TECHNISCHE UNIVERSITÄT
CHEMNITZ

Energie- und Ausführungszeitmodelle zur effizienten Ausführung wissenschaftlicher Simulationen

Dissertation

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften

der

Fakultät für Informatik

Technische Universität Chemnitz

Vorgelegt von

Jens Lang

Tag der Einreichung: 6. Juni 2014
Tag der Verteidigung: 9. Dezember 2014

Gutachter: Prof. Dr. Gudula Rünger
Prof. Dr. Fred Hamker

Referat

Das wissenschaftliche Rechnen mit der Computersimulation hat sich heute als dritte Säule der wissenschaftlichen Methodenlehre neben der Theorie und dem Experiment etabliert. Aufgabe der Informatik im wissenschaftlichen Rechnen ist es, sowohl effiziente Simulationsalgorithmen zu entwickeln als auch ihre effiziente Implementierung.

Die vorliegende Arbeit richtet ihren Fokus auf die effiziente Implementierung zweier wichtiger Verfahren des wissenschaftlichen Rechnens: die Schnelle Multipolmethode (FMM) für Teilchensimulationen und die Methode der finiten Elemente (FEM), die z. B. zur Berechnung der Deformation von Festkörpern genutzt wird. Die Effizienz der Implementierung bezieht sich hier auf die Ausführungszeit der Simulationen und den zur Ausführung notwendigen Energieverbrauch der eingesetzten Rechnersysteme.

Die Steigerung der Effizienz wurde durch modellbasiertes Autotuning erreicht. Beim modellbasierten Autotuning wird für die wesentlichen Teile des Algorithmus ein Modell aufgestellt, das dessen Ausführungszeit bzw. Energieverbrauch beschreibt. Dieses Modell ist abhängig von Eigenschaften des genutzten Rechnersystems, von Eingabedaten und von verschiedenen Parametern des Algorithmus. Die Eigenschaften des Rechnersystems werden durch Ausführung des tatsächlich genutzten Codes für verschiedene Implementierungsvarianten ermittelt. Diese umfassen eine CPU-Implementierung und eine Grafikprozessoren-Implementierung für die FEM und die Implementierung der Nahfeld- und der Fernfeldwechselwirkungsberechnung für die FMM. Anhand der aufgestellten Modelle werden die Kosten der Ausführung für jede Variante vorhergesagt. Die optimalen Algorithmenparameter können somit analytisch bestimmt werden, um die gewünschte Zielgröße, also Ausführungszeit oder Energieverbrauch, zu minimieren. Bei der Ausführung der Simulation werden die effizientesten Implementierungsvarianten entsprechend der Vorhersage genutzt. Während bei der FMM die Performance-Messungen unabhängig von der Ausführung der Simulation durchgeführt werden, wird für die FEM ein Verfahren zur dynamischen Verteilung der Rechenlast zwischen CPU und GPU vorgestellt, das auf Ausführungszeitmessungen zur Laufzeit der Simulation reagiert. Durch Messung der tatsächlichen Ausführungszeiten kann so dynamisch auf sich während der Laufzeit verändernde Verhältnisse reagiert und die Verteilung der Rechenlast entsprechend angepasst werden.

Die Ergebnisse dieser Arbeit zeigen, dass modellbasiertes Autotuning es ermöglicht, die Effizienz von Anwendungen des wissenschaftlichen Rechnens in Bezug auf Ausführungszeit und Energieverbrauch zu steigern. Insbesondere die Berücksichtigung des Energieverbrauchs alternativer Ausführungspfade, also die Energieadaptivität, wird in naher Zukunft von großer Bedeutung im wissenschaftlichen Rechnen sein.

Abstract

Computer simulation as a part of the scientific computing has established as third pillar in scientific methodology, besides theory and experiment. The task of computer science in the field of scientific computing is the development of efficient simulation algorithms as well as their efficient implementation.

The thesis focuses on the efficient implementation of two important methods in scientific computing: the Fast Multipole Method (FMM) for particle simulations, and the Finite Element Method (FEM), which is, e.g., used for deformation problems of solids. The efficiency of the implementation considers the execution time of the simulations and the energy consumption of the computing systems needed for the execution.

The method used for increasing the efficiency is model-based autotuning. For model-based autotuning, a model for the substantial parts of the algorithm is set up which estimates the execution time or energy consumption. This model depends on properties of the computer used, of the input data and of parameters of the algorithm. The properties of the computer are determined by executing the real code for different implementation variants. These implementation variantss comprise a CPU and a graphics processor implementation for the FEM, and implementations of near field and far field interaction calculations for the FMM. Using the models, the execution costs for each variant are predicted. Thus, the optimal algorithm parameters can be determined analytically for a minimisation of the desired target value, i.e. execution time or energy consumption. When the simulation is executed, the most efficient implementation variants are used depending on the prediction of the model. While for the FMM the performance measurement takes place independently from the execution of the simulation, for the FEM a method for dynamically distributing the workload to the CPU and the GPU is presented, which takes into account execution times measured at runtime. By measuring the real execution times, it is possible to response to changing conditions and to adapt the distribution of the workload accordingly.

The results of the thesis show that model-based autotuning makes it possible to increase the efficiency of applications in scientific computing regarding execution time and energy consumption. Especially, the consideration of the energy consumption of alternative execution paths, i.e. the energy adaptivity, will be of great importance in scientific computing in the near future.

Zugehörige Veröffentlichungen

In diese Arbeit flossen Erkenntnisse aus den folgenden Veröffentlichungen ein:

- [59] Dachsel, Holger, Michael Hofmann, Jens Lang und Gudula Rünger: *Automatic Tuning of the Fast Multipole Method Based on Integrated Performance Prediction.* In: *14th IEEE International Conference on High Performance Computing and Communications (HPCC-2012)*, 2012: **Abschnitt 3**

- [18] Balg, Martina, Jens Lang, Arnd Meyer und Gudula Rünger: *Array-based reduction operations for a parallel adaptive FEM.* In: Keller, Rainer, David Kramer und Jan-Philipp Weiß (Herausgeber): *Facing the Multicore Challenge III*, Band 7686 der Reihe *Lecture Notes in Computer Science*, S. 25–36. Springer, Berlin, Heidelberg, 2013. isbn: 978-3-642-35892-0. doi: 10.1007/978-3-642-35893-7_3: **Abschnitt 4.2**

- [162] Lang, Jens und Gudula Rünger: *Dynamic Distribution of Workload Between CPU and GPU for a Parallel Conjugate Gradient Method in an Adaptive FEM.* Procedia Computer Science, Bd. 18, S. 299–308, 2013. doi: 10.1016/j.procs.2013.05.193. Proceedings of the International Conference on Computational Science, ICCS 2013: **Abschnitte 4.4 und 4.5**

- [165] Lang, Jens und Gudula Rünger: *Measuring and modelling energy consumption for a CPU/GPU conjugate gradient method in an adaptive FEM.* In: *Proceedings of the High-Level Programming for Heterogeneous and Hierarchical Parallel Systems workshop at HiPEAC conference 2014*, Wien (Österreich), 2014: **Abschnitt 4.4.3**

- [164] Lang, Jens und Gudula Rünger: *An execution time and energy model for an energy-aware execution of a conjugate gradient method with CPU/GPU collaboration.* Journal of Parallel and Distributed Computing, 2014. doi: 10.1016/j.jpdc.2014.06.001. (zum Druck angenommen): **Abschnitt 4.4**

- [163] Lang, Jens und Gudula Rünger: *High-Resolution Power Profiling of GPU Functions Using Low-Resolution Measurement.* In: Wolf, Felix, Bernd Mohr und Dieter an Mey (Herausgeber): *Euro-Par 2013 Parallel Processing*, Band 8097 der Reihe *Lecture Notes in Computer Science*, S. 801–812, Berlin, Heidelberg, 2013. Springer. isbn: 978-3-642-40046-9. doi: 10.1007/978-3-642-40047-6_80: **Abschnitt 5**

Inhaltsverzeichnis

1 Einleitung

Die Computersimulation hat sich in der wissenschaftlichen Methodenlehre zur dritten Säule neben Experiment und Theorie entwickelt [216, 232, 92, 29]. Computersimulationen bilden das Verhalten eines Systems anhand einer mathematischen Beschreibung rechnergestützt nach [32, s. v. *Computersimulation*]. Sie dienen der Untersuchung der Dynamik von Systemen, der Entwicklung von Hypothesen, Modellen und Theorien, als Ersatz für Experimente, zur Unterstützung von Experimentatoren und als pädagogisches Werkzeug [107]. Durch Simulationen können Sachverhalte erschlossen werden, die Experimenten nicht zugänglich sind – sei es, weil die Experimente zu teuer, das Labor zu klein oder der Untersuchungsgegenstand noch nicht vorhanden ist. Beispiele für Simulationen sind die Entstehung von Galaxien in den Naturwissenschaften, die Entwicklung einer Volkswirtschaft in den Sozialwissenschaften [86] oder das Crash-Verhalten eines neu entwickelten Automobils in den Ingenieurwissenschaften. Obwohl eine Simulation immer nur ein „Als Ob" [73], also nur ein Abbild der Realität ist, versucht sie sich bezüglich eines Sachverhalts der Wirklichkeit bestmöglich anzunähern, um so zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind [261]. Dadurch, dass sie Theorien „anschaulich" macht, trägt die Simulation dazu bei, diese weiterzuentwickeln [232] und so unser Verständnis von der Welt zu erweitern.

Die computergestützte Simulation komplexer Prozesse ist Gegenstand des *wissenschaftlichen Rechnens* (engl. *scientific computing*) [32, s. v. *wissenschaftliches Rechnen*]. Nach Golub und Ortega ist das wissenschaftliche Rechnen „das Sortiment von Werkzeugen, Techniken und Theorien, die zur computergestützten Lösung mathematischer Modelle von Problemen der Natur- und Ingenieurwissenschaften benötigt werden"[1] [90, S. 2]. Die Vorgehensweise im wissenschaftlichen Rechnen ist in der Regel die Folgende [32, s. v. *wissenschaftliches Rechnen*]:

1. Modellbildung durch den Anwender (Physiker, Mediziner, Biologe, ...),
2. Auswahl und Entwicklung geeigneter Rechen- und Näherungsverfahren durch Mathematiker,
3. Entwurf und Implementierung effizienter Algorithmen durch Informatiker.

Im 3. Schritt, der effizienten Implementierung von Simulationsanwendungen im wissenschaftlichen Rechnen, ist diese Arbeit angesiedelt.

Motivation

Die Wissenschaft verfeinert und erweitert ihre Modelle und Simulationsmethoden stetig, um immer genauere und realitätsnähere Ergebnisse zu erhalten. Bei einigen Methoden werden beispielsweise

1 "[...] scientific computing is the collection of tools, techniques, and theories required to solve on a computer mathematical models of problems in science and engineering." [90, S. 2]

empirische, vereinfachte Modelle durch Ab-initio-Methoden ersetzt, also solche, die mit den physikalischen Grundgesetzen rechnen [72, s. v. *supercomputer*]. Beispielsweise wurde auch die Wetter- und Klimasimulation, mit deren Ergebnissen man auch im Alltag häufig konfrontiert wird, über die Jahre immer komplexer: So wurde u. a. das Gitter der diskreten Simulationspunkte, die über die Erde gelegt werden, stetig verfeinert. Dadurch konnte der Prognosefehler für kurzfristige Vorhersagen der Tageshöchsttemperatur, also für 1–2 Tage, im Zeitraum von 1984 bis 2009 von 2,5 K auf 1,6 K im Jahresmittel gesenkt werden. Die Vorhersage für den Bodendruck ist heute für 7 Tage im Voraus besser als die Vorhersage für 2 Tage im Voraus im Jahr 1968, als der Deutsche Wetterdienst mit der Wettersimulation begann [175]. Sowohl der steigende Rechenaufwand als auch die größer werdenden Datenmengen stellen wachsende Anforderungen an die Leistungsfähigkeit der eingesetzten Rechner.

Die heute leistungsfähigsten Rechner sind ausnahmslos Parallelrechner. Sie werden für nahezu alle Simulationen im wissenschaftlichen Rechnen eingesetzt. Eines der drängendsten Probleme beim Einsatz von Parallelrechnern ist heute und in naher Zukunft deren hoher Energieverbrauch [41]. Schreibt man die Steigerung der Rechenleistung der leistungsfähigsten Supercomputer und die notwendige elektrische Leistung der Rechner bis ins Jahr 2020 fort, kann für jenes Jahr ein Parallelrechner erwartet werden, der mit heutiger Technik eine elektrische Leistung von ca. 100 MW benötigen würde. Das ist im Wissenschaftsbereich nicht finanzierbar. Stattdessen wird durch den Einsatz von Energiespartechniken für einen solchen Rechner eine Leistung von ca. 20 MW angestrebt [66]. Auch außerhalb des Hochleistungsrechnens ist es wichtig, dass Computer energieeffizient arbeiten, um so den weltweiten Energieverbrauch des IT-Sektors trotz steigender Computernutzung nicht weiter ansteigen zu lassen, was zum Erreichen der Klimaziele [262] essenziell ist. Energieeinsparung kann nur durch Verbesserungen sowohl auf der Hardware- als auch auf Softwareebene erreicht werden [12].

Neben dem Verlangen nach leistungsfähigeren Rechnern ist eine Konsequenz des Komplexerwerdens wissenschaftlicher Simulationen, dass auch die Implementierung der zugrunde liegenden Algorithmen – gerade auf Parallelrechnern – immer komplexer wird. Es sind Spezialisten notwendig, um eine Simulation auf einem vorhandenen Rechner so effizient wie möglich zu implementieren. An dieser Stelle ergibt sich ein weiteres Problem: Während Software im wissenschaftlichen Rechnen häufig über einen Zeitraum von etwa 20 Jahren im Einsatz ist, hat ein Supercomputer eine durchschnittliche Nutzungszeit von 4 Jahren [174]. Ist die langwierige und teure Anpassung der Software auf einen Rechner durch den Spezialisten abgeschlossen, ist möglicherweise ein guter Teil der Lebenszeit dieses Rechners schon abgelaufen. Nach Sutter und Larus [252] ist „das schwierige Problem […] nicht, Multicore-Hardware zu bauen, sondern sie in einer Art und Weise zu programmieren, die es alltäglichen Anwendungen ermöglicht, aus dem fortschreitenden exponentiellen Wachstum der CPU-Rechenleistung Nutzen zu ziehen.“[2] [252]. Noch schwieriger werden die Verhältnisse, wenn es sich bei dem Parallelrechner um ein heterogenes System, also beispielsweise eines bestehend aus CPUs und Grafikprozessoren, handelt [154].

Eine Möglichkeit, die Anpassung von Software an einen Rechner zu vereinfachen und zu beschleunigen, ist, das Verfahren zu automatisieren. Beim automatisierten Tuning (Autotuning) wird

2 “The difficult problem is not building multicore hardware, but programming it in a way that lets mainstream application benefit from the continued exponential growth in CPU performance.” [252]

ohne Zutun des Anwenders experimentell ermittelt, welche Implementierung eines Algorithmus bzw. welcher Algorithmus auf einem gegebenen Rechner im Hinblick auf ein Optimierungsziel das beste Ergebnis liefert [265]. Derartige Autotuning-Methoden existieren zwar bereits für viele, aber längst nicht für alle Anwendungen. Ebenso konzentriert sich das automatisierte Tuning bisher meist auf die Minimierung der Ausführungszeit und berücksichtigt nur selten den Energieverbrauch. Es wird also ein Verfahren benötigt, das automatisches Tuning in eine wissenschaftliche Anwendung einführt, um ihre Ausführungszeit und ihren Energieverbrauch ohne aufwändige Anwenderinteraktion zu verringern.

Beiträge der Arbeit

Diese Arbeit stellt beispielhaft für zwei Anwendungen des wissenschaftlichen Rechnens anwendungsspezifische Energie- und Ausführungszeitmodelle auf und zeigt, wie man diese Modelle nutzen kann, um mit modellbasiertem Autotuning die Zeit bzw. Energie zu verringern, die für die Ausführung der Anwendung benötigt wird. Ziel ist es also, vorhandene Rechner möglichst effizient zu nutzen, indem die Ausführung der Programme mit einem automatischen Verfahren auf diese Rechner angepasst wird. Anhand von Messungen am konkreten Anwendungscode wird mithilfe des Modells vorhergesagt, wie sich die Ausführungszeit bzw. der Energieverbrauch verschiedener Implementierungsvarianten im nächsten Rechenschritt verhalten wird. So ergibt sich die Möglichkeit, stets die beste Implementierungsvariante im Hinblick auf das Optimierungsziel auszuwählen.

Die Anwendungen des wissenschaftlichen Rechnens, die in dieser Arbeit untersucht werden, sind die *Schnelle Multipolmethode* (engl. *fast multipole method,* FMM) und die *Methode der finiten Elemente* (engl. *finite element method,* FEM). Die schnelle Multipolmethode berechnet Wechselwirkungen zwischen Teilchen im Raum. Sie fasst die Teilchen in baumartig im Raum angeordneten Zellen zusammen, um zu erreichen, dass die Ausführungszeit lediglich linear statt quadratisch mit der Anzahl der Teilchen wächst. Die Wechselwirkungen werden hier für die Berechnung in zwei Klassen aufgeteilt: Als *Nahfeldwechselwirkungen* werden Wechselwirkungen zwischen Teilchen in benachbarten Blattknoten des Baumes direkt berechnet. Als *Fernfeldwechselwirkungen* werden Wechselwirkungen zwischen Teilchen in weiter voneinander entfernt liegenden Zellen über die Verschiebung von Multipolentwicklungen approximiert. Die Ausführungszeit bzw. der Energieverbrauch für die einmalige Berechnung beider Arten der Wechselwirkung wird zum Zeitpunkt der Installation der Anwendung bestimmt. Bei der Ausführung der Methode wird jedes Mal zu Beginn in einem Analyselauf die Methode für die konkreten Eingabedaten durchgespielt, jedoch ohne tatsächlich Berechnungen durchzuführen. Stattdessen wird für jede in Frage kommende Baumtiefe ermittelt, wie häufig jede der Arten der Wechselwirkung jeweils auftritt. Anhand dieser Ergebnisse wird für die tatsächliche Ausführung der Methode im Berechnungslauf die Baumtiefe so bestimmt, dass sich die kürzeste Ausführungszeit bzw. der niedrigste Energieverbrauch ergibt. Beitrag der Arbeit ist es zu zeigen, dass durch den Einsatz des Autotuning-Verfahrens gegenüber der statischen Ermittlung der Kosten, d. h. nicht am konkreten Rechner, sowohl die Ausführungszeit als auch der Energieverbrauch deutlich verringert werden können. Existierende Methoden zur Vorhersage der Ausführungszeit, z. B. [95, 57], beziehen zwar rechnerabhängige Parameter in ihr Modell mit ein, nutzen aber nicht die tatsächlich im Rechner auftretenden Ausführungszeiten. Diese Modelle wurden außerdem nicht auf den Energieverbrauch übertragen.

Einen großen Raum in dieser Arbeit nehmen Untersuchungen an einer adaptiven FEM ein. Die Arbeit liefert in diesem Kontext vier wesentliche Beiträge. In der FEM wird die *Methode der konjugierten Gradienten* (engl. *conjugate gradient method,* CGM) zur Lösung eines linearen Gleichungssystems verwendet, das in einer parallelen Implementierung vorliegt. Ein erster Beitrag ist die Optimierung der dort verwendeten Methode zur Reduktion verteilt liegender Ergebnisvektoren zu einem gemeinsamen Ergebnisvektor: Indem die genutzte explizite, grobgranulare Reduktion durch eine implizite, feingranulare Reduktion mit atomaren Operationen ersetzt wird, verkürzt sich die Ausführungszeit der Routine deutlich. Ein zweiter Beitrag ist eine Untersuchung, ob die Berücksichtigung des Speicherortes der Daten bei der Zuweisung einer Verarbeitungseinheit Vorteile bringt. In einem dritten Beitrag wird basierend auf Experimenten ein anwendungsspezifisches Energie- und Ausführunsgszeitmodell für die Ausführung der CGM auf CPUs und Grafikprozessoren (engl. *graphics processing unit,* GPU) aufgestellt. Das Modell berücksichtigt die Ausführungsgeschwindigkeiten der CPU und der GPU, deren elektrische Leistung, Spannungs- und Frequenzregelung sowie die für die Datenübertragung zwischen CPU- und GPU-Speicher benötigte Zeit und Energie. Anhand des Modells wird vorhergesagt, welche Art der Datenverarbeitung die effizienteste Ausführung ermöglicht: CPU-Ausführung, GPU-Ausführung oder gemeinsame CPU-GPU-Ausführung. Existierende Arbeiten zur Verringerung des Energieverbrauchs der CGM widmen sich nur einzelnen Aspekten wie dem Abspeicherungsformat der dünn besetzten Matrizen [201, 186, 47], der Leerlaufenergie der CPU bei der Ausführung von GPU-Routinen [13] oder stellen zwar ein Modell auf, dieses aber lediglich für Multicore-CPUs [177]. Für die gemeinsame Ausführung der CGM auf CPU und GPU wird als vierter Beitrag ein dynamisches, adaptives Verfahren zur Datenverteilung entwickelt, das die Ausführungszeit minimiert. Dieses Verfahren ist ein Online-Autotuning-Verfahren, das es ermöglicht, zur Laufzeit auf sich verändernde Parameter, die sich beispielsweise aus wachsenden Datenmengen ergeben können, zu reagieren. Andere Ansätze zur gemeinsamen Verarbeitung auf CPU und GPU beherrschen lediglich eine statische Aufteilung der Rechenlast [75, 243, 256, 116] oder sind aus anderen Gründen für die vorliegende Anwendung ungeeignet [7, 64, 37].

Zur Optimierung des Energieverbrauchs sind genaue Methoden zu dessen Messung erforderlich. Mit dem Messintervall des Energiemessgerätes auf der GPU von 20 ms kann der Energieverbrauch von Funktionen, deren Laufzeit kürzer ist bzw. deren Leistungsaufnahme im zeitlichen Verlauf schwankt, nicht adäquat ermittelt werden. Daher stellt die Arbeit in einem weiteren Beitrag eine Methode vor, mit der man trotz der niedrigen zeitlichen Auflösung des Messinstruments hochaufgelöste Leistungsprofile erzeugen kann. Anhand dieser Leistungsprofile lässt sich auch der Energieverbrauch von GPU-Routinen mit sehr kurzer Laufzeit ermitteln. Eine ähnliche Methode zur Untersuchung des Energieverbrauchs von Funktionen, deren Ausführungszeit kürzer als das Messintervall ist, betrachtet nicht die elektrische Leistung und existiert lediglich für CPUs [122]. Dadurch ist es nicht notwendig treten einige in dieser Arbeit betrachtete Probleme, wie die Zeitsynchronisation und die Integration der Leistung zur Energie, nicht auf.

Gliederung der Arbeit

Auf dieses einleitende Kapitel folgt das *2. Kapitel*, das begründet, warum die Steigerung der Energieeffizienz im wissenschaftlichen Rechnen erforderlich ist, und erläutert, wie automatisches Tu-

ning helfen kann, komplexe Systeme effizient zu programmieren. Es gibt einen Überblick über energieeffizientes wissenschaftliches Rechnen und stellt den Stand der Wissenschaft in Bezug auf modellbasiertes und Energie-Autotuning dar. Außerdem gibt es einen Überblick über Methoden zur Messung des Energieverbrauchs von Rechnern. Das *3. Kapitel* erläutert, wie mit Ausführungszeitvorhersage die Baumtiefe der FMM so festgelegt werden können, dass die Ausführungszeit minimiert wird. Außerdem wird dargestellt, wie die Methode auf den Energieverbrauch übertragen wird. Das *4. Kapitel* widmet sich der FEM. Es untergliedert sich in fünf wesentliche Teile: In Unterkapitel 4.1 wird zunächst das untersuchte Verfahren zur Lösung von Gleichungssystemen und dessen parallele Implementierung vorgestellt. In Unterkapitel 4.2 wird die Reduktion des verteilt liegenden Ergebnisvektors optimiert. In Unterkapitel 4.3 wird untersucht, welchen Einfluss der Speicherort der Daten auf die Ausführungszeit bzw. den Energieverbrauch bei Ausführung auf den verschiedenen CPUs hat. In Unterkapitel 4.4 wird das Energie- und Ausführungszeitmodell der CGM aufgestellt. Außerdem werden die Experimente vorgestellt, die das Verhalten der Ausführungszeit und der Energie bei der Ausführung auf der CPU und auf der GPU untersuchen. In Unterkapitel 4.5 wird die Methode zur Verteilung der Rechenlast zwischen CPU und GPU erläutert. Im *5. Kapitel* wird die Methode zur Erzeugung zeitlich hochaufgelöster Leistungsprofile vorgestellt, die die in einer GPU integrierte Leistungsmessfunktionalität mit einer für kurze Funktionen unzureichende zeitliche Auflösung benutzt. Das *6. Kapitel* fasst die Ergebnisse der Arbeit zusammen und schließt mit einem Ausblick ab. ■

2 Motivation der Arbeit und Stand der Wissenschaft

Post und Votta erörtern in [216] drei Herausforderungen für die rechnergestützte Naturwissenschaft (engl. *computational science*):

- die *peformance challenge*, also die Herausforderung, leistungsfähige Rechner zu bauen,
- die *programming challenge*, also die Herausforderung, diese leistungsfähigen und komplexen Rechner zu programmieren, sowie
- die *prediction challenge*, also die Herausforderung, Simulationsanwendungen zu entwickeln, die die Realität möglichst gut abbilden.

Die folgenden Abschnitte zeigen die Relevanz der *peformance challenge* und der *programming challenge* auf. Insbesondere wird gezeigt, warum *leistungsfähige* Rechner heute auch *energieeffizient arbeitende* Rechner sein müssen. Auf energieeffizientes wissenschaftliches Rechnen wird in Abschnitt 2.1 näher eingegangen. Um Energie bei der Berechnung einsparen zu können, ist es notwendig, den Energieverbrauch zu messen. Daher gibt Abschnitt 2.2 einen Überblick über Methoden zur Energiemessung. Einen Beitrag zur Lösung der *programming challenge* liefert das *automatisierte Tuning*, kurz *Autotuning*. Autotuning ermöglicht es, Programme an die Hardware anzupassen, sodass sie schnell oder energiesparend ausgeführt werden, ohne dass der Nutzer selbst großen Aufwand hat. Das Autotuning wird in Abschnitt 2.3 vorgestellt.

Die *prediction challenge* hingegen ist in erster Linie ein Problem der Anwender und des Software-Engineerings, da es bei ihr darum geht, Modelle korrekt auszuwählen, zu implementieren und anzuwenden. Sie ist nicht Teil dieser Arbeit.

2.1 Energieeffizientes paralleles wissenschaftliches Rechnen

Zwar hat sich die Energieeffizienz von Rechnern, also die Anzahl an Berechnungen, die mit einer Kilowattstunde elektrischer Energie durchgeführt werden können, von 1949 bis 2009 alle 1,57 Jahre verdoppelt [155]. Das reicht allerdings nicht aus, wenn im Jahr 2020 der erste Exascale-Supercomputer mit einer Rechenleistung von einem Exaflop pro Sekunde (nach altgriech. *ἑξάς* für *sechs* [137, s. v. *sechs*], also 1000^6 flop/s) in Betrieb genommen werden soll [196]. Für einen Supercomputer im Wissenschaftsbereich ist eine elektrische Leistungsaufnahme von über 20 MW nicht finanzierbar. Die mit heutiger Technik notwendige Leistungsaufnahme wäre jedoch für einen Exascale-Rechner um ein vielfaches höher. Auch außerhalb des Spitzenbereichs ist nach Feng und Cameron das Designparadigma „Rechenleistung um jeden Preis“ nicht mehr haltbar [77]. Zur Steigerung der Energieeffizienz beim Rechnen sind Verbesserungen auf Hardwareebene und auf Softwareebene notwendig.

Auf Hardwareebene existieren im Wesentlichen die folgenden Methoden zum Einsparen von Energie:

- Sowohl im Betrieb einer CPU, also während Berechnungen durchgeführt werden, als auch im Leerlauf, also während sie auf Eingaben etc. wartet, kann ihre Leistungsaufnahme gesenkt werden [41]. Jedoch müssen in den beiden Fällen spezifische Techniken zum Senken der Leistung angewandt werden. Dafür existieren jeweils verschiedene Modi, *Leerlaufmodi* und *Betriebsmodi* [151]:
  - **Leerlaufmodi**: Während eine CPU wartet, können diverse nicht benötigte Komponenten abgeschaltet werden. Diese Komponenten werden wieder aktiviert, sobald die CPU weiterarbeitet. Die Leerlaufmodi werden meist als *Prozessorzustände* bzw. *C-States* bezeichnet und in Abschnitt 2.1.3 näher erklärt.
  - **Betriebsmodi**: Im Betrieb können einzelne Komponenten einer CPU nur bedingt abgeschaltet werden. Solange nicht die volle Rechenleistung benötigt wird, können aber die Taktfrequenz und die Versorgungsspannung abgesenkt werden. Dadurch sinkt die Leistungsaufnahme der CPU. Diese *dynamische Spannungs- und Frequenzregelung* nutzt die als P-States bezeichneten Betriebsmodi. Die Hintergründe werden in Abschnitt 2.1.3 erläutert.

  Barroso und Hölzle fordern *energieproportionale Hardware*, die durch passende Kombination der Leerlauf- und Betriebsmodi erreicht, dass die Leistungsaufnahme der Hardware proportional zu ihrer Auslastung steigt [23].

- Die Schaltkreise selbst können energieeffizient implementiert werden. Das wird zunächst durch sinkende Strukturgrößen in der Halbleiterfertigung automatisch erreicht [155]. Außerdem existiert die Möglichkeit, anwendungspezifische Schaltkreise zu entwerfen. Das können einzelne Befehle in einem Prozessor sein oder ganze Prozessoren:
  - **Spezialbefehle** in CPUs wie z. B. SIMD-Befehle zur gleichzeitigen Verarbeitung mehrerer Datenwörter ermöglichen bei rechenintensiven Anwendungen Energieeinsparungen [170]. Dasselbe gilt für die Maschinenbefehle zur AES-Verschlüsselung *(advanced encryption standard)* der Befehlssatzerweiterung *AES-NI (AES native instructions)* [99]. Der Grund liegt darin, dass bei den Spezialbefehlen zur Durchführung derselben Aufgabe weniger Transistoren aktiviert werden müssen als es bei der Ausführung mit allgemeinen Befehlen der Fall wäre.
  - **Spezialprozessoren**, also Prozessoren, die zwar ggf. nicht turing-mächtig sind, aber dafür genau für einen bestimmten Einsatzzweck entwickelt wurden, können diese Aufgabe deutlich energieeffizienter erledigen als allgemeine Prozessoren. Für Videokodierung wurde das in [109] gezeigt. I/O-Controller wie USB- oder Festplattencontroller entlasten die CPU von der Ansteuerung der Ein- und Ausgabegeräte. Auch Koprozessoren wie der mathematische Koprozessor Intel 80487 [123, s. v. *coprocessor*] oder Beschleuniger wie Grafikprozessoren [145] oder der Intel Xeon Phi [138] lassen sich in die Klasse der Spezialprozessoren einordnen.

Auf Softwareebene sind folgende Zuständigkeiten für die Steigerung der Energieeffizienz denkbar:

- Das Betriebssystem sollte mit geeigneten Heuristiken [211, 212] dafür sorgen, dass automatisch passende Betriebs- oder Leerlaufmodi gewählt werden.
- Der Compiler sollte dafür sorgen, dass der Quellcode, wo möglich, in energieeffiziente Hardwarebefehle wie SIMD- oder AES-NI-Befehle übersetzt wird. Außerdem könnten künftige Compiler einen Optionsschalter anbieten, der nicht im Hinblick auf minimale Ausführungszeit optimiert, sondern für minimalen Energieverbrauch. Dass das sinnvoll ist, zeigen Trefethen und Thiyagalingam [259] für eine parallele Anwendung aus der NAS Benchmark Suite [15]: Dort liegt die minimale Ausführungszeit auf einem Rechner bei *einer* Threadanzahl, der minimale Energieverbrauch jedoch bei einer *anderen* Threadanzahl. Dazu kommt noch, dass beim Einsatz zweier verschiedener Compiler zwar beide die Ausführungszeit bei der gleichen Threadanzahl minimierten, die Threadanzahl mit minimalen Energieverbrauch jedoch unterschiedlich war.
- Anwendungsentwickler können auf verschiedenen Ebenen für eine höhere Energieeffizienz sorgen:
  - Werden Algorithmen für Grafikprozessoren und andere Beschleuniger entwickelt und implementiert, lassen sich diese zur energieeffizienteren Ausführung nutzten [1, 121, 231, 13].
  - Unter anderen schlagen Dongarra et al. [66] sowie Shalf et al. [239] vor, Quelltexte mit Energie- bzw. Ausführungszeitmodellen zu annotieren. Diese könnte das Betriebssystem zur Auswahl der Leerlauf- und Betriebsmodi oder der Scheduler für einen möglichst energiesparenden Ablaufplan nutzen [260].
  - Bestehende Anwendungen lassen sich hinsichtlich ihres Energieverbrauchs optimieren, indem eine Möglichkeit geschaffen wird, über Parameter ihren Ablauf zu modifizieren bzw. verschiedene Implementierungsvarianten zu verwenden. Da bei jeder neuen Hardware das Optimum ggf. mit einer anderen Parameterkonfiguration erreicht wird, bietet es sich an, den Optimierungsvorgang über Autotuning zu automatisieren, s. Abschnitt 2.3.3.
  - Bei einigen Algorithmen lassen sich Berechnungen gegen Datenübertragung eintauschen. Dieser Trade-off kann so gewählt werden, dass ein Energieoptimum getroffen wird [54]. Nach Demmel und Gearhart wird der Energieverbrauch des Speicherverkehrs auf einem Knoten die größte Komponente des Energieverbrauchs für viele Algorithmen werden [62], sodass kommunikationsvermeidende Algorithmen wie in [63] oder [19] vorgestellt interessant werden.

Zur Bezeichnung der elektrischen Energie, die ein Rechnersystem aufnimmt, soll in dieser Arbeit der anschauliche Begriff *Energieverbrauch* genutzt werden. Tatsächlich wird die Energie natürlich nur aus Sicht des Stromnetzbetreibers „verbraucht“, in dem Sinne, dass sie nach dem „Verbrauch“ nicht weiter zur Verfügung steht. Aus physikalischer Sicht wird die aufgenommene elektrische

Energie in Wärme und zu geringen Teilen auch weitere Energieformen wie elektromagnetische Strahlung und kinetische Energie umgewandelt. Daher wird in der Literatur auch häufig der Begriff *Dissipation* (engl. *dissipation*), also der „Übergang irgendeiner Energieform in Wärme“ [102, s. v. Dissipation], verwendet, z. B. in [74, 249].

2.1.1 Definition von Energieeffizienz

In allgemeiner Form wird *Energieeffizienz*, wie zum Beispiel in [215], häufig definiert als das Verhältnis zwischen den „nützlichen Produkten“[1] eines Prozesses und dem Energieeinsatz in einem Prozess. Im wissenschaftlichen Rechnen wird als „nützliches Produkt“ regelmäßig die Durchführung einer Berechnung [155] bzw. einer Rechenoperation (flop) [77] angesehen. Aber auch andere Maße für das Produkt wie „Anzahl sortierter Datenobjekte“ [230] oder „Algorithmusschritte“ [119] werden verwendet. Der Energieaufwand des Rechnersystems wird meist in Kilowattstunden (kW h) oder Joule (J) angegeben. In einigen Arbeiten wird der reziproke Wert, also der Energieaufwand für eine Einheit (Rechen-)Arbeit, verwendet [255, 240], z. B. in [240] als „Energie pro Instruktion“ oder in .

Ein weiteres denkbares Maß für die Energieeffizienz ist das Verhältnis zwischen theoretisch minimal notwendiger Energie und tatsächlich aufgewandter Energie für die Durchführung einer Berechnung bzw. eines Algorithmus. Die theoretisch minimal notwendige Energie lässt sich beispielsweise mit Konzepten wie Lloyds *optimalem Laptop* (engl. *ultimate laptop*) ermitteln. Der optimale Laptop ist ein Modell für einen Rechner, das nur den sich aus physikalischen Gesetzen zwingend ergebenden Energiebedarf für die Ausführung von Algorithmen berücksichtigt. Dazu gehört zum Beispiel die sich aus dem Landauer-Prinzip [159, 26] möglicherweise ergebende Mindestenergie zum Löschen eines Bits oder die sich aus der Heisenbergschen Unschärferelation [108] ergebende Energie, die benötigt wird, um in einer bestimmten Zeit eine logische Operation durchzuführen [39].

In dieser Arbeit wird die erstgenannte Definition der Energieeffizienz genutzt, also die pro Energieeinheit Anzahl durchgeführter Gleitkommaoperationen oder verarbeiteter finiter Elemente der FEM. Die Einheit ist also flop/J oder Elemente/J. Je größer dieser Wert ist, desto energieeffizienter arbeitet das System.

2.1.2 Notwendigkeit von Energieeffizienz

Eine Motivation für die Steigerung der Energieeffizienz ist eine gesellschaftliche: Die Energiewende ist nur zu schaffen, wenn der Energieverbrauch unserer Gesellschaft, wozu auch die Wissenschaft zählt, deutlich reduziert wird. Eine wesentliche Rolle spielt dafür die Steigerung der Energieeffizienz [171, 139, 273]. Die für das wissenschaftliche Rechnen gewonnenen Erkentnisse zur Steigerung der Energieeffizienz lassen sich auf andere Probleme, die beispielsweise in der Industrie häufiger auftreten, übertragen. Von 2006 bis 2008 stieg der Energieverbrauch der Rechenzentren in Deutschland von 8,67 TW h auf 10,1 TW h. Ziel muss es sein, dass sich dieser absolute Wert trotz steigender installierter und nachgefragter Rechenleistung nicht weiter erhöht. Das ist nur durch eine Steigerung der Energieeffizienz möglich. Dasselbe gilt für mobile Anwendungen in Mobiltelefonen oder

1 “useful output”

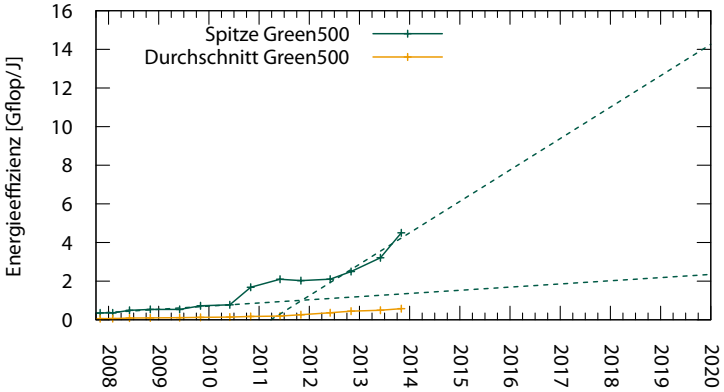


Abbildung 2.1: Entwicklung der Energieeffizienz der Supercomputer der Green-500-Liste [94] von 2007 bis heute; Darstellung der Extrapolation der Leistungsentwicklung (gestrichelte Geraden)

Laptops: Die Kapazität von Akkus lässt sich aus physikalischen Gründen nicht mehr wesentlich steigern, wenn Masse und Volumen vorgegeben sind [194]. Als Folge ist eine höhere mobile Rechenleistung allein durch die Steigerung der Energieeffizienz möglich.

Eine zweite Motivation liegt in der Wissenschaft selbst: Exascale-Computing wird erst dann wirtschaftlich möglich, wenn die Energieeffizienz der Supercomputer deutlich gesteigert worden ist. Der aktuell schnellste Supercomputer in der Wissenschaft ist das chinesische System *Tianhe-2* (chin. 天河二号, *tiānhé-èrhào*) [65] mit einer Rechenleistung von rund 34 Pflop/s [257] und einer elektrischen Leistungsaufnahme von 17,6 MW (ohne Kühlung) [65], also einer Energieeffizienz von 1,9 Gflop/J. Das Spitzensystem auf der Green-500-Liste der energieeffizientesten Supercomputer hat eine Energieeffizienz von 4,5 Gflop/J [94]. Abbildung 2.1 zeigt die Entwicklung der Energieeffizienz der Supercomputer dieser Liste von 2007 bis heute [94]. Es ist erkennbar, dass seit dem Jahr 2010 die Effizienz des jeweiligen Spitzensystems deutlich schneller ansteigt als in der Zeit davor. Würde die Entwicklung seit Mitte 2012 bis zum Jahr 2020 fortgeschrieben, wie es die gestrichelte Regressionsgerade tut, erreichte man dann allerdings lediglich eine Energieeffizienz von ca. 14 Gflop/J. Nötig wären aber 10^{18} flop/s/20 MW = 50 Gflop/J. Das zeigt, dass bis zum Erreichen der Exascale-Marke noch viel Forschungsarbeit zu leisten ist.

2.1.3 Hardwareunterstützung zur Energieeinsparung

Zunächst wird jedoch das Gesetz, das die Leistungsaufnahme von CMOS-Schaltkreisen *(complementary metal-oxide semiconductor)* beschreibt, vorgestellt. Mit dieser Gleichung lassen verschiedene Mechanismen erklären, die eine Energieeinsparung auf Hardwareebene ermöglichen. Diese Mechanismen werden im Anschluss beschrieben.

Leistungsaufnahme von CMOS-Schaltkreisen

Moderne Prozessoren und DRAM-Speichermodule werden in CMOS-Technologie gefertigt [213]. Für diese Art von Schaltkreisen wird die elektrische Leistungsaufnahme üblicherweise durch die Gleichung

$$P = \alpha C U^2 f + \tau \alpha U I_{\mathrm{k}} + U I_{\mathrm{leck}} \tag{2.1}$$

modelliert [192]. Im ersten Summanden des Terms steht α für die Aktivität der Logikgatter, C für die effektive Schaltkapazität der Transistore, U für die Versorgungsspannung des Prozessors und f für dessen Taktfrequenz. Im zweiten Summanden steht zusätzlich I_{k} für den Kurzschlussstrom, der für den Moment τ der Umschaltung der Gatter zwischen der Spannungsquelle und der Masse fließt. Im dritten Summanden steht I_{leck} für den Leckstrom, der jederzeit unabhängig von dem Zustand der Gatter fließt. Besonders die wachsende Strukturdichte neuerer Fertigungsverfahren für Mikroprozessoren, welche zu immer kleineren Transistoren und Leiterbahnen auf den Chips führt, sorgt für ein Anwachsen dieses Leckstroms über die Prozessorgenerationen hinweg [152]. Der Faktor α im ersten Summanden wird benötigt, da nicht jeder Transistor in jedem Takt geschaltet wird [192]. Zur Vereinfachung wird er häufig auf 1 gesetzt, vgl. z. B. [272].

Die elektrische Leistung eines Chips setzt sich grundsätzlich aus einem statischen Anteil P_{stat} und einem dynamischen Anteil P_{dyn} zusammen. Der statische Anteil wirkt stets, auch ohne dass Berechnungen durchgeführt werden. Der dynamische Anteil wird nur wirksam, wenn die Transistoren geschalten werden, also Berechnungs- oder Speichervorgänge durchgeführt werden. Die ersten beiden Summanden der Gleichung (2.1) stellen dabei den dynamischen Anteil und der dritte Summand den statischen Anteil dar. Der zweite Summand ist hierbei gegenüber dem ersten meist relativ klein, sodass er häufig weggelassen wird [152]. Es ergibt sich folglich für die Aufnahme elektrischer Leistung eines CMOS die folgende vereinfachte Formel:

$$P = P_{\mathrm{dyn}} + P_{\mathrm{stat}} \tag{2.2}$$

mit

$$P_{\mathrm{dyn}} = C U^2 f \quad \text{und} \quad P_{\mathrm{stat}} = U I_{\mathrm{leck}} \quad . \tag{2.3}$$

Bei Chips, die in einer älteren Fertigungstechnologie hergestellt wurden, wird meistens sogar noch der statische Anteil an der Leistungsaufnahme vernachlässigt [152].

Neben den in Gleichung (2.1) dargestellten Größen hat auch die Temperatur des Schaltkreis einen Einfluss auf dessen Leistungsaufnahme: Die dynamische Leistung steigt linear mit steigender Temperatur [213, 251]; die statische Leistung, hauptsächlich verursacht durch den steigenden Leckstrom, steigt exponentiell mit steigender Temperatur [213, 283].

Dynamische Spannungs- und Frequenzregelung

Gleichung (2.2) mit Gleichung (2.3) zeigt, dass durch Absenken der Betriebsspannung die Leistungsaufnahme eines Schaltkreises gesenkt werden kann. Allerdings benötigen dann aufgrund der geingeren Steilheit der Taktflanken die Transistoren mehr Zeit zum Umschalten [27], sodass mit der Spannung auch die Taktfrequenz abzusenken ist. Dafür wurde ein linearer Zusammenhang experimentell ermittelt [48, 192, 130]. In der Praxis wird meist vereinfachend eine Proportionalität

zwischen der Taktfrequenz und der Spannung angenommen [223, 284, 156], die hier auch verwendet werden soll.

Ersetzt man also in Gleichung (2.1) U durch f (multipliziert mit einer Konstanten), ergibt sich eine kubische Abhängigkeit der Leistung von der Taktfrequenz. Die Absenkung der Taktfrequenz birgt also ein großes Einsparpotenzial [272]. Diese Technik ist unter dem Begriff *dynamische Spannungs- und Frequenzregelung* (engl. *dynamic voltage and frequency scaling, DVFS*) bekannt. Suda et al. bezeichnen diese als die „effektivste“[2] Möglichkeit, die Leistungsaufnahme softwareseitig zu steuern [251].

Nach dem ACPI-Standard *(advanced configuration and power interface)* [111] können für einen Prozessor bis zu 16 Betriebsmodi, so genannte *P-States*, definiert werden [41], denen jeweils eine Taktfrequenz zugeordnet ist. `P0` ist der P-State mit der maximalen Taktfrequenz. In den weiteren P-States `P1` bis `P15` nimmt die Taktfrequenz und damit auch die Leistungsaufnahme immer weiter ab. Zwischen den P-States kann im laufenden Betrieb gewechselt werden.

Im Betriebssystem Linux werden die P-States durch das Kernelmodul *cpufreq* verwaltet. Durch Auswahl eines *P-State-Reglers* (engl. *governor*) kann der Nutzer die Strategie, nach der der P-State gewählt werden soll, festlegen. Beispielsweise belässt der Regler *performance* stets den P-State `P0`. Der Regler *ondemand* [212] setzt den P-State in Abhängigkeit von der Auslastung der CPU. Der Regler *userspace* macht eine Schnittstelle zum Setzen des P-States im Userspace zugänglich.

Die Schnittstelle des *cpufreq*-Kernelmoduls bilden Dateien, die für den CPU-Kern mit der Ordnungsnummer x im Verzeichnis `/sys/devices/system/cpu/cpu`x`/cpufreq` liegen. Der aktive P-State-Regler kann über die Datei `scaling_governor` ausgelesen bzw. gesetzt werden. Über die Datei `scaling_cur_freq` kann die aktuelle Taktfrequenz gelesen und – bei aktivem *userspace*-Regler – über die Datei `scaling_setspeed` verändert werden. Die Taktfrequenzen der verfügbaren P-States lassen sich aus der Datei `scaling_available_frequencies` auslesen.

Bei älteren Prozessoren wird im P-State `P0` die CPU mit ihrer nominalen Taktfrequenz der CPU getaktet. Hingegen wird bei neueren Prozessoren von AMD und Intel, die die Technologien *Turbo Core* bzw. *Turbo Boost* unterstützen, die nominale Taktfrequenz im P-State `P1` verwendet. Der P-State `P0` ist dann der *Turbo-Modus*. In diesem Modus läuft die CPU mit einer höheren als der nominalen Taktfrequenz, solange bestimmte Randbedingungen eingehalten werden. Dazu gehört unter anderem, dass die benötigte Leistung und die Prozessortemperatur die vom Hersteller spezifizierten Werte nicht überschreiten [125].

Auch Grafikprozessoren bieten Techniken zum Nutzen von Betriebszuständen mit verringerter Leistungsaufnahme, die bei AMD *PowerTune* [3] und bei Nvidia *PowerMizer* [204] genannt werden. Ebenso kann DRAM-Speicher über die beschriebenen Verfahren in Zustände mit niedrigerer Leistungsaufnahme versetzt werden [190].

Prozessorzustände

Bei vielen Anwendungen sind Prozessoren lange Zeit im Leerlauf, da sie auf Daten warten. Das können beispielsweise Benutzereingaben oder Eingabedaten für Berechnungen sein, die im Hauptspeicher liegen. Zum Verringern des Energieverbrauchs während solcher Wartephasen definiert

2 “most effective”

ACPI verschiedene *Leerlaufmodi*, die *Prozessorzustände* bzw. *C-States*. C0 ist dabei der Zustand des normalen Betriebs. Die Zustände C1 bis C*n* werden für Pausen wachsender Dauer verwendet, wobei die genaue Anzahl und Wirkungsweise dieser Zustände vom Standard nicht vorgegeben wird. Mit steigender Ordnungszahl sinkt die Leistungsaufnahme, gleichzeitig steigen jedoch die Latenz und der Energiebedarf für die Rückkehr in den Normalbetriebszustand C0 [34]. Als Techniken zum Einsparen von Energie werden beispielsweise das Anhalten der CPU bzw. verschiedener Taktgeber, ein Verringern der Versorgungsspannung oder das Abschalten nicht benötigter CPU-Komponenten oder ganzer CPU-Kerne genutzt [272, 28, 258]. Im Linux-Kernel werden die C-States vom *cpuidle*-Subsystem verwaltet. In dieser Arbeit werden C-States nicht aktiv genutzt: Es wird davon ausgegangen, dass das Betriebssystem stets den passenden C-State automatisch wählt.

Spezialbefehle

Bei den Spezialbefehlen lassen sich zwei Klassen unterscheiden: *SIMD-Befehle* und *anwendungsspezifische Befehle*. Zu den SIMD-Befehlen gehören beispielsweise bei *amd64*-CPUs die MMX- und SSE-Befehle [129] oder bei ARM-CPUs die NEON-Befehle [14]. Anwendungsspezifische Befehle werden zum Beispiel durch die Befehlssatzerweiterung *AES-NI (AES native instructions)* [99] zur Unterstützung der AES-Verschlüsselung *(advanced encryption standard)* bereitgestellt. FMA-Befehle *(fused multiply-add)* unterstützen numerische Berechnungen, indem sie die Durchführung einer Addition und einer Multiplikation in einem Befehl ermöglichen [191].

Durch Nutzung von SIMD-Befehlen lassen sich bei rechenintensiven Anwendungen Energieeinsparungen erzielen [170]. Begründen lässt sich das mit Formel (2.1): Betrachtet wird idealisiert eine CMOS-Schaltung für einen SIMD-Befehl, der bis zu zwei Datenworte auf einmal verarbeitet. Erfolgt die Verarbeitung nach dem SIMD-Schema, d. h. wird der Befehl wie vorgesehen für zwei Datenworte genutzt, gilt $\alpha = 1$, es werden also alle Transistoren aktiviert. Wird hingegen nur ein Datenwort verarbeitet, gilt $\alpha = \frac{1}{2}$, da dann nur die Hälfte der Transistoren aktiviert. Bei paralleler Verarbeitung beträgt die dynamische Leistung also das Doppelte der dynamischen Leistung von sequentieller Verarbeitung. Allerdings halbiert sich bei paralleler Verarbeitung die Ausführungszeit t. Da $E = Pt$ gilt, bleibt die notwendige dynamische Energie jedoch konstant – unabhängig von der Anzahl der durch die SIMD-Einheit parallel verarbeiteten Datenworte. Durch die Verkürzung der Ausführungszeit verringert sich jedoch die statische Energie, was eine Einsparung im Gesamtenergieverbrauch zur Folge hat.

Anwendungsspezifische Befehle ermöglichen eine Verringerung der Leistungsaufnahme durch Senkung der Aktivität α: Für die Ausführung des Spezialbefehls werden weniger Transistoren benötigt als für die stattdessen notwendigen generischen Befehle. Daher benötigt beispielsweise eine AES-Implementierung unter Nutzung der AES-Maschinenbefehle im Vergleich zur Implementierung mit generischen Maschinenbefehlen nur ca. ein Drittel der Energie zum Ver- und Entschlüsseln der gleichen Datenmenge [161].

Power-Capping

Mit *Power-Capping* wird die mittlere Leistungsaufnahme eines Rechners oder einer Rechnerkomponente in einem Zeitintervall beschränkt, sodass sie unterhalb einer vorgegebenen Schranke bleibt

[81]. Diese Schranke kann vom Nutzer festgelegt werden oder aufgrund von technischen Rahmenbedingungen vorgegeben sein. Für das Power-Capping von CPUs existieren die Technologien *Running Average Power Limit* von Intel [233], *Application Power Management* von AMD [4] und *EnergyScale* von IBM [24, 44, 181]. Diese Techniken basieren meist darauf, den P-State der CPU so einzustellen, dass die vorgegebene Leistungsgrenze gerade eingehalten wird [81]. AMD bietet unabhängig von der CPU auch für den Hauptspeicher Power-Capping [234].

GPUs betreiben Power-Capping, um die maximale Leistung, die die Spezifikation für PCI-Express-Steckkarten vorgibt, nicht zu überschreiten [206, 3].

2.2 Energiemessung in Computern

Zum Messen des Energieverbrauchs eines Rechners bzw. seiner Komponenten beim Ausführen von Algorithmen existieren vielfältige Methoden. Hsu und Poole unterteilen sie in softwarebasierte und hardwarebasierte Methoden [120]. Softwaremethoden zählen Ereignisse oder Maschinenbefehle und berechnen daraus anhand eines Modells den Energieverbrauch. Hardwaremethoden messen physikalische Größen an der Hardware, wie z. B. einen Spannungsabfall bei der Stromversorgung, und ermitteln daraus die Leistung bzw. den Energieverbrauch. Song et al. [244] untergliedern sie weiter in interne und externe Messmethoden. Bei internen Methoden wird der Messwert vom Gerät selber ermittelt und dem Anwender über eine Softwareschnittstelle übergeben. Bei externen Methoden misst externe Hardware die Werte, die dann beispielsweise an einen Auswertungsrechner übertragen werden. Nachfolgend wird ein kurzer Überblick über die Methoden gegeben.

In einigen Fällen liefert die Messmethode die elektrische Leistung zum Messzeitpunkt. In so einem Fall muss die Leistung in ihrem Verlauf aufgezeichnet und über die Zeit integriert werden. Dafür ist eine Zeitmessung notwendig. Geeignet ist zum Beispiel bei *x86*- und *amd64*-Architekturen der Maschinenbefehl RDTSC [247, 214] bzw. allgemein der Linux-Systemruf `clock_gettime` [150, S. 491].

2.2.1 Softwarebasierte Messmethoden

Softwarebasierte Methoden, die den Energieverbrauch einer CPU anhand von Hardware-Performance-Countern [245] abschätzen, wurden beispielsweise in [25], [51], [244], [186], [166] und [35] vorgestellt. [51] erreicht eine mittlere Abweichung der Schätzung vom tatsächlichen Messwert in Höhe von 3,5 %; [35] erreicht eine Abweichung von stets unter 9 % für jedes untersuchte Teilsystem. In [248] wird mit Signifikanzanalyse ermittelt, welche Hardware-Performance-Counter den größten Einfluss auf den Energieverbrauch der Anwendung haben. Das Modell wird dann durch Regression automatisch aufgestellt. Nach [89] reichen gerade einmal 4 Performance-Counter, um eine Abweichung von maximal 10 % zu erreichen. Allerdings stellten McCullough et al. fest, dass Performance-Counter-Methoden in komplexen Situationen möglicherweise ungenau sind [182].

Eine Möglichkeit, auch für noch nicht existierende Hardware den Energieverbrauch abzuschätzen, stellen Simulationsmethoden dar. So wird beispielsweise in [242] ein Prozessorsimulator auf Anweisungsebene um ein taktzyklengenaues Energiemodell erweitert. In [40] wird ein gröberes Modell eines Prozessors entwickelt, das jedes Modul (Logik, Speicher, Taktgeber, Drähte) einzeln

wiederum durch ein parametrisierbares Modell beschreibt. Ein Simulator führt das Anwendungsprogramm aus, sodass der Energiebedarf anhand dessen, wie jedes der Module beansprucht wird, berechnet werden kann. Ebenfalls über Ausführung von Programmen in einem Simulator wird der Energiebedarf in [101] bestimmt. Hier wird jedoch ein kompletter Rechner inkl. RAM, Festplatte usw. modelliert.

2.2.2 Hardwarebasierte Messmethoden

Direkte Methoden zur Energiemessung messen physikalische Größen an der Hardware und ermitteln daraus die aktuelle Leistung bzw. den Energieverbrauch.

Externe Messung

Für die externe Messung wird spezielle Messhardware an die Rechner bzw. die Rechnerkomponenten angeschlossen, für die der Energieverbrauch bestimmt werden soll. Das Grundprinzip wird in [84] beschrieben. In der Regel besteht die Methode darin, den Spannungsversorgungsleiter zu unterbrechen, einen Widerstand an dieser Stelle einzusetzen und den Spannungsabfall ΔU an diesem Widerstand R zu messen. Der fließende Strom I ergibt sich über die Formel $I = \Delta U / R$, mit Kenntnis der Ausgangsspannung U_0 lässt sich über die Formel $P = U_0 I$ die Leistung P ermitteln. Eine andere, zerstörungsfreie Methode wird in [212] gewählt: Dort wird an das Stromkabel ein Hall-Sensor angebracht. Dieser bestimmt die Stärke des stationären Magnetfeldes, das durch den Stromfluss im Leiter entsteht [167, s. v. Hall-Effekt]. Aus der Stärke des Magnetfeldes kann auf den Strom I, der im Leiter fließt, geschlossen werden.

ACPI

ACPI *(Advanced Configuration and Power Interface)* [111] ermöglicht das Auslesen des Ladezustandes des Akkus beispielsweise bei Laptops. Im Betriebssystem Linux wird dessen Wert über die Datei `/proc/acpi/battery/BATx/state` zugänglich gemacht [42]. Hierbei ist x die Nummer der Batterie (bei Zählung beginnend von 0). Ausgegeben wird die vom Akku lieferbare Restenergie sowie die derzeit verbrauchte Leistung [42]. Zur Messung des Energieverbrauchs von Anwendungen wird diese Methode z. B. von [178] und [179] genutzt.

IPMI

Das IPMI *(Intelligent Platform Management Interface)* [135] ermöglicht das Auslesen von Werten, die der vor allem in Server-Systemen vorhandene *Baseboard Management Controller* (BMC) misst. Typischerweise verfügt er unter anderem über Sensoren zur Messung der Mainboard-Temperatur, verschiedener Versorgungsspannungen oder der Lüfterdrehzahlen. Sofern das Netzteil es unterstützt, kann über IPMI auch die aktuelle Leistungsaufnahme des Rechners ausgelesen werden. Die Nutzung von IPMI zur Ermittlung des Energieverbrauchs von Algorithmen wurde z. B. in [105] untersucht.

Neben Servern bietet auch die Beschleunigerkarte *Xeon Phi* die Möglichkeit, die aktuelle Leistungsaufnahme über IPMI zu ermitteln [138, S. 246]. Über das Werkzeug *micsmc*, das für die Überwachung und Verwaltung der Xeon-Phi-Koprozessoren verantwortlich ist, lässt sich die aktuelle

Symbol, Bezeichner	Nr.[1]	Sandy Bridge		Ivy Bridge		Bedeutung
		Client	Server	Client	Server	
MSR_PKG_ENERGY_STATUS	611	+	+	+	+	gesamter CPU-Sockel[2]
MSR_PP0_ENERGY_STATUS	639	+	+	+	+	Prozessorkerne[3]
MSR_PP1_ENERGY_STATUS	641	+	–	+	–	„Uncore"-Geräte[4]
MSR_DRAM_ENERGY_STATUS	619	–	+	–	+	DIMMs des Sockels

1 hexadezimal
2 entspricht der *Package-Energie*
3 inkl. Caches [222] außer Last-Level-Cache [100]
4 für Sandy-Bridge-CPUs: Energieverbrauch des Onboard-Grafikchips; Uncore-Energieverbrauch = PKG−PP0−PP1 [62]

Tabelle 2.1: Semantik der RAPL-MSRs und ihre Unterstützung in Intel-CPUs, Quelle: [133, Bd. 3B, Kap. 14.7]

elektrische Leistung der Koprozessoren vom Hostsystem auslesen [17, 269]. Die Koprozessoren erhalten über die Datei `/sys/class/micras/power` Zugriff auf diese Informationen [267]. Die zeitliche Auflösung der Messung beträgt 50 ms, die Auflösung des Leistungsmesswerts 1 W [17].

GPUs

Aktuelle GPUs unterstützen Power-Capping, s. Abschnitt 2.1.3. Um die Leistung bei Überschreiten eines bestimmten Wertes beschränken zu können, muss sie gemessen werden. Nvdia-GPUs bieten dem Anwender die Möglichkeit, diesen Messwert auch auszulesen. Dazu dient die Funktion `nvmlDeviceGetPowerUsage` der Bibliothek NVML [207]. Der Messwert wird alle 20 ms aktualisiert [163] und hat eine Genauigkeit von ±5 W [207]. Gemessen wird die gesamte Leistungsaufnahme der Grafikkarte am PCI-Express-Slot inkl. der zusätzlichen Spannungsversorgungskabel.

RAPL

Intel-CPUs bieten ab der Mikroarchitektur mit dem Codenamen *Sandy Bridge* die Möglichkeit des Power-Cappings über die Funktionalität *Running Average Power Limit* (RAPL) [233]. Sie macht über maschinenspezifische Register (MSRs) den Energieverbrauch seit einem willkürlichen Startzeitpunkt verfügbar [133, Bd. 3B, Kap. 34.7.2]. Die Werte werden ca. alle 1 ms aktualisiert [133, Bd. 3B, Kap. 34.7.2]. Tabelle 2.1 zeigt die auf den verschiedenen Architekturen verfügbaren maschinenspezifischen Register und ihre Semantik. Die *Uncore*-Energie umfasst u. a. die Spannungsreglungseinheit, den Speichercontroller [134], den PCI-Express-Controller sowie den Cache der letzten Stufe [100].

Der von RAPL übergebene Energiewert wird nicht gemessen, sondern anhand von Hardware-Performance-Countern abgeschätzt [233]. Da das gesamte Verfahren allerdings in der CPU durchgeführt wird, wird die Methode hier den hardwarebasierten Verfahren zugeordnet. Arbeiten, die die Genauigkeit der Abschätzung überprüft haben [224, 233, 268, 122, 105] kommen zum Ergebnis, dass eine starke Korrelation zwischen der RAPL-Schätzung und den gemessenen Werten herrscht. Lediglich Hackenberg et al. stellen fest, dass die Hyperthreading-Funktionalität offenbar von dem Modell, das RAPL zugrunde liegt, nicht ordnungsgemäß abgedeckt wird [105].

Application Power Management

Das *Application Power Management* (APM) [4, Kap. 2.5.2.1.1] wurde von AMD mit der Prozessorgeneration 15h (Codename der Architektur: *Bulldozer*) eingeführt. APM ist wie RAPL ein Power-Capping-Mechanismus und darf nicht mit dem Advanced Power Management, dem Vorläufer von ACPI verwechselt werden.

Die Leistung wird beim Application Power Management über eine „thermisch signifikante“[3] Zeitspanne gemessen [4, Kap. 2.5.2.1.1], an deren Ende sie verfügbar gemacht wird [195]. Das Intervall beträgt etwa 10 ms [105]. Die durchschnittliche Leistung im vorangegangenen Intervall kann der Benutzer ermitteln, indem er die Werte der in [4, Kap. 3.13 MSRC001_0077] angegebenen Register ausliest und in die ebenfalls dort angegebene Formel einsetzt. Zu beachten ist, dass das Auslesen nur über das Sideband-Interface über Advanced Platform Management Link [2] möglich ist. Hackenberg et al. haben die Ausgabe von APM mit einer Hardwaremessung verglichen und kommen zu dem Schluss, dass APM zwar eine konsistente Leistungsabschätzung für die meisten Arten von Rechenlast bietet, aber Turbo- und Leerlaufmodi der CPU nicht korrekt behandelt [105].

2.3 Autotuning

Automatisiertes Performance-Tuning, oder auch kurz *Autotuning* genannt, ist nach Vuduc [265] „ein automatisierter, durch Experimente geführter Vorgang des Auswählens einer Programmimplementierung aus einer Menge von Kandidaten, um ein bestimmtes Leistungsziel zu erreichen. ‚Leistungsziel‘ kann z. B. die Minimierung der Ausführungszeit, des Energie-Zeit-Produktes, des Speichers oder des Approximationsfehlers bedeuten. Ein ‚Experiment‘ ist die Ausführung eines Benchmarks und die Beobachtung seines Ergebnisses im Hinblick auf das Leistungsziel.“[4] [265]. Gegenüber manuellem Tuning hat Autotuning nach Yelick [276] den Vorteil, dass damit schnell auf veränderte Hardware- oder Systemsoftwarebedingungen reagiert werden könne. Außerdem seien damit keine Spezialkenntnisse in Gebieten wie Numerische Analysis, mathematische Software, Compiler und Rechnerarchitektur notwendig [276]. Ebenso begründen Whaley et al. die Notwendigkeit des Autotunings mit immer kürzeren Hardwareentwicklungszyklen: Zu dem Zeitpunkt, zu dem (manuell) hoch optimierter Code für eine Architektur zur Verfügung stehe, sei diese Architektur schon wieder nahezu veraltet [270]. Autotuning erhöht so die wichtige *performance portability* eines Codes, also die Anzahl der Codezeilen, die bei einer Übertragung auf andere Architekturen zur Optimierung der Rechenleistung geändert werden müssen [154].

Vuduc beschreibt folgende Phasen, die ein Autotuner typischerweise durchführt [265]:

- *Identifikation des Kandidatenraumes*: Es werden mögliche Varianten zur Implementierung eines Algorithmus oder alternative Algorithmen zum Erreichen des Ziels identifiziert.
- *Codegenerierung*: Für jede Implementierungsvariante wird der entsprechende Code erzeugt.

3 “thermally significant”

4 “Automated performance tuning, or *autotuning*, is an automated process, guided by experiments, of selecting one from among a set of candidate program implementations to achieve some performance goal. ‘Performance goal’ may mean, for instance, the minimization of execution time, energy delay, storage, or approximation error. An ‘experiment’ is the execution of a benchmark and observation of its results with respect to the performance goal.” [265]

- *Suche des Optimums*: Unter den erzeugten Codevarianten wird diejenige ausgewählt, die das Leistungsziel am besten erfüllt.

Die Suche kann eine erschöpfende Suche im gesamten Kandidatenraum sein, die alle Codevarianten testweise ausführt und evaluiert. Zum Teil werden Heuristiken verwendet, um den Suchraum einzuschränken. Die Suche kann aber auch durch andere Optimierungsverfahren wie Simuliertes Abkühlen, maschinelles Lernen oder genetische Algorithmen durchgeführt werden. Das *modellbasierte Autotuning* nutzt Modelle zur Abschätzung des Leistungsmaßes der Codevarianten, um so die Suche zu verkürzen oder ganz einzusparen, s. Abschnitt 2.3.4. Die drei genannten Schritte des Autotunings werden meist zum Zeitpunkt der Installation der Anwendung bzw. der Bibliothek durchgeführt. In diesem Fall spricht man von *Offline-Autotuning*. Von *Online-Autotuning* spricht man, wenn zumindest die Suche zur Laufzeit des Programms durchgeführt wird, s. Abschnitt 2.3.2.

Bekannte Autotuning-Bibliotheken sind etwa PHIPAC [33] und ATLAS [270] für Lineare Algebra mit dicht besetzten Vektoren und Matrizen, OSKI [264] für Lineare Algebra mit dünn besetzten Vektoren und Matrizen, [80] für die Schnelle Fouriertransformation und [218] für digitale Signalverarbeitung.

2.3.1 Leistungsmaße

Die genannte Definition von Autotuning aus [265] bezieht sich auf ein *performance goal*, das mit *Leistungsziel* übersetzt werden kann. Der Begriff *performance* drückt aus „wie gut oder schlecht etwas arbeitet“[5] [117, S. 1127, s. v. *performance*, 3]. Im Umfeld des High-Performance-Computing ist der Begriff der „Leistung“ eines Algorithmus einer Bedeutungsverengung [6, S. 466] unterlegen und bezieht sich meist auf „Rechengeschwindigkeit“, also durchgeführte Rechenoperationen pro Zeiteinheit [241]. In [55] und [265] wird der Begriff allerdings wieder erweitert und neben der Ausführungszeit als mögliche Leistungsziele beispielhaft genannt: Speicherbedarf, Genauigkeit, Energie-Zeit-Produkt und Näherungsfehler (engl. *approximation error*). Im Folgenden werden die beiden Metriken Ausführungszeit und Energieverbrauch näher beschrieben, die in dieser Arbeit zur Messung der Leistung im Autotuning genutzt werden. Auf die Metrik Näherungsfehler, die auch in verschiedenen Arbeiten verwendet wird [10, 274], soll nicht näher eingegangen werden.

Ausführungszeit

Die meisten traditionellen Benchmarks nutzen die Ausführungszeit als Maß für die Leistung. Die meist entscheidende Frage bei der Einschätzung der Leistungsfähigkeit eines Rechners ist schließlich: Wie schnell führt er eine bestimmte Rechnung durch? Die Benchmarks sind vielfältig und es existieren spezialisierte Benchmarks für alle Arten von Anwendungen, z. B. LINPACK [67] im Hochleistungsrechnen oder SysMark [20] für Büroanwendungen. Neben dem Maß flop/s [71] wird bei Teilchensimulationen auch die Anzahl der Wechselwirkungsberechnungen pro Sekunde [209] verwendet, bei der Simulation neuronaler Netze die Anzahl pro Sekunde verarbeiteter Verbindungen [189] bzw. beim Graph-500-Benchmark die Anzahl der „Kantentraversierungen pro Sekunde“ [193].

5 “how well or badly sth works” [117, S. 1127, s. v. *performance*, 3]

Energieverbrauch

In Analogie zur Ausführungszeit wird gern das Maß flop/J bzw. identisch dazu flop/s/W verwendet, u. a. in [77, 229]. Den Kehrwert J/flop verwendet beispielsweise [255]. Welches Maß man nimmt, ist eine ähnliche Geschmacksfrage wie die, ob der Benzinverbrauch eines Autos in „Meilen pro Gallone“ wie in Amerika oder in „Liter pro 100 Kilometer“ wie in Europa üblich angegeben werden sollte. Nach [155] sei flop/J zu bevorzugen, da der Informatiker gerne die Beziehung „größere Zahl = besseres Ergebnis“ (engl. *“higher is better”*) habe. Für diese Metrik kann also die in Abschnitt 2.1.1 angegebene Formel für die Energieeffizienz verwendet werden.

Außer der reinen Energieeffizienz existiert das *Energie-Zeit-Produkt* (engl. *energy delay*) als Maß, das den Energieverbrauch mit einbezieht. Damit wird versucht, sowohl den Energieverbrauch E als auch die Ausführungszeit t in einer skalaren Größe Θ zu vereinigen [119]. Die Metrik wurde von Horowitz et al. [118, 91] vorgeschlagen und wird in ihrer allgemeinen Form als

$$\Theta(E,t) = E^{\alpha} \cdot t^{\beta} \tag{2.4}$$

mit den Parametern α und β angegeben [119].

Martin entwickelt analog zur Zeitkomplexität eine *Energiekomplexität* für Algorithmen [180]. Er nimmt an, dass sich durch die Veränderung der Frequenz und der Versorgungspannung des Prozessors Ausführungszeit gegen Energieverbrauch eintauschen lässt, vgl. Abschnitt 2.1.3. Das Maß $\Theta(E,t)$ für die Energiekomplexität soll dann folgende Eigenschaften erfüllen:

(i) $\Theta(E,t)$ ist streng monoton wachsend bezüglich E und t.
(ii) $\Theta(E,t)$ ist unabhängig von der Versorgungsspannung des Prozessors.

Für Prozessoren in CMOS-Technologie sind nach Gleichung (2.2) $\alpha = 1$ und $\beta = 2$ gute Werte für die Formel (2.4), also

$$\Theta(E,t) = Et^2 \quad . \tag{2.5}$$

Alle genannten Maße eignen sich für das Leistungsziel im Autotuning. In dieser Arbeit wird jedoch vor allem das erstgenannte Maß, also ein Verhältnis zwischen der Anzahl durchgeführter Berechnungen und der benötigten Energie, verwendet.

2.3.2 Online-Autotuning

Autotuning beruht darauf, ein Leistungsmaß einer Programmimplementierung, z. B. die benötigte Zeit, für mehrere Implementierungsvarianten zu messen und die gewonnenen Informationen für künftige Ausführungen zu nutzen, sodass sich die Leistung verbessert. Während beim Offline-Autotuning die Ausführung der Implementierungsvarianten zum Zeitpunkt der Installation des Programms geschieht, finden beim *Online-Autotuning* diese Messungen im Produktivbetrieb der Anwendung statt. Bei jeder Ausführung wird eine Implementierungsvariante, ggf. auch mit nicht-optimalen Parametern untersucht [250].

Suda nennt mehrere Vorteile des Online-Autotunings gegenüber dem Offline-Autotuning [250]: Neben dem Umstand, dass die für die Installation benötigte Zeit sinkt, werden beim Online-Autotuning häufig genutzte Routinen besonders gut untersucht. Dadurch verbessert sich insbesondere das Verhalten dieser Routinen besonders stark, während der Tuning-Aufwand für selten genutzte Rou-

tinen sinkt. Außerdem passt sich so das Autotuning an die am häufigsten genutzten Eingabedaten an. Als Nachteil sieht Suda an, dass der Overhead der Messung die Ausführung der Anwendung verlangsamen könnte.

Eine Analyse mathematischer Methoden für das Online-Autotuning wird in [250] gegeben. In [251] wird sie erweitert um eine Modellierung der Leistungsaufnahme der Rechner unter Einbezug der Temperatur.

Lu et al. [172] erzeugen anhand eines analytischen Modells mehrere Varianten für die Matrixtransponierung. Diese werden empirisch untersucht und die zwei besten werden behalten: eine mit Lesepuffer, eine ohne. Abhängig von den Eigenschaften der Eingabedaten (Vektorgröße, Ausrichtung im Speicher) wird dann zur Laufzeit eine der beiden Varianten gewählt.

2.3.3 Energie-Autotuning

In den letzten Jahren kamen die ersten Ansätze zur Verwendung des Energieverbrauchs als Leistungsmaß im Autotuning auf: Tiwari et al. [254] untersuchen eine Methode zur Lösung der Poisson-Gleichung. Sie nutzen das *Active-Harmony*-Autotuning-Framework [64] zur Erzeugung von Codevarianten und zur Suche des Optimums. Sie verwenden Energie-Zeit-Metriken mit verschiedenen Exponenten und kommen zu dem Ergebnis, dass die energieeffizienteste Implementierungsvariante ihrer Methode 5,4 % Energie spart und die Ausführungszeit um 4 % verlängert.

Wang et al. [267] nutzen die *Polyhedral Compiler Collection*, um von ihrer Anwendung aus dem Bereich der Hydrodynamik verschiedene Codevarianten durch Transformationen von Schleifen zu erzeugen, z. B. durch Ausrollen, Fusionieren, Vektorisieren oder Parallelisieren. Für alle erzeugten Varianten werden die Ausführungszeit und der Energieverbrauch auf dem Xeon Phi und auf einer Sandy-Bridge-CPU gemessen. In der Regel war über alle Varianten die Ausführungszeit proportional zur Energie. Lediglich eine Variante stach hervor, die zwar die Leistungsaufnahme erhöhte, dafür aber deutlich kürzere Ausführungszeiten und damit einen geringeren Energieverbrauch erzielte.

Studien von Meyer et al. [186, 187] vergleichen zwei Abspeicherungsformate für dünn besetzte Matrizen und die Ausführungszeit bzw. den Energieverbrauch mit diesen Formaten bei der CGM. Sie kommen zu dem Ergebnis, dass es zwischen den Größen keinen einfachen Zusammenhang gibt. Außerdem stellen sie fest, dass sich vorhandene Methoden zur Energieverbrauchsmessung nicht für Online-Autotuning eignen. Ebenfalls mit Routinen der Linearen Algebra, jedoch für dicht besetzte Matrizen und Vektoren, beschäftigt sich [219]. Für die Suche nach dem Optimum kann hier angegeben werden, mit welchem Gewicht Ausführungszeit und Leistungsaufnahme in die Bewertung eingehen sollen. So kann der Energieverbrauch gesenkt werden ohne die Ausführungszeit übermäßig zu verlängern.

Katagiri et al. [143] erweitern das vorhandene Autotuning-Framework *ppOpen-AT* [144] um eine Möglichkeit zur Bewertung der erzeugten Codevarianten anhand des Energieverbrauchs. Untersucht werden CPU- und GPU-Implementierungen einer Anwendung zur Simulation der Vorgänge beim Erstarren flüssigen Aluminiums. Die Implementierung auf der GPU arbeitet leicht energieeffizienter, verliert jedoch durch den notwendigen Datentransfer ihren Vorsprung, wenn die Anzahl der Iterationen mit denselben Daten nicht groß genug ist.

Miceli et al. [188] stellen ein Framework zum Autotuning vor, in dem Plug-ins für die Exploration des Suchraums und das Testen verschiedener Varianten zuständig sind. Eines dieser Plugins berücksichtigt den Energieverbrauch mit RAPL. Ansätze zum Energie-Autotuning für IT-Infrastrukturen werden in [104] und [103] diskutiert.

2.3.4 Modellbasiertes Autotuning

Yotov et al. werfen in [280] die Frage auf, ob die Suche der minimalen Ausführungszeit beim Autotuning wirklich notwendig sei. Werde ein Modell für die Ausführungszeit unter Berücksichtigung aller Parameter, die bei der Codeerzeugung angewandt wurden, aufgestellt, könne das Optimum analytisch ermittelt werden. Damit könne die erschöpfende Suche eingespart werden, die durch das Ausführen aller erzeugten Codevarianten sehr zeitaufwändig wird. Bereits in [279] hatten Yotov et al. für ATLAS gezeigt, dass man auf analytischem Wege bis auf ca. 20 % an das Optimum herankommt, das durch erschöpfende Suche gewonnen wurde. Die von ATLAS erzeugten Codevarianten lassen sich als übliche Compilertransformationen auffassen, für die analytische Modelle existieren [146]. In das Ausführungszeitmodell für die Codevarianten fließen Parameter wie die Größe des Level-1-Caches, die Latenz der CPU und die Verfügbarkeit eines Multiply-Add-Befehls ein. Es wird gezeigt, dass sich durch Nutzen des Modells wesentlich schneller ein Optimum finden lässt als durch Durchprobieren aller Codevarianten.

Nach demselben Prinzip stellen Choi et al. [53] ein Ausführungszeitmodell für die Multiplikation einer dünn besetzten Matrix mit einem dünn besetzten Vektor auf einer GPU auf. Die Methode misst zum Installationszeitpunkt verschiedene Werte und bezieht Hardware-Eigenschaften mit ein, unter anderem die Anzahl der Register, die Größe des Speichers, die Anzahl Warps pro Streaming-Multiprozessor und die Größe eines Warps. Die durch die modellbasierte Methode gefundenen Parameterkonfigurationen ergaben eine maximal um 15 % längere Ausführungszeit als die durch erschöpfende Suche gefundenen Konfigurationen. Das in [124] vorgestellte Modell bestimmt die optimale Blockgröße für Rechenoperationen mit dünn besetzten Matrizen anhand einer Näherung für die Rechenleistung der CPU und einer Approximation der Anzahl „nutzloser“ Berechnungen. „Nutzlose“ Berechnungen entstehen dadurch, dass Teile der dünn besetzten Matrix mit Nullen aufgefüllt werden. Chung und Hollingsworth [55] speichern die Charakteristika aller Ausführungen von Funktionen ihrer Anwendungen in einer Datenbank ab. Bei erneutem Ausführen der Funktion kann anhand der historischen Daten eine Konfiguration ermittelt werden, die eine möglichst gute Ausführungszeit bringt. Fehlende Informationen werden hier durch Inter- bzw. Extrapolation ermittelt.

Yotov et al. entwickeln den in [279] und [280] vorgestellten Ansatz des *rein modellbasierten Autotunings* für ATLAS weiter zu *modellbasiert-empirischem Autotuning* [281]. Mit dem aufgestellten analytischen Modell wird eine Lösung ermittelt, für die erwartet wird, dass sie nahe dem Optimum liegt. Durch empirisches Suchen in der Umgebung dieser Lösung kann eine nahezu optimale Lösung ermittelt werden [281]. Mithilfe des Modells wird beim modellbasiert-empirischem Autotuning der Suchraum stark eingeschränkt, sodass die Suche beschleunigt wird. Im Vergleich zum reinen modellbasierten Autotuning kann so an den am meisten Erfolg versprechenden Stellen genauer gesucht werden [281, 53]. Bei rein empirischer Suche kann bei vielen Parametern, die Einfluss auf die Leistung haben könnten, der Suchraum so groß werden, dass die erschöpfende Suche

impraktikabel wird [61].

Nelson et al. [198] können mit einem ähnlichen modellbasiert-empirischen Ansatz den Suchraum bei einer Anwendung zur Visualisierung molekulardynamischer Prozesse auf 0,3 bis 5 % einschränken und eine Lösung finden, die nur 15 % schlechter als die optimale Lösung ist. Ihre Methode ermöglicht es dem Anwendungsentwickler, dem Autotuning-Werkzeug Modelle und Informationen über Parameter zur Verfügung zu stellen, die seiner Meinung nach besonders großen Einfluss auf die Ausführungszeit haben.

Davidson et al. vergleichen ebenfalls den rein modellbasierten und den modellbasiert-empirischen Ansatz [61]. Die Anwendung ist hier die Lösung großer tridiagonaler Gleichungssysteme, das in verschiedenen Phasen verschiedene Lösungsalgorithmen verwendet, je nach dem Parallelitätsgrad, den die Daten ermöglichen. Für den Wechsel von einem Algorithmus zum nächsten muss ein passender Zeitpunkt bestimmt werden. Sie zeigen, dass die rein modellbasiert ermittelte Lösung durch empirische Suche in ihrer Umgebung noch weiter verfeinert werden kann.

Chen et al. [49] entwickeln den Ansatz weiter und nutzen analytische Modelle für Compilertransformationen, um nur Erfolg versprechende Codevarianten für ATLAS zu erzeugen. Unter diesen Varianten wird dann durch testweise Ausführung die beste gesucht. Als zweites Anwendungsbeispiel demonstrieren Chen et al. ihre Methode an der Jacobi-Relaxation. Ähnlich wird auch in [168], [172] und [30] vorgegangen. In [30] wird das Modell automatisch basierend auf nicht-linearer Regression aufgestellt.

2.4 Anwendungsspezifische Ausführungszeit- und Energiemodelle

Die Modellierung von Ausführungszeiten hat eine lange Tradition im wissenschaftlichen Rechnen [46, 236]. Zahlreiche Methoden zur Modellierung wurden entwickelt. Zur automatisierten Modellierung und Vorhersage der Ausführungszeit paralleler Algorithmen existieren Frameworks wie PACE [202] oder Prophesy [253]. Deutlich weniger Untersuchungen gibt es allerdings bislang zu Energiemodellen.

Generell ist bei den Modellen zwischen *rechnerspezifischen Modellen* und *Algorithmen-* bzw. *Anwendungsmodellen* zu unterscheiden: Rechnerspezifische Modelle beschreiben einen bestimmten Rechner, eine Klasse von Rechnern oder eine Rechnerkomponente. Über Parameter werden Charakteristika des ausgeführten Algorithmus festgelegt. Algorithmenspezifische Modelle hingegen beschreiben einen bestimmten Algorithmus und lassen sich durch Parameter auf bestimmte Rechner sowie Eingabedaten anpassen. Rechnerspezifische Energiemodelle existieren zum Beispiel für folgende Systeme: für Shared-Memory-Systeme [156], für einen BlueGene/L-artigen Supercomputer [177], für Parallelrechner mit aus parallelen und sequentiellen Abschnitten bestehenden Anwendungen [52], für Parallelrechner mit aus Fork-Join-Tasks bestehenden Anwendungen [223], für GPUs [226], für den Xeon Phi [240], für DRAM [190]. In der Regel berücksichtigen die Energiemodelle die Spannungs- und Frequenzregelung der entsprechenden Geräte.

In dieser Arbeit werden hingegen anwendungsspezifische Modelle aufgestellt, die auch direkt in der Anwendung verwendet werden. Der Vorteil von anwendungsspezifischen gegenüber rechnerspezifischen Modellen ist, dass Anwendungscodes häufig über 20 Jahre oder noch länger genutzt werden [174], sodass das Modell über diesen Zeitraum beibehalten werden kann. Lediglich bei

grundlegend neuen Hardwarearchitekturen, die eine Anpassung des Anwendungscodes erforderlich machen, muss auch das Modell überarbeitet werden. Hardware hat hingegen eine regelmäßige Nutzungsdauer von etwa 4 Jahren [174].

Auch die Beispiele für anwendungsspezifische Modelle sind vielfältig: So sind zum Beispiel am Los Alamos National Laboratory über Jahre anwendungszentrierte Ausführungszeitmodelle für dort benötigte Anwendungen entstanden [147, 115, 148]. Sie werden im gesamten Lebenszyklus der Anwendungen eingesetzt, u. a. während des Entwurfs neuer Rechnersysteme, zur Untersuchung von Optimierungsmöglichkeiten von Anwendungen vor deren Optimierung und zur Evaluation künftiger Rechnerarchitekturen [149]. In [227] und [11] wird ein lineares Modell zur Vorhersage der Ausführungszeit von Hartree-Fock- und anderen Rechnungen in einem quantenchemischen Code vorgestellt. [79] sagt für die Multiplikation dünn besetzter Matrizen und Vektoren die Häufigkeit von Level-1 und Level-2-Cache-Misses vorher, die einen wesentlichen Einfluss auf die Ausführungszeit haben.

Energiemodelle für Anwendungen werden beispielsweise für energiebewusstes Scheduling benötigt [260]. Ein Ausführungszeit- und Energiemodell für das Lineare-Algebra-Paket LINPACK wird in [248] automatisiert mit Regressionsanalyse aufgestellt.

Im Folgenden wird auf existierende anwendungsspezifische Energie- und Ausführungszeitmodelle für die in dieser Arbeit betrachteten Anwendungen, die Schnelle Multipolmethode und die Methode der konjugierten Gradienten, eingegangen.

Schnelle Multipolmethode In [95] wird ein Modell für die Ausführungszeit der parallelen FMM in Abhängigkeit von der Anzahl der Teilchen, der Anzahl der Prozessoren und Anzahl der Zellen auf der Blattebene aufgestellt. Außerdem werden 4 Konstanten für die Gleitkommageschwindigkeit des Rechners und die geforderte Genauigkeit des Ergebnisses einbezogen. Eine Betrachtung tatsächlich benötigter Ausführungszeiten auf realen Rechnern erfolgt jedoch nicht. Während [95] von einer homogenen Teilchenverteilung ausgeht, wird das Modell in [57] um eine Betrachtung der Eingabedaten erweitert, sodass es auch für inhomogene Verteilungen geeignet ist. In [57] wird zudem die Rechenlast, die von jedem Knoten des FMM-Baums erzeugt wird, abgeschätzt, um zu einer guten Verteilung der Rechenlast zwischen den Prozessoren, der Kommunikationslast und der Speichernutzung zu gelangen. Jedoch werden auch in [57] keine tatsächlichen Ausführungszeiten einbezogen.

Methode der konjugierten Gradienten In [112] wurde die Ausführungszeit der auch in dieser Arbeit genutzten FEM für verschiedene SMP-Rechner *symmetric multiprocessing machine* untersucht. Das Aufstellen eines formalen Ausführungszeitmodells war jedoch nicht Gegenstand jenes Artikels.

In [157] wird untersucht, welches der Abspeicherungsformate für dünn besetzte Matrizen – CRS *(compressed row storage)*, JDS *(jagged diagonal storage)*, ELLPACK und SS *(segmented scan)* – die kürzeste Ausführungszeit der CGM ermöglicht. Es wird festgestellt, dass die Dispersion, also das Verhältnis zwischen maximalem und durchschnittlichen Besetzungsgrad einer Matrixzeile, und der Besetzungsgrad, also der Anteil der Nichtnullelemente, der gesamten Matrix die entscheidenden Parameter für die Auswahl des optimalen Abspeicherungsformats für eine gegebene Matrix sind. Anhand empirischer Daten wird ein Entscheidungsbaum für die Auswahl des Abspeicherungsformates anhand dieser Parameter aufgestelllt. Ein explizites Modell, das die Vorhersage der Ausfüh-

rungszeit ermöglicht, wird allerdings nicht formuliert.

Bei der CGM benötigt die Matrix-Vektor-Multiplikation den größten Teil der Ausführungszeit des gesamten Verfahrens [278]. Vázquez et al. [266] stellen ein Modell für die Anzahl notwendiger Speicherzugriffe für die Multiplikation dünn besetzter Matrizen mit Vektoren auf GPUs auf. Das Modell bezieht GPU-Parameter wie Warp-Größe und Anzahl der Streaming-Multiprozessoren mit ein, um die Algorithmenparameter der Threads pro Element des Ergebnisvektors und die Größe der CUDA-Blöcke zu ermitteln. Durch die Verschränkung von Speicher- und Rechenoperationen fallen die Rechenoperationen nicht ins Gewicht und die Anzahl der Speicheroperationen ist etwa proportional zur Ausführungszeit. Verschoor und Jalba [263] stellen ein Ausführungszeitmodell für die CGM auf mehreren GPUs auf. Das Modell beruht hauptsächlich auf dem Datendurchsatz als Maß für die Leistungsfähigkeit, da die CGM speicherbeschränkt ist. Es berücksichtigt die Größe des linearen Gleichungssystems und die Anzahl der Nichtnullelemente in der Matrix als Datenparameter. Die GPU-Parameter werden über einen Fit der Ausführungszeiten der einzelnen Unterroutinen ermittelt.

Yang und Lin [275] stellen ein Modell für die parallele Effizienz für ein CGM-Problem auf, das als Problem der kleinsten Quadrate implementiert wurde, was eine äquivalente Formulierung ist [246]. Genauer gesagt wird der Kommunikationsoverhead, der durch die Parallelisierung entsteht, in Relation zur sequentiellen Ausführungszeit betrachtet. Es wird die Kommunikationszeit für das Skalarprodukt sowie die Kommunikationszeit für das Matrix-Vektor-Produkt für zwei verschiedene Partitionierungsschemata in Abhängigkeit von der Systemgröße und der Anzahl der Prozessoren modelliert.

2.5 Zusammenfassung

Zwei der wichtigsten Herausforderungen im wissenschaftlichen Rechnen sind derzeit die Energieeffizienz im Hochleistungsrechnen und die Programmierbarkeit komplexer paralleler Rechner. Die Energieeffizienz ist sowohl auf Hardware- als auch auf Softwareebene zu verbessern. Mit „Energieeffizienz“ ist in dieser Arbeit die Anzahl durchgeführter Operationen pro Energieeinheit gemeint. Die Energieeffizienz einer Anwendung lässt sich beispielsweise durch das Verwenden dynamischer Frequenz- und Spannungsregelung oder durch die Nutzung von Beschleunigern wie Grafikprozessoren steigern. Um die Effektivität der angewandten Methoden evaluieren zu können, ist eine Messung des Energieverbrauchs unerlässlich. Ideal für die Energiemessung direkt in Anwendungen sind zum Beispiel die RAPL-Technik bei modernen Intel-CPUs oder die Möglichkeit über die NVML in Nvidia-GPUs.

Die Programmierung komplexer, z. T. auch heterogener Rechnersysteme wird durch Autotuning stark vereinfacht. Vorhandene Software kann ohne großen Aufwand auf neue Hardware angepasst werden, indem nicht der Nutzer alle Varianten der Implementierung selbst durchprobiert, sondern den Vorgang automatisiert. Neben der Minimierung der Ausführungszeit kann Autotuning auch als Ziel haben, den Energieverbrauch zu minimieren bzw. die Energieeffizienz zu maximieren. Erste Literatur, die dazu bereits existiert, kommt häufig zu dem Schluss, dass die Maximierung der Energieeffizienz auch gleichzeitig die Ausführungszeit minimiert. In einigen Fällen divergieren diese Ziele jedoch. Beim Online-Autotuning wird das Suchen der optimalen Implementierung nicht

wie beim Offline-Autotuning zum Zeitpunkt der Installation der Anwendung auf einem Rechner durchgeführt, sondern im Produktivbetrieb während der Ausführung mit konkreten Daten. Damit kann sich das Autotuning auf die Eingabedaten beziehen und so die Ausführung mit genau diesen Daten optimieren. Besonders beim Online-Autotuning ist es von Vorteil, wenn das Verfahren modellbasiert arbeitet. Hier beschreibt ein Modell die Veränderung der Zielgröße des Autotunings in Abhängigkeit bestimmter Parameter. So kann das Optimum entweder direkt analytisch bestimmt oder zumindest der Suchraum der Optimierung stark eingeschränkt werden. Das verringert die Anzahl notwendiger Testläufe mit verschiedenen Implementierungsvarianten und verkürzt so die für das Tuning benötigte Zeit.

Ausführungszeitmodelle für wissenschaftliche Anwendungen werden schon seit langer Zeit genutzt. Auch für die hier untersuchten Anwendungen, die FMM und die CGM existieren solche Modelle. Seltener waren bislang Energiemodelle für Anwendungen des wissenschaftlichen Rechnens Gegenstand der Forschung. ■

3 Teilchensimulation mit der Schnellen Multipolmethode

Teilchensimulationen vollziehen das räumliche und zeitliche Verhalten einer großer Anzahl miteinander wechselwirkender Teilchen nach [78]. Sie arbeiten in diskreten Zeitschritten. In jedem Zeitschritt werden die zwischen den Teilchen wirkenden Kräfte und die daraus resultierenden Bewegungsvektoren der Teilchen berechnet. Diese Kraftberechnung entspricht dem *n-Körper-Problem*, also der Berechnung der Wechselwirkungen zwischen n Körpern im Raum, zwischen denen eine konservative Kraft wirkt. Beispiele für n-Körper-Probleme sind die Wechselwirkungen von Sternen in Galaxien, die Wechselwirkungen zwischen Atomen innerhalb großer Proteinmoleküle, die u. a. durch Faltung entstehen, oder die Wechselwirkungen zwischen Ionen in einem Plasma [78]. Anhand der für jedes Teilchen berechneten Kräfte werden oft makroskopische Größen wie potentielle Energie, kinetische Energie, Druck oder Diffusionskonstante berechnet [98, S. 10].

Im n-Körper-Problem wirkt von jedem der n Teilchen eine Kraft zu den $n-1$ anderen Teilchen des Systems. Nach dem 3. Newtonschen Gesetz sind dabei die Kraft in der Hin- und in der Rückrichtung betragsmäßig gleich [199, S. 23], sodass $\frac{1}{2}(n^2-n)$, also $O(n^2)$, Wechselwirkungen zu berechnen sind. In großen Teilchensimulationen werden 70 bis 90 Prozent der Ausführungszeit für die Berechnung der Wechselwirkungen aufgebracht [78]. Daher sind effiziente Methoden für diese Berechnung notwendig.

Bei sehr kurzreichweitigen Potentialen wie dem Lennard-Jones-Potential zur Berechnung von Wechselwirkungen zwischen Molekülen [140] lässt sich die Rechenzeit entscheidend verringern, indem ein Cutoff-Radius gesetzt wird. Das heißt, das Potential eines Teilchens wird ab einem bestimmten Abstand auf Null gesetzt, sodass außerhalb dieses Radius keine Wechselwirkungen mit diesem Teilchen mehr stattfinden. Bei langreichweitigen Potentialen wie Coulomb-Potential oder dem Gravitationspotential ist ein solches Vorgehen aufgrund des großen Fehlers, den es verursachen würde, nicht möglich [76, 277].

Die *Schnelle Multipolmethode* (engl. *Fast Multipole Method, FMM*) [96] erzielt für homogene Teilchenverteilungen eine Laufzeit in $O(n)$ [9], indem sie die Wechselwirkungen aus kurzreichweitigen *Nahfeldwechselwirkungen* und langreichweitigen *Fernfeldwechselwirkungen* zusammensetzt und für diese verschiedene Berechnungsverfahren nutzt. Die FMM untergliedert den Raum wie in Abb. 3.1 dargestellt hierarchisch in Zellen, die in einem Baum organisiert sind. Die Teilchen werden in diese Zellen einsortiert. Wechselwirkungen zwischen benachbarten Zellen der Blattebene werden als Nahfeldwechselwirkungen weiterhin direkt mit einer Laufzeit in $O(n^2)$ berechnet. Zur Berechnung der Wechselwirkung zwischen weiter auseinander liegenden Zellen werden die Teilchen einer Zelle zu *Pseudoteilchen* zusammengefasst. Die Fernfeldwechselwirkungen werden nun zwischen den Pseudoteilchen berechnet. Das verwandte Barnes-Hut-Verfahren [22] erreicht

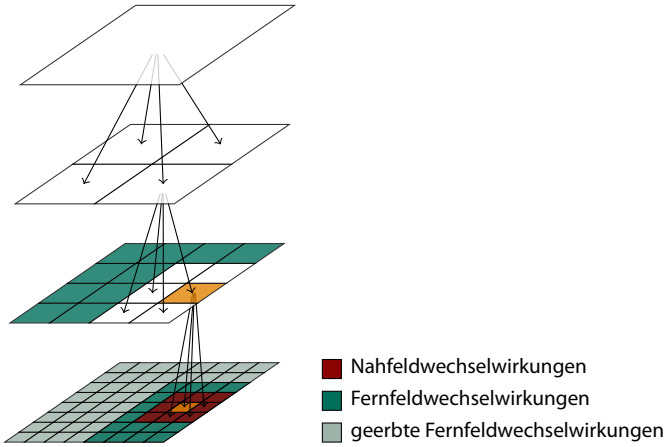


Abbildung 3.1: FMM-Baum der Tiefe 3 mit Wechselwirkungen für die zweidimensionale FMM

so eine Ausführungszeit in $O(n \log n)$. Die FMM erzeugt nun zusätzlich noch Pseudoteilchen aus Pseudoteilchen und berechnet damit die Wechselwirkungen zwischen Zellen, sodass sie eine Ausführungszeit in $O(n)$ erreicht [98, S. 388]. Kindzellen erben dabei die Beiträge der Wechselwirkungen der jeweiligen Elternzelle. Bei dem aufgebauten Baum, dem *FMM-Baum*, handelt es sich im zweidimensionalen Fall wie in Abb. 3.1 um einen Quadtree, im dreidimensionalen Fall um einen Octree.

In diesem Kapitel wird ein Verfahren zum automatisierten Tuning einer FMM-Implementierung vorgestellt, das für konkrete Eingabedaten die Baumtiefe mit der niedrigsten Ausführungszeit ermittelt. Grundlage ist [58], das in einem zweistufigen Fehlersteuerungsverfahren feststellt, nach wie vielen Gliedern die Multipol-Expansion zur Erzeugung der Pseudoteilchen abgebrochen werden kann, um einen vorgegebenen Fehler im Endergebnis nicht zu überschreiten. Dieses Verfahren zählt in einem ersten Durchlauf die Anzahl der benötigten arithmetischen Operationen und berechnet unter Einbezug spezifischer Kosten für die Operationen Addition, Multiplikation, Division und Radizierung eine Schätzung für die Ausführungszeit bei verschiedenen Baumtiefen. Die spezifischen Kosten sind fest im Quelltext einprogrammiert und drücken aus, das Wievielfache der Ausführungszeit einer Addition bzw. Multiplikation eine Division bzw. eine Radizierung benötigt. Außerdem geht eine Konstante für den Verwaltungsoverhead, der für das Traversieren der Baumdatenstruktur etc. notwendig ist, mit in die Schätzung ein. Für die Durchführung der Wechselwirkungsberechnung wird dann die Baumtiefe gewählt, für die die niedrigste Ausführungszeit ermittelt wurde. Das Problem bei der Ausgangsmethode ist, dass der Verwaltungsoverhead als konstant angenommen wird, was er aber nicht ist. Außerdem ist es für jeden neuen Rechnertyp notwendig, die spezifischen Kosten der arithmetischen Operationen sowie des Verwaltungsoverheads per Hand zu ermitteln, um eine präzise Schätzung zu erhalten.

Der Beitrag dieses Kapitels ist es, das manuelle Tuning auf Autotuning umzustellen und zu zeigen, dass das dem [58] inhärente Kostenmodell für die Ausführungszeit des Verfahrens verbessert werden kann, indem man reale Ausführungszeiten, die auch die Variation beim Verwaltungsoverhead einbeziehen, nutzt statt der Näherung durch die Zählung der Rechnenoperationen. Das vor-

gestellte Verfahren wird außerdem genutzt, um neben der Ausführungszeit auch den Energieverbrauch der Ausführung minimieren. Das Leistungsziel des Autotunings ist die Minimierung der Kosten, die von der Ausführungszeit oder dem Energieverbrauch dargestellt werden können.

3.1 Schnelle Multipolmethode

Die FMM wird benötigt, um einen Zeitschritt in einer Teilchensimulation zu berechnen, also die Wechselwirkungen zwischen allen Teilchen im System. Die folgende Beschreibung erläutert den Algorithmus anhand der genutzten Implementierung der FMM in Fortran, die unter [237] veröffentlicht ist. Die Beschreibung folgt dabei den Beschreibungen in [59], und z. T. auch [141], die auf dem gleichen Quelltext basieren.

Alle Teilchen liegen in einem Einheitswürfel im dreidimensionalen Raum, der *Systemzelle*. Die FMM wird zur Berechnung der Gesamtenergie der wechselwirkenden Teilchen benutzt. Die Hauptdatenstruktur der FMM ist ein Octree, also ein Baum, bei dem jeder innere Knoten 8 Kindknoten besitzt. Die Knoten des FMM-Baumes stellen jeweils Zellen, die von der hierarchischen Aufteilung des Raumes herrühren, dar. Der Wurzelknoten auf Ebene 0 stellt die Systemzelle dar. Die Knoten der Ebene 1 repräsentieren die 8 *Kindzellen* der Systemzelle, die entstehen, wenn die Systemzelle in jeder Dimension einmal halbiert wird. Diese Unterteilung wird rekursiv mit allen erzeugten Zellen fortgeführt, bis eine bestimmte Ebene d, deren Knoten die *Blattzellen* repräsentieren, erreicht ist. Jede Ebene l des Octree besteht also aus 8^l Zellen. Mit d wird die *Tiefe des Baumes* bezeichnet.

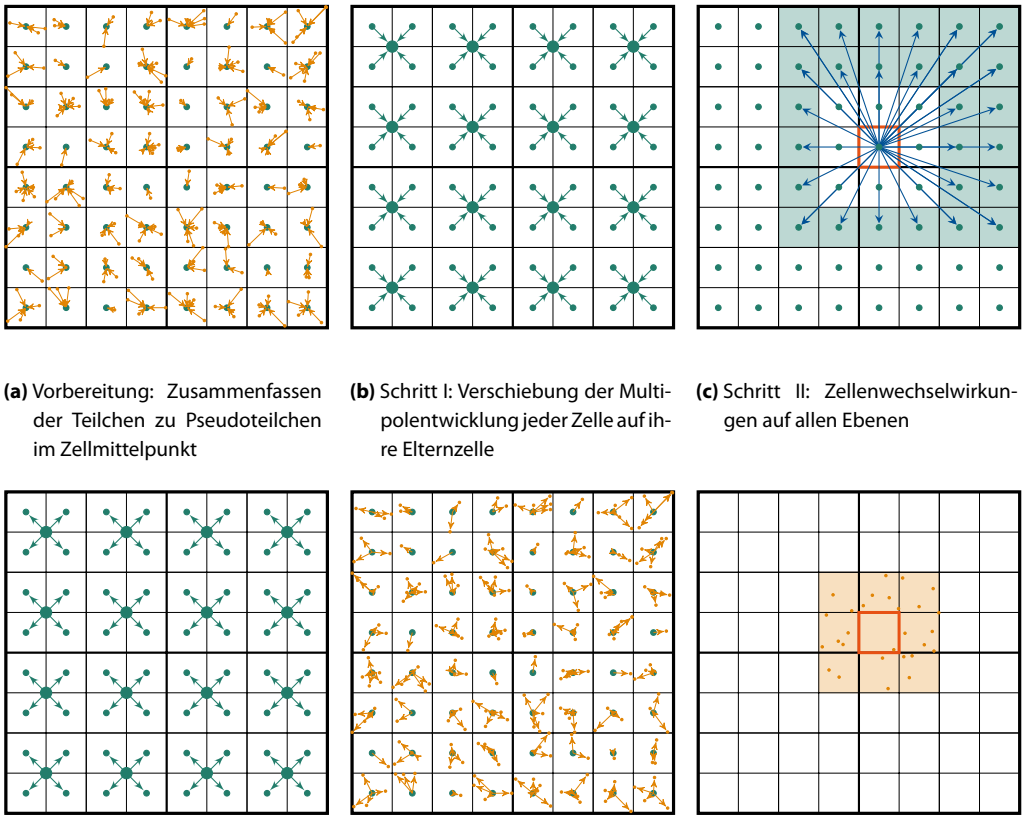
Eingabe der FMM sind eine Teilchenverteilung und ein geforderter Fehler für die zu berechnende Gesamtenergie. Ausgabe ist die Gesamtenergie E. Zu Beginn der Berechnung wird zunächst der FMM-Baum mit der Tiefe d aufgebaut. Die Wahl des Wertes für d wird in Abschnitt 3.2.1 erläutert. Anschließend werden *Pseudoteilchen* erzeugt: Alle Teilchen innerhalb einer Blattzelle werden so behandelt, als seien sie in einem Pseudoteilchen konzentriert, welches sich im Mittelpunkt der Zelle befindet, vgl. Abb. 3.2a. Das Pseudoteilchen wird über eine *Multipol-Expansion* erzeugt. Die Länge der Multipolexpansion, d. h. die Anzahl m der verwandten Multipolmomente, beeinflusst die Genauigkeit des Ergebnisses und die Rechenzeit der FMM. Der Wert für m wird über das in [58] beschriebene Verfahren festgelegt.

Nun wird der FMM-Algorithmus in den folgenden 5 Schritten durchgeführt:

I. Beginnend mit den Blattknoten und endend mit den Knoten der Ebene 2 werden jeweils die Multipolentwicklungen der Kindzellen auf deren Elternzellen verschoben, um dort erneut Pseudopartikel zu erzeugen. Dieser Vorgang wird in Abb. 3.2b dargestellt.

II. Der zweite Schritt berechnet die Wechselwirkungen zwischen den Zellen, also die *Zellenwechselwirkungen*, unter Beachtung des Kriteriums der *Wohlseparierthеit*: Zwei Zellen sind genau dann wohlsepariert, wenn sie nicht benachbart sind, d. h. sie stoßen weder an einer Kante noch an einer Ecke aneinander. Die Wechselwirkungen werden von jeder Zelle z_1 mit jeder Zelle z_2 ausgeführt, die den folgenden Bedingungen genügt:

   1. z_1 und z_2 liegen auf der gleichen Ebene,
   2. z_1 und z_2 sind wohlsepariert und
   3. die Elternzelle von z_1 und die Elternzelle von z_2 sind nicht wohlsepariert.



(a) Vorbereitung: Zusammenfassen der Teilchen zu Pseudoteilchen im Zellmittelpunkt

(b) Schritt I: Verschiebung der Multipolentwicklung jeder Zelle auf ihre Elternzelle

(c) Schritt II: Zellenwechselwirkungen auf allen Ebenen

(d) Schritt III: Verschiebung der lokalen Entwicklung auf die Kindzellen

(e) Schritt IV: Auswertung der lokalen Entwicklung jeder Zelle

(f) Schritt V: Wechselwirkungen mit den Teilchen der Nachbarzellen

Abbildung 3.2: Schritte der FMM

Abbildung 3.2c zeigt, mit welchen Zellen die fett umrandete Zelle wechselwirkt. Jede dieser Wechselwirkungen wird durchgeführt, indem die Multipolentwicklung der Quellzelle in eine lokale Entwicklung der Zielzelle verschoben wird. Dieser Schritt wird für alle Zellen von der Blattebene bis zu Ebene 2 durchgeführt. Die Zellen in den Ebenen 0 und 1 wechselwirken hier nicht.

III. Im dritten Schritt werden die Beiträge der Wechselwirkungen zwischen den Zellen an die Blattzellen verteilt. Dazu werden die lokalen Entwicklungen der Elternzellen jeweils in die Mittelpunkte ihrer Kindzellen verschoben und dort zu den vorhandenen lokalen Entwicklungen addiert. Die gesamte Verschiebung beginnt in Ebene 2 und endet in der Blattebene. Der Vorgang wird für die Verschiebung von Ebene 2 auf Ebene 3 in Abb. 3.2d dargestellt.

IV. Die in den Schritten I bis III berechneten Beiträge der Fernfeldwechselwirkungen zur Gesamtenergie werden berechnet, indem die lokale Entwicklung in jeder Blattzelle ausgewertet wird,

s. Abb. 3.2e.

V. Die Beiträge der Nahfeldwechselwirkungen zur Gesamtenergie werden durch direkte Berechnung der Wechselwirkungen zwischen Teilchen ermittelt. Diese *Teilchenwechselwirkungen* werden zwischen allen Teilchen innerhalb einer Zelle der Blattebene und zwischen Teilchen in Zellen der Blattebene, die nicht wohlsepariert sind, durchgeführt. In Abb 3.2f sind die Zellen der Teilchen, mit denen die Teilchen der fett umrandeten Zelle wechselwirken, orange unterlegt.

3.2 Autotuning durch Anpassung der Baumtiefe

In diesem Abschnitt wird beschrieben, wie die Baumtiefe d für den FMM-Baum so gewählt werden kann, dass die entstehenden Kosten möglichst niedrig bleiben. Als Kosten können hier die Ausführungszeit gemessen in Takten bzw. in Maschinenbefehlen oder der Energieverbrauch verstanden werden. Zunächst wird die vorhandene Methode vorgestellt, die die Kosten basierend auf der Anzahl der arithmetischen Operationen und deren spezifischen Kosten ermittelt. Die spezifischen Kosten geben das Verhältnis zwischen den Ausführungszeiten der verschiedenen arithmetischen Operationen an. Die Gesamtkosten werden für jede Baumtiefe für die konkreten Eingabedaten ermittelt und daraus die Baumtiefe mit den niedrigsten Kosten ausgewählt. Darauf aufbauend wird ein Autotuning-Verfahren vorgestellt, das die Kosten auf dem konkret genutzten Rechner ermittelt, um so eine höhere Genauigkeit zu erzielen. Das Modell wird zunächst mit der Ausführungszeit als Maß für die Kosten entworfen. Später wird es auf den Energieverbrauch übertragen.

3.2.1 Ausgangssituation

Zunächst wird die Methode der automatischen Anpassung der Baumtiefe, wie sie für [58] am Forschungszentrum Jülich implementiert wurde, vorgestellt. Die Methode wurde von Dachsel in [58] bereits angedeutet und in [59] genauer beschrieben.

Zu Beginn wird der Baum mit einer gewissen maximalen Tiefe $d_{\max}$ aufgebaut und die Teilchen in die Zellen einsortiert. Vor der eigentlichen Berechnung der Wechselwirkungen im *Berechnungslauf* wird jedoch für das in [58] beschriebene Fehlersteuerungsverfahren ein *Analyselauf* durchgeführt. In diesem Analyselauf wird der zu erwartende Fehler in Abhängigkeit von der Länge der Multipolentwicklung abgeschätzt, indem für jedes Teilchen anhand seines Abstandes zum Zellmittelpunkt sein Beitrag zum Gesamtfehler ermittelt wird. Der Baum wird im Analyselauf genau so wie auch im Berechnungslauf traversiert. Lediglich die Berechnungen der Wechselwirkungen werden durch Anweisungen zum Erhöhen zweier Zähler für die Zellen- und die Teilchenwechselwirkungen ersetzt. Die Werte dieser Zähler werden für jede Baumtiefe $d \leq d_{\max}$ ermittelt.

Zusätzlich zu den problemabhängigen Zählern werden die rechnerabhängigen Kosten für die Berechnung einer Teilchen- bzw. einer Zellenwechselwirkung benötigt, die *Kostenparameter*. Der Kostenparameter für die Zellenwechselwirkung hängt außerdem von der Länge der Multipolentwicklung ab, also vom Fehler, den der Benutzer für das Ergebnis vorgibt. Die Kostenparameter werden bestimmt anhand der Anzahl der zur Berechnung notwendigen arithmetischen Operationen Addition, Multiplikation, Division und Radizierung sowie deren spezifischen Kosten. Die

spezifischen Kosten einer arithmetischen Operation werden als Vielfaches der Kosten für eine Addition bzw. Multiplikation angegeben. Für ein willkürlich gewähltes System betragen die Kosten für eine Multiplikation `ndiv = 4` sowie für eine Radizierung `nsqr = 31` [237, Datei `master/lib/fmm/src/fmm.f`]. Diese Konstanten wurden per Hand anhand der Ausführungszeiten für diese Operationen gemessen und sind fest in den Quelltext einkodiert.

Anhand der tabellierten Werte für die Kostenparameter und der ermittelten Anzahlen der Wechselwirkungsberechnungen lässt sich vorhersagen, bei welcher Baumtiefe d_{opt} der Berechnungslauf für das aktuelle Problem die geringsten Kosten verursachen wird, d. h. die Ausführungszeit für die arithmetischen Operationen am niedrigsten ist.

Diese Lösung hat zwei Nachteile: Erstens ist es für jede neue CPU-Architektur, auf der die Anwendung ausgeführt wird, notwendig, die Konstanten `ndiv` und `nsqr` zu bestimmen. Da der Vorgang nicht automatisiert ist, ist er sehr aufwändig. Zweitens wird der Aufwand zum Durchmustern des Baumes, z. B. zum Suchen von Nachbar- oder Kindzellen, nur als Konstante einbezogen. Er variiert jedoch in Abhängigkeit von der Verteilung der Teilchen in die Zellen, insbesondere bei stark inhomogenen Teilchenverteilungen. In den folgenden Abschnitten wird für beide Probleme eine Lösungsmöglichkeit vorgestellt.

3.2.2 Berechnung der optimalen Baumtiefe

Zur Steigerung der Genauigkeit der Ausführungszeitvorhersage wird die im vorangegangenen Abschnitt erläuterte Lösung basierend auf dem Zählen von Operationen wie folgt verbessert: Das Kostenmodell wird verbessert, sodass es eine genauere, rechner- und eingabedatenspezifische Vorhersage ermöglicht. Außerdem wird eine dynamische Berücksichtigung des Verwaltungsoverheads ermöglicht. Der Verwaltungsoverhead umfasst u. a. den Aufwand zum Durchmustern des FMM-Baumes, zum Finden von Eltern- und Kindzellen und zum Prüfen zweier Zellen auf Wohlsepariertheit. Die rechnerspezifische Vorhersage wird dadurch ermöglicht, dass die Kosten für Operationen nicht mehr fest im Quelltext einkodiert, sondern zur Installationszeit der Anwendung für jede Operation speziell auf dem genutzten Rechner gemessen werden.

Zentrales Element der Ausführungszeitvorhersage ist der Algorithmus zur Durchmusterung des FMM-Baumes. Er soll im Folgenden beschrieben werden, bevor eine Erläuterung des Verfahrens zum Messen der rechnerspezifischen Kosten für die einzelnen Operationen im *Benchmarklauf* und zum Einbezug des Verwaltungsoverheads im *Analyselauf* gegeben wird.

Algorithmus zur Durchmusterung des FMM-Baumes

In Alg. 3.1 wird die Struktur von verschachtelten Schleifen gezeigt, die die Durchmusterung des FMM-Baumes zur Berechnung der Zellen- und der Teilchenwechselwirkungen durchführen. Dieser Algorithmus stellt ein generelles Schema dar, das im Benchmarklauf und im Analyselauf der FMM angewandt wird. Für den Berechnungslauf wird es wie in Abschnitt 3.2.1 beschrieben minimal abgewandelt: Die Nahfeld- und die Fernfeldwechselwirkungen werden dort in zwei verschiedenen Schritten, Schritt II und Schritt V, berechnet, um durch eine bessere Datenlokalität eine Verringerung der Ausführungszeit zu erzielen: In Schritt II wird nur auf Zellendaten, in Schritt V nur auf Teilchendaten zugegriffen.

```
1  iteriere über Baumebenen l = 2 . . . d
2      iteriere über alle Zellen b der Ebene l
3          finde Elternzelle p der Zelle b
4          wenn Ebene l = Blattebene d, dann
5              berechne Teilchenwechselwirkungen in Zelle b und zwischen den Teilchen der Zelle b und von allen
               anderen nicht-leeren Kindzellen von p
6          iteriere über alle Nachbarzellen n der Elternzelle p
7              iteriere über nicht-leere Kindzellen c der Nachbarzelle n
8                  wenn Zellen c und b sind wohlsepariert, dann
9                      berechne Zellenwechselwirkungen zwischen Zellen c und b
10                 sonst, wenn Ebene l = Blattebene d, dann
11                     berechne Teilchenwechselwirkungen zwischen den Teilchen der Zellen c und b
```

Algorithmus 3.1 : Durchmusterung des FMM-Baumes für Schritt II und Schritt V der Implementierung mit Baumtiefe d [59]

Der Algorithmus geht davon aus, dass die Baumtiefe d größer als 1 ist, da ansonsten das gesamte Teilchensystem nur mit Teilchenwechselwirkungen berechnet würde. Die erste Schleife in Alg. 3.1 (l-Schleife in Zeile 1) läuft von Ebene 2 bis zur letzten Ebene d des FMM-Baumes. Innerhalb der l-Schleife läuft die b-Schleife in Zeile 2 über alle Zellen b der aktuellen Ebene l. Zu jeder Zelle b wird ihre Elternzelle p bestimmt. Falls die aktuelle Ebene l die letzte Ebene d des Baumes ist, wird der erste Teil der Teilchenwechselwirkungen berechnet. Er umfasst die Wechselwirkungen innerhalb von b sowie die Wechselwirkungen zwischen den Teilchen aus b und den Teilchen der weiteren Kindzellen der gemeinsamen Elternzelle p (Zeile 5). Die n-Schleife in Zeile 6 läuft über die 26 Nachbarzellen von p. Diese wird durch drei verschachtelte Schleifen, also eine für jede Raumdimension x, y und z implementiert. Wenn p am Rand der Systemzelle liegt, werden die Nachbarzellen übersprungen, sodass insgesamt weniger als 26 Nachbarzellen betrachtet werden. Die innere c-Schleife läuft über alle Kindzellen der Nachbarzelle n und prüft, ob die Zellen c und b wohlsepariert sind. Ist das der Fall, wird in Zeile 9 eine Zellenwechselwirkung durchgeführt. Andernfalls wird, falls die aktuelle Ebene l die letzte Ebene d des Baumes ist, der zweite Teil der Teilchenwechselwirkungen berechnet. Er umfasst die Wechselwirkungen der Teilchen in Blattzellen b mit den Teilchen in jeder Blattzelle c.

Benchmarklauf

Zum Zeitpunkt der Installation der Anwendung, also vor dem erstmaligen Ausführen auf einem unbekannten Rechnersystem, wird ein *Benchmarklauf* ausgeführt, der die rechnerspezifischen Kostenparameter ermittelt. Es existiert je ein Kostenparameter für

1. die Teilchenwechselwirkung,
2. jede Multipolanzahl $0 \leq m \leq 50$ für die Zellenwechselwirkung,
3. jede der Schleifen in Alg. 3.1.

Die Kosten einer Wechselwirkungsberechnung bzw. einer Schleifeniteration werden mit Hardware-Performance-Countern [245] gemessen. Mit diesen Performance-Countern wird die Anzahl benötigter CPU-Takte bzw. verarbeiteter Maschinenbefehle ermittelt.

Zunächst werden die Kosten einer Zellenwechselwirkung gemessen, indem die entsprechende Routine 1000-mal ausgeführt. Jeweils vor und nach der Ausführung wird der genutzte Hardware-Performance-Counter ausgelesen. Der Vorgang wird für alle Multipolanzahlen $0 \leq m \leq 50$ durchgeführt.

Die Kosten einer Teilchenwechselwirkung bzw. eines Durchlaufs jeder Schleife werden durch testweises Ausführen der Simulation mit einer inhomogenen Teilchenverteilung ermittelt. Die Teilchen sind in dieser Verteilung stark im Zentrum des Systems konzentriert, was zu einer großen Anzahl leerer Zellen und großen Unterschieden in den Teilchenanzahlen der Zellen führt. In Abschnitt 3.3.1 wird diese Xenon-Verteilung genauer beschrieben. Durch die Nutzung von Testdaten mit diesen Eigenschaften wird sichergestellt, dass alle im Baumdurchmusterungsalgorithmus vorkommenden Schleifen und Verzweigungen ausreichend oft ausgeführt werden, sodass ihre Kosten präzise gemessen werden können.

Die Kosten der Schleifen werden einzeln für jede Schleife in einem separaten Benchmarklauf gemessen. Dazu wird Alg. 3.1 jeweils mit den Testdaten und mit einer festen Baumtiefe von $d = 8$ ausgeführt und die Hardware-Performance-Counter vor Zeile 1 und nach Zeile 11 ausgelesen. Zunächst werden die Kosten für die l-Schleife allein bestimmt, indem die inneren Schleifen auskommentiert werden. Es wird ermittelt, wie oft die l-Schleife iteriert wird und die Gesamtkosten durch die Iterationsanzahl geteilt, um die Kosten für eine Iteration der l-Schleife zu erhalten. Fortgefahren wird mit der b-Schleife. Hier wird ebenfalls die Iterationsanzahl ermittelt und die Kosten mit auskommentierten inneren Schleifen gemessen. Dadurch, dass die Gesamtkosten für den Schleifenkomplex mit und ohne die b-Schleife bekannt sind, können anhand der Differenz die Kosten für die b-Schleife allein bestimmt werden. So wird weiter mit der n- und der c-Schleife verfahren. Bei der c-Schleife werden lediglich noch Wechselwirkungsberechnungen auskommentiert.

Abschließend werden die Kosten für die Berechnung einer Teilchenwechselwirkung gemessen. Dazu wird der Schritt V, der das Nahfeld berechnet, separat ausgeführt. Dabei werden die Anzahl der berechneten Wechselwirkungen und die Gesamtkosten ermittelt. Durch Bildung des Quotienten aus Gesamtkosten und Anzahl der Wechselwirkungsberechnungen werden die Kosten für eine Wechselwirkungsberechnung bestimmt.

Ergebnis des Benchmarklaufs sind rechnerspezifische Kostenparameter für die Nahfeld- und Fernfeld-Wechselwirkungsberechnungen sowie für jede der Schleifen in Alg. 3.1. Sie werden für einen Rechner einmal ermittelt und dann abgespeichert, sodass sie bei jeder Ausführung der FMM wieder abgerufen werden können.

Analyselauf

Vor der Durchführung der eigentlichen Berechnung der FMM im Berechnungslauf findet der Analyselauf statt. Er dient zwei Zielen: der Ermittlung der optimalen Multipolanzahl m nach [58] zum Erreichen des geforderten Fehlers des Ergebnisses sowie der Ermittlung der optimalen Baumtiefe, mit der die Kosten minimal werden. Der Analyselauf hat wieder die beschriebene Schleifenstruktur, in der ermittelt wird, wie häufig jeder Codeabschnitt für die Eingabeteilchenverteilung ausgeführt wird. Diese Anzahlen werden multipliziert mit den jeweiligen Kostenparametern und aufsummiert. Für das Fernfeld werden jeweils die Fernfeldkosten der nächstniedrigeren Baumtiefe genommen und die hinzukommenden Operationen dazuaddiert. Die Nahfeldkosten werden für jede Baumtie-

fe neu ermittelt. Für jede Baumtiefe d werden die Gesamtkosten durch Addition der Kosten für das Nahfeld, das Fernfeld und den Verwaltungsoverhead ermittelt. Die optimale Baumtiefe d_{opt} mit minimalen Kosten wird dann für den tatsächlichen Berechnungslauf gewählt.

3.3 Experimente

In diesem Abschnitt werden die Experimente vorgestellt, die durchgeführt wurden, um die vorgeschlagene verbesserte Methode zur Ermittlung der optimalen Baumtiefe zu evaluieren. Für die Vorhersage werden als Kosten für die Ausführungszeit die Anzahl ausgeführter Maschinenbefehle sowie die Anzahl benötigter Taktzyklen verwendet. Außerdem wird der Einfluss der verschachtelten Schleifen auf die Vorhersage untersucht. Weiterhin werden die Genauigkeit der Vorhersage und der durch sie verursachte Overhead in der FMM-Implementierung untersucht. Schließlich wird geprüft, ob sich das für die Minimierung der Ausführungszeit entworfene Modell auch zur Minimierung des Energieverbrauchs eignet.

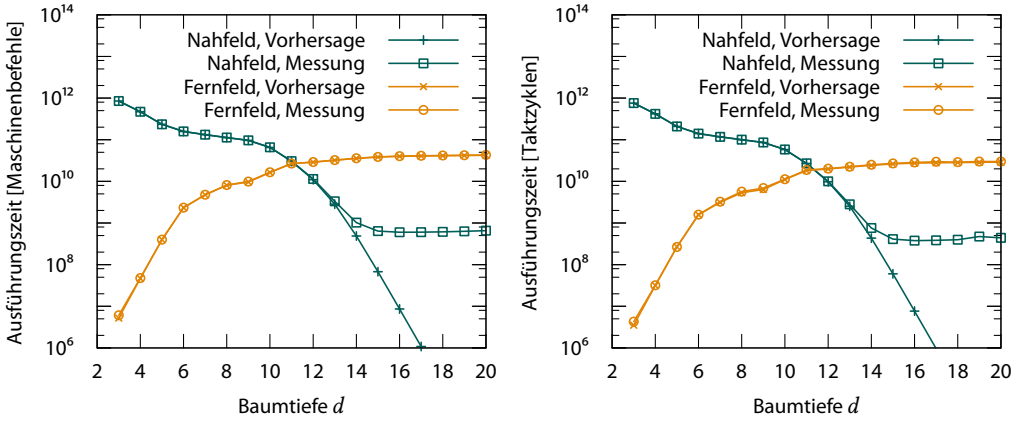
3.3.1 Experimentieranordnung

Für die Experimente wurden drei verschiedene Teilchenverteilungen genutzt. Die *homogene Verteilung* besteht aus 8^6 Teilchen, die regelmäßig auf einem Gitter im Einheitswürfel verteilt sind. Die *Kugelschalenverteilung* wird aus 8 konzentrischen Kugeln geformt, auf deren Oberfläche jeweils 8^5 Teilchen liegen. Die *Xenon-Verteilung* besteht aus 113 000 sehr inhomogen verteilten Teilchen. Die Verteilung entsteht durch eine Coulomb-Explosion von Xenon, also den Übergang von Materie, die von einem fokussierten Laser angeregt wurde, in Plasma. Die Besonderheit der Xenon-Verteilung ist, dass sehr viele Teilchen im Zentrum konzentriert sind, während andere Regionen nur sehr wenige Teilchen enthalten: So ist beispielsweise die Hälfte der Teilchen in 0,025 % des Raums konzentriert. Diese sehr inhomogene Verteilung ruft eine große Anzahl leerer Zellen im FMM-Baum hervor.

Die Messungen wurden auf den folgenden Rechnern durchgeführt: einem Intel-System mit einem Xeon-X5650-Prozessor mit 2,67 GHz Taktfrequenz und 12 GiB RAM sowie einem AMD-System mit einem Opteron-8347-Prozessor mit 1,9 GHz Taktfrequenz und 16 GiB RAM. Der Quelltext wurde mit dem GNU Fortran-Compiler mit dem Optimierungsschalter `-O0` übersetzt. Die Kosten der Ausführung wurden mit der PAPI-Bibliothek [43] gemessen. Zur Messung der Ausführungszeit wurden die PAPI-Ereignisse `PAPI_TOT_INS` für die Anzahl ausgeführter Maschinenbefehle und `PAPI_TOT_CYC` für die Anzahl benötigter Taktzyklen benutzt, die auf die entsprechenden Hardware-Performance-Counter der CPU zurückgreifen. Der Energieverbrauch wurde mit dem PAPI-Ereignis `rapl:::PACKAGE_ENERGY:PACKAGE0`, das wiederum auf das RAPL-Register `MSR_PKG_ENERGY_STATUS` zugreift, gemessen.

3.3.2 Ausführungszeit mit Maschinenbefehlen und Taktzyklen

In Abbildung 3.3a werden die im Analyselauf vorhergesagten Kosten mit den im Berechnungslauf gemessenen Kosten in Abhängigkeit von der Baumtiefe d verglichen. Für die Kosten werden in Abb. 3.3a die Anzahl verarbeiteter Maschinenbefehle und in Abb. 3.3b die benötigten Taktzyklen



(a) Ausführungszeit, gemessen in Maschinenbefehlen

(b) Ausführungszeit, gemessen in Taktzyklen

Abbildung 3.3: Vergleich zwischen der Anzahl vorhergesagter und gemessener Anzahl von Taktzyklen und Machinenbefehlen für die Nahfeld- und die Fernfeldberechnungen der FMM mit der Xenon-Verteilung und einer Fehlerschranke von 10^{-3}

verwendet. Die Ergebnisse wurden mit der Xenon-Verteilung bei einem relativen Fehler von 10^{-3} auf dem Xeon-Rechner erzielt.

Die Vorhersage der Fernfeldberechnungen ist in beiden Fällen sehr präzise. Bei den Nahfeldberechnungen stimmen vorhergesagte und gemessene Werte nur bis zu einer Baumtiefe von $d = 13$ überein. Mit steigender Baumtiefe sinken die vorhergesagten Kosten, während die gemessenen Kosten konstant bleiben. Das wird dadurch verursacht, dass die Kosten der Schleifen über die Teilchen, die zur Nahfeldberechnung benötigt werden, nicht betrachtet werden. Allerdings sind die Abweichungen nicht von Relevanz, da für das Ergebnis nur die Summe der Nahfeld- und der Fernfeldkosten entscheidend ist. Bei einer hohen Baumtiefe überwiegen jedoch, wie in den Diagrammen erkennbar, die Kosten für die Fernfeldberechnung die Kosten für die Nahfeldberechnung um mehrere Größenordnungen.

Bei der Summe der Nahfeld- und der Fernfeldkosten weicht die Vorhersage der Anzahl der Maschinenbefehle um maximal 4 % von der Messung ab. Bei den Taktzyklen beträgt die Abweichung maximal 7 %. Dass die Abweichung für die Taktzyklen etwas höher ist als der Wert für die Maschinenbefehle liegt daran, dass bei der Anzahl der Taktzyklen auch schwer vorhersagbare Effekte wie z. B. Cache-Misses Einfluss haben. Allerdings treten solche Effekte auch im Benchmarklauf auf und sind deshalb zum größten Teil schon in der Vorhersage berücksichtigt. Das wurde in Experimenten mithilfe des PAPI-Zählers für Level-1-Cache-Misses bestätigt. Aufgrund der geringen Abweichung von den gemessenen Werten kann die Vorhersage der Ausführungszeit als sehr präzise angesehen werden.

Tabelle 3.1 führt die rechnerspezifischen Kostenparameter für den AMD-Rechner und für den

		Intel Xeon X5650		AMD Opteron 8347	
Kostenparameter	Zähler	Befehle	Takte	Befehle	Takte
b-Schleife	4680	2367	1567	2251	2260
n-Schleife, z-Richtung	14040	14	3	14	5
n-Schleife, y-Richtung	40080	21	16	21	14
n-Schleife, x-Richtung	114480	31	21	31	21
c-Schleife	52184	65	65	65	106
Nahfeldwechselwirkung	32838560	163	145	163	163
Fernfeldwechselwirkung	2436056	2583	1842	2529	1897

Tabelle 3.1: Verteilungsspezifische Zählerwerte für die homogene Teilchenverteilung mit $d = 5$ sowie rechnerspezifische Kostenparameter für die Anzahl der Maschinenbefehle und der Taktzyklen bei dem Intel- und dem AMD-Rechner

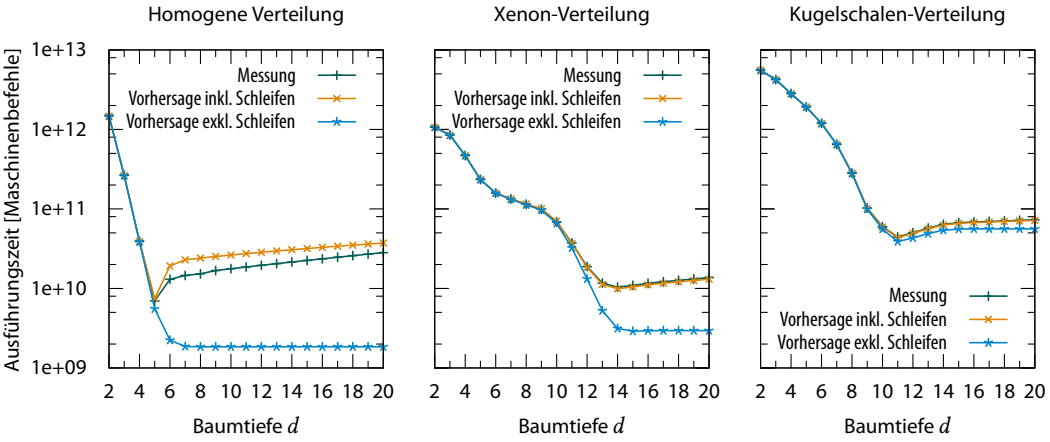


Abbildung 3.4: Vergleich der Vorhersage mit der Messung der Anzahl der Maschinenbefehle mit und ohne Berücksichtigung der verschachtelten Schleifen aus Alg. 3.1, Fehlerschranke 10^{-1}

Intel-Rechner auf, die in zwei Benchmarkläufen für die Maschinenbefehle und für die Taktzyklen ermittelt wurden. Außerdem führt sie die Werte der Zählervariablen auf, die im Analyselauf für die homogene Verteilung mit einer Baumtiefe von $d = 5$ ermittelt wurden. Der vorgegebene relative Fehler war hier 10^{-3}. Die Werte zeigen, dass es bei verschiedenen CPUs zu signifikanten Unterschieden für die Kostenparameter kommen kann. Allerdings hat nicht nur die Hardware, sondern beispielsweise auch der Optimierungsgrad des Compilers Einfluss. Daher ist es unerlässlich, den Benchmarklauf für jede neue Hardware- und Softwarekonfiguration durchzuführen.

3.3.3 Einfluss der verschachtelten Schleifen

Warum der Einbezug der Kosten der verschachtelten Schleifen aus Alg. 3.1 wichtig ist, zeigt Abb. 3.4: Sie vergleicht die Messung mit Vorhersage ohne Einbezug der Schleifen und der Vorhersage mit Einbezug der Schleifen. Die Ergebnisse werden für alle drei untersuchten Teilchenverteilung mit einer

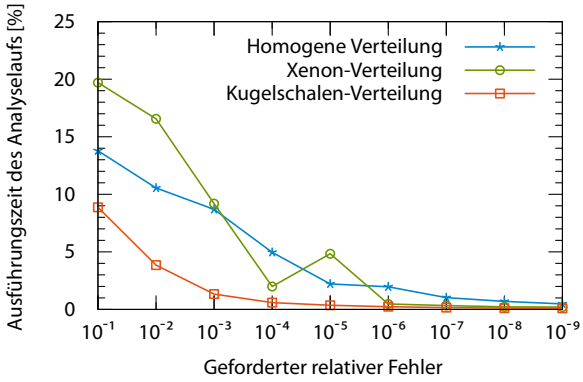


Abbildung 3.5: Anteil des Analyselaufes an der Gesamtausführungszeit der FMM in Abhängigkeit des angeforderten relativen Fehlers

relativen Fehlerschranke von 10^{-1} dargestellt. Bei dieser Fehlerschranke haben die Schleifen einen signifikanten Einfluss: Die Vorhersagen mit und ohne Einbezug der Schleifen unterscheiden sich um bis zu eine Größenordnung. Bei der homogenen und bei der Xenon-Verteilung wird deutlich, dass ohne Berücksichtigung der Kosten der Schleifen das Minimum der Ausführungszeit falsch ermittelt wird: Die Vorhersage ohne Einbezug der Schleifen fällt dort bis zur Baumtiefe $d = 20$ monoton ab, sodass das Minimum bei $d \geq 20$ ermittelt wird. Tatsächlich ist es aber bei der homogenen Verteilung bei $d = 5$ und bei der Xenon-Verteilung bei $d = 14$. Würde beispielsweise bei der homogenen das Minimum fälschlicherweise bei $d = 20$ ermittelt, hätte das eine viermal so lange Ausführungszeit zur Folge.

3.3.4 Overhead der Ausführungszeitvorhersage

Für das Durchspielen der Berechnung im Analyselauf wird eine gewisse Zeit benötigt, die nicht vernachlässigt werden kann. Die Kosten der Zellenwechselwirkungen steigen mit sinkender Fehlerschranke, was die Ausführungszeit der FMM verlängert. Dadurch sinkt der Anteil des Analyselaufes, dessen Ausführungszeit unabhängig von der Fehlerschranke ist. Abbildung 3.5 zeigt den Anteil des Analyselaufes an der Gesamtausführungszeit der FMM für einen geforderten relativen Fehler von 10^{-1} bis 10^{-9}. Bei einem relativen Fehler von 10^{-1} beträgt der Anteil des Analyselaufes bis zu 20 %, bei sehr kleinen Fehlern im Bereich von 10^{-9} beträgt der Anteil ca. 0,5 %. Zu beachten ist allerdings, dass der Analyselauf ohnehin für die Fehlersteuerung nach [58] notwendig ist und daher für die Ausführungszeitvorhersage allein kaum weitere Kosten anfallen.

3.3.5 Abweichung Vorhersage–Messung

Abbildung 3.6 zeigt für die relativen Fehlerschranken von 10^{-1} bis 10^{-10} die relative Abweichung zwischen Vorhersage und Messung, d. h. die Differenz zwischen vorhergesagtem und gemessenem Wert im Verhältnis zum Messwert. Es werden sowohl die Ergebnisse für die ursprünglich vorliegende Variante, die die Gleitkommaoperationen zählt und anhand deren spezifischer Kosten die

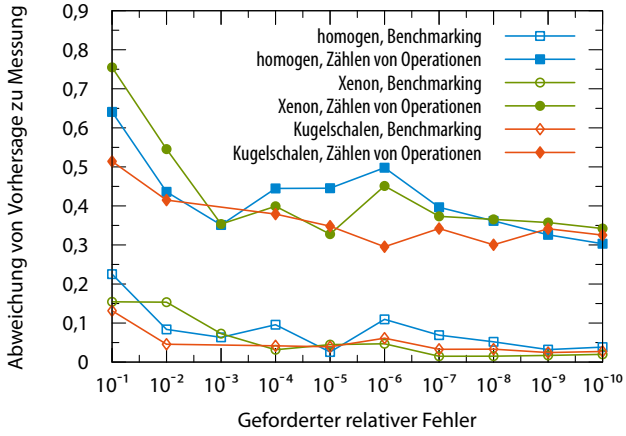


Abbildung 3.6: Relative Abweichung der Vorhersage von der Messung beim ursprünglichen Vorhersageverfahren (Zählen der arithmetischen Operationen) und beim verbesserten Verfahren (Einbezug aller Codeabschnitte durch Benchmarking)

Ausführungszeit abschätzt, als auch für die hier entwickelte Variante, die alle Codeabschnitte einbezieht und deren Kosten auf der genutzten Hardware misst, dargestellt. Für die Abweichung wurde der Wert der optimalen Baumtiefe und der beiden Nachbarwerte gemittelt.

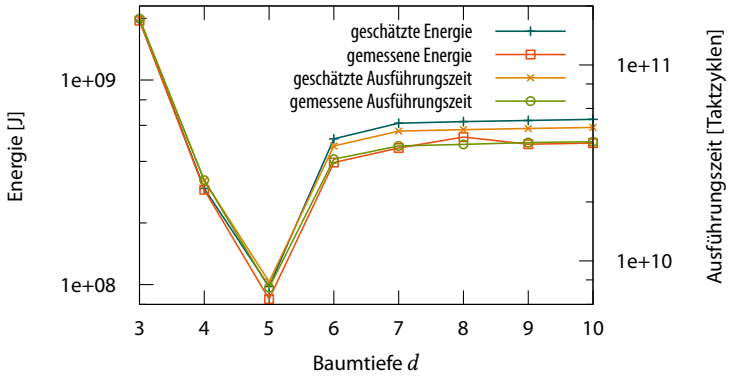
Bei großen relativen Fehlerschranken wie 10^{-1} sinkt die Ausführungszeit der FMM stark und die Vorhersage weicht um bis zu 23 % von der tatsächlichen Ausführungszeit ab. Wird die Fehlerschranke sehr klein gewählt, erhöht sich die Ausführungszeit der FMM und die Abweichung sinkt auf ca. 5 %.

3.3.6 Einbezug der Energie

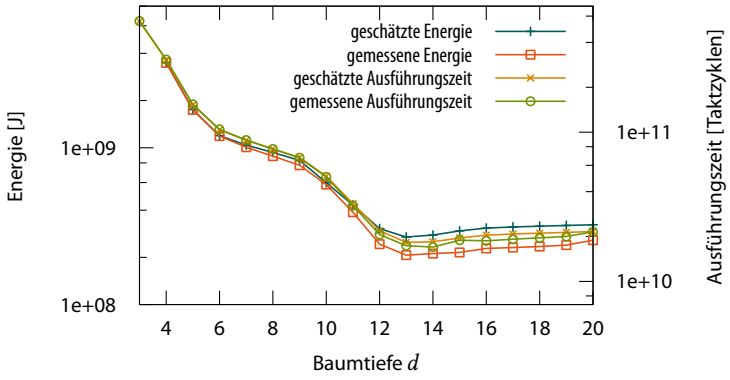
Die Methode, die bislang und in [59] nur zum automatisierten Tuning der Ausführungszeit genutzt wurde, wird nun auf den Energieverbrauch angewandt. In Abb. 3.7 werden die Vorhersage und die Messung des Energieverbrauchs mit RAPL dargestellt. Für alle drei Verteilungen wurde der Versuch mit einer Fehlerschranke von 10^{-3} durchgeführt. Zum Vergleich werden auch die Ergebnisse bei Minimierung der Ausführungszeit dargestellt.

Die Resultate belegen die Robustheit des zur Vorhersage verwendeten Modells: Obwohl bei der Entwicklung nicht an die Energie als Maß für die Kosten gedacht wurde, wird auch der Energieverbrauch gut vorhergesagt. Zu Abweichungen kommt es vor allem bei der Fernfeldberechnung. Bei der homogenen Verteilung und bei der Xenon-Verteilung kommt es zu Abweichungen von bis zu einem Drittel. Das Minimum der Vorhersage liegt trotzdem stets an derselben Stelle wie das der Messung, sodass die Baumtiefe nicht falsch ausgewählt wird.

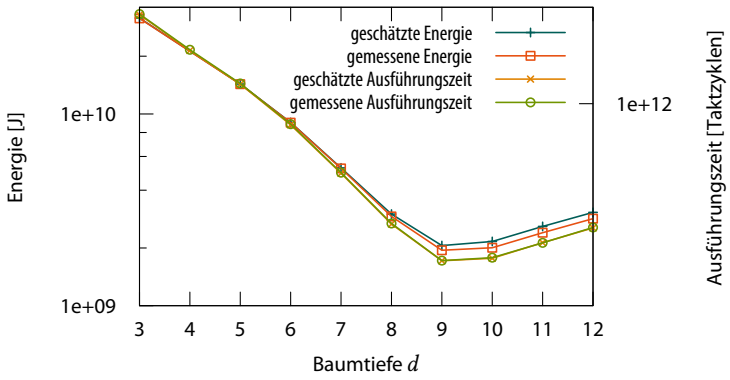
Die Resultate bestätigen die vielfach getätigte Beobachtung [267, 282], dass der Energieverbrauch meist durch eine Minimierung der Ausführungszeit minimiert wird: Bei der homogenen Verteilung und bei der Kugelschalen-Verteilung liegen die Minima für Ausführungszeit und Energie bei der-



(a) Homogene Verteilung



(b) Xenon-Verteilung



(c) Kugelschalen-Verteilung

Abbildung 3.7: Vergleich Laufzeit–Energieverbrauch der FMM in Abhängigkeit von der Baumtiefe für verschiedene Teilchenverteilungen.

selben Baumtiefe. Bei der Xenon-Verteilung liegt das Minimum der Ausführungszeit bei der Baumtiefe $d = 14$ und das Minimum des Energieverbrauchs bei $d = 12$. Allerdings unterscheiden sich die Kosten für beide Baumtiefen nur um rund 2 %.

3.4 Schlussfolgerungen

In diesem Kapitel wurde gezeigt, dass mit dem entwickelten Autotuning-Verfahren die Genauigkeit der Vorhersage gegenüber dem zuvor genutzten Verfahren, das auf dem Zählen von Operationen basiert und den Verwaltungsoverhead als konstant ansieht, verbessert werden kann. Nur durch diese präzise Schätzung der Ausführungszeit kann die optimale Baumtiefe für den FMM-Baum genau vorhergesagt werden. Mit dem genutzten Verfahren ist die Genauigkeit der Vorhersage sowohl, wenn die Ausführungszeit in CPU-Takten gemessen wird, als auch, wenn sie in der Anzahl verarbeiteter Maschinenbefehle gemessen wird, ausreichend. Der für die Kostenvorhersage genutzte Analyselauf verlängert zwar selbst auch die Ausführungszeit, allerdings kann der entstehende Overhead vernachlässigt werden, da der Analyselauf für die Fehlersteuerung der FMM ohnehin benötigt wird.

Eine Übertragung der Methode auf den Energieverbrauch als Kostenmaß zeigte, dass das Modell robust ist und sich mit der entwickelten Methode auch der Energieverbrauch gut vorhersagen lässt. Die Vorhersage zeigte zwar beim Energieverbrauch größere Abweichungen als bei der Ausführungszeit, war aber trotzdem noch ausreichend genau, um stets die Baumtiefe mit minimalem Energieverbrauch korrekt vorherzusagen. Eventuell wird es nötig sein, auch den Schleifenoverhead bei der Nahfeldberechnung mit einzubeziehen, um dort eine ähnliche Genauigkeit wie bei der Ausführungszeit zu erreichen.

3.5 Zusammenfassung

Die FMM berechnet die Wechselwirkungen zwischen n homogen im Raum verteilten Teilchen in einer Zeit in $O(n)$. Sie arbeitet mit einer Baumdatenstruktur, bei der die Tiefe des Baumes einen entscheidenden Einfluss auf die Ausführungszeit des Algorithmus hat. Bei der ursprünglich vorliegenden Implementierung wurde die Ausführungszeit anhand der jeweils durchzuführenden arithmetischen Operationen geschätzt. Die Kosten für eine einzelne Operation wurden von Hand für den genutzten Rechner bestimmt.

Die hier entwickelte verbesserte Methode ermöglicht eine genaue Vorhersage der Ausführungszeit anhand von rechnerspezifischen und eingabedatenspezifischen Parametern. Die rechnerspezifischen Kostenparameter geben die Kosten der Ausführung verschiedener Codeabschnitte an und werden in einem Benchmarklauf konkret für den genutzten Rechner ermittelt. Die eingabedatenspezifischen Parameter werden in einem Analyselauf ermittelt, der für jeden der Codeabschnitte zählt, wie oft er für die gegebenen Eingabedaten durchlaufen wird. Damit kann auch der z. B. durch die Traversierungen im FMM-Baum entstehende Verwaltungsoverhead gut in die Vorhersage einbezogen werden.

Die Resultate zeigen, dass mit dem vorgestellten Verfahren die Ausführungszeit sehr genau vorhergesagt wird. Die falsche Vorhersage der Baumtiefe mit minimaler Ausführungszeit, die bei der

Ausgangsversion teilweise auftrat, wurde mit der vorgestellten Methode behoben. Der durch den Analyselauf verursachte Overhead ist für die fehlergesteuerte FMM vernachlässigbar. Die Übertragung der Methode auf den Energieverbrauch zeigt, dass sich auch jener gut vorhersagen und damit minimieren lässt. Eine spezielle Betrachtung der Energie ist notwendig, da die Baumtiefe mit minimaler Ausführungszeit nicht in allen Fällen gleichzeitig den Energieverbrauch minimiert. ■

4 Methode der Finiten Elemente

Die *Methode der Finiten Elemente* (engl. *finite element method, FEM*) wurde zur numerischen Behandlung elliptischer Differentialgleichungen entwickelt, wird aber inzwischen auch für andere Differentialgleichungen in einer Vielzahl von Anwendungsgebieten genutzt [38]. Das grundlegende Konzept der FEM beruht auf der Diskretisierung eines Raumes in diskrete, finite Elemente. Hier wird eine Implementierung einer adaptiven FEM [31] untersucht, die unter anderem zur Lösung von Deformationsproblemen verwendet wird [18]. Der zu untersuchende Körper wird hier in hexaederförmige Elemente unterteilt. Jedes Element besteht aus 6 Flächen, 12 Kanten und 8, 20 oder 27 Knoten. Jeder Knoten wiederum hat bis zu 3 Freiheitsgrade. Die vorliegende Implementierung wurde in Fortran ausgeführt und ist teilweise parallelisiert [31].

Eingabe für die FEM sind das Grobnetz des Körpers und die Parameter der Simulation. Bis zum Erreichen der gewünschten Genauigkeit werden folgende Schritte wiederholt ausgeführt:

I. adaptive Netzverfeinerung,
II. Assemblierung der Steifigkeitsmatrix,
III. Lösung des entstandenen linearen Gleichungssystems,
IV. Fehlerschätzung.

Die Adaptivität bei der Netzverfeinerung drückt sich dadurch aus, dass das Netz an Stellen, an denen die stärksten Kräfte herrschen bzw. die größte Verformung auftritt, stärker verfeinert wird als an anderen Stellen. Bei dem in Abb. 4.1 dargestellten Objekt, das einen Quader mit einem Bohrloch darstellt, ist diese adaptive Verfeinerung gut erkennbar. Im Vergleich zur vollständigen Verfeinerung wird durch die adaptive Verfeinerung bei gleicher Genauigkeit der Rechenaufwand erheblich gesenkt.

Die Schritte II und III der FEM-Implementierung sind in einer parallelen Version verfügbar. Wie in Abbildung 4.2 dargestellt, handelt es sich um eine Parallelisierung nach dem Master-Slave-Prinzip: Der Master führt die Hauptschritte und die sequentiellen Teile der FEM aus, die Slaves die speicher- und rechenintensiven Teile der Assemblierung der Elementmatrizen sowie die Durchführung von Matrix-Vektor-Multiplikationen. Schritt III ist der zeitaufwändigste Schritt der FEM und wird daher im Folgenden einer genaueren Betrachtung unterzogen.

Die Lösung des Gleichungssystems in Schritt III erfolgt mit der *Methode der konjugierten Gradienten* (engl. *conjugate gradient method, CGM*). In diesem Kapitel werden verschiedene Ansätze zur Verringerung der Ausführungszeit der parallelen Implementierung dieser Methode vorgestellt.

Zunächst wird die Methode der konjugierten Gradienten in *Unterkapitel 4.1* erklärt. In *Unterkapitel 4.2* wird eine Methode vorgestellt, mit der sich in der parallelen Implementierung die Reduktion der verteilt liegenden Ergebnisvektoren zu einem gemeinsamen Ergebnisvektor effizient implementieren lässt. Dafür wird das Speichermodell von verteiltem auf gemeinsamen Speicher umgestellt, und die vorhandene explizite grobgranulare Reduktion wird durch eine implizite fein-

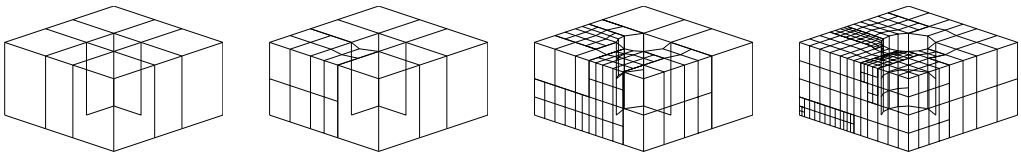


Abbildung 4.1: Verfeinerung des FEM-Netzes für das Beispiel *bohrung* zu Beginn der Rechnung sowie nach dem 1., nach dem 4. und nach dem 7. Verfeinerungsschritt; Grafiken erzeugt durch das Programmpaket SPC-PM3AdH-XX [31], die Ausgangsdaten entstammen der zum Programmpaket gehörigen Bibliothek.

granulare Reduktion ersetzt. Sie findet nun nicht mehr nach Beendigung des parallelen Abschnitts statt: Stattdessen wird jedes Ergebnis einer Teilberechnung durch atomare Operationen synchronisiert direkt auf den gemeinsamen Vektor geschrieben. In *Unterkapitel 4.3* wird untersucht, ob es einen Vorteil bringt, die Verarbeitung der finiten Elemente auf der CPU durchzuführen, in deren Speicherbereich die zugehörigen Daten abgespeichert sind. In *Unterkapitel 4.4* wird zunächst die CGM teilweise für die GPU implementiert. Anschließend werden Experimente durchgeführt, die den Energieverbrauch und die Ausführungszeit der CGM auf CPUs und GPUs untersuchen. Die Experimente berücksichtigen bei der CPU die dynamische Frequenz- und Spannungsregelung und den Energieverbrauch des Speichers sowie die für die Datenübertragung zwischen Haupt- und GPU-Speicher benötigte Zeit und Energie. Anhand der Ergebnisse der Experimente wird ein Energie- und Ausführungszeitmodell für die Ausführung der CGM auf der CPU und auf der GPU aufgestellt. Dieses Modell wird genutzt, um eine Vorhersage zu treffen, welche Art der Ausführung der CGM die energieeffizienteste ist: CPU-Ausführung, GPU-Ausführung oder eine gemeinsame CPU-GPU-Ausführung. Für die gemeinsame CPU-GPU-Ausführung wird in *Unterkapitel 4.5* eine Methode zur dynamischen Verteilung der Rechenlast vorgestellt. Sie minimiert die Ausführungszeit der CGM, indem sie die Rechenlast über ein Online-Autotuning-Verfahren in jedem FEM-Verfeinerungsschritt neu auf CPU und GPU verteilt. Sie misst dabei die Ausführungszeiten und ermöglicht so die ständige Anpassung an sich verändernde Verhältnisse, z. B. aufgrund größer werdender Datenmengen. Es wird gezeigt, dass die genutzte Methode keinen signifikanten Overhead erzeugt. In *Unterkapitel 4.6* werden die Ergebnisse dieses Kapitels diskutiert und in *Unterkapitel 4.7* wird es noch einmal zusammengefasst.

4.1 Methode der konjugierten Gradienten

Das in Schritt III der FEM-Implementierung zu lösende Gleichungssystems

$$Au = b \tag{4.1}$$

wird mit der *Methode der konjugierten Gradienten* gelöst [110, 185]. Die Methode der konjugierten Gradienten prüft für eine Approximation u_k der Lösung, ob das Produkt aus Steifigkeitsmatrix A und Approximation u_k ungefähr so groß ist wie der Lastvektor b, also ob $Au_k \approx b$. Genauer gesagt wird geprüft, ob das Residuum $r_k = |Au_k - b|$ unter einer gegebenen Fehlerschranke e ist.

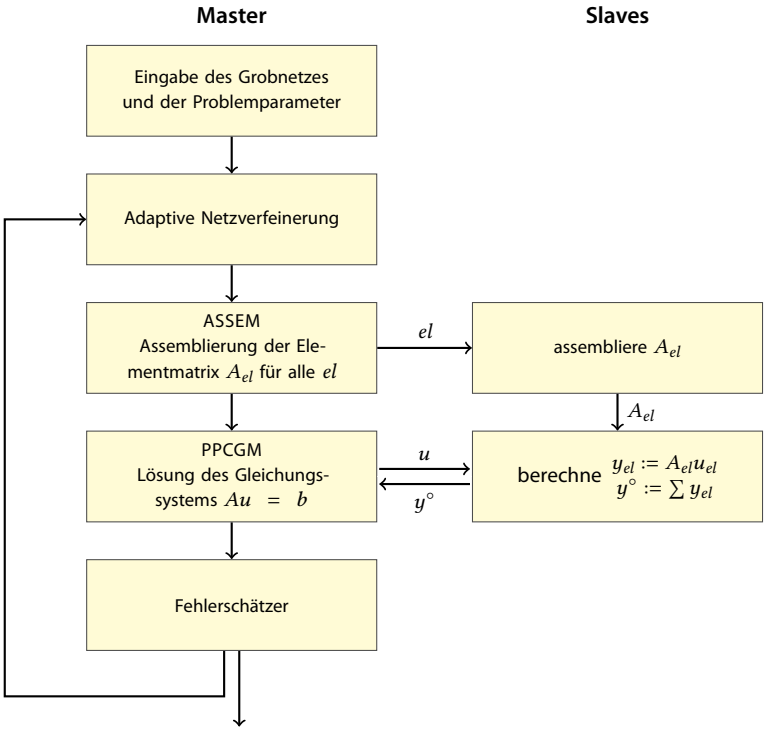


Abbildung 4.2: Darstellung des Programmablaufs für die adaptive FEM im Master-Slave-Modus (angepasst auf Grundlage von [31, S. 15], [87, S. 6] und [184])

Wenn die Bedingung nicht erfüllt ist, wird in einer weiteren Iteration die nächste Approximation u_{k+1} berechnet und überprüft. Die erste Approximation u_1 wird willkürlich, bzw. zur Abkürzung des Verfahrens anhand der Lösung des vorangegangenen Verfeinerungsschrittes, gewählt. Die Nutzung eines Vorkonditionierers für die Wahl des folgenden u_{k+1} beschleunigt die Konvergenz und verringert so die Anzahl notwendiger Iterationen.

Bei der CGM entsteht der Rechenaufwand hauptsächlich durch die vielfache Ausführung der Matrix-Vektor-Multiplikation

$$y = A\,u \quad , \tag{4.2}$$

die einmal pro Iteration durchgeführt wird. Das Ergebnis y wird anschließend mit dem Lastvektor b verglichen.

4.1.1 Parallelisierung

Die parallele Version aus [31] nutzt nicht die Gesamtvektoren u und y und die Gesamtmatrix A, sondern führt die Multiplikation in Gleichung (4.2) elementweise durch:

$$y_{el} = A_{el}\,u_{el} \tag{4.3}$$

für alle finiten Elemente *el*. Während die Größe der Gesamtvektoren proportional zur Anzahl der finiten Elemente wächst und in die hunderttausende gehen kann, bleibt die Größe der Elementvektoren konstant. In Abhängigkeit von der Anzahl der Knoten pro Element und der Anzahl ihrer Freiheitsgrade beträgt sie zwischen 8 und 81. Die Daten, die für ein Element benötigt werden, also die Elementdatenstrukturen A_{el}, u_{el} und y_{el}, werden durch die dafür vorgesehenen Funktionen o2el und el2o aus den Gesamtdatenstrukturen extrahiert und in diese zurück überführt:

$$A_{el} = \text{o2el}(A, el) \quad , \quad u_{el} = \text{o2el}(u, el) \quad , \tag{4.4}$$

$$y = \sum_{el} \text{el2o}(y_{el}) \quad . \tag{4.5}$$

Da die Gleichungen (4.3) bis (4.5) für die Elemente jeweils unabhängig von einander sind, können sie parallel ausgeführt werden. Lediglich die Schreibvorgänge in den Gesamtergebnisvektor y nach Gleichung (4.5) müssen synchronisiert werden.

4.1.2 Implementierung

In Alg. 4.1 wird die parallele Implementierung der CGM dargestellt: Der Routine PPCGM werden die Steifigkeitsmatrix A und der Lastvektor b übergeben. Die Iterationen der CGM werden durch die repeat-until-Schleife in den Zeilen 3 bis 7 durchgeführt. Die elementweise Matrix-Vektor-Multiplikation erfolgt in der Routine AXMEBE. Für jedes Element werden die Elementdatenstrukturen aus den Gesamtdatenstrukturen extrahiert, die Multiplikation durchgeführt und die elementweisen Ergebnisse im Gesamtergebnisvektor y zusammengefasst. Das Schreiben auf y wird durch den kritischen Abschnitt synchronisiert. Eine effiziente Implementierung der Reduktion der verteilt vorliegenden Elementergebnisvektoren y_{el} zum gemeinsamen Gesamtergebnisvektor y wird in Unterkapitel 4.2 diskutiert.

Die bislang genutzte Parallelisierung nutzt MPI [183] zum Nachrichtenaustausch. Da die Anwender ohnehin in der Regel Systeme mit gemeinsamem Speicher nutzten, wurde die Parallelisierung im Rahmen dieser Arbeit in einem ersten Schritt auf OpenMP [210] umgestellt. Der gemeinsam genutzte Speicher besitzt den Vorteil, dass die großen Datenmengen nicht redundant im Speicher vorgehalten werden müssen. Außerdem wurde die Kommunikation zwischen den Knoten erleichtert, da dafür keine expliziten Nachrichtenaustauschoperationen mehr notwendig sind. Aus der Sicht des Softwareentwicklers ist die OpenMP-Version einfacher zu handhaben, da dort die sequentielle und die parallele Version in einem gemeinsamen Quelltext kodiert werden. Auch die Kommunikation wird erleichtert. Die leichte Implementierung mit OpenMP ermöglichte die Parallelisierung vieler Programmabschnitte, die bisher nur sequentiell abgearbeitet wurden.

4.2 Reduktion verteilter Ergebnisvektoren

In diesem Unterkapitel wird eine Methode vorgestellt, mit der sich die Reduktionsoperation in Zeile 20 in Alg. 4.1 effizient darstellen lässt. In der in Alg. 4.1 gezeigten Implementierung ist sie unnötig aufwändig. Bei einer Architektur mit gemeinsamem Speicher ist es nicht sinnvoll, die Reduktion erst am Ende des parallelen Abschnitts durchzuführen. Vielmehr können die Threads das Ergebnis

```
1  PPCGM(A, b)
2  begin
3      repeat
4          call preconditioner
5          calculate next u
6          y := AXMEBE(A, u)
7      until y ≈ b
8      return u
9  end
10 AXMEBE(A, u)
11 begin
12     y := 0 // gemeinsamer Vektor
13     for each el do in parallel
14         y° := 0
15         A_el := o2el(A, el),   u_el := o2el(u, el)
16         y_el := A_el u_el
17         y° := el2o(y_el)
18     next
19     begin critical section
20         y := y + y°
21     end critical section
22     return y
23 end
```

Algorithmus 4.1 : Pseudocode der parallelen Implementierung der CGM

direkt nach seiner Berechnung in den gemeinsamen Speicherbereich schreiben und so die Speicherzugriffe zwischen die Berechnungen schieben. Das kann bewirken, dass für die Reduktion keine oder nur eine geringe Ausführungszeit zusätzlich zur Berechnungszeit nötig ist. Im Gegensatz dazu benötigt die explizite Reduktion am Ende des parallelen Abschnitts Ausführungszeit, die nur von der Geschwindigkeit des Speichers abhängt und während der die CPUs nicht ausgelastet sind.

Für die konkrete Anwendung ist es nicht notwendig, zur Durchführung der Addition den gesamten gemeinsamen Ergebnisvektor zu sperren. Jedes Element des Ergebnisvektors ist nur in wenigen, ca. ein bis zwei, privaten Ergebnisvektoren verschieden von Null. Es erscheint also sinnvoll, die Reduktion feingranular durchzuführen, d. h. für die Schreiboperationen nur die betroffenen Elemente des gemeinsamen Vektors zu sperren. Es ist daher zu untersuchen, welche Technik zur feingranularen Reduktion des Vektors eine effiziente Ausführung ermöglicht.

4.2.1 Feingranulare Reduktion

Bei feingranularer Reduktion eines Vektors wird bei der Durchführung der Reduktion jedes Vektorelement des Ergebnisvektors einzeln gesperrt, im Gegensatz zur grobgranularen Reduktion, bei der der gesamte Ergebnisvektor gesperrt und dann die Reduktionsoperation durchgeführt wird. Enthalten die zu reduzierenden Vektoren viele Elemente, die Nullelemente bzgl. der Reduktionsoperation sind, kann so Ausführungszeit eingespart werden. Es werden zunächst zwei Methoden zur feingranularen Reduktion der Vektoren y° der CGM vorgestellt: *atomare Operationen* und *feingranulare Sperren*.

Atomare Operationen

Atomare Operationen, die mehrere semantische Operationen in einer ununterbrechbaren Funktion vereinigen, können in Hardware und in Software implementiert werden. Für die Reduktion in der CGM wird eine atomare Addition von Gleitkommazahlen mit doppelter Genauigkeit benötigt. *x86*-CPUs ermöglichen schon seit der 80386-Generation das Sperren des Speicherbereiches einer Variablen, indem vor bestimmte arithmetische oder logische Befehle das *LOCK-Präfix* [5, Kap. 1.2.5] gesetzt wird [128, Kap. 15.7]. Erlaubte Datentypen dieser Operationen sind 8-, 16-, 32- und 64-Bit-Integer, sowohl vorzeichenbehaftet als auch vorzeichenlos [5, Kap. 3]. Auch GPUs unterstützen atomare Operationen, sowohl für Integer als auch – im Falle der Addition – für Gleitkommazahlen mit einfacher Genauigkeit [208, Kap. B.12.1].

Da die CGM Gleitkommazahlen doppelter Genauigkeit nutzt, können die in Hardware verfügbaren atomaren Additionsoperationen nicht eingesetzt werden. Stattdessen wird die atomare Additionsoperation in Software mithilfe des *Compare-&-Swap*-Befehl (CAS), der auf den meisten Plattformen verfügbar ist, emuliert. In dieser Arbeit wird der Compare-&-Swap-Befehl nach der in Alg. 4.2 gezeigten Definition verwendet: Der Wert der Variablen *oldVal* wird mit dem Wert an der Speicherstelle **location* verglichen. Sind sie gleich, wird der Wert *newVal* an diese Speicherstelle geschrieben. Der Rückgabewert von CAS ist das Ergebnis des Vergleichs. In Alg. 4.3 wird für die Addition gezeigt, wie jede binäre Operation mithilfe der CAS-Funktion emuliert werden kann: Ein Wert wird aus dem Speicher ausgelesen, über die Operation mit einem zweiten Wert verknüpft und das Ergebnis wird wieder an die Speicherstelle des ersten Wertes zurückgeschrieben. Dieses Zurückschreiben geschieht mit CAS und wird nur dann vollzogen, wenn der Wert im Speicher nicht zwischenzeitlich verändert wurde. Andernfalls wird der Vorgang mit dem veränderten Wert wiederholt.

```
1 bool CAS(T *location, T oldVal, T newVal)
2 begin
3     begin atomic
4         if *location == oldVal then
5             *location = newVal;
6         end
7         return (*location == oldVal);
8     end
9 end
```

Algorithmus 4.2 : Compare & Swap nach [97]

```
1 void ATOMIC_ADD(double *sum, double a)
2 begin
3     repeat
4         double oldSum = *sum;
5         double newSum = oldSum + a;
6     until CAS(sum, oldSum, newSum);
7 end
```

Algorithmus 4.3 : Emulation der atomaren Additionsoperation mit Compare & Swap

Feingranulare Sperren

Die Optimierung der Granularität von Reduktionen ist schon lange Thema der Forschung [228]. In diesem Abschnitt wird eine Technik vorgestellt, die jeweils für eine kleine Anzahl Vektorelemente eine gemeinsame Sperrvariable vorsieht. Vor dem Zugriff auf ein Element i des Vektors y wird die zugehörige Sperre mit LOCK(i) eingerichtet. Nach dem Zugriff wird sie mit RELEASE(i) aufgehoben.

Die Implementierung der LOCK- und UNLOCK-Funktionen wird in Alg. 4.4 dargestellt. Im Feld *locks* werden *n_locks* Sperrvariablen vorgehalten. Indem die Sperrvariable mit den verfügbaren

```
1  int* locks[n_locks]; // initialise with 0
2  void LOCK(int i)
3  begin
4      while (true) do
5          int lock_status = ATOMIC_ADD(&locks[i %
           n_locks], 1)
6          if (lock_status == 0) then
7              break
8          end
9          UNLOCK(i)
10         USLEEP(1)
11     end
12 end
13 void UNLOCK(int i)
14 begin
15     ATOMIC_ADD( &locks[i % n_locks], -1);
16 end
```

Algorithmus 4.4 : Implementierung der Routinen LOCK() and UNLOCK()

atomaren Additionsoperationen auf eine Zahl größer als Null gesetzt wird, wird die Sperre eingerichtet. Aufgehoben wird sie durch die Dekrementierung der Sperrvariablen. Zu beachten ist, dass der Rückgabewert der Funktion ATOMIC_ADD in Zeile 5 der Wert der Variablen vor der Inkrementierung ist. Für den Fall, dass das Erlangen der Sperre fehlschlägt, vermeidet die Anweisung USLEEP(1) in Zeile 9 einen Livelock, indem der Thread für eine Mikrosekunde angehalten wird. In Abschnitt 4.2.3 wird untersucht, welcher Wert für *n_locks* gewählt werden sollte.

4.2.2 Implementierung

Zur Umsetzung der feingranularen Reduktion wird die Funktion AXMEBE aus Alg. 4.1 abgeändert [18]: Die Reduktion in dem kritischen Abschnitt wird entfernt. Stattdessen wird die Reduktion nun feingranular direkt im parallelen Abschnitt durchgeführt. Das wird in Alg. 4.5 für die Reduktion mit atomarer Addition und in Alg. 4.6 für die Reduktion mit Sperren dargestellt. Nach jeder Berechnung eines Elementergebnisvektors y_{el} wird geprüft, welche Elemente des Gesamtergebnisvektors y° verschieden von Null sind. Nur diese werden – synchronisiert durch die atomare Operation bzw. die Sperre – in den gemeinsamen Ergebnisvektor y geschrieben. Dasselbe Prinzip wird auch für eine GPU-Implementierung von AXMEBE benötigt, wie sie in Abschnitt 4.4.1 vorgestellt wird. Da die GPU eine um Größenordnungen höhere Parallelität aufweist als eine Multicore-CPU, ist es unmöglich, für jeden parallelen Thread einen privaten Ergebnisvektor im Speicher anzulegen. Stattdessen wird auf y° verzichtet und direkt in einen gemeinsamen Ergebnisvektor geschrieben.

4.2.3 Experimente

In synthetischen Tests wird außerhalb der Anwendung die Ausführungszeit der verschiedenen Implementierungsvarianten der Reduktionsoperation untersucht. Ebenso wird untersucht, welcher Speed-up für die Ausführungszeit der Funktion AXMEBE mit den verschiedenen Implementierungsvarianten erzielt werden kann. Für die Experimente werden zwei Parallelrechner eingesetzt:

- eine 24-Kern-Maschine mit 4 Intel-Xeon-X5650-CPUs einer Taktfrequenz von 2,67 GHz und mit 12 GiB Speicher sowie

```
1  AXMEBE(A, u)
2  begin
3      y := 0 // gemeinsamer Vektor
4      for each el do in parallel
5          y° := 0
6          A_el := o2el(A, el),   u_el := o2el(u, el)
7          y_el := A_el u_el
8          y° := el2o (y_el)
9          for each element i of y° with y°[i] ≠ 0 do
10             ATOMIC_ADD(&y[i], y°[i])
11         next
12     next
13     return y
14 end
```

Algorithmus 4.5 : Implementierung von AXMEBE mit atomarer Addition

```
1  AXMEBE(A, u)
2  begin
3      y := 0 // gemeinsamer Vektor
4      for each el do in parallel
5          y° := 0
6          A_el := o2el(A, el),   u_el := o2el(u, el)
7          y_el := A_el u_el
8          y° := el2o (y_el)
9          for each element i of y° with y°[i] ≠ 0 do
10             LOCK(i)
11             y[i] := y[i] + y°[i]
12             UNLOCK(i)
13         next
14     next
15     return y
16 end
```

Algorithmus 4.6 : Implementierung von AXMEBE mit expliziten Sperren

- eine 24-Kern-Maschine mit 4 AMD-Opteron-8425-HE-CPUs einer Taktfrequenz von 2,1 GHz und mit 32 GiB Speicher.

Für die Experimente mit der FEM-Implementierung wird das in Abb. 4.1 dargestellte Beispielobjekt *bohrung* [31] verwendet. Es besteht zu Beginn aus 8 Elementen und 32 Knoten.

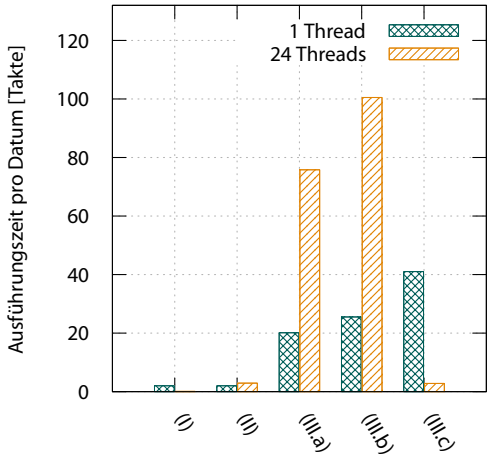
Synthetische Tests

In einem ersten Test wird untersucht, wie viele CPU-Takte der AMD-Rechner benötigt, um parallele Integer-Additionen der Form $a := a + b$ in den folgenden Szenarien durchzuführen:

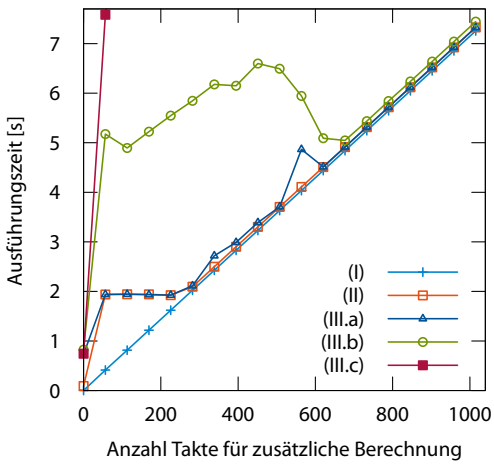
I. a ist in jedem Thread eine private Variable,
II. a ist eine gemeinsame Variable, die unsynchronisiert aktualisiert wird, was möglicherweise zu einem falschen Resultat führt,
III. a ist eine gemeinsame Variable, die synchronisiert aktualisiert wird über
   a. eine atomare Hardwareoperation,
   b. eine atomare Operation, die mit Compare & Swap emuliert wird,
   c. expliziten Sperren.

Für die Messung werden in einer Schleife 100 000 Additionen durchgeführt. Diese Schleife wird einmal von nur einem Thread und einmal von 24 Threads parallel ausgeführt. Die Ausführungszeit der Schleife wird über Hardware-Performance-Counter gemessen, auf die mit der PAPI-Bibliothek [43] zugegriffen wird.

Die Ergebnisse dieses ersten Experiments werden in Abb. 4.3a dargestellt. Der Wert für den einzelnen Thread zeigt, wie viele Takte benötigt werden, um die Addition und, sofern zutreffend, die Synchronisation zusammen durchzuführen. Der Wert für 24 Threads zeigt das Verhalten beim Auftreten von konkurrierenden Zugriffen. Wenn privater Speicher verwendet wird, fällt die Ausführungszeit bei paralleler Ausführung auf $\frac{1}{24}$ des Wertes bei sequentieller Ausführung ab. Wenn der



(a) Anzahl an Takten, die die Addition bei den verschiedenen Implementierungsvarianten benötigt



(b) Ausführungszeit der Additionsoperation mit zusätzlicher Berechnung zwischen zwei Operationen

Abbildung 4.3: Ausführungszeiten der Additionsoperation auf eine einzelne Speicherstelle unter Benutzung von (I) privatem Speicher, (II) unsynchronisierten Zugriffen, (III) synchronisierten Zugriffen mit (III.a) einer atomaren Hardwareoperation zur Addition, (III.b) Compare & Swap sowie (III.c) Sperren

gemeinsame Speicher ohne Synchronisierung verwendet wird, steigt die Ausführungszeit gegenüber jener bei Verwendung privaten Speichers. Das liegt daran, dass der veränderte Speicherinhalt an die Caches aller CPU-Kerne propagiert werden muss, um die Cache-Kohärenz sicherzustellen. Wird die atomare Hardwareoperation zur Addition verwendet, steigt die Ausführungszeit, da die Schreibzugriffe auf das Ergebnis serialisiert werden müssen. Für die mit Compare & Swap emulierte atomare Addition ist die sequentielle Ausführungszeit größer als bei der Hardwareoperation, da für ihre Implementierung mehr Maschinenbefehle benötigt werden. Im Konfliktfall muss ein Thread außerdem mit Busy-Waiting auf die Beendigung der Schreiboperationen der anderen warten, was die parallele Ausführungszeit verlängert. Bei der Verwendung von Sperren treten nur kurze Wartezeiten auf, die in einer Verkürzung der Ausführungszeit auf ca. $\frac{1}{15}$ münden.

In einem zweiten Experiment auf demselben AMD-Rechner wird jeweils zwischen zwei Additionen eine gewisse Menge *zusätzlicher Berechnung* durchgeführt, d. h. die *Intensität* des Codes wird erhöht. Die Intensität eines Codeabschnitts gibt das Verhältnis von Rechen- zu Speicheroperationen an [54]. Die Intensität wird im Experiment schrittweise erhöht, indem die Anzahl der zusätzlich durchgeführten Berechnungen erhöht wird. Die Ergebnisse für die Szenarien (I)–(III.c) werden in Abb. 4.3b dargestellt. Die Kurven zeigen die Ausführungszeit von 24 Threads für die Durchführung von 15 Millionen Additionsoperationen auf einer Variable inklusive der Zeit für die zusätzliche Berechnung. Die Anzahl der Takte der zusätzlichen Berechnung pro Addition ist auf der Abszisse des Diagramms abgetragen. Die Benutzung des privaten Speichers, der keine Synchronisierung erfordert, ergibt im Diagramm eine Gerade. Im Vergleich dazu verursacht sowohl

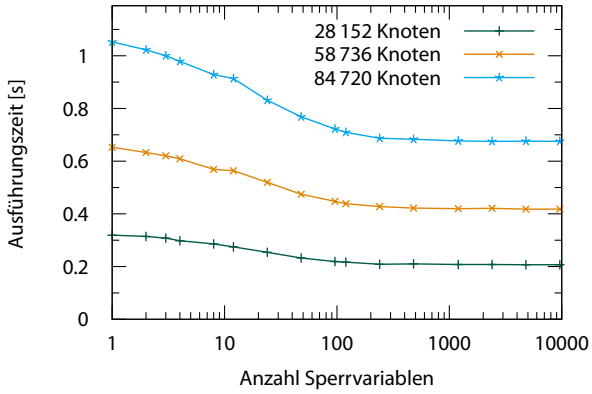


Abbildung 4.4: Ausführungszeit der Funktion PPCGM in Abhängigkeit der Anzahl der Sperrvariablen für verschiedene Feldlängen

der unsynchronisierte als auch der synchronisierte Zugriff auf eine gemeinsame Variable zusätzliche Kosten. Die Ergebnisse zeigen, dass keine substanziellen Zusatzkosten entstehen, wenn die atomare Hardwareoperation mit zusätzlicher Berechnung von mindestens 300 Takten oder Compare & Swap mit mindestens 700 Takten zwischen zwei Additionsoperationen genutzt werden. Im Gegensatz dazu steigt die Ausführungszeit stark an, wenn die explizite Sperre eingesetzt wird.

Aus den Resultaten kann geschlussfolgert werden, dass die atomare Hardwareoperation zur Reduktion genutzt werden sollte. Wenn diese für den benötigten Datentyp nicht existiert, wie bei der untersuchten FEM, die mit Gleitkommazahlen doppelter Genauigkeit rechnet, kann auf die Emulation mit Compare & Swap ausgewichen werden. Vergehen zwischen zwei Zugriffen auf eine Speicherstelle mindestens 300 CPU-Takte, ergibt sich im Falle der atomaren Hardwareoperation kein zusätzlicher Zeitaufwand durch die Synchronisation. Im Falle der Emulation mit Compare & Swap beträgt der Mindestabstand 700 Takte.

Feingranulare explizite Sperren

Für die in Abschnitt 4.2.1 vorgestellte Methode, feingranulare Sperren zu verwenden, wird untersucht, wie sich die Ausführungszeit der Funktion PPCGM bei Veränderung der Anzahl der Sperrvariablen verhält. Als Eingabe für die FEM wird wieder das Objekt *bohrung* verwendet. Abbildung 4.4 zeigt die Ausführungszeit für drei Verfeinerungsschritte der FEM, bei denen das Objekt aus 28 152, 58 736 bzw. 84 720 Knoten besteht und die Ergebnisvektoren eine Länge von 84 456, 176 208 bzw. 254 160 haben. Für alle Feldlängen sinkt die Ausführungszeit mit steigender Anzahl der Sperrvariablen bis ca. 240 rasch ab und bleibt danach etwa konstant. Zur Minimierung des Speicherverbrauches erscheint es also sinnvoll, ca. 240 Sperrvariablen zu verwenden.

4.2.4 Reduktionsoperationen in der FEM-Implementierung

Zur Evaluation der Ausführungszeiten wird die Reduktionsoperation in AXMEBE in den folgenden Varianten implementiert:

(a) als grobgranulare Reduktion nach Alg. 4.1
   – mit einem verteilten Speichermodell in MPI (MVAPICH2 1.5.1),
   – mit einem gemeinsamen Speichermodell in OpenMP (GNU Fortran 4.4.7),
(b) als feingranulare Reduktion
   – mit der atomaren Additionsoperation unter Verwendung von Compare & Swap nach Alg. 4.5,
   – mit expliziten Sperren nach Alg. 4.6,
   – ohne Synchronisation (unter Inkaufnahme falscher Ergebnisse).

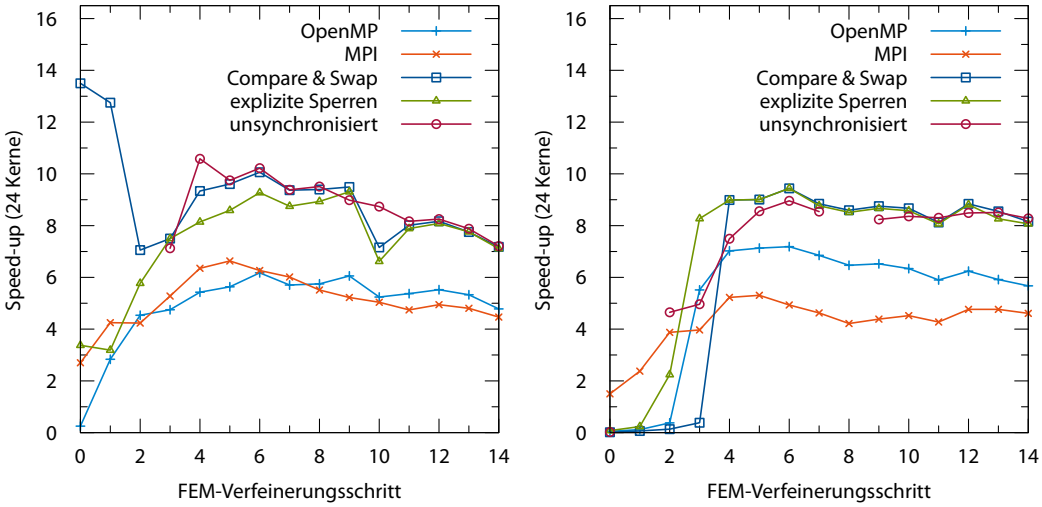
Da auf der genutzten Hardware kein Maschinenbefehl zur atomaren Addition von Gleitkommazahlen doppelter Genauigkeit existiert, wird diese Variante nicht untersucht. Die Experimente werden mit dem Beispielobjekt *bohrung* auf dem Intel- und dem AMD-Rechner durchgeführt. Die Ergebnisse sind in Abb. 4.5 dargestellt. Die Ausführungszeit der OpenMP-Variante, die mit dem GNU-Fortran-Compiler erzielt wird, unterscheidet sich nicht signifikant von Ergebnissen, die mit dem kommerziellen Fortran-Compiler von Intel erzielt werden.

Mit der parallelen Implementierung der Funktion PPCGM, die sowohl sequentielle als auch parallele Anteile enthält, wird mit den Varianten, die grobgranulare Reduktion nutzen, ein Speed-up zwischen 4 und 6 erzielt. Das gewählte Speichermodell hat keinen Einfluss auf den Speed-up. Für die Varianten mit feingranularer Reduktion ist der Speed-up etwas höher: Bei Nutzung von Compare & Swap oder expliziten Sperren beträgt er etwa 8. Das entspricht dem Wert, der erreicht wird, wenn die Addition unsynchronisiert durchgeführt wird. Die Ergebnisse zeigen, dass mit der feingranularen Reduktion zur Verhinderung von Speicherzugriffskonflikten keine zusätzliche Ausführungszeit im Vergleich zur unsynchronisierten Ausführung benötigt wird. Offenbar sind die Zeitintervalle zwischen zwei Zugriffen auf ein Element groß genug, um die Reduktionsoperation zwischen den Berechnungen zu verstecken, wie es in Abschnitt 4.2.1 gezeigt wurde. Die Intensität des Codeabschnitts, der die Berechnung durchführt, ist offenbar groß genug, dass seine Ausführungszeit von der Rechengeschwindigkeit der CPU beschränkt wird, d. h. er ist *compute bound*. Die Intensität der grobgranularen Reduktion hingegen ist sehr niedrig, sodass ihre Ausführungszeit durch die Speichergeschwindigkeit beschränkt wird, sie ist also *memory bound*. Die Wartezeit, die bei Verwendung grobgranularer Reduktion am Ende jedes parallelen Abschnittes entsteht, kann folglich durch die Verwendung von feingranularer Reduktion nahezu vollständig eliminiert werden.

4.2.5 Schlussfolgerungen

In diesem Unterkapitel wurden verschiedene Varianten der Implementierung der Reduktion von Feldern auf Systemen mit verteiltem Speicher untersucht. Feingranulare Implementierungen der Reduktion mit atomaren Operationen bzw. mit Sperren wurden mit grobgranularen Implementierungen in OpenMP und MPI verglichen.

Die Ergebnisse zeigen, dass die Synchronisation der Schreibzugriffe auf einen gemeinsamen Vektor keine zusätzliche Zeit benötigt, wenn die Zeitspanne zwischen den Schreibzugriffen groß genug ist. In der untersuchten Routine AXMEBE haben die lokalen Vektoren nur an wenigen Stellen Nichtnull-Einträge und die Schreibzugriffe auf den gemeinsamen Vektor erfolgen direkt nach der Berechnung, sodass oben genannte Bedingung erfüllt ist. Dadurch wird die Ausführungszeit der Routine PPCGM deutlich verkürzt. Die Ergebnisse zeigen, dass im Gegensatz zu anderen Arbeiten, in denen meist Feldprivatisierung empfohlen wird, z. B. [82] und [169], durch Verwendung der fein-



(a) AMD-Rechner **(b)** Intel-Rechner

Abbildung 4.5: Speed-up der Routine PPCGM für verschiedene Implementierungsvarianten der Reduktionsoperation in der Routine AXMEBE

granularen Reduktion auch das Gegenteil, nämlich das direkte Schreiben auf einen gemeinsamen Vektor, effizient sein kann.

4.3 NUMA-Awareness für Datenverteilung

Heutige Multicore-Systeme mit gemeinsamem Speicher weisen häufig eine *NUMA-Architektur* auf. *NUMA* steht für *non-uniform memory access (ungleichförmiger Speicherzugriff)* und bedeutet, dass die Latenz und die Übertragungsrate eines Speicherzugriffs davon abhängen, in welchem Teil des Speichers das Datum steht [21]. NUMA-Architekturen entstehen, wenn das System aus mehreren CPUs besteht, die jeweils einen eigenen Speichercontroller besitzen [158], und dadurch an jede CPU ein Teil des Speichers angebunden ist. Das Betriebssystem verbirgt diesen Umstand meist, sodass der Speicher als ein großer, gemeinsamer Speicher erscheint [238, Kap. 9.8.7.5]. Erfolgt nun von einer CPU ein Zugriff auf ein Datum, das im Speicher einer anderen CPU abgelegt ist, muss dieser erst den Weg durch das Speicher-Interconnect-System nehmen [158]. Ein solcher Zugriff auf entfernten Speicher hat bei heute typischen Systemen zirka die doppelte Latenz eines Zugriffs auf lokal angebundenen Speicher [158]. Das hier verwendete System besteht aus 2 CPUs mit jeweils 6 Kernen, also auch aus 2 NUMA-Knoten. Die CPUs sind vom Typ Intel Xeon X5650 mit einer Taktfrequenz von 2,67 GHz und jeweils 12 MiB Level-3-Cache. Sie verfügen über Hyperthreading-Funktionalität, sodass auf jeder CPU 6 Threads parallel ausgeführt werden können.

In diesem Unterkapitel wird untersucht, welchen Einfluss die Zuordnung zwischen NUMA-Knoten der Daten und verarbeitender CPU auf die Ausführungszeit der CGM hat. Die effizienteste

Ausführung ist zu erwarten, wenn die Daten jeweils von einem Kern der CPU in dem NUMA-Knoten verarbeitet werden, in dessen Speicher sie auch liegen. Maximale Ineffizienz hingegen ist zu erwarten, wenn die Daten immer gerade auf der anderen CPU verarbeitet werden [160].

4.3.1 Implementierung

Für das Betriebssystem Linux existiert die Bibliothek *libnuma* [153], die ein benutzerfreundliches Programmierinterface für die NUMA-Funktionalität des Linux-Kernels bietet. In der Implementierung der Anwendung wird über die *libnuma* der Speicher für die Steifigkeitsmatrix und alle weiteren Datenstrukturen mit der NUMA-Policy *interleaved* angefordert. Das heißt, die Speicherseiten werden reihum den NUMA-Knoten zugeordnet [153].

Die NUMA-gerechte Implementierung der CGM wird in Alg. 4.7 im Pseudocode dargestellt: Noch vor dem ersten Aufruf der Routine PPCGM erfolgt die Initialisierung: Hier werden die Threads den NUMA-Knoten zugeteilt. Alle Threads, die auf den CPUs eines NUMA-Knotens laufen, bilden eine NUMA-Gruppe. Jedem Thread wird in seiner Gruppe wieder eine eindeutige Nummer zugewiesen.

In der Routine PPCGM werden nach der Vorkonditionierung und vor dem Aufruf der Funktion AXMEBE, also zwischen Zeilen 5 und 6 in Alg. 4.1, die Elementlisten der NUMA-Gruppen aufgebaut: Für jedes Element des Gesamtelementfeldes wird ermittelt, auf welchem NUMA-Knoten es abgespeichert ist. Dazu wird zunächst dessen logische Speicheradresse bestimmt, die an die Funktion `move_pages` der *libnuma* übergeben wird. Eigentlich dient die Funktion dazu, Speicherbereiche auf einen bestimmten NUMA-Knoten zu verschieben. Wird für den Zielknoten jedoch der Nullzeiger übergeben, ist ihr Rückgabewert die Nummer des NUMA-Knotens, auf dem die übergebene Speicheradresse liegt. Die für jeden NUMA-Knoten erzeugte *NUMA-Gruppenliste* enthält die Information, welche Elemente in ihm abgespeichert sind. Diese Liste führt die *NUMA-Awareness* ein, also das Bewusstsein des Algorithmus darüber, auf welchen NUMA-Knoten die Daten liegen.

Der parallele Abschnitt der Funktion AXMEBE wird mit einem Thread pro CPU-Kern ausgeführt. Die in der Initialisierung belegten Variablen mit den Informationen über die Gruppenzugehörigkeit etc. werden hier weiter genutzt. Zu Beginn wird für jeden Thread festgelegt, dass er auf dem NUMA-Knoten seiner Gruppe ausgeführt wird. Dazu dient die von *libnuma* bereitgestellte Funktion `numa_run_on_node`. Durch diese Funktion wird außerdem sichergestellt, dass die Threads nicht zufällig auf eine andere CPU migrieren. Die parallele Schleife über alle Elemente in Zeile 13 des Alg. 4.1 wird nun dahingehend modifiziert, dass auf jeder CPU nur die Elemente auf die Threads verteilt werden, die auch auf diesem NUMA-Knoten abgespeichert sind. Das Element an Stelle i in der NUMA-Gruppenliste des Knotens wird von dem Thread verarbeitet, dessen Threadnummer sich beim Teilen von i durch die Anzahl der Threads auf diesem Knoten als Rest ergibt. Die eigentliche Verarbeitung der Elemente erfolgt wie in Alg. 4.1 angegeben.

Die vorgestellte Methode minimiert den Datenverkehr, indem sie sicherstellt, dass keine der Daten, für die NUMA-Awareness hergestellt wurde, zwischen den CPUs übertragen werden. Diese Methode wird verglichen mit einer zweiten Methode, die den Datenverkehr maximiert, indem sie jedem Element gerade jene CPU zur Verarbeitung zuweist, die zum anderen NUMA-Knoten gehört. Die Daten werden also absichtlich auf dem jeweils falschen Knoten verarbeitet. Die Differenz der so ermittelten Werte entspricht dem maximalen Gewinn, der durch die Einführung der NUMA-

```
   // Initialisierung:
1  paralleler Abschnitt
2      T_global := Anzahl der laufenden Threads
3      t_global := Nummer des ausführenden Threads
4      G := Anzahl der NUMA-Knoten
5      T_gruppe := T_global div N // Anzahl Threads in einer NUMA-Gruppe
6      g := t_global div T_gruppe // NUMA-Gruppe des Threads
7      t_gruppe := t_global mod T_gruppe // Nummer des Threads innerhalb der NUMA-Gruppe

   // PPCGM:
8  ...
9  Aufruf des Vorkonditionierers, Berechnung von u
10 iteriere g := 0 ... G
11     erzeuge leere Liste gruppenelementliste[g]
12 iteriere Elemente el ∈ elementfeld
13     numa_knoten := ermittle NUMA-Knoten von Adresse elementfeld[el]
14     füge Elementnummer von el ans Ende der Liste gruppenelementliste[numa_knoten] ein
15 Aufruf von AXMEBE
16 ...

   // AXMEBE:
17 paralleler Abschnitt
18     führe Thread auf NUMA-Koten g aus
19     n_el,gruppe := Länge der Liste gruppenelementliste[g]
20     iteriere über k = 0 ... n_el,gruppe div T_gruppe
21         elnr := hole Element k · T_gruppe + t_gruppe aus gruppenelementliste[g]
22         el := Element mit der Nummer elnr
23         verarbeite el wie in Alg. 4.1
```

Algorithmus 4.7 : Modifikation von PPCGM und AXMEBE zur NUMA-gerechten Verarbeitung der finiten Elemente

Awareness für diese Daten erzielt werden kann.

4.3.2 Experiment

Im Experiment wurden nun die Ausführungszeit und der Energieverbrauch für das Testobjekt *bohrung* im 11. Verfeinerungsschritt mit 9584 Elementen gemessen. Für die Ermittlung des Energieverbrauchs wurde RAPL verwendet. Der Energieverbrauch der CPUs wurde über das Register MSR_PKG_ENERGY_STATUS und der Energieverbrauch für den Speicher wurde über das Register MSR_DRAM_ENERGY_STATUS ausgelesen.

Die Ergebnisse sind inklusive der 95-%-Konfidenzintervalle in Tab. 4.1 angegeben. Die Unterschiede zwischen beiden Implementierungen können als nicht signifikant angesehen werden, da sich die Konfidenzintervalle für die korrekte und die falsche Zuordnung überlappen. Die Ausführungszeit ändert sich möglicherweise deshalb nicht signifikant, weil bei der Entwicklung des Programms großer Wert auf Cache-Freundlichkeit gelegt wurde. Sind die Daten einmal im Cache vor-

	Zuordnung NUMA-Knoten zu CPU	
	korrekt	**falsch**
Ausführungszeit	(1,005 ± 0,029) s	(1,012 ± 0,011) s
Energie Package	(72,5 ± 2,1) J	(73,2 ± 1,0) J
Energie DRAM	(8,21 ± 0,28) J	(8,08 ± 0,11) J

Tabelle 4.1: Ausführungszeit und Energieverbrauch für die Ausführung von `PPCGM` mit korrekter und mit absichtlich fehlerhafter Zuordnung zwischen NUMA-Knoten und CPU

handen, macht es keinen Unterschied mehr, ob sie vorher im lokalen oder im entfernten NUMA-Knoten abgelegt waren. Beim Energieverbrauch werden keine Unterschiede sichtbar, weil möglicherweise die für die Berechnungen notwendige Energie deutlich größer ist als die für die Datenübertragung notwendige Energie. Für den DRAM ergibt sich keine signifikante Differenz, weil es für dessen Energieverbrauch keinen Unterschied machen dürfte, ob die Daten direkt an die anbindende CPU ausgeliefert werden oder ob diese sie dann an eine andere CPU weitersendet.

Die vorgestellte Methode erzielt also weder bei der Ausführungszeit noch beim Energieverbrauch einen Vorteil. Wenn es schon nicht zwischen den beiden extremen Implementierungen einen signifikanten Unterschied gibt, wird auch gegenüber einer NUMA-agnostischen Implementierung kein Vorteil zu erzielen sein.

4.3.3 Diskussion

Die vorgestellte Methode vermindert den Speicherverkehr zwischen den NUMA-Knoten, indem sie die Daten auf demselben NUMA-Knoten verarbeitet, auf dem sie auch abgespeichert sind. Die Methode bringt allerdings beim untersuchten Beispiel keinen Erfolg, stattdessen erhöht sie lediglich die Komplexität des Quelltextes. Es erscheint derzeit wenig sinnvoll, den Ansatz weiterzuverfolgen. Daher wird die Implementierung verworfen. Teilweise kann sie jedoch weiterverwendet werden: Die aufgestellten Listen eignen sich ebenso, um die Elemente in solche, die auf der CPU verarbeitet werden, und solche, die auf der GPU verarbeitet werden, zu unterteilen. Das wird in Abschnitt 4.4 genutzt.

Trotzdem könnte die NUMA-Awareness künftig noch eine Bedeutung bekommen: Lameter erwartet, dass kommende Prozessorgenerationen wachsende Unterschiede in der Zugriffszeit zwischen lokalen und entfernten NUMA-Knoten aufweisen werden [158]. Für entfernte NUMA-Knoten wird ein Absinken der Datenrate auf $\frac{1}{4}$ bis $\frac{1}{8}$ gegenüber dem lokalen NUMA-Knoten erwartet [239]. Möglicherweise werden die Unterschiede zwischen den verschiedenen Implementierungen dann signifikant. Nach Demmel und Gearhart wird der Energieverbrauch des Speicherverkehrs auf einem Knoten die größte Komponente des Energieverbrauchs für viele Algorithmen werden [62], sodass das vorgestellte Schema zur Minimierung des Speicherverkehrs möglicherweise zu einem späteren Zeitpunkt zur Steigerung der Energieeffizienz benötigt wird.

4.4 Energie- und Ausführungszeitmodell

In diesem Unterkapitel wird ein anwendungsspezifisches Energie- und Ausführungszeitmodell für die CGM aufgestellt. Dieses soll es ermöglichen, den Energieverbrauch bzw. die Ausführungszeit der CGM zu minimieren und damit eine energiebewusste Ausführung zu erreichen. Dazu wird zunächst die vorhandene CPU-Implementierung der CGM auf Grafikprozessoren (GPUs) übertragen. Für viele Probleme wurde gezeigt, dass GPUs sie schneller [113, 83, 200] oder energieeffizienter [1, 121, 231] lösen.

Beide Typen von Verarbeitungseinheiten, CPUs und GPUs, sind gleichermaßen für die Ausführung der CGM geeignet. Die Möglichkeit, die Berechnung frei zwischen den Verarbeitungseinheiten verschieben zu können, eröffnet die Möglichkeit, die schnellste bzw. energieeffizienteste Art der Ausführung zu wählen. In Abhängigkeit von den Merkmalen des Rechnersystems ist die beste Verteilung der Rechenlast eine der folgenden:

1. Verarbeitung der gesamten Rechenlast auf der CPU,
2. Verarbeitung der gesamten Rechenlast auf der GPU,
3. gemeinsame CPU-GPU-Verarbeitung, wobei jede Verarbeitungseinheit einen Teil der Rechenlast verarbeitet.

Das aufgestellte Energie- und Ausführungszeitmodell wird anhand verschiedener Experimente aufgestellt und berücksichtigt die Ausführungsgeschwindigkeiten von CPU und GPU, deren elektrische Leistung, Spannungs- und Frequenzregelung sowie den Energieverbrauch und die benötigte Zeit für die Datenübertragung zwischen Haupt- und Grafikspeicher.

4.4.1 Implementierung

Die in Alg. 4.1 dargestellte Implementierung der CGM wird erweitert, sodass auch die GPU zur Ausführung der Funktion AXMEBE mit herangezogen wird. Der parallele Abschnitt in den Zeilen 13 bis 18 des Alg. 4.1 ist sowohl für die Ausführung auf der CPU als auch auf der GPU geeignet. Durch die parallele Ausführung auf beiden Verarbeitungseinheiten wird die hohe Rechenleistung der GPU ausgenutzt, ohne die CPU währenddessen im Leerlauf zu lassen.

Die Implementierung wird in Alg. 4.8 gezeigt. Für die parallele Ausführung auf p CPU-Kernen und auf der GPU wird die Rechenlast W in disjunkte Mengen – für jede Verarbeitungseinheit eine – aufgeteilt (Zeile 3). W besteht aus n_{el} zu verarbeitenden finiten Elementen. $W_{\text{cpu}k}$ bezeichnet die Rechenlast, die auf dem k-ten CPU-Kern verarbeitet wird, $1 \leq k \leq p$. W_{gpu} bezeichnet die auf der GPU verarbeitete Rechenlast. Die Daten, die auf der GPU benötigt werden, werden vor der Ausführung dorthin übertragen und die Ergebnisse nach Beendigung der Ausführung in den Hauptspeicher zurück übertragen. Der Lastvektor u und der Ergebnisvektor y_{gpu} werden für jeden Aufruf von AXMEBE übertragen (Zeilen 20 und 29). Für die Elemente el, die auf der GPU verarbeitet werden, werden bestimmte Daten einmal pro PPCGM-Aufruf übertragen. Das sind die Teile der Steifigkeitsmatrix A und die Permutationsvektoren, die zur Erzeugung der entsprechenden Elementsteifigkeitsmatrizen A_{el} benötigt werden (Zeilen 5 und 6).

Die Routine AXMEBE wird von mehreren Threads ausgeführt. Jeder dieser Threads besitzt eine eindeutige Nummer tid, $0 \leq tid \leq p$. Der erste Thread mit $tid = 0$ (Zeile 19) ist verantwortlich dafür, u und y_{gpu} in den Grafikspeicher zu übertragen und die GPU-Kernelfunktionen zu starten. Die

```
1  PPCGM(A, b)
2  begin
3      distribute all el ∈ W to W_gpu and W_cpuk, 1 ≤ k ≤ p
           such that |W_gpu| = n_el,gpu and |W_cpuk| = n_el,cpu
4      for each el ∈ W_gpu do
5          copy part of A for el to GPU memory          | t_copy
6          copy permutation data to GPU memory          | E_copy
7      next
8      repeat
9          call preconditioner                          | t_pre
10         calculate next u                             | E_pre
11         y := AXMEBE_CPU_GPU(A, u)
12     until y ≈ b
13     return u
14 end

15 AXMEBE_CPU_GPU(A, u)
16 begin
17     begin parallel
18         tid := ID of current thread, 0 ≤ tid ≤ p
19         if tid == 0 then
20             transfer u to GPU memory
21             for each el ∈ W_gpu do (on GPU)
22                 A_el := o2el(A, el)
23                 u_el := o2el(u, el)
24                 y_el := A_el u_el                    | t_gpu
25                 begin critical section               | E_gpu
26                     y_gpu := y_gpu + el2o(y_el)
27                 end critical section
28             next
29             transfer y_gpu to main memory
30         else
31             for each el ∈ W_cputid do
32                 A_el := o2el(A, el)
33                 u_el := o2el(u, el)
34                 y_el := A_el u_el                    | t_cpu
35                 begin critical section               | E_cpu
36                     y_cpu := y_cpu + el2o(y_el)
37                 end critical section
38             next
39         end
40     end parallel
41     return y_cpu + y_gpu
42 end
```

Algorithmus 4.8 : Pseudocode für PPCGM und AXMEBE in der CPU-GPU-Implementierung

weiteren p Threads führen die Berechnungen auf den CPU-Kernen durch. Am Ende von AXMEBE werden die beiden Ergebnisvektoren y_{cpu} und y_{gpu} addiert und an PPCGM zurückgegeben.

Für die Vorhersage des Energieverbrauchs und der Ausführungszeit wird der Code in verschiedene Codeabschnitte aufgeteilt. Diese werden durch die vertikalen Linien in Alg. 4.8 dargestellt. Die daneben stehenden Formelzeichen stehen für die Ausführungszeit bzw. den Energieverbrauch der jeweiligen Abschnitte: t_{copy} bezeichnet die Zeit, die benötigt wird, um die Daten in den Grafikspeicher zur übertragen, t_{pre} bezeichnet die für die Vorkonditionierung und die Berechnung des nächsten Approximation benötigte Zeit, t_{gpu} bezeichnet die Zeit, die für die Verarbeitung eines Elementes auf der GPU benötigt wird, und t_{cpu} bezeichnet die Zeit, die für die Verarbeitung eines Elements auf den p CPU-Kernen benötigt wird. Der Energieverbrauch dieser Abschnitte wird entsprechend mit E_{copy}, E_{pre}, E_{gpu} und E_{cpu} bezeichnet.

In Abschnitt 4.4.2 werden Gesetze aufgestellt, die das Verhalten der Größen in Abhängigkeit von Parametern wie die Anzahl verarbeiteter Elemente oder die Taktfrequenz der CPU beschreiben. Anhand dieser Untersuchungen wird in Abschnitt 4.4.3 das anwendungsspezifische Modell für die CGM aufgestellt. Außerdem wird eine Fallunterscheidung entwickelt, anhand der sich mit dem

Modell vorhersagen lässt, wie die CGM am energieeffizientesten ausgeführt werden kann.

4.4.2 Experimente

Für eine möglichst genaue Vorhersage müssen Gesetzmäßigkeiten gefunden werden, die das Verhalten der Größen t_{cpu}, E_{cpu}, t_{gpu}, E_{gpu}, t_{copy} und E_{copy} beschreiben. Außerdem wird die Größe E_{dram} für die Energie, die der DRAM-Speicher während der Verarbeitung auf der CPU benötigt, eingeführt. Die zu findenden Gesetze sollten die Anzahl der auf den jeweiligen Verarbeitungseinheiten verarbeiteten finiten Elemente sowie weitere Merkmale der Daten und Hardwareeigenschaften berücksichtigen. Anhand der in diesem Abschnitt beschriebenen Experimente werden diese Gesetze aufgestellt. Die Größen t_{pre} und E_{pre} werden nicht berücksichtigt, da sie keinen Einfluss auf die Verteilung der Rechenlast auf die Verarbeitung haben.

Parallel zu den Experimenten werden Messmethoden untersucht, die zur Durchführung der Experimente notwendig sind. Zur Verifikation der Messmethoden werden die Messergebnisse verglichen mit Werten, die aus anderen Quellen erhalten wurden. Alle Messmethoden nutzen nur Softwareschnittstellen, es wurde keine zusätzliche Messhardware für die Messungen verwendet.

Experimentieranordnung

Die Experimente wurden auf zwei unterschiedlichen Rechnern durchgeführt, *Westmere* und *Sandybridge*. Ihre technischen Daten werden in Tab. 4.2 aufgeführt. Beim Rechner *Westmere* handelt es sich um einen 2×6-Kern-Xeon-Rechner mit einer GPU vom Typ GeForce GTX 570 HD. Beim Rechner *Sandybridge* handelt es sich um einen 2×4-Kern-Xeon-Rechner mit einer GPU vom Typ Tesla C2075. Die Experimente zur Messung der Ausführungszeit wurden auf beiden Maschinen ausgeführt. Da nur im Rechner *Sandybridge* per Software auslesbare Energie- bzw. Leistungsmesswerte verfügbar sind, wurden die Experimente zur Untersuchung des Energieverbrauchs nur auf diesem Rechner durchgeführt.

Zusätzlich zum Objekt *bohrung*, das in Abb. 4.1 dargestellt wurde, werden hier die Objekte *c8zyl* und *kurbel3f* verwendet, die ebenfalls aus der mit [31] bereitgestellten Bibliothek stammen und in Abb. 4.6 dargestellt werden. Die Eingabeparameter wurden so gewählt, dass jedes finite Element der Beispielobjekte aus 27 Knoten besteht, die jeweils 3 Freiheitsgrade haben, was eine Größe der Elementdatenstrukturen von 81 ergibt.

Alle Messungen wurden mit den gezeigten Testobjekten durchgeführt. Das heißt, dass alle Effekte wie falsche Sprungvorhersage, Cache-Misses usw., die auch im produktiven Einsatz der FEM auftreten, in die Messungen einbezogen werden. Die Funktionen zur Messung der aktuellen Zeit bzw. des Energieverbrauchs werden unmittelbar vor den Codeabschnitten gestartet und unmittelbar danach gestoppt. Wenn sowohl Energie als auch Zeit gemessen werden, wird die Energiemessung nach der Zeitmessung gestartet und vor ihrem Ende gestoppt. Während der Ausführung der FEM wurden keine anderen Anwendungen auf dem Rechner laufen gelassen, um Störeinflüsse zu vermeiden. In einigen Diagrammen mit Messergebnissen werden Regressionskurven dargestellt. Die Regression wurde jeweils im Programm gnuplot [88] mit der Methode der kleinsten Quadrate nach Levenberg und Marquardt durchgeführt.

Rechner	**Westmere**	**Sandybridge**
CPU-Modell	**Intel Xeon X5650**	**Intel Xeon E5-2650**
Taktrate der CPU	2,67 GHz	2,0 GHz
Anzahl CPUs, Kerne, Threads	$2 \times 6 \times 2 = 24$	$2 \times 4 \times 2 = 16$
CPU-Peak-Performance	64 Gflop/s	128 Gflop/s
Thermal Design Power der CPU	95 W	95 W
RAM-Modell	Samsung M393B5670EH1-CH9	Samsung M393B1K70DH0
Anzahl DIMMs, DIMM-Größe	6×2 GiB = 12 GiB	4×8 GiB = 32 GiB
RAM-Geschwindigkeit	1333 MHz	1600 MHz
NUMA-Knoten	2	2
GPU-Modell	**Nvidia GeForce GTX 570 HD**	**Nvidia Tesla C2075**
GPU-Speicher	1,2 GiB	6,0 GiB
GPU-Speichergeschwindigkeit	1,9 GHz	1,5 GHz
CUDA-Kerne	480	448
Taktfrequenz der CUDA-Kerne	1,46 GHz	1,15 GHz
GPU-Peak-Performance	176 Gflop/s	515 Gflop/s
Speicherübertragungsrate	152 GB/s	208 GB/s
Thermal Design Power der GPU	219 W	215 W
Betriebssystem	Linux, Kernel 3.2.21	Linux, Kernel 3.2.46
Version des GPU-Treibers	304.48	304.88
CUDA-Version	4.2	5.0

Tabelle 4.2: Technische Daten der für die Experimente verwendeten Rechner [132, 131, 205, 126, 127, 203]

Messmethoden

Für die CPU und die GPU sind verschiedene Messmethoden notwendig, da sie über verschiedene Schnittstellen zum Auslesen der Zeit-, Energie- und Leistungsmesswerte verfügen. Diese Methoden werden hier beschrieben.

CPU Die Ausführungszeiten wurden gemessen, indem der hochaufgelöste Zeitstempel TSC [247] der CPU ausgelesen wurde. Dafür wurde die Funktion `PAPIF_GET_VIRT_USEC` der PAPI-Bibliothek [43] verwendet, die den `clock_gettime`-Systemruf [150, S. 491] nutzt.

Für die Messung des Energieverbrauchs wurden die maschinenspezifischen Register der RAPL-Technologie von Intel [60] ausgelesen, vgl. Abschnitt 2.2.2. Hier wurde die *Package*-Energie, also das Register `MSR_PKG_ENERGY_STATUS`, für die Berechnungsenergie genutzt. Die damit ermittelte Energie enthält auch die der Caches, der Uncore-Energie usw. und umfasst so alle Komponenten, die für die Berechnung notwendig sind. Für die Messung des Speicherenergieverbrauchs wurde das Register `MSR_DRAM_ENERGY_STATUS` genutzt.

GPU Die Ausführungszeiten auf der GPU werden in derselben Art und Weise wie für die CPU beschrieben gemessen. Die Zeitmessung wurde unmittelbar vor der Datenübertragung zum GPU-Speicher (Zeile 20 in Alg. 4.8) gestartet und nach der Datenübertragung vom GPU-Speicher (Zeile 29) gestoppt. Die Zeitmessfunktionen der GPU werden nicht verwendet, da es hier nicht das Ziel ist, den GPU-Code zu optimieren, sondern eine Auswertung der Ausführungszeiten von AXMEBE auf der GPU aus der CPU-Perspektive zu erhalten.

Für die Messung des Energieverbrauchs auf der GPU wird das integrierte Leistungsmessgerät

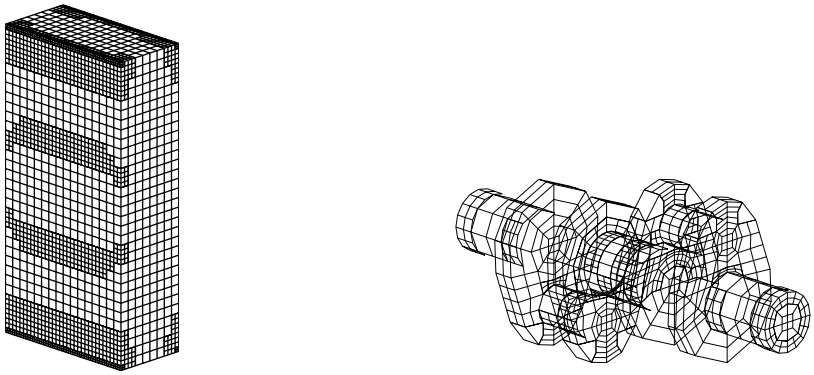


Abbildung 4.6: Beispielobjekte *c8zyl* und *kurbel3f* jeweils nach 8 Verfeinerungsschritten [31]

verwendet und über die Funktion `nvmlDeviceGetPowerUsage` der NVML-Bibliothek ausgelesen [207], vgl. Abschnitt 2.2.2. Da die Leistungsmessung nur alle 20 ms aktualisiert wird, was für die benötigten Messungen nicht ausreicht, wird das Offline-Verfahren der in Kapitel 5 vorgestellten Methode verwendet. Dieses Verfahren führt die zu untersuchende Routine mehrfach (ca. 50- bis 100-mal) zu zufälligen Phasen im Messintervall aus, sodass die Leistungsmessung zu verschiedenen Zeitpunkten während der Ausführung stattfindet. Durch Integration der Leistungsmesswerte über die Ausführungszeit lässt sich der Energieverbrauch ermitteln.

Ausführungszeit und Energie in Abhängigkeit von der Elementanzahl

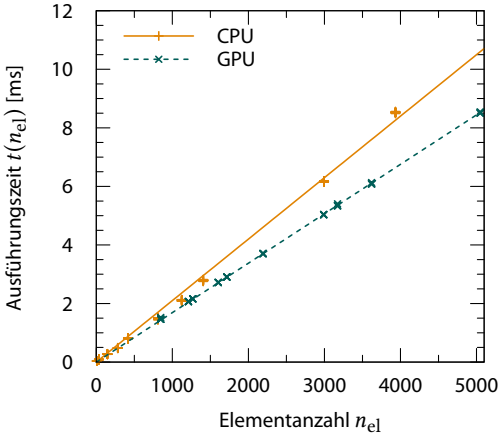
Zunächst werden die Ausführungszeit und der Energieverbrauch der Routine AXMEBE untersucht. Abbildung 4.7a zeigt die Ausführungszeit von AXMEBE auf dem Rechner *Sandybridge* in Abhängigkeit von der Anzahl verarbeiteter finiter Elemente. Für die Messungen wurden die Objekte *bohrung* und *c8cyl* in verschiedenen Verfeinerungsstufen verwendet. Jeder Punkt im Diagramm entspricht einem Experiment mit einer Elementanzahl n_{el}. Für jede Elementanzahl wurde das Experiment fünfmal durchgeführt. Die Regressionsgeraden wurden in der Form gewählt $t_{\mathrm{cpu}}(n_{\mathrm{el}}) = {v_{\mathrm{ex}}}^{-1} n_{\mathrm{el}}$, wobei v_{ex} die *Ausführungsgeschwindigkeit* als „Anzahl pro Zeiteinheit verarbeiteter Elemente“ bezeichnet. Die Resultate zeigen, dass sowohl für die CPU als auch für die GPU die Ausführungszeiten t_{cpu} und t_{gpu} von AXMEBE proportional zu n_{el} sind:

$$t \sim n_{\mathrm{el}} \quad . \tag{4.6}$$

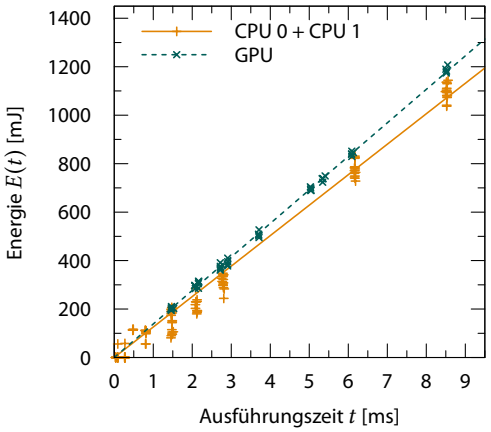
Die Proportionalitätsfaktoren sind die jeweiligen Ausführungsgeschwindigkeiten auf der CPU und auf der GPU:

$$v_{\mathrm{ex,cpu}} = 587 \, \frac{\mathrm{el}}{\mathrm{ms}} \quad \text{und} \quad v_{\mathrm{ex,gpu}} = 592 \, \frac{\mathrm{el}}{\mathrm{ms}} \quad . \tag{4.7}$$

Die CPU-Kerne von *Sandybridge* verarbeiten in einem gegebenen Zeitraum etwa dieselbe Anzahl an Elementen wie die GPU. Die jeweiligen Ausführungszeiten für ein Element auf der CPU bzw. der GPU sind $t_{\mathrm{el,cpu}} = 2{,}10\ \mu\mathrm{s}$ und $t_{\mathrm{el,gpu}} = 1{,}69\ \mu\mathrm{s}$.



(a) Ausführungszeiten t_{cpu} und t_{gpu} in Abhängigkeit von der Elementanzahl n_{el}



(b) Energieverbrauch E_{cpu} und E_{gpu} in Abhängigkeit von der Ausführungszeit t_{cpu} und t_{gpu}

Abbildung 4.7: Ausführungszeit und Energieverbrauch von AXMEBE bei der Ausführung auf CPU und GPU

Abbildung 4.7b zeigt die Messergebnisse der Energiemessung aus demselben Experiment. Da der Rechner zwei CPUs enthält, werden die für beide CPUs ermittelten Werte der Package-Energie aufsummiert. Das Ergebnis zeigt, dass auch der Energieverbrauch E etwa proportional zur Ausführungszeit ist t_{ex}:

$$E \sim t \quad . \tag{4.8}$$

Der Anstieg der Regressionsgeraden entspricht der Leistungsaufnahme der Prozessoren: $E(t) = Pt$. Für die Leistung wurden folgende Werte ermittelt: $P_{ex,cpu} = 126$ W für die CPUs zusammen und $P_{ex,gpu} = 138$ W für die GPU. Diese Werte entsprechen jeweils ca. ⅔ der Thermal Design Power von 95 W pro CPU [126] und 215 W für die GPU [206], was realistisch erscheint.

Anhand der Beziehungen in den Formeln (4.6) und (4.8) kann geschlussfolgert werden, dass auch

$$E \sim n_{el} \tag{4.9}$$

gilt, d. h. der Energieverbrauch ist auf beiden Verarbeitungseinheiten proportional zur Anzahl verarbeiteter Elemente.

Frequenzregelung

Die dynamische Spannungs- und Frequenzregelung erlaubt es, die CPU in einen P-State mit niedrigerer Frequenz und damit auch niedrigerer Leistungsaufnahme zu versetzen, s. Abschnitt 2.1.3. Mit dem hier vorgestellten Experiment wird untersucht, ob es bestimmte P-States gibt, mit denen die CPU die CGM energieffizienter ausführen kann.

Die CPU des Rechners *Sandybridge* unterstützt P-States mit folgenden Frequenzen: 1,2 GHz bis

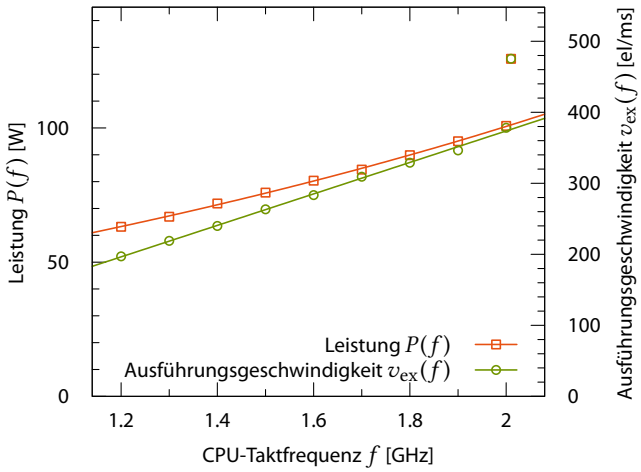


Abbildung 4.8: Leistungsaufnahme P der CPU und Ausführungsgeschwindigkeit v_{ex} bei der Ausführung der CGM bei verschiedenen CPU-Taktfrequenzen f

2,0 GHz in Intervallen von 0,1 GHz sowie 2,001 GHz für den Turbo-Modus [16]. Im Turbo-Modus läuft die CPU schneller als mit ihrer nominalen Taktfrequenz, solange bestimmte Bedingungen eingehalten werden, vgl. Abschnitt 2.1.3. Beispielsweise darf die aktuelle Leistungsaufnahme und die CPU-Temperatur die Herstellerspezifikationen nicht überschreiten [125].

Für das Experiment werden die CPU-Kerne nacheinander in jeden der verfügbaren P-States versetzt. Bei der Ausführung von AXMEBE werden dann Ausführungszeit und Energieverbrauch für verschiedene Elementanzahlen gemessen. Die Leistung P und die Ausführungsgeschwindigkeit v_{ex} werden für jeden P-State nach dem im vorangegangenen Abschnitt genutzten Verfahren durch Regression ermittelt. Das Diagramm in Abb. 4.8 zeigt das Verhalten von Leistung und Ausführungsgeschwindigkeit in Abhängigkeit der CPU-Taktfrequenz f. Die beiden vertikalen Achsen sind so skaliert, dass Leistung und Ausführungszeit für den letzten P-State übereinander zu liegen kommen. Die Ergebnisse werden durch die Kurven

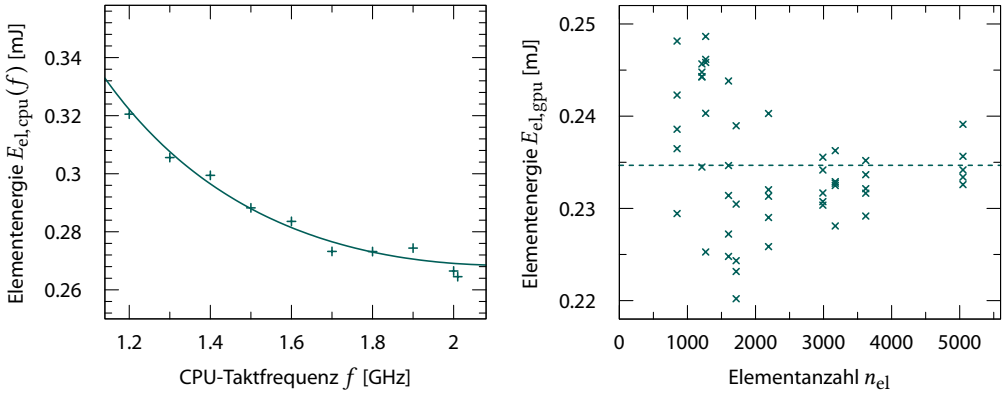
$$P(f) = af^3 + bf + c \quad \text{und} \quad v_{ex}(f) = df + e \tag{4.10}$$

im Intervall von 1,2 bis 2,0 GHz approximiert. Der Turbo-Modus mit 2,001 GHz blieb bei der Regressionsanalyse unberücksichtigt, da seine tatsächliche Frequenz von der CPU intern festgelegt wird und nicht mit der Frequenz, die dem P-State zugeordnet ist, zusammenhängt.

Dass die Leistungsaufnahme am besten durch eine kubische Gleichung angenähert wird, wird motiviert durch die bereits in Abschnitt 2.1.3 eingeführte Gleichung

$$P = CU^2 f + UI_{\text{leck}} \ , \tag{4.11}$$

die häufig zur Modellierung der Leistungsaufnahme von CPUs genutzt wird [192]. Die Frequenz f wird als proportional zur Spannung U angenommen, vgl. Abschnitt 2.1.3. Daher ist der erste



(a) Verarbeitung auf der CPU

(b) Verarbeitung auf der GPU

Abbildung 4.9: Energieverbrauch der CPU und der GPU zur Verarbeitung eines Elements

Summand in Gleichung (4.11) proportional zu f^3 und der zweite Summand proportional zu f. Die durch die Regression ermittelten Parameter sind: $a = 2{,}2 \pm 0{,}4$, $b = 29 \pm 4$ sowie $c = 24 \pm 4$. Der Wert hinter dem ± gibt jeweils den asymptotischen Standardfehler, der bei der Regression ermittelt wurde, an.

Die lineare Abhängigkeit zwischen Ausführungsgeschwindigkeit und Taktfrequenz rührt daher, dass die Ausführungszeit von AXMEBE durch die Rechengeschwindigkeit der CPU beschränkt wird, also *compute-bound* ist. Die Anzahl der Maschinenbefehle in dieser Routine ist fest und ihre Ausführung benötigt eine feste Anzahl Takte. Folglich ist die Ausführungszeit von AXMEBE etwa proportional zu f^{-1}. Die durch die Regression ermittelten Parameter sind $d = 222 \pm 4$ und $e = -70 \pm 6$.

Die Graphen in Abb. 4.8 zeigen deutlich, dass die Ausführungsgeschwindigkeit schneller wächst als die Leistungsaufnahme. Folglich ermöglichen P-States mit höherer Taktfrequenz eine effizientere Ausführung.

Elementenergie

Abbildung 4.9 zeigt die Energie, die benötigt wird, um ein Element auf der CPU bzw. der GPU zu verarbeiten. Bei der CPU in Abb. 4.9a wird diese Elementenergie in Abhängigkeit der Taktfrequenz der CPU dargestellt. Für die Ergebnisse wurden die Werte für die Leistung P durch die Werte für die Ausführungsgeschwindigkeit v_{ex} aus Abb. 4.8 geteilt:

$$E_{\mathrm{el,cpu}}(f) = \frac{P(f)}{v_{\mathrm{ex}}(f)} \quad . \tag{4.12}$$

Die Kurve im Diagramm in Abb. 4.9a ist das Verhältnis der beiden Kurven in Abb. 4.8. Die Ergebnisse zeigen, dass höhere Taktfrequenzen energieeffizientere Ausführung ermöglichen: Bei einer Frequenz von 2,0 GHz, beträgt die Elementenergie $E_{\mathrm{el,cpu}}$ rund 267 µJ und rund 321 µJ bei einer Frequenz von 1,2 GHz. Im Turbo-Modus beträgt die Energie 265 µJ.

Die Elementenergie auf der GPU $E_{\mathrm{el,gpu}}$ wird in Abb. 4.9b dargestellt. Jeder Messpunkt ist das

```
1  SMAMVEKD(n, y[1...n], A[1...n(n+1)/2], x[1...n])
2  begin
3      y[0...n] := (0,...,0)
4      k := 0
5      for i = 1,...,n do
6          y[i] := y[i] + A[k] x[i]
7          k := k + 1
8          for j = i + 1,...,n do
9              y[i] := y[i] + A[k] x[j]
10             y[j] := y[j] + A[k] x[i]
11             k := k + 1
12         end
13     end
14 end
```

Algorithmus 4.9 : Pseudocode der in [31] genutzten Routine SMAMVEKD zur Matrix-Vektor-Multiplikation

Ergebnis einer Messung mit einer gewissen Anzahl an Elementen. Eine klare Abhängigkeit der Energie von der Elementanzahl ist nicht erkennbar. $E_{\mathrm{el,gpu}}$ streut im Bereich von 220 µJ bis 250 µJ. Der durch die gestrichelte Linie markierte Mittelwert der Werte ist 235 µJ. Ein Element auf der GPU zu verarbeiten ist also ein wenig effizienter als es auf der CPU zu verarbeiten.

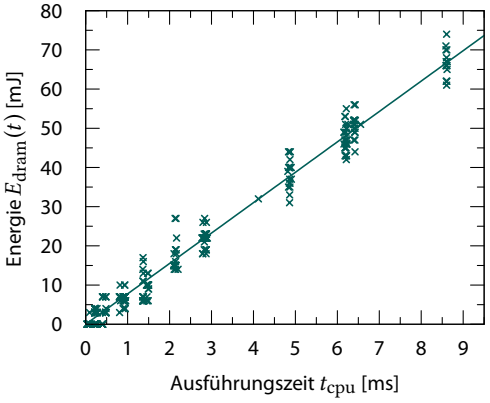
Theoretische Berechnung der Elementenergie

Die Routine, die in AXMEBE die meiste Zeit, nämlich etwa 90 %, benötigt, ist die Matrix-Vektor-Multiplikationsroutine SMAMVEKD. Ihre Semantik ist ähnlich zu der BLAS-Routine DSPMV [68]: Eine symmetrische, als Dreiecksmatrix gepackt gespeicherte Matrix wird mit einem Vektor multipliziert. Der Pseudocode für SMAMVEKD, der vom Quelltext in [31] abgeleitet wurde, wird in Alg. 4.9 angegeben. Die Größe n der Elementdatenstrukturen beträgt maximal 81. Experimente haben gezeigt, dass die in Alg. 4.9 gezeigte Implementierung in Bezug auf die Ausführungszeit für die vorkommenden Größen der Elementdatenstrukturen genauso effizient ist wie eine BLAS-Implementierung.

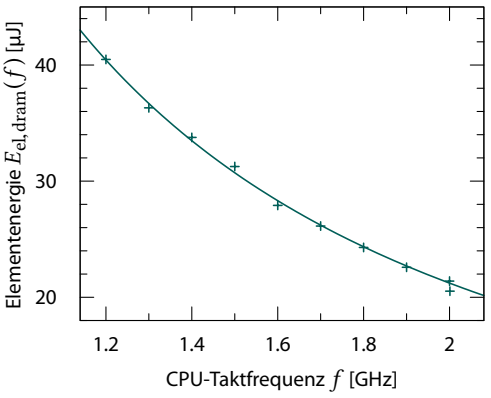
Im Code werden die folgenden Gleitkommaoperationen durchgeführt: Die äußere Schleife wird n-mal durchlaufen und besteht aus einer Addition und einer Multiplikation pro Iteration. Die innere Schleife wird $n(n+1)/2$-mal durchlaufen und besteht aus zwei Additionen und zwei Multiplikationen pro Iteration. Das ergibt insgesamt $2n(n+2)$ Gleitkommaoperationen.

Tolentino und Cameron [255] schätzen den Energieverbrauch für eine arithmetische Gleitkommaoperation für aktuelle Systeme auf 2 . . . 10 nJ. Mit einer Elementdatenstrukturgröße von $n = 81$ kann eine obere Schranke für den Energieverbrauch der Gleitkommaoperationen von 134 µJ pro Element angegeben werden. Dieser Wert erscheint in Bezug auf den gemessenen Wert von 265 µJ für die gesamte Routine AXMEBE realistisch, wenn man in Betracht zieht, dass zusätzlich zu den hier betrachteten Gleitkommaoperationen

- die Daten aus dem Speicher gelesen und wieder zurückgeschrieben werden müssen,
- Maschinenbefehle aus dem Speicher geladen und dekodiert werden müssen,
- pro SMAMVEKD-Aufruf $n(n+3)$ Ganzzahloperationen und Kontrollflusssteuerungsbefehle wie Vergleiche oder Sprünge anfallen,



(a) Energieverbrauch E_{dram} in Abhängigkeit von der Ausführungszeit von AXMEBE



(b) Elementenergie $E_{\text{el,dram}}$ in Abhängigkeit von der Taktfrequenz

Abbildung 4.10: Energieverbrauch des Speichers während der Ausführung von AXMEBE

- in der Routine AXMEBE weitere Operationen ausgeführt werden, die mit ca. 10 % zu ihrer Ausführungszeit beitragen.

Speicherenergie

Neben der Energie, die die CPU zur Durchführung von Berechnungen benötigt, muss auch der Energieverbrauch des Speichers zum Schreiben, Halten und Laden der Daten berücksichtigt werden. Bei der GPU umfasst der über NVML ermittelte Wert beides. Bei der CPU wird der Energieverbrauch der CPU und des DRAM-Speichers separat gemessen und kann aus dem jeweiligen MSR ausgelesen werden. Die Energiewerte der DRAMs beider CPUs werden zum gesamten Speicherenergieverbrauch E_{dram} aufaddiert.

In Abb. 4.10a wird der Energieverbrauch der DRAMs bei der Ausführung von AXMEBE auf der CPU gezeigt. Es wurden 16 Threads auf beiden CPUs im Turbo-Modus ausgeführt. Der Graph zeigt, dass der Energieverbrauch des Speichers E_{dram} proportional zur Ausführungszeit t_{cpu} der Routine AXMEBE ist, d. h.

$$E_{\text{dram}} \sim t_{\text{cpu}} \quad . \tag{4.13}$$

Die Leistungsaufnahme des Speichers wird durch Regression mit $P_{\text{dram}} = 7{,}8$ W ermittelt.

Im Datenblatt des DIMM-Speicherbausteins [235] werden die Ströme angegeben, die in den verschiedenen Betriebszuständen des Speichers fließen. Der minimale Strom ist IDD6, also der *self refresh current*, und der maximale Strom ist IDD7, also der *operating bank interleave read current* [221]. Die genauen Zeitanteile, in denen die Ströme fließen, ist für die genutzte Anwendung unbekannt. Der Strom sollte allerdings immer im Intervall von IDD6 bis IDD7 liegen, d. h. $390\text{ mA} \leq I_{\text{DD}} \leq 3216\text{ mA}$ für ein DIMM. Der Speicher besteht aus 4 DIMM-Riegeln. Mit einer Spannung von $U = 1{,}35$ V ergibt sich durch $P = UI$ für die Leistung ein Bereich von etwa $2{,}1\text{ W} \leq P_{\text{dram}} \leq 17\text{ W}$

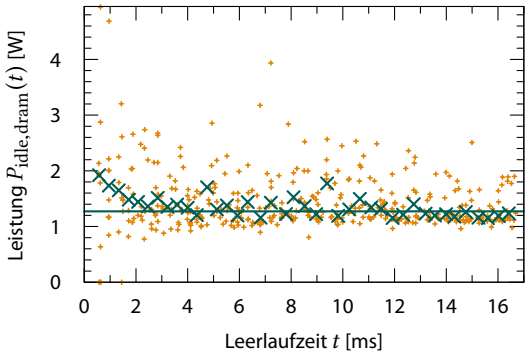


Abbildung 4.11: Leerlaufleistung $P_{\text{idle,dram}}$ des Speichers über verschiedene Zeitintervalle; „+“: eine Messung, „×“: Median aller Messungen eines Zeitintervalls, waagerechte Gerade: Median über alle Messungen

für 4 DIMMs. Das Messergebnis liegt in diesem Bereich.

Die Elementenergie $E_{\text{el,dram}}$ wird für alle CPU-Taktfrequenzen f gemessen und ist in Abb. 4.10b dargestellt: Mit steigender Taktfrequenz sinkt die Elementenergie des Speichers. Die Kurve zeigt, dass die Gesetzmäßigkeiten, die in Abschnitt. 4.4.2 für $E_{\text{el,cpu}}$ gefunden wurden, in etwa auch für $E_{\text{el,dram}}$ gelten. Ebenso ist die Elementenergie des Speichers mit $E_{\text{el,dram}} = 20{,}5\ \mu\text{J}$ im Turbo-Modus am geringsten und mit $E_{\text{el,dram}} = 40{,}5\ \mu\text{J}$ bei der Taktfrequenz 1,2 GHz am höchsten.

In Abb. 4.11 wird die Leerlaufleistung (engl. *idle power*) $P_{\text{idle,dram}}$ des Speichers dargestellt. Die CPUs und der Speicher werden mit der Funktion POSIX-Funktion `nanosleep` für einen gewissen Zeitraum in den Leerlaufmodus (engl. *idle mode*) versetzt. Für die Länge des Zeitraums werden nacheinander verschiedene Werte gewählt und jeweils der Energieverbrauch gemessen. Jeder der kleinen Punkte (+) im Diagramm steht für eine Messung der Leerlaufleistung des Speichers, jeder der großen Punkte (×) für den Median der Messwerte, die bei Messungen der gleichen Zeitintervalllänge entstanden sind. Der Median über alle Werte wird durch die waagerechte Gerade dargestellt. Sein Wert ist $P_{\text{idle,dram}} = 1{,}3$ W. Dieses Ergebnis für $P_{\text{idle,dram}}$ liegt etwas unter der minimalen Leistung von 2,1 W, die anhand des Datenblatts ermittelt wurde, stellt jedoch eine akzeptable Näherung dar.

Datenübertragung

In diesem Abschnitt wird die Datenübertragung der Routine PPCGM in den Zeilen 5 und 6 des Alg. 4.8 untersucht. Die Daten bleiben über alle AXMEBE-Aufrufe eines Verfeinerungsschrittes der FEM konstant. Der Energieverbrauch für die Übertragung der Daten, die für jeden einzelnen AXMEBE-Aufruf nötig ist, ist bereits in der ermittelten Elementenergie für die GPU $E_{\text{el,gpu}}$ enthalten.

Für die Übertragung der Daten vom Haupt- in den Grafikspeicher und umgekehrt bietet CUDA die Funktion `cudaMemcpy` an. In einem ersten Experiment wird die Datenübertragungsrate für zusammenhängend im Speicher abgelegte Daten bestimmt. Für Datenmengen von 1 bis 40 MiB wurde auf dem Rechner *Sandybridge* eine Datenübertragungsrate vom Haupt- zum Grafikspeicher von 3,2 GiB/s und vom Grafik- in den Hauptspeicher von 2,9 GiB/s gemessen.

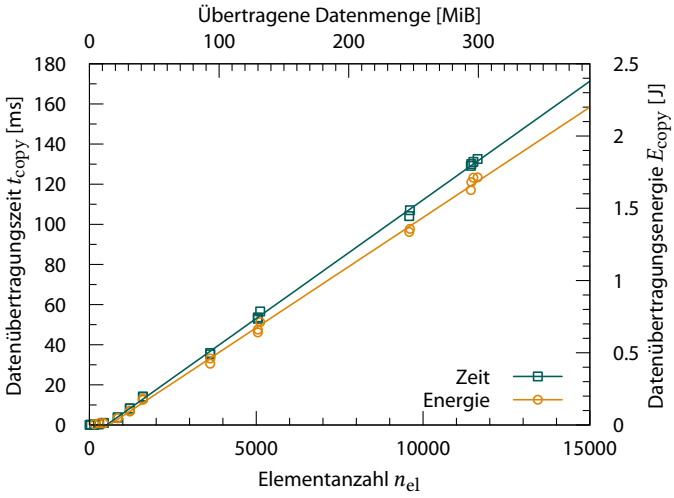


Abbildung 4.12: Dauer und Energieverbrauch des Speichers für die Datenübertragung vom Haupt- in den Grafikspeicher

Allerdings werden in der Anwendung die zu übertragenden Datenstrukturen im Hauptspeicher nicht unmittelbar hintereinander abgelegt. Daher wird experimentell die Datenübertragungsrate und der Energieverbrauch der Datenübertragung innerhalb der Anwendung gemessen. In Abb. 4.12 werden die Größen t_{copy} für die Dauer der Datenübertragung vom Haupt- in den Grafikspeicher und E_{copy} für den Energieverbrauch für verschiedene Elementanzahlen dargestellt: Sowohl die Datenübertragungszeit als auch der Energieverbrauch zeigen ein lineares Verhalten. Die Geraden schneiden die Abszisse bei ca. 500 . . . 600 Elementen, was einer Datenmenge von etwa 12 MiB entspricht. Weniger als 12 MiB Daten zu übertragen benötigt keine Zeit und Energie, was sich auf Caching oder ähnliche Effekte zurückführen lassen könnte.

Die in Abb. 4.12 dargestellten Messwerte werden durch folgende Geraden angenähert:

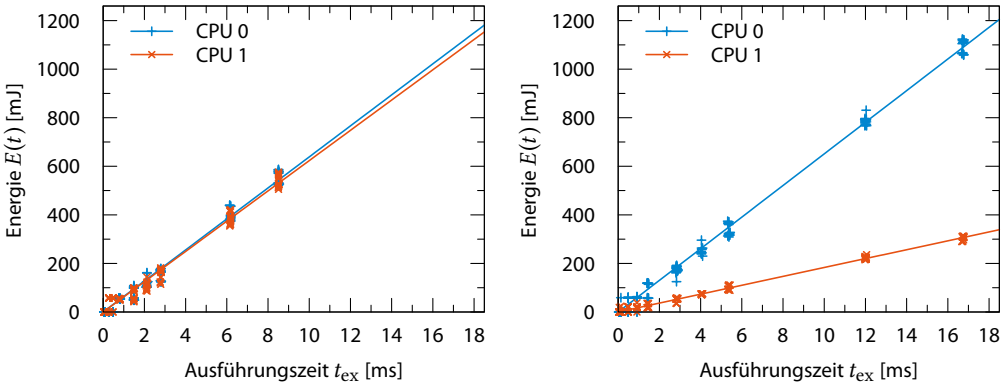
$$E_{\text{copy}}(n_{\text{el}}) = 153\ \frac{\mu\text{J}}{\text{el}} \cdot (n_{\text{el}} - 592\ \text{el}) \tag{4.14}$$

$$t_{\text{copy}}(n_{\text{el}}) = 11{,}8\ \frac{\mu\text{s}}{\text{el}} \cdot (n_{\text{el}} - 512\ \text{el}) \tag{4.15}$$

Die daraus abgeleitete Datenübertragungsrate beträgt etwa 2,1 GiB/s. Es überrascht nicht, dass dieser Wert etwas geringer ist als der Wert von 3,2 GiB/s, der für zusammenhängend abgespeicherte Daten gemessen wurde.

Die Leistungsaufnahme des Speichers wird auf etwa 13 W geschätzt, indem der Anstieg der Energie von 153 µJ/el durch den Anstieg der Zeit von 11,8 µs/el geteilt wird. Bei der Datenübertragung ist die Leistung des Speichers höher als bei der Ausführung von AXMEBE, da letzteres den Speicher durch die geringere Frequenz der Datenzugriffen weniger beansprucht.

Die Leistungsaufnahme der GPU bei der Datenübertragung wurde mit 83,2 W gemessen. Das Schreiben in den Speicher benötigt 25,7 mJ/MiB und das Lesen 29,1 mJ/MiB, also 3,06 nJ/bit bzw.



(a) 16 Threads auf beiden CPUs

(b) 8 Threads auf CPU 1, CPU 2 gedrosselt

Abbildung 4.13: Energieverbrauch der CPU bei der Ausführung von AXMEBE mit und ohne Drosselung

3,47 nJ/bit. Bei der GPU lassen sich die Leistung des Prozessors und des Speichers nicht isolieren, sodass diese Messwerte beide gemeinsam umfassen. Wenn ein Elementvektor aus 81 Einträgen besteht, werden pro PPCGM-Aufruf für jedes auf der GPU verarbeitete Element 27 kB Daten in den GPU-Speicher übertragen. Dieser Wert multipliziert mit dem Energieverbrauch der GPU für das Schreiben in den GPU-Speicher ergibt eine Energie von 661 µJ pro Element.

CPU-Drosselung

Da die GPU eine energieeffizientere Ausführung von AXMEBE als die CPU erlaubt, wäre es denkbar, alle Elemente auf der GPU zu verarbeiten und während dieser Zeit die CPUs zu drosseln, also in einen Leerlaufzustand mit geringer Leistungsaufnahme zu versetzen. Experimente haben jedoch gezeigt, dass das nicht in vollem Umfang funktioniert: Wie auch Anzt et al. feststellten [34], benötigt die erste CPU stets die volle Leistung, selbst wenn sie nur darauf wartet, dass die GPU ihre Aufgabe beendet. Folglich ist es sinnvoll, dass diese CPU während dieser Wartezeit ebenfalls Berechnungen durchführt, also Elemente verarbeitet. Die zweite CPU jedoch wird automatisch gedrosselt und ihre Leistungsaufnahme heruntergefahren, wenn dort kein Thread läuft. 8 Threads auf den Kernen der CPU 1 und keinen auf der CPU 2 laufen zu lassen wird in OpenMP des gcc durch Setzen der Umgebungsvariablen OMP_NUM_THREADS auf 8 und GOMP_CPU_AFFINITY auf 0-7 festgelegt.

Abbildung 4.13a zeigt den Energieverbrauch der CPUs bei der Ausführung von 16 Threads: Beide CPUs arbeiten unter voller Last und haben etwa die gleiche Leistungsaufnahme: $P_{\text{CPU1}} \approx P_{\text{CPU2}} \approx 63$ W. Abbildung 4.13b zeigt, dass im Gegensatz dazu die gedrosselte CPU lediglich eine Leistung von $P_{\text{CPU2}} = 18$ W benötigt, während die andere CPU weiter die volle Leistungsaufnahme von $P_{\text{CPU1}} = 65$ W hat. Natürlich kann auf diese Weise lediglich Leistung, jedoch keine Energie eingespart werden, da eine CPU mit 8 Threads die doppelte Ausführungszeit von zwei CPUs mit 16 Threads benötigt.

Diskussion

Die Experimente zeigen, dass die Ausführungszeiten t_{cpu} und t_{gpu} proportional zu den Elementanzahlen auf den jeweiligen Verarbeitungseinheiten sind. Dasselbe gilt für den Energieverbrauch E_{cpu} und E_{gpu}. Für die Datenübertragung existiert eine lineare Abhängigkeit mit einer Verschiebung von etwa 500 Elementen gegenüber dem Koordinatenursprung. Zur Vereinfachung wird diese Verschiebung vernachlässigt und stattdessen werden die Proportionalitäten

$$t_{\text{copy}} \sim n_{\text{el}} \tag{4.16}$$

$$E_{\text{copy}} \sim n_{\text{el}} \tag{4.17}$$

verwendet.

Für den Rechner *Sandybridge* beträgt die Zeit, die zur Verarbeitung eines Elementes auf der CPU benötigt wird, $t_{\text{el,cpu}} = 1{,}70\ \mu\text{s}$. Auf der GPU beträgt diese Zeit $t_{\text{el,gpu}} = 1{,}69\ \mu\text{s}$. Die für die Verarbeitung eines Elementes benötigte Energie beträgt $E_{\text{el,gpu}} = 235\ \mu\text{J}$ auf der GPU und mindestens $E^*_{\text{el,cpu}} = 286\ \mu\text{J}$ auf der CPU inkl. der Speicherenergie. Für die Übertragung der Daten zur GPU wird eine Energie von $E_{\text{el,copy}} = 814\ \mu\text{J}$ benötigt. Der Energieverbrauch der CPU bei der Datenübertragung wird vernachlässigt, da CUDA die Funktion `cudaMemcpyAsync` anbietet. Diese Funktion überträgt die Daten aus CPU-Sicht asynchron, sodass keine zusätzliche CPU-Zeit hierfür benötigt wird. Folglich ist die Ausführung auf der GPU inklusive der Übertragung energieeffizienter als auf der CPU, wenn die CGM mindestens 15 Iterationen, also AXMEBE-Aufrufe pro PPCGM-Aufruf, benötigt.

Zur Verarbeitung der Rechenlast gibt es drei Möglichkeiten für ihre Verteilung zwischen der CPU und der GPU:

1. Die gesamte Rechenlast wird auf der CPU verarbeitet.
2. Die gesamte Rechenlast wird auf der GPU verarbeitet.
3. Die Rechenlast wird im Gleichgewicht zwischen CPU und GPU aufgeteilt.

Für die dritte Möglichkeit wird eine Verteilungsmethode benötigt, mit der die CPU und die GPU ihren Anteil an der Rechenlast in etwa gleicher Zeit verarbeiten. Eine solche Methode vermeidet Leerlaufzeiten, in denen Energie verbraucht, aber keine Ergebnisse produziert werden. Ebenso liefert möglicherweise eine Variante, die eine CPU drosselt, eine gute Energieeffizienz.

4.4.3 Modell

In diesem Abschnitt wird ein Modell entwickelt, das den Energieverbrauch und die Ausführungszeit der CGM vorhersagt. Das grundlegende Modell, das von dem Pseudocode in Alg. 4.8 abgeleitet wird, wird anhand der Ergebnisse des Abschnitts 4.4.2 verfeinert. Weiterhin wird diskutiert, wie sich der Energieverbrauch durch Änderung der Verteilung der Rechenlast minimieren lässt.

Ausführungszeit

Die Ausführungszeit der CGM besteht aus zwei wesentlichen Teilen: die Zeit, die für die Datenübertragung in den Zeilen 5 und 6 in Alg. 4.8 benötigt wird, und die Ausführungszeit der Schleife in den

Zeilen 8 bis 12. Die Ausführungszeit der Schleife ergibt sich aus dem Produkt der Anzahl Schleifeniterationen l und dem Maximum der Ausführungszeiten zur Ausführung der Routine AXMEBE auf den Verarbeitungseinheiten:

$$t_{\text{cgm}} = t_{\text{copy}} + l \cdot \max\left(t_{\text{cpu}}, t_{\text{gpu}}\right) \quad . \tag{4.18}$$

Die Verarbeitungseinheit, die ihren Anteil der Rechenlast zuerst beendet, wartet darauf, dass die andere ihren Anteil ebenfalls beendet. Die Zeit t_{pre} für den Vorkonditionierer und zur Berechnung der nächsten Approximation in den Zeilen 9 und 10 in Alg. 4.8 sowie weitere Zeiten, die nicht von der Verteilung der Rechenlast auf die Verarbeitungseinheiten abhängen, bleiben unberücksichtigt.

Die Werte t_{cpu}, t_{gpu} und t_{copy} können aufgrund der in Abschnitt 4.4.2 in den Formeln (4.6) und (4.16) gezeigten Proportionalität wie folgt ausgedrückt werden:

$$t_{\text{cpu}} = n_{\text{el,cpu}} \cdot t_{\text{el,cpu}} \tag{4.19}$$

$$t_{\text{gpu}} = n_{\text{el,gpu}} \cdot t_{\text{el,gpu}} \tag{4.20}$$

$$t_{\text{copy}} = n_{\text{el,gpu}} \cdot t_{\text{el,copy}} \quad . \tag{4.21}$$

Der Wert $n_{\text{el,cpu}}$ gibt an, wie viele Elemente auf der CPU verarbeitet werden, $n_{\text{el,gpu}}$ gibt an, wie viele Elemente auf der GPU verarbeitet werden. Ihre Summe ist die Gesamtanzahl der Elemente n_{el}:

$$n_{\text{el}} = n_{\text{el,cpu}} + n_{\text{el,gpu}} \quad . \tag{4.22}$$

Werden in Gleichung (4.18) t_{cpu}, t_{gpu} und t_{copy} durch die entsprechenden Terme aus den Gleichungen (4.19) bis (4.21) ersetzt, ergibt sich

$$t_{\text{cgm}} = n_{\text{el,gpu}} \cdot t_{\text{el,copy}} + l\left(\max\left(n_{\text{el,cpu}} \cdot t_{\text{el,cpu}}, n_{\text{el,gpu}} \cdot t_{\text{el,gpu}}\right)\right) \quad . \tag{4.23}$$

Nur dann, wenn die Bedingung

$$t_{\text{el,copy}} < l \cdot t_{\text{el,cpu}} \tag{4.24}$$

erfüllt ist, ist es sinnvoll, Berechnungen auf die GPU zu verschieben: Wenn man für das Verschieben der Daten in den GPU-Speicher mehr Zeit benötigt, als die CPU zum Verarbeiten benötigt, dann sollten alle Elemente auf der CPU verarbeitet werden, d. h. $n_{\text{el,gpu}} = 0$.

Unter der Annahme, dass die Bedingung in Formel (4.24) erfüllt ist, wird die Ausführungszeit in Gleichung (4.23) minimal, wenn für $n_{\text{el,cpu}}$ und $n_{\text{el,gpu}}$ gilt

$$n_{\text{el,cpu}} \cdot t_{\text{el,cpu}} = n_{\text{el,gpu}} \cdot t_{\text{el,gpu}} \quad . \tag{4.25}$$

Alternativ kann das Ergebnis ausgedrückt werden, indem r für den Anteil der Elemente, die auf der GPU verarbeitet werden, genutzt wird, also $r = n_{\text{el,gpu}}/n_{\text{el}}$ mit $0 \leq r \leq 1$. Dann gilt für $n_{\text{el,cpu}}$ und $n_{\text{el,gpu}}$:

$$n_{\text{el,gpu}} = r n_{\text{el}} \quad , \quad n_{\text{el,cpu}} = (1 - r) n_{\text{el}} \quad . \tag{4.26}$$

Das Einsetzen der Gleichungen (4.26) in Gleichung (4.25) ergibt für r:

$$r = \left(1 + \frac{t_{\text{el,gpu}}}{t_{\text{el,cpu}}}\right)^{-1} \quad . \tag{4.27}$$

Von allen möglichen Verteilungen der Rechenlast zwischen CPU und GPU ist diejenige, bei der r nach Gleichung (4.27) ausgewählt wird, die mit der geringsten Ausführungszeit t_{cgm} für die Routine PPCGM.

Energie

Der Energieverbrauch von PPCGM ergibt sich aus der Summe der Energie für die Datenübertragung E_{copy} und der Energie für die Ausführung der Schleife in Alg. 4.8. Für jede der l Iterationen wird eine Energie von $E_{\text{cpu}} + E_{\text{dram}}$ für die Verarbeitung auf der CPU und E_{gpu} für die Verarbeitung auf der GPU benötigt. Die Energie E_{pre} für den Vorkonditionierer und die Berechnung der nächsten Approximation fällt ebenfalls an, wird aber im Folgenden nicht weiter berücksichtigt, da sie unabhängig von der Verteilung der Rechenlast auf CPU und GPU ist. Der Energieverbrauch der CGM E_{cgm} ergibt sich also wie folgt:

$$E_{\text{cgm}} = l \left(E_{\text{cpu}} + E_{\text{dram}} + E_{\text{gpu}}\right) + E_{\text{copy}} \quad . \tag{4.28}$$

Durch die Fähigkeit der Prozessoren, ihre Leistungsaufnahme im Leerlaufmodus (engl. *idle mode*) zu senken, existieren zwei verschiedene Werte für die Leistung: die Leistung P_{ex}, die aufgenommen wird, wenn der Prozessor Rechenlast verarbeitet, sowie die Leistung P_{idle}, die aufgenommen wird, wenn er sich im Leerlaufmodus befindet, also keine Rechenlast verarbeitet. Folglich kann der Energieverbrauch für E_{cpu}, E_{dram} und E_{gpu} wie folgt formuliert werden:

$$E_{\text{cpu}} = P_{\text{ex,cpu}} \cdot t_{\text{ex,cpu}} + P_{\text{idle,cpu}} \cdot t_{\text{idle,cpu}} \quad , \tag{4.29}$$

$$E_{\text{dram}} = P_{\text{ex,dram}} \cdot t_{\text{ex,dram}} + P_{\text{idle,dram}} \cdot t_{\text{idle,dram}} \quad , \tag{4.30}$$

$$E_{\text{gpu}} = P_{\text{ex,gpu}} \cdot t_{\text{ex,gpu}} + P_{\text{idle,gpu}} \cdot t_{\text{idle,gpu}} \quad . \tag{4.31}$$

t_{ex} und t_{idle} geben jeweils die Zeiten an, in der sich der entsprechende Prozessor im Betriebs- bzw. im Leerlaufmodus befindet. Im Folgenden wird der Energieverbrauch im ersten Summanden der Gleichungen (4.29) bis (4.31) jeweils explizit durch $E_{\text{ex,cpu}}$, $E_{\text{ex,dram}}$ bzw. $E_{\text{ex,gpu}}$ ausgedrückt.

Die in den Formeln (4.9), (4.13) und (4.17) gezeigte Proportionalität zwischen E und n_{el} ergibt die Ausdrücke

$$E_{\text{ex,cpu}} = n_{\text{el,cpu}} \cdot E_{\text{el,cpu}} \quad , \tag{4.32}$$

$$E_{\text{ex,dram}} = n_{\text{el,cpu}} \cdot E_{\text{el,dram}} \quad , \tag{4.33}$$

$$E_{\text{ex,gpu}} = n_{\text{el,gpu}} \cdot E_{\text{el,gpu}} \quad , \tag{4.34}$$

$$E_{\text{copy}} = n_{\text{el,gpu}} \cdot E_{\text{el,copy}} \quad . \tag{4.35}$$

Der Kürze halber werde die Summe der Leerlaufleistungen von CPU und Speicher im Folgenden

als $P^*_{\text{idle,cpu}}$ bezeichnet, also

$$P^*_{\text{idle,cpu}} = P_{\text{idle,cpu}} + P_{\text{idle,dram}} \quad , \tag{4.36}$$

und die Summe der Berechnungs- und Speicherenergie für die Verarbeitung eines Elementes auf der CPU als $E^*_{\text{el,cpu}}$, also

$$E^*_{\text{el,cpu}} = E_{\text{el,cpu}} + E_{\text{el,dram}} \quad . \tag{4.37}$$

Gleichung (4.28), die den Energieverbrauch bei der Ausführung der CGM mit CPU und GPU ausdrückt, wird wie folgt umformuliert:

$$\begin{aligned} E_{\text{cgm}} = l \left(n_{\text{el,cpu}} \cdot E^*_{\text{el,cpu}} + n_{\text{el,gpu}} \cdot E_{\text{el,gpu}} \right) \\ + n_{\text{el,gpu}} \cdot E_{\text{el,copy}} \\ + P^*_{\text{idle,cpu}} \cdot t_{\text{idle,cpu}} + P_{\text{idle,gpu}} \cdot t_{\text{idle,gpu}} \quad . \end{aligned} \tag{4.38}$$

Minimierung des Energieverbrauchs

Für die Ausführung der gesamten Rechenlast auf der CPU wird eine Energie von

$$E_{\text{cgm}} = n_{\text{el}} \cdot E^*_{\text{el,cpu}} \tag{4.39}$$

benötigt. Die Leerlaufenergie der GPU wird vernachlässigt, da sie ausgeschaltet oder komplett entfernt werden kann, wenn sämtliche Berechnungen auf der CPU durchgeführt werden.

Die Summe der Berechnungsenergie für ein Element und der Anteil der Datenübertragungsenergie, der einem Aufruf von AXMEBE zugeschrieben werden kann, wird mit $E^*_{\text{el,gpu}}$ bezeichnet:

$$E^*_{\text{el,gpu}} = E_{\text{el,gpu}} + l^{-1} E_{\text{el,copy}} \quad . \tag{4.40}$$

Wird die gesamte Rechenlast auf der GPU verarbeitet während sich die CPU im Leerlauf befindet, ergibt sich der Energieverbrauch aus der Summe der GPU-Energie und der Leerlaufenergie der CPU für die Dauer der Ausführung:

$$E_{\text{cgm}} = l \left(n_{\text{el}} \cdot E^*_{\text{el,gpu}} + n_{\text{el}} \cdot t_{\text{el,gpu}} \cdot P^*_{\text{idle,cpu}} \right) \quad . \tag{4.41}$$

Für die Entscheidung, wie die Rechenlast zu verteilen ist, ergeben sich drei Möglichkeiten:

1. Wenn $E^*_{\text{el,cpu}} < E^*_{\text{el,gpu}}$,
   dann verarbeite alle Elemente auf der CPU.

2. Wenn $E^*_{\text{el,cpu}} > E^*_{\text{el,gpu}} + P^*_{\text{idle,cpu}} \cdot t_{\text{el,gpu}}$,
   dann verarbeite alle Elemente auf der GPU.

3. Sonst
   verteile die Rechenlast zwischen CPU und GPU.

Für $E_{\text{el,cpu}}$ kann in der Fallunterscheidung der Wert einer beliebigen CPU-Taktfrequenz verwendet werden. Wenn die Bedingung für den 1. Fall für keine Frequenz erfüllt ist und die Bedingung für

Größe	Wert	Bemerkungen
$E_{\text{el,cpu}}$	265 µJ	Turbo-Modus
$E_{\text{el,dram}}$	20,5 µJ	Turbo-Modus
$E_{\text{el,gpu}}$	235 µJ	
$E_{\text{el,copy}}$	814 µJ	
$P_{\text{idle,cpu}}$	83 W	
$P_{\text{idle,dram}}$	1,3 W	
$t_{\text{el,gpu}}$	1,69 µs	
l	32,4	Mittel für die ersten 21 Verfeinerungsschritte von *kurbel3f*

Tabelle 4.3: Messwerte für Größen, die für die energiebewusste Verteilung der Rechenlast benötigt werden

den 2. Fall nur für einige Frequenzen erfüllt ist, ist die Strategie des 3. Falls anzuwenden. In diesem Fall wird die Frequenz mit dem niedrigsten Wert für $E_{\text{el,cpu}}$ genutzt.

Wird im 3. Fall die Rechenlast zwischen der CPU und der GPU verteilt, müssen die Leerlaufzeiten in Gleichung (4.38) minimiert werden, um die Energieeffizienz zu maximieren. Während dieser Leerlaufzeiten wird lediglich Energie verbraucht, aber keine „sinnvollen Ergebnisse“ nach der Definition der Energieeffizienz in Abschnitt 2.1.1 erzeugt. Die Leerlaufzeiten zu minimieren bedeutet sicherzustellen, dass $t_{\text{cpu}} = t_{\text{gpu}}$, also das r mit einer minimalen Ausführungszeit nach Gleichung (4.27) zu finden.

In Tabelle 4.3 werden die Messwerte aufgeführt, die in den Experimenten in Abschnitt 4.4.2 für den Rechner *Sandybridge* ermittelt wurden und nun für die Fallunterscheidung zur energiebewussten Verteilung der Rechenlast benötigt werden. Für die mittlere Anzahl der CGM-Iterationen l wurde der Mittelwert der 21 ersten Verfeinerungsschritte für das Objekt *kurbel3f* genommen. Für andere Objekte kann für die Vorhersage beispielsweise die mittlere Anzahl an CGM-Iterationen, die bei den bisherigen Verfeinerungsschritten benötigt wurde, genutzt werden. Das Einsetzen der in Tab. 4.3 aufgeführten Werte in die Gleichungen (4.37), (4.40) und (4.36) ergibt:

- $E^*_{\text{el,cpu}} = 286$ µJ,
- $E^*_{\text{el,gpu}} = 260$ µJ,
- $P^*_{\text{idle,cpu}} = 84$ W.

Anhand der Werte ist erkennbar, dass für den Rechner *Sandybridge* die Bedingung für Fall 1 nicht erfüllt ist. Da bei der Bedingung für Fall 2 die rechte Seite, also $E^*_{\text{el,gpu}} + P^*_{\text{idle,cpu}} \cdot t_{\text{el,gpu}}$, 401 µJ beträgt, wird auch diese Strategie nicht angewendet. Folglich muss auf dem Rechner *Sandybridge* die dritte Möglichkeit, also die Verteilung der Rechenlast zwischen CPU und GPU für eine möglichst energieeffiziente Ausführung genutzt werden. Eine solche Methode wird im Abschnitt 4.5 vorgestellt.

4.4.4 Schlussfolgerungen

Auf dem Rechner *Sandybridge* wird für die Verarbeitung eines Elements der FEM in der CGM auf der CPU eine Energie von 286 µJ inkl. der Speicherenergie benötigt. Die Verarbeitung auf der GPU

benötigt im Mittel 260 µJ inkl. der Energie für die Datenübertragung in den GPU-Speicher. Die Verarbeitung der Elemente auf der GPU ist folglich energieeffizienter als die Verarbeitung auf der CPU. Trotzdem ist die Verarbeitung auf der GPU allein nicht die energieeffizienteste Lösung: Für jedes Element, das auf der GPU verarbeitet wird, benötigt die CPU eine Leerlaufenergie von 142 µJ. Folglich ist die gemeinsame Verarbeitung auf CPU und GPU, bei der beide Verarbeitungseinheiten gleichzeitig je einen Teil der Elemente verarbeiten, die energieeffizienteste Art und Weise der Ausführung. Bei diesem Verfahren werden die Leerlaufzeiten von CPU und GPU minimiert, um den Gesamtenergiebedarf zu minimieren.

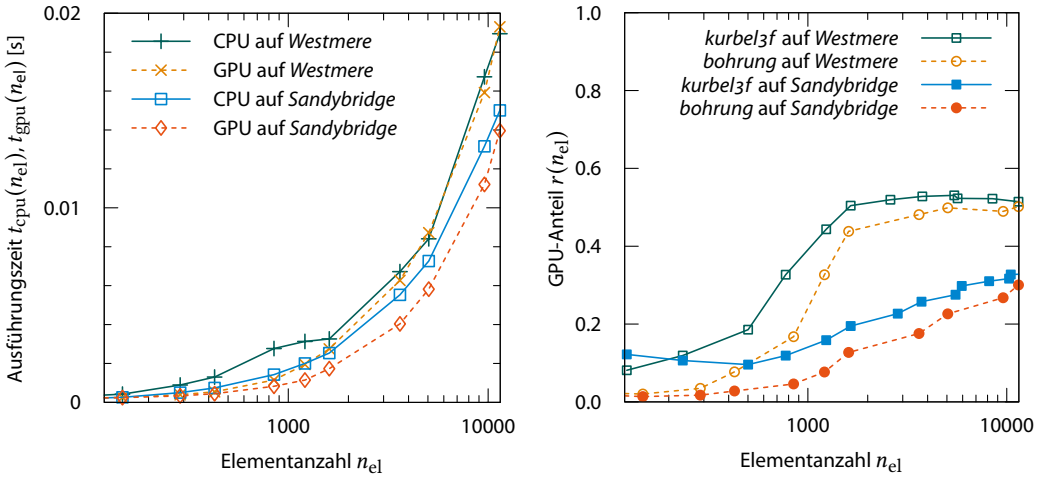
Ein anderes Verfahren zur energieeffizienten Implementierung der CGM wird in [8] vorgestellt. Die Energieeffizienz dieser Implementierung wird dadurch erhöht, dass die CGM umformuliert wird: Die gesamte Berechnung wird dadurch auf der GPU durchgeführt. Die vorherige Implementierung bestand aus mehreren GPU-Kernelfunktionen mit Zwischenabschnitten auf der CPU, sodass viele Synchronisationen notwendig waren.

Modelle zur Vorhersage der Ausführungszeit der CGM werden z. B. in [263] und [266] vorgestellt: Verschoor und Jalba [263] stellen ein Modell für die Ausführungszeit einer parallelen CGM auf mehreren GPUs auf. In dieses Modell fließen die Dimension des Problems und die Gesamtanzahl der gespeicherten Elemente der Matrix ein. An den konkreten Rechner angepasst wird das Modell durch Ausführung mehrerer Tests und einen Fit. Vázquez et al. [266] implementieren die CGM auf der GPU. Sie stellen ein analytisches Modell vor, das zwei Parameter für die Implementierung, nämlich die Threadanzahl und Größe der CUDA-Blöcke, optimal bestimmt. Das Modell nutzt die beiden rechnerspezifischen Parameter Warpgröße und Anzahl der Streaming-Mulitprozessoren der GPU sowie den datenspezifischen Parameter der Länge jeder Matrixzeile. Beide Modelle nutzen die Anzahl der Speicheroperationen zur Vorhersage der Ausführungszeiten. Sie sind auch beide etwas komplexer als das in dieser Arbeit genutzte Modell. Da alle hier in Frage kommenden Matrizen aus einer FEM stammen, sind sie sich untereinander relativ ähnlich. Dadurch verringert sich der Einfluss verschiedener Matrixcharakteristika im Modell. Die verbleibenden Unterschiede müssen nicht explizit modelliert werden, da sie durch die Messung der Ausführungszeiten mit den konkreten Daten in das Modell einfließen. Dasselbe gilt für die Maschinenparameter.

Die Experimente haben gezeigt, dass verbesserte Möglichkeiten zur Energiemessung für eine genauere Analyse notwendig sind. Eine höhere zeitliche Auflösung der Energie-MSRs der CPU und des Leistungsmessers der GPU wären wünschenswert. Insbesondere für die Online-Messung wäre auch eine Energiemessfunktionalität für die GPU wünschenswert. Im Gegensatz zu dem aktuell vorhandenen Leistungsmesser könnte dann nach der Ausführung die verbrauchte Energie ausgelesen werden. Die Funktionalität könnte beispielsweise implementiert werden, indem in der Hardware oder im GPU-Treiber die gemessene Leistung über die Zeit integriert wird.

4.5 Dynamische Lastverteilung

Zur Minimierung des Energieverbrauchs der CGM bei der gemeinsamen CPU-GPU-Ausführung wird in diesem Abschnitt eine Methode zur dynamischen Verteilung der Rechenlast vorgestellt [162]. Sie verwendet die Gleichung (4.27), um eine ausgeglichene Verteilung mit minimaler Leerlaufzeit von CPU und GPU zu erreichen. Die Methode nutzt aus, dass die FEM iterativ vorgeht



(a) Vergleich der Ausführungszeiten auf der CPU und der GPU für das Beispielobjekt *bohrung*

(b) Anteil r der Elemente, die auf der GPU verarbeitet werden, im Verlauf der Netzverfeinerung für die Objekte *kurbel3f* und *bohrung* auf den Rechnern *Westmere* und *Sandybridge*

Abbildung 4.14: Verhalten der Ausführungszeiten und des GPU-Anteils beim Ausgleich der Rechenlast für die gemeinsame Ausführung der CGM auf CPU und GPU

und misst in jedem Verfeinerungsschritt die Werte $t_{el,cpu}$, $t_{el,gpu}$ und $t_{el,copy}$, um sie zur Berechnung der Verteilung für den nächsten Verfeinerungsschritt zu verwenden. Der Anteil r der auf der GPU verarbeiteten Elemente wird also für jeden FEM-Verfeinerungsschritt neu berechnet. Unter der Annahme, dass sich die Verhältnisse zwischen zwei Verfeinerungsschritten nicht drastisch ändern, passt sich die Verteilung dadurch dynamisch auf Veränderungen z. B. aufgrund wachsender Datengrößen an. Algorithmus 4.8 wird dahingehend abgewandelt, dass vor Zeile 3 der neue Wert für r berechnet wird und die Datenverteilung unter Berücksichtigung dieses r erfolgt. Der Anfangswert für r wird auf 0,5 gesetzt. In den folgenden Abschnitten wird die vorgeschlagene Methode evaluiert.

4.5.1 Verteilung der Rechenlast

Um eine ausgeglichene Verteilung zu erreichen, muss die Bedingung der Gleichung (4.25) erfüllt sein: Die Ausführungszeiten der CPU und der GPU für AXMEBE müssen etwa gleich sein. Für das Beispielobjekt *bohrung* wurden beide Zeiten auf den Rechnern *Westmere* und *Sandybridge* bei Benutzung der vorgeschlagenen dynamischen Verteilungsmethode gemessen. Sie werden in Abb. 4.14a dargestellt. Das Diagramm zeigt, dass die Zeiten, die für die Verarbeitung von $n_{el,gpu}$ Elementen auf der CPU und $n_{el,gpu}$ Elementen auf der GPU benötigt werden, etwa gleichgroß sind, wenn $n_{el,gpu}$ und $n_{el,gpu}$ für jeden Verfeinerungsschritt anhand des berechneten r festgelegt werden.

Abbildung 4.14b zeigt, wie sich der Anteil r der auf der GPU verarbeiteten Elemente im Verlauf

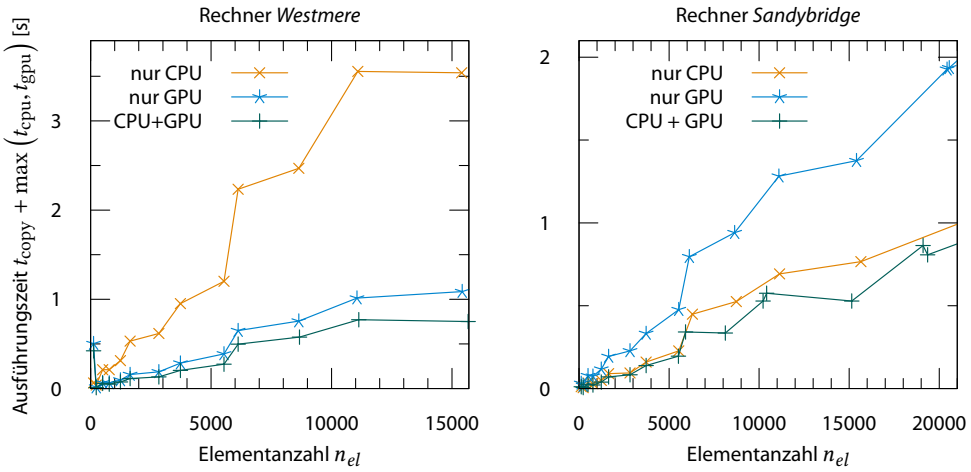


Abbildung 4.15: Ausführungszeiten der CGM bei CPU-Ausführung, GPU-Ausführung und gemeinsamer CPU-GPU-Ausführung mit dem Objekt *kurbel3f*

der Netzverfeinerung verändert. Im Diagramm ist erkennbar, dass r auf dem Rechner *Westmere* ab ca. 2000 Elementen unabhängig vom Testobjekt bei ca. $\frac{1}{2}$ liegt. Auf dem Rechner *Sandybridge* steigt r fortwährend an und liegt am Ende bei 10 000 Elementen bei ca. $\frac{1}{3}$.

Der Anteil der auf der GPU verarbeiteten Elemente wird während der Ausführung der FEM mit steigender Elementanzahl angepasst. Auf dem Rechner *Westmere* benötigt die CPU im Bereich von 400 bis 1500 Elementen signifikant mehr Zeit als die GPU, vgl. Abb. 4.14a. Das hat ein schnelles Ansteigen des GPU-Anteils zur Folge, wie in Abb. 4.14b gut erkennbar ist. Auf dem Rechner *Sandybridge* benötigt die CPU während fast der gesamten Ausführungszeit mehr Zeit für die Verarbeitung ihrer Elemente, was zum fortwährenden Ansteigen des GPU-Anteils in Abb. 4.14b führt.

4.5.2 Ausführungszeitgewinn

Abbildung 4.15 zeigt die Summe der Ausführungszeiten der verteilungsabhängigen Abschnitte der Routine PPCGM für das Objekt *kurbel3f*. Die Messung wurde auf beiden Rechnern mit den drei Implementierungsvarianten durchgeführt. Die Variante mit der gemeinsamen CPU-GPU-Ausführung erreicht auf beiden Rechnern einen Speed-up von etwa 25 % im Vergleich zur jeweils besseren der CPU- und der GPU-Varianten. Während die GPU-Variante auf dem Rechner *Westmere* die bessere ist, ist die CPU-Variante auf dem Rechner *Sandybridge* die bessere. Eine Erklärung dafür könnte in der Rechenleistung der Verarbeitungseinheiten liegen: Die Peak-Performance der CPUs in *Sandybridge* ist doppelt so hoch wie die Peak-Performance der CPUs in *Westmere*. Analog ist die Peak-Performance der GPU in *Sandybridge* dreimal so hoch. Allerdings ist die Speicherübertragungsrate der *Sandybridge*-GPU niedriger. Folglich ist der Einfluss der Datenübertragungsraten überproportional groß, wodurch die GPU auf dem Rechner *Sandybridge* gegenüber der GPU verliert.

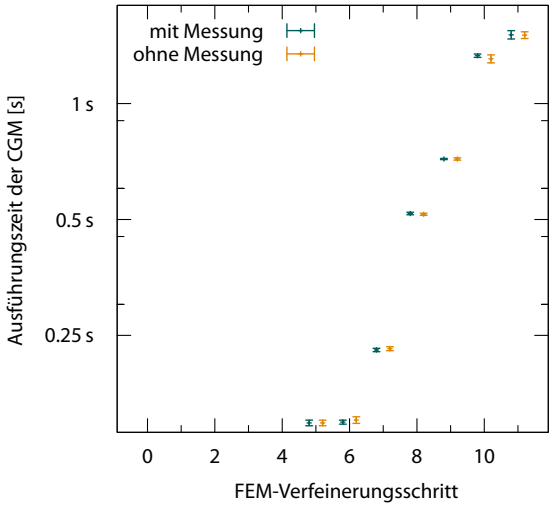


Abbildung 4.16: Ausführungszeit mit PPCGM mit und ohne die Messung der Ausführungszeitparameter während der Ausführung

4.5.3 Overhead

Die Messung der Ausführungszeiten, die nötig ist, um die Parameter t_{copy}, t_{cpu} und t_{gpu} zu erhalten, erzeugt möglicherweise einen Overhead in PPCGM. Dieser wird in einem weiteren Experiment untersucht. Abbildung 4.16 zeigt die Ausführungszeiten der Routine PPCGM mit und ohne diese Messung für das Objekt *bohrung* auf dem Rechner *Westmere*. Der Anteil der auf der GPU verarbeiteten Elemente wird fest auf 0,53 gesetzt, da dies dem ermittelten optimalen Anteil entspricht. Für jeden FEM-Verfeinerungsschritt existiert ein Messpunkt. Bei keinem der Messpunkte ist eine signifikant unterschiedliche Ausführungszeit zwischen den beiden Varianten feststellbar. Das deutet darauf hin, dass die Verwendung dieses Online-Autotuning-Verfahrens keinen signifikanten Overhead erzeugt.

4.5.4 Verwandte Arbeiten

Zur gemeinsamen Verarbeitung von Daten durch CPU und GPU existieren verschiedene Ansätze: Harmony [64] ist ein Programmier- und Ausführungsmodell, das es ermöglicht, Programme für CPU-GPU-Systeme zu schreiben und sie auszuführen. Das Runtime-System von Harmony verteilt die Rechenlast automatisch auf die CPU und die GPU. Allerdings profitieren die meisten Anwendungen nicht von der Zusammenarbeit. Die vorgestellten Ergebnisse zeigen allerdings, dass eine dynamische Verteilung der Rechenlast notwendig ist, da die Ausführungszeit stark rechnerabhängig ist, was die Erkenntnisse dieser Arbeit stützt.

MapCG [116] ist ein MapReduce-Framework, das es ermöglicht, MapReduce-Jobs auf CPUs und auf GPUs auszuführen. Für die durchgeführten Experimente war der Speed-up bei der Zusammenarbeit stets unter 1,1, häufig sogar unter 1. Das wird der Notwendigkeit der Serialisierung/Deserialisierung der Intermediate-Daten für deren Übertragung zugeschrieben.

In [256] wird eine Cholesky-Faktorisierung, die als ein gerichteter kreisfreier Taskgraph formuliert wird, untersucht. Manche dieser Tasks können nur auf der CPU, manche nur auf der GPU und einige auf beiden ausgeführt werden. Die Verteilung der Tasks zu CPU und GPU ist fest, d. h. kann nicht auf deren Ausführungsgeschwindigkeit angepasst werden. [243] untersucht eine blockbasierte Cholesky- und QR-Faktorisierung. Die Ausführungszeiten der Kernel auf den Verarbeitungseinheiten werden während der Installation für verschiedene Blockgrößen gemessen und in einer Bibliothek abgespeichert. Mit dieser Information wird eine optimale Verteilung zwischen CPU und GPU ermittelt.

In [37] wird das DAGuE-Framework [36], das Algorithmen als Anordnung von Tasks in einem gerichteten kreisfreien Graphen auffasst, um GPU-Funktionalitäten erweitert. Tasks werden durch Codelets implementiert. Für einen Task kann es verschiedene Codelets geben, von denen jeder eine andere Hardwareplattform unterstützt. Die CPU-GPU-Zusammenarbeit wird durch die Fähigkeit, Codelets in verschiedenen Versionen zugleich auszuführen, ermöglicht. Im Vergleich zur Ausführung nur auf der GPU mit der MAGMA-Bibliothek [176] erreicht dieses erweiterte DAGuE-Framework einen Speed-up von 1,2.

[75] verteilt basierend auf einer Ausführungszeitvorhersage die `DGEMM`-Matrixmultiplikation auf mehrere CPU-Kerne und eine GPU. Zur Berechnung dieser Vorhersage wird eine Formel entwickelt, die die Berechnungs- und die Datenübertragungszeit berücksichtigt. Für die tatsächliche Berechnung der Verteilung wird die Datenübertragungszeit jedoch nicht berücksichtigt, da sie von der Ausführungszeit dominiert wird. Die Parameter werden dort basierend auf der Peak-Performance abgeschätzt und nicht, wie in dieser Arbeit, basierend auf Messungen. Im Vergleich zur GPU-Ausführung wird durch zusätzliches Nutzen einer 4-Kern-CPU die Rechenleistung um 35 % erhöht.

Für die dreidimensionale Rekonstruktion von durch Elektronenmikroskopie erfassten Körpern wird in [7] ein Schema zur Zusammenarbeit von CPU und GPUvorgestellt. Es nutzt einen Task-Pool zur Verteilung der Rechenlast und erreicht einen Speed-up von 1,4 bis 1,7 im Vergleich zur reinen CPU- oder GPU-Ausführung. Schedulingverfahren für Jobs, die sowohl auf CPUs als auch auf GPUs ausgeführt werden können, werden in [225] vorgeschlagen.

Die meisten der vorgestellten Methoden erreichen einen Speed-up zwischen 1,15 und 1,35 bei gemeinsamer Verarbeitung im Vergleich zur Verarbeitung lediglich auf der CPU bzw. der GPU. Das liegt in der selben Größenordnung wie bei der in diesem Unterkapitel vorgestellten Methode, bei der ein Speed-up von ca. 1,25 erzielt werden konnte. Lediglich bei dem Task-Pool-Ansatz in [7] wurde größerer Speedup von bis zu 1,7 erreicht. Allerdings ist der Task-Pool-Ansatz für das vorliegende Problem nicht umsetzbar, da aufgrund der mehrfachen Iteration die Datenverteilung im Vorhinein feststehen muss. Außerdem ermöglichen alle vorgestellten Ansätze bis auf [7], [64] und [37] lediglich eine statische Verteilung der Rechenlast, keine dynamische wie hier vorgestellt.

4.5.5 Diskussion

In Tab. 4.4 wird für verschiedene Verteilungsstrategien dargestellt, wie viel Energie der Rechner *Sandybridge* für eine Iteration mit dem das Objekt *kurbel3f* mit 10 000 Elementen benötigt. In Abschnitt 4.5.1 wurde gezeigt, dass der GPU-Anteil r bei gemeinsamer Verarbeitung etwa $\frac{1}{3}$ beträgt. Eine Iteration benötigt eine Energie von 1,91 J auf der CPU und eine Energie von 0,87 J für die

Strategie	CPU-Anteil	GPU-Anteil	CPU-Energie[1]	GPU-Energie	Gesamtenergie
nur CPU	1	0	2,86 J	0 J	2,86 J
gemeinsame Verarbeitung	$\frac{2}{3}$	$\frac{1}{3}$	1,91 J	0,87 J	2,78 J
Drosselung eines CPU-Kerns	$\frac{1}{2}$	$\frac{1}{2}$	1,43 J + 0,15 J	1,30 J	2,88 J
nur GPU	0	1	2,86 J + 0,30 J	2,61 J	5,77 J

1 beide CPU-Kerne zusammen bzw. Kern 1 + Kern 2

Tabelle 4.4: Energieverbrauch bei verschiedenen Datenverteilungsstrategien zwischen CPU und GPU für die Verarbeitung von 10 000 finiten Elementen auf dem Rechner *Sandybridge*

Berechnung auf der GPU mitsamt der notwendigen Datenübertragung. Das ergibt einen Gesamtenergieverbrauch von 2,78 J. Wenn nur eine CPU genutzt und die andere in den Leerlaufmodus versetzt würde, wie in Abschnitt 14 vorgeschlagen, würde je die Hälfte der Elemente auf der GPU und der verbleibenden CPU verarbeitet werden. Das ergäbe einen Energieverbrauch von 1,43 J auf der CPU und 1,30 J auf der GPU. Mit einer Leerlaufleistung von 18 W betrüge deren Leerlaufenergie 0,15 J. Das ergäbe einen Gesamtenergieverbrauch von 2,88 J. Folglich ist es für die Ausführung auf dem Rechner *Sandybridge* empfehlenswert, beide CPUs für die gemeinsame CPU-GPU-Ausführung zu verwenden.

In diesem Unterkapitel wurde gezeigt, dass das in Abschnitt 4.4.3 entwickelte Ausführungszeitmodell dazu genutzt werden kann, die Rechenlast so zwischen CPU und GPU zu verteilen, dass die Ausführungszeit minimiert wird. Dieses Schema wird für einen der drei Fälle zur energieminimierenden Ausführung der CGM, die in Abschnitt 4.4.3 verwendet werden, benötigt. Es wurde ein Online-Autotuning-Verfahren eingesetzt, das in jedem Verfeinerungsschritte die Ausführungszeiten auf der CPU und der GPU sowie die Datenübertragungszeit misst und anhand dieser die Parameter zum Finden der Verteilung für den nächsten Verfeinerungsschritt bestimmt. So wird die Verteilung auf die zugrunde liegende Hardware angepasst und gleichzeitig werden Eigenschaften der Daten einbezogen. Mit der Kombination der CPU mit der GPU zur Verarbeitung der finiten Elemente in der CGM wurde zirka ein Viertel der Ausführungszeit im Vergleich zur Verarbeitung nur auf der CPU oder nur auf der GPU eingespart.

4.6 Schlussfolgerungen

Das in Unterkapitel 4.5 vorgestellte Schema zur dynamischen Lastverteilung zwischen CPU und GPU minimiert die Ausführungszeit der CGM. Den Energieverbrauchs minimiert es jedoch nur in einem der drei Fälle der in Abschnitt 4.4.3 vorgestellten Fallunterscheidung. Unter bestimmten Umständen ist es effizienter, nur die CPU bzw. nur die GPU für die Verarbeitung zu nutzen. Dass das Verfahren zur Lastverteilung ein dynamisches sein muss, wird in Abschnitt 4.5.1 gezeigt: Mit steigender Elementanzahl verändert sich der optimale Anteil der Elemente, die auf der GPU verarbeitet werden. Bei dem Online-Verfahren werden derzeit nur Ausführungszeiten berücksichtigt. Um auch die Energieeffizienz in jedem Verfeinerungsschritt erneut bewerten und berücksichtigen

können, sind neue Online-Methoden zur Energiemessung sowohl für CPUs als auch für GPUs notwendig.

In Bezug auf NUMA-Architekturen sollte für künftige Hardwarearchitekturen das Verhältnis zwischen der Zugriffszeit auf entfernte NUMA-Knoten und auf den lokalen NUMA-Knoten im Auge behalten werden. Wird das Verhältnis zu groß, sollte die in Unterkapitel 4.3 beschriebene Methode für die NUMA-gerechte Datenverarbeitung für die CPU-Ausführung genutzt werden.

4.7 Zusammenfassung

In diesem Kapitel wurde die effiziente parallele Ausführung einer Implementierung der Methode der konjugierten Gradienten in einer adaptiven FEM untersucht. „Effizient“ bezieht sich hier sowohl auf die Ausführungszeit als auch auf den Energieverbrauch. Die parallele Ausführung erfolgt auf Multicore-CPUs und einer GPU, wobei die Rechenlast zwischen CPU und GPU aufgeteilt werden kann.

Die energiebewusste parallele Ausführung der CGM ermöglicht das in Unterkapitel 4.4 aufgestellte Energie- und Ausführungszeitmodell. Durch das Modell kann vorhergesagt werden, unter welchen Bedingungen die Ausführung nur auf der CPU, nur auf der GPU oder eine gemeinsame CPU-GPU-Ausführung am energieeffizientesten ist. Die dafür notwendige Fallunterscheidung wird in Abschnitt 4.4.3 aufgestellt. Für die gemeinsame CPU-GPU-Ausführung wird in Unterkapitel 4.5 ein Schema vorgestellt, das die Rechenlast derart auf CPU und GPU verteilt, dass deren Leerlaufzeiten und damit auch der Energieverbrauch und die Ausführungszeiten minimiert werden. Das vorgestellte Schema ist ein Online-Autotuning-Verfahren. Das heißt, es misst die Ausführungszeiten während der Ausführung der CGM in einem Verfeinerungsschritt der FEM und berücksichtigt diese Messungen bei der Verteilung für den nächsten Verfeinerungsschritt.

Zum Aufstellen des Energie- und Ausführungszeitmodells werden Experimente durchgeführt, die den Energieverbrauch und die Ausführungszeit auf der CPU und der GPU in Abhängigkeit von der Anzahl verarbeiteter Elemente untersuchen. Sie ergeben eine Proportionalität zwischen Elementanzahl, Ausführungszeit und Energieverbrauch. Ebenso gilt für den Energieverbrauch des Speichers der CPU eine Proportionaliät zwischen Elementanzahl und Energieverbrauch. Außerdem wurde das Verhalten bei Frequenz- und Spannungsregelung für die CPU und ihren Speicher untersucht: Die niedrigste Energie, die für die Verarbeitung eines Elements benötigt wird, tritt bei der höchstmöglichen CPU-Frequenz auf. Eine lineare Abhängigkeit zwischen Elementanzahl und benötigter Zeit bzw. Energie wird auch bei der Übertragung vom CPU- zum GPU-Speicher beobachtet.

Mit dem in Unterkapitel 4.2 vorgestellten Reduktionsschema kann die für die Reduktion verteilt liegender Ergebnisvektoren notwendige Zeit deutlich gesenkt werden. Nach dem Schema wird jedes Teilergebnis unmittelbar nach seiner Berechnung direkt in den gemeinsamen Ergebnisvektor geschrieben. Der Schreibvorgang wird durch Verwendung atomarer Operationen synchronisiert. Durch die hohe Rechenintensität des Codeabschnitts wird im Vergleich zum unsynchronisierten Schreiben auf den Ergebnisvektor keine zusätzliche Zeit benötigt. Dieses Reduktionsschema wird außerdem bei der GPU-Implementierung in Abschnitt 4.4.1 genutzt.

In Unterkapitel 4.3 wird ein Verarbeitungsschema für die finiten Elemente vorgestellt, das die

Zugehörigkeit eines Elements zu einem NUMA-Knoten berücksichtigt. Jede CPU verarbeitet genau die Elemente, die in dem NUMA-Knoten abgespeichert sind, dem sie auch angehört. Es wird gezeigt, dass dieses Schema weder einen signifikanten Vorteil in der Ausführungszeit noch im Energieverbrauch bringt. Trotzdem können die in der Implementierung genutzten Datenstrukturen für die Implementierung der Verarbeitung auf der GPU weiterverwendet werden. ■

5 Energiemessung für GPU-Routinen

Zur Untersuchung des Energieverbrauchs von Anwendungen, die zumindest teilweise auf der GPU ausgeführt werden, sind genaue Methoden zur Energiemessung auf der GPU notwendig. Neben der externen Hardwaremessung, die an allen Geräten möglich ist (vgl. Abschnitt 2.2.2), bietet manche Hardware eine Softwareschnittstelle zum direkten Auslesen der Energie- bzw. Leistungsmesswerte. Gegenüber der Hardwaremessung vereinfacht das die Messung für den Anwender deutlich. Der Messwert kann dann problemlos direkt in der Anwendung weiter genutzt werden. Eine solche Schnittstelle bietet für GPUs des Herstellers Nvidia die Bibliothek *Nvidia Management Library* (NVML). Sie ermöglicht es, die momentan von der GPU aufgenommene Leistung auszulesen.

Der von der NVML bereitgestellte Leistungsmesswert wird alle 20 Millisekunden aktualisiert. Dieses Messintervall ist für eine genaue Ermittlung des Energieverbrauchs einer GPU-Kernelroutine nicht ausreichend, insbesondere nicht für Routinen mit einer sehr kurzen Ausführungszeit. Derartige Routinen treten in vielen Gebieten auf, beispielsweise in der numerischen Mathematik bei der Lösung linearer Gleichungssysteme (vgl. Abschnitt 4.4.2), in der Physik bei Monte-Carlo-Verfahren [217] oder in der Datenverarbeitung bei der Bildung von Partialsummen [69]. Folglich sind Methoden zur Erzeugung genauer, zeitlich hochaufgelöster Profile der Leistungsaufnahme von GPU-Routinen notwendig. Bei der externen Hardwaremessung werden die Messungen mit Abtastraten von 500 Hz [231], 1 kHz [136] oder sogar 50 kHz [56] durchgeführt. Durch Integration der im Leistungsprofil angegebenen Leistung P über die Ausführungszeit t der Routine erhält man ihren Energieverbrauch E.

Der Beitrag dieses Kapitels ist die Entwicklung einer Methode zum Erzeugen hochaufgelöster Leistungsprofile von GPU-Kernelroutinen unter Verwendung einer Messschnittstelle mit einer niedrigen Abtastrate [163]. Diese Profile können zur Untersuchung und Verminderung des Energieverbrauchs von GPU-Routinen genutzt werden. Es werden eine Offline-Methode und eine Online-Methode vorgestellt. Die Offline-Methode führt die GPU-Routine unabhängig von der Anwendung aus, in der sie genutzt wird, und misst so die Leistungsaufnahme. Die Online-Methode führt die Messungen während der Ausführung der Anwendung durch, um den bei der Offline-Methode entstehenden Overhead zu vermeiden. Durch Nutzung der Online-Methode werden beispielsweise Online-Autotuningverfahren ermöglicht. Beide Methoden basieren darauf, die bei vielfachen Ausführungen der Routine gewonnenen Leistungsmesswerte geeignet zu einem Leistungsprofil der Routine zu kombinieren.

Hähnel et al. [122] stellen eine Methode für den gleichen Zweck vor, die sich allerdings auf die Messung des Energieverbrauchs von CPUs mit RAPL (s. Abschnitt 2.2.2) bezieht. Bei RAPL beträgt das Messintervall 1 ms. Zur Messung des Energieverbrauchs einer kürzeren Routine wird diese genau zu Beginn des Messintervalls gestartet. Vom Ende der Routine bis zum Ende des Messintervalls wird eine Routine ausgeführt, deren Leistungsbedarf bekannt ist. Das Verfahren auf GPUs mit Mes-

sung durch NVML zu übertragen ist nicht sinnvoll, da bei RAPL Energiewerte gemessen werden, durch die GPU aber Leistungswerte. Bei RAPL erfolgt die Integration bereits durch die Hardware. Zusätzlich ist bei der Messung auf der GPU eine Synchronisation der Uhren von CPU und GPU notwendig.

Dieses Kapitel gliedert sich wie folgt: Abschnitt 5.1 gibt einen Überblick über die Messung der Leistungsaufnahme auf Nividia-GPUs. Abschnitt 5.2 stellt die Methode zur Erzeugung von hochaufgelösten Leistungsprofilen vor, die aus niedrig aufgelösten Messungen gewonnen werden. Abschnitt 5.3 zeigt für ausgewählte Routinen Beispiele für solche Leistungsprofile. In Abschnitt 5.4 wird die Methode zu einem Online-Verfahren weiterentwickelt. Abschnitt 5.5 schließt das Kapitel ab und zieht Schlussfolgerungen.

5.1 Messung der Leistungsaufnahme von GPUs

Moderne GPUs nutzen dynamische Spannungs- und Frequenzregelung, um die spezifizierte Thermal Design Power, also die maximal zulässige Leistung, auch bei Routinen mit hoher Leistungsaufnahme nicht zu überschreiten [204, 3]. Für GPUs des Herstellers Nvidia der neuesten Generation bietet die Nvidia Management Library (NVML) [207] eine Schnittstelle zum Auslesen der momentan benötigten elektrischen Leistung an.

5.1.1 Messvorgang

Abbildung 5.1 zeigt den Vorgang der Leistungsmessung auf Nvidia-GPUs: Zu Beginn jedes Zeitintervalls der Länge T, dem Messintervall, wird jeweils die momentane Leistungsaufnahme gemessen und zwischengespeichert. Die GPU-Routine wird im Beobachtungszeitraum mehrmals ausgeführt. Im oberen Diagramm wird die Leistungsaufnahme der GPU für den Fall, dass sie 70 W Leistung im Leerlauf und 130 W bei Ausführung einer GPU-Routine benötigt, dargestellt. Im unteren Diagramm wird die Leistungsaufnahme dargestellt, wie sie dem Nutzer erscheint: Durch die Pufferung des Leistungsmesswertes erhält der Nutzer beim Auslesen unter Umständen einen anderen Wert für die Leistung als er aktuell tatsächlich auftritt. Wie in der Abbildung dargestellt wird, kann das zu fehlerhaften Annahmen über die Leistungsaufnahme der GPU führen.

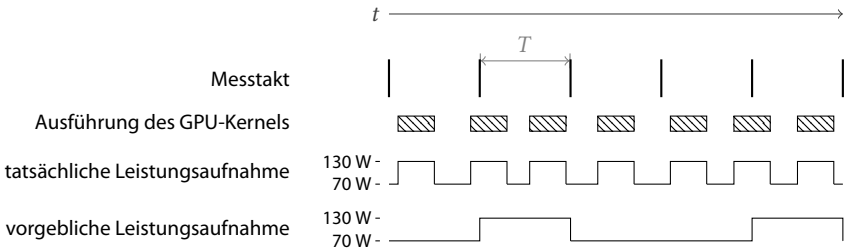


Abbildung 5.1: Wiederholte Ausführung einer Kernelfunktion und die Messung der Leistungsaufnahme

Die Software-Schnittstelle zum Auslesen des Leistungsmesswerte stellt die Routine `nvmlDe-`

viceGetPowerUsage der NVML bereit. Sie gibt den Messwert in Milliwatt zurück. Er umfasst die elektrische Leistung der gesamten Grafikkarte inklusive des Speichers. Er hat einen Messfehler von ±5 W [207]. Die Messungen in diesem Kapitel werden auf einer GPU vom Typ Nvidia Tesla C2075 [206] vorgenommen, die in einem Rechner mit zwei Vierkern-CPUs vom Typ Intel Xeon E5-2650 verbaut ist. Auf dem Rechner läuft als Betriebssystem Linux der Kernel-Version 3.2 mit dem Nvidia-GPU-Treiber der Version 304.64 sowie CUDA der Version 4.2.9.

5.1.2 Länge des Messintervalls

Zunächst wird die Länge T des Messintervalls bestimmt: Das entsprechende Experiment nutzt aus, dass der Messwert starkem Rauschen unterliegt. In den meisten Fällen ändert sich der Wert in zwei aufeinanderfolgenden Intervallen selbst dann, wenn sich der Zustand der GPU nicht verändert hat. Das Messverfahren liest wiederholt den aktuellen Messwert P_{akt} von der GPU aus. Daraus, dass P_{akt} sich von dem zuvor gelesenen Wert unterscheidet, kann geschlossen werden, dass die Hardware ihren Messwert aktualisiert hat. Dann wird die aktuelle Zeit ausgelesen und die Länge des Intervalls ausgegeben. Dieses Verfahren wird 100 000-mal wiederholt, sodass 100 000 Messwerte für die Länge des Messintervalls T vorliegen. In seltenen Fällen (ca. 0,5 ‰) ändert sich P_{akt} zwischen zwei Messintervallen nicht. Das resultiert in Intervalllängen, die zwei- oder dreimal so groß wie die übrigen Intervalllängen sind. Diese werden eliminiert und T als Mittelwert der übrigen Werte berechnet.

Zur Zeitmessung wird die POSIX-Funktion `clock_gettime` genutzt, die die Systemzeit mit einer Genauigkeit von 1 ns zurückliefert. Für die Länge des Messintervalls wurde ein Wert von $T = (20{,}0082 \pm 0{,}0008)$ ms ermittelt. Eine Messintervalllänge von $T \approx 20$ ms bedeutet, dass die Mess- bzw. Abtastfrequenz $f = \frac{1}{T} \approx 50$ Hz beträgt.

5.2 Erzeugung von Leistungsprofilen

Die Abtastfrequenz der genutzten GPU beträgt 50 Hz. Diese Frequenz ist unzureichend zur Ermittlung des Leistungs- bzw. Energiebedarfs von Kernel-Routinen mit einer Ausführungszeit von unter 20 ms. Um dennoch Messungen für solche kurzen Routinen zu ermöglichen, wird eine Methode entwickelt, die für Kernel-Routinen hochaufgelöste Leistungsprofile erzeugt. Diese Leistungsprofile stellen die elektrische Leistungsaufnahme der GPU im Verlauf der Ausführung der Kernel-Routine dar. Die entwickelte statistische Methode führt die Routine wiederholt aus und startet sie dabei zu verschiedenen Phasen des Messintervalls. Dadurch erfolgt die Messung zu unterschiedlichen Zeitpunkten während der Ausführung der Routine und ein hochaufgelöstes Profil kann erzeugt werden.

Die Methode wird in Abb. 5.2 illustriert: Die zu untersuchende GPU-Routine wird mehrfach mit einer zufälligen Wartezeit t_{wait} zwischen den Aufrufen ausgeführt. Nachdem die Ausführung gestartet wurde, wird der Leistungsmesswert überwacht. Bei jeder Veränderung wird er gemeinsam mit der seit dem Start vergangenen Zeit ausgegeben. Diese beiden Werte werden zur schrittweisen Erzeugung des unten in der Abbildung dargestellten Diagramms verwendet. Wenn während der Ausführung der Kernel-Routine keine Aktualisierung des Leistungsmesswertes erfolgt, werden

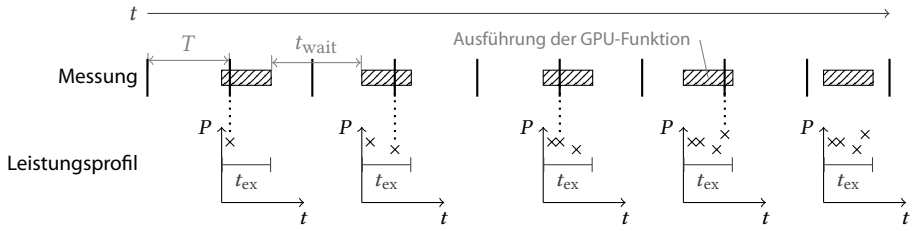


Abbildung 5.2: Erzeugung eines Leistungsprofil durch mehrfaches Ausführen einer Kernel-Routine und Überwachtung des Leistungsmesswertes

```
1  retrieve current time t_start
2  execute GPU kernel asynchronously
3  wait for GPU kernel to finish
4  retrieve current time t_ret
5  t_ex := t_ret − t_start
6  for k = 1 to n_ex do
7      wait a random time t_wait
8      retrieve GPU power P_last
9      execute GPU kernel asynchronously
10     retrieve current time t_start; t_curr := t_start
11     while t_curr < t_start + t_delay + t_ex + T do
12         retrieve GPU power P_curr
13         if P_curr ≠ P_last then
14             retrieve current time t_update
15             emit t_update − t_start, P_curr
16             P_last := P_curr
17         end
18         retrieve current time t_curr
19     end
20 end
```

Algorithmus 5.1 : Auslesen der Leistungsmesswerte für eine GPU-Kernelfunktion

keine Werte ausgegeben, wie es das letzte Diagramm zeigt.

In Alg. 5.1 wird die Erzeugung eines Leistungsprofils im Pseudocode dargestellt: Zunächst wird die GPU-Routine einmal ausgeführt, um ihre Ausführungszeit t_{ex} zu bestimmen (Zeilen 1 bis 5). Der Algorithmus nimmt an, dass die Ausführungszeit der Routine stets gleich ist. Daher sollte sie immer mit den gleichen Parametern aufgerufen werden. Danach wird die Routine n_{ex}-mal aufgerufen. Für die in Abschnitt 5.3 vorgestellten Experimente wurde $n_{\text{ex}} = 100$ gewählt. Dadurch werden im Mittel 100 Leistungswerte pro Messintervall ermittelt, was einer effektiven Abtastfrequenz von 5000 Werten pro Sekunde entspricht. Vor der Ausführung der GPU-Routine wird die CPU für eine zufällig gewählte Zeitspanne angehalten (Zeile 7). So wird die Leistung zu einem zufälligen Zeitpunkt der Ausführung der Routine gemessen. Die Wartezeit t_{wait} ist im Intervall $T \leq t_{\text{wait}} < 2T$ gleichverteilt. Durch die Wahl dieses Intervalls wird sichergestellt, dass nicht die Leistung der vorangegangenen Ausführung der Routine gemessen wird. Die GPU-Routine wird dann asynchron ausgeführt (Zeile 9). In Experimenten hat sich herausgestellt, dass zwischen dem Funktionsaufruf

und dem Ansteigen der Leistung eine Verzögerung auftritt. Zwischen dem Auslesen der Startzeit t_{start} unmittelbar nach der Rückkehr des asynchronen GPU-Aufrufes und dem Ansteigen der Leistung vergeht eine Zeit von $t_{\text{delay}} \approx 6$ ms, vgl. Abschnitt 5.3. Solange die GPU-Routine läuft, liest die Schleife in Zeile 11 wiederholt den Leistungsmesswert aus. Jedesmal, wenn sich der Wert P_{akt} ändert (Zeile 13), wird er gemeinsam mit der seit dem Start der Routine vergangenen Zeit ausgegeben (Zeile 15).

Die Ausgabe von Alg. 5.1, also eine Menge von Paaren (P, t), wird zur Erzeugung eines Diagramms weiterverarbeitet. Die P-Werte liegen im Bereich von der *long idle*-Leistung von 35 W [142] bis zur *thermal design power* von 225 W [206]. Die t-Werte geben an, wieviel Zeit seit dem Start der GPU-Routine vergangen ist. Sie liegen im Bereich von 0 bis $t_{\text{ex}} + T$. Ein Leistungsprofil umfasst alle Punkte, die sich aus den für eine Routine ermittelten (P, t)-Paaren ergeben.

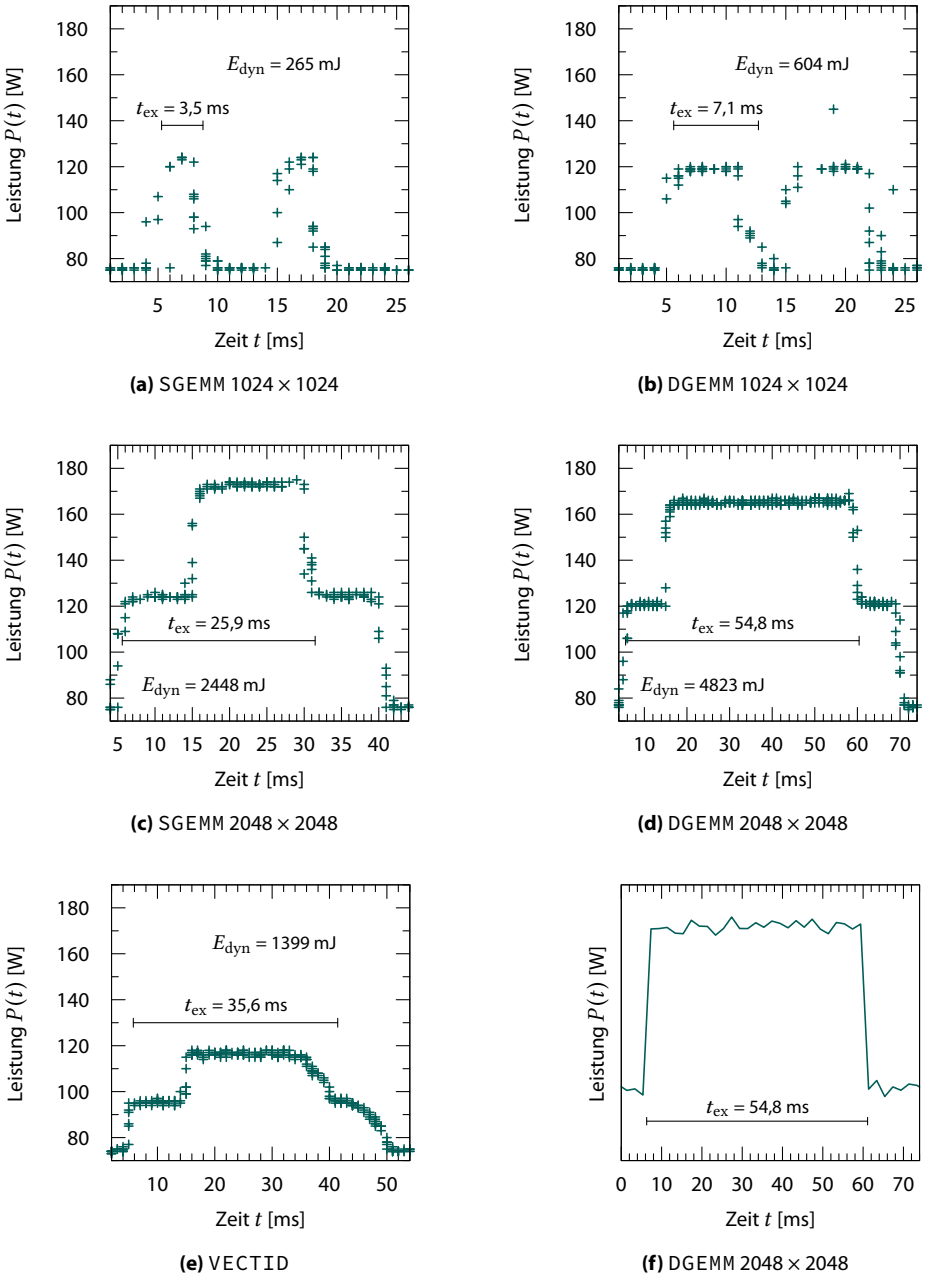
Das in diesem Abschnitt vorgestellte Verfahren lastet einen CPU-Kern im Verlauf der Messung vollständig aus. Daher eignet es sich nur für die Offline-Erzeugung von Leistungsprofilen, also einer Erzeugung unabhängig von der Anwendung, die die GPU-Routine nutzt. Dessen ungeachtet, dass das vorgestellte Verfahren eine hohe CPU-Last erzeugt, hatte das ständige Abfragen des Leistungsmesswertes keine wahrnehmbare Auswirkung auf die Ausführung der GPU-Routine.

5.3 Leistungsprofile für ausgewählte Routinen

Die Diagramme in Abb. 5.3a–e zeigen die Leistungsaufnahme bei der Ausführung der Matrix-Matrix-Multiplikationsroutine *x*GEMM der CUBLAS-Bibliothek und einer weiteren Routine, VECTID. Die Routine VECTID führt arithmetische Operationen und Speicherzugriffe dergestalt aus, dass verschiedene GPU-Threads verschiedene Ausführungszeiten haben. Jeder Punkt im Diagramm steht für ein Zeit-Leistungs-Paar (P, t) wie von Alg. 5.1 ausgegeben.

Die Gesamtleistung P_{ges} wird wie in Abschnitt 2.1.3 vorgeschlagen in die statische Leistung P_{stat} und die dynamische Leistung P_{dyn} aufgeteilt: $P_{\text{ges}} = P_{\text{stat}} + P_{\text{dyn}}$. Die statische Leistung kann anhand der Diagramme mit $P_{\text{stat}} = 76$ W bestimmt werden. Die dynamische Leistung hängt von der Routine ab, die die GPU ausführt. Bei Ausführung der Routine SGEMM mit einer Matrixgröße von 1024×1024 steigt die Leistung auf 123 W und sinkt nach deren Beendigung wieder. 10 ms nach dem ersten Anstieg steigt die Leistung jedoch erneut an. Dieser zweite Ausschlag hat eine ähnliche Form wie der erste. Es liegt nahe, dass die Ursache für diesen zweiten Ausschlag und den Treppeneffekt, der im Fall der 1024×1024-Matrix auftritt, dieselbe ist. Bei dem Treppeneffekt steigt bei Sekunde 6,3 die Leistungsaufnahme auf 124 W und steigt bei Sekunde 16,3 erneut auf dann 172 W. Das Verhalten beim Beenden der Funktion ist analog. Als mögliche Ursache kommt in Frage, dass eine Hälfte der Leistung der GPU bei der Messung um 10 ms verzögert wird. Der tatsächliche Wert für die dynamische Leistung ist also doppelt so hoch wie die beiden Diagramme für die *x*GEMM-Aufrufe mit den 1024×1024-Matrizen suggerieren: Für SGEMM beträgt P_{ges} 172 W statt 124 W mit $P_{\text{dyn}} = 96$ W. Das passt zum Fall der 2048×2048-Matrix mit SGEMM, bei dem P_{ges} ebenfalls 172 W beträgt. Im Falle von DGEMM beträgt die Gesamtleistung $P_{\text{ges}} = 166$ W mit $P_{\text{dyn}} = 92$ W.

Die zur Ausführung benötigte Energie ergibt sich aus der Integration der Leistung über die Zeit. Der Inhalt der Fläche unter der Leistungskurve wird durch die Verzögerung nicht beeinflusst. Allerdings ist aufgrund der Verzögerung die Leistung um 10 ms über die Beendigung der Routine



(a) SGEMM 1024 × 1024 **(b)** DGEMM 1024 × 1024

(c) SGEMM 2048 × 2048 **(d)** DGEMM 2048 × 2048

(e) VECTID **(f)** DGEMM 2048 × 2048

Abbildung 5.3: (a)–(e) Leistungsprofile für die CUBLAS-Routinen *x*GEMM und VECTID; (f) durch Hardwaremessung erzeugtes Leistungsprofil

hinaus zu integrieren.

Die Routine VECTID ist eine CUDA-Funktion mit heterogener Rechenlast für die einzelnen GPU-Threads: Die Ausführungsdauer eines Threads hängt von seiner Threadnummer ab. Die Routine besteht hauptsächlich aus einer Schleife, die von 0 bis $10 \cdot tid$ läuft, wobei *tid* die Threadnummer im Threadblock ist. Es existieren 32 Blöcke mit jeweils 1024 Threads. In der Schleife werden 5 Gleitkommaoperationen ausgeführt. Außerdem werden zwei Feldeinträge aus dem globalen Speicher gelesen und einer geschrieben. Durch die unterschiedlichen Ausführungszeiten der Threads sinkt die Leistungsaufnahme hier weniger steil als bei den *x*GEMM-Routinen. Mit einer gröberen zeitlichen Auflösung der Leistungsprofile wäre ein solcher Effekt nicht erkennbar gewesen. Die Werte für den dynamischen Energieverbrauch, die in Abb. 5.3 angegeben sind, wurden durch Integration der im jeweiligen Diagramm angegebenen Werte mit der Trapezregel [85, S. 607] ermittelt.

Abbildung 5.3f wurde durch Hardwaremessung erzeugt. Dazu wurde der elektrischen Strom, der durch die externen PCI-Express-Spannungsversorgungskabel der GPU-Karte fließt, gemessen. Die elektrische Leistung entspricht dem Produkt von Stromstärke und Spannung. Da direkt durch den Sockelverbinder weiterer Strom fließt, kann anhand der Messwerte nicht auf die genaue Gesamtleistung geschlossen werden. Trotzdem zeigt die Kurve, dass die Leistung der GPU genau für die Zeit der Ausführung der DGEMM-Routine ansteigt. Der Verzögerungseffekt ist also tatsächlich nur ein Effekt beim Messen und tritt nicht in der Realität auf.

Die maximal zulässige Leistung, also die Thermal Design Power von 225 W, konnte mit keinem der *x*GEMM-Experimente erreicht werden. Die CUBLAS-Routine DGEMM mit einer Matrixgröße von $14 \cdot 2^{10} \times 14 \cdot 2^{10}$, die alle vorhandenen Streaming-Prozessoren belegt, benötigt nur 163 W. Wird die GPU vorher auf rund 85 °C vorgeheizt, beträgt die Leistung 171 W. Das entspricht etwa den Ergebnissen von Kasichayanula et al. [142], die eine Leistung von 180 W für die DGEMM-Routine aus MAGMA [176] beobachten: Auch dort liegt die Leistung bei einer Matrixgröße von 8192×8192 deutlich unter der Thermal Design Power.

5.4 Online-Erzeugung von Leistungsprofilen

Der in Abschnitt 5.2 vorgestellte Ansatz zur Erzeugung von Leistungsprofilen wird nun zu einem Online-Verfahren abgewandelt: Dieses erzeugt Leistungsprofile von Routinen allein anhand der Aufrufe, die in der Anwendung ohnehin auftreten. Während dieser Aufrufe wird die Leistung gemessen und in das Leistungsprofil eingetragen. Im Gegensatz dazu muss die betrachtete Routine beim Offline-Verfahren mehrfach speziell für die Messung ausgeführt werden. Das Online-Verfahren verursacht eine geringere CPU-Last als das Offline-Verfahren und erfordert keine zusätzlichen Ausführungen des GPU-Kernels. Das Verfahren wird in Abb. 5.4 skizziert. Wenn der zu untersuchende GPU-Kernel eine Ausführungszeit von weniger als T hat, reicht es aus, den Wert P_{akt} unmittelbar nach dessen Beendigung auszulesen. Aus der Synchronisationszeit t_{sync} und der Periodendauer T kann der Zeitpunkt der letzten Messung berechnet werden. Wenn die Messung während der Ausführung der GPU-Funktion stattfand, kann der Messwert für das Leistungsprofil genutzt werden.

Der beschriebene Ansatz zur Online-Erzeugung von Leistungsprofilen funktioniert, wenn im System sehr genaue Uhren vorhanden sind. Allerdings haben Experimente gezeigt, dass die Uh-

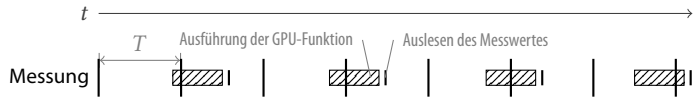


Abbildung 5.4: Erzeugung eines Leistungsprofils mit dem Online-Verfahren

ren der CPU und der GPU leicht auseinanderdriften. Die Leistungsprofile weisen nur dann klare Ausschläge auf, wenn zumindest alle 0,2 s eine Neusynchronisation erfolgt. Wird diese Synchronisation während der Ausführung der GPU-Funktion durchgeführt, erzeugt sie keinen allzu großen Overhead, da die CPU in dieser Zeit häufig nicht vollständig ausgelastet ist. Durch die Vorhersage des Beginns des nächsten Messintervalls und der Beendigungszeit der GPU-Funktion kann abgeschätzt werden, ob die Messung während der Ausführung erfolgen wird.

```
1  function SYNCCLOCKS()
2      retrieve GPU power P_akt
3      P_last := P_akt
4      while P_last = P_akt do
5          P_last := P_akt
6          retrieve GPU power P_akt
7      end
8      retrieve current time t_curr
9      return t_curr
10 end
11 function SYNCIFPOSSIBLE()
12     t_expected_finish := t_start + β t_ex
13     t_next_update := ⌊(t_start − t_sync)/Δt + 1⌋ Δt
14     if t_expected_finish > t_next_update then
15         t_sync := SYNCCLOCKS()
16         n_last_sync := n_call
17     end
18 end
19 function MAIN()
       // main algorithm
20     t_sync := SYNCCLOCKS()
21     n_call := 0
22     n_last_sync := 0
23     while ... do
           // main algorithm
24         n_call := n_call + 1
25         retrieve current time t_start
26         call GPU kernel (asynchronously)
27         SYNCIFPOSSIBLE()
28         wait for GPU kernel to finish
29         retrieve current time t_finish
30         retrieve GPU power P_akt
31         t_φ := t_next_update − t_start
32         emit t_φ, P_akt
33         if n_call − n_last_sync > γ then
34             t_sync := SYNCCLOCKS()
35             n_last_sync := n_call
36         end
           // main algorithm
37     end
38 end
```

Algorithmus 5.2 : Methode zur Online-Erzeugung von Leistungsprofilen

Die Online-Erzeugung der Leistungsprofile erfolgt wie in Alg. 5.2 dargestellt: Die Funktion `syncClocks` (Zeilen 1 bis 9) wartet auf die Aktualisierung des Leistungsmesswerts und gibt die aktuelle Zeit zurück. Die Funktion `main` stellt die Struktur der Anwendung dar, in die die Erzeugung des Leistungsprofils eingebettet ist. Die Schleife in Zeile 23 ruft wiederholt die GPU-Funktion auf (Zeile 26). Bevor die Schleife gestartet wird, wird die Zeit t_{sync} der Aktualisierung des Messwerts bestimmt (Zeile 20).

Die Startzeit t_{start} der GPU-Routine wird in Zeile 25 gemessen. Die Routine wird dann in Zeile 26 asynchron ausgeführt. In der Funktion `syncIfPossible` findet die Synchronisation der Uhren

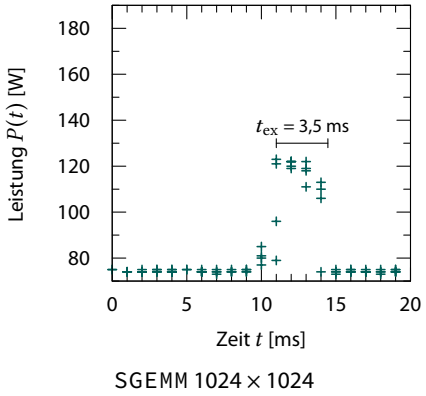


Abbildung 5.5: Mit dem Online-Verfahren erzeugtes Leistungsprofil

während der Ausführung der GPU-Funktion statt: Falls die erwartete Beendigung der GPU-Funktion $t_{\text{expected\_finish}}$ nach der erwarteten Aktualisierung des Leistungsmesswertes $t_{\text{next\_update}}$ liegt, wird der tatsächliche Aktualisierungszeitpunkt durch die Funktion `syncClocks` bestimmt. Das β in Zeile 12 gibt an, wie groß der Sicherheitspuffer vor dem erwarteten Ende der GPU-Funktion ist, in dem keine Synchronisation stattfindet. Hier wurde ein Wert von $\beta = \frac{4}{5}$ gewählt, sodass im letzten Fünftel der Ausführungszeit der GPU-Funktion keine Synchronisation mehr durchgeführt wird. Nach Beendigung der GPU-Funktion (Zeile 28) wird die aktuelle Leistung P_{akt} ausgelesen (Zeile 30). Danach wird t_φ berechnet, also die Phase des Messintervalls, in der die Funktion gestartet wurde (Zeile 31). Die Werte t_φ und P_{akt} werden dann ausgegeben.

Wie oben erwähnt, driften die Uhren von CPU und GPU nach mehr als etwa 0,2 s zu weit auseinander. Deshalb wird eine Synchronisation erzwungen, wenn eine Zeitlang keine Synchronisation durch die Funktion `syncIfPossible` erfolgte (Zeilen 33 bis 35). Der Wert für γ wird so eingestellt, dass die Synchronisation mindestens alle 0,2 s erfolgt.

Der durch die Messung verursachte Overhead in der Anwendung, also die Verlängerung der Gesamtausführungszeit der GPU-Routine, war bei den untersuchten Fällen nicht größer als 10 %. Ein mit dem Verfahren erzeugtes Profil der Routine `SGEMM` mit einer Matrixgröße von 1024×1024 ist in Abb. 5.5 dargestellt. Der Ausschlag ist ähnlich dem in Abb. 5.3a dargestellten. Die Leistungsaufnahme entspricht den in Abschnitt 5.3 ermittelten Werten. Allerdings ist der in Abb. 5.3a sichtbare zweite Ausschlag nicht erkennbar. Möglicherweise erfolgt er zu spät nach Beendigung der Ausführung der GPU-Routine. Aufgrund dieses Effekts können derzeit nur GPU-Routinen mit einer Ausführungszeit von unter 10 ms mit dem Online-Verfahren sinnvoll untersucht werden.

5.5 Diskussion

Das Offline-Verfahren zur Erzeugung der Leistungsprofile arbeitet sehr genau. An den vorgestellten Beispielleistungsprofilen konnten interessante Effekte beobachtet werden wie z. B. das langsame Absinken der Leistung bei einer Routine, deren Threads unterschiedliche Ausführungszeiten haben. Das Verfahren wurde zu einem Online-Verfahren weiterentwickelt, um die Erzeugung der

Leistungsprofile auch während des Ausführens von Anwendungen zu ermöglichen. Das Online-Verfahren verursacht nur einen geringen Overhead. Derart erzeugte Leistungsprofile können beispielsweise für Online-Autotuning genutzt werden. Eine Einschränkung des Online-Verfahrens ist, dass es derzeit nur für Routinen mit einer Ausführungszeit von unter 10 ms genutzt werden kann. Außerdem ist seine Genauigkeit etwas geringer als die des Offline-Verfahrens. Die vorgestellte Methode ist im Allgemeinen nicht auf GPUs beschränkt, sodass sie auch auf andere Messgeräte übertragen werden kann, deren zeitliche Auflösung zu gering für den vorgesehenen Anwendungszweck ist.

5.6 Zusammenfassung

Mit der in diesem Kapitel vorgestellen Methode lassen sich zeitlich hochaufgelöste Leistungsprofile von GPU-Funktionen erzeugen, selbst wenn das Messgerät nur eine niedrige zeitliche Auflösung besitzt. Das Messintervall der durch die NVML angebotenen Schnittstelle beträgt 20 ms. Die daraus resultierende zeitliche Auflösung der Messung ist für GPU-Routinen mit sehr kurzer Ausführungszeit ungenügend. Die Methode führt die Routine mehrfach aus und kombiniert die Messungen zu einem zeitlich hochaufgelösten Leistungsprofil. Damit wird auch der Verlauf der Leistung bei sehr kurzen GPU-Routinen adäquat wiedergegeben. Der Energieverbrauch einer GPU-Routine kann durch Integration der Leistungswerte des Leistungsprofils über die Ausführungszeit berechnet werden werden. Damit können Anwendungsentwickler bei der Optimierung des Energieverbrauchs ihrer Codes unterstützt werden.Die vorgestellte Methode ermöglicht die Erzeugung der Leistungsprofile ohne zusätzliche Hardware.

Zunächst wird ein Offline-Verfahren vorgestellt, das die Routine außerhalb einer Anwendung ausführt und vermisst. Dieses wird erweitert in ein Online-Verfahren, das die innerhalb einer Anwendung ohnehin auftretenden Aufrufe der Routine für die Messungen nutzt. Damit können ohne großen Overhead die Messergebnisse gleich innerhalb der Anwendung weiterverwendet werden, beispielsweise für ein Online-Autotuning. Außerdem werden beispielhaft Leistungsprofile einiger BLAS-Routinen dargestellt und ihr Energieverbrauch angegeben. ■

6 Schluss

Die Arbeit hat sich der effizienten Ausführung zweier Anwendungen des wissenschaftlichen Rechnens gewidmet, einer FMM- und einer FEM-Implementierung. Ziel war die Senkung der Ausführungszeit und des Energieverbrauchs der Anwendungen. Dazu wurden Ausführungszeit- bzw. Energiemodelle aufgestellt. Basierend auf diesen Modellen wurden Autotuning-Verfahren implementiert, die Parameter der Algorithmen bzw. Implementierungen so einstellten, dass die Ausführungszeit bzw. der Energieverbrauch minimiert wurden.

Für die FMM wurde eine Methode entwickelt, die die Kosten einzelner Codeabschnitte misst und anhand eines Analyselaufes mit den Eingabedaten die Gesamtkosten der Ausführung präzise vorhersagt. Sie können minimiert werden, indem anhand der Vorhersage die optimale Baumtiefe für den FMM-Baum ausgewählt wird. Als Kostenmaß wurde zunächst die Ausführungszeit mit der Anzahl verarbeiteter Maschinenbefehle und der Anzahl benötigter CPU-Takte der CPU genutzt. Es wurde gezeigt, dass sich das Kostenmodell auch für die Minimierung des Energieverbrauchs eignet, wenn die benötigte Energie als Kostenmaß verwendet wird.

Bei der FEM wird mit einem Modell die Ausführungszeit der dort genutzten CGM auf der CPU und auf der GPU vorhergesagt. Anhand dieser Vorhersage wird die Rechenlast dynamisch zwischen CPU und GPU aufgeteilt. Das Autotuning-Verfahren ist hier ein Online-Verfahren, sodass auf die sich ändernden Eigenschaften der Daten im Verlauf der Ausführung Rücksicht genommen werden kann. Außerdem wurde für die CGM ein Energiemodell entwickelt, das die energieeffizienteste Art und Weise der Ausführung vorhersagt. Das kann dann entweder eine Ausführung nur auf der CPU bzw. nur auf der GPU sein oder eine gemeinsame Ausführung auf der CPU und der GPU nach der erwähnten Methode zur Minimierung der Ausführungszeit.

In weiteren Beiträgen stellte die Arbeit für die FEM eine Methode zur NUMA-gerechten Verarbeitung der finiten Elemente sowie eine Methode zur Beschleunigung der Reduktion des Ergebnisvektors der CGM vor. Die Reduktionsmethode ersetzt die explizite Reduktion am Ende eines parallelen Abschnitts durch Schreiben auf einen gemeinsamen Vektor mit atomaren Operationen. Auf diese Weise können Berechnungen und Speicheroperationen miteinander verwoben werden, sodass für die Synchronisation des Schreibvorgangs kein zusätzlicher Aufwand entsteht.

Die außerdem vorgestellte Methode zur Energiemessung für GPU-Routinen mit kurzer Laufzeit ist notwendig zum Aufstellen und Verifizieren von Energiemodellen für GPUs. Sie erreicht durch mehrfache Ausführung der GPU-Routinen zu unterschiedlichen Phasen des Messintervalls, dass Leistungsprofile mit einer höheren zeitlichen Auflösung als der von 20 ms, die die GPU anbietet, erstellt werden können. Durch Integration der Leistungsprofile über die Zeit lässt sich der Energiebedarf errechnen.

6.1 Modellbasiertes Autotuning

Die Arbeit hat gezeigt, dass sich mit modellbasiertem Autotuning im wissenschaftlichen Rechnen Energie und Zeit bei der Ausführung von Simulationsanwendungen einsparen lässt. Das modellbasierte Autotuning ersetzt die empirische Suche der besten Implementierungsvariante im Autotuning durch ein analytisches Ermitteln des Optimums anhand eines Modells. Zum Aufstellen eines Ausführungsmodells eines Anwendungsprogramms sind zunächst dessen wesentliche Bestandteile zu ermitteln. Diese werden experimentell untersucht, um Gesetzmäßigkeiten zu finden, die in ein Modell für die Leistung (engl. *performance*) der Implementierung einfließen. Als Leistungsziel kann beispielsweise die Minimierung der Ausführungszeit oder des Energieverbrauchs festgelegt werden.

Zum Aufstellen des Modells hat es sich als sinnvoll erwiesen, das Programm zunächst in Teile zu untergliedern, deren Verhalten sich möglichst einfach beschreiben lässt, und aus Teilmodellen das Gesamtmodell zusammenzusetzen. Das Modell enthält *Algorithmenparameter*, deren Veränderung die Ausführungszeit, den Energieverbrauch, den Grad der Parallelität, die Cacheausnutzung usw. der Ausführung beeinflusst. Außerdem sollte das Modell *rechner- und datenspezifische Parameter* enthalten, damit das Autotuning die Ausführung an die genutzte Hardware bzw. die Eingabedaten anpassen kann.

Bei der FMM ist der Algorithmenparameter die Baumtiefe. Rechnerspezifische Parameter sind die Kosten der einzelnen Codeabschnitte. Datenspezifische Parameter hingegen sind die Anzahlen der Ausführungen jedes Codeabschnitts, die im Analyselauf gemessen werden. Bei der FEM ist der Algorithmenparameter der Anteil der auf der GPU verarbeiteten Elemente. Rechnerspezifische Parameter sind hier die Ausführungszeiten bzw. Energieverbräuche der Ausführung der Routine AXMEBE sowie der Datenübertragung. Ein datenspezifischer Parameter ist die Gesamtanzahl der finiten Elemente in jedem Verfeinerungsschritt.

Das für das modellbasierte Autotuning benötigte Modell wird in der Regel in „kreativer“ Arbeit [173] durch den Anwender aufgestellt werden. Teilweise wird das manuelle Aufstellen von Modellen allerdings als zu „schwer“[1] empfunden [30], weshalb auch Techniken des maschinellen Lernens bzw. der Regressionsanalyse eingesetzt werden [30, 248, 166]. Letztendlich muss aber auch bei diesen (teil-)automatisierten Verfahren der Anwender die zu nutzende Methode und ihre Rahmenbedingungen vorgeben. Anwendungsentwickler kennen meist ihre Software und die verwendeten Algorithmen sehr gut. Bei genauer Kenntnis der Algorithmen sollte es in der Regel möglich sein, dafür Ausführungsmodelle aufzustellen. Eine allgemein gültige Methode zur Herleitung der Modelle kann trotzdem nicht angegeben werden: Rohrlich zufolge gehöre das „Aufstellen theoretischer Modelle“ zur „‚Kunst‘ der Wissenschschaft“[2] [232].

Es reicht aus, wenn das Modell das Ausführungsverhalten der Anwendung nur grob beschreibt. Unzulänglichkeiten im Modell können durch Online-Autotuning ausgeglichen werden: Wenn gewisse entscheidende Parameter immer wieder gemessen werden, kann so dynamisch auf deren Veränderung im Verlaufe der Simulation reagiert werden. Das Weglassen der Beschreibung dieser Veränderung im Modell vereinfacht so dessen Aufstellen für den Anwender. Wie auch Suda et

1 “the analytical model, which remains hard to build by hand” [30]

2 “There are no rules for constructing theoretical models; they are based on educated guesses and highly trained intuition. They belong to the ‘art’ of doing science.” [232]

al. feststellen [251], kann beim Online-Autotuning ohnehin nur eine begrenzte Anzahl von Parametern gemessen werden, sodass es sinnvoll ist, nur wenige Parameter vorzusehen. Ein weiterer Vorteil des Online-Autotunings gegenüber dem Offline-Autotuning ist, dass es den Einbezug der Eingabedaten in das Tuning ermöglicht.

Um das Konzept des modellbasierten Autotunings einer breiteren Anwenderschaft zugänglich zu machen und die Implementierung zu vereinfachen, erscheint es sinnvoll, zwischen Autotuner und Anwendung zu trennen. Dafür sollten einheitliche Schnittstellen definiert werden, wie beispielsweise auch in der Exascale-Roadmap als Meilensteine für die Jahre 2014 bis 2015 skizziert wird [66, Tab. 15, S. 38]. Der Anwender würde dann seinen Quelltext mit einem Modell annotieren und Informationen bereitstellen, wie die rechner-, und datenabhängigen Parameter ermittelt werden können. Diese Informationen können beispielsweise auch Markierungen des Beginns und des Endes von Codeabschnitten sein, für die der Autotuner automatisch die Ausführungszeit oder den Energieverbrauch misst. Nach Durchführung der Optimierung werden die ermittelten Algorithmenparameter an die Anwendung übergeben, sodass sie damit ihre Ausführung optimieren kann.

6.2 Ausblick

Eine offene Frage ist, auf welcher Hardware künftige Hochleistungsrechner aufbauen. Grundsätzlich existieren drei Möglichkeiten [154]: eine schwergewichtige aus herkömmlichen High-End-Multicore-CPUs, eine leichtgewichtige aus energiesparenden, langsameren CPUs sowie eine heterogene aus Multicore-CPUs mit leichtgewichtigen Beschleunigern. In [220] wurde eine homogene leichtgewichtige Architektur vorgestellt, die sich aus energiesparenden RISC-CPUs aufbaut, die eigentlich für eingebettete Geräte und Mobiltelefone entworfen wurden. Bei dieser Art von Cluster müsste die gewohnte Art des parallelen Rechnens nicht aufgegeben werden. Im Gegensatz dazu stehen heterogene Architekturen, die nach [154] eine höhere Energieeffizienz versprechen: Als General-Purpose-Beschleuniger, auf die die Haupt-CPU rechenintensive Teilaufgaben auslagern kann, existieren neben der GPU heute der Xeon Phi [138] von Intel und die Synergistic Processing Elements des Cell-Prozessors [114, 271] von IBM. Alternativ sind auch anwendungsspezifische Beschleuniger denkbar [255]: Hartenstein [106] sowie Shalf et al. [239] erwarten, dass künftig nicht mehr die von-Neumann-Architektur weiterverfolgt wird, sondern dass fest verdrahtete, aber rekonfigurierbare FPGAs *(field-programmable gate array)* als Beschleuniger verwendet werden. FPGAs seien energieeffizienter [50] und viele Beispiele zeigten, dass Speedups von z. T. mehreren Größenordnungen gegenüber CPUs bzw. GPUs möglich sind.

Eine ähnliche Entwicklung, wie sie im wissenschaftlichen Rechnen erwartet wird, kann beim Bitcoin-Mining [197] nachvollzogen werden: Dort wurden bis Mitte 2010 vorrangig CPUs zum Minen benutzt, danach bis Ende 2011 GPUs und von 2012 bis Anfang 2013 FPGAs. Seit Februar 2013 werden ASICs, also anwendungsspezifische integrierte Schaltungen (engl. *application-specific integrated circuit*), verwendet [45]. Beim Bitcoin-Mining kommt es darauf an, dass man für den Vorgang des „Mining“ weniger für die benötigte Energie bezahlt als man durch den Verkauf der erzeugten Bitcoins erzielt. Außerdem ist es wichtig, dass man seine Rechenleistung ständig steigert, da der Aufwand zum „Minen“ eines Bitcoins immer weiter steigt. Da beim Bitcoin-Mining viel Geld

zu verdienen und die Anwendung vergleichsweise wenig komplex ist, kann vermutet werden, dass sich die Entwicklungen hier schneller vollziehen als beim wissenschaftlichen Rechnen.

In dieser Arbeit wurde die Methode des modellbasierten Autotuning nur auf CPU- und GPU-Computing angewandt. Sie sollte aber ohne Probleme auf andere Beschleunigertypen wie Xeon Phi oder FPGAs übertragbar sein: Die Messung der Ausführungszeit bzw. des Energieverbrauchs ist dort genauso möglich wie für GPUs. Sogar die vorgestellte Methode zur dynamischen Verteilung der Rechenlast ist problemlos auf andere Beschleuniger übertragbar: Sofern dort die Routine AXMEBE aus Abschnitt 4.1.2 implementiert und die Übertragung der Daten möglich ist, kann statt der GPU ein anderer Beschleuniger die finiten Elemente verarbeiten.

Des Weiteren erscheint es naheliegend, die Energie- bzw. Ausführungszeitmodelle auch für das Scheduling zu nutzen [260, 120, 70]. Wenn der Autotuner bereits eine präzise Vorhersage der Ausführungszeit besitzt, kann der Scheduler diese über einheitliche Schnittstellen auslesen. Er kann dann besser abschätzen, wann und wo die Ausführung der Anwendung am besten einzuplanen ist. Ist der Energiebedarf bzw. die benötigte elektrische Leistung im Vorhinein bekannt, können die Anwendungen so in einem Cluster verteilt werden, dass die zur Verfügung stehende elektrische Leistung bestmöglich ausgenutzt wird. Häufig existieren Beschränkungen, wie viel Leistung ein bestimmter Teil eines Clusters maximal verbrauchen darf. Die Summe der theoretischen Maximalleistungen der einzelnen Knoten dieses Teils übersteigt jedoch häufig diese Leistungsbeschränkung. Ist nun der Leistungsbedarf der Anwendungen bekannt, kann eine Anwendung mit hohem Leistungsbedarf gemeinsam mit einer Anwendung mit niedrigem Leistungsbedarf in einem Teil des Clusters ausgeführt werden, ohne dass sie sich einschränken müssen. Ergibt sich durch das Modell eine Vorhersage von Wartezeiten, beispielsweise aufgrund von Kommunikation, kann der Leerlaufmodus der CPU passend zur Wartezeit gewählt werden [34]. Sie könnte auch rechtzeitig wieder in den Betriebszustand versetzt werden, sodass bei maximaler Energieeinsparung keine Rechenzeit verlorengeht.

6.3 Fazit

Die Computersimulation wurde nach Gramelsberger notwendig, um die wachsende Kluft zwischen Theorie und Experiment zu schließen, die ab dem 19. Jahrhundert in der Naturwissenschaft auftrat [93, S. 206]: Mit den existierenden Theorien ließen sich nur sehr einfache Phänomene beschreiben, da entweder nur empirische Gesetze für Spezialfälle bekannt gewesen seien oder die Mathematik der allgemeinen Theorien zu komplex war [93, S. 69]. Erst die Computersimulation habe es ermöglicht, Modelle beispielsweise aus nicht-linearen Differentialgleichungen auf praktische Probleme anzuwenden [93, S. 79]. Dabei erhöhe die „Quantität der Rechenkraft“ die Qualität der Ergebnisse [93, S. 255]. Zu dieser Erhöhung der „Quantität der Rechenkraft“ trägt die vorliegende Arbeit (zu einem kleinen Teil) bei. Sie hilft, dass Simulationsrechner – wie es die Verantwortlichen der Green-500-Liste formulieren – „die Klimaveränderung simulieren, aber nicht verursachen“[3] [94]. ■

3 “We […] ensure that supercomputers only simulate climate change and not create it.” [94]

Literaturverzeichnis

[1] Abe, Yuki, Hiroshi Sasaki, Martin Peres, Koji Inoue, Kazuaki Murakami und Shinpei Kato: *Power and Performance Analysis of GPU-accelerated Systems.* In: *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12, Berkeley, CA, USA, 2012. USENIX Association.

[2] Advanced Micro Devices: *Advanced Platform Management Link (APML) Specification*, August 2009. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/09/41918.pdf. 41918 Rev 1.02.

[3] *AMD PowerTune Technology.* Whitepaper, Advanced Micro Devices, Dezember 2010.

[4] Advanced Micro Devices: *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors*, Oktober 2012. http://support.amd.com/us/Processor_TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf. Rev 3.12. [14.01. 2013].

[5] Advanced Micro Devices: *AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions*, Mai 2013. http://developer.amd.com/wordpress/media/2008/10/24594_APM_v3.pdf. Rev 3.20. [28.01. 2014].

[6] Agricola, Erhard, Wolfgang Fleischer und Helmut Protze (Herausgeber): *Kleine Enzyklopädie – Die Deutsche Sprache.* Bibliographisches Institut, Leipzig, 1. Auflage, 1969.

[7] Agulleiro, Jose Ignacio, Francisco Vázquez, Ester Martín Garzón und José Jesús Fernández: *Hybrid computing: CPU+GPU co-processing and its application to tomographic reconstruction.* Ultramicroscopy, Bd. 115, Nr. 0, S. 109–114, 2012. doi: 10.1016/j.ultramic.2012.02.003.

[8] Aliaga, José Ignacio, Joaquín Pérez, Enrique S. Quintana-Ortí und Hartwig Anzt: *Reformulated Conjugate Gradient for the Energy-Aware Solution of Linear Systems on GPUs.* In: *Proceedings of the 42nd International Conference on Parallel Processing 2013*, S. 320–329, 2013. doi: 10.1109/ICCP.2013.41.

[9] Aluru, Srinivas: *Greengard's N-Body Algorithm is not Order N.* SIAM Journal on Scientific Computing, Bd. 17, Nr. 3, S. 773–776, 1996. doi: 10.1137/S1064827593272031.

[10] Ansel, Jason, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman und Saman Amarasinghe: *Language and compiler support for auto-tuning variable-accuracy algorithms.* In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, S. 85–96, Washington, DC, USA, 2011. IEEE Computer Society. isbn: 978-1-61284-356-8. doi: 10.1109/CGO.2011.5764677.

[11] Antony, Joseph, Alistair P. Rendell, Rui Yang, Gary Trucks und Michael J. Frisch: *Modelling the Runtime of the Gaussian Computational Chemistry Application and Assessing the Impacts of Microarchitectural Variations.* Procedia Computer Science, Bd. 4, Nr. 0, S. 281–291, 2011. doi: 10.1016/j.procs.2011.04.030. Proceedings of the International Conference on Computational Science, {ICCS} 2011.

[12] Anzt, Hartwig, Maribel Castillo, Juan C. Fernández, Vincent Heuveline, Francisco D. Igual, Rafael Mayo und Enrique S. Quintana-Ortí: *Optimization of power consumption in the iterative solution of sparse linear systems on graphics processors.* Computer Science – Research and Development, Bd. 27, Nr. 4, S. 299–307, 2012. doi: 10.1007/s00450-011-0195-8.

[13] Anzt, Hartwig, Vincent Heuveline, José I. Aliaga, Maribel Castillo, Juan C. Fernandez, Rafael Mayo und Enrique S. Quintana-Ortí: *Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms.* In: *2011 International Green Computing Conference and Workshops (IGCC)*, S. 1–6, Juli 2011. doi: 10.1109/IGCC.2011.6008594.

[14] ARM: *Cortex™-A9 NEON™ Media Processing Engine: Technical Reference Manual*, 2010. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0409f/DDI0409F_cortex_a9_neon_mpe_r2p2_trm.pdf. Revision r2p2, [Abruf: 2014-04-14].

[15] Bailey, David H., Eric Barszcz, John T. Barton, Don S. Browning, Robert L. Carter, Leonardo Dagum, Rod Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Robert Schreiber, Horst D. Simon, V. Venkatakrishnan und Sisira Weeratunga: *The NAS Parallel Benchmarks.* International Journal of High Performance Computing Applications, Bd. 5, Nr. 3, S. 63–73, 1991. doi: 10.1177/109434209100500306.

[16] Balakrishnan, Ganesh: *Understanding Intel Xeon 5500 Turbo Boost Technology.* How to use Turbo Boost Technology to your advantage, IBM, Juni 2009. http://public.dhe.ibm.com/common/ssi/rep_wh/n/XSW03045USEN/XSW03045USEN.PDF. [Abruf: 2014-04-24].

[17] Balaprakash, Prasanna, Ananta Tiwari und Stefan M. Wild: *Multi-Objective Optimization of HPC Kernels for Performance, Power, and Energy.* Preprint, Argonne National Laboratory, April 2013. http://www.mcs.anl.gov/uploads/cels/papers/P4069-0413.pdf.

[18] Balg, Martina, Jens Lang, Arnd Meyer und Gudula Rünger: *Array-based reduction operations for a parallel adaptive FEM.* In: Keller, Rainer, David Kramer und Jan-Philipp Weiß (Herausgeber): *Facing the Multicore Challenge III*, Band 7686 der Reihe *Lecture Notes in Computer Science*, S. 25–36. Springer, Berlin, Heidelberg, 2013. isbn: 978-3-642-35892-0. doi: 10.1007/978-3-642-35893-7_3.

[19] Ballard, Grey, James Demmel, Olga Holtz und Oded Schwartz: *Minimizing Communication in Numerical Linear Algebra.* SIAM Journal on Matrix Analysis and Applications, Bd. 32, Nr. 3, S. 866–901, 2011. doi: 10.1137/090769156.

[20] BAPCo: *SYSmark 2012.* http://bapco.com/products/sysmark-2012. [Abruf: 2014-02-12].

[21] Bardine, Alessandro, Pierfrancesco Foglia, Cosimo Antonio Prete und Marco Solinas: *NUMA Caches.* In: Padua, David (Herausgeber): *Encyclopedia of Parallel Computing*, S. 1329–1338. Springer, 2011. isbn: 978-0-387-09765-7. doi: 10.1007/978-0-387-09766-4_16.

[22] Barnes, Josh und Piet Hut: *A hierarchical O(N log N) force-calculation algorithm.* Nature, Bd. 324, S. 446–449, Dezember 1986. doi: 10.1038/324446a0.

[23] Barroso, Luiz André und Urs Hölzle: *The Case for Energy-Proportional Computing.* Computer, Bd. 40, Nr. 12, S. 33–37, Dezember 2007. doi: 10.1109/MC.2007.443.

[24] Behle, Brad, Nick Bofferding, Martha Broyles, Curtis Eide, Michael Floyd, Chris Francois, Andrew Geissler, Michael Hollinger, Hye-Young McCreary, Cale Rath, Todd Rosedahl, Guillermo Silva und Christine Wang: *IBM EnergyScale for POWER6 Processor-Based Systems.* IBM, Oktober 2009. http://public.dhe.ibm.com/common/ssi/ecm/en/pow03002usen/POW03002USEN.PDF. [Abruf: 2014-04-24].

[25] Bellosa, Frank: *The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems.* In: *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, EW 9, S. 37–42, New York, NY, USA, 2000. ACM. doi: 10.1145/566726.566736.

[26] Bennett, Charles H.: *Notes on Landauer's principle, reversible computation, and Maxwell's Demon.* Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics, Bd. 34, Nr. 3, S. 501–510, 2003. doi: http://dx.doi.org/10.1016/S1355-2198(03)00039-X. Sonderausgabe „Quantum Information and Computation".

[27] Benz, Benjamin: *Nachbrenner: Prozessor-Turbos von AMD und Intel.* c't – Magazin für Computertechnik, , Nr. 16, S. 170–175, 2010.

[28] Benz, Benjamin: *Leistung ohne Reue: Mainboards für Intels Haswell-Prozessoren.* c't – Magazin für Computertechnik, , Nr. 16, S. 142, 2013.

[29] Berczik, Peter, Rainer Spurzem, Shiyan Zhong, Long Wang, Keigo Nitadori, Tsuyoshi Hamada und Alexander Veles: *Up to 700k GPU Cores, Kepler, and the Exascale Future for Simulations of Star Clusters Around Black Holes.* In: Kunkel, Julian Martin, Thomas Ludwig und Hans Werner Meuer (Herausgeber): *Supercomputing*, Band 7905 der Reihe *Lecture Notes in Computer Science*, S. 13–25. Springer Berlin Heidelberg, 2013. isbn: 978-3-642-38749-4. doi: 10.1007/978-3-642-38750-0_2.

[30] Bergstra, James, Nicolas Pinto und David Cox: *Machine learning for predictive auto-tuning with boosted regression trees.* In: *Innovative Parallel Computing (InPar), 2012*, S. 1–9, May 2012. doi: 10.1109/InPar.2012.6339587.

[31] Beuchler, Sven, Arnd Meyer und Matthias Pester: *SPC-Pm3AdH v1.0 – Programmer's Manual.* Preprint SFB393 01-08, TU Chemnitz, 2001, revised 2003.

[32] Bibliographisches Institut & F. A. Brockhaus: *Brockhaus-Enzyklopädie Online.* http://www.brockhaus-enzyklopaedie.de. [Abruf: 2013-05-29].

[33] Bilmes, Jeff, Krste Asanović, Chee whye Chin und Jim Demmel: *Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology.* In: *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997.

[34] Bircher, W. Lloyd und Lizy K. John: *Analysis of dynamic power management on multi-core processors.* In: *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, S. 327–338, New York, NY, USA, 2008. ACM. isbn: 978-1-60558-158-3. doi: 10.1145/1375527.1375575.

[35] Bircher, W. Lloyd und Lizy K. John: *Complete System Power Estimation Using Processor Performance Events.* IEEE Transactions on Computers, Bd. 61, Nr. 4, S. 563–577, April 2012. doi: 10.1109/TC.2011.47.

[36] Bosilca, George, Aurélien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier und Jack Dongarra: *DAGuE: A generic distributed DAG engine for High Performance Computing.* Parallel Computing, Bd. 38, Nr. 1–2, S. 37–51, 2012. doi: 10.1016/j.parco.2011.10.003.

[37] Bosilca, George, Aurélien Bouteiller, Thomas Herault, Pierre Lemarinier, Narapat Ohm Saengpatsa, Stanimire Tomov und Jack J. Dongarra: *Performance Portability of a GPU Enabled Factorization with the DAGuE Framework.* In: *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, S. 395–402, sep 2011. doi: 10.1109/CLUSTER.2011.51.

[38] Braess, Dietrich: *Finite Elemente: Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie.* Springer-Lehrbuch Masterclass. Springer, Berlin, Heidelberg, 2013. isbn: 978-3-642-34796-2. doi: 10.1007/978-3-642-34797-9.

[39] Bremermann, Hans J.: *Minimum energy requirements of information transfer and computing.* International Journal of Theoretical Physics, Bd. 21, Nr. 3-4, S. 203–217, 1982. doi: 10.1007/BF01857726.

[40] Brooks, David, Vivek Tiwari und Margaret Martonosi: *Wattch: A Framework for Architectural-level Power Analysis and Optimizations.* SIGARCH Computer Architecture News, Bd. 28, Nr. 2, S. 83–94, Mai 2000. doi: 10.1145/342001.339657.

[41] Brown, David J. und Charles Reams: *Toward energy-efficient computing.* Commun. ACM, Bd. 53, Nr. 3, S. 50–58, März 2010. doi: 10.1145/1666420.1666438.

[42] Brown, Len, Konstantin A. Karasyov, Vladimir P. Lebedev, Alexey Y. Starikovskiy und Randy P. Stanley: *Linux Laptop Battery Life: Measurement Tools, Techniques, and Results.* In: *Proceedings of the Linux Symposium*, S. 127–146, Ottawa, Ontario (Kanada), July 2006.

[43] Browne, Shirley, Jack Dongarra, George Ho Nathan Garner und Phil Mucci: *A Portable Programming Interface for Performance Evaluation on Modern Processors.* International Journal of High Performance Computing Applications, Bd. 14, Nr. 3, S. 189–204, 2000. doi: 10.1177/109434200001400303.

[44] Broyles, Martha, Chris Francois, Andrew Geissler, Gordon Grout, Michael Hollinger, Todd Rosedahl, Guillermo J. Silva, Mark Vanderwiel, Jeff van Heuklon und Brian Veale: *IBM EnergyScale for POWER7 Processor-Based Systems.* IBM, März 2013. http://public.dhe.ibm.com/common/ssi/ecm/en/pow03039usen/POW03039USEN.PDF. [Abruf: 2014-01-24].

[45] Bögeholz, Harald und Fabian A. Scherschel: *Währung im Kollektiv: So funktioniert die Kryptowährung Bitcoin.* c't – Magazin für Computertechnik, , Nr. 25, S. 146–149, 2013.

[46] Carrington, Laura, Allan Snavely und Nicole Wolter: *A performance prediction framework for scientific applications.* Future Generation Computer Systems, Bd. 22, Nr. 3, S. 336–346, 2006. doi: 10.1016/j.future.2004.11.019.

[47] Cevahir, Ali, Akira Nukada und Satoshi Matsuoka: *High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning.* Computer Science - Research and Development, Bd. 25, Nr. 1-2, S. 83–91, 2010. doi: 10.1007/s00450-010-0112-6.

[48] Chandrakasan, Anantha P., Samuel Sheng und Robert W. Brodersen: *Low-power CMOS digital design.* IEEE Journal of Solid-State Circuits, Bd. 27, Nr. 4, S. 473–484, April 1992. doi: 10.1109/4.126534.

[49] Chen, Chun, Jacqueline Chame und Mary Hall: *Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy.* In: *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, S. 111–122, Washington, DC, USA, 2005. IEEE Computer Society. isbn: 0-7695-2298-X. doi: 10.1109/CGO.2005.10.

[50] Chen, Doris und Deshanand Singh: *Using OpenCL to evaluate the efficiency of CPUs, GPUs and FPGAs for information filtering.* In: *22nd International Conference on Field Programmable Logic and Applications (FPL), 2012*, S. 5–12, aug 2012. doi: 10.1109/FPL.2012.6339171.

[51] Chen, Xi, Chi Xu, Robert P. Dick und Zhuoqing Morley Mao: *Performance and power modeling in a multi-programmed multi-core environment.* In: *Proceedings of the 47th Design Automation Conference*, DAC '10, S. 813–818, New York, NY, USA, 2010. ACM. isbn: 978-1-4503-0002-5. doi: 10.1145/1837274.1837479.

[52] Cho, Sangyeun und Rami Melhem: *Corollaries to Amdahl's Law for Energy.* IEEE Computer Architecture Letters, Bd. 7, Nr. 1, S. 25–28, Januar 2008. doi: 10.1109/L-CA.2007.18.

[53] Choi, Jee W., Amik Singh und Richard W. Vuduc: *Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs.* SIGPLAN Not., Bd. 45, Nr. 5, S. 115–126, Januar 2010. doi: 10.1145/1837853.1693471.

[54] Choi, Jee Whan und Richard W. Vuduc: *How Much (Execution) Time and Energy Does My Algorithm Cost?* XRDS, Bd. 19, Nr. 3, S. 49–51, März 2013. doi: 10.1145/2425676.2425691.

[55] Chung, I-Hsin und Jeffrey K. Hollingsworth: *Using Information from Prior Runs to Improve Automated Tuning Systems.* In: *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, S. 30–41, November 2004. doi: 10.1109/SC.2004.65.

[56] Collange, Sylvain, David Defour und Arnaud Tisserand: *Power Consumption of GPUs from a Software Perspective.* In: *9th Int. Conf. on Computational Science*, 2009. isbn: 978-3-642-01969-2. doi: 10.1007/978-3-642-01970-8_92.

[57] Cruz, Felipe A., Matthew G. Knepley und Lorena A. Barba: *PetFMM—A dynamically load-balancing parallel fast multipole library.* Internat. J. Numer. Methods Engrg., Bd. 85, Nr. 4, S. 403–428, 2011. doi: 10.1002/nme.2972.

[58] Dachsel, Holger: *An error-controlled fast multipole method.* Journal of Chemical Physics, Bd. 132, Nr. 119901, 2010. doi: 10.1063/1.3358272.

[59] Dachsel, Holger, Michael Hofmann, Jens Lang und Gudula Rünger: *Automatic Tuning of the Fast Multipole Method Based on Integrated Performance Prediction.* In: *14th IEEE International Conference on High Performance Computing and Communications (HPCC-2012)*, 2012.

[60] David, Howard, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna und Christian Le: *RAPL: memory power estimation and capping.* In: *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '10, S. 189–194, New York, NY, USA, 2010. ACM. isbn: 978-1-4503-0146-6. doi: 10.1145/1840845.1840883.

[61] Davidson, Andrew, Yao Zhang und John D. Owens: *An Auto-tuned Method for Solving Large Tridiagonal Systems on the GPU.* In: *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, S. 956–965, May 2011. doi: 10.1109/IPDPS.2011.92.

[62] Demmel, James und Andrew Gearhart: *Instrumenting Linear Algebra Energy Consumption via On-chip Energy Counters.* Technischer Bericht UCB/EECS-2012-168, EECS Department, University of California, Berkeley, Juni 2012.
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-168.html.

[63] Demmel, James, Laura Grigori, Mark Hoemmen und Julien Langou: *Communication-optimal Parallel and Sequential QR and LU Factorizations.* SIAM Journal on Scientific Computing, Bd. 34, Nr. 1, S. A206–A239, 2012. doi: 10.1137/080731992.

[64] Diamos, Gregory F. und Sudhakar Yalamanchili: *Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems.* In: *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, S. 197–200, New York, NY, USA, 2008. ACM. isbn: 978-1-59593-997-5. doi: 10.1145/1383422.1383447.

[65] Dongarra, Jack: *Visit to the National University for Defense Technology Changsha, China.* Reisebericht, Juni 2013.
http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf. [Abruf: 2013-06-17].

[66] Dongarra, Jack, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad van der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski und Kathy Yelick: *The International Exascale Software Project roadmap.* International Journal of High Performance Computing Applications, Bd. 25, Nr. 1, S. 3–60, 2011. doi: 10.1177/1094342010391989.

[67] Dongarra, Jack J.: *Performance of Various Computers Using Standard Linear Equations Software.* Technischer Bericht CS - 89- 85, University of Manchester, Februar 2013. [Abruf: 2014-02-10].

[68] Dongarra, Jack J., Jeremy Du Croz, Sven Hammarling und Richard J. Hanson: *An extended set of FORTRAN basic linear algebra subprograms.* ACM Trans. Math. Softw., Bd. 14, Nr. 1, S. 1–17, März 1988. doi: 10.1145/42288.42291.

[69] Dotsenko, Yuri, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd und John Manferdelli: *Fast Scan Algorithms on Graphics Processors.* In: *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, S. 205–213, New York, NY, USA, 2008. ACM. isbn: 978-1-60558-158-3. doi: 10.1145/1375527.1375559.

[70] Dümmler, Jörg, Thomas Rauber und Gudula Rünger: *Semi-dynamic Scheduling of Parallel Tasks for Heterogeneous Clusters.* In: *10th International Symposium on Parallel and Distributed Computing (ISPDC), 2011*, S. 1–8, July 2011. doi: 10.1109/ISPDC.2011.11.

[71] Eckhardt, Wolfgang, Alexander Heinecke, Reinhold Bader, Matthias Brehm, Nicolay Hammer, Herbert Huber, Hans-Georg Kleinhenz, Jadran Vrabec, Hans Hasse, Martin Horsch, Martin Bernreuther, Colin W. Glass, Christoph Niethammer, Arndt Bode und Hans-Joachim Bungartz: *591 TFLOPS Multi-trillion Particles Simulation on SuperMUC.* In: Kunkel, Julian Martin, Thomas Ludwig und Hans-Werner Meuer (Herausgeber): *Supercomputing*, Band 7905 der Reihe *Lecture Notes in Computer Science*, S. 1–12. Springer Berlin Heidelberg, 2013. isbn: 978-3-642-38749-4. doi: 10.1007/978-3-642-38750-0_1.

[72] *Encyclopædia Britannica.* http://www.britannica.com/. 2013. [Abruf: 2013-06-12].

[73] Enzensberger, Hans Magnus: *Enzensbergers Panoptikum.* Suhrkamp, Berlin, 1. Auflage, 2012. isbn: 9783518069011.

[74] Etinski, Maja, Julita Corbalán, Jesús Labarta und Mateo Valero: *Understanding the future of energy-performance trade-off via DVFS in HPC environments.* Journal of Parallel and Distributed Computing, Bd. 72, Nr. 4, S. 579–590, 2012. doi: 10.1016/j.jpdc.2012.01.006.

[75] Fatica, Massimiliano: *Accelerating linpack with CUDA on heterogenous clusters.* In: *2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, S. 46–51, New York, NY, USA, 2009. ACM.

[76] Feller, Scott E., Richard W. Pastor, Atipat Rojnuckarin, Stephen Bogusz und Bernard R. Brooks: *Effect of Electrostatic Force Truncation on Interfacial and Transport Properties of Water.* The Journal of Physical Chemistry, Bd. 100, Nr. 42, S. 17011–17020, 1996. doi: 10.1021/jp9614658.

[77] Feng, Wu-chun und Kirk W. Cameron: *The Green500 List: Encouraging Sustainable Supercomputing.* Computer, Bd. 40, Nr. 12, S. 50–55, 2007. doi: 10.1109/MC.2007.445.

[78] Fogarty, Joseph, Hasan Aktulga, Sagar Pandit und Ananth Y. Grama: *N-Body Computational Methods.* In: Padua, David (Herausgeber): *Encyclopedia of Parallel Computing*, S. 1259–1268. Springer, 2011. doi: 10.1007/978-0-387-09766-4_97.

[79] Frasca, Michael, Anirban Chatterjee und Padma Raghavan: *Can models of scientific software-hardware interactions be predictive?* Procedia Computer Science, Bd. 4, Nr. 0, S. 322 – 331, 2011. doi: 10.1016/j.procs.2011.04.034. Proceedings of the International Conference on Computational Science (ICCS 2011).

[80] Frigo, Matteo und Steven G. Johnson: *The Design and Implementation of FFTW3.* Proceedings of the IEEE, Bd. 93, Nr. 2, S. 216–231, February 2005. doi: 10.1109/JPROC.2004.840301.

[81] Gandhi, Anshul, Mor Harchol-Balter, Rajarshi Das, Jeffrey O. Kephart und Charles Lefurgy: *Power Capping Via Forced Idleness.* In: *Proceedings of Workshop on Energy-Efficient Design (WEED 09)*, Austin, Texas, Juni 2009.

[82] Gao, Da und Thomas E. Schwartzentruber: *Optimizations and OpenMP implementation for the direct simulation Monte Carlo method.* Computers & Fluids, Bd. 42, Nr. 1, S. 73–81, 2011. doi: 10.1016/j.compfluid.2010.11.004.

[83] Garland, Michael, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang und Vasily Volkov: *Parallel Computing Experiences with CUDA.* IEEE Micro, Bd. 28, Nr. 4, S. 13–27, Juli 2008. doi: 10.1109/MM.2008.57.

[84] Ge, Rong, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li und Kirk W. Cameron: *PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications.* IEEE Transactions on Parallel and Distributed Systems, Bd. 21, Nr. 5, S. 658–671, Mai 2010. doi: 10.1109/TPDS.2009.76.

[85] Gellert, Walter, Herbert Küstner, Manfred Hellwich und Herbert Kästner (Herausgeber): *Kleine Enzyklopädie Mathematik.* Bibliographisches Institut Leipzig, 6. Auflage, 1971.

[86] Gilbert, Nigel und Klaus G. Troitzsch: *Simulation for the Social Scientist.* Open University Press, Maidenhead (Großbritannien), 2. Auflage, 2005. isbn: 0-335-21601-3.

[87] Glänzel, Janine: *Kurzvorstellung der 3D-FEM Software SPC-PM3AdH-XX.* Chemnitz Scientific Computing Preprints CSC/09-03, TU Chemnitz, Chemnitz (Deutschland), Januar 2009. http://nbn-resolving.de/urn:nbn:de:bsz:ch1-200900211.

[88] *gnuplot homepage.* http://www.gnuplot.info/. [Abruf: 2014-02-03].

[89] Goel, Bhavishya, Sally A. McKee, Roberto Gioiosa, Karan Singh, Major Bhadauria und Marco Cesati: *Portable, scalable, per-core power estimation for intelligent resource management.* In: *Green Computing Conference, 2010 International*, S. 135–146, August 2010. doi: 10.1109/GREENCOMP.2010.5598313.

[90] Golub, Gene und James M. Ortega: *Scientific Computing – An Introduction with Parallel Computing.* Academic Press, San Diego, CA, 1993.

[91] Gonzalez, Ricardo und Mark Horowitz: *Energy dissipation in general purpose microprocessors.* Solid-State Circuits, IEEE Journal of, Bd. 31, Nr. 9, S. 1277–1284, 1996. doi: 10.1109/4.535411.

[92] Gordana, Dodig-Crnkovic: *Scientific Methods in Computer Science.* In: *Proc. Conf. for the Promotion of Research in IT at New Universities and at University Colleges in Sweden*, 2002.

[93] Gramelsberger, Gabriele: *Computerexperimente.* transcript Verlag, Bielefeld (Deutschland), 1. Auflage, Januar 2010. isbn: 978-3-89942-986-2.

[94] *The Green500.* http://www.green500.org. [Abruf: 2014-04-28].

[95] Greengard, Leslie und William D. Gropp: *A parallel version of the fast multipole method.* Computers & Mathematics with Applications, Bd. 20, Nr. 7, S. 63–71, 1990. doi: 10.1016/0898-1221(90)90349-O.

[96] Greengard, Leslie und Vladimir Rokhlin: *A fast algorithm for particle simulations.* Journal of Computational Physics, Bd. 73, Nr. 2, S. 325–348, 1987. doi: 10.1016/0021-9991(87)90140-9.

[97] Greenwald, Michael Barry: *Non-blocking synchronization and system design.* Dissertation, Stanford University, Stanford, CA, USA, 1999.

[98] Griebel, Michael, Stephan Knapek, Gerhard Zumbusch und Attila Caglar: *Numerische Simulation in der Moleküldynamik.* Springer-Verlag, 2004. isbn: 3-540-41856-3.

[99] Gueron, Shay: *Intel Advanced Encryption Standard (AES) New Instructions Set.* White Paper 323641-001, Intel Corporation, September 2013. http://download-software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf. Revision 3.01.

[100] Gupta, Vishal, Paul Brett, David Koufaty, Dheeraj Reddy, Scott Hahn, Karsten Schwan und Ganapati Srinivasa: *The Forgotten 'Uncore': On the Energy-Efficiency of Heterogeneous Cores.* In: *Proceedings of USENIX ATC 2012 Short Paper*, 2012.

[101] Gurumurthi, Sudhanva, Anand Sivasubramaniam, Mary Jane Irwin, Narayanan Vijaykrishnan und Mahmut Kandemir: *Using complete machine simulation for software power estimation: the SoftWatt approach.* In: *Eighth International Symposium on High-Performance Computer Architecture, 2002. Proceedings.*, S. 141–150, Februar 2002. doi: 10.1109/HPCA.2002.995705.

[102] Göschel, Heinz (Herausgeber): *Meyers neues Lexikon.* Bibliographisches Institut Leipzig, Leipzig, 2. Auflage, 1971–1978.

[103] Götz, Sebastian, Claas Wilke, Sebastian Cech und Uwe Aßmann: *Architecture and Mechanisms for Energy Auto Tuning.* In: Kaabouch, Naima und Wen-Chen Hu (Herausgeber): *Sustainable Green Computing: Practices, Methodologies and Technologies*, Kapitel 3., S. 45–73. IGI Global, Hershey, PA (USA), 1. Auflage, Juni 2011. isbn: 9781466618398. doi: 10.4018/978-1-4666-1839-8.ch003.

[104] Götz, Sebastian, Claas Wilke, Matthias Schmidt, Sebastian Cech und Uwe Aßmann: *Towards Energy Auto-Tuning.* In: *Proceedings of First Annual International Conference on Green Information Technology (GREEN IT)*, 2010.

[105] Hackenberg, Daniel, Thomas Ilsche, Robert Schöne, Daniel Molka, Maik Schmidt und Wolfgang E. Nagel: *Power measurement techniques on standard compute nodes: A quantitative comparison.* 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Bd. 0, S. 194–204, 2013. doi: 10.1109/ISPASS.2013.6557170.

[106] Hartenstein, Reiner: *The Paramountcy of Reconfigurable Computing.* In: Zomaya, Albert Y. und Young Choon Lee (Herausgeber): *Energy-Efficient Distributed Computing Systems*, S. 81–108. John Wiley & Sons, Inc., 2012. isbn: 9781118342015. doi: 10.1002/9781118342015.ch18.

[107] Hartmann, Stephan: *The World as a Process.* In: Hegselmann, Rainer, Ulrich Mueller und Klaus G. Troitzsch (Herausgeber): *Modelling and Simulation in the Social Sciences from the Philosophy of Science Point of View*, Band 23 der Reihe *Theory and Decision Library*, S. 77–100. Springer Netherlands, 1996. isbn: 978-90-481-4722-9. doi: 10.1007/978-94-015-8686-3_5.

[108] Heisenberg, Werner: *Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik.* Zeitschrift für Physik, Bd. 43, Nr. 3-4, S. 172–198, 1927. doi: 10.1007/BF01397280.

[109] Henkel, Jörg und Yanbing Li: *Energy-conscious HW/SW-partitioning of Embedded Systems: A Case Study on an MPEG-2 Encoder.* In: *Proceedings of the 6th International Workshop on Hardware/Software Codesign*, CODES/CASHE '98, S. 23–27, Washington, DC, USA, 1998. IEEE Computer Society. isbn: 0-8186-8442-9.

[110] Hestenes, Magnus R. und Eduard Stiefel: *Methods of conjugate gradients for solving linear systems.* Journal of Research of the National Bureau of Standards, Bd. 49, Nr. 6, S. 409–436, Dezember 1952.

[111] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation: *Advanced Configuration and Power Interface Specification*, December 2011. http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf. Revision 5.0.

[112] Hippold, Judith und Gudula Rünger: *Performance Analysis for Parallel Adaptive FEM on SMP Clusters.* In: Dongarra, Jack, Kaj Madsen und Jerzy Waśniewski (Herausgeber): *Applied Parallel Computing. State of the Art in Scientific Computing*, Band 3732 der Reihe *Lecture Notes in Computer Science*, S. 730–739. Springer Berlin Heidelberg, 2006. isbn: 978-3-540-29067-4. doi: 10.1007/11558958_88.

[113] Hoffmann, Karl Heinz, Michael Hofmann, Jens Lang, Gudula Rünger und Steffen Seeger: *Accelerating Physical Simulations Using Graphics Processing Units.* it – Information Technology, Bd. 53, Nr. 2, S. 49–59, 2011. doi: 10.1524/itit.2011.0625.

[114] Hofstee, H. Peter: *Power efficient processor architecture and the cell processor.* In: *11th International Symposium on High-Performance Computer Architecture, 2005. HPCA-11*, S. 258–262, Feb 2005. doi: 10.1109/HPCA.2005.26.

[115] Hoisie, Adolfy, Olaf Lubeck und Harvey Wasserman: *Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications.* International Journal of High Performance Computing Applications, Bd. 14, Nr. 4, S. 330–346, 2000. doi: 10.1177/109434200001400405.

[116] Hong, Chun-Tao, De-Hao Chen, Yu-Bei Chen, Wen-Guang Chen, Wei-Min Zheng und Hai-Bo Lin: *Providing Source Code Level Portability Between CPU and GPU with MapCG.* Journal of Computer Science and Technology, Bd. 27, S. 42–56, 2012.

[117] Hornby, Albert Sidney (Herausgeber): *Oxford Advanced Learner's Dictionary.* Oxford Elt, 8. Auflage, 2010. isbn: 978-0194799003.

[118] Horowitz, Mark, Thomas Indermaur und Ricardo Gonzalez: *Low-power digital design.* In: *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, S. 8–11, Oktober 1994. doi: 10.1109/LPE.1994.573184.

[119] Hsu, Chung-Hsing, Jeffery A. Kuehn und Stephen W. Poole: *Towards Efficient Supercomputing: Searching for the Right Efficiency Metric.* In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, S. 157–162, New York, NY, USA, 2012. ACM. isbn: 978-1-4503-1202-8. doi: 10.1145/2188286.2188309.

[120] Hsu, Chung-Hsing und Stephen W. Poole: *Power measurement for high performance computing: State of the art.* In: *Green Computing Conference and Workshops (IGCC), 2011 International*, S. 1–6, 2011. doi: 10.1109/IGCC.2011.6008596.

[121] Huang, Song, Shucai Xiao und Wu-chun Feng: *On the energy efficiency of graphics processing units for scientific computing.* In: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, S. 1 –8, may 2009. doi: 10.1109/IPDPS.2009.5160980.

[122] Hähnel, Marcus, Björn Döbel, Marcus Völp und Hermann Härtig: *Measuring Energy Consumption for Short Code Paths Using RAPL.* In: *GREENMETRICS'12*, London, UK, Juni 2012.

[123] Illingworth, Valerie (Herausgeber): *A Dictionary of Computing.* Oxford University Press, Oxford (Großbritannien), 5. Auflage, 2004. isbn: 0-19-860877-2.

[124] Im, Eun-Jin, Katherine Yelick und Richard Vuduc: *Sparsity: Optimization Framework for Sparse Matrix Kernels.* International Journal of High Performance Computing Applications, Bd. 18, Nr. 1, S. 135–158, 2004. doi: 10.1177/1094342004041296.

[125] Intel: *Intel Turbo Boost Technology—On-Demand Processor Performance.* http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html. [Abruf: 2013-10-25].

[126] Intel: *Intel Xeon Processor E5-2650*, Intel. http://ark.intel.com/products/47922. [Abruf: 2013-11-11].

[127] Intel: *Intel Xeon Processor X5650.* http://ark.intel.com/products/64590. [Abruf: 2013-09-06].

[128] Intel: *Intel 80386 Programmer's Reference Manual*, 1986.

[129] Intel: *Pentium® processor with MMX™ technology*, Juni 1997. http://download.intel.com/design/archives/processors/mmx/docs/24318504.pdf. Bestellnummer: 243185.004, [Abruf: 2014-04-14].

[130] *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor.* White Paper Order Number: 301170-001, Intel, März 2004. ftp://download.intel.com/design/network/papers/30117401.pdf.

[131] Intel: *Intel microprocessor export compliance metrics: Intel Xeon E5-2600 Product Family.* http://download.intel.com/support/processors/xeon/sb/xeon_E5-2600.pdf. September 2011. [Abruf: 2013-11-11].

[132] Intel: *Intel microprocessor export compliance metrics: Intel Xeon Processor 5600 Series.* http://download.intel.com/support/processors/xeon/sb/xeon_5600.pdf. September 2011. [Abruf: 2013-11-11].

[133] Intel: *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2012.

[134] Intel: *Intel Xeon Processor E5-2600 Product Family Uncore Performance Monitoring Guide*, März 2012. http://www.intel.com/content/dam/www/public/us/en/documents/design-guides/xeon-e5-2600-uncore-guide.pdf. Reference Number: 327043-001.

[135] Intel, Hewlett-Packard, NEC, Dell: *IPMI: Intelligent Platform Management Interface Specification, Second Generation*, Februar 2009. http://www.intel.com/content/www/us/en/servers/ipmi/second-gen-interface-spec-v4.html. Document Revision 1.0, June 12, 2009 Markup.

[136] Isci, Canturk und Margaret Martonosi: *Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data.* In: *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, S. 93, Washington, DC (USA), 2003. IEEE Computer Society. isbn: 0-7695-2043-X.

[137] Jacobitz, Karl und Ernst Eduard Seiler: *Wörterbuch der griechiſchen Sprache*, Band 2: Deutſch – Griechiſch. J. C. Hinrichs'ſche Buchhandlung, Leipzig, Zweite vielfach vermehrte und verbeſſerte Auflage, 1871.

[138] Jeffers, Jim und James R. Reinders: *Intel Xeon Phi Coprocessor High-Performance Programming*. Morgan Kaufmann, Waltham, MA, 2013. isbn: 978-0-12-410414-3.

[139] Jochem, Eberhard (Herausgeber): *Steps towards a sustainable development.* novatlantis, Dübendorf (Schweiz), März 2004. isbn: 3-9522705-0-4.

[140] Jones, John Edward: *On the Determination of Molecular Fields.—II. From the Equation of State of a Gas.* Proceedings of the Royal Society of London. Series A, Bd. 106, Nr. 738, S. 463–477, 1924. doi: 10.1098/rspa.1924.0082.

[141] Kabadshow, Ivo: *Periodic Boundary Conditions and the Error-Controlled Fast Multipole Method.* Dissertation, Bergische Universität Wuppertal, September 2010.

[142] Kasichayanula, Kiran, Dan Terpstra, Piotr Luszczek, Stan Tomov, Shirley Moore und Gregory Peterson: *Power Aware Computing on GPUs.* In: *2012 Symposium on Application Accelerators in High-Performance Computing*, 2012.

[143] Katagiri, Takahiro, Cheng Luo, Reiji Suda, Shoichi Hirasawa und Satoshi Ohshima: *Energy Optimization for Scientific Programs Using Auto-tuning Language ppOpen-AT.* In: *2013 IEEE 7th International Symposium on Embedded Multicore Socs*, S. 123–128, Los Alamitos, CA, USA, September 2013. IEEE Computer Society. doi: 10.1109/MCSoC.2013.14.

[144] Katagiri, Takahiro, Satoshi Ohshima und Satoshi Ito: *Early Experiences for Adaptation of Auto-tuning by ppOpen-AT to an Explicit Method.* In: *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on*, S. 153–158, Sept 2013. doi: 10.1109/MCSoC.2013.15.

[145] Keckler, Stephen W., William J. Dally, Brucek Khailany, Michael Garland und David Glasco: *GPUs and the Future of Parallel Computing.* IEEE Micro, Bd. 31, Nr. 5, S. 7–17, 2011. doi: 10.1109/MM.2011.89.

[146] Kennedy, Ken und John R. Allen: *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. isbn: 1-55860-286-0.

[147] Kerbyson, Darren J., Hank J. Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J. Wasserman und Michael Gittings: *Predictive Performance and Scalability Modeling of a Large-scale Application.* In: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, Supercomputing '01, S. 37–37, New York, NY, USA, 2001. ACM. isbn: 1-58113-293-X. doi: 10.1145/582034.582071.

[148] Kerbyson, Darren J., Adolfy Hoisie und Shawn D. Pautz: *Performance Modeling of Deterministic Transport Computations.* In: Getov, Vladimir, Michael Gerndt, Adolfy Hoisie, Allen Malony und Barton Miller (Herausgeber): *Performance Analysis and Grid Computing*, S. 21–39. Springer US, 2004. isbn: 978-1-4613-5038-5. doi: 10.1007/978-1-4615-0361-3_2.

[149] Kerbyson, Darren J., Adolfy Hoisie und Harvey J. Wasserman: *Modeling the Performance of Large-Scale Systems.* IEE Proceedings – Software, Bd. 150, Nr. 4, S. 214–221, August 2003. doi: 10.1049/ip-sen:20030808.

[150] Kerrisk, Michael: *The Linux Programming Interface.* No Starch Press, San Francisco, CA (USA), 1. Auflage, 2010. isbn: 978-1-59327-220-3.

[151] Kidd, Taylor: *C-states and P-states are very different*, Intel. http://software.intel.com/en-us/blogs/2008/03/12/c-states-and-p-states-are-very-different. Dezember 2008. [Abruf: 2014-03-19].

[152] Kim, Nam Sung, Todd Austin, David Baauw, Trevor Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir und Vijaykrishnan Narayanan: *Leakage current: Moore's law meets static power.* Computer, Bd. 36, Nr. 12, S. 68–75, 2003. doi: 10.1109/MC.2003.1250885.

[153] Kleen, Andreas, Richard Brunner, Mark Langsdorf und Meike Chabowski: *A NUMA API for LINUX.* Technical Linux Whitepaper, Novell, April 2005. http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf.

[154] Kogge, Peter und John Shalf: *Exascale Computing Trends: Adjusting to the "New Normal" for Computer Architecture.* Computing in Science Engineering, Bd. 15, Nr. 6, S. 16–26, November 2013. doi: 10.1109/MCSE.2013.95.

[155] Koomey, Jonathan G., Stephen Berard, Marla Sanchez und Henry Wong: *Implications of Historical Trends in the Electrical Efficiency of Computing.* Annals of the History of Computing, IEEE, Bd. 33, Nr. 3, S. 46–54, 2011. doi: 10.1109/MAHC.2010.28.

[156] Korthikanti, Vijay Anand und Gul Agha: *Towards Optimizing Energy Costs of Algorithms for Shared Memory Architectures.* In: *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, S. 157–165, New York, NY (USA), 2010. ACM. isbn: 978-1-4503-0079-7. doi: 10.1145/1810479.1810510.

[157] Kubota, Yuji und Daisuke Takahashi: *Optimization of Sparse Matrix-Vector Multiplication by Auto Selecting Storage Schemes on GPU.* In: Murgante, Beniamino, Osvaldo Gervasi, Andrés Iglesias, David Taniar und BernadyO. Apduhan (Herausgeber): *Computational Science and Its Applications - ICCSA 2011*, Band 6783 der Reihe *Lecture Notes in Computer Science*, S. 547–561. Springer Berlin Heidelberg, 2011. isbn: 978-3-642-21886-6. doi: 10.1007/978-3-642-21887-3_42.

[158] Lameter, Christoph: *NUMA (Non-Uniform Memory Access): An Overview.* Queue, Bd. 11, Nr. 7, S. 40:40–40:51, Juli 2013. doi: 10.1145/2508834.2513149.

[159] Landauer, Rolf: *The physical nature of information.* Physics Letters A, Bd. 217, Nr. 4–5, S. 188–193, 1996. doi: http://dx.doi.org/10.1016/0375-9601(96)00453-7.

[160] Lang, Jens: *Performanzanalyse paralleler Algorithmen auf NUMA-Architekturen.* Studienarbeit, Fakultät für Informatik, Technische Universität Chemnitz, 2008.

[161] Lang, Jens: *Grüner verschlüsseln – Messung des Energieverbrauchs von kryptografischen Algorithmen.* In: Team der Chemnitzer Linux-Tage (Herausgeber): *Chemnitzer Linux-Tage 2014 – Tagungsband.* Technische Universität Chemnitz, Universitätsbibliothek: Universitätsverlag Chemnitz, 2014. isbn: 978-3-941003-52-1.

[162] Lang, Jens und Gudula Rünger: *Dynamic Distribution of Workload Between CPU and GPU for a Parallel Conjugate Gradient Method in an Adaptive FEM.* Procedia Computer Science, Bd. 18, S. 299–308, 2013. doi: 10.1016/j.procs.2013.05.193. Proceedings of the International Conference on Computational Science, ICCS 2013.

[163] Lang, Jens und Gudula Rünger: *High-Resolution Power Profiling of GPU Functions Using Low-Resolution Measurement.* In: Wolf, Felix, Bernd Mohr und Dieter an Mey (Herausgeber): *Euro-Par 2013 Parallel Processing*, Band 8097 der Reihe *Lecture Notes in Computer Science*, S. 801–812, Berlin, Heidelberg, 2013. Springer. isbn: 978-3-642-40046-9. doi: 10.1007/978-3-642-40047-6_80.

[164] Lang, Jens und Gudula Rünger: *An execution time and energy model for an energy-aware execution of a conjugate gradient method with CPU/GPU collaboration.* Journal of Parallel and Distributed Computing, 2014. doi: 10.1016/j.jpdc.2014.06.001. (zum Druck angenommen).

[165] Lang, Jens und Gudula Rünger: *Measuring and modelling energy consumption for a CPU/GPU conjugate gradient method in an adaptive FEM.* In: *Proceedings of the High-Level Programming for Heterogeneous and Hierarchical Parallel Systems workshop at HiPEAC conference 2014*, Wien (Österreich), 2014.

[166] Lee, Benjamin C. und David M. Brooks: *Accurate and efficient regression modeling for microarchitectural performance and power prediction.* SIGOPS Oper. Syst. Rev., Bd. 40, Nr. 5, S. 185–194, Oktober 2006. doi: 10.1145/1168917.1168881.

[167] Lenk, Richard und Walter Gellert (Herausgeber): *Brockhaus abc Physik.* F. A. Brockhaus Verlag, Leipzig, 1. Auflage, 1972, 1973.

[168] Li, Yinan, Jack Dongarra und Stanimire Tomov: *A Note on Auto-tuning GEMM for GPUs.* In: *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, S. 884–892, Berlin, Heidelberg, 2009. Springer-Verlag. isbn: 978-3-642-01969-2. doi: 10.1007/978-3-642-01970-8_89.

[169] Liu, Zhenying, Barbara Chapman, Yi Wen, Lei Huang, Tien-Hsiung Weng und Oscar Hernandez: *Analyses for the Translation of OpenMP Codes into SPMD Style with Array Privatization.* In: Voss, MichaelJ. (Herausgeber): *OpenMP Shared Memory Parallel Programming*, Band 2716 der Reihe *Lecture Notes in Computer Science*, S. 26–41. Springer Berlin Heidelberg, 2003. isbn: 978-3-540-40435-4. doi: 10.1007/3-540-45009-2_3.

[170] Lorenz, Markus, Peter Marwedel, Thorsten Dräger, Gerhard Fettweis und Rainer Leupers: *Compiler Based Exploration of DSP Energy Savings by SIMD Operations.* In: *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, S. 838–841, Piscataway, NJ, USA, 2004. IEEE Press. isbn: 0-7803-8175-0.

[171] Lovins, Amory und Peter Hennicke: *Voller Energie.* Campus Verlag, 1. Auflage, 1999. isbn: 978-3593360386.

[172] Lu, Qingda, Sriram Krishnamoorthy und P. Sadayappan: *Combining Analytical and Empirical Approaches in Tuning Matrix Transposition.* In: *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, S. 233–242, New York, NY, USA, 2006. ACM. isbn: 1-59593-264-X. doi: 10.1145/1152154.1152190.

[173] Luderer, Bernd: *Vorwort.* In: Luderer, Bernd (Herausgeber): *Die Kunst des Modellierens – Mathematisch-ökonomische Modelle*, Teubner Studienbücher Wirtschaftsmathematik. Vieweg+Teubner Verlag, Wiesbaden (Deutschland), 1. Auflage, 2008. isbn: 978-3-8351-0212-5.

[174] Lugato, David: *Model-driven engineering for high-performance computing applications.* In: *Proceedings of the 19th IASTED International Conference on Modelling and Simulation*, MS '08, S. 303–308, Anaheim, CA, USA, 2008. ACTA Press. isbn: 978-0-88986-764-2.

[175] Lux, Gerhard: *Wie gut sind Wettervorhersagen?* Broschüre des Deutschen Wetterdienstes, 2009.

[176] *MAGMA.* http://icl.cs.utk.edu/magma/. [Abruf: 2012-12-11].

[177] Malkowski, Korad, Ingyu Lee, Padma Raghavan und Mary Jane Irwin: *Conjugate gradient sparse solvers: performance-power characteristics.* In: *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, S. 297–297, Washington, DC, USA, 2006. IEEE Computer Society. isbn: 1-4244-0054-6.

[178] Mamatas, Lefteris und Vassilis Tsaoussidis: *Transport Protocol Behavior and Energy-Saving Potential.* In: *Proceedings of the 31st IEEE Conference on Local Computer Networks*, S. 889–896, November 2006. doi: 10.1109/LCN.2006.322196.

[179] Margi, Cintia B., Katia Obraczka und Roberto Manduchi: *Characterizing system level energy consumption in mobile computing platforms.* In: *Wireless Networks, Communications and Mobile Computing, 2005 International Conference on*, Band 2, S. 1142 – 1147 vol.2, june 2005. doi: 10.1109/WIRLES.2005.1549573.

[180] Martin, Alain J.: *Towards an energy complexity of computation.* Information Processing Letters, Bd. 77, Nr. 2–4, S. 181–187, 2001. doi: 10.1016/S0020-0190(00)00214-3.

[181] McCreary, Hye-Young, Martha A. Broyles, Michael S. Floyd, Andrew J. Geissler, S. P. Hartman, Freeman L. Rawson, Todd J. Rosedahl, Juan C. Rubio und Malcolm S. Ware: *Energyscale for IBM POWER6 microprocessor-based systems.* IBM Journal of Research and Development, Bd. 51, Nr. 6, S. 775–786, November 2007. doi: 10.1147/rd.516.0775.

[182] McCullough, John C., Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren und Rajesh K. Gupta: *Evaluating the effectiveness of model-based power characterization.* In: *Proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIXATC'11)*, 2011.

[183] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard*, 2009. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf. Version 2.2.

[184] Meyer, Arnd: *Adaptive FEM – Einführung und Beispielnutzung.* Vortrag vom 26.11.2009 (HFT PE eniPROD).

[185] Meyer, Arnd: *A parallel preconditioned conjugate gradient method using domain decomposition and inexact solvers on each subdomain.* Computing, Bd. 45, S. 217–234, 1990. doi: 10.1007/BF02250634.

[186] Meyer, Jan Christian, Juan Manuel Cebrian, Lasse Natvig, Vasileios Karakasis, Dimitris Siakavaras und Konstantinos Nikas: *Energy-Efficient Sparse Matrix Autotuning with CSX – A Trade-off Study.* In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, S. 931–937, Washington, DC, USA, 2013. IEEE Computer Society. isbn: 978-0-7695-4979-8. doi: 10.1109/IPDPSW.2013.219.

[187] Meyer, Jan Christian, Lasse Natvig, Vasileios Karakasis, Dimitris Siakavaras und Konstantinos Nikas: *Energy-efficient Sparse Matrix Auto-tuning with CSX.* PRACE Whitepaper, Partnership for Advanced Computing in Europe, März 2013. http://www.prace-project.eu/IMG/pdf/wp57_energy_efficient_sparse_matrix_auto-tuning_with_csx.pdf.

[188] Miceli, Renato, Gilles Civario, Anna Sikora, Eduardo César, Michael Gerndt, Houssam Haitof, Carmen Navarrete, Siegfried Benkner, Martin Sandrieser, Laurent Morin und François Bodin: *AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications.* In: Manninen, Pekka und Per Öster (Herausgeber): *Applied Parallel and Scientific Computing*, Band 7782 der Reihe *Lecture Notes in Computer Science*, S. 328–342. Springer Berlin Heidelberg, 2013. isbn: 978-3-642-36802-8. doi: 10.1007/978-3-642-36803-5_24.

[189] Misra, Janardan und Indranil Saha: *Artificial neural networks in hardware: A survey of two decades of progress.* Neurocomputing, Bd. 74, Nr. 1–3, S. 239–255, 2010. doi: 10.1016/j.neucom.2010.03.021.

[190] Mittal, Sparsh: *A Survey of Architectural Techniques for DRAM Power Management.* International Journal of High Performance Systems Architecture, Bd. 4, Nr. 2, S. 110–119, Dezember 2012. doi: 10.1504/IJHPSA.2012.050990.

[191] Montoye, Robert K., Erdem Hokenek und Stephen L. Runyon: *Design of the IBM RISC System/6000 floating-point execution unit.* IBM Journal of Research and Development, Bd. 34, Nr. 1, S. 59–70, Januar 1990. doi: 10.1147/rd.341.0059.

[192] Mudge, Trevor: *Power: a first-class architectural design constraint.* Computer, Bd. 34, Nr. 4, S. 52–58, 2001. doi: 10.1109/2.917539.

[193] Murphy, Richard C., Kyle B. Wheeler, Brian W. Barrett und James A. Ang: *Introducing the Graph 500.* In: *Cray User Group (CUG) 2010 Proceedings*, Mai 2010.

[194] Müssig, Florian: *Strom to go: So funktionieren Lithium-Ionen-Akkus.* c't – Magazin für Computertechnik, , Nr. 2, S. 174–183, 2014.

[195] Naffziger, Samuel D., John P. Petry, Kiran K. Bondalapati und Mom Eng Ng: *Flexible power reporting in a computing system.* Patent US 8442786 B2, 14. Mai 2013.

[196] Nair, Ravi: *Exascale Computing.* In: Padua, David (Herausgeber): *Encyclopedia of Parallel Computing*, S. 638–644. Springer, 2011. doi: 10.1007/978-0-387-09766-4_68.

[197] Nakamoto, Satoshi: *Bitcoin: A peer-to-peer electronic cash system.* https://bitcoin.org/bitcoin.pdf. 2008. [Abruf: 2014-03-03].

[198] Nelson, Yoonju Lee, Bhupesh Bansal, Mary Hall, Aiichiro Nakano und Kristina Lerman: *Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization.* Parallel and Distributed Processing Symposium, International, Bd. 0, S. 1–8, 2008. doi: 10.1109/IPDPS.2008.4536189.

[199] Newton, Isaac: *Philosophiæ naturalis principia mathematica*, Band 1: Tomus Primus. Barrillot & Filii, Genf, Schweiz, 1739.

[200] Nickolls, John und William J. Dally: *The GPU Computing Era.* IEEE Micro, Bd. 30, Nr. 2, S. 56–69, März 2010. doi: 10.1109/MM.2010.41.

[201] Nikas, Konstantinos, Dimitris Siakavaras, Vasileios Karakasis, Jan Christian Meyer, und Lasse Natvig: *An Energy-centric Study of Conjugate Gradient Method.* PRACE Whitepaper, Partnership for Advanced Computing in Europe, August 2013.

[202] Nudd, G. R., D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper und D. V. Wilcox: *Pace–A Toolset for the Performance Prediction of Parallel and Distributed Systems.* International Journal of High Performance Computing Applications, Bd. 14, Nr. 3, S. 228–251, August 2000.

[203] Nvidia: *GeForce GTX 570 | Specifications.* http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-570/specifications. [Abruf: 2013-11-11].

[204] *PowerMizer 8.0: Intelligent Power Management Technology.* Technischer Bericht, Nvidia, June 2008. TB-04051-001_v01.

[205] *Nvidia Tesla C2075 Companion Processor.* Data sheet, Nvidia, Sept. 2011. http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf.

[206] Nvidia: *Tesla C2075 Computing Processor Board. Board Specification*, 2011. http://www.nvidia.com/docs/IO/43395/BD-05880-001_v02.pdf. BD-05880-001_v02.

[207] Nvidia: *NVML API Reference Manual.* http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/NVML/nvml.pdf. April 2012. Version 3.295.45.

[208] Nvidia: *CUDA C Programming Guide*, Juli 2013. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. PG-02829-001_v5.5 [28.01. 2014].

[209] Nyland, Lars, Mark Harris und Jan Prins: *Fast N-Body Simulation with CUDA.* In: Nguyen, Hubert (Herausgeber): *GPU Gems 3*, Kapitel 31. Addison-Wesley Professional, 1. Auflage, August 2007. isbn: 9780321545428.

[210] OpenMP Architecture Review Board: *OpenMP Application Program Interface*, Juli 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf. Version 4.0.

[211] Pallipadi, Venkatesh, Shaohua Li und Adam Belay: *cpuidle—Do nothing, efficiently…* In: *Proceedings of the Linux Symposium*, Band 2, S. 119–126, Ottawa, Ontario (Kanada), Juni 2007.

[212] Pallipadi, Venkatesh und Alexey Starikovskiy: *The Ondemand Governor: Past, Present and Future.* In: *Proceedings of Linux Symposium*, Band 2, S. 223–238, Ottawa, Ontario (Kanada), 2006.

[213] Panda, Preeti Ranjan, Aviral Shrivastava, B. Venkata Naga Silpa und Krishnaiah Gummidipudi: *Basic Low Power Digital Design.* In: *Power-efficient System Design*, S. 11–39. Springer, 2010. isbn: 978-1-4419-6387-1. doi: 10.1007/978-1-4419-6388-8_2.

[214] Paoloni, Gabriele: *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures.* Whitepaper 324264-001, Intel Corporation, September 2010. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

[215] Patterson, Murray G: *What is energy efficiency?: Concepts, indicators and methodological issues.* Energy Policy, Bd. 24, Nr. 5, S. 377–390, 1996. doi: 10.1016/0301-4215(96)00017-1.

[216] Post, Douglass E. und Lawrence G. Votta: *Computational Science Demands a New Paradigm.* Physics Today, Bd. 58, Nr. 1, S. 35–41, 2005. doi: 10.1063/1.1881898.

[217] Preis, Tobias, Peter Virnau, Wolfgang Paul und Johannes J. Schneider: *GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model.* Journal of Computational Physics, Bd. 228, Nr. 12, S. 4468 – 4477, 2009. doi: 10.1016/j.jcp.2009.03.018.

[218] Püschel, Markus, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson und Nicholas Rizzolo: *SPIRAL: Code Generation for DSP Transforms.* Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation", Bd. 93, Nr. 2, S. 232–275, 2005.

[219] Rahman, Shah Faizur, Jichi Guo und Qing Yi: *Automated Empirical Tuning of Scientific Codes for Performance and Power Consumption.* In: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, S. 107–116, New York, NY, USA, 2011. ACM. isbn: 978-1-4503-0241-8. doi: 10.1145/1944862.1944880.

[220] Rajovic, Nikola, Lluis Vilanova, Carlos Villavieja, Nikola Puzovic und Alex Ramirez: *The low power architecture approach towards exascale computing.* Journal of Computational Science, Bd. 4, Nr. 6, S. 439 – 443, 2013. doi: http://dx.doi.org/10.1016/j.jocs.2013.01.002. Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.

[221] RAMpedia: *Memory Power Consumption – DRAM IDD*, RAMpedia. http://www.rampedia.com/index.php/ae2a. [Abruf: 2013-11-25].

[222] Rauber, Thomas und Gudula Rünger: *Towards an Energy Model for Modular Parallel Scientific Applications.* In: *Proceedings IEEE International Conference on Green Computing and Communications (GreenCom), 2012*, S. 523–532, 2012. doi: 10.1109/GreenCom.2012.79.

[223] Rauber, Thomas und Gudula Rünger: *Modeling and analyzing the energy consumption of fork-join-based task parallel programs.* Concurrency and Computation: Practice and Experience, 2014. doi: 10.1002/cpe.3219.

[224] Rauber, Thomas, Gudula Rünger, Michael Schwind, Haibin Xu und Simon Melzner: *Energy Measurement, Modeling, and Prediction for Processors with Frequency Scaling.* Journal of Supercomputing, 2014. (im Druck).

[225] Ravi, V. T., M. Becchi, W. Jiang, G. Agrawal und S. Chakradhar: *Scheduling Concurrent Applications on a Cluster of CPU-GPU Nodes.* In: *12th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, S. 140–147, Mai 2012.

[226] Ren, Da Qi und Reiji Suda: *Investigation on the power efficiency of multi-core and GPU Processing Element in large scale SIMD computation with CUDA.* In: *Proceedings of the International Conference on Green Computing*, GREENCOMP '10, S. 309–316, Washington, DC, USA, 2010. IEEE Computer Society. isbn: 978-1-4244-7612-1. doi: 10.1109/GREENCOMP.2010.5598300.

[227] Rendell, Alistair P., Joseph Antony, Warren Armstrong, Pete Janes und Rui Yang: *Building fast, reliable, and adaptive software for computational science.* Journal of Physics: Conference Series, Bd. 125, Nr. 1, S. 012015, 2008. doi: 10.1088/1742-6596/125/1/012015.

[228] Ries, Daniel R. und Michael Stonebraker: *Effects of locking granularity in a database management system.* ACM Trans. Database Syst., Bd. 2, Nr. 3, S. 233–246, 1977. doi: 10.1145/320557.320566.

[229] Rivoire, S., M.A. Shah, P. Ranganathan, C. Kozyrakis und J. Meza: *Models and Metrics to Enable Energy-Efficiency Optimizations.* Computer, Bd. 40, Nr. 12, S. 39–48, 2007. doi: 10.1109/MC.2007.436.

[230] Rivoire, Suzanne, Mehul A. Shah, Parthasarathy Ranganathan und Christos Kozyrakis: *JouleSort: a balanced energy-efficiency benchmark.* In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, S. 365–376, New York, NY, USA, 2007. ACM. doi: 10.1145/1247480.1247522.

[231] Rofouei, Mahsan, Thanos Stathopoulos, Sebi Ryffel anf William Kaiser und Majid Sarrafzadeh: *Energy-aware high performance computing with graphic processing units.* In: *Workshop on Power Aware Computing and Systems (HotPower'08)*, 2008.

[232] Rohrlich, Fritz: *Computer Simulation in the Physical Sciences.* PSA: Proceedings of the Biennial Meeting of the Philosophy of Science Association, Bd. 1990, S. 507–518, 1990.

[233] Rotem, Efraim, Alon Naveh, Avinash Ananthakrishnan, Doron Rajwan und Eliezer Weissmann: *Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge.* IEEE Micro, Bd. 32, Nr. 2, S. 20–27, 2012. doi: 10.1109/MM.2012.12.

[234] Rountree, Barry, Dong H. Ahn, Bronis R. de Supinski, David K. Lowenthal und Martin Schulz: *Beyond DVFS: A First Look at Performance under a Hardware-Enforced Power Bound.* 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, Bd. 0, S. 947–953, 2012. doi: 10.1109/IPDPSW.2012.116.

[235] Samsung Electronics: *240pin Registered DIMM based on 2Gb D-die 1.35V: datasheet*, August 2011. http://www.samsung.com/global/business/semiconductor/file/2011/product/2011/9/6/476443ds_ddr3_2gb_d-die_based_1_35v_rdimm_rev12.pdf. Revision 1.2.

[236] Sanjay, H.A. und Sathish Vadhiyar: *Performance modeling of parallel applications for grid scheduling.* Journal of Parallel and Distributed Computing, Bd. 68, Nr. 8, S. 1135–1145, 2008. doi: 10.1016/j.jpdc.2008.02.006.

[237] ScaFaCoS. https://github.com/scafacos/scafacos. Quelltext-Repositorium des Projektes „ScaFaCoS". [24.06.2013].

[238] Schneider, Uwe (Herausgeber): *Taschenbuch der Informatik.* Fachbuchverlag Leipzig im Carl Hanser Verlag, München, 7. Auflage, 2012. isbn: 9783446426382.

[239] Shalf, John, Sudip Dosanjh und John Morrison: *Exascale Computing Technology Challenges.* In: Palma, José M. Laginha M., Michel Daydé, Osni Marques und João Correia Lopes (Herausgeber): *High Performance Computing for Computational Science – VECPAR 2010*, Band 6449 der Reihe *Lecture Notes in Computer Science*, S. 1–25. Springer Berlin Heidelberg, 2011. isbn: 978-3-642-19327-9. doi: 10.1007/978-3-642-19328-6_1.

[240] Shao, Yakun Sophia und David Brooks: *Energy characterization and instruction-level energy model of Intel's Xeon Phi processor.* In: *2013 IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, S. 389–394, 2013. doi: 10.1109/ISLPED.2013.6629328.

[241] Sharma, Sushant, Chung-Hsing Hsu und Wu chun Feng: *Making a case for a Green500 list.* In: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006. doi: 10.1109/IPDPS.2006.1639600.

[242] Šimunić, Tajana, Luca Benini und Giovanni De Micheli: *Cycle-accurate Simulation of Energy Consumption in Embedded Systems.* In: *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, S. 867–872, New York, NY, USA, 1999. ACM. isbn: 1-58113-109-7. doi: 10.1145/309847.310090.

[243] Song, Fengguang, Stanimire Tomov und Jack Dongarra: *Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems.* In: *Proceedings of th 26th ACM International Conference on Supercomputing*, ICS '12, S. 365–376, New York, NY, USA, 2012. ACM. doi: 10.1145/2304576.2304625.

[244] Song, Shuaiwen, Chunyi Su, Barry Rountree und Kirk W. Cameron: *A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures.* In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, S. 673–686, Washington, DC, USA, 2013. IEEE Computer Society. isbn: 978-0-7695-4971-2. doi: 10.1109/IPDPS.2013.73.

[245] Sprunt, Brinkley: *The Basics of Performance-Monitoring Hardware.* IEEE Micro, Bd. 22, Nr. 4, S. 64–71, Juli/August 2002. doi: 10.1109/MM.2002.1028477.

[246] Stiefel, Eduard: *Relaxationsmethoden bester Strategie zur Lösung linearer Gleichungssysteme.* Commentarii Mathematici Helvetici, Bd. 29, Nr. 1, S. 157–179, 1955. doi: 10.1007/BF02564277.

[247] Stiller, Andreas: *Spezialkommando: Intrinsics für präzise Zeitmessung und Cache-Flushs.* c't – Magazin für Computertechnik, , Nr. 17, S. 172–174, 2013.

[248] Subramaniam, Balaji und Wu-chun Feng: *Statistical Power and Performance Modeling for Optimizing the Energy Efficiency of Scientific Computing.* In: *Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, S. 139–146, Washington, DC, USA, 2010. IEEE Computer Society. isbn: 978-0-7695-4331-4. doi: 10.1109/GreenCom-CPSCom.2010.138.

[249] Subramaniam, Balaji und Wu chun Feng: *Understanding Power Measurement Implications in the Green500 List.* In: *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, S. 245–251, 2010. doi: 10.1109/GreenCom-CPSCom.2010.140.

[250] Suda, Reiji: *A Bayesian Method of Online Automatic Tuning.* In: Naono, Ken, Keita Teranishi, John Cavazos und Reiji Suda (Herausgeber): *Software Automatic Tuning*, S. 275–293. Springer, New York, 2010. isbn: 978-1-4419-6934-7. doi: 10.1007/978-1-4419-6935-4_16.

[251] Suda, Reiji, Luo Cheng und Takahiro Katagiri: *A Mathematical Method for Online Autotuning of Power and Energy Consumption with Corrected Temperature Effects.* Procedia Computer Science, Bd. 18, S. 1302–1311, 2013. doi: 10.1016/j.procs.2013.05.297. Proceedings of the International Conference on Computational Science (ICCS 2013).

[252] Sutter, Herb und James Larus: *Software and the Concurrency Revolution.* Queue, Bd. 3, Nr. 7, S. 54–62, September 2005. doi: 10.1145/1095408.1095421.

[253] Taylor, Valerie, Xingfu Wu und Rick Stevens: *Prophesy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications.* SIGMETRICS Performance Evaluation Review, Bd. 30, Nr. 4, S. 13–18, März 2003. doi: 10.1145/773056.773060.

[254] Tiwari, Ananta, Michael A. Laurenzano, Laura Carrington und Allan Snavely: *Auto-tuning for energy usage in scientific applications.* In: *Proceedings of the 2011 international conference on Parallel Processing - Volume 2*, Euro-Par'11, S. 178–187, Berlin, Heidelberg, 2012. Springer-Verlag. isbn: 978-3-642-29739-7. doi: 10.1007/978-3-642-29740-3_21.

[255] Tolentino, Matthew und Kirk W. Cameron: *The Optimist, the Pessimist, and the Global Race to Exascale in 20 Megawatts.* Computer, Bd. 45, Nr. 1, S. 95–97, 2012. doi: 10.1109/MC.2012.34.

[256] Tomov, Stanimire, Jack Dongarra und Marc Baboulin: *Towards dense linear algebra for hybrid GPU accelerated manycore systems.* Parallel Computing, Bd. 36, Nr. 5-6, S. 232–240, Juni 2010.

[257] *TOP500 Supercomputer Sites.* http://www.top500.org/. [Abruf: 04.06.2014].

[258] Torres, Gabriel: *Everything You Need to Know About the CPU C-States Power Saving Modes.* http://www.hardwaresecrets.com/article/611. [Abruf: 2013-09-04].

[259] Trefethen, Anne E. und Jeyarajan Thiyagalingam: *Energy-aware software: Challenges, opportunities and strategies.* Journal of Computational Science, , Nr. 0, 2013. doi: 10.1016/j.jocs.2013.01.005.

[260] Valentini, Giorgio Luigi, Walter Lassonde, Samee Ullah Khan, Nasro Min-Allah, Sajjad A. Madani, Juan Li, Limin Zhang, Lizhe Wang, Nasir Ghani, Joanna Kolodziej, Hongxiang Li, Albert Y. Zomaya, Cheng-Zhong Xu, Pavan Balaji, Abhinav Vishnu, Fredric Pinel, Johnatan E. Pecero, Dzmitry Kliazovich und Pascal Bouvry: *An overview of energy efficiency techniques in cluster computing systems.* Cluster Computing, Bd. 16, Nr. 1, S. 3–15, 2013. doi: 10.1007/s10586-011-0171-x.

[261] Verein Deutscher Ingenieure: *VDI 3633 Blatt 1: Simulation von Logistik-, Materialfluss- und Produktionssystemen*, März 2000. Gründruck.

[262] *Resolution 60/1: Ergebnisse des Weltgipfels 2005.* In: *Resolutionen und Beschlüsse der sechzigsten Tagung der Generalversammlung*, Band I – Resolutionen, S. 3–27. Vereinte Nationen, 2006.

[263] Verschoor, Mickeal und Andrei C. Jalba: *Analysis and Performance Estimation of the Conjugate Gradient Method on Multiple GPUs.* Parallel Comput., Bd. 38, Nr. 10-11, S. 552–575, Oktober 2012. doi: 10.1016/j.parco.2012.07.002.

[264] Vuduc, Richard, James W. Demmel und Katherine A. Yelick: *OSKI: A library of automatically tuned sparse matrix kernels.* Journal of Physics: Conference Series, Bd. 16, Nr. 1, S. 521, 2005.

[265] Vuduc, Richard W.: *Autotuning.* In: Padua, David (Herausgeber): *Encyclopedia of Parallel Computing*, S. 102–105. Springer, 2011. doi: 10.1007/978-0-387-09766-4_68.

[266] Vázquez, Francisco, José Jesús Fernández und Ester M. Garzón: *Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach.* Parallel Computing, Bd. 38, Nr. 8, S. 408 – 420, 2012. doi: http://dx.doi.org/10.1016/j.parco.2011.08.003.

[267] Wang, Wei, John Cavazos und Allan Porterfield: *Energy Auto-tuning using the Polyhedral Approach.* In: *International Workshop on Polyhedral Compilation Techniques*, 2014.

[268] Weaver, Vince, Matthew Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Daniel Terpstra und Shirley Moore: *Measuring energy and power with PAPI.* In: *International Workshop on Power-Aware Systems and Architectures (PASA 2012)*, Pittsburgh, PA (USA), September 2012.

[269] Weaver, Vincent M., Dan Terpstra, Heike McCraw, Matt Johnson, Kiran Kasichayanula, James Ralph, John Nelson, Phil Mucci, Tushar Mohan und Shirley Moore: *PAPI 5: Measuring power, energy, and the cloud.* 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), S. 124–125, 2013. doi: 10.1109/ISPASS.2013.6557155.

[270] Whaley, R. Clint, Antoine Petitet und Jack J. Dongarra: *Automated empirical optimizations of software and the ATLAS project.* Parallel Computing, Bd. 27, Nr. 1-2, S. 3–35, 2001. doi: 10.1016/S0167-8191(00)00087-9.

[271] Williams, Samuel, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands und Katherine Yelick: *The Potential of the Cell Processor for Scientific Computing.* In: *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, S. 9–20, New York, NY, USA, 2006. ACM. isbn: 1-59593-302-6. doi: 10.1145/1128022.1128027.

[272] Windeck, Christof: *Spar-o-matic: Stromsparfunktionen moderner x86-Prozessoren.* c't – Magazin für Computertechnik, , Nr. 15, S. 200–207, 2007.

[273] Wünsch, Marco, Ruth Offermann, Friedrich Seefeldt, Karsten Weinert, Inka Ziegenhagen, David Echternacht, Ulf Kasper, Julian Lichtinghagen und Albert Moser: *Positive Effekte von Energieeffizienz auf den deutschen Stromsektor.* Studie 032/01-S-2014/DE, Agora Energiewende, März 2014.

[274] Yang, Chunbai, Changjiang Jia, Wing-Kwong Chan und Yuen Tak Yu: *On Accuracy-Performance Tradeoff Frameworks for Energy Saving: Models and Review.* In: *19th Asia-Pacific Software Engineering Conference (APSEC)*, Band 2, S. 58–65, 2012. doi: 10.1109/APSEC.2012.117.

[275] Yang, Tian-Ruo und Hai-Xiang Lin: *Isoefficiency analysis of CGLS algorithm for parallel least squares problems.* In: Hertzberger, Bob und Peter Sloot (Herausgeber): *High-Performance Computing and Networking*, Band 1225 der Reihe *Lecture Notes in Computer Science*, S. 452–461. Springer, Berlin, Heidelberg, 1997. isbn: 978-3-540-62898-9. doi: 10.1007/BFb0031617.

[276] Yelick, Katherine: *Special Issue on Automatic Performance Tuning, Preface.* International Journal of High Performance Computing Applications, Bd. 18, Nr. 1, S. 19, Februar 2004. doi: 10.1177/1094342004042573.

[277] Yonetani, Yoshiteru: *A severe artifact in simulation of liquid water using a long cut-off length: Appearance of a strange layer structure.* Chemical Physics Letters, Bd. 406, Nr. 1–3, S. 49–53, 2005. doi: 10.1016/j.cplett.2005.02.073.

[278] Yoshizawa, Hiroki und Daisuke Takahashi: *Automatic Tuning of Sparse Matrix-Vector Multiplication for CRS Format on GPUs.* In: *IEEE 15th International Conference on Computational Science and Engineering (CSE), 2012*, S. 130–136, Dec 2012. doi: 10.1109/ICCSE.2012.28.

[279] Yotov, Kamen, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill und Peng Wu: *A comparison of empirical and model-driven optimization.* SIGPLAN Not., Bd. 38, Nr. 5, S. 63–76, Mai 2003. doi: 10.1145/780822.781140.

[280] Yotov, Kamen, Xiaoming Li, Gang Ren, Maria Garzaran, David Padua, Keshav Pingali und Paul Stodghill: *Is Search Really Necessary to Generate High-Performance BLAS?* Proceedings of the IEEE, Bd. 93, Nr. 2, S. 358–386, February 2005. doi: 10.1109/JPROC.2004.840444.

[281] Yotov, Kamen, Keshav Pingali und Paul Stodghill: *Think Globally, Search Locally.* In: *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, S. 141–150, New York, NY, USA, 2005. ACM. isbn: 1-59593-167-8. doi: 10.1145/1088149.1088168.

[282] Yuki, Tomofumi und Sanjay Rajopadhye: *Folklore Confirmed: Compiling for Speed = Compiling for Energy.* Technischer Bericht CS13-107, Colorado State University, Computer Science Department, Fort Collins, CO (USA), August 2013.
http://www.cs.colostate.edu/TechReports/Reports/2013/tr13-107.pdf.

[283] Zhang, Yan, Dharmesh Parikh, Karthik Sankaranarayanan, Kevin Skadron und Mircea Stan: *HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects.* Technischer Bericht, 2003.

[284] Zhuo, Jianli und Chaitali Chakrabarti: *Energy-efficient Dynamic Task Scheduling Algorithms for DVS Systems.* ACM Trans. Embed. Comput. Syst., Bd. 7, Nr. 2, S. 17:1–17:25, Januar 2008. doi: 10.1145/1331331.1331341.

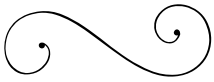
Danksagung

Ich danke

Frau Prof. Gudula Rünger für die Gelegenheit, in ihrer Arbeitsgruppe zu promovieren, für die hervorragende Betreuung meiner Arbeit, für ihre Anregungen bezüglich wissenschaftlich interessanter Fragestellungen und für ihre vielen nützlichen Hinweise zum wissenschaftlichen Schreiben,

Herrn Prof. Fred Hamker für die freundliche Übernahme des Zweitgutachtens,

meinen Arbeitskolleginnen und -kollegen an der Professur Praktische Informatik für das anregende Arbeitsumfeld und die stets angenehme Atmosphäre.



Stichwortverzeichnis

Thesen

I. Modellbasiertes Autotuning ermöglicht die effiziente Ausführung wissenschaftlicher Simulationen sowohl in Bezug auf ihre Ausführungszeit als auch ihren Energieverbrauch.

II. Wird beim Autotuning die meist langwierige Suche einer optimalen Parameterkonfiguration durch eine Vorhersage der Ausführungszeit anhand eines Modells ersetzt, wird Online-Autotuning und damit die Adaption an Eingabedaten ermöglicht.

III. Bei der Schnellen Multipolmethode wird die Genauigkeit der Kostenvorhersage durch Einsatz eines Verfahrens, das die Kosten von Codeabschnitten der Hauptschleife am konkreten Rechner misst, gegenüber einem Verfahren mit statischer Kostenfunktion deutlich verbessert.

IV. Mit dem vorgestellten Verfahren zur Kostenvorhersage für die Schnelle Multipolmethode, das die Vorhersage anhand der gemessenen maschinenspezifischen Kosten eines jeden Codeabschnitts der Hauptschleife und der Häufigkeit seiner Ausführung bei gegebenen Eingabedaten trifft und dabei auch die Kosten der Schleifen mit einbezieht, können die Gesamtkosten bei verschiedenen Baumtiefen genau abgeschätzt werden, sodass die Baumtiefe mit den niedrigsten Kosten für die tatsächliche Berechnung genutzt werden kann.

V. Zur Messung der Kosten von Codeabschnitten der Hauptschleife der Schnellen Multipolmethode ist für manche Codeabschnitte eine mehrfache synthetische Ausführung mit beliebigen Daten, für manche eine Ausführung mit konkreten Daten einer stark inhomogenen Teilchenverteilung zweckmäßig.

VI. Bei der vorliegenden Implementierung der Methode der finiten Elemente (FEM) kann durch Ersetzen der zur Reduktion des verteilt liegenden Ergebnisvektors ursprünglich verwendeten expliziten, grobgranularen Methode durch eine implizite, feingranulare Methode die parallele Ausführungszeit deutlich verkürzt werden.

VII. Die Berücksichtigung des NUMA-Speicherknotens der finiten Elemente bei ihrer Verarbeitung in der Methode der konjugierten Gradienten (CGM) bringt auf heutigen Systemen keine signifikante Verringerung der Ausführungszeit oder des Energieverbrauchs.

VIII. Die Ausführungszeit t_{cgm} und der Energieverbrauch E_{cgm} der CGM in der untersuchten FEM

werden durch die folgenden Formeln beschrieben:

$$
\begin{aligned}
t_{\mathrm{cgm}} &= n_{\mathrm{el,gpu}} \cdot t_{\mathrm{el,copy}} + l\left(\max\left(n_{\mathrm{el,cpu}} \cdot t_{\mathrm{el,cpu}}, n_{\mathrm{el,gpu}} \cdot t_{\mathrm{el,gpu}}\right)\right) \quad , \\
E_{\mathrm{cgm}} &= l\left(n_{\mathrm{el,cpu}} \cdot E^{*}_{\mathrm{el,cpu}} + n_{\mathrm{el,gpu}} \cdot E_{\mathrm{el,gpu}}\right) \\
&\quad + n_{\mathrm{el,gpu}} \cdot E_{\mathrm{el,copy}} \\
&\quad + P^{*}_{\mathrm{idle,cpu}} \cdot t_{\mathrm{idle,cpu}} + P_{\mathrm{idle,gpu}} \cdot t_{\mathrm{idle,gpu}} \quad .
\end{aligned}
$$

IX. Eine hinsichtlich des Energieverbrauchs optimale Verteilung der finiten Elemente der FEM auf die Verarbeitungseinheiten CPU und GPU ergibt sich für die CGM nach folgender Fallunterscheidung:

- Wenn $E^{*}_{\mathrm{el,cpu}} < E^{*}_{\mathrm{el,gpu}}$,
  dann verarbeite alle Elemente auf der CPU.
- Wenn $E^{*}_{\mathrm{el,cpu}} > E^{*}_{\mathrm{el,gpu}} + P^{*}_{\mathrm{idle,cpu}} \cdot t_{\mathrm{el,gpu}}$,
  dann verarbeite alle Elemente auf der GPU.
- Sonst
  verteile die Rechenlast zwischen CPU und GPU.

X. Bei der gemeinsamen Verarbeitung der Rechenlast der CGM auf CPU und GPU ist die Verteilung, durch die sich minimale Leerlaufzeiten der Ausführungseinheiten ergeben, die Verteilung mit der kürzesten Ausführungszeit und gleichzeitig dem niedrigsten Energieverbrauch.

XI. Eine ausführungszeitminimierende Verteilung der Rechenlast der CGM auf CPU und GPU ergibt sich, wenn folgender Anteil r der Elemente auf der GPU verarbeitet wird:

$$
r = \left(1 + \frac{t_{\mathrm{el,gpu}}}{t_{\mathrm{el,cpu}}}\right)^{-1} \quad .
$$

XII. Durch Messung der Ausführungszeiten der CGM auf der CPU und der GPU in jedem Verfeinerungsschritt kann mit dem vorgestellten Verfahren die Verteilung der Rechenlast im nächsten Verfeinerungsschritt dynamisch auf sich verändernde Verhältnisse, z. B. aufgrund der wachsenden Anzahl finiter Elemente, angepasst werden, um so die Leerlaufzeiten von CPU und GPU stets möglichst gering zu halten.

XIII. Die von der GPU-Hardware im Intervall von 20 ms bereitgestellten Energiemesswerte lassen sich mit der entwickelten Methode durch mehrfache Ausführung der zu untersuchenden GPU-Funktion derart kombinieren, dass ein zeitlich hochaufgelöstes Energieprofil entsteht, sodass auch Funktionen mit kürzerer Ausführungszeit als 20 ms adäquat untersucht werden können.

XIV. Die vorgestellte Methode zum modellbasierten Autotuning lässt sich auf andere heterogene Rechenarchitekturen übertragen.