Verbesserung der Programmierbarkeit und Performance-Portabilität von Manycore-Prozessoren[1]

Michel Steuwer[2]

Abstract: Parallele Prozessoren sind heutzutage, in allen Arten von Rechnersystemen zu finden: von großen Datenzentren bis zu den kleinsten mobilen Geräten. Die Programmierung dieser modernen parallelen Rechnersysteme ist aufwändig und fehleranfällig. Um optimale Performance zu erreichen, muss Software zusätzlich speziell angepasst werden. Dabei muss dieser Optimierungsprozess zurzeit für jede neue Prozessorarchitektur wiederholt werden, d.h. Performance ist nicht portabel.

Diese Dissertation widmet sich diesen zwei zentralen Herausforderung der parallelen Programmierung. Das entwickelte und implementierte SkelCL Programmiermodel verbessert die *Programmierbarkeit* moderner paralleler Prozessoren mithilfe wiederkehrender paralleler Muster (sog. algorithmische Skelette). In der Dissertation wurde zusätzlich eine neuartige Technik zur Codegenerierung entworfen, basierend auf formell definierten Transformationsregeln, die *Performance-Portabilität* ermöglicht. Ausgehend von einem nachweislich korrekten und portablen Programm wird automatisch hardware-spezifischer und hoch-effizienter paralleler Code generiert.

1 Einführung

Die Architekturen von Rechnersystemen haben sich in den vergangenen 20 Jahren einem drastischen Wandel unterzogen, um den gestiegenen Anforderungen an Performance und Energieeffizienz Rechnung zu tragen. Prozessoren mit einer Vielzahl von *Kernen* sind, in der Form von Multicore-CPUs und Manycore-GPUs (Graphics Processing Units), nicht mehr nur in Hochleistungsrechnern zu finden, sondern auch in herkömmlichen PCs und zunehmend sogar in mobilen Geräten. Die Programmierung dieser hoch-parallelen Systeme stellt eine erhebliche Herausforderung der praktischen Informatik dar.

Die zurzeit am Weitesten verbreiteten Programmieransätze, wie softwareseitiges Multithreading für Multicore-CPUs oder OpenCL und CUDA für Manycore-GPUs, bieten ein niedriges Programmiernieveau, welches die Softwareentwicklung kompliziert und aufwändig macht. Um ein funktional korrektes Programm zu entwickeln, müssen Programmierer Probleme wie Deadlocks und Race Conditions beachten und eigenständig lösen oder vermeiden. Die Entwicklung hoch performanter Software stellt eine sogar noch größere Herausforderung dar, weil diese eine genaue Kenntnis der Prozessorarchitektur voraussetzt. Dabei sind Optimierungen um hohe Performance zu erreichen häufig spezifisch für eine spezielle Prozessorarchitektur. Um die bestmögliche Performance zu erreichen, sind Programmierer daher gezwungen, den aufwändigen Optimierungsprozess für jede neue Prozessorarchitektur zu wiederholen.

[1] Englischer Titel der Dissertation: *"Improving Programmability and Performance Portability on Many-Core Processors"* [St15a]

[2] The University of Edinburgh, Scotland, UK, michel.steuwer@ed.ac.uk

Diese Dissertation widmet sich diesen zwei zentralen Herausforderungen der parallelen Programmierung: erstens, der vereinfachten *Programmierbarkeit* moderner paralleler Rechnersysteme und zweitens, der *Performance-Portabilität* von Software auf unterschiedlichen Prozessorachitekturen.

In Abschnitt 3 stellen wir das neue SkelCL Programmiermodel vor, welches sich der Herausforderung der *Programmierbarkeit* annimmt. SkelCL vereinfacht die Programmierung von Systemen mit mehreren Manycore-GPUs mithilfe von drei zentralen Features: erstens, die Speicherverwaltung wird durch speziell implementierte *Datencontainer* vereinfacht; zweitens, Berechnungen werden mithilfe wiederkehrender Muster der Parallelität (sog. *algorithmische Skelette*) strukturiert ausgedrückt; drittens, sog. *Distributions* beschreiben deklarativ die Aufteilung der Datencontainer auf die Speicherbereiche mehrerer GPUs. Das SkelCL Programmiermodel ist als eine Software-Bibliothek, kombiniert mit einem eigenen Compiler, implementiert. Der Compiler ermöglicht eine weitgehende Integration mit der Programmiersprache C++, die besonders bei High-Performance Anwendungen verbeitet ist. Unsere Evaluation zeigt, dass SkelCL den Aufwand der GPU Programmierung drastisch reduziert und eine vergleichbare Performance mit manuell optimierten Implementierungen erreicht.

In Abschnitt 4 präsentieren wir einen neuartigen Ansatz zur Codegenerierung, der die Herausforderung der *Performance-Portabilität* in Angriff nimmt. *Parallele high-level Muster* werden zur algorithmischen Beschreibung von Berechnungen genutzt. Neuartige *parallele low-level Muster* beschreiben das OpenCL Programmiermodel in einer strukturierten Art und Weise. Formal definierte semantik-erhaltende *Transformationsregeln* ermöglichen das Umwandeln von abstrakten high-level Programmen in konkrete, OpenCL-spezifische Programme, welche unmittelbar und mit hoher Performance auf GPUs ausgeführt werden können. Die Transformationsregeln drücken dabei sowohl algorithmische Möglichkeiten bei der Implementierung, als auch hardware-spezifische Optimierungen aus. Da häufig mehrere alternative Transformationsregeln angewendet werden können, ergibt sich ein Suchraum, in dem wir mithilfe einer einfachen automatischen Suchstrategie nach der besten OpenCL Implementierung für eine spezifische GPU suchen können. Wir zeigen formell die Korrektheit der semantik-erhaltenden Transformationsregeln und demonstrieren in einer praktischen Evaluation, dass unser Ansatz, ausgehend von portablen abstrakten Programmen, hardware-spezifischen und hoch effizienten OpenCL Code generiert und damit Performance-Portabilität erreicht.

In nächsten Abschnitt beschreiben wir zunächst nötiges Hintergrundwissen zur Programmierung von Manycore-Prozessoren, welches unsere Forschung motiviert.

2 Hintergrund: Manycore-Prozessoren und Ihre Programmierung

Traditionell wurde die Performance von Mikroprozessoren hauptsächlich durch die Erhöhung der Taktrate sowie die Optimierung des Ausführungsablaufens und von Cache-Speichern verbessert. Diese Entwicklung hat sich vor ungefähr 10 Jahren grundsätzlich geändert, wie in Abbildung 1 zu sehen ist. Während die Anzahl der Transistoren weiterhin gemäß dem Mooresches Gesetz zunimmt, erreichten Taktrate und Energieverbrauch

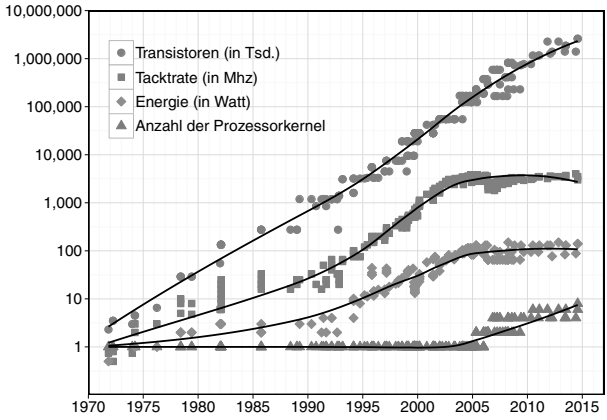

Abbildung 1: Entwicklung von Intel CPUs von 1970 bis 2015. Während die Anzahl von Transistoren weiter exponentiell zunimmt, erreichten Taktrate und Energieverbrauch 2005 ein Plateau. Um die Performance weiter zu steigern, entstanden Multicore-Prozessoren. Abbildung inspiriert von [Su05].

um 2005 ein Plateau. Um die Performance von Prozessoren trotzdem weiter zu steigern, wurden *Multicore-Prozessoren* entwickelt, welche aus mehreren unabhängig operierenden *Kernen* bestehen. Diese Kerne werden zumeist mithilfe von *softwareseitigem Multithreading* explizit programmiert. Diese vorherrschende Art der Programmierung hat sich leider als extrem aufwändig und fehleranfällig erwiesen.

Als eine energieeffiziente Alternative zu Multicore-CPUs haben sich Manycore-GPUs hervorgetan. Ursprünglich zur beschleunigten Berechnung von 3D Grafikszenen entwickelt, ermöglichen moderne GPUs die Ausführung beinahe beliebiger Berechnungen. GPU Prozessorarchitekturen sind für einen hohen Durchsatz von Berechnungen optimiert und in der Lage tausende von Threads zeitgleich in Hardware auszuführen. Dadurch sind GPUs besonders für datenparallele Anwendungen geeignet.

Zur Programmierung von GPUs werden spezifische Programmieransätze verwendet, wobei CUDA und OpenCL die beiden am weitesten Verbreiteten sind. Der Programmierer schreibt spezielle Funktionen, die *Kernel* genannt werden, welche parallel von der GPU ausgeführt werden. Innerhalb des Kernels wird i.d.R. eine eindeutige Thread-Identifikationsnummer verwendet um Berechnungen auf Threads aufzuteilen. Neben den gleichen Problemen des softwareseitigem Multithreadings erfordert die GPU Programmierung zusätzlich das Schreiben von Verwaltungscode der die Ausführung des Kernels auf der GPU organisiert und explizit Daten zwischen der CPU und GPU austauscht. Das Entwickeln von optimierten GPU Anwendungen ist selbst für erfahrene Softwareentwickler eine schwierige Herausforderung, da es genaue Kenntnis der Funktionsweise von GPUs voraussetzt. Darüber hinaus sind viele Optimierungen spezifisch für eine spezielle GPU Architektur, wodurch Performance nicht portabel zwischen unterschiedlichen GPUs oder gar Multicore-CPUs ist. Dadurch müssen zurzeit Optimierungen aufwendig manuell angepasst werden, um die bestmögliche Performance zu erreichen.

3 SkelCL – Ein neuer Ansatz zur GPU-Programmierung

SkelCL ist ein Programmiermodel welches ein hohes Abstraktionsniveau bei der Programmierung von multi-GPU Systemen erlaubt [SKG11]. Durch das Vermeiden von Problemen der Programmierung auf niedrigem Abstraktionsniveau wird die GPU Programmierung signifikant vereinfacht. Im Folgenden stellen wir zunächst das Programmiermodel vor bevor userer Implementierung und Integration des Modells in C++ beschrieben werden. Schließlich evaluieren wir die verbesserte Programmierbarkeit und Performance, die durch SkelCL erreicht werden.

3.1 Das SkelCL Programmiermodel

Das SkelCL Programmiermodel bietet drei zentrale Features, die hier vorgestellt werden.

Parallele Datencontainer In OpenCL müssen Daten explizit zwischen dem Hauptspeicher und dem Speicher der GPU ausgetauscht werden. Die somit notwendige manuelle Programmierung der GPU Speicherverwaltung und des Datenaustausches ist aufwendig und fehleranfällig. SkelCL bietet alternativ dazu eigene parallele Datencontainer, welche auf der CPU und GPU zur Verfügung stehen. Daten werden transparent und automatisch zwischen den Speichern der CPU und GPU transferiert. SkelCL implementiert dazu ein träges (eng. *lazy*) Kopierverfahren, welches Daten nur bei Bedarf überträgt und dadurch unnötige Datentransfers zur Laufzeit vermeidet.

Algorithmische Skelette Um das Abstraktionsniveau zu erhöhen, stellt SkelCL vorgefertigte Muster der parallelen Programmierung, sogenannte *algorithmische Skelette* [Co91], bereit. Dabei spezifiziert ein algorithmisches Skelett nur das Gerüst der parallelen Ausführung von benutzerdefiniertem Code. Formell ist ein algorithmisches Skelett eine Funktion höherer Ordnung, welche vom Programmierer für seine spezielle Anwendung angepasst (instanziiert) wird. Dabei bietet SkelCL effiziente GPU-Implementierungen der vier klassischen datenparallelen Skelette *map*, *zip*, *reduce* und *scan* und führt darüber hinaus zwei neue algorithmische Skelette ein. Das *allpairs* Skelett stellt eine Verallgemeinerung der Matrixmultiplikation dar. Das *stencil* Skelett erleichtert die Programmierung von Stencil-Anwendungen welche häufig in der Bildverarbeitung und bei der Lösung partieller Differentialgleichungen auftreten.

Distributions Bei der Verwendung von mehreren GPUs müssen häufig Daten zwischen den einzelnen GPUs ausgetauscht werden. Die Implementierung eines effizienten Datenaustausches mit OpenCL ist aufwändig und fehleranfällig, da die Aktionen von CPU und mehreren GPUs aufeinander abgestimmt werden müssen. SkelCL bietet sogenannte *Distributions* an, um die Verteilung von Daten auf mehrere GPUs deklarativ zu beschreiben. Während in OpenCL der Datentransfer aus vielen einzelnen Schritten besteht, wird in SkelCL ein Datentransfer indirekt durch das Ändern der Distribution eines Datencontainers ausgelöst. SkelCL bietet drei allgemeine Distributions: *single* – um die Daten auf einer einzelnen GPU zu halten, *block* – um die Daten in Blöcken auf mehrere GPUs aufzuteilen, und *copy* – um die Daten auf alle GPUs zu kopieren. Für das stencil Skelett existiert eine spezielle *overlap* Distribution, welche die Daten in überlappenden Blöcken auf die GPUs verteilt.

3.2 Implementierung von SkelCL in der Programmiersprache C++

Die Implementierung des SkelCL Programmiermodels besteht aus einer Software-Bibliothek, kombiniert mit einem selbst implementierten Compiler, welcher eine weitgehende Integration mit der Programmiersprache C++ erlaubt.

In Listing 1 ist als Beispiel die Berechnung des klassischen Skalarprodukts gezeigt, welches sich als die Summe zweier paarweise multiplizierten Vektoren berechnet: $\sum_i a_i * b_i$. In SkelCL kann dies durch die zwei algorithmischen Skelette *zip* und *reduce* dargestellt werden. Nach der Initialisierung von SkelCL in Zeile 4 werden in Zeilen 6 und 7 die zwei algorithmischen Skelette erzeugt. Das `zip` Skelett wendet die benutzerdefinierte Funktion paarweise auf die Elemente zweier Vektoren an. Im Beispiel wird das Skelett mit der Multiplikation instanziiert, so dass zwei Vektoren paarweise multipliziert werden. Das `reduce` Skelett wird mit der Addition instanziiert, um alle Elemente eines Vektors aufzusummieren. Die angepassten Skelette werden in Zeile 10 auf die zwei Vektoren `A` und `B` angewendet, welche zuvor in Zeile 9 erzeugt wurden. Neben dem eindimensionalen `Vector` Datencontainer, bietet SkelCL auch einen zweidimensionalen `Matrix` Container an. Das gezeigte Programm kann auf einer oder auch auf mehreren GPUs ausgeführt werden. Wird für die Datencontainer keine explizite Distribution angegeben, so wird als Standardverteilung *block* ausgewählt.

SkelCL ermöglicht die Verwendung moderner C++ Features, wie z.B. Lambda-Ausdrücke welche in Zeilen 6 und 7 zu sehen sind und die Anpassung der algorithmischen Skelette erheblich vereinfachen. In OpenCL wird jedoch der Quellcode des Kernels als Strings erwartet, welche zur Laufzeit in ausführbaren Code übersetzt werden. Um dieses Problem zu lösen, übersetzt unser selbst implementierter auf der LLVM-Infrastruktur basierender Compiler den gezeigten Quellcode in Listing 1 in einer Form, in der die Multiplikations- und Additionsfunktionen als Strings dargestellt werden. Diese werden dann von der SkelCL Bibliothek zur Laufzeit mit einer vorgefertigten Implementierung der algorithmischen Skelette zu OpenCL Kerneln kombiniert und schlussendlich ausgeführt.

```
1  #include <SkelCL/SkelCL.h>
2
3  float dotProduct(const float* a, const float* b, int n) {
4    skelcl::init();
5
6    auto mult = zip([](float x, float y) { return x*y; });
7    auto sum  = reduce([](float x, float y) { return x+y; }, 0);
8
9    Vector<float> A(a, a+n); Vector<float> B(b, b+n);
10   Vector<float> C = sum( mult(A, B) );
11   return C.front(); }
```

Listing 1: Berechnung des Skalarprodukts zweier Vektoren in SkelCL. Die Berechnung wird transparent auf einer oder mehrerer GPUs ausgeführt.

3.3 Evaluation der Programmierbarkeit und Performance von SkelCL

Zur Evaluation der Programmierbarkeit haben wir als einfache Metrik die Anzahl der Quellcodezeilen gewählt. Abbildung 2 zeigt für fünf Anwendung aus unterschiedlichen Anwendungsbereichen, dass SkelCL Programme in der Regel erheblich kürzer sind als entsprechende OpenCL Programme. Dies gilt sowohl für die Ergebnisse einfacher Benchmark Anwendungen, wie die Berechnung der Mandelbrot Menge, als auch für komplexe Anwendungen aus der realen Welt z.B. die LM OSEM Anwendung aus der medizinischen Bildgebung. Auffallend ist, dass vor allem der CPU Code Anteil erheblich verringert wird. In OpenCL entspricht dies dem Verwaltungscode, welcher in SkelCL fast vollständig entfällt. Die gezeigten SkelCL Programme können ohne Änderungen auf mehreren statt einer GPU ausgeführt werden, während in OpenCL dafür zusätzliche Codezeilen nötig sind.

Abbildung 3 zeigt die erreichte Performance von SkelCL im Vergleich zu OpenCL für sechs Anwendungen. Bis auf die Skalarprodukt Anwendung, erreichen die SkelCL Programme Performance die vergleichbar mit manuell optimierten OpenCL Programmen ist. Dabei sind die SkelCL Programme jedoch (wie in Abbildung 2 zu sehen) deutlich kürzer und durch das höhere Abstraktionsniveau viel einfacher zu programmieren und zu verstehen. In der OpenCL Implementierung des Skalarprodukts wird ein einzelner Kernel gestartet, während in SkelCL zwei getrennte Kernel ausgeführt werden, was den Performance Unterschied erklärt. Im folgenden Abschnitt 4 werden wir eine neuartige Technik zur Generierung von OpenCL Code vorstellen, welche auch für das Skalarprodukt effizienten Code generiert.

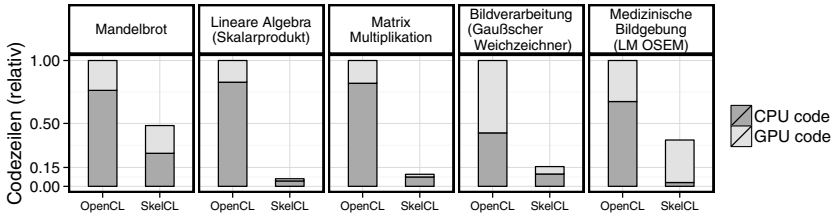

Abbildung 2: Programmieraufwand von SkelCL im Vergleich zu OpenCL. Alle untersuchten Programme sind mehr als 50% kürzer in SkelCL als in OpenCL.

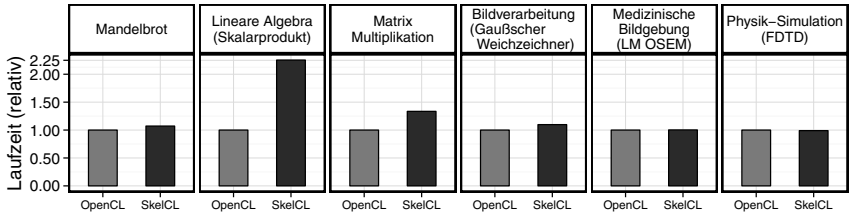

Abbildung 3: Performance von SkelCL im Vergleich zu OpenCL. Für alle untersuchten Programme (bis auf das Skalarprodukt) erreicht SkelCL vergleichbare Performance wie OpenCL. In Abschnitt 4 wird eine Technik vorgestellt um die Performance der Skalarprodukt Anwendung zu verbessern.

4 Ein neuer Ansatz zur Performance portablen Codegenerierung

Die zurzeit vorherrschenden parallelen Programmieransätze bieten keine Performance-Portabilität, da hardware-spezifische Optimierungen manuell angewendet werden, um optimale Performance zu erreichen. In diesem Abschnitt stellen wir einen neuartigen Ansatz vor, der sich dieser Herausforderung der Performance-Portabilität annimmt [St15b]. Dazu führen wir zunächst zwei Arten von parallelen Mustern ein. High-level Muster beschreiben Programme algorithmisch. Low-level Muster drücken OpenCL-spezifische Hardware-Eigenschaften aus. Anschließend, definieren wir Regeln zur automatischen Transformation eines high-level Programms in spezifische low-level Programme. Aus diesen low-level Programmen werden schlussendlich OpenCL Kernel generiert.

4.1 Parallele high-level und low-level Muster

Unser Ansatz basiert auf parallelen Mustern, welche wir als Funktionen modellieren. Tabelle 1 zeigt die parallelen high-level Muster. In der Dissertation werden diese formell definiert und ein *dependent type system* verwendet, um die Länge von Arrays zu erfassen.

Neben den high-level Mustern definieren wir neuartige low-level Muster, welche dem OpenCL Programmiermodell nachempfunden sind. Diese sind in Tabelle 2 zu sehen. Jedes Muster drückt dabei eine spezielle Hardware-Eigenschaft aus, wie z.B. die unterschiedlichen Ebenen der Parallelität, Vektorisierung oder die Speicherhierarchie. Für jedes low-level Muster wird direkt ein enstprechendes Stück OpenCL Code generiert.

High-level Muster	Beschreibung
map	Wendet eine gegebene Funktion auf alle Elemente des Eingabearrays an.
reduce	Berechnet mithilfe eines gegebenen Operators die Reduktion des Eingabearrays.
zip	Erzeugt ein Array durch paarweises kombinieren von zwei Eingabearrays.
split	Teilt das Eingabearray in Blöcke fester Länge auf.
join	Gegenteil von *split*: fügt Blöcke wieder zu einem flachen Array zusammen.

Tabelle 1: Parallele high-level Muster zur algorithmischen Beschreibung von Programmen.

Low-level Muster	Beschreibung
map-global, *map-workgroup*, *map-local*	Jeder **globale Thread**, jede **Gruppe von Threads** oder jeder **lokale Thread** führt parallel eine Berechnung durch.
map-seq, *reduce-seq*	Führe die high-level Muster *map* und *reduce* **sequenziell** aus.
asVector, *asScalar*, *vectorize*	**Vektorisierung** von Daten und Berechnungen.
toLocal, *toGlobal*	Zur Verwendung des schnellen aber kleinen **lokalen Speichers**.
reorder-stride	Speicherzugriffe umordnen, so dass diese von der Hardware zusammengeffast werden (**memory coalescing**).

Tabelle 2: Parallele low-level Muster sind dem OpenCL Programmiermodel nachempfunden.

4.2 Transformationsregeln zur automatischen Optimierung

Der Vorteil einer formalen Definition von parallelen Mustern liegt in der Möglichkeit Programme automatisiert zu transformieren. Dazu definieren wir in der Dissertation eine Reihe von *Transformationsregeln*. Eine solche Regel beschreibt eine syntaktische Transformation des Programms, für welche bewiesen wird, dass sie die Semantik des Programms erhält. In der Dissertation sind insgesamt 29 solche Transformationsregeln definiert und deren Korrektheit formell bewiesen. Dabei unterscheiden wir zwischen zwei Arten von Regeln: erstens, Regeln welche Ausdrücke von high-level Mustern in andere high-level Muster umschreiben und dadurch algorithmische Implementierungsentscheidungen beschreiben, und zweitens, Regeln welche Ausdrücke von high-level Muster auf low-level Muster abbilden und dadurch hardware-spezifische Optimierungen ausdrücken.

Ein Beispiel einer rein *algorithmischen Transformationsregel* ist die *split-join Zerlegung*:

$$map\ f\ xs \quad \rightarrow \quad (join \circ map\ (map\ f) \circ split\ n)\ xs$$

Diese Regel beschreibt, dass es möglich ist, um eine Funktion f auf jedes Element des Arrays *xs* anzuwenden, zuerst das Array in Blöcke der Größe n aufzuteilen ($split\ n$) und dann durch das äußere *map* für jeden Block die ($map\ f$) Funktion anzuwenden bevor die Blöcke mit dem *join* Muster wieder verbunden werden. Das innere *map* wendet dabei die Funktion f auf jedes Element eines einzelnen Blocks an. Durch die Anwendung weiterer Transformationsregeln könnte das äußere *map* parallelisiert werden, das innere *map* jedoch sequenziell ausgeführt werden. In dieser Situation würde n dann den Grad der verwendeten Parallelität kontrollieren. Wir führen insgesamt 16 weitere algorithmische Regeln ein, unter anderem um verschiedene Möglichkeiten der Parallelisierung einer Reduktion zu beschreiben oder mehrere Muster zu einem einzigen Muster zu verschmelzen.

Ein Beispiel einer *hardware-spezifischen Transformationsregel* ist die Regel zur Verwendung des schnellen aber kleinen lokalen GPU Speichers:

$$map\text{-}local\ f\ xs \quad \rightarrow \quad toLocal\ (map\text{-}local\ f)\ xs$$

Diese Regel spezifiziert, dass das *toLocal* Muster nur zusammen mit *map-local* verwendet werden kann, welches den lokalen Threads in OpenCL entspricht die in einer Gruppe organisiert werden. In OpenCL ist nur innerhalb solcher Gruppen die Verwendung des lokalen Speichers erlaubt, was diese Regel formal sicherstellt. In der Dissertation sind 11 weitere hardware-spezifische Regeln beschrieben und bewiesen, welche unter anderem Vektorisierung sowie das Ausnutzen der Parallelitäts- und Speicherhierarchien von GPUs ermöglichen.

Da die Korrektheit der Transformationsregeln formal bewiesen ist können diese automatisiert angewendet werden, ohne die Semantik des ursprünglichen Programms zu verändern. Dies steht im krassen Gegensatz zu traditionellen Compilern, die optimierende Transformationen erst nach aufwändiger Analyse anwenden können. Durch die unterschiedliche Anwendung der Transformationsregeln ergibt sich ein Suchraum von unterschiedlich optimierten OpenCL Implementierungen des ursprünglichen Programms. Eine einfache Suchstrategie basierend auf dem Monte-Carlo-Algorithmus ist in der Dissertation beschrieben.

4.3 Evaluation der erreichten Performance-Portabilität

```
1 scal a xs = map (λ x. a∗x) xs
2 asum xs = reduce + 0 (map abs xs)
3 dot xs ys = reduce + 0 (map ∗ (zip xs ys))
4 gemv mat xs ys a b = map + (zip (map (scal a ∘ dot xs) mat) (scal b ys))
```

Listing 2: Lineare Algebra Anwendungen ausgedrückt durch parallele high-level Muster.

Listing 2 zeigt wie vier Anwendungen aus dem Bereich der linearen Algebra mithilfe der parallelen high-level Muster ausgedrückt werden. Ausgehend von diesen Programmen werden Transformationsregeln angewendet, um unterschiedliche algorithmische und hardware-spezifische Optimierungen vorzunehmen. Aus den umgeschriebenen low-level Programmen wird anschließend OpenCL Code generiert.

Abbildung 4 zeigt die Performance des automatisch generierten OpenCL Codes. Dabei vergleichen wir die Performance mit den hochoptimierten Software-Bibliotheken, die der jeweilige Hersteller speziell für seine Prozessoren zur Verfügung stellt. Für alle untersuchen Anwendungen erreicht – oder übertrifft – unser Ansatz die Performance der Hardware Hersteller auf ihrer eigenen Hardware: einer GPU von Nvidia, einer GPU von AMD und einer CPU von Intel. Insbesondere sehen wir, dass für das Skalarprodukt (**dot**), welche in SkelCL ursprünglich deutlich schlechtere Performance zeigte, die hohe Performance der Hersteller erreicht wird. Eine Transformationsregel zur Verschmelzung der *map* und *reduce* Muster ermöglicht die Generierung eines einzelnen effizienten OpenCL Kernels.

Die gezeigte Performance wurde durch eine vollständig automatisierte Anwendung der Transformationsregeln auf die portablen high-level Programme aus Listing 2 erreicht. Damit erreicht unser Ansatz zur Generierung von effizientem Code auf Basis von semantikerhaltenden Transformationsregeln die erwünschte Performance-Portabilität.

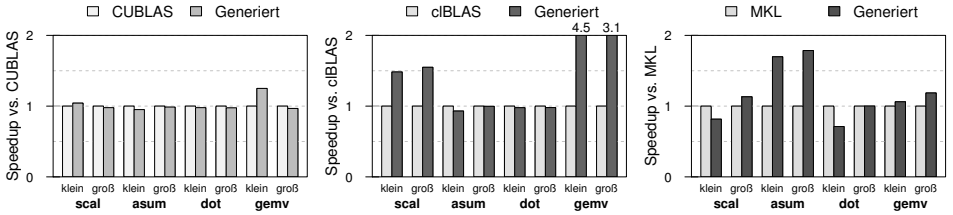

Abbildung 4: Performance des automatisch generierten OpenCL Codes im Vergleich zu den hardware-spezifischen Bibliotheken CUBLAS von Nvidia, clBLAS von AMD und MKL von Intel. In fast allen Fällen zeigt der generiert Code die gleiche oder bessere Performance. Ausgehend vom selben high-level Programm wurde für jeden Prozessor unterschiedlich optimierter Code generiert.

5 Fazit

In dieser Dissertation wurden zwei zentrale Herausforderungen der parallelen Programmierung behandelt. Unser SkelCL Programmiermodel demonstriert, dass es möglich ist die Programmierbarkeit von GPU Anwendungen signifikant zu verbessern ohne Einbußen in der Performance hinzunehmen zu müssen. Ein neuartiger Regel-basierter Ansatz erlaubt die Generierung von spezialisiertem und hoch effizientem OpenCL Code ausgehend von einem portablen high-level Programm. Dieser Ansatz demonstriert erstmals, dass das Ziel der Performance-Portabilität in der Praxis zu erreichen ist.

Durch eine Zusammenführung der beiden Ansätze liegt nahe, dass in der Zukunft eine einfachere Programmierung gekoppelt mit der Generierung von effizientem Code auf einer Vielzahl von parallelen Prozessoren möglich ist. Den Schlüssel dazu stellen die parallelen Muster dar, welche sowohl die Programmierung vereinfachen als auch eine neuartige formelle Darstellung von Optimierungen als Transformationsregeln ermöglichen.

Literaturverzeichnis

[Co91] Cole, Murray: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, 1991.

[SKG11] Steuwer, Michel; Kegel, Philipp; Gorlatch, Sergei: SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW). IEEE, 2011.

[St15a] Steuwer, Michel: Improving Programmability and Performance Portability on Many-Core Processors. University of Münster, 2015.

[St15b] Steuwer, Michel; Fensch, Christian; Lindley, Sam; Dubach, Christophe: Generating Performance Portable Code using Rewrite Rules: From High-Level Functional Patterns to High-Performance OpenCL Code. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. ICFP. ACM, 2015.

[Su05] Sutter, Herb: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 30(3), 2005.

Michel Steuwer wurde am 21. Mai 1985 in Duisburg geboren. Nach dem Besuch der Grund- und Gesamtschule studierte er von 2005 bis 2010 Informatik mit Nebenfach Mathematik an der Westfälische Wilhelms-Universität Münster. Das Studium schloss er mit einem Diplom ab. Anschließend promovierte er unter der Leitung von Prof. Sergei Gorlatch in der Arbeitsgruppe Parallele und Verteile Systeme in Münster. Während seiner Promotion etablierte er eine erfolgreiche Kollaboration mit der University of Edinburgh, welche er mehrfach besuchte. Er verteidigte seine Disseration erfolgreich am 26. Juni 2015 welche mit der Note *summa cum laude* ausgezeichnet wurde.