Modellierung von paralleler Software für homogene und heterogene Multiprozessor-Systeme

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

von
Dipl.-Ing. Oliver Jakob Arndt

geboren am 20. September 1988 in Dortmund, Deutschland

2020

1. Referent:	Prof. Dr.-Ing. Holger Blume
2. Referent:	Prof. rer. nat. Rainer Leupers

Tag der Promotion: 20. Mai 2020

Vorwort

Die vorliegende Dissertation entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Mikroelektronische Systeme (IMS) der Gottfried Wilhelm Leibniz Universität Hannover.

An erster Stelle möchte ich mich bei Herrn Prof. Dr.-Ing. Holger Blume für die fachliche und wissenschaftliche Unterstützung in einer Vielzahl von kritischen, fordernden und fördernden Diskussionen während meiner Promotionszeit bedanken, mit der schließlich diese Arbeit entstehen konnte. Unter seiner Leitung konnte ich während der spannenden Arbeit in Lehre und Forschung sowohl fachliche Fähigkeiten, als auch wertvolle Soft Skills entwickeln. Ebenfalls danke ich Herrn Prof. Dr. rer. nat. Rainer Leupers für die Übernahme des Koreferats, sowie für die gute Zusammenarbeit im PARIS-Projekt. Bei Herrn Prof. Dr. Schneider bedanke ich mich für die Übernahme des Vorsitz der Prüfungskommission.

Eine wichtige Rolle spielte speziell in der Anfangszeit meiner wissenschaftlichen Karriere Herr Dr.-Ing. Daniel Becker, von dem ich ebenfalls unzähliges, kritisches und unschätzbar wertvolles Feedback erhielt. Ebenso wichtig waren die fachlichen Diskussionen mit den Kollegen, aber auch allen Studenten, die mit mir gemeinsam an dieser Thematik arbeiteten. Danke!

Abschließend danke ich meinen Lieben Valentina, Lea und Elias, sowie Petra und Richie die mit mir gemeinsam diese Zeit mit Höhen und Tiefen gemeistert haben und auch besonders meinen Freunden Eric, Jan, Tile, Luis und Jens für die gemeinsame Zeit und all die motivierenden und aufbauenden Worte.

Hannover, Juli 2020

Jakob Arndt

Kurzfassung

Multicore-Prozessoren stellen leistungsstarke Architekturen für die Signal- und Videoverarbeitung dar, die zunehmend auch in Echtzeitanwendungen eingesetzt werden. Da Multicore-Prozessoren in Hochsprachen programmiert werden, bieten sie eine hohe Flexibilität und Portabilität der Software. Eingebettete Multicore-Prozessoren werden beispielsweise im Bereich der kamerabasierten Fahrerassistenzsysteme zur Berechnung anspruchsvoller Algorithmen mit strengen Echtzeitkriterien verwendet. Die Fusion und Auswertung der gesammelten Daten aus einer Vielzahl von Sensoren besitzt eine hohe Rechenkomplexität und einen hohen Speicherbedarf. Gleichzeitig haben derartige Algorithmen kurze Entwicklungszyklen, was den Einsatz von flexibler Multicore-Software attraktiv macht. Eine optimale Auslastung aller Ressourcen und eine gute Portabilität der Software zwischen Plattformen benötigt allerdings eine skalierbare parallele Implementierung. Für die Parallelisierung steht eine Vielzahl von Strategien zur Auswahl, die auf unterschiedlichen Plattformen jeweils andere Laufzeiteigenschaften und Limitierungen, wie z.B. Synchronisationen oder Speicherbandbreite, aufweisen. Moderne Plattformen kombinieren meist heterogene Prozessoren mit applikationsspezifischen Beschleunigern, was eine parallele Programmierung zusätzlich erschwert.

Diese Arbeit adressiert die Programmierbarkeit von parallelen Plattformen mit dem Ziel Entwickler bei der Parallelisierung zu unterstützen. Dazu wird die Abstraktionsschicht *Modular Parallelization Abstraction Layer* (MPAL) vorgestellt, die eine neue Schnittstelle zur parallelen Programmierung von Multicore- und Manycore-Prozessoren, sowie Beschleunigern, wie Grafikprozessoren oder Field Programmable Gate Arrays, bereitstellt. MPAL ermöglicht eine einfache Konfiguration der verwendeten Frameworks und der Parallelität, sowie die Kontrolle des heterogenen Architektur-Mappings. Des Weiteren bietet MPAL Profiling-Methoden, die die Extraktion der Skalierungscharakteristiken erlauben, die wiederum bei der Identifikation von Engpässen helfen. Weiterhin werden in dieser Arbeit Einflussfaktoren auf die Skalierbarkeit untersucht und modelliert, damit diese durch signifikante Modellparameter dargestellt und ausgewertet werden können. Unter Verwendung der Modellparameter als Deskriptoren stellt diese Arbeit eine neue Methode der Performance-Prädiktion vor. Damit können das erwartete Skalierungsverhalten, sowie potentielle Engpässe nach einer Portierung der Software, abgeschätzt werden. Die vorgestellten Methoden werden mit Referenz-Methoden zur parallelen Programmierung und Performance-Prädiktion verglichen und im Anschluss mithilfe von zwei Fahrerassistenzalgorithmen (Stereo-Vision und Fußgängererkennung) evaluiert.

Schlagwörter: Parallelisierung, Skalierbarkeit, MPSoC, Middleware Abstraktionsschicht, Performance-Prädiktion, Architektur-Mapping

Abstract

Multicore processors represent powerful architectures for signal- and video-processing, even applicable for real-time applications. Because these chips are programmable in high-level programming-languages, they offer good flexibility and software portability. For instance in the field of camera-based advanced driver assistance systems, which appear as demanding applications with strict real-time constraints, embedded multicores serve as target platforms. Such algorithms feature not only complex computations and high memory consumption due to the fusion of multiple sensor-data, but they also have short development cycles, which makes the flexibility of multicore-software even more desirable. However, gathering the theoretical peak performance of multicore processors requires parallel programming and a scalable work distribution in order to obtain software that is portable across platforms. Furthermore, parallel programming opens up a huge design space of parallelization strategies each with potential bottlenecks, like synchronization-count or memory bandwidth, which vary between platforms. Modern processors are often equipped with application specific accelerator architectures, resulting in multiple heterogeneous computational units eventually each requiring individual programming techniques.

This work addresses the programmability of parallel platforms with the aim of supporting developers during the parallelization process. In order to ease the parallel programming, the middleware *modular parallelization abstraction layer* (MPAL) is introduced, which offers a new interface for programming multi-/manycores and accelerators (e.g., graphics processors, or field programmable gate arrays), while the actually used frameworks can be configured dynamically. MPAL allows the configuration of the parallelism and to control the architecture mapping including heterogeneous scheduling and data transfers. Furthermore, MPAL offers in-line profiling methodologies, which enable the extraction of the parallel runtime behavior and scalability characteristics, which help to analyze the parallelization success and identify bottlenecks. Most relevant, influencing scalability characteristics are modeled to represent trends in few significant and comparable model-parameters. Using these model-parameters as descriptive features, this work contributes a statistical performance prediction method, which allows for an estimation of the scalability trend and potential bottlenecks after porting a parallel software between platforms. All introduced methods are compared with state-of-the-art programming techniques and prediction methods, and will be evaluated using two driver-assistance algorithms (stereo-vision and pedestrian detection).

Keywords: Parallelization, Scalability, MPSoC, Middleware Abstraction Layer, Performance Prediction, Architecture Mapping

Inhaltsverzeichnis

Abkürzungsverzeichnis

ADAS	*Advanced Driver-Assistance Systems*
ALU	*Arithmetic Logic Unit*
API	*Application Programming Interface*
APU	*Accelerated Processing Unit*
ASIC	*Application Specific Integrated Circuit*
AVX	*Advanced Vector Extensions*
AXI	*Advanced Extensible Interface*
CNN	*Convolutional Neural Network*
CPI	*Cycles per Instruction*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
CV	*Computer Vision*
DCM	*Data Cache Miss*
DMA	*Direct Memory Access*
DSE	*Design-Space Exploration*
DSP	*Digital Signal Processor*
FPGA	*Field Programmable Gate Array*
GEM5	*General Execution-driven Multiprocessor Simulator + M5*

GCC	*GNU Compiler Collection*
GPU	*Graphics Processing Unit*
GPP	*General-Purpose Processor*
HLS	*High-Level Synthesis*
HOG	*Histograms of Oriented Gradients*
HPC	*High-Performance Computing*
HSA	*Heterogeneous Systems Architecture*
HSAIL	*HSA Intermediate Language*
HT	*Hyperthreads*
ICC	*Intel C/C++ Compiler*
IPC	*Inter-Process Communication*
ISP	*Image Signal Processor*
JIT	*Just-In-Time*
LIDAR	*Light Detection and Ranging*
LLC	*Last-Level Cache*
LLVM	*Low-Level Virtual Machine*
MCA	*Machine Code Analyzer*
MMU	*Memory Management Unit*
MMX	*Multi-Media Extension*
MPAL	*Modular Parallelization Abstraction Layer*
MPSoC	*Multi Processor System on Chip*
Mutex	*Mutual-Exclusive Lock*
NEON MPE	*NEON Media Processing Engine*
NoC	*Network on Chip*
NUMA	*Non-Uniform Memory Access*
OpenCL	*Open Computing Language*

OpenMP	*Open Multi-Processing*
OVP	*Open Virtual Platforms*
PAPI	*Performance Application Programming Interface*
PC	*Personal Computer*
PCA	*Principle Component Analysis*
PThread	*POSIX Thread*
QEMU	*Quick Emulator*
Radar	*Radio Detection and Ranging*
SIMD	*Single Instruction Multiple Data*
SGM	*Semi-Global Matching*
SoC	*System on Chip*
SPIR-V	*Standard for Portable Intermediate Representation - V*
SSE	*Streaming SIMD Extension*
STL	*Standard Template Library*
SVM	*Support Vector Machine*
TBB	*Intel Threading Building Blocks*
TLB	*Translation Lookaside Buffer*
TLM	*Transaction Level Modeling*
VHDL	*Very High Speed Integrated Circuit Hardware Description Language*
VLIW	*Very Long Instruction Word*
VSP	*Virtual System Platform*

Formelzeichenverzeichnis

Skalierungsverlauf:

$\boldsymbol{n}$	Anzahl der Threads
$\boldsymbol{t(n)}$	Gesamtausführungszeit bei n Threads
$\boldsymbol{t_i(n)}$	Ausführungszeit des Tasks i bei n Threads
$\boldsymbol{t_0(1)}$	Sequentielle Ausführungszeit (Annahme: Bei sequentieller Ausführung existiert nur ein Task.)
$\boldsymbol{t_{seq}}$	Sequentiell verbleibender nicht-parallelisierbarer Anteil

Laufzeitparameter:

$\boldsymbol{R(n)}$	Redundanz: Prozentualer Anstieg der Summe aller parallelen Taskausführungszeiten im Vergleich zu $t_0(1)$
$\boldsymbol{U(n)}$	Utilization: Prozentualer Anteil der gesamt verfügbaren CPU-Zeit, die für effektive Task-Berechnung verwendet werden kann
$\boldsymbol{l(n)}$	Synchronisation: Prozentualer Anteil der CPU-Zeit, die in Mutex-Locks zur Synchronisation der Threads verwendet wird
$\boldsymbol{w(n)}$	Arbeitsungleichverteilung (engl. *work imbalance*): Prozentuale ungenutzte CPU-Zeit (Idle)
$\boldsymbol{c(n)}$	Task-Erzeugung (engl. *creation*): Prozentuale CPU-Zeit, die für die Erzeugung für Tasks benötigt wird

$\boldsymbol{d(n)}$	Task-Verteilung (engl. *distribution*): Prozentuale CPU-Zeit, die für die Verteilung der Tasks verwendet wird
$\boldsymbol{s(n)}$	Task-Wechsel (engl. *switch*): Prozentuale CPU-Zeit, die für den Wechsel zwischen Tasks verwendet wird
$\boldsymbol{j(n)}$	Task-Synchronisation (engl. *join*): Prozentuale CPU-Zeit, die für die Terminierung von Task-Teams verwendet wird

Prädiktion:

$\boldsymbol{\vec{s}_p}$	Deskriptiver Parameter-Vektor des Skalierungsverlaufs des Parameters $p \in \{R, l, w, c, d, s, j\}$
$\boldsymbol{\vec{sc}}$	Deskriptiver Merkmals-Skalierungsvektor, der einen Skalierungs-Verlauf für $t(n)$ vollständig beschreibt
$\boldsymbol{sc_m}$	Einzelnes Element des Merkmals-Vektors $\vec{sc}$ der Dimension m
$\boldsymbol{\mathcal{D}_a}$	Distanz zwischen zwei algorithmischen Implementierungen auf einer einzelnen Plattform, oder im Durchschnitt
$\boldsymbol{\mathcal{D}_p}$	Distanz zwischen zwei Plattformen anhand von einer einzigen algorithmischen Implementierung oder im Durchschnitt
$\boldsymbol{f(C)}$	Transformationsfaktor der einzelnen Elemente des Skalierungsvektors zur Migrationsprädiktion zur Zielplattform
$\boldsymbol{w}$	Gewichtung der Prädiktionskandidaten für die interpolierende Transformation von $\vec{sc}$
$\boldsymbol{r(w)}$	Zusätzliche Normalisierungsgewichtung, die den Einfluss weit distanzierter Kandidaten (geringere Gewichte w) reduziert

1 Einleitung

Für die Implementierung eines Algorithmus stehen dem Applikationsentwickler eine Vielzahl von potentiellen Zielplattformen zur Verfügung. Dabei kann die Wahl der Plattform für den Erfolg oder Misserfolg der Implementierung entscheidend sein. Es kommen sowohl dedizierte Chips, applikationsspezialisierte Prozessoren, als auch vollkommen frei programmierbare Prozessoren in Frage. Während spezialisierte Hardware-Einheiten gesteigerte Rechenleistung-zu-Verlustleistungscharakteristiken aufweisen, ist ihr Einsatzgebiet auf einen Algorithmus oder eine Algorithmenklasse beschränkt. Dagegen profitieren frei programmierbare Prozessoren von höchster Flexibilität, weisen jedoch eine geringere Rechenleistung und eine höhere Verlustleistungsaufnahme auf [1].

Eine geeignete Wahl zu treffen kostet Entwickler in vielen Fällen großen Aufwand, um eine umfangreiche Entwurfsraumexploration (engl. *Design-Space Exploration* (DSE)) durchzuführen. Die Aufwendung der hohen Entwicklungskosten für dedizierte Hardware lohnt sich beispielsweise nur bei einer sehr hohen Stückzahl. Der Kauf einer fertigen Prozessorkomponente kann dagegen einen höheren Kostenaufwand pro eingesetztem Hardware-Modul mit sich bringen. Durch die gesteigerte Rechenleistung von programmierbaren Prozessoren (engl. *General-Purpose Processor* (GPP)), wie eingebettete Multicore-Prozessoren in Smartphones, und durch ihre sehr hohen Verkaufszahlen können diese Chips als Standardkomponenten günstige Alternativen darstellen. Durch ihre hohe Rechenleistung, ihre Spezialisierung auf 3D-Bildverarbeitung und ihre gleichzeitig geringe Stromaufnahme, bieten sich diese Komponenten beispielsweise sehr gut für den Einsatz in komplexen und rechenintensiven kamerabasierten Fahrerassistenzsystemen (engl. *Advanced Driver-Assistance Systems* (ADAS)) an.

1.1 Parallele Prozessoren

Während sich die Rechenleistung (*Performance*) von Singlecore-Prozessoren meist nur noch durch komplexe spekulative Techniken kleinschrittig verbessern lässt, kann die theoretische Performance von Multicore-Prozessoren durch eine Verdopplung der Kerne um 100 % gesteigert werden. Parallele Prozessoren sind mittlerweile weit verbreitet und werden in Anwendungen der Hochleistungsrechner (engl. *High-Performance Computing* (HPC)), im *Personal Computer* (PC) bis zu mobilen und eingebetteten (engl. *embedded*) Systemen eingesetzt. Eine exemplarische parallele Ausführung ist in Abbildung 1.1 dargestellt. Die Performance von Multicore-Prozessoren kann jedoch abhängig von der Architektur von vielen Faktoren beeinflusst sein, wie der Speicheranbindung, der Cache-Hierarchie oder der Kommunikationstopologie zwischen den Prozessorkernen (engl. *Core*). Dies gilt sowohl für homogene, als auch heterogene Prozessor-Cluster und kann je nach Parallelisierungs-Strategie zu Engpässen führen.

Da bei der Programmierung von parallelen Prozessoren einige potentielle Parallelisierungsfehler auftreten können, wie zum Beispiel *False-Sharing* (ungünstiges Zugriffsmuster paralleler Cores auf dieselbe Cache-Line), *Race-Conditions* (nicht synchronisierte parallele Operationen, die je nach zufälliger Abarbeitungsreihenfolge das Programm/das Ergebnis beeinflussen können) oder *Deadlocks* (blockierte parallele Prozesse, die jeweils aufeinander warten), ist die Parallelisierung einer zunächst sequentiellen Implementierung oft ein aufwändiger Prozess. Auch kann durch ungünstige Parallelisierungs-Strategien zusätzlicher Synchronisationsaufwand oder Wartezeiten der Prozessoren entstehen und dadurch das Laufzeitverhalten negativ beeinflusst werden. Abbildung 1.2 zeigt das Skalierungsverhalten von zwei unterschiedlichen Implementierungen desselben Algorithmus *Semi-Global Matching* (SGM) [2] auf einem Intel Xeon Phi Manycore-Prozessor mit 61 Prozessorkernen mit je bis

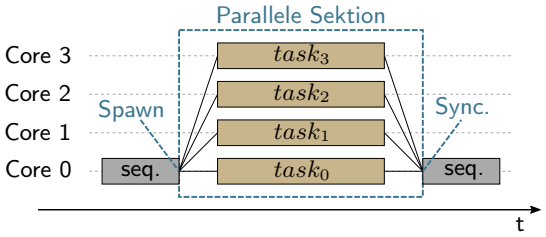
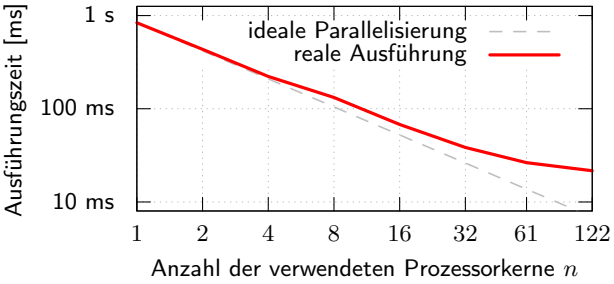
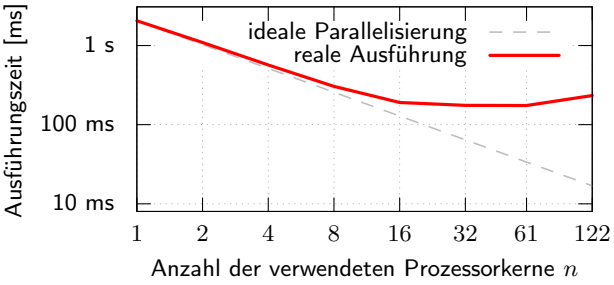


Abbildung 1.1: Schematische Darstellung einer parallelen Ausführung. Von einem sequentiellen Hauptprozess wird durch einen *Spawn*-Aufruf die parallele Sektion betreten und, nach Beendigung aller parallelen Tasks, durch Synchronisation wieder zum sequentiellen Prozess zurückgekehrt.



(a) Skalierungsverlauf der Paralellisierung *Tasking* (Hyperthreading ab $n > 61$).



(b) Skalierungsverlauf der Paralellisierung *Streaming* (Hyperthreading ab $n > 61$).

Abbildung 1.2: Vergleich des Skalierungsverhaltens zweier unterschiedlicher Implementierungen des SGM-Algorithmus mit verschiedenen Parallelisierungs-Strategien. Oben (*Tasking*): Hoher Speicherbedarf, geringe Kommunikation zwischen den Prozessorkernen. Unten (*Streaming*): Hohe Kommunikationsraten, geringer Speicherbedarf durch verbesserte Speicherlokalität.

zu vier Hyperthreads (siehe Kapitel 2.5.2 für *Hyperthreads*). Dargestellt ist der Verlauf der Ausführungszeit über der Anzahl der verwendeten Prozessorkerne. Neben den unterschiedlichen sequentiellen Ausführungszeiten und den variierenden Verläufen der beiden Kurven ist eine deutliche Abweichung von der idealen Skalierung festzustellen, die auf eine Vielzahl von Effekten zurückgeführt werden kann. Die Entwicklung einer Software benötigt daher nicht nur fundierte Expertise über den Algorithmus und Grundkenntnisse der angewendeten Programmiersprache. Sondern, für eine effiziente Umsetzung des gegebenen Algorithmus in parallelisierte Software, auch einen tiefen Einblick in Strategien und Werkzeuge (engl. *Tools*) der parallelen Programmierung. Hierbei liefern Tools eine wesentlichen Hilfestellung; die Expertise des Entwicklers spielt jedoch weiter die zentrale Rolle für den Implementierungserfolg.

Algorithmen bestehen in der Regel aus verschiedenen Modulen, die jeweils nicht auf allen Recheneinheiten gleich gute Ausführungscharakteristiken aufweisen. Daraus resultiert, dass die Ausführung des gesamten Algorithmus auf nur einem Prozessortyp nicht die maximal mögliche Performance erreichen kann. Darum integrieren moderne Computersysteme meist mehrere heterogene Recheneinheiten gleichzeitig, wie *Central Processing Unit* (CPU), *Graphics Processing Unit* (GPU) oder *Digital Signal Processor* (DSP). Durch eine geschickte Zuweisung der algorithmischen Blöcke auf die zur Verfügung stehenden Recheneinheiten (engl. *Architektur-Mapping*), kann so meist eine deutlich effizientere Ausführung im Bezug auf Rechenzeit und Verlustleistung erzielt werden. Um die Verlustleistungscharakteristik weiter zu verbessern, können z.B. nicht verwendete Einheiten temporär deaktiviert werden (engl. *Power-Gating*). Durch die Integration von heterogenen Recheneinheiten (Beschleunigern) können selbst eingebettete Chips zu sehr leistungsstarken Rechensystemen werden. So können beispielsweise moderne Smartphones durch einen stromsparenden *Multi Processor System on Chip* (MPSoC) selbst komplexeste 3D-Grafikverarbeitung in Echtzeit berechnen. Die Komplexität des Architektur-Mappings potenziert sich jedoch mit der Anzahl der verfügbaren heterogenen Recheneinheiten. Algorithmische Blöcke müssen auf Recheneinheiten aufgeteilt werden, wobei diverse Parameter, wie z.B. Parallelisierungs-Strategie, Kommunikationszeiten zwischen den Recheneinheiten, und die individuelle Performance der algorithmischen Blöcke auf den heterogenen Recheneinheiten, berücksichtigt werden müssen.

1.2 Software-Portabilität

Damit eine parallele Implementierung portabel ist, muss sie nicht nur funktional korrekt auf unterschiedlichen Plattformen ausgeführt werden können, sondern dynamisch die Arbeitslast verteilen und die vorhandenen parallelen und evtl. heterogenen Recheneinheiten möglichst effizient ausnutzen (engl. *Performance-portabel*). Diese Eigenschaft verlangt allerdings auch, dass die Software über ein flexibles Architektur-Mapping und skalierbare Partitionierungen der Teilarbeitspakete (engl. *Tasks*) verfügt. Das Anlegen einer großen Menge kleiner Tasks kann zwar eine dynamische Arbeitsverteilung vereinfachen, erzeugt aber auch einen großen Mehraufwand (engl. *Overhead*) zum Erzeugen und Verwalten der Tasks und erschwert die Kontrolle der Datenlokalität. Daher sind oft skalierbare Task-Größen zu bevorzugen.

Bei der Verwendung eines GPPs ist der Software-Code meist in einer Hochsprache, wie C++, geschrieben und mithilfe eines Parallelisierungs-Frameworks für die Verwendung eines gemeinsamen Speicherbereichs (engl. *Shared-Me-*

mory) parallelisiert, wie *Intel Threading Building Blocks* (TBB) oder *Open Multi-Processing* (OpenMP). Dadurch lässt sich der Software-Code einfach zwischen Systemen portieren, skaliert aber nicht zwangsläufig über alle verfügbaren Recheneinheiten. Nachrichtenbasierte Ansätze (engl. *Message Passing*), die nicht nur Prozesse eines Systems synchronisieren, sondern auch mehrere Systeme z.B. über ein Netzwerk verbinden können, werden in dieser Arbeit nicht betrachtet. Schon die Verwendung von heterogenen CPU-Gruppen (engl. *Cluster*) macht die Verteilung der Tasks komplexer, ermöglicht aber zumindest die Verwendung derselben Programmiersprache. Sollen allerdings auch Beschleuniger verwendet werden, wie z.B. GPU, DSP, oder *Field Programmable Gate Array* (FPGA) müssen in den meisten Fällen spezielle Sprachen, wie beispielsweise *Open Computing Language* (OpenCL) oder *Compute Unified Device Architecture* (CUDA) oder sogar synthetisierbare Beschleunigerprogramme (engl. *Compute-Kernel*) für FPGAs zur Verfügung stehen. Bei der Verwendung von Beschleunigern müssen für die Synchronisation und das Übertragen von Daten (z.B. über ein *Direct Memory Access* (DMA)-Modul) meist explizit Aufrufe einer Programmierschnittstelle (engl. *Application Programming Interface* (API)) in der Applikation integriert werden. Dies bedeutet sowohl einen Mehraufwand der Programmierung, als auch einen erheblichen Verwaltungs-Overhead für das Programm. Somit kann die Portabilität eines Programms selbst durch Inkompatibilitäten der APIs beeinträchtigt werden.

Dennoch kann die Wahl zwischen einem Leistungsstarken Singlecore-Prozessor und einer Vielzahl von einfachen Prozessoren in einem Multicore die resultierende Performance einer Applikation stark beeinflussen. Da die Anforderungen einer Software an die Hardware-Ressourcen stark zwischen Implementierungsstrategien variieren können, kann sich in bestimmten Anwendungen aber auch ein leistungsstarker Singlecore besser eignen, als ein Multicore. Zum Beispiel erlaubt eine lineare Berechnung auf untereinander unabhängigen Daten (z.B. Pixel) eine hohe Auslastung von massiv parallelen Prozessoren. Algorithmen, die aus strikt sequentiellen Arbeitsabläufen bestehen, profitieren dagegen eher von einem leistungsstarken Singlecore-Prozessor. Spekulative Techniken, wie *Branch-Prediction* oder *Prefetching*, sowie Parallelität auf Instruktionsebene, wie Superskalare Pipelines oder *Very Long Instruction Word* (VLIW) Architekturen, verbessern dabei die Performance von Singlecores. Ebenfalls können hardwarespezifische Engpässe die Performance der Applikation limitieren, wenn eine Implementierung von einer Hardware-Architektur zu einer anderen portiert wird (Migration). Beispielsweise kann auf einer CPU, die Vektorinstruktionen (*Single Instruction Multiple Data* (SIMD)) unterstützt, eine dichte Anordnung von Integer-Daten in Vektorregister eine Applikation stark beschleunigen. Wird eine Implementierung, die dicht gepackte Vektorregister verwendet, beispielsweise auf eine GPU, oder einen Intel

Xeon Phi [3] portiert, kann es zu einem Performance-Verlust kommen, da diese Plattformen hauptsächlich für größere Gleitkommazahlen optimiert sind, wodurch der Compiler Konvertierungsoperationen (*Unpack-Cast*) einfügen muss. Implementierungen für FPGAs können dagegen von Instruktionen profitieren, die auf Fixpunktzahlen basieren.

Als exemplarische Klasse von rechenintensiven und komplexen Algorithmen haben kamerabasierte Fahrerassistenzsysteme sowohl hohe Leistungsanforderungen an die Recheneinheit, als auch einen hohen Speicherbedarf. Moderne Fahrerassistenzalgorithmen reduzieren deutlich die Zahl der jährlich registrierten dramatischen Verkehrsunfälle [4]. Sie unterstützen nicht nur den Fahrer in Gefahrensituationen, sondern liefern ihm zusätzlich wichtige Informationen oder übernehmen vollständige Aufgaben der Navigation und Fahrzeugführung. Durch die Kombination von mehreren unterschiedlichen Sensorsignalen (z.B. *Light Detection and Ranging* (LIDAR), *Radio Detection and Ranging* (Radar), Ultraschall und Stereokamera – wie in Abbildung 1.3 dargestellt) wird eine umfassende Repräsentation der aktuellen Fahrsituation möglich. Dadurch werden selbst vollständig autonome Fahrzeuge ermöglicht, wobei eine situationsabhängige (beispielsweise wetterabhängige) Kombination aus Algorithmen Fahrentscheidungen berechnet (z.B. Warnsignale, Gefahrenintervention, oder sogar autonome Trajektorienplanung) [5]. Fahrerassistenzsysteme, die eine Vielzahl von unterschiedlichen Sensorsignalen fusionieren, bestehen daher aus komplexen und rechenintensiven Algorithmen zur Verarbeitung großer Datenmengen unter Echtzeitbedingungen. Im Bereich des autonomen Fahrens müssen sicherheitskritische Systeme strikte Echtzeitanforderungen erfüllen. Außerdem ist die zur Verfügung stehende elektrische Leistung oft stark begrenzt. Um diese Anforderungen zu erfüllen, werden energieeffiziente Zielplattformen benötigt, die selbst komplexe Applikationen zur 3D-Szenenrekonstruktion realisieren können. In diesem Bereich findet, auch angetrieben durch die Entwicklung der Mobilprozessoren, eine rasante Weiterentwicklung der potentiellen

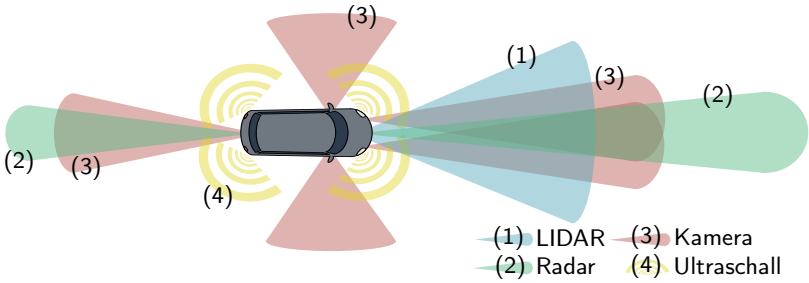


Abbildung 1.3: Darstellung der Abdeckung einiger Sensoren eines modernen Fahrzeugs: LIDAR, Radar, Ultraschall und (Stereo-)Kameras.

Ziel-Plattformen statt, die sich zunehmend auf heterogene Multicore-Systeme (MPSoCs) konzentriert. Aufgrund der hohen Variation der integrierten Systeme und zur Verbesserung der Markteinführungszeit neuer Applikationen wird für die Programmierung von MPSoCs flexible, wartbare und portable Software benötigt.

Durch die vielzähligen Parameter und wechselwirkenden Einflüsse von Hardware und Software, die die resultierende Performance einer Applikation bestimmen, kann der Entwurfsraum einer Software unübersichtlich erscheinen. Um ungewollten Engpässen (engl. *Bottlenecks*) rechtzeitig entgegenwirken zu können und dadurch zusätzliche Entwicklungskosten eines Produktes zu vermeiden, ist die frühzeitige Vorabschätzung der resultierenden Performance für Entwickler ein notwendiges Werkzeug. Eine solche Performance-Prädiktion erfordert je nach Methode eine mehr oder weniger aufwändige Modellierung der Hardware bzw. der Software. Bestehende Verfahren zur Prädiktion unterscheiden sich zum Teil stark sowohl im Aufwand für die Erzeugung eines Modells, in der Zeit für die eigentliche Simulation des Ausführungsverhaltens, als auch in der Präzision der Ergebnisse. So benötigt beispielsweise eine detaillierte Simulation eine aufwändige Modellierung mit anschließender, zeitintensiver Simulationsphase. Aufgrund der Systemkomplexität und der mannigfaltigen Einflussfaktoren kann dagegen durch starke Abstraktion der Systeme beispielsweise in approximativen analytischen Modellen ein großer Fehler entstehen.

1.3 Zielsetzung der Arbeit

In dieser Arbeit wird die Entwicklung der Abstraktionsschicht *Modular Parallelization Abstraction Layer* (MPAL) vorgestellt. MPAL ermöglicht eine Framework-unabhängige Implementierung portabler und skalierbarer paralleler Software für homogene und heterogene Multicore-Systeme. Mit MPAL parallelisierte Implementierungen lassen sich durch flexible Konfiguration der Parallelität und des Architektur-Mappings einfach zwischen unterschiedlichsten Plattformen (wie z.B. heterogene CPU-Cluster, GPUs, oder FPGAs) portieren. In diesem Rahmen werden Methoden zur automatisierten Extraktion von Laufzeitparametern (*Profiling*) und anschließender problemfokussierter Analyse und Charakterisierung des parallelen Laufzeitverhaltens, sowie eine neue Methode zur Performance-Prädiktion paralleler Implementierungen vorgestellt. Diese Methoden ermöglichen auch ungeübten Entwicklern die Implementierung von effizienten parallelen Applikationen für Server-, Desktop- und eingebettete Multicore-Prozessoren, die Programmierung von heterogenen Plattformen, sowie die Durchführung von Bottleneck-Analysen und den Vergleich unter-

schiedlicher Parallelisierungs-Strategien in frühen Entwicklungsphasen. Diese Arbeit umfasst die folgenden Abschnitte:

- **Abstraktionsschicht MPAL zur portablen Parallelisierung:** Es werden Funktion, Aufbau und API der Abstraktionsschicht MPAL für die Performance-portable Parallelisierung vorgestellt.
- **Charakterisierung des Laufzeitverhaltens**: Basierend auf den in MPAL integrierten Profiling-Methoden werden charakteristische Laufzeitparameter generiert und vorgestellt, die eine quantitative Unterscheidung von Software- sowie Hardware-Eigenschaften ermöglichen.
- **Prädiktion der Performance paralleler Applikationen:** Mithilfe der extrahierten quantitativen Parameter der Laufzeitcharakteristik wird eine neue statistische Methode zur Vorabschätzung des parallelen Laufzeit- und Skalierungsverhaltens entwickelt und diskutiert.
- **Beispielimplementierungen:** Anhand von Beispielimplementierungen aus dem Bereich der kamerabasierten Fahrerassistenzsysteme wird der Einsatz der vorgestellten Methoden gezeigt und evaluiert.

Während in Kapitel 2 zunächst die Grundlagen des parallelen Laufzeitverhaltens speziell von eingebetteten Systemen und MPSoCs sowie aktuelle Ansätze der parallelen Programmierung beschrieben werden, wird in Kapitel 3 die Abstraktionsschicht MPAL eingeführt (siehe Anhang A für Zusammenfassung des Befehlssatzes). Anschließend werden in Kapitel 4 die charakteristischen Laufzeitparameter beschrieben, die im Anschluss für die in Kapitel 5 beschriebene statistische Methode zur Prädiktion des parallelen Laufzeitverhaltens angewendet werden. Während in Kapitel 6 die Ergebnisse der vorgestellten Methoden anhand von Fallbeispielen dargestellt und evaluiert werden, schließt das Kapitel 7 diese Arbeit mit einem Fazit ab.

2 Parallele Programmierung

Nicht nur Prozessoren für Spezialanwendungen und Hochleistungsrechenzentren, sondern fast alle GPP in Desktopcomputern, Laptops, Smartphones und Wearables (z.B. Smart-Watches) besitzen heutzutage Mehrkernprozessoren. Um die von der Hardware bereitgestellte Performance nutzen zu können, müssen Anwenderprogramme parallelisiert werden. Mit der Formulierung „the free lunch is over" formulierte Herb Sutter [6] den Zeitenwechsel, nachdem Software-Entwickler nicht mehr ohne eigenes Zutun von der Performance-Steigerung von Singlecore-Prozessoren profitieren können. Stattdessen müssen Entwickler selbst dafür sorgen, dass Software-Implementierungen parallele Prozessoren auslasten können. Des Weiteren soll Software im Allgemeinen die Anforderung der Portabilität erfüllen, was in diesem Sinne nicht nur bedeutet, dass sie auf unterschiedlichen Plattformen ausführbar ist, sondern auch die vorhandenen Ressourcen optimal ausnutzt, um bei möglichst geringen Hardware-Kosten die für Echtzeitanwendungen erforderlichen Performance-Kriterien zu erfüllen (*Performance-Portabilität*). Um Software zwischen verschiedenen Prozessortypen performanceportabel zu halten, muss sie flexibel über eine dynamische Prozessorkern-Anzahl skalierbar sein, weshalb für die folgenden Programmiermethoden besonders Skalierbarkeits- und Portabilitätsaspekte betrachtet werden.

Während applikationsspezifische Hardware in einem *Application Specific Integrated Circuit* (ASIC) exzellente Rechenleistung und Verlustleistungscharakteristiken aufweist, ist die finale Implementierung auf eine einzige Applikation bzw. ein kleines Applikationsfeld begrenzt. Programmierbare Prozessoren besitzen eine geringere Rechenleistung und eine geringere Verlustleistungseffizienz, profitieren aber von höchster Flexibilität [1]. Die Wahl zwischen einem einzigen leistungsstarken Singlecore oder einer Vielzahl von einfacheren Kernen in einem Multicore kann ebenfalls die resultierende Rechenleistung

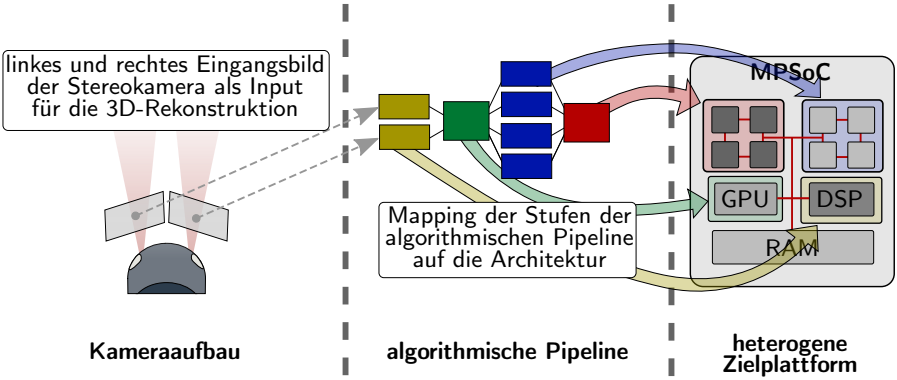


Abbildung 2.1: Darstellung des Mappings einer algorithmischen Pipeline (Stereo-Vision) auf einen heterogenen MPSoC mit zwei CPU-Clustern, einer GPU und einem DSP als verfügbaren Recheneinheiten unter denen die nebenläufigen Tasks eines Algorithmus gleichmäßig verteilt werden müssen.

einer Applikation stark beeinflussen [7]. Da die jeweiligen Anforderungen der Software an die darunterliegende Hardware stark mit der Applikation variieren können, kann in manchen Szenarien ein leistungsstarker Singlecore die Ausführungsgeschwindigkeit eines Multicores überbieten oder umgekehrt. Während manche Anwendungen von massiv parallelen Architekturen (Multi- oder Manycore) oder Beschleunigern (z.B. GPU) profitieren, benötigen strikt sequentielle Prozesse spekulative Ausführungstechniken eines Singlecores. Daher kann nur durch die Kombination von heterogenen Recheneinheiten mit unterschiedlichen Charakteristiken (z.B. heterogene CPU-Cluster, GPUs oder DSPs) eine Hardwarearchitektur geschaffen werden, die für jede Art von algorithmischen Anforderungen die optimale Recheneinheit zur Verfügung stellen kann. Ein solches heterogenes System sowie das Mapping von algorithmischen Blöcken auf die heterogenen Recheneinheiten (*Architektur-Mapping*) ist schematisch in Abbildung 2.1 gezeigt.

Durch die Trennung der Speicheradressbereiche von heterogenen Systemen und die Kombination von kohärenten und nicht kohärenten Speicherdomänen stellt ein solches Architektur-Mapping allerdings eine große Herausforderung für die Programmierung dar. Des Weiteren macht die Verwendung von Beschleunigern in der Regel eine Neuimplementierung der auszulagernden Compute-Kernel in einer geeigneten Sprache nötig, wie zum Beispiel OpenCL [8], oder CUDA für die direkte Programmierung von Beschleunigerprogrammen. OpenACC [9], sowie neuere Versionen von OpenMP [10], bieten dagegen Direktiven, die es für einige Beschleuniger ermöglichen, Bereiche des C++ Host-

Programms auf Beschleuniger auszulagern. Die Verwaltung der unterschiedlichen Recheneinheiten (Initialisieren der Geräte und Kernel, Triggern der Kernel und Synchronisation), sowie Speicherverwaltung erfordert zusätzlich die Verwendung von spezifischen APIs innerhalb des Hauptprogramms. Manche Ansätze fokussieren sich auf die Vereinheitlichung von Programmierschnittstellen beispielsweise durch die Implementierung von Task-Verwaltungs- und Kohärenzmechanismen in einer *Runtime*-Bibliothek (z.B. NVIDIA Unified Memory [11] für NVIDIA GPUs). Andere Entwickler avisieren die Integration von Cache-Kohärenz und Task-Verwaltung in die zugrundeliegende Hardware (z.B. *Heterogeneous Systems Architecture* (HSA) [12] für *HSA-enabled* Plattformen), um dadurch die Programmierbarkeit besonders von heterogenen Plattformen zu erhöhen.

2.1 Frameworks zur Parallelisierung

Während in der wissenschaftlichen Gemeinschaft (engl. *Scientific Community*) verschiedene Ansätze und Werkzeuge entwickelt werden, um Anwendungsentwickler bei der Programmierung von parallelen und heterogenen Plattformen zu unterstützen, ist ein Trend noch nicht sichtbar. Manche Werkzeuge unterstützen den Entwickler mit hilfreichen Hinweisen, wie z.B. Nebenläufigkeiten und Datenabhängigkeitsanalysen, geschätztem Speedup durch Parallelisierung oder parallelen Profiles (*Tracings*). Die Parallelisierung wird bei diesen Ansätzen aber weiterhin manuell ausgeführt. Andere Werkzeuge analysieren dagegen das gegebene Programm selbstständig und führen voll automatisiert eine Parallelisierung durch. Neben dem durch Parallelisierung erreichbaren Speedup darf allerdings auch das Optimierungspotential durch sequentielle Optimierungen nicht vernachlässigt werden, die durch einfache Optimierungen teilweise einen höheren Speedup bringt, als eine Parallelisierung und daher immer bereits vor der eigentlichen Parallelisierung durchgeführt werden [13]. Dabei besteht der Parallelisierungsprozess mit dem Ziel einer Steigerung der Ausführungsgeschwindigkeit aus den folgenden Arbeitsschritten (es wird immer eine bereits sequentiell optimierte Implementierung vorausgesetzt):

0. *Sequentielle Optimierung:* Pointerarithmetik, Schleifenoptimierung, Speicherzugriffsreihenfolgen (Datenlokalität), Vektorisierung, etc.
1. *Algorithmische Partitionierung:* Nebenläufigkeits- und Datenabhängigkeitsanalyse und Arbeitsaufteilung
2. *Parallele Implementierung:* Einsatz eines Frameworks zur Parallelisierung bzw. Implementierung von Beschleuniger-Kerneln

3. *Architektur-Mapping:* Verteilung der verfügbaren parallelen Tasks auf die Recheneinheiten der Zielplattform

C++ bietet keine native Möglichkeit, um Software-Code auf Multicore-Prozessoren, die denselben kohärenten Speicherbereich verwenden, zu parallelisieren (Shared-Memory-Modell). Neben den erst in neueren Versionen von C++ bzw. der C++ *Standard Template Library* (STL) verfügbaren parallelen Algorithmen, wird eine parallele Ausführung daher häufig mithilfe von Frameworks, wie OpenMP [10], OmpSs [16], TBB [17] oder CilkPlus [18] realisiert. Diese Parallelisierungs-Frameworks unterstützen unterschiedliche Methoden, um Software-Code parallel auszuführen, entweder durch explizite Erstellung von Threads (*Fork*), Erstellen von Tasks (*Spawn*), oder durch Verwendung von parallelen Algorithmen (z.B. *for_each* oder *reduce*). Message-Passing-Protokolle, die beispielsweise per Mailbox-Register eine direkte Interprozess-Kommunikation (engl. *Inter-Process Communication* (IPC)) zwischen Prozessorkernen realisieren, werden in dieser Arbeit nicht betrachtet. Tabelle 2.1

Tabelle 2.1: Relevante Parallelisierungs-Frameworks für Multicore-CPUs.

Tool	Merkmale	Portabilitätsaspekte
OpenMP	○ Häufig verwendete Basis-Methode ○ Direktivenbasierter Ansatz (initial fokussiert auf Threading) [14] ○ Beinhaltet Aspekte von OmpSs (Scheduling) und OpenAcc (Offload)	+ Direktiven können ohne Einschränkung ignoriert werden − Direktiven müssen vom Compiler unterstützt werden − Skaliert nicht in allen Experimenten
OmpSs	○ Direktiven basiert ○ Portiert regelmäßig neue Funktionen zu OpenMP-Standards ○ Erweitert OpenMP u.A. um Beschleuniger-Support unter Verwendung von OpenCL und CUDA	+ Durch OpenCL und CUDA portabel über viele Architektur-Typen + Direktiven können ignoriert werden + Runtime-Backend unterstützt erweiterte Task-Scheduling Mechanismen − Erfordert den Einsatz des OmpSs-spezifischen Compilers
TBB	○ Fokussiert Task-Parallelisierung ohne Zugriff auf Threads [15] ○ Performant skalierende parallele Algorithmen und Datenstrukturen ○ Implementiert als Bibliothek ohne weitere Abhängigkeiten	+ Bibliothek ist kompatibel zu allen Standard-Compilern − Kein Beschleuniger Support − Fehlende Thread-Kontrolle erschwert manche Parallelisierungsstrategien
CilkPlus	○ Parallelisierungsindikatoren, Runtime entscheidet parallele Ausführung ○ Array-Slicing Notation ermöglicht explizite Formulierung von Datenparallelität für SIMD-Vektorisierung ○ Wird seit 2018 nicht weiterentwickelt	+ Entscheidung zur Laufzeit, ob parallelisiert wird, reduziert den Overhead − Slicing-Notation erfordert Support des Compilers − Kein Support für Beschleuniger
C++17 STL	○ Threads seit C++11 (*Thread*, *Mutex*, *Atomic*, *Futures* und *Promises*) ○ Parallele Algorithmen seit C++17 (e.g., *for_each* oder *transform*) ○ Support in GCC's STL (libstdc++) seit Version 9.1, in Intel's PSTL oder HPX-Bibliothek verfügbar	○ PSTL verwendet TBB oder OpenMP und übernimmt damit deren Portabilitätseigenschaften − Kein nativer Beschleuniger-Support

zeigt eine Zusammenfassung der relevantesten Parallelisierungs-Frameworks, deren Hauptmerkmale sowie Portabilitätsaspekte.

Während jedes der aufgeführten Frameworks unterschiedliche Eigenschaften aufweist, haben OmpSs und TBB für viele Anwendungsbereiche derzeit die leistungsstärksten Strategien zur Verteilung der Tasks (*Scheduling*). Manche Scheduling-Strategien können zwar Overhead für die nötige Berechnung involvieren, aber durch effiziente Verwaltung der Threads und Tasks auch wesentlich zur Steigerung der Skalierbarkeit und Portabilität beitragen. Allerdings bieten diese Frameworks, mit Ausnahme von OmpSs, keine Möglichkeit, Beschleuniger zu verwalten bzw. zu programmieren, was eine Einschränkung der Portabilität bedeutet. Ab dem Standard OpenMP 5.0 wurde die erweiterte Unterstützung für die Verwaltung von Beschleunigern von OmpSs übernommen. Da OpenMP als die am weitesten verbreitete Methode zur Parallelisierung angesehen wird, die von allen Standard-Compilern unterstützt wird, hat eine leistungsfähigere

```
// Parallelisierung mit OpenMP (Direktiven)
#pragma omp parallel
for( auto &v: vec ) { // Parallelisierung regulaerer Schleifen
        doWork( v );
}
```

(a) Exemplarische Parallelisierung (for-each) mit OpenMP.

```
// Parallelisierung mit TBB (Bibliothek)
using namespace tbb;
// Parallelisierung per Iterator
parallel_for_each( begin(vec), end(vec), []( auto &v ){
        doWork( v );
} );
```

(b) Exemplarische Parallelisierung (for-each) mit TBB.

```
// Parallelisierung mit C++17 (Nativer Bestandteil der STL)
using namespace std;
// Zus. Definition der Execution-Policy "par"
for_each( execution::par, begin(vec), end(vec), []( auto &v ){
        doWork( v );
} );
```

(c) Exemplarische Parallelisierung (for-each) mit C++17.

Algorithmus 2.1: Exemplarische Parallelisierungsbeispiele mit OpenMP, TBB und C++17, wobei für jedes Objekt eines Vektors (std::vector) die Funktion *doWork* aufgerufen wird.

Version von OpenMP das Potential die Portabilität von vielen Implementierungen zu verbessern. Um die unterschiedlichen Programmiertechniken darzustellen, sind Beispielparallelisierungen einer parallelen for-each-Schleife mit OpenMP, TBB und C++17 sind in Algorithmus 2.1 gegeben.

Als weitere Schwierigkeit bei der Parallelisierung variiert die Performance der Frameworks teilweise stark zwischen Plattformen, was als Konsequenz auch die Portabilität einschränkt. Für die Programmierung von CPUs eignen sich besonders Shared-Memory-Frameworks, wie OpenMP, TBB oder CilkPlus, da sie geringeren Overhead involvieren. Für die Programmierung von Beschleunigern muss meist ein spezielles *Offload*-Framework eingesetzt werden, wie OpenCL oder CUDA, mit dem Beschleuniger-Programme (*Compute-Kernel*) implementiert werden. Während CUDA-Kernel nur auf NVIDIA GPUs ausgeführt werden können, bieten OpenCL-Kernel eine weit höhere Portabilität nicht nur für unterschiedliche GPUs, sondern auch für andere Beschleuniger und sogar Multicore-CPUs. Aufgrund der implizit sehr feinen Granularität der Parallelisierung mit OpenCL, die normalerweise für massiv parallele Grafikprozessoren ausgelegt ist, ist die Ausführungsgeschwindigkeit auf einem Multicore meist deutlich geringer als die von Shared-Memory-Frameworks [19]. Manche Compiler, wie der *Intel C/C++ Compiler* (ICC) oder *GNU Compiler Collection* (GCC), unterstützen auch eine Direktiven-basierte Programmierung von Beschleunigern durch Kennzeichnung von Code-Abschnitten innerhalb des C++-Codes entweder durch Compiler-eigene Pragmas, oder durch die Verwendung von neueren Versionen von OpenMP bzw. OmpSs. Auch die C++ STL unterstützt immer weitere Möglichkeiten zur Parallelisierung auf Multicore-Prozessoren, sowie Programmierung von Beschleunigern als Teil der aktuellen Forschung, was hingegen eine umfangreiche Erweiterung der Runtime-Bibliothek erfordert. Aktuell ist die Anzahl der unterstützten Architekturen noch sehr gering, weshalb sich diese Arbeit mit der Verbesserung der Programmierbarkeit von heterogenen, parallelen Prozessoren beschäftigt.

2.2 Automatisierte Parallelisierung

Neben den diskutierten Frameworks zur manuellen Parallelisierung gibt es Werkzeuge, die den sequentiell geschriebenen Software-Code automatisch auf Datenabhängigkeiten untersuchen und dadurch potentielle Nebenläufigkeiten identifizieren. Diese Mechanismen, die durch statische Analyse des Software-Codes die sequentielle Implementierung während des Kompilationsprozesses parallelisieren, sind in allen Standard-Compilern integriert und können daher als sehr portabel angesehen werden. Durch aktivieren dieser Zusatzfunktion in den Standard-Compilern können selbst unerfahrene Entwickler aus

rein sequentiellem Software-Code adäquate Parallelisierungsergebnisse erzielen und so Vorteile aus parallelen Architekturen ziehen. Allerdings beinhaltet rein sequentiell geschriebener Code, der keine dedizierte Separation zwischen unabhängigen algorithmischen Bereichen enthält, häufig auch versehentlich eingebrachte zusätzliche Datenabhängigkeiten (z.B. iterative Akkumulationen oder Modifikationen mit Abhängigkeiten zum vorherigen Ergebnis) und andere Limitierungen, die eine Parallelisierung erschweren. Da diese Implementierungen somit oft nicht den eigentlichen Nebenläufigkeitsgraph des originalen Algorithmus repräsentieren, ist die automatische Parallelisierung nicht in der Lage den optimalen Speedup zu erreichen. Des Weiteren kann häufig die echte Nebenläufigkeit in Bereichen mit angewendeter Pointer-Arithmetik nicht evaluiert werden, da die Kenntnis über die verwendeten Speicherregionen, Datenabhängigkeiten und Konflikte (z.B. *Read-after-Write*) fehlt. Aus diesem Grund können Entwickler, die bereits mit einem hoch optimierten sequentiellen Software-Code arbeiten, wie er in eingebetteten Systemen eingesetzt wird (z.B. in Fahrerassistenzsystemen), nur wenig Speedup von automatisierter Parallelisierung erwarten.

Profilebasierte automatische Parallelisierungsmethoden verwenden Informationen über das dynamische Ausführungsverhalten und messen zur Laufzeit die Speichernutzung und eventuelle Abhängigkeiten, um so Nebenläufigkeiten selbst in algorithmischen Regionen mit Pointer-Arithmetik identifizieren zu können [20, 21]. Einerseits können diese Tools einen wesentlich höheren Speedup erreichen. Andererseits kann es bei der erzeugten Partitionierung allerdings zu unsicherer Nebenläufigkeit kommen, sodass eine spätere manuelle Verifikation durch den Entwickler nötig ist. Die Tool-Sammlung von Silexica zum Beispiel bietet sowohl unterstützende Profiling- und Analysewerkzeuge, als auch eine automatisierte profilebasierte Parallelisierung. Dabei wird eine Code-zu-Code Transformation des sequentiellen Software-Codes durchgeführt und anschließend mit OpenMP-Direktiven annotiert, sodass der Entwickler diese verifizieren oder gar optimieren kann. Dadurch, dass ein erster Parallelisierungsansatz automatisch erstellt wird, kann dies ein geeigneter Einstieg auch für unerfahrene Enwickler sein, die anschließend lediglich die Parallelisierung verifizieren und optimieren müssen. Außerdem werden durch Analyse des assemblierten Codes und unter Verwendung von mitgelieferten Architekturmodellen geschätzte Speedup-Metriken für die parallelisierten Regionen mitgeliefert. Wie in [22] beschrieben, unterstützen auch die Silexica-Tools ein automatisches Architektur-Mapping auf Beschleuniger, wie z.B. DSP oder FPGAs unter Verwendung von *High-Level Synthesis* (HLS), die Software-Code in eine Hardware-Beschreibung übersetzt.

2.3 Parallelisierungs-Strategien

Nebenläufigkeiten können auf unterschiedlichen Ebenen auftreten und mit unterschiedlichen Strategien in einer parallelen Implementierung umgesetzt werden. Dabei spielt die Identifizierung der geeigneten Methode oft eine maßgebliche Rolle für die resultierende Komplexität der Implementierung, den Parallelisierungserfolg (Erreichen eines angestrebten Speedups), sowie die Flexibilität, Skalierbarkeit und Portabilität der parallelen Implementierung. Daher wird in Anlehnung an [23] der Parallelisierungsprozess in dieser Arbeit wie folgt definiert und nachfolgend näher beschrieben (siehe [24, 25, 26, 27, 28] für weiterführende Literatur): Es muss zunächst die Art der Nebenläufigkeiten identifiziert und anschließend eine geeignete Strategie zur Realisierung der parallelen Implementierung gewählt werden (siehe Abbildung 2.2). Für die konkrete Implementierung werden meist parallele Software-Strukturen, vorgegebene Muster (engl. *Pattern*) oder parallele Algorithmen verwendet. Arbeiten Prozesse einer Datenparallelisierung beispielsweise in strikt getrennten Datenbereichen, ist keine Synchronisation nötig. Andernfalls werden entweder manuell eingebrachte explizite Synchronisationen, beispielsweise unter Verwendung eines *Mutual-Exclusive Lock* (Mutex), benötigt, oder es werden parallele Datenstrukturen oder parallele Zugriffsmuster eingesetzt.

Parallelisierungs-Level: Bei der *Task-Level-Parallelisierung* bzw. *Funktions-Level-Parallelisierung* werden voneinander unabhängige Tasks oder Funktionen auf jeweils unterschiedlichen Recheneinheiten ausgeführt, die im algorithmischen Kontext erst zu einem späteren Zeitpunkt synchronisiert werden. Dagegen wird eine *Daten-Level-Parallelisierung* innerhalb einer einzigen Funktion angewendet, wobei dieselbe Berechnung auf einer Vielzahl von – im besten Fall vollständig voneinander unabhängigen – Daten ausgeführt wird. Viele Mikroprozessoren speziell für eingebettete Systeme sowie Beschleuniger implementieren unterschiedliche Formen von *Instruktions-Level-Parallelisierung* in

Abbildung 2.2: Der Parallelisierungsprozess beginnt bei der Identifizierung des Parallelisierungs-Levels (z.B. Task- oder Datenparallelität), um daraus eine geeignete Parallelisierungs-Strategie abzuleiten. Die parallele Implementierung wird dann mithilfe von parallelen Software-Strukturen, parallelen Algorithmen und parallelen Datenstrukturen, Datenzugriffsmustern oder manuellen Synchronisationen realisiert.

Form von (superskalarenm, out-of-order) Instruktions-Pipelines, duplizierten *Arithmetic Logic Unit* (ALU)s (z.B. VLIW) oder SIMD-Vektorinstruktionen.

Parallelisierungs-Strategien: Für die parallele Ausführung unabhängiger Funktionen können explizite *Threads* oder *Tasks* generiert werden. Für eine Datenparallelisierung innerhalb einer algorithmischen Stufe kann eine *Domain-Decomposition* angewendet werden, wobei die Granularität häufig einen entscheidenden Einfluss auf den Speedup hat. Ebenfalls können in parallelen Implementierungen *Producer-Consumer* bzw. *Master-Worker* oder *Pipeline-Ausführung* eingesetzt werden. Einige Programmiersprachen beherrschen auch eine asynchrone Ausführung der Tasks (*Futures* und *Promises*).

Parallele Software-Strukturen: Eine konkrete Parallelisierung kann entweder durch direkte Verwendung von Kernel-Threads (z.B. nativer *POSIX Thread* (PThread) oder OpenMP-Thread) oder durch die Verwendung der taskbasierten Ansätze eines der Parallelisierungs-Frameworks umgesetzt werden. Hierbei können entweder Tasks explizit generiert, gestartet und synchronisiert oder einer der verfügbaren parallelen Algorithmen angewendet werden, wie z.B. *parallel_for* oder *reduce*. *Futures* und *Promises* werden von einigen Parallelisierungs-Frameworks direkt als Datenobjekt angeboten.

Parallele Datenstrukturen: Abhängig von der gewählten Parallelisierungs-Strategie müssen geeignete – eventuell für den parallelen Zugriff abgesicherte – Datenstrukturen gewählt werden. Grundsätzlich muss unterschieden werden, ob es einen Thread- bzw. Tasklokalen Speicherbereich gibt, ein geteilter Speicher nur gelesen wird, oder ob ein geteilter Speicher eventuell auch gleichzeitig geschrieben wird, was eine Synchronisation der Zugriffe erfordert. Nicht nur APIs für Thread-sicheren Zugriff auf Datenstrukturen durch *scoped locking*, wodurch auch Deadlocks verhindert werden, sondern auch spezifische Zugriffs-Muster, wie *reduce* (z.B. $b = \sum_i a[i]$), *gather* (z.B. $a[:] = b[c[:]]$) und *scatter* (z.B. $a[c[:]] = b[:]$) kommen hier zum Einsatz.

Als eine weitere Form der Datenparallelisierung auf Instruktionsebene können Vektorregister mit dazugehörigen SIMD-Operationen verwendet werden. Werden mehrere Datenelemente desselben Typs in einem Vektorregister gespeichert, können SIMD-Operationen dieselbe Operation auf allen im Vektor befindlichen Datenelementen in nur einer Instruktion ausführen. SIMD-Instruktionssatz-Erweiterungen aus verschiedenen Generationen und von verschiedenen Herstellern (z.B. *Multi-Media Extension* (MMX), *Streaming SIMD Extension* (SSE), *Advanced Vector Extensions* (AVX), oder *NEON Media Processing Engine* (NEON MPE)) unterscheiden sich in ihrer Registergröße, den unterstützten Datenformaten und Operatoren (z.B., *gather/scatter*, *reduce*, oder maskierte bzw. bedingte Ausführungen in modernen Instruktionssätzen). Darum kann es eine Herausforderung sein, eine portable Software zu entwickeln,

Tabelle 2.2: Methoden zur Programmierung von SIMD-Instruktionen.

Tool	Merkmale	Portabilitätsaspekte
Intrinsics	○ Direkte Verwendung von SIMD-Instruktionen durch instruktionssatzspezifische Built-In Funktionen ○ Instruktionsbasierte Entwicklung ○ Verwendung der vollen Funktionalität des Instruktionssatzes möglich	− Unflexible implementierung gebunden an verwendeten Instruktionssatz − Portable Software benötigt separate Implementierungen in unterschiedlichen Instruktionssätzen
CilkPlus	○ Explizite Formulierung von datenparallelem Software-Code durch Array-Slicing Notationen ○ Unterstützt komplexe Operationen, wie *gather*, *reduce*, Maskierung ○ Verfügbar in ICC, LLVM, GCC ○ Seit 2018 nicht mehr weiterentwickelt	+ Implementierung ist nicht limitiert auf einzelnen Instruktionssatz + Kompakte Formulierung von Datenparallelität − Nicht verfügbar für ARM-CPUs − Nur noch als veraltete Version (engl. *deprecated*) verfügbar
Auto - Vektorisierung	○ Verfügbar in Standardcompilern ○ Nebenläufigkeitsanalyse basierend auf statischer Code-Analyse (Pattern-Matching) ○ Vektorisierungserfolg sollte immer verifiziert werden durch Analyse / Untersuchen des assemblierten Codes	+ Erste Vektorisierung erfordert keine Änderungen des Software-Codes + Selbst unerfahrene Entwickler können SIMD-Instruktionen nutzen − Fehlgeschlagenes Pattern-Matching oder unpassende Datentypen können zu überflüssigen Konvertierungsoperationen führen
OpenMP	○ Direktiven-basierte SIMD Hinweise verwenden ähnliche Vektorisierungs-Pattern, wie Auto-Vektorisierung ○ Ermöglicht zusätzliche Kontrolle der Parallelitäten und Abhängigkeiten gegenüber Auto-Vektorisierung	+ Direktiven können ignoriert werden + Erweiterte Kontrolle der automatischen Vektorisierung von Patterns − Vektorisierungs-Erfolg sollte verifiziert werden (Assembler-Code)

die den vollen Leistungsumfang aller Vektor-Instruktionenssätze ausnutzen kann. Möglichkeiten, um SIMD-beschleunigten Software-Code zu schreiben sind in Tabelle 2.2 aufgeführt.

2.4 Beschleuniger

Die bislang diskutierten Methoden zur Parallelisierung fokussieren die Programmierung von homogenen Multicore-Prozessoren. Um jedoch alle Ressourcen von heterogenen MPSoCs auszunutzen, muss auch die Programmierung von Beschleunigern und heterogenen CPU-Clustern sowie ein anpassungsfähiges Architektur-Mapping berücksichtigt werden.

Da Speicherbereiche des Hauptprogramm-ausführenden Prozessors (*Host-CPU*) und Beschleunigern oft strikt getrennt sind, können Beschleuniger nicht mit gewöhnlichen Shared-Memory-Frameworks programmiert werden, sondern benötigen einen dedizierten *Offload*. Für einen solchen Offload ist nicht nur die Synchronisation der parallelen Tasks (wie bei Shared-Memory-Paralle-

Tabelle 2.3: Effekte, die das Programmieren heterogener Systeme erschweren.

Speicher-Verwaltung:	◦ Selbst in MPSoC, in denen Host-CPU und Beschleuniger auf denselben Hauptspeicher zugreifen, sind häufig für jede Recheneinheit separierte Speicherregionen reserviert ◦ Daten müssen häufig explizit transferiert werden ◦ Heterogene Recheneinheiten in einem System haben meist keinen kohärenten Speicher ⇒ Da sich Speicherzugriffsstrategien zwischen Systemen unterscheiden, muss eine portable Implementierung verschiedene Strategien beherrschen
Beschleuniger-Verwaltung:	◦ Verwaltungsaufgaben, wie das Triggern von DMAs, Beschleuniger-Kernel einreihen oder synchronisieren muss von der Host-CPU gesteuert werden ◦ Zusätzlicher Programmieraufwand und Rechenressourcen für Ressourcenverwaltung benötigt ⇒ Ressourcenverwaltung kann ein aufwändiger Teil der heterogenen Implementierung sein
Compute-Kernel:	◦ Die meisten Beschleuniger können nicht mit C/C++ programmiert werden, sondern benötigen eine entsprechende Implementierung beispielsweise in CUDA oder OpenCL ◦ Die Verwendung von OpenCL erlaubt die Ausführung des Kernels auf FPGAs durch HLS [29] ◦ Das Migrieren von Beschleuniger Kerneln zwischen Plattformen erfordert häufig Anpassungen der Parameter, wie z.B. die Task-Granularität ⇒ In den meisten Fällen sind mehrere Implementierungen des funktionell selben algorithmischen Tasks erforderlich, um unterschiedliche Architekturen gleichzeitig zu verwenden

lisierung), sondern auch das explizite Kopieren der benötigten Daten bzw. Funktionsargumente nötig. Ist ein DMA vorhanden, kann das Kopieren zwar parallel zu Berechnungen von Tasks geschehen sofern es die algorithmischen Anforderungen erlauben, allerdings müssen diese Kontrollvorgänge manuell in der Programmierung berücksichtigt werden. Somit erschwert sowohl das erforderliche Architektur-Mapping, als auch die Kommunikation zwischen Host-CPU und Beschleunigern für die Synchronisation der Tasks und das explizite Kopieren von Daten die Programmierung von heterogenen MPSoCs. Aspekte, die bei der Programmierung von heterogenen Systemen berücksichtigt werden sollten, sind in Tabelle 2.3 dargestellt.

Um bei einer Software-Migration zwischen Plattformen durch möglichst optimale Auslastung der jeweils verfügbaren Recheneinheiten eine performante Ausführung zu gewährleisten, muss das Architektur-Mapping anpassbar sein. Allerdings ist eine vollständige Suche aller möglichen Kombinationen des Architektur-Mappings sehr zeitintensiv und wegen des großen Entwurfsraums und der Vielzahl von freien Parametern oft nicht anwendbar. Eine Änderung der Scheduling-Strategie oder der Arbeitsverteilung kann daher besonders bei heterogenen Plattformen große Herausforderungen darstellen. Werden heterogene CPU-Cluster zeitgleich verwendet, kann die Arbeit nicht zu gleich

großen Teilen aufgeteilt werden, um Wartezeiten der Prozessoren zu vermeiden, sondern muss abhängig von der individuellen Prozessorleistung ausbalanciert werden. Soll ein Programmabschnitt auf einen Beschleuniger ausgelagert werden, muss dieser Abschnitt in einer für den Beschleuniger geeigneten Sprache (z.B. OpenCL, CUDA oder Vulkan [30]) neu implementiert werden.

Aufgrund von unterschiedlichen Anforderungen der Software und variierenden Hardware-Eigenschaften der Ziel-Architektur, kann es bei der Migration einer bestehenden Implementierung zwischen Plattformen zu Bottlenecks kommen. Wird beispielsweise eine SIMD-optimierte Implementierung mit dichten Vektoren aus kleinen ganzzahligen Datentypen (z.B. uint8) auf eine GPU portiert, die jedoch nur SIMD-Operationen mit größeren Gleitkommazahlen unterstützt (z.B. float32), kann es zu einem Performance-Verlust kommen. In diesem Fall können durch die geringere Packungsdichte der Daten in den Vektorregistern nicht nur weniger Daten pro Operation verarbeitet werden, sondern es kommt zu zeitaufwändigen Konvertierungen vor und nach den jeweiligen Vektoroperationen. Implementierungen, die durch HLS auf einem FPGA ausgeführt werden sollen, profitieren dagegen von Fixpunkt-Zahlenformaten und einer eher grob granularen Compute-Kerneln. Allgemeine Aspekte, die die Portabilität einer heterogenen Software potentiell einschränken können, sind in Tabelle 2.4 zusammengefasst.

Werden mehrere heterogene Recheneinheiten gleichzeitig verwendet, sind komplexere Architektur-Mapping-Strategien notwendig. *Auto-Tuning* Ansätze bieten ein automatisches Balancieren der Arbeitsverteilung zur Laufzeit [31, 32], was allerdings zu einem nicht-deterministischen Verhalten der resultierenden Software führt. Diese Methode hat eine hohe Flexibilität und kann somit für bestimmte Szenarien von großem Vorteil sein, sofern sich die Arbeitspakete feingranular genug aufteilen lassen. Diese Ansätze sind jedoch nicht anwendbar bei Applikationen, bei denen Echtzeitbedingungen eingehalten werden müssen, da eine maximale Latenz bis zur Bereitstellung der Ergebnisse nicht garantiert werden kann. Im Folgenden werden grundlegende Offload-Schemata kurz zusammengefasst, die sich sowohl in den Anforderungen an das Kommunikationsnetz zwischen Recheneinheiten (z.B. CPU, GPU oder *Image Signal*

Tabelle 2.4: Portabilitätslimitierende Faktoren von heterogenen Systemen.

Architekturen	○ Fehlende Spezialisierung des Instruktionssatzes für Algorithmus ○ Unpassende Granularität der Parallelisierung
System-Design:	○ Lange Datentransfer-Zeiten ○ Speicherzugriffs-Barrieren
Programmierbarkeit:	○ Nicht unterstützte Programmiersprachen / APIs ○ Komplexe Konfiguration und Parameter-Feinabstimmung

Processor (ISP)), der resultierenden Anfälligkeit für Arbeitsungleichverteilung und dem Task-Verwaltungs-Overhead teilweise stark unterscheiden. Häufig wird eine Mischform der unten aufgeführten Strategien eingesetzt.

Single-Stage Offload: Als einer der einfachsten Ansätze, ähnlich zu einer Instruktionssatzerweiterung, wird aus einer algorithmischen Pipeline aus mehreren aufeinanderfolgenden Bearbeitungsstufen (*Stages*) nur die rechenintensivste Stufe ausgewählt, für den Beschleuniger implementiert und ausgelagert, während alle anderen Stufen noch auf dem Host berechnet werden. Dieses Offload-Schema kann nur bei geringen Datentransferzeiten zwischen Host und Beschleuniger und einem guten Speedup der beschleunigten Pipeline-Stufe einen Gesamt-Speedup erwirken, da der Speedup den Kontroll- und Kopier-Overhead überwiegen muss.

Domain/Data Decomposition: Um die Gesamtlatenz einer algorithmischen Pipeline zu minimieren, muss die Rechenleistung aller verfügbaren Recheneinheiten für die Bearbeitung eines einzelnen Frames zusammen eingesetzt werden. Dazu muss gegebenenfalls jede Stufe der algorithmischen Pipeline – abhängig von der Algorithmenstruktur – die Arbeit aufteilen und über mehrere Recheneinheiten verteilen. Da die meisten Beschleuniger für bestimmte Klassen von Algorithmen optimiert sind, erreichen manche Offloads nur geringen Speedup, wodurch dieses Schema in vielen Fällen zwar eine gute Latenzminimierung bietet, nicht aber den optimalen Durchsatz erreicht.

Pipeline: Der Durchsatz kann optimiert werden, indem eine Zuweisung der einzelnen algorithmischen Stufen zu Recheneinheiten erfolgt, die jeweils die beste Ausführungseigenschaften aufweisen. Dabei ist es auch möglich, Teile der Recheneinheiten jeweils für die Berechnung unterschiedlicher Frames einzusetzen. Durch Abtausch mit einer höheren Latenz kann so der Durchsatz optimiert werden. Transferzeiten und die definierte Pipeline-Frequenz, mit der Zwischenergebnisse von einer Stufe zur nächsten gereicht werden, begrenzen jedoch die Gesamt-Performance dieses Schemas. Außerdem limitiert die fixe Zuweisung der algorithmischen Stufen zu Recheneinheiten die Flexibilität und Portabilität einer solchen Implementierung, da sie auf einer vorgegebenen Konstellation von Recheneinheiten ausgelegt ist.

Statische Arbeitsteilung: Transferzeiten von großen Datenmengen, wie beispielsweise Zwischenergebnisse der algorithmischen Stufen, können den Offload-Speedup stark begrenzen. Daher kann es sinnvoll sein den gesamten Algorithmus oder zumindest große Teile davon auf einem Beschleuniger auszuführen, da Input- und Output-Daten oft deutlich weniger Transferzeiten benötigen. Währenddessen kann der Host bereits Vorverarbeitungen für den nächsten zu bearbeitenden Frame übernehmen, was unter Verwendung eines Framebuffers den Gesamtdurchsatz maximieren kann.

Im Gegensatz zu statischem Architektur-Mapping sind auch hier nichtdeterminische Ansätze denkbar, die beispielsweise unabhängig von der zu erwartenden Ausführungszeit der algorithmischen Blöcke aus einer globalen Warteschlange Tasks entnehmen und auf der nächsten freien Recheneinheit ausführen. Auch kann die zunächst willkürliche Verteilung der Tasks durch Berücksichtigung von Kosten für die Kommunikation, Eignung der Aufgabe für die jeweils freie Recheneinheit, sowie maximale Deadline für den Task priorisiert bzw. sortiert werden.

Für Algorithmen einer speziellen Applikationsklasse, können Entwickler auf Bibliotheken zurückgreifen, die darunterliegende Beschleuniger-Architekturen nutzen können, aber dem Entwickler durch eine High-Level-API vereinfachten Zugriff auf die Funktionalitäten bieten. Da derartige Frameworks häufig für eine Vielzahl von Beschleuniger-Architekturen implementiert sind, die API jedoch einheitlich ist, haben Programme, die derartige APIs verwenden, generell eine hohe Portabilität. Beispielsweise bietet das *OpenVX*-Framework [33], das auf Basis von OpenCL implementiert ist, eine portable Implementierung für maschinelles Sehen (engl. *Computer Vision* (CV)) und neuronale Netze (engl. *Convolutional Neural Network* (CNN)). *Caffe* [34] und *Tensorflow* [35] bieten eine einheitliche API für Entwickler, während architekturspezifisch beschleunigte Implementierungen für viele Plattformen verfügbar sind. Weiterhin sind applikationsspezifische DSPs, wie z.B. Bildprozessoren (engl. ISP) häufig in Frameworks, wie Video4Linux2 [36] programmierbar oder DSPs für das maschinelle Lernen (engl. *Machine Learning*) (*AI-Beschleuniger*) haben Interfaces zu Caffe oder Tensorflow. Da diese Prozessoren allerdings nicht vollkommen frei programmierbar sind, sondern Beschleuniger-DSPs für spezifische Anwendungsklassen darstellen, werden sie im Kontext dieser Arbeit nicht betrachtet.

2.5 Paralleles Laufzeitverhalten

Während Multicore-Prozessoren eine immer größer werdende Anzahl an Prozessorkernen besitzen, wächst der Aufwand für Verwaltung und Programmierung dieser großen Anzahl von heterogenen, parallelen Geräte. Darum ist es eine komplexe Aufgabe, eine Software zu entwickeln, die die Lücke zwischen theoretisch verfügbarer und tatsächlich erreichter Performance minimiert [6, 37]. Um eine sog. Performance-portable Implementierung eines Algorithmus zu erreichen, muss eine Implementierung auf jeder Plattform dynamisch die derzeit vorhandenen Ressourcen ausnutzen, um damit die bestmögliche Performance zu erreichen. Dazu muss eine gute Skalierbarkeit über eine flexible Anzahl an verwendbaren Prozessorkernen erreicht werden. Werden zunächst homogene

Multicore-Prozessoren betrachtet, muss jedem Prozessor-Kern dieselbe Menge und Größe an Arbeitspaketen zugewiesen werden. Dabei können unterschiedliche Einflussfaktoren, wie Anforderungen der Software, Eigenschaften der Hardware, oder speziell die Kombination bestimmter Software- und Hardware-Eigenschaften, einen großen Einfluss auf das Laufzeit- und Skalierungsverhalten haben. Auch Effekte, die erst explizit durch die Parallelisierung eingebracht werden, sowie zusätzlicher Overhead beeinflussen die parallele Performance maßgeblich.

2.5.1 Skalierbarkeit und Portabilität

Software kann potentiell leicht zwischen Plattformen portiert werden und weist durch die einfache Wartbarkeit eine deutlich höhere Flexibilität als dedizierte Hardware auf. Allerdings muss eine *Performance-portable* Software, die auch nach einer Migration eine möglichst effiziente Auslastung der Ressourcen aufweist, dynamisch über alle Recheneinheiten skalieren, was bereits während der Partitionierung der Arbeit berücksichtigt werden muss. Nicht alle der diskutierten Parallelisierungsschemata und nicht alle algorithmisch auftretenden Nebenläufigkeiten sind hinreichend feingranular aufteilbar und damit beliebig skalierbar. Selbst, wenn theoretisch eine dynamische und feingranulare Partitionierung der Arbeit vorliegt, ist eine Anpassung des Architekturmappings an die aktuelle Plattform nicht trivial. So kann in manchen Implementierungen, die beispielsweise auf einer einfachen Datenparallelität basieren (z.B. in der Bildverarbeitung), leicht in fast beliebig viele voneinander unabhängige Abschnitte unterteilt und zwischen den Prozessoren verteilt werden. Andere Parallelisierungs-Strategien benötigen dagegen ein komplizierteres Mapping der gegebenen Berechnungsschritte auf die darunterliegende Architektur. Eine derartige Aufteilung auf (heterogene) Prozessorkerne und eventuelle Beschleuniger kann entweder automatisch im Programm durchgeführt werden, oder aber vom Anwender bzw. Entwickler durch eine Konfigurationsschnittstelle konfiguriert werden. Werden reguläre x86-Prozessoren verwendet, kann zumindest derselbe Sourcecode wiederverwendet werden, was allerdings nicht die Skalierbarkeit über alle Kerne garantiert. Heterogene Architekturen benötigen für eine dynamische Aufteilung derselben algorithmischen Tasks häufig redundante Implementierungen derselben Funktion (je eine pro Beschleuniger bzw. pro Beschleunigersprache, wie CUDA oder OpenCL) für die jeweiligen Recheneinheiten und Beschleuniger.

Viele Aspekte können sich als performancelimitierende Faktoren einer Applikatuion herausstellen, wie beispielsweise fehlende Datenlokalität, eine unangemessene Speicherzugriffsstrategie oder redundante Speicherzugriffe von

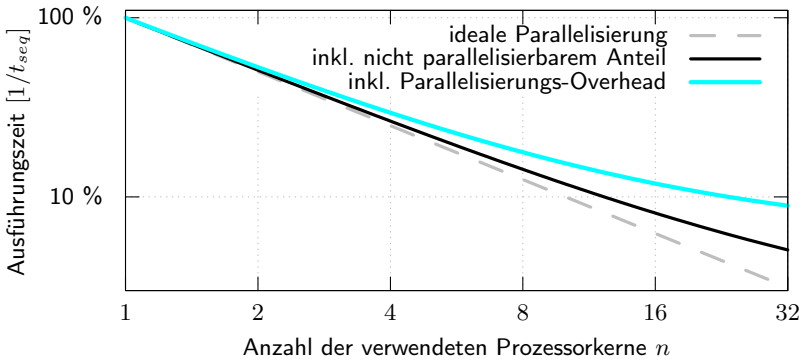


Abbildung 2.3: Ideales gegen beispielhaftes reales Skalierungsverhalten einer parallelen Implementierung. Wird ein sequentieller Anteil berücksichtigt, der nicht parallelisiert werden kann, entsteht eine Lücke zwischen dem idealen und dem tatsächlichen Skalierungsverlauf. Parallelisierungs-Overhead, wie er oben beschrieben wurde, vergrößert diese Lücke und limitiert dadurch die maximal erreichbare parallele Performance der Applikation.

mehreren Prozessorkernen. Allerdings kann selbst durch den zusätzlich benötigten Software-Code und Verwaltungsaufwand bereits ein drastischer Performance-Verlust verursacht werden – beispielsweise durch Overhead für Task-Verwaltung und eine zu feingranulare Aufteilung der Tasks.

Wie in Abbildung 2.3 dargestellt, verläuft das ideale Skalierungsverhalten in Form einer Geraden innerhalb einer logarithmischen Darstellung beider Achsen. Davon abweichend und durch zusätzlichen Parallelisierungs-Overhead beeinflusst, verläuft das realistische Beispiel, das pro zusätzlich verwendetem Prozessorkern eine immer größere Abweichung zur Ideallinie verzeichnet. Auch ein nicht parallelisierbarer Anteil bleibt bestehen und limitiert nach Amdahl's Gesetz [38] den maximalen Speedup. Dieser nicht parallelisierbarer Anteil wird im weiteren Verlauf nicht weiter betachtet. Stattdessen beziehen sich alle Metriken und Charakterisierungen auf die algorithmische Last und den Overhead innerhalb einer parallelen Sektion.

2.5.2 Einflussfaktoren und Skalierungsmetriken

Ausgehend von der sequentiellen Ausführungszeit unter Verwendung von nur einem Prozessorkern kann zunächst bei einer naiven Annahme davon ausgegangen werden, dass sich die Ausführungszeit bei Verdopplung der verwendeten Prozessorkerne – also der Verdopplung der verfügbaren Rechenleistung – auch

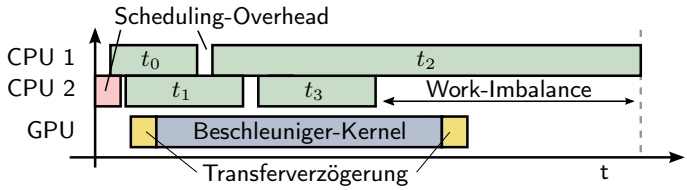


Abbildung 2.4: Schematische Darstellung einer typischen parallelen Task-Verteilung mit Scheduling-Overhead, Work-Imbalance und Offload eines GPU-Kernels. Der Scheduling-Overhead umfasst Zeiten für Task-Erzeugung, Verteilung, Switching und Overhead für die Synchronisation einer parallelen Sektion. Zusätzlicher Overhead für das Verwalten der GPU innerhalb eines CPU-Tasks ist nicht explizit dargestellt.

die Ausführungszeit halbiert. Eine typische Situation einer Task-Verteilung, die verdeutlicht, wodurch sich eine reale parallele Ausführung von der idealen unterscheidet, ist in Abbildung 2.4 skizziert. Metriken, die das reelle Verhalten einer Arbeitsaufteilung auf parallelen Prozessoren beschreiben, werden in Tabelle 2.5 zusammengefasst und im Folgenden erläutert.

Wird ein zunächst sequentiell ausgeführter Task mit Ausführungszeit $t_0(1)$ parallel auf n Prozessorkernen ausgeführt, kann die Summe aller resultierenden parallelen Tasks $t_{task}(n)$ die sequentielle Ausführungszeit deutlich überschreiten. Da nebenläufige Ausführungen durch Wartezeiten von synchronisierenden Mutex-Locks, kommunikationsbedingten Verzögerungen oder Bandbreiten-Limitierungen beeinflusst sein können, wird der Anstieg der Ausführungszeit als *Redundanz* $R(n)$ beschrieben. Je nach Definition wird die Redundanz um die Lock-Wartezeit $t_{lock}(n)$ bereinigt dargestellt, sodass eine Redundanz $R > 1$ maßgeblich auf Bus- und Speicher-Effekte zurückzuführen ist. Ungleiche

Tabelle 2.5: Zusammenfassung der Skalierungsmetriken.

Sequentielle Ausführung [s] :	$t_0(1)$	(ohne sequentiellen Anteil)
Parallele Ausführungszeit [s] :	$t(n) = \frac{t_0(1) \cdot R(n)}{n \cdot U(n)}$,	n: Anzahl der Kerne,
Redundanz [-] :	$R(n) = \frac{t_{task}(n) - t_{lock}(n)}{t_0(1)}$	(ohne Overhead: $R(n) = 1$)
Utilization [-] :	$U(n) = \frac{t_{task}(n)}{t(n) \cdot n} = 1 - Ov(n)$	
Summe aller Tasks [s] :	$t_{task}(n) = \sum_i t_i(n)$,	i: Task-ID
Overhead [-] :	$Ov(n) = \frac{t_{wi}(n) + t_{sched}(n)}{t(n) \cdot n}$	
Work-Imbalance [s] :	$t_{wi}(n)$	
Lock-Delay [s] :	$t_{lock}(n)$	
Scheduling-Overhead [s] :	$t_{sched}(n)$	

Tabelle 2.6: Einflussfaktoren auf das parallele Laufzeitverhalten.

Parallelisierungs-Strategie-Einflüsse:	Einflüsse, die besonders durch die gewählte Parallelisierungs-Strategie und z.B. der angewendeten Datenlokalität bedingt werden bzw. besonders durch Kombination mit speziellen Plattformeigenschaften auftreten z.B. Cache-Größe. Bedingt besonders: **Work-Imbalance**, **Lock-Wartezeiten**
Parallelisierungs-Framework-Einflüsse:	Einflüsse, die durch das Parallelisierungs-Framework bedingt werden und beispielsweise durch Scheduling-Strategien involviert werden. Je nach Plattformeigenschaften, Anforderungen der Parallelisierungs-Strategie und Bottlenecks können sich Effekte aber auch unterschiedlich stark ausprägen. Bedingt besonders: **Scheduling-Overhead**
Hardware-Eigenschaften / Software-Anforderungen (interferierend):	Durch die Kombination der Software-Anforderungen mit den gegebenen Hardware-Eigenschaften entstehende Bottleneck-Situationen, die bei paralleler Ausführung zu Verzögerungen führen: z.B. durch die Speicherhierarchie und Cache-Performance, Speicherbandbreiten, Kommunikationen zwischen Prozessorkernen oder Instruktionssatz-Eigenschaften. Bedingt besonders: **Redundanz**

Arbeitsverteilungen (*Work-Imbalance* $wi(n)$) können ebenfalls mit Wartezeiten der Prozessorkerne beim nächsten Synchronisationspunkt einer parallelen Sektion verbunden sein. *Scheduling-Overhead* ($t_{sched}(n)$) kann beim Verteilen von Tasks und Task-Teams vom Scheduling-Framework involviert werden, wobei sich die Einflüsse der jeweiligen Frameworks stark unterscheiden. Dabei wird in dieser Arbeit zusätzlicher Overhead für das Erzeugen, Verteilen und Synchronisieren (Terminieren) der Tasks separat betrachtet. Die *Utilization* $U(n)$ beschreibt den prozentualen Anteil der verfügbaren CPU-Zeit, die für die tatsächliche Ausführung von algorithmischen Tasks verwendet werden kann. Dabei ist die Utilization durch den Gesamt-Overhead $Ov(n)$ bestehend aus Scheduling-Overhead und Work-Imbalance limitiert.

Im Allgemeinen können die Aspekte, die das parallele Laufzeitverhalten beeinflussen in die folgenden drei Klassen eingeteilt werden: (1) Einflüsse der Parallelisierungs-Strategie, (2) Einflüsse des Parallelisierungs-Frameworks und (3) Limitierungen durch die Eigenschaften der Hardware. Tabelle 2.6 fasst diese zusammen und nennt die jeweils beeinflussten Metriken.

Zusätzliche Sprünge der Ausführungszeit im Skalierungsverlauf einer parallelen Anwendung können entstehen, sobald Grenzen entweder für *Hyperthreads* (HT) (duplizierte Registersätze derselben ALU, die dem Betriebssystem als separate Prozessorkerne repräsentiert werden) oder aber *Non-Uniform Memory Access* (NUMA)-Nodes (separate Prozessor-Chips mit jeweils eigenen Speicherbank-Anbindungen) überschritten werden. Im Fall von Applikationen, die eine gleichartige Operation in allen parallelen Tasks ausführen (z.B. bei Datenparallelität), kann die Verwendung von Hyperthreads zu einer Verlangsamung der Prozesse führen, da dieselben Instruktionseinheiten der ALU

nicht von mehreren Threads gleichzeitig verwendet werden können. Durch die blockierten CPU-Ressourcen müssen dann Instruktionen anderer Threads warten. Werden für alle Threads jeweils Hyperthreads verwendet, verlangsamen sich alle Threads gleichermaßen, wodurch lediglich die Redundanz steigt. Ist nur bei einem Teil der verwendeten Prozessoren jeweils die ALU durch mehrere Hyperthreads belegt, kann dies zusätzlich zu Work-Imbalance führen, da einige Tasks verlangsamt ausgeführt werden, anderen dagegen die volle ALU-Performance zur Ausführung zur Verfügung steht. Wird eine parallele Implementierung auf einer Plattform ausgeführt, die aus mehreren NUMA-Nodes besteht, wobei Threads dieser Anwendung auf jeweils unterschiedlichen NUMA-Nodes ausgeführt werden, erfolgt der Zugriff auf Speicherbereiche unsymmetrisch. Da jedem Prozessor eine eigene Speicherbank zugewiesen ist, aber zusammenhängende Speicherbereiche einer Applikation immer nur in einem der Speicherbänke angelegt werden, müssen Daten zwischen den Chips transferiert werden. Durch die größere Entfernung mancher Threads der Anwendung zum angefragten Speicher, kommt es zu längeren Speicherzugriffverzögerungen, sowie zur Arbeitsungleichverteilung zwischen den Tasks bei unsymmetrischer Aufteilung, wie beim Hyperthreading.

Typischer Weise belegt eine Applikation immer die Prozessoren mit den tiefsten vom Linux-Kernel repräsentierten Prozessor-IDs, wobei die Hyperthreads eines Prozessorkerns jeweils mit einer eigenen ID als separater Prozessorkern repräsentiert werden. Wie in Abbildung 2.5 dargestellt, werden vom Linux-Kernel der Reihenfolge nach zunächst alle Prozessorkerne eines NUMA-Nodes durch nur einen Thread pro Prozessorkern repräsentiert. Anschließend setzt sich die Reihenfolge der IDs auf den zweiten bzw. die weiteren NUMA-Nodes fort, bevor weitere Hyperthreads des ersten NUMA-Nodes aufgelistet werden. Dementsprechend belegen auch parallele Software-Implementierungen, die keine explizite Zuweisung der Threads zu Prozessorkernen berücksichtigen, zunächst jeweils nur einen Hyperthread pro Kern, überschreiten dann die Grenze der NUMA-Nodes und erst danach werden Hyperthreads belegt.

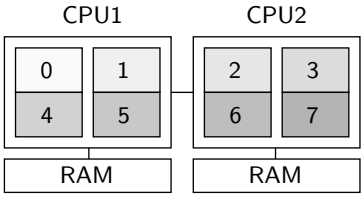


Abbildung 2.5: Vom Linux-Kernel repräsentierte Prozessorreihenfolge auf einem NUMA-System (zwei NUMA-Nodes) mit Hyperthreads (je NUMA-Node zwei Cores mit je zwei Threads).

2.6 Profiling

Zur Analyse des parallelen Laufzeitverhaltens werden Profiling-Tools benötigt, die charakteristische Eigenschaften darstellen. Eine solche Analyse ist bei der Entwicklung paralleler Software unabdingbar, um Nebenläufigkeitsfehler aufzudecken, wie *Data-Races*, *Lock-Contentions*, *Work-Imbalances* oder Bottlenecks zu identifizieren (siehe [24, 25, 26, 27, 28] für Parallelisierungsfehler). Dabei sind besonders parallele Profiles der Task-Ausführungszeiten und der jeweils ausführenden Prozessoren (Traces), sowie zugehörige *Performance-Counter* (z.B. Cache-Zugriffe auf den *Last-Level Cache* (LLC) (engl. *LLC-Load*) oder das Fehlen eines angeforderten Datums im Level-1-Data-Cache (engl. *Data Cache Miss* (DCM)) von Bedeutung. Viele herkömmliche Profiler unterstützen jedoch keine parallele Ausführung, weshalb beispielsweise das Standard-Tool der GNU-Compiler-Collection *GProf* nicht verwendet werden kann (Als Standard-Profiler aber trotzdem unten aufgeführt wird). Umfangreiche und mächtige Profiling-Werkzeuge sind besonders für eingebettete Systeme mit Echtzeitanforderungen, wie Fahrerassistenzsysteme, nicht verfügbar.

Instrumentierende Profiling-Tools basieren auf regelmäßigen Inspektionsaufrufen (*Sampling-Interrupt*), die den aktuellen Programmstatus auslesen, was sowohl zu Ungenauigkeiten, als auch zu hohem Overhead führen kann. Dadurch kann das Extrahieren von Traces und der Performance-Counter-Metriken zu Verzerrungen des parallelen Laufzeitverhaltens führen und bei nicht synchronisierten parallelen Regionen die Deterministik des Ausführungsverhaltens gefährden. Beinhaltet ein Programm beispielsweise Data-Races, die bei regulärer Ausführung versteckt bleiben, können diese durch Verschiebung der Laufzeiten zweier Threads gegeneinander provoziert werden. Es kann aber genauso gut zu dem weitaus verheerenderen Fall kommen, dass Nebenläufigkeitsfehler durch das Profiling gerade nicht auftreten und so unentdeckt bleiben. Allgemein ist die nicht-deterministische Natur der parallelen Ausführung für die Analyse problematisch, da manche Fehlertypen nur in einem Bruchteil der Ausführungen auftreten. Eine Analyse startet dann zunächst damit, den Fehler konsistent bei jedem Durchlauf sichtbar zu machen, um anschließend die Fehlerquelle identifizieren zu können. Im Folgenden werden zunächst gängige Profiler vorgestellt.

GProf ist als Teil der GNU Compiler Collection der vermutlich portabelste Profiler der durch Cross-Kompilieren auf jeder GCC-unterstützten Plattform eingesetzt werden kann [39]. Allerdings gibt GProf unzulängliche Profiler-Informationen zurück, wenn ein Programm parallel auf mehreren Prozessoren ausgeführt wird. Auch dieser instrumentierende Profiler basiert auf regelmäßigen Interrupts, die den Programmablauf für Status-Inspektionen unterbrechen.

Dies führt bei geringer Sampling-Rate zu mangelhaftem Detailgrad der Profiles, bei zu hoher Sampling-Rate allerdings zu unverhältnismäßig hohem Overhead.

VTune Amplifier ist ein kommerzielles Profiling-Tool von Intel, was die Extraktion von Tracing- und Performance-Counter-Informationen von parallelen Prozessen auf Intel-Architekturen erlaubt [40]. Dieses Tool bietet ebenfalls eine grafische Oberfläche zur Inspektion des Laufzeitverhaltens aller verfügbaren Prozessoren, Threads und Tasks, als auch eine aspektfokussierte Analyse. Allerdings ist die grafische Oberfläche nicht intuitiv bedienbar, benötigt großen Einarbeitungsaufwand, um repräsentative Analysen zu erstellen, und tendiert zu regelmäßigen Programmabstürzen. Auch dieses Tool basiert auf instrumentierenden Sampling-Interrupts, die das parallele Laufzeitverhalten deformieren.

Dimemas ist Teil der BSC Performance Tools [41]. Dieses Tool bietet eine Simulation des Laufzeitverhaltens von Implementierungen, die Message-Passing-Interfaces (wie MPI) verwenden. Die extrahierten Tracings können mit dem Tool Paraver visualisiert werden. Basierend auf den im Software-Code verwendeten Parallelisierungs- und Kommunikationsaufrufen, werden Profiling-Anker in das eigentliche Programm integriert, was regelmäßige Interrupts unnötig macht, die Messpräzision erhöht und den Overhead minimiert.

Ein Profiling für heterogene Plattformen zu erstellen stellt eine weitere Herausforderung dar, da nicht nur Tracings für GPP und verteilte Tasks erzeugt werden müssen, sondern auch Warte-, Transfer- und Ausführungszeiten der Beschleuniger-Kernel berücksichtigt werden müssen. Außerdem lassen sich beispielsweise Performance-Counter-Metriken nicht zwangsläufig zwischen Prozessoren vergleichen, da diese Counter unterschiedlich implementiert werden. Beispielsweise erzeugt ein LLC-Miss auf manchen Plattformen immer auch gleichzeitig einen L1-Miss, auf anderen dagegen nicht. Im folgenden Kapitel wird für die vorgestellte Abstraktionsschicht zusätzlich eine Erweiterung für ein leistungsstarkes und automatisiertes Profiling sowohl für homogene, als auch für heterogene Systeme vorgestellt.

3 Abstraktionsschicht

Es existiert eine große Lücke, zwischen der theoretisch verfügbaren Rechenleistung von parallelen (heterogenen) Plattformen und der durch Programmierung erreichbaren Performance. Darum wird in diesem Kapitel eine Abstraktion zur Vereinheitlichung (*Unifikation*) der parallelen Programmierung vorgestellt, um die Programmierung zu vereinfachen. Die Abstraktionsschicht (engl. *Middleware*) *Modular Parallelization Abstraction Layer* (MPAL) befindet sich, wie in Abbildung 3.1 dargestellt, zwischen den Software-Schichten der eigentlichen Applikationsimplementierung und den Parallelisierungs-Frameworks. Durch die Abstraktion der eigentlichen Parallelisierungsaufrufe (*Spawn*) wird die parallele Implementierung unabhängig vom darunterliegenden Parallelisierungs-Framework und damit portabler. Unterstützt werden unterschiedliche Frameworks, je nach Verfügbarkeit auf der aktuell verwendeten Plattform, mit denen MPAL

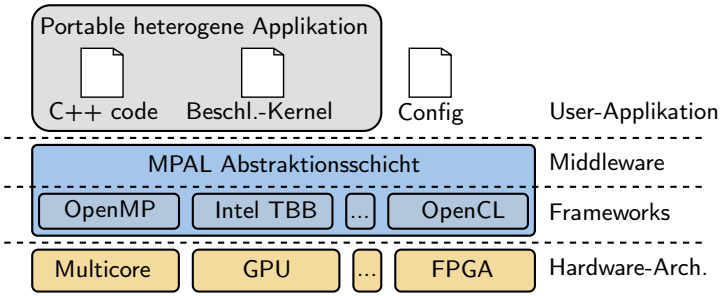


Abbildung 3.1: MPAL-Abstraktionsschicht innerhalb der Software-Schichten zur Parallelisierung von portabler Software. Architektur-Mapping und Parallelisierungseigenschaften werden in der Konfigurationsdatei konfiguriert. Es werden homogene und heterogene CPU-Cluster sowie Beschleuniger (z.B. GPU, DSP oder FPGA durch HLS) unterstützt.

die tatsächliche Parallelisierung realisiert, wie zum Beispiel OpenMP [10], TBB [17] oder CilkPlus [18]. Eine Schnittstelle zu einer Konfigurationsdatei ermöglicht selbst nach dem Compilieren eine Konfiguration des verwendeten Frameworks, sowie weitere Eigenschaften der Parallelität (zum Beispiel Anzahl der verwendeten Prozessoren oder Core-Pinning). Durch den zusätzlichen Support von Beschleuniger-Frameworks (*Offload*-Frameworks), wie CUDA [42] oder OpenCL [8], und das Kapseln des eigentlichen Offload-Aufrufs in der Abstraktionsschicht wird selbst die Programmierung von heterogenen Plattformen ermöglicht. Dadurch, dass alle Parallelisierungsereignisse (engl. *Event*) in dieser Abstraktionsschicht ausgeführt werden, werden auch weitere Funktionalitäten ermöglicht, wie zum Beispiel paralleles Profiling (*Tracing*). MPAL bietet für Software-Implementierungen, die mit dieser API parallelisiert wurden, die folgenden Vorteile:

- Framework-unabhängige portable Parallelisierung
- Hohe Skalierbarkeit und konfigurierbares Architektur-Mapping
- Automatisiertes *Inline*-Profiling mit minimalem Overhead
- Extraktion der Laufzeiteigenschaften zur Bottleneck-Identifikation

Mit den hier beschriebenen Eigenschaften stellt MPAL die Basis für die nachfolgenden Themen der Charakterisierung und Performance-Prädiktion, die in den folgenden Kapiteln 4 und 5 beschrieben werden.

3.1 Unifikationsmethoden

Da die bereits eingeführten Parallelisierungs-Strategien meist entweder nur die Programmierung von GPP oder von Beschleunigern beherrschen, aber nicht beide Typen gleichzeitig, werden Methoden zur Vereinheitlichung (Unifikation) benötigt. In Abbildung 3.2 ist der herkömmliche Entwicklungsablauf der Programmierung von heterogenen Plattformen unter Verwendung von separaten Frameworks für Host- (Kontroll-), Multicore- und Beschleuniger-Code gezeigt. Die genannten Methoden bieten ebenfalls keine direkte Lösung für eine einfache und flexible Konfiguration des Architektur-Mappings an, die allerdings für eine portable Implementierung für heterogene Systeme erforderlich ist. Ansätze zur Vereinheitlichung der Programmierung von homogenen und heterogenen Plattformen lassen sich grob in die drei folgenden Ebenen unterteilen, die im Folgenden näher erläutert werden:

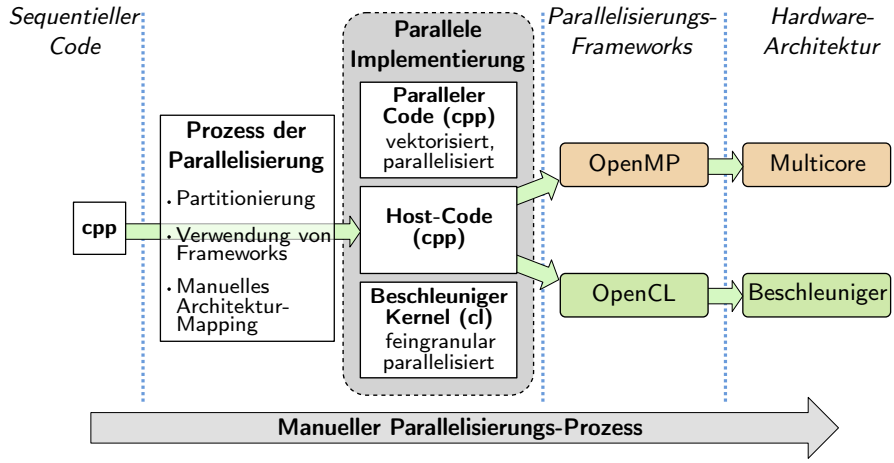


Abbildung 3.2: Beispiel eines manuellen Entwicklungsprozesses einer parallelen Applikation für heterogene Architekturen: Der Entwickler ist verantwortlich für die Partitionierung des Software-Codes, die Instrumentalisierung der Parallelisierungs- und Offload-Frameworks und die Integration von Architektur-Mapping Methoden.

1. Vereinheitlichung des Software-Codes, sodass nur eine Version derselben Funktion für alle Recheneinheiten benötigt wird.
2. Vereinheitlichung der Speicherzugriffe durch gemeinsame kohärente Speicheradressbereiche der einzelnen Recheneinheiten
3. Vereinheitlichtes Architektur-Mapping durch konfigurierbares oder automatisiertes Mapping der Tasks

3.1.1 Single-Source Programmierung

Eine Anforderung der Vereinheitlichung der parallelen Programmierung ist die Zusammenführung des Software-Codes, mit dem eine flexible heterogene parallele Ausführung beschrieben werden kann. Darum bieten manche Frameworks, wie z.B. OpenMP, die Möglichkeit eine einzige Code-Basis, also eine einzige Implementierung einer algorithmischen Funktion für die Programmierung aller verfügbaren heterogenen Recheneinheiten zu verwenden. Dazu ist es nötig den Software-Code (z.B. in C++ geschrieben) für alle Architekturtypen (GPP und Beschleuniger) zu übersetzen. Einige Ansätze, die die Vereinheitlichung der heterogenen parallelen Programmierung verfolgen, sind in Tabelle 3.1 zusammengefasst.

Tabelle 3.1: Frameworks zur heterogenen Single-Source Programmierung.

Tool	Merkmale	Portabilitätsaspekte
OpenMP	◦ Die *target*- und *map*-Direktiven von OpenMP (seit Version 4.0) erlauben das Auslagern von Programm-Sektionen zu Beschleunigern ◦ Ermöglicht die Programmierung von Beschleunigern in C++ ◦ Erweiterter Beschleuniger-Support durch Verwendung von OpenCL und CUDA in Version 5.0 geplant	+ Ermöglicht Programmierung heterogener Plattformen mit einer einzigen Source-Code-Basis + Keine weiteren Kontrollstrukturen oder API-Aufrufe nötig − Kontrolle und Geräte-Unterstützung muss in Compiler integriert werden − Derzeit nur Unterstützung für Intel Xeon Phi und NVIDIA PTX
OpenCL	◦ Verwendung von C-Syntax (C++ seit Version 2.1) für Beschreibung von Beschleuniger-Kerneln ◦ Basiert auf herstellerspezifischen Treibern und Compilern ◦ API-Aufrufe für Beschleunigerkontrolle, Speicherverwaltung(inkl. DMA) und Profiling ◦ Unterstützung für diverse Gerätetypen, wie CPUs, GPUs, DSPs oder FPGAs (via HLS) ◦ Manuelle Anpassung an Beschleunigerarchitektur nötig	+ Deutlich portabler als CUDA durch mehr unterstützte Architekturen − Benötigt separates C/C++ Host-Programm das den Offload kontrolliert + Durch Verwendung von SyCL [43] können OpenCL-Kernel direkt in C++ beschrieben werden − Geringere Performance auf NVIDIA-GPUs, wegen CUDA-Spezialisierung − Geringe Performance auf GPP aufgrund der feingranularen Partitionierung [19].
NVIDIA Thrust	◦ Erlaubt den expliziten Aufruf von relevanten parallelen Algorithmen und Datenstrukturen auf NVIDIA GPUs direkt aus regulärem C++ Code ◦ Syntaktisch identisch zu C++ STL Datenstrukturen, wie *vectors*, *iterator*, *for_each loops*, *scan* oder *sort* ◦ Mittlerweile mit CUDA ausgeliefert	+ Einfache Verwendung von relevanten parallelen Algorithmen und Datenstrukturen auf GPUs − Keine generelle Programmierung: GPU-Verwendung limitiert auf angebotene Strukturen − Nur Unterstützung für NVIDIA GPUs

Wie in Abbildung 3.3 dargestellt, erlaubt beispielsweise OpenMP das Compilieren von Sektionen des C++-Codes zu Beschleuniger-Kerneln. Ab OpenMP 4.0 kann dazu die *target*- und *map*-Direktive verwendet werden. Allerdings ist die Anzahl der von OpenMP unterstützten Plattformen bislang stark limitiert. OpenCL unterstützt die größte Vielfalt von Beschleunigern von diversen Herstellern und unterschiedlichen Architekturtypen. Die Verwendung von OpenCL erfordert allerdings das Vorhandensein einer Implementierung des Compute-Kernels in OpenCL und einen Host-Code in C/C++, der die vorhandenen Beschleuniger, Kernel und Task-Warteschlangen (engl. *Queues*) verwaltet. Ein C++-Frontend für OpenCL ist mit SyCL [43] gegeben, wodurch es möglich ist, OpenCL-fähige Beschleuniger (außer FPGAs) und CPUs direkt in C++ zu programmieren. Ein ähnliches Frontend existiert auch für NVIDIA GPUs mit Thrust [44], wodurch Entwickler NVIDIA GPUs direkt in C++ programmieren können. SyCL und Thrust bieten hierbei spezielle Funktionalitäten, wie Datenstrukturen und Algorithmen an, die von der C++-STL abgeleitet sind, dann aber auf dem Beschleuniger verarbeitet werden.

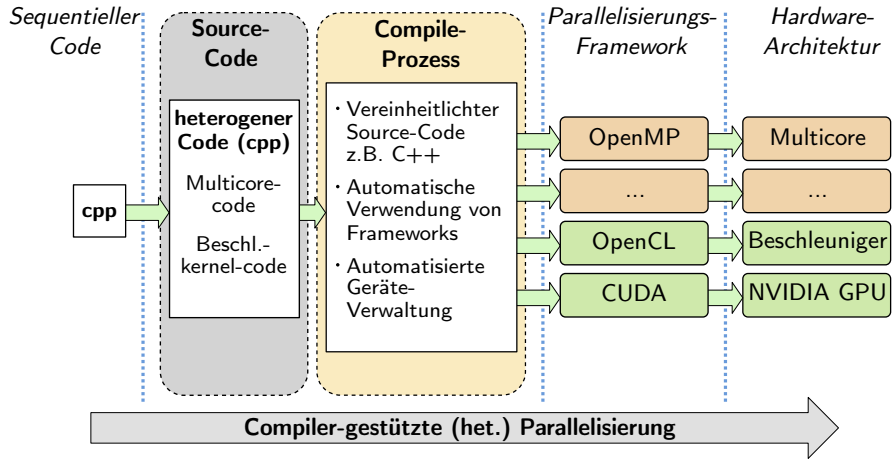


Abbildung 3.3: Beispiel einer parallelen Implementierung bestehend aus einem einzigen Code (z.B. in C++), welcher sowohl Host-, Multicore-, als auch Beschleuniger-Kernel enthält: Der Compiler verwaltet automatisch die Beschleuniger-Hardware und das Speicher-Mapping.

3.1.2 Kohärenter Shared-Memory

Host-CPU und Bechleuniger verwenden üblicherweise separate Speicher bzw. Speicherbereiche. Der Anforderung der Vereinheitlichung der parallelen Programmierung von allen Recheneinheiten einer heterogenen Plattform kann auch mit einem einheitlich erreichbaren geteilten Speicher (engl. *Shared Virtual-Memory*) begegnet werden. Um den Speichertransfer innerhalb einer Applikation zu erleichtern und den Speicherverwaltungsaufwand zu minimieren, werden Kohärenzmethoden für die üblicherweise getrennten Speicherbereiche heterogener Recheneinheiten eingeführt, um auch zwischen CPU und Beschleuniger kohärenten Shared Virtual-Memory verwenden zu können. Bei echter Kohärenz des virtuellen Speichers kann dieselbe Speicheradresse (engl. *Pointer*) von mehreren Recheneinheiten gleichzeitig verwendet werden, ohne dass Caching- oder Buffer-Effekte den Speicherinhalt inkonsistent machen. Dabei kann Kohärenz auf unterschiedlichen Ebenen realisiert werden: von der realen Kohärenzimplementierung in Hardware übergreifend über alle vorhandenen Recheneinheiten, bis zu Software-emulierter Kohärenz in der Gerätetreiber-Ebene der Beschleunigereinheiten.

Heterogeneous System Architecture (HSA) Foundation: Die HSA Foundation [12] verfolgt den Ansatz, die zugrundeliegende Topologie der Speicherverwaltungs-Hardware zu vereinheitlichen und dabei alle heterogenen Recheneinheiten

auf einem *System on Chip* (SoC) denselben kohärenten *zero-copy* Speicher erreichen zu lassen. Somit werden keine expliziten Mapping- bzw. Kopieroperationen im Host-Code benötigt, sondern alle Recheneinheiten können direkt dieselben Host-Pointer zu virtuellen Speicheradressen (auch gleichzeitig) verwenden, was die Verwendung derselben *Memory Management Unit* (MMU) voraussetzt. Des Weiteren implementiert eine HSA-fähige Plattform, wenn das vollständige HSA-Profile umgesetzt wurde, auch Hardware-Einheiten, die eine Task-Verwaltung übernehmen, wodurch weitere Verwaltungsprozesse aus dem Host-Programm eliminiert und weiterer Overhead eingespart werden kann. Parallele Programme, beispielsweise in C++ geschrieben, werden zu einer zu Assembler ähnlichen Zwischensprache *HSA Intermediate Language* (HSAIL) übersetzt (ähnlich, wie die für OpenCL verwendete *Standard for Portable Intermediate Representation - V* (SPIR-V) [45]) und anschließend durch herstellereigene *Finalizer* zum finalen Beschleuniger-Kernel übersetzt. Übersetzte Programm-Binaries, die für eine HSA-Plattform übersetzt wurden, beinhalten zusätzlich zum CPU-Code, auch HSAIL-Blöcke in der Binary-Datei und sind daher kompatibel mit allen verfügbaren Architekturen, was sich positiv auf die Portabilität auswirkt. Ein finales Programm besteht damit nicht aus einer Vielzahl von bereits kompilierten Dateien, sondern aus einer einzigen Binärdatei. Dabei können viele Sprachen zu HSAIL übersetzt werden. Basierend auf der HSA-Runtime wurde in [46] eine Implementierung von *Offload-Policies* für die C++17 PSTL (STL-Implementierung von Intel [47]) vorgestellt. Dabei wurde der zugrundeliegende kohärente HSA-Speicher zwischen CPU und GPU einer AMD Carrizo *Accelerated Processing Unit* (APU) ausgenutzt. In dieser Arbeit wurde gezeigt, dass mithilfe der HSA-Runtime die Ausführungsrichtlinien (engl. *Execution-Policy*) der parallelen Algorithmen aus C++17 um eine *offload*-Policy erweitert werden kann. Der Ansatz verwendet die darunterliegende HSA-Runtime, um den Entwicklungsprozess einer heterogenen Applikation für eine HSA-fähige heterogene Beschleuniger-Hardware in purem C++-Code zu programmieren.

NVIDIA unified memory: Im Gegensatz zur Hardware im HSA-Ansatz besitzen GPU-Beschleuniger im *NVIDIA unified memory*-Ansatz [11] normalerweise jeweils einen separaten Hauptspeicher. Während Entwickler zuvor dedizierte Kopiervorgänge triggern mussten, oder zumindest das Mapping von Speicherbereichen angestoßen werden musste, erlaubt der vereinheitlichte Speicherzugriff von NVIDIA unified memory systemweiten Zugriff auf den virtuellen Speicher, was statt in der Hardware, in der Gerätetreiber-Schicht realisiert wird. Dies erlaubt das automatische, seitenweise Kopieren von Daten zur GPU, sobald die Daten dort angefordert werden, sowie eine entsprechende Zugriffskontrolle auf gemappte Bereiche, wenn auf diese von der CPU zugegriffen wird, um eine Kohärenz zu garantieren. Somit wird der GPU-Speicher ähnlich zu ei-

nem privaten Cache, der alle gängigen Caching-Mechanismen, wie das Spiegeln von Speicherseiten, Speicherseiten-Verdrängungsmechanismen, und Prefetching aufweist. Da dieser im Gerätetreiber verankerte Ansatz ausschließlich auf Software basiert, wird eine Speicherkohärenz auch für Recheneinheiten ermöglicht, die über eine größere Distanz verbunden sind (z.B. PCIe). NVIDIA unified memory fokussiert sich dabei allerdings hauptsächlich auf GPUs, während die HSA auch in der Lage ist, andere Beschleunigertypen anzusprechen.

3.1.3 Middleware-Layer

Auf der einen Seite repräsentieren die diskutierten Programmiermodelle bereits eine Reihe von leistungsstarken Ansätzen und zeigen Trends der aktuellen Forschung und Entwicklung. Auf der anderen Seite können die meisten dieser Ansätze entweder nur ein kleines Set von Architekturen programmieren (entweder CPUs oder Beschleuniger), oder die Ansätze befinden sich derzeit noch nicht in einem Entwicklungsstadium, in dem eine breite Masse von Architekturen effizient verwendet werden kann (z.B. OpenMP). Tatsächlich limitiert nicht nur das fehlende Vorhandensein von geeigneten Programmierwerkzeugen immer noch die eigentliche Portabilität einer parallelen und heterogenen Implementierung, sondern auch architekturspezifische Hardware-Eigenschaften und variierende Performance-Charakteristiken (z.B. Speicherbandbreite) begrenzen die Portierbarkeit. Speicherkohärenzmechanismen sind vielversprechende Ansätze zur Vereinheitlichung der Programmierung von heterogenen Architekturen, sind aber derzeit nur für eine kleine Anzahl von Plattformen verfügbar. Weiterhin stellen diese Ansätze bislang keine Lösung für die Suche nach geeigneten Strategien für das Architektur-Mapping dar. Darum fokussiert ein weiterer Forschungszweig die Abstraktion der eigentlichen parallelen Programmierung durch den Einsatz einer neuen API für die Parallelisierung. In der darunterliegenden Runtime-Bibliothek wird die Parallelität weiterhin durch Verwendung von Standard-Frameworks realisiert, wie in Abbildung 3.4 dargestellt. Im Unterschied zu den in Abbildung 3.3 dargestellten Vereinheitlichungsansätzen benötigen Entwickler weiterhin mehrere Implementierungen der algorithmischen Funktionen jeweils für jeden verwendeten Beschleuniger. Allerdings wird die Verwaltung der Plattform, wie Beschleunigerverwaltung, Speicherverwaltung und Architektur-Mapping innerhalb der Abstraktionsschicht übernommen.

Diese Arbeit stellt die Abstraktionsschicht MPAL vor (detaillierte Beschreibung ab Kapitel 3.2) [50], die die Portabilität von Software-Implementierungen durch bessere Skalierbarkeit, Unabhängigkeit der Implementierung vom angewendeten Parallelisierungs-Framework, sowie ein konfigurierbares Architektur-

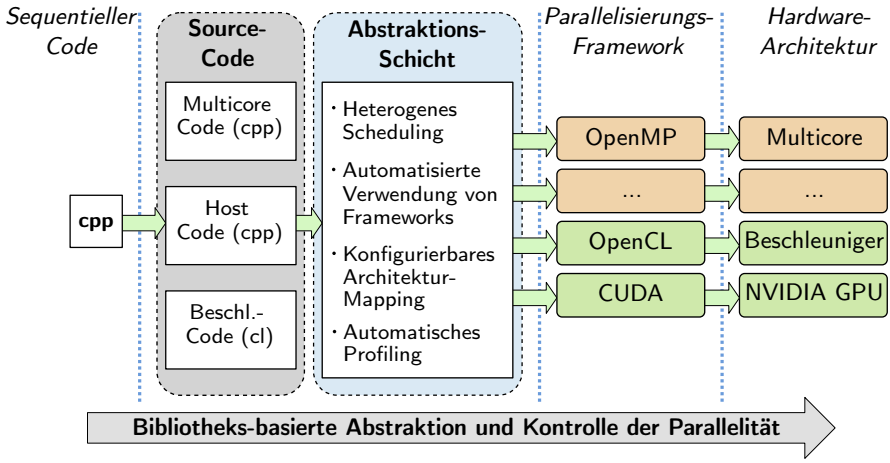


Abbildung 3.4: Beispiel einer parallelen Implementierung unter Verwendung der API einer Abstraktionsschicht, wie die in dieser Arbeit vorgestellte Abstraktionsschicht MPAL, die auf Parallelisierungs-Frameworks zurückgreift: Der Entwickler muss weiterhin architekturspezifische Implementierungen bereitstellen, hat jedoch einfache Konfigurationsmöglichkeiten für das Architektur-Mapping.

Mapping, verbessert. Die Abstrakionsschicht wird zunächst mit zwei weiteren ähnlichen Ansätzen verglichen, bevor Implementierungsdetails und deren Anwendung beschrieben werden. MPAL bietet Unterstützung für heterogene Multicore-Prozessoren durch Verwendung von Parallelisierungs-Frameworks für die Shared-Memory-Parallelisierung (z.B. OpenMP oder TBB), sowie Unterstützung für unterschiedliche Beschleuniger durch die Verwendung von Offload-Frameworks, wie z.B. OpenCL oder CUDA. Die Möglichkeit Tasks bzw. Task-Teams auf bestimmte Prozessorkerne (oder Cluster in heterogenen Systemen) zu fixieren bzw. assoziieren (engl. *Pinning*) wurde in MPAL ebenfalls unter Verwendung von regulären Tasking-Frameworks implementiert [51] und ermöglicht innerhalb gepinnter Task-Teams weitere Unterverzweigungen (engl. *nested* Spawns) und dadurch auch komplexe Verzweigungsstrukturen. Durch die Verwendung von OpenCL als Backend für die Verwaltung von Beschleunigern innerhalb der Abstraktionsschicht, können die erstellten Kernel durch HLS auch auf FPGAs ausgeführt werden, während OpenCL den Datentransfer innerhalb von MPAL realisiert. Für eine vereinfachte Speicherverwaltung kapselt MPAL die Beschleuniger-Buffer in einer generischen hybriden Datenstruktur, die automatisch zwischen Host und Beschleuniger *gemappt* wird, wenn keine Speicherkohärenz zwischen den Geräten vorliegt. Beide Abstraktions-APIs, sowohl die Shared-Memory-Parallelisierung, als auch Beschleuniger-Offloads,

Tabelle 3.2: Abstraktionsschichten, die eine neue Parallelisierungs-API bieten.

Tool	Merkmale	Portabilitätsaspekte
MPAL	◦ Bietet API zur Multicore-Programmierung z.B. via OpenMP und Beschleuniger-Offloads z.B. OpenCL ◦ Fokus: konfigurierbares Architektur-Mapping, Framework-Unabhängigkeit und Extraktion von Charakteristiken (z.B. Timing, Performance-Counter) ◦ Bietet einige Profiling- und Analyse-Tools zur Evaluation der Parallelisierung und Bottleneck-Identifikation	+ MPAL vereinfacht die Entwicklung von heterogenen Parallelisierungen, Arbeitsaufteilung und Charakteristikanalyse bei flexiblen und portierbaren Implementierungen + Unterstützt Mapping auf het. CPUs, GPUs, DSPs und FPGAs (via HLS) + Profiling-Parameter und Architektur-Mapping können nach dem Übersetzen des Programms angepasst werden
Kokkos	◦ Bietet parallele Algorithmen, wie *parallel_for* oder *reduce*, die auch Low-Level Frameworks verwenden [48] ◦ Multicore-Code in C++-Lambdas, separate GPU-kernel in CUDA ◦ Architekturspezifische Speicherstrukturen mit individuellen *Views* auf Speicherbereiche ◦ Profiling kann durch angebotene Profiling-Methoden verwendet werden	+ Automatische architekturspezifische Speicherverwaltung bietet optimierte Performance-Portabilität über unterschiedliche Speicherzugriffsmuster − Derzeit wird nur OpenMP für Multicores und CUDA für NVIDIA GPU Beschleuniger unterstützt
StarPU	◦ Fokus: automatisches Scheduling von Tasks (*Codelets*) über Recheneinheiten von heterogenen Plattformen [49] ◦ Codelets definieren abstrakte algorithmische Tasks mit unterschiedlichen architekturspezifischen Implementierungen, Input und Output zur Definition von Task-Abhängigkeiten	+ Intelligente Scheduling-Methoden für heterogene Plattformen + Verwendung von OpenCL und CUDA für breites Beschleuniger-Spektrum + Bietet sowohl eine C++-API, als auch Compiler-Direktiven

beinhalten die volle MPAL-Funktionalität, die durch virtuelle Methoden in die API integriert werden, sodass mit nur wenig Aufwand durch Überladen der virtuellen Methoden weitere Frameworks integriert werden können.

Tabelle 3.2 fasst die Eigenschaften von MPAL, sowie der beiden Abstraktionsschichten Kokkos [48] und StarPU [49] zusammen, die alle die Verbesserung der Performance-Portabilität von heterogenen Implementierungen fokussieren. Alle vorgestellten Abstraktionsschichten benötigen weiterhin separate Implementierungen für jeden verwendeten Architekturtyp mit derselben algorithmischen Funktion für die parallele Ausführung. Dies ist auf den Fokus auf Vereinheitlichung der Speicherverwaltung, Geräteverwaltung und Architektur-Mapping, anstatt auf Vereinheitlichung des Software-Codes zurückzuführen.

Während StarPU das Mapping von Tasks auf heterogenen Architekturen mit heterogenen CPU-Clustern und Beschleunigern fokussiert, ist es möglich, Beschleuniger sowohl mit OpenCL, als auch CUDA anzusprechen. Kokkos implementiert dagegen Speicherverwaltungsmethoden, unterstützt allerdings nur CUDA für NVIDIA-GPUs als Beschleunigerarchitekturen. Wie beide vorher genannten Frameworks, benötigt MPAL weiterhin eine separate Implementie-

rung von Beschleuniger-Kerneln, abstrahiert allerdings ebenfalls Beschleuniger- und Datenverwaltung (Host-Code). Da alle bislang vorgestellten Vereinheitlichungsmethoden zunächst nur jeweils einen Aspekt adressieren (Software-Code Unifikation, Virtueller Shared-Memory, oder Abstraktion), ist besonders die Verbindung von mehreren Ansätzen vielversprechend. Im Folgenden wird der interne Aufbau und die Anwendung der MPAL-Abstraktionsschicht näher erläutert, wobei auf die folgenden Punkte genauer eingegangen wird:

- Programmierung von homogenen und heterogenen Multicore-Clustern
- Programmierung von Beschleunigern, wie GPUs, DSPs oder FPGAs
- Konfiguration der Parallelität und des Architektur-Mappings
- Profiling zur Bottleneck-Analyse und Extraktion des Laufzeitverhaltens

3.2 Abstraktionsschicht MPAL

Aufgrund der unterschiedlichen Implementierung der Task-Verwaltung der verschiedenen Parallelisierungs-Frameworks, die jeweils für unterschiedliche Anwendungen und Plattformen eine variierende Eignung aufweisen, bietet sich an, den eigentlichen *Spawn*-Aufruf und dazugehörige Funktionalitäten zu abstrahieren. Basierend auf einer einheitlichen API, die Zugriff auf darunterliegende Frameworks erlaubt, wird im Folgenden die Implementierung der Abstraktionsschicht MPAL zur Reduktion der Entwicklungszeit paralleler Software vorgestellt. Die Funktionalitäten von MPAL, sowie Komponenten, API und Anwendungsbeispiele werden dargestellt, wobei drei unterschiedliche Entwicklungsperspektiven betrachtet werden.

Die Implementierung von MPAL basiert auf virtuellen Methoden, die zunächst als funktionslose Platzhalter implementiert werden und als Schnittstelle (engl. *Interface*) fungieren, um Parallelität Framework-unabhängig im Software-Code auszudrücken. Während Vererbungen der Klassen die eigentliche Funktionalität (z.B. Spawn) in Framework-spezifischer Syntax der Platzhaltermethoden implementieren, kann die Auswahl des verwendeten Frameworks (z.B. OpenMP, TBB) selbst zur Laufzeit geändert werden. Des Weiteren erlaubt das Kapseln der eigentlichen Parallelisierungsaufrufe das zusätzliche Einfügen von Messmethoden (*Probes*) bei relevanten Parallelisierungsereignissen, die das parallele Programm auf Parallelisierungsebene analysieren. Durch das Triggern der Messungen an exakten Events innerhalb des Software-Codes (hier *Inline*-Profiling) kann ein herkömmliches Profiling vermieden werden, das durch Instrumentieren des Codes mit regelmäßigem Sampling zur Analyse des

Programmstatus großen Overhead erzeugt. In dieser Arbeit (Kapitel 3.2.2) wird eine Methode des automatischen Profilings für das parallele Laufzeitverhalten aus algorithmischer Sicht vorgestellt, die eine aussagekräftige Analyse der Nebenläufigkeiten und algorithmischen Charakteristiken ermöglicht. Daraus können laufzeitbeeinflussende Parameter, Hotspots und auch der Einfluss des Parallelisierungs-Frameworks extrahiert werden. Ebenfalls werden in MPAL zusätzliche Werkzeuge implementiert, die für eine effiziente parallele Implementierung benötigt werden, wie eine Cache-effiziente Datenstruktur oder Core-Pinning-Methoden, die parallele Threads bzw. Tasks auf Cores oder Core-Cluster fixieren.

3.2.1 Homogene Parallelisierung

In diesem Abschnitt werden die drei Entwicklungsperspektiven betrachtet, die auch in Abbildung 3.5 dargestellt sind.

- *Das Applikationsinterface* steht dem Entwickler von paralleler Software und Anwender der MPAL Abstraktionsschicht zur Verfügung um Nebenläufigkeiten im Software-Code auszudrücken
- *Die Framework-Spezialisierungen*, die auch ohne Kenntnis über den internen Aufbau dynamisch erweitert werden können
- *Die interne Implementierung* der Abstraktionsschicht, die alle virtuellen Methoden, die Realisierung der Messinfrastruktur, und weitere für

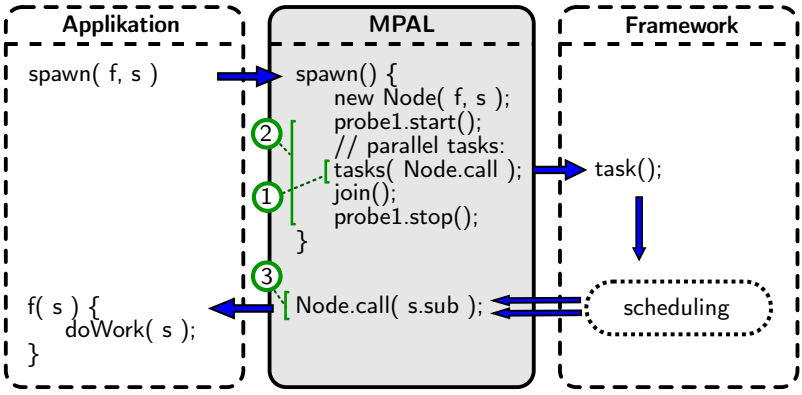


Abbildung 3.5: Entwicklungsperspektiven der Abstraktionsschicht: Parallelisierungs-API, interne Struktur und Framework-Spezialisierung. Die markierten Profiling-Probes (1), (2), (3) werden in Abschnitt 3.2.2 beschrieben.

die parallele Ausführung nötige Software-Strukturen beinhaltet

Applikationsinterface: Die erste Perspektive beschreibt das Applikationsinterface, das verwendet wird, um eine parallele Sektion zu beschreiben, die im Wesentlichen aus der Anzahl der zu erstellenden Tasks und einer Funktion besteht (z.B. Member-Funktion einer Klasse), die von jedem Task ausgeführt werden soll. Für die Parallelisierung von kamerabasierten Fahrerassistenzalgorithmen wird beispielsweise zunächst eine blockbasierte Parallelisierung angeboten, die eine kontinuierliche Sequenz (z.B. Pixel) in einheitliche Teilsequenzen (bei Datenparallelität Unterabschnitte des Datenbereichs) unterteilt, dieselbe Task-Funktion für jede Teilsequenz parallel ausführt und nach jedem Spawn implizit synchronisiert. Diese aufzuteilende Sequenz wird hier als $\vec{s}$ dargestellt und besteht aus dem Tripel $\vec{s} = \{start, end, step\}$ und kann beispielsweise für die datenparallele Prozessierung eines Bildes durch horizontales Teilen der Daten die Menge der Bildzeilen repräsentieren: $\{0, height - 1, 1\}$. Zur Erhaltung der Datenlokalität wird die Taskfunktion nicht für jedes Element der Sektion separat ausgeführt, sondern jedem Task eine zu bearbeitende Teilsequenz als Argument übergeben. Die Bearbeitung der Elemente erfolgt in einer Schleife innerhalb der jeweiligen Tasks. Eine Sequenz kann hierbei sowohl aus ganzzahligen oder Gleitkommazahlen bestehen, sowie positive oder negative Laufrichtung und Begrenzungen aufweisen.

Die nebenläufige Ausführung von Sequenzen, die andernfalls in einer rein sequentiell iterativen Schleife abgearbeitet würden, ist ein häufiges Parallelisierungsschema (*parallel_for*) wobei alle anderen Schemata davon abgeleitet werden können. Auch eingenistete Tasks sind möglich (ein Spawn von Sub-Tasks innerhalb eines bereits parallel ausgeführten Tasks), da der eigentliche Spawn auf einem Tasking-Modell basiert. Der Funktionsrumpf (engl. *Header*) der Task-Funktion, die von jedem Task auszuführen ist und dem Spawn-Aufruf als Argument übergeben wird, muss einer vorgegebenen Form entsprechen, damit eine Teilsequenz übergeben werden kann (siehe Algorithmus 3.1). Die abstrakte Klasse *ParallelWorker* kapselt alle Funktionalitäten, Interfaces und Konfigurationen für die Parallelisierung. Somit ist für einen Spawn, unabhängig vom darunterliegenden Framework, nur ein einziger Aufruf notwendig, wie in Abbildung 3.5 auf der linken Seite zu sehen ist.

Framework-Spezialisierungen: Die zweite Perspektive beschreibt die Spezialisierungen der zunächst virtuellen Methoden der *ParallelWorker*-Klasse zu einem spezifischen Parallelisierungs-Framework. Die Funktionsaufrufe der eigentlichen Task-Funktion, sowie die Berechnung der Teilsequenzen sind vollständig im *ParallelWorker* implementiert. Demnach muss eine Spezialisierung zu einem weiteren Parallelisierungs-Framework lediglich den eigentlichen Aufruf eines parallelen Tasks beinhalten und darin auf die im *ParallelWorker*

```
// wird von jedem Task ausgefuehrt (parallele Ausfuehrung):
void AlgorithmClass::taskFunc( start, end, step ) {
        for( i = start; i <= end; i += step ) doWork( i );
}
```

(a) Implementierung einer Member-Funktion, die von jedem parallelen Task aufgerufen werden soll, wobei der Funktionsheader durch MPAL vorgeschrieben ist.

```
// ausgefuehrt vom sequentiellen Hauptprogramm (z.B. main):
AlgorithmClass algoInstance;
MPAL::ParallelWorker worker( numThreads );
worker.spawn( s⃗, algoInstance::taskFunc, numTasks );
```

(b) Spawn-Aufruf aus einer sequentiellen Ausführung mithilfe des *ParallelWorkers* und unter Angabe der oben definierten Task-Funktion.

```
// sequentieller oder bereits verzweigter paralleler Programmabschnitt
MPAL::ParallelWorker worker( numThreads );
worker.spawn( s⃗, [&]( start, stop, step ){
                    for( i = start; i <= end; i += step ) doWork( i );
        }, numTasks );
```

(c) Alternativer Spawn-Aufruf mit einer Lambda-Funktion (anonyme Funktion, die als Objekt repräsentiert und dadurch als Funktionsargument übergeben werden kann), die denselben Funktions--Header besitzen muss, wie eine parallelisierte Member-Funktion.

Algorithmus 3.1: Grundlegende Software-Struktur für einen Parallelisierungsaufruf (*Spawn*) mit MPAL für eine homogene Task-Parallelisierung.

enthaltenen Funktionalitäten zurückgreifen, wie im mittleren Teil in Abbildung 3.5 gezeigt. Da alle auf ein Framework spezialisierten Klassen kompatibel zu den virtuellen Interface-Methoden sind, sind innerhalb der Applikationen keine Code-Veränderungen nötig, um ein weiteres Framework mit MPAL zu verwenden, während das aktuell verwendete Framework selbst zur Laufzeit noch gewechselt werden kann.

Interne Implementierung: Die dritte Perspektive beschreibt die reinen virtuellen Interface-Methoden und die interne Parameterverwaltung, die die fundamentalen Funktionalitäten von MPAL realisieren, wie in Abbildung 3.5 dargestellt. Das Applikationsinterface des Spawn-Aufrufs erzeugt initial für jeden separaten Spawn ein *TeamNode*-Objekt, das alle nötigen Informationen eines Task-Teams beinhaltet (Sequenz, Task-Funktion und Anzahl der auszuführenden Tasks). Jeder Spawn stellt einen weiteren Knoten innerhalb des eventuell weit verzweigten Parallelisierungsgraphen dar und besitzt einen eigenen *TeamNode*, der alle Parameter beinhaltet, die ein Task-Team charak-

terisieren. Da somit die volle Funktionalität der Task-Verwaltung innerhalb des *TeamNodes* implementiert ist, ist für eine Framework-Spezialisierung nicht mehr nötig, als die Verwendung der MPAL-internen Funktionalitäten, um den vollen Funktionsumfang zu gewährleisten. Neben der Übergabe von Klassen mit den zugehörigen Member-Funktionen, die als Task-Funktion verwendet werden sollen, können auch Lambda-Funktionen (anonyme Funktionen, die als Objekt repräsentiert und dadurch als Funktionsargument übergeben werden können) als Task-Funktionen übergeben werden. Durch Lambda-Funktionen können Parallelisierungskonstrukte ähnlich zu einem regulären *parallel_for* realisiert werden.

3.2.2 Profiling-Probes

Wie in Abbildung 3.5 dargestellt, sind alle relevanten Ereignisse der parallelen Programmausführung in MPAL gekapselt. Dies erlaubt eine Integration von *Probes* (engl. für Messspitzen – hier: generische Software-Funktionen, die Messungen vornehmen können) direkt in den Software-Code (*Inline-Profiling*), um die Funktionalität der MPAL-Abstraktionsschicht um automatische Messmethoden zu erweitern. Dazu werden generische Probes jeweils am Anfang und am Ende einer parallelen Sektion (1), der Framework-spezifischen Latenz für das Anlegen eines Task-Teams (2) und des Aufrufs der jeweils parallel ausgeführten Tasks (3) eingefügt. Darüber hinaus werden Probes um frameworkspezifische Mutex-Locks herum eingefügt, um den Overhead für die Synchronisation und Lock-Wartezeiten analysieren zu können.

Wie in [50] beschrieben, wird durch das Integrieren der Profiling-Probes nur minimaler Overhead erzeugt. Dadurch bietet das direkte Einbringen von Probes in den Software-Code einen wesentlichen Vorteil gegenüber regulären parallelen Profilern, die auf zusätzlichen Debug-Informationen im compilierten Programm basieren und dadurch einen relevanten und teilweise Laufzeitverzerrenden Overhead erzeugen. Tabelle 3.3 zeigt die parallele Ausführungszeit des SGM-Algorithmus [13] (Tasking-Optimierung mit 10 konsekutiven MPAL-Spawns) auf einem Xeon E5-2680 GPP und einem Xeon Phi Manycore (*Knights Corner*). Hier wurde der von MPAL eingebrachte Overhead gemessen, der durch die Abstraktion der Spawn-Aufrufe und durch die Profiling-Probes entsteht. Durch den minimalen Overhead wird, im Gegensatz zu den in Abschnitt 2.6

Tabelle 3.3: Einfluss von Abstraktion und Profiling auf SGM-Algorithmus.

Plattform	Threads	Ausführung	Abstraktion	Profiling
Xeon E5-2680	32	8,452 ms	+30 µs (0,4 %)	+25 µs (0,3 %)
Xeon Phi	122	22,448 ms	+160 µs (0,7 %)	–

beschriebenen Sampling-Profilern, auch das parallele Laufzeitverhalten nur minimal verzerrt und eine Analyse stark erleichtert oder erst ermöglicht.

Um den Overhead zu minimieren, werden Compiler-Optimierungen benutzt, um den flexiblen Wechsel von einem Durchlauf mit automatischer Messung aller parallelen Ereignisse zu einem Ausführungsmodus ohne jeglichen Overhead zu ermöglichen. Probes werden ebenfalls mit C++-Templates formuliert und können zu jeder Art von Messung spezialisiert werden (z.B. Zeitmessung oder Performance-Counter). So, wie Spezialisierungen der zunächst abstrakten Klasse *ParallelWorker* für unterschiedliche Parallelisierungs-Frameworks erstellt werden können, können auch Spezialisierungen der Probes erstellt und als neue Profiling-Funktionalität zu MPAL hinzugefügt werden. Unter Verwendung des Compilers und dessen Linker-Optimierungen kann bei Einfügen von leeren Probes (Funktionsaufrufe zu leeren Funktionen) selbst der Funktionsaufruf eliminiert werden. Dass selbst der Funktionsaufruf zu einer eigentlich leeren Funktion vom Linker unterdrückt wird, kann leicht durch Analyse des assemblierten Programms verifiziert werden. Es existieren Probes für die folgenden Ausführungsmodi, wobei der Modus wie die Auswahl des Parallelisierungs-Frameworks selbst zur Laufzeit geändert werden kann.

1. Tracing (Messung der Zeit und CPU-ID)
2. Messung der Performance-Counter (z.B. Cache-Misses)
3. Ausgeschaltetes Profiling für eine Ausführung ohne Overhead

3.2.3 Heterogene Prozessor-Cluster

Im folgenden Abschnitt wird eine Erweiterung der MPAL-Abstraktionsschicht vorgestellt, die es ermöglicht, nicht nur Parallelisierungen für homogene Multicore-Systeme zu bieten, sondern auch heterogene CPU-Cluster und Beschleuniger, wie z.B. GPUs, oder FPGAs durch HLS, verwalten zu können. Dadurch wird die Portabilität von Implementierungen, die mit MPAL parallelisiert wurden, deutlich erhöht und sie können bei gleichzeitigem Erhalt der Skalierbarkeit auf verschiedene homogene und heterogene parallele Systeme portiert werden. Ebenso, wie für homogene Multicores, werden im entsprechenden Ausführungsmodus für alle heterogenen Recheneinheiten Profiles extrahiert, wodurch auch bei heterogenen Implementierungen beispielsweise eine Bottleneck-Analyse möglich ist. Durch diese Erweiterung bietet MPAL für die Entwicklung von paralleler Software die folgenden Vorteile:

1. Vereinfachungen für den Entwicklungsprozess von parallelisierter Software für homogene und heterogene CPU-Cluster und Beschleuniger,

wie z.B. GPUs, DSPs und FPGAs (via HLS)

2. Flexible, skalierbare und portable Implementierungen für einfache Migration von parallelisierter Software zwischen Plattformen
3. Konfigurierbares Architektur-Mapping zur dynamischen Anpassung an individuelle Plattformeigenschaften, wie Kommunikationstopologie, Rechenleistung oder Speicherbandbreiten
4. Einfache und automatisierte Analyse des Laufzeitverhaltens für Entwurfsraumexploration, Bottleneck-Analysen und Evaluation verschiedener Parallelisierungsstrategien

Diese Eigenschaften bilden die Grundsteine für die im nächsten Kapitel vorgestellte Extraktion von Charakteristiken des parallelen Laufzeitverhaltens, auf dessen Basis anschließend die Performance-Prädiktion aufgebaut wird.

Für die Programmierung von homogenen Multicore-Plattformen, ist zunächst ein Spawn-Aufruf vorgesehen, der eine homogen aufzuteilende Datensequenz und eine auf die Teilsequenzen anzuwendende Task-Funktion als Argumente erwartet. Die Aufteilung einer Datensequenz über heterogene CPU-Cluster erfordert allerdings eine Arbeitsverteilung, die der Rechenleistung der jeweiligen Prozessoren entspricht, um Arbeitsungleichverteilungen zu vermeiden. Darüber hinaus erfordern viele Anwendungen nicht nur eine einzige Funktion, die über mehrere Prozessoren aufgeteilt wird, sondern benötigen die nebenläufige Ausführung von mehreren Funktionen auf jeweils unterschiedlichen CPU-Clustern. Dabei kann eine Zuweisung eines Threads zu einem Core (*Core-Pinning*) über Funktionen im Linux-Kernel realisiert werden, die einem Thread eine Assoziativität zu einem Core zuweisen. Von einem gepinnten Hauptprozess abgespaltene Threads erben anschließend die Core-Assoziativität des Hauptprozesses. Da jedoch der Linux-Kernel die Threads auch im späteren Prozessverlauf untereinander vertauschen könnte, reicht es bei heterogenen Plattformen nicht aus, lediglich eine Assoziativität dem Hauptprozess mitzugeben. Stattdessen muss jeder Thread einem separaten Prozessorkern zugewiesen werden, damit nicht heterogen zugewiesene Teilsequenzen auf ungeeignete Kerne verteilt werden. Bei der *big.LITTLE*-Architektur, wie im Samsung Exynos 5 Octa 5422 verwendet, wird beispielsweise ein Thread-Scheduler angewendet, der automatisch rechenintensive Threads von den stromsparenden *LITTLE*-Kernen auf die High-Performance *big*-Kerne migriert ohne dabei die Auslastung dieser Kerne zu berücksichtigen [52]. Besitzt ein Programm beispielsweise acht Threads auf die jeweils Arbeit (heterogen) verteilt werden soll, um beide CPU-Cluster verwenden zu können, würde der Scheduler alle Threads auf die big-Kerne migrieren. Dadurch muss der gesamte Rechenaufwand auf den big-Kernen berechnet werden, während den LITTLE-Kernen

(a) Automatisches big.LITTLE Scheduling der Threads.

(b) Mit MPAL individuell zu Cores assoziierte Threads.

Abbildung 3.6: Der kernelinterne Thread-Scheduler für die big.LITTLE-Architektur wird alle Threads auf die big-Kerne migrieren, was die gleichzeitige Auslastung der LITTLE-Kerne verhindert.

keine Arbeit zugewiesen ist, wie in Abbildung 3.6 dargestellt. Im Folgenden wird eine Implementierung vorgestellt, die auf Basis der in MPAL enthaltenen Parallelisierungsmethoden ein individuelles Core-Pinning für jeden Thread erzielt.

Thread-individuelles Core-Pinning: Die Verwendung von Tasks, die vom Parallelisierungs-Framework verwaltetet und intern zu Threads zugeordnet werden, erlaubt häufig keinen direkten Zugriff auf die darunterliegenden Threads und deren IDs (Beispiel: Intel TBB). Da in MPAL allerdings nur Funktionen innerhalb solcher Framework-Tasks aufgerufen werden können, wurde das folgende Schema implementiert, bei dem vor Beginn der eigentlichen algorithmischen Arbeit jeweils ein Task pro Thread gestartet wird, der das Pinning von jeweils einem Thread übernimmt. Das unten beschriebene Schema ist ebenfalls in Abbildung 3.7 bzw. in Algorithmus 3.2 dargestellt. Erst nach vollständiger Zuordnung aller Threads (Terminierung und Synchronisation des nicht-algorithmischen Task-Teams) werden funktionale Tasks für die algorithmische Arbeit erzeugt. Anschließend wählt jeder Thread die ihm zugeordneten

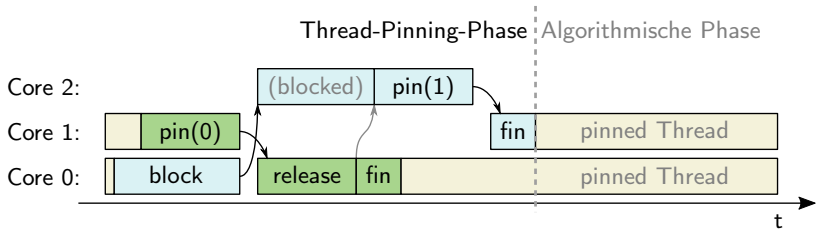


Abbildung 3.7: Thread-individuelles Core-Pinning mithilfe von Framework-Tasks. block: Thread wird blockiert, pin: Thread wird zum assoziierten Core gepinnt, release: release/freigeben der anderen Threads.

```
// sicherstellen, dass jeder Thread genau einen Task ausfuehrt
if( not lastThread() )
    lockArray[ taskID ].lock(); // Thread blockieren        t_x.bl()
// eigentliches Core-Pinning
setCoreAffinity( cores[ taskID ] );                         t_x.pin()
// release/naechsten Thread wieder freigeben
if( nextTask = findNextTask( lockArray ) )
    lockArray[ nextTask ].unlock();                         t_x.rel(t_y)
```

Algorithmus 3.2: Thread-individuelles Core-Pinning, welches jedem Thread nur einen Prozessorkern zur Ausführung assoziiert.

Tasks aus einem zentralen Task-Vorrat (engl. *Task-Pool*) zur Abarbeitung aus.

Da ein nicht-funktionaler Task, der den ausführenden Thread zu einem Core zuweist, nur eine sehr kurze Rechenzeit aufweist, ist es möglich, dass ein Thread mehrere dieser *Pinning*-Tasks nacheinander ausführt (Pinning hin und her), während andere Threads dagegen keinen dieser Tasks ausführen (kein Core-Pinning). Um dieses Problem zu umgehen und ein stabiles Core-Pinning zu gewährleisten, wird jeder Thread, der einen Pinning-Task ausführt, zunächst blockiert und erst freigegeben, wenn alle Pinning-Tasks gestartet wurden (jeder auf einem anderen Thread). Dazu wird ein Array aus Mutex-Locks angelegt, wobei alle Mutex-Locks bereits zu Beginn gelockt sind. Zunächst wird von jedem ausgeführten Task das Mutex im Array mit der ID, die der Thread-ID entspricht, ein weiteres Mal gelockt, wodurch der ausführende Thread blockiert wird, bis das Mutex von einem anderen Thread wieder freigegeben wird. Das Blockieren der Threads verhindert, dass weitere Tasks zu diesem Thread zugewiesen werden und stattdessen die noch nicht ausgeführten Tasks auf freien Threads ausgeführt werden, sodass jeder Thread genau einen Pinning-Task ausführt. Lediglich der letzte Task lockt kein Mutex, sondern beginnt mit dem Pinning von zunächst seinem eigenen Thread, um anschließend den nächsten blockierten Thread freizugeben. In einer Kette werden so alle Threads wieder freigegeben, sodass die ausgeführten Tasks der Reihe nach das Pinning durchführen. Wurde der letzte Task beendet, wird damit auch die Pinning-Sektion terminiert.

Task-zu-Thread Pinning: Im Fall einer heterogenen Ausführung definiert ein *Arbeitspaket* eine spezifische Teilsequenz, die von einer Funktion func_x (der eigentlichen algorithmischen Arbeit) bearbeitet werden soll. Um die korrekte Zuweisung von Arbeitspaketen, die sowohl im Arbeitsaufwand, als auch in der zu bearbeitenden Funktion variieren können, auf die zugehörigen Prozessorkerne zu garantieren, wurde das folgende Schema implementiert.

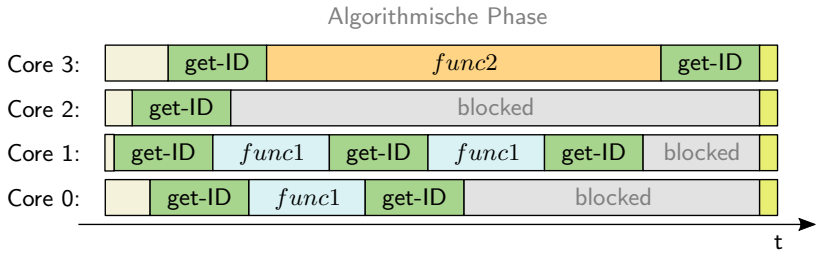


Abbildung 3.8: Bearbeitung von Tasks aus globalem Task-Pool durch gepinnte Threads und anschließendem Blockieren der Threads, wenn keine weitere Arbeit zugewiesen ist. Die Funktion *func1* besitzt drei Tasks und ist Core 0 und 1 zugewiesen. *func2* besitzt nur einem Task und ist Core 3 zugewiesen. Zu Core 2 wurde ein Arbeiter-Thread, wird aber gleich blockiert, da keine Tasks zu Core 2 assoziiert sind.

Dabei wird zunächst angenommen, dass jeder Thread einem einzelnen Prozessorkern zugewiesen wurde und es wird, wie bereits oben beschrieben zunächst ein Mutex-Array angelegt, wobei jedes Mutex-Lock bereits initial gelockt ist. Außerdem wird für jedes Arbeitspaket ein *Framework-Task* erzeugt und zusätzlich ein weiterer Framework-Task pro Thread, der den Thread blockieren kann, falls dort derzeit keine Arbeitspakete bearbeitet werden sollen. Um zu verhindern, dass Framework-Tasks auf Prozessorkerne portiert werden, zu denen momentan keine Arbeitspakete zugewiesen sind, werden diese Threads von den zusätzlichen Framework-Tasks mithilfe des Mutex-Arrays blockiert. Wie in Abbildung 3.8 gezeigt, greift jeder ausgeführte Framework-Task zunächst auf einen globalen Arbeitspaketvorrat (engl. *Task-Pool*) zu und führt das nächste dem ausführenden Thread zugeordnete Arbeitspaket aus. Ist dem jeweiligen Prozessorkern ein Arbeitspaket zugeordnet, das sich im Task-Pool befindet, wird dieses ausgeführt. Andernfalls wird dieser Thread blockiert, um sicherzustellen, dass kein weiterer Framework-Tasks diesem Thread zugewiesen

```
// Finden einer zu diesem Core zugewiesenen Funktion
if( func = getFunc( coreID ) ) { // Funktion. gefunden
    if( not func.popAndExecTask() )
        blockThread(); // auf andere Threads warten
} else // keine weitere Funktion diesem Core zugewiesen
    blockThread(); // auf andere Threads warten
```

Algorithmus 3.3: Verwaltung von Arbeitspaketen (*Work-Packets*) für die Ausführung auf heterogenen CPU-Clustern (Task-Pinning).

wird. In Algorithmus 3.3 wird das Blockieren eines Threads mit der Funktion *blockThread* ausgeführt. Diese Funktion prüft außerdem, ob alle anderen Threads ebenfalls bereits blockiert sind, was auf das Ende einer parallelen Sektion hindeutet, und beendet anschließend alle blockierenden Tasks, wodurch das Task-Team beendet und synchronisiert wird.

Heterogene Sub-Spawns: Die oben beschriebenen Mechanismen ermöglichen ein Scheduling für heterogene CPUs auf *Root-Spawn-Ebene* (erste parallele Ebene), allerdings benötigen moderne Parallelisierungen häufig tiefere (eingenistete) Verzweigungsebenen. Dazu beherrscht MPAL eine Methodik, die erlaubt, dass jeder Task, der einem bestimmten Core-Cluster zugewiesen ist, weitere *Sub-Tasks* erzeugen kann, die folglich auf dasselbe Core-Cluster oder eine Untermenge davon limitiert sind. Um Parallelisierungsfehler zu vermeiden (z.B. Deadlocks oder Race-Conditions), warten alle übergeordneten Tasks immer auf die vollständige Terminierung aller abgeleiteten Sub-Tasks. Daraus folgt eine Unterscheidung der Tasks innerhalb des Task-Pools, wobei sich die Arbeitspakete jeweils durch ihre Verzweigungsebene unterscheiden. Da allerdings eingenistete Framework-Tasks nur in den wenigsten Frameworks auf Cores oder Core-Cluster begrenzt werden können, ist das Erzeugen von echten Framework-Tasks hier nicht angebracht und es wird die folgende Methode angewendet. Wie oben beschrieben, werden Arbeitspakete von Root-Spawns mit entsprechender Core-Assoziativität und der zu bearbeitenden Teilsequenz in einem zentralen Task-Pool verwaltet. Um auf Core-Cluster gepinnte Sub-Spawns zu realisieren, wird zusätzlich zum Task-Pool für Root-Spawns ein weiterer Pool für Arbeitspakete der Sub-Spawns in Form einer Listen-Datenstruktur in MPAL verwaltet. Um Prioritäten der auszuführenden Arbeitspakete einzuhalten und tiefer eingenistete Arbeitspakete zuerst zu bearbeiten, wird immer als erstes der Pool für Sub-Spawns auf auszuführende Arbeitspakete geprüft. Aus dieser Methode wird auch beim Erzeugen von Sub-Spawns (zusätzliche Arbeitspakete im Sub-Spawn-Vorrat) die Anzahl der Framework-Tasks konstant gehalten. Diese Framework-Tasks arbeiten in vorgegebener Reihenfolge zunächst vorhandene Sub-Spawns ab, bevor die eigentlichen Root-Spawns bearbeitet werden. Sub-Spawns existieren nicht von Beginn einer parallelen Sektion an, sondern werden zur Laufzeit erzeugt. Somit kann es allerdings vorkommen, dass Threads, die Arbeitspakete eines Sub-Spawns bearbeiten sollen, zur Erzeugung der Sub-Tasks bereits blockiert sind. Es muss also möglich sein, bereits blockierte Threads wieder aufzuwecken, um beim Eintreffen von Sub-Spawns Arbeitspakete zu bearbeiten.

3.2.4 Beschleuniger (Offload)

Um mit MPAL auch Beschleuniger, wie z.B. GPUs oder DSPs, programmieren zu können, wurde MPAL um eine abstrakte Klasse *Offloader* erweitert, die alle notwendigen Methoden beinhaltet, um einen Offload durchzuführen, wie das Triggern von Compute-Kerneln, Synchronisationen, oder Profiling. So, wie auch die abstrakte Klasse ParallelWorker, kann auch die Offloader-Klasse vererbt und innerhalb jeder Vererbung auf spezifische Offload-Frameworks, wie CUDA oder OpenCL, spezialisiert werden. Die eigentliche Offload-Funktionalität kann dann z.B. mithilfe einer spezialisierten Instanz des OpenCL-Offloaders verwendet werden. Da die Vererbungen vollständig kompatibel mit der abstrakten Klasse sind, die somit als Interface-Klasse verwendet werden kann, müssen Applikationsentwickler die verwendeten Frameworks nicht explizit instanziieren, sondern verwenden lediglich die Aufrufe der virtuellen Klasse. Abhängig vom Framework, das in der MPAL-Konfigurationsdatei gewählt ist, instanziiert MPAL automatisch die richtige Framework-Spezialisierung.

Für ein vollständig kontrollierbares heterogenes Ausführungsverhalten bei gleichzeitig hohem Komfort durch einen hohen Automatisierungsgrad, besitzt MPAL ein Speicherobjekt, das alle Funktionalitäten für die Verwaltung von Buffern kapselt (z.B. Kopieren bzw. Mappen des Speichers oder asynchrone DMA-Operationen). Durch die Verwendung von Funktionen, die entweder einen Host- oder einen Beschleuniger-Pointer (ein Pointer auf den zum Beschleuniger gemappten Buffer, der nur im Adressbereich des Beschleunigers gültig ist) zurückgeben, wird automatisch der aktuelle Kohärenzzustand überprüft und, falls nötig, ein Mapping durchgeführt, bevor der jeweilige Pointer zurückgegeben wird. Ob bei der Herstellung der Kohärenz zwischen den Adressbereichen des Hosts und des Beschleunigers ein DMA verwendet wird, die CPU oder der Beschleuniger während des Kopierens blockieren, oder ob ein zero-copy Speicher vorliegt, kann dabei im Speicherobjekt konfiguriert werden. Dieses Speicherobjekt ist ebenfalls abhängig vom jeweils gewählten Offload-Framework und daher auch Bestandteil einer Spezialisierung zu einem Offload-Framework durch Vererbung der abstrakten Klasse. Dabei sind allerdings die wesentlichen Funktionalitäten bereits in MPAL vorhanden, sodass lediglich die eigentliche Funktion (z.B. für das Erzeugen oder Kopieren eines Buffers) spezialisiert werden muss. Daher benötigt jede neue Framework-Spezialisierung nur wenige Zeilen zusätzlichen Software-Code.

In der aktuellen Version besitzt MPAL eine Spezialisierung für OpenCL und ist durch die Bereitstellung der abstrakten Methoden und internen Verarbeitung von Datensequenzen vorbereitet für weitere Spezialisierungen, wie CUDA- oder OpenMP-Offloader, die sich mit wenig Implementierungsaufwand hinzufügen lassen. Durch die Verwendung von externen Frameworks, um

```
__kernel__ void k( parameters ) {  // Beschleuniger-Kernel
    doWork( rangePos(), parameters );
}
```

(a) Algorithmische Implementierung als Beschleuniger-Kernel in OpenCL.

```
MPAL::ParallelWorker pw;
// Heterogene Arbeitsverteilung:
pw.offload( s_acc, k( parameters ), u ); // Offload
pw.spawn( s_cpu, [&]( s_sub ){ // CPU-spawn
        for( i ∈ s_sub ) doWork( i, parameters );
    } );
pw.sync( u ); // Synchronisation mit Beschleunigern
```

(b) Haupt-Programm (z.B. main) verwaltet den Prozess-Ablauf.

Algorithmus 3.4: Implementierung einer heterogenen MPAL-Applikation, die die Teilsequenz $\vec{s}_{acc}$ den Beschleunigern $\vec{u}$ zuweist (pw.offload), während die Beschleuniger die zugewiesenen Tasks verarbeiten, die Teilsequenz s_{cpu} der Multicore-CPU zuweist (pw.spawn) und im Anschluss CPU und Beschleuniger synchronisiert (pw.sync).

Beschleuniger programmieren zu können, ist es notwendig, dass Entwickler zusätzlich zur parallelen CPU-Implementierung (Algorithmus 3.4(b)) separate Beschleuniger-Kernel in der jeweils vom Offload-Framework benötigten Sprache bereitstellen (Algorithmus 3.4(a)).

Um Ausführungs- und Wartezeiten der Compute-Kernel auf den Beschleunigern messen zu können, werden in MPAL die internen Profiling-Methoden des jeweiligen Offload-Frameworks verwendet. Es wird nicht nur die Kernel-Ausführungszeit gemessen, sondern auch die Wartezeit des Kernels in der Task-Warteschlange (engl. *Queue*) und die Transferzeit der Daten vom Host auf den Beschleuniger. Ebenso kann in MPAL der Overhead für das Anlegen und Verwalten von Tasks im Offload-Framework berechnet werden – ähnlich zum Scheduling-Overhead der regulären Parallelisierungs-Frameworks für CPUs. Die Verwaltung von Beschleunigern und Daten sowie Task-Scheduling und Synchronisation von heterogenen Tasks für Beschleuniger und CPU-Cluster kann einen großen Overhead darstellen, der nicht vorhersagbare Wartezeiten und damit Work-Imbalance in die Ausführung einbringen und dadurch das Laufzeitverhalten deformieren kann. Darum kann es in manchen Strategien für das Architektur-Mapping sinnvoll sein, einen dedizierten Prozessorkern hauptsächlich mit der Verwaltung von heterogenen Tasks zu beauftragen.

Durch die generische API für die Parallelisierung von Software für heteroge-

ne CPUs und Beschleuniger mit freier Wahl des angewendeten Parallelisierungs- und Offload-Frameworks bietet MPAL eine deutliche Verbesserung der Portabilität einer heterogenen Implementierung. Das Architektur-Mapping und die Arbeitsverteilung der Sequenzen zwischen CPUs und Beschleunigern können dabei mithilfe der Konfigurationsdatei angepasst werden, wodurch auch die Skalierbarkeit verbessert und eine schnelle Entwurfsraumexploration ermöglicht wird. MPAL bietet Methoden eine API, sowie Methoden und Werkzeuge für eine effizientere Programmierung und Konfiguration von parallelen und heterogenen Implementierungen, bietet aber keine automatische Lastverteilung der Arbeitspakete. Implementierungen für eingebettete Systeme erfordern häufig eine maximale Optimierung, weshalb automatisierte Scheduling-Ansätze nicht geeignet sind, da sie oft nur lokale Optima erreichen. Da Applikationsentwickler selbst den besten Einblick in die algorithmischen Details und somit die Parallelisierungs- und Optimierungspotentiale haben, wird die Feinabstimmung des Architektur-Mappings bewusst den Software-Entwicklern überlassen.

3.2.5 OpenCL HLS

Durch die Verwendung von OpenCL als Offload-Framework, das neben GPUs auch eine Vielzahl anderer Beschleuniger unterstützt, ist es ebenfalls möglich durch *High-Level Synthesis* (HLS) Compute-Kernel für FPGAs zu synthetisieren. Während jeder OpenCL-Beschleuniger Gerätetreiber und eine OpenCL-Runtime-Bibliothek benötigt, kann die Host-Applikation die Geräte über eine generische und plattformunabhängige API verwalten. Somit können auch Methoden für das Mappen von Speicher und das Triggern von Compute-Kerneln für FPGAs implementiert werden, was wiederum eine Basis für die Verwendung eines FPGAs als Beschleuniger für heterogene Prozesse darstellt. Hierbei kann allerdings nicht wie für die anderen Beschleuniger-Typen, eine *Just-In-Time* (JIT)-Kompilation verwendet werden, sondern die FPGA-Konfigurationen müssen bereits vor der Ausführung der Applikation durch eine HLS erzeugt werden. Die Kommunikation zwischen Host und FPGA wird dabei durch synthetisierte Hardware-Module realisiert, die zusätzlich zu den Compute-Kerneln auf dem FPGA abgebildet werden, wobei auch die Abbildung und Verwendung eines DMAs möglich ist. Da die vollständige OpenCL-API in MPAL gekapselt zur Verfügung steht, können FPGAs ohne weitere Änderungen verwendet werden, wenn hier zuvor erzeugte FPGA-Konfigurationsdateien geladen werden.

In einem ersten Test dieser Methode [51] wurde der SGM-Algorithmus auf einem Nallatech FPGA-Beschleuniger mit einem Intel Stratix-V FPGA und 8 GB lokalem Speicher ausgeführt. Die PCIe-Erweiterungskarte wurde mit

einem i5-2400 Dualcore mit je zwei Hyperthreads bei 3,1 GHz betrieben. Da in diesem Test vor allem die Portabilität untersucht wurde, wurden zunächst keine FPGA-spezifischen Optimierungen an den für einen GPU-Beschleuniger entwickelten Compute-Kerneln vorgenommen. Die zunächst erzeugte FPGA-Konfiguration konnte den vollständigen SGM-Algorithmus bei einer Verwendung von 80 % der FPGA-Ressourcen auf dem FPGA abbilden. Für massiv parallele GPUs werden *ND-range-Kernel* entwickelt, die dann über alle GPU-Kerne verteilt werden, wohingegen für FPGAs meist die Ausführung von *Single-Work-Item Kerneln* bzw. die Instanziierung von nur wenigen Work-Items bevorzugt wird [53]. Da hier keine Optimierungen angewendet wurden, können die OpenCL-Kernel zwar auf dem FPGA abgebildet und ausgeführt werden, erreichen aber eine deutlich geringere Performance als eine GPU. Des Weiteren wird deutlich, dass die Syntheseergebnisse stark variieren in Abhängigkeit von der Anzahl der abzubildenden Compute-Kerneln und der zur Verfügung stehenden Ressourcen. Stehen weitere Ressourcen zur Verfügung, ist der Synthese-Prozess beispielsweise in der Lage, Schleifen auszurollen und dadurch dynamisch weitere Ressourcen, wie Block-RAM oder DSP-Slices, zu allozieren. Somit wird deutlich, dass für das Erreichen einer adäquaten Performance eine Optimierung der OpenCL-Kernel für FPGAs notwendig ist.

3.3 Programmierschnittstelle

Der nachfolgende Abschnitt beschreibt die Grundlagen der MPAL-API und die existierenden Klassen und Methoden zur Parallelisierung und Programmierung von heterogenen Architekturen. Eine Auflistung wesentlicher MPAL-Funktionalitäten ist in Anhang A gegeben.

3.3.1 Software-Schnittstelle / API

Während alle Klassen und Funktionalitäten von MPAL, wie zum Beispiel das Core-Pinning, auch direkt verwendet werden können, bietet MPAL ebenfalls Elemente, die die volle Funktionalität verwalten, durch die Konfigurationsdatei konfiguriert werden und den gesamten Arbeitsablauf automatisieren (Abbildung 3.9). MPAL bietet dafür außerdem ein Makefile-System, mit dem mit nur wenigen Zeilen MPAL-Projekte erstellt werden können und dadurch mit sehr geringem Aufwand der volle Funktionsumfang von MPAL zur Verfügung steht. Ein MPAL-Projekt besteht dabei aus den folgenden Komponenten:

- Die **Host-Applikation (C++)** beinhaltet sowohl das Hauptprogramm und den Kontroll-Code für alle parallelen Recheneinheiten,

als auch die parallelisierten Task-Funktionen für die homogene oder heterogene Parallelisierung auf CPUs.

- Die **Beschleuniger-Kernel (OpenCL, CUDA, usw.)** implementieren die algorithmischen Funktionen, die während einer heterogenen Ausführung auf Beschleuniger ausgelagert werden sollen und implementieren eventuell dieselbe algorithmische Funktionalität, wie die parallele CPU-Implementierung.
- Die **Konfigurationsdatei** beinhaltet Parameter, wie die Konfiguration des Parallelisierungs- und Offload-Frameworks, das Architektur-Mapping und das gewählte Core-Pinning.
- Das **Makefile** definiert das MPAL-Projekt und bindet, sofern die MPAL-Make-Struktur verwendet wird, automatisch alle benötigten Komponenten und Einstellungen mit ein.

Wurde ein ausführbares Programm erzeugt, kann das Architektur-Mapping in der Konfigurationsdatei angepasst werden und dadurch eine schnelle Entwurfsraumexploration durchgeführt werden, ohne das Programm neu zu kompilieren. Auch die Extraktion von Profiling-Informationen ist mit der Konfigurationsdatei einstellbar, was eine schnelle Analyse der Charakteristiken und eine Bottleneck-Identifikation ermöglicht. Für den Aufbau einer parallelen oder heterogenen Applikation stehen die folgenden Klassen zur Verfügung:

***Profiler*-Klasse:** Die Konfigurationsdatei wird von der *Profiler*-Klasse eingelesen, die einen MPAL-internen Parser verwendet und wendet automatisch ein eventuell eingestelltes Core-Pinning an. Der *Profiler*, der die gesamte aktuelle Konfiguration speichert, erstellt die Framework-spezifische *Parallel-Worker*-Klasse und eine *Distributor*-Klasse, die wiederum die Konfiguration

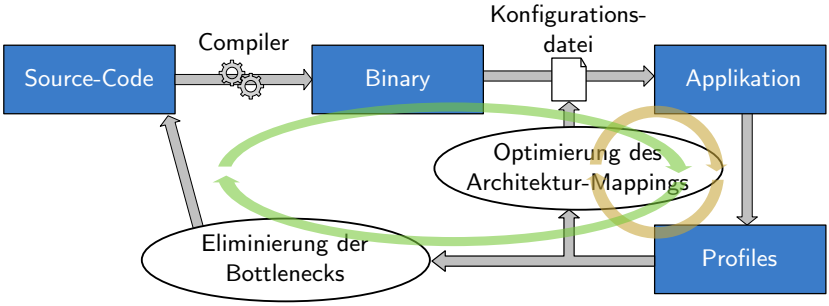


Abbildung 3.9: Programmierverlauf mit automatisiertem MPAL-Profiling und Architektur-Mapping mithilfe der Konfigurationsdatei.

des Architektur-Mappings beinhaltet. Für eine einfachere Handhabung wird die Framework-spezifische *Offloader*-Klasse als Member der *ParallelWorker*-Instanz übergegeben. Alle Offload-Funktionen sind über Member-Funktionen des *ParallelWorker* aufrufbar. Somit können alle heterogenen Tasks allein durch die API der *ParallelWorker*-Klasse erzeugt und kontrolliert werden. Der *Profiler* kontrolliert außerdem den Ablauf des Hauptprogramms, wozu zwei einkapselnde Sektionen definiert wurden, die dem Hauptprogramm Hinweise für das automatisierte Profiling geben: *MPAL_SECTION* und *MPAL_ALGO* (C-Defines). Die *MPAL_SECTION* umschließt den Bereich der Applikation, der für jeden Profiling-Durchlauf durchgeführt werden muss inklusive notwendige Speicherallokation, Initialisierungen und die Erzeugung des *ParallelWorkers*. Ist die Profile-Extraktion aktiviert, wird diese Sektion in einer Schleife mehrfach ausgeführt. Die *MPAL_ALGO*-Sektion definiert die reine algorithmische Ausführung, innerhalb der ein Profile gemessen werden soll. Profiling-Messungen werden automatisch zu Beginn dieses Bereichs aktiviert, während vorangegangene Initialisierungen das Profiling nicht beeinflussen. In Streaming-Applikationen, die eine Initialisierung bzw. einen Reset von Datenstrukturen vor jedem Durchlauf benötigen, muss diese Initialisierung allerdings in die Messung mit einbezogen werden.

***ParallelWorker*-Klasse:** Da das Profiling ein Teil der *ParallelWorker*-Klasse ist, kann die Extraktion von Charakteristiken auch manuell angestoßen werden, wenn keine der oben aufgeführten Sektionen verwendet wird. In diesem Fall wird die Messung bei Erzeugen der *ParallelWorker*-Instanz gestartet und bei Aufruf des Destruktors beendet. Durch die Methoden zum Erzeugen und Kontrollieren von Tasks für homogene und heterogene Multicores, sowie der Offload von Beschleuniger-Kerneln steht der *ParallelWorker* als zentrales Element von MPAL für jede Form der Parallelisierung zur Verfügung. Die *Spawn*-Funktion des *ParallelWorkers* kann mit unterschiedlichen Parametern aufgerufen werden, die anschließend das resultierende Verhalten bestimmen. Für eine reguläre Parallelisierung auf einem homogenen Multicore wird eine Funktion als Argument übergeben, die von jedem parallelen Task ausgeführt werden soll, eine aufzuteilende Sequenz und die gewünschte Anzahl der Tasks. Wird die *Spawn*-Funktion mit einer *PinnedFunctionMap* aufgerufen, die aus einer Liste aus Tupeln aus jeweils einer Task-Funktion, einer aufzuteilenden Sequenz und einem assoziierten Core-Cluster besteht, werden diese Funktionen über heterogene CPU-Cluster verteilt, wie es im Architektur-Mapping definiert ist. Ebenso, wie die homogene Parallelisierung, kann auch eine Funktion zu einem Beschleuniger ausgelagert werden. Die *Offload*-Funktion benötigt als Argument einen Beschleuniger-Kernel (Software-Code, oder FPGA-Konfiguration), eine zu bearbeitende Sequenz und die Bezeichnung des gewünschten Beschleunigers. Da die Instanz des *ParallelWorkers* bereits eine Framework-Spezialisierung

darstellt, können aus dem *ParallelWorker* auch Framework-spezifische Mutex-Locks abgeleitet werden.

***Memory*-Klasse:** MPAL bietet zwei Speicherobjekte für eine verbesserte Kontrolle der Speicherzugriffe auf dem Host-Speicher, sowie dem Beschleunigeradressbereich. Die *Matrix*-Klasse speichert entweder zwei- oder dreidimensionale Datensätze in einem kontinuierlichen Array, während Elemente entweder durch die Angabe von Koordinaten durch einen Zugriffsoperator oder durch direkten Zugriff per Pointer verwendet werden können. Diese beiden Zugriffsmethoden bieten somit auf der einen Seite eine einfache und gut lesbare algorithmische Entwicklung und ermöglichen auf der anderen Seite Speicheroptimierungen für eingebettete Applikationen. Aus der Framework-spezifischen *ParallelWorker*-Klasse können Beschleuniger-Buffer erzeugt werden, die vollständig kompatibel mit der *Matrix*-Klasse sind, sobald der Speicher zum Host-Adressbereich gemappt ist. Die *AcceleratorBuffer*-Klasse speichert intern den aktuellen Mapping-Zustand (konsistent im Host-Adressbereich oder konsistent im Beschleuniger-Adressbereich), um eine Speicherkohärenz zu garantieren. Wird entweder ein CPU-Pointer oder ein Pointer zum Beschleunigeradressbereich angefragt, wird zunächst der Mapping-Zustand geprüft und eventuell ein Mapping durchgeführt, bevor der Pointer zurückgegeben wird.

***Distributor*-Klasse:** Die *ParallelWorker*-Klasse benötigt für die Parallelisierung von Funktionen auf heterogenen Plattformen eine explizite Annotation der Arbeitsverteilung. Eine solche *PinnedFunctionMap* kann von der *Distributor*-Klasse, die alle Informationen über das konfigurierte Architektur-Mapping beinhaltet, automatisch erzeugt und anschließend der *Spawn*-Funktion als Argument übergeben werden. Dabei ermöglicht die *Distributor*-Klasse Zugriff auf alle konfigurierten Core-Cluster und Arbeitsverteilungen für Beschleuniger. Eine automatisch erzeugte *PinnedFunctionMap* kann anschließend um weitere Funktionen erweitert werden, um komplexe heterogene Ausführungs- und Parallelisierungsszenarien zu realisieren.

3.3.2 Konfigurierbarkeit

In diesem Abschnitt wird der Aufbau der Konfigurationsdatei erklärt. Dabei werden zunächst die Hauptsektionen *Parallelization* und *Profiling* erläutert und anschließend drei einfache Beispiele zu den Szenarien *homogene Parallelisierung*, *heterogenes Architektur-Mapping* und *Profiling* präsentiert.

Parallelization: Diese Sektion der MPAL-Konfigurationsdatei definiert die grundlegenden Parameter der Parallelisierung der finalen Applikation. Hier wird das Parallelisierungs-Framework gewählt, welches für die Erzeugung und

```
[Parallelization]
        Framework       = TBB
        Threads         = 16
        CorePinning     = {0:4:2}
        # Core-Pinning: offset, cluster-size, threads-per-cluster
        # (0, 1, 4, 5, 8, 9, ..)
[end]

[Profiling]
        Profiling       = false
[end]
```

Algorithmus 3.5: Beispielkonfiguration einer homogenen Ausführung mit Core-Pinning ohne Extraktion eines Profiles.

Verwaltung der Tasks genutzt wird. Im Fall einer homogenen Parallelisierung (Algorithmus 3.5) wird die Zahl der gewünschten Threads, sowie ein optionales Core-Pinning angegeben. Ein Core-Pinning kann entweder in Form einer Liste mit expliziten Prozessor-IDs oder einem sich wiederholenden Schema, wie z.B. "jeder zweite Kern", angegeben werden. Für eine heterogene Parallelisierung (Algorithmus 3.6) wird hier der Name einer Architektur mit einem zugehörigen Architektur-Mapping definiert. Das Architektur-Mapping wird dann in einer separaten Sektion beschrieben. Für diesen Fall wird die Anzahl der Threads

```
[Parallelization]
        Framework       = TBB
        Architecture    = Exynos5422
        Distribution    = bigLITTLE+GPU
[end]

[Exynos5422]
        bigLITTLE+GPU   = { big:0.35, LITTLE, GPU0:0.5 }
        big             = { 4:7 }
        LITTLE          = { 0:3 }
[end]

[Profiling]
        Profiling       = false
[end]
```

Algorithmus 3.6: Beispielkonfiguration einer heterogenen Ausführung beider CPU-Cluster und der GPU des Exynos 5422 MPSoCs (*big*-Cluster zu 35 %, GPU mit ID 0 zu 50 % und *LITTLE*-Cluster zu den verbleibenden 15 %).

```
[Parallelization]
        Framework       = TBB
        CorePinning     = {0:4:2}
[end]

[Profiling]
        Profiling       = true
        ThreadSequence  = {1:16}
        OutputFile      = "profile"
        InitRuns        = 20
        TraceRuns       = 30
        PCountRuns      = 30
        PCountTypes     = {"L1_DCM"}
[end]
```

Algorithmus 3.7: Beispielkonfiguration einer homogenen Ausführung mit Core-Pinning und Extraktion von Profile-Informationen mit einer Thread-Sequenz (1-16 Threads) zur Analyse des Skalierungsverhaltens (20 nicht gespeicherte Initialisierungsdurchläufe, 30 zeitliche Messdurchläufe und 30 Messungen der Performance-Counter (hier: L1-DCM)).

und das definierte Core-Pinning nicht berücksichtigt.

Profiling: In dieser Sektion der MPAL-Konfigurationsdatei wird eingestellt, ob und welche Profiling-Informationen extrahiert werden sollen (Algorithmus 3.7). Es kann auch die Anzahl der durchzuführenden Profiling-Durchläufe und die zu extrahierenden Messungen und Performance-Counter definiert werden. Ist die Messung deaktiviert, werden alle Profiling-Methoden und die dazugehörigen Funktionsaufrufe vollständig eliminiert, wodurch ein vollständig Overhead-freier *Performance-Run* der Anwendung ermöglicht wird. Für den Fall einer homogenen Parallelisierung kann eine automatisierte Skalierungsanalyse durchgeführt werden, wobei automatisch eine Sequenz von zu verwendenden Threads auf die resultierende Performance getestet wird und alle Laufzeitparameter extrahiert werden. Wird ein heterogenes Architektur-Mapping konfiguriert, muss eine konkrete Verteilung angegeben werden, weshalb eine Skalierungsanalyse nicht in Frage kommt, sondern nur das statisch vorkonfigurierte Mapping gemessen werden kann.

Sobald die Extraktion von Profiling-Informationen in der Konfigurationsdatei aktiviert ist, erzeugt die interne MPAL-Struktur automatisiert ein Profile und speichert alle gewünschten Informationen in eine Datei. Diese Datei kann anschließend offline und systemunabhängig mithilfe von MPAL-zugehörigen Werkzeugen untersucht werden, wodurch auch komplexe Analysen und die Verwendung aufwändiger Werkzeuge, sowie die Verwendung einer grafischen

Oberfläche und das automatische Erzeugen von Plots, ermöglicht wird. Um aussagekräftige Charakteristiken zu extrahieren und eine detaillierte Analyse des Laufzeitverhaltens sowie die Auslastung der vorhandenen Ressourcen zu ermöglichen, bietet MPAL zwei wichtige Werkzeuge: ein Kommandozeilen-Tool und eine grafische Oberfläche.

3.4 Portabilität

Im Bereich der eingebetteten Systeme müssen Applikationen häufig bis zum Äußersten Performance-optimiert werden, weshalb eine manuelle Feinabstimmung nötig ist. Tabelle 3.4 fasst die diskutierten Low-Level Frameworks und Middleware-Abstraktionsschichten zusammen und hebt ihre Eigenschaften hervor. All diese Programmieransätze haben unterschiedliche Einstiegshürden, bieten unterschiedliche Vor- und Nachteile und resultierende Performance. Außerdem unterscheidet sich der Aufbau und die Portabilität des Software-Codes, der mit diesen Tools entwickelt wurde, teilweise stark. Manche Werkzeuge bieten implizite Profiling-Möglichkeiten, während andere auf externe Profiler zurückgreifen müssen. Daher hängt die Wahl des jeweils am besten geeigneten Werkzeugs sowohl von der Zielapplikation, als auch von den Wünschen und Vorlieben des Entwicklers ab.

Während OpenMP als ein Framework für die Shared-Memory Multicore-Parallelisierung zusätzlich Zugriff auf die Thread-Verwaltung ermöglicht, fokussieren sich TBB und CilkPlus hauptsächlich auf die Verwaltung von Tasks und die Implementierung von parallelen Algorithmen, wie *parallel_for* oder *reduce*. Die Unterstützung von komplexeren parallelen Strukturen wurde in C++ ab Version C++17 integriert, die allerdings im GCC-Compiler erst ab Version 9 verfügbar ist und auf Intels PSTL-Implementierung basiert. Um Beschleuniger programmieren zu können, müssen Entwickler meist die Programme vollständig oder zumindest zu großen Teilen in Beschleuniger-Kernel übersetzen (z.B. CUDA oder OpenCL). Eine Schnittstelle zu nativem C++ ist mit Thrust für CUDA-GPUs und SyCL für OpenCL-Beschleuniger gegeben, wodurch die Programmierung von Beschleunigern für C++-erfahrene Entwickler zugänglicher wird. Allerdings benötigen Beschleuniger-Kernel weiterhin Rechen-Overhead durch eine explizite Verwaltung der Geräte und der Speicherbereiche, der lediglich durch einen kohärenten geteilten Speicherbereich reduziert werden kann. Nur durch die Programmierung von Beschleunigern mit OpenMP-Direktiven wird der Compiler eingesetzt, um automatisch die Verwaltung der Beschleuniger-Geräte und des Speichers zu übernehmen, was allerdings erst ab OpenMP 5.0 verfügbar ist. OmpSs-Direktiven bieten grundsätzlich dieselben Möglichkeiten, unterstützen allerdings eine größere Beschleunigervielfalt. MPAL ist ein

Tabelle 3.4: Vergleich der diskutierten Programmier-Methoden und deren Möglichkeiten bzw. Fokussierung.

Framework	Abhängigkeiten	Architekturen	Sprachen	Fokus
Shared-Memory Multicore Parallelisierung				
OpenMP [10]	compiler support	CPU, GPU (limited support)	C/C++	CPUs
OmpSs [16]	compiler support	CPU, GPU	C/C++	CPUs, accelerators
Intel TBB [17]	library	CPU	C++	CPUs
C++17 [54]	compiler supp., lib.	CPU	C++	CPUs
CilkPlus [18]	compiler supp., lib.	CPU	C++	CPUs, vectorization
Beschleuniger-Programmierung				
CUDA [42]	nvcc compiler	NVIDIA GPU	CUDA kernel	NVIDIA GPUs
Thrust [44]	nvcc compiler	NVIDIA GPU	C/C++	NVIDIA GPUs
OpenCL [8]	runtime, library	CPU, GPU, DSP, FPGA (HLS)	OpenCL kernel	div. accelerators
SyCL [43]	runtime, library	CPU, GPU, DSP	C++	div. accelerators
HSA [12]	runtime, driver	CPU, GPU, DSP	C++/OpenCL	het. platforms
Abstrahierende Middleware-Schichten				
MPAL [50]	backends, library	het. CPU, GPU, DSP, FPGA	C++, kernel	conf. arch. mapping
Kokkos [48]	backends, library	CPU, NVIDIA GPUs	C++, kernel	arch. aware memory
StarPU [49]	backends, library	het. CPU, GPU, DSP, FPGA	C++, kernel	advanced scheduling

abstrahierender Middleware-Ansatz, der eine höhere Konfigurierbarkeit der Parallelität und des Architektur-Mappings auf homogenen und heterogenen parallelen Plattformen ermöglicht und gleichzeitig die Applikationen unabhängig vom eingesetzten Low-Level-Framework für Parallelisierung und Offload zu Beschleunigern macht. Gleichzeitig bietet es umfangreiche Profiling- und Analysemethoden. Andere Abstraktionsschichten, wie Kokkos, bieten Speicherverwaltungsmechanismen über verschiedene heterogene Prozessorarchitekturen. StarPU bringt zusätzlich ein automatisiertes heterogenes Task-Scheduling.

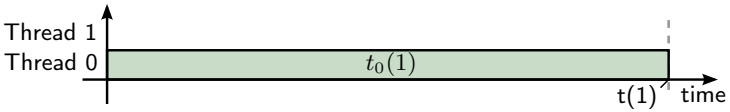
Parallelität kann auf mehreren Ebenen vorhanden sein (z.B. Daten- und Task-Level-Parallelität) und sollte ebenso auf mehreren Ebenen ausgenutzt werden z.B. durch parallele Prozessoren, Beschleuniger und SIMD-Instruktionen. Abgesehen vom Einsatz von automatisierten Parallelisierungs- und Vektorisierungs-Werkzeugen, ist der Entwickler weiterhin dafür verantwortlich eine geeignete Partitionierung des Codes und somit eine möglichst skalierbare Aufteilung der Arbeitspakete zu entwickeln. Auch die Analyse und Fehlerbehebung von parallelen Implementierungen kann aufgrund des nicht-deterministischen Verhaltens und der vielfältigen Einflussfaktoren eine komplexe Aufgabe darstellen. Das Skalierungsverhalten einer Implementierung kann dazu aufschlussreiche Informationen liefern, um Bottlenecks zu identifizieren und das parallele Laufzeitverhalten zu charakterisieren. Vereinheitlichungsansätze, wie die MPAL-Abstraktionsschicht, helfen durch integrierte Methoden zur Extraktion der Laufzeiteigenschaften das Skalierungsverhalten unterschiedlicher Programmier- und Parallelisierungstechniken zu untersuchen. Da die Migration einer Implementierung zwischen unterschiedlichen Architekturen selbst zu starken Performance-Einbußen führen kann, stellen Modellierungen und die Vorabschätzung der Performance einen wesentlichen Gegenstand aktueller Forschung dar, wie es im weiteren Verauf dieser Arbeit behandelt wird.

4 Laufzeit-Charakterisierung

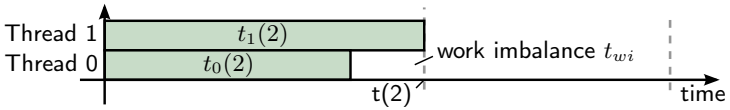
Die meisten Profiler bieten funktionsbasierte Profiles, in denen nach Funktionsaufrufen sortierte Baumstrukturen erstellt werden, und akkumulierte Ausführungszeiten der Funktionen erhoben werden. Werden jedoch parallelisierte Programme und damit einhergehende Parallelisierungseffekte und Bottlenecks untersucht, sollten Profiler exklusiv relevante Parallelisierungsereignisse berücksichtigen. Die in MPAL integrierten generischen Profiling-Probes bieten eine direkte Schnittstelle zu eben diesen Ereignissen der parallelen Ausführung, wie Task-Erzeugung, Task-Ausführungszeiten und ID des ausführenden Prozessorkerns, Synchronisationen oder Locks. In diesem Kapitel werden daraus gewonnene Informationen zu Einflussfaktoren der parallelen Ausführung und daraus abgeleitete Skalierungscharakteristiken eines parallelen Programms, sowie potentielle Skalierungslimitierungen (z.B. fehlerhafte Synchronisationen der Arbeitsungleichverteilungen) beschrieben.

4.1 Laufzeiteigenschaften

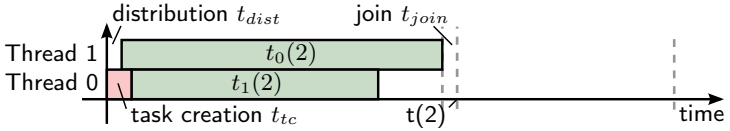
Abbildung 4.1(a) stellt eine sequentielle Ausführung eines Programms auf einem Zweikernprozessor dar. Um den Mehraufwand eines Kontextwechsels zwischen Threads zu vermeiden, wird jeweils nur ein Thread pro Prozessorkern verwendet. Da Hyperthreads im Betriebssystem als separate Kerne repräsentiert werden, kann damit im Folgenden die Anzahl der verwendeten Prozessorkerne, die Anzahl der Hardware-Threads und auch die Anzahl der Software-Threads gleich gesetzt werden. Wird die Gesamtausführungszeit eines parallelen Programms, das n Prozessorkerne verwendet, als $t(n)$ geschrieben, so ist die sequentielle Ausführungszeit $t(1)$. Um Overhead für die Verwaltung von Tasks zu minimieren, sollte in einer sequentiellen Ausführung auch nur ein Task vorhanden sein ($k = 1$, mit Anzahl der Tasks k). Während $t_i(n)$



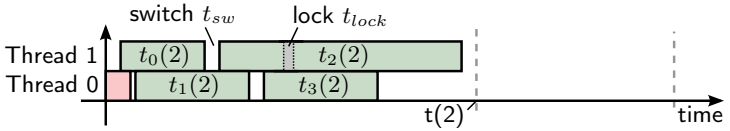
(a) Sequentielle Ausführung mit Gesamtausführungszeit $t(1)$.



(b) Idealisierte parallele Ausführung mit Gesamtausführungszeit $t(2)$.



(c) Zusätzlicher Berechnungsaufwand durch Framework-Einflüsse.



(d) Redundanz durch Locks und Verzögerung durch Task-Verteilung.

Abbildung 4.1: Skizze eines exemplarischen nebenläufigen Programms.

die Ausführungszeit der parallelen Tasks mit $i \in [0, k-1]$ bezeichnet, ist in einer sequentiellen Ausführung also $t_0(1) = t(1)$. Im Folgenden werden unterschiedliche Einflussfaktoren betrachtet, die sich auf das parallele Ausführungsverhalten und die Skalierungscharakteristik auswirken und entweder auf die Parallelisierungsstrategien oder auf den Framework-spezifischen Overhead zurückzuführen sind.

Arbeitsungleichverteilung: Sofern eine Granularität von einem Task pro Thread angenommen werden kann, muss eine ideale Parallelisierung die gesamte parallelisierbare Arbeit ($t_0(1)$) in gleichgroße Teile für jeden Prozessorkern teilen. Wie in Abbildung 4.1(b) für $n = 2$ dargestellt ist, entsteht bei der Teilung der meisten algorithmischen Rechenaufgaben eine nicht vermeidbare Ungleichverteilung. Auch eine nebenläufige Ausführung kann die Ausführung einzelner Tasks deformieren, was ebenfalls zu einer Arbeitsungleichverteilung (engl. *Work-Imbalance*) führt. Zusätzliche Prozessorzeit, die aufgrund von Arbeitsungleichverteilung nicht effektiv für die Berechnung von Tasks verwendet werden kann (*Idle*-Zeit), wird als $t_{wi}(n)$ bezeichnet. Im hier gezeigten Szenario wird angenommen, dass zunächst keine weiteren Einflüsse vom Parallelisie-

rungs-Framework auf das Laufzeitverhalten einwirken. Damit ist auch die Summe der Ausführungszeiten der Tasks auf beiden Prozessorkernen identisch mit der sequentiellen Ausführungszeit ($\sum_{i=0}^{k-1} t_i(n) = t_0(1)$, bei $k = 2$). Die parallele Gesamtausführungszeit ist nur durch $t_{wi}(n)$ beeinflusst:

$$t(n) = \frac{\sum_{i=0}^{k-1} t_i(n) + t_{wi}(n)}{n}. \tag{4.1}$$

Framework-Einflüsse: Wie in Abbildung 4.1(c) gezeigt, beeinflusst auch der Verwaltungsaufwand des Parallelisierungs-Frameworks das Laufzeitverhalten. In den meisten Fällen wird die Erzeugung von Tasks in einer nicht parallelisierbaren Schleife abgearbeitet, in der Tasks nacheinander erzeugt und dem Task-Scheduler übergeben werden. Für den Scheduler werden Tasks als ein Tupel aus einem Funktionspointer (oder Funktionstemplate) und den anzuwendenden Funktionsargumenten repräsentiert. Der zusätzliche Aufwand, der für die Erzeugung dieser Tasks (engl. *Task Creation*) benötigt wird, wird durch die Zeit $t_{tc}(n)$ ausgedrückt. Für die Verteilung der Tasks auf parallele Prozessorkerne (engl. *Distribution*) entsteht aufgrund der konsekutiven Erzeugung und durch Bus-Verzögerungen zwischen Kernen weiterer Overhead $t_{dist}(n)$. Auch das Wechseln zwischen Tasks (engl. *Switch*) auf einem Prozessorkern innerhalb eines Task-Teams (bei feinerer Granularität $k > n$) kann im Framework Verwaltungsaufwand $t_{sw}(n)$ entstehen. Des Weiteren involvieren manche Frameworks ebenfalls relevanten Overhead zur Terminierung und Synchronisation von Task-Teams $t_{join}(n)$. Die Summe dieser vom Framework-Scheduling involvierten Overhead-Zeiten wird als $t_{sched}(n)$ bezeichnet, wodurch sich die Berechnung von $t(n)$ wie folgt erweitert:

$$t(n) = \frac{\sum_{i=0}^{k-1} t_i(n) + t_{wi}(n) + t_{sched}(n)}{n}, \text{ mit} \tag{4.2}$$

$$t_{sched}(n) = t_{tc}(n) + t_{dist}(n) + t_{join}(n) + t_{sw}(n). \tag{4.3}$$

Nebenläufigkeitseffekte: Auf der einen Seite bringt eine gleichmäßige Arbeitsverteilung mit linearer Task-Ausführung und geringen Synchronisationen bei langläufigen Tasks die höchste Effizienz der Parallelisierung. Auf der anderen Seite ermöglicht der gezielte Einsatz des Task-Schedulings mit feingranularer Task-Aufteilung ein verbessertes Skalierungsverhalten durch adaptives Scheduling, das sich auch an in der Zwischenzeit geänderte Last durch andere Programme auf dem System anpassen kann. Allerdings bringt eine feingranulare Task-Aufteilung sowohl einen größeren Mehraufwand für die Erzeugung der Tasks (in Abbildung 4.1(d) rot markiert), als auch Berechnungs- und Kommunikationsaufwand für das Wechseln von Tasks $t_{sw}(n)$ mit sich. In dieser Arbeit wird allerdings von einer exklusiven Nutzung der parallelen

Plattform ausgegangen, sodass keine konkurrierenden Prozesse berücksichtigt werden. Darüber hinaus wird ein Prozessorkern blockiert und im Wartezustand gehalten, während eine angefragte Ressource (z.B. Speicherbereich) von einem anderen Prozessorkern blockiert ist, was typischerweise mit Mutex-Locks realisiert wird. Während die Wartezeiten durch Mutex-Locks ebenfalls die Ausführungszeit der jeweiligen Tasks $t_i(n)$ verlängert, wird sie für die Berechnung der Redundanz $R(n)$, die kennzeichnet, wie stark die Summe der Ausführungszeit aller parallelen Tasks im Vergleich zur sequentiellen Ausführung ansteigt, nicht berücksichtigt. In diesem Fall ist ein Anstieg der parallelen Taskausführungszeiten hauptsächlich auf Speicherengpässe und Latenzen durch Cache-Kohärenz zurückzuführen.

$$R(n) = \frac{\sum_{i=0}^{k-1} t_i(n)}{t_0(1)}. \tag{4.4}$$

Während einer nebenläufigen Ausführung können selbst indirekte oder nur teilweise blockierte Ressourcen, wie Speicherbandbreite, unzureichende Datenlokalität und Einflüsse des Kohärenzprotokolls die Redundanz vergrößern. Würden auch die nicht parallelisierbaren Anteile t_{seq} nach Amdahl berücksichtigt, würden die Gesamtausführungszeit einer parallelisierten Implementierung definiert als

$$t_{total}(n) = t(n) + t_{seq}. \tag{4.5}$$

In dieser Arbeit wird jedoch hauptsächlich jede parallelisierte Sektion separat betrachtet und der sequentielle Anteil zunächst nicht betrachtet, weshalb in dieser Arbeit nicht speziell auf die Gesetze nach Amdahl [38] oder Gustafson [55] eingegangen wird.

Da durch Mutex-Locks blockierte Prozessorzeit durch explizite Messung von $t_{lock}(n)$ von der Redundanz abgezogen wird, kann davon ausgegangen werden, dass die Redundanz maßgeblich durch Speichereffekte der Nebenläufigkeit verursacht wird. Um die Beiträge zu diesem Effekt genauer zu analysieren, können neben Performance-Countern auch Mikrobenchmarks eingesetzt werden, die am Ende dieses Kapitels exemplarisch kurz vorgestellt werden. Für die Darstellung der parallelen Laufzeit werden die normalisierten Overhead-Zeiten $sched(n) = t_{sched}(n)/(t(n) \cdot n)$, $wi(n) = t_{wi}(n)/(t(n) \cdot n)$ und $lock(n) = t_{lock}(n)/(t(n) \cdot n)$ verwendet. $sched(n)$ und $wi(n)$ umfassen alle Zeiten einer parallelen Sektion, die nicht zur eigentlichen Berechnung von Tasks beiträgt, was auch durch die inverse Definition der Auslastung (engl. *Utilization*) $U(n)$ ausgedrückt wird:

$$U(n) = 1 - [sched(n) + wi(n) + lock(n)] \tag{4.6}$$

$$\Rightarrow t_{total}(n) = \frac{t_0(1)}{n} \cdot \frac{R(n)}{1 - [sched(n) + wi(n) + lock(n)]} + t_{seq} \tag{4.7}$$

Datenlokalität: Wohingegen die vorangegangenen Aspekte hauptsächlich Zeitmessungen berücksichtigten, kann die Verwendung von Performance-Countern beispielsweise die Analyse der Effizienz der Datenverwaltung (Datenlokalität) innerhalb einer parallelen Implementierung ermöglichen. Aufgrund von integrierten Kohärenzprotokollen, die geteilte Daten (die „gespiegelt“ in den privaten Caches mehrerer Prozessoren vorliegen) zwischen den Prozessoren kohärent halten, wird durch fehlende Datenlokalität zusätzlicher Synchronisationsaufwand und damit Inter-Core-Kommunikation für die Aktualisierung der Daten generiert. Besonders Tasks, die auf benachbarten Daten arbeiten oder sogar dieselben Daten teilen, sollten daher nicht über zu lange Distanzen zwischen den Prozessorkernen verteilt werden, damit auszutauschende Daten nur über kurze Distanzen synchronisiert werden müssen. Datenlokalitätsprobleme können somit in parallelisierten Implementierungen durch nicht optimale Task-Verteilungsstrategien verursacht werden und zu Verzerrungen der Task-Ausführung führen.

Speicherallokationen: Abhängig von der Hardware kann ein weiterer relevanter Einfluss auf die parallele Ausführung die Anzahl der (gleichzeitigen) Speicherallokationen darstellen, die in der Regel im Betriebssystem-Kernel nicht parallelisiert sind. Dieser eventuell auftretende Overhead hängt stark vom Aufbau des verwendeten Prozessors ab.

4.2 Profile-Datenverarbeitung

Die hier vorgestellten Parallelisierungseinflüsse werden in der vorgestellten MPAL-Abstraktionsschicht und die den dazugehörigen Werkzeugen automatisiert extrahiert und daraus die im nächsten Abschnitt vorgestellten Parallelisierungsparameter extrahiert. Um höchste Flexibilität für die Analyse von Bottlenecks bei möglichst geringem Overhead zu erreichen, werden alle gespeicherten Zeitstempel zunächst in einer Baumstruktur gespeichert, die der Aufrufstruktur des parallelisierten Programms entspricht. Um den Allokations-Overhead von den Messungen zu extrahieren, wird in einem ersten initialen Durchlauf ein leerer Baum initialisiert. Darüberhinaus wird jede Form von Profiling (hier: zeitliches Trace und Performance-Counter) separat und unabhängig von den anderen durchgeführt, um gegenseitige Einflüsse zu minimieren. Somit erzeugt jeder Messdurchlauf für Timings und Performance-Counter einen separaten Baum derselben Struktur. Das bedeutet, dass Programme, die eine zufällige Aufrufstruktur aufweisen (beispielsweise genetische Algorithmen) nicht für sich wiederholende Messungen geeignet sind. Derartige Programme lassen sich nur entweder unter Verwendung eines einzelnen Messdurchlaufs vermessen, wodurch allerdings der Initialisierungs-Overhead

der Baum-Datenstruktur nicht ausgeklammert werden kann, oder durch das Setzen eines Startwerts für die Zufallszahlengenerierung (engl. *Seed*), um die Aufrufstruktur bei jedem Durchlauf zu reproduzieren.

Messergebnisse können entweder in einer grafischen Oberfläche oder per Kommandozeilen-Interface dargestellt werden. Ebenfalls können automatisiert die oben genannten Parallelisierungseinflüsse analysiert oder die nachfolgend beschriebenen Skalierungsparameter ausgewertet werden. Das erlaubt eine schnelle Identifikation von Bottlenecks und limitierenden Faktoren. Das Messen und Auftragen der Parallelisierungseinflüsse über eine steigende Anzahl von verwendeten Prozessorkernen und die Darstellung in einem Graphen kann zum Beispiel direkt die Task-Ausführungszeiten oder das Auftreten von ungeeigneten Synchronisationen visualisieren und zu einer detaillierten Analyse des parallelen Laufzeitverhaltens beitragen.

Für die Evaluation der Work-Imbalance wird die Differenz zwischen den Beendigungszeiten der jeweils letzten Tasks pro Prozessorkern bis zum letzten ausgeführten Task des Spawns auf allen Kernen akkumuliert. Die nebenläufige Ausführung von mehreren parallelen Task-Teams innerhalb von eingenisteten Spawns kann zu Vermischungen der Tasks auf den Prozessorkernen führen, was die Unterscheidung der jeweiligen Skalierungsparameter erschwert. Des Weiteren kann in manchen Parallelisierungsschemata eine solche Interferenz auch absichtlich zur Erzeugung von Verschachtelungen und zur Reduktion von Wartezeiten eingefügt werden. Daher werden in den Evaluationen in dieser Arbeit alle eingenisteten Parallelisierungen gruppiert und es wird ausschließlich der Spawn auf unterster Ebene interpretiert und ausgewertet. Dadurch wird lediglich die Gewichtung zwischen Task-Switching und Work-Imbalance verändert, nicht aber die Summen der Overhead-Zeiten, wodurch auch die Signifikanz der Messungen erhalten bleibt.

4.3 Modellierung der Skalierungsparameter

Diese Parallelisierungsparameter werden automatisiert von der MPAL-Abstraktionsschicht für eine variierende Anzahl von Prozessorkernen n extrahiert. Dadurch wird besonders das Skalierungsverhalten verdeutlicht, was vollständige Informationen über alle wichtigen Eigenschaften und Charakteristiken der parallelen Implementierung beinhaltet. Die *Skalierbarkeit* beschreibt die Fähigkeit einer untersuchten parallelen Implementierung (engl. *Workload*) die Arbeit über eine steigende Anzahl von Prozessorkernen zu verteilen und gleichzeitig deren Anfälligkeit zusätzlichen Overhead beispielsweise für die Task-Verwaltung zu involvieren. Darüberhinaus lässt sich am Skalierungsverhalten nicht nur die Anfälligkeit für beispielsweise Synchronisations-Overhead

oder Work-Imbalance erkennen, sondern auch die Anfälligkeit auf Effekte, die durch die Verwendung von NUMA-Nodes oder Hyperthreads entstehen. Zusammenfassend werden somit folgende Skalierungsparameter betrachtet:

- *Redundanz*: Prozentualer Anstieg der Summe der Ausführungszeiten aller parallelen Tasks $\sum_i t_i(n)$, der durch Speicher-Begrenzungen und Caching-Effekte entsteht, wobei Lock-Zeiten hier nicht eingerechnet werden.
- *Synchronisationen*: Zeit $t_{lock}(n)$, die ein Task im Wartezustand verbringt in Relation zur insgesamt verfügbaren CPU-Zeit.
- *Work-Imbalance*: Prozentualer Anteil der verfügbaren CPU-Zeit, die aufgrund von ungeeigneter Arbeitsaufteilung nicht zur Verarbeitung von effektiven Tasks verwendet werden kann.
- *Scheduling Overhead*: Anteil der verfügbaren CPU-Zeit, die benötigt wird, um Tasks zu verwalten. Dabei wird berücksichtigt: Task-Erzeugung (engl. *Creation*) $t_{tc}(n)$, Task-Verteilung (engl. *Distribution*) $t_{dist}(n)$, Task-Wechsel (engl. *Switching*) $t_{switch}(n)$ und Task-Synchronisation (engl. *Join*) $t_{join}(n)$.

Diese abstrakten Parameter beinhalten direkte Informationen über das parallele Laufzeitverhalten und die Skalierbarkeit einer parallelen Implementierung,

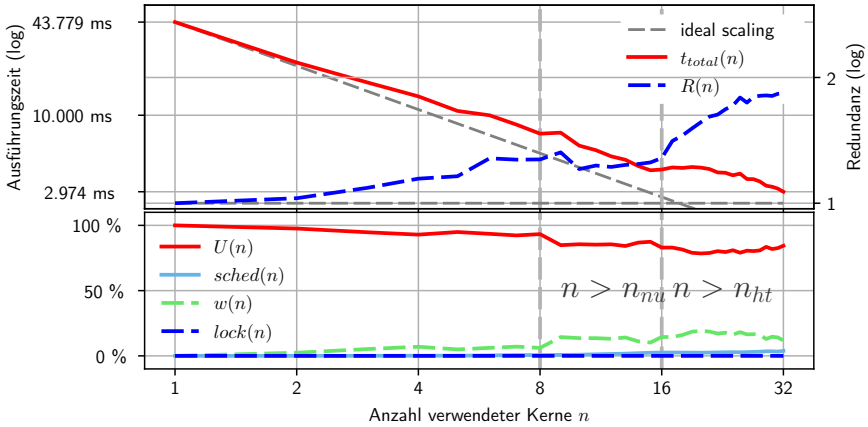


Abbildung 4.2: Exemplarisches Profile, das den Skalierungsverlauf einer parallelen Sektion des SGM-Algorithmus auf einem Xeon E5-2630v3 zeigt.

sowie potentielle Bottlenecks. Ein exemplarisches Profile eines Skalierungsverlaufs ist in Abbildung 4.2 gezeigt. Für eine schnelle Analyse bestimmter Aspekte und einen direkten Vergleich von Profiles, die beispielsweise Unterschiede zwischen Implementierungen oder Unterschiede zwischen Plattformen verdeutlichen, werden im Folgenden Modellfunktionen für den Skalierungsverlauf entwickelt. Es werden die Skalierungsverläufe der in Tabelle 4.1 gezeigten Parameter nachgebildet, wobei der Verlauf jedes Parallelisierungsparameters separat modelliert wird. Die resultierende parallele Gesamtausführungszeit kann wie folgt reproduziert werden:

$$t(n) = \frac{t(1) \cdot R(n)}{n \cdot U(n)} \text{ , mit Utilization } U(n) \tag{4.8}$$

$$U(n) = 1 - sum(l(n), w(n), c(n), d(n), s(n), j(n)) \tag{4.9}$$

Normalerweise werden Software-Threads zu Prozessorkernen nach aufsteigender Core-ID zugewiesen, während die Reihenfolge der Core-IDs zunächst NUMA-Nodes ($n \leq n_{nu}$) vollständig auffüllt, bevor die Cores der anderen NUMA-Nodes verwendet werden (ohne dabei Hyperthreads zu verwenden $n \leq n_{ht}$). Darum wird im Folgenden davon ausgegangen, dass $n_{nu} < n_{ht}$, sodass Effekte, die durch Kommunikation zwischen NUMA-Nodes entstehen, im Skalierungsverlauf früher auftreten, als Hyperthreading-Effekte.

Es zeigt sich, dass alle Verläufe der jeweiligen Skalierungsparameter $p \in \{R, l, w, c, d, s, j\}$ eine einfache Form über steigende n aufweisen und sich durch eine Funktion p mit $p(n) = m \cdot g(n) + b$ beschreiben lassen. Dabei repräsentiert $g(n)$ eine Gradientenfunktion, die in den meisten Fällen einfach $g(n) = n$ ist, und in der hier vorgestellten Konfiguration nur für $d(n)$ die Gaußsche Summe $g(n) = \frac{n \cdot (n-1)}{2}$ verwendet wird. b definiert einen Offset, der für manche Parameter als $b = 0$ angenommen werden kann. Es wird lediglich ein zusätzlicher Offset (plötzlicher Anstieg) und eine Anpassung des Gradienten berücksichtigt,

Tabelle 4.1: Skalierungsparameter eines parallelen Workloads.

Redundanz:	$R(n) =$	$\left[\sum_i t_i(n) - t_{lock}(n)\right] / t(1)$
Synchronisation:	$l(n) =$	$t_{lock}(n)/(t(n) \cdot n)$
Work-Imbalance:	$w(n) =$	$t_{wi}(n)/(t(n) \cdot n)$
Task-Creation:	$c(n) =$	$t_{tc}(n)/(t(n) \cdot n)$
Task-Distribution:	$d(n) =$	$t_{dist}(n)/(t(n) \cdot n)$
Task-Switching:	$s(n) =$	$t_{sw}(n)/(t(n) \cdot n)$
Task-Synchronisation:	$j(n) =$	$t_{join}(n)/(t(n) \cdot n)$

sobald Kommunikationen zwischen NUMA-Nodes oder Hyperthreading-Effekte hinzukommen. Somit kann der Verlauf jedes Skalierungsparameters vollständig durch eine derartige Parametrisierung der Modellfunktion in Gleichung 4.10 mit einem Deskriptor $\vec{s_p}$ aus 6 quantitativen Parametern beschrieben werden. Die Konkatenation dieser deskriptiven Parameter (engl. *Features*) erlaubt einen direkten Vergleich des Laufzeitverhaltens unterschiedlicher Profiles.

$$p(n) = m_{base} \cdot g(n) + b_{base} + nu(n) + ht(n) \tag{4.10}$$

$$nu(n) = \begin{cases} m_{nu} \cdot g(n - n_{nu}) + b_{nu} & n > n_{nu} \\ 0 & else \end{cases} \tag{4.11}$$

$$ht(n) = \begin{cases} m_{ht} \cdot g(n - n_{ht}) + b_{ht} & n > n_{ht} \\ 0 & else \end{cases} \tag{4.12}$$

4.4 Mikrobenchmarks

Wie bereits beschrieben, hat der Aufbau der Speicherhierarchie, sowie die Speicherbandbreite und -latenz einen großen Einfluss auf die Laufzeit paralleler Tasks, was sich wiederum durch Verlängerung der parallelen Tasks auf die Redundanz R auswirkt. Um diesen Einfluss näher zu untersuchen und eventuelle Engpässe zu identifizieren, können Mikrobenchmarks verwendet werden. Auch können Mikrobenchmarks als exemplarische Algorithmen verwendet werden, um eine Plattform gemäß der oben beschriebenen Skalierungsparameter zu charakterisieren und dabei besondere Eigenschaften, speziell unter hoher Belastung, zu testen (z.B. Bus-Bandbreite oder Inter-Core-Kommunikation). Zwei mögliche Benchmarks, die auch in [56] beschrieben und verwendet wurden, um das Kommunikationsverhalten eines ADAS-Algorithmus auf einem Intel Xeon Phi Manycore Prozessor zu modellieren, werden im Folgenden aufgeführt.

Mikrobenchmark Speicherzugriffsverzögerung: Der erste hier gezeigte Benchmark, liest periodisch ein Daten-Array, wobei bei jedem Durchlauf unterschiedlich große Bereiche des Arrays verwendet und bei jedem Lesezyklus die Zugriffszeiten gemessen werden. Abbildung 4.3 zeigt die durchschnittliche Zugriffszeit pro Cache-Line (meistens 64 Byte) über eine variierende Array-Größe. Jeder Lesezyklus (zu einer expliziten Array-Größe) wird mehrfach wiederholt, um einen Durchschnittswert zu erhalten und den Einfluss von Kontrollstrukturen zu minimieren. Die typische Treppenform des Plots wird definiert durch die Größen der jeweiligen Cache-Hierarchie-Ebenen und deren Zugriffsverzögerungen. Bei jedem Überschreiten der Größe einer Cache-Ebene steigt die durchschnittliche Zugriffszeit pro Cache-Line und kennzeichnet die Verzögerungszeit der nächsthöheren Cache-Ebene. Im hier gezeigten Beispiel

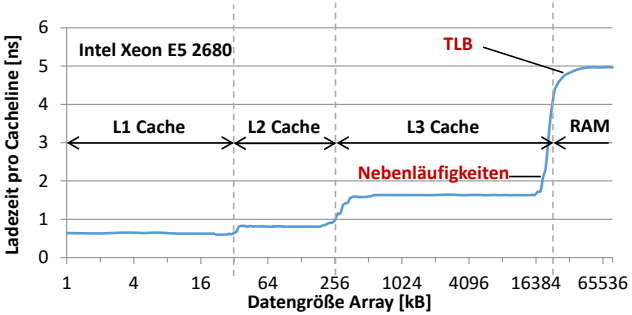
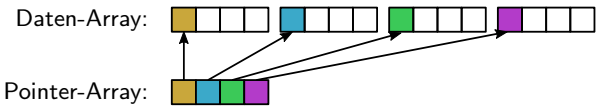


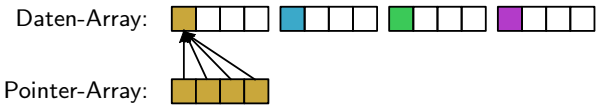
Abbildung 4.3: Ergebnis eines Mikrobenchmarks für die Cache-Hierarchie eines Intel Xeon Prozessors.

eines Intel Xeon E5-2630v3 Prozessors beginnt zunächst der L1-Cache Daten aus dem L2-Cache zu laden, dann der L2-Cache aus dem L3-Cache und anschließend der L3-Cache Daten aus dem Hauptspeicher. Wie zu erwarten ist, ist der Anstieg zwischen den Cache-Ebenen deutlich geringer, als der Sprung zwischen L3-Cache und Hauptspeicher.

Der Speicherzugriff wird mithilfe eines zu Beginn initialisierten Pointer-Arrays realisiert, wobei unterschiedliche Zugriffsmuster realisiert werden können (z.B. zufällige Reihenfolge, fortlaufend, rückwärtslaufend, fortlaufend mit Sprüngen). Dabei wird davon ausgegangen, dass im Ziel-Array pro Cache-Line jeweils maximal ein Element gelesen wird (Abbildung 4.4(a)), wodurch der Einfluss des Auslesens des Pointer-Arrays im Vergleich zum Ziel-Array vernachlässigt werden kann. Um die bereinigten Speicherzugriffsverzögerun-



(a) Pro Cache-Line des Ziel-Arrays ein Zugriff.



(b) Verwendung derselben Adresse des Ziel-Arrays in jedem Element des Pointer-Arrays.

Abbildung 4.4: Speicherzugriffs-Struktur mit Pointer-Array, das in jedem Element eine Adresse des Ziel-Arrays beinhaltet, um durch sequentiellen Durchlauf der Adressen unterschiedliche Zugriffsmuster zu realisieren.

gen zur jeweiligen Speicherebene aus dem Benchmark herauszulesen, muss außerdem der Overhead durch die Pointer-Arithmetik selbst ermittelt und subtrahiert werden. Dazu wird in einer separaten Messung im Pointer-Array in jeder Zugriffsposition dieselbe Speicheradresse initialisiert (Abbildung 4.4(b)), um die Anzahl der Cache-Misses zu minimieren und somit den reinen Einfluss der Pointer-Arithmetik zu messen und entsprechend aus der Gesamtlaufzeit zu eliminieren.

Mikrobenchmark Kern-zu-Kern-Kommunikation: Im zweiten Benchmark, der hier vorgestellt werden soll, wird ein Datum in einem spezifischen Kern vorgeladen (also im privaten L1-Cache dieses Kerns abgelegt) und verändert, wodurch die Cache-Line invalidiert (falsifiziert) wird. Anschließend wird das Datum von benachbarten Prozessorkernen ausgelesen und somit der nötige Overhead des Kohärenzprotokolls provoziert, um die Cache-Line im privaten Cache des benachbarten Kerns zu aktualisieren. Dabei werden Daten in einem Array verwendet, wobei das Array so klein dimensioniert ist, dass es vollständig in den L1-Cache passt, um weitere Cache-Verzögerungen auszuschließen. Da auch die typische Implementierung von Mutex-Locks auf Hardware-Kohärenz zwischen den privaten Caches der Prozessorkernen basiert, bildet dieser Benchmark die tatsächliche Latenz für Synchronisationen sowie für Cache-Misses aufgrund von falsifizierten Cache-Lines ab. Während der Laufzeit des Benchmarks lesen nacheinander alle verfügbaren Prozessorkerne dasselbe Array, wodurch sich durch die ggf. variierende Latenz der logische Abstand der Kerne ablesen lässt. Wie in Abbildung 4.5 zu sehen ist, haben auf dem Intel Xeon 2680 mit zwei NUMA-Nodes besonders benachbarte Kerne auf einem Prozessor-Chip (engl. *Die*) ähnliche Zugriffslatenzen, und der benachbarte Hyperthread hat aufgrund der Verwendung desselben L1-Caches eine deutlich geringere Latenz. Ein Übergang zum zweiten NUMA-Node ist durch einen starken Anstieg der Kommunikationszeit deutlich zu erkennen.

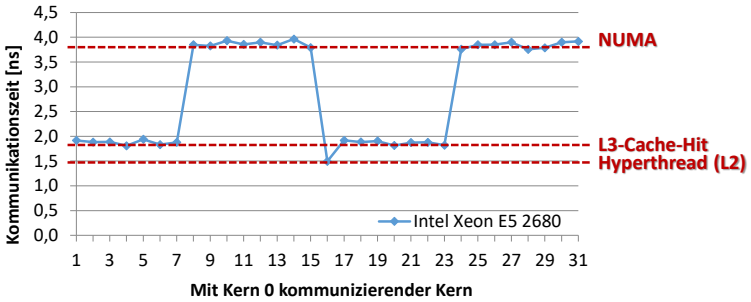


Abbildung 4.5: Ergebnis eines Mikrobenchmarks für die Kern-zu-Kern-Kommunikation eines Intel Xeon Prozessors.

Mit diesen beiden Benchmarks lassen sich bereits die wesentlichsten Einflüsse des Speichernetzwerks auf die Ausführung einer parallelen Implementierung analysieren. In Kombination mit Performance-Countern, die die Auftrittswahrscheinlichkeit eben dieser Speichereffekte wiedergeben, lässt sich nun quantitativ der Einfluss der Speichereffekte genauer beurteilen, wie in [56] gezeigt. Lediglich Reihenfolgen des Speicherzugriffs und beispielsweise Einflüsse von Speicherzugriffs-Vorhersagen (engl. *Prefetching*) und des *Translation Lookaside Buffer* (TLB) werden durch diese Mikrobenchmarks nicht berücksichtigt.

5 Performance-Prädiktion

Das vorherige Kapitel adressiert speziell Unterschiede zwischen Hardware-Eigenschaften und deren Einflüsse auf die Ausführungszeit paralleler Software, sowie Variationen in Parallelisierungs-Strategien und gegenseitige Interferenzen zwischen Anforderungen paralleler Implementierungen und von der Hardware zur Verfügung gestellten Kapazitäten. Daher ist für die Untersuchung der Portabilität von Software besonders die Betrachtung der Performance nach Migration von parallelen Implementierungen auf eine neue bzw. unbekannte Plattform in einer frühen Phase der Software-Entwicklung notwendig und Thema aktueller Forschung. Allerdings ist eine frühe Abschätzung der Performance aufgrund von schwer einzuschätzenden Faktoren, wie superskalaren Prozessor-Pipelines und spekulativen Strategien (z.B. Branch-Prediction oder Prefetching), sowie Caching-Effekten, eine komplexe Aufgabe.

Eine Vielzahl potentieller Bottlenecks und limitierender Faktoren der Hardware, sowie die Implementierungs- und Parallelisierungs-Strategie der Software (z.B. Daten- oder Task-Level Parallelisierung und Task-Granularität) kann die Performance einer Applikation beeinflussen. Nicht nur potentielle Hardware-Einschränkungen, wie Speicherhierarchie und Bandbreite, *Network on Chip* (NoC)-Topologie oder Instruktionssatz-Architektur, müssen betrachtet werden. Auch individuelle Anforderungen der Software, wie Speicheranforderungen oder Synchronisationsverhalten, beeinflussen die parallele Ausführung. Aus diesem Grund ist eine frühe Abschätzung der resultierenden Performance eine notwendige Komponente moderner Tools für die parallele Programmierung im Bereich von Hardware/Software-Co-Design.

In den meisten Fällen werden Prädiktionsszenarien verwendet, die eine der beiden folgenden Perspektiven repräsentieren, die auch in Abbildung 5.1 dargestellt sind.

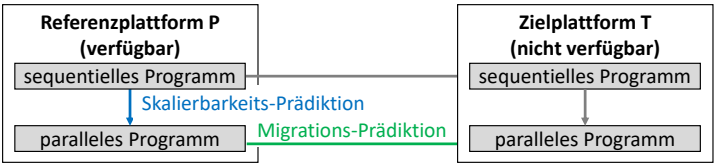


Abbildung 5.1: Szenarien der Performance-Prädiktion: Skalierungsverhalten einer noch nicht parallelisierten Software oder parallele Performance nach einer Migration zwischen Plattformen.

- *Migrations-Prädiktion:* Die parallele Software ist bereits gegeben, aber eine passende Zielplattform wird gesucht – entweder muss eine optimale Plattform aus einem Vorrat existierender Plattformen heraus gesucht werden, oder es wird eine neue Plattform entwickelt.
- *Skalierbarkeits-Prädiktion:* Die Plattform ist gegeben, aber eine noch sequentielle Software soll parallelisiert werden, wozu eine möglichst gut skalierende Parallelisierungs-Strategie gesucht wird.

Existierende Performance-Modellierungstechniken können in die folgenden drei Kategorien eingeteilt werden: *Virtual Prototyping*, *Analytische Modelle* und *Statistische Methoden* [57, 58], die sich jeweils in ihrem Modellierungs- und Simulationsaufwand, sowie in der resultierenden Prädiktionsgenauigkeit unterscheiden, wie in Abbildung 5.2 dargestellt ist.

In diesem Kapitel wird ein neues Konzept eines statistischen Ansatzes für die Prädiktion der Performance von paralleler Software, basierend auf Skalierbarkeitscharakteristiken, vorgestellt [59]. Wie im vorangegangenen Kapitel erwähnt, beschreibt die Skalierbarkeit die Fähigkeit eines Programms die

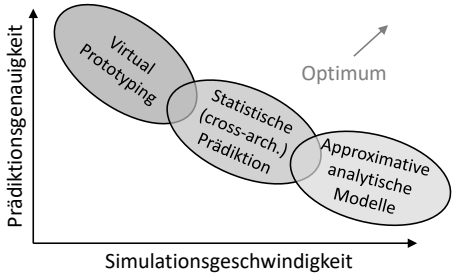


Abbildung 5.2: Prädiktionsgenauigkeit gegenüber Simulationsgeschwindigkeit von unterschiedlichen Prädiktionsmethoden unter der Annahme, dass Hardware- und Software-Modelle bereits existieren.

Arbeit über eine steigende Anzahlen der verwendeten Prozessorkerne gleichmäßig zu verteilen und die Tendenz gleichzeitig Parallelisierungs-Overhead zu generieren. Daraus ergibt sich für jede parallele Implementierung und auf unterschiedlichen Plattformen ein individueller Speedup. Im vorangegangenen Kapitel ist dargestellt, dass alle relevanten Verhaltenscharakteristiken, sowie Bottlenecks, wie Anfälligkeit gegenüber NUMA-Node Distanzen oder zusätzliche Work-Imbalance pro zusätzlich verwendetem Prozessorkern, im Skalierungsgraphen widergespiegelt sind. Darum werden die eingeführten Skalierungsparameter als charakteristische Deskriptoren verwendet, um geeignete Prädiktionskandidaten aus einer vorbereiteten Datenbank zu selektieren und daraus die Ziel-Performance zu rekonstruieren.

Im Gegensatz zu anderen Prädiktionsmethoden basiert dieser Ansatz ausschließlich auf Charakteristiken, die aus Profiles extrahiert werden können, weshalb keine manuelle Modellierung und keine Benutzereingaben erforderlich sind. Außerdem wird in diesem Ansatz, im Gegensatz zu anderen statistischen Methoden, kein Prädiktionsmodell trainiert, sondern es wird die gesamte Datenbank verwendet, um auf Basis von Distanzmaßen zwischen Feature-Vektoren aus geeigneten Prädiktionskandidaten eine gewichtete Interpolation durchzuführen. Es werden Skalierungsparameter $\vec{s}_p$ verwendet, die im vorangegangenen Kapitel beschrieben wurden, die nur ein einfaches mathematisches Modell erfordern, um den gesamten Skalierungsverlauf durch quantitative Parameter zu repräsentieren. Experimente zeigen, dass die Rekonstruktion der gemessenen Skalierungskurven mithilfe des gezeigten mathematischen Modells und unter Verwendung der extrahierten Skalierungsparameter in den meisten Fällen nur einen geringen Fehler von unter 1 % bis 2 % über den gesamten Skalierungsverlauf aufweist. Größere Abweichungen lassen sich meist auf Messungenauigkeiten durch konkurrierende, nicht berücksichtigte Prozesse (z.B. Services des Betriebssystems) zurückführen. Diese Parameter erlauben somit eine relativ präzise Rekonstruktion des Skalierungsverhaltens und bilden ebenfalls komplexe Effekte ab, wie NUMA-Node Distanzen oder die Verwendung von Hyperthreads. Während nachfolgend auch ein Benchmark-Set vorgestellt wird, womit Plattformen charakterisiert und dessen parallele Implementierungen als Prädiktionskandidaten verwendet werden können, basiert der neue statistische Prädiktionsansatz maßgeblich auf folgenden Schritten:

1. Extraktion von Feature-Vektoren aus gemessenen Profiling-Informationen: Fitting der Skalierungsparameter-Kurven
2. Selektion der Prädiktionskandidaten aufgrund von Distanz-Metriken
3. Rekonstruktion (Prädiktion) der Ziel-Performance durch gewichtete Interpolation

5.1 Prädiktionsansätze

Die Modellierung eines komplexen Systems und dessen Prädiktion ist eine schwierige Aufgabe in allen Forschungsbereichen. Systemmodelle müssen entweder eine enorme Komplexität aufweisen, um alle oder zumindest die meisten relevanten Teilaspekte abzudecken, oder sie vernachlässigen durch Abstraktion potentiell wichtige Einflüsse [60]. Eine Grundvoraussetzung bei der Systemmodellierung und für die Definition eines geeigneten Abstraktionslevels ist die Identifikation eines Sets von Grundparametern (Principle Components), die eine möglichst maximale Korrelation mit dem Systemverhalten aufweisen. Allerdings beinhalten viele Systeme Parameter, die gegenseitig interferieren, was den Modellierungsprozess erschwert – insbesondere wenn das System nicht bekannt ist und lediglich als Blackbox gegeben ist. Sofern eine gegebene Datenbank aus ausreichend vielen Daten besteht, können Machine-Learning Algorithmen (z.B. Deep Learning) verwendet werden, die automatisch Features, Grundparameter und Korrelationen extrahieren. Im Bereich der Prozessor- und Software-Modellierung ist allerdings die Anzahl der gegebenen Daten meist durch die hohen Kosten für Hardware-Plattformen und den hohen Entwicklungsaufwand für neue parallele Software limitiert. Daher verwenden selbst statistische Methoden für die Multicore-Performance-Prädiktion oft traditionelle (manuell erstellte) Modelle zur Feature-Extraktion. Anstatt beliebige Workloads auf allen Prozessor-Varianten zu prädizieren fokussieren sich daher einige Ansätze auf die Prädiktion von Basisblöcken (z.B. *Stencil Codes*).

Existierende Prädiktionsmethoden können in drei Kategorien unterteilt werden und erfordern bislang zumindest in geringem Maße Modellierungsaufwand (Benutzereingaben): *Analytische Methoden* verwenden mathematische Modelle, die die architekturellen Leistungsmerkmale, wie z.B. Cache-Größe oder Hauptspeicherbandbreite, sowie Software-Charakteristiken, wie Datenlokalität oder Branch-Distanzen, nachbilden. Analytische Methoden bieten eine gute Modellierungs- und Simulationsgeschwindigkeit, erlauben aber derzeit nur Prädiktionen mit moderater Präzision. *Virtual Prototyping* ist ein Ansatz, bei dem ein vorhandenes compiliertes Software-Binary auf in Software simulierter Hardware ausgeführt wird. Dazu ist entweder eine voll detaillierte Modellierung der ausführenden Hardware erforderlich, oder eine abstrakte Simulation wird potentiell wichtige Informationen über das zeitliche Verhalten ignorieren. *Statistische Methoden* können auch als Machine-Learning Ansätze angesehen werden, da diese auf einer Datenbank aus gegebenen Performance-Messwerten (z.B: Profiles) basieren, woraus anschließend Performance-Werte prädiziert werden. Während diese Ansätze potentiell eine lange Trainingsphase benötigen, ist die Prädiktionsgenauigkeit maßgeblich von der Qualität der Datenbank und der Features abhängig.

5.1.1 Virtual-Prototyping

Virtual-Prototyping emuliert eine volle funktionale Hardware-Plattform in einer Software-Simulation, die kompilierte Software-Binaries und auch vollständige Betriebssysteme ausführen kann. Während sich diese Plattformen in ihrer Abstraktion unterscheiden können, bietet eine zeitlich weniger präzise, funktional orientierte Simulation einen Speedup bei gleichzeitig geringerer Prädiktionsgenauigkeit. Modulare Systemkomponenten, wie Speicherkontroller, CPUs oder NoCs, sind in System-Level-Design Sprachen, wie dem Quasi-Standard SystemC für C++, oder Hardware-Beschreibungssprachen, wie *Very High Speed Integrated Circuit Hardware Description Language* (VHDL), implementiert. Um Interaktionen zwischen Modulen auf unterschiedlichen Abstraktionsebenen zu modellieren, bietet *Transaction Level Modeling* (TLM) einen Kommunikations-zentrierten Ansatz.

Auf Basis von Grundlagen, wie TLM und SystemC, bieten existierende Virtual-Prototyping-Frameworks, wie *General Execution-driven Multiprocessor Simulator + M5* (GEM5) [61], *Imperas Open Virtual Platforms* (OVP) [62], *Quick Emulator* (QEMU) [63] oder *Cadence Virtual System Platform* (VSP) [64], auch vollständige *out-of-the-box* verwendbare virtuelle Plattformen. QEMU bietet zum Beispiel die Möglichkeit in kurzer Zeit eine Emulation zu konfigurieren, die unterschiedliche Instruktionssatz-Architekturen, sowie vollständige Plattformen, wie den RaspberryPi oder das Xilinx Ultrascale+ simuliert. QEMU verwendet ein hohes Abstraktions-Level (funktionale Simulation) und erlaubt dafür eine schnelle Evaluation und Verifikation der Funktionalität einer Applikation. Allerdings gibt es auch Ansätze, die zeitliche Informationen in Form von Timing-Datenbanken in die QEMU-Simulation integrieren oder TLM und SystemC-Module verwenden, um eine zeitlich präzisere Simulation durchführen zu können.

GEM5 ist ein Open-Source Framework, das sowohl im wissenschaftlichen wie im industriellen Bereich weit verbreitet ist. GEM5 verwendet ein modular erweiterbares System für das Virtual Prototyping. Einige verwendbare Komponenten werden sogar direkt von IP-Core-Herstellern, wie ARM, zur direkten Verwendung zur Verfügung gestellt. Während die Simulatoren für einzelne Komponenten (z.B. Instruktionssatz-Simulatoren für CPU-Cores) oft in C++ implementiert sind, kann ein Virtual Prototype in GEM5 einfach über eine Python-Schnittstelle konfiguriert werden (Zusammenstellung der Teilkomponenten), wobei sogar heterogene CPU-Cluster oder GPU-beschleunigte Plattformen erstellt werden können. Eine so konfigurierte Plattform repräsentiert ein voll funktionsfähiges System, sodass Software entweder *bare-metal* oder mit einem Betriebssystem, wie Linux oder Android, direkt ausgeführt werden kann. GEM5 verwendet ein eigenes, TLM-ähnliches System für

Tabelle 5.1: Überblick verschiedener Frameworks für Virtual-Prototyping.

Framework	Abstraktion	Prozessor-Architekturen
QEMU [63]	funktionell	ARM, x86, microblaze, ...
ARM FVP [66]	funktionell	ARM
Imperas OVP [62]	Instruktions-level	ARM, MIPS, POWERPC, RISCV, ...
GEM5 [61]	funktionell oder Zyklen	ARM, MIPS, x86, ...
Cadence VSP [64]	funktionell oder Instruktionen	ARM, MIPS, Renesas, PowerPC

die Kommunikation zwischen den Komponenten, kann aber auch auf TLM zurückgreifen, sowie mit SystemC-Komponenten erweitert werden. Butko et al. [65] präsentierten ein Fallbeispiel für die Prädiktion der Performance einer ARM big.LITTLE-Plattform (Cortex-A15 + Cortex-A7) mit einem durchschnittlichen Prädiktionsfehler von 20 %. Tabelle 5.1 zeigt einen Überblick über verschiedene Frameworks für das Virtual-Prototyping.

5.1.2 Analytische Modelle

Analytische Prädiktionen verwenden separate Modelle der Hardware-Eigenschaften und Software-Charakteristiken [67], die anschließend in einer mathematischen Evaluation kombiniert werden. Carlson et al. [68] haben ein mechanistisches Prozessormodell für analytische Prädiktionen evaluiert, die normalerweise auf architekturunabhängigen Charakterisierungen des Workloads basieren (z.B: Cache-Reuse-Histograms [69]). In anderen Prädiktionsansätzen werden auch weitere charakteristische Software-Metriken verwendet, wie Sprungwahrscheinlichkeiten oder durchschnittliche Anzahl der Schleifendurchläufe, die beispielsweise vom *Low-Level Virtual Machine* (LLVM)-Compiler extrahiert werden können. Van den Sten et al. [70] prädizierten mit Ihrem Ansatz eine Single-Thread-Performance mit einem durchschnittlichen Fehler von 13 %. Der Ansatz wurde anschließend für Multicore-Prädiktion von De Pestel et al. [71] erweitert. Ein anderes analytisches Prozessormodell von Jongerius et al. [72] (eingesetzt im IBM ExaBounds Framework) prädiziert Leistungsaufnahme und Performance und wurde mit einem Xeon E5-2697 v3 und einem ARM Cortex-A15 evaluiert. Mit diesem Modell konnte die Single-Thread-Performance mit einem Fehler von 59 % und der Einfluss von Nebenläufigkeiten mit einem zusätzlichen Fehler von 11 % prädiziert werden.

IBM ExaBounds verwendet ebenfalls plattformunabhängige Software-Profiles, die mit LLVM extrahiert werden. Das Framework bringt ein kleines Set von vorkonfigurierten Prozessormodellen mit, wobei das Hinzufügen von eigenen Prozessormodellen für die Erstellung einer passenden Parametrisierung detaillierte Kenntnisse über die Architektur erfordert. ExaBounds erzeugt umfangreiche Prädiktionen der Laufzeiteigenschaften, *Cycles per Instructi-*

Tabelle 5.2: Überblick über verschiedene analytische Prädiktionsmethoden.

Framework	Prozessor-Architekturen
ExaBounds (nach Jongerius et al. [72])	x86
SLX	x86, ARM
LLVM MCA	x86, ARM, MIPS, ...
nach Carlson et al. [68]	x86, ARM
nach de Pestel et al. [71]	x86, ARM

on (CPI), Cache-Miss-Raten und einiges mehr. Ein ähnlicher Ansatz wird von Silexica entwickelt und in die SLX-Parallelisierungs-Werkzeuge integriert. SLX prädiziert ebenfalls die parallele Performance aus LLVM-basierten Software-Profiles und einem XML-Prozessormodell. Zusätzlich zur Performance-Prädiktion schlägt das SLX-Werkzeug aussichtsvolle Parallelisierungen, Vektorisierungen und Offload-Strategien auf Basis von einer sequentiellen Implementierung für jede bereits modellierte Architektur vor. LLVM bietet nicht nur plattformunabhängige Profiles zur Verwendung in Prädiktionsmethoden von Drittanbietern, sondern integriert auch selbst die *Machine Code Analyzer* (MCA)-Prädiktionsmethode ab LLVM 7. Das MCA-Werkzeug verwendet ebenfalls dieselben plattformunabhängigen Profile-Informationen des LLVM-Compilers und fügt ein Modell einer Prozessor-Pipeline hinzu, um das parallele Laufzeitverhalten eines Programms zu prädizieren. Eine Zusammenfassung von analytischen Prädiktionsmethoden ist in Tabelle 5.2 gezeigt.

5.1.3 Statistische Methoden

Bisherige statistische Prädiktionen verwenden Datenbanken aus zuvor extrahierten Laufzeitmetriken, aus denen separate deskriptive Feature-Vektoren für die Software (Workload) und die Hardware-Plattform gewonnen werden, um damit Machine-Learning-Algorithmen zu trainieren (siehe Abbildung 5.3). Darum ist für die Prädiktionspräzision die Identifikation von geeigneten Deskriptoren, wie beispielsweise Performance-Counter oder Registerzugriffsverhalten, essentiell. Statistische Prädiktionsmethoden beruhen somit stark auf der Signifikanz von Deskriptoren, wobei extrahierte Metriken, wie Performance-Counter, aufgrund von Variationen zwischen Architekturen teilweise eine schlechte Interpretierbarkeit aufweisen. Aus diesem Grund fokussieren einige Ansätze die Prädiktion auf spezifische Architekturfamilien, wie beispielsweise GPUs [73, 57].

In [74] verwenden Ardalani et al. Mikroarchitektur-unabhängige Charakteristiken, wie deskriptive Workload-Features aus dem CPU-Code (siehe [75]), um die Performance der korrespondierenden, aber noch nicht existenten GPU-Implementierung zu prädizieren. In dieser Methode wird ein Set aus Ma-

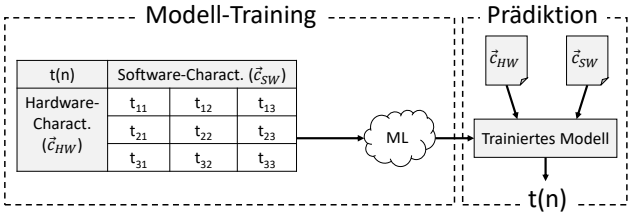


Abbildung 5.3: Statistische Prädiktionsmethoden unter Verwendung eines durch Machine-Learning (ML) trainierten Modells.

chine-Learning Algorithmen in einer Ensemble-Learning-Technik [76] für die Prädiktion eingesetzt, womit ein durchschnittlicher Fehler von 27 % für eine Maxwell- und 36 % für eine Keppler-Architektur erreicht wurde. Ein weiterer Ansatz verwendet Code-Skelette (*Skeletons*) des CPU-Codes um die entsprechende GPU-Performance zu prädizieren [77]. Wan et al. [78] verwenden Benchmark-Ergebnisse und Prozessor-Klassifikationen aus zwei herkömmlichen CPU-Benchmarks *SPEC* und *Geekbench*, um die Performance hauptsächlich für x86-Architekturen zu prädizieren. Diese Benchmarks bieten hauptsächlich Single-Thread-Codes. Es wurde eine *Principle Component Analysis* (PCA) durchgeführt, um aussagekräftige Deskriptoren zu identifizieren. Unter Verwendung von Deep-Learning, das bessere Ergebnisse lieferte als eine lineare Regression, wurde ein durchschnittlicher Fehler von 5 % (SPEC) bzw. 11 % (Geekbench) für die Prädiktion einer neuen Prozessorarchitektur und 26 % (SPEC) bzw. 14 % (Geekbench) für die Prädiktion eines neuen Workloads erreicht. Einen Überblick über verschiedene statistische Prädiktionsmethoden ist in Tabelle 5.3 gezeigt.

Tabelle 5.3: Überblick über verschiedene statistische Prädiktionsmethoden.

Framework	Prozessor-Architekturen
nach Ardalani et al. [74]	CPU (sequentiell) zu GPU
nach Meng et al. [77]	CPU (skeletons) zu GPU
nach Wang et al. [78]	Intel x86-CPU
Zusammenfassungen [73] und [57]	GPU
Ansatz dieser Arbeit	x86, ARM

5.2 Skalierbarkeitsbasierter statistischer Ansatz

Bisherige statistische Ansätze basieren auf separaten Features $\vec{c}_{HW}$ einer Plattform P und $\vec{c}_{SW}$ eines algorithmischen Workloads A, sodass die parallele

Ausführungszeit von A auf P ausgedrückt werden kann als:

$$t = f(\vec{c}_{HW}, \vec{c}_{SW}). \tag{5.1}$$

Im Gegensatz dazu wird in diesem Ansatz nicht nur die Laufzeitcharakteristik unter Verwendung aller verfügbaren Prozessorkerne betrachtet, sondern der vollständige Skalierungsverlauf mit einbezogen. Darum wird der Workload A unter variierenden Anzahlen von verwendeten Prozessorkernen n gemessen, wobei die Messung bei einer sequentiellen Ausführung $t(n = 1)$ startet und die Anzahl der verwendeten Prozessorkerne bis zur maximalen Anzahl verfügbarer Prozessorkerne $n = n_{max}$ erhöht wird. Damit entsteht eine Laufzeitcharakteristik der parallelen Implementierung, die von n abhängt (Skalierungsverhalten):

$$t(n) = f(\vec{c}_{HW}, \vec{c}_{SW}, n). \tag{5.2}$$

Da das parallele Laufzeitverhalten durch eine Vielzahl von Effekten beeinflusst wird, wie bereits im vorangegangenen Kapitel beschrieben, wird die Gesamtausführungszeit nicht in einem Schritt prädiziert. Stattdessen wird die Gesamtausführung in die vorgestellten abstrakten Parallelisierungs-Effekte, wie *Work-Imbalance* oder *Scheduling-Overhead*, aufgeteilt und der Trend jedes Parameters über den gesamten Skalierungsverlauf separat modelliert bzw. prädiziert. Dabei werden keine explizit separierten Features für Hardware und Software, sondern aus Profiles extrahierte Features verwendet, die

- Hardware- und Software-Einflüsse beinhalten,
- eine abstrakte *verhaltensorientierte* Perspektive repräsentieren,
- potentielle Bottlenecks direkt anzeigen und
- ohne Modellierungsaufwand aus Profiles extrahiert werden können.

Andere statistische Prädiktionsmethoden verwenden eine Datenbank mit Charakteristiken und Performance-Metriken, womit ein Modell trainiert wird. Im Gegensatz dazu wird in diesem Ansatz bei jedem Prädiktionsprozess die Datenbank als Ganzes verwendet, um aus den existierenden Daten auf Basis von Distanzmaßen Prädiktionskandidaten zu selektieren und zwischen allen selektierten Kandidaten durch eine interpolative Transformation die Performance der Ziel-Platform (engl. *Target*) zu prädizieren. Die explizite Charakterisierung einer Plattform ist damit definiert als die Kombination aller einzelnen Charakteristiken mehrerer Benchmarks, die auf einer Plattform vermessen wurden beziehungsweise die Charakterisierung eines Workloads als die Kombination aller einzelnen Charakteristiken eines Workloads, die auf verschiedenen Plattformen vermessen wurden.

Als charakteristische, deskriptive Features werden die im vorherigen Kapitel vorgestellten und in Tabelle 4.1 zusammengefassten Parameter verwendet. Wie im folgenden Kapitel beschrieben, werden die Feature-Vektoren aus den quantitativen Skalierungsparametern zusammengesetzt, die durch Parametrisierung der Modellfunktionen entstehen. Außerdem werden zusätzlich Performance-Counter aus den Profiles dem Feature-Vektor angehängt. Die folgenden Abschnitte beschreiben den gesamten Ablauf der Prädiktion, sowie die einzelnen Teilschritte der Prädiktion:

1. Extraktion der Feature-Vektoren (aus Skalierungskurven)
2. Selektion von Prädiktionskandidaten basierend auf Distanzmaßen
3. Prädiktion durch Rekonstruktion mit gewichteter Interpolation

5.2.1 Deskriptoren

Der gesamte Prädiktionsprozess ist in Abbildung 5.4 dargestellt und basiert auf einer Datenbank, die charakteristische Profile-Informationen von Referenz- und zu prädizierender Zielplattform beinhaltet. Um Plattformen zu charakterisieren, wurde eine Sammlung von Implementierungen (Workloads) als Benchmarks B_i vorbereitet, wobei jede Implementierung eine individuelle Charakteristik besitzt (z.B. unterschiedliches Speicherzugriffsverhalten, Parallelisierungs-Strategien, etc.). Auf einer Referenzplattform P, die bereits charakterisiert wurde (durch

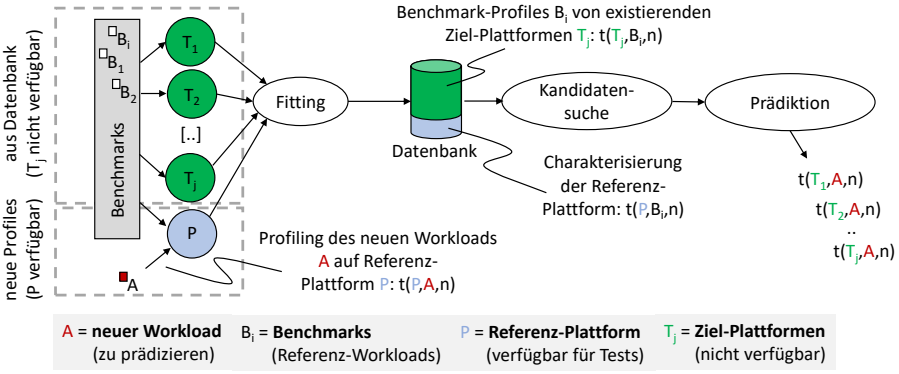


Abbildung 5.4: Prädiktionsprozess, wodurch die Performance eines neuen Workloads, der zuvor auf einer Referenz-Plattform gemessen wurde, auf einer vorher durch Benchmarks vermessenen aber nicht verfügbaren Target-Plattform prädiziert werden kann.

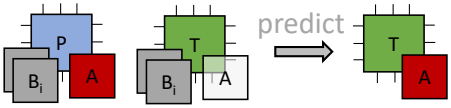
Profiling die Benchmarks B_i vermessen wurden), kann ein neuer Workload A getestet werden, um anschließend dessen Performance auf jeder anderen bekannten Target-Plattform T_j aus der Datenbank zu prädizieren. Konsekutive parallele Abschnitte (beispielsweise durch *Barriers* oder durch explizite Spawns voneinander getrennt) werden als separate Workloads betrachtet, da diese zumeist auch unterschiedliche Charakteristiken aufweisen.

Die Prädiktion lässt sich in zwei Teilschritte unterteilen, die ebenso separat verwendet werden können: (1) Prädiktion der sequentiellen Ausführungszeit $t(1)$ und (2) Prädiktion des Skalierungsverhaltens $\vec{sc}$. Damit werden die folgenden Prädiktionsszenarien ermöglicht, die auch in Abbildung 5.5 dargestellt werden:

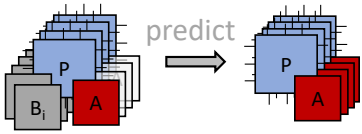
- **Performance-Prädiktion** eines vorhandenen Workloads A, vermessen auf der Referenzplattform P, für eine weitere bereits für die Datenbank vermessene Target-Plattform T, die derzeit nicht vorhanden bzw. verfügbar ist ($t_{P,A}(n) \rightarrow t_{T,A}(n)$). Das beinhaltet die Prädiktion der sequentiellen Ausführungszeit von $t(1)$, sowie des gesamten Skalierungsverlaufs, dargestellt als $\vec{sc}$.
- **Skalierbarkeits-Prädiktion** einer noch nicht parallelen Implementierung A auf einer vorhandenen Architektur P unter Verwendung der Referenzen aus der Datenbank ($t_{P,A}(1) \rightarrow t_{P,A}(n)$). Das Anwenden von Skalierungseigenschaften der existierenden Benchmarks, die nach Parallelisierungs-Strategien klassifiziert wurden, kann einen Ausblick auf den Parallelisierungserfolg einer noch sequentiellen Implementierung geben. Da Profiles von sequentiellen Workloads keine Informationen über die Skalierbarkeit liefern (sondern nur Performance-Counter), ist die Skalierungsprädiktion allerdings limitiert.
- **Prädiktion eines virtuellen Workloads** V, wobei die frei wählbaren Parameter beispielsweise *Speicherverwaltung*, *Synchronisationsverhalten* oder *Arbeitsaufteilung* sind ($t_{P,A}(n) \rightarrow t_{P,V}(n)$).

Da eine Plattform-Charakterisierung im Wesentlichen aus einem Satz von getesteten Benchmarks besteht, ist eine virtuelle Plattform nur schwer zu prädizieren. Freie Parameter für eine solche virtuelle Plattform sind diese, die direkt im Modell des Skalierungsverhaltens von $t(n)$ angewendet werden: n_{nu}, n_{ht}, n_{max}.

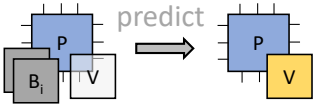
Als charakteristische Merkmale (engl. *Features*) des Skalierungsverlaufs werden die zuvor vorgestellten Skalierungsparameter $p \in \{R, l, w, c, d, s, j\}$ verwendet. Für die Erzeugung eines deskriptiven *Skalierungsvektors* $\vec{sc}$ (Spaltenvektor) werden die durch Fitting erzeugten Parametervektoren $\vec{s}_p$ (ebenfalls Spaltenvektoren), sowie aus den Profiles gewonnene Performance-Counter $\vec{pc}$



(a) Migrations-Prädiktion ($t(n)$) von A: von P zu T).



(b) Skalierbarkeits-Prädiktion (A auf P: von $t(1)$ zu $t(n)$).



(c) Prädiktion virtueller Workloads ($t(n)$) auf P: von B_i zu V).

Abbildung 5.5: Prädiktionsszenarien, die durch die vorgestellte statistische Prädiktionsmethode ermöglicht werden.

konkateniert, wie in Gleichung 5.4 beschrieben. Außerdem werden nicht nur die Performance-Counter selbst verwendet, sondern auch daraus abgeleitete Laufzeiteigenschaften, wie *L1-Load-Misses/Instructions*. Einige Elemente des Vektors $\vec{sc}$ können als *nicht valide* gekennzeichnet werden, da nicht alle Plattformen beispielsweise mehrere NUMA-Nodes oder Hyperthreads bieten, oder die angefragten Performance-Counter nicht bereitstellen. In diesem Fall werden die nicht verfügbaren Elemente gekennzeichnet und von den weiteren Berechnungen ausgenommen.

$$\vec{s}_p = [m_{base}, b_{base}, m_{nu}, b_{nu}, m_{ht}, b_{ht}]^\mathsf{T} \tag{5.3}$$

$$\vec{sc} = [\vec{s}_R^\mathsf{T}, \vec{s}_l^\mathsf{T}, \vec{s}_w^\mathsf{T}, \vec{s}_c^\mathsf{T}, \vec{s}_d^\mathsf{T}, \vec{s}_s^\mathsf{T}, \vec{s}_j^\mathsf{T}, \vec{pc}^\mathsf{T}]^\mathsf{T} \tag{5.4}$$

5.2.2 Distanzmaße

Der Prädiktionsprozess startet mit einer Suche nach Prädiktionskandidaten basierend auf Distanzen, die zwischen Skalierungsvektoren ermittelt werden. Zunächst werden die einzelnen Elemente (Dimensionen) des Vektors $\vec{sc}$ (skalares Element sc_m mit Dimensions-ID m) innerhalb der jeweiligen Dimension über

alle in der Datenbank vorhandenen Skalierungsvektoren auf den Wertebereich $[0, 1]$ normalisiert. Es wird lediglich das globale absolute Minimum und Maximum zu jeder Dimension gespeichert, um den finalen Ziel-Skalierungsvektor rekonstruieren zu können.

$$max_m = max(sc_m(P, A), \{sc_m(T_j, B_i)\}) \tag{5.5}$$

$$min_m = min(sc_m(P, A), \{sc_m(T_j, B_i)\}) \tag{5.6}$$

$$sc_{m,norm} = (sc_m - min_m)/(max_m - min_m) \tag{5.7}$$

Allgemein wird die Distanz $\mathcal{D}$ zwischen zwei Vektoren $\vec{sc}_\alpha$ und $\vec{sc}_\beta$ als geometrische Distanz definiert, die allerdings mit der maximal möglichen Distanz in Abhängigkeit der jeweils gültigen Dimensionen normiert wird, sodass $0 \leq \mathcal{D} \leq 1$. Zunächst wird die L2-Norm zur Bestimmung der geometrischen Distanz verwendet, wobei Dimensionen ignoriert werden, in denen einer der beiden oder beide Vektoren als *nicht valide* gekennzeichnet sind. Die Anzahl der gültigen Dimensionen, in denen keiner der beiden Vektoren als *nicht valide* gekennzeichnet ist, wird im Folgenden als $dim(\vec{sc}_\alpha, \vec{sc}_\beta)$ geschrieben. Daher ist die maximale Distanz in einem Raum, in dem der Wertebereich aller Dimensionen auf 1 beschränkt ist, $\sqrt{dim(\vec{sc}_\alpha, \vec{sc}_\beta)}$.

$$\mathcal{D}(\vec{sc}_\alpha, \vec{sc}_\beta) = \frac{\|\vec{sc}_\alpha - \vec{sc}_\beta\|_2}{\sqrt{dim(\vec{sc}_\alpha, \vec{sc}_\beta)}} \tag{5.8}$$

Das nachfolgende Beispiel zeigt die Berechnung der Distanz aus zwei hypothetischen Minimal-Skalierungsvektoren, die nur aus den Basis-Parametern der Redundanz ($R_{m,base}$, $R_{b,base}$), sowie den LLC-Misses (LLC-M) als Performance-Counter-Messwert bestehen:

$$\begin{pmatrix} R_{m,base} \\ R_{b,base} \\ LLC-M \end{pmatrix} \quad \vec{sc}_\alpha = \begin{pmatrix} 0,01 \\ 0,04 \\ 0,20 \end{pmatrix}, \quad \vec{sc}_\beta = \begin{pmatrix} 0,20 \\ 0,01 \\ inv. \end{pmatrix} \tag{5.9}$$

In diesem Beispiel sind die LLC-M-Werte in $\vec{sc}_\beta$ nicht verfügbar (als nicht valide gekennzeichnet), wodurch sich die Anzahl der berücksichtigten Dimensionen für die Distanzberechnung auf zwei beschränkt, sodass

$$\mathcal{D}(\vec{sc}_\alpha, \vec{sc}_\beta) = \frac{\sqrt{(-0,19)^2 + (0,03)^2}}{\sqrt{2}} = 0,136. \tag{5.10}$$

Im Folgenden wird die Distanz zwischen Skalierungsvektoren zweier Workloads A und B, die beide auf der Plattform P gemessen wurden, als $\mathcal{D}_a(P, A, B)$ und die Distanz zwischen Skalierungsvektoren desselben Workloads A gemessen

auf den Plattformen P und T, als $\mathcal{D}_p(A, P, T)$ definiert. Eine mittlere Distanz $\mathcal{D}_a(A, B)$ zwischen Workloads A und B kann durch den Durchschnitt der Workload-Distanzen auf allen Plattformen T_j, auf denen die beiden Workloads ausgeführt wurden, ausgedrückt werden. Die mittlere Distanz $\mathcal{D}_p(P, T)$ zwischen Plattformen P und T kann durch die durchschnittliche Distanz über alle Workloads B_i aus der Datenbank ausgedrückt werden, die auf beiden Plattformen ausgeführt wurden.

$$\mathcal{D}_a(P, A, B) = \mathcal{D}(\vec{sc}(P, A), \vec{sc}(P, B)) \tag{5.11}$$

$$\mathcal{D}_p(A, P, T) = \mathcal{D}(\vec{sc}(P, A), \vec{sc}(T, A)) \tag{5.12}$$

$$\mathcal{D}_a(A, B) = mean(\mathcal{D}_a(T_j, A, B)) \tag{5.13}$$

$$\mathcal{D}_p(P, T) = mean(\mathcal{D}_p(B_i, P, T)) \tag{5.14}$$

5.2.3 Kandidatenselektion

Ist ein Satz von Referenz-Profiles in der Datenbank gegeben (Benchmarks B_j), werden anschließend die vielversprechendsten Kandidaten für die Prädiktion des Workloads A auf einer Ziel-Plattform T aufgrund der geringsten Distanzen selektiert. Wurde A lediglich auf einer Referenz-Plattform P vermessen, können Prädiktionskandidaten auf Basis von $\mathcal{D}_a$ selektiert werden. Werden mehrere Referenz-Plattformen P_x bereitgestellt, wie in Abbildung 5.6 gezeigt, wodurch grundsätzlich die Prädiktionsgenauigkeit gesteigert werden kann, wird ebenfalls $\mathcal{D}_p$ zur Kandidaten-Selektion berücksichtigt. Bei der Kombination der Algorithmen- mit der Plattform-Distanz zu einer *Tupel-Distanz* $\mathcal{D}_t$ werden Distanzen als inverse Ähnlichkeiten (Wahrscheinlichkeiten) interpretiert, sodass $\mathcal{D}_t$ wie folgt berechnet wird:

$$\mathcal{D}_t = 1 - [(1 - \mathcal{D}_p) \cdot (1 - \mathcal{D}_a)]\,. \tag{5.15}$$

Tupel können entweder durch die Verwendung von jeweils individuellen oder aber durchschnittlichen Distanzen für Algorithmen und Plattformen gebildet werden, wie in Abbildung 5.6 visualisiert wird. Potentielle Kandidaten zur Prädiktion werden anschließend durch eine minimale *Tupel-Distanz* ausgewählt. Eine Referenzplattform P kann beispielsweise zunächst durch die durchschnittliche Plattform-Distanz $\mathcal{D}_p(P, T)$ ausgewählt werden, wogegen die Workload-Kandidaten durch individuelle Distanzen $\mathcal{D}_a(P, A, B_i)$ auf der zuvor selektierten Plattform P ausgewählt werden. Um spezifische Eigenheiten von einzelnen Prädiktionskandidaten zu eliminieren, können auch mehrere Benchmarks als Prädiktionskandidaten ausgewählt werden, wie im nachfolgenden Abschnitt erklärt wird.

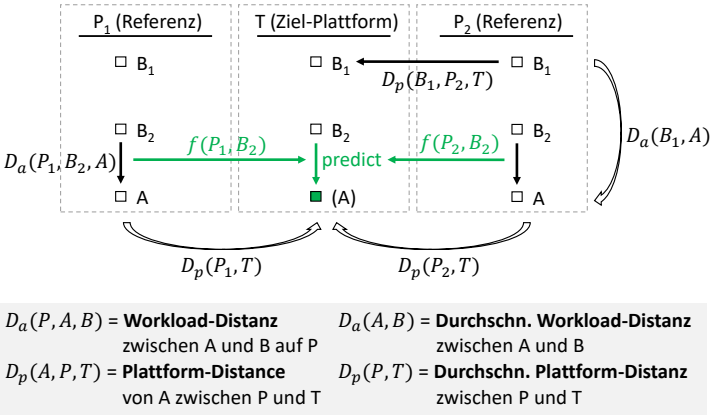


Abbildung 5.6: Prädiktionsszenario unter Verwendung von zwei Referenz-Plattformen und zwei ausgewählten Benchmarks als Kandidaten. Es sind workload- und plattformindividuelle Distanzen, sowie Durchschnittswerte markiert.

5.2.4 Rekonstruktion

Wurde lediglich ein Prädiktionskandidat selektiert (B vermessen auf P und T), um den Workload A auf der Zielplattform T zu prädizieren, kann eine einfache, gewichtete Transformation angewendet werden, um entweder die sequentielle Ausführungszeit $t(1)$ oder die jeweiligen Elemente sc_m des Skalierungsvektors $\vec{sc}$ zu rekonstruieren. Da die Prädiktion von $t(1)$ und der Elemente sc_m denselben Regeln folgt, wird im Weiteren nur die Transformation der Elemente des Skalierungsvektors sc_m beschrieben. Um ein Element sc_m zu prädizieren, wird ein Transformationsfaktor f zwischen A und B, jeweils auf P vermessen, verwendet und daraus das Ziel-Element abgeleitet.

$$sc_m(T, A) = sc_m(T, B) \cdot f \text{ , with } f = \frac{sc_m(P, A)}{sc_m(P, B)} \tag{5.16}$$

In dem Fall, dass mehrere Kandidaten zur Prädiktion ausgewählt wurden, wird eine gewichtete Interpolation verwendet, die allerdings keine Extrapolation erlaubt, sodass die Variation der Charakteristiken der Benchmarks im aktuellen Ansatz auch alle Extremwerte der Parameter-Wertebereiche beinhalten muss. Für jeden zu prädizierenden Parameter wird eine Serie von Transformationsfaktoren $f(C)$ erstellt, je einer pro Kandidat C. Anschließend werden die Faktoren durch die inversen Distanzen zum Ziel-Skalierungsvektor gewichtet ($w = 1 - \mathcal{D}$), wobei unterschiedliche Gewichtungen getestet

wurden (individuelle oder durchschnittliche Distanzen zwischen Plattformen, Workloads oder Tupeln). Da Kandidaten mit geringerer Distanz potentiell eine höhere Ähnlichkeit des Skalierungsverhaltens aufweisen, kann eine zusätzliche Gewichtung verwendet werden (*Penalty*), die größere Distanzen bestraft. Dadurch wird der Einfluss der Kandidaten zwischen 100 % für minimale Distanz w_{min} und 0 % für maximale Distanz entweder linear, quadratisch oder kubisch normalisiert.

$$r(w) = \left(\frac{w - w_{min}}{w_{max} - w_{min}} \right)^k, \quad k \in \{0, 1, 2, 3\} \tag{5.17}$$

Folglich kann die finale Prädiktion eines Elementes sc_m eines Skalierungsvektors (oder die sequentielle Ausführungszeit) aus einer Auswahl von Kandidaten C wie folgt berechnet werden:

$$sc_m(T, A) = \frac{\sum_C \left[sc_m(C) \cdot f(C) \cdot r(1 - \mathcal{D}(C)) \right]}{\sum_C r(1 - \mathcal{D}(C))}. \tag{5.18}$$

5.2.5 Benchmark-Set

Da die Prädiktionsgenauigkeit stark von der Qualität der Datenbank abhängt, ist die Grundidee dieses Ansatzes, eine umfangreiche Datenbank zu erstellen, um die Wahrscheinlichkeit für die Existenz nahe gelegener Prädiktionskandidaten zu erhöhen. Alle Benchmarks und benötigten Bibliotheken, wie MPAL für die Parallelisierung, Profiling und Parameter-Extraktion, oder *Performance Application Programming Interface* (PAPI) [79] für die Verwendung von Performance-Countern, wurden dazu in einem TAR-Ball (Archiv) komprimiert. Ein Makefile extrahiert automatisch den gesamten Inhalt, kompiliert Bibliotheken, führt nacheinander alle Benchmarks aus und komprimiert alle Profiling-Ergebnisse, sowie alle extrahierten Charakteristiken, in einem weiteren TAR-Ball. Die Zusammenstellung der Benchmarks beinhaltet Implementierungen mit unterschiedlichen Charakteristiken und Parallelisierungs-Strategien. Alle Benchmarks verwenden statische Größen der Input-Daten, und jede separate parallele Sektion der Algorithmen wird als individueller Workload betrachtet. Während zum einen reale Algorithmen aus dem Bereich der Fahrerassistenzsysteme, wie Stereo-Vision, Fußgängererkennung oder Optical Flow, verwendet werden, werden ebenfalls Standard-Benchmarks eingefügt, wie beispielsweise Speckle Noise Reduction oder K-D-Tree Implementierungen. Das aktuelle Set von insgesamt 17 Workloads beinhaltet variierende Parallelisierungen, wie *Domain Decomposition*, *eingenistete und rekursive Spawns*, und *Wellenfront Parallelisierung*, sowie unterschiedliche *Lock und Synchronizationstrategien*. Die zusätzliche Verwendung von Mikrobenchmarks zur Provokation von bestimmten Hardware-Effekten, wie Inter-Core-Kommunikation, könnte den

Entwurfs- und Parameterraum weiter ausreizen und damit die Prädiktion weiter verbessern.

5.2.6 Prädiktionsgenauigkeit

Eine Vielzahl der in der Literatur präsentierten Prädiktionsfehler sind nur schwer miteinander vergleichbar, da die Ergebnisse unterschiedlich interpretiert wurden, wie für parallele Benchmark-Ergebnisse auch von Hoefler und Belli [80] zusammengefasst wurde. Während manche Autoren absolute Abweichungen angeben, präsentieren andere nur Korrelationen zwischen Prädiktion und gemessenen Laufzeiten oder es werden die prädizierten Ergebnisse aller paralleler Sektionen aggregiert, die sowohl positive, als auch negative Abweichungen beinhalten, und werten lediglich den Gesamtfehler aus. Im Folgenden wird daher der durchschnittliche Fehler über den gesamten Skalierungsverlauf angegeben (als Durchschnitt der absoluten Abweichung zwischen Prädiktion und Messung über den Verlauf von $t(n)$), was die Prädiktionsgenauigkeit der vorgestellten Methode über den gesamten Skalierungsverlauf repräsentiert.

Für eine umfangreiche Evaluation der Prädiktion wurde eine Datenbank mit Profiles unterschiedlicher Plattformen aus den drei unterschiedlichen Anwendungsbereichen (Plattform-Familien) Server, Desktop und eingebettete Prozessoren verwendet, die jeweils aus unterschiedlichen Generationen mit variierenden Mikroarchitekturen stammen (siehe Tabelle 5.4). Die Prädiktion wird in drei Schritten ausgewertet: $t(1)$, $\vec{sc}$ (rekonstruierter Skalierungsverlauf/Speedup), und die volle Prädiktion ($t(1)$ plus $\vec{sc}$). Da die Prädiktion von $t(1)$ und $\vec{sc}$ jeweils sowohl positive, als auch negative, Abweichungen aufweisen

Tabelle 5.4: Für die Prädiktions-Evaluation verwendete Plattformen.

CPU	Cores	Threads	μ-Arch.	Wortbreite
Xeon Gold 6148	2×20	80	Skylake	64 bit
Xeon E5-2630 v3	2×8	32	Haswell	64 bit
Xeon E5-2680	2×8	32	Sandy Bridge	64 bit
Opteron 6220	2×8	16	Bulldozer	64 bit
Opteron 6172	2×12	24	K10	64 bit
Xeon E5620	2×4	16	Westmere	64 bit
i9-9900K	8	16	Coffee Lake	64 bit
i5-8500T	6	6	Coffee Lake	64 bit
i7-8550U	4	8	Kaby Lake	64 bit
i5-7300HQ	4	4	Kaby Lake	64 bit
i7-4771	4	8	Haswell	64 bit
i7-3667U	2	4	Ivy Bridge	64 bit
A53 (Amlogic S905)	4	4	ARMv8	64 bit
A15 (Exynos 5422)	4	4	ARMv7	32 bit
A7 (BCM2836)	4	4	ARMv7	32 bit

Tabelle 5.5: Prädiktionsfehler unterschiedlicher Datenbank-Konfigurationen mit zusätzlichen Angaben des Medians, des minimalen bzw. maximalen Fehlers, sowie der 70 %-, 80 %-, sowie 95 %-Perzentilen.

Fehler [%]	Durchschn.	Median	Min.	Max.	70 %	80 %	95 %
all: $t(1)$	15,4	9,5	0,0	107,8	15,5	23,7	59,0
all: $\vec{sc}$	12,7	6,7	0,1	195,4	11,5	17,8	38,9
all: full	**19,9**	11,1	0,0	185,6	20,4	29,7	69,8
server: $t(1)$	18,0	10,7	0,0	116,6	20,8	23,7	70,5
server: $\vec{sc}$	16,0	11,0	0,3	101,1	17,7	22,9	57,8
server: full	25,5	16,3	0,2	103,7	29,7	46,4	72,1
desktop: $t(1)$	6,8	4,4	0,2	41,8	6,8	9,9	23,6
desktop: $\vec{sc}$	8,0	4,5	0,1	136,6	7,6	9,8	21,5
desktop: full	9,3	5,3	0,1	129,2	9,4	13,7	29,0
embed.: $t(1)$	28,9	20,4	2,0	112,6	33,9	39,1	100,6
embed.: $\vec{sc}$	3,2	2,4	0,0	16,6	4,1	4,5	11,7
embed.: full	29,0	21,8	0,0	111,2	32,8	39,7	100.8

kann, stellt der Prädiktionsfehler der vollen Prädiktion nicht zwangsläufig die Aggregation der beiden Teilfehler dar, sondern Fehler können sich gegeneinander aufheben und kompensieren. Die Prädiktion wurde ausgewertet, indem jeweils ein Wert aus der Datenbank eliminiert, durch die gezeigte Methode prädiziert und anschließend die Prädiktion gegen diesen Ground-Truth-Wert verglichen wurde. Es wurden alle in der Datenbank vorhandenen Performance-Werte getestet, wobei eine einzelne Prädiktion 1,6 s benötigte. Zunächst wurde eine Zusammenstellung der Datenbank verwendet, bei der alle vorhandenen Plattformen als Prädiktionskandidaten zur Verfügung stehen, was zum einen die Anzahl der verfügbaren Kandidaten erhöht, als auch eine Prädiktion von einer Plattform-Familie zu einer anderen ermöglicht. Anschließend wurde eine Prädiktion nur innerhalb der einzelnen Plattform-Typen getestet (z.B. nur Desktop). Für die Prädiktion der sequentiellen Ausführungszeit $t(1)$ wurden maximal 5 Workloads und 5 Plattformen für die Kandidatensuche zugelassen, während die Kandidaten anhand der durchschnittlichen Workload-Distanz, aber individuellen Plattform-Distanz sortiert wurden. Die Transformationsfaktoren der jeweiligen Kandidaten wurden mithilfe der durchschnittlichen Workload-Distanz ohne zusätzliche Penalty gewichtet ($k = 0$). Die Skalierbarkeit wird prädiziert, indem die Kandidaten nach Tupeln aus individuellen Plattform- und Workload-Distanzen (3 Plattformen und bis zu 10 Workloads) sortiert werden. Die Transformationsfaktoren werden mit individuellen Workload-Distanzen mit kubischer Penalty ($k = 3$) gewichtet.

Tabelle 5.5 zeigt den durchschnittlichen Fehler über den Skalierungsverlauf als Durchschnitt und Median über alle Prädiktionen, sowie die minimalen und maximalen Fehler und 70 %-, 80 %- und 95 %-Perzentilen. Während $t(1)$ im Durchschnitt mit einem Fehler von 15,4 % prädiziert werden kann, erreicht

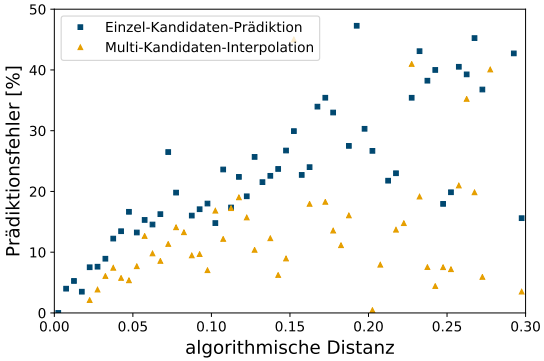


Abbildung 5.7: Durchschnittlicher Prädiktionsfehler unter Verwendung von Kandidaten mit variierenden algorithmischen Distanzen $\mathcal{D}_a(P, A, B)$.

die Prädiktion des Skalierungsverlaufs (Speedup über variierende Anzahl von Prozessorkernen n) im Durchschnitt über alle verfügbare Plattformen einen mittleren Fehler von 9,3 %. Die Kombination beider Teil-Prädiktionen in der vollen Prädiktion, die folglich eine vollständige Migrationsprädiktion eines parallelen Workloads darstellt, erreicht einen durchschnittlichen Prädiktionsfehler von 19.9 %. Es wird deutlich, dass die Präzision der Prädiktion stark von der Zusammensetzung und Qualität der Datenbank abhängt. Wird beispielsweise lediglich innerhalb der Server-Klasse prädiziert, sinkt die Präzision bis auf einen Fehlerwert von 25,5 %. Werden dagegen nur Desktop-Prozessoren verwendet, erreicht die Prädiktion einen Fehlerwert von 9,3 %. Während die Prädiktionen innerhalb von Desktop-Prozessoren geringe Fehler aufweisen, haben Server-Prozessoren zusätzliche Einflüsse durch Effekte, wie z.B. NUMA-Node-Kommunikation und eine deutlich höhere Anzahl von Prozessorkernen, wodurch sich auch geringe Abweichungen nach einem langen Skalierungsverlauf stärker auf den finalen Speedup auswirken und so zu größeren Prädiktionsfehlern führen. Die geringe Anzahl der verfügbaren Referenz-Plattformen beeinträchtigt die Präzision der Prädiktion innerhalb der Embedded-Prozessoren. Im Gegensatz zu einer naiven Abschätzung mithilfe des idealisierten Speedups n, die in den meisten Fällen durchschnittliche Fehler von über 200 % bringt, liefert die vorgestellte Methode präzise Prädiktionen verschiedener Nebenläufigkeitseinflüsse und Bottlenecks auf allen Multicore-Architekturen.

Zusätzlich zur vorangegangenen Evaluation des mittleren Prädiktionsfehlers, die bereits zeigt, dass die vorgestellte Methode schon mit relativ kleinen Datenbanken gute Prädiktionen liefert, wurde die Korrelation zwischen den verwendeten Distanzen und dem resultierenden Prädiktionsfehler untersucht. Abbildung 5.7 zeigt den durchschnittlichen Fehler für eine Reihe getesteter

(a) Einzel-Kandidaten-Prädiktion. (b) Multi-Kandidaten-Prädiktion.

Abbildung 5.8: Schematische Darstellung einer Einzel-Kandidaten-Prädiktion gegenüber eine Multi-Kandidaten-Prädiktion mit gewichteter Interpolation. Kreise sind vorhandene Kandidaten der Datenbank, die Zur Prädiktion des rot markierten Datenpunkt genutzt werden können.

Prädiktionen, während Kandidaten mit unterschiedlichen algorithmischen Distanzen $\mathcal{D}_a$ verwendet wurden, um eine Prädiktion entweder mit nur einem einzelnen Kandidaten, oder eine interpolierte Prädiktion (selbe Konfiguration, wie zuvor) mit mehreren Kandidaten durchzuführen (siehe Abbildung 5.8). Bei Betrachtung der Einzel-Kandidaten-Prädiktion ist eine klare Korrelation zwischen dem ermittelten Distanzmaß und dem resultierenden Prädiktionsfehler erkennbar. Diese Beziehung kann wie folgt ausgedrückt werden: *je kleiner die ermittelte algorithmische Distanz, desto größer die Wahrscheinlichkeit für eine präzise Prädiktion.* Diese Korrelation unterstützt die Signifikanz der verwendeten Skalierungscharakteristiken als deskriptive Features, sowie der daraus bestimmten Distanzmaße. Des Weiteren zeigt die Betrachtung der Multi-Kandidaten-Prädiktion, dass sich durch diese interpolative Transformation eine deutliche Verbesserung des Prädiktionsfehlers, selbst unter Verwendung von Kandidaten mit größeren Distanzen, ergibt.

6 Methoden-Bewertung

In diesem Kapitel werden die in den vorangegangenen Kapiteln vorgestellten Methoden evaluiert und ihre Anwendung und Wirkung anhand von Fallbeispielen (engl. *Case Study*) gezeigt. Dabei wird entsprechend der allgemeinen Struktur dieser Arbeit zunächst die Abstraktionsschicht besonders im Hinblick auf Portabilität untersucht, wobei sowohl homogene, als auch heterogene Implementierungen betrachtet werden. Anschließend werden Skalierungscharakteristiken betrachtet, die mit Hilfe der Profiling-Methoden extrahiert werden können, die in die MPAL-Abstraktionsschicht integriert wurden. Abschließend wird anhand einer Fallstudie die vorgestellte statistische Prädiktionsmethode mit anderen *state-of-the-art*-Prädiktionsansätzen aus den Bereichen Virtual-Prototyping und Analytische Modelle verglichen. Um sowohl die Portabilität hervorzuheben, als auch die zum Teil große Varianz zwischen Plattformen, werden im Folgenden für die Darstellung der jeweiligen Aspekte unterschiedliche homogene und heterogene Plattformen verwendet.

6.1 Implementierungen

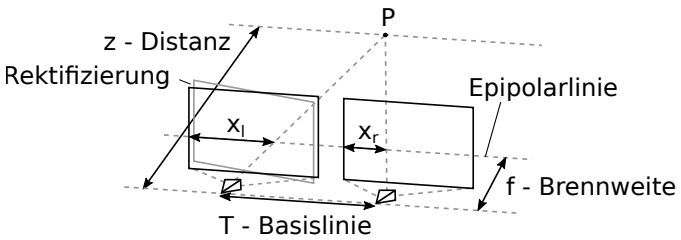
Folgend werden zunächst die beiden Algorithmen *Semi-Global Matching* (SGM) [2] für Stereo-Vision (Disparitätsschätzung) und *Histograms of Oriented Gradients* (HOG) [81] für die Fußgängererkennung aus dem Bereich der kamerabasierten Fahrerassistenzalgorithmen vorgestellt, die teilweise bereits in vorherigen Beispielen verwendet wurden. Beide Algorithmen bestehen aus mehreren konsekutiven algorithmischen Stufen (engl. *Stages*), die jeweils auch als separate Algorithmen betrachtet werden können, da sie sich in ihrer Rechenkomplexität, ihrer Parallelisierungs-Strategie, sowie den Anforderungen an die Hardware unterscheiden. In vorangegangenen Arbeiten wurden unterschiedliche homogene und heterogene Implementierungen der beiden Algorithmen entwickelt

und auf diversen Signalverarbeitungs-Architekturen evaluiert. Daher werden alle Untersuchungen zu den vorgestellten Methoden anhand dieser beiden Fallstudien aufgestellt.

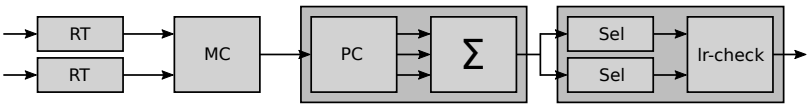
6.1.1 Disparitätsschätzung

In diesem Abschnitt wird zunächst der SGM-Algorithmus und der dazugehörige geometrische Stereo-Kamera-Aufbau erläutert. Im Anschluss werden zwei unterschiedliche Implementierungs- bzw. Parallelisierungs-Strategien (Optimierungen) erläutert, die bereits im Verlauf der Arbeit angesprochen wurden und auch in diesem Kapitel wieder aufgegriffen werden.

Algorithmus: Für die Rekonstruktion einer pixelgenauen Disparitätskarte, die durch korrekte Parametrisierung direkt in eine Tiefenkarte umgerechnet werden kann, verwendet der *Semi-Global Matching* Algorithmus zwei Stereo-Bilder, die jeweils aus unterschiedlichen Perspektiven aufgezeichnet wurden. Basierend auf den beiden rektifizierten Eingangsbildern (Base- und Match-Bild) aus einem Stereokamera-Aufbau, die eine sogenannte Epipolar-Geometrie repräsentieren (siehe Abbildung 6.1(a)), kann die Objektdistanz z aus Setup-Parametern, wie Brennweite f und Abstand zwischen den Kameras T, und der geschätzten Disparität d zwischen korrespondierenden Pixeln in beiden Eingangsbildern bestimmt werden. Unter Verwendung des Strahlensatzes kann die Distanz z zwischen einem dreidimensionalen Objektpunkt und der Stereo-



(a) Stereo-Vision-Setup: Zwei rektifizierte Eingangsbilder mit Epipolargeometrie.



(b) Algorithmische Stufen des Semi-Global Matching Algorithmus.

Abbildung 6.1: Semi-Global Matching: Geometrier Stereo-Kamera-Aufbau und algorithmische Pipeline.

Kamera mit der folgenden Formel bestimmt werden:

$$z = \frac{f \cdot T}{x_l - x_r} = \frac{f \cdot T}{d}. \tag{6.1}$$

Jedes Stereo-Bildpaar wird in einer linearen algorithmischen Pipeline bearbeitet, die in Abbildung 6.1(b) gezeigt ist. Zunächst wird in einer Rank-Transformation (RT) jedes Bild von Belichtungsunterschieden und Subpixel-Ungenauigkeiten bereinigt. Dadurch entsteht aus jedem der beiden zweidimensionalen Eingangsbilder ebenfalls wieder ein zweidimensionales Ausgangsbild derselben Größe. Durch Vergleich der Intensitäten von Korrespondenz-Kandidaten, die entlang der Epipolarlinie mit variierender Disparität $d = x_l - x_r$ in beiden rank-transformierten Bildern aus Base- und Match-Perspektive zu finden sind, wird in der *Matching-Cost*-Stufe (MC) eine Reihe von absoluten Intensitätsdifferenzen für jeden Pixel des Base-Bildes gebildet, wodurch eine dreidimensionale Ausgangsdatenstruktur entsteht ($x \times y \times d$). Da diese generierten Matching-Costs starke Schwankungen beinhalten, werden in der nachfolgenden *Path-Cost*-Stufe (PC) örtliche Lokalitätsinformationen ausgenutzt. Dazu werden in unterschiedlichen Laufrichtungen (je im 45 °-Winkel zueinander) Schwankungen entlang der Laufrichtung mit höheren Kosten versehen, um diese Schwankungen auszugleichen. Im Gegensatz zum originalen Algorithmus, in dem alle acht Richtungen durchlaufen werden, verwendet diese Implementierung lediglich vier, wodurch die Gesamtausführungszeit stark reduziert werden kann, der Einfluss auf die Qualität des Ausgangsbildes jedoch gering bleibt. Anschließend werden die dreidimensionalen Ausgangsdaten jeder Laufrichtung zu einem Gesamtergebnis akkumuliert, woraus danach die Disparität mit den minimalsten Intensitätsunterschieden zur Speicherung in einer Disparitätskarte selektiert wird. Durch das Parsen der Daten in zwei unterschiedlichen Richtungen, werden Disparitätskarten jeweils für die Base- und Match-Perspektive generiert, wodurch eine Konsistenzprüfung zwischen beiden Perspektiven ermöglicht wird. Dabei werden nur Disparitäten, die in beiden Bildern innerhalb einer geringen Toleranz liegen verwendet und große Abweichungen (beispielsweise bei teilweiser Überdeckung von Objekten in einer der Perspektiven) als ungültig markiert.

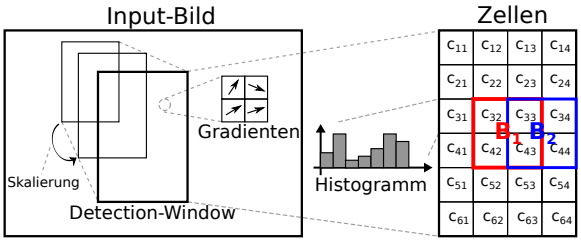
Parallelisierung und Optimierungen: Im Folgenden wird eine homogen parallelisierte Implementierung des SGM-Algorithmus vorgestellt [13], bei der alle algorithmischen Stufen in konsekutiver Reihenfolge abgearbeitet werden und eine Parallelisierung durch Datenaufteilung (Data Decomposition) in jeder separaten Stufe durchgeführt wird. Zunächst wurde die Performance auf einem eingebetteten homogenen Achtkern-Prozessor untersucht. Basierend auf den Ergebnissen werden zwei Optimierungen angewendet. Die erste Optimierung (*Tasking*) verwendet eine feingranulare Aufteilung der Arbeit,

um ein dichteres Packen der Tasks und dadurch eine Reduktion der Work-Imbalance zu ermöglichen. Um allerdings den Speicherverbrauch zu minimieren und gleichzeitig die Datenlokalität zu erhöhen, wurde eine zweite Optimierung angewendet (*Streaming*), die die Stufen MC und PC vollständig ersetzt und beide algorithmischen Schritte in einer Wellenfront-Transformation abarbeitet, ohne dabei zu speichernde Zwischenergebnisse zu erzeugen. Während der Berechnung der Pfadkosten hängt jeder Pixel von den Ergebnissen der vier benachbarten Pixel ab (vorherige Pixel entlang der Pfad-Laufrichtungen). Daher werden Threads mithilfe eines Mutex-Arrays zueinander synchronisiert, indem der Zugriff auf noch nicht verfügbare Ressourcen (Pixel) blockiert wird. Da Daten, die für die Berechnung des aktuellen Pixels notwendig sind, meist nicht im lokalen Cache vorrätig sind, resultiert diese Optimierung in hohen Synchronisations- und Kommunikationskosten zwischen den Cores und Caches. Da allerdings die Latenzen für Cache-Misses und Inter-Core-Kommunikation stark zwischen Plattformen variieren, konnte gezeigt werden, dass sich beide Optimierungen auf unterschiedlichen Plattformen gegenseitig übertreffen. Die Implementierung wurde vollständig mit MPAL parallelisiert und bietet daher den vollen Funktionsumfang für die Verwendung von homogenen und heterogenen Multicore-Clustern. Außerdem wurde in [51] eine Erweiterung der Implementierung um OpenCL-Kernel zu jeder Stufe vorgestellt, sodass eine vollständig heterogene Ausführung auch auf OpenCL-fähigen Beschleunigern ermöglicht wird.

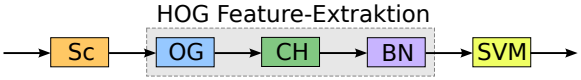
6.1.2 Fußgängererkennung

In diesem Abschnitt wird zunächst der Ablauf und anschließend die parallele Implementierung des HOG-Algorithmus beschrieben, die eine verzweigte Parallelisierung verwendet.

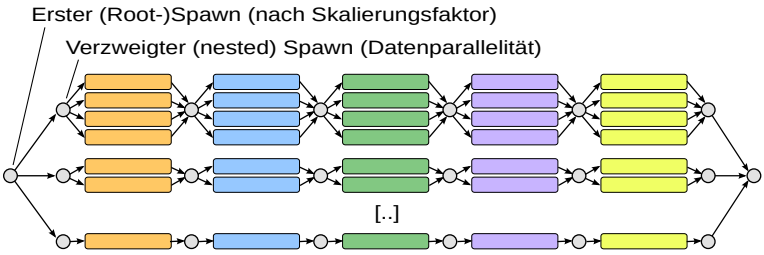
Algorithmus: In [82] wird eine parallele Implementierung des HOG-Algorithmus für die Fußgängererkennung vorgestellt, die zunächst für den homogenen Manycore Intel Xeon Phi (Knights Corner) entwickelt wurde. Während andere Implementierungen Deskriptor-Kaskaden verwenden, um eine Erkennung von Fußgängern in unterschiedlichen Skalierungen zu ermöglichen, wird in dieser Implementierung lediglich eine einzelne *Support Vector Machine* (SVM) mit einer festen Feature-Vektor-Größe verwendet. Um für alle Skalierungsstufen einen einheitlichen Deskriptor zu ermöglichen, wird vor der Feature-Extraktion in der Stufe Sc eine inverse Skalierung des Bildes um den Faktor $1/s$ durchgeführt, und jedes skalierte Bild $I_s(\vec{p})$ wird an eine separate HOG-Feature Extraktions-Pipeline übergeben. Während für das Upsampling der Bilder eine bilineare Interpolation angewendet wird, verwendet das



(a) Schematische Darstellung des Histograms of Oriented Gradients Algorithmus.



(b) Algorithmische Pipeline: Skalierung, Extraktion der HOG-Features und Klassifikation.



(c) Parallelisierungs-Schema mit zwei Parallelisierungs-Ebenen.

Abbildung 6.2: Algorithmus und Parallelisierung des HOG-Algorithmus

Downsampling eine Mittelwertbildung innerhalb einer adaptiven Maske. Wie in Abbildung 6.2(a) und Abbildung 6.2(b) gezeigt, werden für die Feature-Extraktion zunächst pixelweise orientierte Gradienten (OG) in karthesischen Koordinaten durch Masken der Form $[-1, 0, 1]$ berechnet. Bevor daraus Histogramme innerhalb von Zellen der Größe 8×8 Pixel über den quantifizierten Gradientenwinkeln (Binning) ermittelt werden, werden die Gradienten in sphärische Koordinaten übertragen. In [82] wird eine Optimierung vorgestellt, durch die trigonometrische Berechnungen umgangen werden können, ohne dabei Fehler durch Approximationen einzufügen. In der *Cell-Histogram*-Stufe (CH) werden anschließend individuelle Histogramme mit je 12 Binning-Positionen innerhalb jeder Zelle erzeugt, die eine Orientierung bis maximal 180 ° repräsentieren. Die Block-Normalisierung (BN) konkateniert und normalisiert die Histogramme jeweils innerhalb von 4 Zellen (2×2 Zellen) auf die Gesamtlänge 1, wobei sich die Block-Positionen um je einen Block überlappen. Der resultierende HOG-Feature-Vektor wird anschließend durch Konkatenation der normalisierten Block-Histogramme innerhalb eines Detection-Windows erzeugt

und durch die vortrainierte SVM klassifiziert.

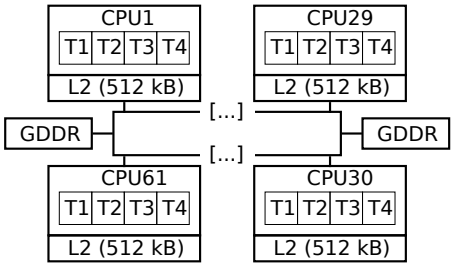
Parallelisierung: Ursprünglich wurde jede Detection-Window-Position in einer eigenen Prozess-Kette verarbeitet. In [82] wird eine Optimierung vorgeschlagen, in der nur jeweils eine nebenläufige Prozess-Kette für jede Skalierungsstufe existiert, und die Positionen der Detection-Windows lediglich in der letzten Stufe (BN) unterschieden und somit aus unterschiedlichen Zellen konkateniert werden (siehe Abbildung 6.2(c)). Die Aufteilung des Gesamtprozesses nach Prozess-Ketten für jede Skalierungsstufe stellt in der Implementierung die erste Parallelisierung dar (Root-Spawn). Da s den Skalierungsfaktor der Höhe und Breite eines Detection-Windows definiert, steigt der Arbeitsaufwand zur Feature-Extraktion quadratisch mit steigendem s. Wie in Abbildung 6.2(c) gezeigt, wird eine weitere (eingenistete) Parallelisierung jeweils innerhalb jeder algorithmischen Stufen der HOG-Feature-Extraktion durch Datenaufteilung (Domain Decomposition) durchgeführt. Da eine reine Datenaufteilung für klein skalierte Eingangsbilder keine ausreichenden Nebenläufigkeiten aufweist, um auf bis zu 244 Threads des Xeon Phi Manycores parallel ausgeführt zu werden, ermöglicht nur eine Parallelisierungs-Strategie mit mehreren Spawn-Ebenen eine höhere und ausreichende Parallelität. Allerdings erfordert diese Strategie eine große Anzahl an parallelen Tasks, Synchronisationen und Taskverwaltungs-Overhead, weshalb die optimale Granularität für jede Skalierungsstufe durch Modellierung ermittelt wurde.

6.2 Portabilitätsuntersuchung

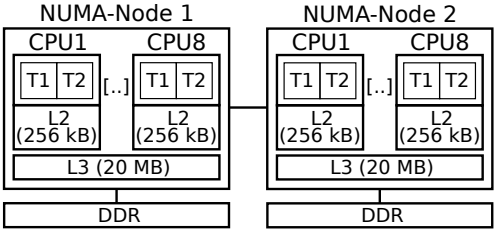
Da die Implementierungen der beiden Algorithmen auf MPAL basieren, lassen sie sich mit geringem Aufwand zwischen Plattformen migrieren, wobei unterschiedliche Laufzeiteigenschaften sichtbar werden. In diesem Abschnitt wird zunächst das Skalierungsverhalten der parallelen Implementierungen auf homogenen Multicore-Prozessoren untersucht, wobei allgemein der Parallelisierungserfolg (Speedup) und Vektorisierungs-Speedup diskutiert werden. Dadurch lassen sich die im Vorfeld diskutierten Einflussfaktoren auf die parallele und heterogene Laufzeit verdeutlichen. Anschließend wird zur spezifischeren Untersuchung der Portabilität, teils unter Verwendung der OpenCL-Erweiterung beider Implementierungen, eine Migration auf unterschiedliche heterogene Plattformen durchgeführt und evaluiert. Dabei wurden für den SGM-Algorithmus Eingangsbilder von 450×375 px und eine maximale Disparität von 64 bei vier Pfad-Richtungen verwendet. Aufgrund der unterschiedlichen Datenbanken, die für SGM zur Verarbeitung von Stereo-Bildern und für HOG zur Personen-Detektion, benötigt werden, wurde für den HOG-Algorithmus eine andere Größe der Eingangsbilder von 557×631 px verwendet.

6.2.1 Homogene Multi- und Manycore-Prozessoren

Im Folgenden werden Skalierungsverläufe der beiden Algorithmen SGM und HOG auf einem Xeon E5-2680 General-Purpose-Prozessor mit 32 Threads und einem Xeon Phi mit bis zu 244 Threads verglichen (siehe Abbildung 6.3). Diese Fallstudie wurde zuerst in [82] vorgestellt. Der Xeon E5-2680 besteht aus zwei NUMA-Nodes mit insgesamt 16 Prozessorkernen bei 2,7 GHz, die je zwei Hyperthreads aufweisen. Der Xeon Phi Prozessor ist als PCIe-Co-Prozessor angebunden und läuft mit einem eigenen Linux-Betriebssystem, sodass er als eigenständiges System betrachtet werden kann. Er besitzt 61 Prozessoren bei 1,2 GHz mit je vier Threads, wobei eine Verwendung von mehr als zwei Threads pro Kern keinen Speedup bringt und auch nicht empfohlen ist. Daher werden hier nur maximal 122 Threads verwendet. Die Kommunikation zwischen den Prozessorkernen wird durch einen bidirektionalen Ringbus realisiert, während die privaten 512 kB L2-Caches ein Hashing-Protokoll verwenden, um Daten über den Bus zu verteilen. Es wird zunächst der Vektorisierungs-Speedup unter Verwendung von einem Prozessorkern und anschließend der Parallelisierungs-Speedup unter Verwendung der vollen Vektorisierung evaluiert.



(a) Darstellung des Xeon Phi.



(b) Darstellung des Xeon E5-2680.

Abbildung 6.3: Schematische Darstellung der Prozessoren.

Vektorisierungs-Speedup: Im Gegensatz zum Xeon E5-2680, der MMX, SSE und AVX als Vektor-Befehlssätze mit bis zu 256 bit-Vektoren anbietet, besitzt der Xeon Phi Prozessor Vektorregister mit bis zu 512 bit. Die Vektorisierungs-Ergebnisse der Vektorisierung aller algorithmischen Stufen des HOG-Algorithmus sind in Tabelle 6.1 dargestellt. Da häufig Integer-Typen mit 8 bit oder 16 bit Länge verwendet werden, kann für die 512 bit großen Vektorregister des Xeon Phis theoretisch mit einer bis zu 64-fachen Beschleunigung der Berechnung gerechnet werden. Da allerdings der Vektor-Instruktionssatz des Xeon Phi Prozessors lediglich Datentypen bis minimal 32 bit unterstützt, ist der maximale Speedup auf 16 (Xeon Phi) bzw 8 (Xeon E5-2680) limitiert.

Tabelle 6.1: HOG Vektorisierungs-Speedup der algorithmischen Stufen.

	Single-Core-Ausführung	**Sc**	**OG**	**CH**	**BN**	**SVM**	**total**
Xeon Phi	**keine Vektorisierung** [ms]	804	1264	161	210	1141	3600
	inkl. Vektorisierung [ms]	433	194	54	17	134	851
	Speedup	1,9	6,5	3,0	12,1	8,5	4,2
Xeon E5-2680	**keine Vektorisierung** [ms]	53	88	7	14	152	314
	inkl. Vektorisierung [ms]	26	45	6	2	17	96
	Speedup	2,0	2,0	1,1	9,1	9,0	3,2

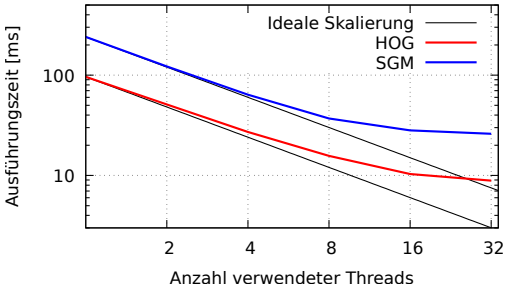
Des Weiteren mussten algorithmische Anpassungen angewendet werden, um eine Vektorisierung zu ermöglichen, wodurch nur ein reduzierter Speedup in den Stufen *Sc*, *OG*, und *CH* erreicht werden kann. Beispielsweise kann für die Stufe *OG* aufgrund von notwendigen Umsortierungen der Operationen, um bedingte Sprünge zu umgehen, nur ein maximaler theoretischer Speedup von etwa 8 (Xeon Phi) und 4 (Xeon E5-2680) erwartet werden. Während aufgrund von unvorhersagbaren Pixel-Positionen in der Stufe *Sc* ein nicht vektorisierter Anteil verbleibt, musste in *CH* eine Speichererweiterung integriert werden, wodurch der Speedup ebenfalls reduziert wird. Allerdings gibt die Umformulierung des Software-Codes in bestimmten Fällen dem Compiler die Möglichkeit eine optimalere Hardware-Auslastung zu finden, wodurch der vermutete Speedup übertroffen werden kann. Beispielsweise übersteigen in den Stufen *BN* und *SVM* die Speedup-Werte von 9,1 und 9,0 den theoretisch maximalen Speedup von 8 auf dem Xeon E5-2680. Allgemein wird der Erfolg der Vektorisierung durch zwei wesentliche Aspekte bedingt:

1. Zusammenspiel zwischen algorithmischen Anforderungen und Beschaffenheit des Vektor-Instruktionssatzes. Beispielsweise könnte bei einer häufigen Verwendung von 8 bit Integer-Typen die Ausführungszeit davon profitieren, wenn auch der Instruktionssatz eine höhere Packungsdichte der Vektoren zulässt.

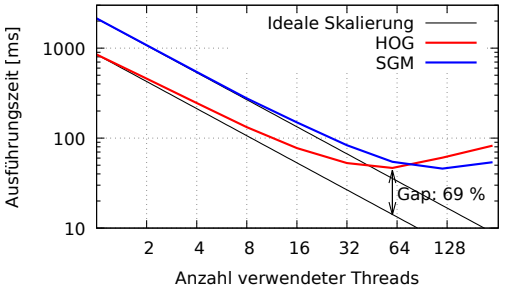
2. Verständlichkeit der in Software formulierten Parallelität für den Com-

piler. Beispielsweise kann die Verwendung von unpassenden Datentypen den Compiler veranlassen, zusätzliche Unpack- oder Cast-Operationen an ungeeigneten Stellen zu integrieren.

Parallelisierungs-Speedup: In Abbildung 6.4 sind die Skalierungsverläufe für die beiden Algorithmen HOG und SGM auf den beiden Plattformen gezeigt. In dieser Messung ist bereits eine volle Vektorisierung beider Implementierungen erfolgt, wobei zunächst die *Tasking*-Optimierung des SGM-Algorithmus verwendet wird. Messergebnisse sind in Tabelle 6.2 aufgelistet. Unter Annahme eines maximalen Speedups von 16 bei 16 Prozessorkernen des E5-2680 respektive 61 bei 61 Prozessorkernen des Manycores, erreicht die SGM-Implementierung eine Auslastung der Plattform von 74 % bzw. 75 %. Der HOG-Algorithmus erreicht auf dem Xeon E5-2680 einen nur geringfügig



(a) Skalierungsverhalten auf dem Xeon E5-2680 (2. NUMA-Node ab $n > 8$ Threads, Hyperthreading ab $n > 16$ Threads).



(b) Skalierungsverhalten auf dem Intel Xeon Phi MIC (Hyperthreading ab $n > 61$ Threads).

Abbildung 6.4: Skalierungsverlauf der HOG und SGM Algorithmen auf einem General-Purpose-Prozessor und einem Xeon Phi Manycore.

Tabelle 6.2: Parallelisierungs-Speedup (S) inkl. Vektorisierung und prozentualer Speedup im Vergleich zum theoretischen Maximum ($S_{max} = n_{max}$).

inkl. Vekt.	Intel Xeon Phi			Intel Xeon E5-2680		
	sequentiell	parallel	S (S/S_{max})	sequentiell	parallel	S (S/S_{max})
HOG	850,7 ms	44,5 ms	19,1 (31 %)	96,2 ms	8,6 ms	11,2 (70 %)
SGM	2139,2 ms	45,8 ms	45,7 (74 %)	240,4 ms	20,2 ms	11,9 (75 %)

abweichenden Wert von 70 %, aber nur 31 % auf dem Xeon Phi. Ein spezifisches Profiling des Laufzeitverhaltens des HOG-Algorithmus auf dem Xeon Phi deutet sowohl auf einen hohen Overhead für Task-Verwaltung (Erzeugen und Synchronisieren), als auch auf eine hohe Auslastung im Kernel-Thread hin, die auf die hohe Anzahl der Speicherallokationen zurückgeführt werden kann. Somit wird deutlich, dass der Xeon Phi aufgrund seiner in-order Pipeline und einfacheren internen Prozessor-Architektur für die Verarbeitung von synchronisationsintensiven parallelen Tasks nicht gut geeignet ist.

6.2.2 Heterogenes MPSoC

In [83] wurde die parallelisierte HOG-Implementierung um OpenCL-Kernel für jede einzelne algorithmische Stufe erweitert und daraus eine heterogene Implementierung entwickelt, die gleichzeitig die CPU und GPU eines MPSoC ausnutzt. Eine OpenCL-Erweiterung für den SGM-Algorithmus (*Tasking*-Optimierung) wurde in [13] vorgestellt. Dabei wurden sowohl die Ausführungszeiten beider heterogenen CPU-Cluster einer *big.LITTLE*-Architektur, die reine GPU-Ausführung, als auch die vollständig heterogene Ausführung evaluiert. Beide heterogenen Erweiterungen wurden vollständig in MPAL integriert, sodass eine einfache Konfiguration des aktuellen Architektur-Mappings möglich ist. Da beide Algorithmen aus unterschiedlichen algorithmischen Stufen bestehen, die jeweils eine individuelle Charakteristik aufweisen, wird exemplarisch für den SGM-Algorithmus die Performance und die notwendige Datentransferzeit jeder Stufe separat gemessen und untersucht.

Als eingebettete heterogene Plattform wurde exemplarisch der Samsung Exynos 5422 MPSoC verwendet, der eine heterogene big.LITTLE CPU-Architektur neben einer Mali-GPU besitzt (Abbildung 6.5). Vier ARM Cortex-A7 Prozessoren bilden das energiesparsame *LITTLE*-Cluster, die bis auf 1,6 GHz getaktet werden können. Für rechenintensive Aufgaben stehen im *big*-Cluster vier leistungsstarke Cortex-A15 Prozessoren mit maximal 2 GHz zur Verfügung, die allerdings einen deutlich höheren Stromverbrauch aufweisen. Während beide CPU-Cluster kohärente L2-Caches besitzen, ist der L2-Cache des A7-Clusters 512 kB groß, der L2-Cache des A15-Clusters dagegen 2 MB. Beide CPU-Cluster

Tabelle 6.3: Ausführungs- und Transferzeiten der algorithmischen SGM-Stufen für unterschiedliche Recheneinheiten des MPSoCs.

Latenzen	RT	MC	PC	Sel+L/R-C
big.LITTLE	7,34 ms	7,36 ms	116,03 ms	19,06 ms
Mali-GPU	5,94 ms	13,50 ms	62,37 ms	16,24 ms
Datengröße	165 kB	330 kB	10,3 MB	41,2 MB	165 kB
Transfer zu Mali	0,61 ms	0,84 ms	11,13 ms	21,04 ms	-
Transfer von Mali	-	0,81 ms	11,45 ms	21,01 ms	0,33 ms

und die GPU sind über einen *Advanced Extensible Interface* (AXI)-Bus mit demselben Hauptspeicher verbunden. Da allerdings die GPU nur mit OpenCL in Version 1.1 programmiert werden kann, wird kein *zero-copy* unterstützt, sodass zwischen CPU und GPU geteilte Daten explizit gemappt werden müssen. Die Mali-T620 MP6 GPU besitzt sechs Kerne, die bei 600 MHz takten und in zwei virtuelle GPUs mit vier respektive zwei Kernen unterteilt sind.

Um die Entwicklung eines heterogenen Ausführungs-Schemas (Architektur-Mapping) darzustellen, wird der Vorgang im Folgenden für den SGM-Algorithmus und auf der eingebetteten MPSoC-Plattform detailliert dargestellt. Ergebnisse einer ersten Messung aller algorithmischen Stufen, zunächst exklusiv auf allen verfügbaren Recheneinheiten, sowie Datentransferzeiten der Eingangs- und Ausgangsdaten zwischen CPU-Clustern und der GPU, sind in Tabelle 6.3 gezeigt.

Da der SGM-Algorithmus eine im Wesentlichen konsekutive algorithmische Pipeline besitzt, bietet sich nicht per se eine funktionale Aufteilung der Stufen an. Dies wäre erforderlich, wenn für latenzkritische Applikationen alle verfügbaren heterogenen Recheneinheiten für einen einzelnen Frame eingesetzt werden sollen. Um trotzdem eine funktionale Aufteilung zu erzwingen, könnte eine der beiden unabhängigen Rank-Transformationen auf die GPU ausgelagert werden, wodurch allerdings Transferzeiten berücksichtigt werden müssen. Die algorithmische Stufe, mit dem größten Einfluss auf die Gesamtausführungs-

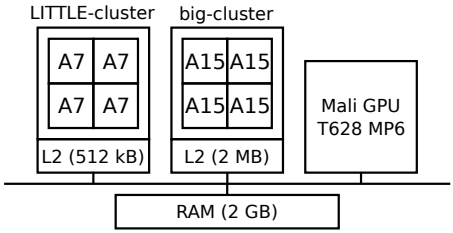


Abbildung 6.5: Aufbau des Samsung Exynos 5 Octa 5422 mobile MPSoC.

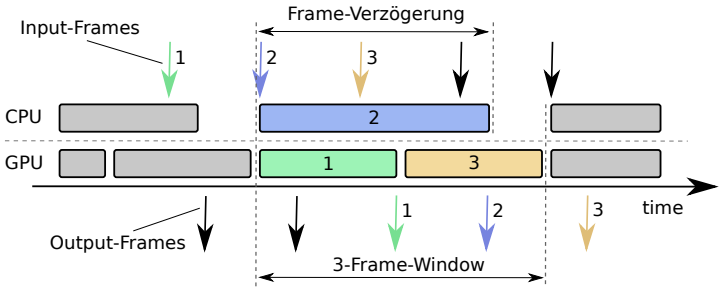


Abbildung 6.6: Heterogene Ausführung des SGM auf den CPU- und GPU-Clustern des Exynos 5422.

zeit, ist die Pfadkosten-Berechnung (PC), in der die Berechnung der vier Pfade funktional auf heterogene Recheneinheiten verteilt werden könnten. Eine Aufteilung der Berechnung nach Pfaden zwischen Recheneinheiten erfordert allerdings die Speicherung der Zwischenergebnisse jedes Pfades und bedingt das zwischenzeitliche Verdrängen der Daten aus dem Cache, wodurch auch das spätere Akkumulieren der Ergebnisse deutlich verlangsamt wird. Die fortlaufende Abarbeitung der Pfade ermöglicht dagegen eine implizite Akkumulation während der Berechnung, wodurch eine deutlich höhere Gesamt-Performance erreicht werden kann, wenn alle Pfade auf einem Gerät berechnet werden. Aufgrund der relativ geringen Ausführungszeit der Compute-Kernel und der hohen Transferzeiten erscheint auch eine Aufteilung der Daten in voneinander unabhängige Blöcke, die dann auf jeweils separaten Recheneinheiten berechnet werden, nicht sinnvoll.

Wie in Tabelle 6.3 zu erkennen ist, benötigen die Eingangs- und Ausgangsdaten der Path-Cost-Stufe lange Transferzeiten zwischen CPU- und GPU-Speicher. Daher wurde entschieden lediglich ein gesamtes Eingangs-Bildpaar der Stereo-Kamera auf die GPU zu kopieren und dort den gesamten SGM-Algorithmus auszuführen, um so das zwischenzeitliche Kopieren der Daten zu vermeiden. Dadurch sind allerdings die vorhandenen Rechenressourcen der CPU ungenutzt, weshalb eine gleichzeitige Berechnung von mehreren Frames in einem vorgegebenen Zeitfenster implementiert wurde. Wie in Abbildung 6.6 dargestellt, können auf der schnelleren GPU zwei Frames berechnet werden, während zeitgleich auf der CPU ein Frame verarbeitet wird. Dabei werden beide CPU-Cluster verwendet und die Arbeit zwischen den heterogenen Cores so verteilt, dass keine Work-Imbalance entsteht. In der hier gezeigten Version werden nur drei der vier Cortex A7-Cores für die SGM-Berechnung verwendet, da ein A7-Prozessor für die Verwaltung der GPU eingesetzt wird. Die Berechnung eines Frames auf der GPU benötigt 103 ms und auf den CPU-

Clustern 175 ms. Um eine möglichst geringe Latenz zu erreichen, wird ein Fenster von drei gleichzeitig zu bearbeitenden Frames gewählt, sodass die Berechnung von zwei Frames auf der GPU und einem Frame auf der CPU zu einer Arbeitsungleichverteilung von 31 ms der CPU-Cluster führt. Dadurch wird ein Durchsatz von $\frac{3\,\text{fr.}}{206\,\text{ms}} = 14,6\,\frac{\text{fr}}{\text{s}}$ mit einer maximalen Latenz von 175 ms erreicht.

Der HOG-Algorithmus besteht aus einer verzweigten Parallelisierung, die auch funktionale Nebenläufigkeiten beinhaltet und daher vielversprechendere Offload-Schemata ermöglicht. Wie in [83] dargestellt, können einige algorithmische Stufen durch Auslagerung auf die GPU stark beschleunigt werden, wohingegen die Skalierung (Sc) und die SVM-Berechnung, die einen hohen Speicherdurchsatz benötigt, keine guten Ergebnisse auf der Mali-GPU erzielen. Daher wurden alle Eingangsbilder auf der CPU vorprozessiert, dann teilweise zur Feature-Extraktion auf die GPU geladen und anschließend gemeinsam von GPU und CPU in der SVM-Stufe klassifiziert. Die optimale Verteilung der Daten zwischen CPU und GPU wurde durch eine vollständige Suche des Parameterraums evaluiert.

6.2.3 High-Level-Synthese

In [51] wurde ein exemplarisches Mapping der OpenCL-Kernel des SGM-Algorithmus auf einen FPGA durch HLS evaluiert. Dazu wurde ein i5-2400 Dualcore-Prozessor bei 3,1 GHz und mit 6 MB L3-Cache mit einer OpenCL-fähigen FPGA-Beschleuniger Karte von Nallatech erweitert (siehe Abbildung 6.7). Die Nallatech PCIe-385N besitzt einen Stratix-V FPGA und 8 GB lokalen Speicher. Sie wird per PCIe mit dem Host verbunden, und das OpenCL-interne Diagnose-Tool gibt einen Durchsatz von 1,39 GB/s an. Durch Verwendung der OpenCL-HLS, die in der Intel Quartus Software enthalten ist, können dieselben Kernel, die zunächst für GPU-Beschleuniger implementiert wurden, auch zur Konfiguration eines FPGAs verwendet werden. Dabei wird die Kommunikation zwischen Host und Beschleuniger durch zusätzlich auf das FPGA

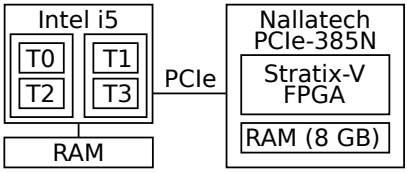


Abbildung 6.7: Darstellung des Host-PCs mit FPGA-Beschleuniger: Intel i5 PC via PCIe verbunden mit einem Nallatech FPGA-Beschleuniger.

Tabelle 6.4: Ausführungs- und Transferzeiten der SGM-Stufen für unterschiedliche Recheneinheiten des i5-Prozessors und den FPGa-Beschleuniger.

Latenzen	**RT**	**MC**	**PC**	**Sel+L/R-C**
Intel i5	1,96 ms	3,78 ms	56,97 ms	12,10 ms
Stratix-V FPGA	7,72 ms	46,38 ms	1333,65 ms	2434,21 ms

Datengröße	165 kB	330 kB	10,3 MB	41,2 MB	165 kB
Transfer zu Stratix-V	0,28 ms	0,63 ms	8,61 ms	32,33 ms	-
Transfer von Stratix-V	-	0,32 ms	6,83 ms	26,72 ms	0,34 ms

konfigurierte Kommunikationsmodule realisiert, sodass aus einem C-Programm ein einfacher OpenCL-Buffer mit OpenCL-API-Aufrufen zum FPGA kopiert werden kann. Weil die gesamte Verwaltung des FPGAs in OpenCL gekapselt ist, kann für die heterogene Ausführung mit FPGA-Beschleuniger auch MPAL zur Konfiguration des Architektur-Mappings verwendet werden. Da hier speziell die Portabilität der Implementierungen untersucht wird, wurden zunächst keine FPGA-spezifischen Optimierungen der Kernel vorgenommen (z.B. Fixpunkt-Arithmetik, Optimierungen durch Verwendung des Blockspeichers oder Anpassung der Granularität).

Unter einer Auslastung von etwa 80 % der verfügbaren Ressourcen des FPGAs konnte eine erste FPGA-Konfiguration durch HLS erzeugt und erste Messungen der Laufzeit und Speichertransferzeiten unternommen werden, welche in Tabelle 6.4 gezeigt sind. Aufgrund der nicht angepassten Kernel, die zunächst für die Verwendung von GPUs optimiert wurden, werden hier deutlich schlechtere Ergebnisse erzielt, als beispielsweise auf der Mali-GPU. Außerdem ist zu erwähnen, dass die Performance der synthetisierten Kernel mit der Gesamtauslastung des FPGAs stark variieren kann, da die Synthese unterschiedliche Prioritäten für das Ausrollen von Schleifen oder die Verwendung von Block-Speichern oder DSP-Blöcken anwendet. Aufgrund dessen und wegen der heuristischen Charakteristik des *Place-And-Route*-Prozesses schwanken die Ressourcen, die für einen algorithmischen Block verwendet werden, stark, sobald weitere Blöcke hinzugefügt oder entfernt werden. Allgemein wurde festgestellt, dass GPU-Kernel nicht ohne weitere Anpassungen an FPGA-spezifische Eigenschaften eine gute Beschleunigung erzielen können. Die Portabilität von OpenCL-Kerneln wird somit durch die spezifischen Optimierungen stark eingeschränkt, die für unterschiedliche Geräte-Typen notwendig sind.

6.2.4 Zusammenfassung

Die verwendeten Implementierungen basieren vollständig auf MPAL, was eine einfache Kontrolle der Parallelität und des Architektur-Mappings, sowie eine

einfache Extraktion von Profiles ermöglicht. Durch anpassen der MPAL-Konfigurationsdatei kann mit geringem Aufwand jede der gezeigten Migrationen zwischen homogenen und heterogenen Plattformen durchgeführt werden, sofern eine flexible Implementierung vorliegt, die eine dynamische Arbeitsaufteilung erlaubt. So ist eine schnelle Portierung zwischen homogenen oder heterogenen Plattformen möglich, ohne Änderungen des Software-Codes vornehmen zu müssen. Eine Zusammenfassung der Laufzeiten auf unterschiedlichen homogenen und heterogenen Plattformen, ist in Tabelle 6.5 gezeigt. So ist hier beispielsweise auch zu sehen, dass die Ausführung des HOG Algorithmus auf den big.LITTLE CPUs eine bessere Ausführungszeit aufweist, als auf der GPU. Es wird also deutlich, dass nicht alle algorithmischen Stufen immer für eine bestimmte Beschleuniger-Familie geeignet sind, weshalb ein flexibles Architektur-Mapping besonders wichtig ist. Für den Fall, dass das verwendete Parallelisierungs-Framework auf der Zielplattform nicht vorhanden ist oder keine ausreichenden Skalierungseigenschaften aufweist, kann dieses per einfacher Konfiguration gewechselt werden. Es wurde bereits evaluiert, dass die Zwischenschicht nur minimalen bis gar keinen Overhead erzeugt und somit keine Nachteile durch die Verwendung von MPAL entstehen. Eine Einschränkung in der Programmierung von Beschleunigern ist sicherlich, dass Beschleuniger-Kernel erst in OpenCL implementiert und, wie im letzten Fallbeispiel zu sehen ist, gegebenenfalls für die Zielplattform optimiert werden müssen. Darüber hinaus bietet MPAL die Möglichkeit ohne Aufwand und ohne das Laufzeitverhalten zu verzerren, ein umfangreiches Profile zu erstellen und daraus Charakteristiken abzuleiten, wie im folgenden Abschnitt gezeigt wird.

Tabelle 6.5: Vergleich der Ausführungszeiten von HOG und SGM.

Plattform	SGM (*Tasking*)	HOG
4 x A7	587 ms	715 ms
4 x A15	219 ms	270 ms
big.Little	151 ms	196 ms
Mali-GPU	103 ms	700 ms
big.Little + Mali	70 ms/fr, 175 ms latency	161 ms
i5	75 ms	-
Stratix V (HLS)	3823 ms	-
i5 + Stratix V (HLS)	75 ms	-
Xeon Phi	46 ms	45 ms
Xeon E5-2680	20 ms	9 ms

6.3 Charakterisierungen

In diesem Abschnitt wird anhand eines praktischen Fallbeispiels gezeigt, wie eine Untersuchung des Skalierungsverhaltens deutlich Aufschluss über das

Laufzeitverhalten und die Charakteristik von parallelen Implementierungen gibt. Die gezeigten Messungen wurden automatisiert mit MPAL durchgeführt. Als exemplarisches Skalierungsverhalten wird hier der Verlauf von zwei unterschiedlichen Implementierungen desselben Algorithmus betrachtet. Es werden dabei die beiden Parallelisierungs-Optimierungen *Tasking* und *Streaming* des SGM-Algorithmus untersucht, die sich auf unterschiedlichen Plattformen in ihrer Charakteristik stark unterscheiden.

6.3.1 General-Purpose Desktop-Prozessor

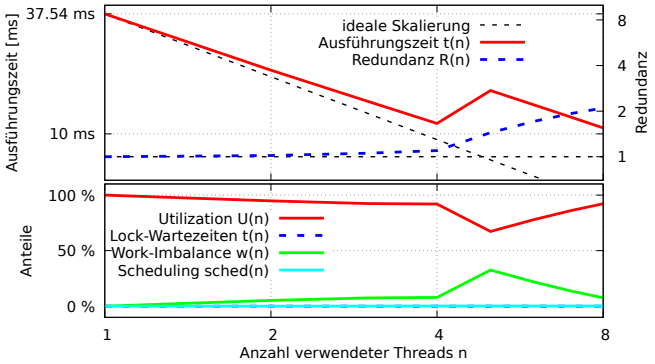
Alle im Folgenden gezeigten Analysen, Charakteristiken und Plots können mit den Werkzeugen des MPAL-Profilings automatisiert extrahiert bzw. generiert werden. Als erstes Beispiel, das hier analysiert wird, wird das Ausführungsverhalten beider Implementierungen auf einem i7-Quadcore bei 3,5 GHz betrachtet (Abbildung 6.8). Beide Optimierungen zeigen einen Anstieg der Redundanz bis auf etwa $R = 2$, wobei die *Tasking*-Optimierung ab Überschreiten der Hyperthreading-Grenze (in Abbildung 6.8 bei $n = 4$ Threads) Work-Imbalance aufweist, die *Streaming*-Optimierung dagegen Verzögerungen aufgrund von Synchronisationen (Locks).

Wie anhand der Performance-Counter-Daten in Tabelle 6.6 zu erkennen ist, minimiert die *Streaming*-Optimierung die Anzahl der benötigten Zugriffe auf den Hauptspeicher. Allerdings involviert sie dagegen eine hohe Synchronisationsrate, sowie zusätzliche Kommunikation zwischen den Prozessorkernen durch das Cache-Kohärenzprotokoll. Während die Anzahl der L1-Zugriffe aufgrund der benötigten Synchronisation für die *Streaming*-Optimierung höher ist, als für die *Tasking*-Optimierung, bleibt diese für beide Implementierungen konstant über die Anzahl der verwendeten Cores. Die verbesserte Datenlokalität wird durch die geringere Anzahl der LLC-Cache-Misses der *Streaming*-Optimierung im Vergleich zur *Tasking*-Optimierung unter Verwendung von nur einem Prozessorkern deutlich. Allerdings steigt die Anzahl der LLC-Misses im Gegensatz zu den L1-Zugriffen mit steigender Anzahl verwendeter Prozessorkerne.

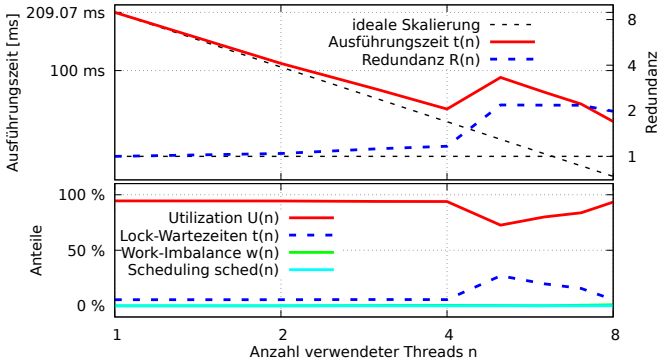
Unter Betrachtung der Verläufe der Cache-Performance-Counter in Abbil-

Tabelle 6.6: Performance-Counter beider Optimierungen (ein Thread).

Threads	**Ausführungszeit**		**L1-Zugriffe (LD+ST)** [Elemente]		**LLC-Misses (LD+ST)** [Cache-Lines]	
	Tasking	Streaming	Tasking	Streaming	Tasking	Streaming
1	154 ms	262 ms	583×10^6	832×10^6	93×10^3	19×10^3
8	42 ms	70 ms	583×10^6	831×10^6	162×10^3	145×10^3



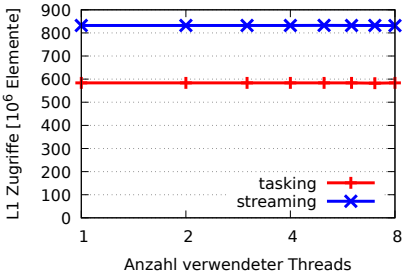
(a) SGM mit *Tasking*-Optimierung (ein Pfad).



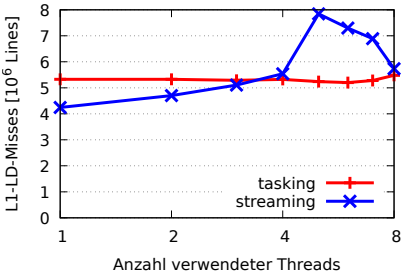
(b) SGM mit *Streaming*-Optimierung (vier Pfade).

Abbildung 6.8: Skalierungsverhalten und Skalierungsparameter beider SGM-Optimierungen auf einem i7-4771-Quadcore mit Hyperthreading ($n > 4$).

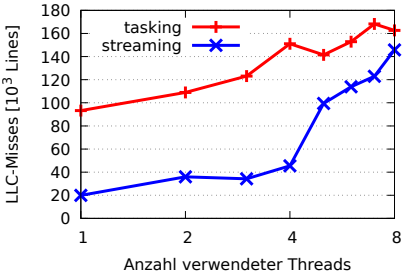
dung 6.9 werden Details der Skalierungscharakteristiken deutlich. Die *Tasking* Optimierung verteilt die Arbeit der Tasks durch eine Domain-Decomposition, wodurch die Anzahl der L1-Cache-Zugriffe weitestgehend konstant gehalten wird. Hyperthreading ($n > 4$) führt sogar zu einer geringen Verbesserung, da Cache-Lines im privaten L1-Cache für beide Threads verwendet werden können. Notwendige Synchronisationen einer Wellenfront-Parallelisierung, sowie notwendige Zugriffe auf die privaten Caches benachbarter Cores der *Streaming*-Optimierung sorgen dagegen für eine steigende L1-Cache-Miss-Rate mit steigender Anzahl verwendeter Kerne. Außerdem kann ein abrupter Anstieg der L1-Cache-Misses beobachtet werden, sobald die Grenze für Hyperthreading überschritten wird. Da sich jeweils zwei Hyperthreads einen physikalischen



(a) Anzahl der Zugriffe auf den L1-Cache.



(b) Anzahl der Load-Misses im L1-Cache



(c) Anzahl der Cache-Misses im LLC.

Abbildung 6.9: Verlauf der Performance-Counter beider Optimierungen (L1-ST-Misses sind nicht verfügbar) (Hyperthreading für $n > 4$).

Prozessorkern (und damit die Ressourcen der ALU) teilen, sind Hyperthreads häufig langsamer, als wenn nur je ein Thread pro Kern aktiv ist.

Diese Implementierung verwendet Threads, die sich untereinander synchronisieren, sodass auch die Threads, die keine Hyperthreads verwenden, deutlich verlangsamt werden, was auch im plötzlichen Anstieg der Lock-Verzögerungen in Abbildung 6.8 zu sehen ist. Da mit weiter ansteigender Anzahl an verwendeten Prozessorkernen auch die Anzahl der Hyperthreads steigt, die sich einen Kern teilen, wodurch dieser Effekt wieder kompensiert wird, sinken die L1-Misses wieder im weiteren Verlauf. In der *Streaming*-Optimierung wird eine Synchronisierung realisiert, indem noch nicht verfügbare Daten durch Mutex-Locks blockiert werden, und erst wenn Daten von einem Thread prozessiert und damit verfügbar gemacht wurden, löst dieser Thread die Blockierung auf und erlaubt damit die Verwendung der Daten durch andere Threads. Bei einer geringen Anzahl an Threads, die noch keine geteilte Hyperthread-Ressourcen

verwenden, laufen die Threads synchronisiert in einer Art Lock-Step und arbeiten daher auch auf derselben Region innerhalb des größeren Datensatzes, was zunächst die Datenlokalität verbessert. Die sich unterscheidenden Prozessierungsgeschwindigkeiten und die nur sehr unwahrscheinliche Tatsache, dass Hyperthreads, die gemeinsam auf einem Prozessorkern laufen, Daten aus einer benachbarten Region verarbeiten, beeinträchtigt die Datenlokalität, was auch anhand der LLC-Misses in Abbildung 6.9 sichtbar ist.

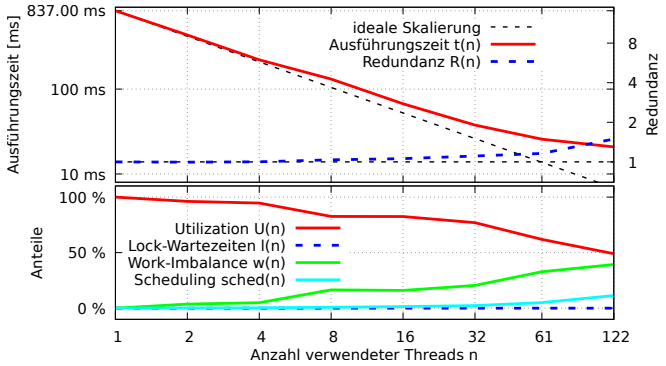
6.3.2 Intel Xeon Phi Manycore

In Abbildung 6.10 wird das Skalierungsverhalten der beiden Optimierungen des SGM-Algorithmus auf einem Intel Xeon Phi Manycore gezeigt, der bereits im oben aufgeführten Beispiel verwendet wurde. Abbildung 6.10 zeigt die großen Unterschiede des Skalierungsverhaltens der beiden Implementierungen auf diesem Prozessor. Während sich die Gesamtlaufzeit beider Optimierungen auf dem General-Purpose Prozessor zwar auf unterschiedliche Effekte zurückführen ließ, sich aber dennoch ähnliche Verläufe zeigten, sind auf diesem Prozessor deutliche Abweichungen in der resultierenden Gesamt-Performance zu erkennen.

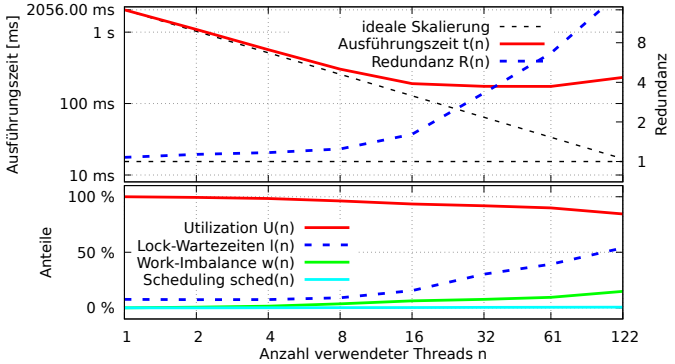
Das Skalierungsverhalten der *Tasking*-Optimierung zeigt, dass der Grund für eine nicht optimale Skalierbarkeit für eine steigende Anzahl an Prozessorkernen in der Abnahme der Utilization liegt. Wie in Abbildung 6.10(a) gezeigt, steigt die Redundanz nur gering aufgrund von Speichereffekten, wie Speicherbandbreite, Cache-Misses und Kohärenz-Protokoll ($R_{mem}(n) = 1,49$). Zum einen ist der Einfluss der Work-Imbalance ein stark limitierender Faktor, zum anderen beeinflusst auch das Parallelisierungs-Framework die parallele Ausführung aufgrund von Verzögerungen durch die initiale Verteilung der Tasks auf die große Anzahl der Kerne. Des Weiteren werden die zu verteilenden Tasks umso kleiner, je mehr Threads involviert sind, wodurch der Einfluss des Schedulings größer wird.

Im Gegensatz zur *Tasking*-Optimierung nimmt die Utilization bei der *Streaming*-Optimierung nur moderat ab. Allerdings ist die Skalierbarkeit auf eine minimale Ausführungszeit begrenzt, die bei einer Verwendung von 23 Prozessorkernen erreicht wird. Wie in Abbildung 6.10(b) zu sehen ist, ist die limitierte Skalierbarkeit einem Anstieg der Redundanz geschuldet. In diesem Fall zeigen Messungen, dass Wartezeiten aufgrund von Synchronisationen mit Mutex-Locks $l(n) = t_{lock}/(t(n) \cdot n)$ für die Verlängerung der Task-Ausführungszeiten verantwortlich sind.

Die *Tasking*-Optimierung minimiert die Anzahl der notwendigen Syn-



(a) SGM mit *Tasking*-Optimierung (ein Pfad).



(b) SGM mit *Streaming*-Optimierung (vier Pfade).

Abbildung 6.10: Skalierungsverhalten beider SGM-Optimierungen auf einem Intel Xeon Phi mit TBB parallelisiert (Hyperthreading ab $n > 61$).

chronisationen und Locks, um Kommunikationen zwischen den Kernen zu reduzieren. Da für jede individuelle algorithmische Stufe eine eigene Parallelisierung angewendet wird, werden mehrfache Spawns eingebracht. Daher wird diese Implementierung maßgeblich durch die involvierte Work-Imbalance und den Task-Verwaltungs-Overhead dominiert, die besonders für eine hohe Anzahl verwendeter Prozessorkerne die Laufzeit beeinflusst. Im Gegensatz dazu verwendet die *Streaming*-Optimierung eine Parallelisierung, die lediglich einen Spawn verwendet und den Speicherbedarf minimiert, dafür aber eine Vielzahl an Synchronisationen benötigt.

Die hier exemplarisch gezeigten Charakterisierungen von Skalierungsverläu-

fen ermöglichen die Analyse und den Vergleich des parallelen Laufzeitverhaltens. Dies ermöglicht wiederum dem Entwickler eine schnelle Identifikation der limitierenden Faktoren und damit eine effiziente Optimierung der Implementierung. Des Weiteren werden die hier vorgestellten Charakteristiken, die das Laufzeitverhalten aus einer abstrakten Perspektive darstellen, für die im nächsten Kapitel exemplarisch evaluierte Performance-Prädiktion genutzt.

6.4 Fallbeispiel Performance-Prädiktion

In dieser Fallstudie [84] wird die Performance der einzelnen algorithmischen Stufen des HOG- und des SGM-Algorithmus auf den vier eingebetteten Prozessoren eines Xilinx Zynq Ultrascale+ EG prädiziert. Es handelt sich um vier Cortex-A53 Prozessoren bei 1,2 GHz mit einer in-order Pipeline und 4 GB RAM (siehe Abbildung 6.11). Dazu wurde sowohl die in dieser Arbeit vorgestellte statistische Methode verwendet, als auch die beiden state-of-the-art-Prädiktionsmethoden GEM5 aus dem Bereich Virtual Prototyping und das Analytische Modell ExaBounds. Nachdem mit allen drei Methoden eine vollständige Prädiktion erzeugt wurde, werden Modellierungsaufwand, Simulationszeit und Prädiktionsgenauigkeit miteinander verglichen.

6.4.1 Prädiktionssetup

Der Aufbau einer vollständigen Prädiktionsumgebung variiert stark zwischen den verschiedenen Ansätzen, da alle unterschiedlichen Modellierungsmethoden mit variierendem Detailgrad und resultierenden Simulationsgeschwindigkeiten aufgebaut sind, wie in Abbildung 6.12 dargestellt. Daher wird im Folgenden die Einrichtung der Prädiktionen für alle drei Methoden gezeigt, wobei ein Virtual-Prototype mit GEM5 aufgebaut, ein analytisches Prozessormodell mit ExaBounds erstellt und eine umfangreiche Datenbank für den statistischen

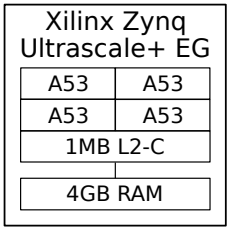


Abbildung 6.11: ARM Cortex-A53 Cores des Xilinx Zynq Ultrascale+ EG.

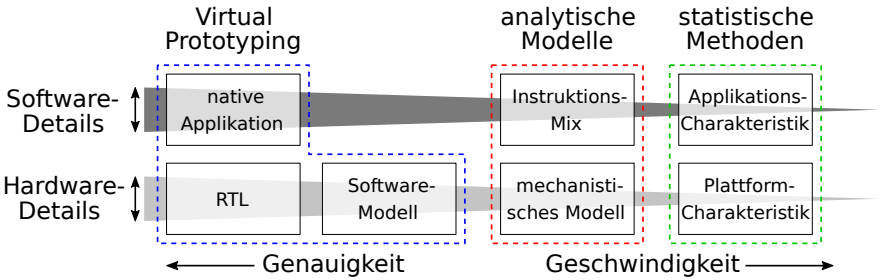


Abbildung 6.12: Detailgrad der unterschiedlichen Prädiktionsansätze.

Ansatz aus dieser Arbeit erzeugt wird. Zum Modellierungsprozess gehört für den Virtual-Prototype die Komposition aus Systemkomponenten und deren Parametrisierung. In ExaBounds wird ein generisches Prozessormodell zur Verfügung gestellt, das entsprechend der zu prädizierenden Hardware parametrisiert werden muss. Für diese Parametrisierung dienen im ersten Schritt die statischen Parameter, wie beispielsweise Anzahl der Prozessorkerne oder Cache-Größen. Es zeigt sich, dass eine detailliertere Konfiguration mit dynamischen Parametern, wie Speicherbandbreiten oder Cache-Miss-Latenzen, notwendig ist, da diese einen starken Einfluss auf die Prädiktionsgenauigkeit haben. Nicht für jede Plattform stehen alle dieser Parameter zur Verfügung, sodass sie beispielsweise durch Benchmarks extrahiert werden müssen, wodurch ein großer Mehraufwand für Modellierung entsteht und Messungenauigkeiten und dadurch Modellierungsfehler auftreten können. Der Aufbau einer Datenbank für die statistische Methode benötigt dagegen lediglich das Profiling der verfügbaren Benchmarks, wobei die Extraktion der Charakteristiken im Prozess automatisiert ist.

Virtual-Prototype: Als ein gängiges und weit verbreitetes state-of-the-art Framework dient GEM5 als Virtual-Prototyping-Framework mit einer zyklengenauen Simulation für die hier vorgestellte Fallstudie. Die Vorbereitung zur Prädiktion besteht aus den folgenden drei Schritten:

1. *Erstellung des Virtual Prototypes:* Zur Modellierung der ARM-Cores des Ultrascale+ SoCs wird zunächst das in GEM5 integrierte generische *HPI* ARMv8-Modell verwendet und mit statischen Parametern angepasst. Die Auswahl der weiteren Systemkomponenten erfordert nicht nur umfangreiche Kenntnisse über die Simulatoren, sondern auch über intere Details der Zielplattform. In diesem Fall wurden Standard-Komponenten aus GEM5 für beispielsweise RAM, Real-Time-Clock und serielle Konsole verwendet. Die Verbindung der einzelnen Komponenten, die in der realen Plattform über einen AXI-Bus verbunden

sind, wird hier mit einer zu TLM ähnlichen, ereignisbasierten Bus-Simulation realisiert. Die Zusammensetzung der einzelnen Komponenten und deren High-Level-Parametrisierung, wie Prozessor-Frequenz und Speichergröße, kann über eine Python-Schnittstelle konfiguriert werden.

2. *Software-Umgebung:* Im hier gezeigten Fall wird im sogenannten Full-System-Modus simuliert, was die Verwendung eines Betriebssystems (hier ein an die virtuelle Umgebung angepasster Linux-Kernel) ermöglicht, um die reale Ausführung möglichst präzise nachzubilden. Um das vollständige Dateisystem aufzubauen und das System mit Bibliotheken und für die Messung notwendigen Tools auszustatten, wird eine virtuelle Umgebung in QEMU (funktionale Emulation) verwendet. Diese ermöglicht es durch einen *Changeroot*-Befehl auf einem regulären Host-Rechner innerhalb einer Session ein beliebiges Verzeichnis als Root-Verzeichnis anzunehmen und dort Zielplattform-Binaries und Bibliotheken auszuführen. Durch die Verwendung von MPAL auch innerhalb des Virtual-Prototypes wird ohne manuelle Anpassungen des Software-Codes die Messung der simulierten Ausführungszeit möglich.

3. *Prädiktionsprozess:* Damit lange Systemstart-Zeiten vermieden werden können, wird der Betriebssystem-Kernel im Simulator zunächst in einer funktionalen Simulation gestartet und erst anschließend in die zyklengenaue Simulation gewechselt. Die zyklengenaue Simulation ist sehr zeitaufwändig, und die Simulation der beiden Algorithmen HOG und SGM benötigt etwa 10 Stunden, wohingegen die reale Plattform für deren Ausführung lediglich 550 ms benötigt. Der Simulationsprozess wird auch dadurch verlangsamt, dass selbst parallele Komponenten, wie die Prozessorkerne eses Multicore-Prozessors, nur in einem sequentiellen Prozess simuliert werden können. Daher benötigt in diesem Fall die Simulation der vier parallelen Prozessoren etwa 20 % mehr Simulationszeit als die Simulation einer sequentiellen Ausführung, aufgrund der zu simulierenden Kommunikation zwischen den Prozessorkernen.

Analytisches Modell: Der Aufbau der Prädiktionsumgebung und der Prädiktionsprozess mit ExaBounds, das ebenfalls ein sehr aktuelles analytisches Modell zur Performance Prädiktion verwendet [85], besteht aus den folgenden drei Schritten:

1. *Prozessor-Parametrisierung:* Das Prozessormodell, auf dem ExaBounds basiert, musste zunächst auf eine in-order Pipeline-Struktur mit mehreren Gleitkomma- und Ganzzahl-Operatoren, sowie Vektoreinheiten

konfiguriert werden. Zusätzlich wurden Prozessor-spezifische dynamische Parameter konfiguriert. Die Speicherhierarchie des vorgegebenen Modells mit privaten L2-Caches pro Kern musste angepasst werden, sodass der L2-Cache die Größe null und keine Latenzen hat, sodass der L3-Cache als Last-Level-Cache den eigentlich geteilten L2-Cache der Zielplattform repräsentiert.

2. *Software-Profiling:* Sobald das Prozessormodell aufgebaut ist, muss die Software durch ein architekturunabhängiges Profile charakterisiert werden, wozu im Modellierungsprozess ebenfalls auf das LLVM-Profiling zurückgegriffen wird. Sofern die Implementierung mit OpenMP parallelisiert ist, kann das Profiling parallel ausgeführt und dadurch entsprechend beschleunigt werden. Die Profiling-Zeit skaliert linear mit der Gesamtausführung des Algorithmus. Aufgrund der Verwendung externer Profiling-Tools musste eine manuelle Aufteilung der einzelnen algorithmischen Stufen erfolgen, um diese separat vermessen und prädizieren zu können.

3. *Analytische Berechnung:* Das Prozessormodell liegt in Form eines Wolfram Mathematica Notebooks vor, in das die gesamte Plattform-Beschreibung und das Software-Profile geladen werden. Die eigentliche Prädiktion kann in nur wenigen Sekunden pro algorithmischer Stufe berechnet werden, wobei ausführliche Informationen zur Laufzeit, aber auch Systemstatistiken, wie Cache-Misses oder verwendete Speicherbandbreite, generiert werden. Außerdem wird auch der Energieverbrauch der Plattform prädiziert.

Statistische Methode: Die statistische Prädiktion, die in dieser Arbeit vorgestellt wurde, basiert nicht auf einem zuvor manuell erstellten oder durch Machine-Learning trainierten Modell. Stattdessen besteht die Charakterisierung einer neuen Plattform und auch eines zu prädizierenden Workloads im Wesentlichen aus der Gesamtheit von mehreren Profile-Informationen. Aus diesem Grund wird angenommen, dass eine ausreichende Anzahl an Benchmarks für die Erstellung einer Datenbank vorhanden ist [59]. Der Prädiktionsablauf besteht hier insbesondere aus den folgenden zwei Punkten:

1. *Erstellung der Datenbank:* Mit dem bereitgestellten Satz aus insgesamt 18 Benchmarks werden sowohl die Zielplattform, als auch eine oder mehrere Referenzplattformen vermessen. Mit den jeweils unterschiedlichen Parallelisierungs-Strategien und individuellen Charakteristiken werden für diese Fallstudie sechs x86-Server, drei x86-Desktop-PC und zwei eingebettete ARM-Prozessoren der Datenbank hinzugefügt. Die

Benchmarks zeigen unterschiedliche Verhaltensweisen auf den verschiedenen Plattformen. Da für die präzise Parametrisierung des Virtual-Prototypes und des analytischen Modells ebenfalls großer Aufwand investiert wurde, wird auch für diesen Prädiktionsansatz eine ähnliche Plattform mit vier Cortex-A53 Prozessoren bei 1,5 GHz und 2 GB RAM der Datenbank hinzugefügt. Der Prozess der Extraktion von Charakterisierungen ist dabei vollständig automatisiert und benötigt keine weiteren Benutzereingaben.

2. *Berechnung der Prädiktion:* Der Prädiktionsprozess ist in einem Python-Skript implementiert und wird über die Kommandozeile aufgerufen, wobei die Datenbank übergeben wird, die Profiles der Benchmarks auf Ziel- und Referenzplattformen enthalten, sowie Profiles des zu prädizierenden Workloads auf den Referenzplattformen. Der Prädiktionsprozess schätzt neben der Gesamtausführungszeit immer einen vollständigen Skalierungsverlauf, sowie alle Skalierungsparameter, wie Locks, Work-Imbalance oder Redundanz.

6.4.2 Prädiktionsergebnisse

Abbildung 6.13 zeigt den prozentualen Prädiktionsfehler im Vergleich zur gemessenen Ausführungszeit auf der physikalischen Zielplattform für jede algorithmische Stufe HOG_{1-5}/SGM_{1-8} und für alle drei Prädiktionsmethoden. Aufgrund der unterschiedlichen Charakteristiken und Hardware-Anforderungen ergeben sich für alle Prädiktionsmethoden sowohl positive, als auch negative Abweichungen, was grundsätzlich für eine gute Parametrisierung spricht. Dadurch, dass sich positive und negative Fehler bei Akkumulation der einzelnen

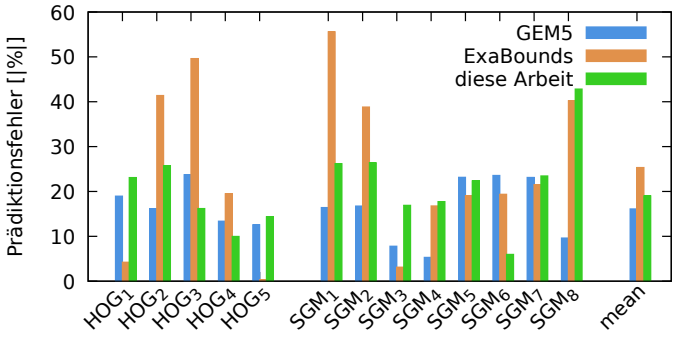


Abbildung 6.13: Fehler der drei getesteten Prädiktionsmethoden.

Ausführungszeiten für einen gesamten Algorithmus kompensieren, können alle Prädiktionsmethoden für die Prädiktion der gesamten Algorithmen SGM und HOG einen Fehler von $< 2\,\%$ erreichen. Da der akkumulierte Fehler allerdings keine Aussage über die tatsächliche Prädiktionsgenauigkeit zulässt, werden im Folgenden die Mittelwerte der absoluten Fehler aller einzelnen Stufen betrachtet. Der Virtual-Prototype generiert die präzisesten Prädiktionen mit einem mittleren Fehler von 16,1 %, was auf die zyklengenaue Simulation des Gesamtsystems in GEM5 und die zusätzliche Verwendung des Betriebssystems zurückzuführen ist, wie es auch auf der Zielplattform zum Einsatz kommt. Das analytische Modell liefert Prädiktionen mit dem höchsten mittleren Fehler von 25,3 %, da hier zum einen der Einfluss des Betriebssystems nicht betrachtet wird und zum anderen das Modell der Speicher-Hierarchie nicht mit der Zielplattform übereinstimmt. Des Weiteren wurde das architekturunabhängige Profile des zu prädizierenden Workloads auf einem x86-Prozessor erstellt; die Anzahl der Operationen unterscheidet sich hierbei aber stark von der ARM-Zielplattform. Der mittlere Fehler der statistischen Prädiktionsmethode liegt trotz geringer Komplexität der mathematischen Modelle und minimalem Modellierungsaufwand mit 19,0 % zwischen den beiden Vergleichswerten, was durch eine gute Qualität der Datenbank erreicht werden konnte.

Tabelle 6.7: Zusammenfassung der vergleichenden Fallstudie.

	Flexibilität	Präzision	Prädiktionsdauer	Modellierungsaufwand	Singlecore	Multicore	Heterogenität
GEM5	++	+	−− (10 Std.)	−− (32 Tage)	√	√	√
IBM ExaBounds	−	o	++ (< 5 Sek.)	− (10 Tage)	√	√	X
stat. Methode	o	+	++ (< 5 Sek.)	+ (< 1 Std.)	√	√	X

Wie in dieser Fallstudie gezeigt und in Tabelle 6.7 ausgewertet wurde, bietet Virtual-Prototyping die präziseste Prädiktion, erfordert aber sowohl einen hohen Modellierungsaufwand (hier ca. einen Monat) und lange Simulationszeiten (hier ca. 10 Stunden). Dafür sind die resultierenden Systemsimulatoren nicht auf Multicores beschränkt, sondern es können eine Vielzahl von unterschiedlichen Systemen auch mit heterogenen Beschleuniger-Komponenten simuliert werden. Analytische Modelle bieten eine sehr schnelle Prädiktion (wenige Sekunden), sofern ein Prozessormodell vorliegt, wobei die Parametrisierung eines neuen Prozessors ebenfalls einen Modellierungsaufwand erfordern kann (hier etwas mehr als eine Woche). Das Erstellen des architekturunabhängigen Profiles erfordert ebenfalls eine lange Messung (hier ca. 6 Stunden) und zudem manuellen Aufwand zum Aufteilen der algorithmischen Stufen, um diese separat messen und charakterisieren zu können. Das Erzeugen einer umfangreichen und qualitativen Datenbank für die statistische Methode erfordert das Profilen der Benchmarks auf verschiedenen Plattformen. Sind die Plattform-Charakterisierungen in einer Datenbank bereits gegeben, ist für das Hinzufügen einzelner

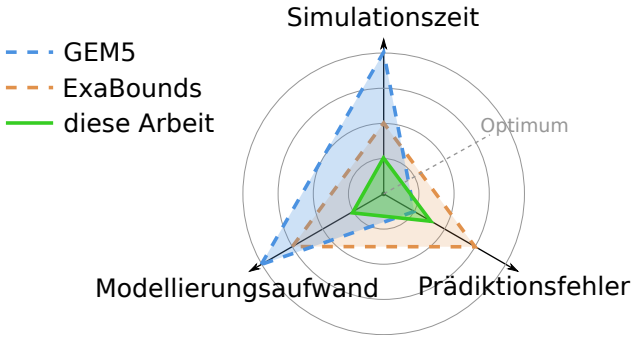


Abbildung 6.14: Klassifikation der drei evaluierten Prädiktionsmethoden.

Plattformen oder neuer Workloads nur ein geringer Profiling-Aufwand nötig. Bei gegebener Datenbank kann eine anschließende Prädiktion ebenfalls in wenigen Sekunden berechnet werden.

Eine Zusammenfassung der Ergebnisse ist in Abbildung 6.14 gegeben. Demnach liefert die instruktionsgenaue Simulation mit dem GEM5-Virtual-Prototype die präziseste Performance-Prädiktion, benötigt aber sowohl einen hohen Modellierungsaufwand, als auch eine lange Simulationsdauer. Virtual-Prototyping bietet außerdem den Vorteil, dass jeder Architekturtyp simuliert werden kann, wodurch auch die Simulation von heterogenen parallelen Plattformen möglich ist. Das analytische ExaBounds-Modell benötigt einen geringeren Modellierungsaufwand, liefert aber einen größeren Prädiktionsfehler. Die in dieser Arbeit vorgestellte statistische Prädiktionsmethode basiert ausschließlich auf extrahierten Profiles, wodurch der Aufwand für die Modellierung sehr klein wird, wenn bereits eine Datenbank existiert.

7 Zusammenfassung

Diese Arbeit behandelt im Wesentlichen die Programmierung von homogenen und heterogenen parallelen Architekturen von HPC-Plattformen bis zu eingebetteten MPSoCs. Dabei wurden speziell Shared-Memory-Parallelisierungen betrachtet, wie sie bei den gängigsten Multicore-Anwendungen, wie beispielsweise bei eingebetteten Systemen im Bereich des automatisierten Fahrens, eingesetzt werden. Es wurden Programmiermodelle, Untersuchungen und Modellierungen von Effekten der Nebenläufigkeit, sowie Methoden zur Prädiktion der parallelen Performance speziell im Hinblick auf die Entwicklung von Performance-portabler Software vorgestellt.

Zunächst wurden Effekte betrachtet, die durch eine nebenläufige Programmausführung entstehen, und das Laufzeitverhalten und die Skalierbarkeit einer Anwendung beeinflussen. Es wurden unterschiedliche Programmiermodelle vorgestellt, die für die Entwicklung homogener und heterogener Performance-portabler Software eingesetzt werden können. Dabei wurde sowohl die notwendige sequentielle Optimierung von Software, Möglichkeiten zur Vektorisierung, sowie die Parallelisierung für Mehrkernprozessoren und Beschleuniger, wie GPUs oder FPGAs, behandelt. Es wird deutlich, dass trotz einiger Lösungsansätze eine einheitliche und portable Programmierung weiterhin nicht oder nur eingeschränkt möglich ist und die Erstellung einer flexiblen, parallelen Implementierung eine komplexe Aufgabe darstellt. Des Weiteren zeigt sich, dass das Zusammenwirken der von der Software geforderten Ressourcen und von der Hardware tatsächlich angebotenen Kapazitäten die parallele Performance stark beeinflusst, weshalb eine Parallelisierungs-Strategie speziell im Hinblick auf die Zielplattform gewählt werden sollte.

Um eine Vereinheitlichung der parallelen Programmierung zu erreichen, wurde die Abstraktionsschicht MPAL vorgestellt, die speziell die Portierbarkeit von parallelen Implementierungen adressiert. Da auch das angewendete Pa-

rallelisierungs-Framework durch die implementierte Scheduling-Strategie das parallele Laufzeitverhalten stark beeinflussen kann, bietet MPAL eine eigene API zur parallelen Programmierung an. Im Hintergrund kann durch einfache Konfiguration in einer Konfigurationsdatei das tatsächlich angewendete Parallelisierungs-Framework gewählt werden, sodass für jede Plattform eines der verfügbaren bzw. das jeweils am besten skalierende Framework angewendet wird. Auf die gleiche Weise lassen sich auch heterogene Implementierungen für Beschleuniger realisieren, wodurch das Architektur-Mapping ebenfalls mithilfe der Konfigurationsdatei flexibel konfigurierbar gemacht wird. Darüber hinaus bietet die Kapselung aller relevanten Parallelisierungs-Aufrufe in der Zwischenschicht, den Vorteil des vollen Zugriffs auf das parallele Laufzeitverhalten. Es wurden in MPAL integrierte Profiling-Methoden vorgestellt, die es ermöglichen, ein hochpräzises und Nebenläufigkeits-fokussiertes paralleles Profiling zu erstellen, ohne dabei Sampling-Overhead zu generieren.

In der Methoden-Bewertung wurden unterschiedliche homogene und heterogene Implementierungen vorgestellt, die auf verschiedenen Architekturen ausgeführt und evaluiert wurden. Es wurden in Fallbeispielen detaillierte Analysen der Parallelisierungs-Effekte vorgestellt, die zeigen, dass es große Unterschiede in den Laufzeiten nach einer Migration geben kann. Außerdem wird deutlich, dass die Kombination der Parallelisierungs-Strategie mit der Architektur der Zielplattform einen massiven Einfluss auf das resultierende Laufzeitverhalten und den Skalierungsverlauf hat. Durch die Verwendung der Abstraktionsschicht MPAL konnten auch die heterogenen Implementierungen schnell in ihrem Architektur-Mapping verändert und dadurch leicht portiert werden. Die hohe Anzahl der gezeigten Portierungen der beiden Algorithmen SGM und HOG zeigt, dass durch die Verwendung von MPAL die Portabilität stark gesteigert werden kann. Auch ermöglichen die integrierten Profiling-Methoden eine einfache Analyse des Skalierungsverhaltens, das schnell Aufschluss über Effekte der Nebenläufigkeit und mögliche Limitierungen gibt.

Aus den gewonnenen detaillierten Profiles des parallelen Laufzeitverhaltens wurden Metriken und Charakterisierungen entwickelt und exemplarisch Unterschiede im Skalierungsverlauf betrachtet. Neben einer schnellen Analyse des Verhaltens, u.a. zur Identifikation von Bottlenecks, kann das Verhalten durch Modellfunktionen nachgebildet und durch nur wenige signifikante Modell-Parameter repräsentiert werden. Dies erlaubt anschließend einen direkten quantitativen Vergleich von Skalierungseigenschaften. Daraus können umso schneller kritische Eigenschaften des Laufzeitverhaltens sowie Limitierungen identifiziert werden. Die extrahierten Modell-Parameter können durch Konkatenation mit Performance-Countern ebenfalls als deskriptiver Feature-Vektor verwendet werden, der für Machine-Learning-Ansätze zur Prädiktion der Performance angewendet werden kann. Der Skalierungsvektor hat zusätzlich den Vorteil,

dass der vollständige Skalierungsverlauf und alle Parallelisierungsparameter daraus rekonstruiert werden können.

In einer neuen statistischen Prädiktionsmethode wurde gezeigt, wie die extrahierten Skalierungsvektoren verwendet werden können, um die parallele Performance nach der Migration auf eine nicht verfügbare Plattform vorherzusagen. Dabei wird nicht nur die Gesamt-Performance unter Verwendung aller verfügbaren Prozessorkerne, sondern der gesamte Skalierungsverlauf mitsamt aller Parallelisierungs-Parameter prädiziert. Somit können bereits vor einer Migration das Verhalten und mögliche Limitierungen und Bottlenecks abgeschätzt werden. Eine Fallstudie, in der die neue statistische Methode mit einem Virtual-Prototype und einer analytischen Prädiktionsmethode verglichen wurde, zeigte, dass das vorgestellte Verfahren gute Ergebnisse liefert und nur minimalen Modellierungs-Aufwand benötigt. Da zum aktuellen Zeitpunkt nur eine kleine Datenbank und nur wenige Benchmarks verwendet wurden, ist davon auszugehen, dass diese vielversprechende Forschungsrichtung in Zukunft noch präzisere Prädiktionen liefern kann.

Diese Arbeit – besonders die einleitenden Kapitel – kann als Zusammenfassung von Methoden und Werkzeugen und somit als Hilfestellung für die effiziente parallele Programmierung auf unterschiedlichen Ebenen von Hardware und Software gesehen werden. Die vorgestellte Charakterisierung stellt dabei eine systematische Untersuchung der Effekte dar. Daraus können weitere Klassifikatoren und Merkmale (wie Parallelisierungs-Strategie oder Anfälligkeiten auf bestimmte Hardware-Begrenzungen) abgeleitet werden, mithilfe derer in Zukunft automatisiert Parallelisierungs-Hinweise erstellt werden können. So könnten beispielsweise dem Entwickler automatisch Problematiken einer vorliegenden parallelen Implementierung aufgezeigt oder Vorschläge für eine geeignete Parallelisierungs-Strategie einer vorliegenden sequentiellen Implementierung gemacht werden. Ebenso ist die Prädiktion Teil einer Reihe von Werkzeugen, die dem Entwickler bereits im frühen Stadium der parallelen Entwicklung eine Abschätzung des Parallelisierungs-Erfolgs bereitstellen. Die Prädiktion kann grundsätzlich für die Beantwortung von Fragestellungen aus zwei Perspektiven genutzt werden: Entweder für die Suche nach einer geeigneten Plattform für eine vorhandene parallele Implementierung oder für die Wahl der richtigen Parallelisierungs-Strategie, wenn die Zielplattform bereits gegeben ist.

A MPAL Befehlssatz

Befehl	Definition
class ParallelWorker (CPU-Parallelisierung):	
spawn($\vec{s}$, f, t)	Homogener Spawn der Funktion f mit der aufzuteilenden Sequenz $\vec{s}$ (z.B. Datenbereich) über t Tasks.
spawn($\{\vec{s_i}, f_i, t_i, \vec{c_i}\}_i$)	Heterogener Spawn der Funktionen f_i, die den CPU-Cores c_i zugewiesen sind (Architektur-Mapping wird meist automatisch aus Konfigurationsdatei erzeugt).
createMutex()	Erzeugt framework-unabhängig ein Mutex-Lock.
getNumThreads()	Rückgabe der Anzahl verwendeter Threads.
getCPU()	Rückgabe des aktuellen CPU-Cores.
getTID()	Rückgabe der aktuellen Thread-ID.
class Offloader (gekapselt in ParallelWorker):	
loadKernel(file)	Lädt Kernel aus Datei (Source-Code für JIT-Compile oder Bitstream für FPGA)
offload($\vec{s}$, k, u)	Startet den Kernel k auf dem Beschleuniger u.
wait(k)	Synchronisiert mit dem Kernel k (blockiert).
createBuffer(size)	Erstellt framework-abhängig einen hybriden Buffer, der je nach Abfrage des CPU- oder Beschleuniger-Pointers die Daten mappt.
getNumDevices()	Rückgabe der Anzahl verwendbarer Beschleuniger.

class Profiler (gekapselt in ParallelWorker):	
start-/stopMeasure()	Startet/stoppt die Messung (wird beim Erzeugen / Zerstören des ParallelWorker oder beim Betreten / Verlassen der MPAL_ALGO-Sektion aufgerufen).
saveProfile()	Speichert ein Profile zur Analyse (wird bei Verwendung der MPAL_SECTION-Sektion und Konfigurationsdatei automatisiert).
Profile-Cruncher:	
calltree	Zeigt den Verzweigungsgraph eines parallelen Programms mit Task-Ausführungszeiten.
coretree	Zeigt zeitliche Abfolge der Tasks je CPU-Core.
parameter	Zusammenfassung der Parallelisierungs-Parameter.
parameter-per-spawn	Zeigt Parallelisierungs-Parameter je Spawn-Aufruf.
Performance Evaluation Tool:	
showProfile	Visualisiert Skalierungsverläufe (alle Parameter).
fitProfile	Extrahiert Modell-Parameter aus Profiles.
showParameters	Visualisiert extrahierte Modell-Parameter zum direkten Vergleich (bspw. Bottlenecks-Analyse).
predict	Nutzt Modell-Parameter aus Benchmarks zur Prädiktion eines Skalierungsverlaufs.

Literaturverzeichnis

[1] H. Blume, H. Hübert, H. T. Feldkämper, and T. G. Noll. Model-based Exploration of the Design Space for Heterogeneous Systems on Chip. In *Intl. Conf. Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2002.

[2] H. Hirschmüller. Accurate and Efficient Stereo Processing by Semi-Global Matching and Mutual Information. In *Intl. Conf. Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2005.

[3] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High-Performance Programming*. Newnes, 2013.

[4] H. Winner, S. Hakuli, F. Lotz, and C. Singer. *Hansbuch Fahrerassistenzsysteme*. Springer, 3 edition, 2015.

[5] G. Payá-Vayá and H. Blume. *Towards a Common Software/Hardware Methodology for Future Advanced Driver Assistance Systems*. River Publishers, 2017.

[6] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[7] G. Blake, R. G. Dreslinski, and T. Mudge. A Survey of Multicore Processors. *Signal Processing Magazine*, 26(6):26–37, 2009.

[8] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[9] OpenACC. OpenACC – More Science, Less Programming. `https://www.openacc.org/`, 2019.

[10] OpenMP. OpenMP – The OpenMP API Specification for Parallel Programming. `https://www.openmp.org/`, 2019.

[11] R. Landaverde, T. Zhang, A. K Coskun, and M. Herbordt. An Investigation of Unified Memory Access Performance in CUDA. In *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014.

[12] HSA. HSA Foundation, ARM, AMD, Imagination, MediaTek, Qualcomm, Samsung, TI. `http://www.hsafoundation.com/`, 2019.

[13] O. J. Arndt, D. Becker, C. Banz, and H. Blume. Parallel Implementation of Real-Time Semi-Global Matching on Embedded Multi-Core Architectures. In *Intl. Conf. Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE, 2013.

[14] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. Lulu.com, 2 edition, 2015.

[15] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[16] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[17] Intel. Intel Threading Building Blocks Documentation. `https://software.intel.com/en-us/node/506039`, 2019.

[18] Intel. Intel CilkPlus. `https://www.cilkplus.org`, 2019.

[19] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance Gaps between OpenMP and OpenCL for Multi-core CPUs. In *Intl. Conf. Parallel Processing Workshops (ICPPW)*. IEEE, 2012.

[20] G. Tournavitis, Z. Wang, B. Franke, and M. O'Boyle. Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping. *SIGPLAN Notices*, 44(6):177–187, 2009.

[21] J. Castrillón Mazo. *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. Dissertation, Rheinisch–Westfälische Technische Hochschule Aachen (RWTH), 2013.

[22] M. A. Aguilar, R. Leupers, G. Ascheid, and L. G. Murillo. Automatic Parallelization and Accelerator Offloading for Embedded Applications on Heterogeneous MPSoCs. In *Design Automation Conference (DAC)*. ACM, 2016.

[23] T. G. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 1 edition, 2004.

[24] A. Williams. *C++ Concurrency in Action*. Manning Publications, 2 edition, 2019.

[25] R. Grimm. *Modernes C++: Concurrency meistern*. Hanser Verlag, 1 edition, 2018.

[26] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 1 edition, 2012.

[27] U. Gleim and T. Schüle. *Multicore-Software: Grundlagen, Architektur und Implementierung in C/C++, Java und C#*. dpunkt, 1 edition, 2012.

[28] T. Rauber and G. Rünger. *Parallele Programmierung*. Springer, 2 edition, 2007.

[29] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[30] Khronos Group. Vulkan – Industry Forged. `https://www.khronos.org/vulkan/`, 2019.

[31] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Intl. Symp. Microarchitecture (MICRO)*. IEEE/ACM, 2009.

[32] C. Nugteren and V. Codreanu. CLTune: A Generic Auto-Tuner for OpenCL Kernels. In *Intl. Symp. Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2015.

[33] Khronos Group. OpenVX – Portable, Power-efficient Vision Processing. `https://www.khronos.org/openvx/`, 2019.

[34] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Intl. Conf. Multimedia*. ACM, 2014.

[35] TensorFlow. TensorFlow – An end-to-end open source machine learning platform. `https://www.tensorflow.org/`, 2019.

[36] LinuxTV.org. Video for linux two api speciifcation. `https://www.linuxtv.org/downloads/legacy/video4linux/API/V4L2_API/spec-single/v4l2.html`, 2019.

[37] J. L. Manferdelli, N. K. Govindaraju, and C. Crall. Challenges and Opportunities in Many-Core Computing. *Proceedings of the IEEE*, 96(5):808–815, 2008.

[38] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *joint computer conference*, pages 483–485. ACM, 1967.

[39] Free Software Foundation. GNU Binutils. `hhttp://www.gnu.org/software/binutils/`, 2019.

[40] Intel. Intel VTune Amplifier. `https://software.intel.com/en-us/vtune`, 2019.

[41] BSC-CNS. Performance Tools. `https://www.bsc.es/discover-bsc/organisation/scientific-structure/performance-tools`, 2019.

[42] NVIDIA. CUDA Zone. `https://developer.nvidia.com/cuda-zone`, 2019.

[43] Khronos Group. SyCL – C++ Single-source Heterogeneous Programming for OpenCL. `https://www.khronos.org/sycl/`, 2019.

[44] NVIDIA. Thrust :: CUDA Toolkit Documentation. `https://docs.nvidia.com/cuda/thrust/index.html`, 2019.

[45] Khronos Group. SPIR – The Industry Open Standard Intermediate Language for Parallel Compute and Graphics. `https://www.khronos.org/spir/`, 2020.

[46] P. Jääskeläinen, J. Glossner, M. Jambor, A. Tervo, and M. Rintala. Offloading C++ 17 Parallel STL on System Shared Virtual Memory Platforms. In *Intl. Conf. High Performance Computing: ISC High Performance Workshops*. Springer, 2018.

[47] M. Voss, R. Asenjo, and J. Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. Apress, 2019.

[48] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[49] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[50] O. J. Arndt, T. Lefherz, and H. Blume. Abstracting Parallel Programming and Its Analysis Towards Framework Independent Development. In *Intl. Symp. Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2015.

[51] O. J. Arndt, F. D. Träger, T. Moß, and H. Blume. Portable Implementation of Advanced Driver-Assistance Algorithms on Heterogeneous Architectures. In *Intl. Symp. Parallel and Distributed Processing Workshops (IPDPSW)*. IEEE, 2017.

[52] ARM. Ten Things to Know About big.LITTLE, 2013. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/ten-things-to-know-about-big-little.

[53] B. Veenboer and J. W. Romein. Radio-Astronomical Imaging: FPGAs vs GPUs. In *Intl. European Conf. Parallel and Distributed Computing (Euro-Par)*. Springer, 2019.

[54] M. Gregoire. *Professional C++*. Wiley, 4 edition, 2018.

[55] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[56] O. J. Arndt, C. Spindeldreier, K. Wohnrade, D. Pfefferkorn, M. Neuenhahn, and H. Blume. FPGA Accelerated NoC-Simulation: A Case Study on the Intel Xeon Phi Ringbus Topology. In *Intl. Symp. Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*. ACM, 2017.

[57] S. Madougou, A. Varbanescu, C. de Laat, and R. van Nieuwpoort. The landscape of GPGPU performance modeling tools. *Parallel Computing*, 56:18–33, 2016.

[58] D. Gill and H. M. Pandey. Approaches for Software Performance Modelling, Cloud Computing and Openstack. *Intl. Journal of Computer Application*, 119(22):31–35, 2015.

[59] O. J. Arndt, M. Lüders, and H. Blume. Statistical Performance Prediction for Multicore Applications Based on Scalability Characteristics. In *Intl. Conf. Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2019.

[60] J. Teich. Hardware/Software Codesign: The Past, the Present, and Predicting the Future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 2012.

[61] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, S. Rathijit, K. Swell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wook. The gem5 Simulator. *Computer Architecture News*, 39(2):1–7, 2011.

[62] Imperas Software. Open virtaul platforms – the source of fast processor models & platforms. `http://www.ovpworld.org/`, 2019.

[63] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Annual Technical Conference (ATEC)*. USENIX, 2005.

[64] Cadence. Virtual system platform – an open, connected, and scalable virtual prototyping solution. `https://www.cadence.com/content/cadence-www/global/ja_JP/home/tools/system-design-and-verification/software-driven-verification/virtual-system-platform.html`, 2019.

[65] A. Butko, A. Gamatié, G. Sassatelli, L. Torres, and M. Robert. Design Exploration For Next Generation High-Performance Manycore On-chip Systems: Application To big.LITTLE Architectures. In *Computer Society Annual Symposium on VLSI*. IEEE, 2015.

[66] ARM. Fast models. `https://developer.arm.com/tools-and-software/simulation-models/fast-models`, 2019.

[67] G. Marin and J. Mellor-Crummey. Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models. In *SIGMETRICS Performance Evaluation Review*. ACM, 2004.

[68] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An Evaluation of High-Level Mechanistic Core Models. *Transactions Architecture and Code Optimization (TACO)*, 11(3):28:1–28:25, 2014.

[69] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache Contention and Application Performance Prediction for Multi-Core Systems. In *Intl. Symp. Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2010.

[70] S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Micro-architecture independent analytical processor performance and power modeling. In *Intl. Symp. Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2015.

[71] Sander De Pestel, Sam Van den Steen, Shoaib Akram, and Lieven Eeckhout. RPPM: Rapid Performance Prediction of Multithreaded Applications on Multicore Hardware. *Computer Architecture Letters*, 17(2):183–186, 2018.

[72] R. Jongerius, A. Anghel, G. Dittmann, G. Mariani, E. Vermij, and H. Corporaal. Analytic Multi-Core Processor Model for Fast Design-Space Exploration. *Transactions on Computers*, 67(6):755–770, 2018.

[73] Unai Lopez-Novoa, Alexander Mendiburu, and Jose Miguel-Alonso. A Survey of Performance Modeling and Simulation Techniques for Accelerator-Based Computing. *Transactions on Parallel and Distributed Systems*, 26(1):272–281, 2014.

[74] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu. Cross-Architecture Performance Prediction (XAPP) Using CPU Code to Predict GPU Performance. In *Intl. Symp. Microarchitecture*. ACM, 2015.

[75] K. Hoste and L. Eeckhout. Microarchitecture-Independent Workload Characterization. *Micro*, 27(3):63–72, 2007.

[76] T. G. Dietterich. Ensemble Methods in Machine Learning. In *Multiple Classifier Systems*. Springer, 2000.

[77] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram. GROPHECY: GPU Performance Projection from CPU Code Skeletons. In *Intl. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2011.

[78] Y. Wang, V. Lee, G.-Y. Wei, and D. Brooks. Predicting New Workload or CPU Performance by Analyzing Public Datasets. *Transactions on Architecture and Code Optimization (TACO)*, 15(4):53:1–53:21, 2019.

[79] D. Terpstra, H. Jagode, H. You, and Jack Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[80] T. Hoefler and R. Belli. Scientific Benchmarking of Parallel Computing Systems: Twelve ways to tell the masses when reporting performance results. In *Intl. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*. ACM/IEEE, 2015.

[81] N. Dalal and B. Triggs. Histograms of Oriented Gradients for Human Detection. In *Intl. Conf. Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2005.

[82] O. J. Arndt, D. Becker, F. Giesemann, G. Payá-Vayá, C. Bartels, and H. Blume. Performance Evaluation of the Intel Xeon Phi Manycore Architecture Using Parallel Video-Based Driver Assistance Algorithms. In *Intl. Conf. Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE, 2014.

[83] O. J. Arndt, T. Linde, and H. Blume. Implementation and Analysis of the Histograms of Oriented Gradients Algorithm on a Heterogeneous Multicore CPU/GPU Architecture. In *Global Conf. Signal and Information Processing (GlobalSIP)*. IEEE, 2015.

[84] M. Lüders, O. J. Arndt, and H. Blume. Multicore Performance Prediction – Comparing Three Recent Approaches in a Case Study. In *Intl. European Conf. Parallel and Distributed Computing Workshops (Euro-Par Workshops)*. IEEE, 2019.

[85] R. Jongerius, G. Mariani, A. Anghel, G. Dittmann, E. Vermij, and H. Corporaal. Analytic Processor Model fori Fast Design-Space Exploration. In *Intl. Conf. Computer Design (ICCD)*. IEEE, 2015.

Publikationen des Autors

Buchbeiträge:

Arndt, O. J.; Rallapalli, P.; Blume, H.: *Portable Implementations for Heterogeneous Hardware Platforms in Autonomous Driving Systems (Chapter 6)*, Big Data Analytics in Cyber-Physical Systems, Elsevier (2019)

Journalbeiträge:

Arndt, O. J.; Lüders, M.; Riggers, C.; Blume, H.: *Multicore Performance Prediction with MPET – Using Scalability Characteristics for Statistical Cross-Architecture Prediction*, In Journal of Signal Processing Systems, Springer (2020)

Konferenzbeiträge:

Lüders, M.; Arndt, O. J.; Blume, H.: *Multicore Performance Prediction – Comparing Three Recent Approaches in a Case Study*, Intl. European Conf. Parallel and Distributed Computing (Euro-Par 2019), Springer (2019)

Arndt, O. J.; Lüders, M.; Blume, H.: *Statistical Performance Prediction for Multicore Applications Based on Scalability Characteristics*, Intl. Conf. Application-specific Systems, Architectures and Processors (ASAP-19), IEEE (2019)

Arndt, O. J.; Spindeldreier, C.; Wohnrade, K.; Pfefferkorn, D.; Neuenhahn, M.; Blume, H.: *FPGA Accelerated NoC-Simulation – A Case Study on the Intel Xeon Phi Ringbus Topology*, Intl. Symp. Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2017), ACM (2017)

Arndt, O. J.; Träger, F. D.; Moß, T.; Blume, H.: *Portable Implementation of Advanced Driver-Assistance Algorithms on Heterogeneous Architectures*, Heterogeneity in Computing Workshop (HCW-17), hosted at Intl. Parallel and Distributed Processing Symp. Workshops (IPDPSW 2017), IEEE (2017)

Arndt, O. J.; Lefherz, T.; Blume, H.: *Abstracting Parallel Programming and its Analysis Towards Framework Independent Development*, Intl. Symp. Embedded Multicore/Many-Core Systems-on-Chip (MCSoC-15), IEEE (2015)

Arndt, O. J.; Linde, T.; Blume, H.: *Implementation and Analysis of the Histograms of Oriented Gradients Algorithm on a Heterogeneous Multicore CPU/GPU Architecture*, Global Conf. Signal & Information Processing (GlobalSIP 2015), IEEE (2015)

Behmann, N.; Arndt, O. J.; Blume, H.: *Parallel Implementation of Real-Time Block-Matching based Motion Estimation on Embedded Multi-Core Architectures*, ICT.OPEN (2015)

Arndt, O. J.; Becker, D.; Giesemann, F.; Payá Vayá, G.; Bartels, C.; Blume, H.: *Performance Evaluation of the Intel Xeon Phi Manycore Architecture Using Parallel Video-Based Driver Assistance Algorithms*, Intl. Conf. Embedded Computer Systems (SAMOS XIV), IEEE (2014)

Arndt, O. J.; Becker, D.; Banz, C.; Blume, H.: *Parallel Implementation of Real-Time Semi-Global Matching on Embedded Multi-Core Architectures*, International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), IEEE (2013)

Arndt, O. J.; Scheuermann, B.; Rosenhahn, B.: *"RegionCut" — Interactive multi-label segmentation utilizing cellular automaton*, Workshop on Applications of Computer Vision (WACV), IEEE (2013)

Eingeladene Vorträge::

Arndt, O. J.; Blume, H.: *Multicore and Manycore Architectures for Video-Based Advanced Driver Assistance Systems*, Eingeladener Vortrag bei Konferenz "Multicore@Siemens 2015 - Parallel Software Development", Nürnberg (2015)

Arndt, O. J.; Becker, D.; Blume, H.: *Parallele Implementierung eines Semi-Global Matching auf eingebetteten Multi-Core Architekturen unter Echtzeitbedingungen*, Eingeladener Vortrag bei Konferenz "Parallel 2013", Karlsruhe (2013)

Lebenslauf

Persönliche Angaben:

Name:	Oliver Jakob Arndt
Geburtsdatum, -ort:	20.09.1988, Dortmund

Wissenschaftlicher Werdegang:

2008 – 2013	Studium Elektrotechnik (Dipl.-Ing.), Vertiefungsrichtung: technische Informatik, Leibniz Universität Hannover
2011	Studienarbeit: *Konzeptionierung und Implementierung eines Low-Power Netzwerkprotokolls für Wireless Sensor Networks*
2013	Diplomarbeit: *Strategien zur parallelen Implementierung von Fahrerassistenzalgorithmen auf Many-Core-Plattformen*
2009 – 2013	Hilfswissenschaftler, Institut für Informationsverarbeitung (TNT), Leibniz Universität Hannover
2013 – 2020	Wissenschaftlicher Mitarbeiter und Promotionsstudium, Institut für Mikroelektronische Systeme (IMS), Leibniz Universität Hannover