Technical University of Munich
Ludwig-Maximilians-University Munich

Proceedings of the 3rd Seminar on HPC Systems: Current Trends and Developments

Winter Semester 2017/2018

January 30-31
Frauenchiemsee, Germany

Eds.: Karl Fürlinger, Josef Weidendorfer, Carsten Trinitis, Dai Yang
TUM University Library

Proceedings of the 3rd Seminar on HPC Systems: Current Trends and Developments
Winter Semester 2017/2018

Editors:
Karl Fürlinger, Ludwig-Maximilians-University Munich
Carsten Trinitis, Technical University of Munich
Josef Weidendorfer, Technical University of Munich
Dai Yang, Technical University of Munich

Cataloging-in-Publication Data

Seminar High Performance Computing: Current Trends and Developments
Proceedings of the 3rd Seminar on HPC Systems: Current Trends and Developments
WS2017/2018
30-31.01.2018 Frauenchiemsee & Munich, Germany

DOI: 10.14459/2018md1430700
Published by TUM University Library

Preface

The hardware and software requirements for High Performance Computing (HPC) Systems is demanding and versatile. Due to its large scale, even the smallest improvement in efficiency and performance may have great impact on these systems, which eventually leads to higher cost efficiency. Therefore, cutting-edge technologies are often deployed rapidly on HPC-Systems. This unique character challenges users and programmers of HPC-systems: how can these new functionalities be efficiently used?

In the joint seminar “High Performance Computing: Current Trends and Developments” students worked on recent publications and research in HPC technologies. New developments and trends in both hardware and software have been discovered and discussed. The seminar features a technical committee of active HPC researchers from Technical University of Munich, Ludwig-Maximilians-University Munich, and various industry partners. Eight students participated in the seminar and submitted a peer-reviewed seminar paper. An academic conference, in which the students present their work in a 30-minute talk, was held at Abtei Frauenwörth on the island of Frauenchiemsee in January 2018.

These are the first proceedings of our seminar series. By publishing the seminar work, we intend to spread the knowledge on recent developments in HPC-Systems and make HPC related education and research more interesting to a broader audience.

Karl Fürlinger, Carsten Trinitis, Josef Weidendorfer and Dai Yang
08.02.2018 in Munich, Germany

 DOI: 10.14459/2018md1430700

Seminar Lecturers

Professors

Martin Schulz, Technical University of Munich
Dieter Kranzlmüller, Ludwig-Maximilians-University Munich

Seminar Chairs

Karl Fürlinger, Ludwig-Maximilians-University Munich
Josef Weidendorfer, Technical University of Munich
Carsten Trinitis, Technical University of Munich

Advisors

David Büttner, Siemens AG
Roger Kowalewski, Ludwig-Maximilians-University Munich
Matthias Maitert, Intel & Leibniz Super Computing Center
Dai Yang, Technical University of Munich

Homepage

http://www.nm.ifi.lmu.de/teaching/Seminare/2017ws/HPCTrends/
https://www.lrr.in.tum.de/lehre/wintersemester-1718/seminare/hochleistungsrechner-aktuelle-trends-und-entwicklungen/

 DOI: 10.14459/2018md1430700

Master-Seminar: Hochleistungsrechner - Aktuelle Trends und Entwicklungen
Aktuelle GPU-Generationen (NVidia Volta, AMD Vega)

Stephan Breimair
Technische Universität München

23.01.2017

Abstract

GPGPU - General Purpose Computation on Graphics Processing Unit, ist eine Entwicklung von Graphical Processing Units (GPUs) und stellt den aktuellen Trend bei NVidia und AMD GPUs dar.

Deshalb wird in dieser Arbeit gezeigt, dass sich GPUs im Laufe der Zeit sehr stark differenziert haben. Während auf technischer Seite die Anzahl der Transistoren stark zugenommen hat, werden auf der Software-Seite mit neueren GPU-Generationen immer neuere und umfangreichere Programmierschnittstellen unterstützt. Damit wandelten sich einfache Grafikbeschleuniger zu multifunktionalen GPGPUs. Die neuen Architekturen NVidia Volta und AMD Vega folgen diesem Trend und nutzen beide aktuelle Technologien, wie schnellen Speicher, und bieten dadurch beide erhöhte Anwendungsleistung. Bei der Programmierung für heutige GPUs wird in solche für herkömmliche Grafikanwendungen und allgemeine Anwendungen differenziert. Die Performance-Analyse untergliedert sich in High- und Low-Level-Analyse. Entscheidend für das GPU-Profiling ist, ob Berechnungskosten Speicherkosten verdecken beziehungsweise umgekehrt oder mangelnde Parallelisierung vorliegt.

Insgesamt konnte mit dieser Arbeit dargelegt werden, dass die aktuellen GPU-Architekturen eine weitere Fortführung der Spezialisierung von GPUs und einen anhaltenden Trend hin zu GPGPUs darstellen. Allgemeine und wissenschaftliche Anwendungen sind heute also ein wesentliches Merkmal von GPUs geworden.

1 Einleitung

Grafikbeschleuniger existieren bereits seit Mitte der 1980er Jahre, wobei der Begriff „GPU“, im Sinne der hier beschriebenen Graphical Processing Unit (GPU) [1], 1999 von NVidia mit deren Geforce-256-Serie eingeführt wurde.

Im strengen Sinne sind damit Prozessoren gemeint, die die Berechnung von Grafiken übernehmen und diese in der Regel an ein optisches Ausgabegerät übergeben. Der Aufgabenbereich hat sich seit der Einführung von GPUs aber deutlich erweitert, denn spätestens seit 2008 mit dem Erscheinen von NVidias „GeForce 8“-Serie ist die Programmierung solcher GPUs bei NVidia über CUDA (Compute Unified Device Architecture) möglich.

Da die Bedeutung von GPUs in den verschiedensten Anwendungsgebieten, wie zum Beispiel im Automobilsektor, zunehmend an Bedeutung gewinnen, untersucht diese Arbeit aktuelle GPU-Generationen, gibt aber auch einen Rückblick, der diese aktuelle Generation mit vorhergehenden vergleicht. Im Fokus stehen dabei die aktuellen Architekturen, deren technische Details beziehungsweise die Programmierung sowie Performance-Analyse von und für GPUs.

Im Abschnitt 2 wird dementsprechend diskutiert, wie sich GPUs seit ihrer Einführung in den Markt der elektronischen Rechenwerke, entwickelt und spezialisiert haben. Abschnitt 3 gibt einen tieferen Überblick über die Architekturen aktueller GPUs von NVidia und AMD, technische Details und die Leistungsfähigkeit. Anschließend werden in Abschnitt 4 Möglichkeiten der

 DOI: 10.14459/2018md1430700

Programmierung für GPUs vorgestellt. Da es jedoch auch im GPU-Bereich nötig beziehungsweise sinnvoll ist, eine Performance-Analyse durchzuführen, soll dies im Abschnitt 5 genauer auf verschiedenen Leveln diskutiert werden.
Insgesamt liegt der Fokus dieser Arbeit aber auf aktuellen Architekturen für technische und wissenschaftliche Berechnungen. Herkömmliche Grafikprogrammierung, wie zum Beispiel mit C++ und DirectX oder OpenGL, sollen deshalb nur am Rande Erwähnung finden.

2 Entwicklung der GPUs

Dieser Abschnitt gibt einen Überblick über die Entwicklung der GPUs seit ihrer Einführung.

2.1 Rückblick [13]

Die ersten Jahre der GPUs sind in Tabelle 1 abgebildet. Bei AMD folgten danach bis heute (Stand 12.2017) noch die HD Radeon-R200-Serie, die Radeon-R300-Serie, die Radeon-400 und Radeon-500-Serie. Bei diesen ist die erstmalige Verwendung von HBM-Speicher mit dem Fiji-Grafikprozessor in der Radeon R9 Fury X (gehört zur Radeon-R300-Serie) hervorzuheben. Außerdem nutzten die Polaris-Grafikchips (Radeon-400-Serie) bereits die GCN 4.0 Architektur, wodruch die Effizienz gesteigert werden konnte. Allerdings war beziehunsweise ist diese Generation und auch die nachfolgende Rebrand-Generation (Radeon-500-Serie) nicht in der Lage die Leistung des Konkurrenten NVidia zu erreichen. Die neueste Grafikkarten-Serie (Stand 12.2017) seitens AMD ist die Radeon-Vega-Serie, die die GCN-Architektur bereits in der 5. Ausbaustufe verwendet und DX 12.1, OpenGL 4.5, Open CL 2.0+ sowie Vulkan 1.0 unterstützt sowie 12,5 Milliarden Transistoren besitzt. Für weitere Details zum Vega-Grafikchip siehe Unterunterabschnitt 3.1.2.
Neben AMD und NVidia existierten beziehungsweise existieren noch andere Grafikchip-Hersteller mit teilweise eigenen technischen Neuerungen, die aber hier der Übersichtlichkeit halber keine Erwähnung finden sollen.

2.2 Differenzierung der GPUs

Der größte Ausbau der Maxwell-Architektur (GM200) von NVidia verfügte gegenüber dem Vorgänger (Kepler) aber nicht mehr über erweiterte Double-Precision-Fähigkeiten (FP64), wozu bis zu Kepler separate ALUs verbaut waren. Dies geschah vermutlich aus Platzgründen. [2] Während Double-Precision-Operationen für 3D-Anwendungen wie Spiele nicht benötigt wurden, machte das Fehlen von Double-Precision-Fähigkeiten den Einsatz in den professionelleren Tesla-Karten unmöglich. Deshalb wurde dort mit GK210 eine verbesserte Variante der älteren Kepler-Architektur eingesetzt. Dies kann als Beginn einer echten Differenzierung der GPUs betrachtet werden. Auch die zweite Generation der Maxwell-Architektur (GM204) wies eine zu geringe Double-Precision-Leistung auf, um in den Tesla-Karten zum Einsatz zu kommen, wo also weiterhin die Kepler-Architektur verwendet wurde. Die GeForce-10-Serie für den Desktop- und Workstation-Bereich nutzt die Pascal-Architektur im 16 nm Fertigungsprozess. Die Ausbaustufen reichen von GP108 bis GP100, wobei sich der größte GP100-Chip stark von den restlichen unterscheidet. Durch den kleineren Fertigungsprozess entfielen die Limitierungen wegen zu großer Die-Fläche, wie die zu geringe Double-Precision-Leistung, allerdings entschied sich NVidia wieder, für die professionellen Tesla-Karten einen eigenen, nämlich den sich stark unterscheidenden GP100-Chip, einzusetzen. Dort wurden ein anderer Clusteraufbau, High Performace Computing Fähigkeiten und ein ECC-HBM2-Speicherinterface verwendet. Die noch aktuellere Volta-Architektur ist derzeit (Stand Dezember 2017) nur für die Tesla-Karten verfügbar und erhöht durch ein Schrumpfen des Fertigungsprozesses von 16 nm auf 12 nm und eine Vergrößerung der Die-Fläche die Transistorenzahl auf 21 Milliarden (gegenüber 15 Milliarden bei Pascal). Weitere Details zur aktuellen Volta-Architektur finden sich in Unterunterabschnitt 3.1.1.
Die Separierung in Desktop- bzw. Workstation-Karten und Tesla-Karten umfasst jedoch nicht die ganze vorhandene Differenzierung von GPUs. Mit NVidia Quadro existieren GPU-Karten, die den Desktop- bzw. Workstation-Pendants sehr ähneln und sogar die gleichen Treiber verwenden, allerdings durch eine andere Beschaltung eine andere Chip-ID erhalten, was wiederum den Treiber veranlasst spezielle Funktionen für professionelle Anwendungen im CAD-, Simulations- und Animationsbereich freizuschalten. Von ATI beziehungsweise AMD existieren entsprechende Karten unter den Bezeichnungen FireGL- und FirePro bezie-

 DOI: 10.14459/2018md1430700

Jahr	Produkt / Serie	Transistoren	CUDA-Kerne	Besondere Technologie(n) / Besonderheiten
1990er			-	Entwicklung dedizierter GPUs beginnt
1997	NV Riva 128	3 Millionen	-	Erster 3D-Grafikbeschleuniger
1995-98	ATI Rage	8 Millionen	-	Twin Cache Architektur
2000	NV GeForce 256	17 Millionen	-	Erste, echte GPU; DX7, OpenGL
2000	ATI Radeon-7000	30 Millionen	-	Effizienzsteigerungen (HyperZ)
2000	NV GeForce 2	25 Millionen	-	Erste GPU für Heimgebrauch
2001	NV GeForce 3	57 Millionen	-	Erste programmierbare GPU; DX8, OpenGL; Pixel- u. Vertex-Shader
2002	ATI Radeon-9000	>100 Millionen	-	Einstieg ATIs in High-End-Markt
2003	NV GeForce FX	135 Millionen	-	32-bit floating point (FP) programmierbar
2004	NV GeForce 6	222 Millionen	-	GPGPU-Cg-Programme; DX9; OpenGL; SLI; SM 3.0
2006	NV GeForce 8	681 Millionen	128	Erste Grafik- und Berechnungs GPU; CUDA; DX10; Unified-Shader-Architektur
2007	NV Tesla			GPUs für wissenschaftliche Anwendungen
2008	NV GeForce-200		240	CUDA; OpenCL; DirectCompute
2008	ATI-Radeon-HD-4000		-	GDDR5; Übernahme ATIs durch AMD
2009	NV Fermi / GeForce-400	3 Milliarden	512	Zunehmende GPGPU-Bedeutung; C++ Unterstützung; DX11; ECC-Speicher; 64-bit Addressierung
2010	ATI-Radeon-HD-5000	∼2 Milliarden	-	Terrascale-Architektur; Optimierungen für GPU-Computing
2012	NV Kepler / GeForce-600	∼3,5 Milliarden	1536	GPU-Boost; Graphics Processing Cluster (GPCs)
2012	AMD-Radeon-HD-7000	4,3 Milliarden	-	Neue GCN-Architektur; Vulkan-API

Tabelle 1: Entwicklung der GPUs

hungsweise seit der 4. Generation der GCN Architektur unter dem Namen Radeon Pro.

2.3 Spezialisierung der GPUs

Hinsichtlich Spezialisierung von GPUs, sind die bereits aus den vorhergehenden Abschnitten bekannten Differenzierungen, wie Tesla- und Quadro-Karten, zu nennen.

Allerdings geht die Differenzierung von GPUs inzwischen soweit, dass von Spezialisierung auf einzelne Anwendungsgebiete gesprochen werden muss. Für autonomes Fahren bietet NVidia beispielsweise Drive PX an, ein Fahrzeugcomputer mit künstlicher Intelligenz, der bezüglich Umfang von Einzelprozessoren für einzelne Fahrzeugfunktionen bis zu Kombinationenen mehrerer Grafikprozessoren für autonome Robotertaxis reicht. Laut NVidia [3] kombiniert die Plattform Deep Learning, Sensorfusion und Rundumsicht, was sogar Level-5, also vollständige, Fahrzeugautonomie bieten soll.

Darüber hinaus bietet NVidia mit NVidia DGX-1 eine Plattform für die KI-Forschung und den KI-Einsatz in Unternehmen. Letztendlich kommt aber hierbei ebenfalls eine aktuelle Tesla V100 und spezialisierte Software zum Einsatz. [4]

Zuletzt existiert unter dem Namen NVidia Tegra auch ein komplettes System-on-a-Chip (SoC) für mobile Geräte, aber auch Autos, das neben der Grafikeinheit auch ARM-Kerne, Chipsatz, teilweise ein Modem und weiteres enthält. Dieses SoC trägt bei NVidia den Namen Tegra, das in der aktuellen Generation (Tegra X1) allerdings nur eine Mawell-Grafikeinheit enthält. Hervorzuheben ist jedoch, dass trotz des mobilen Anwen-

 DOI: 10.14459/2018md1430700

dungsfokus CUDA unterstützt wird.

3 Architekturen aktueller GPUs

Nachdem im vorhergehenden Abschnitt die Entwicklung der GPUs dargelegt wurde, sollen im folgenden die technischen Details der aktuellen Architekturen von NVidia und AMD, sowie deren Leistungsfähigkeit, Kosten und Anwendungsgebiete dargestellt werden.

3.1 Technische Details

Bei den aktuellen Architekturen soll bei NVidia der Fokus auf der Volta-Architektur in Form der Tesla V100 GPU, die auf eben dieser Architektur basiert, und bei AMD auf Vega-Chips liegen, der 5. Ausbaustufe der GCN-Architektur.

3.1.1 NVidia Tesla V100 GPU [14]

Die Volta-Architektur wird gegenwärtig nur in Form der Tesla V100 GPU (GV100) umgesetzt, die unter anderem die folgenden Neuerungen und Verbesserungen mitbringt:

- Neue Streaming Multiprozessor (SM) Architektur für Deep Learning
- NVLink 2.0
- HBM2 Speicher mit höherer Effizienz
- Volta Multi-Prozess Dienst (MPS)
- Verbesserte Unified Memory and Address Translation Dienste
- Modi für maximale Performance und Effizienz
- Cooperative Groups and Launch APIs
- Für Volta optimierte Software
- Weitere, wie zum Beispiel unabhängiges Thread-Scheduling

Daneben enthält die Tesla-GPU 21,1 Milliarden Transistoren auf einer Die-Fläche von 815 mm^2 und wird im TSMC 12 nm FFN (FinFET NVidia) Fertigungsprozess hergestellt.

Durch die neue Streaming Multiprozessor (SM) Architektur ist diese 50% energieeffizienter als diejenige in der Pascal-Architektur, bei gleichzeitig deutlich erhöhter FP32- und FP64-Leistung. Mithilfe neuer Tensor-Kerne steigt die Spitzen-TFLOPS-Leistung auf das Zwölffache für das Training und das Sechsfache bei der Inferenz.

NVLink 2.0 ermöglicht eine höhere Bandbreite, mehr Links und eine bessere Skalierbarkeit für Multi-GPU- und CPU-Systeme. Deshalb unterstützt Volta GV100 bis zu sechs NVLinks mit einer Gesamtbandbreite von 300 GB/s.

Das Subsystem für die HBM2-Speicherverwaltung erreicht laut NVidia eine Spitzenleistung von 900 GB/s Speicherbandbreite. Durch die Kombination des neuen Volta-Speichercontrollers und einer neuen Generation Samsung HBM2-Speichers, erreicht Volta insgesamt eine 1,5-fache Speicherbandbreite gegenüber Pascal.

Der Volta Multi-Prozess Dienst (MPS) hardwarebeschleunigt kritische Komponenten des CUDA-MPS-Servers, wodurch eine bessere Leistung, Entkopplung und besseres QoS erreicht wird, wenn mehrere Anwendungen für wissenschaftliche Berechnungen die GPU teilen müssen. Dadurch steigt die erlaubte Anzahl an MPS-Clients von 16 bei Pascal auf nun 48.

Verbesserte Unified Memory- und Address Translation-Dienste erlauben eine genauere Migration der Speicherseiten zu demjenigen Prozessor, der diese am häufigsten nutzt. Dadurch wird erneut die Effizienz beim Zugriff auf Speicherbereiche, die von mehreren Prozessoren verwendet werden, erhöht. Auf der sogenannten IBM-Power-Plattform gestattet es eine entsprechende GPU-seitige Unterstützung derselben auf die Seitentabellen der CPU direkt zuzugreifen.

Die Thermal Design Power (TDP) von Volta ist mit 300 Watt spezifiziert, die jedoch nur im Maximum-Performance-Mode genutzt wird, wenn Anwendungen höchste Leistung und höchsten Datendurchsatz benötigen. Dagegen eignet sich der Maximum-Efficiency-Mode für Datencenter, die die Leistungsaufnahme regulieren und die höchste Leistung pro Watt erzielen wollen.

Mit NVidia CUDA 9 wurden Cooperative Groups (kooperative Gruppen) für miteinander kommunizierende Threads eingeführt. Mithilfe der Cooperative Groups können Entwickler die Granularität bestimmen mit welcher Threads kommunizieren, worduch effizientere, parallele Dekompositionen von Anwendungen erstellt werden können. Alle GPUs seit Kepler unterstützen diese Gruppen, Pascal und Volta aber auch die Launch-APIs, die die Synchronisation zwischen CUDA-Thread-Blöcken ermöglichen.

Der Aufbau einer Volta-GPU besteht aus sechs GPU Processing Clusters (GPCs), die jeweils über sieben Texture Processing Clusters (TPCs) und 14 Strea-

ming Multiprozessoren (SMs) verfügen. Jeder Streaming Multiprozessor hat dabei:

- 64 FP32 Kerne
- 64 INT32 Kerne
- 32 FP64 Kerne
- 8 Tensor Kerne
- Vier Textureinheiten

Insgesamt verfügt jede volle GV100 GPU also über 5376 FP32- und INT32-, 2688 FP64-, 672 Tensor-Kerne und 336 Textureinheiten. Zusätzlich besitzt eine vollständige GV100-GPU acht 512-bit Speichercontroller mit insgesamt 6144 KB L2 Cache.
Die Anzahl der Streaming Multiprozessoren hat sich gegenüber der GP100, die auf der Pascal-Architektur aufbaut, nicht nur von 56 auf 80 erhöht, sondern diese enthalten nun auch erstmalig 8 Tensor-Kerne, die explizit für Deep Learning vorgesehen sind.
In Abbildung 1 ist ein einzelner Streaming Multiprozessor einer Volta GV100 dargestellt.
Wie leicht zu erkennen ist, nehmen die Tensor-Kerne eine nennenswerte Fläche innerhalb des Streaming Multiprozessors ein, was insofern bemerkenswert ist, als dass es sich um die erste GPU-Generation überhaupt handelt, die diese Tensor-Kerne beinhaltet. Von diesen Tensor-Kernen ist jeder in der Lage 64 floating point fused multiply-add (FMA) Operationen pro Takt durchzuführen. Diese Berechnungsmethode unterscheidet sich von der herkömmlichen Berechnungsmethode für Addition und Multiplikation dadurch, dass die Berechnung mit voller Auflösung durchgeführt wird und eine Rundung erst am Ende erfolgt. [16] Wie bereits erwähnt ist Tesla V100 mithilfe der Tensorkerne in der Lage 125 Tensor-TFLOPS für Training- und Inferenzanwendungen zu liefern.
Jeder Tensor-Kern rechnet auf einer 4×4-Matrix und führt die Operation $D = A \times B + C$ auf dieser aus, wobei A, B, C und D 4×4 Matrizen sind. In der Praxis werden mit den Tensor-Kernen Operationen auf größeren, auch mehrdimensionalen, Matrizen durchgeführt, die hierzu in diese 4×4 Matrizen zerlegt werden. Volta Streaming Multiprozessoren besitzen zudem ein verbessertes L1 Datencache- und Shared Memory-System, das die Effizienz steigern und die Programmierbarkeit vereinfachen soll. Durch die Kombination in einem Block verringert sich die Latenz bei

Abbildung 1: Volta GV100 Streaming Multiprocessor (SM) [14]

gleichzeitig besserer Bandbreite. Shared Memory bietet hohe Bandbreite, geringe Latenz und konstante Performance, muss jedoch bei der CUDA-Programmierung explizit verwaltet werden. Bei der Kombination von Cache und Shared-Memory wird fast die gleiche Leistung erreicht (ca. 7% weniger), jedoch ohne die Notwendigkeit dies in Anwendungen manuell zu steuern.
Neben den bereits teilweise zuvor erwähnten neuen Funktionalitäten, ist die Volta-Architektur im Gegensatz zu Pascal zuletzt in der Lage FP32- und INT32-Operationen gleichzeitig auszuführen, da separate FP32- und INT32-Kerne vorhanden sind, wodurch sich zum Beispiel bei FMA Geschwindigkeitsvorteile in bestimmten Anwendungsszenarien ergeben können.

3.1.2 AMD [10]

Auch wenn von AMD selbst als Vega-Architektur bezeichnet, so handelt es sich bei Vega streng genommen um die 5. Ausbaustufe der GCN-Architektur. Im Fol-

genden werden diese Begrifflichkeiten jedoch der Einfachheit halber synonym verwendet.
Für diese neue Architektur beziehungsweise Ausbaustufe sind verschiedene Implementierungen geplant, wobei die aktuelle als Vega 10 bezeichnet wird. Abbildung 3 zeigt die im 14 nm LPP FinFet Fertigungsprozess hergestellte Architektur Vega 10 mit ihren 12,5 Milliarden Transistoren. Durch das kleinere Fertigungsverfahren beträgt der Boost-Takt nun 1,67 Ghz gegenüber 1 Ghz bei Produkten mit Architekturen, die im 28 nm Verfahren hergestellt wurden. Bei Vega werden als Zielgruppen Gaming, Workstation, HPC und Machine Learning genannt.
Mit 64 Rechneneinheiten der nächsten Generation (Next-generation compute units (NCUs)) besitzt Vega 10 4096 Stream-Prozessoren, deren Anzahl bisherigen Radeon GPUs ähnelt, die aber aufgrund des Takts und der Architekturverbesserungen Instruktionsanweisungen schneller ausführen. Dadurch sind diese in der Lage 13,7 Teraflops von Berechnungen einfacher Genauigkeit und 27,4 Teraflops mit halber Genauigkeit durchzuführen. Die neue Architektur ist in Abbildung 3 dargestellt.
Des weiteren bietet Vega sogenanntes Infinity Fabric mit geringen Latenzen, das Logikeinheiten innerhalb des Chips verbindet und gleichzeitig Quality of Service und Sicherheitsapplikationen zur Verfügung stellt.
Ebenfalls neu in der Vega-Architektur ist eine neue Speicherhierarchie mit einem Cache Controller für hohe Bandbreite (HBCC). Üblicherweise werden Daten von den Registern der Rechenelemente aus dem L1-Cache bezogen, der seinerseits den L2-Cache nutzt. Dieser L2-Cache wiederum stellt einen Zugriff mit hoher Bandbreite und geringer Latenz auf den Videospeicher der GPU bereit. Dabei müssen das gesamte Datenset und alle Ressourcen im Videospeicher gehalten werden, da es ansonsten zu Einschränkungen bezüglich der Bandbreite und Latenz kommt. Bei Vega verhält sich der Videospeicher nun seinerseits wie eine Art letzter Cache, wozu dieser Speicherseiten, die noch nicht im GPU-Speicher vorliegen, selbst über den PCIe Bus bezieht und im Cache mit hoher Bandbreite speichert. Bisher wurde stattdessen die komplette Berechnung der GPU verzögert, bis der Kopiervorgang abgeschlossen war. Diese Neuheit der Speicherhierarchie bei der Vega-Architektur ist möglich, da die Speicherseiten meist

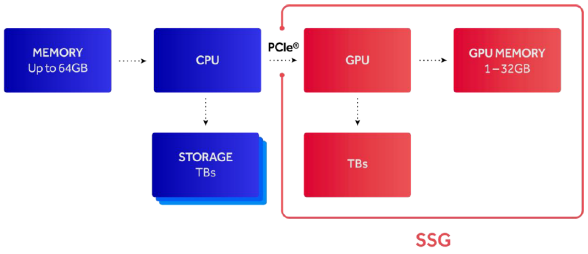


Abbildung 2: Systemspeicherarchitektur mit SSG [9]

deutlich kleiner als die kompletten Ressourcen sind und folglich schnell vom neuen Controller für hohe Bandbreite (HBCC), einem neuen Bestandteil der Speichercontrollerlogik, kopiert werden können.

[9] Davon profitiert insbesondere die neue Radeon Pro SSG, eine GPU mit Solid State Disk (SSD), die Latenz und CPU-Overhead deutlich reduzieren soll. Diese spezielle GPU verhält sich, als besäße sie ein Terabyte lokalen Videospeichers. Bei der bisher üblichen Systemarchitektur für Massenspeicher in einem CPU-GPU-System erfolgt die Datenübertragung zwischen zwei Endpunkten gewöhnlich in den folgenden drei Schritten: Zuerst werden die Daten von einem PCIe-Endpunkt in den System-Cache gelesen und anschließend von dort in den Prozesscache kopiert, von wo sie zuletzt zum anderen PCIe-Endpunkt kopiert werden.

Abbildung 2 zeigt wie die Systemspeicherarchitektur mit der Radeon Pro SSG aufgebaut ist. Dort werden sogenannte Ende-zu-Ende Leseoperationen ermöglicht, indem zunächst, über PCIe exponierter, Videospeicher reserviert wird, anschließend die SSD angewiesen wird eine Leseoperation durchzuführen und zuletzt die Daten auf der GPU verarbeitet werden. Dies hat auch den Vorteil, dass die CPU, sobald der Pfad zwischen GPU und NVMe-Speicher hergestellt ist, zum Datentransfer nicht mehr benötigt wird und folglich für andere Aufgaben verwendet werden kann, was die Systemperformance weiter erhöht, da Datensätze nur so schnell von der GPU verarbeitet werden können, wie diese dieser zur Verfügung stehen. Die Größe solcher Datensätze nimmt bei modernen Anwendungen kontinuierlich zu, was jedoch durch immer schnellere GPU-Architekturen kompensiert wird. Allerdings wird dann häufig die Datenübertragung zur GPU zum Flaschenhals, der mit der CPU-GPU-Systemarchitektur der Ra-

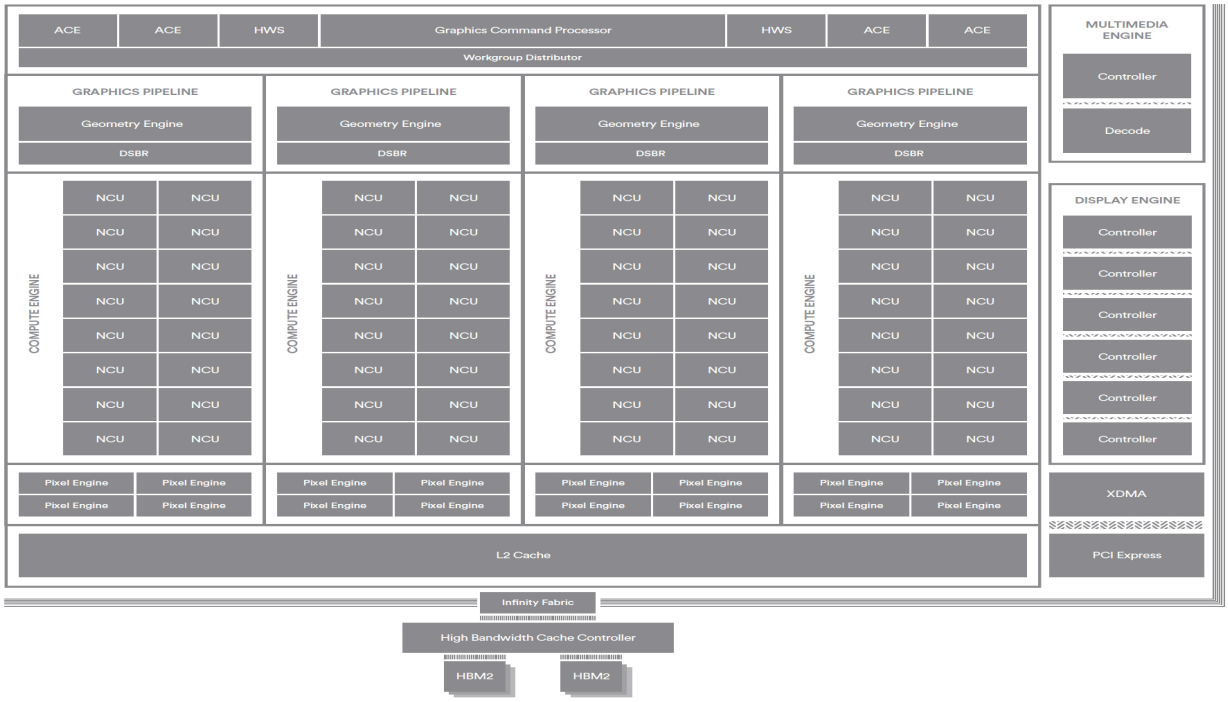


Abbildung 3: AMD Vega 10 Architektur [10]

deon Pro SSG entfällt.

Zur Vega 10-Archiektur zurückkommend gilt es weiterhin hervorzuheben, dass alle Grafikblöcke nun Clients des gemeinsamen L2-Cache sind, womit Datenredundanz verringert wird. Aus diesem Grund wurde der L2-Cache mit 4 MB gegenüber der Vorgängergeneration verdoppelt.

Außerdem wird eine Geometrie-Engine der nächsten Generation (NGG) in Vega integriert, die einen viel höheren Polygon-Durchsatz pro Transistor erreicht. Hierzu werden „primitive“ Shader unterstützt, womit Teile der Geometrie-Pipeline mit einem neuen Shader-Typ kombiniert und ersetzt werden. In üblichen Szenarien wird rund die Hälfte sogenannter Primitiven der Geometrie verworfen, was bisher durch die Geometrie-Pipelines erst nach der Vertex-Verarbeitung geschehen konnte, nun aber früher erfolgen kann. Dadurch erreichen die vier Geometrie-Engines von Vega 10 nun 17 statt nur 4 Primitive pro Takt.

Ebenfalls mit Vega wird sogenanntes „Rapid Packed Math“ für spezialisierte Anwendungen wie Machine Learning und Videoverarbeitung eingeführt, wobei 16-bit Datentypen, wie float und Integer, zum Tragen kommen. Dies ist ebenso Bestandteil der Vega NCU wie die Unterstützung neuer Instruktionen, zum Beispiel die Summe der absoluten Differenzen (SAD), die häufig in Video- und Bildverarbeitung benötigt wird.

Nicht zuletzt wurde auch die Pixel-Engine der Vega-Architektur überarbeitet, unter anderem in Form des Draw-Stream Binning Rasterizers (DSBR), der unnötige Berechnungen und Datentransfer vermeiden soll. Beim Rendern des Bildes wird dieses hierbei zunächst in ein Gitter von Kacheln aufgeteilt, wobei Stapel von Primitiven gebildet werden. Der DSBR traversiert anschließend diese Stapel nacheinander und überprüft, welche vollständige oder Teile von Primitive enthalten. Dadurch und durch weitere Maßnahmen soll die Leistungsaufnahme verringert und höhere Bildwiederholungsraten erreicht werden.

Abschließend werden mit der Vega Architektur DirectX in Feature Level 12.1, verbesserte Stromsparfunktionen und Display- und Multimediaverbesserungen wie HDR

integriert beziehungsweise unterstützt.

3.2 Leistungsfähigkeit

Dieses Kapitel beschreibt die Leistungsfähigkeit aktueller GPU-Architekturen und deren Umsetzungen. Ein direkter Vergleich ist jedoch schwierig beziehungsweise selten möglich, da die Architekturen zum einen sehr neu sind (Stand 12.2017) und zum anderen deren Umsetzungen unterschiedliche Andwenundgsfokusse haben.

3.2.1 NVidia Tesla V100 GPU [14]

Der Vergleich zwischen Pascal und Volta zeigt, dass die Matrixmultiplikation von 4×4 Matrizen und die dazu nötigen 64 Operationen von Volta zwölfmal so schnell berechnet werden. In cuBlAS, einer Bibliothek für Berechnungen der linearen Algebra, die auf CUDA aufbaut, weist Volta gegenüber Pascal bei einfacher Genauigkeit eine bis zu 1,8-fach und bei gemischter Genauigkeit (FP16 Input, FP32 Output) bis zu 9,3-fach höhere Geschwindigkeit auf.

3.2.2 AMD [10]

Die Radeon Pro SSG besitzt in einem CPU-GPU-System gegenüber einem herkömmlichen CPU-GPU-System eine rund fünf Mal höhere Performance bei der Videoverarbeitung in 8K. Weiterhin ist Vega, mithilfe der neuen Geometrie-Engine der nächsten Generation (NGG) und Fast Path, in der Lage über 25 Gigaprimitive, gegenüber ca. 5 bei Fiji, zu verwerfen.

Obwohl die GPUs von NVidia und AMD aufgrund der Spezialisierung nicht vollständig vergleichbar sind, existieren Bechmarks, wie der PassMark Benchmark [5], die die verschiedenen Karten dennoch vergleichen. In besagtem PassMark Benchmark (Stand 14.12.2017) führt die Titan V100 das Testfeld mit 16,842 Punkten mit einigem Vorsprung an. Eine GeForce GTX 1080 Ti, die zweithöchste Ausbaustufe der Pascal-Architektur, erreicht dagegen nur 13,634 Punkte, ist damit aber immer noch deutlich schneller als die schnellste Radeon Vega Karte im Test mit 11,876 Punkten. Damit wäre die aktuelle Umsetzung der Tesla-Architektur circa 50% schneller als diejenige der Vega-Architektur.

4 Programmierung für GPUs

Dieser Abschnitt beschreibt übliche Vorgehensweisen bei der GPU-Programmierung und dazu nötige Software. Die programmierbaren Einheiten einer GPU verfolgen dabei ein single-program multiple-data (SPMD) Programmiermodell, was bedeutet, dass eine GPU aus Effizienzgründen parallel mehrfach das gleiche Programm auf verschiedenen Elementen ausführt. Deshalb sind Programme für GPUs auch immer derart strukturiert, dass es mehrere gleichzeitig zu verarbeitende Elemente und parallele Verarbeitung dieser Elemente von einem einzigen Programm gibt.
Die Programmierung für Grafikanwendungen unterscheidet sich dabei von der Programmierung für allgemeine Anwendungen. Erstere erfolgt dabei folgendermaßen:

1. Der Programmierer spezifiziert die Geometrie, die eine bestimmte Region des Bildschirms abdeckt. Der Rasterer generiert daraus an jeder Pixelstelle ein Fragment, das von der Geometrie abgedeckt ist.
2. Jedes Fragment wird von einem Fragment-Programm bearbeitet.
3. Das Fragment-Programm berechnet dabei den Wert des Fragments mithilfe einer Kombination aus mathematischen Operationen und der globale Speicher liest von einem globalen Texturspeicher.
4. Das resultierende Bild kann dann als Textur in weiteren Durchläufen innerhalb der Grafik-Pipeline weiter verwendet werden.

und für allgemeine Anwendungen wie folgend:

1. Der Programmierer spezifiziert direkt den Berechnungsbereich als strukturiertes Raster von Threads.
2. Ein SPMD-Progamm für allgemeine Anwendungen berechnet den Wert jedes Threads.
3. Der Wert wird dabei erneut durch eine Kombination mathematischer Operationen, sowie lesendem und schreibendem Zugriff auf den globalen Speicher berechnet. Der Buffer kann dabei sowohl für lesende und schreibende Operationen verwendet werden, um flexiblere Algorithmen zu ermöglichen (z.B. In-Place-Algorithmen zur Reduzierung der Speichernutzung).
4. Der resultierende Buffer im globalen Speicher kann wiederum als Eingabe für weitere Berechnungen genutzt werden.

Früher musste bei der Programmierung der Umweg über geometrische Primitive, den Rasterer und

Fragment-Programme, ähnlich wie bei der Grafikprogrammierung, gegangen werden.
Dies legt bereits nahe, dass als Softwareumgebung für die GPGPU-Programmierung früher direkt die Grafik-API genutzt wurde. Auch wenn dies vielfach erfolgreich genutzt wurde, unterscheiden sich, wie bereits oben erwähnt, die Ziele herkömmlicher Programmiermodelle und einer Grafik-API, denn es mussten Fixfunktionen, grafik-spezifische Einheiten, wie zum Beispiel Texturfilter und Blender, verwendet werden.
Mit DirectX 9 wurde die high-level shading language (HLSL) eingeführt, die es erlaubt die Shader in einer C-ähnlichen Sprache zu programmieren. NVidias Cg verfügte über ähnliche Fähigkeiten, war allerdings in der Lage für verschiedene Geräte zu kompilieren und stellte gleichzeitig die erste höhere Sprache für OpenGL bereit. Heute ist die OpenGL Shading Language (GLSL) die Standard-Shading-Sprache für OpenGL. Nichtsdestotrotz sind Cg, HLSL und GLSL immer noch Shading-Sprachen, wobei die Programmierung grafikähnlich erfolgen muss.
Inzwischen gibt es deshalb sowohl von NVidia, als auch von AMD, spezielle GPGPU Programmiersysteme. NVidia stellt hierzu das bekannte CUDA-Interface zur Verfügung, das wie Cg über eine C-ähnliche Syntax besitzt, über zwei Level von Parallelität (Daten-Parallelität und Multithreading) verfügt und nicht mehr grafikähnlich programmiert werden muss.
AMD bietet Hardware Abstraction Layer (HAL) und-Compute Abstraction Layer (CAL). Außerdem wird OpenCL unterstützt, das seinerseits von CUDA inspiriert wurde und auf einer großen Bandbreite von Geräte genutzt werden kann. Bei OpenCL erfolgt die Programmierung in OpenCL C, das zusätzliche Datentypen und Funktionen für die parallele Programmierung bereitstellt. Es handelt sich also wie bei der Programmierung von CUDA um eine C-ähnliche Syntax. [15], [13]

5 Performance-Analyse bei GPUs

Progammierwerkzeuge zur Laufzeitanalyse werden im allgemeinen Profiler genannt, die den Programmierer dabei unterstützen ineffiziente Programmteile zu identifizieren. Der Fokus liegt hierbei auf Funktionen, die am häufigsten durchlaufen werden, da diese den größten Einfluss auf die Gesamtlaufzeit eines Programms haben. Auch wenn moderne Profiler Nebenläufigkeit unterstützen, so stellt diese enorme Anforderungen an etwaige Analysetools. Schon bei einer Threadanzahl wie sie bei einer CPU vorkommt kann es zu Schwierigkeiten bei der Erkennung von Laufzeitproblemen, wie etwa durch Deadlocks, kommen.

5.1 High-Level Analyse

Als High-Level wird im Folgenden die Analyse der GPU-Performance mit Softwaretools bezeichnet, welche aber oft ihrerseits Low-Level-Analyse nutzen und diese in Form einer API oder Ähnlichem dem Programmierer zur Verfügung stellen. Die einzelnen Tools und ihre Funktionsweise ist genauer in Unterabschnitt 5.3 beschrieben.
Als Beispiel, wie High-Level-Analyse durchgeführt werden kann, soll die Arbeit von Sim, Dasgupta, Kim und Vuduc [17] dienen, die feststellen, dass es sich äußerst schwierig gestaltet, bei Plattformen mit vielen Kernen die Ursachen für Engpässe festzustellen. Deshalb stellen die genannten Autoren ein Framework namens GPUPerf Framework vor, das solche Engpässe aufdecken soll. Es kombiniert ein genaues analytisches Modell für moderne GPGPUs und gut interpretierbare Metriken, die direkt die möglichen Performance-Verbesserungen für verschiedene Klassen von Optimierungstechniken zeigen. Das zugrunde liegende analytische Modell, das als MWP-CWP-Modell (siehe Abbildung 4) bezeichnet wird, nutzt folgende Eingaben: Anzahl der Instruktionen, Speicherzugriffe, Zugriffsmuster sowie architekturelle Eigenschaften wie DRAM-Latzen und -Bandbreite.
Auf der einen Seite beschreibt der Memory Warp Parallelism (MWP) die Anzahl an Warps, Ausführungseinheiten einer GPGPU, pro Streaming-Multiprozessor (SM), die gleichzeitig auf den Speicher zugreifen können. Auf Ebene des Speichers spiegelt MWP also Parallelität wider. MWP ergibt sich dabei als Funktion aus Speicherbandbreite, Parameter von Speicheroperationen, wie Latzenz und der aktiven Anzahl von Warps in einem SM. Das Framework modelliert die Kosten von Speicheroperationen durch die Anzahl von Anfragen über MWP. Auf der anderen Seite entspricht Computation Warp Parallelism (CWP) der Anzahl von Warps, die ihre Berechnungsperiode innerhalb einer Speicherwarteperiode plus eins abschließen können.

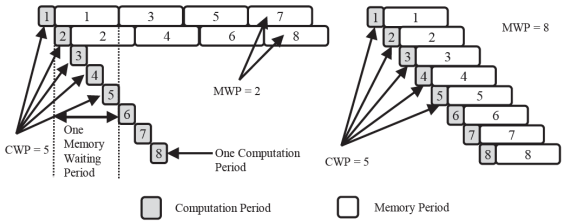


Abbildung 4: MWP-CWP-Modell, links: MWP $<$ CWP, rechts: MWP $>$ CWP [17]

Eine Berechnungsperiode ergibt sich dabei aus den durchschnittlichen Berechnungszyklen pro Speicherinstruktion. Das MWP-CWP-Modell dient dazu, zu identifizieren, welchen Kosten durch Multi-Threading verborgen werden können. Dazu wird in drei Fälle unterschieden:

1. MWP $<$ CWP: Die Kosten einer Berechnung werden durch Speicheroperationen verdeckt. Die gesamten Ausführungskosten ergeben sich durch die Speicheroperationen. (vgl. Abbildung 4, links)
2. MWP $\geq$ CWP: Die Kosten der Speicheroperationen werden durch die Berechnung verdeckt. Die gesamten Ausführungskosten ergeben sich aus der Summe der Berechnungskosten und einer Speicherperiode. (vgl. Abbildung 4, rechts)
3. Nicht genug Warps: Aufgrund mangelnder Parallelisierung werden sowohl Berechnungs- als auch Speicherkosten verdeckt.

Das Framework enthält nun einen sogenannten Performance-Advisor, der Informationen über Performance-Engpässe und möglichen Performance-Gewinn durch Beseitigung dieser Engpässe bereitstellt. Dazu nutzt dieser Advisor eben obiges MWP-CWP-Modell und stellt vier Metriken bereit, die sich entsprechend Abbildung 5 visualisieren lassen. Die vier enthaltenen Metriken sind:

- B_{itilp}: zeigt den potentiellen Performance-Gewin durch Erhöhung der Inter-Thread Parallelität auf Instruktionslevel.
- B_{memlp}: zeigt den potentiellen Performance-Gewin durch Erhöhung der Parallelität auf Speicherlevel.
- B_{fp}: spiegelt den potentiellen Performance-Gewinn wider, wenn die Kosten für ineffiziente Berechnungen ideal entfernt werden. In der Praxis können hier keine 100% erreicht werden, da der Kernel immer auch Operationen wie Datentransfer haben muss.
- B_{serial}: zeigt die Kosteneinsparung, wenn Overhead durch Serialisierungseffekte wie Synchronisation und Ressourcenkonflikte entfernt werden.

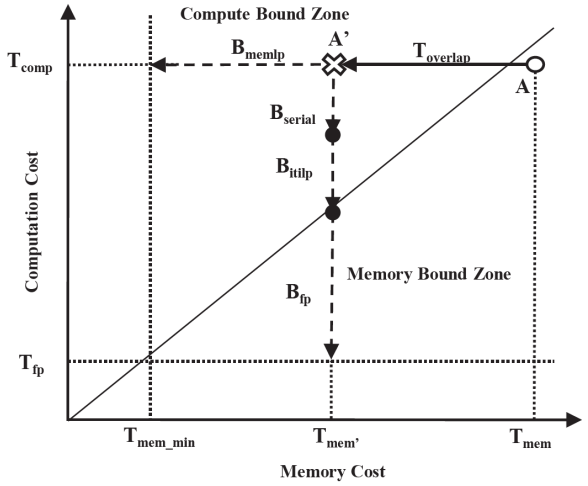


Abbildung 5: Potentieller Performance-Gewinn [17]

Neben allgemeiner High-Level-Analyse gibt es auch Performance-Modellierung und -Optimierung, die sich auf spezielle Operationen bezieht. Guo, Wang und Chen [11] stellen zum Beispiel ein Analysetool vor, das Sparse-Matrix-Vektor-Multipliktion auf GPUs optimieren kann. Dazu stellt es, unter Verwendung einer integrierten analytischen und profilbasierten Performance-Modellierung, ebenfalls verschiedene Performance-Modelle bereit, um die Berechnungsdauer präzise vorherzusagen. Außerdem ist in dem genannten Tool ein automatischer Selektionsalgorithmus enthalten, der optimale Sparse-Matrix-Vektor-Multiplikation-Werte hinsichtlich Speicherstrategie, Speicherformaten und Ausführungszeit für eine Zielmatrix liefert.
Weitere Frameworks sind in Unterabschnitt 5.3 beschrieben.

5.2 Low-Level Analyse

Auf GPU-Basis ist die Low-Level Analyse mithilfe sogenannter Performance-Counter und anderen Methoden noch relativ neu. Verschiedene Hersteller wie NVidia

(The PAPI CUDA Component) und IBM (Parallel Environment Developer Edition) stellen Technologien bereit, die es ermöglichen auf Hardware-Counter innerhalb der GPU zuzugreifen. Prinzipiell ist der Übergang und die Definition von High- und Low-Level Analyse aber natürlich fließend.
[6] Der Einsatz der GPU-Counter soll anhand der Performance API (PAPI) gezeigt werden, die daneben aber auch noch sehr viele weitere Performanceanalyse-Funktionen bereitstellt. PAPI bietet hierzu ein Counter-Interface, das den Zugriff, das Starten, Stoppen und Auslesen von Countern für eine spezifizierte Liste von Events ermöglicht. Die Steuerung beziehungsweise Programmierung von PAPI erfolgt mittels C oder Fortran. Außerdem kann PAPI sogenannte benutzerdefinierte Gruppen von Hardware-Events namens Event-Sets verwalten, was für Programmierer einem feingranulares Mess- und Kontrollwerkzeug gleichkommen soll.
Auch die Umsetzung beziehungsweise Implementierung des zuvor genannten GPUPerf Frameworks kann als Low-Level Analyse bezeichnet werden.
Zuletzt gibt es auch Empfehlungen und Vorgehensweisen, beispielsweise von NVidia selbst [12], für Performance-orientierte Implementierungen. Allgemeine Empfehlungen sind in diesem Kontext zum Beispiel, mit 128-256 Threads pro Threadblock zu sarten, nur Vielfache der Warpgröße (32 Threads) zu nutzen und als Rastergröße 1000 oder mehr Threadblöcke einzusetzen.

5.3 Hilfsmittel (Tools)

Zur Analyse der Laufzeit gibt es auch bei GPUs eine Reihe von Profiling-Tools. Neben den bereits, zur Veranschaulichung von High- und Low-Level Analyse, vorgestellten Frameworks, ist eines dieser Tools NVidia PerfKit. [18]
Durch die enthaltene PerfAPI werden in PerfKit Performance-Counter zugänglich, die aufzeigen, wie ein Programm die GPU nutzt, welche Performance-Probleme vorliegen und ob nach einer Veränderung diese Probleme gelöst wurden. PerfKit kann dedizierte Experimente auf individuellen GPU-Einheiten ausführen, worduch Performance-Charakteristiken gewonnen werden, nämlich der Speed of Light (SOL)- und Bottleneck-Wert. Ersterer gibt im Allgemeinen an, wie stark der entsprechende GPU-Teil genutzt wird; er spiegelt die aktive Zeit der entsprechenden GPU-Einheit wider. Letzterer zeigt welchen zeitlichen Anteil eine GPU-Einheit einen Engpass darstellt. Es kann eine große Anzahl an verschiedenen GPU-Counter untersucht werden.
Der AMD Radeon GPU Profiler stellt Low-Level Timing-Daten für Barriers, Warteschlangensignale, Wellenfront-Belegung, Event-Timings und weitere bereit. [7]
Ebenfalls für AMD Chips steht die AMD CodeXL Entwicklungssuite bereit, die neben einem CPU-Profiler, einem GPU debugger, einem statischen Shader/Kernel Analysator auch einen GPU Profiler beinhaltet. Dieser sammelt Daten von den Performance-Countern, vom Anwendungsablauf, Kernelbelegung und bietet eine Hotspot-Analyse. Der Profiler sammelt Daten der OpenCL Runtime und von der GPU selbst und dient ebenfalls zur Reduzierung von Engpäassen und der Optimierung der Kernel-Ausführung. [8]
Die Autoren Zhang und Owens [19] schlagen ein quantitatives Performance-Analyse-Modell basierend auf einem Microbenchmark-Ansatz vor, das den nativen Instruktionssatz einer GPU nutzt.
Daneben existiert natürlich noch eine Vielzahl weiterer Frameworks und Tools, die zum GPU Profiling eingesetzt werden können, aber hier der Übersichtlichkeit halber keine Erwähnung finden sollen. Es ist stets eine Vergleichbarkeit mit CPU-Profiling gegeben, mit dem Unterschied, dass Parallelität bei GPUs naturgemäß eine viel entscheidendere Rolle spielt.

6 Zusammenfassung

Mit dieser Arbeit konnte zu Beginn dargelegt werden, wie sich Graphical Processing Units (GPUs). Seit dem Beginn der Entwicklung dedizierter Grafikchips hat sich der Funktionsumfang dieser kontinuierlich erweitert und die Transistorenzahl stark erhöht. Außerdem konnte ein Wandel von reinen Grafik-Chips hin zu multifunktionalen GPGPUs identifiziert werden, der mit dem Einzug von Programmierschnittstellen wie CUDA seinen Anfang fand. Immer umfangreichere und effizientere Programmierschnittstellen wie OpenGL, DirectX und die Vulkan API werden in ihren neuen Versionen in der Regel mit neueren Architekturen der Grafik-Chips integriert. Die Leistungsfähigkeit der GPUs hat im Lau-

fe der Zeit stark zugenommen, wobei diese heute primär vom Anwendungsgebiet abhängt.

Die neuen Architekturen von NVidia und AMD, nämlich Volta und Vega, haben gemeinsam, dass sie schnellen HBM2 Speicher verwenden. Bei den Neuheiten der Volta-Architektur sind besonders die neue Streaming-Multiprozessor-Architektur mit Tensor-Kernen für Matrixmultiplikationen und bei der AMD Vega-Architektur die 64 Next-generation compute units (NCUs) und eine neue Speicherhierarchie mit eine Cache Controller für hohe Bandbreite (HBCC) hervorzuheben.

Die Leistungsfähigkeit profitiert von diesen Architektur-Veränderungen zum Teil erheblich. Neben allgemeinen Geschwindigkeitsvorteilen in Benchmarks, erreicht die AMD Radeon Pro SSG, eine auf Vega basierende GPU mit eigener SSD, eine rund fünfmal höhere Performance als eine herkömmliche Systemarchitektur, wodurch ein flüssiges Videoprocessing bei 8K ermöglicht wird. NVidias Volta erreicht mithilfe der Tensor-Kerne bei der Matrixmultiplikation gegenüber der Vorgängergeneration Pascal eine zwölffache Geschwindigkeit.

Bei der Programmierung, wie diese Arbeit darlegt, muss heute sehr stark in Programmierung für herkömmliche Grafikanwendungen und allgemeine Anwendungen differenziert werden.

Die Performance-Analyse erfolgt auch bei GPUs mithilfe von Profilern, die im Wesentlichen eine High-Level Analyse für den Programmierer bereitstellen, indem sie selbst zum Teil eine Low-Level Analyse durchführen. Entscheidend für das GPU-Profiling ist, ob Berechnungskosten Speicherkosten verdecken beziehungsweise umgekehrt oder mangelnde Parallelisierung vorliegt.

Weiterer Forschungsbedarf besteht zum Beispiel in der Performance-Analyse bei Machine-Learning, insbesondere bei und in Kombination mit den neuen Tensor-Kernen der Volta-Architektur.

Insgesamt zeigt sich mit den aktuellen GPU-Architekturen eine weitere Fortführung der Spezialisierung von GPUs und ein anhaltender Trend hin zu GPGPUs, wobei inzwischen sogar der Hauptfokus auf wissenschaftlichen und allgemeinen Anwendungen zu liegen scheint.

Literatur

[1] https://www.duden.de/rechtschreibung/GPU_Grafikprozessor_EDV (aufgerufen am 06.12.2017).

[2] http://www.3dcenter.org/artikel/launch-analyse-nvidia-geforce-gtx-titan-x (aufgerufen am 06.12.2017).

[3] https://www.nvidia.de/self-driving-cars/drive-px/ (aufgerufen am 06.12.2017).

[4] https://www.nvidia.de/data-center/dgx-1/ (aufgerufen am 07.12.2017).

[5] https://www.videocardbenchmark.net/high_end_gpus.html (aufgerufen am 14.12.2017).

[6] http://icl.cs.utk.edu/projects/papi/wiki/Counter_Interfaces (aufgerufen am 14.12.2017).

[7] https://gpuopen.com/gaming-product/radeon-gpu-profiler-rgp/ (aufgerufen am 14.12.2017).

[8] https://gpuopen.com/compute-product/codexl/ (aufgerufen am 14.12.2017).

[9] A. —. R. T. Group. *RADEON PRO - Solid State Graphics (SSG) Technical Brief*. Techn. Ber. https://www.amd.com/Documents/Radeon-Pro-SSG-Technical-Brief.pdf (aufgerufen am 26.11.2017).

[10] A. —. R. T. Group. *Radeon's next-generation Vega architecture*. Techn. Ber. https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf (aufgerufen am 26.11.2017). Juni 2017.

[11] P. Guo, L. Wang und P. Chen. "A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* 25.5 (2014), S. 1112–1123.

[12] P. Micikevicius. "GPU performance analysis and optimization". In: *GPU technology conference*. Bd. 84. 2012.

[13] J. Nickolls und W. J. Dally. "The GPU computing era". In: *IEEE micro* 30.2 (2010).

[14] NVidia. *NVIDIA TESLA V100 GPU ARCHITECTURE. THE WORLD'S MOST ADVANCED DATA CENTER GPU"*. Techn. Ber. http://www.nvidia.com/object/volta-architecture-whitepaper.html (aufgerufen am 26.11.2017). Aug. 2017.

[15] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone und J. C. Phillips. "GPU computing". In: *Proceedings of the IEEE* 96.5 (2008), S. 879–899.

[16] E. Quinnell, E. E. Swartzlander und C. Lemonds. "Floating-point fused multiply-add architectures". In: *Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on*. IEEE. 2007, S. 331–337.

[17] J. Sim, A. Dasgupta, H. Kim und R. Vuduc. "A performance analysis framework for identifying potential benefits in GPGPU applications". In: *ACM SIGPLAN Notices*. Bd. 47. 8. ACM. 2012, S. 11–22.

[18] N. P. Toolkit. "NVIDIA PerfKit". In: (2013).

[19] Y. Zhang und J. D. Owens. "A quantitative performance analysis model for GPU architectures". In: *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE. 2011, S. 382–393.

Seminar HPC Trends
Winter Term 2017/2018

New Operating System Concepts for High Performance Computing

Fabian Dreer
Ludwig-Maximilians Universität München
dreer@cip.ifi.lmu.de

January 2018

Abstract

When running large-scale applications on clusters, the noise generated by the operating system can greatly impact the overall performance. In order to minimize overhead, new concepts for HPC OSs are needed as a response to increasing complexity while still considering existing API compatibility.

In this paper we study the design concepts of heterogeneous kernels using the example of *mOS* and the approach of library operating systems by exploring the architecture of *Exokernel*. We summarize architectural decisions, present a similar project in each case, *Interface for Heterogeneous Kernels* and *Unikernels* respectively, and show benchmark results where possible.

Our investigations show that both concepts have a high potential, reduce system noise and outperform a traditional Linux in tasks they are already able to do. However the results are only proven by micro-benchmarks as most projects lack the maturity for comprehensive evaluations at the point of this writing.

1 The Impact of System Noise

When using a traditional operating system kernel in high performance computing applications, the cache and interrupt system are under heavy load by e.g. system services for housekeeping tasks which is also referred to as noise. The performance of the application is notably reduced by this noise.

Even small delays from cache misses or interrupts can affect the overall performance of a large scale application. So called *jitter* even influences collective communication regarding the synchronization, which can either absorb or propagate the noise. Even though asynchronous communication has a much higher probability to absorb the noise, it is not completely unaffected. Collective operations suffer the most from propagation of jitter especially when implemented linearly. But it is hard to analyse noise and its propagation for collective operations even for simple algorithms. Hoefler et al. [5] also suggest that "at large-scale, faster networks are not able to improve the application speed significantly because noise propagation is becoming a bottleneck." [5]

Hoefler et al. [5] also show that synchronization of point-to-point and collective communication and

 DOI: 10.14459/2018md1430700

OS noise are tightly entangled and can not be discussed in isolation. At full scale, when it becomes the limiting factor, it eliminates all advantages of a faster network in collective operations as well as full applications. This finding is crucial for the design of large-scale systems because the noise bottleneck must be considered in system design.

Yet very specialized systems like BlueGene/L [8] help to avoid most sources of noise [5]. Ferreira et al. [3] show that the impact is dependent on parameters of the system, the already widely used concept of dedicated system nodes alone is not sufficient and that the placement of noisy nodes does matter.

This work gives an overview of recent developments and new concepts in the field of operating systems for high performance computing. The approaches described in the following sections are, together with the traditional Full-Weight-Kernel approach, the most common ones.

The rest of this paper is structured as follows. We will in the next section introduce the concept of running more than one kernel on a compute or service node while exploring the details of that approach at the example of *mOS* and the *Interface for Heterogeneous Kernels*. Section 3 investigates the idea of library operating systems by having a look at *Exokernel*, one of the first systems designed after that concept, as well as a newer approach called *Unikernels*. Section 4 investigates *Hermit Core* which is a combination of the aforementioned designs. After a comparison in Section 5 follows the conclusion in Section 6.

2 Heterogeneous Kernels

The idea about the heterogeneous kernel approach is to run multiple different kernels side-by-side. Each kernel has its spectrum of jobs to fulfill and its own dedicated resources. This makes it possible to have different operating environments on the partitioned hardware. Especially with a look to heterogeneous architectures with different kinds of memory and multiple memory controllers, like the recent Intel Xeon Phi architecture, or chips with different types of cores and coprocessors, specialized kernels might help to use the full potential available.

We will first have an in-depth look at *mOS* as at this example we will be able to see nicely what aspects have to be taken care of in order to run different kernels on the same node.

2.1 mOS

To get a light and specialized kernel there are two methods typically used: The first one is to take a generic Full-Weight-Kernel (FWK) and stripping away as much as possible; the second one is to build a minimal kernel from scratch. Either of these two approaches alone does not yield a fully Linux compatible kernel, which in turn won't be able to run generic Linux applications [4].

Thus the key design parameters of *mOS* are: full linux compatibility, limited changes to Linux, and full Light-Weight-Kernel scalability and performance, where performance and scalability are prioritized.

To avoid the tedious maintenance of patches to the Linux kernel, an approach inspired by FUSE has been taken. Its goal is to provide internal APIs to coordinate resource management between Linux and Light-Weight-Kernels (LWK) while still allowing each kernel to handle its own resources independently.

"At any given time, a sharable resource is either private to Linux or the LWK, so that it can be managed directly by the current owner." [11] The resources managed by LWK must meet the following requirements: i) to benefit from caching and reduced TLB misses, memory must be in physically contiguous regions, ii) except for the ones of the applications no interrupts are to be generated, iii) full control over scheduling must be provided, iv) memory regions are to be shared among LWK processes, v) efficient access to hardware must

be provided in userspace, which includes well-performing MPI and PGAS runtimes, vi) flexibility in allocated memory must be provided across cores (e.g. let rank0 have more memory than the other ranks) and, vii) system calls are to be sent to the Linux core or operating system node.

mOS consists of six components which will be introduced in one paragraph each:

According to Wisniewski et al. [11], the Linux running on the node can be any standard HPC Linux, configured for minimal memory usage and without disk paging. This component acts like a service providing Linux functionality to the LWK like a TCP/IP stack. It takes the bulk of the OS administration to keep the LWK streamlined, but the most important aspects include: boot and configuration of the hardware, distribution of the resources to the LWK and provision of a familiar administrative interface for the node (e.g. job monitoring).

The LWK which is running (possibly in multiple instantiations) alongside the compute node Linux. The job of the LWK is to provide as much hardware as possible to the applications running, as well as managing its assigned resources. As a consequence the LWK does take care of memory management and scheduling [11].

A transport mechanism in order to let the Linux and LWK communicate with each other. This mechanism is explicit, labeled as *function shipping*, and comes in three different variations: via shared memory, messages or inter-processor interrupts. For shared memory to work without major modifications to Linux, the designers of *mOS* decided to separate the physical memory into Linux-managed and LWK-managed partitions; and to allow each kernel read access to the other's space. Messages and interrupts are inspired by a model generally used by device drivers; thus only sending an interrupt in case no messages are in the queue, otherwise just queueing the new system call request which will be handled on the next poll. This avoids floods of interrupts in bulk-synchronous programming. To avoid jitter on compute cores, communication is in all cases done on cores running Linux [11].

The capability to direct system calls to the correct implementor (referred to as *triage*). The idea behind this separation is that performance critical system calls will be serviced by the LWK to avoid jitter, less critical calls, like signaling or `/proc` requests handles the local Linux kernel and all operations on the file system are offloaded to the operating system node (OSN). But this hierarchy of system call destinations does of course add complexity not only to the triaging but also to the synchronization of the process context over the nodes [11].

An offloading mechanism to an OSN. To remove the jitter from the compute node, avoid cache pollution and make better use of memory, using a dedicated OSN to take care of I/O operations is already an older concept. Even though the design of *mOS* would suggest to have file system operations handled on the local linux, the offloading mechanism improves resource usage and client scaling [11].

The capability to partition resources is needed for running multiple kernels on the same node. Memory partitioning can be done either statically by manipulating the memory maps at boot time and registering reserved regions; or dynamically making use of hotplugging. These same possibilities are valid for the assignment of cores. Physical devices will in general be assigned to the Linux kernel in order to keep the LWK simple [11].

We have seen the description of the *mOS* architecture which showed us many considerations for running multiple kernels side-by-side. As the design of *mOS* keeps compatibility with Linux core data structures, most applications should be supported. This project is still in an early development stage, therefore an exhaustive performance evaluation is not feasible at the moment.

2.2 Interface for Heterogeneous Kernels

This project is a general framework with the goal to ease the development of hybrid kernels on many-

core and accelerator architectures; therefore *attached* (coprocessor attached to multi-core host) and *builtin* (standalone many-core platform) configurations are possible. It follows the two design principles of keeping the interface minimal on the one hand, and providing a requisite utility library for kernels on the other hand.

Similar to *mOS* yet less strict, IHK defines the requirements for a hybrid kernel approach to be i) management of kernels and an interface to allocate resources, ii) resource partitoning and, iii) a communication mechanism among kernels. In IHK it is assumed that one kernel manages at most one processor.

The framework consists of the following components:

IHK-Master has the ability to boot other kernels. The mechanisms needed to do so are the same as discussed in the architecture description of *mOS* about partitioning resources. The master kernel also makes the user interface available.

IHK-Slave defines an interface for slave kernels to work with each other. Kernels of this type only run in their assigned space, retrieve that information from, and are booted by, the master kernel.

The IHK-IKC (communication model) provides rudimentary functions for the use of channels, where a channel is a pair of message queues. Master and slave kernels have an interface to use the IKC. This slightly differs from what we've seen in *mOS*, as IHK provides a library for using interrupts and queuing where the exact implementation is free. The included IKC library provides functions to setup a client-server layout among the kernels with a master channel to share control messages [10].

For easier development of LWKs, a bootstrap library is part of IHK. Currently an implementation for x86_64 is available. The delegation of system calls works in concept exactly like we've seen in *mOS*, with the difference that there is no operating system node where file system operations can be sent to.

"While IHK/McKernel is in a more advanced phase than *mOS* at this moment, both projects are too early in their development cycle for doing an exhaustive performance study." [4]

To still have an idea what can be expected from the heterogeneous kernel approach, Figure 1 shows benchmark results from *FusedOS* [9], which was the first prototype incorporating the idea. The work of Wisniewski et al. [11] is also based on *FusedOS*, therefore the overall tendency should be comparable.

The x-axis of Figure 1 shows time while the y-axis shows the number of iterations performed during a certain quantum of time. We see the performance of the FusedOS PECs (Power-Efficient-Cores) — which can be thought of as the LWKs of *mOS* — in purple above the red Linux. High-frequency noise as well as occasional large spikes can be seen in the Linux curve. Especially these large spikes are detrimental to the performance on large-scale clusters. In comparison, the *FusedOS* PEC curve has the form of a straight line, thus not displaying any spikes; for that reason we would tend to believe that the behavior of the application running on *FusedOS* is deterministic [9].

To sum up, even though some prototypes of heterogeneous kernels are still in their early phases, the concept itself looks promising. Light-Weight-Kernels run almost completely deterministically and show superior noise properties.

3 Library Operating Systems

The job of a traditional OS is to abstract away the hardware and isolate different processes, owned by potentially multiple users, from one another, as well as from the kernel. This abstraction is commonly realized by the differentiation, and therefore separation, into kernel space and user space.

But as this makes the abstraction fix, this concept can also limit performance and freedom of implementation. As a result applications are denied the possibility of domain-specific optimizations; this

presetting also discourages changes to the abstractions.

Conceptually a library operating system (libOS) is built around an absolutely minimalistic kernel, which exports all hardware resources directly via a secure interface. The operating system, which implements higher level abstractions, uses this interface.

Therefore the (untrusted) OS lives entirely in user space, which effectively moves the whole resource management to user space. This has the advantage that parts like e.g. virtual memory are user defined and offer more flexibility as well as specialization to the application. Another strong point of this design is the reduction of context switches for privileged operations the kernel would normally have to execute [2].

In contrast to the previously described heterogeneous kernel approach, libraryOS concepts work with exactly one kernel which is then used by multiple libOSs. Similar to our investigation of the heterogeneous kernel approach, we will discuss first the concept of *Exokernel* in detail, then take a look at the younger variants named *Unikernels*.

3.1 Exokernel

The challenge for the exokernel approach is to give the libOSs maximal freedom while still secluding them in such a way that they do not affect each other. A low-level interface is used to separate protection from management. The kernel performs the important tasks of i) keeping track of resource ownership, ii) guarding all resource binding points and usage as well as, iii) revoking resource access.

In order to protect resources without managing them at all, the designers of *Exokernel* decided to make the interface in such a way that all hardware resources could be accessed as directly as possible because the libOS knows best about its needs. This is supposed to be possible by exposing allocation, physical names, bookkeeping data structures and revocation. Additionally a policy is needed to handle competing requests of different libOSs. In case of an exokernel the decisions to make are all about resource allocation and are handled in a traditional manner with e.g. reservation schemes or quotas. In the following paragraphs we will have a closer look at some other mechanisms of the exokernel [2] architecture.

Exokernel uses a technique referred to as *secure bindings* in order to multiplex resources so that they are protected against unintended use by different libOSs. The point in time where a libOS requests allocation of a resource is called *bind time*, subsequent use of that resource is known as *access time*. By doing authorization checks only at *bind time* this mechanism improves efficiency. Another aspect is that this way of handling checks strengthens the separation of management and protection as the kernel does not need to know about the complex semantics of resources at *bind time*. *Secure bindings* are implemented with hardware mechanisms, caching in software and the download of application code into the kernel. As this downloading mechanism is not as common as the other two, an example can be found in the paragraph about network multiplexing in this section.

The multiplexing of physical memory is done with *secure bindings* as well. When the libOS requests a page of memory, the kernel creates a binding for that page with additional information on the capabilities of this page. The owner of a page is allowed to manipulate its capabilities, and these capabilities are used to determine the access rights for that memory page. Therefore applications can grant memory access to other applications which makes resource sharing easier.

Multiplexing of network resources efficiently is rather hard with the design philosophy requiring separation of protection, which includes delivering packets to the correct libOS, and management, e.g. creating connections and sessions. To deliver the packets correctly it is necessary to understand the logic of their contents. This can be done by either requesting each possible recipient (every libOS), or, more efficiently, with the use of downloaded application code in the packet filter to handle the packet.

 DOI: 10.14459/2018md1430700

This code can be run with immediate execution on kernel events which avoids costly context switches or otherwise required scheduling of each application. As this code is untrusted, it should be combined with security mechanisms like sandboxing [2].

Finally there must be a way to reclaim allocated resources. For this a *resource revocation protocol* has been implemented. Typically a kernel does not inform the OS when physical memory is allocated or deallocated. But the design of exokernels strives to give the libOS the most direct access to hardware possible. Therefore the revocation is visible for most resources which allows the libOS to react to a revocation request accordingly by e.g. saving a certain state. In cases where the application becomes unresponsive there is an *abort protocol* which can be understood as issuing orders instead of requests. Still if the libOS cannot comply, *secure bindings* to allocated resources must be broken by force. To complement this behavior, the libOS actively releases resources no longer needed.

In sum we have seen that exokernel approaches do not provide the same functionality as a traditional OS, but offer a way of running specialized systems with high performance implemented mostly in user space. Engler et al. [2] already show that the concept of exokernels and their implementation can be very efficient, and that it is efficient as well to build traditional OS abstractions on application level. Yet, as the experiments are quite a few years old already, we will investigate more recent approaches based on this concept in the next subsection.

3.2 Unikernels

As it is difficult to support a wide range of real-world hardware with the exokernel approach, libOSs have never been widely deployed. This problem can be solved with the use of virtualization hypervisors which are already very popular today especially in cloud environments [7].

The key difference between the previously shown exokernel architecture and a unikernel is that unikernels are single-purpose, single-image and single-address-space applications that are, at compile-time, specialized into standalone kernels. To make the deployment of unikernels easier, the configuration is integrated into the compilation process. In comparison, Linux distributions rely e.g. on complex shell scripting to pack components into packages. When deployed to e.g. a cloud platform, unikernels get sealed against modifications. In return they offer significant reduction in image size, improved efficiency and security, and should also reduce operational costs [7].

In the following paragraphs we will now have a closer look at the architecture of such a single-purpose application, often referred to as an *appliance*.

Application configurations are usually stored in dedicated files, one for each service. The view of services in the unikernel architecture is not the one of independent applications but are seen as libraries of one single application. As a result the configuration is either done at build time for static parameters or with library calls for dynamic ones. This concept eases the configuration of complex layouts and also makes configurations programmable, analyzable and explicit.

Another advantage of linking everything as a library, even functions that would normally be provided by an operating system, results in very compact binary images. The system can be optimized as a whole without including unnecessary functionalities. And the static evaluation of linked configurations helps to eliminate dead code segments. This compile time specialization is also a measure of security, especially effective in the combination with the isolating hypervisor and possibly, as it is the case for the work of Madhavapeddy et al. [7], type-safe languages.

A special protection mechanism made possible by the design of unikernels is *sealing*. When the appliance starts up, it allocates all the memory it needs in a set of page tables with the policy that no page is writable and executable. After this a call to the hypervisor is used to seal these pages, which in turn makes the heap size fixed. The hypervisor has to be

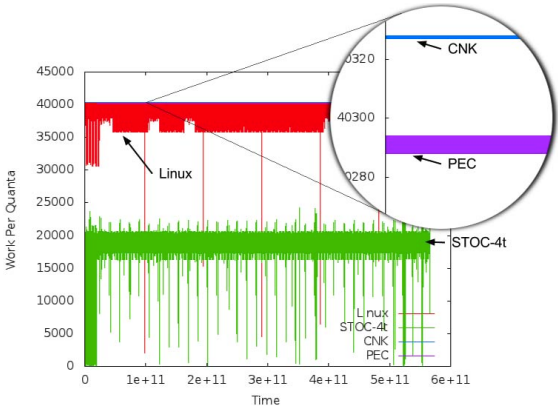


Figure 1: Fixed Time Quanta benchmark for Linux and FusedOS; adapted from Park et al. [9].

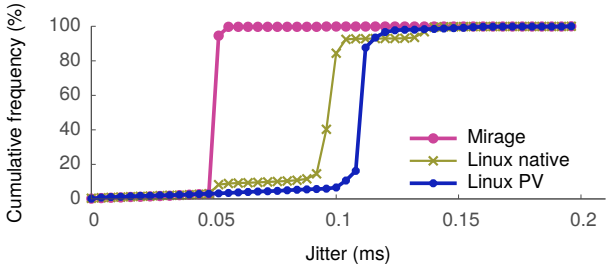


Figure 2: Figure 2: CDF of jitter for 10^6 parallel threads in Linux and Mirage OS; adapted from Madhavapeddy et al. [7].

modified in order to provide the sealing mechanism. This action makes all code injection attacks ineffective. The use of this technique is optional. The second security mechanism for unikernels is possible because most of the time a reconfiguration requires recompilation. Therefore the address space layout can be randomized at compile time [7].

Madhavapeddy et al. [7] show with their *Mirage* prototype that CPU-heavy applications are not affected by the virtualization "as the hypervisor architecture only affects memory and I/O." [7]

According to Briggs et al. [1] the performance of Mirage OS, the prototype of Madhavapeddy et al. [7], is not easily evaluated as e.g. the DNS server as well as the HTTP server are still example skeletons. They are missing important features or are unstable. But the evaluation of the Mirage DNS server nevertheless showed that it is able to handle much higher request rates than a regularly used one on Linux. Mirage OS might need some more time to mature but shows other advantageous results, such as lower and also more predictable latency which can be seen in Figure 2 [7].

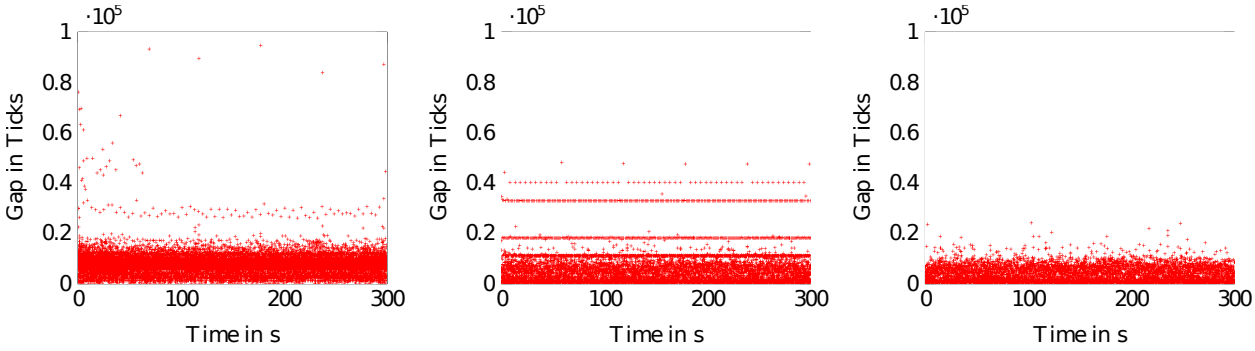
In conclusion, the drawback of the Mirage prototype is that it only runs specifically ported applications written in OCaml, like the system itself. But by this design it provides "type-safety and static analysis at compile-time." [1]

4 Hermit Core

Hermit Core is the combination of the approaches we have seen above. It combines a Linux kernel with a unikernel and promises maximum performance and scalability. Common interfaces and non-performance critical tasks are realized by Linux [6]. But as this project is focused on HPC programming models (e.g. MPI, OpenMP), performance has been improved in exchange for full POSIX compliance.

Hermit Core is extremely versatile. It can be run as a heterogeneous kernel, standalone like a pure unikernel, in a virtualized environment as well as directly on hardware as single- or multi-kernel. It can be booted without a Linux kernel directly by virtualization proxies, but in multi-kernel mode a special loader is required.

When running as a multi-kernel, one instance of *Hermit Core* runs on each NUMA node abstracting this fact so the application is presented a traditional UMA architecture.



(a) Linux (b) Hermit Core (c) Hermit Core without IP thread

Figure 3: Scatter plots observing OS noise in different configurations adapted from Lankes et al. [6].

The communication between the kernel instances are realized either by using a virtual IP device or via a message passing library. Also the communication with the Linux kernel in the heterogeneous setup happens over an IP connection as well.

Just as in the unikernel design previously seen, some parameters, like the number of system threads, are set at compile time. As a result internal data structures can be built as static arrays, which provides fast accesses and good cache-usage. For tasks like garbage collectors in the runtime of managed languages it is still necessary for *Hermit Core* to provide a scheduler. Yet the scheduling overhead is reduced by the fact that the *Hermit Core* kernel does not interrupt computation threads. Computation threads run on certain cores and are not using a timer which would be a source for OS noise.

For building applications a slightly modified version of the *GNU binutils* and *gcc* is used with the build target *x86_64-hermit*. By this design, hermit core applications can be built in any language supported by *gcc*. But it is even possible to use a different C-compiler by including the special *Hermit Core* header files instead of the Linux ones.

Figure 3 shows scatter plots from the *Hourglass* benchmark. The gaps in the execution time are used to indicate points in time where the operating system took time from an application process for maintenance tasks.

In comparison to a standard Linux, which can be seen in Figure 3a, the *Hermit Core* with a networking thread, seen in Figure 3b, shows significantly smaller gaps than the Linux. But *Hermit Core* is designed to spawn only one thread for handling IP packets, therefore all computation threads run with a profile similar to Figure 3c which shows the smallest noise distribution. Lankes et al. [6] also show that their prototype is approximately twice as fast, with regard to basic system services on the Intel Haswell architecture, as Linux.

We have seen *Hermit Core*, which is a versatile, well performing symbiosis of Linux and unikernels designed for HPC. The benchmark results still show OS noise to be present, but on a much smaller scale than on Linux.

5 Comparison

As the *mOS* project is still in a prototype phase, and the *Interface for Heterogeneous Kernels* as well is in an early stage, the stability of the projects has still to show as they both progress. As the concept of library operating systems is much older, the systems investigated in this work seem to be stable yet are facing different issues.

The development of applications to run on a heterogeneous kernel should be not much different than for traditional Full-Weight-Kernel systems as both projects presented set Linux compatibility as one of their goals. Additionally the symbiotic system presents itself as a single system to applications. Even with a stable exokernel already present, the implementation of an application together with a specialized OS suited for it involves much more manual work than with the heterogeneous kernel concept. Virtualization helps to cope with hardware abstractions for the exokernel, but the libOS has to be written for every application. *Hermit Core* provides the combination of the aforementioned concepts by uniting a libOS in form of a unikernel with a Full-Weight-Kernel. It makes it possible to use high-level languages for application development and provides an adapted software collection with familiar build tools.

If we take the performance evaluation of *FusedOS* into account, as *mOS* as well as IHK are not ready yet for macro-benchmarks, the Light-Weight-Kernels run with much less jitter and deterministic behavior. Results of the unikernel prototype *Mirage OS* benchmarks are all at the micro-level as most applications for it are still stubs or have limited functionality. Yet this approach as well shows a clear reduction in jitter and more predictable behavior can be expected. As can be seen in Figure 3, *Hermit Core* offers a considerable reduction in OS noise. Additionally the performance for basic system services and scheduling has been shown to be higher compared to Linux.

6 Conclusion

On the way to exascale computing there is a need for new concepts in HPC OS design in order to make full use of the hardware. One step in this direction is the elimination of jitter.

In this paper we introduced the two currently most popular concepts for new operating system designs focusing on high performance computing. To sum up, both approaches show great promise for performance and the reduction of OS noise. Even their combination is possible and the so constructed prototype system performs equally well. Yet all projects are missing comprehensive evaluation results as a consequence of their youth. Heterogeneous kernel concepts seem to have high potential yet are not mature enough to be considered at the moment. Application development should be straightforward and compatibility with already existing ones should be provided. The concept of library operating systems has been around for a long time and it might be a good option if the performance boost compensates the cost for virtualization layers, but more manual work is involved in order to write a tailored libOS in addition to the desired application. A combination of both concepts is possible and seems to have excellent properties.

References

[1] Ian Briggs, Matt Day, Yuankai Guo, Peter Marheine, and Eric Eide. A performance evaluation of unikernels. 2015.

[2] Dawson R Engler, M Frans Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*, volume 29. ACM, 1995.

[3] Kurt B Ferreira, Patrick G Bridges, Ron Brightwell, and Kevin T Pedretti. The impact of system design parameters on application noise sensitivity. *Cluster computing*, 16 (1):117–129, 2013.

[4] Balazs Gerofi, Masamichi Takagi, Yutaka Ishikawa, Rolf Riesen, Evan Powers, and Robert W Wisniewski. Exploring the design space of combining linux with lightweight kernels for extreme scale computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, page 5. ACM, 2015.

[5] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.12. URL `https://doi.org/10.1109/SC.2010.12`.

[6] Stefan Lankes, Simon Pickartz, and Jens Breitbart. Hermitcore: A unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '16, pages 4:1–4:8, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4387-9. doi: 10.1145/2931088.2931093. URL `http://doi.acm.org/10.1145/2931088.2931093`.

[7] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, volume 48, pages 461–472. ACM, 2013.

[8] José Moreira, Michael Brutman, Jose Castano, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, et al. Designing a highly-scalable operating system: The blue gene/l story. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 53–53. IEEE, 2006.

[9] Yoonho Park, Eric Van Hensbergen, Marius Hillenbrand, Todd Inglett, Bryan Rosenburg, Kyung Dong Ryu, and Robert W Wisniewski. Fusedos: Fusing lwk performance with fwk functionality in a heterogeneous environment. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 211–218. IEEE, 2012.

[10] Taku Shimosawa, Balazs Gerofi, Masamichi Takagi, Gou Nakamura, Tomoki Shirasawa, Yuji Saeki, Masaaki Shimizu, Atsushi Hori, and Yutaka Ishikawa. Interface for heterogeneous kernels: A framework to enable hybrid os designs targeting high performance computing on manycore architectures. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10. IEEE, 2014.

[11] Robert W Wisniewski, Todd Inglett, Pardo Keppel, Ravi Murty, and Rolf Riesen. mos: An architecture for extreme-scale operating systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, page 2. ACM, 2014.

Master-Seminar: Hochleistungsrechner - Aktuelle Trends und Entwicklungen
Winter Term 2017/2018

Fault Tolerance in High Performance Computing

Simon Klotz
Technische Universität München

Abstract

Fault tolerance is one of the essential characteristics of High-Performance Computing (HPC) systems to reach exascale performance. Higher failure rates are anticipated for future systems because of an increasing number of components. Furthermore, new issues such as silent data corruption arise. There has been significant progress in the field of fault tolerance in the past few years, but the resilience challenge is still not solved. The most widely used global rollback-recovery method does not scale well enough for exascale systems which is why other approaches need to be considered. This paper gives an overview of established and emerging fault tolerance methods such as rollback-recovery, forward recovery, algorithm-based fault tolerance (ABFT), replication and fault prediction. It also describes how LAIK can be utilized to achieve fault tolerant HPC systems.

1 Introduction

Fault tolerance is the ability of an HPC system to avoid failures despite the presence of faults. Past HPC systems did not provide any fault tolerance mechanisms since faults were considered rare events [1]. However, with the current and next generation of large-scale HPC systems faults will become the norm. Due to power and cooling constraints, the increase in clock speed is limited, and it is expected that the number of components per system will continue to grow to improve performance [2] following the trend of the current Top500 HPC systems [3]. A higher number of components makes it more likely that one of them fails at a given time. Furthermore, the decreasing size of transistors increases their vulnerability.

The resilience challenge [4] encompasses all issues associated with an increasing number of faults and how to deal with them. Commonly used methods such as coordinated checkpointing do not scale well enough to be suited for exascale environments. Thus, the research community is considering other approaches such as ABFT, fault prediction or replication to provide a scalable solution for future HPC systems. This paper presents existing and current research in the field of fault tolerance in HPC and discusses their advantages and drawbacks.

The remainder of this paper is structured as follows: Section 2 introduces the terminology of fault tolerance. Fault recovery methods such as rollback-recovery, forward recovery and silent error detection are covered in Section 3. Section 4 discusses fault containment methods such as ABFT and replication. Section 5 is concerned with fault avoidance methods including failure prediction and migration. Finally, Section 6 presents LAIK, an index space management library, that enables all three classes of fault tolerance.

 DOI: 10.14459/2018md1430700

2 Background

First, essential concepts of the field have to be introduced for an efficient discussion of current fault tolerance methods. This paper relies on the definitions of Avizienis [5], who defines faults as causes of errors which are deviations from the correct total state of a system. Errors can lead to failures which imply that the external and observable state of a system is incorrect. Faults can be either active or dormant whereas only active faults lead to an error. However, dormant faults can turn into active faults by the computation or external factors. Furthermore, faults can be classified into permanent and transient faults. Transient faults only appear for a certain amount of time and then disappear without external intervention in contrast to permanent faults which do not disappear. Errors can be classified as detected, latent or masked. Latent errors are only detected after a certain amount of time and masked errors do not lead to failures at all. Errors that lead to a failure are also called fail-stop errors whereas undetected latent errors that only corrupt the data are called silent errors. Silent errors have been identified to be one of the greatest challenges on the way to exascale resilience [1]. As an example, cosmic radiation impacting memory and leading to a discharge would be a fault. Then, the affected bits flip which would be an error. If an application now uses these corrupted bits the observable state would deviate from the correct state which is a failure.

An important metric to discuss fault tolerance mechanisms is the Mean Time Between Failures (MTBF). This paper is following the definition of Snir et al. [6] who define it as

$$MTBF = \frac{Total\ runtime}{Number\ of\ failures}$$

Several studies [7, 8] were conducted to classify failures and determine their source in HPC systems. In order to analyze failures researchers define different failure classes based on their cause. Schroeder et al. [7] distinguish between hardware, software, network, environment, human, and undetermined failures whereas Oliner et al. [8] only distinguish between software, hardware, and undetermined failures. Depending on the studied system the occurence of different failure classes differ significantly. A possible explanation is the discrepancy in system characteristics and used failure classes. Nevertheless, most researchers agree that software and hardware faults are the most frequent causes of failures [7]. Hardware faults can occur in any component of an HPC system, such as fans, CPUs, or memory. One of the most important classes of hardware faults are memory bit errors due to cosmic radiation. Software faults can occur at any part of the software stack. The most common faults include unhandled exceptions, null reference pointers, out of memory error, timeouts and buffer overflows [6]. Since hardware and software faults are most common in HPC systems, the remainder of this paper will focus on these two classes since also most fault tolerance methods apply to them.

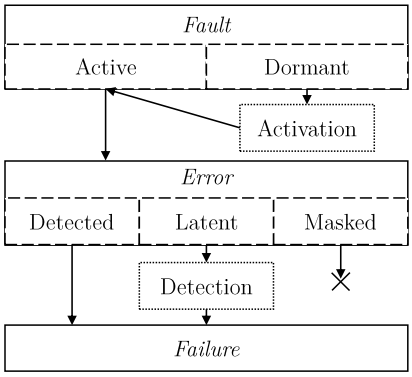


Figure 1: Classification of faults, errors, and failures

3 Fault Recovery

The objective of fault recovery is to recover a normal state after the application experienced a fault, thus preventing a complete application failure. Fault recovery methods can be divided into rollback-recovery, forward recovery, and silent error detection. Rollback-recovery is the most widely investigated approach and commonly used in many

large-scale HPC systems. Rollback-recovery methods periodically take snapshots of the application state and in case of failure the execution reverts to that state. Forward recovery, avoids this restart by letting the application recover by itself using special algorithms. A novel research direction is silent error detection which can be combined with rollback-recovery and forward recovery to recover from faults.

3.1 Rollback-recovery

Rollback-recovery, which is also called checkpoint-restart, is the most widely used and investigated fault tolerance method [9, 10]. Checkpointing methods can be classified into multiple categories according to how they coordinate checkpoints and at which level of the software stack they operate. Each class has different approaches trying to solve the challenge of saving a consistent state of a distributed system which is not a trivial task to achieve since large-scale HPC systems generally do not employ any locking mechanisms. The two major approaches to checkpoint a consistent state of a HPC system are coordinated checkpointing and uncoordinated checkpointing with message-logging [10] as depicted in Figure 2.

Coordinated protocols store checkpoints of all processes simultaneously which imposes a high load on the file system. They enforce a synchronized state of the system such that no messages are in transit during the checkpointing process by flushing all remaining messages. The advantage of coordinated checkpointing is that each checkpoint captures a consistent global state which simplifies the recovery of the whole application. Furthermore, coordinated protocols have a comparatively low overhead since they do not store messages between processes [9]. However, if one process fails all other processes need to be restarted as well [11]. Also, with a growing number of components and parallel tasks the cost of storing checkpoints in a coordinated way steadily increases.

Message-logging protocols store messages between nodes with a timestamp and the associated content, thus capturing the state of the network [9]. They are based on the assumption that the application is piecewise deterministic and that the state of a process can be recovered by replaying the recorded messages from its initial state [9]. In practice, each process periodically checkpoints its state so that recovery can continue from there instead of the initial state [12]. In contrast to coordinated protocols, only the processes affected by failure need to be restarted which makes it more scalable. However, each process needs to maintain multiple checkpoints which increases memory consumption and inter-process dependencies can lead to the so-called Domino effect [9]. Furthermore, all messages and their content need to be stored as well.

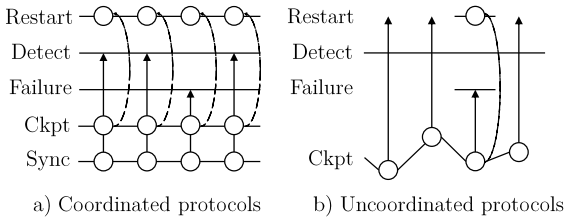


Figure 2: Difference of coordinated protocols and uncoordinated protocols with message-logging [11]

Message-logging protocols can be further divided into: optimistic, pessimistic and causal protocols [13]. Optimistic protocols store checkpoints on unreliable media, e.g. the node running a process itself. Furthermore, a process can continue its execution during logging [11]. This method has less overhead compared to others but can only tolerate a single failure at a time and if a node itself fails the whole application has to be restarted. Pessimistic protocols store all checkpoints on reliable media and processes have to halt execution until messages are logged. This enables them to tolerate multiple faults and complete node failures at the cost of a higher overhead using reliable media and halting execution. Causal protocols also do not use reliable media but log the messages in neighboring nodes with additional causality information. This omits the need for reliable media and an application is still able to recover if one node experiences a complete failure.

Checkpointing methods can operate on different levels of the software stack which has an important impact on transparency and efficiency. Checkpointing can be implemented by the operating system, a library, or the application itself [14]. Operating system extensions can do global checkpoints that provide transparent checkpointing of the whole application state and do not require modifications of the application code. There are also libraries that act as a layer on top of MPI and enable global and transparent checkpointing. However, global checkpointing has a large overhead since the storage costs are proportional to the memory footprint of the system [15] which often includes transient data or temporary variables which are not critical for recovery. Approaches such as compiler-based checkpointing intend to solve this issue by automatically detecting transient variables and excluding them from checkpoints which reduces the total size of checkpoints.
All of the previously mentioned approaches automatically store checkpoints which does not allow any control of the procedure. User-level fault tolerance libraries assume that developers know best when to checkpoint and what data to store which can significantly decrease the checkpoint size. However, this also increases application complexity [15] and an application developer can be mistaken and forget about critical data.
Researchers introduced different variations and additions to the general checkpointing techniques to reduce the number of needed resources to reach feasible performance for exascale HPC systems. One of these approaches is diskless checkpointing [16] which stores checkpoints in memory. Other approaches include multi-level checkpointing [17] which hierarchically combines multiple storage technologies with different reliability and speed allowing a significantly higher checkpointing frequency since checkpointing on the first layer is generally fast.

3.2 Forward recovery

Forward recovery avoids restarting the application from a stored checkpoint and lets the application recover by itself. It is only applicable for algorithms that can accept faults but ultimately converge to a correct state, sometimes at the cost of additional computations. Forward recovery algorithms are non-masking since they can display erratic behavior due to transient faults but finally display legitimate behavior again after a stabilization period [18]. One class of algorithms having this property are self-stabilizing algorithms.
Forward recovery can also be manually implemented by application developers. Resilient MPI implementations provide an interface which can be used by application developers to specify behavior in case a fault occurs. Using this, developers can implement custom forward recovery routines that recover a consistent, fault-free application state. The advantage of forward recovery methods is that they often allow faster recovery compared to rollback-recovery. However, they are application specific and not applicable to all algorithms.

3.3 Silent Error Detection

The objective of silent error detection methods is to detect latent errors such as multi-bit faults. Silent error detection is a relatively novel research area of increasing importance since smaller components are more vulnerable to cosmic radiation [1]. It is application specific and uses historic application data to detect anomalous results during the computations of an application which indicates a silent data corruption (SDC).
Several methods can be used to detect anomalies in the application data. Vishnu et al. [19] use different machine learning methods able to correctly detect multi-bit errors in memory with a 97% chance. After a silent error is detected it has to be corrected. Gomez et al. [20] just replace corrupted values with their predictions. Since it is not always desirable to use approximate predictions, rollback-recovery can be used, to revert to a consistent application state instead. Application developers can also implement application-specific routines to recover a normal state.
One drawback of silent error detection is that it introduces computational overhead. Another limita-

tion is the false positive rate which has a significant impact on the overall performance of an application. Furthermore, it takes significant effort to develop silent error detection methods since they are application specific. Nevertheless, silent error detection is one of the few methods able to deal with silent errors and has less resource overhead compared to replication [21].

4 Fault Containment

Fault containment methods let an application continue to execute until it terminates even after multiple faults occur. Compensation mechanisms are in place to contain the fault and avoid an application failure. They also ensure that the application returns correct results after termination. The most commonly used fault containment methods are replication and ABFT. Replication systems run copies of a task on different nodes in parallel and if one fails others can take over providing the correct result. ABFT includes all algorithms which can detect and correct faults, that occur during computations.

4.1 Replication

Replication methods replicate all computations on multiple nodes. If one replica is hit by a fault the others can continue computation and the application terminates as expected.
Replication methods can be distinguished into methods that either deal with fail-stop or silent errors. MR-MPI [22] deals with fail-stop errors and uses replicas for each node. As soon as one node fails the replica is taking over and continues execution. Only if both nodes fail the application needs to be restarted. In contrast, RedMPI [23] is not addressing complete node failures but silent errors. It compares the output of all replicas to correct SDCs. A novel research direction is to use approximate replication to detect silent errors. A complementary node is running an approximate computation which is then compared to the actual result to detect SDCs. Benson et al. [24] use cheap numerical computations to verify the original results which reduces the overhead compared to other replication methods.
Replication generally has a high overhead because multiples of all resources are needed depending on the replication factor. Furthermore, it is costly to check the consistency of all replications and it is difficult to manage non-deterministic computations [25]. Also, an overhead of messages is needed to communicate with all replicas. Nevertheless, Ferreira et al. [25] showed that replication methods outperform traditional checkpointing methods if the MTBF is low.

4.2 ABFT

ABFT was first proposed by Huang and Abraham in 1984 [26] and includes algorithms able to automatically detect and correct errors. ABFT is not applicable to all algorithms but researchers are continuously adapting new algorithms to be fault tolerant because of the generally low overhead compared to other methods. The core principle of ABFT is to introduce redundant information which can later be used to correct computations.
ABFT algorithms can be distinguished into online and offline methods [11]. Offline methods correct errors after the computation is finished whereas online methods repair errors already during the computation. The most prominent examples for offline ABFT are matrix operations [27] where checksum columns are calculated which are later used to verify and correct the result. Online ABFT can also be applied to matrix operations [28] whereas they are often more efficient than their offline counterparts since they can correct faults in a timely manner, when their impact is still small, compared to offline ABFT methods that correct faults only after termination.
The main drawback of ABFT is that it requires modifications of the application code. However, since most applications rely on low-level algebra libraries the methods can be directly implemented in those libraries hidden from an application programmer. Another disadvantage of ABFT is that it cannot compensate for all kinds of faults, e.g.

complete node failures. Furthermore, if the number of faults during the computation exceeds the capabilities of the algorithm it cannot correct them anymore. ABFT also introduces overhead in the computations which is however comparatively small to other fault tolerance methods.

5 Fault Avoidance

In contrast to fault recovery and fault containment, fault avoidance or proactive fault tolerance methods aim at preventing faults before they occur. Generally, fault avoidance methods can be split in two phases. First, predicting the time, location and kind of upcoming faults. Second, taking preventive measures to avoid the fault and continue execution. Proactive fault tolerance is a comparatively new but promising field of research [1]. The prediction of fault occurrences mainly relies on techniques from Data Mining, Statistics and Machine Learning which use RAS log files and sensor data. The most common preventive measure is to migrate tasks running on components which are about to fail to healthy components.

5.1 Fault Prediction

It is important to consider which data and model to use to accurately predict faults. Engelmann et al. [29] derived four different types of health monitoring data that can be used to predict faults whereas most prediction methods rely on Type 4 which utilizes a historical database that is used as training data for machine learning methods. It mostly includes sensor data such as fan speed, CPU temperature and SMART metrics but also RAS logfiles containing failure histories.
The success of fault avoidance methods mainly depends on the accuracy of the predictions. It is important that predictors can predict both location and time of a fault in order to successfully migrate a task [30]. Furthermore, the accuracy of the prediction is of high importance. If too many faults are missed the advantage of a prediction-based model is small compared to other approaches such as rollback-recovery. On the other hand, if the predictor detects too many false positives too much time is spent on premature preventive measures. Models used for fault prediction mainly include Machine Learning models [31] and statistical models [32].

5.2 Migration

Once a fault is predicted measures have to be taken to prevent it from happening. The most common approaches for prevention are virtualization-based migration and process migration which both move tasks from failing nodes to healthy ones. Both approaches can be divided into stop&copy and live migration [33]. Stop&copy methods halt the execution as long as it takes to copy the data to a target node which depends on the memory footprint, whereas live migration methods continue execution during the copy process reducing the overall downtime.
Both virtualization and process-based methods need available nodes for migration. They can either be designated spare nodes, which are expected to become a commodity in future HCP systems, unused nodes, or the node with the lowest load [33]. Two tasks sharing the same node can however result in imbalance and an overall worse performance. Most approaches generally use free available nodes first, then spare nodes, and as a last option the least busy node [33].
Nagarajan et al. [34] use Xen-virtualization-based migration while there approach is generally transferrable to other virtualization techniques as well. Each node runs a host virtual machine (VM) which is responsible for resource management and contains one or more VMs running the actual application. Xen supports live migration of VMs [35] which allows MPI applications to continue to execute during migration.
Process-based migration [36] has been widely studied on several operating systems. It generally follows the same steps as virtualization-based migration but instead of copying the state of a VM it copies the process memory [33]. Process-based migration also supports live migration.

Applications do not need to be modified for both process-based and VM-based migration. Virtualization offers several advantages such as increased security, customized operating system, live migration, and isolation. Furthermore, VMs can be more easily started and stopped compared to real machines [37]. Despite of these advantages virtualization-based techniques have not been widely adopted in the HPC community because of their larger overhead.
Fault avoidance cannot fully replace other approaches such as rollback-recovery since not all faults are predictable. Nevertheless, fault avoidance can be combined with other approaches such as ABFT, replication and rollback-recovery. Since it can significantly increase the MTBF it allows an application to take less checkpoints which is shown by Aupy et al. [38] who study the impact of fault prediction and migration on checkpointing.

6 LAIK

LAIK (Leichtgewichtige Anwendungs-Integrierte Datenhaltungs Komponente) [39] is a library currently under development at Technical University Munich which provides lightweight index space management and load balancing for HPC applications supporting application-integrated fault tolerance. Its main objectives are to hide the complexity of index space management and load balancing from application developers, increasing the reliability of HPC applications.
LAIK decouples data decomposition from the application code so that applications can adapt to a changing number of nodes, computational resources and computational requirements. The developer needs to assign application data to data containers using index spaces. Whenever required, LAIK can trigger a partitioning algorithm which distributes the data across nodes where the respective data container is needed. This allows higher flexibility in regard to fault tolerance and scheduling mechanisms. It also avoids application specific code which can become increasingly complex and hard to maintain. Furthermore, LAIK is designed to be portable by defining an interface which supports multiple communication backends and allowing application developers to incrementally port their application to LAIK. Figure 3 shows an overview of how LAIK is integrated in HPC systems.

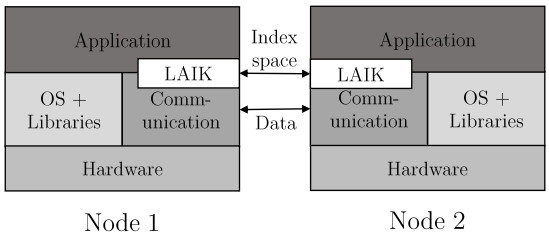


Figure 3: Integration of LAIK in HPC systems [39]

The capabilities of LAIK can be utilized for application-integrated fault tolerance. It is possible to implement fault recovery, fault containment and fault avoidance methods as layers on top of LAIK using its index space management and partitioning features.
LAIK can be used for local checkpointing by maintaining copies of relevant data structures on neighboring nodes [39]. Once a node failure is detected, the application can revert to the checkpointed state of these copies and resume execution from there on a spare node using LAIKs partitioning feature.
Furthermore, it is possible to implement replication in a similar way as checkpointing using LAIK. Again multiple redundant copies of the data containers are maintained on different nodes whereas one node acts as the master node. Instead of only using the redundant data after a failure as in checkpointing, tasks can be run in parallel on each replicated data container. Additional synchronization mechanisms have to be utilized to keep the replicas in a consistent state. In case the master node experiences failure one of the replicas becomes the new master node and execution continues without interruption.
LAIK also allows proactive fault tolerance. Once a prediction algorithm predicts a failure the nodes are synchronized, execution is stopped, and the failing node is excluded. Then, LAIK triggers the repartitioning algorithm and execution can be continued

on a different node preventing a complete application failure.

7 Conclusion

Fault tolerance methods aim at preventing failures by enabling applications to successfully terminate in the presence of faults. Several distinct fault tolerance methods were developed to solve the resilience challenge. Since the most common method global rollback-recovery is not feasible for future systems new methods have to be considered. However, currently no other method is able to fully replace rollback-recovery. Nevertheless, methods such as user-level fault tolerance, multi-level checkpointing and diskless checkpointing aim at increasing the efficiency of checkpointing. Also, additional methods such as replication, failure prediction, and process migration can be used complementary to rollback-recovery to decrease the MTBF and application specific methods such as ABFT, forward recovery and silent error detection can further improve the fault tolerance.
It is challenging to use the various existing fault tolerance libraries complementary since they rely on different technologies. In order to reach exascale resilience an integrated approach, which is able to combine fault tolerance methods, is of high importance. LAIK has the capabilities to act as a basis for such an approach. Researchers could build layers on top of LAIK implementing new methods in a standardized and compatible way utilizing LAIKs features. Furthermore, additional partitioning algorithms, data structures and communication backends could be implemented because of LAIKs modular architecture. This would enable a complementary use of multiple fault tolerance methods to significantly increase the overall fault tolerance of HPC systems.

References

[1] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[2] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *International Conference on High Performance Computing for Computational Science*, pp. 1–25, Springer, 2010.

[3] E. Strohmaier, "Top500 supercomputer," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, (New York, NY, USA), ACM, 2006.

[4] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[6] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[7] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," in *Journal of Physics: Conference Series*, vol. 78, p. 012022, IOP Publishing, 2007.

[8] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pp. 575–584, IEEE, 2007.

[9] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery

protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[10] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni, "Unified model for assessing checkpointing protocols at extreme-scale," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 17, pp. 2772–2791, 2014.

[11] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *The International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, 2009.

[12] S. Rao, L. Alvisi, and H. M. Vin, "The cost of recovery in message logging protocols," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 160–173, 2000.

[13] L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, causal, and optimal," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 149–159, 1998.

[14] E. Roman, "A survey of checkpoint/restart implementations," in *Lawrence Berkeley National Laboratory, Tech*, Citeseer, 2002.

[15] J. Dongarra, T. Herault, and Y. Robert, "Fault tolerance techniques for high-performance computing," in *Fault-Tolerance Techniques for High-Performance Computing*, pp. 3–85, Springer, 2015.

[16] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, 1998.

[17] N. H. Vaidya, "A case for two-level distributed recovery schemes," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 23, pp. 64–73, ACM, 1995.

[18] S. Tixeuil, "Self-stabilizing algorithms," in *Algorithms and theory of computation handbook*, pp. 26–26, Chapman & Hall/CRC, 2010.

[19] A. Vishnu, H. van Dam, N. R. Tallent, D. J. Kerbyson, and A. Hoisie, "Fault modeling of extreme scale applications using machine learning," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pp. 222–231, IEEE, 2016.

[20] L. A. B. Gomez and F. Cappello, "Detecting and correcting data corruption in stencil applications through multivariate interpolation," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pp. 595–602, IEEE, 2015.

[21] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello, "Lightweight silent data corruption detection based on runtime data analysis for hpc applications," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 275–278, ACM, 2015.

[22] C. Engelmann and S. Böhm, "Redundant execution of hpc applications with mr-mpi," in *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, pp. 15–17, 2011.

[23] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 78, IEEE Computer Society Press, 2012.

[24] A. R. Benson, S. Schmit, and R. Schreiber, "Silent error detection in numerical time-stepping schemes," *The International Journal of High Performance Computing Applications*, vol. 29, no. 4, pp. 403–421, 2015.

[25] K. Ferreira, J. Stearley, J. H. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pp. 1–12, IEEE, 2011.

[26] K.-H. Huang *et al.*, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[27] Z. Chen and J. Dongarra, "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10–pp, IEEE, 2006.

[28] P. Wu, C. Ding, L. Chen, F. Gao, T. Davies, C. Karlsson, and Z. Chen, "Fault tolerant matrix-matrix multiplication: correcting soft errors on-line," in *Proceedings of the second workshop on Scalable algorithms for large-scale systems*, pp. 25–28, ACM, 2011.

[29] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, "Proactive fault tolerance using preemptive migration," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pp. 252–257, IEEE, 2009.

[30] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into hpc systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 77, IEEE Computer Society Press, 2012.

[31] E. W. Fulp, G. A. Fink, and J. N. Haack, "Predicting computer system failures using support vector machines.," *WASL*, vol. 8, pp. 5–5, 2008.

[32] L. Yu, Z. Zheng, Z. Lan, and S. Coghlan, "Practical online failure prediction for blue gene/p: Period-based vs event-driven," in *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pp. 259–264, IEEE, 2011.

[33] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration and back migration in hpc environments," *Journal of Parallel and Distributed Computing*, vol. 72, no. 2, pp. 254–267, 2012.

[34] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive fault tolerance for hpc with xen virtualization," in *Proceedings of the 21st annual international conference on Supercomputing*, pp. 23–32, ACM, 2007.

[35] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273–286, USENIX Association, 2005.

[36] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Computing Surveys (CSUR)*, vol. 32, no. 3, pp. 241–299, 2000.

[37] W. Huang, J. Liu, B. Abali, and D. K. Panda, "A case for high performance computing with virtual machines," in *Proceedings of the 20th annual international conference on Supercomputing*, pp. 125–134, ACM, 2006.

[38] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, "Checkpointing algorithms and fault prediction," *Journal of Parallel and Distributed Computing*, vol. 74, no. 2, pp. 2048–2064, 2014.

[39] J. Weidendorfer, D. Yang, and C. Trinitis, "Laik: A library for fault tolerant distribution of global data for parallel applications," in *Konferenzband des PARS'17 Workshops*, p. 10, 2017.

Hochleistungsrechner: Aktuelle Trends und Entwicklungen
Wintersemester 2017/2018

Zen Mikroarchitektur

Thomas Ledwon
Ludwig-Maximilians-Universität München

Zusammenfassung

Thema dieser Arbeit ist Zen, die neueste Entwicklung von AMD auf dem Gebiet der Mikroprozessor-Architektur. Dabei wird auf Erneuerungen gegenüber der Vorgängerversion Bulldozer beziehungsweise auf die darauf aufbauenden Architekturen eingegangen. Den neuesten Prozessoren von Intel, dem Marktführer im Consumer- und HPC-Bereich, wird der AMD Ryzen 7 als neueste Implementierung der Zen Architektur gegenübergestellt. Anhand des VASP-Codes [9], der im HPC Bereich als Performanceindikator angesehen werden kann, werden Intel-Prozessoren und das Flaggschiff von AMD Ryzen 1800X in Bezug auf die beiden Leistungsindikatoren Performance und Energieeffizienz miteinander verglichen. Es zeigt sich, dass AMD mit der Zen Prozessorenfamilie seine Fähigkeit bewiesen hat, weiterhin Prozessoren fertigen zu können, die die Konkurrenz mit Intel nicht zu scheuen brauchen.

1 Einführung

Im Jahr 2011 brachte AMD auf Basis der Bulldozer-Mikroarchitektur die FX-Prozessorenfamilie mit den Modellen FX8150, FX8120 und FX6100 auf den Markt. Einige Jahre später wurden diese durch die Modelle FX8350, FX8320 und FX6300 ersetzt. Diese waren aber im Vergleich zu den Intel Prozessoren in den Bereichen Instruktionen, Optionen sowie IPC unterlegen [13]. Zen, die folgende Mikroarchitekturgeneration wurde von Grund auf neu entworfen. Die darauf basierenden CPUs heißen Ryzen beziehungsweise EPYC für den Serverbereich [6]. Die EPYC Prozessoren bieten bis zu 32 Kerne mit 64 Threads und 2 TB an DDR4-Speicherkapazität über 8 Kanäle. Produkte mit diesen Prozessoren kommen Ende des Jahres 2017 auf den Markt. [2] Ziel von AMD ist es, im Vergleich zu Intel in punkto Performance konkurrenzfähig zu werden und Marktanteile zurück zu gewinnen [8]. Am 2. März 2017 wurden die ersten Hardwarereviews zu AMDs neuem Flaggschiff-Prozessor Ryzen 7 1800X online gestellt. Die AMD Ryzen Familie besteht aus drei Typen, dem AMD Ryzen 3 im Einsteigersegment, AMD Ryzen 5 im Mittelklassesegment und dem AMD Ryzen 7 im hochpreisigen Segment . Vor allem im höherpreisigen Segment möchte AMD mit geringere Preise und durch höhere Rechenleistung, als vergleichbare Intel-Prozessoren aufweisen können, Druck auf den Marktführer Intel aufbauen [11].

Das Paper ist nun wie folgt gegliedert. Zuerst werden Neuerungen der Zen Mikroarchitektur gegenüber der Vorgängerversion, der Bulldozer Architektur, vorgestellt. Danach werden Unterschiede zu Intel Prozessoren der Coffe Lake Generation hervorgehoben. In dem darauffolgendem Abschnitt werden der AMD Ryzen 1800x, einige Intel Prozessoren und der IBM Power 7 CPU in den Kriterien Performance und Energiekonsum miteinander verglichen. Der letzte Abschnitt dieser Arbeit beinhaltet das Fazit.

 DOI: 10.14459/2018md1430700

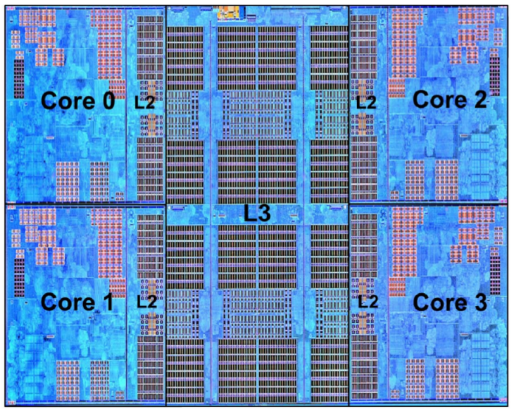


Abbildung 1: Zen Core Complex Die (Quelle [14])

2 Neuerungen in der Architektur

Das Quad-Core Complex (CCX) (Abbildung 1) Design der Zen Architektur verfolgt einen anderen Ansatz als der Vorgänger, die Bulldozer-Architektur. Die Bulldozer CPUs sowie deren Derivate (Piledriver und Excavator) setzten auf Zweikern-Module, wo jeder Kern einige Ressourcen mit dem anderen Kern teilt. Dabei hat jeder Kern eine eigenständige Integerberechnung, geteilt werden der Fetch, das Dekodieren, Floating-Point Berechnungen, und der L2-Cache [12]. Die Vierkern-Module in Zen haben eine diskrete Logik, in der von den Kernen nur der Zugang zum L3-Cache geteilt wird.

Die vierkernigen Bulldozer-Chips nutzen ein Zwei-Modul-Design, die Achtkern-Chips ein Vier-Modul-Design. Die achtkernigen AMD-Chips, z.B. der FX-8370, sind somit im Endeffekt Vierkern-CPUs. Wohingegen bei Zen jeder Kern abgesehen vom L3-Cache vollkommen autonom arbeitet [8]. Verbesserungen am internen Schaltkreis durch neue Techniken wie wordline boost [17], contention-free dynamic logic, supply drop detection mit mitigation [14], erhöhen die Performance und die Energieeffizienz [15].

Die Zen Architektur verfügt nun über Advanced Vector Extensions 2 (AVX2) [3], die von Intel mit den Haswell Chips eingeführt wurde. Diese unterstützen das parallele Ausführen von speziellen Instruktionen in 256-Bit-Vektorregistern. In den Registern können somit zum Beispiel 8 Float oder 4 Double Werte gespeichert werden [10]. AVX2 unterstützt Floating-Point-Arithmetik und Fused-Multiply-Add (FMA) mit drei Operaden, wie z.B. die Instruktion A = A * B + C. In der Performance ist AVX2 bei bestimmten 64-Bit-Operationen 4 mal schneller also vorherige Instruction Sets [5].

Eine weitere Neuerung beim AMD ist ein integrierter op-Cache. In diesem kleinen Speicher werden Instruktionen gespeichert, die zuvor schon dekodiert wurden. Jedes Mal wenn die CPU eine Instruktion benötigt, die schon in diesem Cache liegt, werden Zeit und Energie gespart, da das Fetching und das Dekodieren entfällt [6].

AMD hat unter dem Vermarktungsnamen AMD SenseMIT fünf neue Technologien eingeführt. Zu SenseMIT gehören Neural Net Prediction, Smart Prefetch, Pure Power, Precision Boost and Extended Frequency Range.

Die Neural Net Prediction erlaubt es dem Prozessor sich über, eine KI selbst zu trainieren und dadurch die benötigten Instruktionen vorzuladen, so dass diese schneller ausgeführt werden können. Wenn Net Prediction herausgefunden hat, welches die benötigten Instruktionen sind, lernt die Smart Prefetch Technologie vorherzusagen, welche Daten eine Anwendung benötigt und versucht, diese schon bereitzustellen, bevor sie benötigt werden.

Sensoren, die im Prozessor integriert sind, sind Voraussetzung für Pure Power und Precision Boost. Pure Power und Prescion Boost arbeiten Hand in Hand um den Energieverbrauch und die Frequenz im Millisekunden Bereich bestmöglich an die gegenwärtige Situation anzupassen. Dies geschieht dadurch, dass nicht benötigte Komponenten des Chips heruntergefahren oder in reduziertem Umfang ausgeführt werden. Dies geschieht auf Hardwareebene vollautomatisch.

Die Extended Frequency Range Technologie ist ähnlich wie die GPU boost Technologie von Nvidia Grafikkarten. Hier ist die Taktfrequenz nicht von fest definierten Parametern abhängig, sondern so

 DOI: 10.14459/2018md1430700

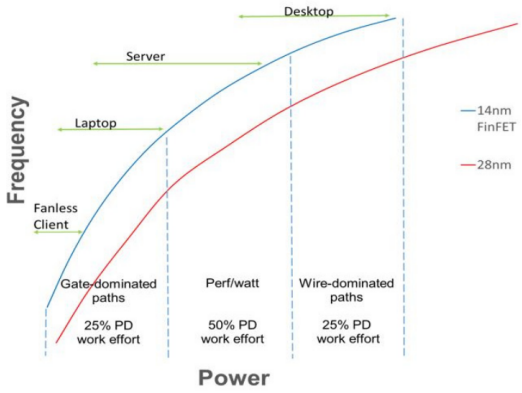


Abbildung 2: Zen CCX Optimierung für unterschiedliche Marktsegmente (Quelle [14])

gut wie komplett von der Hitzeentwicklung und der Kühllösung. SenseMIT ist somit eine automatische Übertaktungsmöglichkeit der CPU, die nur von der Kühlleistung abhängig ist und je höher deren Leistung, desto höher deren Frequenz [8].

Das Konzept der Zen Mikroarchitektur wurde im Hinblick auf Performance komplett überarbeitet. Zusammen mit einer höheren Bandbreite und einem Cachesystem mit sehr niedriger Latenz, stellt dies einen signifikanten Leistungszuwachs gegenüber der älteren Bulldozer-Architektur dar. Die Fertigungsgröße der FinFet [7] Transistoren schrumpft dabei auf 14nm (Abbildung 2). Der IPC, also die Instruktionen die pro Zyklus beendet werden, wurde hauptsächlich durch eine Verbesserung der branch prediction erhöht. Der IPC wurde im Vergleich gegenüber der Intel-Architektur als eine der Schwachstellen der Bulldozer Architektur angesehen. Zen kann nun 6 Instruktionen pro Zyklus senden, während es bei Bulldozer nur 4 Instruktionen waren. Im Cachsystem wurde die Bandbreite des L1 und L2-Cache verdoppelt und der L3-Cache verfünffacht, der L2-Cache kann 50 Prozent schneller und der L3-Cache 70 Prozent schneller angesprochen werden [6]. Dies sollte besonders bei HPC Anwendungen einen Ausschlag geben. Das SMT Multithreading wurde ausgebaut mit Kernen, die auf SMT basieren um konkurenzfähig zu Intels HyperThreading zu werden [8].
Durch all diese Verbesserungen konnte der IPC erhöht, der Stromverbrauch konstant gehalten und AMD Ryzen attraktiver für den Markt gemacht werden.

3 Unterschiede zu Intel Coffee Lake

Intel CPUs gehören zu den am meisten installierten Komponenten in HPC Systemen [16]. Intels performantester Prozessor auf Basis der Intel Coffee Lake Mikroarchitektur (Intel Generationen ab 2011: Sandy Bridge, Ivy Bridge, Haswell, Skylake, Caby Lake, Coffe Lake), der im selben Marktsegment wie der AMD Ryzen liegt, hat nur 6 Kerne im Vergleich zu AMDs Ryzen mit 8 Kernen. Wegen eines höheren IPC Durchsatzes und einer höheren Frequenz hat Intel jedoch theoretisch eine höhere Performance pro Kern. Sowohl Intels Core i7 wie auch AMD Ryzen Chips bieten Kerne, die ein oder auch zwei Threads parallel bearbeiten können. Somit hat AMD mit seinen 8 Kernen 16 virtuelle Kerne wohingegen Intel nur auf 12 virtuelle Kerne kommt.
Intels Memory Geschwindigkeit mit DDR4-2666 stimmt mit der schnellsten Memory Lösung des AMD Ryzen mit single-rank Memory und einem DIMM pro Kanal überein. Beide Prozessoren bieten Dual-Channel Speicher. Intel bietet einen höheren Speichertakt, der aber nur bei extremem Übertakten zum Tragen kommt. Die Ryzen Prozessoren haben dafür einen höheren L2 und L3-Speicher.
Intel hat eine höhere Leistungsentfaltung und eine performantere Boost Option. Intel setzt auf seine Turbo Boost Technologie, wenn gerade nicht alle Prozessorkerne benötigt werden. Dadurch hat Intel einen bis zu 1 GHz größeren Vorteil auf Basis seines Boost. (Tabelle 1) Gegenüber Ryzen, das die Übertaktungsfrequenz über alle Kerne automatisch ausführt, ist bei Intel das Übertaktungspotential höher. Der TDP ist bei den Flaggschiffen von Intel

Tabelle 1: Spezifikationen der Flagschiff Prozessoren (Quellen [3] [1])

	Intel Core i7-8700K	Intel Core i7-8700	Ryzen 7 1800X [3]
Cores/Threads	6 / 12	6 / 12	8 / 16
Base Frequency	3.7 GHz	3.2 GHz	3.6 GHz
Boost Frequency	4.7 GHz	4.6 GHz	4GHz
Memory Speed	DDR4-2666	DDR4-2666	DDR4-1866 bis DDR4-2667
Memory Controller	Dual-Channel	Dual-Channel	Dual-Channel
Unlocked Multiplier	ja	nein	ja
Cache (L2+L3)	13.5MB	13.5MB	20MB
Process	14nm	14nm	14nm
TDP	95 W	95 W	95 W

und Ryzen gleich [1].

4 Performanceunterschiede

Um die Performanceunterschiede des AMD Ryzen 1800X mit unterschiedlichen Intel Prozessoren zu vergleichen hat Vladmir Stegailov und Vyacheslav Vecher in dem Paper "Efficiency Analysis of Intel and AMD x86_64 Architectures for Ab Initio Calculations: A Case Study of VASP" Tests mit VASP Code durchgeführt. VASP steht für Vienna Ab Initio Simulation Package [9] und gehört zu den meist genutzten Anwendungen bei Berechnungen von elektronischen Strukturen mit ab initio Methoden, also nicht empirischen Methoden. Dieser Code ist für 15-20 Prozent der Rechenleistung aller weltweiten Berechnungen mit Supercomputern verantwortlich. Er ist somit ein guter Indikator zur Bestimmung der Performance von CPUs im HPC-Bereich [16].
In HPC Systemen ist heutzutage die Energieeffizienz eines der größten Probleme, das wohl auch in Zukunft so bleiben wird. Die Anstieg des Stromverbrauchs und die Entwicklung von Hitze sind die Hauptprobleme bei der Messung von Performancetests. Dadurch werden solche Messungen von Performance und Energie immer mehr beweisbasiert geführt.

In diesem Abschnitt werden nun einige Intel CPUs, der AMD Ryzen 1800X mit den Ergebnissen aus Messungen mit dem IBM Power 7 CPU aus dem "Best Practice Guide — IBM Power 775" von IBM [4] verglichen. In Tabelle 2 werden die Leistungsmerkmale der Systeme aufgeführt. Als Betriebssytem für das Testsystem kommt Ubuntu Linux zum Einsatz. VASP kommt für Intel als Version 5.4.1 mit Intel Fortran, Intel MPI und Intel MKL für BLAS, LAPACK und FFT calls. Für das AMD System VASP 5.4.1 mit gfortran ver. 6.3 mit OpenMPI, OpenBLAS und FFTW libraries.

Das VASP Modell repräsentiert einen GaAs Kristall, der aus 80 Atomen in der Superzelle zusammengesetzt ist. Der Parameter τ_{iter} dient als Parameter für die Messung der Zeit bei der Berechnung. τ_{iter} Werte dauern 10 bis 100 sec und gehören zu einem einzigen Knoten eines HPC Cluster.
Der Stromverbrauch wird bei dem Single Socket System über digitales sampling mit dem Linux apcupsd Treiber während den VASP Berechnungen gemessen. Es wird der gesamte Stromverbrauch der CPU, des Memory, des Motherboards und der PSU gemessen.

Performancemessungen mit unterschiedlichen CPUs, unterschiedlichen Frequenzen sowie unterschiedlicher Peak Performance sind problematisch durchzuführen. Aus diesem Grund wird in dem Pa-

 DOI: 10.14459/2018md1430700

Tabelle 2: Eigenschaften der Systeme für den Performancetest (Quelle [16])

	N_{cores}	$N_{mem.ch}$	L3 (MB)	CPU_{freq} (GHZ)	$DRAM_{freq}$ (MHZ)
Single socket, Intel X99 chipset					
Xeon E5-2620v4	8	4	20	2.1	2133
Core i7-6900K	8	4	20	2.1 - 3.2	2133-3200
Xeon E5-2660v4	14	4	35	2.0	2400
Single socket, AMD B350 chipset					
Ryzen 1800X	8	2	16	3.6	2133 - 2400
Dual socket, Intel C602 chipset (the MVS10P cluster					
Xeon E5-2690	8	4	20	2.9	1600
Dual socket, Intel c612 chipset (the MVS1P5 cluster					
Xeon E5-2697v3	14	4	35	2.6	2133
Dual socket, Intel C612 chipset (the IRUS17 cluster					
Xeon E5-2698v4	20	4	50	2.2	2400
Quad socket, IBM Power 775 (the Boreasz cluster					
Power 7	8	4	32	3.83	1600

per von Vladimir Stegailov und Vyacheslav Vecher mit dem reduzierten Parameter $R_{peak\_\tau\_iter}$, der die Peakperformance von τ_{iter} repräsentiert, und um das Speichersystem zu beschreiben mit N_{cores} / $N_{mem.ch.}$ gearbeitet. Die Speicherbandbreite wird hierbei vernachlässigt. Die Abbildung 5 zeigt die gleichen Daten wie die Abbildung 4 aber mit den reduzierten Parametern. Dadurch wurden die Unterschiede in der floating point Performace der unterschiedlichen CPU Kerne und die Unterschiede in der Anzahl der Speicherkanäle entfernt. Der Anstieg von $R_{peak\_\tau\_iter}$, der proportional zu der Anzahl von CPU Zyklen führt, zeigt den Anstieg des Overhead aufgrund der limitierten Speicherbandbreite. Die weitere Verteilung der Datenpunkte im Abbildung 4 kann zum Teil auf die unterschiedlichen L3 Cache Größen der unterschiedlichen CPU zurückgeführt werden. In Abbildung 6 werden die Datenpunkte aus Abbildung 4 ausgewählt die zu dem reduzierten Parameter $N_{cores}/N_{mem.ch.}$ = 1-2 zählen und plotten die $R_{peak\_\tau\_iter}$ Werte als eine Funktion der L3 Cachegröße pro Kern. Daraus folgt, je größer die L3 Cachegröße pro Kern ist, desto kleiner ist der $R_{peak\_\tau\_iter}$ Wert.

In Abbildung 7 und Abbildung 8 werden die durchschnittliche und die totale verbrauchte Energie als Funktion von τ_{iter} dargestellt. Diese Experimente mit dem Core i7 6900K zeigen, dass die Erhöhung der DRAM Frequenz von 2133 auf 3200 zu einem 10 Prozent höheren Energiebedarf führen, aber dafür 10 Prozent kleinere Iterationen für 4 und 8 Kerne benötigt werden. Für die E5-2620v4 und E52660v4 lässt sich schlussfolgern, dass nicht aktive Kerne nicht signifikant zu einer Erhöhung des Energiebedarfs führen. Der AMD Ryzen 1800x hat ein ähnliches Maß an Energiebedarf. Nur beim Übergang von 1 zu 2 Kernen ist der durchschnittliche Energiebedarf höher als bei den Intel Broadwell Prozessoren. Dies hat wahrscheinlich mit der Aktivierung der beiden Quad-Core CPU Complexen des AMD Ryzen zu tun. Die CPU mit der besten Leistung und Energieeffizienz der untersuchten CPUs ist der Intel E5-2660v4 mit 4 Kernen [16].

 DOI: 10.14459/2018md1430700

5 Fazit

Es wurde gezeigt, dass der neue AMD Ryzen von der Performance mit den Intel CPUs (Sandy Bridge, Haswell und Broadwell) vergleichbar ist. Die komplette Überarbeitung der Architektur mit von Intel Prozessoren bekannten Technologien und eigenen Techniken macht sich bemerkbar. Im Test mit dem VASP Code, als Indikator für HPC Anwendungen, konnte gezeigt werden, dass für diesen Code die optimale Anzahl von Memory Channels bei 1 bis 2 liegt. Mehr als 2 Kerne pro Kanal bringen keine Beschleunigung. Die VASP Performance erhöht sich stark mit größerem L3 Cache. Jedes weitere MB an L3 Cache pro Kern verringert die Zeit bis zur Lösungsberechnung um 30 bis 50 Prozent. Es lässt sich allgemein sagen, dass Prozessoren mit größerer L3 Cachegröße bei gleicher Performace weniger Energie benötigen. AMD hat mit der Erhöhung des L3 Cache bei der Zen Mikroarchitektur sowie Technologien, die schon von Intel bekannt sind und neuen Techniken, zu Intel aufschließen können.
Durch die Zen Plattform, dessen Implementierung der AMD Ryzen darstellt, ist AMD zu einem ernstzunehmenden Konkurrenten für den Marktführer Intel geworden.

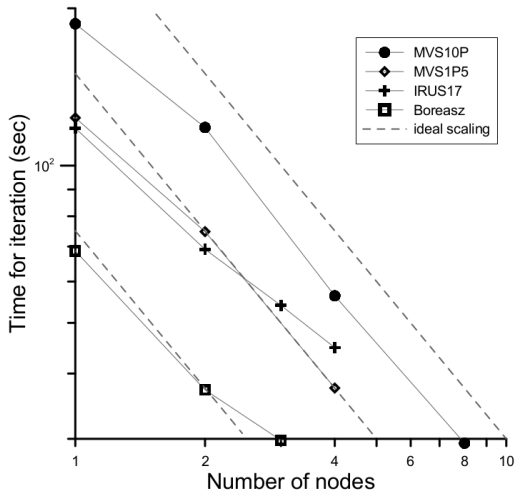


Abbildung 3: Performancetest mit 8 Kernen pro Socket (Quelle [16])

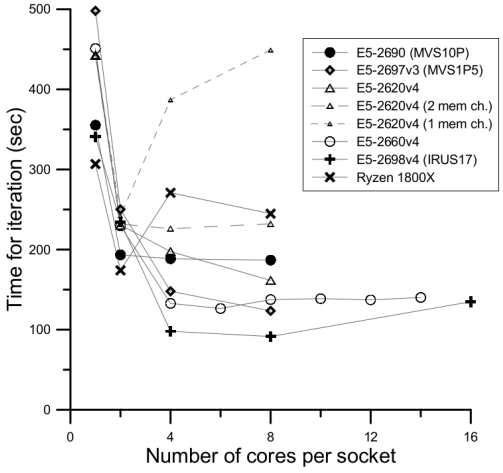


Abbildung 4: Die Zeit für den ersten Berechnungsdurchgang (Quelle [16])

 DOI: 10.14459/2018md1430700

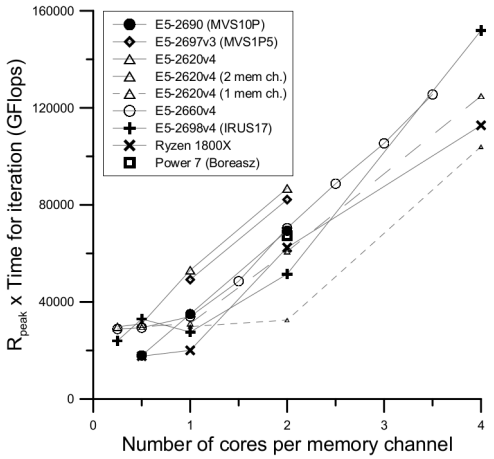


Abbildung 5: Die Zeit für den ersten Berechnungsdurchgang mit dem reduzierten Paramter $R_{peak\_\tau\_iter}$ und $N_{cores}/N_{mem.ch.}$ (Quelle [16])

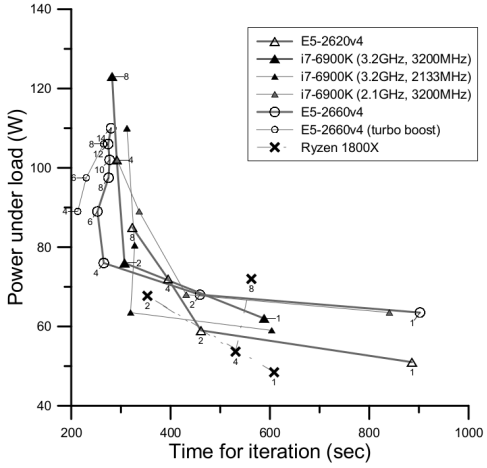


Abbildung 7: Durchschnittliche Leistungsaufnahme unter Last. Die Anzahl der Kerne ist in der Abbildung gekennzeichnet. (Quelle [16])

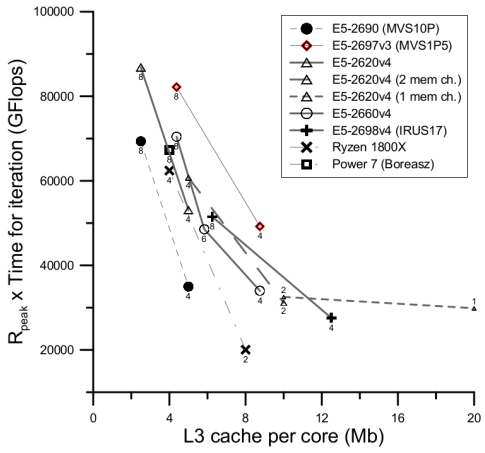


Abbildung 6: Die Abhängigkeit des Parmater $R_{peak\_\tau\_iter}$ vom L3-Chache pro Kern. (Quelle [16])

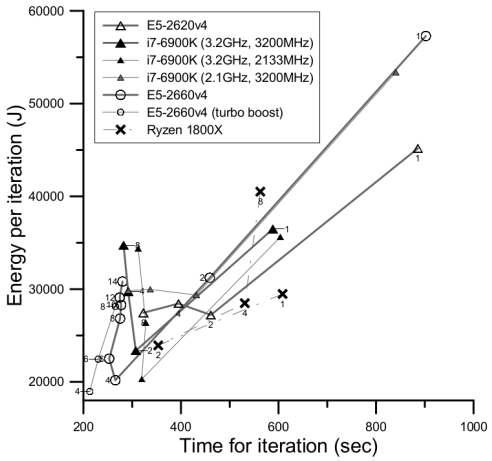


Abbildung 8: Durschnittlicher Energiebedarf pro Berechnungsschitt. Die Anzahl der Kerne ist in der Abbildung gekennzeichnet. (Quelle [16])

Literatur

[1] Paul Alcorn. Intel coffee lake vs. ryzen: A side-by-side comparison. http://www.tomshardware.com/news/intel-coffee-lake-amd-ryzen,35546.html, besucht: 2017-12-12, 2017.

[2] AMD. Amd epyc 7000. http://www.amd.com/de/products/epyc-7000-series, besucht: 2017-12-12, 2017.

[3] AMD. Amd ryzen 7 1800x. https://www.amd.com/en/products/cpu/amd-ryzen-7-1800x, besucht: 2017-12-12, 2017.

[4] Maciej Cytowski. Best practice guide — ibm power 775. http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-IBM-Power-775.pdf, besucht: 2017-12-12, 2013.

[5] Armando Faz-Hernández and Julio López. Fast implementation of curve25519 using avx2. In *International Conference on Cryptology and Information Security in Latin America*, pages 329–345. Springer, 2015.

[6] Linley Gwennap. Epyc: Designed for effective performance. 2017.

[7] Digh Hisamoto, Wen-Chin Lee, Jakub Kedzierski, Hideki Takeuchi, Kazuya Asano, Charles Kuo, Erik Anderson, Tsu-Jae King, Jeffrey Bokor, and Chenming Hu. Finfet-a self-aligned double-gate mosfet scalable to 20 nm. *IEEE Transactions on Electron Devices*, 47(12):2320–2325, 2000.

[8] Dave James. Amd ryzen reviews, news, performance, pricing, and availability, 2017. https://www.pcgamesn.com/amd/amd-zen-release-date-specs-prices-rumours/, besucht: 2017-12-12.

[9] Georg Kresse and J Furthmüller. Vienna ab-initio simulation package (vasp). *Vienna: Vienna University*, 2001.

[10] Chris Lomont. Introduction to intel® advanced vector extensions. https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions, besucht: 2017-12-12.

[11] R Lutsishin. The cheapest and most powerful microprocessor? *Collection of theses of the All-Ukrainian Student Scientific and Technical Conference: Natural and Humanitarian Sciences. Topical Issues*, 1:60–60, 2017.

[12] H. McIntyre, S. Arekapudi, E. Busta, T. Fischer, M. Golden, A. Horiuchi, T. Meneghini, S. Naffziger, and J. Vinh. Design of the two-core x86-64 amd; bulldozer; module in 32 nm soi cmos. *IEEE Journal of Solid-State Circuits*, 47(1):164–176, Jan 2012.

[13] SL MERISTATION MAGAZINE. Amd ryzen, en detalle,¿ amenaza al imperio intel?, 2017. http://meristation.as.com/reportaje/amd-ryzen-en-detalle-amenaza-al-imperio-intel/2180726, besucht: 2017-12-12.

[14] T. Singh, S. Rangarajan, D. John, C. Henrion, S. Southard, H. McIntyre, A. Novak, S. Kosonocky, R. Jotwani, A. Schaefer, E. Chang, J. Bell, and M. Co. 3.2 zen: A next-generation high-performance x86 core. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 52–53, Feb 2017.

[15] T. Singh, A. Schaefer, S. Rangarajan, D. John, C. Henrion, R. Schreiber, M. Rodriguez, S. Kosonocky, S. Naffziger, and A. Novak. Zen: An energy-efficient high-performance x86 core. *IEEE Journal of Solid-State Circuits*, PP(99):1–13, 2017.

[16] Vladimir Stegailov and Vyacheslav Vecher. Efficiency analysis of intel and amd x86_64 architectures for ab initio calculations: A case study of vasp. In *Russian Supercomputing Days*, pages 430–441. Springer, 2017.

[17] Yu-Hsiung Tsai and Po-Hao Lee. Word line boost circuit, August 13 2013. US Patent 8,509,026.

Hauptseminar Hochleistungsrechner: Aktuelle Trends und Entwicklungen
Wintersemester 2017/2018

Speichertechnologien und ihre Nutzung

Maurice Rang
Ludwig-Maximilians-Universität München

30.01.2018

Abstract

Die Bedeutung von nicht-flüchtigen Speichertechnologien sowie deren Verwendung steigt nach wie vor an. Die vorliegende Seminararbeit liefert einen kurzen Überblick über die in nicht-flüchtigen Speichertechnologien verwendete Hardware und kann als Einstieg in das Thema genutzt werden. Dazu wird zunächst deren grundlegender Aufbau vorgestellt. Darüberhinaus werden Unterschiede zwischen den einzelnen Technologien erläutert und analysiert. Abschließend werden in diesem Kapitel verschiedene Programmiermodelle und Einsatzmöglichkeiten für nicht-flüchtige Speichertechnologien aufgezeigt. Zum Ende dieser Arbeit gibt es eine kurze Darstellung der Speichertechnologie 3D Dram und dazu zwei passende Beispiele.

1 Einleitung

In den letzten Jahren ist die Datenmenge, die durch die Menschheit produziert wurde, auf ein sehr hohes Niveau angewachsen. Schaut man sich das Jahr 2007 an stellt man fest, dass sich hier die Kapazität des weltweiten Datenbestandes noch auf etwa 295 Exabyte (EB) beziffern ließ [14]. Circa 10 Jahre später im Jahre 2016 ist dieser Bestand schon auf 16 Zettabyte (ZB) angewachsen und damit um über 5000% angestiegen. Schenkt man Prognosen einiger Experten Glauben soll sich der weltweite Datenbestand im Jahre 2025 auf über 163 Zettabyte (ZB) belaufen und damit nochmals um mehr als 1000% im Vergleich zum Jahr 2016 anwachsen [13]. Eine Begleiterscheinung, die durch diese rasante Entwicklung entsteht, ist die immer größer werdende Nachfrage nach neuen und verbesserten Speichertechnologien, um diese Menge an Daten effizient und schnell verarbeiten zu können. Parallel dazu ist die Weiterentwicklung und konstante Verbesserung der benötigten Mikroprozessoren genauso unabdingbar, da die Verarbeitung dieser Größenordnung an Daten ein hohes Maß an Rechenleistung benötigt. In Bezug auf die Entwicklung von Mikroprozessoren stellte Gordon Moore einer der Mitgründer von Intel im Jahr 1965 eine Theorie auf, die von da an als Mooresches Gesetz bekannt wurde. Diese Theorie besagt, dass "sich die Anzahl der Komponenten, die sich auf einem Chip befindet, jedes Jahr ungefähr um das Doppelte erhöht" [10].
Auf den nachfolgenden Seiten werden zunächst vier Speichertechnologien vorgestellt, die zur Familie der nicht flüchtigen Speicher gehören. Dabei werden von jeder Technologie die technischen Grundlagen und ihre Funktion dargelegt. Im Anschluss daran wird auf die potenziellen Einsatzgebiete und Programmiermodelle von nicht-flüchtigen Speichern eingegangen. Den Schlussteil der vorliegenden

 DOI: 10.14459/2018md1430700

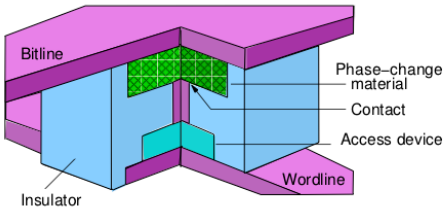


Abb. 1: Aufbau eines Phase-Change Memory Device [12]

Arbeit bildet das Kapitel 3D Stacked DRAM. Hier wird wiederum zunächst die technsiche Funktionalität dieser Technologie und dann deren Funktionalität beschrieben.

2 NVRAM

Im folgenden Kapitel werden vier unterschiedliche Speichertechnologien mit ihrem technischen Aufbau und ihrer Funktion vorgestellt, die zur Familie der nicht-flüchtigen Speicher gehören.

2.1 Definition

NVRAM ist eine Form des neuen und revolutionären Speichers, der auf der einen Seite Zugriffscharakteristiken von herkömmlichen RAM Speichern wie DRAM oder SRAM bereitstellt und zum anderen die nicht-flüchtigen Speichereigenschaften von Sekundärspeichern wie Flash miteinander vereint. Da die Halbleitertechnologie sehr große Fortschritte macht ist davon auszugehen, dass NVRAM zukünftig ein fester Bestandteil von eingebetteten Systemen und Computern wird [6].

NVRAM weist im Vergleich zu anderen Speichertechnologien zahlreiche Vorteile auf, vor allem gegenüber Flash-Speichern. Ein großer Vorteil, den nicht-flüchtige Speicher im Gegensatz zu Flash-Speichern haben ist ihre schnelle Zugriffszeit, die hauptsächlich auf Schreibzugriffe bezogen ist. Während die Zugriffszeit bei Flash-Speichern in

Parameter	DRAM	NAND Flash	NOR Flash	PCM
Density	1X	4X	0.25X	2X-4X
Read Latency	60ns	25 us	300 ns	200-300 ns
Write Speed	≈1 Gbps	2.4 MB/s	0.5 MB/s	≈100 MB/s
Endurance	N/A	10^4	10^4	10^6 to 10^8
Retention	Refresh	10yrs	10yrs	10 yrs

Abb. 2: PCM im Vergleich [12]

einem Bereich von mehreren hundert Millisekunden liegt, beträgt die Schreibzeit bei nicht-flüchtigen Speichern wenige Nanosekunden. Dadurch eignet sich nicht-flüchtiger Speicher besonders für Situationen, in denen häufige Schreibvorgänge aufftreten [6].

2.2 Phase-Change Memory (PCM)

Phase-Change Memory (PCM) ist eine Speichertechnologie, die zur Familie der nicht-flüchtigen Speicher gehört und ein sogenanntes ”phase-change material” zur Sicherung von Daten verwendet. Dieses Material kann zwei physische Zustände annehmen: kristallin oder amorph. Der Vorteil dieses Materials liegt darin, dass es seinen Zustand sehr zuverlässig, schnell und oft wechseln kann. Der amorphe Zustand hat eine geringe optische Reflexivität und einen hohen elektrischen Widerstand. Der kristalline Zustand dagegen hat eine hohe Reflexivität und einen geringen Widerstand. Abbildung 1 zeigt den Aufbau eines Phase-Change Memory Device’. Das Material befindet sich zwischen einer oberen und unteren Elektrode mit einem Hitzelement, das die Erweiterung zur unteren Elektrode darstellt. Das Kristallisieren des Materials wird als sogenannte SET-Methode bezeichnet. Diese Operation wird durch moderate Leistung sowie eine längere Dauer von Elektroimpulsen kontrolliert. Dabei befindet sich die Speicherzelle in einem Zustand mit geringem Widerstand. Die RESET-Methode dagegen zeichnet sich durch hohe Stromimpulse aus, die die Speicherzelle in einen Zustand mit hohem Widerstand versetzen. Die gespeicherten Daten der Zelle können mit

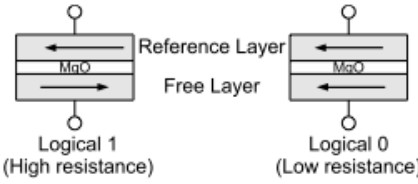
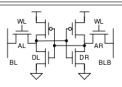
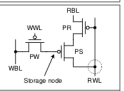


Abb. 3: Magnetic Tunnel Junction (MTJ) [16]

Hilfe von einer sehr geringen Stromzuführung ausgelesen werden. In der oberen Tabelle wird die Speichertechnolgie PCM mit anderen Technologien in Bezug auf die Parameter Dichte, Lese-Latenz, Schreibgeschwindigkeit, Standhaftigkeit und Speicherung verglichen. Schaut man sich nun den Vergleich an sieht man, dass PCM die vielversprechensten Charakteristiken besitzt. In Bezug auf die Dichte weist PCM (2X - 4X) ähnliche Werte auf wie sie bei NAND Flash (4X) zu finden sind. Sieht man sich nun den Parameter Lese-Latenz an, stellt man fest, dass es hier Parallelen zu NOR Flash (300 ns) gibt, welche circa vier mal langsamer ist als die Lese-Latenz bei DRAM (60 ns). Vergleicht man abschließend den Paramter Schreibgeschwindigkeit ist zu erkennen, dass diese bei PCM mit circa 100 MB/s höher ist als die bei den zwei Flash-Speichern [12].

2.3 Spin-Transfer Torque RAM (STT-RAM)

Spin-Transfer Torque RAM (STT-RAM) ist eine entstehende nicht-flüchtige Speichertechnologie, die als Universalspeicher eingesetzt werden kann [15]. Sie gehört zu den sogenannten magnetoresistiven Speicherarten (MRAM). Wie bei herkömmlichen magnetoresistiven Speicherarten nutzt eine STT-RAM Zelle zum Speichern von binären Daten eine Magnetic Tunnel Junction (MTJ). Eine MTJ besteht aus zwei ferromagnetischen Schichten, dem Reference Layer und Free Layer und zusätzlich aus einem Tunnel Barrier Layer (MgO). Die magnetische Ausrichtung des Reference Layer ist beständig und ändert sich nicht, wohingegen die magnetische Ausrichtung des Free Layers geändert werden

	(A) SRAM	(B) STT-RAM	eDRAM	
			(C) 1T1C	(D) Gain cell
Cell schematic				
Process	CMOS	CMOS + MTJ	CMOS + Cap	CMOS
Cell size (F^2)	120 - 200	6 - 50	20 - 50	60 - 100
Data storage	Latch	Magnetization	Capacitor	MOS gate
Read time	Short	Short	Short	Short
Write time	Short	Long	Short	Short
Read energy	Low	Low	Low	Low
Write energy	Low	High	Low	Low
Leakage	High	Low	Low	Low
Endurance	10^{16}	$> 10^{15}$	10^{16}	10^{16}
Retention time	-	-	< 100 us *	< 100 us *
Features	(+) Fast (-) Large area (-) Leakage	(+) Non-volatile (+) Potential to scale (-) Extra process (-) Long write time (-) High write energy (-) Poor stability	(+) Low leakage (+) Small area (-) Extra process (-) Destructive read (-) Refresh	(+) Low leakage (+) Decoupled read/write (-) Refresh

* 32 nm technology node

Abb. 4: STT-RAM im Vergleich [16]

kann. Die magnetische Ausrichtung zwischen den beiden Schichten erzeugt unterschiedliche Widerstände beim MTJ, die zum repräsentieren der in den Zellen gespeicherten Daten genutzt werden. Wenn das magnetische Feld zwischen den beiden Schichten parallel zueinander ist und der Widerstand des MTJ niedrig ist, repräsentiert dieser Zustand eine logische 0. Sobald das Feld antiparallel ist und die MTJ einen hohen Widerstand hat, wird eine logische 1 repräsentiert (siehe Abbildung 3). Während des Lesevorgangs liegt eine kleine negative Spannung zwischen der Bit und Word line an. Um eine logische 0 schreiben zu können, wird eine positive Spannung zwischen der Bit und Word line angelegt. Eine logische 1 wird durch eine negative Spannung geschrieben, die einen Strom in die entgegengesetzte Richtung erzeugt [16]. Abbildung 4 zeigt STT-RAM im Vergleich zu den Speichertechnologien SRAM und eDRAM hinsichtlich verschiedener Kriterien. Sieht man sich nun die einzelnen Punkte an stellt man fest, dass STT-RAM nicht viel schlechter abschneidet als die anderen beiden Technologien. Ein wichtiger Vorteil, den STT-RAM gegenüber den anderen Technologien besitzt ist, dass die Eigenschaft des nicht flüchtigen Speichers. Nachteile, die diese Technologie im Vergleich zu SRAM und eDRAM dennoch aufweist sind eine lange Schreibzeit, ein hoher Energieverbrauch beim Schreiben und eine geringe Beständigkeit.

 DOI: 10.14459/2018md1430700

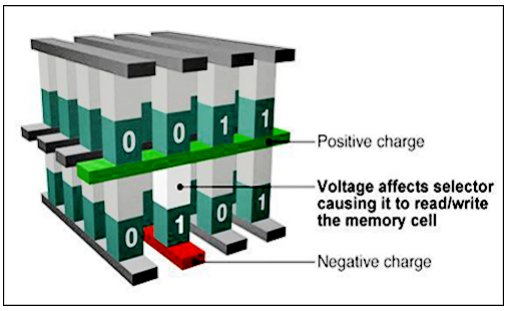


Abb. 5: 3D-Crosspoint-Architektur [1]

2.4 Resistive Random Access Memory (R-RAM)

Resistive Random Access Memory (R-RAM) ist ein weiteres Beispiel für eine nicht-flüchtige Speichertechnologie. Diese Technologie besteht im Grunde aus zwei Operationen: Unipolares Switching und bipolares Switching. Innerhalb dieses Kontextes ist das unipolare Switching für das Ausführen von Programmen und das Löschen zuständig. Dies geschieht mit Hilfe von kurzen und langen Impulsen oder mit Hilfe von hoher und niedriger Stromspannung mit der selben Spannungspolarität. Im Gegensatz dazu werden bei bipolarem Switching kurze Impulse mit einer umgekehrten Spannungspolarität verwendet. Ein Beispiel für unipolares Switching ist ein filament-basiertes R-RAM Gerät. Ein Filament wird in Form eines isolierten Dielektrikums erzeugt nachdem eine hohe Spannung angesetzt wurde. Nachdem das Filament geformt wurde, kann es durch angemessene Spannung gesetzt oder zurückgesetzt werden. Ein typisches Beispiel für bipolares Switching ist ein PCM Gerät, welches aus zwei metallischen Elektroden besteht. Ein dünner Film aus Elektrolyten befindet sich zwischen den Elektroden. Wenn eine negative Spannung an der inaktiven Elektrode angelegt wird fließen metallische Ionen in die Elektrolyte und werden durch die inaktive Elektrode vermindert. Abschließend formen die Ionen einen kleinen "Nanodraht" zwischen den Elektroden. Daraus resultiert, dass der Widerstand zwischen den Elektroden stark gesenkt wird. Soll die Zelle gelöscht werden wird eine postive Spannung angelegt was zur Folge hat, dass die Ionen zurück in die Elektrolyte fließen. Der erzeugte "Nanodraht" wird zerstört und der Widerstand wieder erhöht [9].

2.5 3D XPoint

3D XPoint ist eine von Intel und Micron entwickelte nicht-flüchtige Speichertechnologie, die auf der sogenannten CrossPoint-Architektur basiert. In Abbildung 5 ist ein Querschnitt dieser Architektur zu sehen. Die Besonderheit hierbei liegt in der Adressierung und in der Kontroll-Architektur der Speicherzellen. Der Einsatz von Silizium für die Herstellung der elektronischen Komponenten ist nach wie vor gegeben. Der Unterschied zu den anderen Technologien liegt jedoch darin, dass nun keine Notwendigkeit mehr darin besteht für jeden Bereich einer einzelnen Zelle oder mehrerer Zellen einen Transistor einzusetzen. Dadurch wird es ermöglicht die Speicherdichte auf der einzelnen Fläche eines Chips um das Zehnfache zu erhöhen. Jede einzelne Speicherzelle besteht aus einem Speicherwert-Selektor (in Abbildung 5 hellgrau markiert) und einem Speicher-Bit (in Abbildung 5 grün markiert). Diese werden in mehreren Schichten gestapelt und von sogenannten Adressierungsleitungen miteinander verbunden (in Abbildung 5 als graue Balken dargestellt). Durch die Adressierungsleitungen liegt an jeder Speicherzelle eine Spannung an, die es ermöglicht zu bestimmen, ob es sich bei der auszuführenden Aktion um einen Lese- oder Schreibvorgang handelt. Das Weglassen der Transistoren für jede einzelne Zelle und das Zusammenschließen mehrerer Zellen hat zur Folge, dass die Architektur des Speichers stabiler und langlebiger wird [2]. Setzt man 3D XPoint nun in Vergleich mit NAND und DRAM Speichern ist festzustellen, dass 3D XPoint ähnliche Werte in Sachen Lese- und Schreibgeschwindigkeit aufweist wie sie bei DRAM vorkommen. NAND Speicher sind bezüglich dieses Parameters langsamer als die beiden anderen Technologien. Einen Vorteil, den NAND sowie 3D XPoint Speicher aufweisen sind die niedrigen Kosten. Diese sind viel geringer als die bei DRAM

Speichern der Fall sind. Zu erwähnen ist jedoch, dass 3D XPoint Speicher immer noch ungefähr zehn mal teurer sind als NAND Speicher. Einen weiteren Vorteil, den 3D XPoint und NAND gegenüber DRAM haben ist, dass sie nicht-flüchtig sind [3].

3 Programmiermodelle für NVRAM

Die nicht-flüchtige Eigenschaft von Speichern erlaubt eine Vielzahl an unterschiedlichen Programmiermodellen für nicht-flüchtige Speicher, die diese gegen Programm- und auch gegen Systemabstürze sicher machen. Einige dieser Programmiermodelle werden nun im Folgenden erläutert.

Ein erster Ansatz ist das sogenannte "Mnemosyne" ein Interface, das von Volos et al. zum Programmieren von SCM's entwickelt wurde. Dieses Interface erlaubt es Applikationen statische Variablen zu deklarieren deren Werte selbst über einen Systemneustart hinaus bestehen bleiben. Darüber hinaus lässt es zu, dass Applikationen Speicher reservieren, der durch persistenten Speicher unterstützt wird. Die Schreibreihenfolge für persistente Speicher wird hierbei durch eine Software geregelt, die non-cached Schreibmodi, cache-line flush Anweisungen und Speicherbarrieren verwendet.
Ein weiterer Ansatz, der nach Coburn et al. benannt wurde ist der NV-heap. Dies ist ein persistentes Objektsystem, das eine transaktionale Semantik bereitstellt. Um sicherstellen zu können, dass die Anwendungen konsistent ausgeführt werden, werden bei NV-heap die Daten in flüchtige und nicht-flüchtige Daten separiert.
Ein Ansatz, der sich von den beiden oben Dargestellten differenziert wurde von Narayanan et al. entwickelt. Dieser präsentierte eine Möglichkeit, bei der der gesamte Systemspeicher nicht-flüchtig ist. Dazu verwenden ein Prinzip, das als "flush-on-fail" bezeichnet wird. Bei diesem Prinzip wird der vorübergehende Status, der sich in den Registern der CPU und den Cache-lines befindet erst dann gespeichert wird, wenn es zu einem Fehlerzustand kommt. Bei einem Neustart des Systems werden die aktuellen Applikations- und OS-Stati durch einen transparenten Checkpoint wieder hergestellt.
Zhao et al. haben eine Herangehenweise präsentiert, die einen nicht-flüchtigen last-level cache und nicht-flüchtigen Systemspeicher verwendet, um eine persistente Speicherhierarchy schaffen zu können. Hierbei ist es so, dass wenn eine Cache-line aktualisiert wurde beinhaltet sie die aktuelle Version während die Daten, die sich im nicht-flüchtigen Speicher befinden noch die alte Version beinhalten. Wenn die "dreckige" Cache-line bereinigt wurde, wird automatisch der alte Stand im nicht-flüchtigen Speicher durch den neuen Stand ersetzt.
Eine letzte Technik, die noch zu erwähnen bleibt ist die von Condit et al. Sie beschäftigten sich mit einer Methodik, bei dem ein transaktionales Filesystem zum Einsatz kommt, das die byte addressierbarkeit von Speichern so beeinflusst, sodass die Menge an Metadaten, die während einer Aktualisierung geschrieben wird, drastisch reduziert wird. Dieses Filesystem soll garantieren, dass das Schreiben im Filesystem dauerhaft wird und jede Operation des Filesystems automatisch in einer bestimmten Reihenfolge ausgeführt wird [11].

4 Einsatzmöglichkeiten von NVRAM

Da jede einzelne der vorgestellten Speichtechnologien ihre Vor- und Nachteile besitzt haben sich einige Wissenschaftler gedacht, dass man die Vorteile besser nutzen könnte, wenn man die einzelnen Technolgien miteinander kombiniert. Im folgenden Abschnitt werden nun drei dieser Kombinationen dargestellt [11].

4.1 Flash + PCM

Sun et al. hat die Kombination zwischen Flash-Speicher und PCM vorgeschlagen. Hierbei soll PCM als Protokollumgebung für den NAND Flash-Speicher genutzt werden. Dieses Verfahren reduziert die Lese- sowie Löschoperationen für Flash,

 DOI: 10.14459/2018md1430700

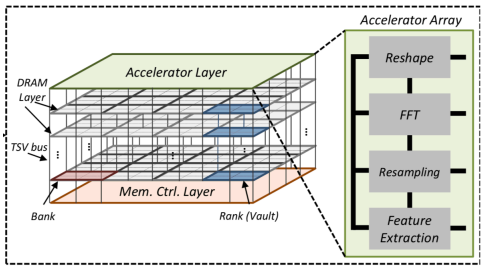


Abb. 6: Architektur eines 3D Stacked DRAM Systems [7]

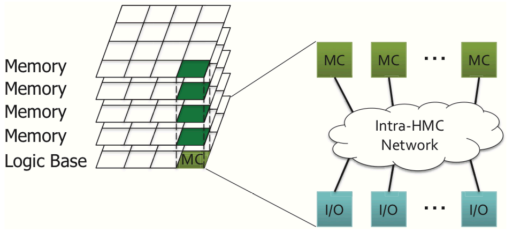


Abb. 7: Architektur eines HMC [8]

was die Lebenszeit, die Performance und Energieeffizeienz des Flash-Speichers verbessert. Durch die byte-addressierbarkeit von PCM wird zusätzlich die Performance für Leseoperationen verbessert.

4.2 DRAM + PCM

Qureshi et al. haben einen Ansatz hervorgebracht, der aus einem hybriden Hauptspeicher besteht, bei dem DRAM als "Page Cache" genutzt wird, um die Vorteile in Bezug auf Latenz des DRAMs mit den Vorteilen der Kapazität von PCM kombinieren zu können. Bei einem Seitenfehler wird die Seite, die von der Hard Disk geholt wurde, nur auf den DRAM Speicher geschrieben. Die Seite wird nur dann auf auf den PCM Speicher geschrieben, wenn sie aus dem DRAM Speicher entfernt wurde als "dreckig" gekennzeichnet wurde.

4.3 PCM + Flash + HDD

Kim et al. haben das Potential von PCM in Speicherhierarchien bezüglich Kosten und Performance bewertet. Sie beobachteten, dass das Schreiben auf Flash-Speicher basierend auf dem materiellen Level wesentlich langsamer ist als das Schreiben auf PCM. Basierend auf dem Systemlevel ist das Schreiben auf flash-basierte SSD's schneller als das auf PCM-basierte SSD's was durch den Stromverbrauch bedingt ist. Weiterhin beobachteten sie, dass das Hinzunehmen von PCM die Performance in einem abgestuften Speichsystem verbessert und auch eine höhere Performance pro Dollar hervorrufen kann.

5 3D Stacked DRAM

3D Stacked DRAM ist eine Speichertechnologie, bei der mehrere DRAM Matrizen und logische Schichten übereinander gestapelt werden und durch sogenannte TSV's (Through Silicon Vias) verbunden sind. Durch viele TSV Verbindungen ist eine Kommunikation mit hoher Bandbreite und geringer Latenz zwischen den einzelnen Schichten möglich. Abbildung 6 zeigt den Aufbau einer 3D DRAM Architektur. Hierbei ist zu erkennen, dass diese Struktur aus mehreren DRAM Schichten besteht, die an sich auch nochmal aus mehreren Teilstücken (in der Abbildung als *banks* gekennzeichnet) bestehen. Das vertikale Aufeinanderstapeln von banks wird als *vault* bezeichnet. Jedes vault hat seine eigene TSV-Verbindung und seinen eigenen Vault-Controller. Die logische Schicht (in Abbildung 6 als Mem. Ctrl. Layer bezeichnet) besteht aus dem Memory Controller, einem Crossbar Switch und aus Vault und Link Controllern. Diese Kontrolleinheiten belegen nicht den gesamten Platz der logischen Schicht, sondern lassen einen Raum für Erweiterungen [4].

5.1 Hybrid Memory Cube (HMC)

Als Beispiel für die oben erläuterte Speichertechnologie wird die Technologie Hybrid Memory Cube (HMC) gewählt. Die Architektur von HMC ist ähn-

lich der in Abbildung 6 dargestellten Struktur. Der Cube besteht aus mehreren übereinander gestapelten DRAM Schichten und einer logischen Schicht am unteren Ende. Die einzelnen Schichten sind durch TSV's miteinander verbunden. Jede Speicherschicht des Cubes ist in einzelne Segmente aufgeteilt und das vertikale Zusammenschließen mehrerer Segmente wird als Vault bezeichnet. Der Prozessor kommuniziert mit dem HMC-seitigen Memory Controller indem er Nachrichten schickt, die Speicheraktionen, wie Schreiben oder Lesen, Speicheradressen und/oder Speicherdaten enthalten sendet. Innerhalb der logischen Schicht wird zusätzlich ein Switch benötigt, um die verschiedenen Vault Memory Controller und die I/O-Ports zu verbinden [8].

5.2 High Bandwidth Memory (HBM)

Ein weiteres Beispeil für 3D DRAM Speicher ist High Bandwidth Memory (HBM). Diese Technologie wird vorwiegend von Graphikprozessoren und als Nachdolger für GDDR5 verwendet. HBM wird von AMD, Nvidia und Hynix entwickelt. Dieser Standard zielt auf leistungsstarke Grafikanwendungen und Netzwerkanwendungen ab, die eine hohe Performance benötigen. HBM hat einen niedrigen Stromverbrauch, ultra-weite Kommunikationsleitungen und eine neuartige Stapelstruktur. Es ist vorgesehen, dass bis zu 8 Speicherchips übereinander gestapelt werden und mit Hilfe von TSV's verbunden werden. Die Prozessoranordnung bei HBM besteht aus einem Kernprozessor und vier Speicherstapeln. Die Speicherstapel an sich befinden sich auf einer logischen Schicht. Der Silizium Interposer ist ein Chip mit metallischen Schichten, die keine Logik besitzen. Der Aufbau von HBM ist nochmals detailiert in Abbildung 8 dargestellt [1].

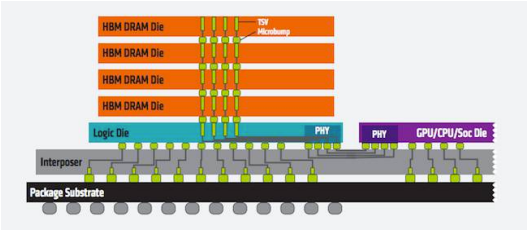


Abb. 8: Architektur eines HBM [1]

6 Zusammenfassung

Im vorliegenden Paper wurden zu Beginn vier Speichertechnologien vorgestellt, die zur Familie der nicht-flüchtigen Speicher gehören. Hierbei wurde der Fokus auf den technischen Aufbau dieser Technologien und deren Verwendung gelegt. Jede der dargelegten Technologien hat eine andere Funktionsweise und ist in sich anders aufgebaut. Durch diese Unterschiede weisen sie bezüglich der wichtigsten Parameter für Speichertechnologien Dichte, Schreib- und Lesegeschwindigkeit sowie Dauerhaftigkeit unterschiedliche Werte auf. Vergleicht man nun abschließend die vier Speichertechnologien untereinander in Bezug auf die zuvor genannten Parameter stellt man fest, dass jede einzelne Speichertechnologie ihre Vor- und auch Nachteile besitzt. Beispielsweise weist STT-RAM in Bezug auf die Lesegeschwindigkeit excellente Werte auf wohingegen die Lesegeschwindigkeit bei PCM eine Dauer von 200 bis 300 ns hat. Abschließend werden in diesem Kapitel unterschiedliche Programmiermodelle und Einsatzmöglichkeiten für NVRAM erläutert. In Abschnitt 5 wurde zunächst der grundlegende Aufbau von 3D DRAM Speichern dargelegt. Hierbei lag der Fokus wiederum auf dem hardwareseitigen Aufbau und dessen Funktionalität. Abschließend wurden zwei Beispiele aufgeführt, die die Verwendung von 3D DRAM Speichern verdeutlichen sollten. Die Bedeutung von nicht-flüchtigen Speichertechnologien wird in Zukunft noch mehr an Bedeutung gewinnen, da sie universell und vielseitig einsetzbar sind.

Literatur

[1] Amd high bandwidth memory (hbm). http://www.semiconductorscentral.com/memory_page.html.

[2] 3d xpoint technologie: 1000mal schneller als ssd. https://www.krollontrack.de/blog/3d-xpoint-technologie-1000mal-schneller-als-ssd/4928/, December 2015.

[3] Intel® optane™: Die revolution mit ultraschneller speichertechnik beginnt 2016. https://www.intel.de/content/www/de/de/it-managers/non-volatile-memory-idf.html, 2015.

[4] B. Arkin, J. Hoe, and F. Franchetti. Hardware accelerated memory layout transform within 3d-stacked dram. Technical report, Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh.

[5] M. Chang, P. Rosenfeld, S. Lu, and B. Jacob. Technology comparison for large last-level caches (l3cs): Low-leakage sram, low write-energy stt-ram, and refresh-optimized edram. Technical report, University of Maryland.

[6] I. Doh, D. Lee, J. Choi, and S. Noh. Exploiting non-volatile ram to enhance flash file system performance. Technical report, Hongik University and Dankook University, 2007.

[7] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. Low, L. Pileggi, J. Hoe, and F. Franchetti. 3d-stacked memory-side acceleration: Accelerator and system design. Technical report, Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh.

[8] G. Kim, J. Kim, J. Ahn, and J. Kim. Memory-centric system interconnect design with hybrid memory cubes. Technical report, Seoul National University, KAIST, 2013.

[9] H. Li and Y. Chen. An overview of non-volatile memory technology and the implication for tools and architectures. Technical report, Alternative Technology Group Seagate Technology LLC, 2009.

[10] C. Mack. Fifty years of moore's law, 2011.

[11] S. Mittal and J. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. Technical report, IEEE, 2015.

[12] M. Qureshi, V. Srinivasan, and J. Rivers. Scalable high performance main memory system using phase-change memory technology. Technical report, IBM Research, 2009.

[13] D. Reinsel, J. Gantz, and J. Rydning. Data age 2025: The evolution of data to life-critical. Technical report, IDC Headquarter, 2017.

[14] Christoph Schrader. Explosion des cyberspace. *Süddeutsche Zeitung*, 2011.

[15] C. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. Stand. Relaxing non-volatility for fast and energy-efficient stt-ram caches. Technical report, Department of Computer Science and Department of Electrical and Computer Engeneering University of Virginia.

[16] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. Energy reduction for stt-ram using early write termination. Technical report, Department of Electrical and Computer Engeneering, Department of Computer Science University of Pittsburgh.

Seminar Hochleistungsrechner - Aktuelle Trends und Entwicklungen
Wintersemester 2017/2018

Maschinelles Lernen im HPC

Manuel Rothenberg
Technische Universität München

30.01.2018

Zusammenfassung

Im vorliegenden Paper geht es um die Anwendung von Maschinellem Lernen im Bereich von High Performance Computing (HPC). Es werden grundlegende Methoden im maschinellen Lernen, wie 'Support Vector Machine' und Neuronale Netze vorgestellt. Diese werden anschließend auf verschiedene Bereiche zur Optimierung von HPC angewandt. Dazu gehören frühzeitige Fehlererkennung von Hardware, sowie Optimierungen zur Ausnutzung der vorhandenen Ressourcen, was sowohl Energie als auch Hardware betrifft. Es werden die Vorgehensweise sowie die Resultate in diesen Anwendungen vorgestellt und bewertet. Insgesamt wird gezeigt, dass mit maschinellem Lernen Optimierungen durchgeführt werden können, die sonst ungenutzt geblieben wären.

1 Einführung

Maschinelles Lernen fokussiert sich auf die Algorithmen-Konstruktion um Vorhersagen anhand von Daten zu machen. Dabei wird versucht eine Funktion $f : X \to Y$ zu finden bzw. zu erlernen, welche die Input-Domäne X (Daten) einer Output-Domäne Y (mögliche Vorhersagen) zuordnet. Die Elemente X und Y sind anwendungsspezifische Repräsentationen von Daten-Objekten bzw. Vorhersagen. [11]

Um eine Input- mit einer Outputdomäne mit Hilfe einer Funktion zu verknüpfen gibt es verschiedene Methoden. Die bekanntesten Methoden werden später noch vorgestellt. Dabei ist es jedoch immer wichtig, ein gutes Modell des Anwendungszwecks zu bestimmen. Anhand von diesem Modell kann dann eine tieferliegende Verknüpfung zwischen Input- und Outputdomäne hergestellt werden. Diese Verbindung ist oft nicht einfach über die menschliche Intuition zu finden. Auch Formel-basierte Ingenieursarbeit ist dazu oft nicht in der Lage, da die Anwendungsumgebung oft hoch-komplex und nicht-linear miteinander agiert [7].

Maschinelles Lernen ist daher interessant um Muster zu erkennen und Vorhersagen zu treffen, die z.B. im Bereich des High Performance Computing (HPC) verwendet werden können um Prozesse zu optimieren. Im Bereich von HPC agieren viele Rechner miteinander. Das führt dazu, dass diese Struktur organisiert werden muss um die Rechenkapazität eines Rechenzentrums auch effizient ausnutzen zu können. Dazu gehört unter Anderem das Scheduling, um Tasks auf dem am besten geeigneten Rechner auszuführen, sowie die Kommunikation zwischen Rechnern. Auch die Energienutzung eines ganzen Rechenzentrums muss organisiert werden. In dieser Hinsicht muss z.B. bestimmt werden, ob und welche Rechner hoch- bzw. runtergetaktet werden können. Auch die Kühlung ist ein sehr großer Posten in der Energiebilanz eines Rechenzentrums und kann mit Hilfe von Vorhersagen optimiert bzw.

 DOI: 10.14459/2018md1430700

reduziert werden. Des Weiteren muss auch sichergestellt sein, dass die benötigte Hardware dauerhaft verfügbar ist. In dieser Hinsicht bietet sich eine Vorhersage der Ausfallwahrscheinlichkeit von bestimmten Geräten an, um möglichst schon vor einem Defekt reagieren zu können.
All diese Prozesse können mit Hilfe von Maschinellem Lernen optimiert werden, um die Effizienz eines Rechenzentrums zu verbessern. Dies hilft letztendlich Hardware, Energie und nicht zuletzt Geld zu sparen, sowie die Umwelt durch weniger Ressourcen-Nutzung zu schonen.
Das vorliegende Dokument zielt im zweiten Kapitel darauf ab, Methoden des Maschinellen Lernens vorzustellen. Im dritten Kapitel wird auf die Anwendungsmöglichkeiten zur Optimierung von HPC mit Hilfe dieser Methoden näher eingegangen. Im vierten Kapitel folgt dann eine Zusammenfassung, sowie ein Ausblick.

2 Methoden in Maschinellem Lernen

Maschinelles Lernen kann mit Hilfe von verschiedenen Methoden angewandt werden. Dieses Kapitel soll einige davon vorstellen.

2.1 Überwachtes und unüberwachtes Lernen

In den meisten Fällen geht es beim maschinellen Lernen um die Klassifizierung von Datensätzen. Das System lernt aus vorhandenen Datensätzen und versucht anhand der Daten Muster und Gesetzmäßigkeiten zu erkennen, die auf zukünftige Datensätze angewandt werden können um diese zu klassifizieren. Alle Methoden des maschinellen Lernen können grob in zwei Kategorien aufgeteilt werden.

- 'Supervised Learning' (überwachtes Lernen)
- 'Unsupervised Learning' (unüberwachtes Lernen)

Bei beiden Kategorien erhält das System viele unterschiedliche vorhandene Datensätze mit jeweils vielen verschiedenen Merkmalen, anhand derer es Muster erkennen soll. Das Ziel ist dann, dass jeder Datensatz klassifiziert werden kann.
Der Unterschied besteht darin, dass beim Supervised Learning die Trainings-Datensätze schon klassifiziert sind. Das System kann dann lernen, welche Merkmaleigenschaften zu welcher Klassifizierung führen.
Beim Unsupervised Learning ist diese Klassifizierung der Trainings-Daten nicht vorhanden. Das System muss selbst erkennen, welche charakteristischen Merkmale zusammengehören und welche sich abgrenzen. Abbildung 1 zeigt grafisch den Unterschied zwischen Supervised und Unsupervised Learning. Links sieht man, dass die Datensätze bereits klassifiziert sind und dass das System anhand dieser Daten versucht eine mögliche Trennlinie zu finden mit Hilfe derer dann in Zukunft Datensätze klassifiziert werden können.
Rechts sieht man, dass alle Datensätze als Kreise dargestellt sind, die noch nicht klassifiziert wurden. Die farblich markierte Klassifizierung wird dann während des Unsupervised Learning ermittelt. Hier muss das System selbst herausfinden, welche Datensätze zu welcher Klassifizierung gehören.

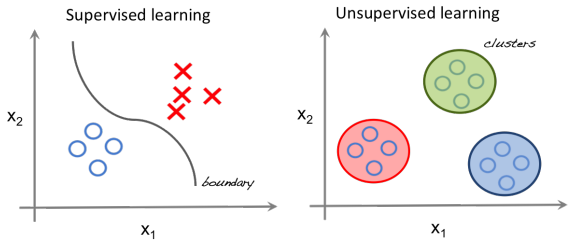


Abbildung 1: Grafische Erklärung von Supervised/Unsupervised Learning [1]

2.2 Support Vector Machine

Support Vector Machines (SVM) gehören zu den am Meisten genutzten Algorithmen zur Klassifizierung und Regression [11]. Eine SVM dient der

 DOI: 10.14459/2018md1430700

binären Klassifizierung. Die grundlegende Idee ist, eine Linie zu finden, die die Datensätze perfekt in ihre Klassen trennt. Dies soll auch möglich sein, wenn die Datensätze nicht linear trennbar sind. Um dies zu erreichen, muss eine Hyperebene gefunden werden, welche die Datensätze in eine höhere Dimension überführt, in der die Daten trennbar sind [3]. Diese Hyperebene ist mit Hilfe von Stützvektoren definiert. Um eine Balance zwischen korrekter und falscher Erkennung zu schaffen, kann der Algorithmus belohnt oder bestraft werden, je nachdem ob eine Klassifizierung korrekt oder falsch war.

2.3 Neuronale Netze

Neuronale Netze werden z.B. in den Bereichen Mustererkennung, Klassifizierung, Prognose und Regelung eingesetzt. Das Funktionsprinzip ist jedoch immer das gleiche [2]. Ein neuronales Netz ist ein System, das einen Datensatz als Eingabevektor erhält. Dieser Eingabevektor besteht aus den zu betrachtenden Merkmalen des Datensatzes. Anhand dieses Eingabevektors ermittelt das neuronale Netzwerk dann einen Ausgabevektor, der die gesuchte Information enthält. Ein- und Ausgabevektor müssen dabei nicht die selben Merkmale enthalten.
Das neuronale Netzwerk selbst besteht dabei aus beliebig vielen Neuronen (Units), welche den Eingabevektor verarbeiten und schließlich auch den Ausgabevektor erzeugen. Abbildung 2 zeigt die unterschiedlichen Arten von Neuronen. Neuronen die übereinander angeordnet sind, werden als Layer (Schicht) zusammengefasst. Zwischen Input-Layer und Output-Layer können mehrere sogenannte Hidden-Layer enthalten sein. Im Beispiel aus Abbildung 2 ist ein Hidden-Layer enthalten. Die Merkmale des Input-Vektors fließen von links nach rechts.

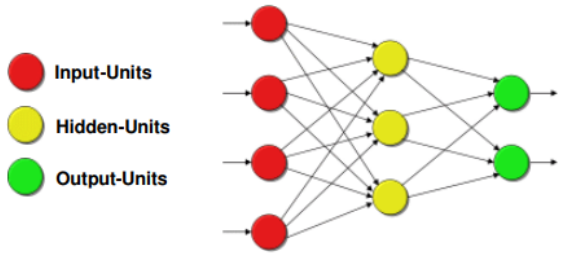


Abbildung 2: Neuronales Netz [13]

Neuronen sind über Kanten miteinander verbunden. Jede Kante hat ein Gewicht welches ausdrückt wie stark der Einfluss des sendenden Neurons auf das empfangende Neuron ist.

- $Gewicht > 0$: erregender Einfluss
- $Gewicht = 0$: kein Einfluss
- $Gewicht < 0$: hemmender Einfluss

Das Gewicht drückt das *Wissen* des neuronalen Netzes aus. Lernen ist demzufolge eine Anpassung des Kantengewichts.
Der Input, den ein Neuron von einem anderen Neuron erhält, hängt von dem Output des sendenen Neurons ab. Dieser Output wird mit dem Gewicht der verbindenden Kante multipliziert. Der gesamte Input eines Neurons wird Netzinput genannt und besteht aus den aufsummierten Inputs von allen Neuronen, die an dieses Neuron senden. [13]
Ein Neuron verarbeitet diesen Netzinput dann mit einer internen Übertragungsfunktion, die einem Netzinput ein Aktivitätslevel zuordnet. Diese Funktion kann z.B. linear, binär oder sigmoid sein. Das Aktivitätslevel wird in den meisten Fällen direkt als Output dieses Neurons verwendet. Abbildung 3 zeigt exemplarisch ein einzelnes Neuron und die

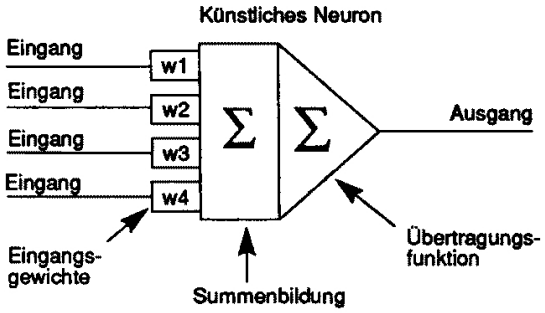


Abbildung 3: Ein einzelnes Neuron [5]

 DOI: 10.14459/2018md1430700

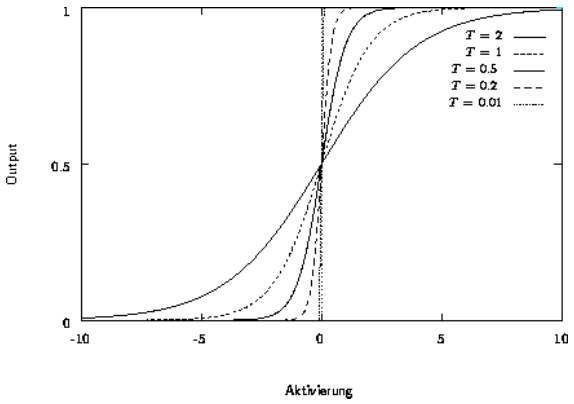


Abbildung 4: Sigmoide Funktionenschar [12]

Verarbeitung der Inputs. Abbildung 4 zeigt exemplarisch einige Sigmoid-Funktionen, die als Übertragungsfunktion dienen können.
Die Kombination vieler Neuronen und deren Verbindung ergibt dann ein neuronales Netz. Dieses Netz muss trainiert werden, was mit Hilfe von Trainingsdaten erfolgt. Diese Phase wird Trainingsphase genannt und kann sowohl in die Kategorie Supervised als auch Unsupervise Learning fallen. Während der Trainingsphase werden die Kantengewichte angepasst um ein funktionierendes Netz zu erhalten. Dies erfolgt mit Hilfe von Lernregeln. Anschließend erfolgt eine Testphase, in der das neuronale Netz nicht mehr angepasst wird, sondern nur noch verwendet wird um Ergebnisse zu validieren.

3 Maschinelles Lernen zur Optimierung von HPC

Nachdem nun ein Grundverständniss für maschinelles Lernen vorhanden ist, wird im folgenden Kapitel näher auf die Anwendung von maschinellem Lernen zur Optimierung von HPC eingegangen. Es werden dabei verschiedene Optimierungsansätze erläutert.

3.1 Ausfallvorhersage/Fehlertoleranz

In großen Rechenzentren wird im Regelfall viel Hardware betrieben. Dies hat zur Folge, dass mehr Geräte ausfallen können. Um den Betrieb nicht zu gefährden, wäre es deshalb von Vorteil, wenn vorhergesagt werden könnte, welche Geräte mit hoher Wahrscheinlichkeit in naher Zukunft ausfallen. Man könnte dann diese Geräte vorsorglich austauschen.
Errin Fulp, Glenn Fink und Jereme Haack haben in ihrem Artikel zum *USENIX Workshop on the Analysis of System Logs* vorgestellt, wie SVMs helfen können, Ausfälle von Festplatten anhand von System-Log Dateien vorherzusagen, siehe dazu auch [8]. Im folgenden Text wird deren Vorgehen näher erläutert.

3.1.1 Vorgehensweise

In großen Cluster-Systemen werden üblicherweise System-Log Einträge gespeichert, die über den Systemzustand und über Systemereignisse Auskunft geben können. Wird für dieses Logging das Unixtool *syslog* verwendet, besteht ein Log-Eintrag unter Anderem aus einer Identifikation des Nodes, einem Zeitstempel, einer Nachricht und einem Tag, der die Priorität der Nachricht repräsentiert. Umso kleiner der Tag-Wert, umso wichtiger ist dabei die Nachricht.
Des Weiteren haben inzwischen fast alle Festplatten *Self-Monitoring Analysis and Reporting Technology (SMART)* integriert. Dies ist ein System, welches Festplattenintern den eigenen Zustand (z.b. Lesefehler, Temperatur, Anlaufzeit, ...) überwacht. Werden bestimmte Grenzwerte überschritten, meldet die Festplatte selbst einen möglichen Fehler an das System, welcher dann auch als Syslog-Eintrag gespeichert wird. Diese Selbstüberwachung kann helfen Fehler zu erkennen, ist aber alleine nicht ausreichend [4].
In Abbildung 5 werden beispielhaft einige Syslog-Einträge gezeigt. Die letzte Nachricht mit dem 'Tag' 1 (höchste Priorität) zeigt z.B. einen Festplattenausfall. Für die SVM werden diese Daten aufbereitet. Das heißt, die Syslog-Einträge werden in Sequenzen der Länge k zusammengefasst. Anschließend werden die Häufigkeit der Tags pro Sequenz daraus extrahiert, so dass Paare von *tag:anzahl* entstehen. Aus dem Beispiel aus Abbildung 5, würde

 DOI: 10.14459/2018md1430700

Host	Facility	Level	Tag	Time	Message
198.129.8.6	local7	notice	189	1171061732	sysstat
198.129.8.6	kern	info	6	1171061732	kernel md: using maximum available idle IO bandwidth
198.129.8.6	cron	info	78	1171061733	crond 2500 (root) CMD (/usr/lib/sa/sa1 1 1)
198.129.8.6	auth	info	38	1171062445	rsh(pam_unix) 2215 session opened for user by (uid=0)
198.129.8.6	auth	info	38	1171062445	in.rshd 2216 root@hpcs2.cs.edu as root: cmd=/root/temps
198.129.8.6	daemon	info	30	1171062590	smartd 88 Device: /dev/twe0 SMART Prefailure Attribute
198.129.8.6	auth	notice	37	1171062590	sshd(pam_unix) 12430 auth failure; logname=el-fork-o
198.129.8.6	kern	info	6	1171062590	kernel md: using 512k, over a total of 12287936 blocks.
198.129.8.6	cron	info	78	1171062601	crond 2500 (root) CMD (/usr/lib/sa/fork-it 1 1)
198.129.8.6	kern	alert	1	1171062692	kernel raid5: Disk failure on sde1, disabling device

Abbildung 5: Syslog Auszug [8]

dies also so aussehen:

$(1:1\ ,\ 6:2\ ,\ 30:1\ ,\ 37:1\ ,\ 38:2\ ,\ 78:2\ ,\ 189:1)$

Dies sind die aggregierten Klassifizierungs-Merkmale. Des Weiteren werden nicht nur diese Merkmale, sondern auch die zeitliche Reihenfolge der Tags innerhalb einer Sequenz betrachtet. Dazu werden die Syslog-Einträge anhand des Tags in beispielsweise drei Prioritäts-Kategorien eingeteilt.

- $tag < 10$: hoch (0)
- $10 <= tag <= 140$: medium (1)
- $tag > 140$: niedrig (2)

Das Beispiel aus Abbildung 5 hätte somit die angepasste Sequenz:

$$(2, 0, 1, 1, 1, 1, 1, 0, 1, 0)$$

Mit $k = 10$, gibt es $3^{10} = 59.049$ mögliche angepasste Sequenzen, daher sollte k reduziert werden, um weniger mögliche angepasste Sequenzen zu erhalten. Anschließend kann jeder möglichen angepassten Sequenz eine ID zugeordnet werden um dann die Häufigkeit der angepassten Sequenz innerhalb einer Sequenz von Sequenzen zu bestimmen. Dies ist die Spektrum-Repräsentation, welche als Spektrum-Klassifizierer verwendet wurde.
Wenn in den Syslog-Einträgen ein Festplattenausfall auftritt, können dann die vorhergehenden Syslog-Einträge herangezogen und wie beschrieben aufbereitet werden. Die so entstandenen Vektoren können dann klassifiziert werden. Mit Hilfe dieser aufbereiteten Vektoren (Tag-Häufigkeit in einer Sequenz, Sequenz-Häufigkeit in einer Sequenz von Sequenzen) sowie der Klassifizierung *Fehler aufgetreten*, können dann SVMs trainiert werden. Während des SVM-Trainings sollten aber ebenso viele Sequenzen einbezogen werden, die keinen Fehler verursachen, anonsten fehlt die zweite Klassifizierung *kein Fehler aufgetreten*. Die SVMs versuchen dann eine Hyperebene zu finden, welche die Vektoren entsprechend der Klassifizierung trennt. Mit Hilfe der entstehenden SVMs, können dann zukünftige Syslog-Eintragssequenzen klassifiziert werden um Ausfälle frühzeitig zu erkennen.

3.1.2 Ergebnisse

Über einen Zeitraum von 24 Monaten wurden auf einem 1024-node Linux-Cluster Syslog-Einträge gesammelt. Dabei traten 100 eindeutige Festplatten-Ausfälle auf. Die jeweils 1200 Syslog-Einträge die vor einem Fehler auftraten, wurden dann aufbereitet und als Input einer SVM herangezogen. Um beide Seiten der Klassifizierung zu erhalten wurden ebensoviele Syslog-Einträge herangezogen, die zu keinem Fehler führten. Die Hälfte der gesamten Syslog-Einträge wurde fürs Training der SVM verwendet, die andere Hälfte für Tests der trainierten SVM.
Um eine bestmögliche Trainigskonfiguration zu finden, wurden mehrere Trainingsdurchläufe durchgeführt, die jeweils einen Teil M der 1200 Einträge verwendeten. Das Subset startet dabei immer beim ersten Eintrag der 1200 Einträge. M lag zwischen 400 und 1100. Bei $M = 1100$ ver-

bleiben z.B. nur noch 100 Einträge (durchschnittlich ca. 30 Stunden) bis der Fehler auftritt. Außerdem wurden sowohl ausschließlich aggregierte Klassifizierungs-Merkmale als auch eine Kombination aus aggregierten sowie auch Spektrum-Klassifizierungsmerkmalen verwendet.

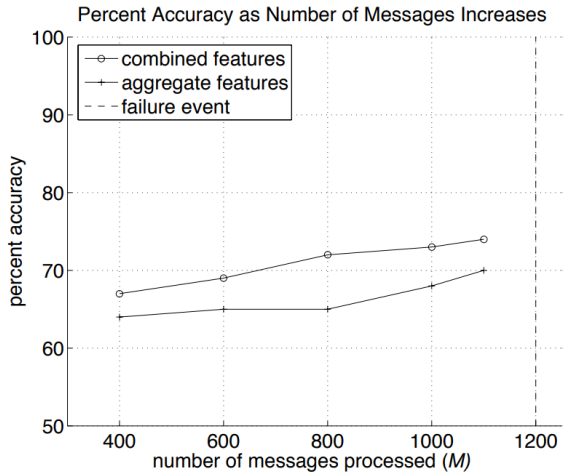


Abbildung 6: Vorhersagegenauigkeit der SVMs [8]

Abbildung 6 zeigt die Genauigkeit, mit der die SVMs Festplattenausfälle vorhersagen konnten. Es zeigt sich, dass bei $M = 1100$ die Genauigkeit am Besten ist, was aber zu erwarten war, da mehr Daten zu besseren Modellen führen. Außerdem wird gezeigt, dass eine Kombination aus aggregierten sowie auch Spektrum-Klassifizierungsmerkmalen besser ist, als nur die aggregierten Klassifizierungsmerkmale. Insgesamt konnte das System bei $M = 1100$ und einer Kombination der Merkmale eine Genauigkeit von 74% erreichen, bei $M = 1000$ eine Genauigkeit von 73%. Das heißt das 100 Syslog-Nachrichten bevor ein Festplatten-Ausfall auftritt, die SVM zu 74% dieses Ereignis vorhersagt.
Abbildung 7 zeigt, dass die SVMs auch gute Sicherheit bezüglich falscher Fehlermeldungen bietet. Die Kurven wölben sich alle nach oben links. Der Punkt (0/1) oben links würde eine 100% Trefferquote bedeuten. Die Fläche unter der Kurve (AUC) wäre dann genau $AUC = 1$, was dem besten Fall entsprechen würde. $AUC = 0,5$ würde einer zufälligen Klassifizierung entsprechen. Bei kombinierten Merkmalen und $M = 1000$, liegt dieser Wert jedoch schon bei $AUC = 0,79$.

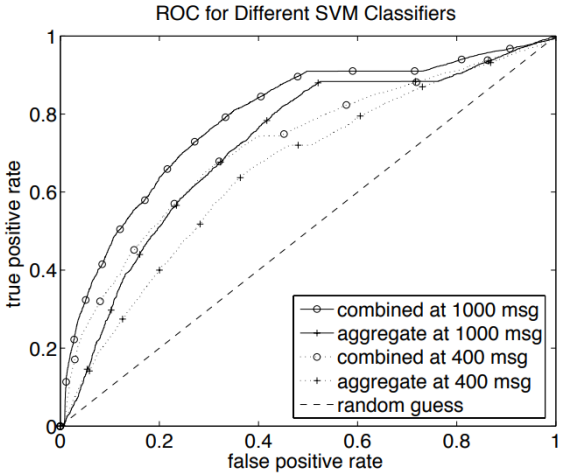


Abbildung 7: ROC Kurve der SVMs [8]

Insgesamt ist dieses Verfahren vielversprechend und kann helfen Festplatten-Ausfälle rechtzeitig vorherzusagen. Das System könnte noch erweitert werden, indem noch mehr Merkmale zur Klassifizierung hinzugezogen werden. Außerdem kann dieses Verfahren nicht nur auf Festplatten angewandt werden, sondern überall wo Daten gesammelt werden.

3.2 Energieoptimierung

Wie eingangs erwähnt, kann maschinelles Lernen verwendet werden um Rechenzentren in Hinsicht auf die Energieeffizienz zu optimieren. Um die Effizienz eines Rechenzentrums zu bewerten, wird von Google, aber auch von vielen anderen Institutionen das Power usage effectiveness (PUE) Verhältnis verwendet [6]. Diese Zahl ist das Verhältnis von insgesamt im Rechenzentrum verbrauchter Energie zur Energieaufnahme der Rechner. Umso näher der PUE-Wert sich 1 nährt, umso effektiver ist das Rechenzentrum.

$$PUE = \frac{gesamte\ Energieaufnahme}{Energieaufnahme\ der\ Rechner}$$

 DOI: 10.14459/2018md1430700

Ein hoher PUE-Wert bedeutet, dass viel Energie für Wärme und Wärmeabfuhr verbraucht wird.

Google gelang es mit Hilfe von maschinellem Lernen, den PUE eines Rechenzentrums um 15% zu senken, was einer Energieeinsparung bei der Kühlung von 40% entspricht [7]. Dies wurde durch das Training eines neuronalen Netzwerks ermöglicht. Trainingsdaten hierbei waren Aufzeichnungen von Kühlwassertemperaturen, Energieverbrauch, Pumpengeschwindigkeit, Anzahl und Art der Kühlsysteme, Wetterbedingungen, ... Mit Hilfe dieser Trainigsdaten konnte das Netzwerk den zu erwartenden PUE mit einer Genauigkeit von 0,4% bestimmen. Außerdem wurden zwei weitere neuronale Netz trainiert, um die zukünftige Temperatur sowie Auslastung des Rechenzentrums zu bestimmen. Anhand der PUE-Vorhersage konnten dann

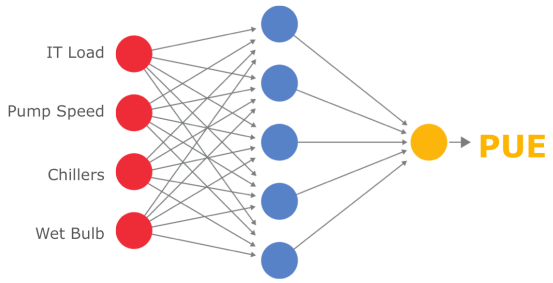


Abbildung 8: Vereinfachtes Neuronales Netzwerk zur Ermittlung des erwarteten PUE [10]

Änderungen am Rechenzentrum vorgenommen werden, die zu einer besseren Nutzung der Ressourcen führt. Mit Hilfe der Temperatur- und Auslastungsvorhersage kann außerdem sichergestellt werden, dass die Änderungen im anwendbaren Rahmen stattfinden, ohne das Hardware überstrapaziert wird. Des Weiteren ist die Möglichkeit der Rechenzentren-Simulation ein Anwendungszweck. So können Parameter modifiziert werden, um die dadurch folgenden Auswirkungen zu bestimmen, denn im realen Betrieb kann nicht einfach die Konfiguration beliebig modifiziert werden.

3.2.1 Vorgehensweise

Im folgenden Text wird näher darauf eingegangen, wie Google dieses neuronale Netzwerk entwickelt hat, siehe dazu auch [6].
Die Merkmale, die das neuronale Netzwerk zur Ermittlung des PUE als Input-Vektor erhält, sind so ausgewählt, dass möglichst wenig linear abhängige Merkmale vorhanden sind. Dies reduziert die Trainingszeit und reduziert die Chancen von Overfitting (zu starke Anpassung des Kantengewichts).
Der Trainingsprozess des neuronalen Netzwerks kann in effektiv vier Schritte aufgeteilt werden:
1. zufällige Initialisierung
Die Gewichte der Kanten werden zufällig initialisiert. Wichtig hierbei ist, dass die Katen nicht mit 0 initialisiert werden. Würden alle Kanten mit 0 initialisiert werden, hätte dies zur Folge, dass die nachfolgenden Schritte keine Anpassungschance erhalten, da alle Werte immer mit 0 multipliziert werden würden. Aus diesem Grund werden die Kanten mit zufälligen Werten im Bereich $[-1, 1]$ initialisiert.
2. Forward Propagation - Algorithmus anwenden
In diesem Schritt wird schichtweise der Netzinput jedes Neurons berechnet, da jede Schicht auf den Outputs der vorhergehenden Schicht basiert. Es wird dabei eine sigmoide Übertragungsfunktion g(z) implementiert, die das Feuern eines biologischen Neurons immitiert.

$$g(z) = \frac{1}{1 + e^{-z}}$$

Nachdem alle Schichten berechnet wurden, enthalten die Output-Neuronen konkrete Werte.
3. Kostenfunktion berechnen
Die Kostenfunktion ist üblicherweise der quadratische Fehler zwischen vorhergesagtem und tatsächlichem Output. Diese Größe sollte mit jeder Iteration kleiner werden.
4. Back Propagation - Algorithmus anwenden
Der Wert der Kostenfunktion wird nun schichtweise rückwärts durch das Netzwerk geleitet. Dabei wird jedes Kantengewicht anhand des mitverschuldeten

 DOI: 10.14459/2018md1430700

Fehlers angepasst.

5. Iteration

Die Schritte 2-4 müssen solange wiederholt werden, bis das neuronale Netzwerk konvergiert oder die maximale Anzahl an Iterationen erreicht ist.

Googles Neuronales Netzwerk enthält fünf versteckte Schichten mit jeweils 50 Neuronen. Als Regulierung wird ein Error-Grenzwert von 0,001 festgelegt. Die Input-Datensätze enthalten 19 Merkmale, der Output-Vektor nur eines, den PUE. Es stehen 185.435 Datensätze zur Verfügung, von denen 70% zu Trainings- und 30% zu Validierungszwecken verwendet werden. Da die Input-Merkmale in ihrem Wertebereich sehr unterschiedlich sind, werden diese zuvor auf einen Wertebereich zwischen $[-1, 1]$ normalisiert.

$$z_{NORM} = \frac{z - MEAN(z)}{MAX(z) - MIN(z)}$$

z ist der Input-Vektor und z_{NORM} der normalisierte Input-Vektor.

3.2.2 Ergebnisse

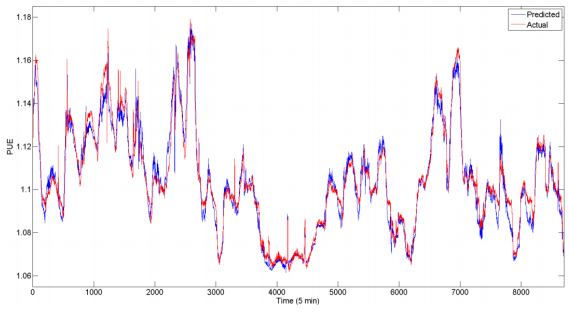


Abbildung 9: Vorhergesagter und tatsächlicher PUE [6]

Abbildung 9 zeigt in blau den vom neuronalen Netzwerk vorhergesagten PUE im Vergleich zum roten tatsächlichen PUE eines Google Rechenzentrums. Der mittlere absolute Fehler liegt bei 0,004 und die Standardabweichung bei 0,005. Generell ist der Fehler höher bei PUE-Werten größer als 1,14.

Mit Hilfe des erstellten Neuronalen Netzwerks zur PUE-Vorhersage konnten diverse Simulationen durchgeführt werden um einzelne Parameter zu optimieren. Ein Beispiel hierfür ist die Erhöhung des Temperatur-Schwellwerts des verlassenden Kühlwassers um ca. 1,6 Grad Celsius, was zu einem um durchschnittlich 0,005 reduzierten PUE führte.
Die PUE-Vorhersage kann auch zum aufspüren von Fehlerquellen im laufenden Betrieb genutzt werden, indem Anomalien aufgezeigt werden. Beispielsweise hat Google im Jahr 2011 Gas als Energiequelle zum Betreiben eines Rechenzentrums hinzugezogen. Einige Sensoren meldeten den Gaszufluss mit Impulsen alle 1000 Gaseinheiten, andere alle 100 oder sogar bei jeder Gaseinheit. Die Rechenzentrum-Betreiber konnten diesen Fehler feststellen, weil der reale PUE im Vergleich zum vorhergesagten PUE um bis zu 0,1 höher war, sobald Gas als Energiequelle verwendet wurde.
Ein weiteres Beispiel der Nützlichkeit von PUE-Simulationen hat sich gezeigt, als Google ein Datenzentrum erweiterte und dadurch während der Arbeiten ein Teil des Server-Traffics umgeleitet werden musste. Durch PUE-Simulationen konnte eine Konfiguration für diese Situation gefunden werden, die den PUE um 0,02 senkte.
Viele derartige Optimierungen summieren sich schnell auf und resultieren in einer großen Einsparung von Energie und somit Geld und nicht zuletzt auch einer besseren Umweltbilanz.
Anhand dieser Beispiele lässt sich schon erkennen, dass Maschinelles Lernen zu Optimierungszwecken sehr nützlich sein kann. Allerdings müssen die ausgewählten Input-Merkmale sorgfältig ausgewählt werden. Außerdem müssen viele Datensätze zur Verfügung stehen um ein präzises Modell zu erhalten.

3.3 Scheduling

Die bisherigen Anwendungszwecke von maschinellem Lernen zur Optimierung von HPC bezogen sich mehr auf die Hardware. Maschinelles Lernen kann aber auch zur Optimierung der Software verwendet werden. An der Universität von Barcelona wurde dazu im Bereich CPU-Scheduling maschinelles

 DOI: 10.14459/2018md1430700

Lernen verwendet [9]. Scheduling bestimmt, zu welcher Zeit und auf welchem Rechner ein Task läuft. Um das Scheduling so weit wie möglich auf energiesparende Taskzuteilung zu optimieren, ist es von Vorteil, wenn der Ressourcenverbrauch des Tasks bereits vor Ausführung bekannt ist. Außerdem ist in dieser Hinsicht wichtig, dass alle Rahmenbedingungen der Ausführung eingehalten werden und der Task innerhalb der vorgegebenen Zeit beendet wird und dauerhaft genügend Ressourcen zur Verfügung hat (Service Level Agreement SLA).
Beide dieser Parameter sind vor Ende der Taskausführung nicht bekannt, werden aber benötigt um ein optimales energiesparendes Scheduling erreichen zu können. Deshalb wurde an der Universität von Barcelona mittels vorhandener Daten ein Maschinenlern-System entwickelt, welches genau diese zwei Größen anhand von Taskparametern vorhersagt. Das System verwendet an dieser Stelle den M5P-Lernalgorithmus, welcher eine abschnittsweise lineare Funktion erzeugt. Mittels dieser Vorhersagen werden dann bekannte Scheduling-Algorithmen verwendet um Tasks auf dem bestmöglichen System zu starten oder einen Task auch auf ein anderes System zu verschieben.
Dieses Scheduling wurde dann mit demselben Scheduling-Algorithmus verglichen, der allerdings vom Benutzer vorbestimmte Parameter anstatt der Vorhersagen verwendete. Es hat sich gezeigt, das bei vielen unterschiedlichen Tasks, der Scheduler mit Vorhersagen vom Maschinenlern-System ca. 24% weniger Rechen-Nodes benötigte um die selben Tasks zu verteilen. Bei vielen sehr ähnlichen Tasks, hat der Scheduler mit vom Benutzer bestimmten Parametern besser abgeschnitten und ca. 30% weniger Rechen-Nodes benötigt um dieselben Tasks zu verteilen [9]. Allerdings war dieses Ergebnis zu erwarten, da homogene Tasks sehr gut verteilt werden können und Vorhersagen erst nützlich werden, wenn die Tasks sehr unterschiedlich sind.

4 Zusammenfassung

Nachdem nun mehrere Anwendungszwecke für maschinelles Lernen im HPC vorgestellt wurden, lässt sich sagen, dass maschinelles Lernen in diesem Bereich sehr hilfreich sein kann. In allen aufgezeigten Bereichen konnte ein Nutzen erzielt werden, welcher HPC effektiver macht. Dies kann die Vorhersage von Ausfällen sein, was hilft, fehlerhafte Komponenten rechtzeitig auszutauschen um die Verfügbarkeit zu erhöhen. Aber auch die Rechenzentren-Konfiguration bzw. -Steuerung kann optimiert werden um die vorhandenen Ressourcen besser zu nutzen. Auch die Task-Verteilung innerhalb des Systems kann verbessert werden, um effektiv mehr Rechenleistung zu erhalten.
Mit Hilfe von maschinellem Lernen können *Stellschrauben* angepasst werden, an denen mit menschlicher Intuition oder auch mit Formel-basierter Ingenieursarbeit evtl. niemals etwas verändert werden würde. Maschinelles Lernen hilft dabei, tiefere Verbindungen zwischen Merkmalen zu entdecken. Allerdings funktioniert dieser Prozess nicht von alleine. Die zu verwendenden Trainings-Merkmale müssen sorgfältig ausgewählt und auf einander abgestimmt werden, um brauchbare Resultate zu erhalten.
Bei sorgfältiger Anwendung sind jedoch noch viele weitere Optimierungen der vorgestellten Anwendungszwecke möglich. Auch viele weitere Anwendungen sind denkbar, da maschinelles Lernen überall angewandt werden kann, wo Daten vorhanden sind, aus denen man lernen kann.

Literatur

[1] Overview of Data Science, 2016. http://beta.cambridgespark.com/courses/jpm/01-module.html, Technischer Report, abgerufen Dezember 2017.

[2] W. Bibel, A. Scherer, and R. Kruse. *Neuronale Netze: Grundlagen und Anwendungen*. Computational Intelligence. Vieweg+Teubner Verlag, 2013.

[3] Dustin Boswell. Introduction to support vector machines. August 2002. Artikel.

[4] Wolf-Dietrich Weber Eduardo Pinheiro and Luiz André Barroso. Failure trends in a large disk drive population. *5th USENIX Conference on File and Storage Technologies*, 2007.

[5] Thorsten Schmidt David Fuchs. Neuronale Netze Das biologische Vorbild. http://www.geosimulation.de/methoden/neuronale_netze.htm, Website, abgerufen Dezember 2017.

[6] Jim Gao. Machine learning applications for data center optimization. 2014. https://static.googleusercontent.com/media/www.google.com/en//about/datacenters/efficiency/internal/assets/machine-learning-applicationsfor-datacenter-optimization-finalv2.pdf, Artikel.

[7] Richard Evans Jim Gao. DeepMind AI Reduces Google Data Centre Cooling Bill by 40%, 2016. https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/, Technischer Report, abgerufen Dezember 2017.

[8] Errin W. Fulp Glenn A. Fink Jereme N. Haack. Predicting computer system failures using support vector machines. *USENIX Workshop on the Analysis of System Logs*, 2008. http://static.usenix.org/legacy/events/wasl08/tech/full_papers/fulp/fulp.pdf.

[9] Josep Ll. Berral Íñigo Goiri Ramón Nou Ferran Julià Jordi Guitart Ricard Gavaldà Jordi Torres. Towards energy-aware scheduling in data centers using machine learning. *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, 2010. https://upcommons.upc.edu/bitstream/handle/2117/7182/eenergy10.pdf.

[10] Joe Kava. Better data centers through machine learning. https://googleblog.blogspot.de/2014/05/better-data-centers-through-machine.html, abgerufen Dezember 2017.

[11] Ron Bekkerman Mikhail Bilenko John Langford. *Scaling Up Machine Learning - Parallel and Distributed Approaches*. Cambridge University Press, 2012.

[12] Uli Middelberg. Künstliche Neuronale Netze, 1995. http://www2.inf.uos.de/papers_html/dipl_95_uli/bp2.html, Diplomarbeit, abgerufen Dezember 2017.

[13] G.D. Rey and K.F. Wender. *Neuronale Netze: eine Einführung in die Grundlagen, Anwendungen und Datenauswertung*. Aus dem Programm Huber: Psychologie-Lehrbuch. Huber, 2011.

Hochleistungsrechner - Aktuelle Trends und Entwicklungen
Winter Term 2017/2018

Task-based Programming Models in HPC

Jakob Schneider
Ludwig-Maximilians Universität München

January 19, 2018

Abstract

As programming for large clusters becomes harder and harder, every tool that simplifies the life of an HPC programmer is valuable. Task-based programming (TBP) models are one such tool. This paper describes the basic mechanisms of TBP on shared memory systems and shows how these functionalities are extended to fit distributed memory systems. To this end, a few libraries on shared and distributed memory systems are investigated and their specific features highlighted. To conclude we try to evaluate TBP against other common programming models in terms of performance, efficiency, and scalability.

1 Introduction

With the increasing complexity in high performance computing hardware systems, the software side of things mirrors this intricacy. With threads inside of cores inside of proccesors and the addition of accelerators such as the Xeon Phi co-processor or GPGPUs, application development is getting much harder. Synchronizing all these components is a very complex task (Venkat Subramaniam [15] calls this the *"syncronize and suffer"* model), but thankfully there are advancements that could simplify an HPC programmer's job significantly. Task based programming (TBP) is such a tool.

In TBP, the developer creates tasks with dependencies and the scheduler arranges the task execution, data distribution and synchronization. Tasks can be suspended and resumed on the same or other CPUs, and load balancing is done via work stealing.
TBP can facilitate parallel programming, but it needs to be carefully evaluated, as there are scenarios, in which threads can perform better than tasks. One such scenario may be an application with a lot of blocking (mutexes, waiting for user input). If a task is blocking, the worker thread that this task is assigned to is not able to perform any work while waiting.
The rest of this paper is structured as follows: Section 2 first introduces general features of task based programming models and show specific examples in existing frameworks for shared and distributed memory systems. In Section 3, the different frameworks are evaluated against each other and classic programming models in terms of scalability, performance, and efficiency. Section 4 provides a conclusion about the presented techniques.

2 Task-Based Programming Models

The important features of TBP are the following:

- **Task Spawning:** The programmer can de-

 DOI: 10.14459/2018md1430700

clare parts of code as tasks and these tasks can be spawned, that is, put into a queue of a worker thread and then executed.

- **Dependency Declaration:** Tasks most likely have dependencies, for example results from previous tasks, or the current results are needed by subsequent tasks. These dependencies must be declared, so that the scheduler can arrange the order of execution correctly.
- **Suspension and Resumption of Tasks:** Tasks can be suspended at specific points, for example when waiting for data. Afterwards, the task can be resumed at this point, and can even be transfered to another worker thread for resumption.
- **Load Balancing:** The runtime system schedules the given tasks according to different strategies. If a worker thread has no work, it can steal tasks from other workers.

2.1 Shared Memory

First, we will look at programming models for systems with shared memory to understand what benefits come with task based programming, before looking into how the task model can be implemented in distributed memory systems.

2.1.1 OpenMP 3.0

Task parallelism for OpenMP was proposed in 2006 and made available with the release of OpenMP 3.0 in May 2008. Tasks in OpenMP are defined as follows:
A structured block, executed once for each time the associated task construct is encountered by a thread. The task has private memory associated with it that stays persistent during a single execution. The code in one instance of a task is executed sequentially. [4]
A task is declared by the preprocessor directive
`#pragma omp task [clause]`
followed by a code block. With the clauses the programmer can specify dependencies between tasks, for example stating that a task needs a variable from another task. Furthermore variables can be declared private (initialized inside the task), firstprivate (initialized outside before the task, but then private to each task using it) or shared.
The proposal specifies two important features of tasks:

- **Suspend / resume points:** Tasks can be suspended at these specified points, so that other tasks can be executed first. The suspended tasks will be resumed at the point of interruption. These points are specified with the preprocessor directive

  `#pragma omp taskyield`
- **Thread switching:** If a task is suspended at a suspend point, a different thread can pick it up and continue at the same point.

There are two different types of task barriers in OpenMP 3.0:

- **taskwait** The current task waits for every child task that is generated since the beginning of the current task.
- **taskgroup** The program waits for all tasks in the taskgroup to complete.

2.1.2 Intel Cilk Plus

Intel Cilk Plus adds simple language extensions to C and C++ to allow the programmer to use task and data parallelism. It adds three keywords (`cilk_for`, `cilk_spawn` and `cilk_reduce`), as well as so-called hyperobjects, that simplify shared states between tasks without race conditions or locks.
The keywords are pretty self-explanatory:

- `cilk_for` parallizes loops,
- `cilk_spawn` spawns a task,
- `cilk_sync` waits for all spawned tasks in function before the program continues.

The most common hyperopbject is a *reducer*. The following example computes $\sum_0^N foo(i)$ in parallel:

 DOI: 10.14459/2018md1430700

```
cilk::reducer_opadd<float> result(0);
cilk_for (int i = 0; i < N; i++)
    result += foo(i);
```

Each worker thread has a deque instead of a classic queue. It pushes and pops tasks to and from the tail (and works thus in a LIFO order), which tends to improve locality. Additionally, if a worker tries to steal work from another worker, it does so from the head of the other worker's deque. The tasks at the head of the deque are more likely to be larger and spawn a lot of child tasks, and thus may lead to less overhead created by stealing.
Intel Cilk Plus will be deprecated with the release of the Intel Software Development Tools in 2018 and remain in deprecation mode for two years. Intel recommends migrating to OpenMP or Intel Threading Builing Blocks.

2.1.3 Intel Threading Building Blocks

Intel Threading Building Blocks (or TBB) is a C++ template library that again allows the division of the program in tasks. In contrast to Cilk Plus, TBB offers no keywords, but algorithms (e.g. `parallel_for`), containers (e.g. `concurrent_vector`), constructs for memory allocation (e.g. `scalable_malloc`), synchronization (e.g. `mutex`) and atomic operations (e.g. `fetch_and_add`):
It also provides the Intel Flow Graph to express dependencies and data flow:

```
//create the graph
tbb::flow::graph g;

//create a node
tbb::flow::continue_node
  <tbb::flow::continue_msg>
  h (g, [](const continue_msg &)
    {std::cout << "Hello ";});


//create a second node
tbb::flow::continue_node
  <tbb::flow::continue_msg>
  w (g, [](const continue_msg &)
    {std::cout << "World!";});

//link the nodes
tbb::flow::make_edge(h,w);

//start the first node
h.try_put(continue_msg());

//wait for the whole graph to finish
g.wait_for_all();
```

These nodes fulfill the same purpose as tasks.
There is also a visual graph design tool, called Flow Graph Designer, which lets you create, arrange and connect nodes. The Flow Graph Designer then generates the C++ code and you just have to fill in the content of the nodes.

2.1.4 Task Parallel Library

The Task Parallel Library (TPL) is a library for .NET. It provides types and functions that facilitates exposing parallelism in a program. It makes heavy use of generics and delegates. Leijen et al. [13] offer an overview of the design and implementation of the TPL.
Similar to Cilk Plus, it introduces tools to parallelize loops (`Parallel.For()`) and create tasks or futures. It also introduces the `Interlocked` class, that provides function for thread-safe incrementing, decrementing or adding a value to a variable. Fork-join parallelism is realized through the `Parallel.Do()` method.
The task queues, which are double ended like in Cilk Plus, are mostly lock free. Only if there are not enough tasks in a queue for popping and possible work-stealing, the local thread takes a lock. The freedom from locks is bought by adding states (`init`, `running` and `done`) to the tasks, which are set with atomic compare-and-swap operations, to ensure that each task is only run once.

2.1.5 Ordered Read Write Locks

The "Ordered Read-Write Locks"(ORWL)-Model builds upon the concept of read-write locks. Clauss et al. [6] explain, that while read-write locks fulfill their purpose, the programmer often has to take further actions to prevent deadlocks or starvation. The solution proposed by the authors is a synchronization overlay coupled with queues for the locks.

Additionally, handles are introduced, by which the locks are accessed, so a task can request a lock and may continue other work until the lock is granted.

The synchronization overlay is an abstraction of the data dependencies between the tasks. It is generated and optimized at startup and ensures a lock free execution.

Gustedt et al. [9] enrich the ORWL model with an automated system to improve data locality in systems with distributed memory. It utilizes hwloc, a framework presented by Broquedis et al. [5], to obtain information on the hardware topology and generate an allocation strategy, that aims to reduce communication between the nodes and optimise the shared caches in the nodes.

2.1.6 StarPU-MPI

StarPU is a software tool that simplifies the creation of highly parallel applications for heterogeneous multicore architectures. It is presented by Augonnet et al. [3].

It features a runtime support library, which handles the scheduling of its tasks. StarPU tasks consist of a *codelet* (computational kernel for a worker, can be a CPU, CUDA or OpenCL device), a data set on which the codelet works and information on its data dependencies. Task dependencies in StarPU are deduced by data dependencies by default, but can also be specifically set by the programmer.

As the dependencies can be specified, so can the scheduling. The programmer can choose to implement her own algorithms or choose from various designs offered by StarPU. Available queue designs span FIFOs, priority FIFOs, stacks and deques, which can be adapted at will. Data transfers are automatically handled by the StarPU runtime system.

Augonnet et al. [2] augment StarPU with MPI, which allows its use on distributed systems, but shows, that the network poses a severe bottleneck, but that could change as newer CUDA drivers enable the direct communication between the GPU and and the network cards.

2.2 Distributed Memory

In this section, we will take a look at frameworks that are specifically designed for systems with distributed memory and use the task based programming model.

2.2.1 Charm++

Charm++ is a programming language based on C++ that provides high level abstractions to simplify the development of parallel programs. Programs written in Charm++ can be executed unchanged on systems with shared and distributed memory. Kale et al. [12] presents the design principles behind Charm++:

Portability: Charm++ lays a heavy focus on portability, and it achieves it through tasks, but they are called `Chares`. Chares have the ability to create other chares and communicate with them.

Latency Tolerance: Instead of blocking-receive-based communication, Charm++ uses message driven execution, meaning all computations are initiated by received messages. Futures are supported as well, and while some chare may need to wait on data, others can be executed.

Object Orientation: Charm++ consists of the following classes of objects:

- **Sequential Objects** give the programmer the choice of having non-parallelized parts of the program.
- **Concurrent Objects** are the chares.
- **Shared Objects** contain data of any type, and can be distributed data structures. There are different classes of shared objects for example read-only, accumulator objects or distributed tables.
- **Communicaton Objects** describe the message entities.

Several load balancing and memory management strategies are available: *random*, *share with neighbours*, *central manager/scheduler* and *prioritized task creation*.

 DOI: 10.14459/2018md1430700

Zheng et al. [17] introduce FTC-Charm++, which extends Charm++ with a fault tolerance protocol based on checkpoints. The reason why it is based on checkpoints rather than on logging mechanisms is explained by a look to the evolution of applications that need fault tolerance at all. In the past these applications were mission-critical programs that under no circumstances should fail. Efficiency loss or overhead imposed by the fault tolerance techniques were neglectable, as long as the application didn't crash.
In most of today's high performance computing, a strong focus is set on efficiency. A computation, that runs for hours or days, fails and has to be restarted again, costs a lot of time and with it energy. Logging techniques entail a lot of overhead on communication, which again makes the application less efficient.
Zheng et al set the following requirements for a fault tolerance system in high performance computing:

- No reliance on fault-free storage
- Low impact on fault-free run time
- Low recovery time after a crash
- High execution efficiency after a restart (and maybe a loss of a processor)

The proposed solution is based on *double in-memory checkpointing* or *double in-disk checkpointing*. The checkpoints are saved in the local memory (or on a local disk) of a process instead of a central storage location. Additionally, every processor gets a *buddy processor*, and both their checkpoints are saved in both their memories or disks. That way, the system is not dependent on fault-free storage. In-memory checkpointing, opposed to in-disk checkpointing, ensures the low impact on fault-free run time.
To maintain a high efficiency after a fault, the task based model of Charm++ plays an important role, as the work of lost processors can be dynamically load balanced onto the remaining ones. Figure 1 shows the performance of LeanMD (a molecular dynamics simulation program) after a crash with and without load balancing.

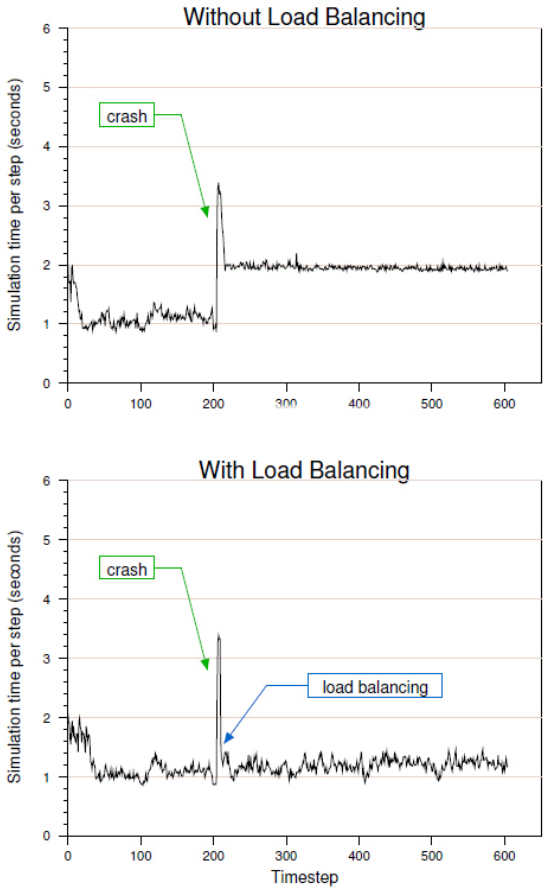


Figure 1: Performance with and without load balancing after a crash [16].

2.2.2 High Performance ParalleX

Kaiser et al. [11] introduce High Performance ParalleX (HPX) as a general purpose C++ parallel runtime system. The paper describes HPX as follows:

HPX represents an innovative mixture of a global system-wide address space, fine grain parallelism, and lightweight synchronization combined with implicit, work queue based, message driven computation, full semantic equivalence of local and remote execution, and explicit support for hardware accelerators through percolation.

Design Principles In this paper, four main factors are identified, that prevent scaling. They are appropriately named the **SLOW** factors: **S**tarvation, **L**atencies, **O**verheads and **W**aiting for contention resolution. To reduce these factors, the following design principles are established:

- **Latency Hiding instead of Latency Avoidance:** It is pointed out, that latency is tightly related to an operation waiting on a resource. Instead of focusing on minimizing this waiting time, it could be employed to do useful work.

- **Fine-grained Parallelism instead of Heavyweight Threads:** This is closely related to Latency Hiding, as even very small latencies can be used to do work, but only if context switching is fast enough to gain performance.

- **Avoid Global Barriers:** The paper mentions, that many common operations, such as loops parallelized with OpenMP, entail global barriers. But usually there is no need to wait on all iterations to finish before doing other work.

- **Adaptive Locality Control:** Static data distribution may lead to inhomogenous workloads across the system, therefore an dynamic data placement system is required. HPX implements a global, uniform adress space and a decision engine that is able to distribute data at runtime.

- **Move Work instead of Data:** As operations are commonly much smaller than the data required to perform them, the work should be moved to the data and not the other way around.

- **Avoid Message Passing:** When two threads communicate, they have to synchronize, in other words wait for each other, therefore message driven computation should be prefered.

Subsystems: HPX's architecture consists of five subsystems, each of which encapsulates a specific functionality.

- **The Parcel Subsystem:** The parcel subsystem handles network communication. Parcels are a form of active messages, which means that they are capable of performing processing on their own. In HPX, a parcel contains the global address of its destination object, a reference to one of the object's methods, the arguments for the method and optionally a continuation. The communicating entities are called *localities*. A locality transmits and receives parcels asynchronously.

- **The Active Global Address Space:** The AGAS constructs a global address space for all localities the application currently runs on. Opposed to partitioned global address space, the AGAS is dynamic and adaptive, it evolves over the lifetime of an application.

- **The Threading Subsystem:** When a locality receives a parcel, it is scheduled, run and recycled by the threading subsystem. Context switching between HPX threads does not require a kernel call, so that millions of threads can be executed per second on each core. There is usually one OS-thread per core, that the HPX threads are mapped onto. The threading subsystem uses work-stealing for intra-locality load balancing and threads are never interrupted by the scheduler, but can suspend themselves when waiting for data.

- **Local Control Objects:** LCOs handle parallelization and synchronization, they provide means of shared resource protection and execution flow organization. Examples for LCOs are `futures`, `dataflow objects`, `mutexes` and `barriers`.

- **Instrumentation and Adaptivity:** HPX implements ways to analyse performance measured by hardware, the OS, HPX runtime services and the application. It can use this data

 DOI: 10.14459/2018md1430700

to adapt the resource management system and therefore increase performance.

To determine if an optimal task size for specific applications exit on the HPX runtime system, Grubel et al. Grubel et al. [8] used an one-dimensional stencil operation, the heat distribution benchmark HPX-Stencil, with varying task sizes. In this benchmark, the calculations for each grid point are wrapped in futures. Additionally, the grid points are partitioned, and these partitions are represented as futures, too. This way, the number of calculations per future can be fine-tuned.
Measurements were taken by the previously mentioned instrumentation. Over the course of the experiments, the total number of grid points was always 100 million, but the size of the partitions ranged from 160 to 100 million points. Taking into account the idle-rate (computation to thread management ratio), thread management overhead and wait time per thread, a clear optimal range for the task size can be determined. Figure 2 shows the different measurements exemplarily on a Haswell 8 core CPU. Wait time in this figure can be negative, because of the way it is calculated: Wait time is defined as the difference between the measured average task duration and the task duration of the same experiment on one core. The time on one core can be higher, because of behaviours such as caching effects [8].
This knowledge can easily be used to statically and potentially even dynamically assign optimal task sizes.

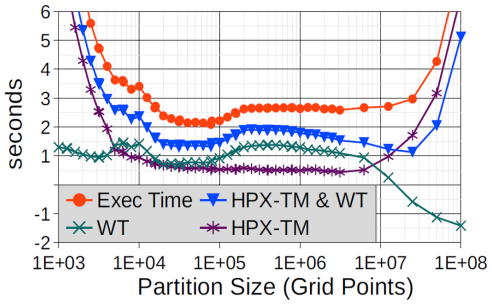


Figure 2: Execution Time (Exec Time), combined thread management and wait time (HPX-TM & WT), wait time(WT), and thread management (HPX-TM) on a Haswell 8 core CPU.

3 Evaluation

In this chapter we will look into several benchmarks on different kinds of systems to compare scalability, performance and efficiency.

3.1 Scalability

Scalability is very important for applications, otherwise the application needs to be adjusted to every system, or maybe even completely rewritten.

The first interesting benchmarking result is from the work of Kaiser et al. [11]. They used the Homogeneous-Timed-Task-Spawn benchmark on a Intel Ivybridge system with two sockets, and evaluated HPX against Qthreads and TBB. The result show a clear performance drop at the socket boundary for Qthreads and TBB (see Figure 3). 2.5 million tasks were user for each core, and these tasks did no work and returned immediately, simulating very fine grained tasks.

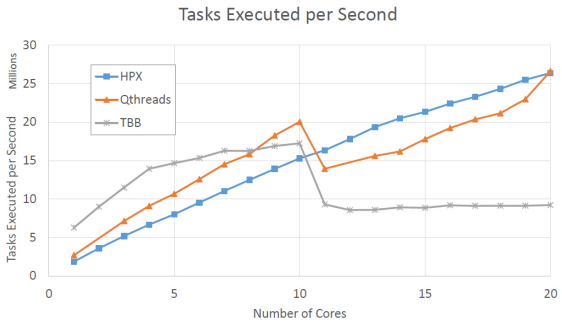


Figure 3: Tasks executed per second on a two-socket Intel Ivybridge system [11].

Another interesting benchmark was conducted by Bryce Adelstein-Lelbach from the Center of Computation and Technology (CTT) at Louisiana State

University. He spawned 500000 tasks on a HP DL785 G6 node with 48 cores on eight sockets[1]. The tasks did not perform any communication and had artificial workloads of 0, 100 and 1000 μs, which exposes the task overheads of each framework. Figure 4 shows, that TBB and HPX are almost tied when tasks are fine grained, but with growing task sizes, HPX is more stable in scaling than TBB.
Note that the links provided to the SWARM library no longer works and no information on this library could be found.

3.2 Performance

Kaiser et al. [11] show that HPX can maintain 98% of the theoretical peak performance of a system on a single node (16 cores) and 87% with 1024 nodes. It outperforms the MPI implementation by a factor of 1.4 (see Figure 5). On a Xeon Phi co-processor, HPX is able to reach 89% of the theoretical peak performance.

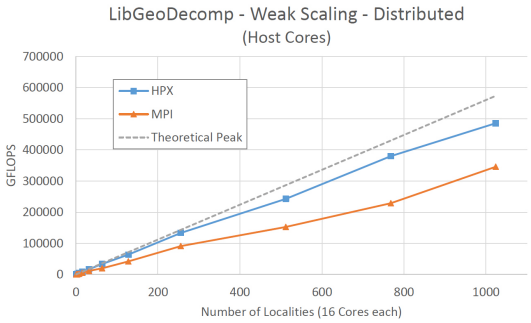


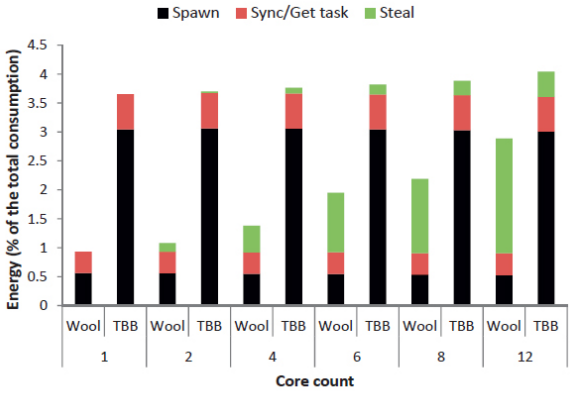
Figure 5: HPX vs MPI in terms of efficiency [11].

Interestingly, Perez et al. [14] mention, that an application using their task based programming model can rival highly tuned libraries in terms of performance, without much tuning effort.
Most other benchmarks evaluate the different task-based programming models among themselves, which is not relevant for this paper, as it tries to measure task-based to classic apporaches.

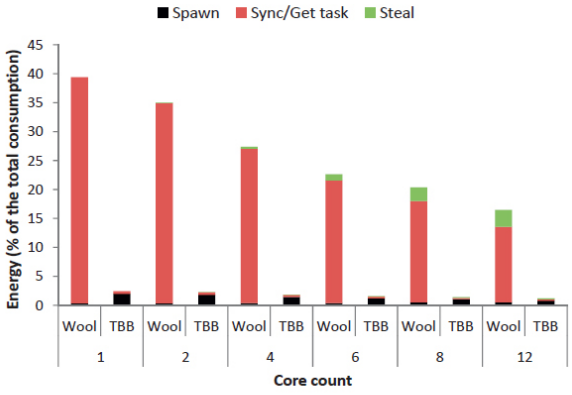
[1] http://stellar.cct.lsu.edu/2012/03/benchmarking-user-level-threads/

3.3 Efficiency

As power consumption has become an increasingly important factor in HPC, it is mandatory to take a look at the efficiency of the applications.
Iordan et al. [10] compare the overheads for different task operations, such as spawning new tasks, stealing and synchronizing. They use TBB and Wool ([7]), an experimental library with the goal of extremely low overhead task creation and synchronization.



(a) Stress



(b) MemStress

Figure 6: Percentage of total energy consumption of different operations in CPU and memory bound situations [10].

 DOI: 10.14459/2018md1430700

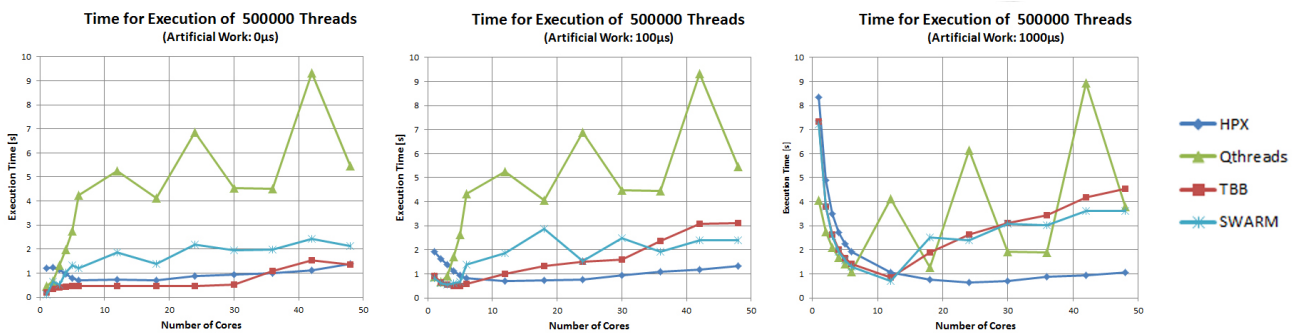


Figure 4: Task overheads for HPX, Qthreads, TBB and SWARM

The results (see Figure 6) show, that Wool surpasses TBB in spawning new tasks, but as soon as the application is bound through memory access, Wool has huge overheads in synchronizing (many workers).
In a third test with matrix multiplications, where tasks are relatively big and few, TBB surpasses Wool in energy efficiency, as Wool has a very aggressive work stealing algorithm, whereas TBB puts workers, that fail to steal tasks several times to sleep and wakes them up later.
Iordan et al. developed FASTA ([10]), a tool that speeds up simulation times by only calculating specified samples. This tool can not be used for all applications, but there are some for which a 12x speedup can be reached with an accuracy error of only 2.6%.
But in the end, Iordan et al. [1] point out, that as the number of cores grows, more power gets consumed by synchronization and management overhead.

4 Conclusion

This paper provides an overview over task-based programming models, for both shared memory and distributed memory systems. First it introduced the common concepts and illustrates the specific implementations of frameworks for shared memory systems, two of which later received support for distributed memory systems. Then it introduced three libraries that were specifically built for distributed memory and emphasizes the differences.
The evaluation showed that, compared to the usual programming models, TBP shows great promise, especially in the field of scalability and ease of use. With minimal optimization effort, performance can rival the common techniques, and as workload distribution is handled by the scheduler, adjustment for different hardware setups may be non-existent.
The presented advantages (scalability, ease of use, and efficiency) show that task-based programming models are a promising tool worth looking into.

References

[1] Iordan Alexandru, Artur Podobas, Lasse Natvig, and Mats Brorsson. Investigating the potential of energy-savings using a fine-grained task based programming model on multi-cores. 2011. QC 20120223.

[2] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In Siegfried Benkner Jesper Larsson Träff and Jack Dongarra, editors, *The 19th European MPI Users' Group Meeting (EuroMPI 2012)*, volume 7490 of *LNCS*, Vienna, Austria, September 2012. Springer.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Ernesto Su, Priya Unnikrishnan, and Guansong Zhang. A proposal for task parallelism in openmp. In *Proceedings of the 3rd International Workshop on OpenMP: A Practical Programming Model for the Multi-Core Era*, IWOMP '07, pages 1–12, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186, Feb 2010.

[6] Pierre-Nicolas Clauss and Jens Gustedt. Iterative Computations with Ordered Read-Write Locks. *Journal of Parallel and Distributed Computing*, 70(5):496–504, 2010.

[7] Karl-Filip Faxen. Wool-a work stealing library. 36:93–100, 01 2008.

[8] P. Grubel, H. Kaiser, J. Cook, and A. Serio. The performance implication of task size for applications on the hpx runtime system. In *2015 IEEE International Conference on Cluster Computing*, pages 682–689, Sept 2015.

[9] J. Gustedt, E. Jeannot, and F. Mansouri. Optimizing locality by topology-aware placement for a task based programming model. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 164–165, Sept 2016.

[10] A. C. Iordan, M. Jahre, and L. Natvig. On the energy footprint of task based parallel applications. In *2013 International Conference on High Performance Computing Simulation (HPCS)*, pages 164–171, July 2013.

[11] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[12] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, October 1993.

[13] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. Atlanta, FL, September 2009. ACM SIGPLAN.

[14] Josep M Perez, Rosa M Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151. IEEE, 2008.

[15] Venkat Subramaniam. *Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors.* Pragmatic Bookshelf, Dallas, Tex., 2011.

[16] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *SIGOPS Oper. Syst. Rev.*, 40(2):90–99, April 2006.

[17] Gengbin Zheng, Lixia Shi, and L. V. Kale. Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, pages 93–103, Sept 2004.

Hauptseminar Hochleistungsrechner: Aktuelle Trends und Entwicklungen
Winter Term 2017/2018

Accelerators for Deep Learning

Mirjam Trapp
Ludwig-Maximilians-Universität München

19.01.2018

Abstract

Deep learning has become an increasingly important topic in computer science, accelerators aiming at improving performance of deep learning applications have become a subject of interest. In this paper, we are taking a look at three different current accelerators considered to be representative of general directions for accelerators in deep learning, namely NVIDIA Tesla GPU, Google TPU and IBM TrueNorth and try to draw some comparison between them. We compare these three regarding their usability in deep learning and conclude that each has its usefulness for different aspects which reflects their primary design point. However, those three accelerators are built for different purposes and one cannot simply replace one of them by one of the others.

1 Introduction

Artificial Intelligence in general and the related topic of machine learning have lately been a big topic due to increasing technological possibilities, even though the subject itself has been discussed for decades.
A specific approach to artificial intelligence which is recently becoming an item of discussion due to new implementations is artificial neural networks.
Deep Learning can generally be seen as a subtopic of neural networks, which can in its simplest form be described as a set of connected processors, with those connections typically being weighted [13].
Deep Learning is implicating the existence of a certain hierarchy in the neural network, with different levels also providing different levels of abstraction. As for many topics in computer science, the increasing amount of data accessible to us is contributing to the improvement of prediction regarding statistical methods of computer science. [14]
There are different types of deep learning, the most common one being supervised learning, where the neural network is trained by using a set of data that has already been classified in advance in order to try to adjust its weights to minimize classification errors, most commonly using Stochastic Gradient Descent (SDG). [14]
A slightly different version of unsupervised learning in neural networks are backpropagation architectures, which aim at adjusting the weights of the connections as specifically as possible, tracking the impact of the specific weights on the total error regarding the outcome by going backwards from neuron to neuron, following the connections. This way, it is possible to change only those weights having an actual effect on the undesired outcome. [14]
There are also approaches of unsupervised learning with neural networks, as in having the network

 DOI: 10.14459/2018md1430700

find own rules based on which to separate the given training data. A classic example for this is Hebbian Learning. [16]
Hardware accelerators were originally developed to be tools mainly for faster graphic processing in a computer - which they are still used for today - , since rendering and displaying graphics requires a limited set of instructions to be performed very frequently and could therefore be optimized by running on a specialized device.
They are for the same reason also perfectly suitable for increasing performance of deep learning applications in artificial neural networks, where the same type of operation, namely matrix multiplications, usually has to be performed over and over again.

2 Motivation

Today the most commonly used accelerators for deep learning are GPUs.
As pointed out by LeCun in [14] the first deep learning application that used backpropagation was successfully built in 2009 using GPUs.
Following Moore's Law, the performance capabilities of GPUs have steadily increased over the last decades making them a more useful acceleration tool then ever, without being limited to their original task of fast processing of graphic images anymore.
Among classic GPUs the most well know company would probably be NVIDIA who have also more recently been in focus due to the increasing public interest in self-driving cars. NVIDIA's Tesla GPU range is for instance used in the NVIDIA DGX and NVIDIA Drive PX systems [23] that are used by well-known car companies such as Audi, Mercedes and Tesla.
Their newest accelerator architecture is NVIDIA Volta which was launched in June, 2017.
However, there are new approaches entering the market targeting the field of deep learning with its specific requirements. The Google TPU for example received massive media attention in 2016 when Google's AlphaGo program managed to beat Lee Sedol, one of the world's best Go players [19]. The deep learning processes needed to train the algorithm were reportedly performed using Google TPU [20].
Interestingly enough, Google TPU is an application-specific integrated circuit (short ASIC). In October 2017, it was even reported that Google had developed a new version of AlphaGo that was able to beat its predecessor that had succeeded against Lee Sedol [20].
Another interesting development would be the efforts to make accelerators that are directly inspired by the human brain. There seems to be a general public interest in simulating the functioning of the human brain, presumably partly due to the potential medical uses of such a model. There are several projects that aim at building such simulations, most commonly known the Human Brain Project which is supported by the European Union [21] and the BRAIN Initiative [22] that is funded by the US government.
Technology developed in such efforts, such as neuromorphic chips, are however not necessarily limited to being a part of large-scale brain simulations, but can also quite obviously be used for deep learning tasks in general. One good example for this would be IBM TrueNorth which got developed as part of the DARPA SyNAPSE program, another program aimed at the development of neuromorphic technology in the US [11].
Though not being publicly released yet, results of tests performed by IBM have shown that the chip performs very well on image recognition tasks [17].

3 Methodology

The following chapter provides a deeper insight into three different types of accelerators, examining their architecture, the corresponding software environment and benchmarks.
The accelerators are furthermore compared regarding performance, efficiency and usability. The accelerators as examples for new developments in those three different categories of GPUs, ASICs and neuromorphic chips are NVIDIA Tesla V100, Google TPU and IBM TrueNorth.

 DOI: 10.14459/2018md1430700

This is mainly due to the fact that those three accelerators have received a lot of public attention prior and after their release, being praised as futuristic takes on accelerators.
Conveniently, NVIDIA's Tesla V100, Google TPU and IBM TrueNorth are representing three very different approaches, which makes them interesting to review in comparison to each other, even though a direct comparison may be difficult due to their different architectures and the different objectives they were designed for.

4 Analysis of Accelerators for Deep Learning

The following chapter will discuss the architecture, software environment and benchmarks of NVIDIA V100, Google TPU and IBM TrueNorth and further draw a comparison between them.

4.1 NVIDIA V100

This section will give a description of the architecture, software environments and benchmarks of the NVIDIA V100.

4.1.1 Architecture

NVIDIA V100 is the first accelerator using NVIDIA's new volta architecture and will first be used in the NVIDIA Titan V graphics card that got announced by NVIDIA on December 7th, 2017.
It comes in two different versions depending on the system interface that is used, Tesla V100 PCIe and Tesla V100 SKM2, the first one using PCIe Gen3 and the second one using NVIDIA NVLink. Tesla V100 PCIe comes with 640 NVIDIA tensor cores and achieves an interconnect bandwidth of 32GB/sec, whereas Tesla V100 SKM2 has 5120 CUDA cores and an interconnect bandwidth of 300GB/sec. [8]
According to NVIDIA, the Tesla V100 is highest performing parallel computing processor thus far. Regarding the accelerator's architecture, it consists of multiple GPU processing clusters, each of which is containing seven Texture Processing Clusters, as well as 84 Volta Streaming Multiprocessors and eight 512-bit memory controllers. [7]
According to the authors of the Tesla V100 whitepaper, a main goal when designing the accelerator was to also increase its efficiency, which is why the V100 can run in two different modes: Maximum Performance Mode and Maximum Efficiency Mode, stating that the GPUs could still achieve a potential performance of 75 to 85 percent of its full performance when running in Maximum Efficiency Mode. [7]
A new feature of the V100 compared to previous NVIDIA architectures is the fact that the Streaming Multiprocessor can perform 32-bit floating point operations and 32-bit integer operations in parallel, since it has separate cores for these two purposes. This design was chosen in order to improve deep learning matrix arithmetics. [7]

4.1.2 Software Environment

The compatible software APIs for this GPU include NVIDIA CUDA and OpenCL and it is suited for several deep learning frameworks such as Caffe. [8]
The most commonly known software framework used for NVIDIA GPU programming would be the CUDA toolkit.
CUDA was developed in 2006 and is supposed to be a general-purpose parallel computing platform typically with a software environment using C, though other languages such as FORTRAN and C++ are supported as well [10].
The main objective when creating CUDA was to enable scalable parallelism in order to maximize the overall performance. Therefore, threads are organized in a hierarchical structure, with several threads building a thread block and several thread blocks forming a grid. The premise for thread blocks is that they have to be fully independent from one another, in other to being able to schedule execution flexibly. Within thread blocks, threads coordinate their actions via shared memory. [10]
In a combined CPU andGPU architecture, those threads generally run on a different device (GPU)

than their host (CPU), with device and host typically not sharing any memory. The GPUs accessible to the host are identified via a so-called compute capability, which is essentially a version number, from which the CPU can derive information about the GPU's hardware and therefore what functions are supported by that specific GPU. [10]
Grids of kernel functions that are invoked by the CPU are then distributed for execution onto the GPUs as evenly as possible. Thread blocks are further grouped into warps with regards to the GPU hardware. [10]
Specifically in the Tesla Volta architecture, there is so-called Independent Thread Scheduling, giving each thread its own execution state, program counter and call stack in order to minimize the risk of deadlocks within warps. There is also a feature called schedule optimizer, through which it is for instance possible to build even smaller subgroups of threads within warps. [10]
There are several newer extensions to CUDA, for example Cooperative Groups, which is enabling the user to specify in detail to which degree which thread shall be able to communicate with what other threads. [10]
Another extension is CUDA Dynamic Parallelism, which confers the possibility to create new work tasks on the GPU itself instead of necessarily having to be called by the host [10].

4.1.3 Benchmarks

The peak performance given by NVIDIA is 7 TFLOPs/sec on V100 PCIe and 7.8 TFLOPs/sec on V100 SKM2 regarding double precision as well as 14 TFLOPs/sec on V100 PCIe and 15.7 TFLOPS on V100 SKM2 for single precision, causing a maximum power consumption of 250W on V100 PCIe and 300W on V100 SKM2. [8]
Furthermore, it is stated that the peak performance for training and inference is 125 tensor TFLOPs [7].

4.2 Google TPU

In the following, the architecture, software environments and benchmarks of Google TPU shall be discussed.

4.2.1 Architecture

Google TPU was built by Google in order to have a customized accelerator for neural networks, regarding the increasing need of computational resources regarding Google's own data centers. The main idea was to work with quantization to lower the cost of matrix multiplications in neural networks. Via quantization, floating point values are reduced to 8-bit integers, which makes operations on them computationally much cheaper. [12]
Specifically, quantization maps floating point values to integer values based on the position of that floating point value in the total range of values that the floating point number could have - given a floating point range from -10.0 to 30.0, the value 30.0 would then be mapped to 255 in the integer representation [9].
It should be noted that quantization works for inference - the model classifying some given input based on its internal classification rules - but usually not for training itself, where the aim is to optimize those classification rules. This is due to the general robustness towards errors that the neural network should have when classifying new input. [9]
The accelerator's design is explicitly specialized for neural network prediction, therefore it is using a CISC architecture. Instead of a classic ALU that stores and fetches values from registers, Google TPU uses a systolic array in which several cells are connected to and pass on result values between each other. Each cell can also store its result value without the need of using a register. [12]
As described in [15], Google TPU furthermore consists of a matrix multiply unit, with the weights for the addition and multiplication operations being stored in a weight FIFO.
The TPU communicates with the host CPU over a PCIe Gen3 x16 bus, transferring 14 GiB/sec [15].
Google TPU has a size of 28nm and needs 40W of energy, running on a 700MHz frequency. It comes as an external card that matches a SATA harddisk slot. [12]

4.2.2 Software Environment

The software library used for running Python code on Google TPU is called TensorFlow and was made specifically for Google's machine learning applications, though having been turned into an open source platform lately.
TensorFlow consists of a number of different APIs enabling the user to build and train their own neural networks at different levels of control, with the most low level one being TensorFlow Core, and the higher level APIs being built on top of TensorFlow Core. The concept giving the library its name is tensors, which are basically just an array which can have a varying number of dimensions. The other concept that shows in TensorFlow's name is computational graphs, which represent a series of operations that will be run by a TensorFlow program after building the graph. TensorFlow also offers a graphic visualization tool for these computational graphs, called TensorBoard. [9]
Computational graphs are generally built using three different functions, inference, loss and training, where, as the names would suggest, the graph is build by the inference function, loss calculation is included into the graph by the loss function and training gives the functions used for minimization of the loss. After the training stage, an evaluation graph can be generated to enable the user to revise one's program. [9]
Whereas the most low level API TensorFlow Core requires advanced skills with regards to programming and machine learning, the high-level API tf.estimator is recommended to unexperienced users and takes a lot of work from the developer. It offers a range of predefined classification models that can be used on one's own data sets for training, while still offering the possibility to create one's own classification functions as well as defining the preprocessing operations applied to the data beforehand. It also comes with several functions to convert data sets into different tensor formats. [9]
As mentioned above, TensorFlow is not restricted to Google TPU or Google's neural network applications in general, but can also be used on classic CPU/GPU architectures. There are therefore a number of benchmarks available for different NVIDIA GPUs testing common classification models [9], unfortunately it apparently has not been tested for the NVIDIA Tesla V100 yet.
As for more recent developments, there is also a new compiler for linear algebra called XLA being developed which is trying to improve TensorFlow with regards to memory usage, execution speed, portability, etc. [5]

4.2.3 Benchmarks

The peak performance of Google TPU is stated to be 92 Teraops per second on its matrix unit, this is however referring to integer operations due to its architecture [9].
Comparisons with Intel's Haswell Xeon CPU and NVIDIA's K80 GPU on different benchmarks for convolutional neural networks, recurrent neural networks and multilayer perceptrons furthermore showed Google TPU to be 15 to 30 times faster than Intel's CPU and NVIDIA's GPU [5].

4.3 IBM TrueNorth

The following section shall give an overview of the architecture, software environment and benchmarks of the IBM TrueNorth.

4.3.1 Architecture

The IBM TrueNorth chip is described in [11] by one of its developers, Dharmendra Moda. According to him, a main focus in the development process was reducing the power consumption of a chip that is inspired by the human brain, regarding the fact that a hypothetical computer simulating the brain would need around 12GW in energy, whereas the actual human brain only runs on 20W. The IBM TrueNorth chip consists of around one million artificial neurons, connected by 256 million artificial synapses, each of them programmable. This is made possible by 4096 cores in total (parallel and distributed).
Furthermore, there is a customizable on-chip communication network, as well as the possibility

to build a customizable network of chips. IBM TrueNorth is considered to be very energy-efficient, it takes 70mW to run what is described as a typical recurrent network. [11]
The neurosynaptic cores on the TrueNorth chip are implemented as a number of axons which are connected to a number of neurons, where each axon is assigned a so-called activity bit with a corresponding time stamp modeling event flow. Each connection is furthermore weighted. [18]
On the hardware side, the axon-neuron connections are modeled as a 1024 x 256 SRAM crossbar, taking in the axon activity as input (using an input decoder) and giving the neuron spikes as output (using an output decoder) [18].

4.3.2 Software Environment

Along with the chip, IBM also introduced a new programming language with a matching simulator, environment and libraries, which is described in [2] by Amir et al. Seeing that the classic sequential programming model would not be suitable for TrueNorth's architecture, there has been developed a completely new programming paradigm based on the concept of so-called Corelets, which are an abstraction of a neural network, as well as a Corelet Language, Corelet Library and Corelet Laboratory. [2]
The Corelet Language is a turing-complete language, but it is different to common programming languages since the program itself is the neural network specified by the developer with its inputs, outputs, parameters and connectives. The goal of this programming language implementation is to hide the internal processes which are specified by the developer and only show the external in- and outputs to potential users of the corelet. [2]
The whole concept follows a divide-and-conquer approach with each corelet being made of a number of subcorelets, forming a tree with neurosynaptic cores being the leaves of that tree. Therefore, large neural networks are essentially built from smaller neural networks. [2]
The simplest version of a corelet as described in the article is the seed corelet, a program which is basically just one neurosynaptic core, which on its inside consists of neurons and axons, neurons being spike sources and axons being spike destinations. There is also a list called output connector, listing all of the neurons that put out spikes to the external environment of the Seed corelet, as well as a similar input connector list for axons. [2]
When a new corelet is formed from subcorelets, the output connectors of the subcorelets become the new spike sources, whereas input connectors turn into spike destinations. [2]
The main symbols of the Corelet Language are therefore quite obviously neurons, neurosynaptic cores and corelets. As the Corelet Language is object-oriented, this translates to a neuron class, core class and an additional connector class. Here, the neuron class internally consists of the initial membrane potential, reset mode, leaks and thresholds, along with some corresponding set- and get-methods. The neurosynaptic core class internally specifies a vector of 256 neuron objects, another list of 256 axom types and a 256 x 256 matrix showing connections between axoms and neurons. The corelet class quite intuitively contains its subcorelets, neurons, neurosynaptic cores and connectors, with the connector class specifying a list of pins that are storing the inter- an intra-corelet connections of the neural network. [2]
Generally, the communication between neurosynaptic cores is performed using all-or-none spike events that are transported over a message-passing network [2].
The Corelet Language got implemented with MATLAB OOP. Since it is object-oriented, there are features such as objects and classes as described above as well as inheritance and polymorphism. [2]
The Corelet Library is basically a container for premade corelets that can be reused by developers who can also add their own corelets into the repository. When the article [2] was written, there were about 100 corelets in the repository, for instance aggregators, linear filters, a Discrete Fourier Transform, etc. The Corelet Library can be seen as a tree itself, as all corelets are subcorelets to a base class. [2]
Corelet Laboratory is the program environment to create and execute code on. The underlying con-

cepts here are composition and decomposition, with composition being the creation of corelets from smaller elements. In order to actually run such a program, it is necessary to decompose the whole structure of corelets into an actual TrueNorth program that can be compiled and simulated. Corelet Laboratory further offers support for program verification. As in most deep learning frameworks, the simulation results can also be plotted and furthermore visualized in a video. [2]

4.3.3 Benchmarks

It is stated by Merolla et al. in [17] that TrueNorth achieves a peak performance of 46 billion SOPS (synaptic operations per second) per watt, comparing it to the peak performance of the current most energy-efficient supercomputer, which has a peak performance of 4.5 billion FLOPS per watt.
This again points out the great energy efficiency of IBM TrueNorth.

4.4 Comparison

With regard to power consumption, it is quite easy to see that IBM TrueNorth and Google TPU clearly outperform NVIDIA's Tesla V100, with IBM TrueNorth only needing 70mW to run a network and Google TPU which needs 40W compared to the Tesla V100 with a consumption of 250 to 300W.
Regarding the peak performance, the accelerators taken into account in this paper get difficult to compare, mainly due to the use of quantization in Google TPU leading to the fact that only integer operations are performed on the chip. It is likely that Google TPU will outperform NVIDIA's V100 in most neural network prediction tasks, since increasing the efficiency in this field was the reason it got designed by Google.
It would be interesting though to compare IBM TrueNorth to a Tesla V100 GPU using the same classification tasks as the ones that IBM researchers previously used for testing TrueNorth.
Overall, the lack of explicit benchmark values for Google TPU and IBM TrueNorth as well as the different units of measurement associated with the accelerators by the manufacturers make them very hard to compare when it comes to performance.
Taking a look at usability, it is obvious that Google TPU which is specialized on Google's needs for neural network prediction will not come in handy for many other tasks than that, whereas a GPU architecture such as Tesla V100 can be more universally used as seen in their systems for self-driving cars mentioned above.
With IBM TrueNorth, one big issue is the fact that is has not actually been released yet, so it is hard to estimate the degree to which is will be used in a commercial way.
Overall, it can probably be stated that NVIDIA's Tesla V100 is the most universal accelerator out of the three since it can be applied for both training and inference tasks in multiple different settings. Also, there are several different software environments that can be used on Tesla V100, from CUDA to open-source OpenCL and frameworks such as Caffe.
Regarding the corresponding software environments, it will always be a matter of personal preference which one will be preferred by the majority of developers when having to implement one's own applications and it is quite hard to build an opinion on their user-friendliness without having a lot of experience using any of them, since a software development kit generally should be tested by both beginners and advanced developers regarding intuitiveness and the degree of control over one's implementation.

5 Discussion

It is not a big surprise that Google TPU and IBM TrueNorth are more energy-efficient than NVIDIA's Tesla V100 keeping in mind that the first two accelerators were specifically built with the goal to reduce energy consumption, as well as both architectures being designed for very specific purposes in contrast to NVIDIA's Tesla V100 being designed to be a more general-purpose accelerator, making Google TPU and IBM TrueNorth less universally

usable.
On the flip side, the fact that Tesla V100 will outperform the two other accelerators on most tasks - except the ones those two were designed for - is equally little surprising.
Overall, it is definitely difficult to draw a comparison between those three different architectures. Generally spoken, which accelerator should be preferred over the other ones is very much dependent on the actual task one trying to use them for.
In many cases, a combination of different architectures might also be beneficial as in using a GPU architecture for training and a specialized ASIC or a neuromorphic chip for the actual inference tasks.

6 Related Work

In an article from 2015 Rodrigues et al. showed the capabilites of GPUs in comparison to multicore architectures, namely comparing a Gainward GeForce GTX 580 Phantom GPU by NVIDIA to two Intel Xeon X5550 cores, running the same recommender algorithms in already existing versions for CPUs on the Intel Xeon X5550 cores and a CUDA version of them on the Gainward GeForce GTX 580 Phantom. The tested GPU was able to achieve a speedup of maximum 14.8 compared to the multicore implementation. [4]
Another topic that is frequently mentioned regarding accelerators for deep learning is FPGA technology.
In a 2017 article Nurvitadhi et al. discuss in detail the performance of FPGAs in comparison to GPUs in relation to new developments in deep learning. The author argues that upcoming deep neural networks will work with network sparsity at lot more so then done in the past which leads to more irregular parallelism and furthermore use more custom, compact data types. Hence he is suggesting to use FPGAs as an alternative, which are strongly customizable and in addition also typically more energy-efficient than GPUs. His test results show a better performance of current FPGAs on binary, sparse, compact narrow-bandwidth and ternary deep neural networks compared to a modern GPU. [1]
Wen et al. in 2016 suggested a new learning method for IBM TrueNorth in order to improve its performance on inference tasks since TrueNorth is like Google TPU working with low-resultion integers. The authors therefore propose a probability-biased learning method. [6]
The approach led to small improvements regarding accuracy, but an increased overall performance with a speed-up of 6.5 and a reduction of 68.8 percent regarding the cores needed to perform computations [6].
In an article from 2016, Gu et al. also describe attempts to transform caffe, a CUDA-based framework for deep learning on neural networks into an OpenCL version to make deep learning more accessible for hardware architectures which are not based on CUDA [3].
OpenCL is as mentioned before an open source standard and supported by many different manufacturers [3] as well as NVIDIA [8].
The OpenCL version could not compete with the CUDA version yet regarding performance, however the authors point out that there is still a lot of effort needed to deal with the different OpenCL extensions available in the future development of OpenCL caffe [3].

7 Conclusion and Future Work

In summary, this paper gives an overview of three different trends in accelerators, examining NVIDIA Tesla V100 representing classic GPU-based accelerators, Google TPU as an ASIC and IBM TrueNorth as a neuromorphic chip, reviewing their architectures, corresponding software environments and accessible benchmarks, giving an overview of the different accelerators' characteristics and capabilities. It would be difficult to recommend one of the accelerators over the others, since they were all built with different concepts in mind. It is easy to see that the NVIDIA Tesla V100 accelerator is the one achieving overall best performance, whereas TrueNorth is incredibly energy-efficient. Since

 DOI: 10.14459/2018md1430700

Google TPU is an ASIC, it performs very well on its designated task while at the same time being very energy-efficient.
It would obviously be interesting to try to compare all of the three accelerators reviewed in this paper on a set of actual classification tasks, namely running for each accelerator one or several tasks that it is "specialized" on on itself and the two other accelerators to point out strengths and weaknesses regarding tasks that the accelerator wasn't necessarily optimized for, as well as taking a deeper look into similarities and differences between the different architectures.
Yet another processor that has been a recent topic of discussion is Intel's Xeon Phi Knights Mill processor, which is presumably going to be specialized on deep learning as well.
Since Knights Mill was neither yet released nor was there a lot of information available on it when this paper was written, it hasn't been included into my comparison. With that being said, Knights Mill would be an interesting subject for future work on this topic once there is more information on it released.
On the general topic of trends for accelerators in deep learning, it will be interesting to see what the next developments are going to be and how the approaches on architectures are going to evolve. As discussed above in this paper, there are different new models challenging classic GPU accelerators, from neuromorphic chips to ASICs and FPGAs.
FPGAs and ASICs however are naturally specialized constructions for very specific applications and the energy-efficiency of Google TPU and IBM TrueNorth for example comes with the cost of lower precision, which makes them unsuitable for training artificial neural networks, hence why GPUs are still needed for the training stage before performing the actual classification tasks.
Overall, successful architecture will always be tied to the needs of current deep learning applications, so one will have to keep an eye on research developments in deep learning in general to predict future trends in accelerator types and architectures.

References

[1] Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R., Ong Gee Hock, J., ... & Boudoukh, G. (2017, February). Can FPGAs beat GPUs in accelerating next-generation deep neural networks?. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 5-14). ACM.

[2] Amir, A., Datta, P., Risk, W. P., Cassidy, A. S., Kusnitz, J. A., Esser, S. K., ... & McQuinn, E. (2013, August). Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores. In Neural Networks (IJCNN), The 2013 International Joint Conference on (pp. 1-10). IEEE.

[3] Gu, J., Liu, Y., Gao, Y., & Zhu, M. (2016, April). OpenCL caffe: Accelerating and enabling a cross platform machine learning framework. In Proceedings of the 4th International Workshop on OpenCL (p. 8). ACM.

[4] Rodrigues, A. V., Jorge, A., & Dutra, I. (2015, April). Accelerating recommender systems using GPUs. In Proceedings of the 30th Annual ACM Symposium on Applied Computing (pp. 879-884). ACM.

[5] Abadi, M., Isard, M., & Murray, D. G. (2017, June). A computational model for TensorFlow: an introduction. In Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (pp. 1-7). ACM.

[6] Wen, W., Wu, C., Wang, Y., Nixon, K., Wu, Q., Barnell, M., ... & Chen, Y. (2016, June). A new learning method for inference accuracy, core occupation, and performance co-optimization on TrueNorth chip. In Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE (pp. 1-6). IEEE.

[7] NVIDIA. Nvidia Tesla V100 GPU Architecture: The World's Most Advanced

Data Center GPU. Retrieved from http://www.nvidia.com/object/tesla_product_literature.html

[8] NVIDIA. Tesla V100 Performance Guide: Deep Learning and HPC Applications. Retrieved from http://www.nvidia.com/object/tesla_product_literature.html

[9] TensorFlow. (2017). Getting Started With TensorFlow. Retrieved from https://www.tensorflow.org/get_started/get_started

[10] NVIDIA. (n.d.). Programming Guide :: CUDA Toolkit Documentation. Retrieved from http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[11] IBM. (n.d.). IBM Research: Brain-inspired Chip. Retrieved from http://www.research.ibm.com/articles/brain-chip.shtml

[12] Google Cloud Platform. (n.d.). An in-depth look at Googles first Tensor Processing Unit (TPU). Retrieved from https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

[13] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. Neural networks, 61, 85-117.

[14] LeCun, Y., Bengio, Y., LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.

[15] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., ... & Boyle, R. (2017, June). In-datacenter performance analysis of a tensor processing unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture (pp. 1-12). ACM.

[16] Song, S., Miller, K. D., & Abbott, L. F. (2000). Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. Nature neuroscience, 3(9), 919-926.

[17] Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., ... & Brezzo, B. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. Science, 345(6197), 668-673.

[18] Merolla, P., Arthur, J., Akopyan, F., Imam, N., Manohar, R., & Modha, D. S. (2011, September). A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm. In Custom Integrated Circuits Conference (CICC), 2011 IEEE (pp. 1-4). IEEE.

[19] Wired. (2016). In Two Moves, AlphaGo and Lee Sedol Redefined the Future. Retrieved from https://www.wired.com/2016/03/two-moves-alphago-lee-sedol-redefined-future/

[20] top500. (2017). Google Latest AlphaGo AI Program Crushes Its Predecessors. Retrieved from https://www.top500.org/news/google-latest-alphago-ai-program-crushes-its-predecessor/

[21] Human Brain Project. (n.d.). Retrieved from https://www.humanbrainproject.eu/en/

[22] The BRAIN Initiative. (n.d.). Retrieved from www.braininitiative.org/

[23] Nvidia. (n.d.). Autonomous Car Development Platform from NVIDIA DRIVE PX2. Retrieved from https://www.nvidia.de/self-driving-cars/drive-px/