisierung prädestiniert. Weiterhin sind die Datenabhängigkeiten zwischen den einzelnen Individuen (Threads) gering und beschränken sich meist auf den Austausch der besten Lösungen. Dieser Faktor spielt bei der Ausführung von Algorithmen auf parallelen Maschinen eine große Rolle, da Informationen zwischen Threads meist nur über begrenzten Speicher (auf GPGPUs) oder mit Hilfe von Synchronisierungspunkten ausgetauscht werden können. Der Informationsaustausch zwischen den einzelnen Threads bildet dementsprechend einen Engpass bei der Ausführung und sollte möglichst minimiert werden.

Die beschriebenen Probleme und Herausforderungen an die Entwickler bei der Programmierung von parallelen Programmen erfordern Werkzeuge, die sowohl die Komplexität minimieren, als auch den zeitlichen Overhead, den das

Entwickeln von parallelen Programmen zwangsläufig mitbringt. Diese Art von Werkzeugen bilden das Thema dieser Arbeit. Die Arbeit konzentriert sich auf zwei Phasen der Softwareentwicklung, die sogenannten *Pre-Code* und *Post-Code*-Phasen. Bei der *Pre-Code*-Phase handelt es sich um den Zeitpunkt in der Softwareentwicklung, in der noch kein Quellcode geschrieben wurde, wo jedoch Informationen über das spätere Programm vorliegen. Solche Informationen können z.B. die Art der Funktionen sein, die ein Programm berechnen soll. In der *Post-Code*-Phase wiederum existiert bereits lauffähiger serieller Quellcode, jedoch kein paralleler Code. Diese Arbeit stellt dem Entwickler in den zwei Phasen der Softwareentwicklung Werkzeuge zur Verfügung, die in drei Hauptkategorien unterteilt sind:

1. Werkzeuge zur Bestimmung des potentiellen Speedups des seriellen Programms bei einer potentiellen parallelen Ausführung auf die vorhandene Hardware.

2. Werkzeuge zur Bestimmung von Laufzeitparametern für das potentiell parallele Programm.

3. Werkzeuge, die die Informationen über den zu erwartenden Speedup verwenden, um einzelne Code-Abschnitte semi-automatisch zu parallelisieren.

1.2 Problembeschreibung

Aus dem vorangegangenen Abschnitt ergeben sich unterschiedliche Problemstellungen bei der Parallelisierung von Quellcode, die jeweils in unterschiedliche Teilgebiete der Softwareentwicklung einzuordnen sind:

1. Parallelisierung von Software ist aufwendig und erfordert die Einarbeitung in spezialisierte Frameworks.

2. Der erzielte Speedup eines parallelen Programms kann nicht ohne Weiteres vorausgesagt werden und basiert zum größten Teil auf Erfahrungswerten.

3. Die Wahl der zu parallelisierende Programmabschnitte beruht ebenso auf Erfahrungswerten und es werden tendenziell die Abschnitte parallelisiert, die die meiste Berechnungszeit beanspruchen.

4. Die Ausführung von parallelen Programmen ist hochgradig von den richtigen Laufzeitparametern (wie z.B. der Anzahl der genutzten Threads) abhängig. Die korrekte Konfiguration lässt sich nur durch Ausprobieren und Erfahrungswerte bestimmen.

5. Bei heterogenen Computerarchitekturen mit mehreren unterschiedlichen Berechnungseinheiten ist die Wahl der besten Abbildung auf die korrekte Hardware nicht trivial.

6. Einige Algorithmen laufen parallel ausgeführt langsamer, als seriell. In solchen Fällen kann eine Transformation des Quellcodes zu einem Geschwindigkeitsgewinn bei der Ausführung führen.

1.2.1 Probleme während der Entwicklung von parallelen Programmen

In diesem Abschnitt werden die oben genannten Probleme bei der Entwicklung eines (parallelisierbaren) Programms beschrieben. Diese sind mit erheblichen Herausforderungen für den Entwickler bei der Planung der Software verbunden, sollte eine parallele Implementierung vorgesehen sein. Zuerst ist die richtige Hardware für das Programm zu wählen, was gegebenenfalls eine Einarbeitung in die benötigten Tools und Architekturen notwendig macht. Die Wahl der Hardware beruht im Allgemeinen auf Erfahrungswerten des Entwicklers und kann erst nach einer konkreten Parallelisierung des Programms evaluiert werden. Das kann dazu führen, dass bei einer falschen Annahme das Programm auf einer anderen Hardware eine weitaus kürzere Laufzeit aufweisen würde. Ohne geeignete Werkzeuge fehlt dem Entwickler jeglicher Vergleich der Laufzeit des gleichen Programms für unterschiedliche Architekturen. Zuletzt ist es erforderlich zu entscheiden, welche Programmabschnitte parallelisiert werden sollen. Meistens führt eine komplette Parallelisierung des Programms zu Ge-

schwindigkeitseinbußen, da die Laufzeiten von einzelnen Abschnitten zu kurz sind, um den Overhead [18] einer Parallelisierung zu rechtfertigen. Weiterhin kann es vorkommen, dass die vorhandenen Operationen prinzipiell schneller seriell ausgeführt werden können. Dies kann z.B. bei häufig notwendigen Synchronisierungen der Fall sein. Auch hier gilt, die korrekte Wahl der Abschnitte kann letztendlich nur durch konkretes Ausprobieren mit Sicherheit festgestellt werden. Einzelne Algorithmen oder Teile davon eignen sich möglicherweise nicht dazu, auf paralleler Hardware ausgeführt zu werden. In solchen Fällen können möglicherweise Transformationen im Quellcode zu einer effizienteren parallelen Ausführung des Programms führen.

1.2.2 Probleme bei der Ausführung von parallelen Programmen

Den Hauptteil der Probleme bei der Parallelisierung bilden die im letzten Abschnitt beschriebenen Probleme während der Entwicklung. Jedoch hängt die optimale Ausführung von parallelen Programmen ebenso von der Wahl der korrekten Laufzeitparameter ab. Je nach verwendeter Architektur sind hierbei unterschiedliche Parameter zu berücksichtigen. Alle parallelen Architekturen benötigen eine korrekte Anzahl an Threads, mit denen ein Programm durchgeführt wird. Ist diese Anzahl zu niedrig gewählt, wird Potential bei der Laufzeit nicht ausgereizt. Ist sie hingegeben zu hoch, führt dies zu einem großen Overhead bei der Threadverwaltung, und dadurch ebenso zu längeren Laufzeiten des Programms. Die korrekte Wahl der Threadgröße ist somit entscheidend für eine optimale Laufzeit [19]. Einige Architekturen benötigen weiterhin eine komplexe Thread-Topologie (s. Abschn. 2.1.2.1), was diese Problematik weiter verschärft.

1.3 Beitrag zur Forschung

Die vorliegende Dissertation beschäftigt sich mit den in diesem Kapitel beschriebenen Problemen und nutzt Techniken aus unterschiedlichen Bereichen der Forschung, um diese zu lösen. Hierzu wurde das Framework *Code Analysis*

Speedup Estimation and Parallelization (*CASEP*) entwickelt. Wie in Abb. 1.1 dargestellt, unterstützt *CASEP* den Entwickler sowohl während der Entwicklungsphase von paralleler Software, als auch während der Produktivphase (zur Laufzeit des fertigen Programms).

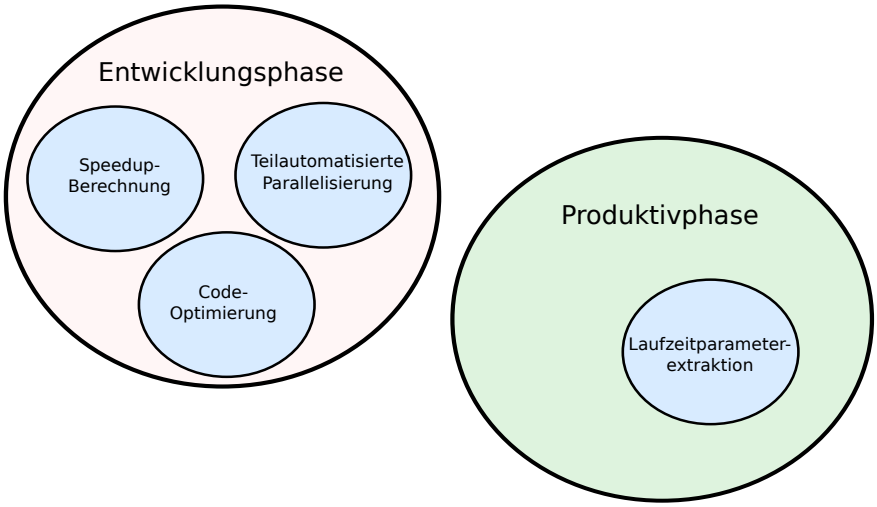


Abbildung 1.1: Forschungsfelder dieser Arbeit

Es werden von *CASEP* alle sechs (in Abschn. 1.2) beschriebenen Probleme abgedeckt. Es wurden zwei verschiedene Ansätze für die Speedup-Vorhersage serieller Software entwickelt, die mit unterschiedlichen Methoden eine Teilannahme für den Speedup bereitstellen, aus denen von *CASEP* die endgültige Speedup-Vorhersage erstellt wird. Es wurde bewusst auf ein breites Vorhersagemodell gesetzt, mit dem Ziel, die Schwächen der einzelnen Methoden durch Verwenden von weiteren Vorhersagemethoden auszugleichen. Die so ermittelte Speedup-Vorhersage bietet Informationen darüber, auf welcher vorhandenen Hardware (sowohl parallele als auch serielle Hardware) der größte Speedup bei der Ausführung des zu untersuchenden Programms zu erwarten ist. Im nächsten Schritt kann die Vorhersage dazu verwendet werden, teilautomatisiert einzelne Bereiche des Programms zu parallelisieren. *CASEP* führt dabei gegebenenfalls Transformationen im Quellcode durch, die der Optimierung der Laufzeit und/oder der Optimierung der Ergebnisse dienen.

CASEP bietet mit Hilfe der Analysetools, die ebenso für die Speedup-Vorhersage verwendet werden, die Möglichkeit der Extraktion von Laufzeitpa-

rametern. Dabei werden unterschiedliche Parameter untersucht und diejenige vorgeschlagenen, die die besten Laufzeitergebnisse liefern. Somit deckt diese Dissertation die in diesem Kapitel beschriebenen Probleme vollständig ab und bietet für den Entwickler ein Werkzeug, das die Entwicklung von Software für parallele Hardware signifikant vereinfacht.

Der Schwerpunkt der untersuchten Algorithmen als Eingabe für die in dieser Arbeit beschriebenen Prinzipien bilden populationsbasierte Algorithmen (s. Abschn. 2.2). Diese Algorithmenklasse wurde aufgrund ihrer Grundstruktur exemplarisch ausgewählt, da sie sich potentiell besonders gut für eine Parallelisierung und Optimierung eignet. Einzelheiten zu den gewählten Algorithmen finden sich im referenzierten Abschnitt. Obwohl die Methoden dieser Arbeit anhand dieser Algorithmenklasse beschrieben werden, sind die meisten der hier beschriebenen Techniken auch auf andere Algorithmenklassen erweiterbar, was in Abschnitt 5 diskutiert wird.

1.4 Aufbau der Arbeit

Die hier vorliegende Arbeit ist in folgende Kapitel gegliedert: Zuerst werden in Kapitel 2 der Stand der Technik der verwendeten Technologien und verwandte Arbeiten diskutiert. Als nächstes wird in Kapitel 3 die Systemarchitektur von *CASEP* beschrieben, auf die in Kapitel 4 im Detail eingegangen wird. Eine Generalisierung von *CASEP* wird in Kapitel 5 diskutiert. Die Evaluationsergebnisse der beschriebenen Module und des Gesamtframeworks werden in Kapitel 6 aufgezeigt und diskutiert. Die Arbeit schließt mit einer Zusammenfassung in Kapitel 7 und einem Ausblick auf mögliche Erweiterungen.

2 Stand der Technik

Inhaltsangabe

Das nachfolgende Kapitel behandelt den aktuellen Stand der Forschung im Bereich dieser Dissertation und beschreibt die verwendeten Frameworks und Technologien. Abschnitt 2.1 enthält aktuelle parallele Architekturen und Programmiermodelle mit ihren Vor-und Nachteilen in Hinblick auf diese Arbeit. In Abschnitt 2.2 werden die Algorithmenklasse der populationsbasierten Algorithmen beschrieben und Anhand von zwei populären Mitgliedern dieser Klasse, die den Schwerpunkt der Algorithmen des *CASEP* Frameworks bilden, Gemeinsamkeiten und Differenzen aufgezeigt. Abschnitt 2.4 beschäftigt sich mit verwandten Arbeiten im Bereich der Speedup-Vorhersage, während Abschnitt 2.5 Beiträge im Bereich der automatisierten Parallelisierung würdigt. Im letzten Abschnitt (2.6) wird das in *CASEP* eingesetzte Code-Manipulationswerkzeug vorgestellt.

2.1 Parallele Architekturen und Programmiermodelle

Parallele Architekturen haben eine lange Geschichte in der Computerarchitektur. 1956 initiierte IBM in Los Alamos das Projekt *STRETCH*[20] mit dem Ziel, die Rechenkraft damaliger Maschinen durch Parallelisierung zu verhundertfachen. Der Trend zur Entwicklung immer größerer Maschinen mit parallelen Recheneinheiten zog sich durch die nächsten Jahrzehnte. Während parallele Architekturen den Nutzern von Rechenzentren vorenthalten waren, änderte sich dies 2001 mit der Entwicklung des *Power4* [3], des ersten nicht eingebetteten Multi-Prozessors, grundlegend.

Heutige parallele Architekturen lassen sich grob in drei Ebenen einteilen, die in Abb. 2.1 abgebildet sind. Die oberste Ebene bildet die *globale Parallelisierung*. In dieser Ebene sind einzelne Rechenknoten über ein Verbindungsnetzwerk zu einem Rechencluster zusammengeschaltet. Üblicherweise wird hierfür eine *Ethernet*-Verbindung [21] oder die schnellere *InfiniBand*-Verbindung [22] verwendet. Die einzelnen Knoten besitzen keinen gemeinsamen Speicher. Zum Ausnutzen der Parallelität wird i. A. der nachrichtenbasierte Ansatz *Message Passing Interface* (MPI) [23] verwendet. In der nächsten Ebene (*grobkörnige Parallelisierung*) wird die Parallelität der einzelnen Rechenknoten ausgenutzt. Rechenknoten heutiger Cluster bestehen in der Regel aus mehreren Multi-Core CPUs (s. Abschn. 2.1.1) mit einem gemeinsamen Speicher (*Dynamic Random Access Memory*; DRAM). Diese Recheneinheit wird im folgenden als *host* bezeichnet. Um ein Programm parallel auf einer Multi-Core CPU auszuführen, hat sich die C/C++ Erweiterung *OpenMP* (s. Abschn. 2.1.1.1) als Standard etabliert. Dabei wird ein Programm auf den einzelnen Kernen einer Multi-Core CPU ausgeführt. Für die Parallelisierung auf mehreren Multi-Cores eines Knotens wird sowohl MPI, als auch OpenMP verwendet. MPI wird dazu verwendet Nachrichten zwischen den einzelnen Multi-Cores auszutauschen, während OpenMP für die Parallelität auf den jeweiligen Multi-Cores zuständig ist. Rechenknoten besitzen weiterhin häufig GPGPUs (s. Abschn. 2.1.2), die über eine Schnittstelle (in der Regel *PCI Express* [24]) mit dem *host* verbunden

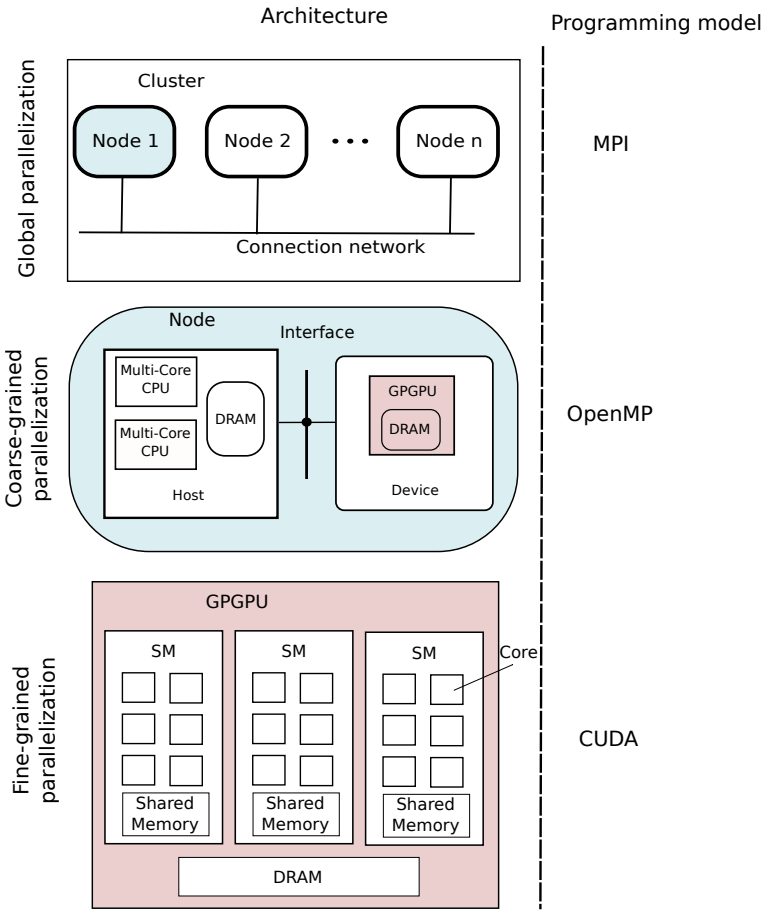


Abbildung 2.1: Übersicht verschiedener Parallelisierungsebenen

ist. Die GPGPU Recheneinheit wird im Folgenden als *device* bezeichnet. Die Parallelisierung auf der GPGPU (s. Abschn. 2.1.2) bildet die dritte Parallelisierungsebene (*feinkörnige Parallelisierung*). Eine GPGPU ist in mehrere *Streaming Multi-Processors* (SM) unterteilt, die jeweils mehrere Rechenknoten (cores) beinhalten. Die *cores* eines SM können dabei über einen gemeinsamen Speicher kommunizieren. Zur Kommunikation zwischen den *cores* verschiedener SMs wird ein gemeinsamer DRAM verwendet. Zur parallelen Programmierung auf GPGPUs hat sich mittlerweile das von NVIDIA entwickelte CUDA (s. Abschn. 2.1.2.1) als Standard durchgesetzt.

In den folgenden Abschnitten werden die hier beschriebenen parallelen Architekturen und Programmiermodelle betrachtet und es wird auf ihre Vor-und

Nachteile in Bezug auf diese Arbeit eingegangen. Die oberste Ebene (*globale Parallelisierung*) wird in dieser Arbeit nicht weiter verwendet und somit hier nicht näher betrachtet.

2.1.1 Multi-Cores - Grobkörnige Parallelisierung

Bei Multi-Core CPUs handelt es sich um einen Verbund von mehreren Rechenkernen auf einem Chip, denen eine komplexe Cache-Hierarchie und Pipeline-Architektur für die Berechnungen zur Verfügung steht. Obwohl heutige Multi-Core CPUs üblicherweise aus vier bis zwölf physischen Kernen bestehen (Intel Sandy Bridge E), ist ihre Leistungsfähigkeit mit den im vorherigen Abschnitt beschriebenen GPGPUs vergleichbar [25]. Dafür sorgt die weitaus komplexere Architektur der einzelnen Rechenkerne zusammen mit der Cache- und Pipeline-Architektur, die eine optimierte Ausführung von Quellcode ermöglichen. Während GPGPUs darauf optimiert sind, einen hohen Durchsatz zu gewährleisten, sind Multi-Cores darauf ausgelegt, den vorhanden Quellcode möglichst effizient zu berechnen. Dies erfolgt beispielsweise durch Sprungvorhersagen, bei denen die zu erwartenden Befehle und Daten zuvor geladen werden, bevor sie bei der Ausführung benötigt werden.

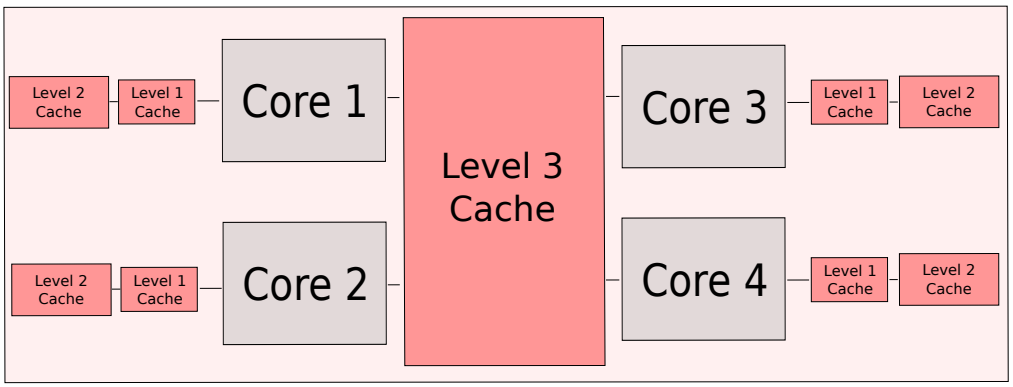


Abbildung 2.2: Multi-Core CPU Architektur

Abb. 2.2 stellt eine vereinfachte Multi-Core Architektur (in diesem Fall *Quad-Core*) dar. Zu erkennen sind die vier Recheneinheiten (Core 1-4) und die zugehörigen privaten Speicher (Level 1 und 2 Caches). Der Informationsaustausch findet über den gemeinsamen größeren, aber auch langsameren, Level 3

Cache statt. Die Hardwarehersteller versuchen, diese Cache-Hierarchie immer weiter zu optimieren, indem sie beispielsweise nicht verwendeten Cache von einzelnen Kernen den anderen Cores zur Verfügung stellen, oder indem eine weitere Hierarchie-Ebene eingeführt wird. In dieser können z.B. jeweils zwei Kerne über einen gemeinsamen Cache verfügen.

Bei der Implementierung von Programmen auf Multi-Cores wird, anders als auf GPGPUs, der Großteil der Verwaltung vom Compiler und der darunter liegenden Architektur übernommen (s. Abschn. 2.1.1.1). Die Entwickler müssen sich keine Gedanken über das korrekte Kopieren von Daten auf die jeweiligen Speicher machen. Nichtsdestotrotz erfordert ein optimales Abbilden von Software auf Multi-Core Prozessoren das Beachten der Hardware-Parameter.

2.1.1.1 OpenMP

OpenMP [26] ist ein Programmiermodell zur speicherbasierten Parallelisierung von Quellcode auf Multi-Core-Maschinen. Bei speicherbasierten Verfahren werden die Informationen zwischen den einzelnen Rechenkernen, bzw. Threads (auf der Quellcode-Ebene), über einen gemeinsamen Speicher übertragen und synchronisiert. OpenMP ist mittlerweile zum Standard bei der Parallelisierung auf Multi-Core Maschinen geworden und wird von allen gängigen C/C++ Compilern als Erweiterung zu C/C++ unterstützt. OpenMP unterstützt weiterhin auch die Programmiersprache *Fortran* [27], die jedoch nicht relevant für die vorliegende Arbeit ist.

Für das Markieren der zu parallelisierenden Abschnitte nutzt OpenMP Direktiven, die in den C/C++ Code eingefügt werden. Diese Direktiven werden von den Compilern nicht beachtet, wenn bei der Kompilierung nicht explizit mittels einer Option OpenMP unterstützt werden soll. Somit kann OpenMP Code sowohl seriell, als auch parallel ausgeführt werden und muss nicht umgeschrieben werden.

2.1.2 Grafikkarten - Feinkörnige Parallelisierung

Grafikkarten wurden zu Beginn ihrer Entwicklung ausdrücklich zur Darstellung von visuellen Effekten auf Bildschirmen verwendet. NVIDIA entwickelte

1999 die erste Grafikkarte (*GeForce 256*), die hardwaregestützte Transformationen und Lichteffekte berechnen konnte. Diese Grafikkarte war ein Meilenstein in der Entwicklung, was auch eine Umbenennung von Grafikkarten in *Graphics Processing Units* (GPUs) zu Folge hatte [28]. Entwickler fingen an für die Vertex- und Pixel-Shader der Grafikkarte Programme zu schreiben, indem sie die Programmier-Schnittstellen OpenGL[1] oder DirectX[2] nutzen. Shader sind Recheneinheiten, die die Berechnung der darzustellenden Objekte übernehmen. Dabei übernehmen Vertex-Shader die Berechnung von geometrischen Transformationen, während die Pixel-Shader für die Berechnung der einzelnen Bildpunkte zuständig sind. Die Firma ATI veröffentlichte für die Spielekonsole Xbox 360 die erste GPU mit gemeinsamen Shadern [29], NVIDIA folgte mit der ersten GPU mit gemeinsamen Shadern für den PC (GeForce 8800). Die komplexen gemeinsamen Shader ermöglichten den Entwicklern, komplexere und aufwendigere Programme für die GPUs zu schreiben, was den Begriff der *General Purpose Graphics Processing Unit* (GPGPU) prägte.

Die gemeinsamen Shader-Prozessoren entwickelten sich bei NVIDIA unter dem Namen CUDA-Cores. NVIDIA veröffentlichte zur vereinfachten Programmierung auf GPGPUs eine Erweiterung für die Programmiersprache C/C++ mit dem Namen CUDA (s. Abschn. 2.1.2.1). Dabei lehnt sich die Programmierung der Grafikkarte stark an die darunter liegende Architektur an. In Bild 2.3 ist eine vereinfachte Darstellung einer GPGPU Architektur von NVIDIA dargestellt. Sie enthält alle für diese Arbeit wichtige Module. Eine detaillierte Beschreibung und Darstellung der einzelnen Komponenten findet sich in [30].

Die abgebildete Architektur entspricht dabei dem Stand der sogenannten *Tesla-Architektur*. Es ist deutlich eine Trennung zwischen der Host-CPU und der GPGPU zu erkennen. Der Informationsaustausch erfolgt über eine Schnittstelle (*Host-Interface*). Bei aktuellen Systemen bildet *PCI-Express* die Schnittstelle zwischen GPGPU und Host-System[3]. Die GPGPU ist hierarchisch aufgebaut. Dabei bilden die *Texture/Processor Cluster* (TPC) die oberste Hier-

[1] http://www.opengl.org/

[2] http://msdn.microsoft.com/en-us/library/windows/apps/bg182880.aspx

[3] Mit *host* ist im weiteren Verlauf der Arbeit das CPU-System gemeint, also der Teil des Computers, der die Berechnungen auf der CPU durchführt

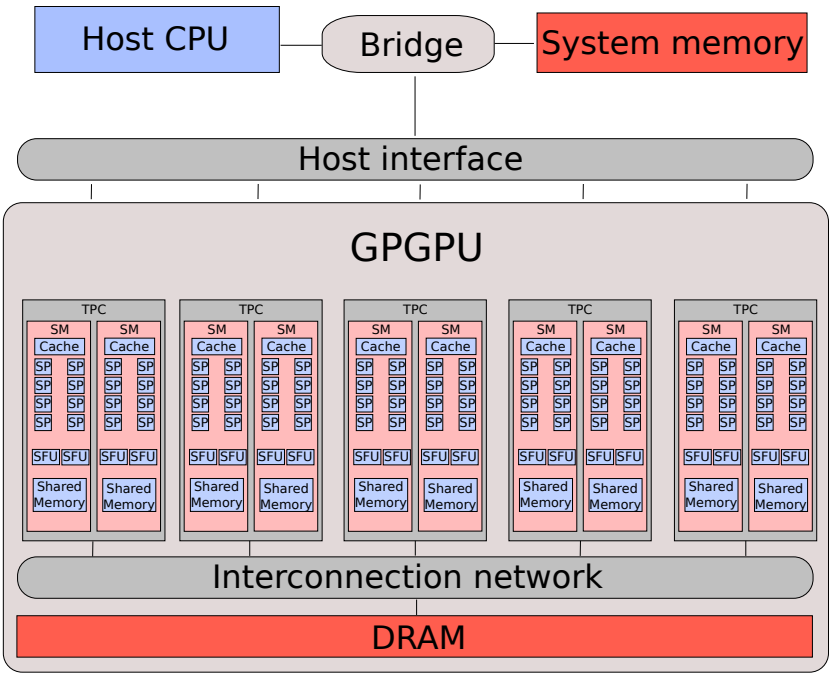


Abbildung 2.3: GPGPU Architektur

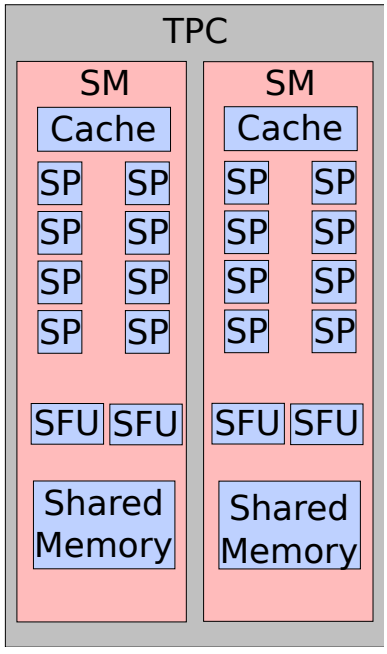


Abbildung 2.4: Streaming Multi-Processor

archie. Die Verbindung und der Datenaustausch zwischen den einzelnen TPC findet über den vergleichsweise langsamen gemeinsamen Hauptspeicher (*Dynamic Random Access Memory* (DRAM)) der GPGPU statt. Ein TPC besteht bei dieser Architektur aus zwei *Streaming Multi-Processors* (SMs). Die einzelnen SMs wiederum bestehen aus mehreren *Streaming Processors* (SPs). Die SPs bilden die atomaren Berechnungseinheiten bei GPGPUs, denen ein Thread zugeordnet wird und auf dem die jeweiligen Berechnungen durchgeführt werden. Jeder SM besitzt eine *special function unit* (SFU) auf der transzendente Funktionen berechnet werden, also Funktionen, die nicht algebraisch sind, wie trigonometrische Funktionen oder die Exponentialfunktion. In Abbildung 2.4 sind die inneren Module nochmal vergrößert dargestellt.

Die Threads werden in sogenannten *Warps* zusammengefasst. Üblicherweise besteht ein Warp aus 32 Threads und ein SM kann mehrere Warps verwalten. Bei der *Tesla* Architektur sind es maximal 32 aktive Warps, während bei der neueren *Fermi* Architektur 48 und bei der *Kepler* Architektur 64 aktive Warps unterstützt werden [31]. Aktive Warps sind Warps, die von dem SM abgearbeitet werden. Dabei wählt der SM dynamisch jeweils den Warp, bei dem die zu berechnenden Daten im Speicher zur Verfügung stehen. Dies dient dazu mögliche Speicherzugriffe zu kaschieren. Ein wichtiger Aspekt bei GPGPUs ist der *context-switch*, der im Vergleich zu einem context-switch auf der CPU mit sehr geringem Overhead durchgeführt werden kann. Ein context-switch beschreibt dabei das Wechseln der Threads, die aktuell ausgeführt werden, sollten die Daten für die Ausführung noch nicht vorhanden sein. Eine SM wechselt dabei den gerade ausgeführten Warp mit einem anderen aus dem Pool der aktiven Warps aus. Dementsprechend ist es essentiell, ein Programm mit vielen Warps auszuführen, um Latenzen zu vermeiden.

Definition 1 (Speedup) *Der Speedup ist in dieser Arbeit definiert als der Geschwindigkeitsgewinn der Ausführung eines parallelen Programms (T_{par}) im Vergleich zur sequentiellen Ausführung des gleichen Programms (T_{seq}). Er wird formal berechnet aus Speedup = $\frac{T_{seq}}{T_{par}}$. Dabei wird in dieser Arbeit immer die Ausführungszeit des kompletten Programms betrachtet und nicht nur die zu parallelisierenden Teilabschnitte.*

Die enorme parallele Rechenleistung von GPGPUs implizieren einen enormen Leistungsschub im Vergleich zu herkömmlichen CPUs. Untersuchungen haben jedoch ergeben, dass der erreichte Speedup (s. Def. 1) von vielen Faktoren abhängig ist und nicht immer ein großer Speedup erreicht wird [25]. In der Literatur finden sich Speedup Angaben von 3 bis 200 [32]. Die Firma Intel hat in einer eigenen Studie sogar gezeigt, dass in vielen Problemen eine Multi-Core CPU einer GPGPU überlegen ist [25]. Tatsächlich ist der Speedup hochgradig vom Problem und der Implementierung abhängig. Insbesondere die Ausnutzung der schnellen Speicher auf der GPGPU und das korrekte Ansprechen selbiger ist essentiell für eine schnelle Ausführung. Dies kann jedoch zu Problemen mit der Größe des vorhandenen Speichers führen. Während der langsame DRAM meist ausreicht um selbst größere Probleme zu berechnen, ist der schnellere Speicher (*Shared-Memory*) begrenzt und für jeden Thread-Block stehen meist nur mehrere tausend Bytes zur Verfügung. Auf die Thread-Hierarchien (Thread-Blöcke) wird in Abschn. 2.1.2.1 eingegangen. Der begrenzte Speicher stellt die Entwickler vor große Herausforderungen. Es muss darauf geachtet werden, dass ausreichend Shared-Memory vorhanden ist, bzw. ob das Problem aufgeteilt werden und nacheinander berechnet werden muss, was jedoch nicht bei allen Problemen möglich ist.

GPGPUs sind für bestimmte Problemstellungen besonders gut geeignet, jedoch muss auf viele Kriterien geachtet werden, um das volle Potential ausnutzen zu können. Insbesondere die Synchronisierung zwischen den einzelnen Threads und Thread-Blöcken ist nicht trivial und bildet oft ein Bottleneck [33], [34] bei der Berechnung. Des Weiteren ist die korrekte Verwendung der komplexen Speicherarchitektur der GPGPU für die performante Ausführung von Programmcode essentiell und kann die Art der Implementierung des Algorithmus beeinflussen.

2.1.2.1 CUDA

NVIDIA bietet mit CUDA [35] [36] eine Programmierschnittstelle für die Programmierung auf NVIDIA GPGPUs. Dabei bildet CUDA eine Obermenge der

Programmiersprache C++ mit zusätzlichen Direktiven, die für die Allozierung der Daten auf die GPGPU zuständig sind und die Ausführung der Programmabschnitte auf selbiger. Bereiche des Quellcodes, die mit Hilfe von CUDA auf der Grafikkarte ausgeführt werden sollen, werden als *Kernel* bezeichnet. Diese sind ähnlich wie C/C++ Funktionen aufgebaut, außer dass ein Kernel zur Laufzeit mit multiplen Daten parallel ausgeführt wird. Auf die jeweiligen Daten wird dabei üblicherweise mit Hilfe von Indizes zugegriffen, die über die jeweiligen Threads definiert werden.

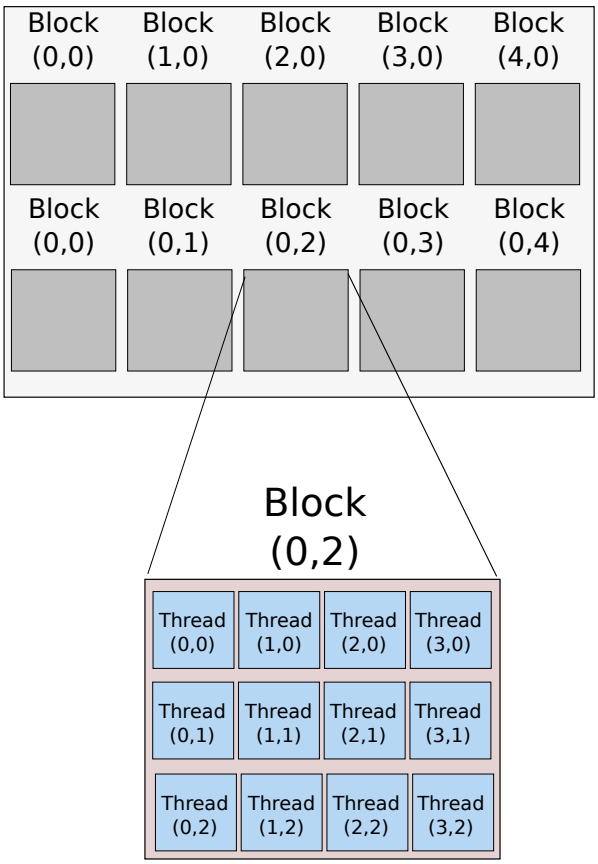


Abbildung 2.5: Block und Thread Hierarchie bei CUDA

Die Thread-Hierarchie bei CUDA stellt einen essentiellen Bestandteil bei der Programmierung auf GPGPUs dar. Die Threads sind in hierarchischen Strukturen (s. Abb. 2.5) angeordnet. Die unterste Hierarchieebene bildet der *Thread-Block*, in dem die einzelnen Threads über einen gemeinsamen Speicher

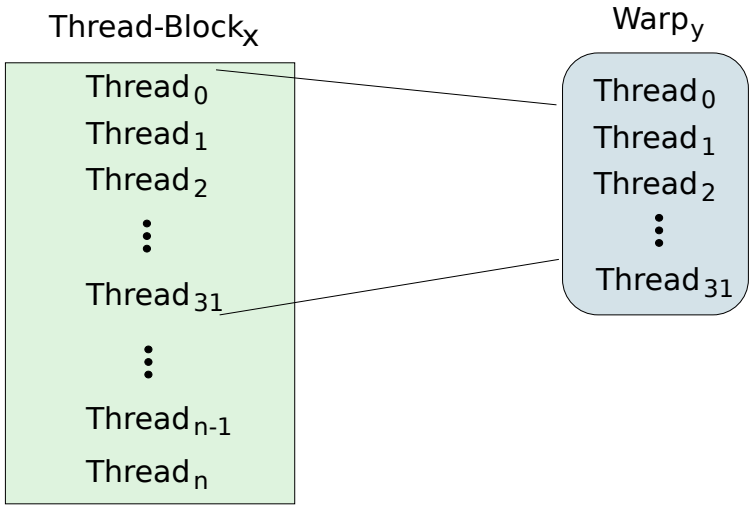


Abbildung 2.6: Thread-Block / Warp Zusammenhang

Informationen austauschen können. Außerdem ist es möglich, die Threads in dieser Ebene untereinander zu synchronisieren, was in den höheren Ebenen nicht möglich ist. Die Threads in einem Block werden für die Ausführung in *Warps* zusammengefasst. Auf aktuellen Grafikkarten bilden 32 Threads üblicherweise einen Warp. Die Threads in diesem Warp führen alle die gleichen Operationen auf einem SM (s. Abschn. 2.1.2) aus. In Abb. 2.6 ist der Zusammenhang zwischen einem Thread-Block und einem Warp abgebildet. In dem Thread-Block sind n Threads angeordnet. Aus diesen n Threads werden 32 Threads ausgewählt, die zusammen einen Warp bilden. Dabei ist zu beachten, dass Threads in einem Warp aus dem gleichen Thread-Block ausgewählt werden. Diese Auswahl wird von CUDA automatisch durchgeführt und kann vom Entwickler nicht gesteuert werden.

Ein großes Problem auf GPGPUs stellen divergente Verzweigungen dar. Divergente Verzweigungen sind Abschnitte im Quellcode, bei denen die Threads unterschiedliche Ausführungspfade nehmen können (z.B. IF-Verzweigungen). In diesem Fall muss auf die divergierenden Threads gewartet werden. Dies ist ein Problem, da ein SM für die Ausführung eines Warps zuständig ist und die nicht benötigten Ressourcen in der Zwischenzeit von keinem anderen Thread verwendet werden können. Jeder der Threads in einem Block hat eine eindeutige ID, über die er angesprochen werden kann. Abb. 2.5 stellt eine

2-dimensionale Struktur dar, wo jeder Thread eine zweidimensionale Koordinate besitzt. Auch 1- und 3-dimensionale Strukturen sind üblich und von dem Problem, welches gelöst werden soll, abhängig. Beispielsweise eignen sich 2-dimensionale Strukturen besonders gut für die Berechnung von Matrizen, da jedem Thread ein Eintrag in der Matrix zugeordnet werden kann.

```
1 int a[100];
2
3 for(int i = 0; i < 100; i++){
4   a[i] = i*i;
5 }
```

Listing 2.1: C++ Loop

```
1 __global__ void loop(int *a){
2   int id = threadIdx.x;
3   if(id < 100){
4   a[id] = id*id;
5   }
6 }
```

Listing 2.2: CUDA Loop

In den Linstings 2.1 und 2.2 ist eine Transformation einer einfachen C++ Schleife in einen CUDA-Kernel dargestellt. Dort sieht man, dass auf den einzelnen Indizes des Arrays (*a) über die ID des Threads zugegriffen wird. Um zu vermeiden, dass ein Thread mit einer größeren ID als der Größe des Arrays drauf zugreift, muss wie im Beispiel mit einer Abfrage die ID ermittelt werden. Eine andere Möglichkeit besteht darin, dafür zu sorgen, dass die Anzahl an Threads durch die Array-Größe beschränkt wird.

Die einzelnen Thread-Blöcke sind in einem *Block-Grid* angeordnet, wobei jeder Block, ähnlich der Threads, durch eine ID gekennzeichnet ist. Die Threads zwischen verschiedenen Blöcken können Informationen nur über den langsamen globalen Speicher der GPGPU austauschen und nicht untereinander synchronisiert werden. Dies führt bei der Implementierung von Algorithmen oft zu Problemen, die durch eine Anpassung der Algorithmen gelöst werden müssen.

2.2 Populationsbasierte Algorithmen

Dieser Abschnitt beschäftigt sich mit populationsbasierten Algorithmen, der Beispielproblemklasse für das in dieser Arbeit vorgestellte *CASEP* Framework. *PBAs* sind naturanaloge Heuristiken [37]. Verwendung finden sie vorwiegend in Problemstellungen, in denen eine analytische Lösungsfindung nicht möglich

oder zu zeitaufwendig ist. Heuristische Verfahren beruhen auf Erfahrungswerten, mit denen versucht wird, eine Lösung für das gesuchte Problem zu finden. *PBAs* versuchen dazu, diese Erfahrungswerte iterativ zu erhöhen und somit bei jeder Iteration der Lösung näher zu kommen. Üblicherweise wird hierzu zu Beginn des Algorithmus der Suchraum möglichst großflächig abgesucht und potentiell bessere Suchbereiche (Bereiche mit besseren Lösungen) in späteren Iterationen genauer untersucht. Dies entspricht dem allgemeinen Exploration-Exploitation Problem [38].

Definition 2 (Agenten) *Agenten (A) sind im Zusammenhang mit PBAs als die atomaren Berechnungseinheiten innerhalb des Algorithmus definiert. Jeder Agent führt die definierten PBA-Operationen iterativ auf seinen privaten Daten aus. Je nach PBA können Agenten auch andere Bezeichner erhalten, wie z.B. Individuen bei Genetischen Algorithmen oder Partikel bei Particle Swarm Optimization.*

Definition 3 (Population) *Die Population (Pop) eines PBAs ist definiert als die Summe aller Agenten ($Pop = \sum_i A_i$). Die Agenten einer Population können je nach PBA-Implementierung private Daten untereinander austauschen.*

Zum Durchsuchen des Suchraums werden sogenannte *Agenten* (s. Def. 2) verwendet, die in ihrer Gesamtheit die *Population* (s. Def.2) der *PBAs* bilden. Diese Agenten führen jeweils die gleichen Operationen aus, um eine (möglichst gute) Lösung (s. Def. 4) für das gestellte Problem zu finden. Die iterativ durchgeführten Operationen dienen dazu, die Lösungen der einzelnen Agenten zu verbessern[4]. Die so gefundenen Lösungen werden anhand ihrer Güte bewertet; diese Güte wird bei *PBAs* als *Fitness* (s. Def. 5) bezeichnet. Die zur Verbesserung der Fitness verwendeten Operationen sind in zwei Kategorien eingeteilt.

[4]Im Abschnitt 2.2.2 und 2.2.1 werden die Operationen von zwei häufig verwendeten *PBAs* genauer erläutert.

Zum einen werden lokale Operationen durchgeführt, d.h. Operationen, die unabhängig von Lösungen anderer Agenten durchgeführt werden können. Zum anderen werden Lösungen von anderen Agenten dazu verwendet, die eigene Lösung zu verbessern. Während die erste Kategorie ohne Kommunikationskosten auskommt, stellt die zweite Kategorie aufgrund der Kommunikation mit anderen Agenten eine Herausforderung speziell bei der Parallelisierung dar. Insbesondere auf GPGPUs stellt Kommunikation ein großes Problem dar, da nur zwischen einzelnen Threads in Thread-Blöcken synchronisiert werden kann und der Datenaustausch entweder über den sehr begrenzten Shared-Memory oder den langsamen Global-Memory durchgeführt werden kann (s. Abschn. 2.1.2). Auf die konkreten Probleme und Lösungswege wird in späteren Kapiteln eingegangen.

Definition 4 (Lösung) *Die Lösung eines Agenten ist definiert als ein Parametervektor zur Lösung des definierten Problems. Beispielhaft sei dies an einer Optimierungsfunktion aufgezeigt: Die reellwertige Funktion* $f(\vec{x}) = \sum_i x_i^2$ *sei das zu lösende Problem. Dabei soll die Funktion einen möglichst kleinen Wert annehmen. Der Parametervektor des Agenten ist der Vektor* $\vec{x}$*. Eine Lösung ist eine Belegung des Vektors* $\vec{x}$ *mit reellen Zahlen.*

Definition 5 (Fitness) *Die Fitness einer Lösung ist definiert als die Güte der Lösung bezogen auf die optimale Lösung des Problems. Beim Beispiel der Optimierungsfunktionen ist die optimale Lösung das zu findende Minimum oder Maximum (je nach Optimierungsdefinition). Je näher die Lösung des Agenten an dem Optimum liegt, desto höher (besser) ist die Fitness zu bewerten.*

Im Folgenden werden zwei der prominentesten Vertreter der *PBAs* (*Genetische Algorithmen* und *Particle Swarm Optimization*) ausführlich behandelt. Obwohl die vorliegende Arbeit auf die Klasse der *PBAs* ausgelegt ist, sind die

meisten Beispiele und Implementierungen auf diese beiden Arten von *PBAs* beschränkt. Dies hat größtenteils mit dem Verbreitungsrad der erwähnten Algorithmen zu tun. Außerdem decken die verwendeten Algorithmen die meisten Charakteristika von *PBAs* ab, da Unterklassen von *PBAs* meistens eine Permutation typischer *PBA* Funktionen abbilden und keine grundlegend neue Techniken einführen [39]. In dieser Arbeit werden in den jeweiligen Abschnitten die generischen Eigenschaften des Frameworks hervorgehoben und es wird auf die für bestimmte *PBAs* spezifische Eigenschaften eingegangen. Außerdem wird darauf eingegangen, welche Modifikationen gegebenenfalls notwendig sind, um einzelne spezifische Eigenschaften für andere *PBAs* zu generalisieren.

2.2.1 Genetische Algorithmen

Genetische Algorithmen (GA) [15] sind *PBAs*, die ihre Analogie aus dem Bereich der Genetik und dort im Speziellen aus dem Bereich der *Rekombination* einzelner Chromosomen haben. Entwickelt wurden sie 1988 von Holland et. al. [16]. Die Lösungen der einzelnen Agenten werden beim *GA Chromosomen* genannt, dies kann z.B. ein Lösungsvektor für eine Funktion oder eine Liste von Punkten bei einem Wegfindungsproblem sein. Jeder Wert in diesem Lösungsvektor wird als *Gen* bezeichnet. Jeder Agent in der Population des *PBAs* hat hierbei seine eigene lokale Lösung des Problems, die zu Beginn des Algorithmus meist zufällig erzeugt wird.

Die anfänglich erzeugten Lösungen der einzelnen Agenten werden iterativ mit Hilfe von vier charakteristischen Operationen verbessert: *Rekombination*, *Mutation*, *Fitnessberechnung* und *Selektion*. Der Grundaufbau eines *GAs* wird in Abb. 2.7 dargestellt. Zu Beginn des Algorithmus werden zufällige Lösungen für alle Agenten erzeugt. Als Nächstes folgt die *Rekombination*, die neue Lösungen (Individuen) durch die Kombination von Lösungen zweier Eltern-Individuen erzeugt. In der nächsten Phase werden einzelne Abschnitte der Lösungen zufällig verändert (mutiert). Dies ist notwendig, da sonst durch die *Rekombination* nur Lösungen entstehen würden, die ähnlich der Lösungen der Eltern wären, also keine neuen Gebiete im Lösungsraum durchsucht werden würden. Von den so neu erzeugten Individuen wird im nächsten Schritt die

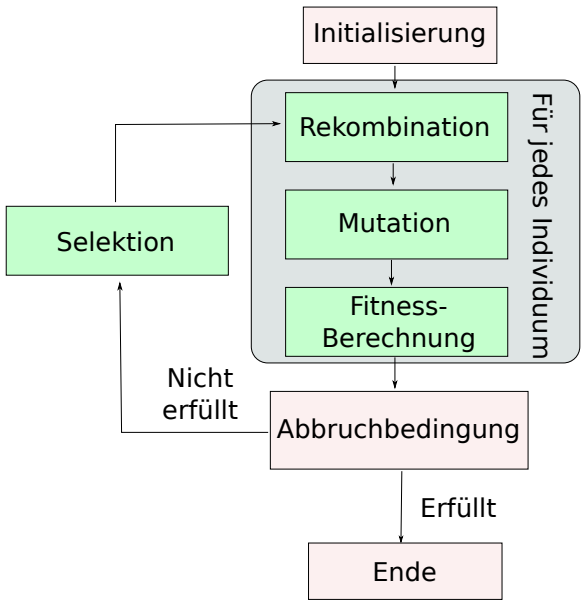


Abbildung 2.7: Schematische Darstellung eines GA

Güte (Fitness) ihrer Lösung berechnet. Sollte nach der Iteration die Abbruchbedingung erfüllt sein, wird der Algorithmus beendet. Dies kann entweder nach einer festgelegten Anzahl an Iterationen sein, nach einer festgelegten Zeit oder wenn das Ziel erreicht ist. Sollte die Abbruchbedingung nicht erfüllt sein, wird eine weitere Iteration durchgeführt. Die beschriebenen Operationen können vielfältig gestaltet sein und werden im Folgenden genauer beschrieben. Insbesondere wird auf die häufigsten Vertreter der Operationen eingegangen.

2.2.1.1 Rekombination

Je nach Problemstellung werden verschiedene Arten der *Rekombination* (*Crossover*) bei *GAs* bevorzugt. Eine ausführliche Beschreibung verschiedener Methoden ist in [40] zu finden. Die einfachste Form der Rekombonation ist die sogenannte *One-Point-Crossover* Methode wie in Abb. 2.8 dargestellt. Hierbei wird ein zufälliger Punkt bei den beiden Eltern gewählt. Das erste Individuum, dass erstellt wird, besteht aus dem ersten Abschnitt des ersten Elternteils und aus dem zweiten Abschnitt des zweiten Elternteils, beim zweiten Individuum entsprechend umgekehrt. Um eine größere Verteilung der einzelnen Abschnit-

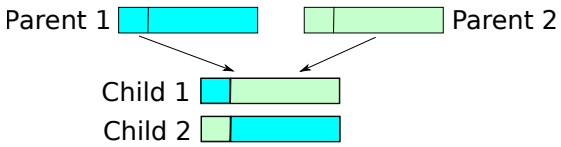


Abbildung 2.8: One-Point-Crossover

te zu erreichen, kann die Teilung an mehreren Stellen durchgeführt werden (s. Abb. 2.9). Diese Form wird als *Two-Point-Crossover* bezeichnet und ist

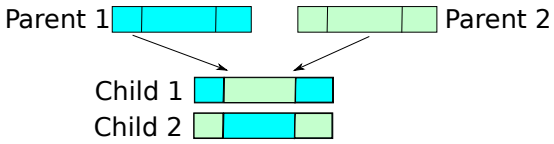


Abbildung 2.9: Two-Point-Crossover

gemeinsam mit dem sogenannten *Uniform-Crossover* (s. Abb. 2.10) die am häufigsten verwendete Form der *Rekombination*. Hierbei wird für jede Position (s. Def. 6) der Lösung mit einer Wahrscheinlichkeit (üblicherweise 50%) entweder die Position vom ersten oder die vom zweiten Elternteil verwendet.

Definition 6 (Position) *Mit Position einer Lösung wird der Wert im Lösungsvektor des Agenten an einer bestimmten Stelle bezeichnet. Im Beispiel der Funktionsoptimierung wäre bei einem zweidimensionalen Vektor* (x_0, x_1) *die erste Position* x_0.

Die zu verwendende Methode hängt hauptsächlich von der Größe der Population ab [40]. *Uniform-Crossover* kann 2^n (bei einer Problemgröße n) unterschiedliche neue Individuen erzeugen. Für jede Position der Lösung wird entweder die Position des ersten Elternteils oder die des zweiten Elternteils gewählt. Im Vergleich dazu können aus zwei Elternteilen beim *One-Point-Crossover* vier unterschiedliche Individuen entstehen und beim *Two-Point-Crossover* neun unterschiedliche Individuen. Somit lässt sich mit *Uniform-Crossover* der Suchraum besser durchsuchen, da vergleichsweise viele unterschiedliche Lösungen

im Suchraum entstehen können. Insbesondere ist dies bei kleinen Populationen vom Vorteil, bei denen mit wenigen Individuen ein großer Bereich im Suchraum durchsucht werden kann. Bei größeren Populationen übernehmen die Exploration des Suchraums die große Anzahl an Individuen, die durch die zufällige Anfangsverteilung bereits einen größeren Bereich des Suchraums abdecken. Somit eignen sich für größere Populationen eher Methoden wie *One-* oder *Two-Point-Crossover*, zumal *Uniform-Crossover* durch die ausgedehnte Verteilung der Lösungen im Suchraum langsamer zu einem bestimmten Punkt (Lösung) konvergiert.

Definition 7 (Suchraum) *Der Suchraum ist definiert als die Summe aller möglichen Lösungen für ein bestimmtes Problem.*

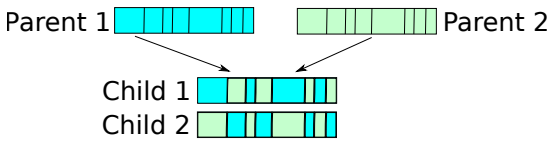


Abbildung 2.10: Uniform-Crossover

2.2.1.2 Mutation

Der Mutationsoperator dient in erster Linie dazu, neue Lösungen zu generieren, die nicht mit Hilfe von *Crossover* erreicht werden können. Angenommen die beste Lösung eines Problems hat an einer Position n den Wert 0 und es existiert kein Individuum, das an dieser Position den Wert 0 besitzt. Dann ist es für den Algorithmus nicht möglich, die optimale Lösung nur durch *Crossover* zu erreichen. Um dieses Problem zu umgehen, wird an zufälligen Positionen im Lösungsvektor mit einer festgelegten Wahrscheinlichkeit der Wert modifiziert. Die Modifikation hängt hierbei von der Art der Repräsentation des Vektors ab. Bei reellwertigen Zahlen beispielsweise wird eine reelwertige Zahl hinzu addiert bzw. subtrahiert. Die Wahrscheinlichkeit für eine Mutation wird üblicherweise bei einem Wert von ungefähr 1.0%-10.0% gewählt [41] [42]. Eine zu hohe

Wahrscheinlichkeit führt zur Divergenz des Algorithmus, während bei einer zu niedrigen Wahrscheinlichkeit zu wenig neue Gebiete im Suchraum durchsucht werden.

2.2.1.3 Fitness Berechnung

Das eigentliche Problem, welches durch den *GA* berechnet werden soll, bildet gleichzeitig die Fitnessfunktion für den Algorithmus. Am Beispiel der Funktionsoptimierung bildet die Funktion, die optimiert werden soll, die Fitnessfunktion. Die Lösung des Individuums ist dementsprechend der Fitnesswert. Je näher die jeweilige Lösung am Optimum ist, desto höher ist somit auch die Fitness des Individuum.

Bei bestimmten Problemen (z.B. Funktionsoptimierung) können sehr große Fitnesswerte entstehen, was hauptsächlich ein Problem für die Lesbarkeit der Ausgabe ist. Um dieses Problem zu umgehen, werden üblicherweise die Fitnesswerte auf einer Skala zwischen 0,0 und 1,0 normiert, wobei 1,0 die beste Lösung und 0,0 die schlechteste darstellt. Bei Funktionsoptimierungen, bei denen das Ziel ist das Funktionsminimum zu ermitteln, hat die Funktionsbelegung mit dem kleinsten Wert die größte Fitness. In diesem Fall wird die Funktion normiert, indem man die beste Lösung (größte Fitness) f_{max} als Basiswert verwendet und die normierte Lösung aus $f_{normiert} = |\frac{f_{max}}{f_i}|$ berechnet. Dabei ist der Wert f_i die i-te Lösung der Population. Sollte die beste Lösung den größten Wert haben (z.B. bei der Ermittlung des Funktionsmaximums), müssen die Quotienten dementsprechend vertauscht werden ($f_{normiert} = |\frac{f_i}{f_{max}}|$)

2.2.1.4 Selektion

Durch *Crossover* und gegebenenfalls Mutation entstehen neue Individuen. Um ein stetiges Wachstum der Population (*Pop*) durch die *Rekombination* zu vermeiden, wird für jede Iteration x die Population P_x selektiert, auf die *GA* Operationen erneut durchgeführt werden sollen. In der Literatur finden sich viele unterschiedliche Methoden, die diese Selektion durchführen können. Die Ausführungen in dieser Arbeit beschränken sich auf zwei der am häufigsten verwendeten Methoden, *Roulette-Wheel-Selection* und *Tournament-Selection*

[43].

Bei der *Roulette-Wheel-Selection* werden die einzelnen Lösungen anhand ihrer Fitness auf ein virtuelles Roulette abgebildet. Dabei werden die einzelnen Lösungen einem Abschnitt abhängig der Güte (Fitness) f ihrer Lösung zugeordnet, d.h. je besser die Lösung, desto größer der zugehörige Abschnitt, wie in Abb. 2.11 beispielhaft zu sehen ist. Hier wird der Lösung j ein kleinerer

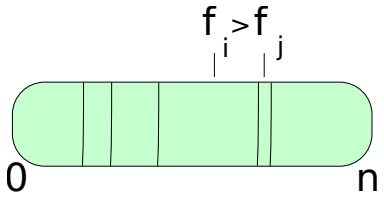


Abbildung 2.11: Roulette-Wheel-Selection

Abschnitt zugeordnet als Lösung i, da die Fitness von Lösung i größer ist. Als nächstes wird eine Zufallszahl zwischen 0 und n generiert und das Individuum mit der Lösung gewählt, zu der der Abschnitt gehört, in dem die Zufallszahl liegt. Dementsprechend werden Individuen bevorzugt, die eine bessere Fitness, d.h. einen größeren Abschnitt besitzen.

Die *Tournament-Selection* bevorzugt im Grundaufbau ebenso Lösungen mit einer besseren Fitness, dies kann jedoch mit Hilfe von Parametern angepasst werden. Zu Beginn des Verfahrens wird zufällig eine Menge an Individuen aus der Population gewählt. Diese Operation ist unabhängig von der Fitness der einzelnen Individuen. Als nächstes wird das Individuum mit der größten Fitness ausgewählt, welches in die neue Population aufgenommen wird. Unterschiedliche Implementierungen verwenden nicht nur das Individuum mit der besten Fitness für die Wahl, sondern die n besten Individuen. Je nach Größe der Auswahl kann die Wahrscheinlichkeit, mit der Individuen mit einer schlechteren Fitness ausgewählt werden, variiert werden. Je kleiner die Auswahl, desto wahrscheinlicher ist es, dass auch Individuen mit einer kleinen Fitness ausgewählt werden.

Beide Methoden eigenen sich im Grundaufbau für eine Parallelisierung, da die Operationen für die Wahl unabhängig für jedes neue Individuum durchgeführt werden können und keine Kommunikation zwischen den einzelnen In-

dividuen notwendig ist. Die Wahl hängt somit von der Art des zu lösenden Problems ab. Auch kann die Problemgröße ein Faktor bei der Wahl sein und muss für jedes Problem separat untersucht werden, eine generische Aussage ist nicht möglich [43].

2.2.2 Particle Swarm Optimization

Der 1995 von Kennedy und Eberhart [17] vorgestellte *Particle Swarm Optimization* (PSO) Algorithmus bedient sich der Analogie von Vogel- oder Fischschwärmen, die bei der Futtersuche bestimmten Mustern folgen. Den Grundgedanken bildet dabei die Ausrichtung von weniger guten (geringer Fitnesswert; s. Abschn. 2.2.1.3) Individuen (beim *PSO Partikel* genannt) an die Position von besseren Nachbarpartikeln bzw. am global besten Partikel. Dabei bildet jedes Partikel gleichzeitig eine Lösung für das zu untersuchende Problem, basierend auf seiner Position im Suchraum. Durch die Ausrichtung entsteht eine Schwarmbewegung im Suchraum von den schlechteren Positionen weg hin zu den eher besseren Positionen. Für diese Bewegung besitzt jedes Partikel einen Positions- und einen Geschwindigkeitsvektor. Der Grundalgorithmus ist in Abb. 2.12 dargestellt. Zuerst werden die Partikel zufällig im Suchraum verteilt. Danach bestimmt jedes Partikel seine persönliche Fitness, also die Güte seiner Lösung. Nun wird die beste persönliche Lösung des Partikels (pbest) bestimmt, d.h. die beste je gefundene Lösung. Im nächsten Schritt wird das Partikel aus der Nachbarschaft (s. Def. 8) mit der besten Lösung in dieser Iteration bestimmt. Diese Informationen werden dazu genutzt, den Geschwindigkeits- und Positionsvektor des Partikels anzupassen. Die einzelnen Operationen werden im Folgenden im Detail beschrieben. Auf eine Beschreibung der Fitnessberechnung wird verzichtet, da die Prozedur analog der des *GAs* ist. Die Beschreibung richtet sich nach [44].

Definition 8 (Nachbarschaft) *Die Nachbarschaft beim PSO ist definiert über die absolute Differenz (Entfernung E) des Lösungsvektors zweier Partikel ($E = |\vec{x_i} - \vec{x_j}|$). Partikel, die eine festgelegte Entfernung zueinander nicht*

überschreiten, bilden eine Nachbarschaft. Am Beispiel der Funktionsoptimierung seien $\vec{x_1} = \{2, 3\}$ *und* $\vec{x_2} = \{1, 4\}$ *die Lösungsvektoren zweier Partikel. Die absolute Differenz der Partikel beträgt zwei. Somit sind die beiden Partikel in der gleichen Nachbarschaft, wenn die Nachbarschaft kleiner/gleich zwei definiert wurde.*

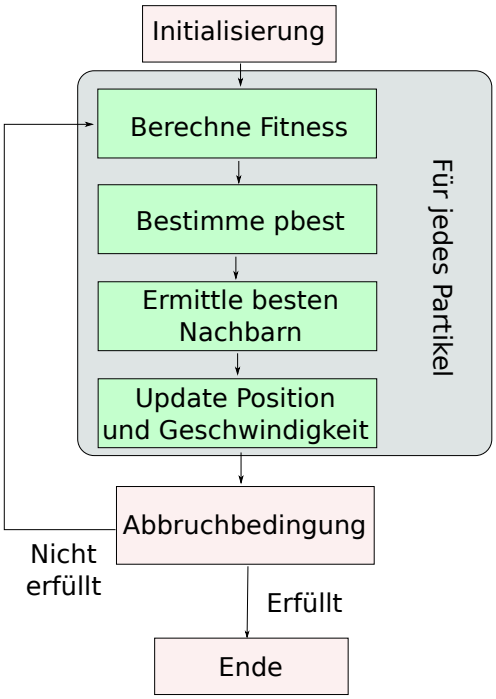


Abbildung 2.12: Particle Swarm Optimization Algorithmus

2.2.2.1 Bestimme *pbest*

Nachdem für jedes Partikel die Fitness berechnet wurde, wird die aktuelle Lösung mit der besten je gefunden Lösung (*pbest*) vom Partikel verglichen. Ist die neue Lösung besser als *pbest*, wird *pbest* entsprechend aktualisiert. Damit wird erreicht, dass sich das Partikel eher zu besseren Lösungen hinbewegt, als zu schlechteren und dient der allgemeinen Konvergenz des Algorithmus.

2.2.2.2 Ermittle besten Nachbarn

Als nächstes bestimmt jedes Partikel im Schwarm die beste Fitness aus seiner persönlichen Nachbarschaft. Das beste Partikel in der Nachbarschaft bildet den lokalen Attraktor und ist im Allgemeinen für jedes Partikel verschieden. Die Größe der Nachbarschaft ist ein wichtiger Parameter für einen *PSO* Algorithmus und muss je nach Aufgabenstellung gegebenenfalls angepasst werden. Je größer die Nachbarschaft ist, desto unwahrscheinlicher ist es, dass schlechtere Partikel als lokale Attraktoren verwendet werden, was zu einer schnelleren Konvergenz des Algorithmus führt. Viele *PSO* Implementierungen verwenden hierzu das global beste Partikel, die Nachbarschaft hat somit die Größe der Population [44].

2.2.2.3 Update von Position und Geschwindigkeit

Die Informationen über *pbest* und den besten Nachbarn werden verwendet, um den Geschwindigkeitsvektor anzupassen. Dieser wird im nächsten Schritt dazu verwendet, die Position des Partikels anzupassen. In Abb. 2.13 ist exemplarisch ein Partikel ($Partikel_i$) mit der Position x_i und einem Geschwindigkeitsvektor v_i in einem zweidimensionalen Raum dargestellt. Je nach Problemstellung kann der Raum über n Dimensionen aufgespannt werden. Die Vektoren haben dabei immer die selbe Dimensionszahl, die sich an der Dimension des aufgespannten Raums richtet (im Beispiel sind es zwei Dimensionen). In der Grundfassung des *PSO* Algorithmus findet die Anpassung des Geschwindigkeitsvektors v_i über die in Formel 2.1 dargestellte Weise statt.

$$\vec{v_i} \leftarrow \vec{v_i} + \vec{U}(0, \phi_1) \otimes (\vec{p_i} - \vec{x_i}) + \vec{U}(0, \phi_2) \otimes (\vec{p_g} - \vec{x_i}) \tag{2.1}$$

Wobei $\vec{U}(0, \phi_1)$ einen Vektor von Zufallszahlen zwischen 0 und $\vec{\phi_i}$ repräsentiert, der in jeder Iteration neu erstellt wird und $\otimes$ die komponentenweise Multiplikation. $\vec{p_i}$ ist die beste je gefundene Lösung des Partikels, $\vec{x_i}$ die aktuelle Position des Partikels und $\vec{p_g}$ die Position des besten Nachbarn. Dementsprechend werden zu dem aktuellen Geschwindigkeitsvektor zwei Vektoren addiert. Ein Vektor bestehend aus der Differenz zwischen der besten lokalen Position

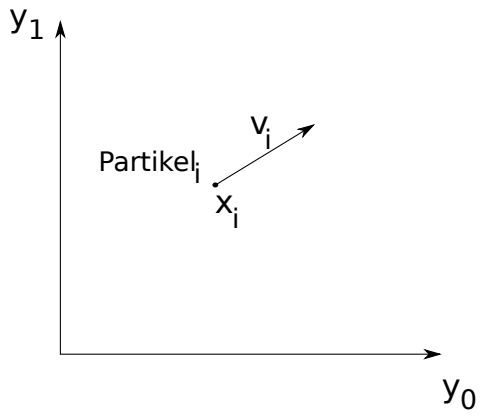


Abbildung 2.13: Positions- und Richtungsvektor eines Partikels

(pbest) und aktuellen Position und ein Vektor aus der Differenz zwischen der Position des besten Partikels aus der Nachbarschaft und der aktuellen eigenen Position. Zu diesen beiden Vektoren werden zwei Zufallsvektoren hinzu addiert, mit denen sich die Konvergenz und das Suchverhalten des *PSO* Algorithmus steuern lassen. Die neue Position des Partikels wird bestimmt, indem auf die aktuelle Position der neue Geschwindigkeitsvektor $\vec{v_i}$ addiert wird.

2.3 Parallelisierungsmethoden popualtionsbasierter Algorithmen

Der Grundaufbau von *PBAs* besteht aus einer Anzahl n von Optimierungsfunktionen, die von m Individuen iterativ durchgeführt werden, bis das gewünschte Ergebnis erreicht ist oder eine Abbruchbedingung erreicht wurde. Dabei führt jedes Individuum die gleichen Operationen auf jeweils verschiedenen Daten aus, was dem klassischen *Single Instruction Multiple Data* (SIMD) [45] Paradigma entspricht. Diese Art von Problemen eignet sich traditionell besonders gut für die Parallelisierung. Insbesondere auf GPGPUs, die ursprünglich dazu entwickelt wurden, einzelne Pixel parallel zu berechnen, indem die gleichen Operationen für jeweils unterschiedliche Daten ausgeführt werden, lassen sich solche Probleme gut abbilden. Bei GPGPUs spricht man auch von *Single Instruction Multiple Threads* (SIMT). Anders als bei SIMD, können

bei einer SIMT Architektur einzelne Threads auch einen anderen Datenfluss nehmen (z.B. bei *if-Verzweigungen*). Dies ist für *PBAs* besonders wichtig, da Operationen oft mit einer bestimmten Wahrscheinlichkeit ausgeführt werden. Somit werden einzelne Operationen nur von einer Teilmenge der Threads durchgeführt.

In der Literatur finden sich verschiedene Ansätze zur effizienten Parallelisierung von *PBAs*. In [46] von 1995 wird ein Überblick über frühe Parallelisierungsversuche von *GAs* gegeben. Die Autoren unterscheiden dabei grundsätzlich zwischen drei verschiedenen Ansätzen der Parallelisierung, 1) *globale Parallelisierung*, 2) *grobkörnige Parallelisierung* und 3) *feinkörnige Parallelisierung* (vgl. Abschn. 2.1). Die globale Parallelisierung beruht auf dem Verwenden einer einzigen Population, die auf n Prozessoren aufgeteilt wird. Dabei wird die Kommunikation auf die Start- und die Endphase des Algorithmus minimiert, sodass kein signifikanter Overhead durch Kommunikation entsteht. Sollte es sich bei der Zielarchitektur um eine Architektur ohne gemeinsam genutzten Speicher (shared-memory) handeln, ist auch eine Master-Slave Aufteilung möglich. Dabei übernimmt ein Prozessor die Aufgaben des Masters und verteilt die Individuen auf die jeweiligen Slave-Knoten zur Berechnung der Fitness. Die Ergebnisse werden an den Master zurückgeschickt, welcher für die Erzeugung der neuen Population (*Mutation*, *Crossover*) zuständig ist. Der Master und die Kommunikation zwischen den Prozessoren und dem Master bilden bei diesem Ansatz einen Flaschenhals bei der Ausführung des Algorithmus. Die Überlegungen bei der globalen Parallelisierung führten zur grobkörnigen Parallelisierung, bei der Sub-Populationen in einer Topologie angeordnet werden, die die Kommunikation zwischen den einzelnen Populationen bestimmt. Die grundlegende Änderung zur globalen Parallelisierung ist dabei die Migration von einzelnen Individuen in eine andere benachbarte Sub-Population. Einzelne Sub-Populationen können nur mit anderen Sub-Populationen kommunizieren, wenn sie durch die gewählte Topologie miteinander verbunden sind. Die feinkörnige Parallelisierung eignet sich besonders gut für hochparallele Architekturen. Der optimale Fall wird dadurch gebildet, dass jede Berechnungseinheit für ein Individuum zuständig ist. Dadurch entsteht zwangsläufig ein großer Kommunikations-Overhead, was dazu führt, dass Architekturen für

eine feinkörnige Parallelisierung schnelle Kommunikation zwischen den einzelnen Berechnungsknoten voraussetzen.

PBA Parallelisierungsbeispiele: Die feinkörnige Parallelisierung eignet sich besonders gut für GPGPUs, auf denen jedes Individuum einem Berechnungsknoten auf der GPGPU zugeordnet werden kann. Die Kommunikation zwischen den einzelnen Individuen kann über einen sehr schnellen gemeinsamen Speicher erfolgen. Die Autoren in [47] nutzen diese Art der Parallelisierung zur Lösung von Optimierungsproblemen auf GPGPUs und erzielen je nach Komplexität der zu lösenden Funktion Speedups im Bereich von 1,4 bis 73,6. Feinkörnige Parallelisierung eignet sich grundsätzlich für jede Art von *PBA* und ist nicht auf *GAs* beschränkt. In [48] nutzen die Autoren feinkörnige Parallelisierung für einen parallelen *Ant Colony Optimization* (ACO) Algorithmus, einen Algorithmus, der auf der Naturanalogie der Ameisen beruht und häufig dazu verwendet wird, kürzeste Pfade in einem verbundenen Graphen zu finden. In [47] wird mit Hilfe dieser Methode ein *GA* auf GPGPUs parallelisiert, während die Autoren in [49] einen grobkörnigen Ansatz wählen, um einen *PSO* auf einer Multi-Prozessor Architektur zu parallelisieren. In der Literatur finden sich weitere Parallelisierungsversuche für *PBAs*, die jedoch alle eine der drei weiter oben beschriebenen Methoden verwenden. Für einen vollständigen Überblick wird an dieser Stelle auf [50], [51], [52] verwiesen, die zusammen mit den hier beschriebenen Arbeiten einen Überblick bieten.

2.4 Speedup-Berechnung auf parallelen Maschinen

Das maßgebliche Bewertungskriterium für den Erfolg einer Parallelisierung eines Programms ist der *erzielte Speedup* gegenüber einer seriellen Version. Einen ersten Anhaltspunkt über den zu erwartenden Speedup können hierbei Erfahrungswerte mit ähnlichen schon erprobten Programmkonstrukten bereitstellen. Diese Erfahrungswerte sind jedoch beschränkt, da sie insbesondere auf neuen Architekturen meist keine zuverlässigen Werte liefern. Aus diesem

Grund wurden verschiedene Methodiken entwickelt, die den Entwicklern auf andere Weise Anhaltspunkte über den zu erwartenden Speedup liefern.

Synthetische Benchmarks: Einer der ersten entwickelten synthetischen Benchmarks bildet die 1973 in *Algol 60* geschriebene Benchmark-Suite [53] zur Messung der Performance auf unterschiedlichen Maschinen. Synthetische Benchmarks sind Programme, die das Verhalten von reellen Algorithmen abbilden und meist mit unterschiedlichen Parametern ausgeführt werden. Sie werden oft dazu genutzt, das Verhalten und die Laufzeit von bestimmten Programmklassen zu untersuchen, ohne die jeweiligen Programme vorher implementieren zu müssen. Synthetische Benchmarks sind jedoch nur bedingt dazu geeignet, reelle Programme abzubilden: Das verwendete Modell spielt eine große Rolle und je näher es an dem echten Modell ist, desto genauer werden die Ergebnisse, desto weniger generisch werden jedoch die Benchmarks.

Profiling: Eine weitere Herangehensweise, die Leistungsfähigkeit von parallelen Programmen zu untersuchen, besteht darin, das bereits parallelisierte Programm auszuführen und seine Ausführungscharakteristika zu untersuchen. *OmpP* [54] ist ein *Profiling-Tool* zur Messung der Effektivität von parallelen OpenMP Programmen. Es erkennt Bottlenecks, z.B. bei der Threaderzeugung und bei Synchronisierungspunkten, d.h. bei Abschnitten im Programm, auf dem auf andere Threads gewartet wird. Ein ähnliches Tool stellt NVIDIA mit *Nsight* [55] bereit, welches vergleichbare Aufgaben für CUDA Programme übernimmt.

Die folgenden Unterabschnitte beschreiben zwei grundlegende Methoden zur Bestimmung des Speedups, benchmarkbasierte Verfahren (2.4.1) und modellbasierte Verfahren (2.4.2). Einige dieser Verfahren [32] benötigen spezielle Informationen über die parallele Hardware (z.B. Pipeline-Parallelität), die entweder von den Hardwareherstellern bereitgestellt werden müssen oder aber mit Hilfe von API Aufrufen abgefragt werden können. Da gerade bei neuerer Hardware diese Informationen nicht immer zur Verfügung stehen, beschäftigt sich die Forschung mit Methoden, wie diese Information zu extrahieren sind. Die Autoren in [56] beschreiben in ihrer Arbeit eine Benchmark-Suite, die

dazu dient, die von Hardwareherstellern nicht dokumentierte Charakteristika der GPGPUs zu extrahieren. In [57] wird eine Benchmark-Suite beschrieben, die auf GPGPUs von AMD spezialisiert ist und relevante Parameter wie die Speicherstruktur und Ausführungslatenzen extrahiert.

2.4.1 Benchmarkbasierte Verfahren

Benchmarkbasierte Verfahren zur Speedup-Bestimmung versuchen mit Hilfe von synthetischen Benchmarks das Verhalten von Programmen zu bestimmen, ohne diese konkreten Programme tatsächlich parallelisieren zu müssen. Es reichen Informationen über signifikante Charakteristika, wie die Funktionen, die berechnet werden sollen oder (im Speziellen bei *PBAs*) Populationsgröße und Konfigurationsparameter des *PBAs*. In der Literatur finden sich eine Reihe von Arbeiten, die spezielle Probleme *benchmarken*. In der Arbeit *SpeComp* von *Aslot* et al. [58] wird die *Spec* Benchmark-Suite auf parallele Maschinen mit Hilfe von OpenMP abgebildet. Ein weiterer bekannter Benchmark, NAS, wurde von der NASA auf parallele Maschinen abgebildet [59]. NAS ist ein Benchmark, der ursprünglich für physikalische Berechnungen in der Luftfahrt entwickelt wurde, aber auch auf andere wissenschaftliche Bereiche angewendet werden kann.

GPU-spezifische Benchmarks decken meist nur den Bereich der 3D Bildbearbeitung ab. Der wohl bekannteste Vertreter dieser Art von Benchmarks ist 3DMark [60], welcher die Performance der Grafikkarte in Bezug auf aufwendige 3D Animationen analysiert. Mit *Rodina* [61] existiert eine Benchmark-Suite, die heterogene Architekturen, bestehend aus Multi-Cores und GPGPUs, mit Hilfe unterschiedlicher Problemstellungen und Eingaben analysiert. Eine auf *PBAs* spezialisierte Benchmark-Suite konnte in der Literatur nicht gefunden werden.

2.4.2 Modellbasierte Verfahren

Zur Bestimmung des Speedups eines seriellen Programms auf paralleler Hardware können auch Modelle über die zugrunde liegende Hardware und dem zu

parallelisierenden Programm genutzt werden. Dabei besteht immer die Problematik des Abstraktionsgrads des Modells. Ein sehr abstraktes Modell des zu untersuchenden Problems ist generisch und kann auf mehrere Probleme der gleichen Problemklasse angewendet werden. Außerdem erfordert ein abstrakteres Modell weniger Informationen über das zu untersuchende Problem als ein konkreteres Modell. Je abstrakter jedoch das Modell ist, desto mehr unterscheidet sich für gewöhnlich sein Verhalten vom konkreten Problem. Beispielsweise kann ein abstraktes Modell die Grundform eines PBAs abbilden, mit dem man z.B. bestimmen kann, wie ein allgemeiner PBA mit der Problemgröße skaliert. Der konkrete Algorithmus besteht jedoch aus bestimmten Arten von PBA-Operation, die sich je nach Implementierung anders verhalten. Beispielsweise ist die Art der Crossover-Funktion beim GA (*One-Point-Crossover* oder *Uniform-Crossover*; s. Abschn. 2.2.1) für die genaue Laufzeit des Algorithmus ausschlaggebend oder die Art der Fitnessfunktion, die berechnet werden soll. Je mehr spezifische Informationen über den Algorithmus zur Verfügung stehen, desto genauer kann das Verhalten des Programms mit einem Modell vorhergesagt werden (s. Abschn. 4.2.1).

In [32] untersuchen die Autoren ein Verfahren, um mit Hilfe von Informationen über die Hardware, welche sie größtenteils von den Hardwareherstellern erhalten, ein Modell der GPGPU Hardware zu erstellen, auf dem das zu lösende Problem abgebildet werden kann. Die notwendigen Informationen sind z.B. die Anzahl der Zyklen, die die GPGPU benötigt, um eine bestimmte Rechenoperation durchzuführen, oder die Art der Pipeline. Problematisch wird die Informationsbeschaffung bei neuerer Hardware, da diese Informationen meist der Geheimhaltung seitens der Hersteller unterliegen und nicht öffentlich zugänglich sind. Während das Modell versucht, die unterschiedlichen Speicherhierarchien der GPGPU abzubilden, finden sich Limitierungen in der Betrachtung von Thread Divergenzen. Dies kann bei Verzweigungen im Quellcode auftreten, die dazu führen, dass Threads in einem Thread-Block unterschiedliche Operationen ausführen, oder bei der Synchronisierung der Threads.

Die Autoren in [62] beschreiben in ihrer Arbeit *Kismet* eine auf hierarchischer Code-Analyse basierende Methode zur Speedup-Vorhersage auf Multi-Core-Maschinen. Dabei wird der kritische Pfad im Code bestimmt, also der

Code-Abschnitt, der die längste Ausführungszeit besitzt und nicht parallelisiert werden kann. Für die potentiell parallelisierbaren Abschnitte wird mittels eines Modells der Speedup berechnet, in dem Parameter wie Cache Effekte (z.B. *Cache Miss*) und Anzahl Cores eingehen.

In [63] stellen die Autoren ein Verfahren vor, mit dem sie die Performance von Programmen auf CPUs vorhersagen können, ohne das Programm vollständig auszuführen. Dies spielt vor allem bei lange laufenden Programmen eine große Rolle, bei denen die Laufzeit mehrere Tage oder sogar Wochen beträgt. Hierzu instrumentalisieren sie das Programm in der ersten Phase. Das so modifizierte Programm wird ausgeführt und die so gewonnenen Informationen dazu genutzt, mit Hilfe eines Modells der Hardware (im Beispiel eine MIPS R10000 Maschine) die Laufzeit vorherzusagen. Der modellbasierte Ansatz wurde aufgrund der hier vorgestellten Limitierungen und Abhängigkeiten von Informationen von Hardwareherstellern in dieser Arbeit nicht weiter verfolgt.

2.5 Parallelisierung von seriellem Programmcode

Der Bereich der automatischen Parallelisierung ist ein wichtiger Aspekt von *CASEP*. Die Informationen über den zu erwartenden Speedup werden dazu verwendet, einzelne Teile des Programms auf der Zielhardware zu parallelisieren. Aufgrund der Komplexität dieser Aufgabe, die für sich genommen mehrere wissenschaftliche Arbeiten einnimmt, werden für diese Arbeit verwandte Ansätze hinzugezogen und auf das konkrete Problem der *PBAs* hin optimiert. In Abschnitt 2.5.2 werden verschiedene Frameworks zur Parallelisierung auf ihre Anwendbarkeit in Bezug auf die in dieser Arbeit zu lösenden Probleme hin untersucht. Ein anderer Ansatz wird in Abschnitt 2.5.1 beschrieben, welcher jedoch aufgrund wenig entwickelter Werkzeuge, die diesen Ansatz unterstützen, in dieser Arbeit nicht weiter verfolgt wird. Weiterhin gibt es Ansätze, die eine Transformation von CUDA-Code in parallelen C-Code, zur Vereinheitlichung der Programmiermodelle, beschreiben [64]. Für die hier vorliegende Arbeit ist

jedoch der umgekehrte Fall relevant und es wird somit nicht weiter auf diese Technik eingegangen.

2.5.1 Polyeder Modell

Das Polyeder Modell ist ein theoretisches Verfahren zur Parallelisierung von seriellem Code. Es eignet sich nur für lineare Programme, also Programme, bei denen die Array-Zugriffe lineare Funktionen von Laufvariablen und globalen Variablen sind [65]. Bei dem Polyeder Modell handelt es sich um ein mathematisches Modell, bestehend aus linearen Ungleichungen. Das zu untersuchende Programm wird in die Polyeder-Form überführt und mit Hilfe von affinen Transformationen in eine parallele Form überführt [66].

Obwohl *PBAs* im Grunde lineare Programme sind und sich somit das Polyeder Modell eignet, wurde in dieser Arbeit der Ansatz nicht weiter verfolgt. Die Gründe hierfür sind zum einen die nicht vorhandenen Tools zur Realisierung dieses Ansatzes. Zwar verfügt das in dieser Arbeit eingesetzte Transformations-Tool *ROSE* (s. Abschn. 2.6) Module zur Behandlung von Polyeder Modellen, diese befinden sich jedoch im Anfangsstadium. Zum anderen existieren ausgereiftere Frameworks zur Parallelisierung von seriellen Programmen, auf die im folgenden Abschnitt eingegangen wird.

2.5.2 Frameworks zur Parallelisierung

Für das in dieser Arbeit verwendete Modul zur automatisierten Parallelisierung wurden Tools aus der Literatur und Industrie auf ihre Tauglichkeit hin verglichen, basierend auf der Zusammenstellung in [67]. Zuerst werden die einzelnen betrachteten Frameworks vorgestellt, als nächstes werden in einer Übersicht die relevanten Charakteristika verglichen und zuletzt wird die Entscheidung des in dieser Arbeit verwendeten Frameworks diskutiert.

2.5.2.1 PGI Accelerator

Beim PGI Accelerator [68] markiert der Benutzer die Abschnitte im Quellcode, die parallelisiert werden sollen, mit sogenannten *Accelarator Compute*

Regions Direktiven. PGI übersetzt diese Abschnitte in CUDA-Kernel, wobei auch der Kopiervorgang der Variablen in den Grafikkarten-Speicher übernommen wird. Der *PGI Accelerator* besitzt jedoch einige Limitierungen [67]. So werden beispielsweise dem Entwickler keine manuelle Optimierungsmöglichkeiten des Programmcodes ermöglicht, diese werden automatisch vom Compiler übernommen. Weiterhin ist es nicht möglich, explizit verschiedene Grafikspeicher (s. Abschn. 2.1.2) anzusprechen, was zu weiteren Leistungseinbrüchen bei der Ausführung des parallelen Programms führen kann.

2.5.2.2 OpenACC

OpenACC [69] ist ein auf dem PGI Accelerator basierender Ansatz, der ähnlich wie PGI auf Direktiven basiert. Der Datenaustausch zwischen der GPGPU und dem Hostsystem wird mit Hilfe von Direktiven durchgeführt. Ebenso lassen sich Teile markieren, die in CUDA-Kernel überführt werden. Eine Besonderheit ergibt sich bei der Behandlung von verschachtelten Schleifen. Der Benutzer kann mit sogenannten *kernels constructs* die verschachtelten Schleifen in jeweils separate Kernel überführen oder mit Hilfe von *parallel constructs* einen einzelnen Kernel definieren. Außerdem ist es mit *OpenACC* möglich, explizit aus einer Schleife einen *Reduction Kernel* (s. Abschn. 4.4.7) zu erzeugen, was zu einer erheblichen Leistungssteigerung führen kann. Aufgrund der *PGI Accelerator* Basis, verfügt der *OpenACC* Compiler über ähnliche Limitierungen wie der *PGI Accelerator* [67].

2.5.2.3 HMPP

Bei HMPP (*Hybrid Multicore Parallel Programming*) [70] werden die zu parallelisierenden Abschnitte vom Benutzer ebenfalls mit Direktiven markiert. Das Framework beherrscht jedoch nicht nur eine Parallelisierung auf CUDA, sondern auch auf den offenen Standard OpenCL [71]. Hierfür findet eine Abstraktion der GPGPU Programmierung statt und das Überführen in parallelen Code wird mit sogenannten *codelets* bewerkstelligt. Codelets sind Funktionen,

die auf GPGPUs ausgeführt werden können, jedoch keinen Rückgabewert besitzen. Dementsprechend können, je nach Programm, Änderungen im Quellcode notwendig sein. Weiterhin besitzen *codelets* einen festen Speicherplatz auf der GPGPU. Sollte das gleiche *codelet* von einem anderen Bereich im Programmcode nochmal mit anderen Daten aufgerufen werden, so überschreiben die neuen Daten die Daten des alten *codelets*. Dies kann dazu führen, dass für das erste *codelet* die Daten noch einmal auf die GPGPU kopiert werden müssen, was zu einem *Overhead* im Datenaustausch führt [67].

2.5.2.4 OpenMPC

OpenMPC [72] ist eine Erweiterung von OpenMP [73]. Mit OpenMP lassen sich Bereiche im Source-Core markieren, die vom Framework in parallelen Code für Multi-Core Maschinen überführt werden. OpenMPC erweitert die Überführung dieser Bereiche auf GPGPUs, wobei die spezielle Speicher-Architektur der GPGPUs berücksichtigt wird. Mit bestimmten Direktiven kann bestimmt werden, auf welchen Speicher (z.B. *shared*- oder *global*; s. Abschn. 2.1.2) die Dateien gespeichert werden sollen. Weiterhin kann festgelegt werden, ob Daten zwischen der GPGPU und dem Host-System kopiert werden sollen.

2.5.2.5 hiCUDA

Den am wenigsten abstrahierten Ansatz stellen Tianyi et al. mit ihrem Framework *hiCUDA* [74] bereit. Ähnlich wie die anderen hier vorgestellten Frameworks lassen sich mit Direktiven Stellen zur Parallelisierung im Quellcode markieren. Bei hiCUDA lassen sich jedoch fast alle Möglichkeiten von CUDA ausschöpfen, wie z.B. die Allozierung der Daten auf verschiedenen Speicherebenen und das Setzen von Synchronisierungspunkten zwischen den verschiedenen Kerneln.

2.5.2.6 R-Stream

R-Stream [75] basiert anders als die zuvor beschriebenen Ansätze auf dem Polyeder Modell. Somit ist es auf lineare Programme beschränkt, ist jedoch,

aufgrund der Abstraktion auf das Polyeder Modell, universell auf unterschiedliche parallele Architekturen anwendbar. Beispielsweise gibt es Versionen von R-Stream, die OpenMP unterstützen und andere die CUDA unterstützen. Weiterhin übernimmt R-Stream das Datenmanagement für die unterschiedlichen Speicher, der Benutzer muss nur die parallelen Abschnitte im seriellen Quellcode markieren.

2.5.2.7 Vergleich

Die hier vorgestellten Frameworks haben jeweils unterschiedliche Stärken und Schwächen. Zum besseren Vergleich wurden sie anhand von sechs relevanten Kriterien für die hier vorgestellte Arbeit miteinander verglichen [65]:

1. Ist der Datentransfer zwischen GPU und CPU beeinflussbar?
2. Kann der Datenspeicher auf der GPGPU frei alloziert und freigegeben werden?
3. Ist jede Ebene der Speicherhierarchie von CUDA ansprechbar?
4. Sind Schleifenoptimierungen möglich?
5. Wird C++ unterstützt?
6. Ist die Lizenz kostenlos?

Tabelle 2.1 stellt die vorgestellten Kriterien im Vergleich dar. Dabei bedeutet ein „+“, dass das Framework dieses Feature unterstützt und ein „-“, dass dieses Feature vom Framework nicht unterstützt wird. Auffallend ist, dass OpenMPC als einziges unter den hier untersuchten Frameworks eine freie Lizenz bietet und bis auf die C++ Überstützung alle Kriterien erfüllt. Als einzige Alternative, die alle notwendigen Kriterien unterstützt, bietet sich HMPP an, welches jedoch keine freie Lizenz bietet. Dementsprechend sind die zwei Kandidaten für das hier verwendete Framework HMPP und OpenMPC. Die C++ Unterstützung ist für große Projekte essentiell, jedoch beschränkt sich *CASEP* auf *PBAs*, welche zum größten Teil ohne großen Aufwand in C implementiert werden können. Üblicherweise wird nur eine kleine Teilmenge der

Kriterium	PGI	OpenACC	HMPP	OpenMPC	hiCUDA	R-Stream
1	+	+	+	+	+	+
2	+	+	+	+	+	-
3	-	-	+	+	+	-
4	-	-	+	+	-	-
5	+	+	+	-	-	-
6	-	-	-	+	-	-

1: Datentransfer, 2: Datenspeicher, 3: Speicherhierarchie, 4: Schleifenoptimierungen, 5: C++, 6: Lizenz kostenlos

Tabelle 2.1: Vergleich der Frameworks auf Basis der notwendigen Kriterien

von C++ bereitgestellten Konstrukte für *PBAs* benötigt, wie Vektoren und Klassen. Diese lassen sich in späteren Arbeiten mit Hilfe von Parsern und Übersetzern nachträglich in *CASEP* einfügen, ohne eine Re-Implementierung des kompletten Frameworks durchzuführen. Die Vektoren können dabei von einem separaten Modul in Arrays umgewandelt werden, während die Klassen als C-Structs dargestellt werden können. Dementsprechend fiel die Wahl für *CASEP* auf OpenMPC.

2.6 ROSE

Die Implementierung von *CASEP* erfordert ein Werkzeug zum Parsen und Manipulieren von Quellcode. Der folgende Abschnitt stellt den in dieser Arbeit dafür genutzten *ROSE*-Compiler vor.

Bei ROSE [76] handelt es sich um ein komplexes Quellcode-Manipulationswerkzeug, mit dem Programme sowohl geparsed, als auch manipuliert und transformiert werden können. *ROSE* wandelt hierzu das eingegebene Programm in sogenannte *Abstrakte Syntaxbäume* (AST [77]) um. Diese abstrakte Repräsentation des Quellcodes lässt sich manipulieren und anschließend mit entsprechenden Übersetzern in die gewünschte Programmiersprache überführen. *ROSE* bietet verschiedene Übersetzer, u.a. auch für OpenMP, an. Für das in dieser Arbeit benötigte CUDA (s. Abschn. 2.1.2.1) wird ebenfalls ein Übersetzer angeboten, welcher sich jedoch zum Zeitpunkt dieser Arbeit noch in

der Entwicklung befand und einige Fehler aufwies. Auf die konkreten Probleme und Lösungen wird im jeweiligen Abschnitt (4.4) dieser Arbeit genauer eingegangen.

2.7 Zusammenfassung

In diesem Kapitel wurden die für diese Arbeit signifikanten Technologien beschrieben und verwandte Arbeiten gewürdigt.

Zuerst wurde auf die verwendeten parallelen Architekturen und Programmiermodelle eingegangen. Es wurde der Grundaufbau der GPGPUs und Multi-Core Architekturen beschrieben und die jeweiligen Charakteristika herausgearbeitet. Dabei wurde festgestellt, dass sich GPGPUs aufgrund ihrer Bauweise besonders gut für die parallele Ausführung von vielen gleichartigen Operationen (*many-thread*) eignen. Multi-Core CPUs eignen sich hingegen besser für die Ausführung von vergleichsweise wenigen parallelen Threads (*multi-thread*), wobei jedoch jeder einzelne Thread auf einer Multi-Core CPU weitaus schneller abgearbeitet werden kann. Dies ergibt sich aus den leistungsstärkeren einzelnen Kernen von Multi-Core CPUs im Vergleich zu den Kernen von GPGPUs und den komplexeren Pipeline- und Cache-Architekturen der Multi-Core CPUs.

Als nächstes wurden die *PBAs* eingeführt. Dabei wurde zuerst der Grundaufbau von *PBAs* beschrieben und auf die zwei häufig verwendeten *PBAs* *GA* und *PSO* eingegangen, die den Fokus für die in dieser Arbeit beschriebenen Techniken bilden. Es wurden die einzelnen charakteristische Funktionen beschrieben und Parallelisierungsmethoden der vorgestellten *PBAs* aus der Literatur diskutiert.

Im weiteren Verlauf des Kapitels wurden Arbeiten aus dem Bereich der automatisierten Parallelisierung aus der Literatur vorgestellt. Diese wurden anhand von für diese Arbeit wesentlichen Charakteristika miteinander verglichen und es wurde die Wahl der für *CASEP* verwendeten Technologie diskutiert. Zuletzt wurde auf das in dieser Arbeit verwendete Code-Manipulationswerkzeug *ROSE* eingegangen.

3 System Design - Überblick und Einordnung

Inhaltsangabe

Dieses Kapitel beschreibt die Systemarchitektur von *CASEP* und ordnet es in das zu unterstützende Umfeld ein. Zuerst wird in Abschn. 3.1 das Anwendungsfeld von *CASEP* vorgestellt. In Abschn. 3.2 werden die Systemgrenzen definiert und die in dieser Arbeit verwendete Nomenklatur festgelegt. Abschließend wird in 3.3 die Architektur von *CASEP* vorgestellt und die Abgrenzung dieser Arbeit von verwandten Arbeiten aus der Literatur in Abschn. 3.4 diskutiert.

3.1 Anwendungsfeld

Das Hauptziel dieser Arbeit ist die Entwicklung eines generischen Frameworks zur Entwicklerunterstützung bei der Erstellung von parallelen Programmen. Dies umfasst insbesondere drei wesentliche Blöcke: 1) *Unterstützung bei der*

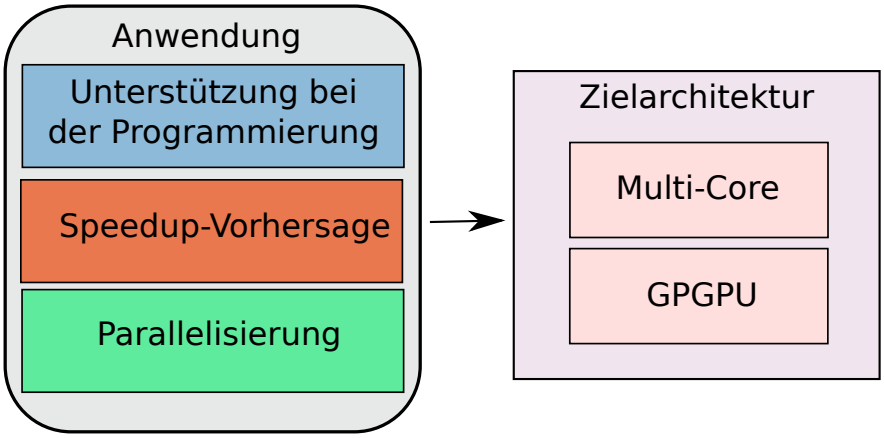


Abbildung 3.1: Anwendungsbereiche von CASEP

Programmierung, 2) *Speedup-Vorhersage* und 3) *Parallelisierung*, wie in Abb. 3.1 dargestellt. Diese drei Blöcke werden im Folgenden detailliert beschrieben.

CASEP unterstützt den Entwickler bei der Erstellung von parallelen Programmen, indem Teile des Quellcodes automatisiert in parallele Code-Sequenzen umgewandelt werden, die entweder zur Verringerung der Laufzeit und/ oder zur Verbesserung der Ergebnisse führen. Diese Transformationen werden autonom durchgeführt, wobei vom Entwickler lediglich die gewünschten Bereiche im Quellcode markiert werden müssen. Dieser Schritt ist notwendig, da aufgrund der Polymorphie von Programmen die benötigten Abschnitte nur teilweise automatisiert gefunden werden können. Operationen (mathematische Operationen, Variablendeklarationen, Funktionsnamen, usw.) können im Programmcode auf unterschiedliche Weise dargestellt werden. Beispielsweise kann es vorkommen, dass gleiche Operationen in unterschiedlichen Teilen des Quellcodes vorkommen. Diese können in *PBA*-spezifische Operationen (wie *Mutation*, *Crossover* usw.) vorkommen, aber auch in Bereichen, die keiner spezifischen Funktion zugeordnet sind. In diesem Fall ist es nicht möglich automatisiert zu unterscheiden, welche Operationen die spezifische Funktion abbilden, die parallelisiert werden soll.

Der zweite Bereich, der von *CASEP* abgedeckt wird, ist der Bereich der Speedup-Vorhersage (s. Abschn. 2.4). Dabei handelt es sich um den vorhergesagten Speedup von seriellen Programmen auf parallelen Architekturen, ohne

die Programme vorher explizit parallelisieren zu müssen. Von *CASEP* werden mehrere unterschiedliche Hardwarearchitekturen als Zielarchitektur für die Parallelisierung analysiert und das bestmögliche Mapping (kürzeste Laufzeit des Programms) bestimmt. Dadurch wird es dem Entwickler ermöglicht, sein Programm auf eine bestimmte Hardware zu optimieren, ohne vorher das Programm für die unterschiedlichen Architekturen zu implementieren und durch Ausführen von parallelem Quellcode das beste Mapping zu bestimmen. Weiterhin ist es möglich *CASEP* zu erweitern, um *hybride Mappings* zu erzeugen, also Mappings, bei denen die einzelnen Bereiche des Quellcodes auf unterschiedlichen Architekturen ausgeführt werden, wie in Abb. 3.2 beispielhaft dargestellt. In diesem Beispiel besteht das Programm aus drei Schleifen (Loop 1, Loop 2, Loop 3), die auf die vorhandene Hardware abgebildet werden. Dabei wird Loop 1 seriell auf der Multi-Core CPU mit einem Thread ausgeführt, Loop 2 ebenso auf der CPU, aber mit mehreren Threads und Loop 3 wird parallel auf der GPGPU ausgeführt.

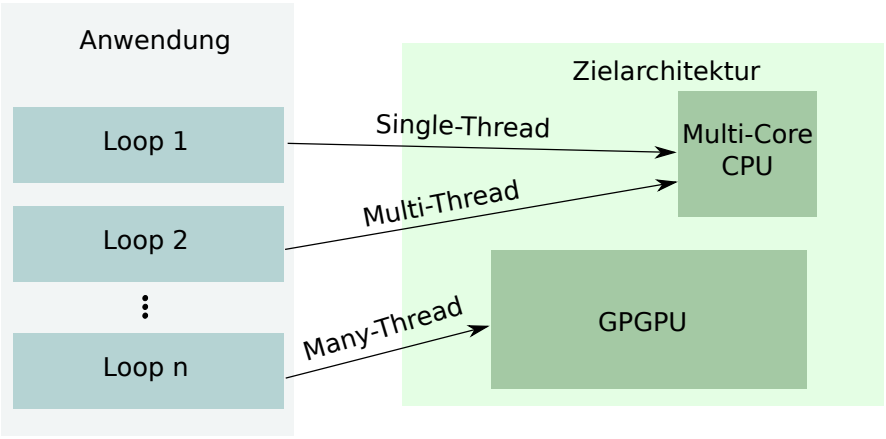


Abbildung 3.2: Beispiel für Software-Hardware-Mapping

Der letzte Bereich ist die Parallelisierung (s. Absch.2.5). *CASEP* verwendet die vorhergesagten Speedup-Ergebnisse zur teilautomatisierten Erzeugung von parallelem Quellcode. Auch für diesen Bereich gilt, dass aufgrund der Polymorphie, der Entwickler einzelne Bereiche markieren muss. Beispielsweise ist es notwendig zu markieren, in welchem Bereich die Fitnessfunktion (s. Abschn. 2.2) berechnet wird, damit Code-Optimierungen durchgeführt werden können.

Die hier vorgestellten Bereiche (*Unterstützung bei der Programmierung*, *Speedup-Vorhersage* und *Parallelisierung*) finden sich in den nächsten Kapitel wieder und werden jeweils zur besseren Orientierung in den hier vorgestellten Farben dargestellt. Teilweise übernehmen einzelne Module mehrere Aufgaben, in diesem Fall werden die Module jeweils mit mehreren Farben dargestellt.

3.1.1 Ermittlung der optimalen Ausführungsbereiche - Software-Hardware-Mapping

Bei der Implementierung von Software ist die Entscheidung für eine Architektur nicht trivial, sofern die kürzeste Laufzeit des Programms erreicht werden soll. Im Allgemeinen kann diese Entscheidung nicht getroffen werden, ohne dass das Programm auf der jeweiligen Architektur ausgeführt wird. *CASEP* unterstützt den Entwickler dabei die Architekturen zu bestimmen, die bei bestimmten Problemgrößen die geringste Ausführungszeit erreichen. Das Problem ist in Abb. 3.3 abgebildet. Auf der x-Achse ist die Berechnungsgröße abgebildet. In den in dieser Arbeit betrachteten Fällen ist dies die Problemgröße, bestehend aus Populationsgröße mal Dimensionsgröße (s. Abschn. 6). Auf

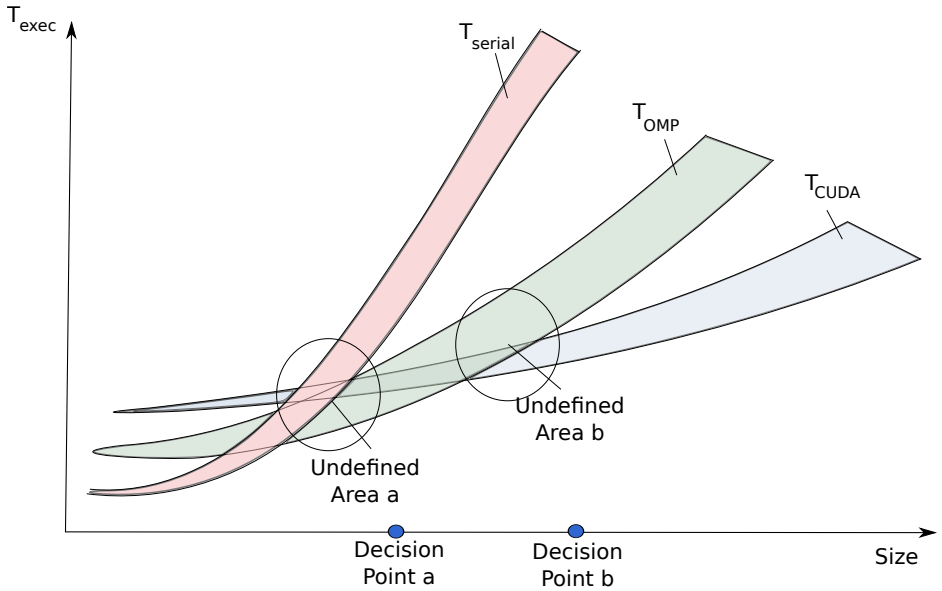


Abbildung 3.3: Ausführungsbereiche des Eingabeprogramms

der y-Achse ist die Ausführungszeit des jeweiligen Programms abgebildet. Es sind exemplarisch die Ausführungszeiten von drei Versionen des gleichen Programms abgebildet: 1) eine serielle Version, 2) eine mit OpenMP parallelisierte Version und 3) eine CUDA-Version des Programms. Die Ausführungszeiten richten sich dabei an den Evaluationsergebnissen (s. Abschn. 6) dieser Arbeit und stellen ein typisches Ausführungsverhalten dar. In der Regel hat die serielle Ausführung, aufgrund des geringen Overheads, die kürzeste Ausführungszeit bei kleinen Berechnungsgrößen. Bei mittleren Berechnungsgrößen ist in der Regel die OpenMP-Version des Programms am schnellsten, während für große Berechnungsgrößen die CUDA-Version die kürzeste Ausführungszeit aufweist. Das Problem des Entwicklers ist die jeweiligen Punkte zu bestimmen, bei denen sich eine Parallelisierung mit Hilfe von OpenMP (*Decision Point a*) lohnt und eine Parallelisierung mit Hilfe von CUDA (*Decision Point b*).

Diese Entscheidung kann nicht allgemeingültig für bestimmte Berechnungsgrößen bestimmt werden. Sie ist abhängig von der Zielarchitektur, der Art der Operationen und der Last auf dem Zielsystem. Weiterhin existieren Bereiche (*Undefined Area a/b*), in denen sich die Ausführungszeiten überlappen und somit eine genaue Vorhersage nicht möglich ist. *CASEP* hilft dem Entwickler dabei diese Bereiche zu bestimmen, indem die voraussichtlichen Ausführungszeiten der drei Programmversionen für unterschiedliche Programmkonfigurationen und Zielarchitekturen bestimmt werden. Dadurch ist es möglich, das Programm in der Version (seriell, parallel mit OpenMP oder parallel mit CUDA) zu implementieren, in der die kürzeste Ausführungszeit zu erwarten ist.

3.2 Systemgrenzen

CASEP ist im Umfeld von Analyse- und Parallelisierungs- und Optimierungswerkzeugen von Quellcode einzuordnen. Der in *CASEP* stattfindende Prozess ist semi-automatisiert, Eingaben vom Benutzer sind notwendig, um die nötige Genauigkeit bei den Analysen zu erhalten. Es handelt sich somit um eine semi-transparente *Blackbox* aus Sicht des Benutzers. Auch wenn die grundsätzlichen Methoden von *CASEP* auf eine Vielzahl von Architekturen und Algorithmen

anwendbar sind, liegt der Fokus dieser Arbeit auf einer Untermenge dieser. Im Bereich der Algorithmen konzentriert sich die Arbeit auf *PBAs* (s. Abschn. 2.2), die eine Untermenge von *Naturanalogen Verfahren* bilden. *PBAs* lassen sich besonders gut parallelisieren (s. Abschn. 2.2 und 2.3), außerdem erlaubt eine Fokussierung auf eine bestimmte Algorithmenklasse eine stärkere Optimierung. Weiterhin wurden *PBAs* aufgrund ihres simplen Grundaufbaus bestehend aus einer Iterationsschleife und mehreren spezifischen Operationen gewählt. Dies erlaubt eine genauere Betrachtungsweise des Grundproblems. Andere Algorithmen aus der Klasse biologisch inspirierter Algorithmen besitzen ebenso einen einfachen Grundaufbau, sind jedoch in der Praxis nicht so weit verbreitet wie *PBAs* und bieten, anders als *PBAs*, keine eins zu eins Abbildung von Individuen auf Thread-Ebene (s. Abschn. 2.3). In der Arbeit werden in den jeweiligen Stellen die generischen und speziellen Abschnitte des Frameworks hervorgehoben (s. Kapitel 5). Des Weiteren wird auf potentielle Generalisierungen der einzelnen Module eingegangen.

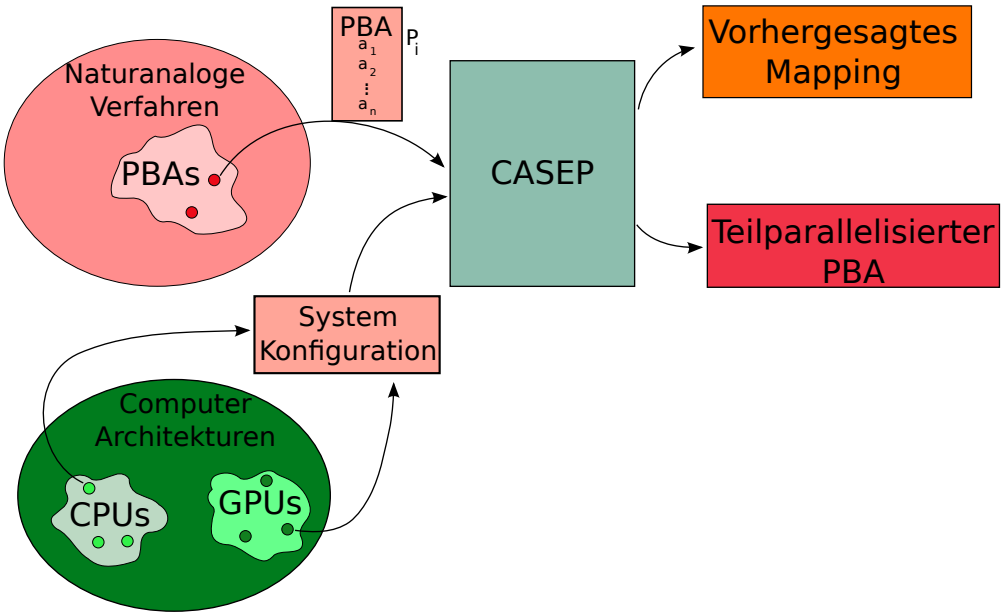


Abbildung 3.4: CASEP System Modell

Außer den Algorithmen, die als Eingabe fungieren, benötigt *CASEP* eine gültige Systemkonfiguration (Hardwareparameter), unter der das *CASEP* Fra-

mework ausgeführt wird, als weiteren Eingabeparameter. Ähnlich dem Bereich der Algorithmen, sind die bei *CASEP* verwendeten Methoden weitestgehend auf unterschiedliche Hardwarekonfigurationen anpassbar. Die Anwendbarkeit von *CASEP* auf unterschiedliche Hardware wird in Abschn. 5.2 behandelt. Der Fokus dieser Arbeit liegt auf GPGPUs (s. Abschn. 2.1.2) und Multi-Core CPUs (s. Abschn. 2.1.1), eine der häufigsten Konfigurationen bei heutigen Computersystemen. Diese Wahl wurde aufgrund der weiten Verbreitung bei Entwicklern getroffen. Spezialisierte parallele Co-Prozessoren (z.B. FPGAs [78] [79]) werden für gewöhnlich für Software verwendet, bei denen eine weitaus komplexere und weitreichende Optimierung des Programmcodes notwendig ist, als *CASEP* bereitstellen kann.

In Abb. 3.4 ist eine Systemübersicht abgebildet. $a_1, a_2, ...a_n$ sind die einzelnen Abschnitte (*Crossover*, *Mutation*, ...) im Quellcode des *PBAs* P_i, der als Eingabe für *CASEP* ausgewählt wird. Es wird nur dann eine parallele Version erstellt, wenn die serielle Ausführungszeit des Programms größer als die Ausführungszeit einer parallelen Version ist. Folglich muss gelten, dass die Summe der Ausführungszeiten der einzelnen Abschnitte des sequentiellen Programms größer als die Summe der Ausführungszeiten des teilparallelen Programms sein muss:

$$t_{seq}(P_i) = \sum_{i=1}^{n} t(a_i) > \sum_{i=1}^{n} t(a_i*) = t_{par}(P_i)$$

Dabei indiziert der $*$ Operator, dass mindestens einer der Abschnitte von a_i* parallelisiert ist. *CASEP* erzeugt zwei Ausgaben, ein von der eingegebenen Systemkonfiguration abhängiges Mapping des eingegebenen *PBAs* auf die Hardware und eine teilparallelisierte Version des *PBAs*, basierend auf dem vorhergesagten Mapping.

3.2.1 Vergleichbarkeit der Ergebnisse

Eine der größten Herausforderungen bei der Parallelisierung ist die Vergleichbarkeit der Ergebnisse zu garantieren. Die Ergebnisse der parallelen Version sollen im Idealfall identisch mit denen der seriellen Version sein. Für den Fall

der Funktionsoptimierung, der in dieser Arbeit in der Evaluation betrachtet wird, bedeutet dies, dass in der gleichen Anzahl an Iterationen die gleiche Lösungsgenauigkeit erreicht werden soll. Die Lösung bei *PBAs* hängt von der Reihenfolge der ausgeführten Operationen ab, die bei parallelen Programmen in der Regel nur durch Synchronisierungspunkte zwischen den einzelnen Threads garantiert werden kann. Synchronisierungspunkte führen jedoch zu einer Erhöhung der Ausführungszeit und verringern die Leistung des parallelen Programms.

Das Problem der Synchronisierung wird anhand eines Beispiels erläutert: Die Aufgabe besteht darin die *Rekombination*-Operation eines *GAs* (s. Abschn. 2.2.1) zu parallelisieren. Bei der *Rekombination* entstehen neue Individuen, aus denen im nächsten Rekombinationsschritt neue Elternindividuen ausgewählt werden können. Die Populationsgröße n ändert sich somit nach jedem Rekombinationsschritt zu $n_{neu} = n + 2$ (n Individuen und zwei neue Kindindividuen). Im seriellen Fall würde die Rekombination der Individuen in einer bestimmten Reihenfolge ausgeführt werden. Bei gleichem Zufalls-Seed würden immer die gleichen Elternindividuen ausgewählt werden. Im parallelen Fall kann dies i.A. nicht garantiert werden, da die Reihenfolge der Rekombination nicht festgelegt ist. Um eine gleiche Rekombination zu gewährleisten, müsste nach jeder Operation eine Synchronisierung stattfinden. Dies würde jedoch dazu führen, dass die Rekombination gewissermaßen sequentiell ausgeführt wird. Hinzu kommt außerdem die Zeit, die für die Synchronisierung notwendig ist, was die Ausführungszeit weiter erhöhen würde. Gleiche Ergebnisse können somit bei einer parallelen Ausführung nur garantiert werden, wenn die einzelnen Operationen vollständig unabhängig voneinander ausgeführt werden können. Bei *PBAs* ist dies somit in den meisten Operationen nicht möglich, was jedoch nicht zu falschen Ergebnissen führt. Dies liegt daran, dass die *PBA*-Operationen zum größten Teil einen Zufallsfaktor beinhalten und keine allgemeingültige richtige Ausführungsreihenfolge bestimmt werden kann. Es ist somit für eine Parallelisierung notwendig, dass die Ergebnisse vergleichbar sind, also ähnlich gute Ergebnisse berechnet werden und eine ähnliche Konvergenzgeschwindigkeit erreicht wird, verglichen mit der seriellen Version. Identische Ergebnisse können nur bei komplett unabhängigen Operationen ohne

Leistungsverlust garantiert werden.

3.3 Systemarchitektur

In diesem Abschnitt wird die grundlegende Architektur von *CASEP* vorgestellt. In Abb. 3.5 ist die Architektur mit den in Abschn. 3.1 vorgestellten Anwendungsbereichen dargestellt. Die *CASEP* Architektur besteht aus vier Schichten. In der untersten Schicht (*Information Crawling*) werden die Informationen bereitgestellt, die für die oberen Schichten notwendig sind. Als nächstes wird in der *Speedup Estimation* Schicht der potentielle Speedup des Programms bei einer Parallelisierung anhand von zwei unterschiedlichen Methoden bestimmt. Die beiden so ermittelten Speedup-Werte werden in der *Mapping* Schicht zu einer globalen Speedup-Vorhersage aggregiert und an die letzte Schicht weitergeleitet (*Parallelization* Schicht). Dort wird das serielle Programm anhand der Vorhersage semi-automatisch parallelisiert. Das Ziel und die Methodik der einzelnen Schichten werden im Folgenden im Detail beschrieben.

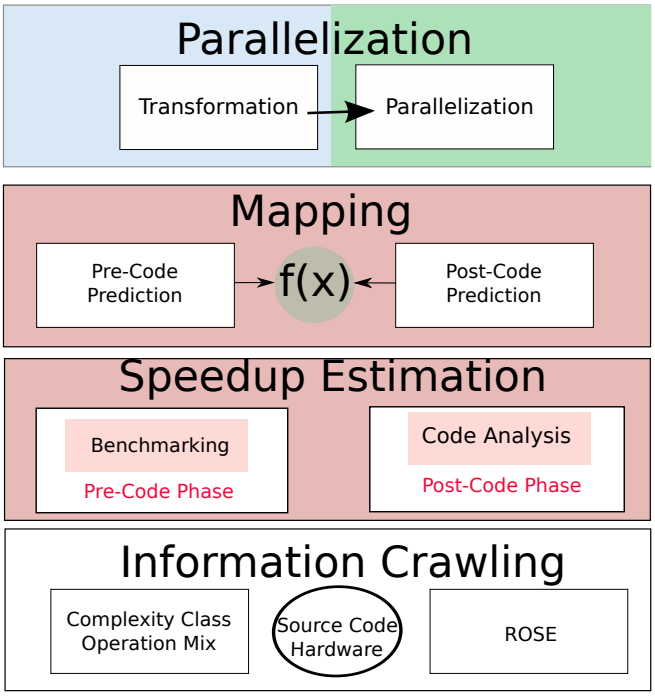


Abbildung 3.5: CASEP Systemarchitektur

3.3.1 Information Crawling Schicht

Ziel:

In der untersten Schicht werden die notwendigen Informationen über die Umgebung, in der *CASEP* eingesetzt wird, gesammelt. Bei diesen Informationen kann es sich um Hardwareparameter oder Informationen über das zu untersuchende Programm handeln. Diese Informationen werden den darüberliegenden Schichten bereitgestellt.

Methodik:

Die Informationen werden anhand einer Analyse des eingegebenen Quellcodes und der Hardware, auf der der potentiell parallele Quellcode ausgeführt werden soll, gesammelt. *CASEP* bedient sich hierzu sowohl API-Aufrufen, als auch Code-Analyse Programmen (ROSE, s. Abschn. 2.6) und Messungen mit Hilfe von *Micro-Benchmarks*. Micro-Benchmarks bilden im Vergleich zu herkömmlichen Benchmarks nur einen sehr kleinen Berechnungsabschnitt ab. Dies kann

z.B. eine bestimmte Funktion oder Transformation im Programmcode sein. Außerdem werden Eingaben vom Benutzer dazu verwendet die Genauigkeit des Frameworks zu verbessern. Diese Eingaben können z.B. Informationen über die vorhandenen Operationen (Kosinus, Addition, ...) im Quellcode sein (s. Abschn. 4.2.1).

3.3.2 Speedup Estimation Schicht

Ziel:
Den Fokus der *Speedup Estimation* Schicht (SES) bildet die Speedup-Vorhersage für das Eingabeprogramm, das im vorangegangenen Schritt von der *ICS* untersucht wurde. Die Informationen der *ICS* werden dieser Schicht übergeben, um anhand von zwei unterschiedlichen Methoden den potentiellen Speedup des seriellen Eingabeprogramms zu ermitteln. Hier soll eine genauere Vorhersage erreicht werden, indem die Schwächen der einzelnen Vorhersagen von der jeweils anderen Vorhersage ausgeglichen werden. Die Schwächen der einzelnen Methoden werden experimentell in Abschnitt 6.3 analysiert und diskutiert. Weiterhin unterscheiden sich die beiden Methoden in ihrer zeitlichen Anwendung während des Entwicklungsprozesses. Im Fokus dieser Arbeit wird zwischen zwei Phasen unterschieden, der *Pre-Code*-Phase und der *Post-Code*-Phase. In der *Pre-Code*-Phase ist nicht das konkrete serielle Programm vorhanden, sondern nur einzelne Informationen darüber. Diese können z.B. die Art des Algorithmus oder bestimmte Berechnungsfunktionen sein. Bei der *Post-Code*-Phase ist das konkrete serielle Programm vorhanden und es stehen dementsprechend mehr Informationen zur Verfügung. Während beide Vorhersagemethoden in der *Post-Code*-Phase verwendet werden können, kann nur die weiter unten beschriebene *Benchmarking*-Methode in der *Pre-Code*-Phase verwendet werden, da für die *Code-Analysis*-Methode der Quellcode des seriellen Programms vorhanden sein muss. Das Verwenden der Vorhersage in der *Post-Code*-Phase ermöglicht es dem Entwickler, eine erste Einschätzung über die Parallelisierung des Programms durchzuführen, und gegebenenfalls den Entwicklungsprozess daran anzupassen. Der *Benchmarker* kann weiterhin unterschiedliche Laufzeitparameter untersuchen, die für die spätere Ausführung

des parallelen Programms notwendig sind, und die besten auswählen (s. Abschn. 4.2.1).

Die *SES* hat für das Gesamtframework mehrere Ziele:

1. Die Bestimmung des zu erwartenden Speedups des seriellen Programms auf der vorhandenen Hardware für ein optimales Software-Hardware-Mapping mit dem Ziel einer möglichst kurzen Gesamtausführungszeit.

2. Die Bestimmung des zu erwartenden Speedups in der *Pre-Code*-Phase zur besseren Anpassung des Programms und als ersten Anhaltspunkt für die zu erwartenden Ausführungszeiten auf unterschiedlichen Architekturen.

3. Als drittes Ziel dient die Extraktion von Ausführungsparametern für das potentiell parallele Programm (z.B. die optimale Threadanzahl). Diese Parameter sollen zu einer möglichst optimalen Ausführungszeit des parallelisierten Programms führen.

Methodik

In der *Pre-Code*-Phase werden mit Hilfe von Benchmarks, die das Verhalten des reellen Programms simulieren, Rückschlüsse über die Laufzeit der einzelnen Programmabschnitte auf verschiedener Hardware geschlossen. Die Bestimmung des Speedups in der *Post-Code*-Phase beruht auf statischer und dynamischer Code-Analyse und wird vom *Code-Analysis*-Modul durchgeführt. Die vom Programm auszuführenden Operationen werden analysiert und für jeden zu betrachtenden Code-Abschnitt werden Micro-Benchmarks erzeugt, mit denen die Ausführungszeiten dieser Operationen auf unterschiedlicher Hardware untersucht werden können. Üblicherweise sind die Anzahl der Iterationen, die ein *PBA* ausgeführt, nicht zur Compile-Zeit bekannt, jedoch für die Ausführung der Micro-Benchmarks zwingend notwendig. Die Operationen der Micro-Benchmarks müssen in der Häufigkeit der Operationen des reellen Programms durchgeführt werden, um vergleichbare Ergebnisse zu erhalten. Zur Auflösung dieser Problematik wird eine dynamische Code-Analyse durchgeführt, bei der Laufzeitparameter bestimmt werden können. Es wird

ebenso eine gemeinsame Betrachtung des Speedups der beiden hier vorgestellten Methoden in der *Post-Code*-Phase betrachtet. Dadurch soll eine genauere Vorhersage erreicht werden.

3.3.3 Mapping Schicht

Ziel

Der Fokus dieser Schicht liegt darin, aus den in der *SES* erfolgten (möglicherweise unterschiedlichen) Speedup-Vorhersagen eine für die nachfolgende Schicht gültige Speedup-Vorhersage zu erstellen. Durch die Aggregierung soll eine genauere globale Vorhersage erreicht werden.

Methodik

Zur Erzeugung der aggregierten Vorhersage wird in der *Mapping*-Schicht eine Funktion genutzt, welche aus Versuchen (s. Abschn. 6.3) abgeleitet ist, um eine möglichst genaue finale Vorhersage aus den unabhängigen Grundvorhersagen der zweiten Schicht zu berechnen. Diese Funktion gewichtet die einzelnen Vorhersagen, beruhend auf den aus Versuchen abgeleiteten Genauigkeiten für die unterschiedlichen in der Evaluation betrachtete Probleme. Es wurde eine optimale Gewichtung der Vorhersagemodule für die hier untersuchte Klasse der *PBAs* untersucht. Andere Klassen besitzen möglicherweise eine andere Gewichtung der Mapping-Funktion und werden in dieser Arbeit nicht weiter behandelt.

3.3.4 Parallelization Schicht

Die oberste Schicht von *CASEP* wird in diesem Abschnitt beschrieben.

Ziel

Im letzten Schritt wird in der obersten Schicht (*Parallelization*) das serielle Programm, basierend auf dem vorhergesagten Software-Hardware-Mapping, parallelisiert. Dabei werden nur die Abschnitte parallelisiert, die anhand der Vorhersage einen Geschwindigkeitsvorteil gegenüber einer seriellen Version des Programms versprechen. Des Weiteren werden in der *Parallelization* Schicht

Code-Optimierungen durchgeführt, die zu einem Geschwindigkeitsvorteil und/ oder einer Verbesserung der Ergebnisse führen.

Das Ziel der *Parallelization* Schicht ist ein zum Teil parallelisiertes Programm auf der Zielhardware, welches eine kürzere Ausführungszeit als das serielle Eingabeprogramm vorweist. Weiterhin muss das Ergebnis des parallelen Programms mit dem Ergebnis des seriellen Programms vergleichbar sein. Die Parallelisierung soll möglichst automatisiert erfolgen und die vom Entwickler benötigten Eingaben minimiert werden.

Methodik
Für die Parallelisierung wird ein offenes Tool (OpenMPC; s. Abschn. 2.5.2.4) verwendet, welches mit Hilfe von für diese Arbeit geschriebenen Module an die hier vorliegende Problemklasse angepasst wurde. Die Anpassungen dienen zum einen dazu, Fehler bei der Parallelisierung zu vermeiden und zum anderen die Eingaben des Entwicklers bei der Parallelisierung zu minimieren. Weiterhin führt die Schicht Transformationen im Quellcode durch, die eine schnellere Ausführung von parallelen *PBAs* ermöglichen und das Ergebnis der *PBAs* verbessern.

3.4 Abgrenzung zum Stand der Technik

In Kapitel 2 wurden unterschiedliche Techniken und Technologien, bezogen auf parallele Programmierung, aus der Fachliteratur aufgezeigt. Die vorgestellten Arbeiten beschäftigen sich mit der Lösung von Teilproblemen, die in ähnlicher Form von dem in dieser Arbeit vorgestellten Werkzeug *CASEP* bearbeitet werden. Diese Dissertation grenzt sich jedoch in folgen Punkten von den dort beschriebenen Arbeiten ab:

1. *Vollständigkeit*: *CASEP* beschäftigt sich mit dem vollständigen Prozess der parallelen Softwareentwicklung. Während die Arbeiten aus der Literatur einzelne Teilprobleme mit Erfolg lösen, nutzt *CASEP* Synergien aus den verschiedenen Abschnitten der Softwareentwicklung. Hierdurch ist es beispielsweise möglich, den berechneten potentiellen Speedup als

Faktor für ein korrektes Software-Hardware-Mapping und letztendlich für die Parallelisierung zu nutzen.

2. *Redundante Speedup-Berechnung*: *CASEP* nutzt unterschiedliche Arten der Speedup-Vorhersage. Dadurch ist es möglich, kleinere Ungenauigkeiten einzelner Speedup-Vorhersage-Methoden auszugleichen, und eine genauere globale Vorhersage zu erhalten.

3. *Laufzeitparameter:* Die in der Literatur vorgestellten Arbeiten behandeln die Softwareentwicklungsphase und nicht die Phase, in der das parallele Programms tatsächlich ausgeführt wird. *CASEP* bietet dem Entwickler durch die Laufzeitparameterextraktion Informationen, die die Laufzeit des parallelen Programms erheblich optimieren können.

4. *Algorithmenoptimierungen:* Verwandte Arbeit nutzen teilweise Optimierungen im Programmcode bei der Transformation in parallelen Code. Diese sind jedoch allgemein gehalten und nicht speziell auf den zu betrachtenden Algorithmus ausgelegt. *CASEP* verwendet hierfür speziell auf die verwendete Algorithmenklasse angepasste Optimierungen, mit dem Ziel, dadurch eine höhere Leistungsfähigkeit und Genauigkeit des zu parallelisierenden Programms zu erhalten.

5. *Offen: CASEP* beruht fast vollständig auf offenen Standards und offener Software. Dadurch ergibt sich eine potentiell große Nutzerbasis und Transparenz gegenüber den entwickelten Techniken. Einzig der verwendete *CUDA*-Compiler ist proprietär, wobei er sehr gut dokumentiert und frei nutzbar ist. Es wurde aufgrund möglicher Leistungsverluste [80] darauf verzichtet, für den Compiler einen offenen Standard wie *OpenCL* zu verwenden.

6. *Erweiterbarkeit:* Alle in *CASEP* verwendete Module sind erweiterbar und austauschbar. Dies wird sowohl über die modulare Architektur, als auch über die verwendeten offenen Standards bewerkstelligt. Diese Thematik wird in Kapitel 5 ausführlich diskutiert.

7. *Pre-Code Phase:* Bei den untersuchten Arbeiten aus der Literatur bildet der fertige Programmcode den Fokus. *CASEP* kann vom Entwickler teilweise bereits in der *Pre-Code*-Phase verwendet werden. Dies ermöglicht eine Fokussierung auf die potentiell leistungsstärkste Version für das Programm (sequentiell, parallel und auf GPGPUS).

Die aufgezeigten Punkte belegen eine deutliche Abgrenzung zu dem Stand der Technik. Gleichzeitig wird die Relevanz von Werkzeugen für die Generierung von parallelem Programmcode anhand der Vielzahl an Arbeiten, die sich mit dieser Problematik beschäftigen, aufgezeigt.

3.5 Zusammenfassung

Das vorliegende Kapitel hat einen Überblick über die Architektur von *CASEP* und in das eingesetzte Umfeld gegeben. Es wurden die drei Anwendungsgebiete 1) *Unterstützung bei der Programmierung*, 2) *Speedup-Vorhersage* und 3) *Parallelisierung* vorgestellt und *CASEP* in die jeweiligen Gebiete eingeordnet. Weiterhin wurden die Systemgrenzen von *CASEP* diskutiert, mit den sich daraus ergebenden Limitierungen und Chancen. Zuletzt wurden die einzelnen Module von *CASEP* beschrieben und jeweils auf das zu erreichende *Ziel* und die dafür verwendete *Methodik* eingegangen. Im Anschluss wurden die Abgrenzungen dieser Dissertation mit verwandten Arbeiten aus der Literatur aufgezeigt und diskutiert.

Das nachfolgende Kapitel beschreibt die einzelnen Module im Detail. Es werden die genutzten Konzepte, Technologien und implementierte Erweiterungen beschrieben, die zu präziseren Ergebnissen bei der Speedup-Vorhersage und-/oder schnelleren Laufzeiten des zu parallelisierenden Programms führen. Weiterhin werden die entstehenden Synergien der einzelnen Module betrachtet.

4 CASEP Architektur

Inhaltsangabe

Das nachfolgende Kapitel beschreibt die in Kapitel 3 vorgestellte Systemarchitektur im Detail. Dabei wird auf die einzelnen Module, deren Aufbau und deren Einordnung in das Gesamtkonzept eingegangen. Zur besseren Übersicht wurde bei der Beschreibung der Module die farbliche Markierung, die in Abschn. 3.1 eingeführt wurde, verwendet. Diese indiziert das Anwendungsfeld (*Unterstützung bei der Programmierung*, *Speedup-Vorhersage* und *Parallelisierung*) der jeweiligen Module.

Zuerst wird in Abschn. 4.1 die unterste Schicht, das Information Crawling, vorgestellt. Als nächstes folgt die Speedup-Vorhersage-Schicht in Abschn. 4.2. Das Zusammenführen der einzelnen Speedup-Vorhersagen wird in Abschn. 4.3 beschrieben, während der Abschn. 4.4 sich mit der Parallelisierungs-Schicht des Frameworks beschäftigt. Abschließend wird in Abschn. 4.5 das Kapitel zusammengefasst.

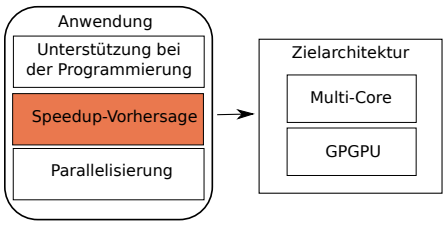
4.1 Information Crawling

Die unterste Schicht des Frameworks ist für das Sammeln von Informationen für die konkrete Speedup-Vorhersage und Parallelisierung durch die oberen Schichten zuständig. Hierzu werden zwei verschiedene Ansätze der Informationsgewinnung unterschieden; vom Benutzer einzugebende Parameter und automatisch ermittelte Informationen.

Auch wenn *CASEP* zum größten Teil automatisiert abläuft, können einige Informationen nicht ohne die Hilfe des Benutzers ermittelt werden. Beispielsweise sind, für die in der zweiten Schicht (Speedup Estimation) stattfindenden Benchmarks, in der Pre-Code-Phase Informationen über die Komplexitätsklassen der zu berechnenden Funktionen notwendig, um die Vorhersage zu bestimmen. Diese Informationen können nicht automatisch ermittelt werden, da zu dem Zeitpunkt der Code-Entwicklung das konkrete Programm nicht implementiert sein muss. Weiterhin muss der Entwickler die Abschnitte im Quellcode markieren, die bestimmte Funktionen von *PBAs* (s. Abschn. 2.2) implementieren, da es aufgrund der Polymorphie von Quellcode nicht möglich ist, die Semantik von Funktionen automatisch zu bestimmen. In der Beschreibung der jeweiligen Module (s. Abschn. 4.2.1, Abschn. 4.2.2 und Abschn. 4.4) wird auf die vom Benutzer notwendigen Informationen im Detail eingegangen.

Außer den notwendigen Benutzereingaben werden von *CASEP* Informationen über die Hardware benötigt, auf der das finale parallelisierte Programm ausgeführt werden soll. Diese Informationen werden zum größten Teil mit Hilfe von API- und Systemaufrufen ermittelt. NVIDIA stellt über CUDA eine Vielzahl von Aufrufen bereit, mit denen sich signifikante Parameter der Grafikkarte extrahieren lassen, wie beispielsweise die Anzahl der Streaming Multiprozessoren oder die Taktrate (s. Abschn. 2.1.2.1).

4.2 Speedup Estimation



In diesem Abschnitt werden die Module zur Bestimmung des Speedups vorgestellt. Grundsätzlich wird zwischen Methoden, die in der Pre-Code-Phase ausgeführt werden, und solchen, die in der Post-Code-Phase durchgeführt werden, unterschieden. *CASEP* verwendet zur Pre-Code-Analyse das Modul Benchmarking, während für die Post-Code-Analyse das Code-Analyse-Modul verwendet wird.

4.2.1 Benchmarking

Das Benchmarking-Modul verwendet *synthetische Benchmarks* [81] [82] zur Bestimmung des zu erwartenden Speedups des zu untersuchenden Programms. Synthetische Benchmarks sind Programme, die das Verhalten von reellen Programmen nachbilden und somit Rückschlüsse auf die Ausführungszeiten von Klassen von Programmen erlauben. Anders als Anwendungsbenchmarks [83] [84], mit denen sich *Echtweltprobleme* abbilden lassen, können mit synthetischen Benchmarks einzelne Abschnitte des Quellcodes genauer untersucht werden. Die Benchmarks werden dabei so allgemein wie möglich gehalten, um eine möglichst große Menge von Programmen abbilden zu können, ohne das charakteristische Verhalten der Programme, in diesem Fall von *PBAs*, zu verlieren.

Eine schematische Darstellung des in *CASEP* verwendeten Benchmarking-Moduls ist in Abbildung 4.1 dargestellt. Dieses besteht aus zwei Teilmodulen, dem *Benchmark Creator* und dem *Executer*. Der Benchmark Creator (BC) ist dafür zuständig die Benchmarks zu erzeugen. Hierzu werden vom Benutzer Eingaben benötigt, die dabei helfen die Benchmarks an das konkrete Programm anzupassen und sie genauer zu machen. Für jeden der unterstützten

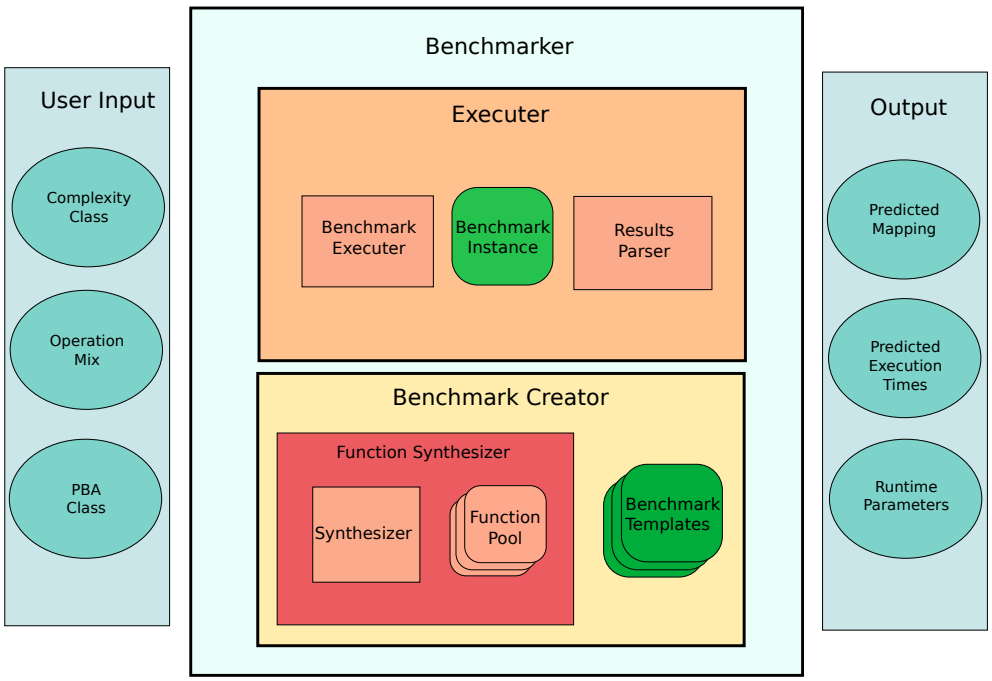


Abbildung 4.1: Schematische Darstellung des Benchmarkers

PBAs existiert ein eigenständiges Template, das die Grundstruktur des *PBAs* abbildet. Diese Templates werden dazu verwendet, eine konkrete Instanz zu erzeugen, indem vom Benutzer Eingaben zu dem Problem, welches er lösen möchte, abgefragt werden. Hierzu zählen der zu verwendete *PBA* (PBA-Class), aber auch die Charakteristika der zu lösenden Funktionen, wie die Art der Operationen und die Komplexitätsklasse. Mit Hilfe dieser Informationen werden die Templates durch den *Function Synthesizer* angepasst, welcher nachfolgend ausführlich erläutert wird. Die vom BC erzeugten Template-Instanzen werden dem Executer-Modul übergeben, der die Benchmarks mit unterschiedlichen, für *PBAs* charakteristischen, Parametern ausführt.

Eine der signifikanten Parameter ist die Populationsgröße des *PBAs*. Üblicherweise lohnt sich eine Parallelisierung erst ab einer bestimmten Berechnungsgröße, hier Populationsgröße. Als Ausgabe erhält man geschätzte Ausführungszeiten für die serielle Maschine und die betrachteten parallelen Zielarchitekturen, wie in Abb. 4.2 beispielhaft dargestellt. Es existieren somit vier Abschnitte. Der erste ist der Bereich bis zum Schnittpunkt *a*. In die-

sem Bereich ist die serielle Ausführung die schnellste. Der zweite Bereich ist zwischen dem Schnittpunkt a und b. In diesem Bereich liefert die parallele Version auf der Multi-Core CPU die kürzeste Ausführungszeit. Im dritten Bereich (zwischen b und c) ist sowohl die GPGPU-Ausführung als auch die Multi-Core-Ausführung des Programms schneller als die serielle Ausführung. Für diesen Bereich ist meistens eine genaue Festlegung auf die Zielarchitektur, aufgrund der Nähe der Ausführungszeiten der parallelen Versionen, nicht möglich. Im letzte Bereich (ab Schnittpunkt c) erzeugt die GPGPU-Version die kürzeste Ausführungszeit des Programms. Die hier dargestellten Kurven

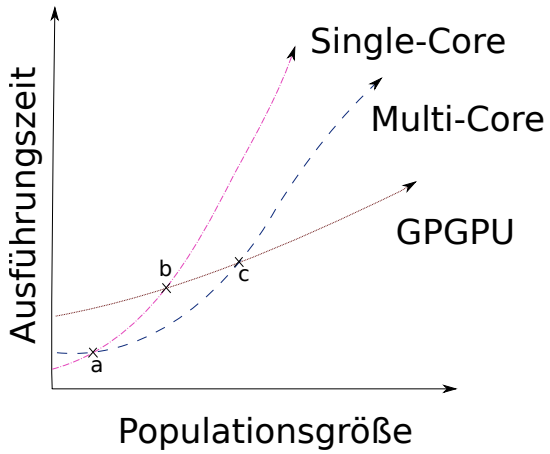


Abbildung 4.2: Vergleich der Ausführungszeiten

beruhen auf Ergebnisse, der im Rahmen dieser Arbeit stattgefundenen Simulationen (s. Abschn. 6). Üblicherweise lohnt sich eine Parallelisierung erst ab einer bestimmten Populationsgröße. Je größer die Population gewählt wird, desto größer ist die Wahrscheinlichkeit, dass die Ausführung auf der GPGPU schneller ist, als die auf einer Multi-Core CPU. Der Benchmarker hilft hierbei den richtigen Schnittpunkt in Abhängigkeit von Parametern, wie beispielsweise der Populationsgröße, zu ermitteln und somit das beste Mapping vorherzusagen. Ein weiterer Parameter, welcher bei der Ausführungszeit von *PBAs* ebenfalls eine gewichtige Rolle spielt, ist die Anzahl der von Algorithmus durchgeführten Iterationen, also wie oft iteriert werden muss, bis das gewünschte Ergebnis erreicht ist (s. Abschn. 2.2). Auch hier werden, ähnlich wie bei der Populationsgröße, unterschiedliche Iterationszahlen vom Benchmarker unter-

sucht und die jeweiligen Ausführungszeiten in Relation zueinander gesetzt. Bei den CUDA Benchmarks werden außerdem verschiedene Thread- und Blockgrößen (s. Abschn. 2.1.2.1) bei der Ausführung kombiniert, um die optimale Ausführungskonfiguration zu bestimmen.

Für jede der oben aufgeführten Konfigurationen werden m Simulationen durchgeführt, um mögliche Messabweichungen zu minimieren. Somit ergeben sich $\prod_{i=0}^{n} |\alpha_i|$ unterschiedliche Kombinationen, wobei α_i der jeweilige Parameter (z.B. Popoluationsgröße, Problemgröße, ...) ist und $|\alpha_i|$ die Menge der möglichen Werte, die angenommen werden können (z.B. 30, 100, ...). Durch die $m-$fache Durchführung der Simulationen erhält man somit $m * (\prod_{i=0}^{n} |\alpha_i|)$ unterschiedliche Benchmarks, die vom Executer-Modul durchgeführt werden.

Die ermittelten Werte werden im letzten Schritt an das *Results Parser*-Modul übergeben, wo sie ausgewertet werden. Dieser ermittelt für jede Populationsgröße und Iterationszahl jeweils die beste Kombination aus Thread- und Blockgröße. Außerdem werden die kürzesten Ausführungszeiten der verschiedenen Architekturlösungen miteinander verglichen, um zu bestimmen, auf welcher Hardware (GPGPU, CPU, Multi-Core CPU) das Programm am schnellsten durchgeführt werden kann. Daraus ergibt sich eine Ausführungszeit, die berechnet wird aus:

$$\begin{aligned} t_{exec} = t_{alloc} + Iterations * (\sum_{i}(t_{mean}PBA\_Operation_i) + \\ \sum_{j}(t_{CPU} \rightarrow t_{GPGPU}) + \sum_{k}(t_{GPGPU} \rightarrow t_{CPU})) \end{aligned} \tag{4.1}$$

Dabei ist t_{alloc} die Zeit, die benötigt wird, um den Speicherplatz für die Variablen auf der Hardware zu allozieren und $t_{mean}PBA\_Operation_i$ die durchschnittliche Zeit, die benötigt wird, um eine *PBA* Operation auf der jeweiligen Hardware durchzuführen. Dies kann beispielsweise die durchschnittliche Zeit für die *Crossover*-Operation (s. Abschn. 2.2) sein. Die Kopierzeiten der Daten von der CPU zur GPU werden mit $\sum_{j}(t_{CPU} \rightarrow t_{GPGPU})$ dargestellt, während das Kopieren zurück auf die CPU durch $\sum_{k}(t_{GPGPU} \rightarrow t_{CPU})$ dargestellt wird.

Sollten alle *PBA* Operationen nur auf der CPU durchgeführt werden, entfallen die Kopierkosten ($j = k = 0$). Ähnlich ist es bei Ausführung aller Operationen auf der GPGPU, hier müssen die Kopieroperationen nur einmal durchgeführt werden ($j = k = 1$). Handelt es sich um eine hybride Implementierung, also eine Implementierung in der Teile des Algorithmus auf unterschiedlicher Hardware laufen, müssen die Daten mehrmals kopiert werden ($j > 1, k > 1$).

Der Entwickler erhält dementsprechend drei Ausgabewerte vom Benchmarker: Die ermittelten Ausführungszeiten auf den einzelnen Architekturen, das daraus abgeleitete beste Mapping der Software auf die Hardware und die optimale Thread- und Block-Konfiguration für die GPGPU. Im Folgenden wird auf einige Aspekte des Benchmarkers detaillierter eingegangen.

Funktionssynthese: Die wichtigste Komponente bei den hier aufgezeigten Benchmarks sind die zu lösenden Funktionen. Die Funktionen sind die vom Algorithmus zu berechnenden Probleme (z.B. Fitnessfunktion bei *PBAs*). Zum Zeitpunkt der Benchmarks sind diese Funktionen jedoch meist nicht genau bestimmbar, bzw. können sich im Laufe der Entwicklung verändern. Das Funktionssynthese Modul erzeugt aus den vorhandenen Informationen, die der Benutzer *CASEP* zur Verfügung stellt, eine Funktion, die möglichst nahe an der Funktion des späteren Programms ist. Hierzu werden zwei Metriken verwendet, die eine Funktion charakterisieren, die Komplexitätsklasse ($\mathcal{O}(n)$, $\mathcal{O}(n^2)$, ...) und die Art der Operationen (Multiplikation, Kosinus, ...), die die Funktion ausführt. Der Synthesizer nutzt die übergebenen Informationen zusammen mit einem Pool aus Funktionen, um daraus die zu simulierende Funktion zu erstellen. *CASEP* bedient sich hierzu bekannter Benchmark Funktionen aus der Literatur [85], die eine große Bandbreite an unterschiedlichen Problemen abbilden.

In Abb. 4.3 ist die Vorgehensweise des Synthesizer-Moduls schematisch dargestellt. Dieses benötigt drei Eingaben, die vom Benutzer oder dem *CASEP* Framework bereitgestellt werden: *Complexity class* (Entwickler), *functioons pool* (CASEP) und *operations mix* (Entwickler) und liefert als Ausgabe eine an das tatsächliche Problem angepasste Funktion (*adjusted function*). Die wichtigste Information für die Erstellung einer genauen Benchmark-Funktion ist

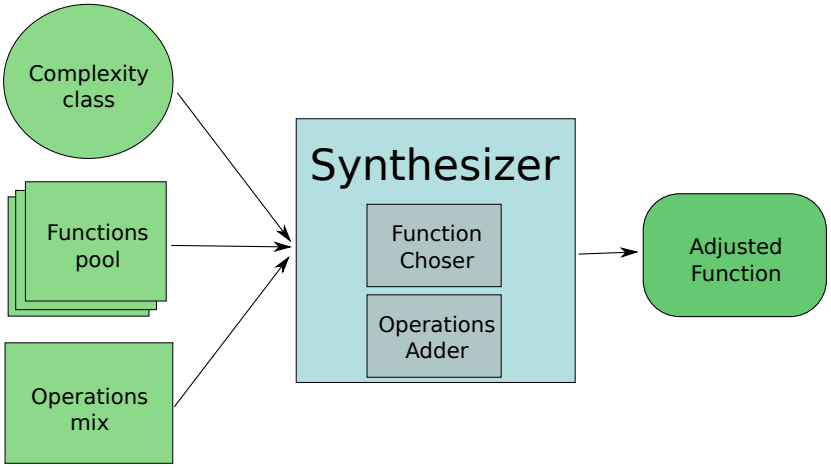


Abbildung 4.3: Function Synthesizer

die Komplexitätsklasse der zu berechnenden Funktion im reellen Programm, diese wird vom Entwickler in der $\mathcal{O}$ Notation angegeben, z.B. $\mathcal{O}(n^2)$ (quadratische Komplexität). Diese Information ist jedoch nicht ausreichend um die Berechnungskomplexität einer Funktion auf der Hardware zu definieren. Programmtechnisch gesehen ergibt sich die Komplexität über die Schleifenverschachtelung, während die elementare Operationen in $\mathcal{O}(1)$ berechnet werden können. Somit ist es irrelevant, welche und wie viele Operationen genau in den einzelnen Funktionen berechnet werden. Diese Annahme ist jedoch für die Ausführung von Operationen auf reeller Hardware nicht korrekt. Ungleiche Operationen haben auf unterschiedlicher Hardware unterschiedliche Laufzeiten. Insbesondere besitzen GPGPUs spezielle Kerne, die für bestimmte Arten von Funktionen optimiert sind (SFUs, s. Abschn. 2.1.2). Diese Kerne können bestimmte Operationen in einem Zyklus berechnen, sind jedoch nur in einer begrenzten Menge vorhanden. Daraus ergibt sich die Notwendigkeit, Funktionen nicht nur anhand ihrer Komplexitätsklasse, sondern auch anhand der durchgeführten Operationen zu klassifizieren. Weiterhin gibt die Komplexitätsklasse keinen Aufschluss darüber, wie viele Operationen tatsächlich in einer Funktion ausgeführt werden. Zwei Funktionen einer Klasse können eine unterschiedliche Anzahl an Operationen berechnen, was sich in einer unterschiedlichen Laufzeit auf einer reellen Maschine bemerkbar macht. Beispielsweise liegen die

Funktionen $f_1(x) = \sum_{i=0}^{n-1}(\sum_{j=0}^{i} x_i)^2$ und $f_2(x) = \sum_{i=1}^{n}\sum_{i=1}^{n} x_i$ in der gleichen Komplexitätsklasse $\mathcal{O}(n^2)$, die Funktion $f_1(x)$ führt jedoch $\frac{n*(n+1)}{2}$ Operationen durch, während Funktion $f_2(x)$ n^2 Operationen ausführt.

Die notwendige Angleichung der Benchmark-Funktion mit der reellen Funktion erfolgt beim Function Synthesizer durch eine Modifikation der Benchmark-Funktion mit vom Entwickler bereitgestellten Operationen. Auch wenn zum Zeitpunkt des Benchmarkers das reelle Programm und somit die zu berechnenden Funktionen nicht bekannt sind, sind möglicherweise die Art der Operationen bekannt, also ob beispielsweise trigonometrische Funktionen im Algorithmus vorkommen. Die Prozedur des Synthesizers sieht somit vier Teilschritte vor:

1. Abfrage der Komplexitätsklasse
2. Abfrage der zu unterstützenden Operationen (z.B. Kosinus und Multiplikation)
3. Wahl einer Funktion aus dem Funktionspool (z.B. $f(\vec{x}) = \sum_{i=0}^{n}(x_i^2)$)
4. Zusammenführung der Operationen und der gewählten Funktion

Zuerst werden vom Entwickler Informationen über die zu erwartende Komplexitätsklasse und den zu unterstützenden Operationen als Eingabe ausgewertet. Diese Informationen werden als nächstes dazu genutzt, um aus dem bereitgestellten Pool an Benchmark-Funktionen, die zu der Komplexitätsklasse der zu berechnenden Funktion im reellen Programm entsprechende Funktion auszuwählen. Im letzten Schritt werden die Eingabeoperationen zu der ausgewählten Operation hinzugefügt. Dadurch ergibt sich eine angepasste Funktion, die der simulierten Funktion ähnlicher ist, als die ursprünglich aus dem Pool der Benchmark Funktionen ausgewählte Funktion.

Extraktion von Ausführungsparametern: Die Ausführung von parallelen Programmen hängt von vielen Faktoren ab. Insbesondere für Many-Core-Architekturen stellt die Aufteilung des Programms auf die einzelnen

Threads dabei eine der größten Herausforderungen für den Programmierer dar. Auf GPGPUs werden die einzelnen parallel auszuführenden Kernel in Thread-Blöcken mit jeweils einer bestimmten Anzahl Threads aufgeteilt (s. Abschn. 2.1.2.1). Die einzelnen SMs (s. Abschn. 2.1.2) bearbeiten in der Regel 32 Threads (1 Warp) gleichzeitig. Dieser Wert ist architekturabhängig und spiegelt aktuelle GPGPUs von NVIDIA wieder. Sollten die Daten für den aktuellen Wert noch nicht zugreifbar sein, wird ein anderer Warp ausgeführt. Dies führt dazu, dass je nach Problemstellung, also je nach Art und Größe des Problems, eine unterschiedliche Konfiguration von Anzahl der Threads pro Thread-Block zu optimalen Ausführungszeiten führt. Dieser Wert ist nicht trivial bestimmbar und muss vom Entwickler anhand von Erfahrungswerten festgelegt werden.

Das *Benchmarking*-Modul stellt die optimalen Ausführungsparameter für das zu untersuchende Programm bereit, indem die Benchmarks mit unterschiedlichen Parametern ausgeführt werden. Dabei wird für jeden möglichen Algorithmusparameter, also beispielsweise Populationsgröße oder Anzahl an Iterationen, jede mögliche Kombination aus einem vorher definierten Pool von Ausführungsparametern, wie Thread-Anzahl und Block-Größe, ein Benchmark durchgeführt. Anhand der gemessenen Ausführungszeiten dieser Benchmarks wird die beste Kombination aus Ausführungsparametern gewählt.

4.2.2 Code-Analysis

Anders als das in Abschn. 4.2.1 vorgestellte *Benchmarking*-Modul, ist das *Code-Analyse*-Modul dem Kontext der *Post-Code*-Phase einzuordnen, d.h. zum Zeitpunkt der Code-Analyse existiert bereits die serielle Version des zu untersuchenden Programms. Dies ermöglicht genauere Vorhersagen, da bestimmte Parameter fest definiert sind und nicht interpoliert werden müssen. Bei der Code-Analyse werden die Operationen und Strukturen im Quellcode analysiert und ein Ausführungsprofil für die vorhandene Zielhardware erstellt. Dabei wird zwischen statischer und dynamischer Code-Analyse unterschieden. Bei der statischen Code-Analyse werden nur Parameter verwendet, die zur Compile-Zeit feststehen, während die dynamische Code-Analyse das Laufzeit-

verhalten des Programms analysiert und somit für die Analyse der Laufzeitparameter zuständig ist.

In Abb. 4.4 ist eine schematische Darstellung des *Code-Analysis*-Moduls dargestellt. Der zu überprüfende Quellcode wird zuerst vom *Analyser* geparst und ausgewertet. Danach folgt die dynamische Code-Analyse mit Hilfe des *Exploitation*-Moduls, welches Laufzeitparameter bestimmt. Diese Informationen werden dem nächsten Modul, dem *Benchmarker*, übergeben. Der *Benchmarker* ist dafür zuständig, aus den vorhandenen Informationen Micro-Benchmarks zu erstellen, die den Quellcode repräsentieren. Diese Benchmark-Instanzen werden dem letzten Modul übergeben (*Speedup Predictor*), welcher die Benchmarks ausführt und die Ergebnisse der einzelnen Messungen vergleicht. Als Ausgabedateien werden sowohl der erzielte Speedup einzelner Funktionen, als auch der Gesamt-Speedup des Programms ausgegeben. Nachfolgend werden die Module im Einzelnen genauer beschrieben.

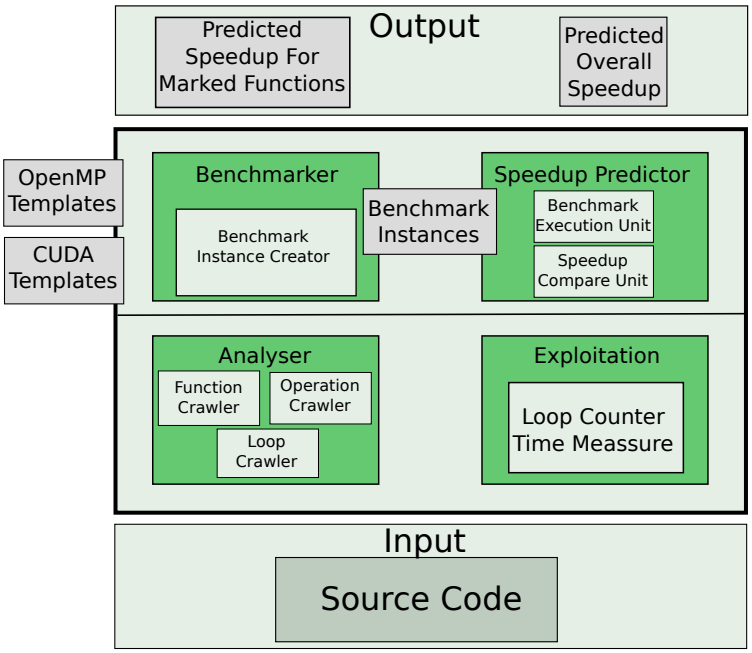


Abbildung 4.4: Code-Analysis Modul

Analyser: Das *Analyser*-Modul parst den Quellcode und unterteilt das Programm in Teilabschnitte, die von den anderen Modulen weiterverarbeitet wer-

den können. Diese Module können Schleifen, Funktionen oder andere definierte Konstrukte im Quellcode sein. Aufgrund der Fokussierung dieser Arbeit auf *PBAs* wurde insbesondere Wert auf Schleifen gelegt. Dies ergibt sich aus dem iterativem Ausführen der PBA-Operationen, die den Großteil der Berechnungszeit ausmachen. Dementsprechend ist der *Loop Crawler* für die Identifizierung und Speicherung von Schleifen im Programm zuständig. Nach dessen Ausführung werden die darin befindlichen Operationen durch den *Operation Crawler* gespeichert und anhand ihrer Datentypen (z.B. float, double, ...) und Operationsarten (Kosinus, Summe, ...) gruppiert. Weiterhin werden rekursive Aufrufe von Funktionen mit Hilfe des *Function Crawlers* berücksichtigt, womit die Überprüfung von Verschachtelungen im Quellcode möglich ist.

Exploitation: Die Ausführung von Programmen hängt von vielen Faktoren ab, die teilweise zur Compilezeit nicht bekannt sind. Ein bekanntes Beispiel ist die Laufzeit von Schleifen, die an eine dynamischen Abbruchbedingung (z.B. Ausführung nur, bis ein bestimmter Fitnesswert erreicht wurde) gekoppelt sind. Solche Schleifen sind bei *PBAs* häufig anzutreffen. Zur Bestimmung der Ausführungszeiten eines Programms ist jedoch die genaue Iterationszahl von Schleifen notwendig. Diese Informationen werden vom *Exploitation*-Modul heuristisch bereitgestellt, indem eine dynamische Code-Analyse durchgeführt wird.

Bei der dynamischen Code-Analyse wird jede Schleife im seriellen Ursprungsprogramm mit Schleifenzählern versehen. Darauf folgend wird das Programm m fach ausgeführt, um eine durchschnittliche Laufzeit der Schleifen zu ermitteln. Die so ermittelten Informationen werden dem Benchmarker zur Erstellung der Micro-Benchmarks übergeben.

Funktionen mit großem Störfaktor oder vielen lokalen Minima bzw. Maxima führen oft zu stark abweichenden Ausführungszeiten des Programms. Diese Abweichungen werden berücksichtigt, indem das Exploitation-Modul den Bereich der Abweichung (*Deviation Area*) ermittelt, wie in Abb. 4.5 dargestellt. In der Abbildung ist der Bereich der seriellen, als auch der parallelen Ausführung abgebildet. Indem die besten und schlechtesten Prognosen für die Ausführungszeiten betrachtet werden, erhält man drei verschiedene Bereiche.

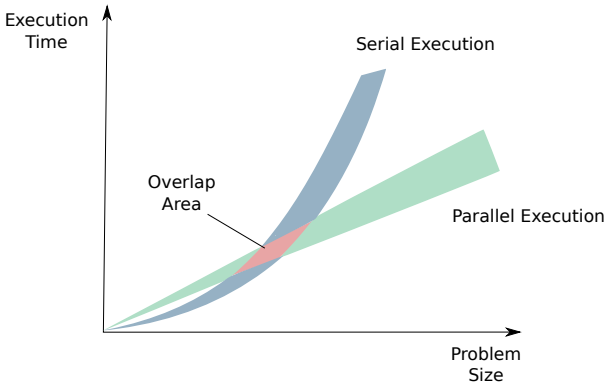


Abbildung 4.5: Deviation Area

Einen Bereich, in dem die serielle Version die besten Ergebnisse liefert, einen Bereich, bei dem die parallele Version am schnellsten ausgeführt wird und einen Überlappungsbereich, in dem keine genaue Vorhersage möglich ist. Das Exploitation-Modul ermittelt für die weitere Berechnung des Speedups die durchschnittlichen Ausführungszeiten der einzelnen Versionen.

Benchmarker: Die vom Analyser- und Exploitation-Modul ermittelten Daten werden vom Benchmarker-Modul verwendet, um Micro-Benchmarks zu erstellen, die zur Bestimmung des möglichen Speedups einer parallelen Version des Programms dienen. Für jede Schleife im Quellcode wird ein Micro-Benchmark erstellt. Der Benchmarker nutzt hierzu spezielle Templates für die einzelnen *PBAs*. Dieser Ansatz hat gegenüber einer automatisierten Parallelisierung mit beispielsweise OpenMPC [72] Vorteile, da bestimmte Charakteristika von *PBAs* manuell implementiert und optimiert werden können, was zu genaueren Ergebnissen der Benchmarks führt. Manuell transformierter Code kann beispielsweise die vorhandene Speicherarchitektur besser ausnutzen oder, wie weiter unten im Abschnitt gezeigt wird, optimiert werden, um Cache-Effekte zu vermeiden, die zu einem ungenauen Ergebnis führen würden. Außerdem hat die Arbeit mit dem in *CASEP* entwickelten Parallelisierungswerkzeug (s. Abschn. 4.4) gezeigt, dass automatisierte Tools zu Fehlern im transformierten Quellcode führen können, was die Ergebnisse weiter verfälschen würde.

Die Erstellung der Micro-Benchmarks ist in zwei Teilschritten aufgeteilt:

Zuerst wird für jede gefundene Schleife ein OpenMP (s. Abschn. 2.1.1.1) Benchmark erstellt. Diese Prozedur ist relativ simpel durchzuführen, indem im Quellcode OpenMP Pragma-Direktiven eingefügt werden und über die Threads iteriert wird, statt über eine Laufvariable, wie bei seriellen Programmen üblich. Als nächstes werden aus den OpenMP-Benchmarks CUDA-Benchmarks erstellt. Hierzu muss lediglich der Schleifenkörper in die speziell angepassten Templates eingefügt werden. Die benötigten Variablen werden automatisiert dem Benchmarker vom Analyser-Modul übergeben und in das Template eingefügt. Ebenso erfolgt die Initialisierung und das Kopieren der nötigen Dateien automatisiert, indem die Daten in die dafür zuständigen Bereiche des Templates kopiert werden.

Das Verfahren wird anhand eines Minimalbeispiels genauer erklärt: Im Listing 4.1 ist eine einfache Schleife abgebildet, die vom Analyser extrahiert wird.

```
1 float a[iterationCount];
2 for(int i = 0; i < iterationCount; i++){
3 a[i] = cos(a[i]);
4 }
```

Listing 4.1: Serielle Schleife

Die Schleife wird zunächst in OpenMP Code transformiert, wobei die Variablen in eine für den OpenMP-zu-CUDA-Übersetzer lesbare Form umgewandelt werden. Weiterhin werden OpenMP Pragmata eingefügt, die für die parallele Ausführung notwendig sind, und die Laufzeitvariablen durch die jeweiligen Thread-IDs ersetzen (idx). Der transformierte Quellcode ist in Listing 4.2 abgebildet.

```
1 float f32Array[iterationCount];
2 #pragma omp parallel for
3 for(int idx=0; idx < iterationCount; idx++){
4 a[idx] = cos(a[idx]);
5 }
```

Listing 4.2: OpenMP Schleife

Diese Transformation erfolgt komplett automatisch für definierte Datentypen. Für die hier vorliegende Arbeit wurden die gängigsten Datentypen wie float- oder double-Arrays implementiert, jedoch ist das Framework auch auf weitere Datentypen erweiterbar. Zuletzt wird der OpenMP Benchmark in ein CUDA-Benchmark umgewandelt. Der so transformierte Quellcode ist in Listing 4.3 dargestellt. Der *Benchmarker* kommentiert die nicht benötigten *defines* im CUDA-Template automatisch aus, hierzu werden die vom Analyser gesammelten Informationen verwendet. Alle nicht benötigten Variablen im Funktions-Header werden zu *null*-Pointern und beeinflussen den Programmablauf nicht. Diese Herangehensweise ist notwendig, um die Templates so generisch wie möglich zu halten. So kann eine möglichst große Menge an Eingabeprogrammen abgedeckt werden. Die Variablen-Deklaration und Kopieroperationen sind in diesem Listing nicht abgebildet. Die vollständigen Templates für OpenMP und CUDA können im Anhang C nachvollzogen werden.

```
1  ...
2  //#define USE_INT64
3  #define USE_FLOAT32
4  //#define USE_FLOAT64
5  ...
6  __global__ void benchmark(int* i32Array, long long int* i64Array,
7      float* f32Array, double* f64Array) {
8    unsigned long long blockID = blockIdx.x + blockIdx.y * gridDim.x
         ;
9    unsigned long long idx = blockID * blockDim.x + threadIdx.x;
10   if (idx < iterationCount) {
11     a[idx] = cos(a[idx]);
12   };
13 }
```

Listing 4.3: CUDA Schleife

Speedup Predictor: Das Modul *Speedup Predictor* führt die Benchmark-Instanzen mehrfach aus. Die Ausführungszeit τ wird bestimmt, indem die Ausführungszeiten der einzelnen Operationen op_i und die Kopierzeiten der

Daten op_{copy}, wie in Formel 4.2 gezeigt, aufsummiert werden.

$$\tau = (\sum_{i=1}^{n} op_i + op_{copy}) + CUDA_{init} \tag{4.2}$$

Die Ausführungszeit wird außerdem von der anfänglichen Initialisierungsphase der CUDA Bibliothek $CUDA_{init}$ beeinflusst, welche nur einmal für die gesamte Ausführungsphase einbezogen werden muss.

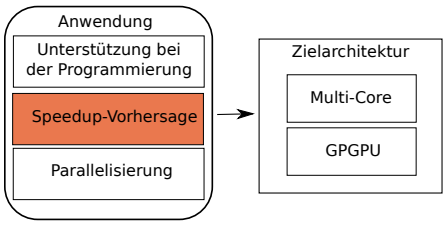
In den meisten Fällen ist es nicht sinnvoll, das gesamte Programm zu parallelisieren, sondern nur diejenigen Abschnitte, die den größten Teil der Berechnungszeit beanspruchen. In diesem Fall müssen die Kopierzeiten zwischen den einzelnen Modulen im Quellcode berücksichtigt werden. Wird beispielsweise bei einem *PBA* mit n unterschiedlichen Modulen ein Modul auf der GPGPU ausgelagert und die restlichen Module auf der CPU ausgeführt, müssen in jeder Iteration die Daten für das Modul auf die GPGPU transferiert werden ($op_{copyToDevice}$).Dabei ist ein Modul eine Operation eines *PBAs* (Mutation, Fitnessberechnung ...). Am Ende des Moduls müssen die Daten wieder zurück auf die CPU transferiert werden ($op_{copyToHost}$). Daraus ergibt sich die optimierte Formel 4.3 für die Ausführungszeit, bei der die einzelnen Kopieroperationen berücksichtigt werden. α und β indizieren dabei, wie oft die Kopieroperationen ausgeführt werden müssen.

$$\tau = (\sum_{i=1}^{n} op_i + op_{copy} + \alpha \cdot op_{copyToDevice} + \beta \cdot op_{copyToHost}) + CUDA_{init} \tag{4.3}$$

Verhinderung von Cache-Effekten: Die von der Code-Analyse erstellten Micro-Benchmarks nutzen für die Ausführung der Operationen zuvor adressierte Daten (die zu berechnenden Werte). Aufgrund der einfachen Struktur der Micro-Benchmarks, kann es zu dem Effekt kommen, dass die Operationen und Daten komplett im Cache gespeichert werden und somit zu kurzen Zugriffszeiten führen, die die Messungen der Micro-Benchmarks verfälschen. Um diese sogenannte *immediate* Adressierung zu umgehen, wurde für die Code-Analyse ein Verfahren entwickelt, welches einer immediate Adressie-

rung entgegenwirkt. Es wird zunächst ein Array erstellt, welches die Größe $Populationsgröße * Dimensionsgröße * Anzahl\ Benchmarkoperationen$ hat. Im nächsten Schritt wird das Array mit Zufallszahlen gefüllt. Bei jedem Aufruf einer Operation im Micro-Benchmark, wird nun jeweils ein Wert aus dem Array verwendet. Dies hat zwar zur Folge, dass sich die Zufallszahlen nach jeder Iteration wiederholen. Dieser Effekt ist jedoch irrelevant für die Benchmarks, da bei der Ausführung die Güte der berechneten Werte keine Rolle spielt, sondern nur die hierzu benötigte Zeit. Durch das Verwenden der Zufallszahlen aus dem Array wird eine immediate Kodierung der Operanden verhindert und Cache-Effekte können nicht auftreten.

4.3 Mapping



In der *Mapping*-Schicht werden die Vorhersage der einzelnen Module zu einer Gesamtvorhersage aggregiert. Hierzu muss zuerst aus den einzelnen Modulen der Speedup für das Gesamtprogramm extrahiert werden. Beim *Benchmarker* muss hierzu nichts weiter berechnet werden. Die erzeugten Benchmarks bilden das komplette Programm ab und ermitteln somit die Gesamtausführung des Programms. Beim *Code-Analysis*-Modul werden jedoch einzig für die vom Entwickler zu parallelisierenden Bereiche Micro-Benchmarks erstellt. Es wird somit einzig der Speedup für diese Bereiche berechnet. Um den Gesamtspeedup des Programms zu berechnen, muss der Speedup der einzelnen Bereiche in Relation gesetzt werden. Hierzu wird die Gesamtausführungszeit des seriellen Programms (T_R) und die serielle Ausführungszeit der Abschnitte, die vom Entwickler markiert wurden (t_R) gemessen. Die serielle Ausführungszeit, ohne die markierten Bereiche beträgt dementsprechend:

$$T_{Rest} = T_R - t_R$$

Als nächstes werden die Ausführungszeiten der Micro-Benchmarks für die markierten Bereiche gemessen. Daraus ergeben sich drei Ausführungszeiten: 1) die vorausgesagte serielle Ausführungszeit t_S, 2) die vorausgesagte OpenMP-Ausführungszeit t_{OMP} und 3) die vorausgesagte CUDA-Ausführungszeit t_{CUDA}. Um den Gesamtspeedup zu bestimmen, muss zur Restausführungszeit (serielle Ausführungszeit ohne markierte Bereiche) der zu erwartende Speedup addiert werden:

$$Speedup = T_{Rest} + \frac{t_R}{S_R}$$

Dabei ist S_R der zu erwartende Speedup für eine der zwei beschriebenen Implementierungen (OpenMP oder CUDA) und berechnet sich aus:

$$S_R = \frac{t_S}{t_{parallel}}$$

Wobei $t_{parallel}$ entweder die t_{OMP} oder t_{CUDA} ist.

Die beiden berechneten Speedup-Werte müssen durch das *Mapping*-Modul im nächsten Schritt gewichtet werden. Hierzu wurde multilineare Regression [86] verwendet. Bei der multilinearen Regression werden Messwerte (in dem Fall die Vorhersagen des *Benchmarkers* und der Code-Analyse) in einem n-dimensionalen Raum (in diesem Fall 2-dimensional) verteilt, wobei der Wert auf der Y-Achse durch den gewünschten Wert dargestellt wird (in diesem Fall der reale Speedup). Als nächstes wird eine Ebene durch die Punkte gelegt, mit dem Ziel, den durchschnittlichen Fehler (Quadratsumme) der einzelnen Punkte zur Ebene zu minimieren. Daraus ergibt sich eine Gewichtung der Punkte. Diese Gewichtung wird dazu verwendet, die beiden Vorhersagemethoden zu aggregieren. Die hierzu notwendigen Messpunkte leiten sich aus den Messungen in der Evaluation ab (s. Abschn. 6.3). Durch die Regression soll folgende Formel gelöst werden:

$$\vec{Y} = \alpha * \vec{x_1} + \beta * \vec{x_2}$$

Dabei ist $\vec{Y}$ ein Vektor mit den gewünschten Werten (in diesem Fall die realen Speedup-Werte), $\vec{x_1}$ die vorhergesagten Speedup-Werte des *Benchmarkers* und $\vec{x_2}$ die vorhergesagten Speedup-Werte des *Code-Analysis*-Moduls. Die Werte α und β sind die Gewichtungen, die mit Hilfe der Regression berechnet werden

sollen. Die sich daraus ergebende Regressionsebene ist exemplarisch in Abb. 4.6 dargestellt.

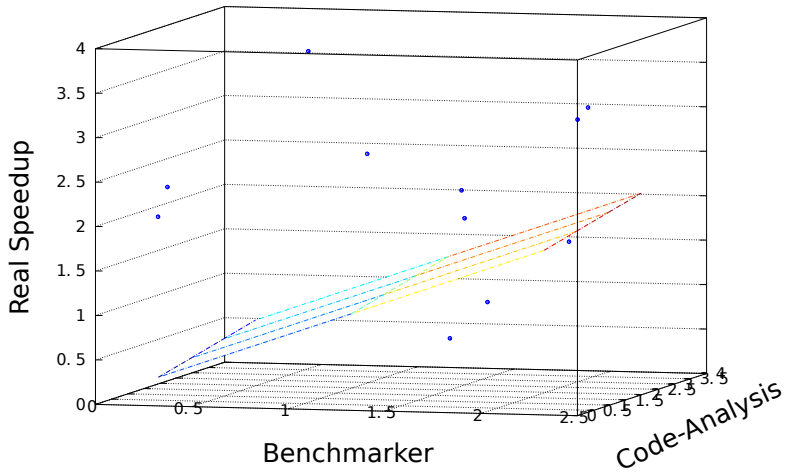
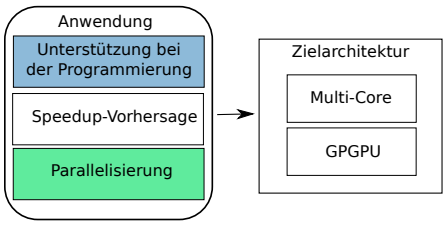


Abbildung 4.6: Beispiel einer multilinearen Regression

Das *Mapping*-Modul verwendet die so berechneten Gewichtungen, um aus den beiden Vorhersagen eine gemeinsame Vorhersage zu erstellen. Bei möglichen Erweiterungen von *CASEP* können zusätzliche Vorhersagemodule integriert werden, indem die Regressionsebene für die hinzugekommenen Messpunkte neu berechnet wird.

4.4 Parallelization



Das vom *Mapping*-Modul ermittelte beste Mapping des seriellen Programms auf die vorhandene parallele und serielle Hardware wird schließlich an das letzte Modul (*Parallelization*) übermittelt. Hier werden die jeweiligen Code-Abschnitte teilautomatisiert zu parallelem Quellcode umgewandelt.

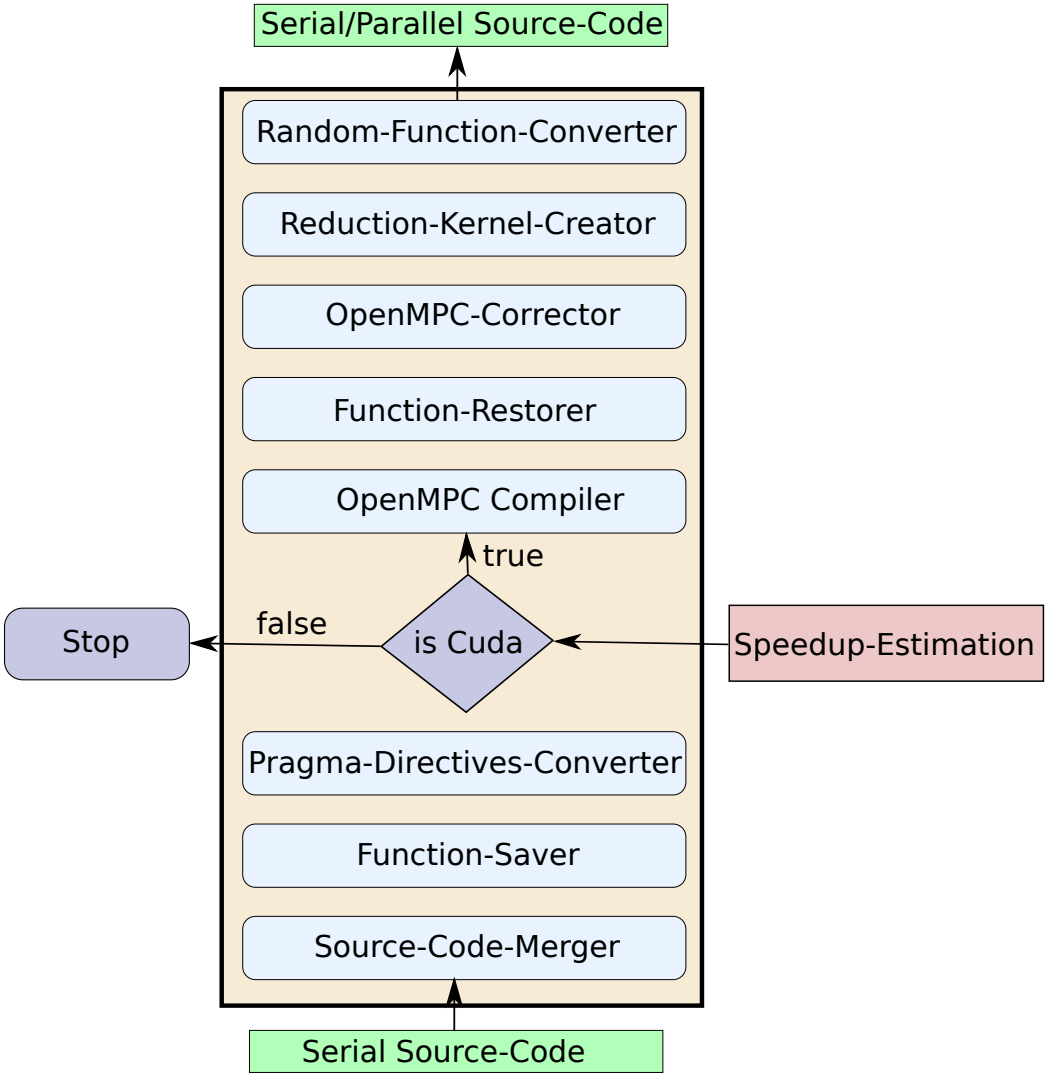


Abbildung 4.7: PBA2CUDA Framework

In Abb. 4.7 ist der prinzipielle Aufbau vom *Population Based Algorithm to CUDA* (PBA2CUDA) Moduls abgebildet. Der serielle Quellcode dient als Eingabe. Zuerst werden die einzelnen Dateien des Quellcodes in eine Datei zusammengeführt. Dies ist notwendig, da das Programm sonst nicht von den genutzten Tools weiterverarbeitet werden kann. Als nächstes werden im *Function-Saver* die Funktionsdeklarationen gespeichert, da sie bei der Transformation durch den verwendeten Compiler gelöscht werden können. Das nächste Teilmodul, der *Pragma-Directives-Converter*, wandelt vereinfachte Parallelisierungsdirektiven in vom Transformationswerkzeug lesbare Direktiven um. Dieser Schritt dient der Vereinfachung, sowohl für den Entwickler, als auch für die Nutzung von *PBA2CUDA* im übergeordneten *CASEP* Framwork. Als nächstes wird die Speedup-Vorhersage dazu genutzt, zu bestimmen, welche Teilabschnitte im Quellcode tatsächlich parallelisiert werden sollen (*is cuda*=true). Diese werden von einem externen Compiler, dem *OpenMPC Compiler*, in parallelen Code umgewandelt. Die zuvor gespeicherten Funktionen werden im nächsten Schritt wiederhergestellt. Mögliche Fehler, die bei der Umwandlung entstehen können, werden im folgenden Teilmodul, dem *OpenMPC-Corrector*, korrigiert. Ein möglicher Fehler ist die falsche Markierung von Funktionen als *host*-Funktionen (also Funktionen, die auf der CPU ausgeführt werden), obwohl sie auf der GPGPU ausgeführt werden sollen. Die nächsten beiden Module dienen der Optimierung des Quellcodes. Im *Reduction-Kernel-Creator* wird die Suche nach einem global besten Ergebnis aus einer Menge von Ergebnissen optimiert, während im *Random-Function-Converter* die erzeugten Zufallszahlen auf der Grafikkarte optimiert werden. Die Optimierung erfolgt sowohl in der Güte (unabhängige, sich nicht wiederholende Zufallszahlen) als auch in der Geschwindigkeit, in der die Zufallszahlen berechnet werden. Als Ausgabe von *PBA2CUDA* wird ein teilautomatisierter Quellcode erzeugt. In den nachfolgenden Abschnitten werden die jeweiligen Module im Detail beschrieben.

4.4.1 Source-Code-Merger

Die von *CASEP* genutzten Tools haben einige Limitierungen. Eine dieser Limitierungen ist die Notwendigkeit, das eingelesene Programm in einer einzigen Datei vorliegend zu haben. Insbesondere bei großen Programmen erschwert dies die Lesbarkeit und Wartbarkeit. Um diese Limitierung aufzuheben, wurde das *Source-Code-Merger* Modul entwickelt, welches die eingelesenen Dateien in eine gemeinsame Datei transformiert.

Hierzu werden zuerst alle vorhandenen Source-Dateien und die zugehörigen Header-Dateien im Programm in einer Datenstruktur gespeichert (Schritt 1). Aus diesem Pool von Dateien wird die Datei mit der *main*-Methode ermittelt. Diese Datei dient als Hauptdatei (main-Datei) für das spätere Programm (Schritt 2). Im folgenden Schritt wird der Inhalt der restlichen Source-Dateien in die zugehörigen Header-Dateien kopiert (Schritt 3). Somit existieren zum Zeitpunkt der Transformation eine main-Datei und 1 bis n Header-Dateien. Als nächstes müssen die vorhanden Header-Dateien in der richtigen Reihenfolge in die main-Datei eingefügt werden (Schritt 4). Hierzu wird die main-Datei solange rekursiv aufgerufen, bis alle Header-Datein eingefügt wurden. Bei jedem Aufruf wird die *include*-Anweisung durch den Inhalt der entsprechenden Header-Datei ersetzt. Dabei muss sichergestellt werden, dass jede Header-Datei nur ein mal in die Datei integriert wird, um Syntax-Fehler zu verhindern. Zu diesem Zweck werden die schon eingesetzten Header-Dateien in einer Datenstruktur gespeichert und es wird bei der Integration überprüft, ob die jeweilige Datei bereits vorhanden ist. Somit können auch Zyklen in den Include-Anweisungen verhindert werden, die zu einer Endlosschleife bei der Erstellung führen würden. Ein Ablaufdiagramm des Source-Code-Mergers ist in Abb. 4.8 abgebildet.

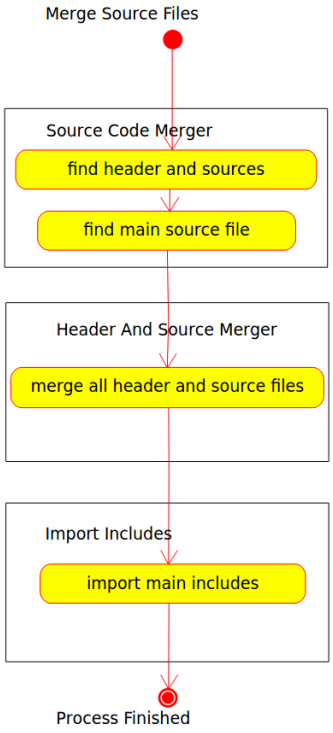


Abbildung 4.8: Ablaufdiagramm Source Code Merger

4.4.2 Function-Saver

Das *Function-Saver* (FS) Modul ist dafür zuständig, fehlerhafte Transformationen von Funktionspointern zu umgehen. Diese entstehen durch das externe Parallelisierungswerkzeug OpenMPC (s. Abschn. 2.5.2.4). OpenMPC löscht bei der Transformation des seriellen Quellcodes in parallelen Code Funktionen, die nur als Funktionspointer referenziert werden. Dies führt zu zwei Fehlern im Programm: Zum einen sind die Funktionsdeklarationen nicht mehr vorhanden und zum anderen bleiben die Aufrufe auf die nicht vorhandenen Funktionen im Quellcode erhalten. Außerdem werden *Defines* durch OpenMPC ersetzt. Diese werden jedoch von einigen später folgenden Teilmodulen benötigt und führen somit zu weiteren Fehlern. Das *FS* Modul löst dieses Problem in mehreren Schritten, die in Abb. 4.9 als Ablaufdiagramm abgebildet sind.

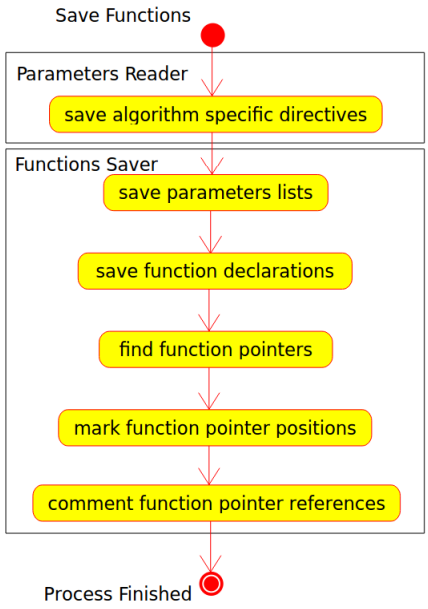


Abbildung 4.9: Ablaufdiagramm Function-Saver

Zuerst werden in der Source-Datei vorhandene Defines durch den *Parameters Reader* ermittelt und gespeichert. Als nächstes werden die Parameterlisten und die Funktionskörper der einzelnen Funktionen gesichert. Darauf folgend werden die Positionen der Funktionen markiert, die nur als Funktionspointer referenziert werden, damit sie von einem späteren Modul wiederhergestellt

werden können. Als letztes werden die Referenzen der Funktionspointer auskommentiert, um Fehler bei der Kompilierung zu vermeiden.

4.4.3 Pragma-Directives-Converter

Die für die Parallelisierung erforderlichen OpenMPC Markierungen im Quellcode erfordern vom Entwickler das Erlernen von neuer Syntax, die eine weitere Hürde bei der Erstellung von paralleler Software darstellt. Weiterhin erschwert die komplexe Syntax die automatisierte Parallelisierung durch das *CASEP* Framework. Eine zusätzliche Hürde bildet die korrekte Deklaration von Variablen in parallelen Abschnitten. Variablen können *private* oder *shared* deklariert werden, je nachdem ob der Zugriff auf die jeweilige Variable exklusiv für einen Thread bestimmt ist oder mehrere Threads gleichzeitig Zugriff auf die Variable haben sollen.

Der *Pragma-Directives-Converter* (PDC) wurde dafür entwickelt, die oben beschriebenen Probleme zu minimieren. Aus einfachen Direktiven werden komplexe OpenMPC Direktiven erzeugt. Außerdem werden bereits einzelne Variablen privat deklariert, bzw. dem Entwickler ein einfaches Interface bereitgestellt, um alle Variablen den richtigen Bereichen zuzuordnen. Dies ist notwendig, um Fehler bei der Programmausführung zu vermeiden. Sollte beispielsweise eine Variable *shared* deklariert werden, auf der jeder Thread seine lokalen Daten abspeichert, kann jeder andere Thread die lokalen Daten überschreiben. Dies ist nicht gewollt und führt in der Regel zu Fehlern in der Ausführung. In Abb. 4.10 ist der Ablauf des *PDCs* abgebildet.

Zuerst werden über ein Interface die Speedup-Vorhersagen ausgewertet, bei denen festgelegt wird, ob der Programmabschnitt mit OpenMP oder CUDA parallelisiert werden soll und ob sich eine Parallelisierung generell lohnt. Sollte dies der Fall sein, wird die zu parallelisierende Stelle im Quellcode gesucht und die Variablen werden extrahiert. Laufvariablen von Schleifen werden dabei direkt als *private* deklariert, da auf diese nicht von allen Threads konkurrierend zugegriffen werden darf, um eine korrekte Ausführung des Programms zu garantieren. Für die übrigen Variablen muss der Anwender explizit angeben, ob diese als *shared* oder *private* deklariert werden sollen. Im *Pragma Generator*

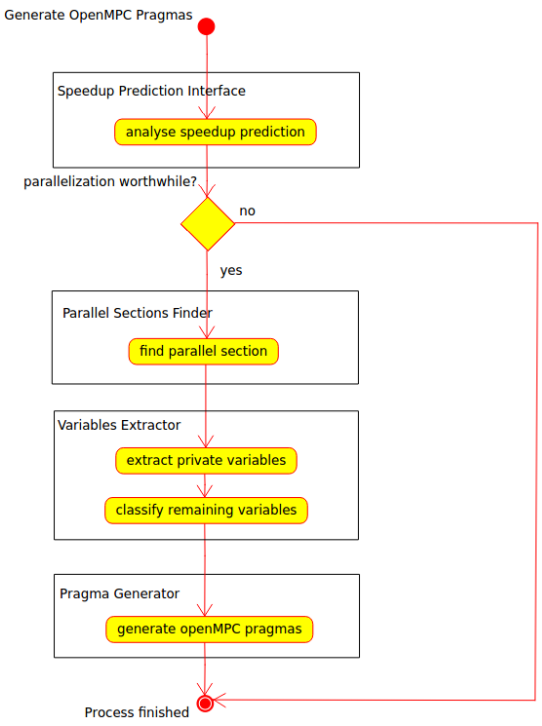


Abbildung 4.10: Ablaufdiagramm Pragma-Directives-Converter

werden die gesammelten Informationen dazu verwendet, aus den vereinfachten Direktiven korrekte OpenMPC Direktiven zu erzeugen.

```
1 #pragma parallel optimize
2       {
3     for (i = 0; i < 100; i++) {
4     updateParticles(p, pb, v, g, i);
5     calcFitness(p,i);
6     checkBest(p, pb, g,i);
7     }
```

Listing 4.4: Vereinfachte PBA2CUDA-Direktive

Im Listing 4.4 ist ein Beispiel eines mit einer PBA2CUDA-Direktiven markierten Quelltexts dargestellt. Die Direktive ist *#pragma parallel optimize*. Diese besteht aus den zwei fest definierten Bezeichnern *#pragma* und *parallel* und einem frei wählbaren Bezeichner (in diesem Fall *optimze*). Bei mehreren

parallelen Bereichen im Quellcode, muss für jeden Bereich ein anderer frei wählbarer Bezeichner verwendet werden. Im Listing 4.5 ist die vom *Pragma-Directives-Converter* umgewandelte Direktive abgebildet. Die private Variable i wurde dabei automatisch ermittelt. Die *shared*-Variablen können nicht automatisch ermittelt werden und müssen vom Entwickler bestätigt werden.

```
1 #pragma omp parallel private(i) shared(p,pb,v,g)
2 {
3 #pragma omp for nowait
4         for (i = 0; i < 100; i++) {
5           updateParticles(p,pb,v,g,i);
6           calcFitness(p,i);
7           checkBest(p,pb,g,i);
8         }
9       }
```

Listing 4.5: Umgewandelte OpenMPC-Direktive

4.4.4 OpenMPC Compiler

Beim *OpenMPC Compiler* handelt es sich um ein externes Modul, das aufgerufen wird, um den seriellen Code in parallelen Code zu transformieren. Hierzu wird der in Abschn. 2.5.2.4 beschriebene OpenMPC Compiler genutzt. Der so transformierte Quellcode enthält teilweise Fehler, die von den hier beschriebenen Modulen korrigiert werden. Weiterhin ist der vom OpenMPC Compiler erzeugte Quellcode nicht für die zugrunde liegenden Algorithmen optimiert, was zu suboptimalen Laufzeiten führen kann. *CASEP* führt in den nachfolgend beschriebenen Modulen automatisiert Optimierungen durch, die zu Laufzeitverbesserungen und Verbesserungen der Ergebnissen führen.

4.4.5 Function-Restorer

Das *Function-Restorer* (FR) Modul stellt die vom *FS* gesicherten Funktionen nach der Parallelisierung des Quellcodes wieder her. Der genaue Ablauf ist in Abb. 4.11 dargestellt. Zuerst wird für jede gesicherte Funktion im *Function*

Body Restorer untersucht, ob die Funktion bei der Parallelisierung entfernt wurde. Sollte dies der Fall sein, wird sie wieder in das parallele Programm eingefügt. Als nächstes werden die auskommentierten Funktionspointer vom *Function Pointer Restorer* wieder in den Quellcode einkommentiert. Diese wurden vom *FS* auskommentiert, da OpenMPC diejenigen Funktionsdeklarationen entfernt, deren Funktionen nur als Funktionspointer verwendet werden. Der von *CASEP* genutzte Parser (ROSE, s. Abschn. 2.6) unterstützt jedoch nur syntaktisch korrekten Quellcode, somit ist dieser Schritt notwendig, um den Quellcode korrekt weiterverarbeiten zu können.

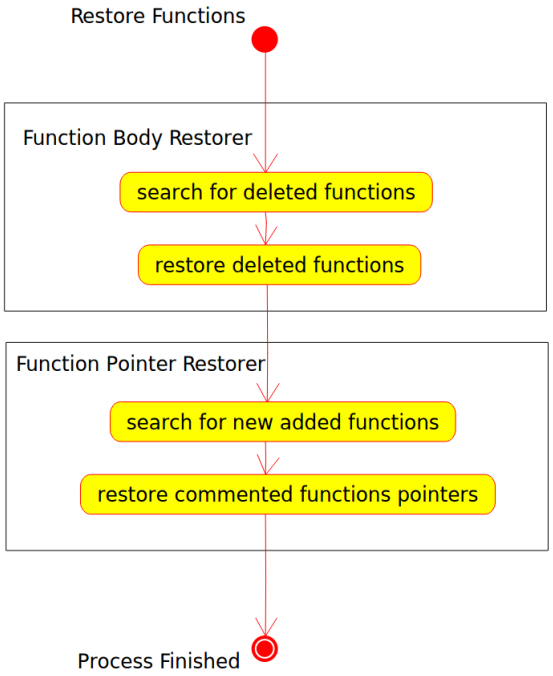


Abbildung 4.11: Ablaufdiagramm Function-Restorer

4.4.6 OpenMPC-Corrector

Der *OpenMPC-Corrector* (OC) löst die durch den *FS* und *FR* nicht abgedeckten Konflikte auf. Die in *CASEP* eingesetzten Werkzeuge verwenden teilweise unterschiedliche syntaktische Konstrukte, was den *OC* zur Konfliktauflösung notwendig macht. Für *CASEP* sind zwei Grundprobleme relevant:

Der OpenMPC Compiler ersetzt *NULL*-Werte im Quellcode durch den Ausdruck *(void*0)*. Dieser Ausdruck kann zwar korrekt von den gängigen Compilern übersetzt werden, ROSE erzeugt hier jedoch beim Parsen Fehlermeldungen. *OC* ersetzt diesen Ausdruck durch den ursprünglichen *NULL*-Wert. Einen weitaus gravierenderen Einfluss hat die nicht immer korrekt stattfindende Markierung von *__device__* und *__host__* Funktionen, also von Funktionen, die entweder auf der GPGPU oder CPU ausgeführt werden sollen. OpenMPC markiert teilweise Funktionen fälschlicherweise als reine *device* Funktionen. Der *OC* korrigiert den Fehler, indem im Quellcode nach *device* Funktionen gesucht wird und die *__device__* Direktive durch eine *__host__ __device__* Direktive ersetzt wird. Somit wird sichergestellt, dass die Funktion sowohl auf der CPU (Host), als auch auf der GPGPU ausgeführt werden kann. Dabei muss jedoch sichergestellt werden, dass reine *host*-Funktionen, wie *rand()* oder *gettimeofday*, nicht fälschlicherweise als device Funktionen deklariert werden, was zu einem Fehler bei der Kompilierung des Programms führen würde.

4.4.7 Reduction-Kernel-Creator

Bei den letzten beiden Modulen handelt es sich um Optimierungen, die das Laufzeitverhalten und die Ergebnisse des Programms weiter verbessern sollen. Eine der wichtigsten Operationen bei *PBAs* besteht darin, in jeder Iteration die beste Lösung aus einer Menge an Lösungen zu bestimmen. Ein nicht optimierter Ansatz besteht darin, alle Lösungen miteinander zu vergleichen, was zu einer Laufzeit von $\mathcal{O}(n)$ für den Vergleich von n Lösungen führt. Der *Reduction-Kernel-Creator* (RKC) [33] reduziert die Laufzeit von solchen Vergleichen auf $\mathcal{O}(\log n)$. Hierzu werden zu Beginn der Prozedur $n/2$ Threads erstellt, von denen jeder Thread zwei Lösungen in der ersten Iteration miteinander vergleicht. Die somit ermittelten $n/2$ Lösungen werden in der zweiten Iteration von $n/4$ Threads wiederum verglichen. Die Iterationsschleife wird abgebrochen, wenn nach m Iterationen nur noch eine Lösung übrig ist, die die beste Lösung aus der Menge an Lösungen repräsentiert. Eine schematische Darstellung der Reduktion-Prozedur ist in Abb. 4.12 dargestellt.

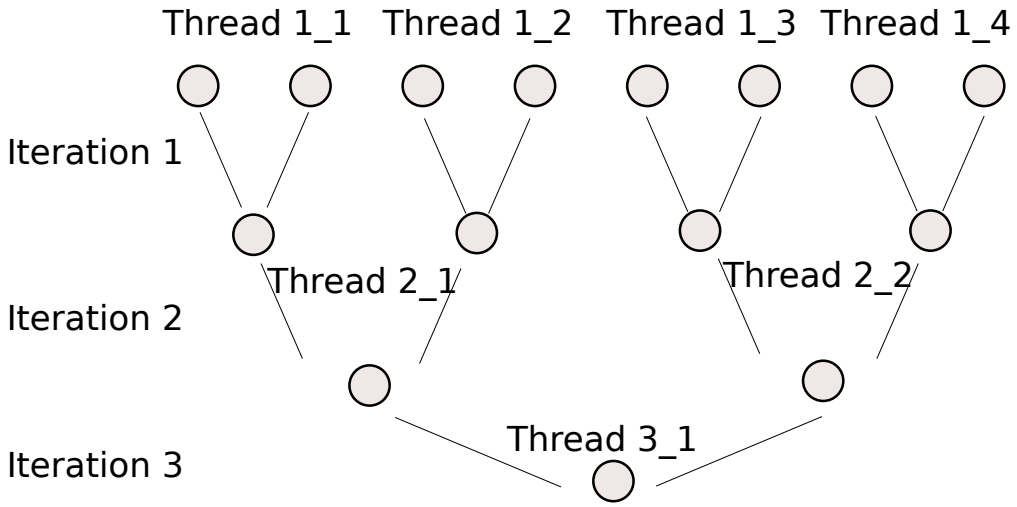


Abbildung 4.12: Schematische Darstellung Reduktion-Prozedur

4.4.8 Random-Function-Converter

Zufallszahlen spielen bei heuristischen Verfahren wie *PBAs* eine entscheidende Rolle. Der *Random-Function-Converter* (RFC) modifiziert Aufrufe von Zufallszahlen im parallelisierten Quellcode, um Fehler zu korrigieren und bessere Zufallswerte zu erzeugen. Ein grundsätzliches Problem von OpenMPC ist das Fehlen einer automatisierten Umwandlung von Host-Funktionen in Device-Funktionen, also Funktionen, die auf der GPGPU ausgeführt werden können. Bei dem *rand()* Aufruf für serielle Programme handelt es sich um eine typische Host-Funktion, mit der Zufallszahlen erzeugt werden. Der *RFC* wandelt jeden Aufruf von *rand*() in einen Zufallszahlenaufruf mit der CUDA Bibliothek *Thrust* um. Hierbei sind grundsätzlich zwei Fälle zu beachten:

1. Fall 1 - Der *rand*() Aufruf befindet sich direkt in der Kernel-Funktion
2. Fall 2 - Der *rand*() Aufruf befindet sich verschachtelt in einer von der Kernel-Funktion aufgerufenen Funktion.

Sollte sich der *rand*() Aufruf direkt in der Kernel-Funktion befinden, so wird auf dem Host (CPU) ein *Seed* erzeugt und dieser an die Kernel-Funktion weitergereicht. Dort wird der Seed mit der jeweiligen Thread-ID verknüpft, um für jeden Thread unabhängige und unterschiedliche Zufallszahlen zu gewähr-

leisten. Zu diesem Zweck muss der Funktionskopf um den Seed-Parameter erweitert werden und der *rand*() Aufruf in der Funktion durch einen äquivalenten Thrust-Aufruf ersetzt werden, der auf der GPGPU ausgeführt werden kann.

Der zweite Fall, bei dem sich der *rand*() Aufruf in einer Funktion befindet, die von der globalen Kernel-Funktion aufgerufen wird, ist weitaus komplexer. Die globale Kernel Funktion (global function) ist diejenige Funktion, die vom Host aufgerufen wird; sie dient somit als Schnittstelle zwischen dem Host und dem Device (GPGPU), wobei sie selbst auf der GPGPU ausgeführt wird. Jede globale Funktion kann wiederum weitere Kernel-Funktionen (device functions) aufrufen, die dann entsprechend auf der GPGPU ausgeführt werden. Sollten sich *rand*() Aufrufe in einer device function befinden, muss zuerst ein Funktionsbaum erstellt werden, wie er in Abb. 4.13 abgebildet ist. Der oberste

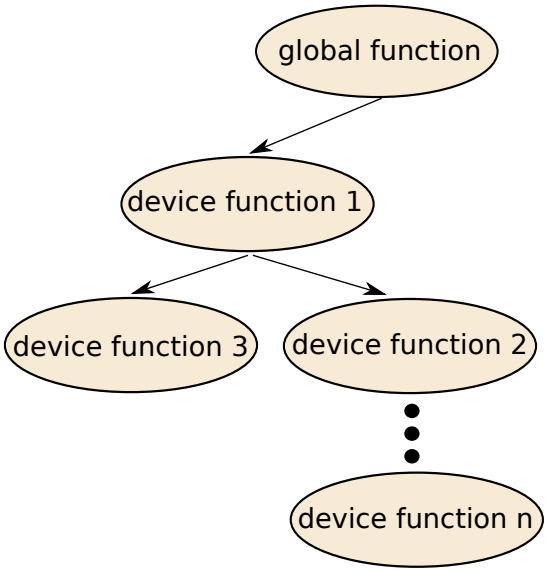


Abbildung 4.13: Funktionsbaum Beispiel

Knoten bildet hierbei den Wurzelknoten, von dem aus verschiedene device functions aufgerufen werden können. Sollte ein *rand*() Aufruf in einem der Knoten gefunden werden, wird die Funktionskette vom Wurzelknoten zum jeweiligen Blattknoten ermittelt (global function → device function 1 → ... → device function n). Im nächsten Schritt wird der Funktionskopf von jeder in

der Funktionskette enthaltenden Funktion um den Seed-Parameter erweitert, um den Seed bis zur Funktion mit dem *rand*()-Aufruf weiterzuleiten. Zuletzt wird, wie bei Fall 1, der *rand*()-Aufruf durch einen Thrust-Aufruf ersetzt.

4.5 Zusammenfassung

Dieses Kapitel hat die Architektur und die einzelnen Module von *CASEP* im Detail beschrieben. Dabei wurden die einzelnen Module den jeweiligen Anwendungsfeldern (*Unterstützung bei der Programmierung*, *Speedup-Vorhersage* und *Parallelisierung*; s. Abschn. 3.1) zugeordnet. Das beschriebene Framework erlaubt dem Entwickler, ein auf Speedup-Vorhersage basierendes Hardware-Software-Mapping zu erstellen. Das Mapping wird mit dem Ziel erstellt, eine möglichst optimale Laufzeit von *PBAs* auf parallelen Architekturen (Multi-Core CPU und GPGPU) zu erzielen. Des Weiteren ermöglicht *CASEP* eine automatisierte Parallelisierung des Programms, basierend auf das vorhergesagte Mapping.

Die beiden Speedup-Vorhersage-Module *Benchmarking* und *Code-Analysis* wurden in ihrer Grundform und mit den in dieser Arbeit implementierten Erweiterungen beschrieben. Das *Benchmarking*-Modul ermittelt die zu erwartende Ausführungszeit indem Benchmarks erzeugt werden, die die Charakteristika des Ursprungsprogramms möglichst genau abbilden. Diese Benchmarks werden in einer seriellen Version, parallelen Version auf Multi-Core CPUs und parallelen Version auf GPGPUs erstellt. Die Laufzeiten der Benchmarks werden in Relation zueinander gesetzt, um den zu erwartenden Speedup zu bestimmen. Das *Code-Analysis*-Modul analysiert die Operationen aus markierten Bereichen im Quellcode. Aus den analysierten Operationen werden Micro-Benchmarks erstellt. Diese Micro-Benchmarks werden, ebenso wie beim *Benchmarker*, in drei Ausführungen erstellt (seriell, parallel auf Multi-Cores und parallel auf GPGPUs) und ausgeführt. Aus den sich daraus ergebenen Laufzeiten wird der relative Speedup der parallelen Ausführungen im Vergleich zur seriellen Ausführung ermittelt. Beim *Benchmarking*-Modul wird eine Verbesserung der Speedup-Vorhersage erzielt, indem das zu untersuchende Problem mit Hilfe

des *Function Synthesizers* modelliert wird. Beim *Code-Analysis*-Modul wurde eine Methode vorgestellt, die Cache-Effekte minimiert, die die Vorhersage verfälschen können.

Das vorgestellte *PBA2CUDA*-Modul ist für die automatisierte Parallelisierung von PBAs zuständig. Es wurden die einzelnen Teilmodule und ihre genaue Funktionsweise vorgestellt. *PBA2CUDA* ist in vier Ausführungsphasen eingeteilt. In der ersten Phase wird der serielle Quellcode für die Parallelisierung vorbereitet. Die zweite Phase besteht aus der eigentlichen automatischen Parallelisierung. In der dritten Phase werden mögliche Fehler in der Parallelisierung korrigiert. Die dritte Phase ist abhängig von dem verwendeten Algorithmus. In dieser Phase werden für *PBAs* mögliche Optimierungen im Quellcode durchgeführt, die zu einer Verringerung der Laufzeit und Verbesserung der Ergebnisse führen.

Das nachfolgende Kapitel beschäftigt sich mit der Verallgemeinerung der für *CASEP* vorgestellten Methoden. Dabei werden die folgende Generalisierungsaspekte behandelt: 1) Generalisierung für andere Algorithmenklassen, 2) Generalisierung für andere Hardware, 3) das Verwenden alternativer Parallelisierungsmethoden und 4) Generalisierung zum Verwenden anderer Programmiersprachen.

5 Generalisierung von CASEP

Inhaltsangabe

In den vorangegangenen Kapiteln wurde das *CASEP* Framework im Umfeld der *PBAs* auf GPGPUs und Multi-Core CPUs betrachtet. Dieses Kapitel beschäftigt sich mit der Generalisierung von *CASEP*. Dabei werden sowohl alternative Algorithmenklassen, als auch unterschiedliche Hardwarearchitekturen und verschiedene Parallelisierungsmethoden betrachtet. In den jeweiligen Abschnitten wird auf die entsprechenden Module, die für eine Generalisierung angepasst werden müssen, eingegangen. Dabei werden die jeweils anzupassenden Module farblich markiert und diskutiert.

Abschn. 5.1 beschäftigt sich mit verschiedenen Algorithmenklassen, während in Abschn. 5.2 auf alternative Hardware eingegangen wird. Abschn. 5.3 beschreibt die Anpassungen, die für alternative Parallelisierungsmethoden notwendig sind. Weiterhin wird auf das Verwenden von anderen Programmiersprachen in Abschn. 5.4 eingegangen. Abschn. 5.5 zeigt die in diesem Kapitel

beschriebenen Methoden anhand eines Beispielszenarios und Abschn. 5.6 fasst die Ergebnisse dieses Kapitels zusammen.

5.1 Anpassung an andere Algorithmenklassen

CASEP wurde entwickelt, um das Parallelisieren von *PBAs* für Entwickler zu vereinfachen. Die grundlegenden Prinzipien lassen sich jedoch auch auf andere Algorithmenklassen erweitern, sowohl auf die Obermenge der *PBAs* (biologisch inspirierte Algorithmen), als auch auf Algorithmen aus anderen Bereichen. Dieser Abschnitt beschäftigt sich mit den Modulen aus der Speedup-Vorhersage und Parallelisierung und es werden die notwendigen Anpassungen für eine Generalisierung aufgezeigt.

5.1.1 Anpassungen an die Speedup-Vorhersage

5.1.1.1 Benchmarker

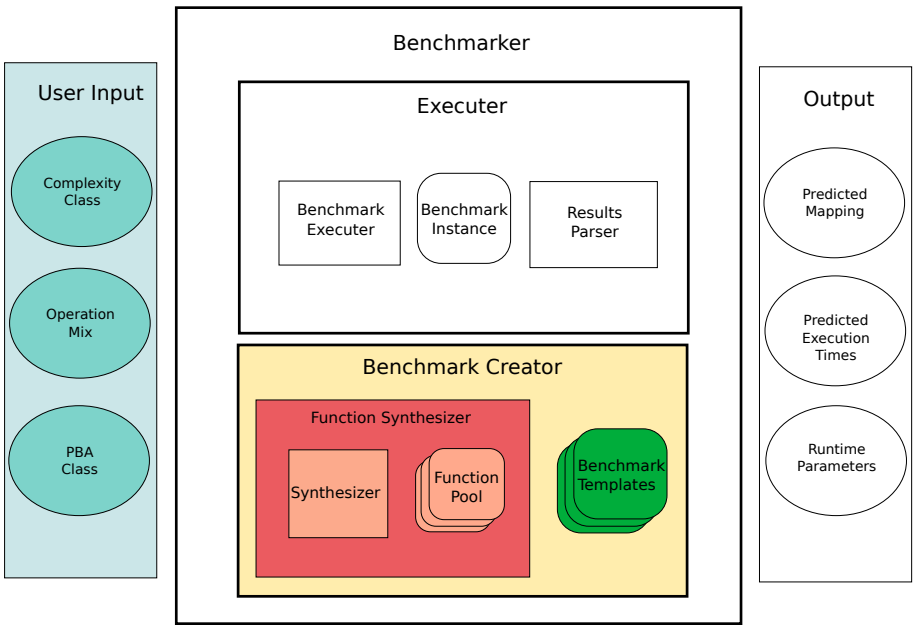


Abbildung 5.1: Generalisierung des Benchmarkers - Algorithmenklassen

Das vorgestellte Benchmark-Modul (s. Abschn. 4.2.1) besteht aus zwei

Teilmodulen (s. Abb. 5.1). Während der Executer nicht angepasst werden muss, ist der *Benchmark Creator* auf *PBAs* ausgelegt. Insbesondere müssen für eine Generalisierung die Benchmark-Templates angepasst werden. Diese müssen jeweils eine generalisierte Form der zu betrachtenden Algorithmen abbilden. Weiterhin sind möglicherweise Anpassungen im *Function-Synthesizer* notwendig. Sollte die betrachtete Algorithmenklasse keine Funktionen als Problemlösung beinhalten, muss ein Synthesizer für die zu untersuchende Problemklasse implementiert werden. Beispielsweise könnte für die Problemklasse der String-Operationen ein Synthesizer entwickelt werden, der unterschiedlich komplexe String-Operationen beinhaltet. Für den Fall, dass kein Synthesizer für das zu untersuchende Problem entwickelt werden kann, kann dieser auch entfallen, da er keinen essentiellen Bestandteil des Benchmarkers darstellt, sondern nur der Verbesserung der Ergebnisse dient. Dies kann z.B. der Fall sein, wenn das zu lösende Problem zu speziell ist und nicht durch eine Menge an Benchmark-Problemen abgebildet werden kann, wie es bei der Funktionsoptimierung der Fall ist. Je nach verwendeter Problemklasse müssen weiterhin die Eingaben des Entwicklers angepasst werden. Der Benchmarker benötigt die Komplexitätsklasse des Problems und, je nach Synthesizer, Informationen über das zu lösende Problem.

5.1.1.2 Code Analysis

Das in Abschn. 4.2.2 vorgestellte Code-Analysis-Modul (s. Abb. 5.2) ist nahezu vollständig auf andere Algorithmenklassen anwendbar. Seine Funktionsweise beruht auf dem zu untersuchenden Quellcode und ist unabhängig von den zu untersuchenden Algorithmen. Das einzige Modul, das je nach Problemstellung angepasst werden muss, ist das *Operation-Crawler-Modul*. Dieses untersucht die im Quellcode befindlichen Operationen. *CASEP* ist auf mathematische Operationen ausgelegt, sollten für andere Algorithmenklassen andere Operationen notwendig sein, muss der *Crawler* auch diese Erkennen können. Diese Anpassung ist jedoch einfach durchzuführen, da dies in *CASEP* vorgesehen und das Modul leicht erweiterbar ist.

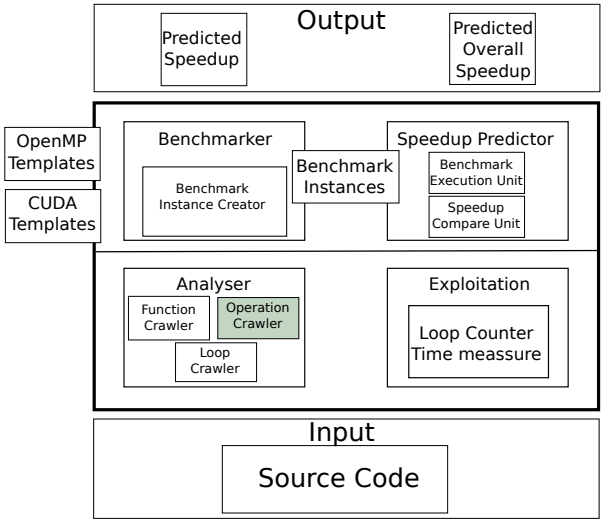


Abbildung 5.2: Generalisierung der Code-Analyse - Algorithmenklassen

5.1.2 Anpassung an die Parallelisierung

Die grundlegende Funktionsweise von *PBA2CUDA* (s. Abschn. 4.4) beruht auf dem zu transformierenden Quellcode und ist unabhängig von der Algorithmenklasse. Jedoch führt *PBA2CUDA* Optimierungen im Quellcode durch, die auf *PBAs* ausgelegt sind. Die Implementierung der Optimierungen befinden sich im *Random-Function-Converter* und *Reduction-Kernel-Creator*. Beide Optimierungsroutinen können allerdings auch auf andere Algorithmen angewandt werden, sollten sie ähnliche Strukturen aufweisen. Dies ist der Fall, wenn im zu untersuchenden Algorithmus Zufallszahlenaufrufe vorkommen und/oder aus einer Menge von Lösungen die beste Lösung berechnet werden soll. Sollte dies nicht der Fall sein, müssen diese Teilmodule deaktiviert werden. *PBA2CUDA* kann jedoch uneingeschränkt weiterverwendet werden. Alternativ wären Optimierungsmodule auch für andere Algorithmenklassen denkbar, diese müssen dann für den jeweiligen Algorithmus implementiert werden und können aufgrund des modularen Aufbaus von *PBA2CUDA* einfach in das vorhandene Framework eingefügt werden.

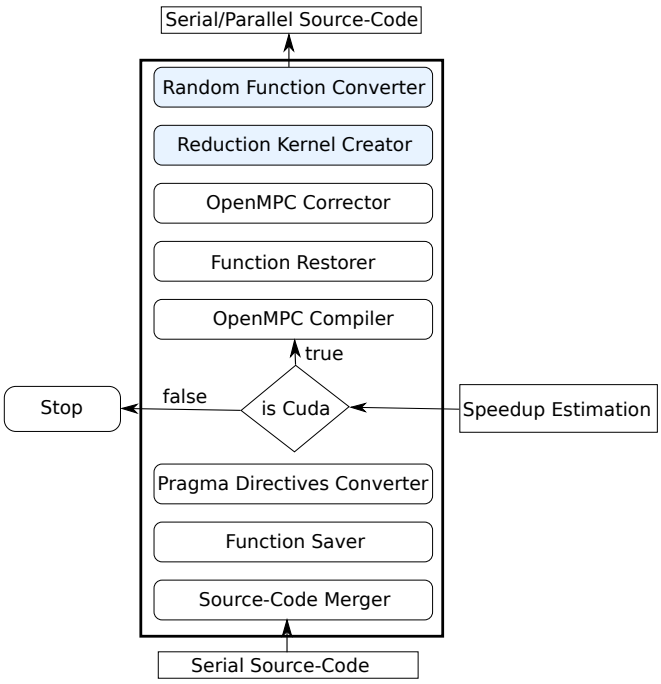


Abbildung 5.3: Generalisierung von PBA2CUDA - Algorithmenklassen

5.2 Anpassungen an andere Hardware

Die Zielplattformen des in dieser Arbeit vorgestellten Framework sind CUDA GPGPUs und Multi-Core CPUs. Jedoch lassen sich die Prinzipien von *CASEP* auf andere Plattformen generalisieren. Hierzu sind an verschiedenen Stellen des Frameworks Modifikationen durchzuführen, die im Nachfolgenden beschrieben werden.

5.2.1 Anpassungen an die Speedup-Vorhersage

5.2.1.1 Benchmarker

Das Benchmarker Modul von *CASEP* ist nahezu architekturunabhängig und kann in dieser Form auf anderen Architekturen ausgeführt werden. Einzig die Lauffähigkeit der erzeugten Benchmark-Instanzen muss gewährleistet werden. Hierzu muss jeweils der passende Compiler für die darunterliegende Architektur verwendet werden. Außerdem müssen bei den Benchmark-Templates die

für die Architektur üblichen Direktiven verwendet werden. Die erzeugten Ergebnisse lassen sich von *CASEP* ausnahmslos nutzen. In Abb. 5.4 sind das

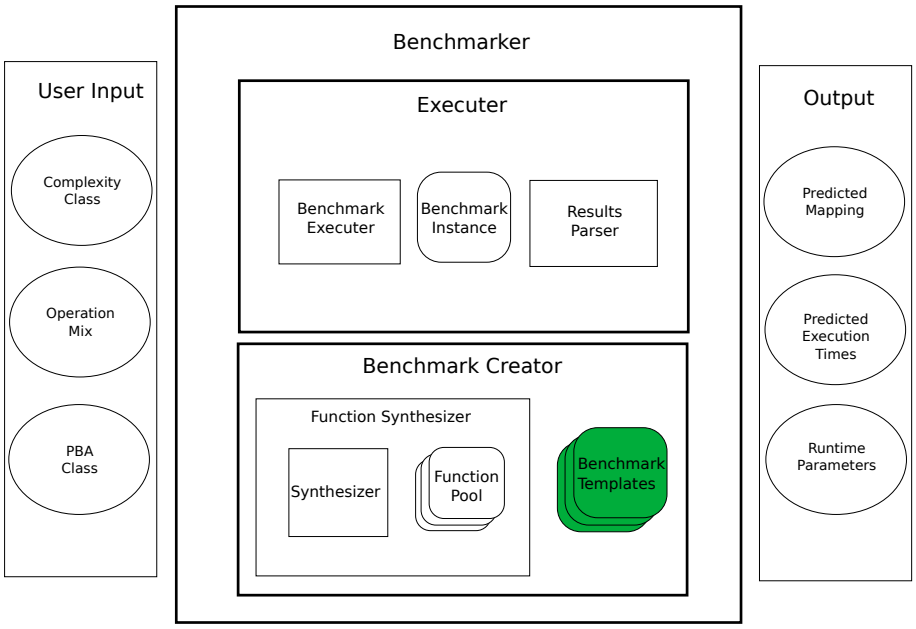


Abbildung 5.4: Generalisierung des Benchmarkers - Architektur

Modul und der Bereich, der angepasst werden muss, nochmals farblich dargestellt.

5.2.1.2 Code Analysis

Ähnlich wie bei dem Benchmarker, muss bei der Code-Analysis lediglich die Lauffähigkeit der Micro-Benchmarks auf der jeweiligen Hardware garantiert werden (s. Abb. 5.5).

Hierzu müssen die CUDA- und OpenMP-Templates durch Templates für die Zielarchitektur ausgetauscht werden. Außerdem muss sichergestellt werden, dass der passende Compiler für die Architektur verwendet wird, um die Lauffähigkeit der Micro-Benchmarks zu garantieren.

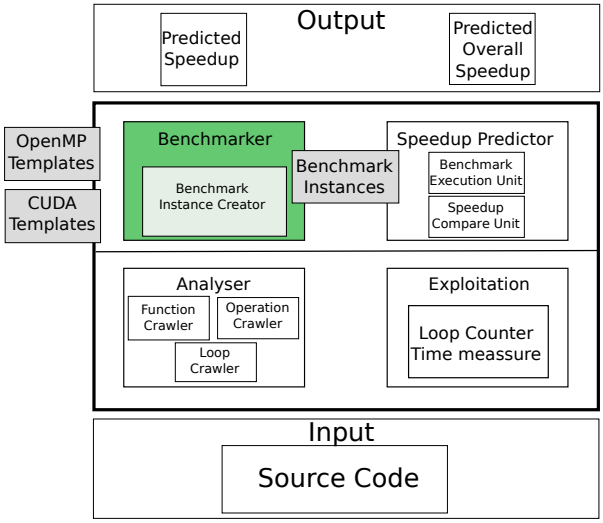


Abbildung 5.5: Generalisierung der Code-Analyse - Architektur

5.2.2 Anpassung an die Parallelisierung

Das *PBA2CUDA* Modul verwendet für die Parallelisierung einen externen Parallelisierer. Dementsprechend muss für die Anpassung an andere Hardware ein äquivalenter Parallelisierer für die Zielplattform verwendet oder entwickelt werden (s. Abb. 5.6). Mögliche Probleme bei der Transformation müssen mit separaten Modulen, ähnlich dem *OpenMPC-Corrector*, korrigiert werden. Weiterhin müssen die Optimierungen, die von *PBA2CUDA* durchgeführt werden (*Reduction-Kernel-Creator* und *Random-Function-Converter*), an die jeweilige Architektur angepasst oder entfernt werden.

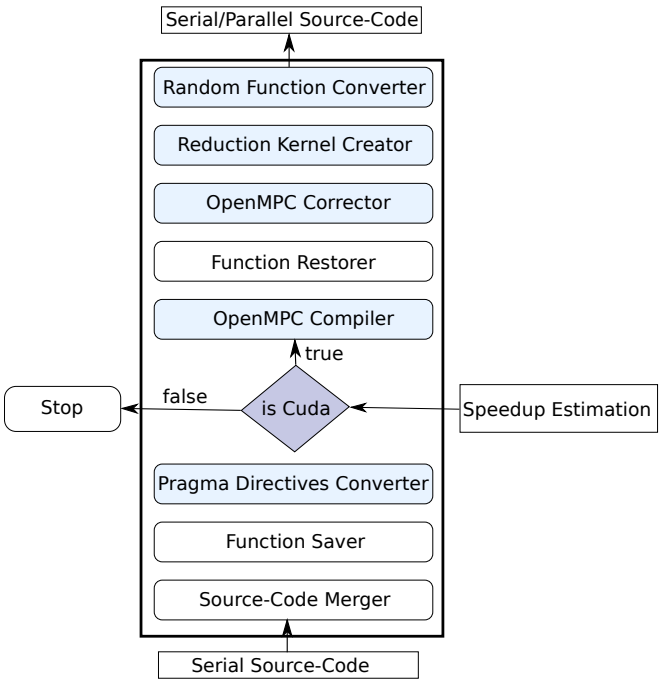


Abbildung 5.6: Generalisierung von PBA2CUDA - Architektur

5.3 Generalisierung von Parallelisierungsmethoden

Das *CASEP* Framework beschäftigt sich mit einfachen Parallelisierungsmethoden, die vom einem Entwickler ohne großen Aufwand durchgeführt werden können und zu einem Speedup gegenüber einer seriellen Version des Programms führen. Für die Parallelisierung sind jedoch viele unterschiedliche Methoden in der Literatur vorhanden, wie in Abschn. 2.3 beschrieben wurde, die möglicherweise zu einem besseren Laufzeitverhalten führen können. Sollen weitere Methoden verwendet werden, müssen einzelne Module von *CASEP* angepasst werden, was in diesem Abschnitt behandelt wird.

5.3.1 Anpassungen an die Speedup-Vorhersage

Dieser Abschnitt beschreibt die Anpassungen der einzelnen Speedup-Module, die für die Nutzung alternativer Parallelisierungs-Pattern notwendig sind.

5.3.1.1 Benchmarking

Die Anpassungen an das Benchmarking-Modul sind vergleichsweise gering. Es müssen lediglich die Benchmark Templates (s. Abb. 5.7) an die jeweiligen

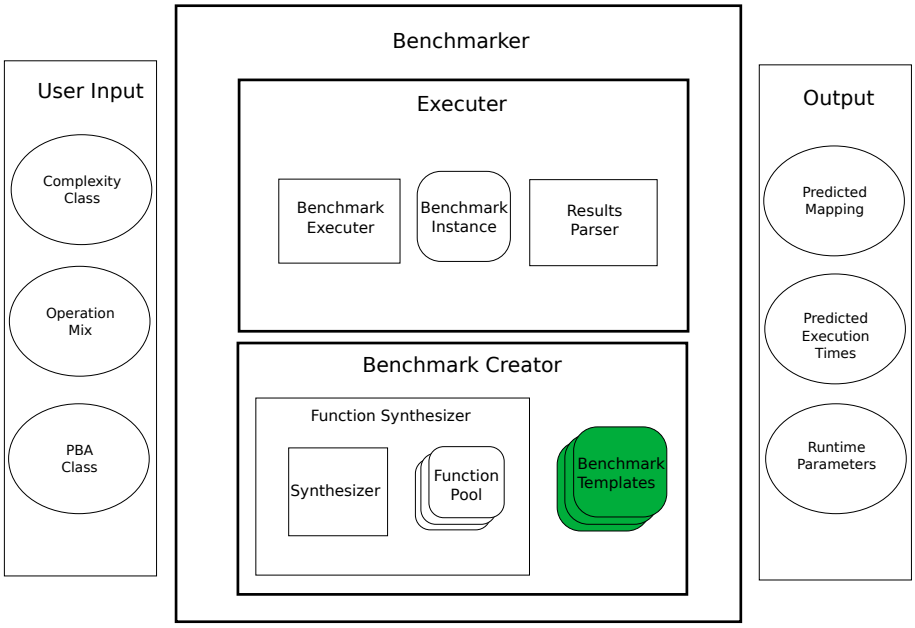


Abbildung 5.7: Generalisierung des Benchmarkers - Parallelisierungsmethoden

Pattern angepasst werden. Hierzu müssen Templates erzeugt werden, die das jeweilige Pattern generisch darstellen und vom Benchmarker weiterverarbeitet werden können. Die restlichen Module funktionieren unabhängig davon und können ohne weitere Modifikation verwendet werden.

5.3.1.2 Code Analysis

Auch bei der Code-Analysis sind lediglich Anpassungen an den vorhandenen Templates durchzuführen (s. Abb. 5.8). Ähnlich dem Benchmarking müssen auch hier die Templates die gewünschten Parallelisierungs-Pattern nachbilden. Die erzeugten Ergebnisse können dann vom Code-Analysis-Modul ohne Modifikation weiterverarbeitet werden.

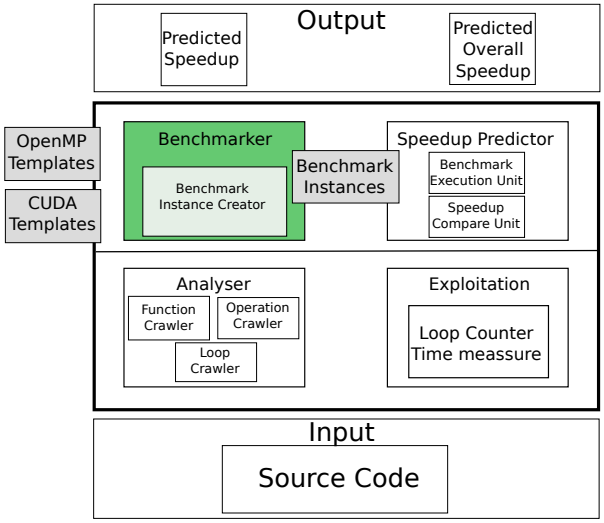


Abbildung 5.8: Generalisierung Code-Analysis - Parallelisierungsmethoden

5.3.2 Anpassung an die Parallelisierung

Das *PBA2CUDA* Modul unterscheidet sich von den übrigen Modulen in Bezug auf die vorzunehmenden Modifizierungen für alternative Parallelisierungsmethoden. Für eine Anpassung muss ein weiteres Modul hinzugefügt werden, welches den *PBA* zuerst in das neue Pattern überführt, beispielsweise in ein Island-Modell (s. Abschn. 2.3). Der transformierte Quellcode kann dann ohne weitere Modifikationen aus Sicht von *PBA2CUDA* in parallelen Quellcode transformiert werden.

5.4 Verwendung von alternativen Programmiersprachen

CASEP wurde mit der Intention entwickelt, seriellen Quellcode ohne großen Aufwand in möglichst performanten parallelen Code zu transformieren. Aus diesem Grund fiel die Wahl auf die Programmiersprachen C/C++. Diese Programmiersprachen eignen sich besonders bei der Programmierung, wenn es um Geschwindigkeitsvorteile gegenüber anderen Hochsprachen geht [87] [88]. Weiterhin sind C/C++, zusammen mit *Fortran* [89], die von NVIDIA für CUDA

nativ unterstützen Programmiersprachen. Zwar gibt es offene Implementierungen für andere Programmiersprachen, wie z.B. Java mit *JCuda* [90], diese erreichen jedoch nicht die Leistungsfähigkeit von nativen CUDA Anwendungen. Aus diesem Grund wurde in dieser Arbeit darauf verzichtet, alternative Programmiersprachen zu betrachten. Bei einer möglichen Anpassung sind in nahezu allen Modulen von *CASEP* Modifikationen durchzuführen, was den Rahmen dieser Arbeit sprengen würde.

Alternativ könnte auch ein Konverter für Programmiersprachen verwendet werden, der ein Programm von einer Programmiersprache zu einer anderen konvertiert. In der Literatur finden sich verschiedene Ansätze, wie beispielsweise *j2c* [91], der Java Code in C++ Code umwandelt. Mit einem solchen Ansatz würden die Modifikationen für *CASEP* entfallen, jedoch entstünden andere Probleme. Automatisierte Transformationstools erzeugen für gewöhnlich für den Entwickler schwer lesbaren Code. Somit ist die Wiederverwendbarkeit des so erzeugten Codes nicht gewährleistet. Weiterhin ist in dieser Form erzeugter Code möglicherweise von *CASEP* nicht als Eingabe geeignet und muss durch neu entwickelte Module in eine passende Form transformiert werden.

5.5 Generalisierungsbeispiel

In diesem Abschnitt wird die Generalisierung von *CASEP* anhand eines theoretisch betrachteten Beispiels aufgezeigt. Es wird das zu implementierende Szenario vorgestellt und die notwendigen Schritte für die Anpassung von *CASEP*. Die zu betrachtenden Parameter sind folgende:

1. Anpassung an andere Algorithmenklassen: Es sollen Algorithmen aus dem Bereich der *Neuronalen Netze* (NN) [92] betrachtet werden, die sich besonders gut mit CUDA und OpenMP parallelisieren lassen [93], [94]. *NN* sind wie *PBAs* biologisch inspirierte Algorithmen, die verbundene Neuronen abbilden, und häufig zur Mustererkennung verwendet werden. Die Verbindungen der einzelnen Neuronen werden dabei durch Parameter gewichtet, mit denen sich das Verhalten des *NN* steuern lässt. Somit

lassen sich für das *NN* durch Modifikation dieser Parameter unterschiedliche Ausgabewerte für bestimmte Eingaben (z.B. ein zu untersuchendes Muster) festlegen. Die Aufgabe bei *NN* besteht darin, diese Parameter bestmöglich zu optimieren, um das gewünschte Ergebnis zu erhalten.

2. Anpassungen an andere Hardware: Der zu betrachtende Algorithmus soll auf generischen GPGPUs ausgeführt werden und nicht ausschließlich auf CUDA fähige GPGPUs beschränkt sein.

3. Generalisierung von Parallelisierungsmethoden: Dieser Punkt wird implizit durch die Verwendung einer anderen Algorithmenklasse abgedeckt, bei der eine unterschiedliche Parallelisierung notwendig ist.

4. Verwendung von alternativen Programmiersprachen: Das zu untersuchende Beispiel soll aufgrund der in Abschn. 5.4 beschriebenen Punkte weiterhin auf die Programmiersprache C/C++ beschränkt sein.

Anpassung an andere Algorithmenklassen: Eine Veränderung der Algorithmenklasse erfordert hauptsächlich Änderungen in den genutzten Speedup-Vorhersage-Modulen. Insbesondere müssen die Templates des *Benchmarkers* komplett neu konzipiert werden, um die neue Algorithmenklasse abzubilden. *NN* lassen sich programmatisch als Matrizen abbilden, bei denen die Spalten die Gewichte zwischen den einzelnen Neuronen repräsentieren. Somit ist es notwendig bei den verwendeten Templates die genutzten Matrix-Operationen (beispielsweise Matrix-Vektor Multiplikation) abzubilden. Weiterhin werden ähnlich wie bei *PBAs* Fitnessfunktionen verwendet, die die Güte der Konfiguration (Werte der Gewichte) bestimmen. Diese Fitnessfunktionen können zum größten Teil von den Templates der *PBAs* übernommen werden und müssen nur an das zu lösende Problem angepasst werden.

Für die Code-Analyse sind weitaus weniger Anpassungen notwendig. Das *Code-Analysis*-Modul erstellt aus den im Quellcode vorhandenen Operationen Micro-Benchmarks und ist somit in seiner Funktionsweise unabhängig von der zu untersuchenden Architektur. Es muss nur beachtet werden, dass das *Code-Analysis*-Modul die im Quellcode vorhandenen Operationen erkennt. *CASEP*

ist in der hier vorgestellten Version auf mathematische Funktionen beschränkt. Sollten die *NN* andere Operationen berechnen, muss der *Operation Crawler* vom Modul dementsprechend erweitert werden. *PBA2CUDA* kann ohne große Einschränkungen für die neue Algorithmenklasse verwendet werden. Jedoch können die in Abschn. 4.4 aufgezeigten Optimierungen möglicherweise für den Quellcode des *NN* nicht weiter verwendet werden. Für eine optimale Parallelisierung müssen die Module für die Optimierung angepasst und/oder erweitert werden oder es wird eine einfache Parallelisierung ohne Optimierungen verwendet.

Anpassungen an andere Hardware: Für die Anpassung an alternative Hardware muss die Lauffähigkeit der Speedup-Vorhersage-Methoden und des kompilierten Quellcodes, der von *CASEP* automatisiert parallelisierten wurde, garantiert werden. Dementsprechend müssen in beiden Bereichen Anpassungen vorgenommen werden. Für das Beispiel soll das Programmierframework *OpenCL* [71] anstatt CUDA verwendet werden, mit dessen Hilfe sich *GPGPU*-Code auf unterschiedlichen GPGPU-Architekturen ausführen lässt. Für die Anpassung der Speedup-Vorhersage-Methoden müssen die CUDA spezifischen Befehle im Quellcode der Benchmarks und Micro-Benchmarks durch OpenCL-Direktiven ausgetauscht werden. Weiterhin muss bei der Kompilierung der Benchmarks der offene OpenCL-Compiler, statt des proprietären CUDA-Compilers, verwendet werden. Für die Parallelisierung (*PBA2CUDA*) sind größere Anpassungen notwendig. Der genutzte OpenMPC Compiler unterstützt nur CUDA und kann somit nicht für OpenCL verwendet werden. Zwei Vorgehensweisen sind möglich: 1) Umwandlung des CUDA-Codes in OpenCL-Code. Hierzu gibt es verschiedene Ansätze aus der Literatur [95],[96]. 2) Das Verwenden eines Compilers, ähnlich von OpenMPC, der seriellen Code in OpenCL transformieren kann. Hierzu kann beispielsweise der in Abschn. 2.5.2 beschriebene *HMPP* Compiler verwendet werden. Das Austauschen des Compilers erfordert jedoch eine komplette Restrukturierung der anderen Module von *PBA2CUDA*, die an *OpenMPC* angelehnt sind. Es müssen mögliche Fehler von *HMPP* korrigiert werden und mögliche Optimierungen für den Quellcode implementiert werden. Somit erfordert die Nutzung anderer Hardware den

größten Aufwand bei der Anpassung von *CASEP*, wobei bei einer nachträgliche Transformierung des CUDA-Codes in OpenCL-Code weniger Module in *PBA2CUDA* angepasst werden müssen.

Es wurden anhand eines theoretischen Beispiels die konkreten Schritte für eine Generalisierung aufgezeigt, basierend auf den in diesem Kapitel beschriebenen Methoden. Der Großteil der Anpassungen kann dabei ohne großen Aufwand durchgeführt werden. Lediglich die Anpassung an neue Hardware erfordert eine größere Restrukturierung von *CASEP*.

5.6 Zusammenfassung

Dieses Kapitel beschäftigte sich mit der Anpassung von *CASEP* an neue Hardware- und Software-Architekturen. Weiterhin wurde das Verwenden von *CASEP* im Rahmen von alternativen Algorithmenklassen behandelt und zuletzt das Verwenden von alternativen Programmiersprachen diskutiert. Die in diesem Kapitel durchgeführten Betrachtungen ergeben unterschiedlich komplexe Anpassungsschritte für die einzelnen Module, um die Anforderungen zu erfüllen. Die aufgezeigten Methoden wurden anhand eines theoretischen Szenarios beispielhaft diskutiert. Zusammengefasst wurde gezeigt, dass die in *CASEP* aufgezeigten Prinzipien auf andere Problembereiche generalisierbar sind.

6 Experimentelle Ergebnisse

Inhaltsangabe

In diesem Kapitel wird das in dieser Arbeit beschriebene Framework *CASEP* evaluiert. Es werden sowohl die einzelnen Module evaluiert, als auch das Gesamtframework. Bei den jeweiligen Modulen wird dabei auf verschiedene Erweiterungen eingegangen. Zuerst werden in Abschn. 6.1 die Ziele der einzelnen Versuche beschrieben. Als nächstes folgt die Definition der für die Evaluation verwendeten Metriken in Abschn. 6.2. Die Speedup-Vorhersage-Module werden in Abschn. 6.3 evaluiert und die aggregierte Vorhersage der Speedup-Module in Abschn. 6.4. Der nachfolgende Abschn. 6.5 beschäftigt sich mit den Ergebnissen der automatisierten Parallelisierung durch das *PBA2CUDA* Modul. Zuletzt werden die Ergebnisse aus diesem Kapitel in Abschn. 6.6 zusammengefasst und abschließend bewertet.

6.1 Ziele der Evaluation

Mit dem *CASEP* Framework wird der Entwickler bei zwei Hauptproblemen bei der Entwicklung von paralleler Software unterstützt: a) Das Problem der Speedup-Vorhersage, aus dem sich das Software-Hardware-Mapping ergibt, und b) das Problem der Parallelisierung. Dementsprechend wurden zwei unterschiedliche Experimente untersucht, die jeweils einen Aspekt von *CASEP* abdecken.

6.1.1 Untersuchung des vorhergesagten Speedups

Für die Untersuchung des Speedups wurden zwei *PBA* Referenzimplementierungen umgesetzt, mit denen sich der von *CASEP* vorhergesagte Speedup vergleichen lässt. Die Referenzimplementierungen wurden dabei jeweils in einer seriellen Version, einer mit OpenMP auf Multi-Cores parallelisierten Version und einer mit CUDA auf GPGPUs parallelisierten Version realisiert. Daraus ergeben sich drei unterschiedliche Laufzeiten:

1. Die serielle Laufzeit T_{Ser}
2. Die Laufzeit der OpenMP Version T_{OMP}
3. Die Laufzeit der CUDA Version T_{CUDA}

Der reale Speedup S_{OMP} für die OpenMP Version ergibt sich dementsprechend aus $S_{OMP} = \frac{T_{Ser}}{T_{OMP}}$ und der reale Speedup S_{CUDA} der CUDA Version aus $S_{CUDA} = \frac{T_{Ser}}{T_{CUDA}}$. Es wurde jeweils der Speedup des Gesamtprogramms betrachtet, wobei je nach Referenzimplementierung nur einige PBA-Operationen parallelisiert wurden (s. Abschn. 6.1.3). Somit ergibt sich die Gesamtausführungszeit T_{OMP} der OpenMP Version aus einem seriellen Teil $T_{Part\_Ser\_OMP}$ und einem parallelen Teil $T_{Part\_Par\_OMP}$:

$$T_{OMP} = T_{Part\_Ser\_OMP} + T_{Part\_Par\_OMP}$$

Gleichermaßen ergibt sich die Gesamtausführungszeit der CUDA Version aus dem seriellen Teil $T_{Part\_Ser\_CUDA}$, der auf der CPU ausgeführt wird, und dem

parallelen Teil $T_{Part\_Par\_CUDA}$:

$$T_{CUDA} = T_{Part\_Ser\_CUDA} + T_{Part\_Ser\_CUDA}$$

Die ermittelten realen Laufzeiten werden mit den Vorhersagen von *CASEP* verglichen. Dabei werden drei unterschiedliche Konfigurationen untersucht: 1) Die Vorhersage des *Benchmarkers*, 2) die Vorhersage des *Code-Analsysis*-Moduls und 3) eine aggregierte Vorhersage beider Verfahren. Die Konfigurationen werden miteinander verglichen, um die optimale Konfiguration für *CASEP* zu bestimmen.

Als Anwendungsszenario wurden Funktionsoptimierungen gewählt. Bei der Funktionsoptimierung wird versucht, analytisch oder heuristisch das Optimum einer Funktion zu bestimmen. Dabei kann das Optimum als Maximum oder Minimum der Funktion definiert sein. Insbesondere bei großen Dimensionsgrößen ist das Finden eines Optimums mit Hilfe analytischer Methoden nicht in angemessener Zeit findbar. *PBAs* sind Heuristiken, die besonders gut dazu geeignet sind, in relativ kurzer Zeit ein ausreichend gutes Ergebnis zu bestimmen [97], bei denen der Einsatz von analytischen Methoden aufgrund der Berechnungszeit nicht sinnvoll ist. Das Verwenden von Funktionsoptimierung als Anwendungsszenario hat folgende Gründe:

1. *Generalisierung*: Nahezu alle Probleme lassen sich durch Funktionen abbilden. Sollte ein Problem nicht direkt über eine Funktion abbildbar sein, lässt sich zumindest die entstehende Last über eine Funktion abstrahieren.

2. *Skalierbar*: Funktionen können anhand ihrer Dimension und den verwendeten Operatoren leicht skaliert werden. Dies erlaubt unterschiedliche Lastbetrachtungen der zu lösenden Probleme. Somit kann *CASEP* in unterschiedlichen Konfigurationen untersucht werden.

3. *Verbreitung*: Funktionsoptimierung ist in der Literatur weit verbreitet und gut analysiert. Dies erleichtert das Verwenden der richtigen Parameter für die Evaluation und ihre Reproduzierbarkeit.

Es wurden vier bekannte Benchmarkfunktionen aus der Literatur [85] unterschiedlicher Berechnungskomplexität für die Evaluation gewählt. Zur besseren Vergleichbarkeit und Orientierung, wurde die Namensgebung aus der Literatur beibehalten (f1, f5, f8, f11). Die Funktionen wurden nach folgenden Kriterien ausgewählt: 1) Komplexität der Berechnung und 2) Komplexität der Operationen. Die Kombination dieser Kriterien ergibt vier unterschiedliche Funktionstypen:

1. Funktionen mit wenigen, einfachen Operationen:

$$f_1(\vec{x}) = \sum_{i=0}^{n-1} x_i^2$$

2. Funktionen mit vielen, einfachen Operatoren:

$$f_5(\vec{x}) = \sum_{i=0}^{n-1} \left(100 \cdot (x_{i+1} - (x_i)^2)^2 + (x_i - 1)^2\right)$$

3. Funktionen mit wenigen, komplexen Operationen:

$$f_8(\vec{x}) = \sum_{i=0}^{n-1} -x_i \cdot \sin\left(\sqrt{|x_i|}\right)$$

4. Funktionen mit vielen, komplexen Operationen:

$$f_{11}(\vec{x}) = \tfrac{1}{4000} \left(\sum_{i=0}^{n-1} x_i^2\right) + \left(\prod_{i=0}^{n-1} \cos(\tfrac{x_i}{\sqrt{i+1}})\right) + 1$$

Als Dimensionen (die Laufvariable n in den beschriebenen Formeln) für die einzelnen Funktionen wurden Werte (30, 100) aus der Literatur verwendet [85], [97]. In der Literatur [98] finden sich weiterhin Problemstellungen, die größere Dimensionsgrößen (Problemgrößen) betrachten. Aus diesem Grund wurden die Funktionen ebenso für sehr große (500) Dimensionen betrachtet. Ebenso ist die Populationsgröße an Werten aus der Literatur [97] angelehnt. Für die Betrachtung der Funktionen mit 500 Dimensionen wurde die Populationsgröße ebenso auf 500 festgelegt, um eine hohe Last auf den Rechenknoten zu simulieren. Die Berechnung der Fitnessfunktion von einem Individuum zählt als eine Funktionsevaluierung. Somit werden z.B. bei einer Populationsgröße von 100 pro Iteration 100 Funktionsevaluierungen durchgeführt. Die Anzahl an durchgeführten Iterationen der *PBA* wurden anhand der Funktionsevaluierungen

Versuch	Problemgröße	Populationsgröße	Iterationszahl
1	30	100	1000
2	30	100	5000
3	30	100	10000
4	100	200	1000
5	100	200	5000
6	100	200	10000
7	500	500	1000
8	500	500	5000
9	500	500	10000

Tabelle 6.1: Simulationsparameter der einzelnen Versuche

festgelegt. In der Literatur [97] finden sich Werte von über 100.000 für die Anzahl an durchzuführenden Funktionsevaluierungen. Somit müsste ein *PBA* mit einer Populationsgröße von 100 in mindestens 1000 Iterationen ausgeführt werden. Um diese Werte zu erreichen, wurden die Versuche mit drei unterschiedlichen Iterationsgrößen durchgeführt (1000, 5000 und 10000). Aus der Kombination der Parameter ergeben sich 9 unterschiedliche Versuche für jede Funktion, die in Tabelle 6.1 abgebildet sind. Um Messfehler zu minimieren, wurden die Versuche mehrmals wiederholt. Für die Laufzeit der realen Ergebnisse wurden die Versuche 100-mal wiederholt. Aufgrund der großen Anzahl an Operationen konnte für die Prädiktoren keine so große Wiederholungszahl gewählt werden. Um die Laufzeiten in einer angemessenen Zeit durchführen zu können, wurden die Versuche des *Code-Analyse*-Moduls-50 mal wiederholt, während die Versuche des *Benchmarkers* 30-mal wiederholt wurden.

6.1.2 Untersuchung der automatisiert parallelisierten Versionen

In der Evaluation dieser Arbeit wird der automatisiert erzeugte Quellcode mit manuell optimiertem Quellcode verglichen. Für die manuelle Referenzimplementierung werden die selben Programme (s. Abschn. 6.1.3) verwendet, die auch bei dem Vergleich der Speedup-Vorhersage verwendet wurden. Es wird dabei der erzielte Speedup der automatisiert generierten Version mit dem er-

zielten Speedup der manuell implementierten Version verglichen. Weiterhin wird der von *CASEP* vorhergesagte Speedup mit dem von der automatisierten Version generierten Speedup verglichen. Für den Vergleich der Ergebnisse werden die automatisiert generierten Programme 100-mal wiederholt.

6.1.3 Referenzimplementierungen

In diesem Abschnitt werden die Charakteristika der Referenzimplementierungen beschrieben. Für die Evaluation dieser Arbeit werden zwei Algorithmen implementiert, ein *GA* (s. Abschn. 2.2.1) und ein *PSO* (s. Abschn. 2.2.2).

Genetischer Algorithmus: Beim *GA* wurde eine Referenzimplementierung erstellt, bei der sich für bestimmte Problemgrößen eine Parallelisierung lohnt. Dafür wurden unterschiedliche Implementierungen untersucht, bei denen verschiedene Abschnitte parallelisiert wurden. Für die in dieser Arbeit untersuchten Funktionen ergaben Messungen nur einen Geschwindigkeitsvorteil bei den Implementierungen, in denen nur die Fitnessberechnung parallelisiert wurde. Andere Implementierungen erzielten keinen Speedup bzw. nur bei sehr großen (> 10000 Individuen) Populationsgrößen. Die für OpenMP und CUDA Version verwendete Partitionierung ist in Abb. 6.1 dargestellt.

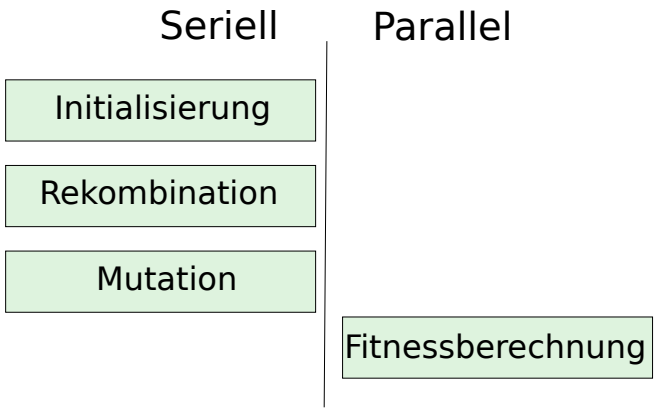


Abbildung 6.1: Partitionierung des Referenz-GAs

Particle Swarm Optimization: Ähnlich wie beim Referenz-*GA* wurden für den *PSO* unterschiedliche Implementierungen untersucht. Für die unter-

suchten Funktionen ergab eine Teilparallelisierung längere Laufzeiten des Programms, als bei einer vollständigen Parallelisierung. Aus diesem Grund wurden, bis auf die Initialisierung, alle Funktionen des *PSOs* parallelisiert (s. Fig. 6.2). Die Aufteilung der Algorithmen ist nicht allgemeingültig und architekturabhängig und gilt für die in dieser Arbeit verwendete Versuchsumgebung (s. Abschn. 6.1.4). In einer anderen Versuchsumgebung, in der z.B. die Operationen der *Mutation*-Operation des *GAs* schneller auf der GPGPU ausgeführt werden können, ist unter Umständen eine andere Partitionierung zu wählen. Die Laufzeit einer kompletten Parallelisierung ist weitaus komplexer vorauszu-

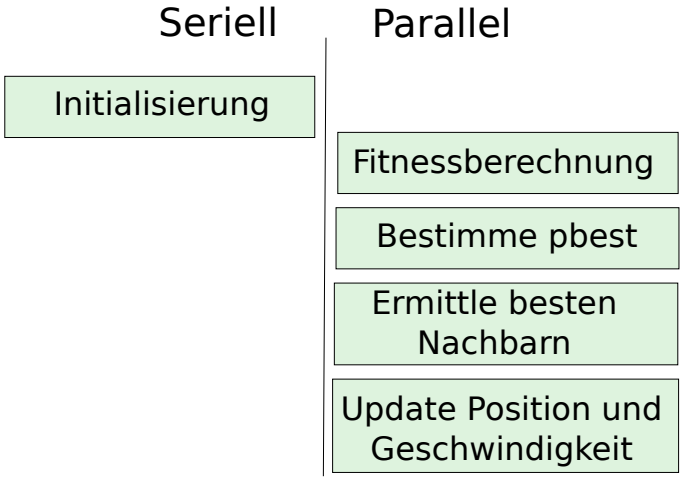


Abbildung 6.2: Partitionierung des Referenz-PSOs

sagen, da Cache- und Pipeline-Effekte für das gesamte Programm einen größeren Einfluss haben können, als für kleinere Abschnitte. Durch die zwei unterschiedlichen Referenzimplementierungen wurden zwei unterschiedlich komplexe Vorhersageszenarien untersucht, das einfachere *GA*-Szenario mit nur einer parallelen Funktion und das komplexere *PSO*-Szenario mit nur einer seriellen Funktion und vier parallelen Funktionen.

Insgesamt wurden für den *GA* und den *PSO* zusammen 72 unterschiedliche Versuche durchgeführt. Dies ergibt sich aus den neun unterschiedlichen Versuchsparametern (s. Tabelle 6.1), die für jede der vier Funktionen und der beiden *PBAs* untersucht wurden ($9 \cdot 4 \cdot 2$).

6.1.4 Versuchsumgebung

Als Versuchsumgebung diente der Rechencluster des Fachgebiets *System- und Rechnerarchitektur* des Instituts für *Systems Engineering* an der Leibniz Universität Hannover. Die Hardwarekonfiguration ist in Tabelle 6.2 abgebildet.

Hardware	Anzahl
Anzahl an Rechenknoten	17
Six-Core X5650 Intel CPU, 2,66 GHz	2 pro Rechenknoten
M2075 GPGPU	2 pro Rechenknoten
Anzahl CUDA Cores	448 pro GPGPU
DDR3-13333 RAM	48 GB pro Rechenknoten

Tabelle 6.2: Hardwareparameter des Rechencluster für die Simulationen

Als Betriebssystem ist ein 64 bit *Linux CentOS 6.5* installiert. Zur Kompilierung des C/C++ Codes wurde der *gcc* Compiler in der Version 4.6.4 verwendet. Für die Kompilierung des CUDA Codes wurde der NVIDIA Compiler *nvcc* in der Version 4.2 verwendet.

6.2 Metriken

Für die beiden in Abschn. 6.1 beschriebenen Versuchsaufbauten wurden unterschiedliche Metriken zur Evaluation der Ergebnisse verwendet. Zur Bewertung des vorausgesagten Speedups und der Güte des vorhergesagten Software-Hardware-Mappings wurden folgende Metriken verwendet:

1. *Primäre Metrik - Mapping-Vorhersage*: Die Hauptmetrik für die Evaluation ist das korrekte Vorhersagen des Software-Hardware-Mappings (s. Abschn. 3.1.1). *CASEP* gibt drei Speedup-Werte für das eingegebene Programm aus. Den Speedup des seriellen Programms (Referenzprogramm), welcher immer eins ist, den Speedup der OpenMP Version und den Speedup der CUDA Version (s. Abb. 6.3). Daraus ergibt sich das vorhergesagte Mapping aus dem größten vorhergesagten Speedup der drei Versionen. Es sind drei potentielle Mappings möglich: 1) Ein serielles Mapping (*Serial Area*), ein paralleles Mapping mit Hilfe von OpenMP

(*OMP Area*) und ein paralleles Mapping mit Hilfe von CUDA (*CUDA Area*). Das vorhergesagte Mapping wird im folgenden Schritt mit dem aus dem Versuchsaufbau (Abschn. 6.1) beschriebenen realen Mapping verglichen. Bei einem korrekten Mapping gilt die Metrik als erfüllt, ansonsten als fehlgeschlagen.

2. *Sekundäre Metrik - Vorhersagegenauigkeit*: Die sekundäre Metrik ist notwendig, um Vorhersagen miteinander vergleichen zu können, die das selbe Mapping vorausgesagt haben. In diesem Fall wird die Abweichung des vorausgesagten Speedups zum realen Speedup (prozentuale Abweichung) als sekundäre Metrik verwendet, damit die Güte des Mappings bestimmt werden kann. Die prozentuale Abweichung $A_{speedup}$ berechnet sich folgendermaßen:

$$A_{speedup} = \frac{|S-G|}{G} \cdot 100 \text{ (Angaben in \%)}$$

Dabei ist S der vorhergesagte Speedup und G der reale Speedup (Grundwert). Je geringer die Abweichung ist, desto besser ist die Vorhersage.

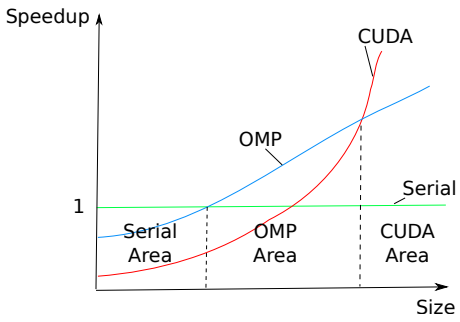


Abbildung 6.3: Software-Hardware-Mapping Anhand des vorhergesagten Speedups

Für die Untersuchung der von *CASEP* automatisiert erzeugten parallelen *PBA* Versionen wurden zwei Metriken verwendet:

1. *Abweichung zur manuellen Version:* In dieser Metrik wird die Ausführungszeit der automatisiert erzeugten Version bewertet. Hierzu wird die Abweichung des erzielten Speedups zum Speedup der manuell erzeugten

Version berechnet. Ähnlich wie bei den Metriken zur Beurteilung des vorhergesagten Speedups wurde auch hier die prozentuale Abweichung $A_{pba2cuda}$ des Speedups berechnet:

$$A_{pba2cuda} = \frac{|S-G|}{G} \cdot 100 \text{ (Angaben in \%)}$$

Dabei ist S der erzielte Speedup der automatisierten Version und G der Speedup der manuell erzeugten Version. Je geringer die Abweichung, desto höher ist die Güte der automatisiert erzeugten Version einzuordnen.

2. *Genauigkeit zum vorhergesagten Speedup:* Diese Metrik bewertet die Synergie der einzelnen Module von *CASEP*. Es wird der vorhergesagte Speedup durch *CASEP* mit dem erzielten Speedup der von *CASEP* erzeugten parallelen Version verglichen. Sollte die gleiche automatisiert parallelisierte Version (OpenMP oder CUDA) den schnellsten Speedup erzielen, wie die durch *CASEP* vorausgesagte Version, gilt die Metrik als erfüllt.

6.3 Evaluierung der Speedup-Prädiktoren

In diesem Abschnitt werden die zwei Speedup-Prädiktoren *Benchmarking* und *Code-Analysis* von *CASEP* evaluiert. Das *Benchmarking*-Modul wird in zwei unterschiedlichen Versionen evaluiert, einmal mit *Enhanced Function Synthesizer* (EFS) und einmal ohne *EFS*. Dabei bedeutet der aktivierte *EFS*, dass beim *Function Synthesizer* (s. Abschn. 4.2.1) zusätzliche Informationen über die Art der vorhandenen Operationen mit angegeben werden, wie im zugehörigen Abschnitt beschrieben wurde. Beispielsweise kann der Entwickler, außer der Komplexitätsklasse, die Anzahl an Kosinus-Operationen mit angeben. Die drei Versionen werden miteinander anhand der in Abschn. 6.2 beschriebenen Metriken ihrer Leistung bezogen auf den beiden beschriebenen Referenzimplementierungen (s. Abschn. 6.1.3) verglichen. Zuletzt wird die Leistungsfähigkeit von *CASEP* anhand einer gemeinsamen Vorhersage, basierend auf der Vorhersage des *Benchmarkers* und *Code-Analysis*, untersucht.

6.3.1 Evaluationsszenario 1 - GA

In diesem Abschnitt werden die Ergebnisse des Szenarios mit dem *GA* untersucht. Es werden exemplarisch die Ergebnisse der Funktion f8 (s. Abschn. 6.1.1) im Detail diskutiert. Diese liegt gemessen an den durchgeführten Operationen in ihrer Komplexität zwischen der komplexesten Funktion f11 und der einfachen Funktionen f1. Sollte sich der Trend der Ergebnisse zwischen den Funktionen unterscheiden, wird an der jeweiligen Stelle darauf eingegangen. Darüber hinaus werden am Ende des Abschnitts die Ergebnisse aller Funktionen tabellarisch zusammengefasst und diskutiert.

6.3.1.1 Repräsentative Betrachtung der Funktion f8

Die Ergebnisse der Funktion f8 werden, wie in Abschn. 6.1 beschrieben, in drei Konfigurationen betrachtet:

1. Populationsgröße 100 und Problemgröße (Dimensionsgröße) 30.
2. Populationsgröße 200 und Problemgröße 100.
3. Populationsgröße 500 und Problemgröße 500.

Für jede der drei Konfigurationen wird das Vorhersageverhalten der einzelnen Module bei 1000 Iterationen des betrachteten Algorithmus, 5000 Iterationen und 10000 Iterationen analysiert. In Abb. 6.4 sind die Ergebnisse für Konfiguration 1 abgebildet, in Abb. 6.5 die Ergebnisse für Konfiguration 2 und in Abb. 6.6 die Ergebnisse für Konfiguration 3. In jeder der drei Abbildungen sind die Vorhersagen für das *Benchmarking*-Modul ohne *EFS* (obere Abbildung), die Vorhersagen des *Benchmarking*-Moduls mit *EFS* (mittlere Abbildung) und die Vorhersagen des *Code-Analysis*-Moduls (untere Abbildung) dargestellt. Dabei ist auf der Y-Achse der vorhergesagte bzw. reale Speedup abgebildet und auf der X-Achse die Anzahl an Iterationen, die der zu untersuchende Algorithmus ausgeführt wird. Der reale Speedup ist der Speedup, der von der manuell erstellen Referenzimplementierung erreicht wird.

Deutlich ist für alle drei Konfigurationen die Verbesserung der Vorhersagegenauigkeit abhängig von den durchgeführten Iterationen und Problemgrößen

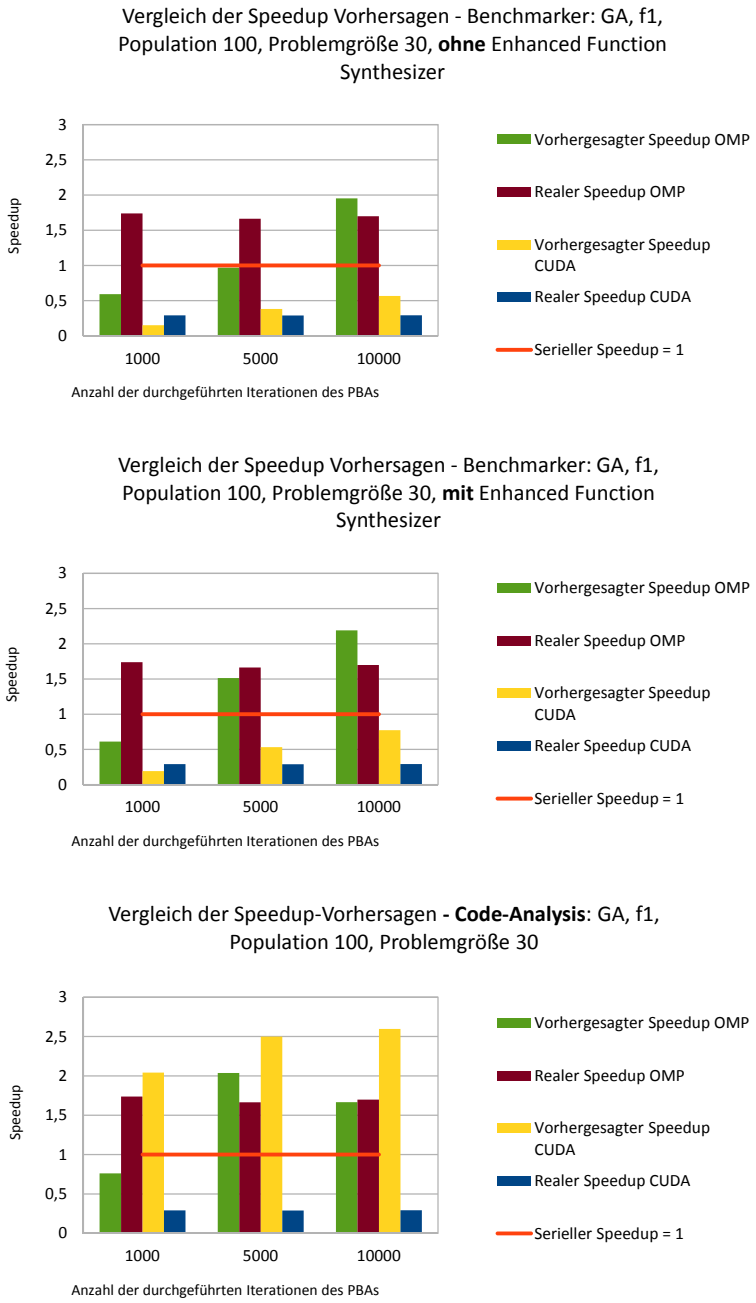


Abbildung 6.4: Vergleich: Funktion f_8 mit Populationsgröße 100 und Problemgröße 30

bzw. Populationsgrößen zu erkennen. Der Grund für dieses Verhalten ist die relativ kurze Laufzeit des Algorithmus bei kleinen Iterationszahlen und Problemgrößen. Für 1000 Iterationen in der Konfiguration 1 liegt die Laufzeit des Algorithmus im einstelligen Sekundenbereich, während die Laufzeit bei Konfiguration 3 und 10000 Iterationen im Minutenbereich liegt. Bei solch kurzen Laufzeiten wie bei Konfiguration 1 haben Ungenauigkeiten bei der Vorhersage einen weitaus größeren Einfluss und führen zu vergleichsweise ungenauen Ergebnissen. In der Regel werden Programme mit höheren Laufzeiten parallelisiert, bei denen der relative Geschwindigkeitsgewinn einer parallelen Version größer ausfällt. Dennoch sind die Ergebnisse für Programme mit kurzen Laufzeiten nicht zu vernachlässigen. In Szenarien, bei denen sehr häufig Programme mit kurzer Ausführungszeit benötigt werden, kann auch ein relativ geringer Geschwindigkeitsgewinn in der Summe relevant sein. Dies kann z.B. bei logistischen Problemen [99] der Fall sein, bei denen bestimmte Aufgaben immer wieder durchgeführt werden müssen, die jedoch für sich genommen eine sehr kurze Ausführungszeit besitzen.

Auffallend ist das Überschätzen der GPGPU-Performance des *Code-Analysis*-Moduls (z.B. $229,68\%$ im Falle von 10000 Iterationen und Populationsgröße 100, Problemgröße 30). Dieses Verhalten lässt sich bei jeder der durchgeführten Evaluationen beobachten. Der Grund dafür liegt im Aufbau der generierten Micro-Benchmarks. Während beim *Benchmarking*-Modul das konkrete Programm und die notwendigen Steueroperationen (z.B. if-Anweisungen) modelliert werden können, generiert das *Code-Analysis*-Modul Micro-Benchmarks, bei denen die vorhandenen Operationen der Reihe nach berechnet werden. Dadurch ergibt sich ein optimaler Zugriff auf die Speicherbereiche der Variablen. Der korrekte Zugriff auf die Speicherbereiche der GPGPU kann das Programm erheblich beschleunigen [36]. Es wurde versucht, Cache-Effekte und Speicherzugriffseffekte beim *Code-Analysis*-Modul weitestgehend zu minimieren (s. Abschn. 4.2.2), dennoch bilden die Micro-Benchmarks den optimalen Fall bei der Ausführung auf der GPGPU. Bei den OpenMP-Micro-Benchmarks tritt der Effekt nicht auf. Dies ergibt sich aus dem relativ einfachen Aufbau des OpenMP-Codes, der überwiegend identisch zum seriellen Quellcode ist. Somit lassen sich OpenMP-Benchmarks weitaus einfacher automatisiert erstellen, die

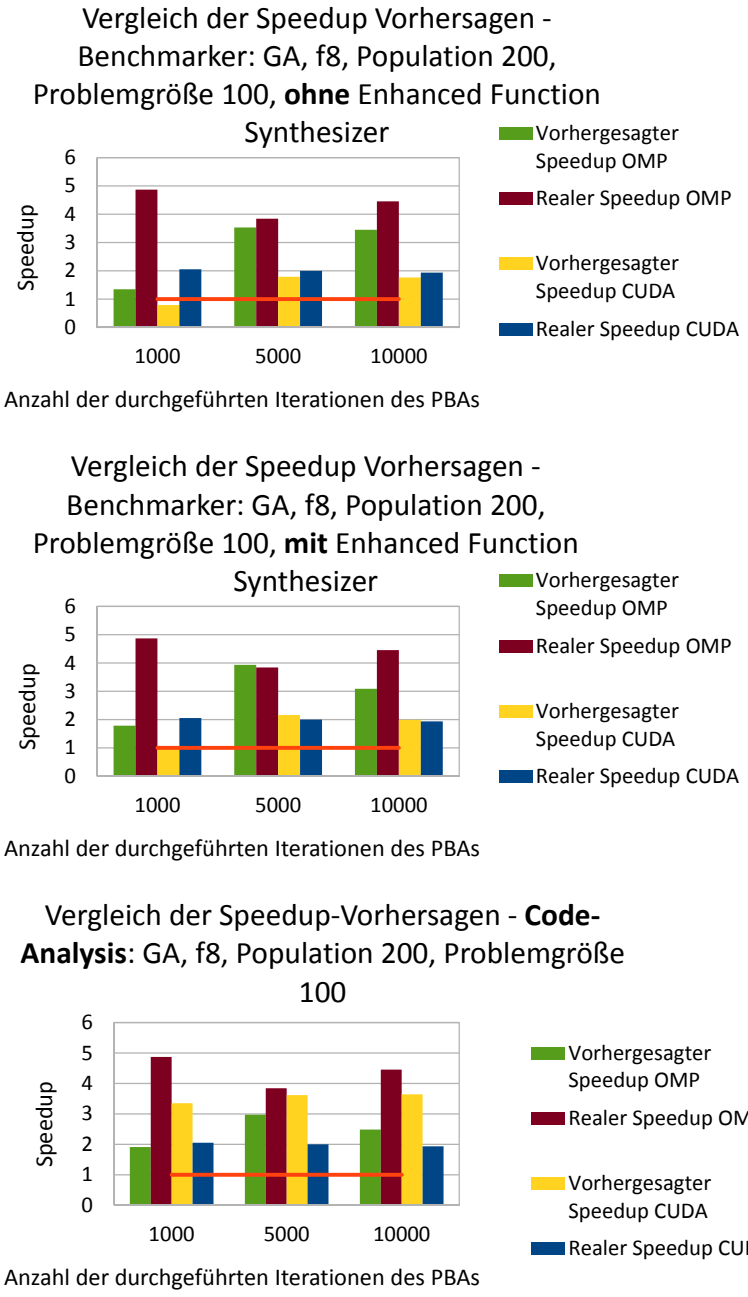
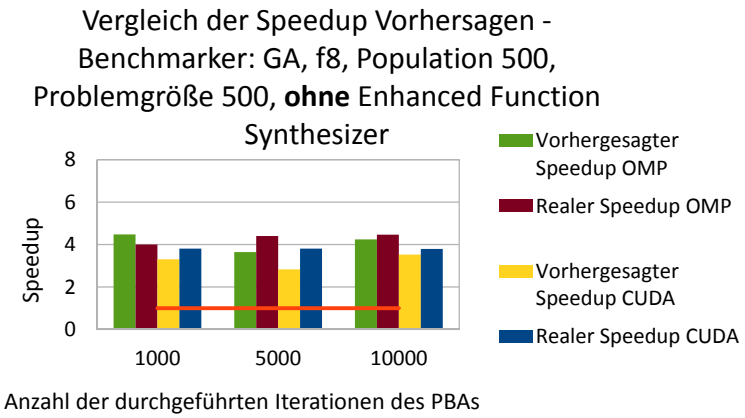
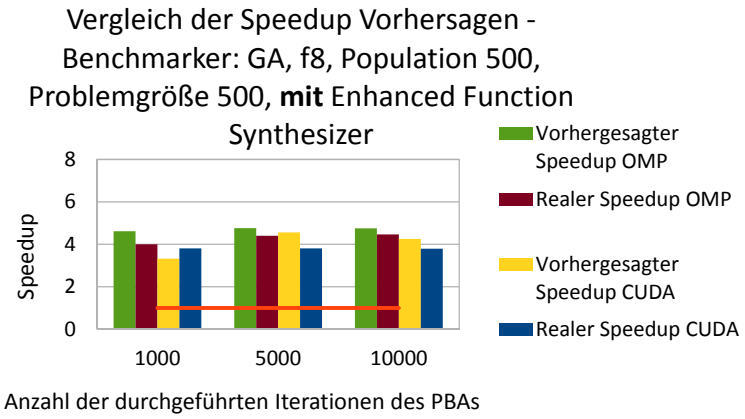


Abbildung 6.5: Vergleich: Funktion f_8 mit Populationsgröße 200 und Problemgröße 100

das Verhalten von manuell erstelltem OpenMP-Code simulieren.

Beim Vergleich der beiden *Benchmarking*-Versionen fällt die etwas genauere Vorhersage des Moduls mit aktivierten *EFS* auf. Bei fast allen Problem- und Populationsgrößen fällt der Vorhersagefehler der Version mit *EFS* geringer aus. Der durchschnittliche Fehler (sekundäre Metrik; s. Abschn. 6.2) bei der OpenMP-Vorhersage liegt für das *GA*-Szenario bei der Version ohne *EFS* bei $36,25\%$ (über alle Versuche betrachtet), während sie bei der Version mit *EFS* bei $26,58\%$ liegt. Bei der CUDA-Vorhersage beträgt die Abweichung $33,06\%$, bzw. $23,85\%$. Insgesamt liegt die durchschnittliche Abweichung bei der Version ohne *EFS* bei $34,66\%$ und bei der Version mit *EFS* bei $25,22\%$. Allgemein für alle Funktionen lässt sich dieser Trend jedoch nicht bestätigen (s. Anhang A). Im Durchschnitt liegt der Fehler der Vorhersage bei der Version ohne *EFS* mit $47,52\%$ sogar niedriger, als bei der Version mit *EFS* ($49,94\%$). Der Grund dafür ist in Abb. 6.7 abgebildet. Der *Function Synthesizer* (s. Abschn. 4.2.1) des *Benchmarking*-Moduls wählt aus einer Menge von Funktionen aus der vom Entwickler angegebenen Komplexitätsklasse ($\mathcal{O}(n)$, $\mathcal{O}(n^2)$, ...) eine zufällige Funktion für jede Evaluation. Dabei kann es sich um vergleichsweise einfache Funktionen (beispielsweise ähnlich der hier vorgestellten Funktion f1) handeln oder komplexere Funktionen (beispielsweise ähnlich der Funktion f11). Die Evaluationen des *Benchmarkers* werden eine festgelegte Anzahl (in diesem Fall 30 mal) wiederholt, um Messfehler auszugleichen. In jeder Evaluation wählt der Benchmarker dabei eine andere Funktion aus. Dadurch ergibt sich eine mittlere Komplexität, die in einem Bereich zwischen den einfachen und komplexen Funktionen liegt, wie in der Abb. 6.7 abgebildet ist. Der *EFS* modifiziert die anfänglich ausgesuchten Funktionen durch zusätzliche Operationen, die der Entwickler angegeben hat. Durch diese Operationen werden die vom *Function Synthesizer* ausgewählten Funktionen ähnlicher zu der Funktion des Eingabeprogramms modelliert. Allerdings erhöht sich ebenso die Komplexität in Richtung zu den komplexeren Funktionen aus dem *Functions-Pool* (s. den markierten Bereich in der Abbildung 6.7). Es werden somit Funktionen vom *EFS* für die Benchmarks generiert, die ähnlicher zu der Funktion des Ursprungsprogramms sind, sich jedoch in der Grundkomplexität unterscheiden können. Je mehr sich der Bereich der generierten Funktionen von





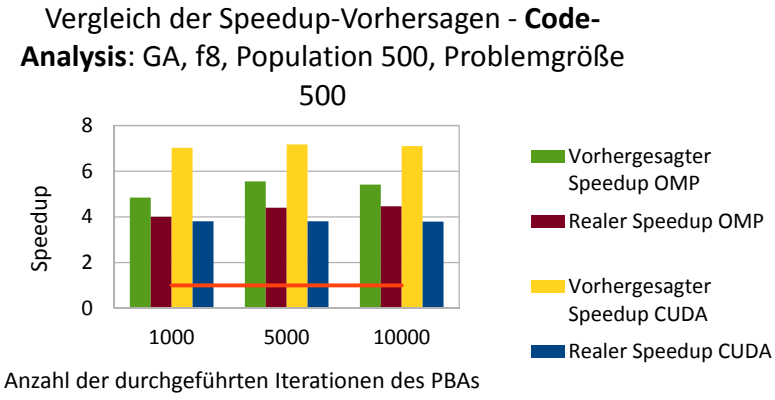


Abbildung 6.6: Vergleich: Funktion f_8 mit Populationsgröße 500 und Problemgröße 500

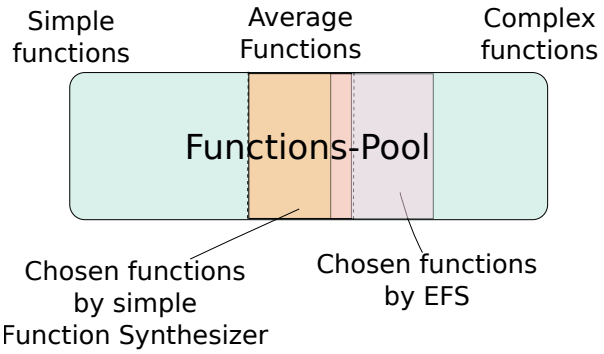


Abbildung 6.7: EFS-Auswahlverfahren der Funktionen

dem Bereich, in dem sich die Ursprungsfunktion befindet, unterscheidet, desto ungenauer werden die generierten Benchmarkarks des *Benchmarking*-Moduls. Zur Verringerung dieses Verhaltens sind mehrere Ansätze möglich:

1. Erhöhung der Anzahl der Funktionen im *Functions-Pool*: Durch eine Erhöhung der Anzahl an Funktionen, aus denen der *Benchmarker* wählt, entsteht ein größerer Abstand in der Komplexität zwischen simplen Funktionen und komplexen Funktionen. Dadurch entsteht eine geringere Verschiebung der Komplexität beim Hinzufügen der Operationen durch den *EFS*.

2. Verschiebung in beide Richtungen erlauben: In der vorliegenden Version verschiebt sich die Komplexität in Richtung der komplexeren Funktionen. Denkbar ist auch eine Verschiebung zu einfacheren Funktionen. Dies kann beispielsweise dadurch erreicht werden, dass der Entwickler bestimmte Operationen ausschließt, die dann von den Benchmark-Funktionen entfernt werden.

3. Verzicht auf den *Functions-Pool:* Bei dieser Implementierung wird für das Erzeugen der Funktionen für den *Benchmarker* komplett auf Funktionen aus dem *Functions-Pool* verzichtet. Es werden nur Informationen vom Entwickler verwendet: Je genauer die Funktion des Eingabeprogramms beschrieben wird, desto genauer wird sie von den Funktionen aus dem *Benchmarker* abgebildet.

Die drei beschriebenen Methoden haben unterschiedliche Vor- und Nachteile. Die Erhöhung der Anzahl der Funktionen im *Functions-Pool* ist am einfachsten zu implementieren. Sie hat jedoch den geringsten Einfluss bei der Verschiebung (s. Abb. 6.7) im Komplexitätsbereich. Die zweite Methode hat den Vorteil, dass sie skalierbar ist. Sie funktioniert ohne zusätzliche Eingaben vom Entwickler und ist in diesem Fall identisch mit dem in dieser Arbeit beschrieben Verfahren. Je mehr Eingaben der Entwickler zur Funktion machen kann, desto genauer wird diese Methode. Die letzte Methode (Verzicht auf den *Functions-Pool*) benötigt die meisten Eingaben vom Entwickler und kann somit nicht in jeder Situation eingesetzt werden. Sollte zum Zeitpunkt der Ausführung des *Benchmarkers* keine Information über die Funktion des Eingabeprogramms bekannt sein, kann diese Methode nicht verwendet werden. Beim Vorhandensein jeglicher Informationen der Eingabefunktion ist diese Methode jedoch die genaueste der drei beschriebenen, da genau die Funktion gebenchmarkt wird, die auch im tatsächlichen Eingabeprogramm vorkommt.

6.3.1.2 Untersuchung des vorhergesagten Mappings und des Vorhersagefehlers beim GA

In diesem Abschnitt wird das vorhergesagte Mapping des *Benchmarkers* ohne *EFS*, des *Benchmarkers* mit *EFS* und des *Code-Analysis-Moduls* verglichen (primäre Metrik; s. Abschn. 6.2). Weiterhin wird die prozentuale Abweichung der Speedup-Vorhersagen zum realen Speedup der manuell erzeugten Programme verglichen (sekundäre Metrik). Die Ergebnisse des Mappings sind in den Tabellen 6.3, 6.5, 6.7 und 6.9 zusammengefasst. In jeder Zeile sind die Ergebnisse der drei Vorhersage-Methoden und das optimale Mapping (reales Mapping), beruhend auf den Laufzeiten der realen Programme, dargestellt. Ein korrektes Mapping ist grün markiert, während falsche Mapping-Vorhersagen rot markiert sind. Es sind jeweils die Ergebnisse aller drei Konfigurationen von Populationsgröße und Problemgröße für die drei untersuchten Iterationszahlen dargestellt (s. Abschn. 6.1). Die prozentuale Abweichung der Vorhersagen ist in den Tabellen 6.4, 6.6, 6.8 und 6.10 dargestellt. Der Grundaufbau der Tabellen ist identisch mit denen der Mapping-Ergebnisse. In jeder Zelle der Tabelle

sind drei Werte abgebildet: Die Gesamtabweichung (oberer Wert), die Abweichung der OpenMP-Vorhersage (unterer, linker Wert) und die Abweichung der CUDA-Vorhersage (unterer, rechter Wert). In jeder Zeile sind die besten Werte der drei Vorhersage-Methoden grün markiert (Gesamtabweichung, OpenMP-Abweichung, CUDA-Abweichung).

Die Mapping-Vorhersage des *Code-Analysis*-Moduls ist bei nahezu allen Versuchen falsch. Dies liegt an der überschätzten CUDA-Vorhersage, auf die im letzten Abschnitt bereits eingegangen wurde. Betrachtet man den Fehler der OpenMP-Vorhersage, sieht man, dass dieser bei der Code-Analyse ähnlich gering ausfällt, wie bei den beiden *Benchmarking*-Versionen. Aufgrund der schlechten CUDA-Vorhersage ist jedoch der Gesamtvorhersagefehler bei der Code-Analyse größer.

Die beiden *Benchmarking*-Versionen besitzen eine ähnliche Vorhersagegenauigkeit. Während das Modul ohne *EFS* 30 von 36 Mapping-Vorhersagen korrekt berechnet, ist das Modul mit *EFS* um eine Vorhersage (31 von 36) geringfügig besser. Beide Module sagen bei hohen Problem- und Populationsgrößen das Mapping korrekt voraus und haben lediglich bei sehr kleinen Laufzeiten (vgl. dazu die zuvor diskutierten Probleme bei kleinen Populations- und Problemgrößen) Schwierigkeiten, das korrekte Mapping zu bestimmen. Anhand des Mappings und des prozentualen Vorhersagefehlers lässt sich bei keiner der beiden Methoden ein signifikanter Vorteil erkennen. Auffallend ist, dass bei sehr großen Populationsgrößen (500) und Problemgrößen (500) das Modul ohne *EFS* durchgehend einen geringeren Fehler aufweist. Dies hängt mit der Verschiebung der Komplexität der Funktion zusammen, die im letzten Abschnitt erklärt wurde. Bei sehr großen Populationsgrößen hat diese Verschiebung einen größeren Einfluss auf die Gesamtlast auf der Maschine (CPU, GPGPU). Durch die höhere Last ergibt sich ein Vorteil der parallelen Ausführungen gegenüber der seriellen Ausführung, was zu höheren Speedup-Vorhersagen des Moduls mit *EFS* führt.

Zusammenfassend lässt sich eine deutlich bessere Leistung bei den *Benchmarking*-Modulen gegenüber dem *Code-Analysis*-Modul feststellen. Insbesondere bei der Vorhersage des CUDA-Speedups ist der Unterschied signifikant. Beim Vergleich der Module mit und ohne *EFS* lässt sich kein entscheidender

Funktion f1	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis	Reales Mapping
Pop: 100 Prob: 30	1000	seriell	seriell	CUDA	OMP
	5000	seriell	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP
Pop: 200 Prob: 100	1000	OMP	OMP	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP
Pop: 500 Prob: 500	1000	OMP	OMP	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP

Tabelle 6.3: Mapping-Voraussagen für den GA - Funktion f1

Vorteil bei einer der beiden Versionen erkennen.

Funktion f1	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis
Pop: 100 Prob: 30	1000	56,99%	49,37%	328,70%
		65,94% | 48,03%	64,82% | 33,92%	56,22% | 601,17%
	5000	37,01%	46,46%	392,96%
		41,79% | 32,23%	9,05% | 83,87%	22,45% | 763,47%
	10000	54,55%	96,79%	395,41%
		14,92% | 94,17%	28,95% | 164,63%	1,94% | 788,88%
Pop: 200 Prob: 100	1000	32,54%	5,18%	180,15%
		28,64% | 36,45%	5,23% | 5,12%	15,40% | 344,89%
	5000	53,84%	122,93%	218,17%
		33,53% | 74,15%	88,60% | 157,26%	46,84% | 389,51%
	10000	114,13%	75,64%	229,34%
		78,41% | 149,85%	31,12% | 120,15%	50,15% | 408,53%
Pop: 500 Prob: 500	1000	157,93%	161,56%	203,76%
		164,08% | 151,78%	159,79% | 163,33%	91,39% | 316,13%
	5000	139,70%	193,88%	211,22%
		118,48% | 160,93%	155,09% | 232,68%	97,46% | 324,98%
	10000	97,40%	161,72%	210,81%
		74,24% | 120,56%	127,35% | 196,09%	89,98% | 331,64%

Tabelle 6.4: Durchschnittliche prozentuale Abweichung der Vorhersagen für den GA - Funktion f1

Funktion f5	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis	Reales Mapping
Pop: 100 Prob: 30	1000	seriell	seriell	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	seriell	OMP	CUDA	OMP
Pop: 200 Prob: 100	1000	OMP	OMP	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP
Pop: 500 Prob: 500	1000	OMP	OMP	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP

Tabelle 6.5: Mapping-Voraussagen für den GA - Funktion f5

Funktion f5	Iterationen	Benchmarker ohne EFS		Benchmarker mit EFS		Code-Analysis	
Pop: 100 Prob: 30	1000	52,82%		58,12%		139,86%	
		87,73%	17,91%	53,95%	62,30%	2,96%	276,75%
	5000	26,75%		33,34%		197,08%	
		31,11%	22,40%	51,60%	15,09%	28,28%	365,89%
	10000	38,99%		34,74%		212,53%	
		74,11%	3,87%	58,34%	11,15%	32,13%	392,94%
Pop: 200 Prob: 100	1000	53,20%		51,27%		81,43%	
		36,13%	70,26%	42,79%	59,74%	14,52%	148,35%
	5000	33,52%		14,08%		84,60%	
		33,79%	33,26%	23,13%	5,03%	3,81%	165,38%
	10000	16,24%		5,16%		92,39%	
		13,79%	18,69%	0,64%	9,68%	8,43%	176,36%
Pop: 500 Prob: 500	1000	11,37%		38,94%		91,37%	
		21,56%	1,18%	39,32%	38,56%	18,07%	164,67%
	5000	3,68%		12,75%		101,35%	
		2,83%	4,54%	10,66%	14,85%	27,77%	174,94%
	10000	4,07%		14,94%		102,80%	
		2,91%	5,24%	7,05%	22,83%	30,55%	175,04%

Tabelle 6.6: Durchschnittliche prozentuale Abweichung der Vorhersagen für den GA - Funktion f5

Funktion f8	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis	Reales Mapping
Pop: 100 Prob: 30	1000	seriell	seriell	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	OMP	OMP	OMP
Pop: 200 Prob: 100	1000	OMP	OMP	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP
Pop: 500 Prob: 500	1000	OMP	OMP	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP

Tabelle 6.7: Mapping-Voraussagen für den GA - Funktion f8

Funktion f8	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis
Pop: 100 Prob: 30	1000	55,52%	59,13%	78,10%
		83,57% | 27,47%	62,69% | 55,57%	1,35% | 154,85%
	5000	60,67%	28,19%	133,17%
		64,70% | 56,64%	22,78% | 33,59%	52,94% | 213,40%
	10000	63,82%	24,01%	121,79%
		40,97% | 86,66%	27,38% | 20,63%	13,89% | 229,68%
Pop: 200 Prob: 100	1000	66,91%	56,36%	61,91%
		72,34% | 61,48%	63,36% | 49,36%	60,72% | 63,10%
	5000	9,28%	5,23%	51,76%
		8,07% | 10,50%	2,41% | 8,04%	22,59% | 80,92%
	10000	15,74%	16,73%	66,18%
		22,56% | 8,91%	30,66% | 2,80%	44,17% | 88,19%
Pop: 500 Prob: 500	1000	12,58%	14,07%	52,78%
		11,92% | 13,23%	15,37% | 12,76%	21,10% | 84,46%
	5000	21,42%	13,94%	57,25%
		17,18% | 25,66%	8,15% | 19,73%	26,16% | 88,34%
	10000	5,98%	9,30%	54,23%
		4,94% | 7,02%	6,43% | 12,16%	21,27% | 87,20%

Tabelle 6.8: Durchschnittliche prozentuale Abweichung der Vorhersagen für den GA - Funktion f8

Funktion f11	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis	Reales Mapping
Pop: 100 Prob: 30	1000	seriell	seriell	CUDA	OMP
	5000	OMP	seriell	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP
Pop: 200 Prob: 100	1000	OMP	OMP	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP
Pop: 500 Prob: 500	1000	OMP	OMP	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP

Tabelle 6.9: Mapping-Voraussagen für den GA - Funktion f11

Funktion f11	Iterationen	Benchmarker ohne EFS		Benchmarker mit EFS		Code-Analysis	
Pop: 100 Prob: 30	1000	82,89%		72,80%		131,04%	
		86,21%	79,57%	72,81%	72,80%	66,35%	195,73%
	5000	53,50%		66,57%		159,94%	
		64,86%	42,15%	76,36%	56,78%	69,86%	250,01%
	10000	49,69%		26,57%		153,45%	
		61,94%	37,43%	33,55%	19,59%	43,31%	263,59%
Pop: 200 Prob: 100	1000	71,02%		56,53%		73,04%	
		70,13%	71,91%	53,37%	59,70%	50,39%	95,69%
	5000	29,92%		40,92%		78,74%	
		22,21%	37,63%	44,99%	36,85%	31,35%	126,13%
	10000	25,53%		26,07%		71,33%	
		22,54%	28,53%	31,24%	20,91%	23,80%	118,85%
Pop: 500 Prob: 500	1000	33,25%		37,35%		59,27%	
		24,20%	42,29%	30,22%	44,48%	4,07%	114,47%
	5000	45,34%		27,05%		65,12%	
		39,56%	51,12%	21,12%	32,97%	7,02%	123,22%
	10000	23,04%		40,28%		65,02%	
		21,06%	25,02%	35,40%	45,15%	10,10%	119,95%

Tabelle 6.10: Durchschnittliche prozentuale Abweichung der Vorhersagen für den GA - Funktion f11

6.3.2 Evaluationsszenario 2 - PSO

Dieser Abschnitt beschreibt die Ergebnisse des Szenarios mit dem *PSO*-Algorithmus. Entsprechend der Evaluation des *GAs* werden die Ergebnisse der Funktion f8 im Detail beschrieben. Ebenso werden die Ergebnisse aller Funktionen am Ende des Abschnitts tabellarisch zusammengefasst.

6.3.2.1 Repräsentative Betrachtung der Funktion f8

Der *PSO*-Algorithmus wurde, anders als der *GA*, nahezu komplett parallelisiert (s. Abschn. 6.1.3). Dadurch ergibt sich für hohe Laufzeiten ein größerer Speedup bei der CUDA-Implementierung, als dies bei der OpenMP-Implementierung der Fall war. In Abb. 6.8, 6.9 und 6.10 sind die Ergebnisse der drei Evaluationskonfigurationen für Funktion f8 abgebildet. Der Aufbau der Diagramme ist identisch mit denen der *GA*-Betrachtung aus dem vorangegangen Abschnitt.

Durch die zum *GA* vergleichsweise bessere Leistung der realen CUDA-

Implementierung sind die Vorhersagen des *Code-Analysis*-Moduls beim *PSO* genauer. Insbesondere bei kleinen Problem- und Populationsgrößen ist bei den meisten durchgeführten Versuchen die Vorhersage genauer als die von den beiden untersuchten *Benchmarking*-Modulen. Je größer die untersuchte Populations- und Problemgröße gewählt wird, desto ungenauer wird die CUDA-Vorhersage der Code-Analyse. Der Grund für dieses Verhalten ist das Überschätzen des CUDA-Speedups durch das Modul, was sich bei den meisten untersuchten Funktionen in Relation zur Problemgröße vergrößert. Bei der Funktion f1 (s. Anhang B.1) lässt sich der umgekehrte Fall beobachten. Dieser ergibt sich aus dem sehr großen Speedup der realen CUDA-Version, bei dem das Überschätzen des *Code-Analysis*-Moduls keine negativen Auswirkungen auf die Vorhersage hat. Die OpenMP-Vorhersage des Moduls ist vergleichbar mit der Vorhersage bei den GA-Versuchen aus dem vorangegangenen Abschnitt. Zusammen mit der besseren CUDA-Vorhersage ist die Vorhersage des *Code-Analysis*-Moduls näher an der Vorhersagequalität der *Benchmarking*-Versionen.

Die Vorhersagen der *Benchmarking*-Module sind bei sehr kleinen Problemgrößen (Population: 100 und Problemgröße 30) ungenau und sagen einen sehr geringen Speedup voraus. Aufgrund der sehr geringen benötigten Datenmenge in dieser Konfiguration passen die Daten in den Cache-Speicher der Multi-Core-CPU. Durch den Cache-Effekt erreicht die reale OpenMP Implementierung einen sehr hohen Speedup. Die implementierten Benchmarks bilden diesen Cache-Effekt nicht ab, was zu einer geringeren Speedup-Vorhersage für diese Konfiguration führt. Der Effekt relativiert sich für größere Problemgrößen, bei denen die Daten nicht mehr komplett im Cache bereitgestellt werden können. Das Verhalten bei den anderen Konfigurationsgrößen zwischen den beiden *Benchmarking*-Versionen ist ähnlich zu dem festgestellten Verhalten bei den *GA*-Versuchen. Bei den kleineren Problemgrößen ist die Version mit *EFS* fast durchgehend besser, während die Version ohne *EFS* bei größeren Problemen bessere Ergebnisse liefert. Dieses Verhalten wurde bei den Evaluationsergebnissen des *GAs* bereits beschrieben und erklärt.

Ähnlich verhält es sich mit dem vorhergesagten Mapping, bei dem die Version mit *EFS* bei kleineren Problemengrößen genauer ist und die Version

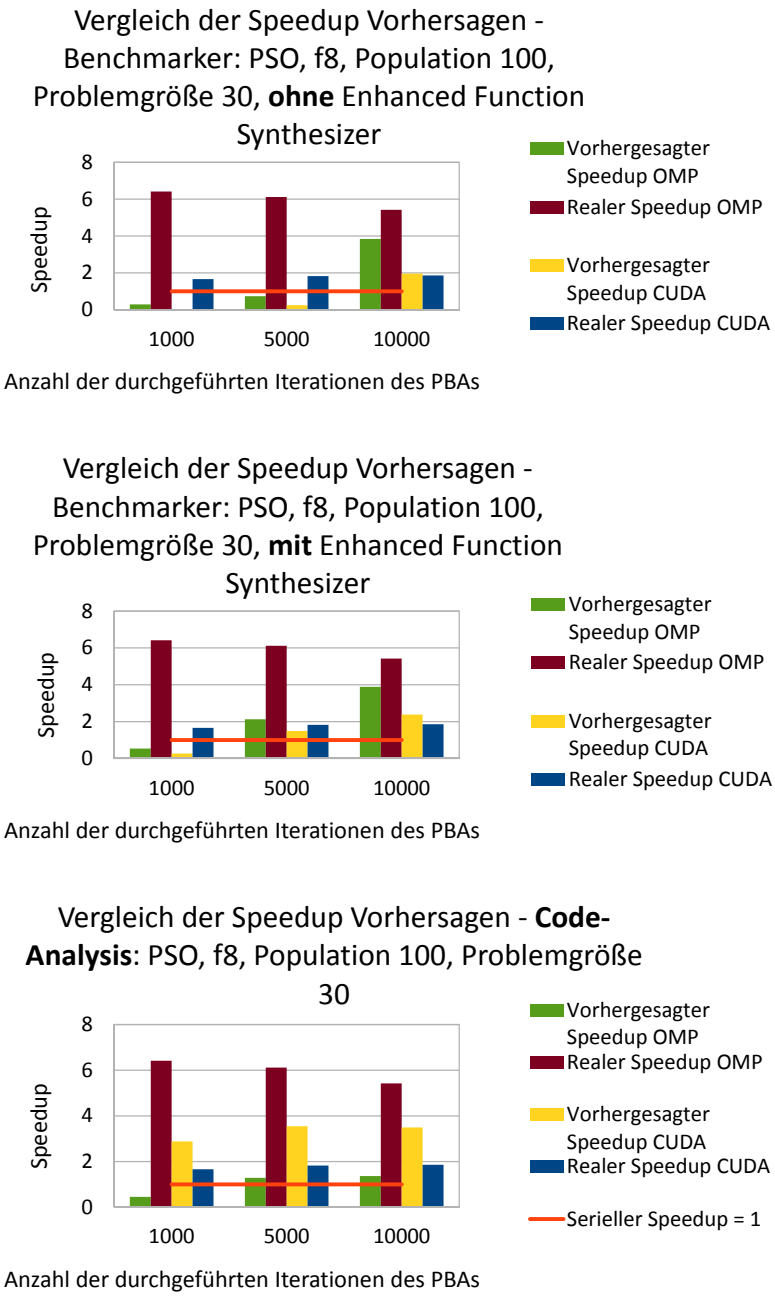


Abbildung 6.8: Vergleich: Funktion f_8 mit Populationsgröße 100 und Problemgröße 30

ohne *EFS* bei größeren Problemgrößen. Das *Code-Analysis*-Modul sagt aufgrund der Überschätzung der CUDA-Leistung, wie beim *GA*, beim Großteil der Versuche das falsche Mapping voraus. Anders als beim *GA* liefert beim *PSO* jedoch die CUDA-Version den größeren Speedup. In diesen Fällen ist die Vorhersage der Code-Analyse korrekt. Insgesamt sind auch beim PSO die Mapping-Voraussagen der *Benchmarking*-Module genauer als die Voraussagen des *Code-Analysis*-Moduls.

6.3.2.2 Untersuchung des vorhergesagten Mappings und des Vorhersagefehlers beim PSO

Die Ergebnisse für die Mapping-Vorhersagen für alle Funktionen sind in den Tabellen 6.11, 6.13, 6.15 und 6.17 abgebildet. Die zugehörigen prozentualen Abweichungen sind in den Tabellen 6.12, 6.14, 6.16 und 6.18 zusammengefasst. Auffallend ist die vergleichsweise gute Vorhersage (sowohl Mapping-Genauigkeit, als auch prozentuale Abweichung) des *Code-Analsis*-Moduls im Vergleich zu den *GA*-Experimenten. Insbesondere bei der prozentualen Abweichung ist die Code-Analyse in vielen Konfigurationen genauer, als die der *Benchmarker*. Bei der Funktion f11 ist sowohl die OpenMP-Gesamtvorhersage, als auch die CUDA-Gesamtvorhersage genauer (im Vergleich zu den Vorhersagen der jeweiligen *Benchmarker*).

Beim Vergleich der beiden *Benchmarker*-Versionen ist der Trend der prozentualen Abweichung ähnlich der *GA*-Betrachtung. Bei kleineren Problemen ist die Version mit *EFS* in der Regel genauer, während für größere Progblemgrößen in den meisten Fällen die Version ohne *EFS* bessere Ergebnisse liefert. Das Verhalten wurde im vorangegangenen Abschnitt beschrieben. Die Mapping-Vorhersage ist bei der Version ohne *EFS* mit 31 von 36 korrekten Vorhersagen genauer als die Version mit *EFS* (26 von 31). Die falschen Vorhersagen der *EFS*-Version hängen jedoch nicht immer mit einer schlechteren Vorhersage im Vergleich zur Version ohne *EFS* zusammen. Beispielsweise ist bei der Funktion f1 mit einer Populationsgröße von 200 und Problemgröße von 100 für 5000 Iterationen (s. Anhang B.1) der gegenteilige Fall zu erkennen. Die prozentuale Abweichung der *EFS*-Version beträgt für diese Konfiguration

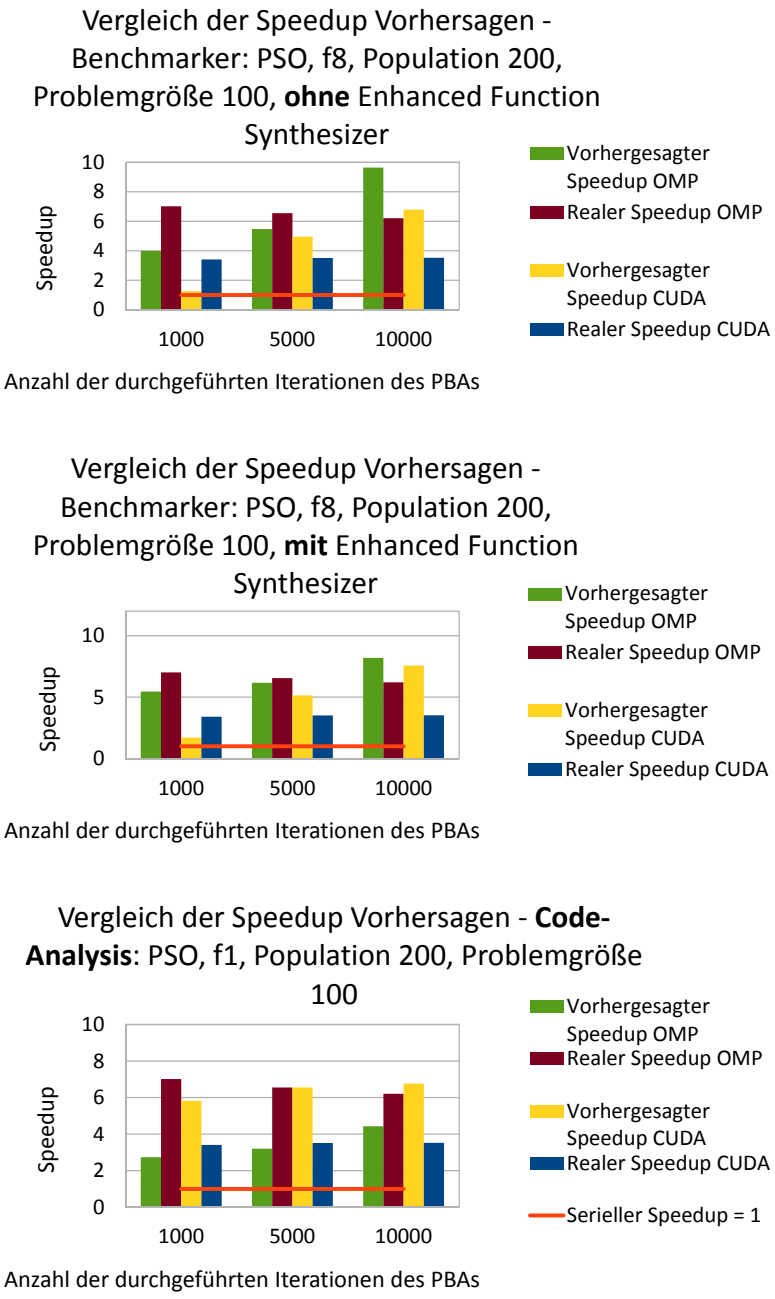
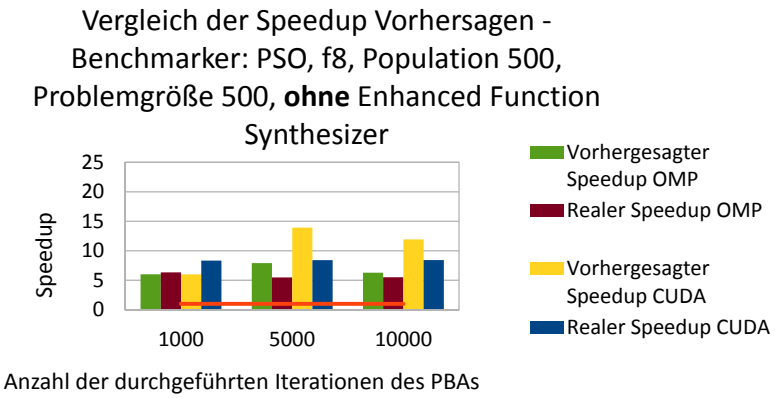
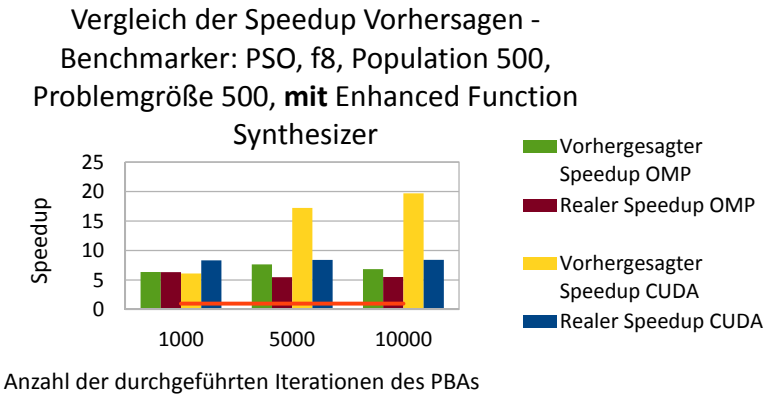


Abbildung 6.9: Vergleich: Funktion f_8 mit Populationsgröße 200 und Problemgröße 100





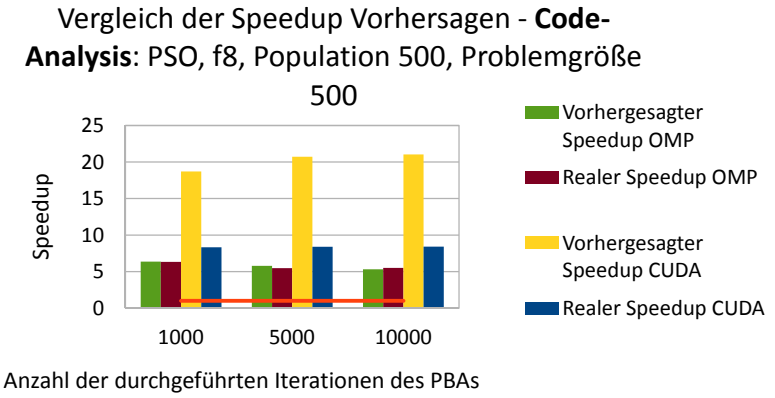


Abbildung 6.10: Vergleich: Funktion f_8 mit Populationsgröße 500 und Problemgröße 500

Funktion f1	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis	Reales Mapping
Pop: 100 Prob: 30	1000	seriell	seriell	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP
Pop: 200 Prob: 100	1000	OMP	OMP	CUDA	OMP
	5000	OMP	CUDA	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP
Pop: 500 Prob: 500	1000	CUDA	CUDA	CUDA	CUDA
	5000	CUDA	CUDA	CUDA	CUDA
	10000	CUDA	CUDA	CUDA	CUDA

Tabelle 6.11: Mapping-Voraussagen des PSOs für die Funktion f1

$34,20\%$, während sie für die Version ohne *EFS* $80,88\%$ beträgt. Dennoch ist die Mapping-Vorhersage der Version ohne *EFS* (OpenMP) korrekt, während die mit *EFS* CUDA vorhersagt und somit falsch ist (s. Tabelle 6.11). Der Effekt tritt auf, da der CUDA-Speedup von der *EFS*-Version genauer vorhergesagt wird (ohne *EFS*=$1,12\%$, mit *EFS*=$5,27\%$ und real=$5,19\%$), der OpenMP-Speedup von beiden Versionen viel zu gering vorhergesagt wird (ohne *EFS*=$2,34\%$, mit *EFS*=$4,64\%$ und real=$13,98\%$). Dadurch ergibt es sich, dass obwohl die Version ohne *EFS* genauer ist, die Mapping-Vorhersage dennoch falsch ist. Dies zeigt, dass beide beschriebenen Metriken (sekundäre und primäre; s. Abschn. 6.2) gemeinsam betrachtet werden müssen, um die beste Modulkonfiguration zu bestimmen.

Der allgemeine durchschnittliche Fehler der beiden *Benchmarking*-Module ist beim *PSO* nahezu identisch (ohne *EFS*=$55,99\%$, mit *EFS*=$52,11\%$). Die Vorhersagegenauigkeit des *Code-Analysis*-Moduls ist weitaus genauer im Vergleich zu den Vorhersagen bei den Experimenten mit dem *GA*. Während die durchschnittliche prozentuale Abweichung beim *GA* bei $139,11\%$ liegt, ist sie beim *PSO* aufgrund der besseren CUDA-Vorhersage bei $73,97\%$. Insgesamt ist die Vorhersagequalität der drei Module beim *PSO* ähnlicher als beim *GA*. Das *Code-Analysis*-Modul ist jedoch auch bei diesen Versuchen aufgrund der überschätzten CUDA-Leistung am ungenauesten.

Funktion f1	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis
Pop: 100 Prob: 30	1000	80,82%	81,76%	111,56%
		85,54% | 76,09%	89,19% | 74,33%	75,54% | 147,58%
	5000	77,60%	73,47%	41,13%
		84,05% | 71,15%	79,79% | 67,14%	72,37% | 9,89%
	10000	69,77%	58,46%	34,39%
		77,38% | 62,15%	61,83% | 55,09%	65,60% | 3,18%
Pop: 200 Prob: 100	1000	76,44%	29,53%	178,89%
		68,11% | 84,77%	46,39% | 12,67%	61,87% | 295,90%
	5000	80,88%	34,20%	99,61%
		83,28% | 78,48%	66,82% | 1,58%	69,31% | 129,91%
	10000	16,28%	24,05%	85,38%
		23,75% | 8,81%	41,50% | 6,60%	60,55% | 110,22%
Pop: 500 Prob: 500	1000	11,37%	6,51%	119,43%
		12,45% | 10,29%	1,35% | 11,68%	13,45% | 225,42%
	5000	36,16%	26,99%	60,67%
		31,52% | 40,80%	40,66% | 13,32%	46,03% | 75,31%
	10000	31,82%	19,95%	49,93%
		24,92% | 38,72%	26,08% | 13,83%	42,21% | 57,66%

Tabelle 6.12: Durchschnittliche prozentuale Abweichung der Vorhersagen für den PSO - Funktion f1

Funktion f5	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis	Reales Mapping
Pop: 100 Prob: 30	1000	OMP	seriell	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP
Pop: 200 Prob: 100	1000	OMP	OMP	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	CUDA	CUDA	OMP
Pop: 500 Prob: 500	1000	OMP	OMP	CUDA	CUDA
	5000	CUDA	CUDA	CUDA	CUDA
	10000	CUDA	CUDA	CUDA	CUDA

Tabelle 6.13: Mapping-Voraussagen des PSOs für die Funktion f5

Funktion f5	Iterationen	Benchmarker ohne EFS		Benchmarker mit EFS		Code-Analysis	
Pop: 100 Prob: 30	1000	83,14%		82,08%		78,16%	
		83,45%	82,83%	84,23%	79,93%	71,10%	85,22%
	5000	40,11%		40,07%		74,71%	
		32,45%	47,77%	60,49%	19,64%	63,46%	85,95%
	10000	26,31%		21,48%		57,28%	
		46,01%	6,62%	10,05%	32,92%	59,07%	55,48%
Pop: 200 Prob: 100	1000	63,27%		54,28%		76,51%	
		63,42%	63,12%	55,37%	53,20%	39,20%	113,82%
	5000	17,26%		44,26%		70,64%	
		8,86%	25,67%	11,51%	77,01%	23,05%	118,22%
	10000	36,59%		128,12%		70,43%	
		44,79%	28,39%	21,43%	128,12%	11,21%	129,65%
Pop: 500 Prob: 500	1000	14,17%		59,37%		96,01%	
		3,45%	24,89%	41,59%	77,15%	5,13%	186,89%
	5000	73,61%		17,00%		119,06%	
		53,21%	94,02%	3,23%	30,76%	15,70%	222,41%
	10000	103,10%		148,20%		120,70%	
		73,68%	132,53%	49,84%	246,55%	15,44%	225,95%

Tabelle 6.14: Durchschnittliche prozentuale Abweichung der Vorhersagen für den PSO - Funktion f5

Funktion f8	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis	Reales Mapping
Pop: 100 Prob: 30	1000	seriell	seriell	CUDA	OMP
	5000	seriell	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP
Pop: 200 Prob: 100	1000	OMP	OMP	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	OMP	CUDA	OMP
Pop: 500 Prob: 500	1000	CUDA	OMP	CUDA	CUDA
	5000	CUDA	CUDA	CUDA	CUDA
	10000	CUDA	CUDA	CUDA	CUDA

Tabelle 6.15: Mapping-Voraussagen des PSOs für die Funktion f8

Funktion f8	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis
Pop: 100 Prob: 30	1000	96,54%	87,73%	83,21%
		95,49% | 97,60%	91,58% | 83,88%	92,92% | 73,51%
	5000	87,32%	41,72%	86,63%
		87,99% | 86,64%	65,20% | 18,23%	78,92% | 94,34%
	10000	17,19%	28,22%	81,31%
		29,15% | 5,23%	28,27% | 28,17%	74,80% | 87,82%
Pop: 200 Prob: 100	1000	53,09%	36,02%	65,94%
		43,15% | 63,03%	22,22% | 49,82%	61,03% | 70,85%
	5000	28,77%	26,09%	68,83%
		16,45% | 41,08%	5,79% | 46,39%	51,11% | 86,55%
	10000	73,97%	73,47%	60,27%
		55,37% | 92,57%	32,05% | 114,89%	28,59% | 91,94%
Pop: 500 Prob: 500	1000	16,55%	13,45%	62,55%
		5,11% | 28,00%	0,26% | 26,64%	0,57% | 124,53%
	5000	55,04%	72,34%	76,14%
		44,39% | 65,69%	39,72% | 104,95%	5,71% | 146,58%
	10000	27,82%	79,11%	76,67%
		14,00% | 41,64%	24,21% | 134,01%	3,46% | 149,88%

Tabelle 6.16: Durchschnittliche prozentuale Abweichung der Vorhersagen für den PSO - Funktion f8

Funktion f11	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis	Reales Mapping
Pop: 100 Prob: 30	1000	seriell	seriell	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	seriell	CUDA	OMP
Pop: 200 Prob: 100	1000	OMP	OMP	CUDA	OMP
	5000	OMP	OMP	CUDA	OMP
	10000	OMP	CUDA	CUDA	OMP
Pop: 500 Prob: 500	1000	CUDA	CUDA	CUDA	CUDA
	5000	CUDA	CUDA	CUDA	CUDA
	10000	CUDA	CUDA	CUDA	CUDA

Tabelle 6.17: Mapping-Voraussagen des PSOs für die Funktion f11

Funktion f11	Iterationen	Benchmarker ohne EFS	Benchmarker mit EFS	Code-Analysis
Pop: 100 Prob: 30	1000	90,10% 93,02% | 87,18%	85,45% 88,69% | 82,21%	74,74% 81,78% | 67,70%
	5000	80,09% 86,23% | 73,95%	67,89% 72,31% | 63,48%	45,77% 86,29% | 5,25%
	10000	58,12% 62,10% | 54,14%	90,65% 91,39% | 89,91%	39,23% 72,47% | 5,99%
Pop: 200 Prob: 100	1000	60,93% 50,95% | 70,90%	46,43% 28,93% | 63,93%	50,18% 38,69% | 61,67%
	5000	58,53% 61,66% | 55,40%	45,35% 48,72% | 41,98%	32,98% 58,79% | 7,16%
	10000	42,79% 38,57% | 47,02%	27,55% 34,92% | 20,18%	29,35% 55,11% | 3,59%
Pop: 500 Prob: 500	1000	33,42% 26,75% | 40,09%	15,88% 7,18% | 24,58%	54,31% 0,62% | 108,01%
	5000	34,81% 29,98% | 39,65%	52,21% 28,21% | 76,22%	63,95% 0,30% | 127,60%
	10000	53,79% 46,79% | 60,79%	90,05% 34,78% | 145,33%	66,43% 2,11% | 130,75%

Tabelle 6.18: Durchschnittliche prozentuale Abweichung der Vorhersagen für den PSO - Funktion f11

6.3.3 Zusammenfassung der Ergebnisse der Speedup-Prädiktoren

Die drei beschriebenen Prädiktoren wurden anhand von zwei Szenarien (*GA* und *PSO*) mit unterschiedlichen Parametern evaluiert. Das Verhalten der Vorhersage-Module war in beiden Szenarien ähnlich. Es lässt sich ein signifikanter Vorteil der *Benchmarking*-Module im Vergleich zum *Code-Analysis*-Modul in der Gesamtvorhersage (OpenMP und CUDA) feststellen. Die durchschnittliche Abweichung (sekundäre Metrik; s. Abschn. 6.2) über beide Evaluationsszenarien bei der Speedup-Vorhersage des *Benchmarkers* ohne *EFS* beträgt $50,26\%$, beim *Benchmarker* mit *EFS* $51,09\%$ und beim *Code-Analysis*-Modul $106,54\%$. Dies ist durchgehend auf die überschätzte CUDA-Leistung durch das *Code-Analysis*-Modul zurückzuführen. Bei der Vorhersage des OpenMP-Speedups lies sich kein signifikanter Unterschied bei den drei untersuchten Modulen feststellen. Die durchschnittliche Abweichung bei der separaten Betrachtung der OpenMP-Vorhersage beträgt beim *Benchmarker* ohne

EFS 47, 92%, beim Benchmarker mit *EFS* 43, 49% und bei der *Code-Analysis* 39, 06%. Somit ist die Vorhersagegenauigkeit bei der Code-Analyse genauer, wenn die OpenMP-Vorhersage einzeln betrachtet wird.

Die beiden *Benchmarking*-Versionen besitzen eine nahezu identische durchschnittliche Abweichung. Dennoch sind ihre Mapping-Vorhersagen unterschiedlich. Während das Modul ohne *EFS* bei 61 von 72 untersuchten Konfigurationen das korrekte Mapping vorhersagt, sind es beim Modul ohne *EFS* 57 von 72. Auf den dazu führenden Effekt wurde in Abschnitt 6.3.2 eingegangen. Die falschen Vorhersagen treten dabei bei beiden Modulen hauptsächlich bei kleinen Problem- und Populationsgrößen auf. Vergleicht man jedoch die durchschnittliche Abweichung der beiden Module, ist das Modul mit EFS bei 40 Konfigurationen genauer, während das Modul ohne *EFS* bei 32 Konfigurationen genauer ist. Somit ist das Modul mit *EFS* in der Regel näher an den tatsächlichen Speedup-Werten, bei einigen Konfigurationen ist die Abweichung jedoch signifikant, wodurch sich eine ähnliche Gesamtleistung, wie beim Modul ohne *EFS* ergibt.

Die drei beschriebenen Speedup-Prädiktoren eignen sich unterschiedlich gut für die von *CASEP* benötigten Aufgaben. Während das *Code-Analysis*-Modul eine gute Vorhersage für die OpenMP-Vorhersage berechnet, ist die CUDA-Vorhersage generell zu optimistisch und somit nur bei einigen hier beschriebenen Konfigurationen anwendbar. Die *Benchmarker*-Module sind sowohl bei der Mapping-Vorhersage als auch bei der prozentualen Abweichung in den untersuchten Szenarien anwendbar. Die Vorhersagen sind zum größten Teil akkurat und die Abweichung gering. Die *Benchmarking*-Module unterscheiden sich hauptsächlich in ihrer Leistung bei unterschiedlichen Problemgrößen, bei denen das Modul ohne *EFS* bei größeren Problemgrößen bessere Ergebnisse liefert, während das Modul mit *EFS* in der Regel bessere Ergebnisse bei kleinen Problemgrößen berechnet. Je nach Gewichtung der beiden beschriebenen Metriken sind die Module unterschiedlich einzuordnen. Bei einer höheren Gewichtung der ersten Metrik ist die Leistung des Moduls ohne *EFS* besser. Bei einer höheren Gewichtung der zweiten Metrik hingegeben liefert das Modul mit *EFS* die besseren Ergebnisse.

6.4 Evaluierung der aggregierten Speedup - Vorhersagen

Die Ergebnisse der drei Speedup-Vorhersagemethoden wurden dazu verwendet, im *Mapping*-Modul (s. Absch. 4.3) eine aggregierte Vorhersage zu erstellen. Hierzu wurde das *Benchmarking*-Modul mit *EFS* und das *Code-Analysis* Modul verwendet. Das *Benchmarking*-Modul ohne *EFS* wurde aufgrund der schlechteren Leistung bei der Anzahl an prozentualen Abweichungen gegenüber dem Modul mit *EFS* nicht verwendet. Das *Code-Analysis*-Modul wurde aufgrund den guten Ergebnissen bei der OpenMP-Vorhersage und den teilweise besseren Ergebnissen bei den CUDA-Vorhersagen beim *PSO* ausgewählt.

Die Gewichtung der beiden Vorhersage-Module wurde mit Hilfe von multilinearer Regression durchgeführt (s. Abschn. 4.3). Für die Regression sind Eingabewerte notwendig, mit deren Hilfe eine Annäherung der richtigen Gewichtungen durchgeführt werden kann. Die Regressionsebene soll möglichst das gesamte Spektrum der hier untersuchten Funktionen abbilden. Hierzu wurden die Ergebnisse der Funktion f1 und Funktion f11 ausgewählt. Die Funktion f1 ist von der Komplexität her die einfachste der hier betrachteten Funktionen, während die Funktion f11 die komplexeste Funktion darstellt. Für die beiden verwendeten Module soll eine Annäherung gefunden werden, bei der gilt:

$$\vec{Y} = \alpha \cdot \vec{x_1} + \beta \cdot \vec{x_2}$$

Dabei stellt $\vec{Y}$ einen Vektor mit allen realen Speedup-Werten für OpenMP und CUDA dar, $\vec{x_1}$ ist ein Vektor mit allen vorhergesagten Speedup-Werten des *Benchmarkers* mit *EFS* und $\vec{x_2}$ ist ein Vektor mit den Vorhersagen des *Code-Analysis*-Moduls. Die Werte α und β sind Gewichte, durch die der durchschnittliche Fehler der Regression möglichst gering ausfallen soll. Die sich daraus ergebene Regressionsebene ist in Abb. 6.11 abgebildet. Für kleinere Speedup-Werte erfolgt eine relativ genaue Annäherung mit einem geringen Fehler. Dies bedeutet, dass durch die gewichtete Addition der Werte aus $\vec{x_1}$ und $\vec{x_2}$ der sich ergebende Wert sehr nah an dem zugehörigen Wert aus dem $\vec{Y}$ Vektor ist. Bei den größeren Speedup-Werten gibt es einige Punkte im Raum,

die durch die Ebene nicht sehr genau angenähert werden können. Eine Verschiebung der Ebene zu diesen Werten würde den Gesamtfehler der Regression bei den kleineren Werten erhöhen. Der Gesamtfehler für die hier abgebildete Regressionsebene ist 1,1697. Somit ist der durchschnittliche Fehler bei der gewichteten Addition der beiden Vektoren zum realen Speedup 1,1697, was bei den in den letzten Abschnitten gezeigten Speedup-Werten gering ist. Die aus der Regression ermittelten Gewichte für den *Benchmarker* (α) und das *Code-Analysis*-Modul (β) betragen $\alpha = 0{,}73924$ und $\beta = 0{,}19970$.

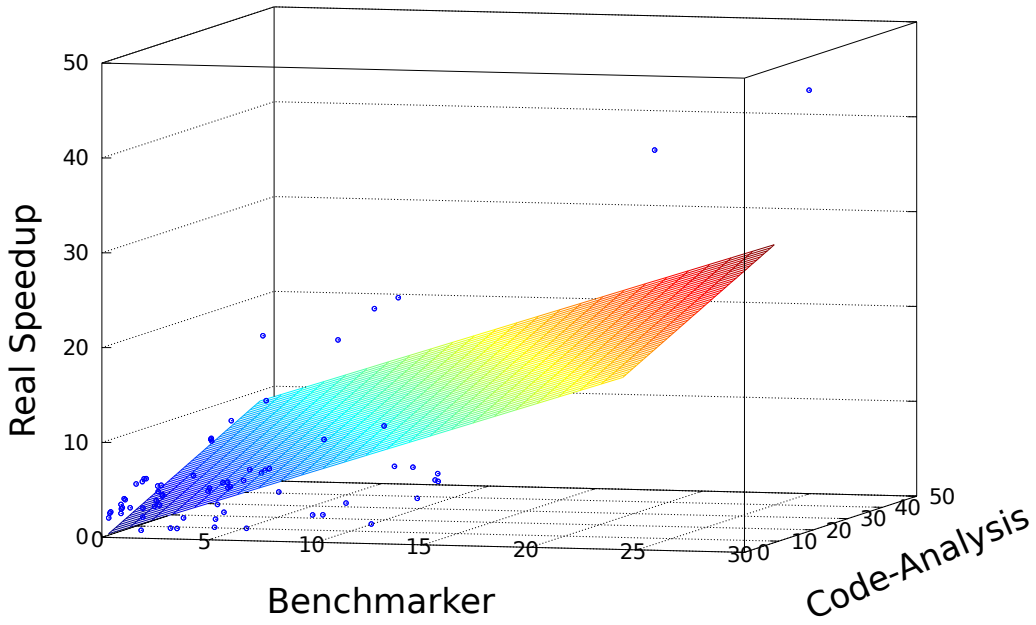


Abbildung 6.11: Regressionsebene für den Benchmarker mit EFS und das Code-Analsis-Modul

Die beiden gewichtete Modulen wurden dazu verwendet, das Mapping der Funktionen f5 und f8 vorherzusagen. Die Funktionen f1 und f11 wurden als Eingabeparameter für die Regression verwendet und werden somit hier nicht weiter evaluiert. Es wurde sowohl das Verhalten beim *PSO*, als auch beim *GA* untersucht. Es wurde der in Abschn. 6.1 beschriebene Versuchsaufbau verwendet.

In den Tabellen 6.19 und 6.21 sind die Mapping-Vorhersagen für die *GA*- und *PSO*-Experimente abgebildet. Die korrekten Vorhersagen sind mit grün

markiert, während die falschen Vorhersagen rot markiert sind. Die Module sagen bei 26 von 36 Vorhersagen das Mapping korrekt voraus. Dieser Wert ist besser als die *Code-Analysis* alleine, jedoch schlechter als die beiden *Benchmarking*-Module (ohne und mit *EFS*) das Mapping vorausgesagt haben. Auffallend ist, dass bei 8 von 10 falschen Vorhersagen CUDA vorhergesagt wurde, obwohl das korrekte Mapping OpenMP war. Dies ist auf das Überschätzen durch das *Code-Analysis*-Modul zurückzuführen. Dies ist trotz der geringen Gewichtung des *Code-Analsyis*-Moduls ein entscheidender Faktor. Insbesondere wenn die prozentuale Abweichung bei der CUDA-Vorhersage sehr groß ist.

Die prozentuale Abweichung der aggregierten Vorhersage ist in den Tabellen 6.20 und 6.22 abgebildet. Dabei sind die Zellen grün markiert, wenn die Abweichung geringer als bei allen drei im letzten Abschnitt beschriebenen Vorhersagemethoden ist. Obwohl die aggregierten Vorhersagen zum Teil besser als die Einzelvorhersagen sind, liegt der durchschnittliche Gesamtfehler mit $48,53\%$ über dem des *Benchmarking*-Moduls ohne *EFS* ($40,73\%$) und dem des *Benchmarking*-Moduls mit *EFS* ($41,39\%$) für die hier betrachteten Funktionen. Das *Code-Analysis*-Modul ist mit einem durchschnittlichen Fehler von $89,05\%$ das ungenauste der vier betrachteten Verfahren. Der höhere Gesamtfehler des aggregierten Verfahrens ergibt sich aus dem relativ hohen Vorhersagefehler des CUDA-Speedups (durchschnittlich $62,59\%$), der weitaus höher als bei dem Benchmarker ohne *EFS* ($41,73\%$) und mit *EFS* ($50,17\%$) liegt. Beim OpenMP-Speedup hingegeben liegt die Leistung mit einer Abweichung von $35,41\%$ zwischen den beiden *Benchmarking*-Modulen ($39,74\%$ und $32,60\%$).

Eine einfache 50% Gewichtung der Ergebnisse aus dem letzten Abschnitt des *Benchmarking*-Moduls ohne *EFS* und des *Code-Analysis*-Moduls führt zu weitaus schlechteren Ergebnissen. Zwar ist die OpenMP-Vorhersage mit $32,01\%$ leicht besser als die hier beschriebene Regressions-Methode, die CUDA-Vorhersage ist jedoch mit $98,42\%$ deutlich ungenauer, was zu einem Gesamtfehler von $65,21\%$ führt. Die aus der Regression abgeleiteten Werte führen dementsprechend zu einer leichten Verschlechterung der OpenMP-Vorhersage und verbessern die CUDA-Vorhersage signifikant.

PBA: GA	Iterationen	Funktion f5	Funktion f8
Pop: 100 Prob: 30	1000	serial	CUDA
	5000	OMP	OMP
	10000	OMP	CUDA
Pop: 200 Prob: 100	1000	OMP	CUDA
	5000	OMP	OMP
	10000	OMP	OMP
Pop: 500 Prob: 500	1000	OMP	OMP
	5000	CUDA	OMP
	10000	CUDA	OMP

Tabelle 6.19: Mapping-Voraussagen der aggregierten Vorhersage - GA

PBA: GA	Iterationen	Funktion f5		Funktion f8	
Pop: 100 Prob: 30	1000	32,08%		52,99%	
		60,89%	3,28%	63,61%	42,38%
	5000	62,32%		65,43%	
		61,4%	63,24%	43,72%	87,14%
	10000	69,15%		82,06%	
		42,95%	95,35%	49,66%	114,45%
Pop: 200 Prob: 100	1000	25,1%		35,92%	
		38,93%	11,26%	70,15%	1,69%
	5000	25,21%		23,69%	
		9,89%	40,53%	11,91%	35,47%
	10000	24,61%		39,41%	
		7,18%	42,05%	38,2%	40,63%
Pop: 500 Prob: 500	1000	43,77%		19,55%	
		30,83%	56,71%	11,2%	27,9%
	5000	16,94%		11,68%	
		0,23%	33,66%	5,09%	18,27%
	10000	34,19%		29,48%	
		34,65%	90,02%	9,87%	26,77%

Tabelle 6.20: Durchschnittliche prozentuale Abweichung der aggregierten Vorhersagen - GA

Verbesserung der gewichteten Vorhersage: Die für *CASEP* beschriebene Regressionsmethode hat Schwächen bei der Vorhersage des CUDA-Speedups. Dieser ist auf die zu hohe Vorhersage des Speedups durch das *Code-Analysis*-Modul und für große Problemgrößen auf das *EFS*-Modul des *Benchmarkers* zurückzuführen (s. Abschn. 6.3.1.1). Auf die möglichen Modifikationen des *EFS*-Moduls zur Verbesserung der Vorhersage wurde im Abschnitt 6.3.1.1 eingegangen. Diese werden an dieser Stelle nicht weiter behandelt. Die drei möglichen Verbesserungen sind: 1) Erhöhung der Anzahl der Funktionen im *Functions-Pool*, 2) Verschiebung der Komplexität der generierten Funktionen in beide Richtungen erlauben (zu komplexeren und weniger komplexen Funktionen) und 3) Verzicht auf den *Functions-Pool*.

Eine Verbesserung der Vorhersage ohne das *Code-Analysis*-Modul zu verändern besteht darin, die beiden Module anders zu gewichten. In der in dieser Arbeit vorgestellten Regression werden die OpenMP- und CUDA-Vorhersagen des *Benchmarkers* durch α gewichtet und die beiden Vorhersagen der Code-Analyse durch β. Die einzelnen Vorhersagen können jedoch auch separat gewichtet werden. Hierzu werden durch die Regression zwei Werte für den *Benchmaker* ermittelt (α_1 für OpenMP und α_2 für CUDA) und zwei Werte für die *Code-Analysis* (β_1 und β_2). Durch die Einzelgewichtung kann das Überschätzen des *Code-Analysis*-Moduls ausgeglichen werden und die Gesamtvorhersage wird genauer. Eine weitere Möglichkeit, den Fehler des *Code-Analysis*-Moduls zu minimieren, besteht darin, die CUDA-Vorhersage zu ignorieren und nur die Vorhersage des *Benchmarkers* für CUDA zu verwenden. Dies hat jedoch den Nachteil, dass bei einigen Konfigurationen die Code-Analyse genauer ist und in diesen Fällen keine Verbesserung durch die Regression erfolgen würde. Die erste Methode ist somit für eine verbesserte Vorhersage zu präferieren.

PBA: PSO	Iterationen	Funktion f5	Funktion f8
Pop: 100 Prob: 30	1000	OMP	serial
	5000	OMP	OMP
	10000	CUDA	OMP
Pop: 200 Prob: 100	1000	OMP	OMP
	5000	OMP	CUDA
	10000	CUDA	OMP
Pop: 500 Prob: 500	1000	CUDA	CUDA
	5000	CUDA	CUDA
	10000	CUDA	CUDA

Tabelle 6.21: Mapping-Voraussagen der aggregierten Vorhersage - PSO

PBA: PSO	Iterationen	Funktion f5		Funktion f8	
Pop: 100 Prob: 30	1000	62,77%		67,71%	
		82,46%	43,07%	86,79%	48,63%
	5000	32,8%		21,2%	
		60,85%	4,75%	37,21%	5,19%
	10000	35,05%		42,91%	
		50,44%	19,66%	27%	58,82%
Pop: 200 Prob: 100	1000	21,91%		31,27%	
		42,7%	1,12%	28,40%	34,13%
	5000	58,58%		42,04%	
		21,21%	95,94%	6,89%	77,18%
	10000	86,69%		85,22%	
		30,67%	142,71%	37,2%	133,24%
Pop: 500 Prob: 500	1000	14,11%		28,94%	
		2,35%	25,87%	27,78%	30,1%
	5000	100,28%		86,18%	
		41,77%	158,8%	38,5%	133,86%
	10000	132,49%		103,17%	
		33,35%	231,63%	28,68%	177,66%

Tabelle 6.22: Durchschnittliche prozentuale Abweichung der aggregierten Vorhersagen - PSO

6.5 Evaluation des PBA2CUDA Moduls

In diesem Abschnitt werden die durch *CASEP* automatisiert generierten parallelen Versionen des *GAs* und *PSO*-Algorithmus evaluiert. Als Metrik wird der in Abschn. 6.2 beschriebene Speedup im Vergleich zum Speedup einer manuell parallelisierten Version des Algorithmus verwendet. Weiterhin wird verglichen, ob das durch *CASEP* (aggregierte) vorhergesagte Mapping mit dem durch die automatisierten Versionen der Algorithmen sich ergebenden Mapping übereinstimmt. Für den Vergleich mit der aggregierten Vorhersage aus dem vorherigen Abschnitt wurden die Funktionen f5 und f8 untersucht. Beim Verhalten der Speedup-Vorhersage zwischen den einzelnen Iterationskonfigurationen (1000, 5000 und 10000) wurde in den Evaluationen aus den letzten Abschnitten kein nennenswerter Unterschied festgestellt. Aus diesem Grund beschränkt sich diese Evaluation auf die Iterationsgröße 5000.

In Tabelle 6.23 wird das durch *CASEP* vorausgesagte Mapping mit dem sich aus den automatisiert parallelisierten Versionen ergebendem Mapping verglichen. Dabei wird verglichen, ob die schnellste Version (OpenMP oder CUDA) von den beiden durch *PBA2CUDA* erzeugten Versionen auch die tatsächlich von *CASEP* vorausgesagte Version ist. Bei 10 von 12 Konfigurationen stimmt die Vorhersage mit den durch *PBA2CUDA* erzeugten Ergebnissen überein. Bei den beiden Konflikten sagt *CASEP* das falsche Mapping voraus, wohingegen die Ergebnissen von *PBA2CUDA* korrekt sind (bezogen auf die Ergebnissen des realen Speedups; s. Abschn. 6.3.1.2 und 6.3.2.2). Somit ist bei allen untersuchten Versuchen das erwartete Ergebnis für *PBA2CUDA* erreicht worden.

		GA		PSO	
Funktion	Pop./Prob.	PBA2CUDA	CASEP	PBA2CUDA	CASEP
f5	100/30	OMP	OMP	OMP	OMP
	200/100	OMP	OMP	OMP	OMP
	500/500	OMP	CUDA	CUDA	CUDA
f8	100/30	OMP	OMP	OMP	OMP
	200/100	OMP	OMP	OMP	CUDA
	500/500	OMP	OMP	CUDA	CUDA

Tabelle 6.23: Mapping-Genauigkeit CASEP/PBA2CUDA

PBA: GA		Speedup PBA2CUDA		Speedup real	
Funktion	Pop./Prob.	OpenMP	CUDA	OpenMP	CUDA
f5	100/30	2,01	0,54	3,04	0,62
	200/100	3,84	1,68	3,88	1,83
	500/500	2,76	2,68	3,86	3,06
f8	100/30	3,06	0,67	3,50	0,69
	200/100	4,17	1,80	3,84	2,00
	500/500	4,57	3,77	4,40	3,81

Tabelle 6.24: Vergleich des realen Speedups mit dem Speedup durch PBA2CUDA - GA

Die Tabellen 6.24 (GA) und 6.25 (PSO) vergleichen den erzielten Speedup der durch *PBA2CUDA* parallelisierten Versionen mit den manuell automatisierten Versionen der Algorithmen. Die manuell erzeugten Versionen der Algorithmen besitzen dabei alle von *PBA2CUDA* in dieser Arbeit beschriebenen Code-Optimierungen, wie einen *Reduction Kernel* und die für CUDA angepassten Zufallszahlenzugriffe. In der Regel erzielen die *PBA2CUDA*-Versionen einen leicht schlechteren Speedup. Bei 17 von 24 Versuchen ist die manuell erzeugte Version schneller. In 7 Fällen erzielen die durch *PBA2CUDA* erzeugten Versionen einen höheren Speedup. Dies ist auf die Art der Parallelisierung durch den OpenMPC-Compiler zurückzuführen, der von *PBA2CUDA* verwendet wird. Der so generierte Quellcode ist im Aufbau komplexer als manuell parallelisierter Quellcode. Der Compiler setzt automatisiert bestimmte Schwellwerte für z.B. Shared-Memory auf der GPGPU oder maximal nutzbareren globalen Speicher fest. Die automatisiert festgelegten Werte führen in den meisten Fällen zu einer leichten Verschlechterung der Laufzeit. In einigen wenigen Fällen können sie jedoch auch im Rahmen der Messgenauigkeit zu einer leichten Verbesserung gegenüber einer manuellen Version führen, bei der diese Werte durch den CUDA-Compiler verwaltet werden. Insgesamt betrachtet liegen die Speedup-Werte der automatisiert parallelisierten Versionen durch *CASEP* im Rahmen der manuell erzeugten Versionen der untersuchten Algorithmen.

PBA: PSO		Speedup PBA2CUDA		Speedup real	
Funktion	Pop./Prob.	OpenMP	CUDA	OpenMP	CUDA
f5	100/30	6,00	2,27	5,16	1,93
	200/100	6,37	3,90	6,43	3,24
	500/500	6,62	9,53	5,78	7,50
f8	100/30	4,89	1,46	6,11	1,83
	200/100	5,07	2,28	6,88	3,51
	500/500	5,05	5,62	5,48	8,41

Tabelle 6.25: Vergleich des realen Speedups mit dem Speedup durch PBA2CUDA - PSO

6.6 Zusammenfassung der Evaluationsergebnisse

Im vorliegenden Kapitel wurden die einzelnen Module von *CASEP* evaluiert. Die Evaluation behandelte dabei die beiden Hauptaspekte des Frameworks: a) das Ermitteln des besten Software-Hardware-Mappings basierend auf dem vorhergesagten Speedup und b) die automatisierte Parallelisierung von *PBA*-Algorithmen. Als Evaluationsszenario wurde der Bereich der Funktionsoptimierung gewählt, durch den sich unterschiedliche Problemstellungen und Problemgrößen formal darstellen lassen. Es wurden Funktionen unterschiedlicher Rechenkomplexität in verschiedenen Parameterkonfigurationen verwendet, um einen möglichst großen Bereich der Funktionsoptimierung abzubilden. Dadurch sollte untersucht werden, ob sich *CASEP* bei verschiedenen Problemgrößen unterschiedlich verhält. Die verwendeten Algorithmen waren die populationsbasierten Verfahren *Genetischer Algorithmus* und *Particle Swarm Optimization*.

Die Ergebnisse der Evaluation zeigen eine ausreichend gute Vorhersage des Mappings durch das *CASEP*-Framwork. Die Evaluation hat ergeben, dass die CUDA-Vorhersage der Code-Analyse in den meisten Fällen zu optimistisch ist und dadurch einen zu hohen Speedup vorhersagt. Das *Benchmarking*-Modul berechnet in der Regel einen genaueren Speedup. Die geringe Abweichung entsteht dadurch, dass den Benchmarks nicht die vollständigen Informationen der realen Programme zur Verfügung stehen und somit nicht komplett das

Laufzeitverhalten abbilden können. Trotz der geringen Abweichung des von *CASEP* vorhergesagten Speedups, ist die Vorhersage hinreichend genau, um ein korrektes Software-Hardware-Mapping zu gewährleisten. Falsche Vorhersagen ergeben sich in der Regel in Bereichen, in denen der Speedup der einzelnen Implementierungen (CUDA, OpenMP und seriell) ähnlich sind. Im suboptimalen Fall wird dementsprechend ein Mapping gewählt, bei dem der Speedup geringfügig geringer als ein alternatives Mapping ist.

Im Vergleich der durch *CASEP* erzeugten parallelen Algorithmen zu den manuell erzeugten parallelen Algorithmen konnte ein leichter Speedup-Vorteil der manuell erzeugten Versionen gegenüber den automatisiert erzeugten Versionen festgestellt werden. Dies stellt ein vielversprechendes Ergebnis dar. Bei einer handoptimierten Version steht in der Regel mehr Wissen zur Verfügung und ist dementsprechend in den meisten Fällen leistungsfähiger. Dennoch sind die automatisiert erzeugten Versionen der *PBAs* in ihrer Leistung ähnlich zu den manuell erzeugten Versionen.

Zusammenfassend haben die Evaluationsergebnisse die Eignung von *CASEP* in den zwei untersuchten Aufgaben (Mapping-Vorhersage und Parallelisierung der Algorithmen) gezeigt.

7 Zusammenfassung und Ausblick

Inhaltsangabe

7.1 Zusammenfassung

In dieser Dissertation wurde das Framework *CASEP* zur Unterstützung des Entwicklers bei der Programmierung von paralleler Software vorgestellt. Dabei wird er in der Entwicklungsphase unterstützt, bei der noch kein paralleler Quellcode implementiert ist, in der jedoch der serielle Quellcode oder zumindest Abschnitte davon bereits vorhanden sind. Es wurden zwei Grundprobleme bei der Erstellung von paralleler Software behandelt:

1. Vorhersage des besten Software-Hardware-Mappings von serieller Software auf parallele Hardware.

2. Automatisierte Parallelisierung der Software auf der Zielhardware.

Für die Vorhersage des besten Mappings wurden zwei Verfahren untersucht: 1) ein Benchmark-basiertes Verfahren (*Benchmarking*-Modul) und 2) ein Verfahren auf Basis von Code-Analyse (*Code-Analysis*-Modul). Die beiden Verfahren bestimmen das Mapping auf Basis des zu erwartenden Speedups einer parallelen Version gegenüber einer seriellen Implementierung. Das beste Mapping ergibt sich aus der schnellsten Ausführung des Programms, wobei drei Implementierungsmethoden untersucht werden: 1) serielle Version des Programms,

2) parallele Version des Programms auf einer Multi-Core CPU und 3) parallele Version des Programms auf einer GPGPU. Es wurde ein Verfahren entwickelt, mit der sich die Benchmarks des *Benchmarking*-Moduls optimieren lassen, um das reale Eingangsprogramm besser abbilden zu können. Des Weiteren wurde ein Verfahren entwickelt, mit dem sich Cache-Effekte bei der Code-Analyse reduzieren lassen, die die Speedup-Vorhersage negativ beeinflussen können. Zuletzt wurde eine aggregierte Vorhersagemethode entwickelt, bei der die beiden Vorhersage-Module gewichtet wurden, um eine gemeinsame Vorhersage zu erhalten.

Im weiteren Verlauf der Arbeit wurde ein Verfahren zur automatisierten Parallelisierung der Eingabealgorithmen beschrieben. Hierfür wurde der offene OpenMPC-Compiler erweitert und modifiziert, um Fehler bei der Parallelisierung zu vermeiden und den Quellcode zu optimieren. Es wurden zwei Optimierungsschritte vorgestellt: 1) Verbesserung der Genauigkeit der Ausgaben des Eingabealgorithmus und 2) Optimierungen, die die Laufzeit des Algorithmus verringern sollen.

Der Fokus des vorgestellten Frameworks liegt auf *PBAs*. Dadurch konnten einzelne Module (z.B. der *Random Function Converter* des automatisierten Parallelisierers *PBA2CUDA*) verfeinert und auf eine bestimmten Algorithmusklasse hin optimiert werden. Im Rahmen dieser Arbeit wurde auch eine Generalisierung von *CASEP* untersucht. Folgende Aspekte wurden dabei betrachtet:

1. Anpassung an andere Algorithmenklassen.
2. Anpassung an andere Hardware.
3. Generalisierung von Parallelisierungsmethoden.
4. Verwendung von alternativen Programmiersprachen.

Die analytische Betrachtung der Generalisierung ergab, dass das Framework auf die beschriebenen Bereiche erweiterbar ist. Je nach Generalisierungsaspekt sind die notwendigen Anpassungen dabei unterschiedlich komplex. Die Anpassungen an andere Algorithmenklassen und Parallelisierungsmethoden sind

vergleichsweise einfach durchzuführen. Die Anpassungen zum Verwenden einer alternativen Programmiersprache sind hingegen aufwendig und erfordern eine fast vollständige Reimplementierung des Frameworks.

Die experimentellen Ergebnisse ergaben eine Eignung der in *CASEP* verwendeten Methodiken zur Bestimmung des optimalen Mappings für *PBAs*. Weiterhin ergaben Messungen, dass automatisiert erstellte Versionen der *PBAs* durch *CASEP* eine ähnliche Leistung erreichen, wie manuell optimierte parallele Versionen der Algorithmen. Die Eignung des vorgestellten Frameworks für die definierten Problemstellungen wurde in dieser Arbeit gezeigt. Dennoch existieren Forschungspunkte, die in dieser Arbeit nicht betrachtet wurden und mögliche zukünftige Forschungsziele darstellen.

7.2 Zukünftige Forschungsziele

Im Laufe der Arbeit haben sich verschiedene Forschungsmöglichkeiten für zukünftige Arbeiten ergeben. Diese werden im Folgenden vorgestellt:

- Optimierung des EFS-Moduls: Das für den *Benchmarker* vorgestellte *EFS*-Modul hat bei einigen der durchgeführten Versuche zu einer Verschlechterung der Vorhersage gegenüber einer Version ohne *EFS* geführt. Es wurden drei mögliche Modifikationen vorgestellt (s. Abschn. 6.3.1.1), die eine mögliche Verbesserung des Moduls zur Folge haben:

  1. Die Anzahl an Benchmarkfunktionen erhöhen, die für den *EFS* verwendet werden.
  2. Das *EFS*-Modul modifizieren, um eine Verschiebung der erzeugten Benchmarkfunktionen in Richtung einfacherer Funktionen zu erlauben.
  3. Auf den Funktionspool des *EFS* verzichten und einzig Eingaben des Benutzers zur Erzeugung der Funktionen für die Benchmarks zulassen.

- Optimierung der CUDA-Vorhersage bei der Code-Analyse: Die durchgeführten Evaluationen haben ein durchgehendes Überschätzen der CUDA-

Vorhersagen des *Code-Analysis*-Moduls gezeigt. Diese Überschätzung ist auf eine ungenaue Abbildung der Speicherzugriffe auf der GPGPU zurückzuführen. In der verwendeten Code-Analyse wird der optimale Fall beim Speicherzugriff angenommen, was jedoch nicht immer zutrifft. Beispielsweise ist ein Zugriff auf ein Array schneller, wenn jeder Thread auf das Element zugreift, das seiner Thread-ID entspricht (Thread 1 auf Arrayposition 1, Thread 2 auf Arrayposition 2, usw.). Es müssen Methoden entwickelt werden, mit denen sich unterschiedliche Speicherzugriffsarten darstellen lassen. Weiterhin muss evaluiert werden, unter welchen Voraussetzungen eine bestimmte Zugriffsart am wahrscheinlichsten ist, um bei der Code-Analyse die korrekte verwenden zu können.

- Unterschiedliche Gewichtung der Vorhersagen: Bei der aggregierten Vorhersage, bestehend aus der Vorhersage des *Benchmarkers* und der Code-Analyse, wurden die CUDA- und OpenMP-Vorhersagen mit Hilfe jeweils eines Parameters für jede Vorhersagemethode gewichtet. Diese Gewichtung erfolgte auf Basis einer multilinearen Regressionsanalyse. Es wurde anhand von Experimenten festgestellt, dass eine Gewichtung beider Architekturen (GPGPU, Multi-Core CPU) durch einen Parameter, zu ungenauen Ergebnissen führen kann. Dies ist der Fall, wenn eine der beiden Vorhersagen (CUDA oder OpenMP) eines Vorhersagemoduls im Vergleich zu den anderen Vorhersagen eine hohe Abweichung besitzt. Es gilt zu untersuchen, ob eine Regression, mit der die CUDA- und OpenMP-Vorhersage separat gewichtet werden, den Fehler minimieren und zu einem genaueren Gesamtergebnis führen.

- Weitere Vorhersagemethoden untersuchen: Für *CASEP* wurden zwei unterschiedliche Vorhersagemethoden untersucht: Eine benchmarkbasierte-Vorhersage und eine Vorhersage basierend auf Code-Analyse. In der Literatur finden sich jedoch auch andere Verfahren. Ein vielversprechendes Verfahren ist die modellbasierte Methode (s. Abschn. 2.4.2). Bei diesem Verfahren werden im ersten Schritt zwei Modelle erstellt. Ein Modell bildet die Architektur ab und ein zweites Modell die Software. Im weiteren Schritt wird mathematisch das Modell der Software auf die Hardware

abgebildet. Der Vorteil dieses Verfahrens besteht darin, dass die Berechnungen in kürzerer Zeit durchgeführt werden können als beispielsweise Benchmarks, die eine gewisse Anzahl wiederholt werden müssen, um Messfehler auszugleichen. Somit eignen sich modellbasierte Verfahren in der Regel besser, um in Echtzeit auf Veränderungen reagieren zu können (z.B. Laständerung auf der ausführenden Maschine). Ein Nachteil dieser Verfahren besteht darin, dass detaillierte Informationen über die Hardware vorhanden sein müssen (z.B. Pipeline-Parallelität), die in der Regel nicht automatisiert extrahiert werden können.

- Weitere Parallelisierungsmethoden unterstützen: Als Parallelisierungsmethode der Algorithmen wurde in dieser Arbeit eine gleichmäßige Aufteilung der Funktionen auf mehrere Threads untersucht. Es sind jedoch weitere Parallelisierungsmethoden möglich (s. Abschn. 2.3), die möglicherweise zu einer Erhöhung der Leistung führen. Es gilt zu untersuchen, unter welchen Bedingungen alternative Parallelisierungsmethoden zu einer kürzen Ausführungszeit führen. Weiterhin sind diese Muster sowohl auf die Vorhersagemodule als auch auf das Parallelisierungsmodul *PBA2CUDA* abzubilden.

- Optimierungen des PBA2CUDA-Moduls erweitern: Für das parallelisierungsmodul *PBA2CUDA* wurden zwei Optimierungsschritte vorgestellt, die die Laufzeit der *PBAs* verkürzen und die Ergebnisse verbessern sollen. Für zukünftige Arbeiten sind weitere Optimierungen vorstellbar. Beispielsweise kann die Speicherarchitektur von GPGPUs besser ausgenutzt werden. Hierzu können die Daten dynamisch in schnellere Speicherbereiche (z.B. Shared-Memory) kopiert werden, um Zugriffszeiten zu minimieren. Weiterhin können Kopieroperationen optimiert werden, indem die Daten asynchron auf die GPGPU kopiert werden. Dies führt dazu, dass für die Berechnung auf der GPGPU nicht bis zum kompletten Kopieren der Daten gewartet werden muss. Die Berechnungen auf der GPGPU können bereits gestartet werden, wenn ein Teil der Daten zur Verfügung steht. Dieser Schritt führt vor allem bei großen Datenmenge zu einer Verkürzung der Ausführungszeit.

Publikationen

[1] Ioannis Zgeras, Juergen Brehm, Michael Knoppik. Pba2cuda-a framework for parallelizing population based algorithms using cuda. *ARCS 2014*, 2014.

[2] Ioannis Zgeras. Casep - code analysis, speedup estimation and parallelization. *OC-DDC*, 2013.

[3] Ioannis Zgeras, Juergen Brehm, Mark Akselrod. Function based benchmarks to abstract parallel hardware and predict efficient code partitioning. *Architecture of Computing Systems*, 2013.

[4] Ioannis Zgeras, Juergen Brehm, Abdreas Reisch. Parallel function optimisation using evolutionary algorithms and deterministic neighbourhood search. *PARS*, 2011.

[5] Ioannis Zgeras, Juergen Brehm, Tobias Sprodowski. A model based approach for computing speedup on parallel machines using static code analysis. *PDCS*, 2011.

[6] Joerg Haehner Sven Tomforde, Ioannis Zgeras and Christian Mueller-Schloer. Adaptive control of sensor networks. *ATC*, 2010.

Referenzen

[1] Robert Yung, Stefan Rusu, and Ken Shoemaker. Future trend of microprocessor design. In *Solid-State Circuits Conference, 2002. ESSCIRC 2002. Proceedings of the 28th European*, pages 43–46. IEEE, 2002. 1

[2] Keith Underwood. Fpgas vs. cpus: trends in peak floating-point performance. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180. ACM, 2004. 1

[3] Joel M Tendler, J Steve Dodson, JS Fields, Hung Le, and Balaram Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002. 1, 10

[4] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005. 1

[5] John Owens. The gpgpu programming model. In *IEEE Visualization*, 2005. 1

[6] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. 1

[7] Shameen Akhter and Jason Roberts. *Multi-core programming*, volume 33. Intel press Hillsboro, 2006. 1

[8] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012. 1

[9] George Karypis. Introduction to parallel computing. *Addison Weslesy,*, 2003. 2

[10] Massimiliano Piscozzi. Cuda architecture, 2008. 2

[11] Rohit Chandra. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001. 2

[12] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990. 2

[13] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009. 2

[14] David Pellerin and Scott Thibault. *Practical FPGA programming in C*. Prentice Hall Press, 2005. 2

[15] Melanie Mitchell. *An Introduction to Genetic Algorithms (Complex Adaptive Systems)*. A Bradford Book, third printing edition, February 1998. 0262631857. 3, 23

[16] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988. 3, 23

[17] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942 –1948 vol.4, nov/dec 1995. 3, 29

[18] Haoqiang Jin, Michael Frumkin, and Jerry Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report, Technical Report NAS-99-011, NASA Ames Research Center, 1999. 6

[19] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In

Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 73–82. ACM, 2008. 6

[20] Stephen W Dunwell. Design objectives for the ibm stretch computer. In *Papers and discussions presented at the December 10-12, 1956, eastern joint computer conference: New developments in computers*, pages 20–22. ACM, 1956. 10

[21] Robert M Metcalfe and David R Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976. 10

[22] Gregory F Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001. 10

[23] Marc Snir. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998. 10

[24] Don Anderson, Tom Shanley, and Ravi Budruk. *PCI express system architecture*. Addison-Wesley Professional, 2004. 10

[25] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010. 12, 17

[26] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. 13

[27] Fredric Stuart. *Fortran programming*. Wiley, 1970. 13

[28] Enhua Wu and Youquan Liu. Emerging technology about gpgpu. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 618–622. IEEE, 2008. 14

[29] David Luebke and Greg Humphreys. How gpus work. *Computer*, 40(2):96–100, 2007. 14

[30] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008. 14

[31] CUDA NVidia. C programming guide version 5.0. 16

[32] K. Kothapalli, R. Mukherjee, M.S. Rehman, S. Patidar, P.J. Narayanan, and K. Srinathan. A performance prediction model for the cuda gpgpu platform. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 463 –472, dec. 2009. 17, 35, 37

[33] Mark Harris et al. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2:45, 2007. 17, 89

[34] David Tarjan, Kevin Skadron, and Paulius Micikevicius. The art of performance tuning for cuda and manycore architectures. *Birds-of-a-feather session at SC'09*, 2009. 17

[35] CUDA Nvidia. Nvidia cuda programming guide, 2011. 17

[36] NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara 95050, USA. *CUDA C Best Practices Guide*, 4.0 edition, May 2011. 17, 121

[37] Leandro N De Castro and Fernando J Von Zuben. *Recent developments in biologically inspired computing*. Igi Global, 2005. 20

[38] James G March. Exploration and exploitation in organizational learning. *Organization science*, 2(1):71–87, 1991. 21

[39] Emad Elbeltagi, Tarek Hegazy, and Donald Grierson. Comparison among five evolutionary-based optimization algorithms. *Advanced engineering informatics*, 19(1):43–53, 2005. 23

[40] Mandavilli Srinivas and Lalit M Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994. 24, 25

[41] Russell C Eberhart and Yuhui Shi. Comparison between genetic algorithms and particle swarm optimization. In *Evolutionary Programming VII*, pages 611–616. Springer, 1998. 26

[42] John J Grefenstette et al. Genetic algorithms for changing environments. In *PPSN*, volume 2, pages 137–144, 1992. 26

[43] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. *Urbana*, 51:61801–2996, 1991. 28, 29

[44] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1:33–57, 2007. 10.1007/s11721-007-0002-0. 29, 31

[45] Keith Balmer, Robert J Gove, Karl M Guttag, and Nicholas K Ing-Simmons. Multi-processor reconfigurable in single instruction multiple data (simd) and multiple instruction multiple data (mimd) modes and method of operation, May 18 1993. US Patent 5,212,777. 32

[46] Erick Cantú-Paz. A summary of research on parallel genetic algorithms. 1995. 33

[47] Jian-Ming Li, Xiao-Jing Wang, Rong-Sheng He, and Zhong-Xian Chi. An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In *Network and parallel computing workshops, 2007. NPC workshops. IFIP international conference on*, pages 855–862. IEEE, 2007. 34

[48] Jianming Li, Xiangpei Hu, Zhanlong Pang, and Kunming Qian. A parallel ant colony optimization algorithm based on fine-grained model with gpu acceleration. *International Journal of Innovative Computing, Information and Control*, 5(11):3707–3716, 2009. 34

[49] Bo Li and Koichi Wada. Parallelizing particle swarm optimization. In *Communications, Computers and signal Processing, 2005. PACRIM. 2005 IEEE Pacific Rim Conference on*, pages 288–291. IEEE, 2005. 34

[50] You Zhou and Ying Tan. Gpu-based parallel particle swarm optimization. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 1493–1500. IEEE, 2009. 34

[51] Sanaz Mostaghim, Jürgen Branke, and Hartmut Schmeck. Multi-objective particle swarm optimization on computer grids. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 869–875. ACM, 2007. 34

[52] Kuk-Hyun Han, Kui-Hong Park, Ci-Ho Lee, and Jong-Hwan Kim. Parallel quantum-inspired genetic algorithm for combinatorial optimization problem. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 2, pages 1422–1429. IEEE, 2001. 34

[53] Tij Si H. J. Curnow, B. A. Wichmann. A synthetic benchmark. In *The Computer Journal*, pages 43–49, 1973. 35

[54] Karl Fürlinger and Michael Gerndt. ompp: A profiling tool for openmp. In *OpenMP Shared Memory Parallel Programming*, pages 15–23. Springer, 2008. 35

[55] NVIDIA. Nvidia parallel nsight. 35

[56] Papadopoulou, Sadooghi-Alvandi, and Henry Wong. Micro-benchmarking the gt200 gpu. *Processing*, pages 235–246, 2009. 35

[57] R. Taylor and XiaoMing Li. A micro-benchmark suite for amd gpus. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 387 –396, sept. 2010. 36

[58] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, WesleyB. Jones, and Bodo Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In Rudolf Eigenmann and MichaelJ. Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 2104 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 2001. 36

[59] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM. 36

[60] B. Gruppe. *Benchmark-Software: 3dmark, Everest, Linpack, Sisoftware Sandra, Benchit, Super Pi, Fraps.* General Books LLC, 2010. 9781158815784. 36

[61] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, October 2009. 36

[62] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: parallel speedup estimates for serial programs. In *ACM SIGPLAN Notices*, volume 46, pages 519–536. ACM, 2011. 37

[63] David Ofelt and John L Hennessy. *Efficient performance prediction for modern microprocessors*, volume 28. ACM, 2000. 38

[64] John A Stratton, Sam S Stone, and W Hwu Wen-mei. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In *Languages and Compilers for Parallel Computing*, pages 16–30. Springer, 2008. 38

[65] Michael Knoppik. Gesteuerte parallelisierung populations-basierter algorithmen. Master's thesis, Leibniz Universitaet Hannover, 2013. 39, 42

[66] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16. IEEE Computer Society, 2004. 39

[67] Seyong Lee and Jeffrey S Vetter. Early evaluation of directive-based gpu programming models for productive exascale computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 23. IEEE Computer Society Press, 2012. 39, 40, 41

[68] Michael Wolfe. Implementing the pgi accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50. ACM, 2010. 39

[69] Ruymán Reyes, Iván López-Rodríguez, Juan J Fumero, and Francisco de Sande. accull: An openacc implementation with cuda and opencl support. In *Euro-Par 2012 Parallel Processing*, pages 871–882. Springer, 2012. 40

[70] Stéphane Bihan, Georges-Emmanuel Moulard, Romain Dolbeau, Henri Calandra, and Rached Abdelkhalek. Directive-based heterogeneous programming–a gpu-accelerated rtm use case. In *Proceedings of the 7th International Conference on Computing, Communications and Control Technologies*, 2009. 40

[71] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010. 40, 107

[72] Seyong Lee and Rudolf Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010. 41, 73

[73] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. 41

[74] Tianyi David Han and Tarek S Abdelrahman. hi cuda: a high-level directive-based language for gpu programming. In *Proceedings of 2nd*

Workshop on General Purpose Processing on Graphics Processing Units, pages 52–61. ACM, 2009. 41

[75] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 51–61. ACM, 2010. 41

[76] Qing Yi Quinlan Dan, Rich Vuduc and Markus Schordan. Rose compiler infrastructure, 2012. 43

[77] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilerbau*. Addison-Wesley, 1988. 43

[78] Richard Wain, Ian Bush, Martyn Guest, Miles Deegan, Igor Kozin, and Christine Kitchen. *An overview of FPGAs and FPGA programming: Initial experiences at Daresbury*. Council for the Central Laboratory of the Research Councils, 2006. 51

[79] Taho Dorta, Jaime Jiménez, José Luis Martín, Unai Bidarte, and Armando Astarloa. Overview of fpga-based multiprocessor systems. In *Reconfigurable Computing and FPGAs, 2009. ReConFig'09. International Conference on*, pages 273–278. IEEE, 2009. 51

[80] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010. 59

[81] Reinhold P Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984. 63

[82] Harold J Curnow and Brian A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, 1976. 63

[83] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991. 63

[84] David B Golub, Randall W Dean, Alessandro Forin, and Richard F Rashid. Unix as an application program. In *UsENIX summer*, pages 87–95, 1990. 63

[85] Jakob Vesterstrom and Rene Thomsen. A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, pages 1980–1987. IEEE, 2004. 67, 112

[86] Leona S Aiken, Stephen G West, and Steven C Pitts. Multiple linear regression. *Handbook of psychology*, 2003. 78

[87] Todd L Veldhuizen and M Ed Jernigan. Will c++ be faster than fortran? In *Scientific Computing in Object-Oriented Parallel Environments*, pages 49–56. Springer, 1997. 104

[88] Lutz Prechelt. An empirical comparison of c, c++, java, perl, python, rexx and tcl. *IEEE Computer*, 33(10):23–29, 2000. 104

[89] John Backus. The history of fortran i, ii, and iii. In *History of programming languages I*, pages 25–74. ACM, 1978. 104

[90] Yonghong Yan, Max Grossman, and Vivek Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *Euro-Par 2009 Parallel Processing*, pages 887–899. Springer, 2009. 105

[91] T Keishiro. J2c java. class to c translator. *URL: http://www. webity. co. jp/-info/andoh/java/j2c. html*, 1995. 105

[92] Teuvo Kohonen. An introduction to neural computing. *Neural networks*, 1(1):3–16, 1988. 105

[93] Jayram Moorkanikara Nageswaran, Nikil Dutt, Jeffrey L Krichmar, Alex Nicolau, and Alex Veidenbaum. Efficient simulation of large-scale spiking neural networks using cuda graphics processors. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 2145–2152. IEEE, 2009. 105

[94] Honghoon Jang, Anjin Park, and Keechul Jung. Neural network implementation using cuda and openmp. In *Digital Image Computing: Techniques and Applications (DICTA), 2008*, pages 155–161. IEEE, 2008. 105

[95] W.M.W. Hwu. Automatic translation of cuda to opencl and comparison of performance optimizations on gpus. 2011. 107

[96] Matt J Harvey and Gianni De Fabritiis. Swan: A tool for porting cuda programs to opencl. *Computer Physics Communications*, 182(4):1093–1099, 2011. 107

[97] Emre Cakar. *Population-based runtime optimisation in static and dynamic environments*. PhD thesis, 2011. 111, 112, 113

[98] Weicai Zhong, Jing Liu, Mingzhi Xue, and Licheng Jiao. A multiagent genetic algorithm for global numerical optimization. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(2):1128–1141, 2004. 112

[99] Ganid Srimool, Putchong Uthayopas, and Juta Pichitlamkhen. Speeding up a large logistics optimization problems using gpu technology. In *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2011 8th International Conference on*, pages 450–454. IEEE, 2011. 121

[100] Patrice Roger Calégari. *Parallelization of population-based evolutionary algorithms for combinatorial optimization problems*. PhD thesis, Université Claude Bernard, Lyon, France, 1999.

[101] Erick Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.

[102] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[103] Mark Harris. Optimizing cuda. *SC07: High Performance Computing With CUDA*, 2007.

[104] Randy L Haupt and Sue Ellen Haupt. *Practical genetic algorithms*. John Wiley & Sons, 2004.

[105] Georg Heinze, Christian Hubrich, and Thomas Halfmann. Stopped light and image storage by electromagnetically induced transparency up to the regime of one minute. *Phys. Rev. Lett.*, 111:033601, Jul 2013.

[106] Peter Kok Keong Loh, Wen Jing Hsu, Cai Wentong, and Nadarajah Sriskanthan. How network topology affects dynamic loading balancing. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 4(3):25–35, 1996.

[107] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Gpu-based island model for evolutionary algorithms. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10*, page 1089, New York, New York, USA, July 2010. ACM Press.

[108] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[109] Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eliezer Weissmann. Power-management architecture of the intel

microarchitecture code-named sandy bridge. *Micro, IEEE*, 32(2):20–27, 2012.

[110] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, October 1984.

[111] Henry Wong, Misel myrto Papadopoulou, Maryam Sadooghi-alv, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *In ISPASS*, pages 235–246, 2010.

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang

A Ergebnisse des GAs - Benchmarker und Code-Analysis getrennt

A.1 Ergebnisse der Funktion f1

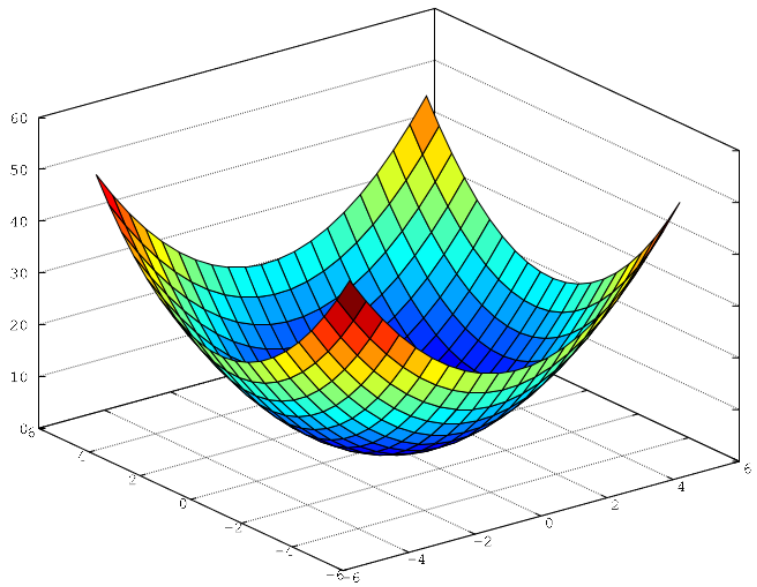
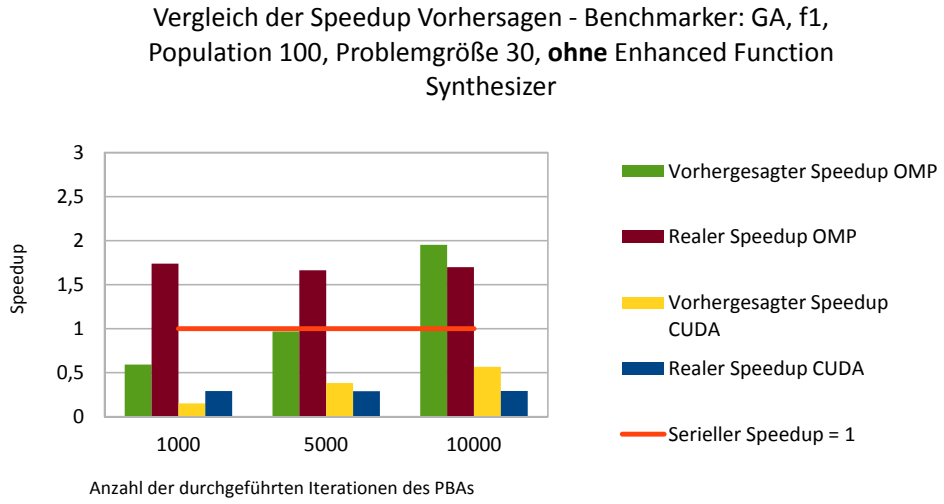
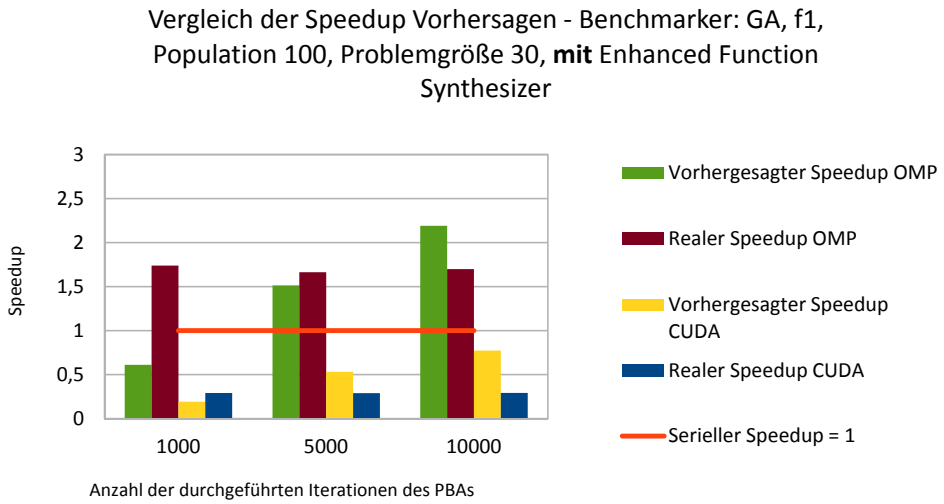


Abbildung A.1: Funktion f1





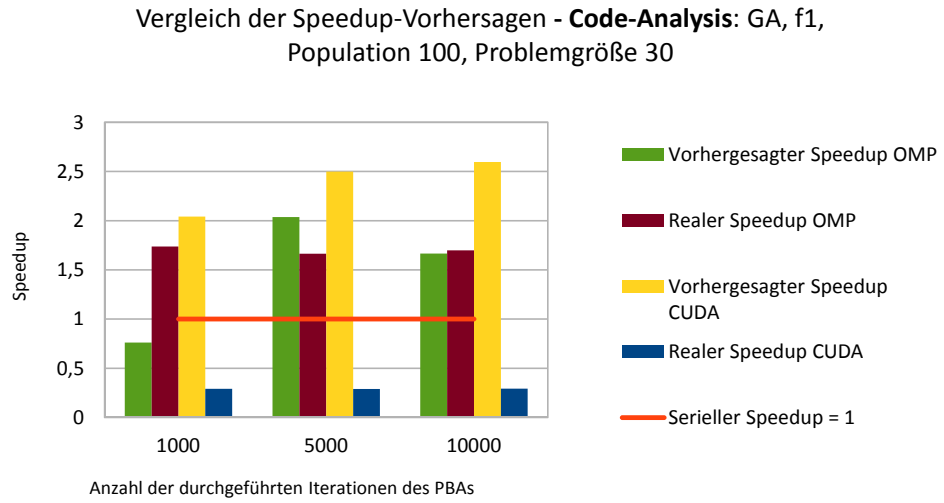
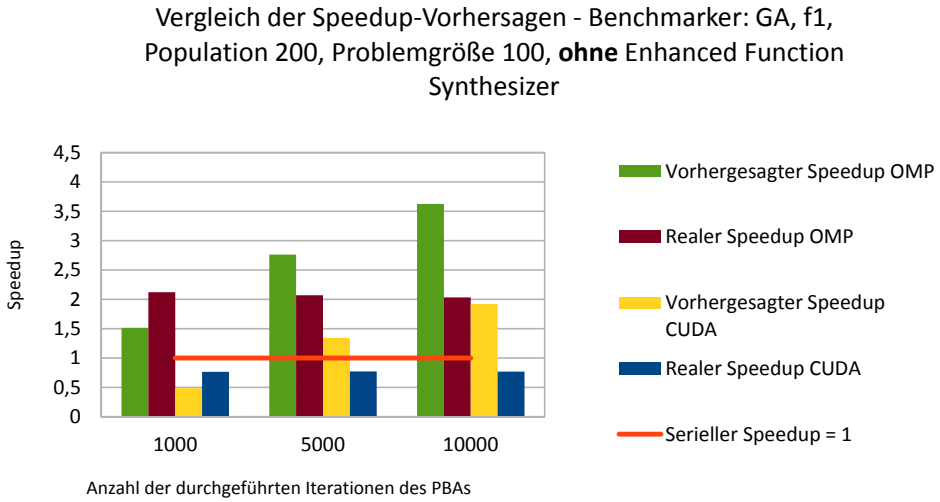
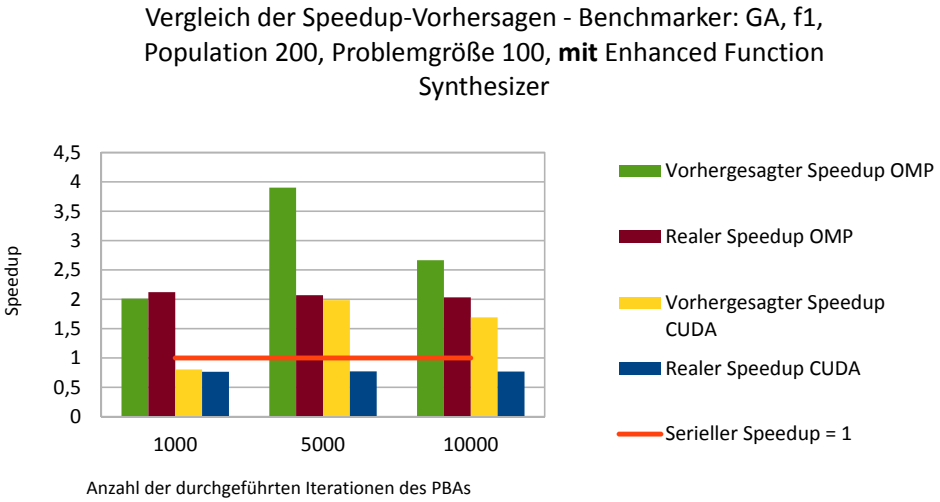


Abbildung A.2: Populationsgröße 100 und Problemgröße 30





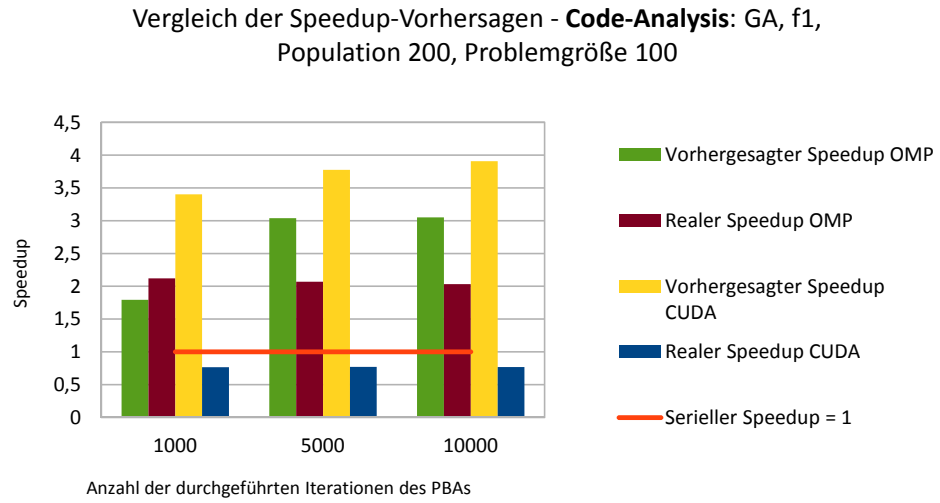
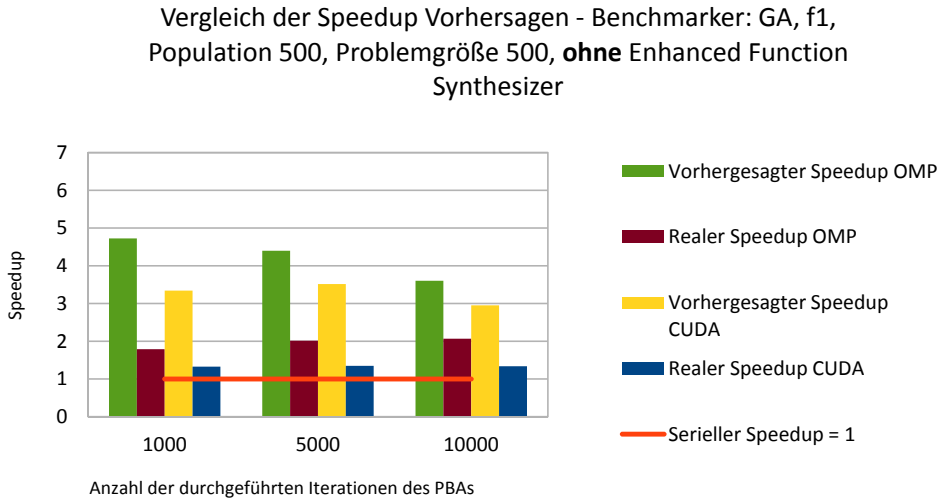
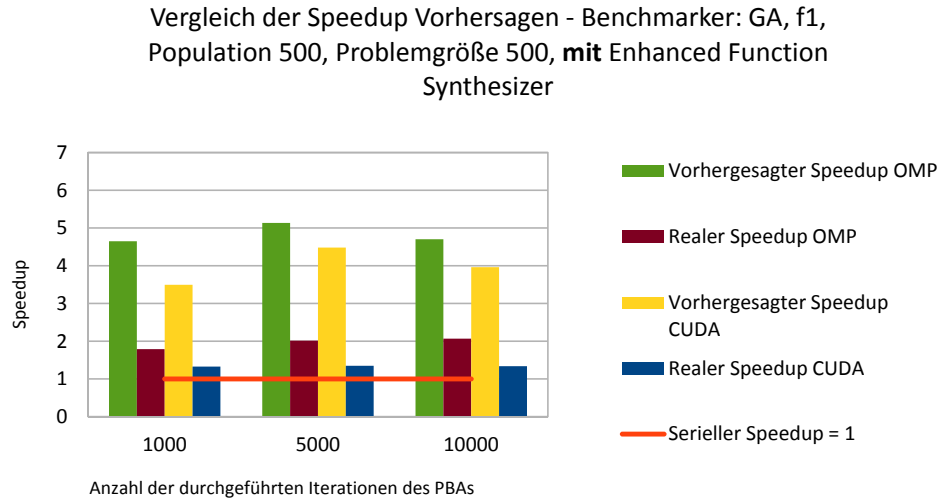


Abbildung A.3: Populationsgröße 200 und Problemgröße 100





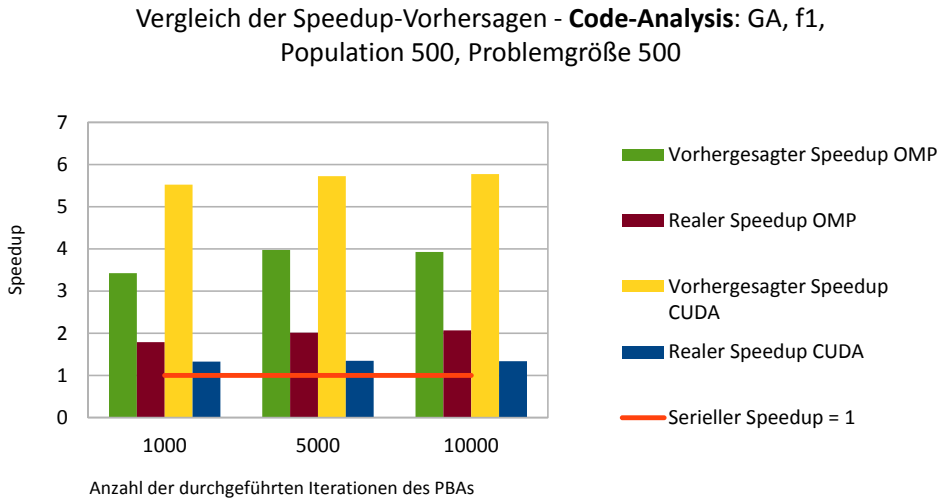


Abbildung A.4: Populationsgröße 500 und Problemgröße 500

A.2 Ergebnisse der Funktion f5

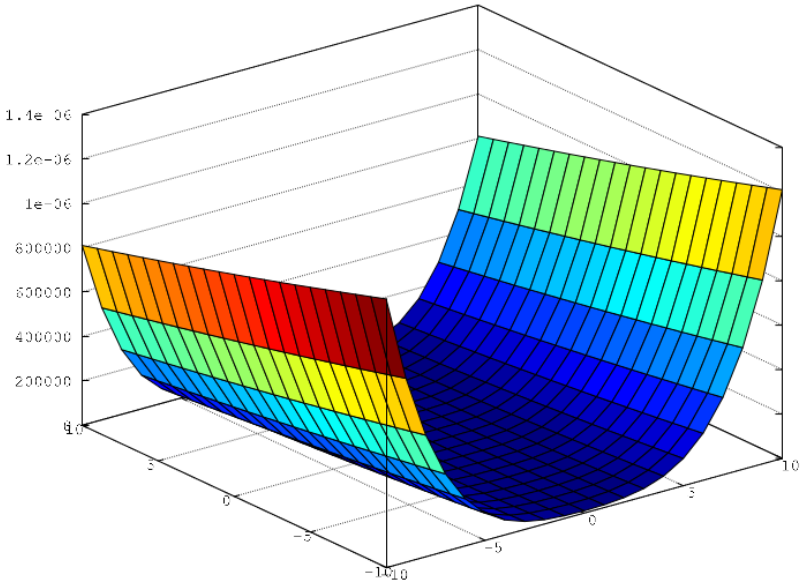
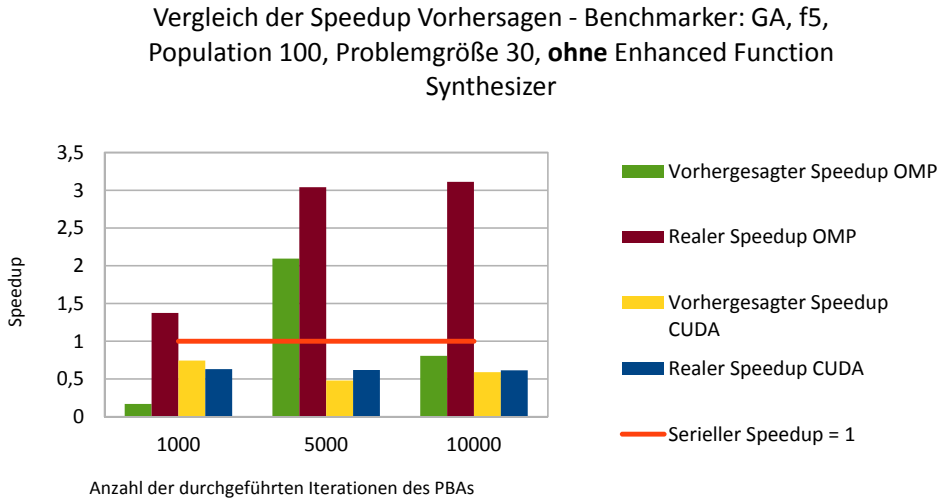
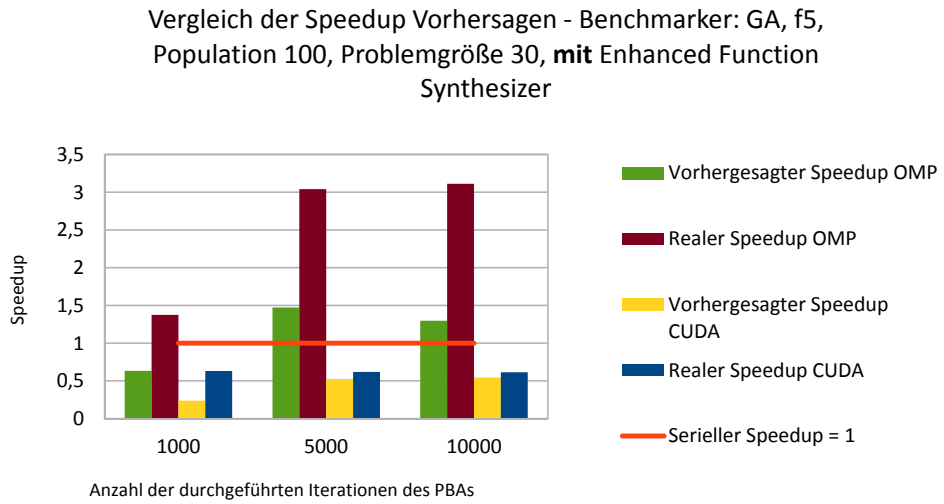


Abbildung A.5: Funktion f5





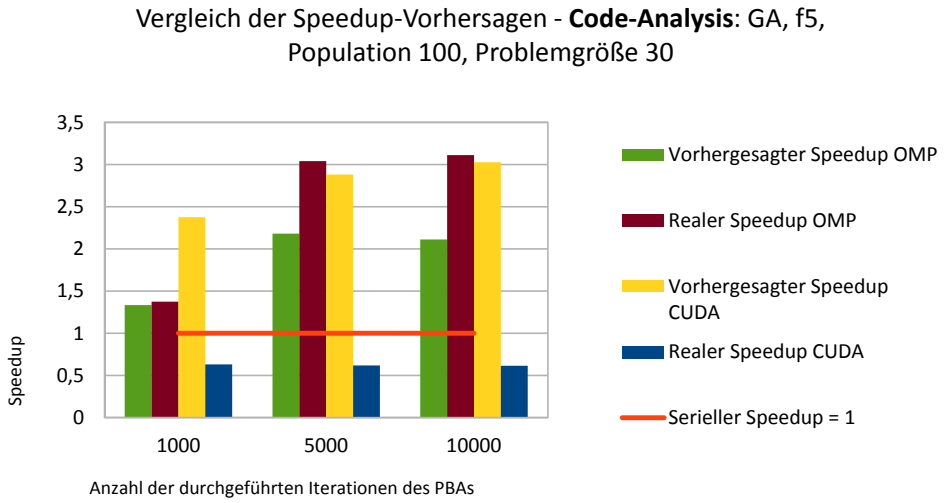
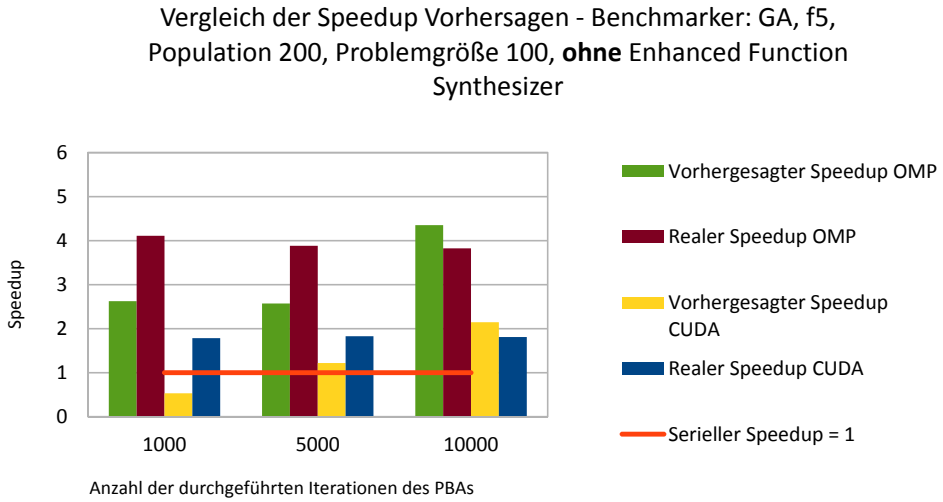
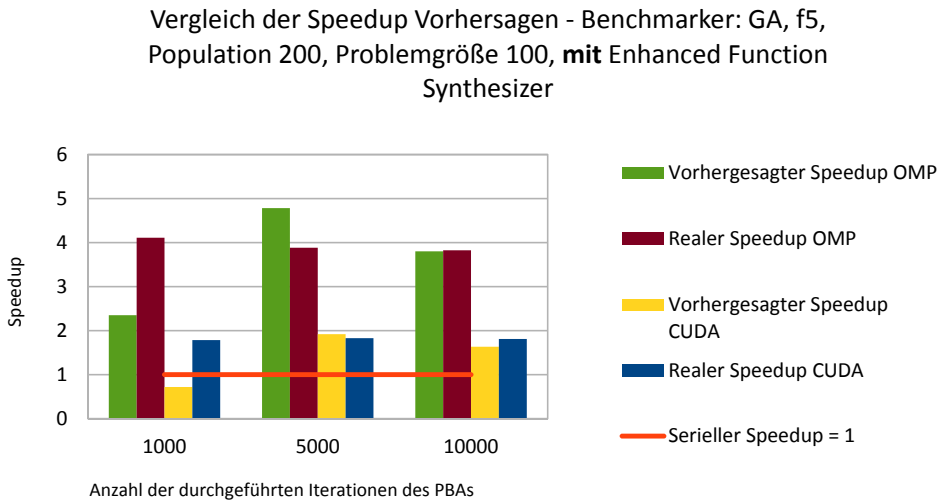


Abbildung A.6: Populationsgröße 100 und Problemgröße 30





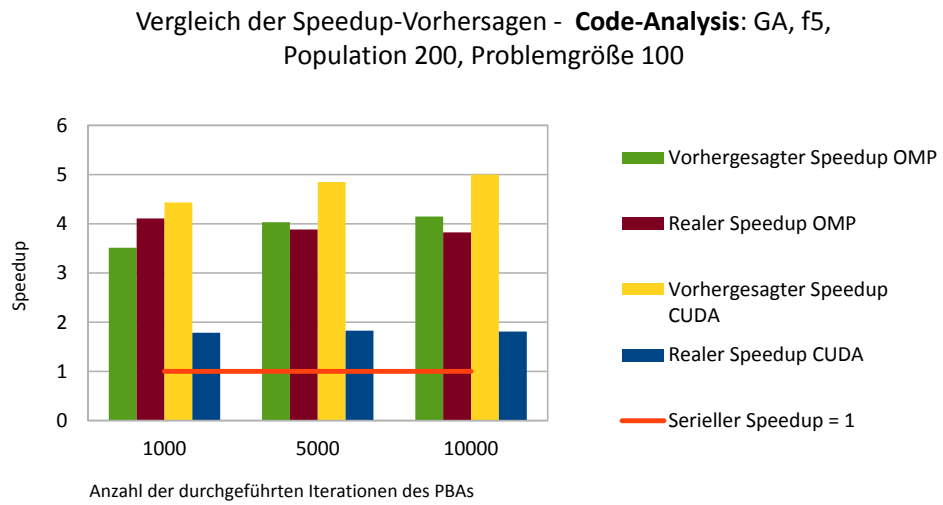
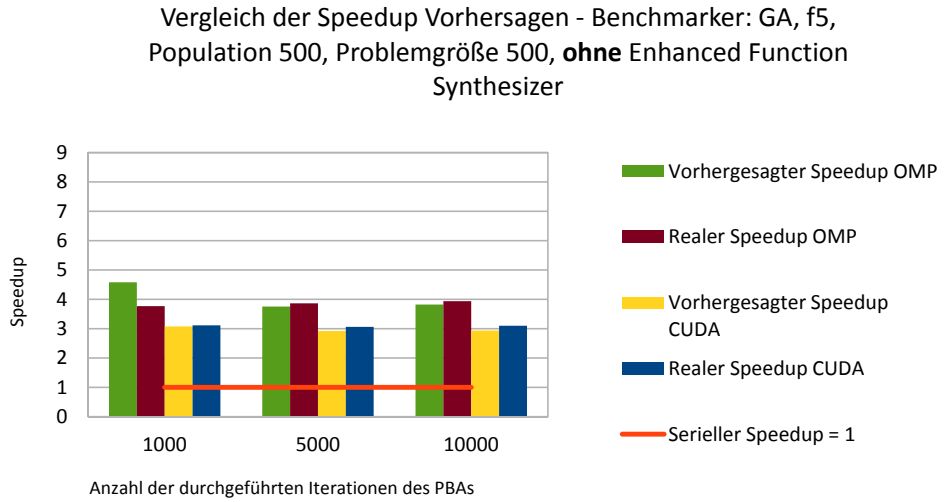
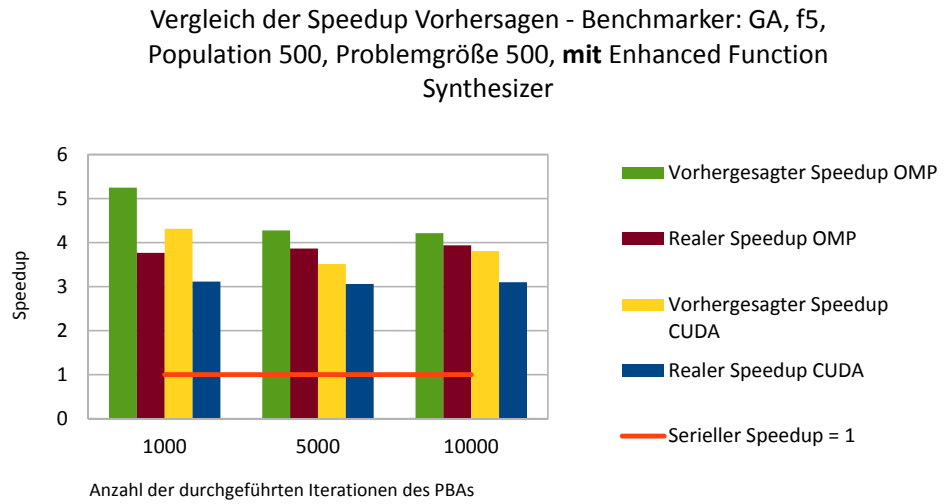


Abbildung A.7: Populationsgröße 200 und Problemgröße 100





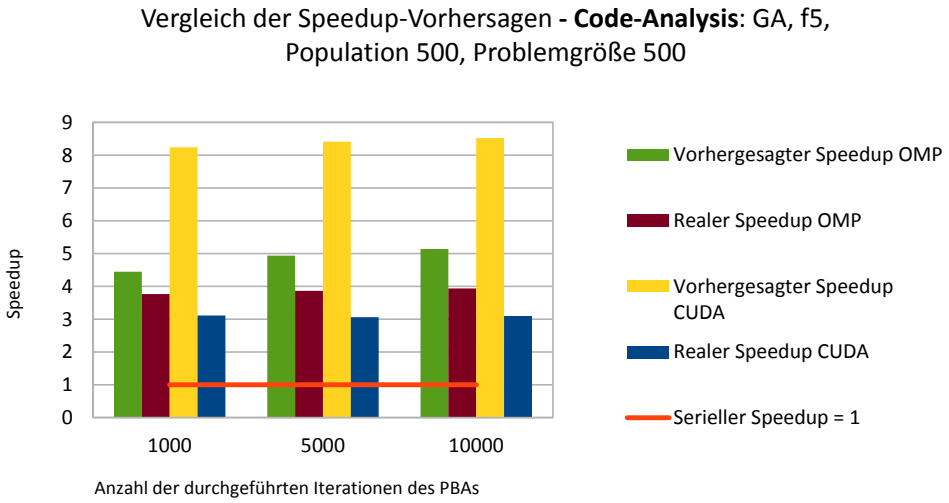


Abbildung A.8: Populationsgröße 500 und Problemgröße 500

A.3 Ergebnisse der Funktion f8

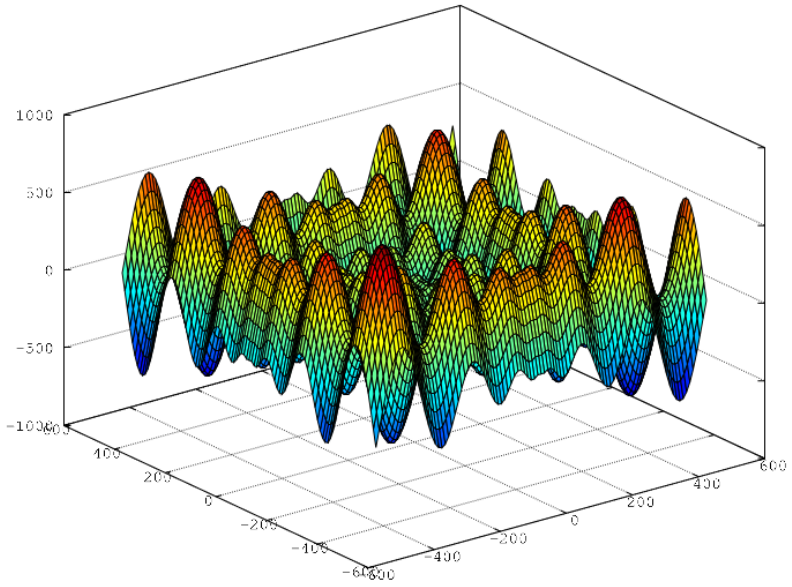
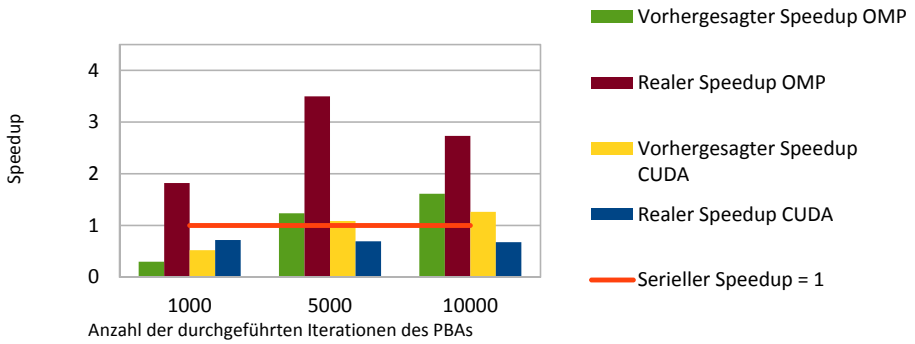
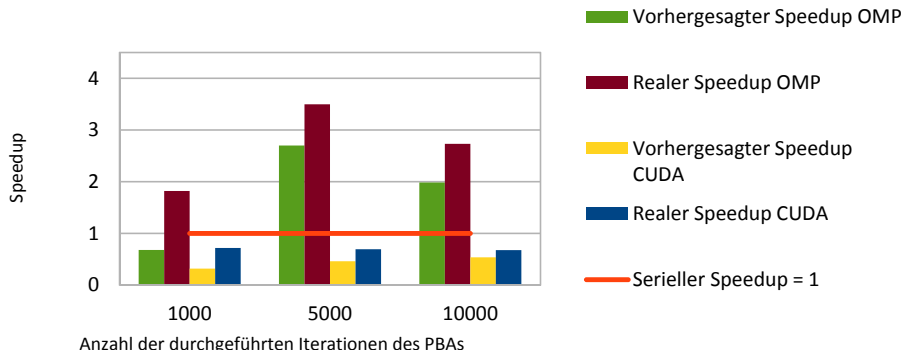


Abbildung A.9: Funktion f8

Vergleich der Speedup Vorhersagen - Benchmarker: GA, f8, Population 100, Problemgröße 30, **ohne** Enhanced Function Synthesizer



Vergleich der Speedup Vorhersagen - Benchmarker: GA, f8, Population 100, Problemgröße 30, **mit** Enhanced Function Synthesizer



Vergleich der Speedup-Vorhersagen - **Code-Analysis**: GA, f8, Population 100, Problemgröße 30

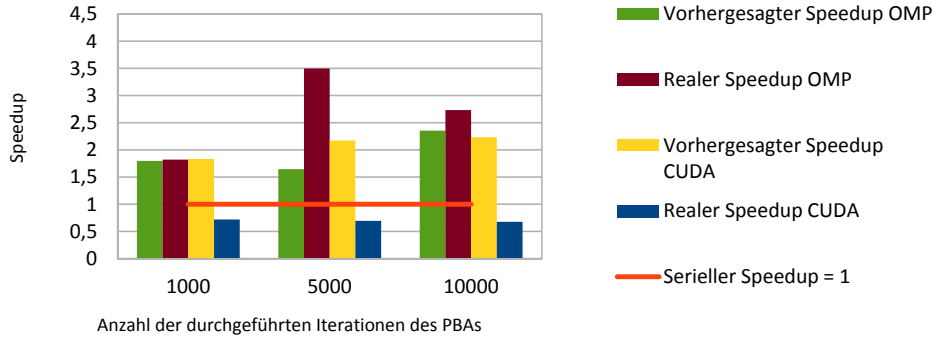
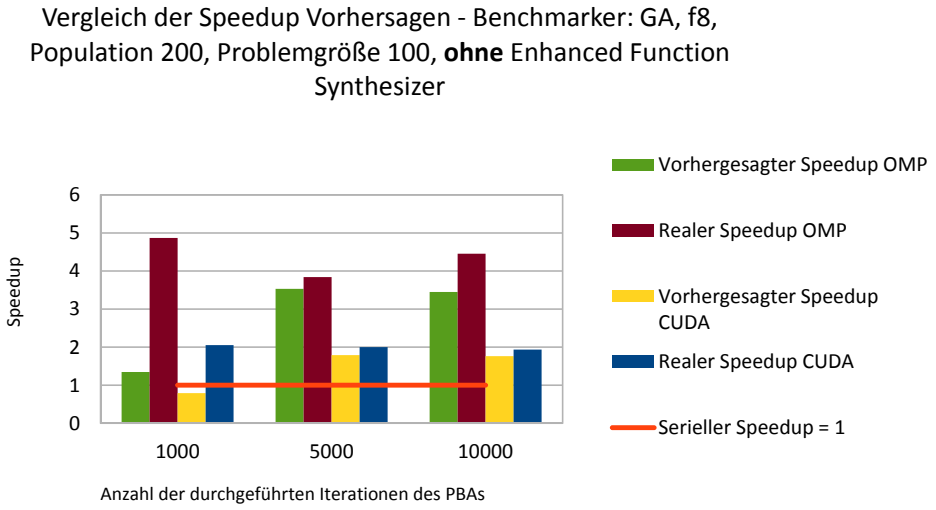
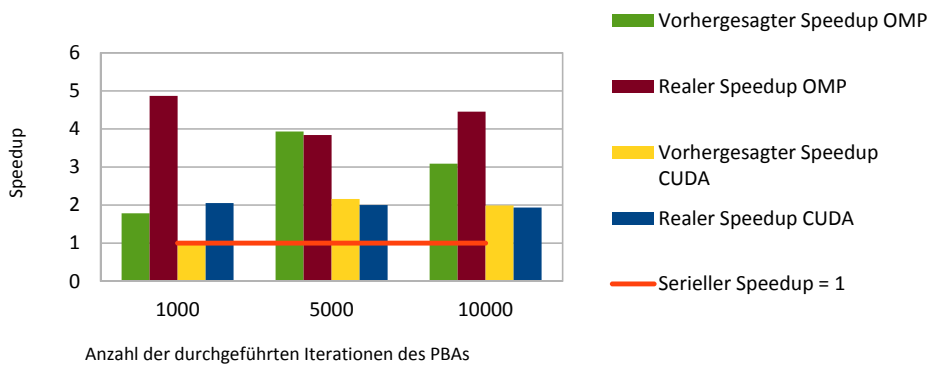


Abbildung A.10: Populationsgröße 100 und Problemgröße 30





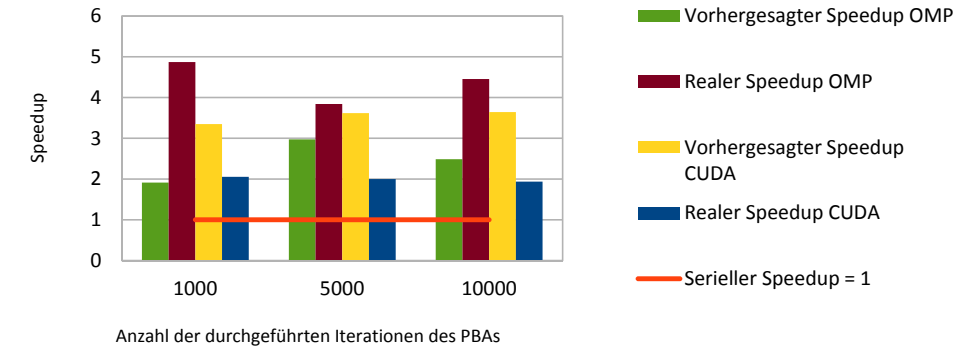
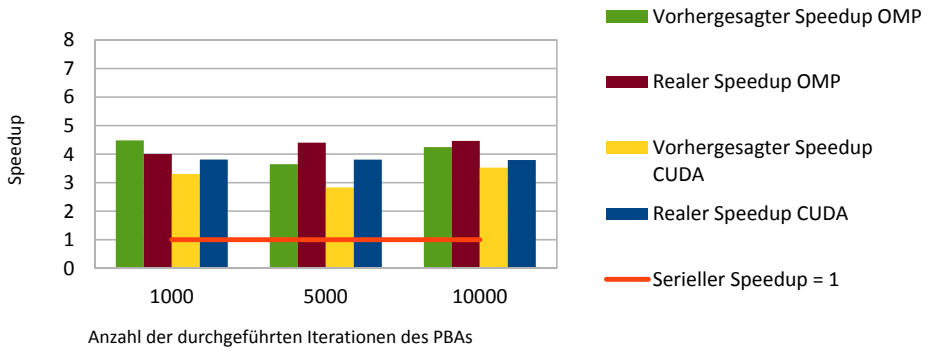
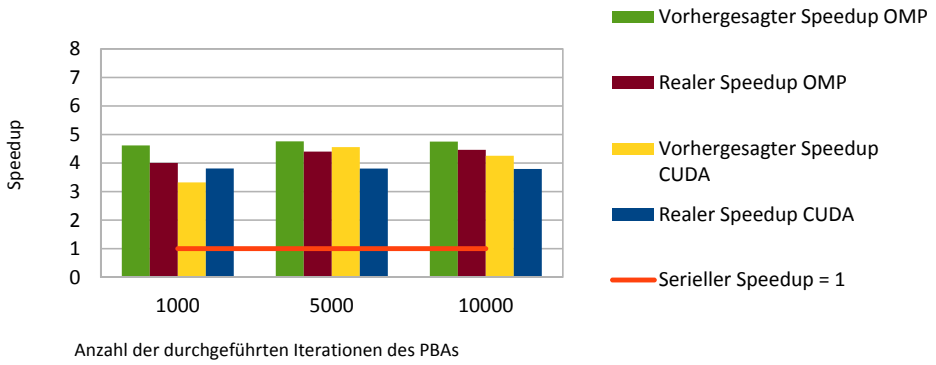


Abbildung A.11: Populationsgröße 200 und Problemgröße 100

Vergleich der Speedup Vorhersagen - Benchmarker: GA, f8, Population 500, Problemgröße 500, **ohne** Enhanced Function Synthesizer



Vergleich der Speedup Vorhersagen - Benchmarker: GA, f8, Population 500, Problemgröße 500, **mit** Enhanced Function Synthesizer



Vergleich der Speedup-Vorhersagen - **Code-Analysis**: GA, f8, Population 500, Problemgröße 500

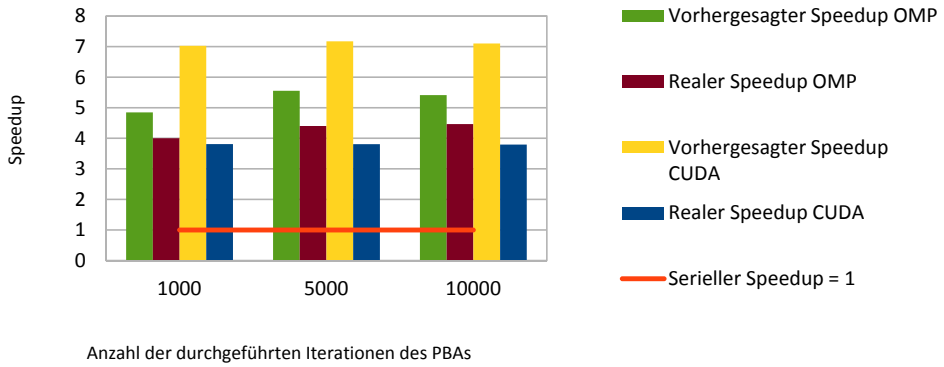


Abbildung A.12: Populationsgröße 500 und Problemgröße 500

A.4 Ergebnisse der Funktion f11

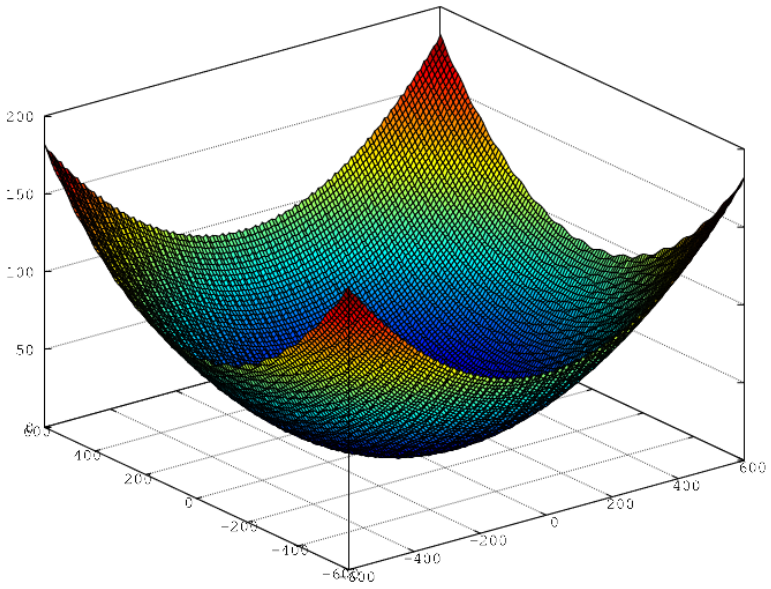
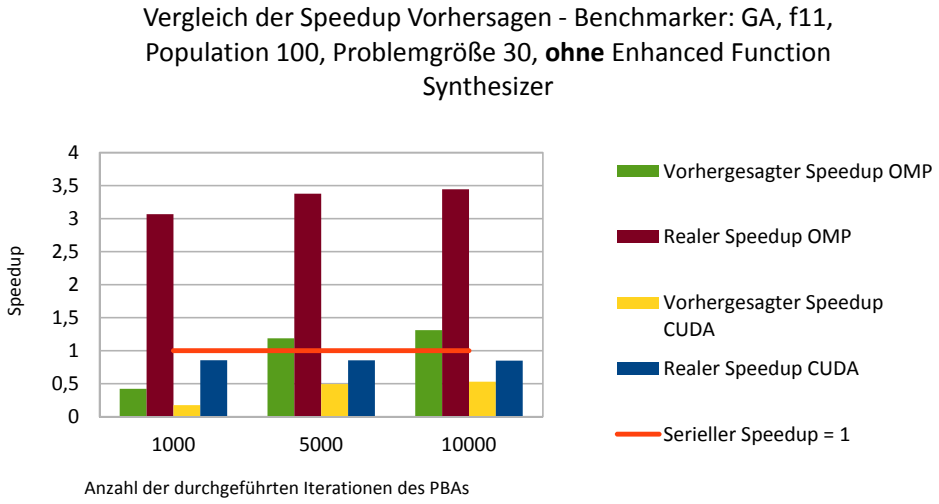
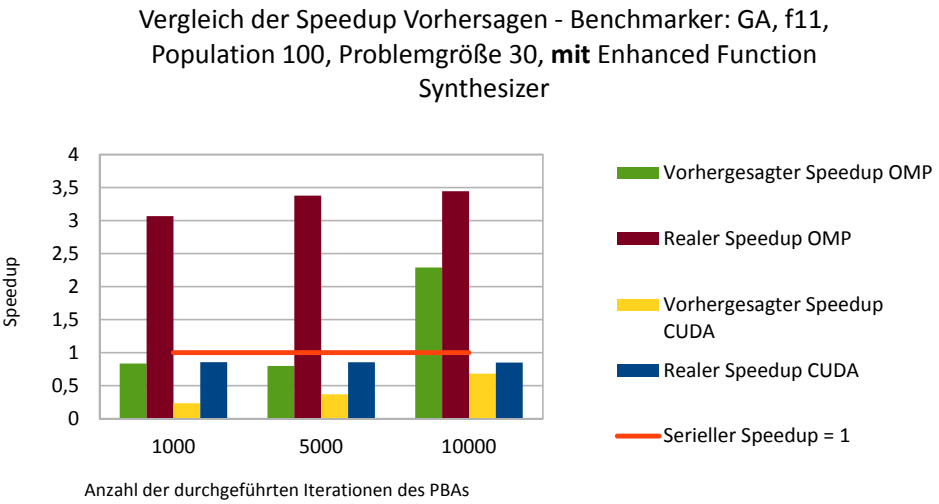


Abbildung A.13: Funktion f11





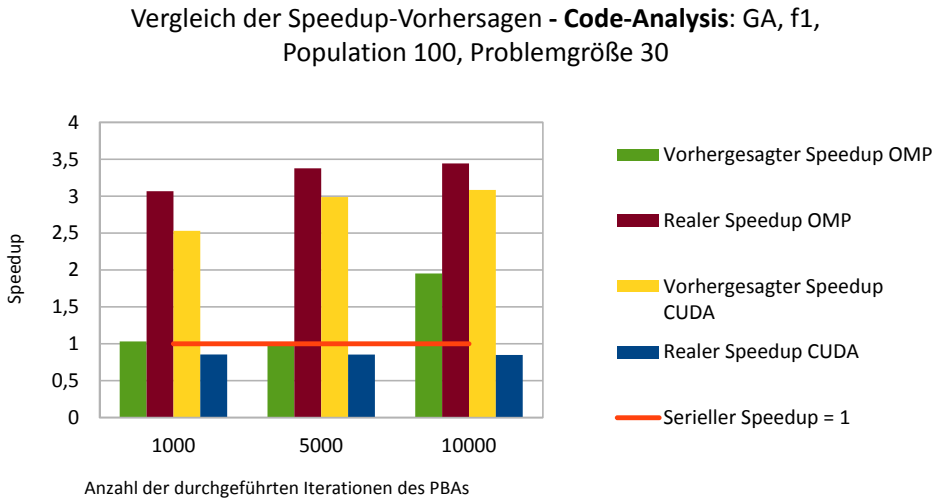
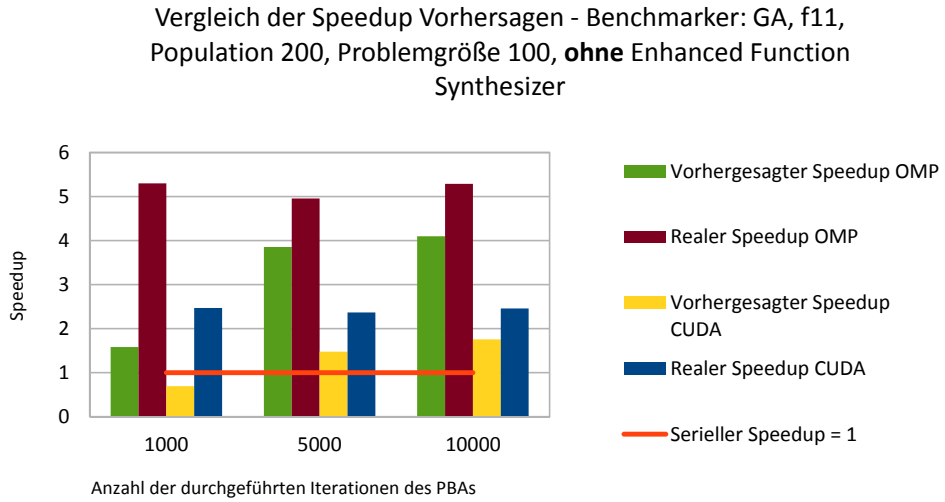
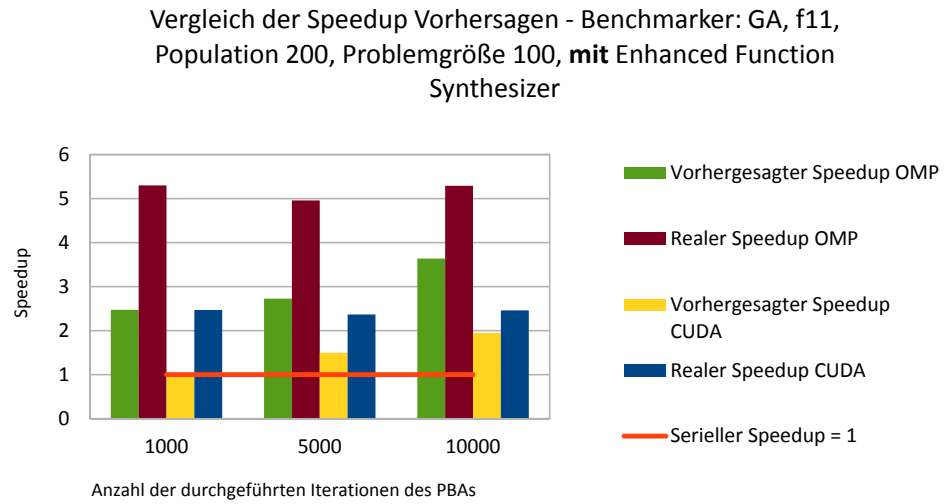


Abbildung A.14: Populationsgröße 100 und Problemgröße 30





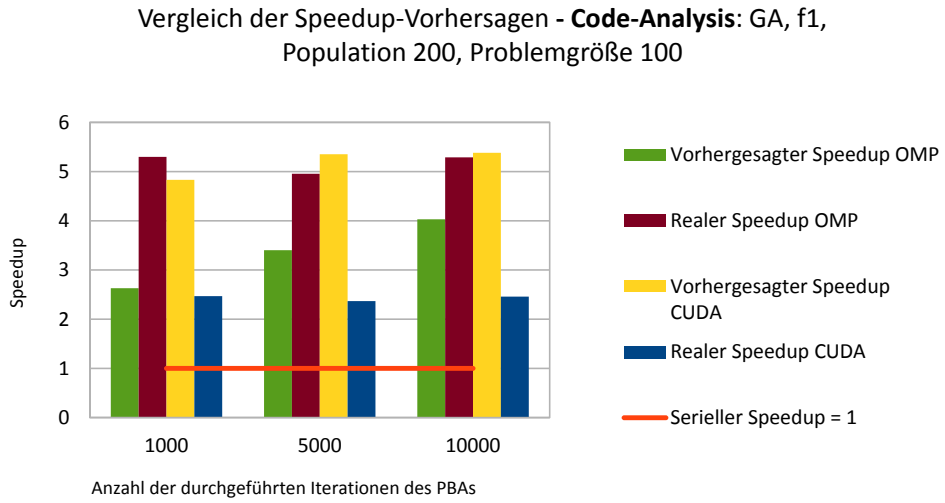
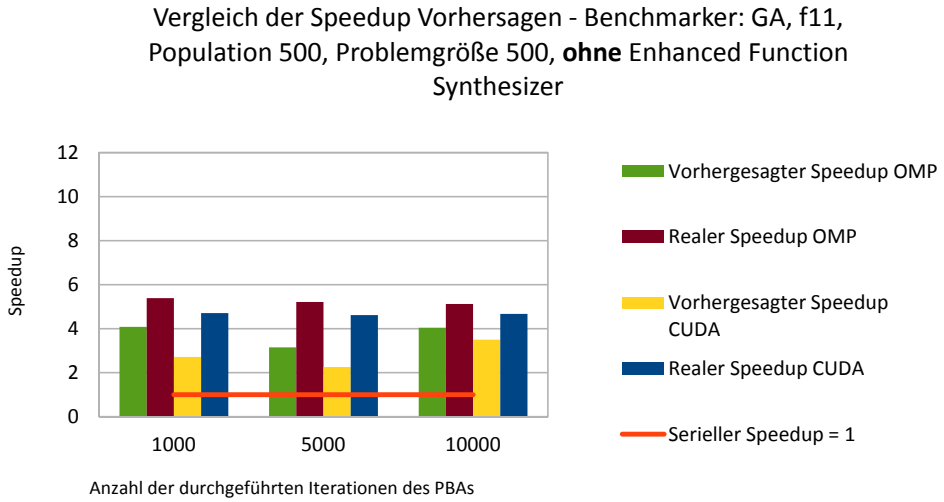
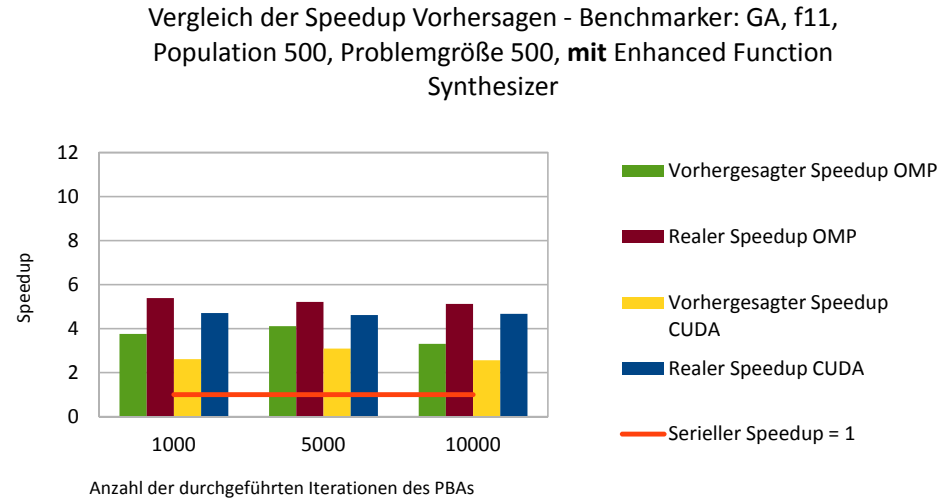


Abbildung A.15: Populationsgröße 200 und Problemgröße 100





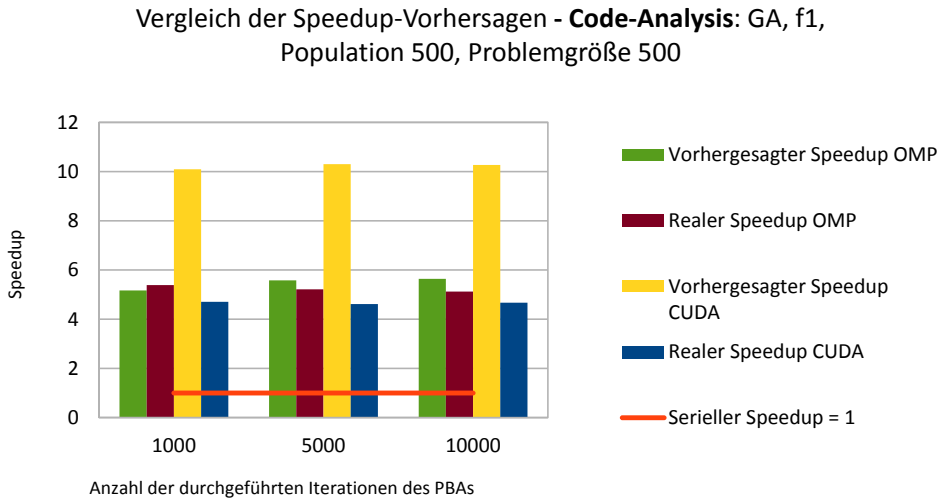


Abbildung A.16: Populationsgröße 500 und Problemgröße 500

B Ergebnisse des PSOs - Benchmarker und Code-Analysis getrennt

B.1 Ergebnisse der Funktion f1

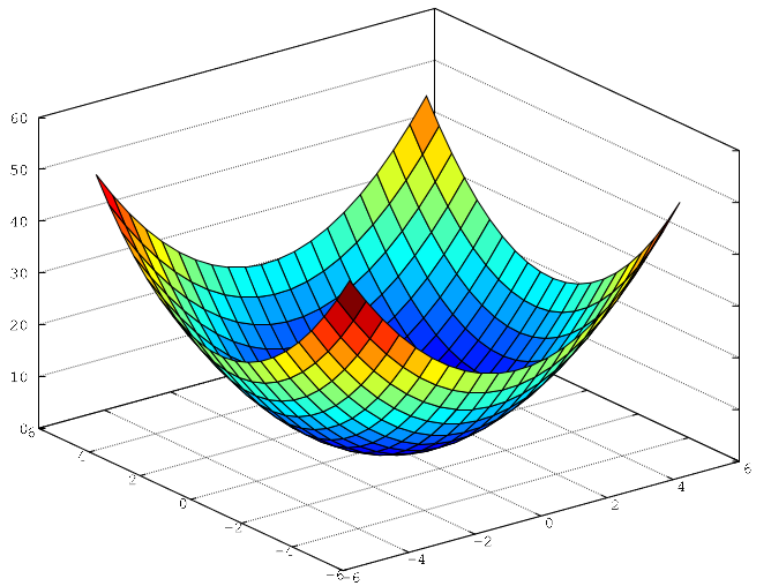
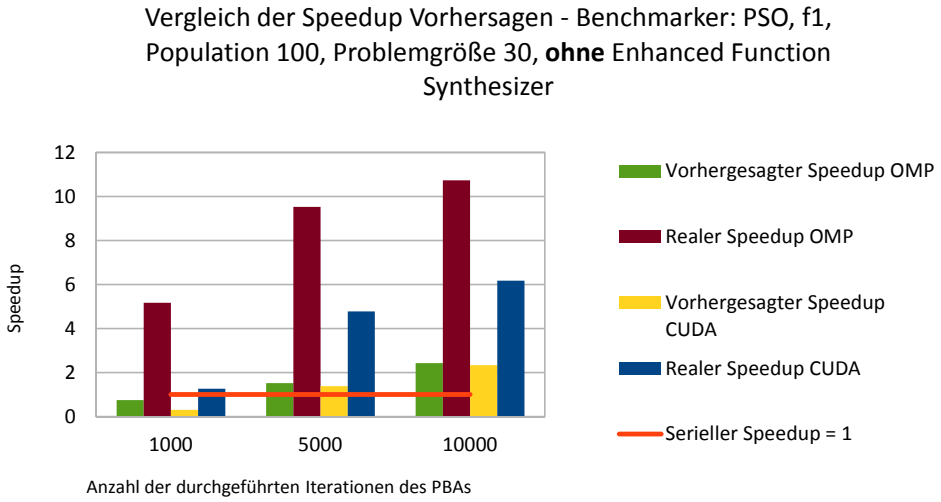
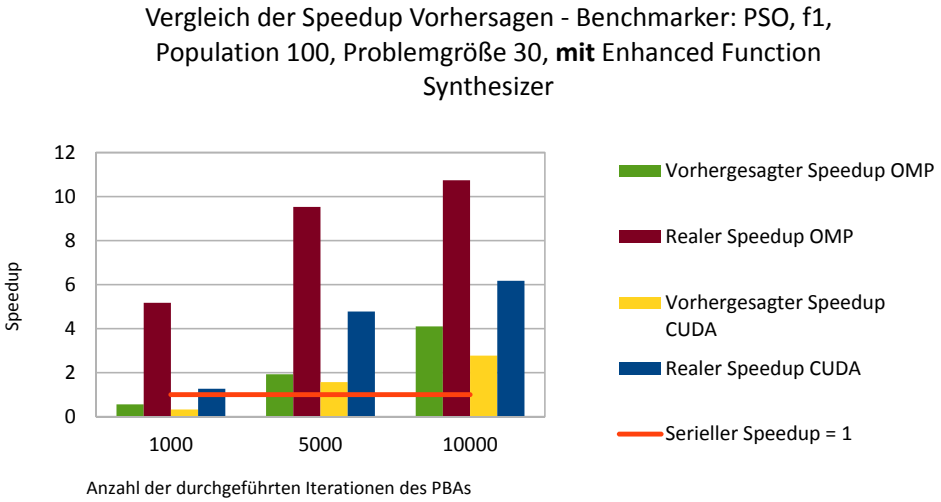


Abbildung B.1: Funktion f1





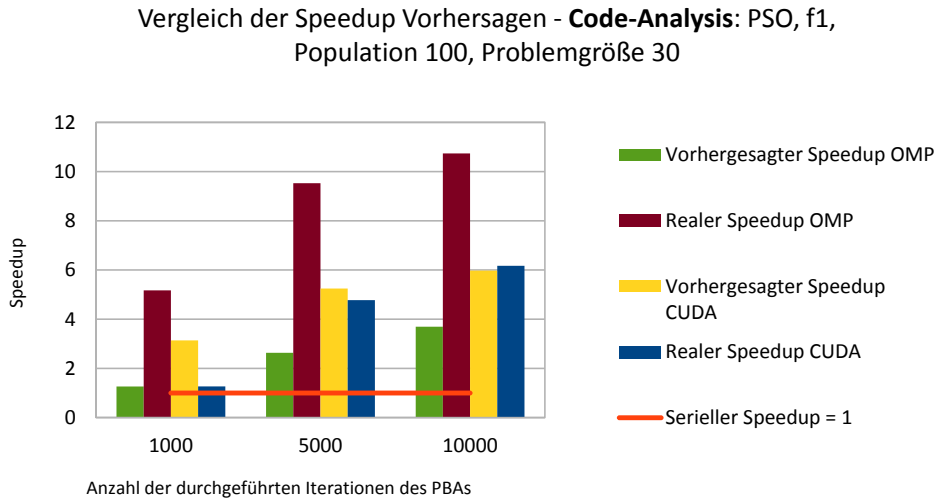
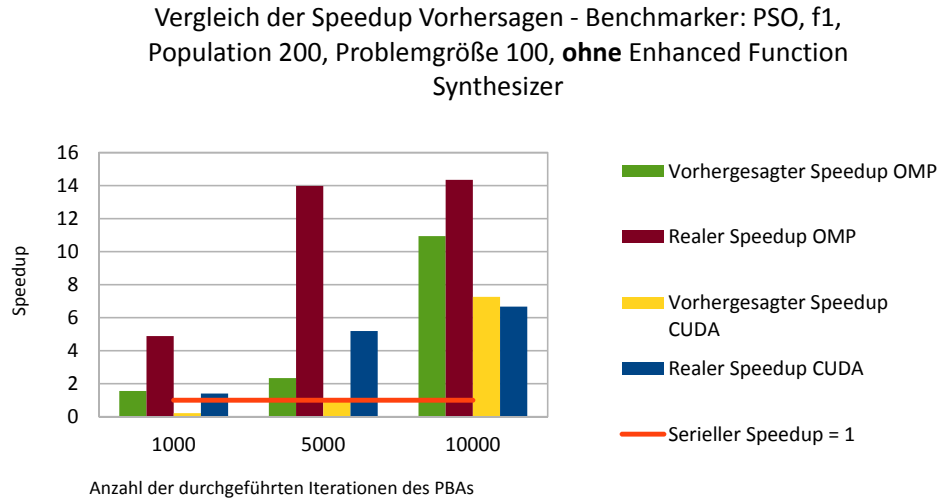
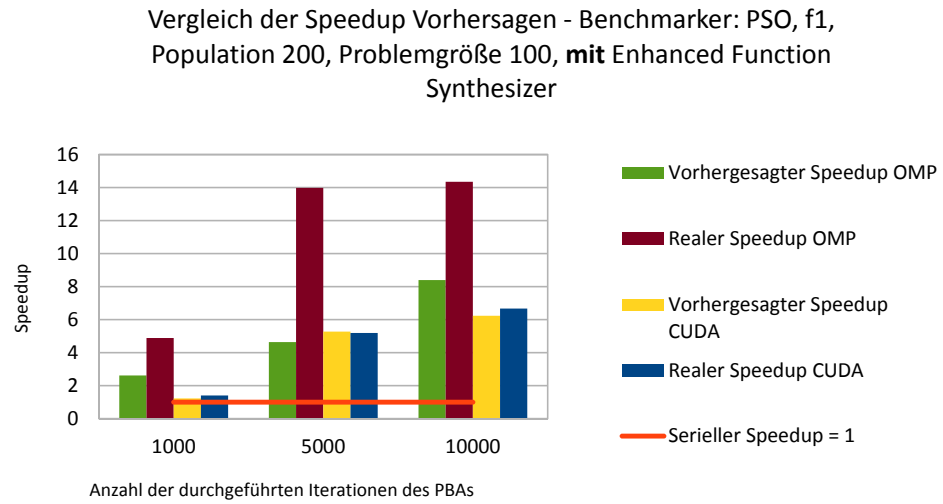


Abbildung B.2: Populationsgröße 100 und Problemgröße 30





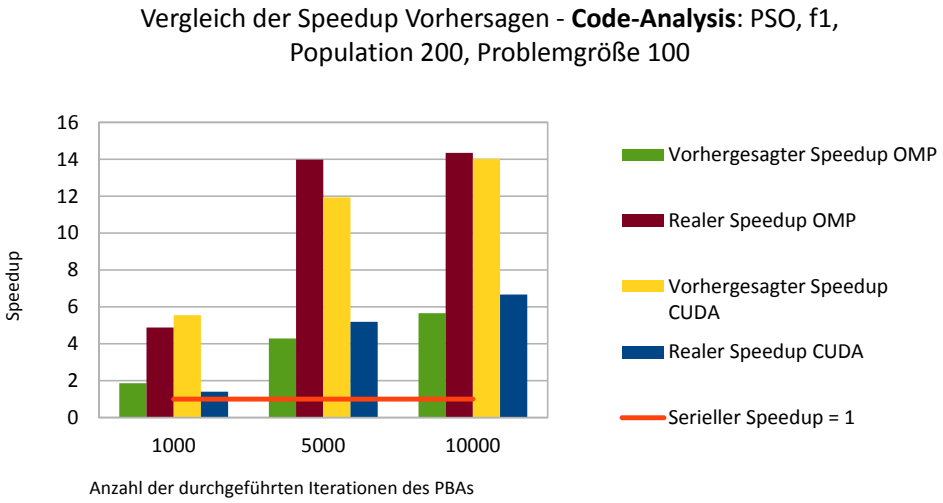
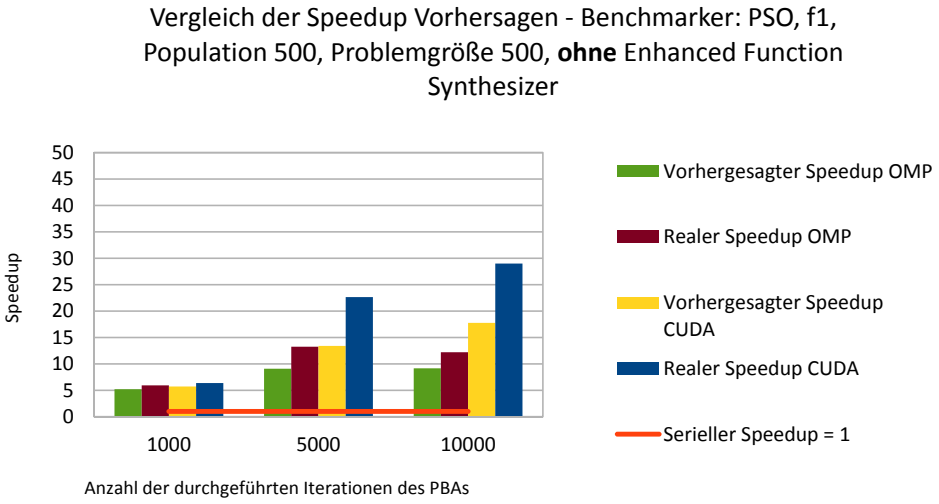
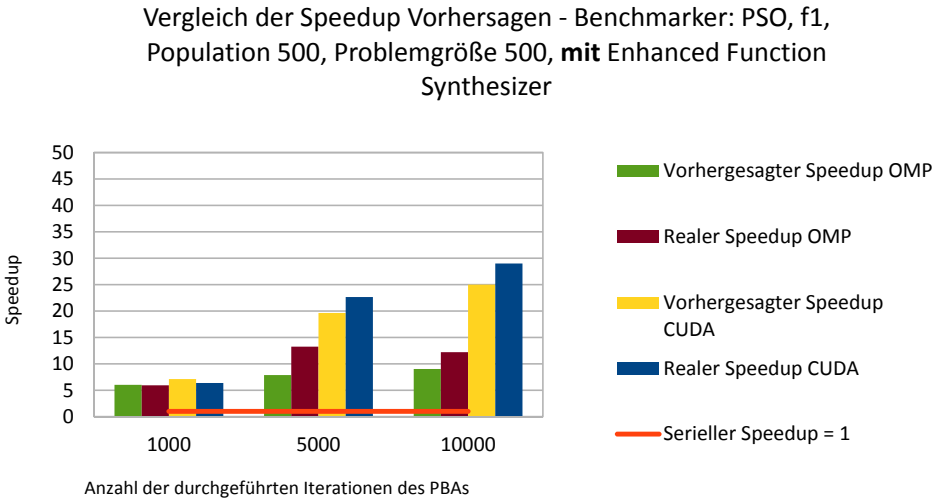


Abbildung B.3: Populationsgröße 200 und Problemgröße 100





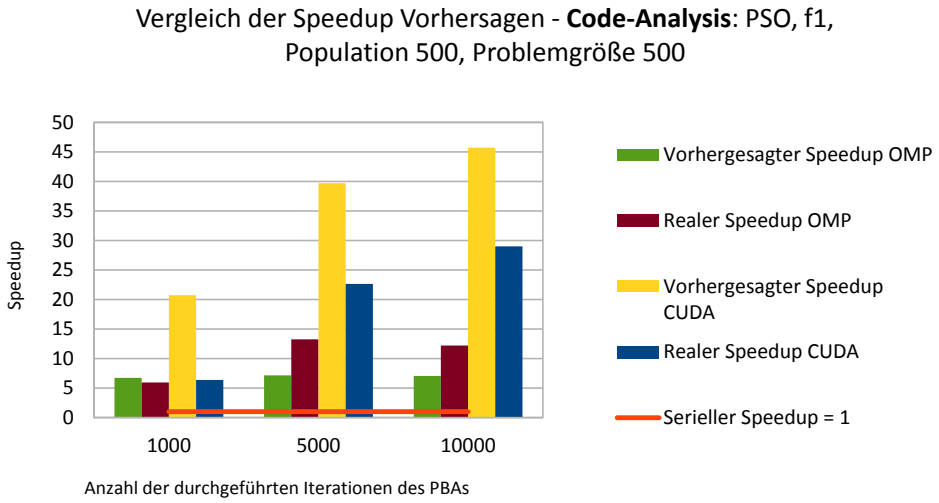


Abbildung B.4: Populationsgröße 500 und Problemgröße 500

B.2 Ergebnisse der Funktion f5

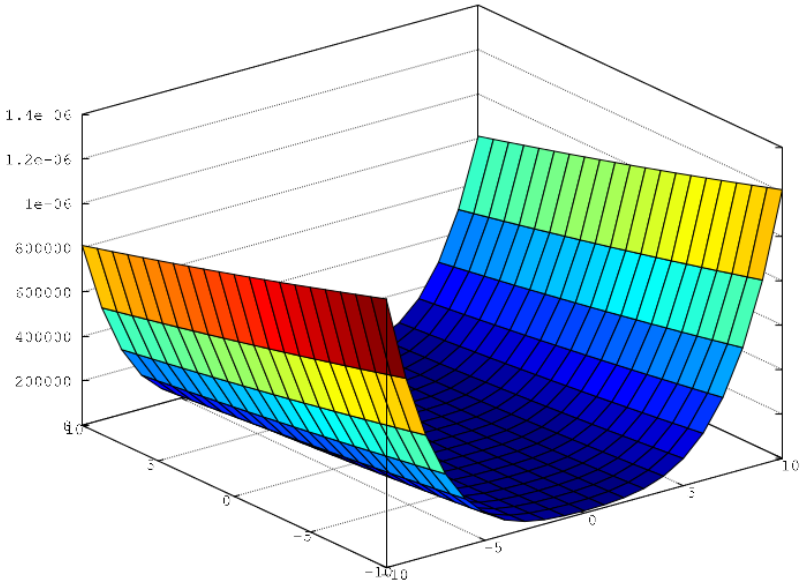
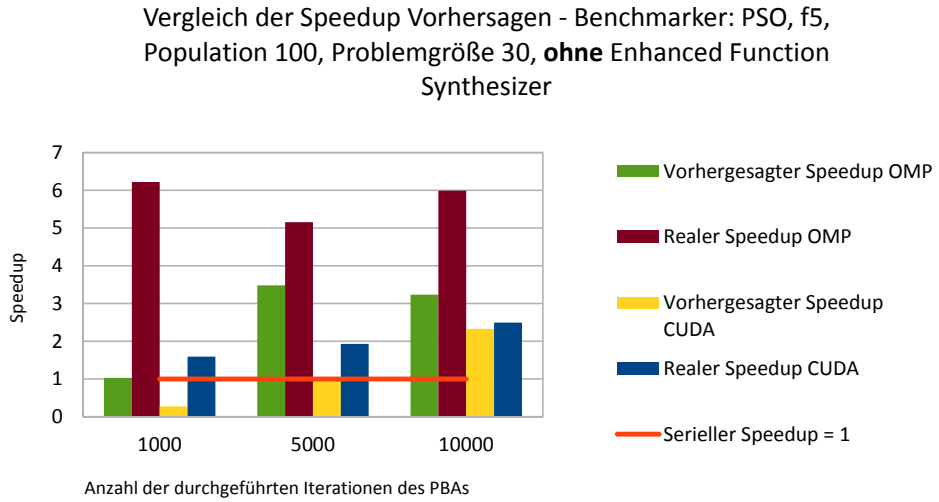
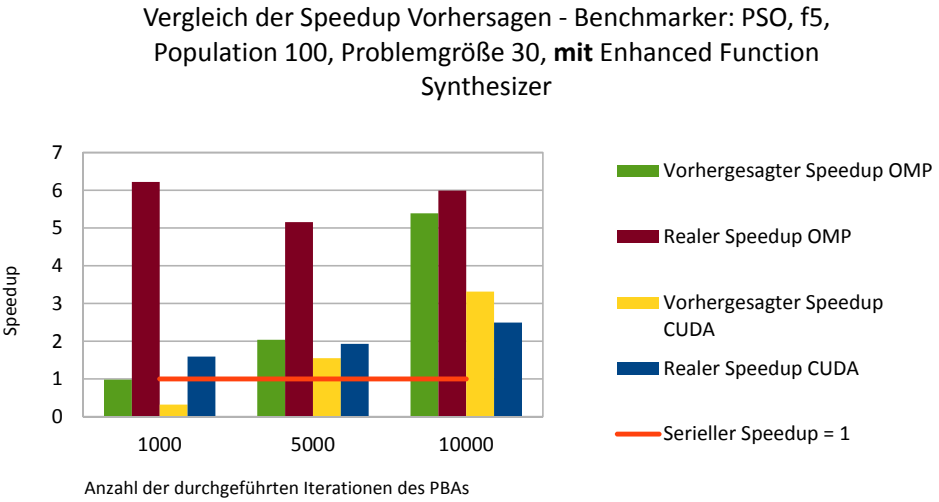


Abbildung B.5: Funktion f5





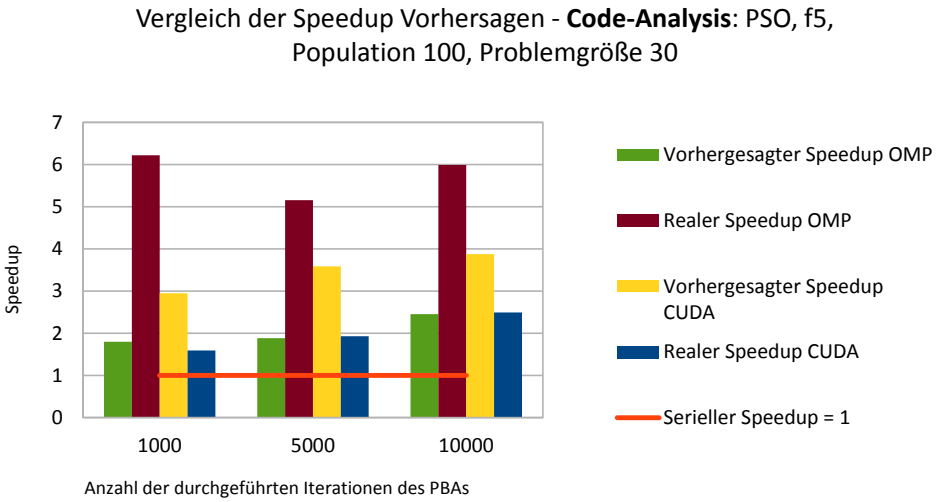
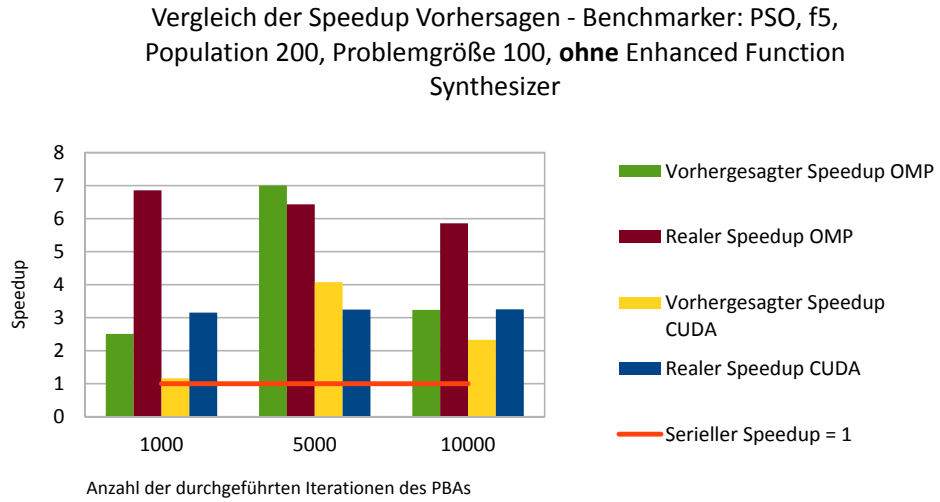
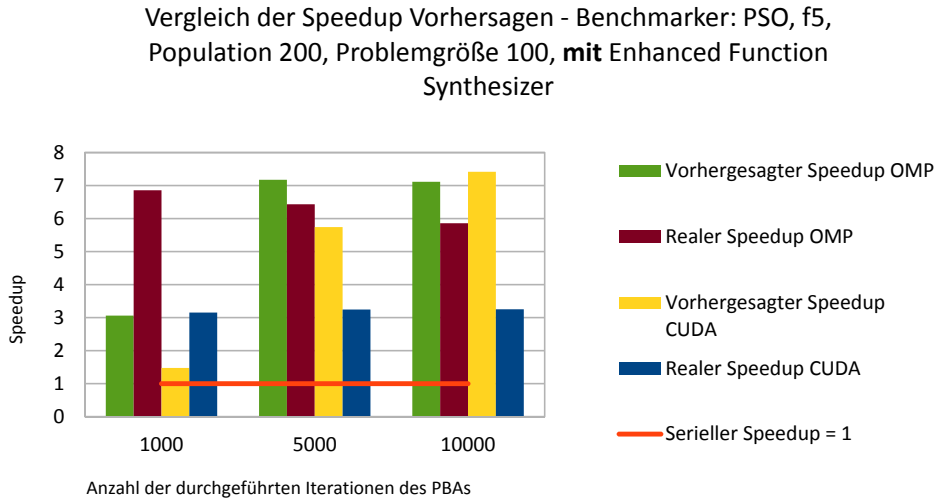


Abbildung B.6: Populationsgröße 100 und Problemgröße 30





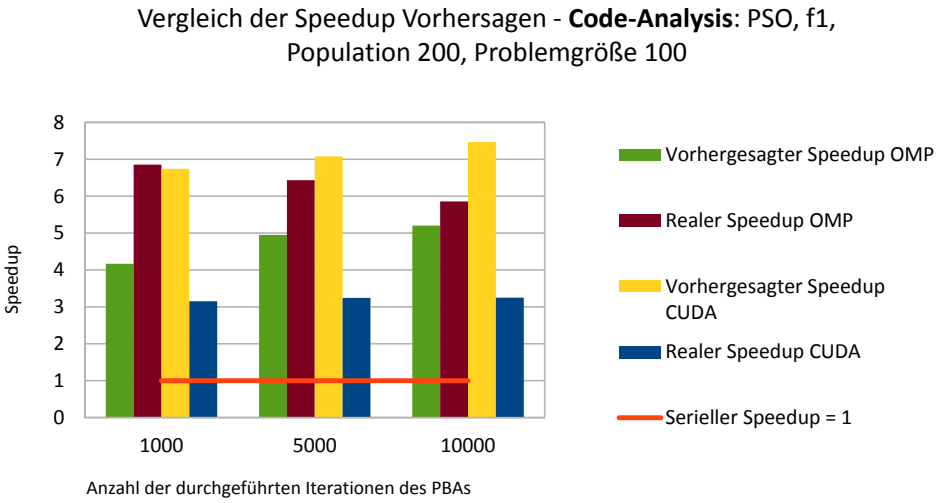
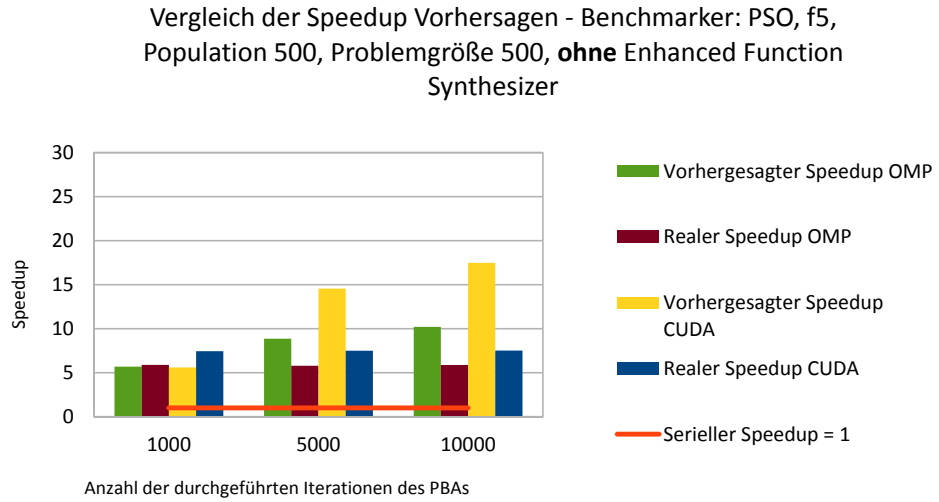
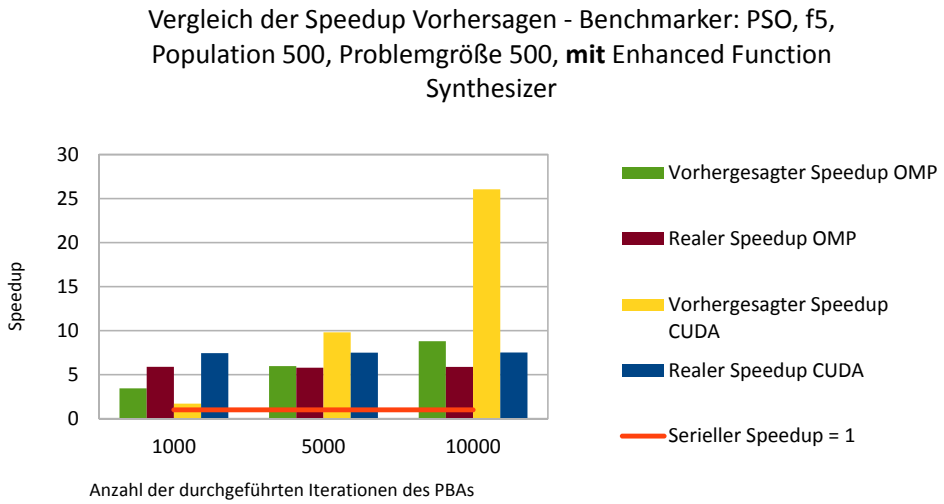


Abbildung B.7: Populationsgröße 200 und Problemgröße 100





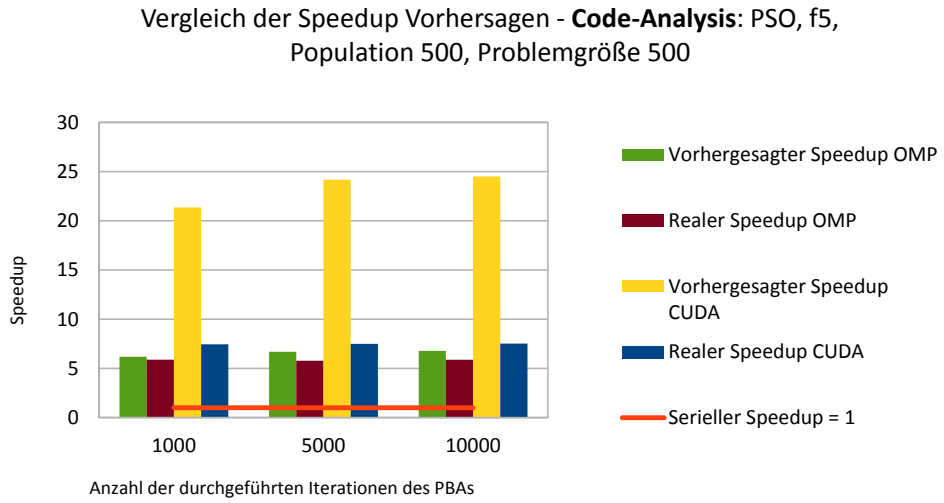


Abbildung B.8: Populationsgröße 500 und Problemgröße 500

B.3 Ergebnisse der Funktion f8

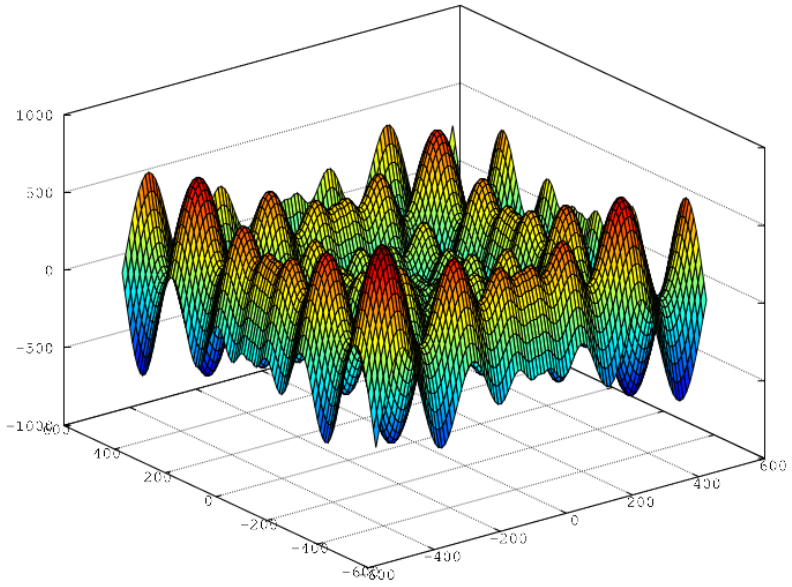
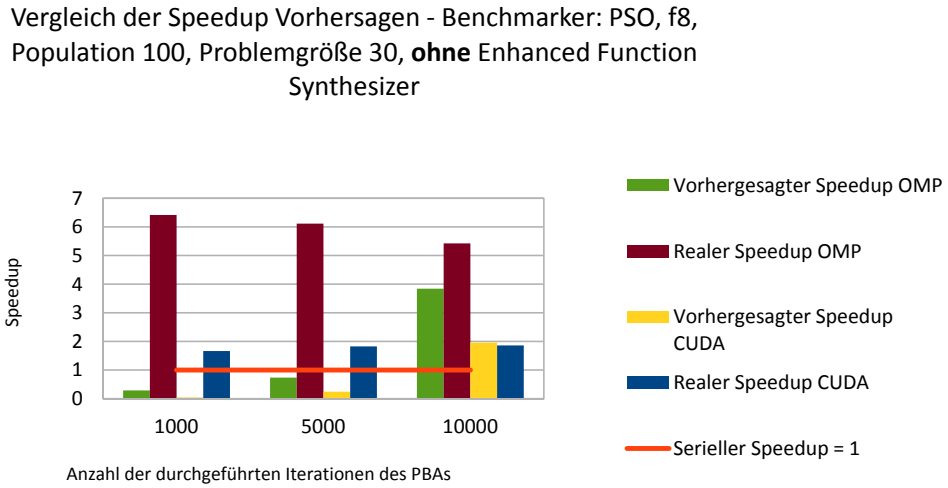
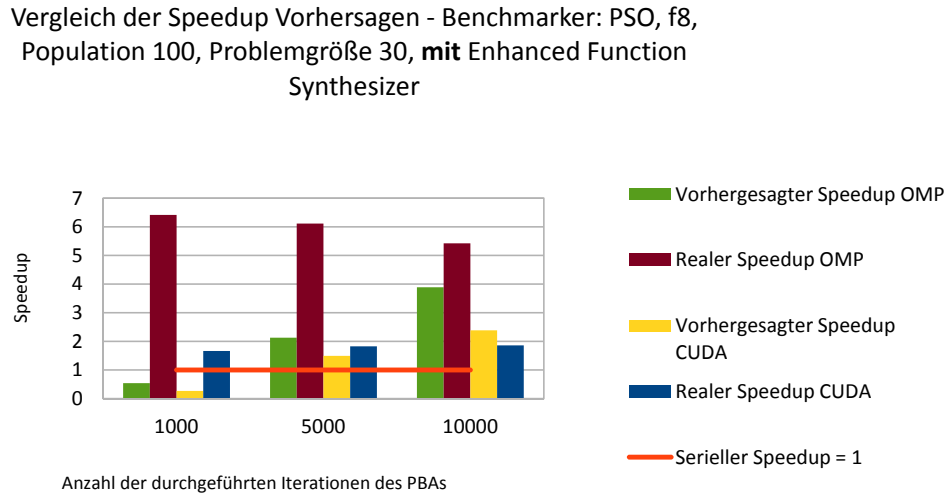


Abbildung B.9: Funktion f8





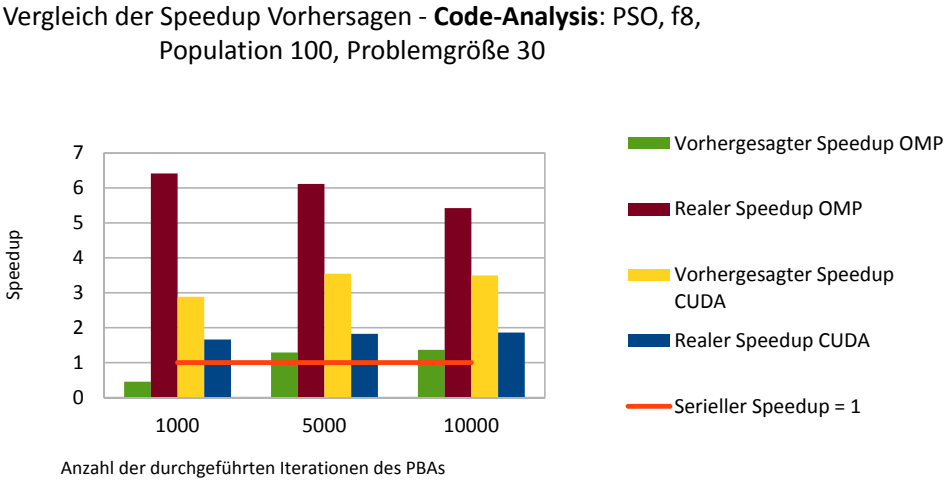
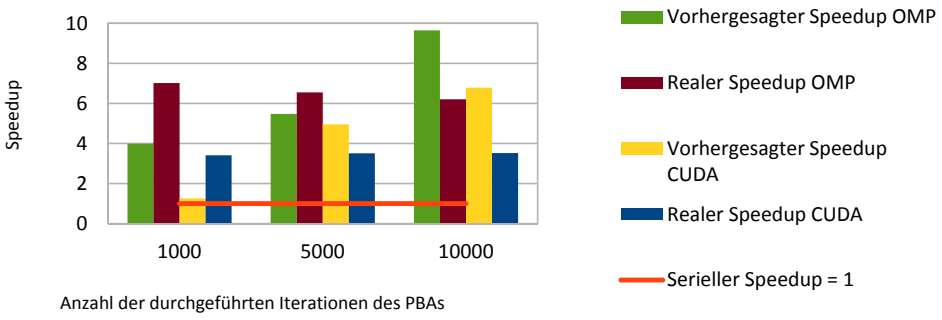
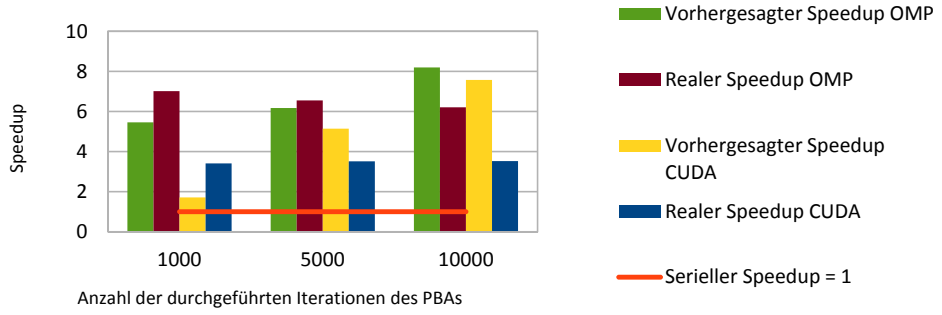


Abbildung B.10: Populationsgröße 100 und Problemgröße 30

Vergleich der Speedup Vorhersagen - Benchmarker: PSO, f8, Population 200, Problemgröße 100, **ohne** Enhanced Function Synthesizer



Vergleich der Speedup Vorhersagen - Benchmarker: PSO, f8, Population 200, Problemgröße 100, **mit** Enhanced Function Synthesizer



Vergleich der Speedup Vorhersagen - **Code-Analysis**: PSO, f1, Population 200, Problemgröße 100

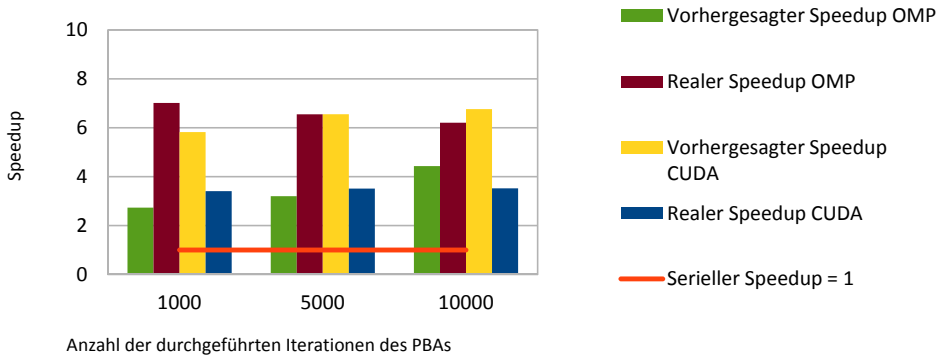
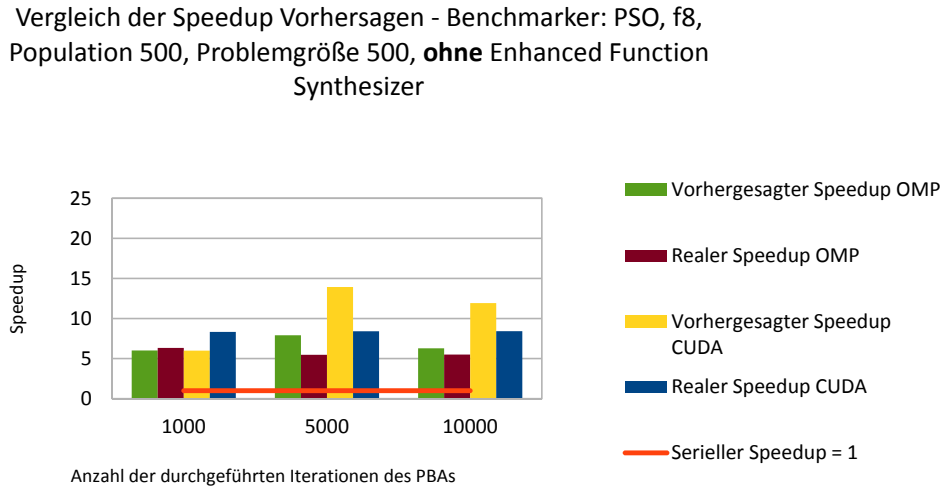
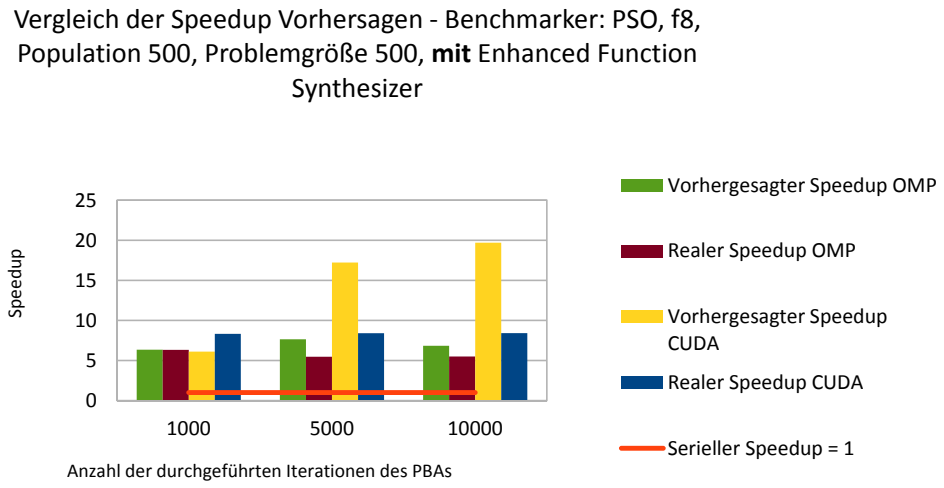


Abbildung B.11: Populationsgröße 200 und Problemgröße 100





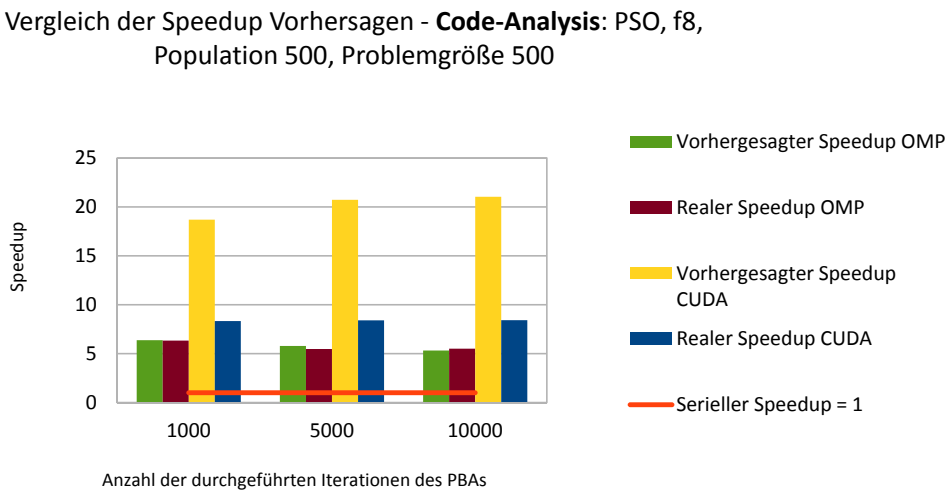


Abbildung B.12: Populationsgröße 500 und Problemgröße 500

B.4 Ergebnisse der Funktion f11

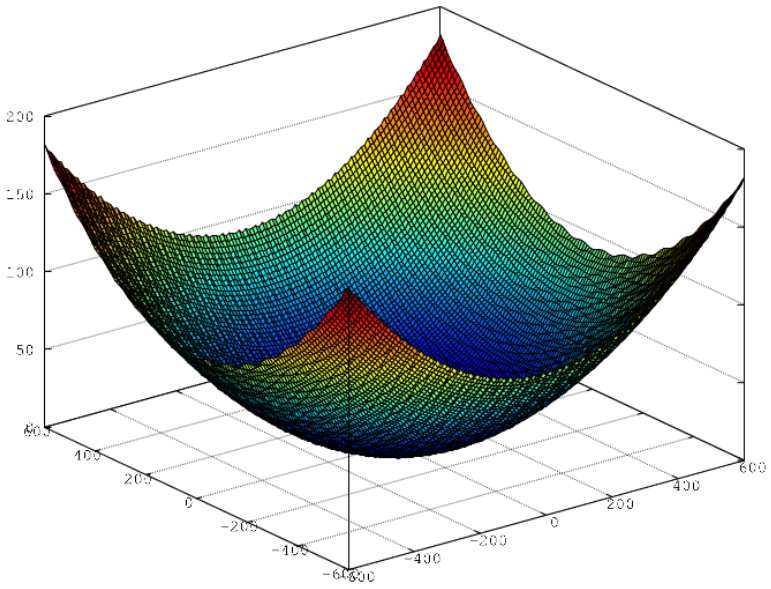
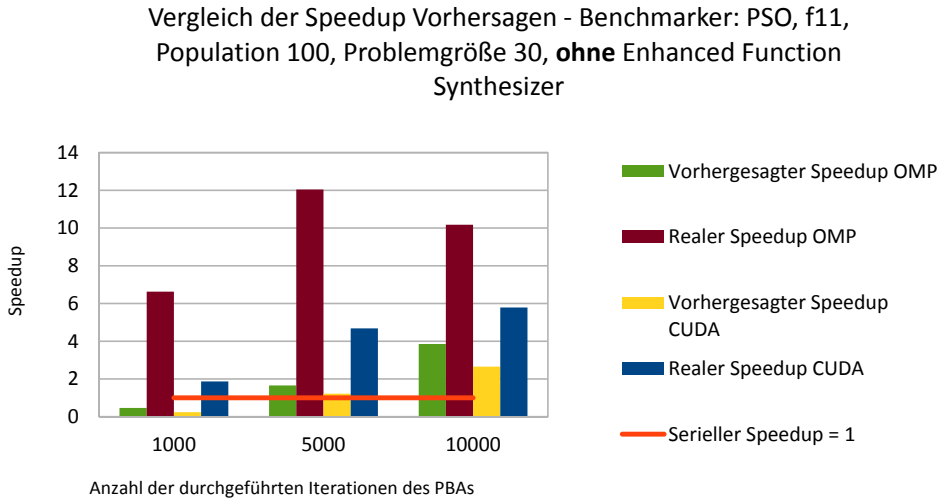
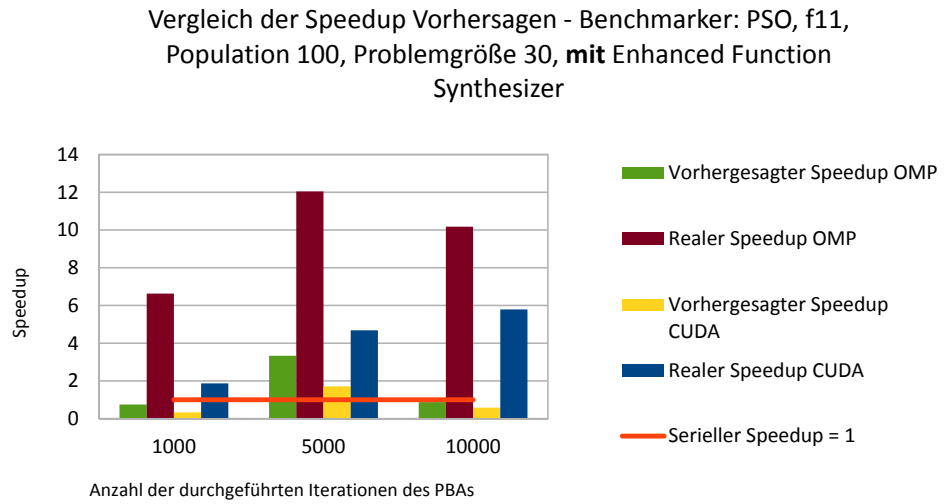


Abbildung B.13: Funktion f11





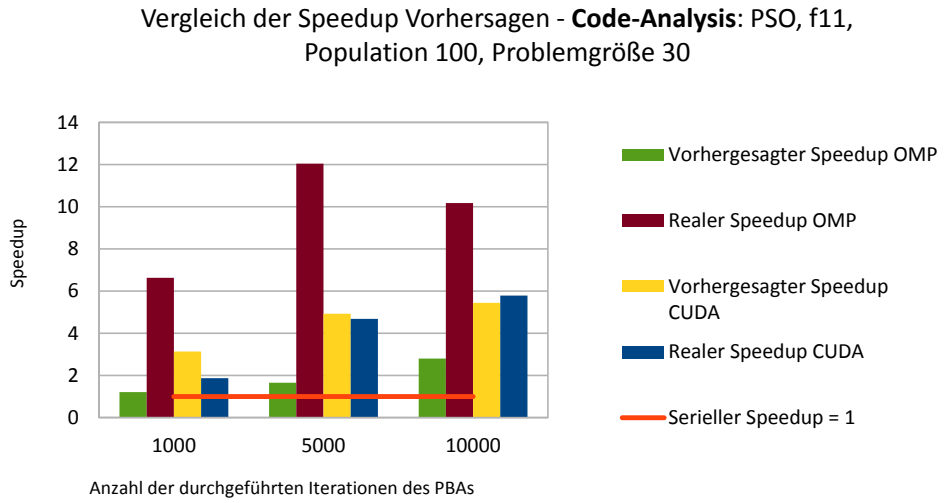
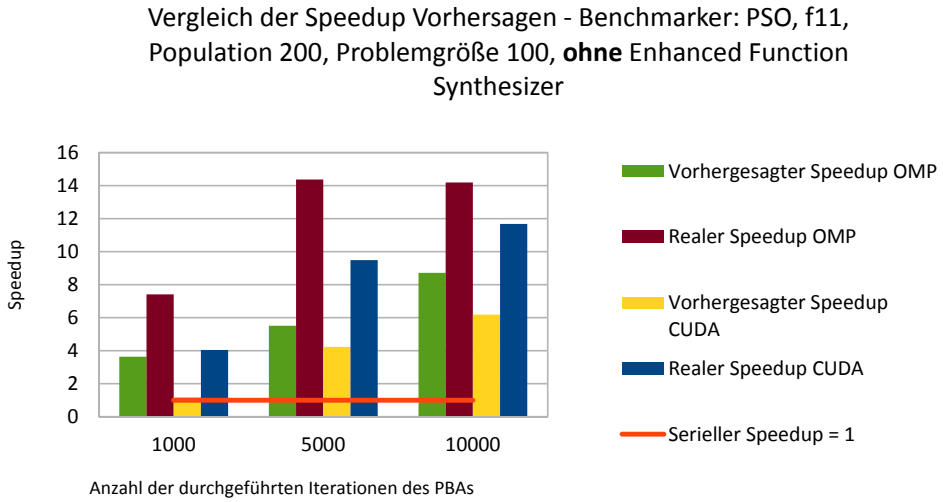
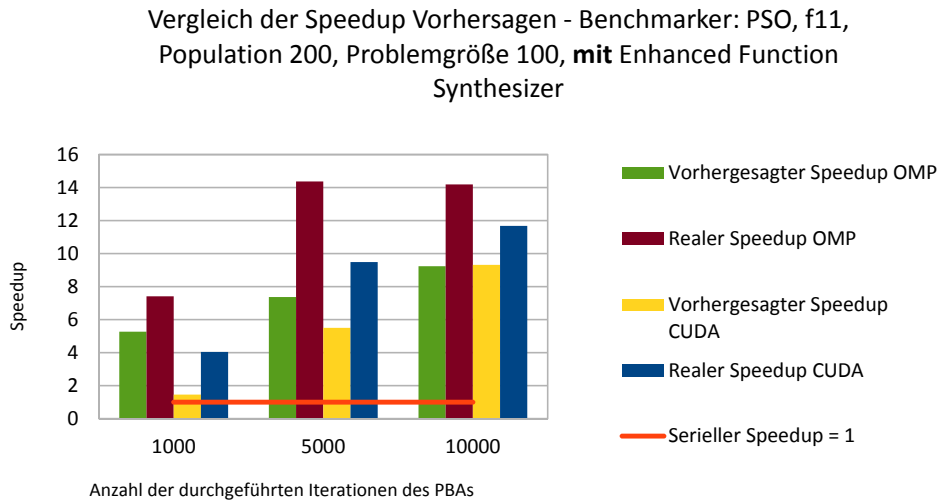


Abbildung B.14: Populationsgröße 100 und Problemgröße 30





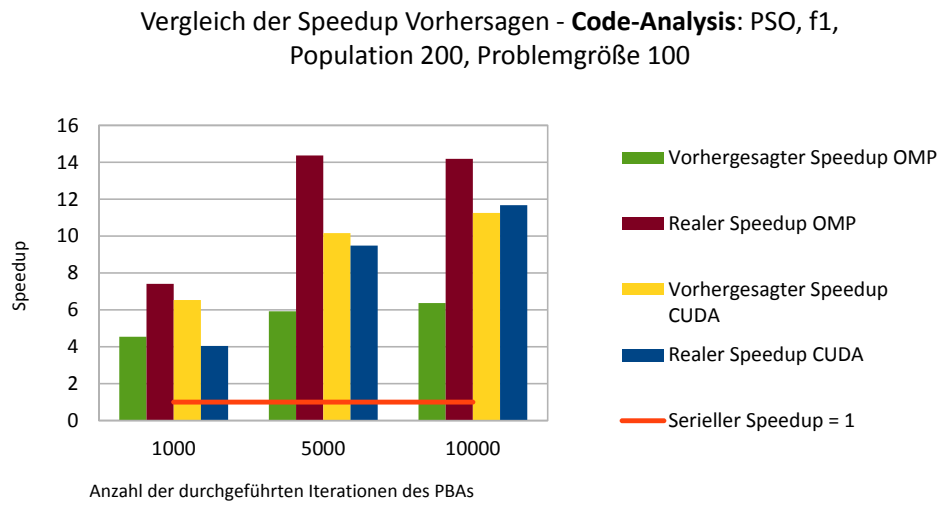
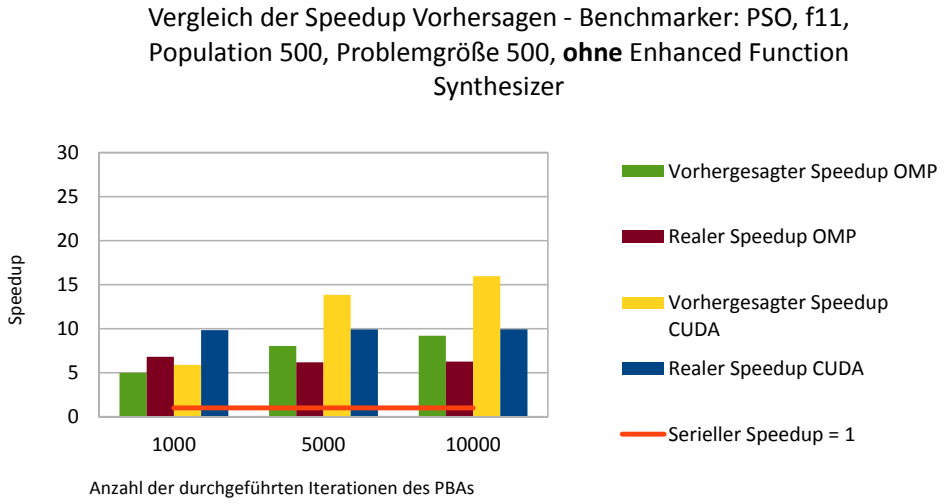
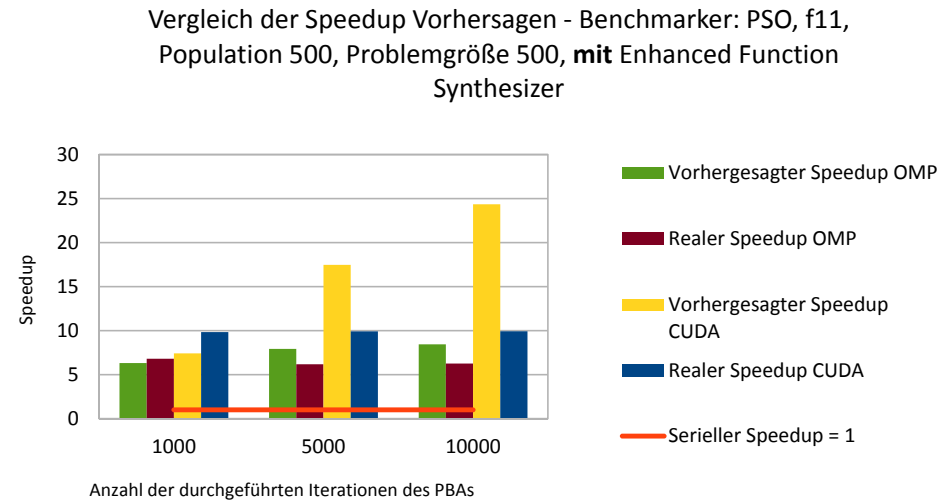


Abbildung B.15: Populationsgröße 200 und Problemgröße 100





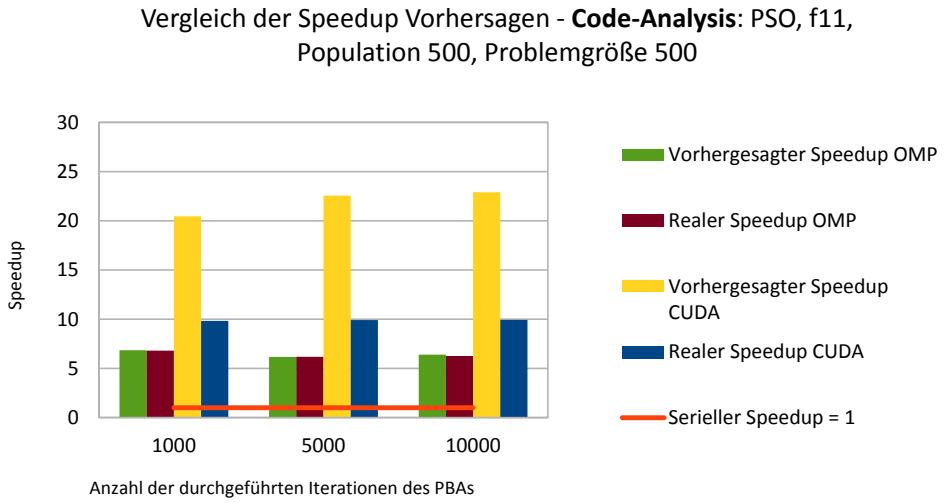


Abbildung B.16: Populationsgröße 500 und Problemgröße 500

C | Templates für das Code-Analysis Modul

```
1 #include "benchmarkLib.h"
2 #include <cmath>
3 #include <cstdlib>
4
5 int finish() {
6   return 0;
7 }
8
9 // arguments: temp file, iterations, dimensions
10 int main(int argc, char* argv[]) {
11   if(argc < 2) {
12     return -1;
13   }
14   StopWatch watch(argv[1]);
15
16   return finish();
17 }
```

Listing C.1: OpenMP Template für die Code-Analyse

```
1 #include "benchmarkLib.h"
2 #include <stdlib.h>
3 #include <time.h>
4 #include <utility>
5 #include <iostream>
6 #include <vector>
7
```

```
8  #define CREATE_RND_ARRAY(type, prefix, count, rnd)  type prefix ##
       array[count]; \
9    for(unsigned long i = 0; i < count; ++i) { \
10     prefix ## array[i] = rnd(); \
11   } \
12 arrays.push_back(HostDeviceArray((void*)prefix ## array, (void**)&
     prefix ## device, sizeof(prefix ## array)))
13
14 using namespace std;
15
16 class HostDeviceArray {
17 public:
18   HostDeviceArray(void* host, void** device, size_t size) : host(
       host), device(device), size(size) {}
19
20   void allocAndCopy() {
21     cudaMalloc(device, size);
22     cudaMemcpy(*device, host, size, cudaMemcpyHostToDevice);
23   }
24
25   void copyAndFree() {
26     cudaMemcpy(host, *device, size, cudaMemcpyDeviceToHost);
27     cudaFree(*device);
28   }
29
30 private:
31   void* host;
32   void** device;
33   size_t size;
34 };
35
36 int irnd() {
37   return rand() % 255 + 1;
38 }
39
40 float frnd() {
41   return 0.9f + rand() * 0.2f / RAND_MAX;
42 }
43
```

```
44 __global__ void benchmark(int* i32array, long long int* i64array,
     float* f32array, double* f64array, unsigned long iterations) {
45   unsigned long long blockID = blockIdx.x + blockIdx.y * gridDim.x
       ;
46   unsigned long long i = blockID * blockDim.x + threadIdx.x;
47   if (i < NUM_ITERATIONS) {
48 #if I32_USES > 0
49     int* i32ptr;
50     int i32accu = 1;
51 #endif
52 #if I64_USES > 0
53     long long int* i64ptr;
54     long long int i64accu = 1;
55 #endif
56 #if F32_USES > 0
57     float* f32ptr;
58     float f32accu = 1;
59 #endif
60 #if F64_USES > 0
61     double* f64ptr;
62     double f64accu = 1;
63 #endif
64     // @SPEEDIE: benchmark insert
65   };
66 }
67
68 // arguments: iterations, population size, dimensions
69 int main(int argc, char* argv[]) {
70   if(argc < 2) {
71     return -1;
72   }
73   srand(time(NULL));
74   StopWatch watch(argv[1]);
75
76   int* i32device;
77   long long * i64device;
78   float* f32device;
79   double* f64device;
80
```

```
81    vector<HostDeviceArray> arrays;
82  #if I32_USES > 0
83    CREATE_RND_ARRAY(int, i32, I32_USES, irnd);
84  #endif
85  #if I64_USES > 0
86    CREATE_RND_ARRAY(long long int, i64, I64_USES, irnd);
87  #endif
88  #if F32_USES > 0
89    CREATE_RND_ARRAY(float, f32, F32_USES, frnd);
90  #endif
91  #if F64_USES > 0
92    CREATE_RND_ARRAY(double, f64, F64_USES, frnd);
93  #endif
94
95    dim3 blockSize(64);
96    dim3 gridSize;
97    dim3 maxGrid(1024, 1024, 64);
98
99    unsigned long long blocksNeeded = NUM_ITERATIONS / blockSize.x
100       + (NUM_ITERATIONS % blockSize.x == 0 ? 0 : 1);
101   gridSize.x = min<unsigned int>(maxGrid.x, blocksNeeded);
102   gridSize.y = max(1,
103       min<unsigned int>(maxGrid.y,
104           blocksNeeded / maxGrid.x
105               + (blocksNeeded % maxGrid.x == 0 ? 0 : 1)));
106
107   {
108     void* dummy;
109     cudaMalloc((void **) &dummy, 0);
110     cudaFree(dummy);
111   }
112
113   watch.start();
114   for(unsigned i = 0; i < arrays.size(); ++i) {
115     arrays[i].allocAndCopy();
116   }
117   watch.stop(); // copy to device time
118
119   watch.start();
```

```
120   for(int i = 0; i < NUM_EXECS; ++i) {
121     benchmark<<<gridSize, blockSize>>>(i32device, i64device,
          f32device, f64device, NUM_ITERATIONS);
122   }
123   watch.stop(); // kernel time
124
125   watch.start();
126   for(unsigned i = 0; i < arrays.size(); ++i) {
127     arrays[i].copyAndFree();
128   }
129   watch.stop(); // copy to host time
130
131   return 0;
132 }
```

Listing C.2: CUDA Template für die Code-Analyse