Master Projekt

IP813

Parallel Computer Vision:

Heterogeneous System Architecture & Enhanced Head-Tracking

Lang Christian

Matrikelnummer: 08-169-047

Advisor:

Prof. Dr. Christoph Stamm, FHNW

Abgabedatum: 30.08.2013

Abstract

Die vorliegende Arbeit befasst sich mit heterogenen System Architekturen (HSA) und dem Detektieren von Objekten in Echtzeit-Videos. Zu klären ist, welches der drei HSA-Systeme (AMP, OpenCL und OpenACC) die beste Performance und den einfachsten Einstieg bietet. Zudem wird ermittelt, ob ein Detektor für Köpfe trainiert werden kann, der einerseits genügend präzise, andererseits schnell genug ist, um den Anforderungen der Personenerkennung in Live-Videos gerecht zu werden. Der dazu verwendete Cascade-Classifier soll mittels OpenCV trainiert werden.

Um diese Fragen zu beantworten werden ausführliche Messungen mit allen drei HSAs durchgeführt, welche das effektive Beschleunigungspotential der einzelnen Systeme liefern. Zusätzlich werden Erfahrungen bei der Benutzung gesammelt und so einen Einblick in die Komplexität des Systems und dessen Anwenderfreundlichkeit gewonnen.

Abschliessend kann das etablierte und stark konfigurierbare OpenCL als leistungsfähigste HSA empfohlen werden. AMP hingegen profitiert von der modernen und gut verständlichen Realisierung, kombiniert mit einer moderaten Beschleunigungsleistung. OpenACC ist noch unausgereift und nur mit CUDA-Geräten kompatibel und erfüllt deshalb nicht die Anforderungen einer universel einsetzbaren HSA.

Weiter wurde erfolgreich ein Detektor mit Kopf-Samples trainiert, welcher für die Weiterentwicklung der Personenerkennungssoftware verwendet werden kann. Dieser Classifier verwendet Local Binary Patterns (LBP) als Features, die in kurzer Zeit trainierbar sind und zu guten Erkennungsraten führen.

Inhaltsverzeichnis

1 Einleitung 1
1.1 Motivation 1
1.2 Aufgabenstellung 1
2 Heterogene Systeme 2
2.1 Unterschiede Host / Device 2
2.1.1 Allgemeine Hardwareunterschiede 3
2.1.2 nVidia CUDA 5
2.1.3 HSA-Begriffe 8
2.1.4 Branching 9
2.1.5 Kompilierung 11
2.1.6 Problematiken bei HSA-Code 11
2.2 Testanwendung 13
2.3 C++ AMP 14
2.3.1 Installation 14
2.3.2 Einstieg in die Technologie 14
2.4 OpenCL 16
2.4.1 Installation 17
2.4.2 Einstieg in die Technologie 17
2.5 OpenACC 20
2.5.1 Installation 21
2.5.2 Einstieg in die Technologie 21
2.6 Rahmenbedingungen Messung 23
2.6.1 Spezielle Testumstände OpenACC 24
2.6.2 Durchführung der Messreihen 25
2.6.3 Auswertung der Messreihen 26
2.7 Analyse: Performance 26
2.7.1 AMP 26
2.7.2 OpenCL 30
2.7.3 OpenACC 35
2.7.4 Vergleich 37
2.8 Analyse: Handhabung 40
2.8.1 AMP 40
2.8.2 OpenCL 41
2.8.3 OpenACC 42
2.8.4 Vergleich 43
2.9 Empfehlung 44
3 Kopfdetektion 45
3.1 Machine Learning 45
3.1.1 Classifier 45
3.1.2 Features 47
3.1.3 Boosting 49
3.1.4 Rejection Cascades 50
3.1.5 Soft-Cascade 51
3.1.6 Einsatz und Beispiel eines Classifiers 53

3.2 Trainieren eines Classifiers mit OpenCV 53
3.2.1 Einrichten OpenCV 53
3.2.2 Datasets / Samples 54
3.2.3 Performance-Test 55
3.2.4 Trainieren 56
3.2.5 Auswertung 58
3.2.6 Trainieren einer Soft Cascade 61
4 Diskussion 62
4.1 Heterogene Systeme 62
4.2 Kopfdetektion 63
4.3 Ausblick 63

1 Einleitung

1.1 Motivation

Da heutzutage in beinahe jedem PC, Smartphone oder Tablet eine Art von Grafikbeschleunigung vorhanden ist, macht es Sinn, diese GPUs für rechenaufwändige Aufgaben nebst der Grafikberechnung zu verwenden. Um als Entwickler mit nur einer einzigen generellen Schnittstelle alle vorhandenen Beschleuniger eines Systems anzusprechen, bieten sich heterogene System Architekturen (HSA) an. Diese ermöglichen den Algorithmus einmalig zu schreiben und über das mitgelieferte Framework auf CPUs, GPUs und APUs effizient und ressourcenfüllend auszuführen.

Dabei soll sich der Entwickler so wenig wie möglich mit den Besonderheiten der verwendeten Beschleuniger auseinandersetzten müssen, um die gewünschte Beschleunigung zu erhalten. Für Fälle, in welchen einer bestimmten Hardware die maximale Leistung entlockt werden soll, muss das System die Flexibilität bieten, spezifische Optimierungsmassnahmen der Hardware von Hand vorzunehmen, ohne zu viel Overhead durch die Abstraktionsschicht zu verursachen.

Eine dieser HSAs soll zukünftig in der Software für Personenerkennung aus dem Projekt 7 (Parallel Computer Vision: Person Data Extraction) eingesetzt werden. Betreffend Personenerkennung bietet OpenCV mächtige Machine-Learning-Algorithmen an wie z.B. Rejection-Classifier. Diese können zu Detektoren trainiert werden und so Köpfe in einem Live-Video schnell und präzise erkennen.

1.2 Aufgabenstellung

Die drei folgenden HSAs sollen ausführlich getestet werden: AMP, OpenCL und OpenACC. Dabei soll die Handhabung und Performance bewertet und analysiert werden. Es ist von Interesse, wie die einzelnen HSAs aufgebaut sind und wie sie die Hardware ansteuern. Dadurch ergibt sich die Komplexität der Systeme und somit der Lernaufwand der nötigen Einarbeitung. Zudem definiert der Systemaufbau, wie und ob gewisse Probleme damit gelöst werden können. Die Performance-Messungen sollen mit einem Problem aus dem Bereich der Bildverarbeitung durchgeführt werden.

Des Weiteren soll die Software aus dem Projekt 7 weiterentwickelt werden, damit sie die Personen im Video detektieren und somit Daten zu den einzelnen Personen extrahieren kann. Diese Daten bilden ein Modell, welches die Wiedererkennung der bekannten Personen ermöglicht. Als nächsten Schritt soll deshalb ein Detektor trainiert werden, der Köpfe aus allen Blickrichtungen erkennt. Diese erkannten Bereiche werden weiterverwendet um die Köpfe der Personen mittels Tracking-Learning-Detection (TLD) über das Bild zu verfolgen.

Die komplette Aufgabenstellung ist in der Klärung in Anhang A ersichtlich.

2 Heterogene Systeme

Um den steigenden Leistungsanforderungen von modernen Anwendungen gerecht zu werden, sollen möglichst alle verfügbaren Ressourcen eines PCs ausgenutzt werden. Die neben der üblich benutzten CPU auch meist vorhandene GPU, soll deshalb auch für allgemeine Berechnungen verwendet werden. Unter allgemein wird hier alles verstanden, was nicht unter den Begriff „Grafische Berechnungen" fällt.

Anders als die bis anhin häufig verwendeten Technologien für GPGPU-Programmierung, wie z.B. CUDA, wird in Zukunft aber eine höhere Abstraktion verlangt, welche durch das Prinzip von heterogenen System Architekturen (HSA) erreicht wird. In diesen wird nicht nur ermöglicht, die GPU einfacher anzusprechen, sondern alle Recheneinheiten eines Systems werden über ein einheitliches Interface verfügbar gemacht. Über die Abstraktion der sogenannten Beschleuniger lassen sich Aufgaben definieren, welche an unterschiedliche Recheneinheiten verteilt werden, ohne auf die jeweiligen Hardwarespezialitäten achten zu müssen. Dies erledigen ein entsprechender Compiler und/oder ein Framework.

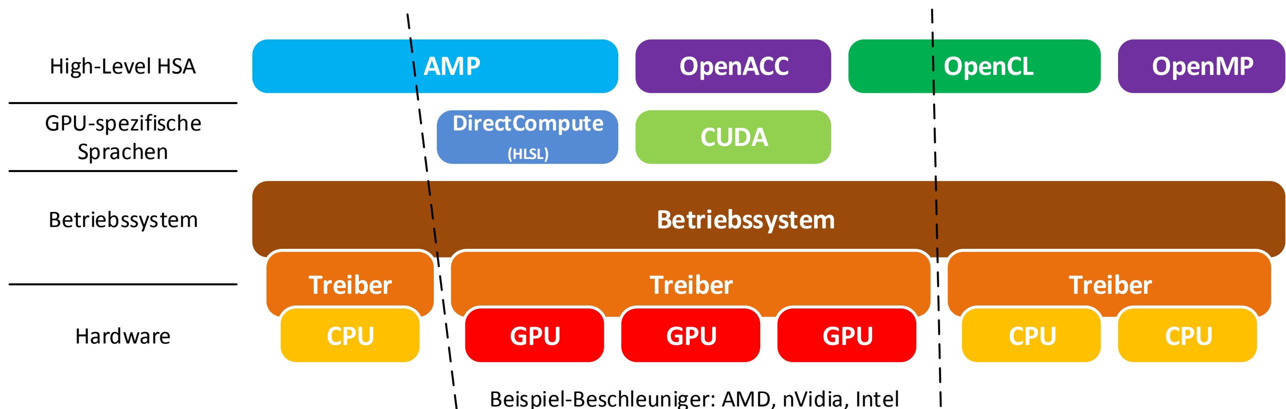


Abbildung 2.1: Die zu testenden HSAs in hierarchischer Darstellung: AMP, OpenACC und OpenCL.

In diesem Kapitel sollen drei HSAs (Abbildung 2.1) betrachtet und anhand einer 2D-Faltung getestet und verglichen werden. Dabei sollen jedoch nicht nur das Laufzeitverhalten, sondern vielmehr auch die Komplexität und Nutzbarkeit der Systeme analysiert werden. Um den Einstieg zu erleichtern, werden zuerst allgemeine Konzepte und Problematiken von HSAs vorgestellt und danach in die drei Technologien eingeführt.

2.1 Unterschiede Host / Device

Durch die doch sehr unterschiedliche Architektur von CPUs und GPUs werden eindeutige Bezeichnungen benötigt. Deshalb wird Code für eine CPU-Architektur als „Host-Code" bezeichnet und Code für Beschleuniger als „Device-Code". Die Bezeichnung des Hosts leitet sich aus dem Fakt ab, dass einzig die CPU die Initialisierung und Verwaltung von Programmen übernehmen kann und somit auch die Ausführung auf den Beschleunigern steuert. Eine andere Bezeichnung für Device-Code ist „Kernel", was ursprünglich den Programmcode einer GPU bezeichnet. Da jedoch meistens GPUs als Beschleuniger eingesetzt werden, ist auch im HSA-Umfeld häufig von Kerneln die Rede. Nicht zu verwechseln mit dem Kern einer CPU!

Ebenfalls aus diesem Grund kann man bei der Implementierung von HSA-fähigem Code viele GPU-spezifische Eigenheiten ausmachen. Und obwohl jeder Hersteller unterschiedliche Begriffe verwendet, handelt es sich meistens um dieselbe Eigenschaft oder dasselbe technische Bauteil. Um diese Begriffe besser überblicken zu können und auch die Unterschiede zur CPU zu kennen, werden diese Grundlagen in den folgenden Abschnitten eingeführt. Dabei bietet Baxter [1] eine sehr ausführliche Einführung in die Hardware und Programmierung von modernen GPUs.

2.1.1 Allgemeine Hardwareunterschiede

Eine Recheneinheit (z.B. CPU) ist ein komplexes Gebilde, das aus diversen Bestandteilen wie Steuerwerken, Rechenwerken oder Speichern besteht. CPUs und GPUs unterscheiden sich primär durch die Art der vorhandenen Rechenwerke. Tabelle 2.1 listet die gebräuchlichen Begriffe für die Recheneinheiten heutiger Computer und deren Zusammengehörigkeit auf. Zudem wird jeweils die Einteilung zur Flynnschen Klassifikation[1] angegeben.

	Bedeutung	**Beschreibung**
CU	Control Unit	Steuer- oder Leitwerk welches die Abarbeitung der Befehle steuert.
SSE	Streaming SIMD Extension	Rechenwerk oder Erweiterung eines Rechenkerns für beschleunigtes Abarbeiten von SIMD-Instruktionen. Auch Vektoreinheit genannt. Besitzt 128bit breite Zusatzregister. Es existieren folgende Versionen: SSE, SSE2, SSE3, SSE4, SSE4a, AVX (aktuell neuste Version mit neu 256bit), SSE5 (verworfene Version von AMD).
FPU	Floating Point Unit	Rechenwerk für Gleitkommaberechnungen.
ALU	Arithmetic Logical Unit	Rechenwerk für Ganzzahlberechnungen.
SFU	Special Function Unit	Rechenwerk für spezialisierte Funktionen wie z.B. Sinus, Cosinus, Wurzel, etc.
EXE	Execution Unit	Vereint alle Rechenwerke eines Kerns, z.B. bei CPU also ALU und FPU.
	Core	Rechenkern. Beinhaltet immer die Execution-Unit, im Fall der CPU aber auch das Steuerwerk.
	Compute Unit	Verbund mehrerer Rechenkerne einer GPU inklusive Cache und Steuerwerk.
CPU	Central Processing Unit	Hauptrecheneinheit oder Hauptprozessor. Besteht aus einem oder mehreren Rechenkernen. Jeder Rechenkern enthält eine CU, ALU und FPU. Des Weiteren enthält eine CPU meist mehrere SSE- oder SFU-Einheiten. Sie kann SISD-, SIMD- oder bei Mehrkernprozessor MIMD-Instruktionen verarbeiten.
GPU	Graphics Processing Unit	Grafische Recheneinheit mit vielen FPUs und ALUs aber wenigen CUs. Verarbeitet SIMD- und MIMD-Instruktionen.
APU	Accelerated Processing Unit	Kombination einer CPU und einer unterstützenden Recheneinheit (meist GPU). Begriff durch AMD geprägt.

Tabelle 2.1: Unterschiedliche Rechenwerke und Recheneinheiten und deren Bedeutung.

Dabei ist wichtig zu erkennen, dass sich CPU und GPU vor allem durch die Anzahl CUs unterscheiden. Abbildung 2.2 zeigt eine grafische Abstraktion der beiden Recheneinheiten. Während die CPU in jedem Kern eine CU besitzt, hat die GPU mehrere Kerne pro einzelne CU. Aus diesem Grund eignet sich eine CPU besser für allgemeinere Aufgaben, da alle Kerne gleichzeitig völlig unabhängige Instruktionen ausführen können. Die GPU hingegen ist für die Abarbeitung von grossen Datenmengen gedacht, wobei auf allen Daten dieselben Instruktionen ausgeführt werden müssen. Deshalb besitzt jede CU ein Array von Kernen, welche alle gleichzeitig identische Aufgaben durchführen, allerdings dabei mehrere Datensätze bearbeiten. Da eine GPU zudem mehrere Compute-Units enthält, ist es ihr möglich MIMD-Instruktionen auszuführen.

[1] Unterteilung der vier unterschiedlichen Rechenarchitekturen nach Michael J. Flynn (1966). Diese beschreibt die Anzahl Befehls- und Datenströme, welche gleichzeitig abgearbeitet werden können: Single Instruction Single Data (SISD), Multiple Instruction Multiple Data (MIMD) etc.

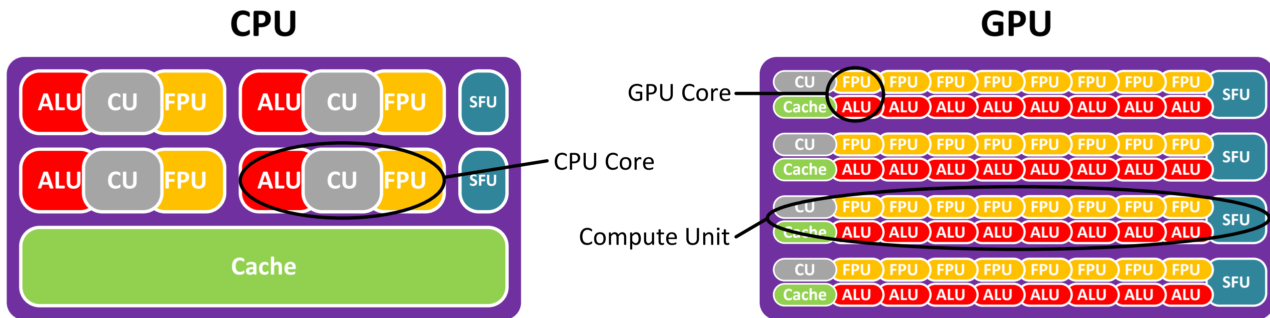


Abbildung 2.2: Vergleich CPU und GPU bezogen auf die enthaltenen Rechenwerke. CPUs enthalten mehr Cache und pro Kern eine CU. GPUs enthalten mehr Kerne, welche zu einer Compute-Unit mit nur einer CU aggregiert werden.

Nicht nur die Rechenwerke unterscheidet sich bei GPUs und CPUs, sondern auch die Organisation des Speichers intern und extern der GPU. Abbildung 2.3 zeigt die grundsätzlichen Unterschiede der Speicherorganisation von CPUs respektive GPUs, während Tabelle 2.2 die Begrifflichkeiten erklärt.

	Bedeutung	Beschreibung
MMU	Memory Management Unit	Verwaltet die Schnittstelle zwischen Recheneinheiten und globalem Speicher.
	Cache	Schneller Zwischenspeicher für Daten aus dem globalen Speicher. Ist meist hierarchisch aufgebaut und kann als Software- oder Hardware-Cache vorhanden sein. Wobei Hardware-Cache ohne weiteres Zutun die optimalsten Daten cached, der Software-Cache allerdings explizit angewiesen werden muss, was gecached werden soll.
	Local Memory	Lokaler Speicher für jede Compute-Unit. Es können nur die Rechenwerke innerhalb der Compute-Unit auf diesen Speicher zugreifen. Funktioniert wie ein Software-Cache.
	Global Memory	Hauptspeicher, welcher über die MMU angesprochen werden kann und sich ausserhalb der Recheneinheit befindet. Alle Rechenwerke können darauf zugreifen.

Tabelle 2.2: Begriffe der Speicher-Hardware und deren Bedeutung.

Hauptmerkmal ist dabei die Art der verwendeten Caches. Während bei CPUs hochoptimierte automatische Caches in einer hierarchischen Form eingesetzt werden, kommen bei GPUs programmierbare Speicher zum Einsatz, welche spezifisch auf das Problem optimiert werden können und sollen. Diese werden auch als Hardware- respektive Software-Caches bezeichnet.

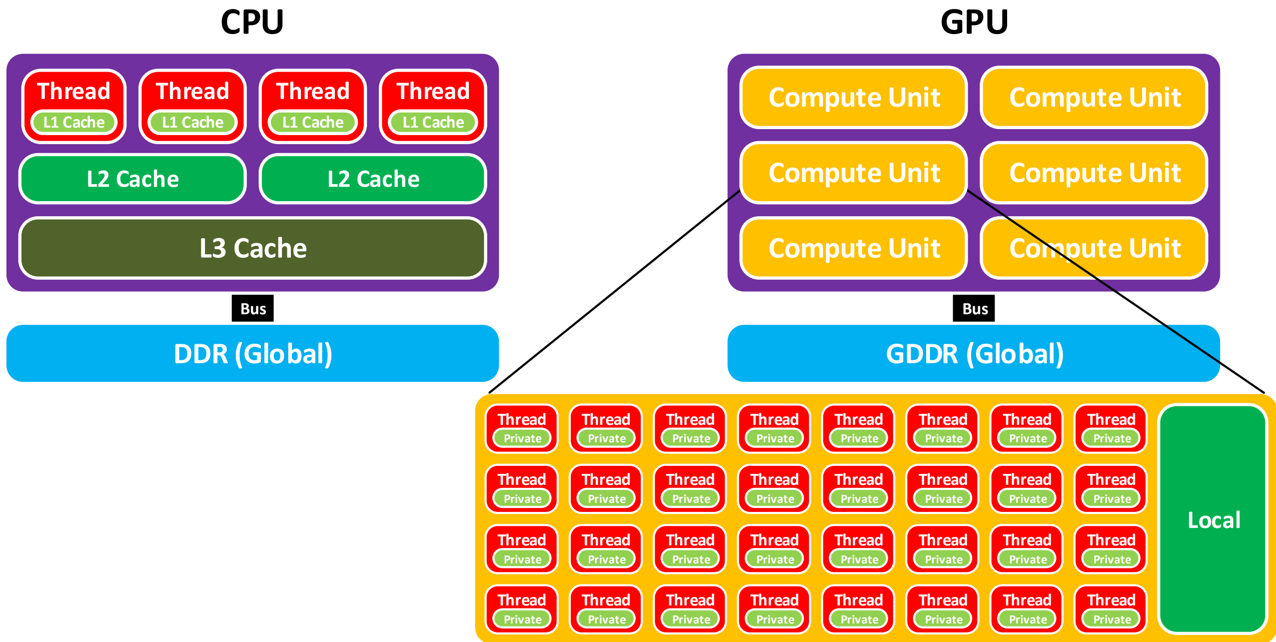


Abbildung 2.3: Speicherstrukturen von CPU und GPU. Während die CPU automatische (Hardware) Caches für die optimierte Speicherverwaltung verwendet, kommen bei der GPU programmierbare (Software) Caches zum Einsatz, welche vom Programmierer optimal eingesetzt werden müssen.

Die wichtigste Eigenschaft der Software-Caches in GPUs ist die in Tabelle 2.2 erwähnte Sichtbarkeit, wobei die lokalen Speicher jeweils klein und schnell sind, umgekehrt der globale Speicher gross und eher langsam ist. Um ein Kernel auf der GPU möglichst schnell zu machen, müssen deshalb häufig verwendete Daten vom globalen Speicher in den schnelleren lokalen Speicher kopiert, also gecached, werden. Dieser Aufwand lohnt sich allerdings nur dann, wenn die Daten mehr als einmal verwendet werden.

Die optimale Wahl der Recheneinheit hängt somit von der Organisation der Aufgabe ab. Wenn sehr viele unterschiedliche Berechnungen auf jeweils wenigen Daten gemacht werden sollen, ist die CPU die bessere Wahl, da sie durch Pipelining und Branch-Prediction Instruktionen vorbereiten kann und so die Wartezeit zwischen den einzelnen Instruktionen minimiert. Da die Verzögerung beim Lesen von Daten auf diese Weise nicht verringert werden kann, wird der Zugriff darauf durch sehr grosse Hardware-Caches unterstützt.

Falls die Aufgabe lautet, sehr viele Daten auf dieselbe Weise zu bearbeiten, wie z.B. bei Rendering, eignet sich eine GPU eher, da die einzelne Instruktion in die Compute-Unit geladen wird und alle Kerne diese auf unterschiedlichen Daten ausführen. Dabei lässt sich die Geschwindigkeit weiter erhöhen, indem die Problemdaten so aufgeteilt werden, dass jede Compute-Unit benachbarte Daten (Data Locality) erhält. So kann die GPU diese in ihren Software-Cache (Local Memory) laden, um die Mehrfachzugriffe darauf erheblich zu beschleunigen. Zudem nutzen auch GPUs Pipelining, um Wartezeiten zu vermindern.

Im Allgemeinen will eine CPU also die Flexibilität durch viele Steuerwerke (MIMD) erhalten und Speicherverzögerungen durch grosse Caches verringern. Die GPU will den arithmetischen Rechendurchsatz mit sehr vielen einzelnen Kernen maximieren, verzichtet dabei aber auf die Möglichkeit, auf allen Kernen unterschiedliche Befehle auszuführen (SIMD). Korrekterweise muss hier aber bemerkt werden, dass sich die beschriebenen Strukturen zunehmend aneinander angleichen. Z.B. enthalten moderne GPUs auch hierarchisch organisierte Hardware-Caches.

2.1.2 nVidia CUDA

Um die Funktionsweise einer GPU besser zu verstehen, wird am Beispiel der Compute Unified Device Architecture (CUDA) von nVidia [2, 3, 4] erklärt, wie die Hardware aufgebaut ist und welche Auswirkungen

dies auf die Software-Ausführung hat. CUDA befindet sich aktuell in der vierten Generation, wobei die verwendeten Testsysteme (Kapitel 2.6) GPUs der zweiten und dritten Generation angehören. Zudem ist die Fermi-Architektur näher an den erläuterten Hardwareabstraktionen und wird deshalb als Referenz verwendet. Tabelle 2.3 zeigt eine Übersicht über die unterschiedlichen CUDA-Architekturen und deren wichtigsten Daten. Zu beachten ist, dass der Begriff „Tesla" einerseits als Name der GT200-Architektur, andererseits als Grafikkarten-Familie im GPGPU-Bereich Verwendung findet. Weitere Daten und Grafiken finden sich im Anhang B.

Gen.	Architektur	Grafikkarten (GeForce)	Erscheinung	Compute Capability	SM/SMX	CUDA Cores pro SM/SMX	Neuigkeiten dieser Generation
1	G80 / G92	Serie 8 / 9	2006 / 2008	1.0 / 1.1	16 SM	8	Unified Shader Architecture, Direct3D 10
2	GT200 (Tesla)	200 / 300	2008	1.2 / 1.3	30 SM	8	Double Precision Floating Point Arithmetics
3	GF100, GF110 (Fermi)	400 / 500	2010	2.0 / 2.1	16 SM	32/48	Improved Floating Point Arithmetics
4	GK104, GK110 (Kepler)	600 / 700	2012	3.0 / 3.5	15 SMX	192	Hyper-Q, Dynamic Parallelism
5	Maxwell	800 / 900	2014				ARM CoProzessor, Unified Virtual Memory
6	Volta	1000 / 1100	2016				Stacked DRAM

Tabelle 2.3: Architektur-Übersicht CUDA. Die vierte Generation (Kepler) ist die aktuelle Version.

Als erstes fällt die steigende Gesamtanzahl SMs auf, welche bereitgestellt wird. Als „Streaming Multiprocessor" (SM) bezeichnet nVidia die Compute-Units der GPU, welche eine unterschiedliche Anzahl Kerne enthalten; „Streaming Processor" (SP) genannt. Ab der vierten Generation werden die SMs als SMX bezeichnet. Das Local-Memory wird unter CUDA Shared-Memory genannt. Tabelle B.1 zeigt, dass dies ab Fermi 64kB gross ist und dynamisch der Verwendung als Shared-Memory oder als L1-Cache zugeteilt werden kann. Mögliche Konfigurationen sind 16kB L1-Cache und 48kB Shared-Memory oder umgekehrt. Auf der Kepler-Architektur kommt noch die Variante 32/32 dazu. Abbildung B.3 verdeutlicht diese Entwicklung, welche darstellt, wie sich die GPUs weiter an die CPU-Architekturen annähern.

Der Aufbau eines Fermi-Chips ist in Abbildung B.1 gezeigt, wobei die 16 SMs deutlich identifzierbar sind. Zudem erkennt man den neuen L2-Cache, die 6 Speicherkontroller und die GigaThread-Einheit, welche für die Kommunikation mit dem Host zuständig ist und über welche die Kernel gestartet werden. In Abbildung B.2 wird ein einzelner SM dargestellt, welcher vier Ausführungsgruppen mit 32 Kernen, 16 Lade/Speichereinheiten und vier SFUs beinhaltet. Wichtig sind die Warp-Scheduler und Dispatch-Einheiten, womit zwei Instruktionspakete zeitgleich in der SM gestartet werden können.

Ein Kernel, welcher auf solch einer CUDA-Architektur ausgeführt werden soll, muss sich an die Hierarchie aus Abbildung 2.4 anlehnen. Die kleinste Einheit ist der Thread, welcher nur auf sein Private-Memory zugreifen kann. Zudem hat jeder Thread eine eigene Thread-ID innerhalb seines Thread-Blocks. Ein Thread-Block ist eine Gruppe von mehreren unabhängigen Threads, welche über das Shared-Memory des Blocks Daten austauschen können. Die Anzahl Threads in einem Block kann beliebig sein und hängt vom Problem ab. Weiter können die einzelnen Threads innerhalb desselben Blocks mit Barrier-Synchronisation koordiniert werden. Die Blocks selbst werden in einem Grid gruppiert, welches einen Kernel bildet. Somit ist jedes Grid ein separater Kernel, wobei ein Programm mehrere Kernel beinhalten kann. Dieser hat auf das Global-Memory Zugriff und verwaltet die Ausführung der Grids und dessen Blöcke. Die Blöcke innerhalb eines Grids können dabei Abhängigkeiten aufweisen, z.B. dass der eine Block zuerst Daten berechnen muss, bevor diese vom anderen Block weiterverwendet werden können.

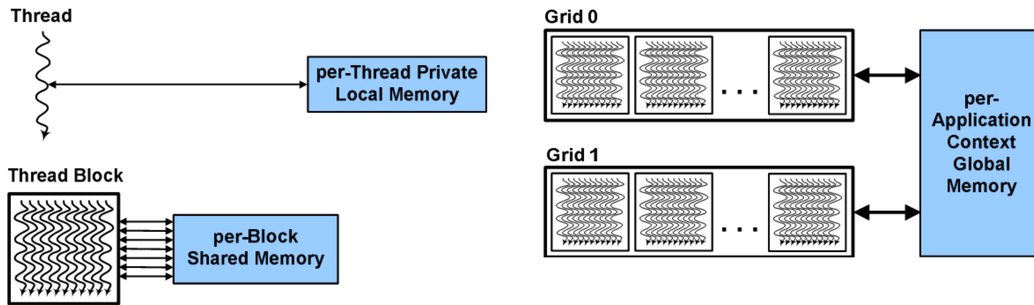


Abbildung 2.4: Hierarchie der Softwareteile unter CUDA. Kleinste Einheit ist der Thread, welcher auf einem einzelnen CUDA-Kern ausgeführt wird. Weiter sind die verfügbaren Speicher auf der jeweiligen Ebene dargestellt. Bild aus [2].

Wird nun ein Kernel an die GPU geleitet, verteilt die GigaThread-Einheit die Blöcke aus dem Grid an die SMs, welche die Blöcke in den Warp-Scheduler weiter aufspalten, so dass die gesamte SM mit Threads ausgelastet ist. Diese Teilblöcke werden Warps genannt und entsprechen 32 Threads. Allerdings werden die 32 Threads eines Warps nicht eins zu eins auf die 32 SPs verteilt, was zum Abarbeiten eines kompletten Warps pro Taktzyklus führen würde.

Diese Tatsache lässt sich zum einen durch den Fakt begründen, dass nicht alle GPU-Typen 32 SPs pro SM besitzen, zum anderen möchte man die Wartezeiten zwischen den einzelnen Befehlen durch Pipelining verkürzen. Diese Technik wird verwendet, um viele Instruktionen schnell hintereinander und ressourcenfüllend auf einem Rechenwerk ausführen zu können. Dabei werden grosse Befehle in kleine einheitliche Instruktionen aufgeteilt, was dann insgesamt zu einer längeren Ausführungszeit der meisten Befehle führt. Durch diese Aufteilung können aber neue Befehlsabarbeitungen begonnen werden, bevor ein noch nicht beendeter Befehl abgeschlossen wurde. Die Architektur, welche diese einzelnen Instruktionsrechenwerke beinhaltet, wird als „Instruction-Pipeline“ [5] bezeichnet. Eine solche mit Tiefe vier ist in Abbildung 2.5 dargestellt.

Um nun einen Befehl vollständig auszuführen, werden vier Taktzyklen benötigt. Da jedoch die vier Teilschritte unabhängig voneinander arbeiten, kann bereits der erste Teil des zweiten Befehls abgearbeitet werden, wenn der zweite Teil des ersten Befehls im zweiten Taktzyklus verarbeitet wird.

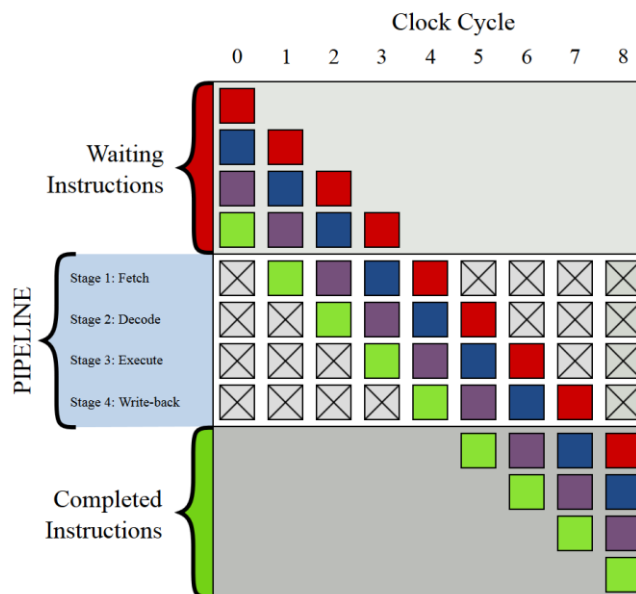


Abbildung 2.5: Abstrahierung einer Instuktions-Pipeline mit Tiefe 4. Bild von [5].

Trotzdem benötigt jeder komplette Befehl vier Taktzyklen, allerdings können vier vollständige Befehle bereits in acht Zyklen bearbeitet werden, was in Abbildung 2.5 genau der Fall ist.

Da das Pipelining nur möglich ist, wenn unabhängige Instruktionen verfügbar sind und da aus Prinzip nur unabhängige Threads in einem Warp enthalten sein sollten, teilt man den Warp auf zwei oder vier Teile auf und benutzt pro Teil eine Ausführungsgruppe des SMs. So können die einzelnen Teilwarps direkt für das Pipelining verwendet werden und der Warp ist nach zwei oder vier Taktzyklen abgearbeitet. Zusätzlich wird dadurch keine Branch-Prediction benötigt. Natürlich werden im Fall der Fermi-Architektur zwei Halbwarps verwendet, wobei jeder auf einer der 16 SP grossen Ausführungsgruppe abgearbeitet wird. Soll ein Warp mit Spezialaufgaben ausgeführt werden, wird dieser in acht Teilwarps aufgeteilt und auf den SFUs ausgeführt, weshalb dieser dann acht Taktzyklen zur kompletten Ausführung benötigt. Eine Beschränkung stellt in solchen Fällen die Anzahl Warp-Scheduler dar. Denn obwohl bei Fermi vier Ausführungsgruppen verfügbar wären, können nur maximal zwei Warps gleichzeitig ausgeführt werden.

Ein Spezialfall betreffend Warps ist das Branching. Unter einem Branch versteht man die Verzweigung von Programmcode nach bedingten Sprüngen, was in Threads innerhalb eines Warps zu unterschiedlichen Zweigen führen kann. Da in diesem Fall nur ein Zweig zur Zeit berechnet werden kann und dies folglich zu mehrfacher Ausführungszeit führt, sollte Branching innerhalb eines Warp wenn möglich verhindert werden. Dies wird erleichtert, da die Warps immer mit aufeinanderfolgenden Threads gefüllt werden. Also z.B. Thread-ID 0 bis 31. Somit sollte darauf geachtet werden, dass aufeinanderfolgende Thread-IDs dasselbe Verhalten zeigen. Diese Problematik wird in Kapitel 2.1.4 ausführlicher behandelt.

Um die optimale Performance einer GPU ausnutzen zu können, muss das Pipelining und das Shared-Memory geschickt ausgenutzt werden. Vor allem das Pipelining ist der Schlüssel zu einer guten Grund-Performance. Hierbei müssen der GPU ausreichend viele und genügend grosse Blöcke bereitgestellt werden, damit sie den einzelnen SMs ausreichend Warps zum kontinuierlichen Füllen der Pipeline liefern kann. Denn genau dieses „Latency-Hiding“ wird benötigt, um die enorme Rechenleistung einer GPU auszunutzen, was diese schliesslich so viel schneller macht als eine CPU. Allerdings ist die Wahl der Anzahl und Grösse der Blöcke nicht einfach, da es einerseits Hardwarebeschränkungen gibt, andererseits das Maximum der Werte selten zur schnellsten Ausführung führt. Deshalb ist es in den folgenden Messungen so wichtig, mit unterschiedlichen Blockgrössen zu experimentieren, um die maximale Geschwindigkeit finden zu können.

2.1.3 HSA-Begriffe

Eine weitere Schwierigkeit beim Einstieg in die Welt von HSA ist, dass jeder Hardwarehersteller aber auch jeder Standard eigene Bezeichnungen für dieselben oder ähnliche Dinge einführt. Die am häufigsten verwendeten Bezeichnungen sind in Tabelle 2.4 aufgelistet und beschrieben. Um Verwechslungen zu vermeiden, wird für allgemeine Erklärungen im Folgenden die Terminologie von OpenCL verwendet.

Beschreibung	OpenCL	nVidia CUDA	C++ AMP	OpenACC
Verbund mehrerer Rechenkerne einer GPU inklusive Cache und Steuerwerk. Bei CPU ein Kern oder bei aktivem Hyperthreading ein Thread.	Compute Unit	Streaming Multiprocessor (SM)		Processing Element (PE)
Rechenkern (Core). Vereint alle Rechenwerke eines Kerns, im Fall der GPU die ALU und FPU.	Processing Element	Streaming Processor (SP)		
Abstrakte Gruppe bestehend aus 32 Threads. Wird verwendet um die Work-Group / den Thread Block in gleich grosse Teile aufzuteilen, welche dem Scheduler der Compute-Unit übergeben werden kann. Alle Threads innerhalb des Warps sind, bis auf Barrier-Synchronisation, unabhängig.		Warp		Worker
Enthält mehrere Thread-Blöcke. Ein Grid entspricht einem Kernel. Die Blöcke im Grid werden zur Laufzeit auf die verfügbaren SMs aufgeteilt.		Grid		
Gruppe, die alle logisch zusammengehörenden Threads sammelt. Innerhalb der Work-Group können über das Local-Memory Daten ausgetauscht werden.	Work Group	Thread Block	Tile	Gang
Eine Arbeitseinheit, welche durch ein Processing-Element abgearbeitet wird.	Work Item	Thread	Thread	Vector
Der Hauptspeicher des Beschleunigers, welcher jeder Compute-Unit zur Verfügung steht. Jeder Thread kann darauf zugreifen.	Global Memory	Global Memory / L2 Cache	Global Memory	
Der Speicher, welcher nur der jeweiligen Work-Group zur Verfügung steht. Schneller und kleiner als das Global-Memory.	Local Memory	Shared Memory / L1 Cache	Tile static Memory	
Ein privater Speicher, welcher nur dem Work-Item zur Verfügung steht.	Private Memory	Private (Local) Memory		

Tabelle 2.4: Unterschiedliche Bezeichnungen für dieselben Dinge bei Hardware- und Software-Herstellern im Bereich GPU. Enthält Begriffe aus Tabelle 2.1 und Tabelle 2.2.

2.1.4 Branching

Ein Problem, das sowohl bei GPU als auch bei CPU Anwendungen auftritt ist das Branching. Dieses entsteht bei Verzweigungen im Code, z.B. durch „Conditionals“ oder durch diverse „Loops“. Während bei der Ausführung auf der CPU nur die zusätzliche Optimierung leidet, welche durch Vorhersagen der nächsten Instruktion (Branch-Prediction) erzielt wird, ergibt sich im SIMD Umfeld ein schwerwiegenderes Problem. Eine Compute-Unit kann nur bei identischen Instruktionen mehrere Daten parallel verarbeiten. Sobald unterschiedliche Instruktionsabläufe in derselben Compute-Unit ausgeführt werden sollen, muss sich diese für einen einzigen Ablaufstrang entscheiden. Somit wird die massive Parallelität vermindert.

Folgich wird im CUDA-Umfeld nicht von SIMD, sondern von Single Instruction Multiple Threads (SIMT) gesprochen. Dieser Ausdruck soll verdeutlichen, dass zwar mehrere Threads in einem Multiprozessor gleichzeitig ausgeführt werden können, aber jeder dieser Threads genau dieselbe Instruktion ausführen muss. Sobald dies nicht mehr gewährleistet ist, werden unterschiedliche Threads sequentiell ausgeführt. Zu erwähnen ist, dass das SIMT-Paradigma für die Korrektheit des Codes nicht von Bedeutung ist und nur bezogen auf Performance relevant ist.

Abbildung 2.6 zeigt die Problematik anhand des Beispiels in Bild a). Nach einem einheitlichen Teil wird eine Verzweigung durch eine If-Abfrage erzwungen, wodurch zwei unterschiedliche Abläufe folgen. Betrachtet man die Ausführung auf einer CPU mit vier Kernen in Bild b), wird der jeweilige Thread einfach den Pfad wählen, welcher beim Auswerten der Bedingung erzwungen wird und diesen ausführen. Dies ist nur

möglich, da jeder Kern der CPU ein eigenes Steuerwerk besitzt. Im Beispielfall wählen zwei Threads den Pfad A und zwei den Pfad B.

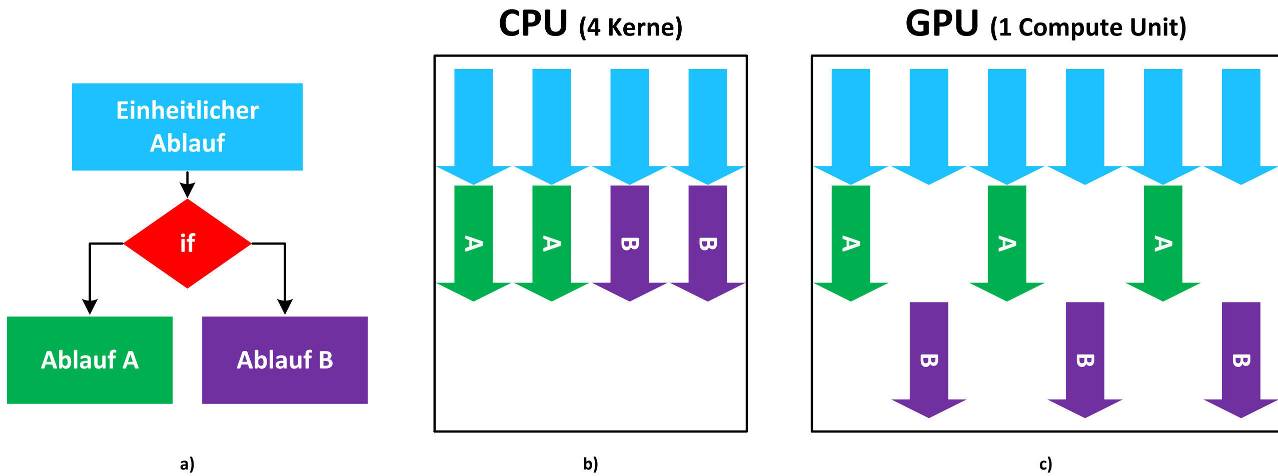


Abbildung 2.6: Branching-Problematik. a) Struktur des Programms mit logischer Verzweigung. b) Paralleler Ablauf auf Mehrkernprozessor. c) Paralleler Ablauf auf GPU mit nur einem Multiprozessor.

Die mehrfache Ausführung dieses Codes auf einer GPU führt zu einem Problem, da die Compute-Unit immer nur einen Instruktionsablauf auf einmal ausführen kann, weil sie nur ein Steuerwerk für alle Work-Units besitzt. Im Bild c) führt dies dazu, dass drei Threads mit dem Pfad A weiterrechnen und die anderen drei warten müssen, bis der Ablauf A abgeschlossen ist. In Realität wird dies meist so implementiert, dass die Compute-Unit trotzdem für alle Threads beide Pfade durchrechnet, jedoch die nicht benötigten Resultate verwirft.

In der Theorie führt diese Variante deshalb zu einer längeren Ausführungszeit: $t(A) + t(B)$. In der Praxis entscheiden jedoch auch die Problemgrösse und die Anzahl Threads pro Compute-Unit, ob dieser Fall überhaupt eintritt, oder ob der Compiler das Problem durch automatische Optimierungen verhindern kann. Trotzdem ist es ratsam, Branching zu vermeiden oder manuell zu optimieren.

Um die Theorie zu bestätigen, wird ein kurzes Testprogramm geschrieben, welches 10‘000 Multiplikationen berechnet. Dass eine Integervariable das Resultat dieser Kalkulation nicht fassen kann und es deshalb zu Überläufen kommt, ist für die Messung irrelevant. Die Berechnung wird jeweils 10‘000 mal, in zwei unterschiedlichen, Varianten durchgeführt. Bei der Variante A wird in allen Threads folgendermassen gerechnet: $1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot 10'000$. In der Variante B rechnen die Threads mit einer ungeraden ID ebenfalls die aufsteigende Variante, diejenigen mit gerader ID jedoch umgekehrt, also wie folgt: $10'000 \cdot 9'999 \cdot \ldots \cdot 1$.

Somit wird der Fall in Abbildung 2.6 c) erreicht, in welcher zwei unterschiedliche Pfade abgearbeitet werden müssen. Die Resultate der Laufzeitmessung bestätigen die Theorie und sind in Tabelle 2.5 ersichtlich.

	Daten vorbereiten [ms]	Berechnung [ms]	Resultate abholen [ms]
Variante A	0.224 ± 0.005	3.776 ± 0.062	0.633 ± 0.021
Variante B	0.223 ± 0.002	5.999 ± 0.059	0.653 ± 0.035

Tabelle 2.5: Resultate des Branching-Tests (Werte in Millisekunden). Obwohl die Problemgrösse dieselbe ist, benötigt die Variante B ca. doppelt so viel Zeit.

Im CUDA-Umfeld erkennt man das Problem beim Betrachten des Aufbaus eines Warps. Da ein Warp mit 32 aufeinanderfolgenden Threads gefüllt wird, werden durch die Unterscheidung von geraden und ungeraden

Thread-IDs 16 Threads mit Ablauf A und 16 mit Ablauf B zu einem Warp zusammengefügt. Soll dieser Warp ausgeführt werden, kann das Steuerwerk nur eine Ablaufvariante auf einmal initiieren, wodurch sich die Ausführungszeit verdoppelt.

2.1.5 Kompilierung

Es existieren weitere relevante Unterschiede in Bezug auf die GPGPU-Programmierung. Einer ist die Art der Code-Generierung, welche nicht statisch von einem Compiler des Entwicklers erledigt wird, sondern dynamisch bei der Ausführung des Kernels auf dem jeweiligen System. So kann der Vielfältigkeit der unterschiedlichen Plattformen (AMD, nVidia, Intel) und Devices (GPU, x86-CPU, x64-CPU) Rechnung getragen werden, ohne unüberschaubar viele Binärcode-Variationen kompilieren zu müssen. Dieses „Just in Time Compiling“ (JIT) kann je nach HSA manuell oder automatisch beim ersten Aufruf durchgeführt werden.

2.1.6 Problematiken bei HSA-Code

Aus den vorgängig beschriebenen Eigenheiten der beiden Plattformen ergeben sich Schwierigkeiten beim Implementieren von HSA-fähigem Code. Einerseits soll dieser allgemein gehalten werden, damit er auf allen möglichen Devices lauffähig ist, andererseits soll er so weit spezialisiert werden, um das jeweilige Optimierungspotential möglichst gut ausnutzen zu können. Zhang et al. [6] beschreiben ähnliche Problematiken und mögliche Optimierungen.

Als erstes ist die Work-Group-Grösse zu erwähnen, welche bei der Anwendung auf GPU nicht einfach zu wählen ist. Zusätzlich symbolisiert diese Grösse bei OpenCL-Code bei der Ausführung auf CPU nicht mehr die Anzahl Threads, welche zu einem Block zusammengefasst werden, sondern die Länge der Vektoreinheit der CPU, also die Anzahl Befehle, welche zu einer Vektorinstruktion zusammengefasst werden. Dabei werden Schleifen „abgerollt“ (Loop unrolling) und die Befehle, welche neu innerhalb einer einzigen Iteration durchgeführt werden, zeitgleich von einer SSE-Einheit ausgeführt. Eine gute Übersicht über Vektorisierung bietet der Guide von Sabahi [7]. Listing 2.1 zeigt ein Code-Beispiel eines Loop-Unrollings. Zu beachten ist, dass dieser Code nur bei einer `size` funktioniert, die ein Mehrfaches von vier ist.

```
// scalar code
static const int size = 8;
int a[size], b[size], c[size];
for (int i=0; i<size; i++) {
      c[i] = a[i] + b[i];
}

// vector code
static const int size = 8;
int a[size], b[size], c[size];
for (int i=0; i<size; i+=4)  {
      c[i] = a[i] + b[i];
      c[i+1] = a[i+1] + b[i+1];
      c[i+2] = a[i+2] + b[i+2];
      c[i+3] = a[i+3] + b[i+3];
}
```

Listing 2.1: Beispiel eines Loop-Unrolling. Es wird Verwaltungsaufwand der Schleife eingespart und jeweils vier Befehle in einem Durchlauf berechnet. Diese vier Befehle können zu einem SSE-Befehl zusammengefasst werden.

Nachdem die Schleife abgerollt wurde, können die vier identischen Befehle durch die SSE-Einheit innerhalb eines einzigen Taktes ausgeführt werden. Dazu werden die 128bit (oder 256bit bei AVX) breiten Zusatzregister von SSE verwendet. Diese erlauben ab SSE2 vier 32bit-Integer gleichzeitig abzuspeichern und den entsprechenden SIMD-Befehl darauf abzuarbeiten. Jedoch muss darauf geachtet werden, dass die Daten nebeneinander liegen, da die Befehle ansonsten nicht vektorisiert werden können. Zusätzlich dürfen keine Abhängigkeiten zwischen den Befehlen oder Daten bestehen, oder bedingte Verzweigungen

vorkommen. Alle möglichen Belegungen der SSE2-Register sind in Abbildung 2.7 dargestellt. Grundsätzlich unterstützt SSE die gebräuchlichsten arithmetischen Operationen. Eine Übersicht über die SIMD-Befehle bis und mit SSE2 findet man unter [8].

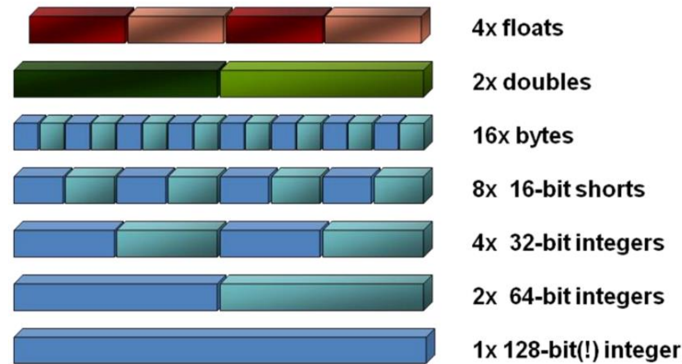


Abbildung 2.7: Mögliche Datentypen die in SSE-Register ab SSE2 passen. Bild aus [7].

Die erste Version von SSE wurde bereits 1999 vorgestellt, weshalb moderne CPUs mindestens Version 4 unterstützen. Folglich kann OpenCL die Work-Group-Grösse im CPU-Bereich für die Vektorisierung verwenden und somit diesen Code zusätzlich beschleunigen. Allerdings sind optimale Werte der Work-Group-Grösse auf CPU eher im Bereich 2-16, da sie von der Grösse der SSE-Register und des Variablentyps abhängig sind, wobei im GPU-Bereich Werte von 32-256 geeigneter sind, wegen den auszulastenden Warps. Eine Work-Group-Grösse von 1 führt zur Deaktivierung des Auto-Vektorisierers.

Eine weitere Eigenschaft ist die Datenlokalität bei der optimalen Handhabung von mehrdimensionalen Arrays. Da diese nicht mehrdimensional im Speicher abgebildet werden können, werden sie als Stream gespeichert. Auf der CPU ist C++ eine Sprache, die dazu das „Row Major“-Prinzip verwendet. Das heisst, es werden im 2D-Fall jeweils die kompletten Zeilen hintereinander in den Speicher geschrieben. Somit wird die Matrix in Abbildung 2.8, wie gezeigt, abgespeichert.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \rightarrow \{\{1,2,3\},\{4,5,6\}\} \rightarrow \{1,2,3,4,5,6\}$$

Abbildung 2.8: Speicherung einer 2D-Matrix in Row-Major-Anordnung.

Wenn dieses 2D-Array abgearbeitet werden soll, macht es aufgrund des Cachings Sinn, in der inneren Schleife durch die Zeilen zu laufen und in der äusseren Schleife auf die nächste Zeile zu springen. Diese Iterationsweise ist in Listing 2.2 gezeigt und wird als Row-Major bezeichnet. Diese Art der Iteration nutzt die Eigenschaft aus, dass der Cache bei einer Speicheranfrage mehr als nur die verlangte Speicherzelle in den Cache lädt. Meistens wird eine ganze Page geladen, welche danach ohne Verzögerung weiterverwendet werden kann. Da die Page die nachfolgenden Speicherzellen der aktuellen Zeile enthält, sollten diese verarbeitet werden und nicht auf die nächste Zeile gesprungen, der Cache gelöscht und eine neue Page angefordert werden. Dies ist jedoch nur bei genügend grossen Arrays der Fall, in welchen nicht auch die nächste Zeile in derselben Page gespeichert ist und sich deshalb bereits im Cache befindet.

```
// column-major
for (int c=0; c<columns; c++)
     for (int r=0; r<rows; r++)
          // data processing

// row-major
for (int r=0; r<rows; r++)
     for (int c=0; c<columns; c++)
          // data processing
```

Listing 2.2: Beispielcode für die beiden unterschiedlichen Iterationsvarianten.

Diese Funktionalität des Caches wird von der CPU ausgenutzt. Es zeigt sich, dass ein einzelner CPU-Thread möglichst benachbarte Daten bearbeiten sollte, wobei die Problemdaten z.B. für einen Vierkern-Prozessor

gleichermassen auf vier Teile (z.B. vier Zeilen) aufgeteilt werden. Anders dagegen arbeitet die GPU mit Work-Groups, welche jeweils möglichst zusammenhängende Daten verarbeiten sollten. Dies, da die neu zu bearbeitenden Daten für alle Cores jeweils gleichzeitig aus dem Speicher geholt werden. Da das Global-Memory als Streaming-Memory betrachtet werden muss, welches bei sequentiellem Datenzugriff die beste Performance aufweist, ist es optimal, wenn die Daten pro Work-Group nebeneinander liegen. Dadurch sind die Daten für die einzelnen GPU-Threads, und somit auch der einzelnen Aufgaben, nicht zusammenhängend.

Betrachtet man die Ausführung eines solchen Kernels, wird als erstes die Work-Group 0 geladen und ausgeführt. Das heisst, jeder Core hat danach den ersten Teil seines Aufgabenbereichs erledigt (dies entspricht der ersten Spalte der 2D-Daten). Als nächstes wird die Work-Group 1 geladen und somit jeweils der zweite Teil der Aufgabe jedes Cores erledigt; und so weiter. Der Softwarecache der GPU lädt also jeweils explizit die Daten für die zu bearbeitende Work-Group und muss sich nicht auf automatische Caching-Strategien verlassen. Wie die Ergebnisse aus [6, 9] zeigen, verhält sich daher eine Column-Major-Anordnung auf einer GPU optimaler. Abbildung 2.9 zeigt eine Abstraktion der beiden Varianten.

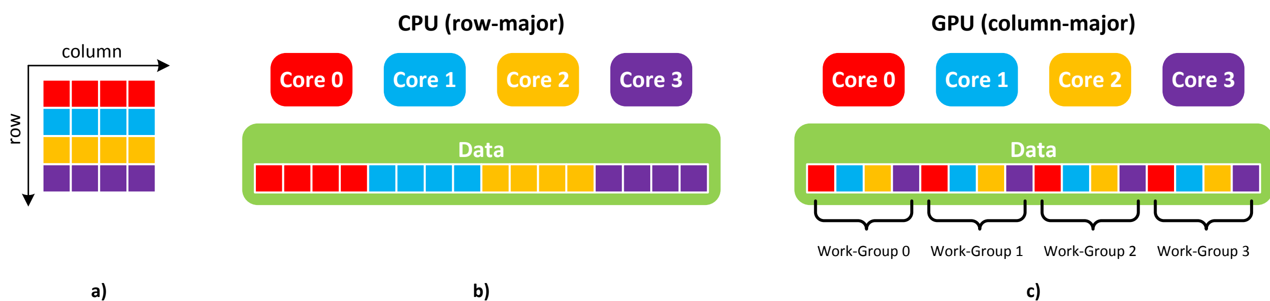


Abbildung 2.9: Unterschied zwischen b) Row-Major- und c) Column-Major-Anordnung. a) zeigt der Aufbau der Problemdaten. In c) sind vier Kerne einer einzelnen Compute-Unit gezeigt. Die Farben markieren unterschiedliche zusammenhängende Aufgabenteile, die von den Cores abgearbeitet werden.

Da sich jedoch nicht nur der Iterationsablauf unterscheidet, sondern ein Softwarecache auf der GPU zum Einsatz kommt, sorgt dieser beim Ausführen des HSA-Codes auf der CPU unter Umständen zu Performance-Einbussen, da er emuliert werden muss.

Des Weiteren tauchen technologiespezifische Unterschiede zwischen CPU- und GPU-Ausführung des HSA-Codes auf; besipielsweise die Ausführung auf CPU mittels WARP-Emulator unter AMP.

2.2 Testanwendung

Als Testanwendungsfall für eine HSA sind nur Algorithmen geeignet, welche parallelisierbar und rechenintensiv sind. Da sich die gesamte Masterarbeit mit der Thematik der Bildverarbeitung befasst, ist die Faltung eines Bildes naheliegend. Zudem ist die Komplexität dieses Problems über die Grösse der Eingangsmatrizen steuerbar. Listing 2.3 zeigt den vereinfachten und nicht optimierten C++ Code, welcher bereits OpenCV mit der Klasse `cv::Mat` und `cv::Point` verwendet.

```
cv::Mat input, filter, output;
cv::Point anchor;

// calculate weight of filter
int filterWeight = 0;
for (int y=0; y<filter.rows; y++) {
      for (int x=0; x<filter.cols; x++) {
            filterWeight += filter.at<uchar>(y, x);
      }
}

// iterate through all pixels of destination image
for (int v=0; v<output.rows; v++) {
      for (int u=0; u<output.cols; u++) {

            // iterate through all pixels of filter kernel
            int value = 0;
            for (int y=0; y<filter.rows; y++) {
                  for (int x=0; x<filter.cols; x++) {
                        int posX = u + x - anchor.x;
                        int posY = v + y - anchor.y;
                        value += input.at<uchar>(posY, posX) * filter.at<uchar>(y, x);
                  }
            }

            // calculate final value and set it to destination-image
            output.at<uchar>(v, u) = value / filterWeight;
}     }
```

Listing 2.3: Beispielanwendung der Faltung als sequentieller CPU-Code.

Bis auf die Verwaltung der Eingangs-/Ausgangsmatrizen und die Randbehandlung, ist der Code komplett. Dieser dient als Startpunkt für die jeweiligen HSA-Implementierungen und bietet zudem die Möglichkeit, die Korrektheit der berechneten Resultatbilder zu überprüfen. Die Laufzeitanalyse dieses Codes bietet hingegen nur geringe Vergleichsmöglichkeiten betreffend den Beschleunigungsmöglichkeiten von HSAs. Dies liegt vor allem daran, dass der Code ohne jegliche Optimierung oder Parallelisierung auskommt. Der Aufbau der Messungen und die Rahmenbedingungen etc. weden im Kapitel 2.6 genauer beschrieben.

2.3 C++ AMP

Microsoft C++ Accelerated Massive Parallelism (AMP) ist eine offene Spezifikation für heterogene Systeme. Sie enthält Klassen für die Abstraktion von Beschleunigern, aber auch andere Techniken zur Parallelisierung wie z.B. Tasks. Aktuell ist die einzig verfügbare Implementierung von Microsoft, welche stark in Visual Studio 2012 integriert wurde und sehr viele Tools zur Unterstützung bietet. Dadurch, dass AMP das objektorientierte Modell von C++ verwendet und teilweise bereits Konzepte von C++11 implementiert, bietet es eine bessere Abstraktion und einfacher zu verwendende Module, als andere auf C basierende Technologien.

2.3.1 Installation

Die Verwendung von AMP wird ein Visual Studio 2012 inklusive des dazugehörigen Compilers von Microsoft benötigt. Als Beschleuniger können alle Direct3D fähigen GPUs zum Einsatz kommen. Um GPU-Debugging und den WARP-Emulator unter Windows 7 verwenden zu können, wird ein Plattform-Update [10] benötigt.

2.3.2 Einstieg in die Technologie

Um in die Welt von AMP einzutauchen und die Technologie verwenden zu lernen, sind diverse Quellen verfügbar. MSDN [11] bietet eine übersichtliche Einführung in die Bestandteile von AMP, wobei zusätzliche

Web-Blogs z.B. von MSDN [12, 13] oder andere [14] die Teilbereiche vertiefen. Sehr hilfreich und umfassend ist vor allem das Buch von Kate Gregory [15], welches sich stark mit GPGPU-Programmierung auseinandersetzt und einen guten Einstieg in AMP ermöglicht. AMP besteht aus einer Bibliothek und einer Spracherweiterung, wobei nur die beiden Schlüsselwörter `restrict` und `tile_static` hinzukommen. Die Bibliothek hingegen beinhaltet sehr viele neue Klassen, welche eine Abstraktion der Beschleuniger erlauben. Zudem verwendet AMP vor allem das in C++11 neue Paradigma der Lambdas. Diese werden benötigt, um den „Kernel", also die Funktion, welche auf dem Beschleuniger ausgeführt werden soll, zu definieren.

Listing 2.4 zeigt den grundsätzlichen Aufbau eines solchen Lambda-Aufrufs, welcher mehrmals, wenn möglich parallel, auf dem Beschleuniger ausgeführt wird. In diesem Beispiel wird ein C-Array mit fünf Elementen erzeugt, für welches dann ein `array_view` generiert wird. Diese „View" lässt sich als Referenz auf die effektiven Daten betrachten und wird mit dem Typ der Daten und der Anzahl Dimensionen (Rang) getypt. Der hierzu verwendete Konstruktor benötigt die Grösse des Arrays und den Zeiger darauf. Für andere Arten von Daten existieren diverse weitere Konstruktoren, z.B. für eine Datenstruktur, welche Iteratoren verwendet. Die View bietet die Abstraktion über die Kopiervorgänge der Daten auf den jeweiligen Beschleuniger und lässt den Benutzer darüber auf die Daten zugreifen.

```
int a[] = {1,2,3,4,5};
array_view<int, 1> av(5, a);
parallel_for_each(av.extent, [=](index<1> idx) restrict(amp) {
      av[idx] *= 7;
});
```

Listing 2.4: Beispielcode mit AMP und Verwendung eines Lambdas.

Die Funktion `parallel_for_each` nimmt den Bereich der zu prozessierenden Daten in Form eines `extent` und ein Lambda, in welchem der Kernel als `restrict(xx)` definiert ist. Anstelle von `xx` kann `cpu`, `amp`, `direct3d` oder auch die Kombination dieser stehen. Damit wird definiert, auf welcher Art von Prozessor der Code ausgeführt werden darf, wobei `amp` für alle von AMP unterstützten Beschleunigern steht. Mithilfe der Klassen `accelerator` und `accelerator_view` kann gewählt werden, auf welchem Beschleuniger der Code ausgeführt werden soll. Dabei wird als Default immer die beste GPU im System verwendet. Zusätzlich wird unter Windows 7 und 8 die „Windows Advanced Rasterization Platform" (WARP) [16] angeboten, welche als Fallback dient, falls in einem System keine GPU verfügbar ist. WARP emuliert eine GPU und nutzt dazu alle Cores und eventuellen SIMD-Fähigkeiten der CPU aus. Trotzdem kann WARP nicht als Ersatz für echte CPU-Parallelisierung betrachtet werden. Dazu soll die Microsoft „Parallel Pattern Library" (PPL) verwendet werden. Das gross geschriebene WARP im AMP-Umfeld darf dabei nicht mit dem normal geschriebenen Warp von nVidia verwechselt werden.

Das Lambda muss jeweils einen Parameter für die Indexierung des zu prozessierenden Bereichs haben. In Listing 2.4 wird dies mithilfe der Klasse `index` gemacht, welche, wie die View, mit dem Rang getypt wird. Über diesen Index lässt sich jedes zu prozessierende Element über die View ansprechen. Anzumerken ist zudem, dass der Lambda-Body auf alle ausserhalb definierten Variablen mit der in `[]` definierten Art Zugriff hat. Einschränkungen dieser Definition sind jedoch, dass `array_views` nur „by Value" und `arrays` nur „by Reference" angesprochen werden dürfen. Ein ähnliches Beispiel findet sich im Kapitel „C++ AMP Overview" von [11].

Das gezeigte Beispiel lässt den Code des Lambdas nun auf einem AMP-Beschleuniger laufen, welcher im Fall einer GPU mehrere Iterationen parallel verarbeitet. Somit wird der Code automatisch parallelisiert. Um die besonderen Eigenschaften von GPUs oder anderen Beschleunigern besser ausnutzen zu können, lässt sich der Code durch diverse Veränderungen und spezielle Konstrukte optimieren.

Die dabei wichtigste Variante ist das sogenannte „Tiling". Es ermöglicht, die einzelnen Iterationen zu „Tiles" zusammenzufassen. Diese Tiles entsprechen den Work-Groups in OpenCL und werden als Einheit betrachtet, welche auf gemeinsamen Daten operiert. Dazu kann pro Tile ein spezieller Cache (Local Memory) des Beschleunigers ausgenutzt werden, welcher meist ca. eine um den Faktor 1000 bessere Zugriffszeit als der globale Speicher hat. Um das in Listing 2.4 gezeigte Beispiel für das Tiling vorzubereiten,

muss das `extent` und der `index` wie in Listing 2.5 angepasst werden. Die Methode `tile()` des `extent` unterteilt diesen in gleich grosse Tiles, welche in diesem Fall jeweils zwei Elemente enthalten. Die zwingende Regel, dass ein `extent` ein Vielfaches der Tile-Grösse sein muss, kann im einfachsten Fall mit dem Aufruf der Methode `pad()` sichergestellt werden. Des Weiteren muss die angepasste Indexklasse `tiled_index` verwendet werden, welche wie die Methode `tile()` mit der Tile-Grösse getypt wird. Da durch das Padding das `extent` grösser ist als der effektive Speicherbereich, muss bei jedem Schreibzugriff getestet werden, ob das jeweilige Feld im ursprünglichen `extent` enthalten ist. Dies wird mit der if-Abfrage sichergestellt. Zudem muss über das Feld `global` der Indexklasse auf die bisherig verwendeten Koordinaten zugegriffen werden.

```
parallel_for_each(av.extent.tile<2>().pad(), [=](tiled_index<2> idx) restrict(amp) {
      if (av.extent.contains(idx.global)) {      // test for padding
            av[idx.global] *= 7;
      }
});
```

Listing 2.5: Erweiterung des „parallel_for_each“ Aufrufs mit Tiling.

Die Klasse `tiled_index` verhält sich anders als die normale Indexklasse `index`. Sie muss zwischen dem globalen und dem lokalen Index, innerhalb des Tiles, unterscheiden können. Dazu enthält sie die Felder, welche in Tabelle 2.6 ersichtlich sind und ermöglicht dadurch den gezielten Zugriff auf die entsprechenden Positionsdaten.

Feldname	Beschreibung
`global`	Globale Position im Problembereich
`local`	Tile-lokale Position im Problembereich
`tile_origin`	Ursprungspunkt des aktuellen Tiles
`tile`	Der Index des aktuellen Tiles

Tabelle 2.6: Die Felder der Klasse „tiled_index“, welche für die gezielte Indexierung bei Tiling benötigt werden.

Um einen Tilecache innerhalb des Lambda-Bodys zu definieren, kommt das Schlüsselwort `tile_static` zum Einsatz. Der Code in Listing 2.6 zeigt alle Ergänzungen, um die Daten zu cachen und am Schluss beim Berechnen des Resultats zu verwenden. Natürlich sollten nur Daten gecached werden, welche mehrmals gelesen werden, da sich ansonsten der Mehraufwand nicht auszahlt.

```
parallel_for_each(av.extent.tile<2>().pad(), [=](tiled_index<2> idx) restrict(amp) {
      tile_static int cache[2];                 // create cache
      cache[idx.local[0]] = av[idx.global];     // caching
      idx.barrier.wait();                       // wait for end of caching

      if (av.extent.contains(idx.global)) {      // test for padding
            av[idx.global] = 7 * cache[idx.local[0]];
      }
});
```

Listing 2.6: Beispielcode für die Verwendung des Tile-Cache in AMP.

Weitere Informationen zum Thema Tiling findet man in [11] im Kapitel „Using Tiles“ oder im Buch [15] im Kapitel 4 „Tiling“.

2.4 OpenCL

Der Standard 1.2 der offenen Spezifikation OpenCL ist seit Ende 2011 verfügbar. Die Version 1.0 wurde bereits Ende 2008 veröffentlicht, weshalb OpenCL vermutlich als die bekannteste HSA gilt.

Dementsprechend unterstützen sehr viel Hardwarehersteller diesen Standard und liefern entsprechende Treiber zu ihren Produkten. Die auf dem Beschleuniger auszuführenden Kernel werden in einer erweiterten Variante des C99-Standards geschrieben. Die Verwaltung dieser Kernel wird hostseitig über eine API erledigt, welche ursprünglich für C, neuerdings aber auch als C++-Variante verfügbar ist.

2.4.1 Installation

Um OpenCL verwenden zu können, wird grundsätzlich nur der jeweilige Hardwaretreiber benötigt. Im Fall von Intel wird dies mit dem SDK [17] für OpenCL erledigt, nVidia hingegen liefert alles im CUDA-Treiber [18] mit. Danach muss noch im Projekt auf die entsprechenden `include`- und `lib`-Verzeichnisse verwiesen und die OpenCL.lib dem Linker angegeben werden. Unter Windows befindet sich bei nVidia z.B. alles Benötigte im folgenden Verzeichnis:

```
c:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\
```

Um die C++-API verwenden zu können, muss zusätzlich die Datei cl.hpp [19] von der Khronos-Webseite heruntergeladen und im Ordner `\include\CL\` platziert werden. Im Projekt muss diese dann mit folgendem Include eingebunden werden:

```
#include <CL\cl.hpp>
```

Wenn mehrere Geräte von unterschiedlichen Herstellern verwendet werden, muss trotzdem nur auf eine Bibliothek verwiesen werden. Der in allen Implementationen enthaltene „Installable Compiler Driver“ Loader (ICD Loader) stellt sicher, dass alle installierten OpenCL-Beschleuniger erkannt werden und auswählbar sind.

Falls Visual Studio 2010 als IDE verwendet wird, kann diese durch das gratis erhältliche nVidia Nsight [20] mit OpenCL-Unterstützung erweitert werden. Allerdings wird dies für die OpenCL-Entwicklung nicht zwingend benötigt.

2.4.2 Einstieg in die Technologie

Als Einführung und zur Überblickgewinnung stellt Khronos ein PDF [21] zur Verfügung, worin grundsätzliche Konzepte erklärt und Referenzen auf Lehrbücher enthalten sind. Ansonsten finden sich viele Tutorials zu OpenCL im Netz. Ergänzend zu diesen wurde eine Tutorialserie [22] aus dem Technikmagazin iX des Heise-Verlags verwendet, welche kompakt in die Grundlagen und erweiterte Problemstellungen einführt. Als Nachschlagewerk kann die Quick-Reference-Card [23] verwendet werden.

Anders als bei AMP oder OpenACC kann nicht einfach bestehender Code durch Annotationen oder kleine Änderungen für die HSA vorbereitet werden. Vielmehr muss jeweils ein echter Kernel in OpenCL-C geschrieben werden. Dieser kann entweder direkt im eigentlichen Code als String oder als separate Quelldatei abgelegt werden. Der Kernel stellt dabei den Code dar, welcher effektiv auf dem Beschleuniger ausgeführt wird. Das heisst, dieser Code muss so aufgebaut werden, dass er mehrmals parallel mit unterschiedlichen Parametern aufgerufen werden kann. Listing 2.7 zeigt einen solchen Kernel, welcher aus einer normalen Funktionsimplementierung und dem vorangehenden Schlüsselwort `__kernel` besteht.

```
// kernel.cl
__kernel void MultSeven(__global uchar* a) {
    a[get_global_id(0)] *= 7;
}
```

Listing 2.7: Beispielcode eines OpenCL-Kernels.

Damit der Kernel weis, welches Work-Item er vom gesamten Problembereich verarbeiten muss, kann mit der Funktion `get_global_id(dim d)` der globale Index der Dimension `d` abgefragt werden. Die Definition und Aufteilung des Problembereichs wird in OpenCL im Host-Teil des Programms erledigt. Um zusätzliche Daten über den Problembereich im Beschleuniger-Teil des Codes abzufragen und auch Optimierungsstrategien einbauen zu können, werden die Funktionen aus Tabelle 2.7 benötigt.

Funktion	Rückgabewert
`uint get_work_dim()`	Anzahl verwendeter Dimensionen
`size_t get_global_size(uint D)`	Globale Anzahl Work-Items
`size_t get_global_id(uint D)`	Globale ID des aktuellen Work-Items
`size_t get_local_size(uint D)`	Lokale Anzahl Work-Items
`size_t get_local_id(uint D)`	Lokale ID des aktuellen Work-Items
`size_t get_num_groups(uint D)`	Anzahl Work-Groups
`size_t get_group_id(uint D)`	ID der aktuellen Work-Group
`size_t get_global_offset(uint D)`	Globaler Offset zur aktuellen Work-Group

Tabelle 2.7: Eingebaute Funktionen für Datenabfrage betreffend Problembereich.

Des Weiteren existieren mehrere Kennzeichner, um den Speicherort und den Typ von Daten genauer zu definieren und dadurch dem OpenCL-Compiler mehr Informationen für Optimierungsstrategien zu liefern. Tabelle 2.8 zeigt diese zusätzlichen Schlüsselwörter. Die Variante `__private` ist dabei der Default-Wert und sollte für alle Funktionsparameter verwendet werden. Die drei Weiteren werden für Zeiger verwendet, wobei kein Casting zwischen unterschiedlichen Typen erlaubt ist.

Kennzeichner	Beschreibung
`__global`	Die Daten werden im globalen Speicher (RAM) abgespeichert und alle Compute-Units können gleichzeitig darauf zugreifen.
`__constant`	Gleich wie `__global`, nur sind die Daten nicht veränderbar.
`__local`	Die Daten sind nur lokal für die aktuelle Work-Group sichtbar. Es werden die kleinen aber schnellen Caches verwendet.
`__private`	Die Daten sind nur im aktuellen Work-Item sichtbar.

Tabelle 2.8: Speicherplatzkennzeichner für OpenCL-Kernels.

Um den Kernel-Code für den Beschleuniger zu kompilieren und auszuführen, werden diverse Schritte im Host-Code benötigt. Diese werden in Listing 2.8 aufgezeigt. Als erstes muss ein entsprechendes Device gesucht werden, auf welchem OpenCL-Code ausgeführt werden kann. Die einfachste Variante nimmt die erste Plattform und den ersten Beschleuniger aus der Liste. Danach muss der Kernel-Code eingelesen und für die Ausführung vorbereitet werden. Nachdem die Daten mit einem entsprechenden `Buffer`-Objekt auf die GPU geladen wurden, kann der Kernel ausgeführt werden.

```
// enable C++-Exceptions
#define __CL_ENABLE_EXCEPTIONS

try {

// get available platforms
std::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);

// get specific context
cl_context_properties cp[] = \
      {CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms[0])(), 0};
cl::Context context(CL_DEVICE_TYPE_GPU, cp);

// read source file
std::ifstream sourceFile("kernel.cl");
std::string code(std::istreambuf_iterator<char>(sourceFile), \
      (std::istreambuf_iterator<char>()));

// get a list of devices on this platform
std::vector<cl::Device> devices = context->getInfo<CL_CONTEXT_DEVICES>();

// create a command queue and use the first device
cl::CommandQueue queue(context, devices[0]);

// load source
cl::Program::Sources source(1, std::make_pair(mSourceCode->c_str(), code->length()+1));

// make program of the source code in the context
cl::Program program(context, source);

// build program for these specific devices
program.build(devices);

// make kernel
cl::Kernel kernel(program, "MultSeven");

// upload data
int a[] = {1,2,3,4,5};
const int size = 5;
cl_int* aArray = new cl_int[size];
for (int i=0; i<size; i++) {
      aArray[i] = a[i];
}

cl::Buffer aBuffer(context, CL_MEM_READ_WRITE, size*sizeof(cl_int), (void*)aArray);

// set arguments to kernel
kernel.setArg(0, aBuffer);

// run the kernel on specific ND range and wait for execution
cl::Event event;
cl::NDRange offsetRange(0);
cl::NDRange globalRange(size);
cl::NDRange localRange(1);
queue.enqueueNDRangeKernel(kernel, offsetRange, globalRange, localRange, NULL, &event);
event.wait();

// receive the result buffer
queue.enqueueReadBuffer(aBuffer, CL_TRUE, 0, size*sizeof(cl_int), (void*)aArray);

} catch (Error err) {}
```

Listing 2.8: Vorbereitung und Ausführung des OpenCL-Kernels.

Mit der Methode `enqueueNDRangeKernel` wird der Kernel in die Warteschlange eingereiht. Dabei wird zudem der zu bearbeitende Problembereich definiert. Dies geschieht mit den drei `NDRange`-Variablen: `offsetRange`, `globalRange` und `localRange`. Dabei definiert die `globalRange` die Grösse des Problembereichs. Die `offsetRange` definiert die Abweichung von 0, falls die Daten nicht bei Index 0 beginnen. Ähnlich wie bei dem Tiling von AMP kann bei OpenCL eine Work-Group-Grösse angegeben werden, um manuell zu definieren, wie viele Work-Items zu einer Work-Group zusammengefasst und gemeinsam ausgeführt werden sollen. Dies geschieht mithilfe des `localRange`. Falls die Work-Group-Grösse nicht manuell gewählt werden soll oder es keinen Offset gibt, können die jeweiligen Variablen durch `NullRange` ersetzt werden.

Ebenfalls wie bei AMP muss beim manuellen Einstellen der Work-Group-Grösse sichergestellt werden, dass die `globalRange` ohne Rest durch die `localRange` teilbar ist. Auch hier ist die simpelste Variante ein Padding, wobei als erstes die Problemgrösse (`size`) auf ein Vielfaches der `localRange` erweitert werden muss, wie in Listing 2.9 gezeigt.

```
int globalSize = size + (localWorkSize - (size % localWorkSize));
cl::NDRange globalRange(globalSize);      // whole problem
cl::NDRange localRange(localWorkSize);    // problem part
```

Listing 2.9: Erweitern des Problembereichs auf ein Vielfaches der Teilgrösse.

Danach muss im Kernel verhindert werden, dass er auf einem undefinierten Wertebereich, also im Bereich des Paddings, ausgeführt wird. Listing 2.10 zeigt den Beispielcode mit einer solchen Ergänzung. Wichtig zu erwähnen ist, dass die Grösse des Problembereichs zwingend an den Kernel übergeben werden muss, da dieser ansonsten nicht entscheiden kann, welcher Bereich zum Padding gehört und welcher nicht.

```
// kernel.cl
__kernel void MultSeven(__global uchar* a, const int size) {
      if (get_global_id(0) < size) {
            a[get_global_id(0)] *= 7;
      }
}
```

Listing 2.10: Einfache Erkennung des Paddings im OpenCL-Kernel.

Um die optimale Grösse der Work-Group zu finden, müssen verschiedene Werte getestet werden. Allerdings sind solche Werte immer an die Hardware gebunden und können daher auf anderen Systemen zu stark abweichenden Resultaten bei Performance-Messungen führen.

2.5 OpenACC

Der offene Standard für die Abstraktion unterschiedlicher Beschleuniger durch Code-Annotationen OpenACC befindet sich momentan in Version 1.0 mit einem aktuellen Entwurf für Version 2.0. Das Ziel des Standards ist, alle Ressourcen der Beschleunigerhardware ohne spezielle Kenntnisse dieser durch einfache Annotationen auszunutzen, ähnlich wie dies OpenMP 3.1 für Mehrkernrechner ermöglicht. Als Langzeitziel ist deshalb die Fusionierung der beiden Standards zu OpenMP Version 4.0 vorgesehen, mit welcher man ein einziges Mittel für die Ausnutzung der kompletten PC-Hardware zur Verfügung haben wird.

2.5.1 Installation

Wie auch AMP und OpenCL schreibt der Standard nur die Schnittstellen und die übergeordneten Verwaltungsstrukturen, wie z.B. das Memory-Modell, vor. Wie der Standard schlussendlich implementiert wird, ist offen. Aktuell bieten die folgenden drei Firmen Implementierungen von OpenACC an:

- The Portland Group: PGI Accelerator Compiler
- CAPS Entreprise: CAPS Compilers
- Cray Corporation: Compiling Environment

Dabei sind nur die ersten beiden für Standard-PC-Hardware geeignet. Da jedoch beide kommerziell vertrieben werden, ist eine kostenlose Verwendung nur über einen zeitlich begrenzten Testzugang [24, 25] möglich. CAPS bietet eine Testversion des Compilers für Linux-Systeme an.

Wichtig ist als erstes die korrekte Installation des CUDA-Treibers, Compilers und der Beispiele [26]. Danach kann der CAPS-Compiler mit der gelieferten `run`-Datei installiert und aktiviert [27] werden. Zu beachten ist, dass die Umgebungsvariablen der Konsole korrekt initialisiert werden. Dazu wird die Datei `.bashrc` mit den Zeilen aus Listing 2.11 ergänzt.

```
# Cuda Env
export CUDA_HOME=/usr/local/cuda-5.0
export LD_LIBRARY_PATH=${CUDA_HOME}/lib64:${CUDA_HOME}/lib
export PATH=$PATH:${CUDA_HOME}/bin:${LD_LIBRARY_PATH}

# OpenCL Env
export OPENCL_HOME=${CUDA_HOME}
export OPENCL_INC_PATH=${OPENCL_HOME}/include
export OPENCL_LIB_PATH=${OPENCL_HOME}/lib64:${OPENCL_HOME}/lib

# CAPS Env
source '/home/christian/CAPSCompilers-3.3.2/bin/caps-env.sh'
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/christian/CAPSCompilers-3.3.2/lib
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/christian/CAPSCompilers-3.3.2/slib

# OpenCV Env
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

Listing 2.11: Umgebungsvariablen für CUDA, OpenCL, CAPS-Compiler und OpenCV unter Linux.

Des Weiteren können diverse Probleme mit Abhängigkeiten zu anderen Tools auftretten, weshalb die Pakete in Listing 2.12 installiert werden sollten.

```
sudo apt-get install gcc
sudo apt-get install g++
sudo apt-get install ia32-libs
```

Listing 2.12: Benötigte Pakete des CAPS-Compilers unter Linux für die Kompilation von OpenACC-Code.

Danach kann mit dem im Ordner `check_installation/` mitgelieferten Test-Skript geprüft werden, ob alle Voraussetzungen für den CAPS Compiler `capsmc` erfüllt sind. Zudem sollte beachtet werden, welche CUDA-Version von der GPU unterstützt wird. Da `capsmc` ohne weitere Definitionen nur Code für CUDA Compute-Capability 2.0 und höher generiert, muss bei älterer Hardware einer der beiden auf [28] beschriebenen Anpassungen vorgenommen werden.

2.5.2 Einstieg in die Technologie

Die betrachteten Implementationen von PGI und CAPS unterscheiden sich grundsätzlich von den bekannten für AMP oder OpenCL. Denn anstatt direkt ausführbaren Code zu generieren, erzeugen sie aus den

annotierten Schleifen jeweils CUDA-C Code, welcher dann normal vom nVidia-Compiler und dem verwendeten C/C++ Compiler verarbeitet wird. Somit stellt der OpenACC Compiler nur eine Vorstufe, also die Extraktion und Generierung der GPU Kernel, vor der endgültigen Kompilation dar. Durch die Verwendung von CUDA bieten sie zudem keine Kompatibilität zu GPUs anderer Herstellern oder der Ausführung des parallelen Codes auf CPUs.

Die Annotation geschieht unter C/C++ mithilfe von `#pragma` Compilerbefehlen. Eine Einführung und erweiterte Informationen, unter anderem als Video-Podcasts, findet man auf der offizielle Webseite [29] des OpenACC Standards unter dem Punkt „Education". Zusätzlich sind unter „Download Area/ Specifications" die genauen Spezifikationen verfügbar. Zwei ausführliche Einführungen in OpenACC stellt zudem die Dr. Dobbs Webseite [30, 31] zur Verfügung. Nachfolgend eine Liste von verfügbaren Pragmas in C/C++:

- `#pragma acc kernels`
- `#pragma acc parallel`
- `#pragma acc data`
- `#pragma acc loop`
- `#pragma acc wait`

Die oben aufgeführten Pragmas stehen entweder direkt vor einer Schleife oder öffnen einen Codeblock mit geschweiften Klammern. Um die Funktionalität genauer zu definieren, können sie zusätzlich mit unterschiedlichen Klauseln erweitert werden. Die wichtigsten sind die `copy`, `copyin`, `copyout` Klauseln, welche dem Compiler Optimierungen beim Kopieren der Daten auf den Beschleuniger und zurück ermöglichen. Die einfachste Möglichkeit eine bestehende Schleife parallel auf einem Beschleuniger auszuführen, ist die `kernels` Annotation. Diese überlässt es dem Compiler, eine optimale Aufteilung der Schleife und der Datenallokation zu finden. Listing 2.13 zeigt ein einfaches Beispiel dieser Variante. Zusätzlich wird bereits festgelegt, wie und welche Daten in der Schleife kopiert werden. Da es sich in diesem Beispiel um eine Variable mit Eingabe- und Ausgabewerte handelt, müssen die Daten jeweils vor der Ausführung vom Host auf das Device und danach vom Device auf den Host kopiert werden.

```
int a[] = {1,2,3,4,5};
#pragma acc kernels copy(a[0:5])
for (int i=0 ; i<5 ;i++) {
      a[i] *= 7;
}
```

Listing 2.13: Beispielcode mit OpenACC-Annotation.

Die Variationen von `copy` mit den `in` und `out` Suffixe bewirken, dass die Daten jeweils nur in die eine Richtung kopiert werden. Falls es sich bei den Variablen um Zeiger auf Daten handelt, muss jeweils zwingend der zu kopierende Bereich der Daten angegeben werden. Dieser wird in eckigen Klammern angegeben `copyin(a[0:N])`, wobei der erste Wert den Anfangsindex, der zweite Wert (`N`) die Anzahl Werte definiert. Je nach Implementation können diese Definitionen jedoch vom Standard abweichen.

Um mehr Kontrolle über das Verhalten des Beschleunigers und der generierten CUDA-Kernel zu erhalten, können die in Listing 2.14 gezeigten Annotationen verwendet werden. Mit dem `#pragma acc data` kann definiert werden, wann Daten vorgängig auf das Device kopiert werden und wie lange diese dort verfügbar sind. Dabei sind erneut die `copy` Klauseln sehr wichtig, wobei die Varianten mit dem Präfix `present_or_` zum Zuge kommen. Diese veranlassen, dass überprüft wird, ob die zu übertragenden Daten bereits kopiert worden sind und führen die Übertragung folglich nur dann aus, wenn dies noch nicht der Fall ist. Zur einfacheren Handhabung können diese in Kurzform nur mit einem vorangehenden „p", wie z.B. in `pcopy`, geschrieben werden.

Mit `#pragma acc parallel` wird, ähnlich wie bei OpenMP, der Bereich definiert, in welchem parallele Ausführungen stattfinden und die benötigten Gangs und Vectors generiert und vorbereitet werden. Dabei entspricht eine Gang einer Work-Group und ein Vector einem Work-Item (siehe Tabelle 2.4). Die Definition des eigentlichen Parallelisierungsbereiches findet mit der `loop` Annotation statt.

```
#pragma acc data pcopy(A[0:size][0:size]) pcopyin(B[0:size][0:size])
{
	// do something
	#pragma acc parallel num_gangs(16), vector_length(32)
	{
		// do something
		#pragma acc loop gang
		for (int x=0; x<size; x++) {
			#pragma acc loop vector
			for (int y=0; y<size; y++) {
				A[x][y] *= B[x][y];
}	}	}	}
```

Listing 2.14: Zur Kontrolle des Beschleunigerverhaltens werden die Pragmas „data“, „parallel“ und „loop“ eingesetzt. Zudem lassen sich die Grössen der Gangs, Workers und Vectors einstellen.

Mit der `loop` Annotation kann zudem festgelegt werden, ob die jeweilige Schleife auf Gang-, Worker- oder Vector-Parallelität gemappt wird. Die jeweilige Anzahl kann bei einem beliebigen Pragma, typischerweise bei `#pragma acc parallel`, mit den Klauseln `num_gangs` (Anzahl Work-Groups), `num_workers` (Anzahl Warps) und `vector_length` (Work-Group-Grösse) eingestellt werden. Diese Definitionen lassen spezifische Optimierungen des Codes zu, welche jedoch immer architektur- und problemspezifisch sind und deshalb jeweils neu angepasst werden müssen. In Listing 2.14 werden 16 Work-Groups mit jeweils 32 Work-Items generiert. Das heisst, dass ein Grid mit 16 Blöcken à 32 Threads erstellt wird, welches von den CUDA SMs abgearbeitet wird. Eine solche Aufteilung macht z.B. bei einer Fermi-GPU Sinn, bei der die 16 Blöcke auf die 16 SMs und die 32 Threads auf die 32 Cores pro SM gemappt werden können.

Zur optimalen Aufteilung und Optimierung der Schleifen ist deshalb ein sehr gutes Verständnis der CUDA-Architektur und der jeweils eingesetzten GPU erforderlich. Eine Einführung in die Optimierung von OpenACC-Code auf CUDA Architektur findet sich im PDF [32] von nVidia. Zur Unterstützung bieten die OpenACC-Compiler teilweise Analysewerkzeuge an. Seitens nVidia existiert der Visual Profiler, welcher mit dem CUDA-SDK ausgeliefert wird. Damit kann man visualisieren, wie die einzelnen Iterationen der Schleifen auf die Hardwarestrukturen der GPU abgebildet werden. Diese Aufteilung kann dann mit den Definitionen der Anzahl Gangs etc. gesteuert und eventuell verbessert werden.

2.6 Rahmenbedingungen Messung

Da die drei HSA-Technologien miteinander verglichen werden sollen, wird eine einheitliche Struktur des Problems (siehe Kapitel 2.2) und der Testumgebung erstellt. Zudem müssen die vorgegebenen Bedingungen in die Auswertung miteinbezogen werden. Tabelle 2.9 listet dazu die Hardware-Eigenschaften der beiden Testsysteme auf. Bis auf die Messreihen von OpenACC werden alle auf dem Windowssystem durchgeführt. Alle Programme werden im x64-Release-Modus kompiliert.

	System Windows	System Linux
Betriebssystem	Windows 7 x64 SP1	Ubuntu 12.04 LTS x64
Motherboard	ASUS Rampage II Extreme	ASUS P7H55D-M
CPU	Intel Core i7 950 3.06 GHz	Intel Core i5 660 3.33 GHz
CPU Vektoreinheit	SSE4.2	SSE4.2
GPU	ZOTAC GTX-460 AMP 1GB DDR5	ASUS EVGA GT-220 1GB DDR2
GPU Warpsize	32	32
Arbeitsspeicher	Mushkin Redline 3x2GB DDR3-1600	8GB
HDD	Samsung 2x SATA 500GB als RAID 0	SATA 160 GB

Tabelle 2.9: Auflistung der Hardware für die HSA-Tests. System Windows für OpenCL und AMP, System Linux für OpenACC.

2.6.1 Spezielle Testumstände OpenACC

Da die Testversion des CAPS-Compiler nur als Linux-Variante zur Verfügung steht, wird ein separater PC mit Ubuntu und den nötigen Tools verwendet. Weil sich diese Hardware erheblich vom Windowssystem unterscheidet (siehe Tabelle 2.9), liefert eine kleine Testreihe einen „Hardwarefaktor", mit welchem die Ergebnisse der Linux-Maschine auf die theoretischen Werte der Windows-Maschine umgerechnet werden können. Hierbei wird vor allem der grosse Leistungsunterschied der beiden GPUs korrigiert. Dazu wird neben OpenACC auch OpenCL auf dem Linuxsystem eingerichtet und der einfachste Testfall der OpenCL-Testreihe auf beiden Systemen gemessen. Aus diesen beiden Messreihen kann für jede Problemgrösse ein Hardwarefaktor berechnet werden.

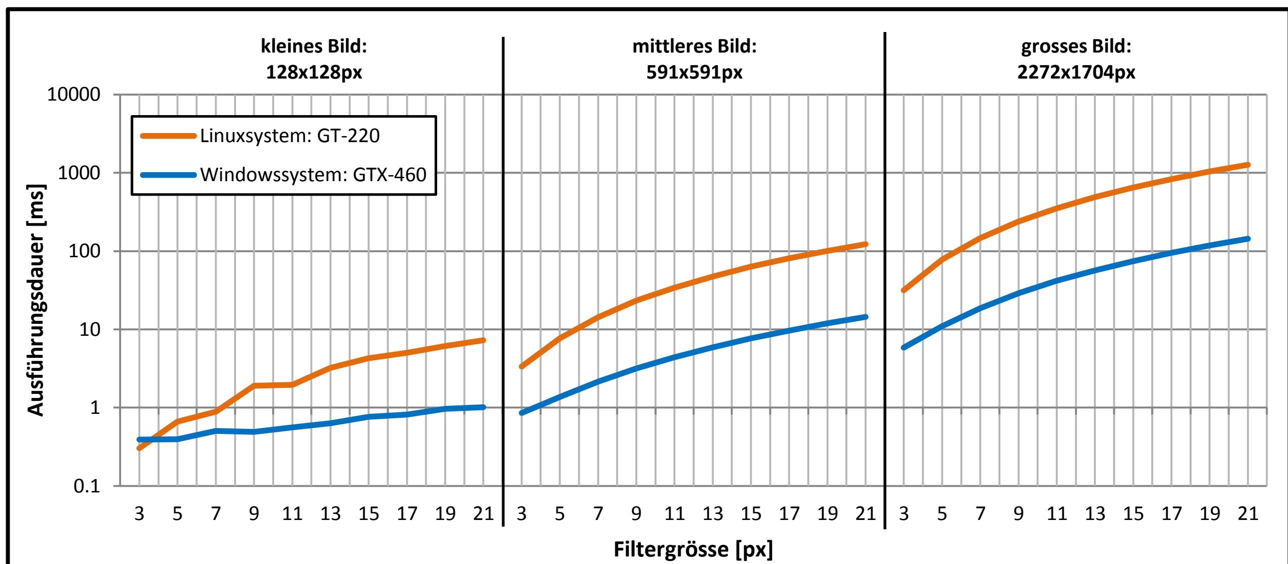


Abbildung 2.10: Ausführungsdauer für verschiedene quadratische Boxfilter auf drei unterschiedlichen Bildern. Die Problemgrösse ergibt sich aus Bildgrösse und Filtergrösse. Die Dauer bezieht sich auf die Ausführung auf der GPU des jeweiligen Systems.

Abbildung 2.10 zeigt die entsprechenden Messungen, mit logarithmischer Ordinate, welche die Ausführungszeit pro Problemgrösse darstellt. Diese ist jeweils in Form der Grösse des Eingabebildes und des Filters gegeben.

Da das Windowssystem mit besserer Hardware, insbesondere mit der neueren nVidia-GPU, ausgestattet ist, sind die Ausführungszeiten der Tests kürzer. Aus den gemessenen Werten kann für jede Problemgrösse ein

Korrekturfaktor berechnet werden, mit welchem die Resultate der OpenACC-Tests umgerechnet werden können. So ergeben sich theoretische Werte, die der Hardwarekonfiguration des Windowssystems entsprechen.

2.6.2 Durchführung der Messreihen

Die drei C++-Projekte für die Testreihen haben dieselbe Struktur und unterscheiden sich grundsätzlich nur durch die Verwendung der jeweiligen `Convolver`-Klasse. In dieser wird der HSA-spezifische Teil implementiert. Jede der darin enthaltenen `do`-Methoden stellt eine Variante der Faltungs-Implementation dar. Alle diese Methoden erhalten beim Aufruf Referenzen auf die Eingangs- und Ausgangsmatrizen, wie auch auf die drei Arrays, welche die Zeitdauer für Datenupload, Ausführung des Algorithmus und Datendownload beinhalten. Somit kann jede Implementation die Zeitmessung auf die geeignetste Weise durchführen und die Werte zurückliefern.

Als Eingangsdaten werden drei unterschiedlich grosse Bilder verwendet, die in Anhang C.C.1 abgebildet sind. Diese werden mit einem Boxfilter der Grössen 3x3, 5x5, bis 21x21 Pixel gefiltert. Zur Speicherung der Matrizen werden die Klassen `Mat` und für den Schwerpunkt des Filters die Klasse `Point` aus OpenCV verwendet. Die einzelnen Resultate werden mit den Resultaten der sequentiellen CPU-Implementation verglichen, um die Korrektheit der Algorithmen sicher zu stellen.

Um die Resultate der Messungen einheitlich und einfacher weiterverwendbar zu speichern, wird die eigens dafür geschriebene Klasse `LogWriter` verwendet. Diese ist portabel einsetzbar und ermöglicht eine Speicherung der Daten im csv-Format. Um die Verwendung des Loggers zu vereinfachen und trotzdem viel Information abspeichern zu können, werden ihm jeweils drei Arrays übergeben, welche alle Ausführungszeiten des Uploads, Algorithmus und Downloads enthalten. Diese Werte werden danach intern mit den berechneten Totalzeiten, Durchschnittswerten und den Standardabweichungen ergänzt. Des Weiteren kann beim Abschliessen einer Messung angegeben werden, ob der Testlauf ein korrektes Resultat erzielt hat. So ist in der automatisch generierten Schlusszusammenfassung des Messprotokolls sofort ersichtlich, ob es Fehler in den Implementationen gibt.

Die verschachtelte Ausführung aller Tests ist in Abbildung 2.11 dargestellt. Die Anzahl Durchläufe sowie alle Parameter der Tests sind jeweils am Anfang der Datei definiert. Die Grafik verdeutlicht zudem die Aufrufe der Loggermethoden.

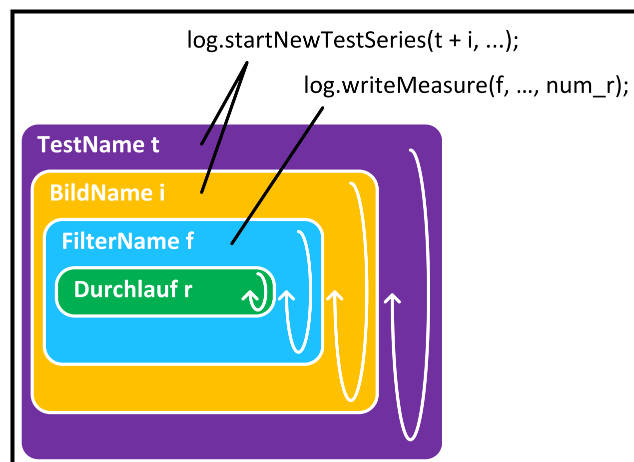


Abbildung 2.11: Iterationsverschachtelung der Testreihen und die Aufrufe der Loggermethoden.

Um gute Messresultate zu erhalten, wird jede Messung fünfmal hintereinander durchgeführt und der Mittelwert der Resultate verwendet. Zudem wird vorgängig sichergestellt, das etwaiges JIT-Compiling oder Starten der Laufzeitumgebung bereits erledigt ist. Dies wird durch einen ersten Aufruf des Tests, welcher separat als „Preparation Time“ gemessen wird, erzwungen. Die erhaltenen Ausführungszeiten werden direkt dem `LogWriter` übergeben, welcher die Statistiken berechnet und die Daten abspeichert.

2.6.3 Auswertung der Messreihen

Nach der Ausführung eines Tests existiert im definierten Resultatordner eine csv-Datei sowie alle gefilterten Resultatbilder. Die Resultatbilder werden nicht weiter betrachtet, da bereits während dem Test überprüft wird, ob das Resultat der getesteten Operation mit dem Resultat der simplen CPU-Implementation übereinstimmt. Die Daten aus der Schlusszusammenfassung werden dann in separaten Excel-Dateien verwendet, um die Grafiken zu erstellen. Dabei müssen die Grafiken mit allen drei Problemgrössen von denen mit nur zwei oder einem Problembereich unterschieden werden. Bei denjenigen mit allen Bereichen wird die Ordinate jeweils logarithmisch, bei den anderen linear dargestellt. Weiter ist zu beachten, dass die Ausführungszeiten des Up- und Downloads jeweils aufsummiert und als gesamthafter Datentransfer bezeichnet werden.

2.7 Analyse: Performance

In den folgenden Unterkapiteln wird die Leistungsfähigkeit der drei Technologien; C++ AMP, OpenCL und OpenACC zuerst einzeln betrachtet, um dadurch deren Verhalten bezogen auf die Problemgrössen und Optimierungsmassnahmen zu analysieren. Im letzten Teil werden die Systeme untereinander verglichen. Es wird dabei zwischen folgenden Messungen unterschieden.

- Laufzeit des Algorithmus
- Dauer des Datentransfers (Up- und Downloads)
- Gesamtlaufzeit

Zusätzlich werden die Messergebnisse der Ausführung des HSA-Codes auf der CPU diskutiert. Es werden jeweils nur die relevanten Messreihen[1] dargestellt. Bei Bedarf sind zusätzliche Diagramme in Anhang C ersichtlich.

2.7.1 AMP

Die AMP-Messreihen werden auf dem Windowssystem mit Visual Studio 2012 durchgeführt. Da Visual Studio 2012 und die AMP Implementation auf Windows 8 ausgerichtet sind, werden die Tests unter Windows 8 und 7 ausgeführt und verglichen. Dabei können jedoch keine signifikanten Unterschiede in Bezug auf Geschwindigkeit festgestellt werden.

Die Implementation verfügt nebst der simplen Methode (`AMPSimpleGPU`) über die Variante mit Tiling (`AMPTiledXGPU`). Letztere ist als Template implementiert, welchem man die Grösse des Tiles mitgeben kann. Diese wird auf 1x1, 2x2, 3x3, 4x4, 8x8, 16x16 und 32x32 getestet, wobei laut Theorie ab Variante 8x8 (64 Tiles) gute Resultate auf der GPU erreicht werden sollte, da die Warp-Grösse 32 beträgt und so die SMs genügend ausgelastet werden. Abbildung 2.12 zeigt alle Varianten in der Übersicht.

[1] Alle in diesem Dokument gezeigten und weitere Darstellungen finden sich in den original Excel-Dateien unter „Bericht\Resultate“.

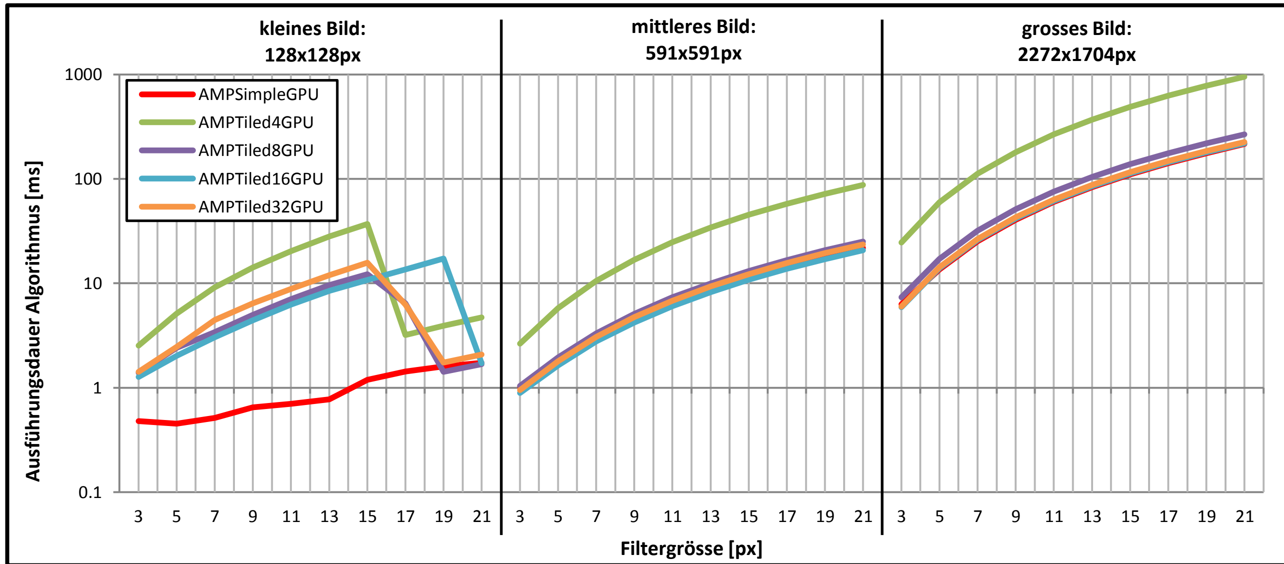
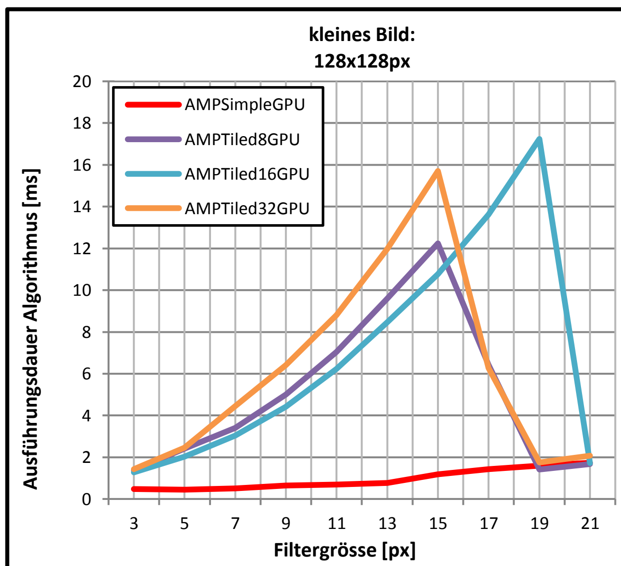
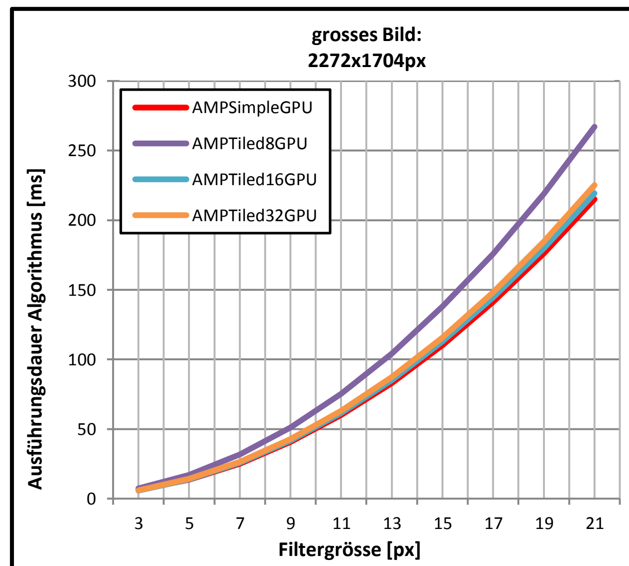


Abbildung 2.12: Ausführungsdauer des Algorithmus der AMP-Messreihen auf der GPU des Windowssystems. Gezeigt sind nur die relevanten Varianten des Algorithmus.

In Abbildung 2.12 ist ersichtlich, dass sich die Zeiten hauptsächlich im kleinen Problembereich voneinander unterscheiden. Ausgenommen sind dabei die Ausführungen mit Tile-Grösse kleiner als 8x8, welche in allen Bereichen deutlich höhere Ausführungsdauern aufweisen. Gezeigt ist nur noch die Tile-Grösse 4x4, welche unter den kleinen Tile-Grössen am besten abschneidet. Die Abbildung 2.13 stellt einerseits die Messung des kleinen Bildes, andererseits diejenige des grossen Bildes dar. Im Bild a) können sehr grosse Unterschiede betreffend Ausführungszeit ausgemacht werden. Ersichtlich ist, dass die simple Implementation die optimierten Varianten deutlich schlägt. Die Geschwindigkeitszunahme beträgt bis zu einem Faktor 13 zwischen der Variante mit Tile-Grösse 16x16 und der simplen Implementierung. Weshalb dieser grosse Unterschied entsteht, kann nicht festgestellt werden, denn die automatische Variante erzeugt 256 Threads pro Tile, was der manuellen 16x16 Variante entspricht und in den grösseren Problembereichen zu ähnlichen Werten führt. Es lässt sich zudem erkennen, dass die Anzahl von 256 Threads die GPU mit Warp-Grösse 32 scheinbar am besten auslastet und so zu den besten Messwerten führt.



a)



b)

Abbildung 2.13: Ausführungsdauer des Algorithmus der AMP-Messreihen im Detail. a) Kleines Bild. b) Grosses Bild mit. Beide mit der simplen Variante als beste Implementation.

In Abbildung 2.13 b) werden die Unterschiede zwischen den einzelnen Varianten verschwindend klein. Trotzdem ist auch hier die simple Implementation, vor derjenigen mit Tile-Grösse 16, an der Spitze. Die Leistungssteigerung beträgt allerdings maximal 2.7%.

Um die Aufwände des Datentransfers der einzelnen Varianten zu vergleichen, wird die Anzahl Durchläufe der Messungen auf 50[1] erhöht. Dies soll für zusätzliche Stabilität der Daten sorgen. Zudem werden nur die Datentransfers durchgeführt und die Faltung übersprungen. Dennoch führen diese Testreihen zu Schwierigkeiten und erzeugen bei mehrmaligem Ausführen der gesamten Reihe nicht immer gleiche Werte. Es ist deshalb schwierig, eine gültige Aussage zu machen, auch deshalb, da vor allem die simple Variante sehr unterschiedliche Messwerte erzeugt. Beispielsweise erreicht sie im kleinen Bild sporadisch bessere Werte als die manuellen Varianten. Aus Abbildung 2.14 wird ersichtlich, dass die Datentransfers beim kleinen Bild immer etwa zwischen 0.4 und 0.8ms und beim grossen Bild ca. 31ms dauern. Die mittlere Problemgrösse benötigt ca. 3.65ms.

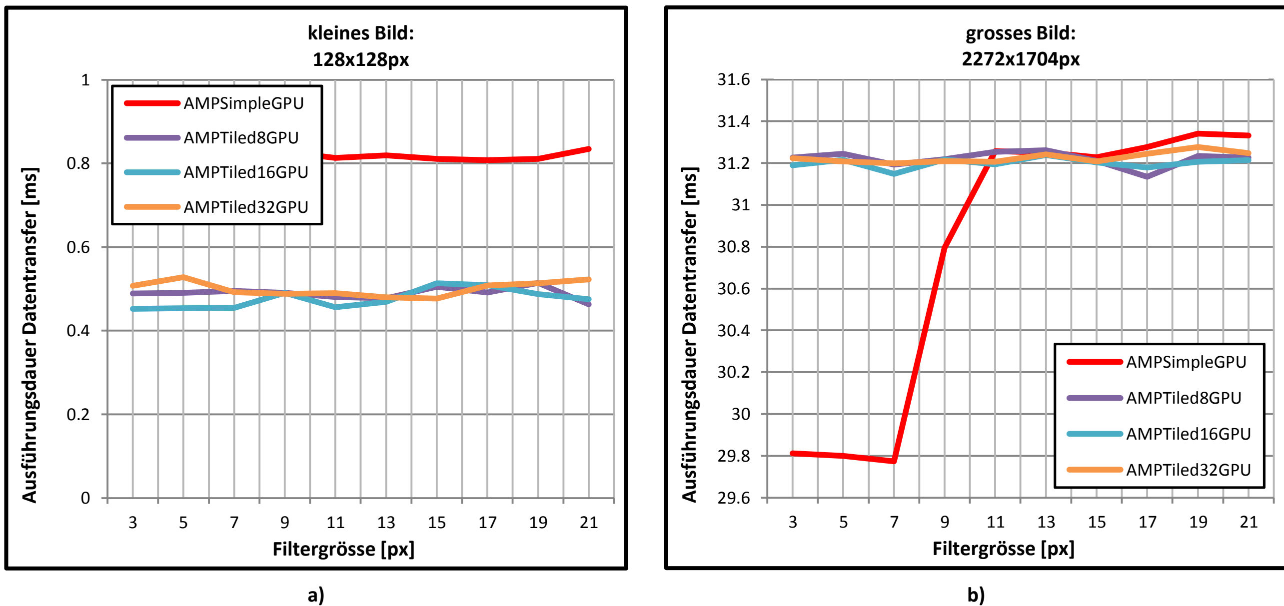


Abbildung 2.14: Aufwände für Datentransfers zur und von der GPU mit AMP. Für diese Messungen werden 50 anstatt 5 Durchläufe jeder Testreihe gemessen. a) Kleines Bild. b) Grosses Bild.

Wird die Gesamtlaufzeit der jeweiligen Ausführungen in Abbildung 2.15 analysiert, so kann erkannt werden, dass sich die simple Implementation im kleinen Problembereich klar durchsetzten kann. Hier erzielt sie um Faktor 1.3 bis 11.4 schnellere Ergebnisse. Im grossen Problembereich ist sie, gemessen an der Gesamtlaufzeit, ebenfalls besser, erzielt jedoch im Schnitt nur um den Faktor 1.014 schnellere Resultate.

[1] Diese separaten Messungen findet man unter „Bericht\Resultate“ in den Dateien „Performance_AMP_DataOnly50.xlsm“ und „Performance_AMP_DataOnly50_2.xlsm“.

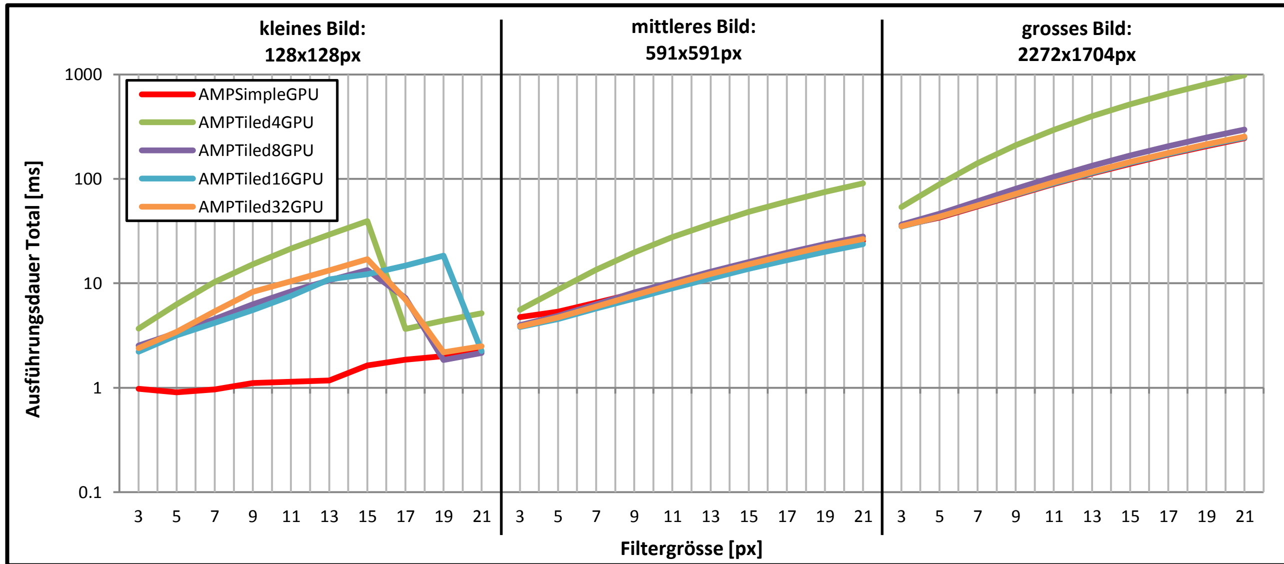


Abbildung 2.15: Gesamtlaufzeit der simplen und manuellen Varianten mit AMP auf der GPU.

Beachtenswert ist vor allem der mittlere Problembereich in Abbildung 2.16, im welchem die Variante mit Tile-Grösse 16x16 die besten Messwerte liefert. Hier reicht der Beschleunigungsfaktor von ca. 1.07 bis 1.20 und beweist, dass die automatischen Einstellungen der simplen Variante in gewissen Situationen durch Handeinstellungen übertroffen werden können.

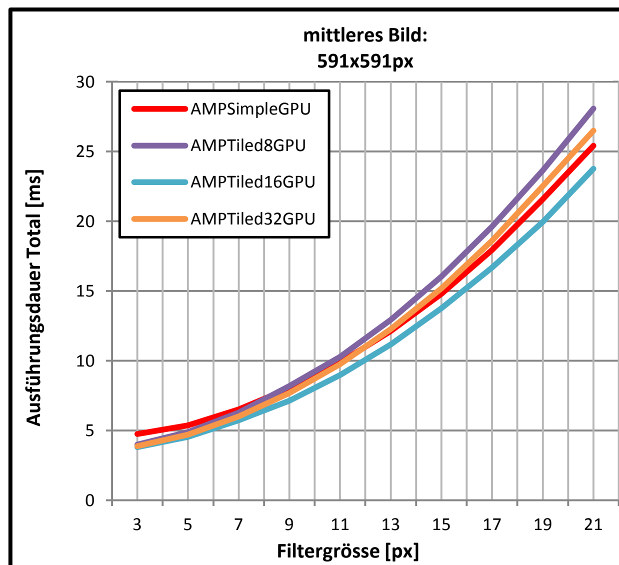


Abbildung 2.16: Gesamtlaufzeit von AMP auf der GPU. Im hier gezeigten mittleren Problembereich vermag die Variante mit Tile-Grösse 16x16 die simple Implementation zu schlagen.

Um den plattformunabhängigen Code auf einem Device das keine GPU ist zu testen, wird dieser zusätzlich auf der CPU ausgeführt und mit einer parallelen Implementation von OpenMP verglichen. Der AMP-Code wird dabei unverändert auf dem WARP-Emulator ausgeführt. Beide Varianten nutzten alle vier Kerne der CPU aus. Abbildung 2.17 zeigt die beste Variante dieser Messung.

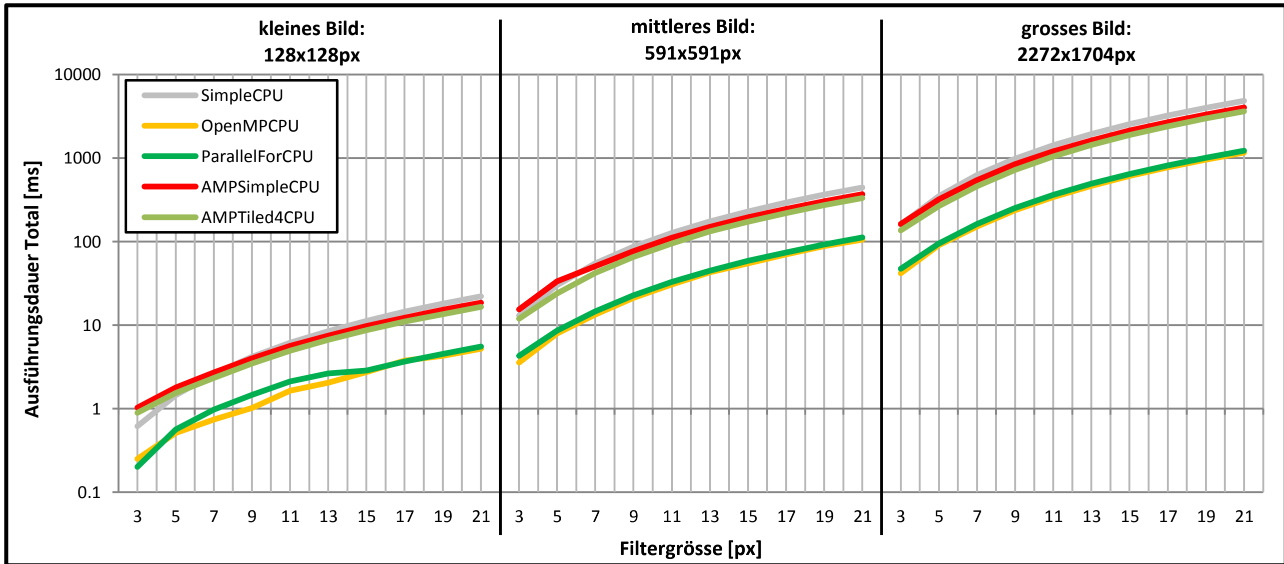


Abbildung 2.17: Gesamtlaufzeit der besten Varianten bei Ausführung der Ausführung auf CPU.

Ersichtlich wird, dass OpenMP eindeutig die besten Ausführungszeiten erreicht. Die beiden AMP-Varianten können sich nur knapp vor die primitive CPU-Implementation stellen. In Zahlen bedeutet dies einen Faktor von 3.1 bis 4.2 von OpenMP gegenüber der primitiven CPU-Implementation. Die beste der manuellen AMP-Varianten erreicht hingegen gerade mal einen durchschnittlichen Faktor von 1.26. Wichtig zu erwähnen ist, dass anders als bei den AMP-Varianten, die OpenMP-Variante keine Daten transferieren muss, was bei kleinen Problemgrössen bis zu 39.6% der Gesamtlaufzeit ausmacht. Dennoch erklärt sich die geringe Leistungssteigerung von AMP auf CPU dadurch, dass der WARP-Emulator als allgemeiner Fallback für GPU-Anwendungen gedacht ist und nicht für CPU-Parallelisierung. Der Sinn davon ist, dass Anwendungen, welche für GPUs entwickelt werden, auch auf Systemen ohne GPU ausgeführt werden können. Daher ist der WARP, wie der Name sagt, ein Emulator einer GPU, um entsprechenden Code auszuführen, welcher zwar alle Kerne einer CPU ausnutzt, diese aber hauptsächlich mit dem Overhead der Emulation belastet. Weitere Infos zu WARP finden sich in der MSDN-Bibliothek [16].

Aus diesem Grund wird zusätzlich die Variante mit der parallel_for-Funktion aus der PPL implementiert. Das macht Sinn, da Microsoft grundsätzlich diese für Parallelisierungen auf CPU vorschlägt. PPL kann mit OpenMP mithalten und erreicht im Schnitt 92.4% der Performance von OpenMP.

Zusammenfassend ist zu sagen, dass mit der simplen Variante bereits eine gute Beschleunigung erreicht werden kann, ohne sich vertieft in die Materie einzuarbeiten und den Code auf die Hardware oder die Problembeschleunigung zu spezialisieren. Jedoch lassen die Messungen vermuten, dass es beim Datentransfer durchaus Potential zur Optimierung gegenüber dem simplen Code gibt. AMP als Alternative zu üblichen Parallelisierungsmethoden auf CPU in Erwägung zu ziehen ist allerdings unsinnig. Stattdessen sollte die für solche Anwendungen gedachte PPL verwendet werden.

2.7.2 OpenCL

Die OpenCL-Messreihen werden unter Visual Studio 2010 auf dem Windowssystem durchgeführt. Dabei gibt es einerseits die simple OpenCL Implementation (`OpenCLSimpleGPU`), welche dem Framework die Wahl der Work-Group-Grösse überlässt, indem der Methode `enqueueNDRangeKernel` ein `NullRange` übergeben wird. Andererseits wird in der zweiten Variante (`OpenCLLocalWorkSizeXGPU`) jeweils eine explizite Grösse der Work-Group definiert. Die dritte Variante (`OpenCLLocalWorkSizeNoVectXGPU`)verhält sich gleich, bis auf die Tatsache, dass die automatische Vektorisierung abgeschaltet ist. Die beiden letztgenannten Varianten werden als Template implementiert und mit den folgenden Grössen getestet: 1x1, 2x2, 3x3, 4x4, 8x8, 16x16, 32x32. Wie bei AMP sollte in diesem Fall die Variante die besten Ergebnisse liefern, welche die 32 Threads im Warp der verwendeten nVidia GPU genügend auslasten kann (8x8). Abbildung 2.18 zeigt alle

Varianten in der Übersicht. Daraus wird deutlich, dass alle relevanten Implementationen sehr ähnliche Werte liefern.

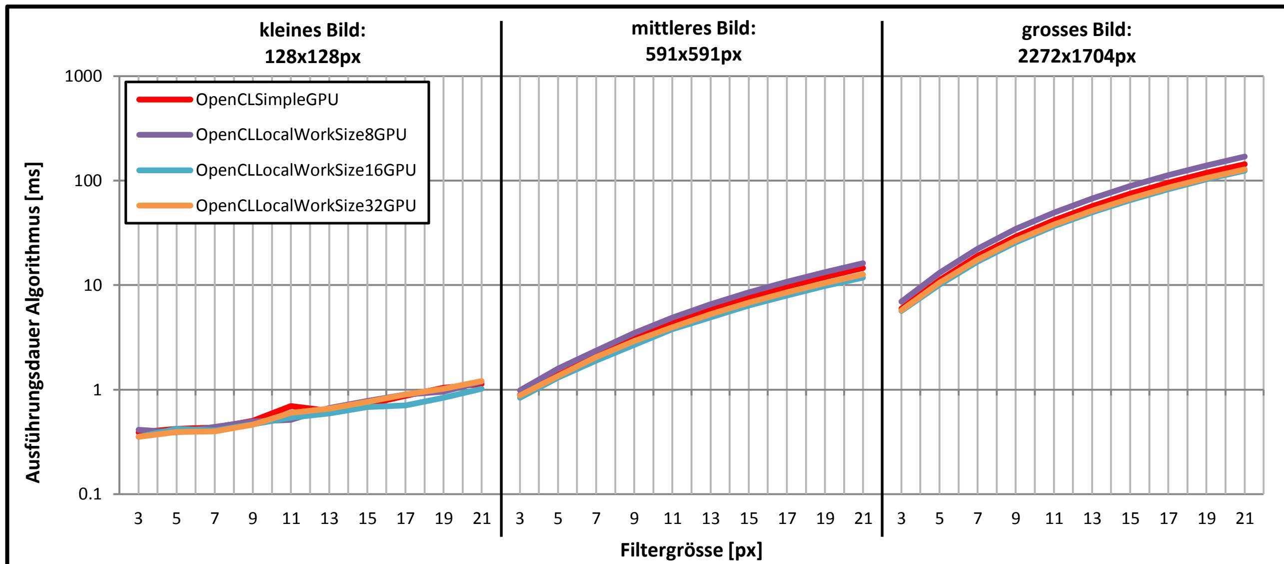
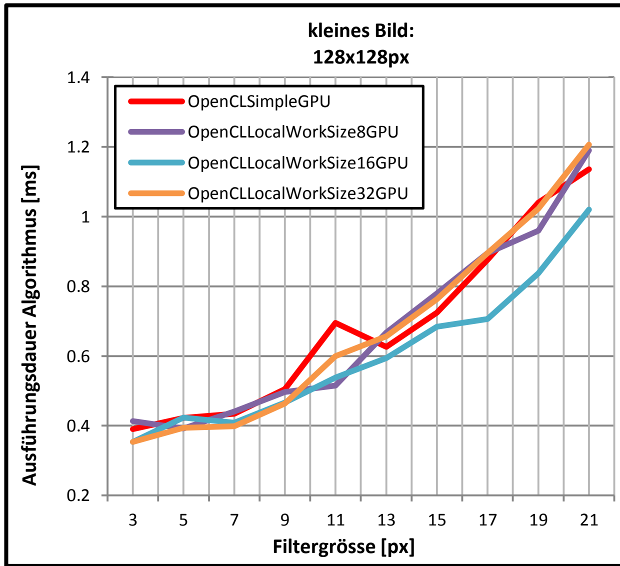


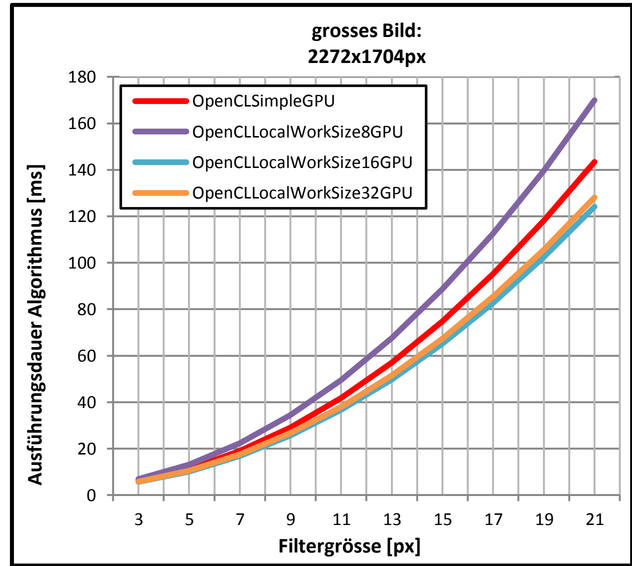
Abbildung 2.18: Ausführungsdauer des Algorithmus der OpenCL-Messreihen auf der GPU des Windowssystems. Gezeigt sind nur die GPU-relevanten Implementationen des Algorithmus.

Nicht gezeigt sind einerseits alle Varianten mit einer kleinen Work-Group-Grösse, da diese die GPU nicht optimal auslasten, andererseits diejenigen mit deaktiviertem Auto-Vektorisierer, welche bei der Ausführung auf GPU durch diese Einstellung nicht beeinflusst werden und deshalb zu denselben Ergebnissen führen.

Abbildung 2.19 zeigt die Ausführung auf der GPU im Detail. Interessant ist, dass alle Varianten grundsätzlich nahe beieinander liegen und sich, bis auf die kleine Problemgrösse, eine eindeutig beste Variante herauskristallisiert. Ähnlich wie bei AMP ist dies jedoch nicht die Version mit Work-Group-Grösse 8x8, sondern diejenige mit der Grösse 16x16. Ein deutlicher Unterschied zu AMP ist, dass die simple Variante immer von der 16er-Variante geschlagen wird. Im Schnitt beträgt die Geschwindigkeitssteigerung zur simplen Variante 1.14, im Maximum sogar 1.29.



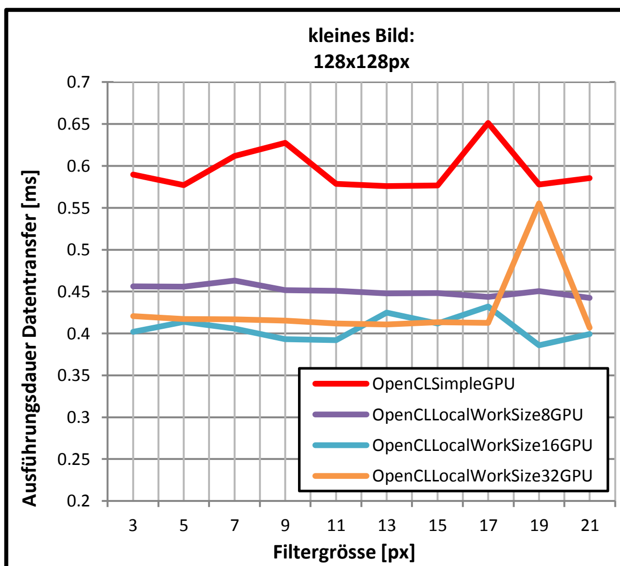
a)



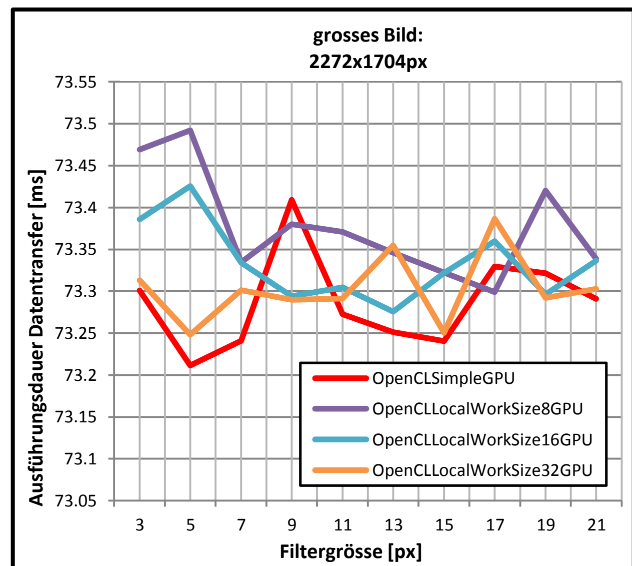
b)

Abbildung 2.19: Ausführungsdauer des Algorithmus der OpenCL-Messreihen im Detail. Gezeigt sind nur die simple und die optimierte Variante mit manueller Wahl der Work-Group-Grösse. a) Problemgrösse: kleines Bild. b) Problemgrösse: grosses Bild.

In Bezug auf den Datentransferaufwand kann ähnlich wie bei AMP nur schlecht eine Aussage gemacht werden. Es werden aber ebenfalls 50 Durchläufe für die Messungen verwendet, welche einzig eine mittlere Übertragungsdauer der einzelnen Bilder zulassen. Abbildung 2.19 zeigt, dass das kleine Bild zwischen 0.4 und 0.6ms und das grosse Bild ca. 73.3ms Transferzeit benötigt. Für die mittlere Problemgrösse werden im Mittel 3.4ms verwendet.



a)



b)

Abbildung 2.20: Ausführungsdauer des Datentransfers von und zu der GPU bei den OpenCL-Messreihen. Für diese Messungen werden jeweils 50 anstatt fünf Durchläufe jeder Testreihe gemessen. a) Kleine Problemgrösse. b) Grosse Problemgrösse.

Betrachtet man den Gesamtaufwand in Abbildung 2.21 a), erreichen alle gezeigten Varianten im Bereich der kleinen Problemgrösse ähnliche Resultate. Im grösseren Bereich b) dominiert, wenn auch knapp, die 16er-Variante vor der 32er, wobei sich diese im Schnitt um 1.4% unterscheiden. Die 16er-Variante erzielt

im grösseren Bereich durchschnittlich 7.9% und maximal 17.6% bessere Ergebnisse als die simple Implementation.

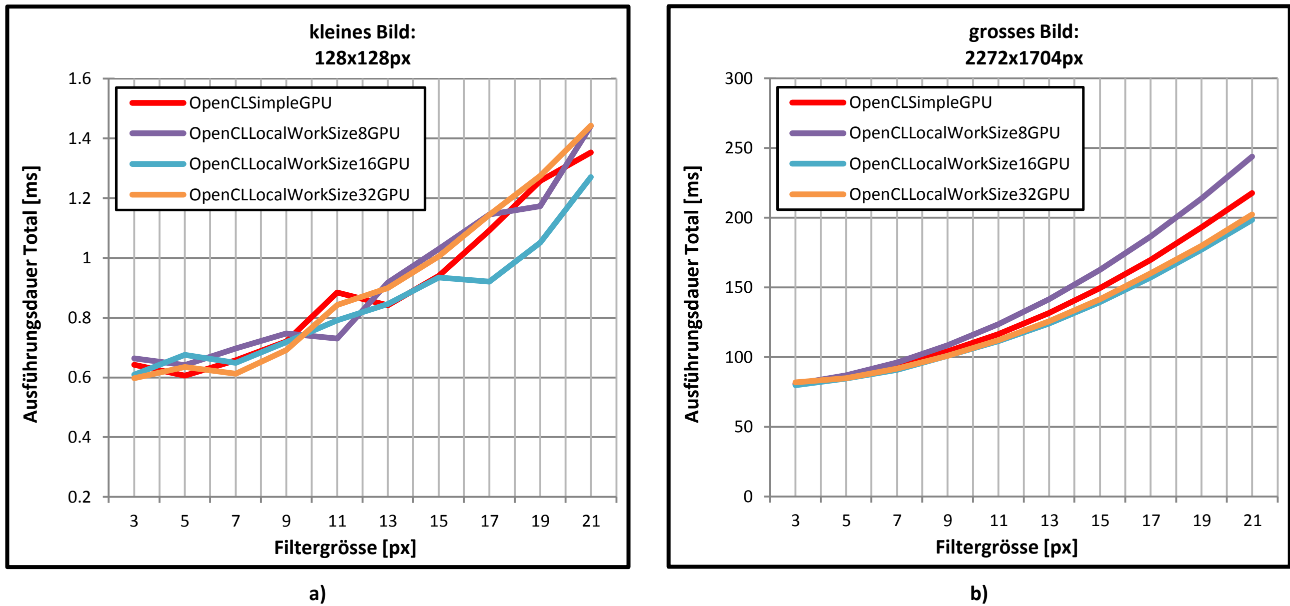


Abbildung 2.21: Gesamtausführungsdauer bei den OpenCL-Messreihen. a) Kleine Problemgrösse mit bestem Ergebnis der simplen Variante. b) Grosse Problemgrösse mit bestem Ergebnis der manuellen Variante mit Work-Group-Grösse 16x16.

Um den plattformunabhängigen Code ebenfalls auf einer CPU zu testen, wird dieser mit einer parallelen Implementation von OpenMP verglichen. Der OpenCL-Code wird unverändert auf der CPU ausgeführt, wobei alle vier Kerne der CPU ausgenutzt werden. Abbildung 2.22 zeigt die besten Varianten dieser Messung. Obwohl die einzelnen Kurven nicht genau unterschieden werden können, lässt sich eine wichtige Aussage machen: Die gezeigten OpenCL-Varianten erreichen vergleichbare Zeiten wie OpenMP.

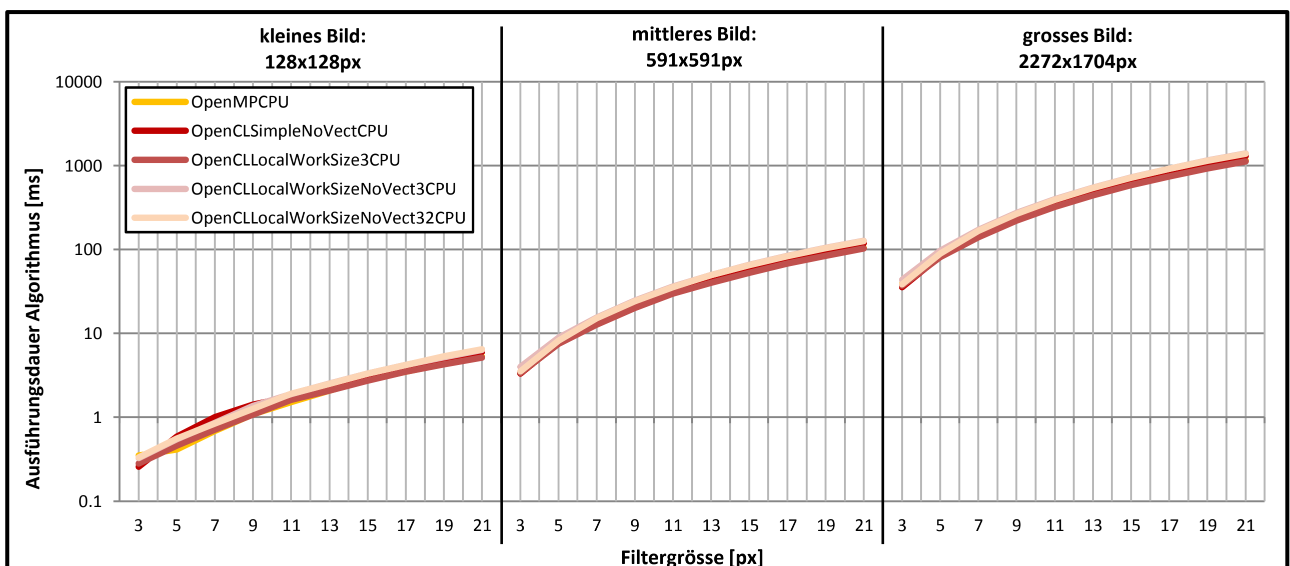


Abbildung 2.22: Algorithmuslaufzeit der besten Varianten bei Ausführung des OpenCL-Codes auf CPU.

Ähnlich wie bei der AMP-Messreihe erzielt OpenMP die besten Resultate. Trotzdem schlägt sich OpenCL deutlich besser als AMP. Abbildung 2.23 zeigt die Messung im Detail, aus welcher hervorgeht, dass die Variante mit aktiviertem Auto-Vektorisierer und einer Work-Group-Grösse von 3x3 die OpenMP-Variante

sogar übertrifft. Dies wird dadurch ermöglicht, dass OpenMP keine Art der Vektoreinheiten einer CPU ausnutzt. Betrachtet man die Ergebnisse der OpenCL-Messreihen ohne Vektoreinheit, erreichen diese nicht ganz so gute, aber doch ähnliche Werte.

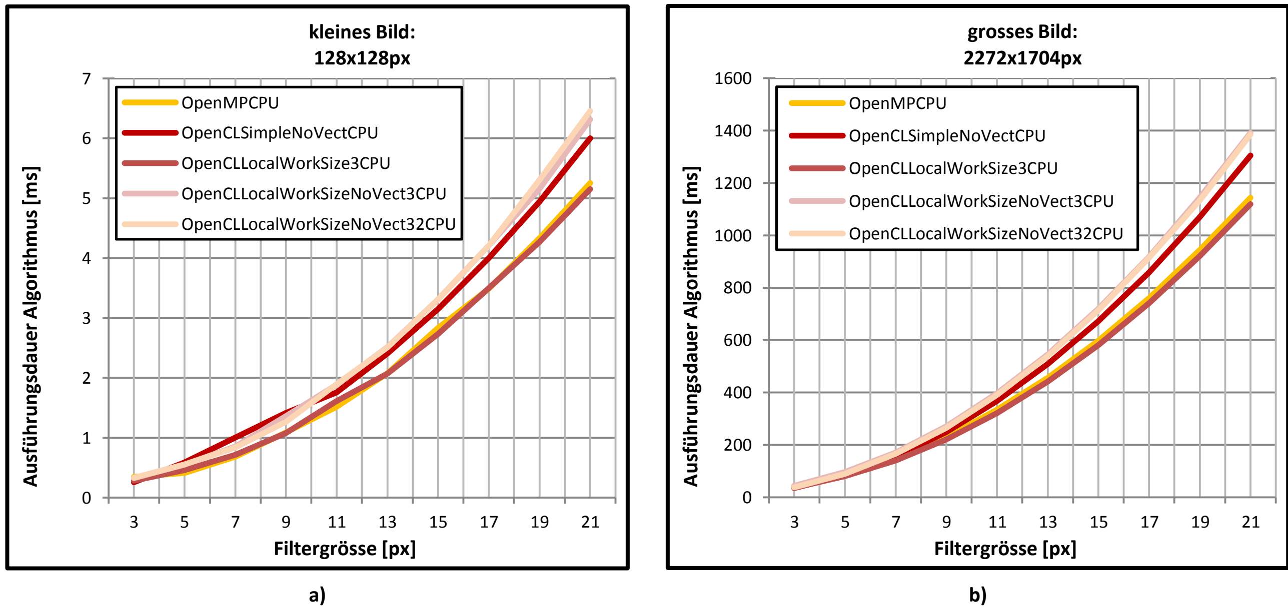


Abbildung 2.23: Ausführungszeit des Algorithmus der besten OpenCL-Varianten auf CPU.

Abbildung C.6 zeigt die OpenCL-Varianten mit aktiviertem Auto-Vektorisierer. Dabei fällt auf, dass die Variante mit Work-Group-Grösse 1x1 besonders bei kleinen Filtern schlechter ist als OpenMP. Dies lässt sich damit erklären, dass, obwohl keine Vektorisierung stattfindet, trotzdem die für den Vektorisierer notwendige Umwandlung von `uchar` zu `int` stattfindet und so überflüssiger Overhead erzeugt wird, welcher bei kleinen Laufzeiten negativ ausfällt.

Abbildung 2.24 zeigt die Gesamtlaufzeiten der besten OpenCL-Varianten. Darin lässt sich der Einfluss des Datentransfers beobachten. Obwohl die Daten im RAM liegen und von der CPU verwendet werden, müssen diese vorbereitet und kopiert werden. Folglich verschieben sich die Kurven um jeweils konstante Werte nach oben. Trotzdem erreicht OpenMP im Mittel nur 1.25 bessere Werte, wobei OpenCL im Schnitt um den Faktor 3.19 schneller ist als die simple CPU-Implementation. Um den Einfluss des Datentransfers auf CPU zu vermindern, könnte der Speicher direkt angesprochen werden, indem der Buffer mit dem Flag `MEM_ALLOC_HOST_PTR` initialisiert wird.

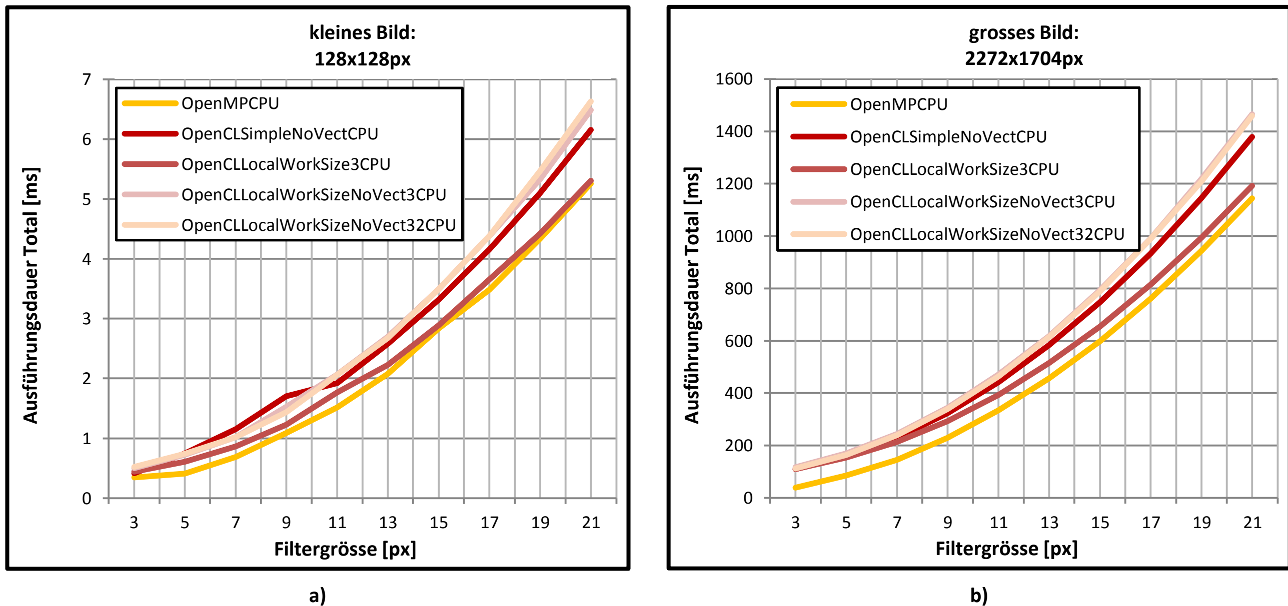


Abbildung 2.24: Gesamtausführungsdauer der besten OpenCL-Varianten auf CPU. a) Kleines Bild. b) Grosses Bild.

Zusammengefasst erscheint OpenCL als interessante HSA, welche bei kleinen Problemgrössen bereits mit der simplen Implementation zu guten Ergebnissen führt. Bei mittleren und grossen Problemen macht es jedoch Sinn, unterschiedliche Work-Group-Grössen auszuprobieren und die optimale für die entsprechende Aufgabenstellung und Hardware zu finden. Bezogen auf die Parallelisierung der CPU leistet OpenCL ebenfalls einiges und erreicht ähnliche Werte wie OpenMP, mit dem Vorteil, den Code bei vorhandener GPU noch weiter beschleunigen zu können.

2.7.3 OpenACC

Die OpenACC-Messreihen werden auf einem Linuxsystem durchgeführt. Damit die Messdaten mit denen des Windowssystems verglichen werden können, werden sie mit einem Hardwarefaktor angepasst, welcher durch die Ausführung der simplen OpenCL-Variante auf beiden Systemen ermittelt wird (siehe Kapitel 2.6.1). Es werden zwei unterschiedliche Varianten gemessen. Einerseits die simple Implementierung (`OpenACCSimpleGPU`), andererseits die von Hand optimierte Variante (`OpenACCOptGPU`).

Die simple Variante wird mit einem einfachen `#pragma acc kernels loop` parallelisiert und der Datentransfer mit den `copyin` und `copyout` Klauseln definiert. Um den Code manuell zu optimieren, werden die einzelnen Schritte aufgeteilt. Als erstes werden mit dem `#pragma data pcopyin` die Daten für die GPU vorbereitet. Die äussere For-Schleife wird dann mit `#pragma acc kernels loop` und die innere mit `#pragma acc loop` parallelisiert, wobei bei beiden die Anzahl der Gangs, Workers oder Vectors definiert werden kann. In der Messung wird die äussere Schleife mit 32 Gangs und die innere Schleife mit 16 Gangs und 32 Vectors definiert. Die Ausführung beider Varianten ist in Abbildung 2.25 aufgetragen.

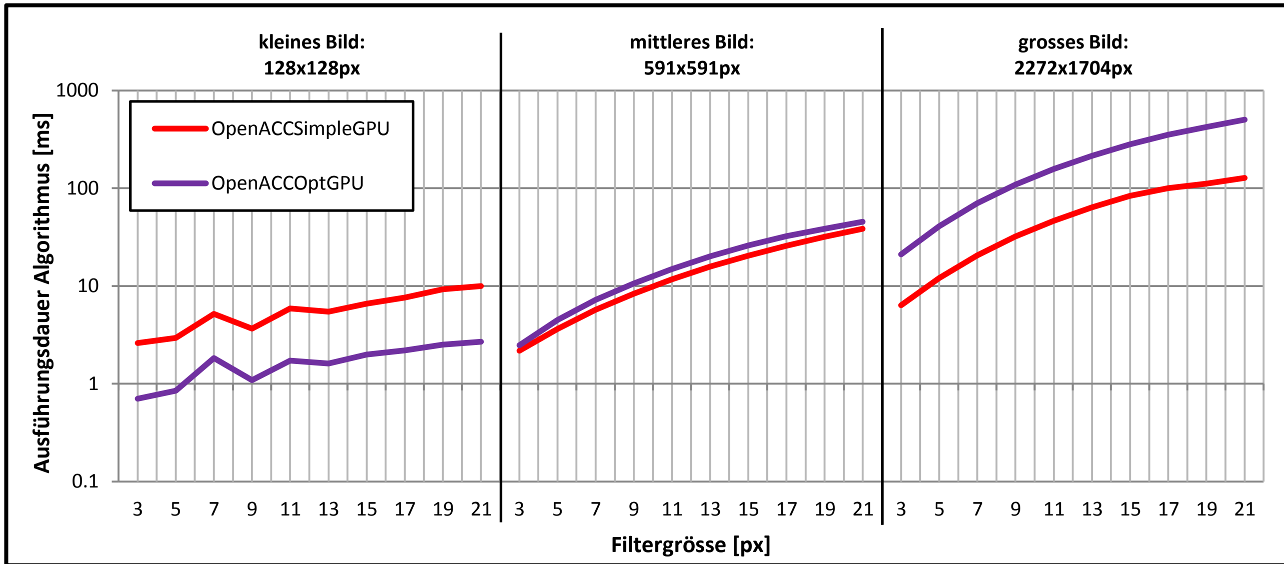


Abbildung 2.25: Ausführungsdauer des Algorithmus der OpenACC-Messreihen auf der GPU des Linuxsystems. Gezeigt sind alle durchgeführten Varianten des Algorithmus.

Es kann erkannt werden, dass bei kleinen Problemgrössen die optimierte Variante im Schnitt um den Faktor 3.43 schneller ist als die simple Implementierung. Umgekehrt schneidet die simple Variante in den grösseren Problembereichen um den Faktor 1.23 bzw. 3.49 besser ab.

Bei der Betrachtung der Gesamtzeit in Abbildung 2.26 zeigt sich kein neues Bild. Einzig die Geschwindigkeitszunahmen haben sich minimal geändert. So verringert sich der durchschnittliche Faktor im kleinen Problembereich auf 3.11, und im grossen Bereich auf 2.98. Der durchschnittliche Wert im mittleren Bereich verändert sich nur marginal.

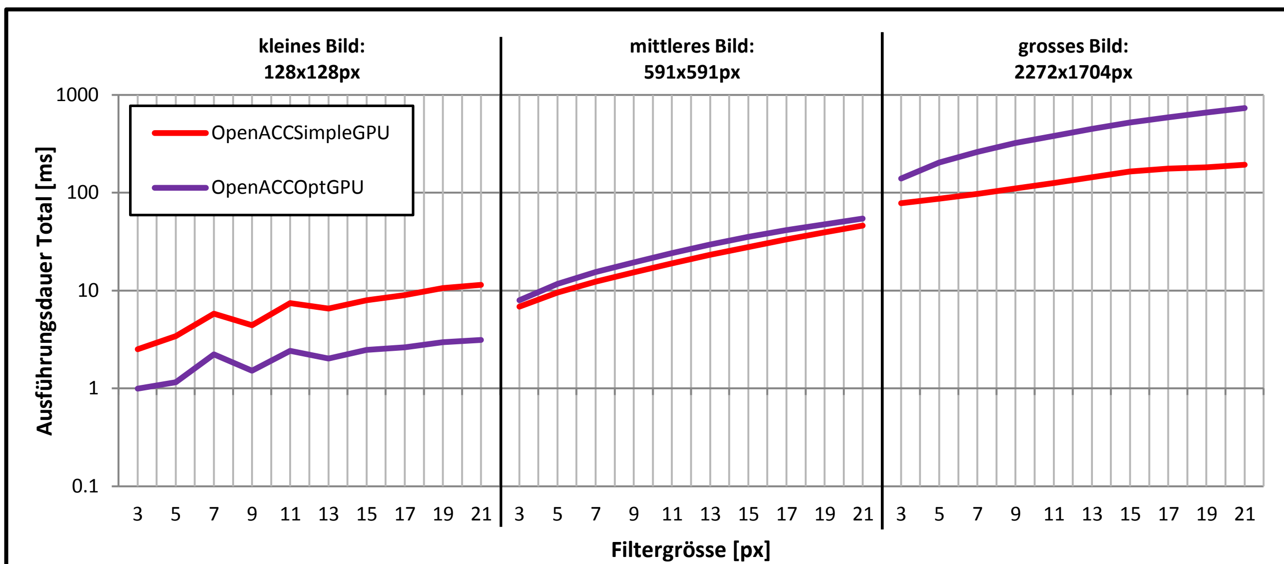


Abbildung 2.26: Logarithmische Gesamtausführungsdauer der OpenACC-Messreihen auf der GPU des Linuxsystems.

Da die erhältlichen OpenACC Implementationen CUDA-Code erzeugen, kann diese HSA nur bei vorhandener nVidia-GPU verwendet werden. Auch eine Parallelisierung auf CPU ist nicht möglich.

Zurückblickend kann nicht viel über die Performance von OpenACC ausgesagt werden. Zumindest aber ist es möglich ist, die simple Implementation durch manuelles Einstellen der Parameter zu optimieren.

2.7.4 Vergleich

In diesem Abschnitt werden alle getesteten HSAs bezüglich Performance miteinander verglichen. Dabei werden nur die simple und die am besten optimierte Variante pro Problemgrösse betrachtet. Abbildung 2.27 zeigt eine Übersicht über die relevanten Messungen bezogen auf die Ausführungsdauer des Algorithmus auf der GPU. Die simple AMP-Version liegt, ausser im kleinen Problembereich, jeweils direkt unter der Kurve der optimierten Version.

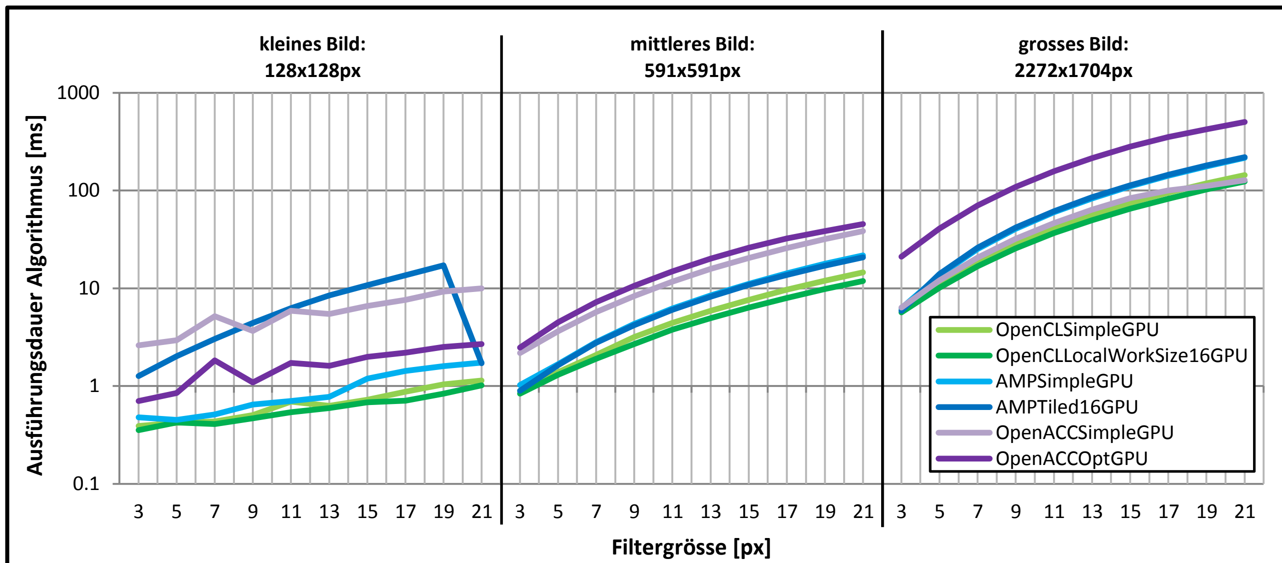


Abbildung 2.27: Ausführungsdauer des Algorithmus aller Messreihen auf der GPU. Gezeigt sind die simple und jeweils die am besten optimierte Variante des HSA-Codes.

Die OpenCL Implementationen dominieren bei allen Problemgrössen, wobei die AMP- und OpenACC-Varianten sich in der Platzierung abwechseln. Wird nur die jeweils beste Variante betrachtet, ergeben sich die Grafiken in Abbildung C.7 und Abbildung C.8. Bei kleiner Problemgrösse siegt die optimierte OpenCL-Variante, welche im Mittel um den Faktor 1.50 schneller als die simple Variante von AMP und 2.84 schneller als die optimierte OpenACC-Variante ist. Im mittleren Bereich schlägt ebenfalls die optimierte OpenCL-Variante die optimierte AMP-Implementation um den Faktor 1.55 und die simple OpenACC-Variante um den Faktor 3.08. Bei der grössten Problemgrösse erreicht die optimierte Variante von OpenCL eine Geschwindigkeitszunahme um den Faktor 1.19 gegenüber der simplen OpenACC-Implementation und einen Faktor 1.57 bezogen auf die simple Implementation von AMP. Somit übertrifft die OpenCL-Variante mit 16x16 Work-Items pro Work-Group alle anderen HSAs.

Da die Hersteller der HSAs grundsätzlich die Verwendung der simplen Varianten empfehlen, weil diese von automatischen Updates profitieren und sich selbständig für alle unterschiedlichen Devices konfigurieren, wird der durchschnittliche Leistungsunterschied über alle Problemgrössen betrachtet. Dabei schneidet die simple Variante von AMP mit dem Faktor 3.70 und OpenACC mit 1.67 besser ab, als ihre optimierten Pendants. Einzig OpenCL hat in der optimierten Implementierung im Schnitt die um den Faktor 1.09 besseren Werte als das simple Gegenstück, was jedoch kaum von Bedeutung ist.

Allgemein kann deshalb gesagt werden, dass die vorgeschlagenen simplen Varianten eine gute Wahl sind, auch wenn in Spezialfällen eine Leistungssteigerung möglich ist. Wird jedoch der Testsieger OpenCL betrachtet, kann auf diese Spezialfälle durchaus verzichtet werden, da im Vergleich zu AMP und OpenACC trotzdem eine grosse Geschwindigkeitszunahme erzielt werden kann.

Des Weiteren ist zu erwähnen, dass AMP in Bezug auf Datentransferraten bei grossen Daten eine besondere Position einnimmt. Denn wie in Abbildung 2.28 b) erkennbar ist, benötigt AMP in diesem Fall weniger als die Hälfte der Zeit der anderen HSAs für den Datentransfer. Dadurch erreicht AMP in der Gesamtlaufzeit sogar einen besseren Wert als OpenCL, was in Abbildung C.10 b) ersichtlich wird.

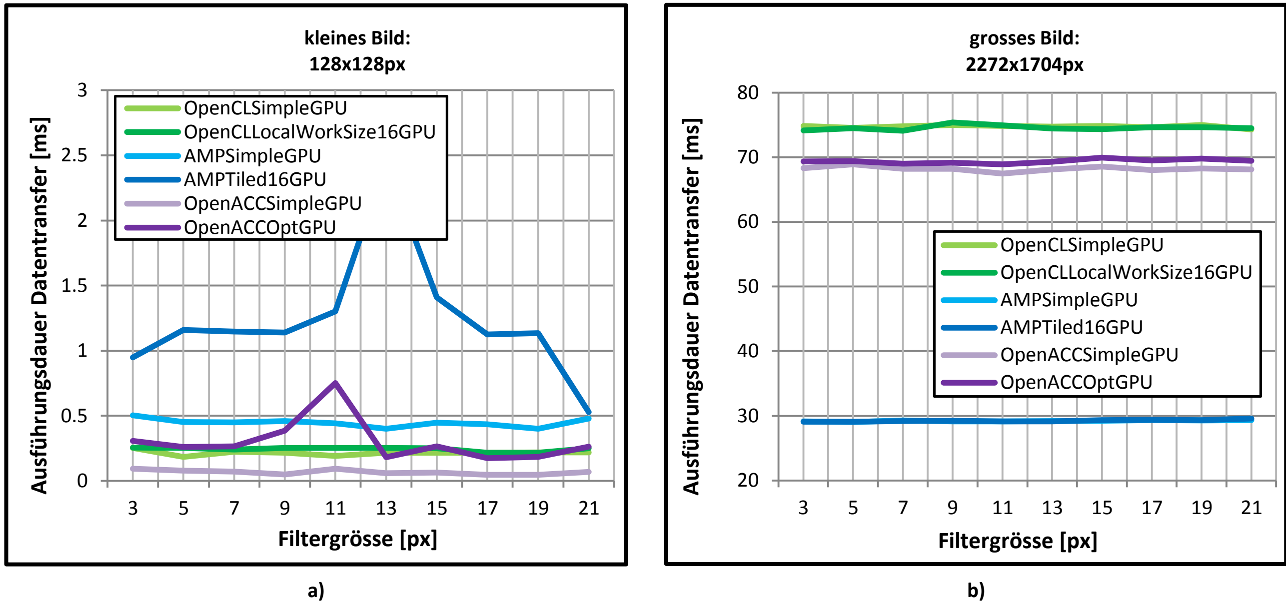


Abbildung 2.28: Ausführungsdauer des Datentransfers von und zu der GPU aller Messreihen. a) Kleine Problemgrösse mit bestem Ergebnis der simplen OpenACC-Variante. b) Grosse Problemgrösse mit bestem Ergebnis der simplen AMP-Variante.

Im Bereich Parallelisierung auf CPU werden nur AMP und OpenCL betrachtet. Abbildung 2.29 zeigt alle Varianten inkl. der primitiven CPU- und der OpenMP-Implementierung. Die optimierte OpenCL-Variante übertrifft mit Work-Group-Grösse 3x3 alle anderen HSA-Implementierungen. Bei AMP erweist sich eine Tile-Grösse von 8 als optimal, was dennoch nicht annäherungsweise zu solch guten Resultaten wie bei OpenCL führt. Deshalb wird hier die PPL-Implementierung als beste Variante der Microsoft-Sparte betrachtet. Die Kurve von OpenMP liegt jeweils unter der Kurve von PPL und der optimierten OpenCL-Variante.

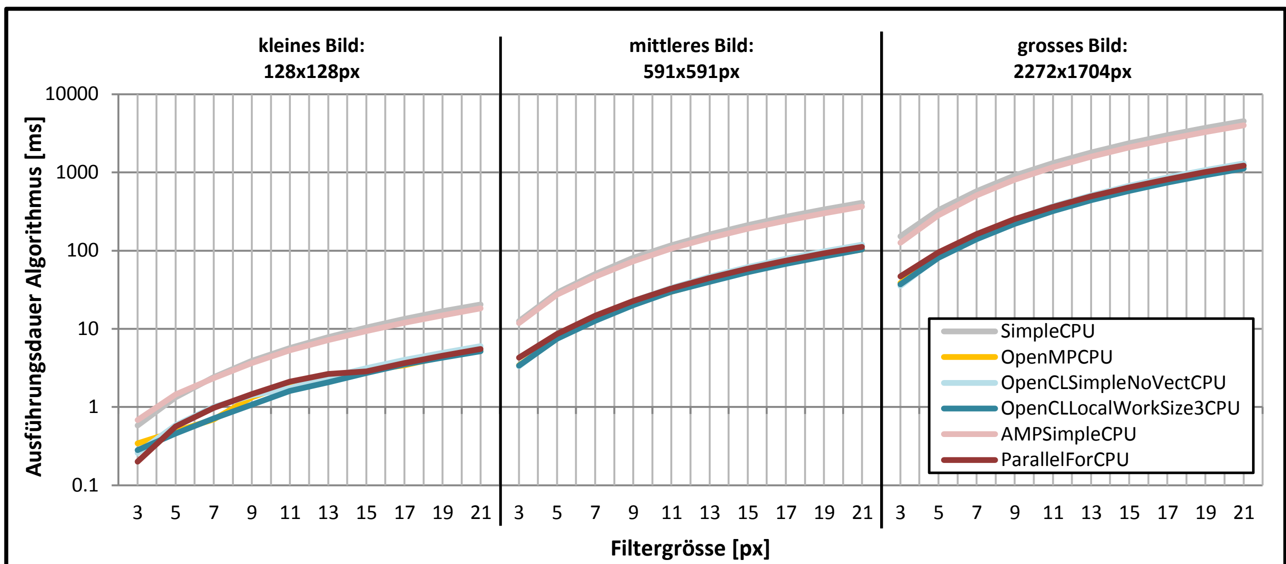


Abbildung 2.29: Ausführungsdauer des Algorithmus aller Messreihen auf der CPU. Gezeigt sind die simple und jeweils am besten optimierte Variante des HSA-Codes.

In Zahlen ist OpenCL im Schnitt 1.05mal schneller als OpenMP, dieses aber nur 1.09mal schneller als PPL. Dabei darf die Transferdauer der Daten zum Beschleuniger nicht ausser Acht gelassen werden. Denn gegenüber OpenCL benötigt OpenMP und PPL keinen Datentransfer. Die Gesamtdauer der beiden Technologien ist in Abbildung 2.30 ersichtlich.

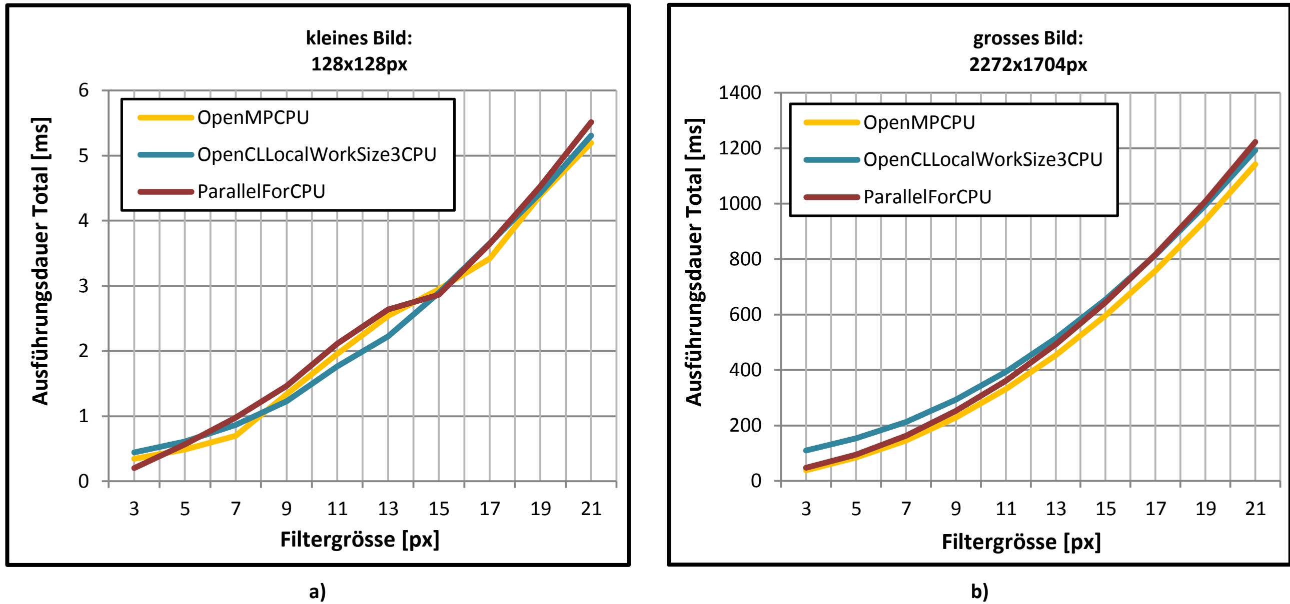


Abbildung 2.30: Gesamtausführungsdauer der besten HSAs auf der CPU.
a) Kleine Problemgrösse. b) Grosse Problemgrösse.

Unter den HSAs auf CPU kann kein eindeutiger Sieger gekürt werden. Denn obwohl AMP mit seinem WARP-Emulator deutlich von OpenCL geschlagen wird, kann die PPL-Implementation die Ehre von Microsoft retten. Diese kann OpenCL mit Auto-Vektorisierer jedoch nur übetreffen, da sie einen Vorteil beim Datentransfer hat. Allerdings ist es unter OpenCL möglich, auch in dieser Hinsicht noch Boden gut zu machen, weshalb OpenCL als die schnellste HSA betrachtet werden kann.

Soll also eine HSA-Implementation zusätzlich auf einer CPU alle Recheneinheiten optimal ausnutzten, empfiehlt sich eine optimierte Variante zu wählen. Vor allem im Fall von OpenCL kann dies bedenkenlos auf diese Weise gehandhabt werden, da ein optimaler Wert der Work-Group-Grösse häufig in der Nähe von 8 sein wird, wegen der meist 128bit-langen Vektoreinheit. Die Datentransferzeiten können jedoch zu einem Flaschenhals werden, weshalb diesen besondere Aufmerksamkeit geschenkt werden muss. Allgemein werden bei der Verwendung einer HSA jeweils zwei spezielle Implementation benötigt, welche jeweils auf CPU oder GPU optimiert sind.

In Bezug auf OpenACC bleibt abzuwarten, ob in Zukunft eine Implementation des CAPS-Compilers oder OpenMP 4.0 die Ausführung auf anderen GPUs oder CPUs unterstützen wird.

2.8 Analyse: Handhabung

Die folgenden Abschnitte befassen sich mit der Handhabung der getesteten HSA-Technologien. Darunter fallen folgende Teilbereiche:

- Installation
- Verständnissaufbau für simple Implementierung
- zusätzlicher Verständnissaufbau für optimierte Implementierung
- Code einer simplen Implementierung
- zusätzlicher Code einer optimierten Implementierung
- Handhabung unterschiedlicher Devices
- Kompatibilität zu unterschiedlichen Beschleunigern
- Vorgehen bei Messungen (Warm Up)

Diese müssen nicht zwingend für alle Technologien relevant oder existent sein. Zudem wird die allgemeine Benutzbarkeit miteinbezogen.

2.8.1 AMP

Als relativ junger Standard konzentriert sich AMP auf eine dicke Abstraktionsschicht und die Vermeidung von Problemen seiner Konkurrenten (OpenCL, CUDA, etc.). Dabei ist die fehlende oder schwache Abstraktion durch die Verwendung von C anstatt C++ eines der grössten Probleme, die AMP erfolgreich meistert.

Die Installation und Einrichtung von AMP gestaltet sich als sehr einfach, da bereits alles Notwendige in Visual Studio 2012 enthalten ist. Somit steht der Nutzung von AMP nichts mehr im Weg, vorausgesetzt es wird der neuste Compiler (vc11) mit C++11-Unterstützung verwendet. Auch ein spezieller GPU-Treiber wird nicht benötigt, da alle Beschleuniger über DirectCompute angesprochen werden. Um eine vorhandene Schleife parallelisiert auf einem Beschleuniger auszuführen, müssen die Daten entsprechend in `array`-Objekte kopiert oder durch eine `array_view` gekapselt werden. Dies ist relativ simpel, benötigt jedoch ein grundlegendes Verständnis des Speicherkonzeptes. Danach kann die for-Schleife durch ein Lambda ersetzt werden, welches der Methode `parallel_for_each` übergeben wird. Nun noch die entsprechenden Daten- und Indexobjekte innerhalb des Lambdas verwenden und fertig ist der HSA-Code. Das alles klappt innerhalb kürzester Zeit und ohne viel Literaturstudium.

Damit ein solcher Code mit Tiling und static Caching optimiert werden kann, ist wesentlich mehr Hintergrundwissen notwendig. Obwohl der Umbau auf Tiling durch „mechanisches“ Anpassen des Codes machbar ist, wird bei weitem keine optimale Geschwindigkeit erreicht. Des Weiteren muss sehr auf die Grösse der Problemdaten geachtet werden, da bei entsprechenden Tile-Grössen ein Padding oder Ähnliches zwingend notwendig ist. Um sich in die Theorie hinter AMP einzulesen empfiehlt sich das Buch von Gregory und Miller [15].

Um den Code auf unterschiedlichen Beschleunigern auszuführen, werden die Klassen `accelerator` und `accelerator_view` verwendet. Diese abstrahieren die einzelnen Devices und deren Kontext und ermöglichen eine einfache Handhabung. Dabei tauchen unterschiedliche Typen von Beschleunigern in der Liste auf:

- direct3d\ref: Softwareemulation eines Beschleunigers. Sehr langsam und nur für Debugging gedacht.
- direct3d\warp: Parallelisierter Emulator, welcher alle Cores der CPU ausnutzt. Dient als Fallback.
- cpu: Nicht verwendbar. Wird für die Datentransferierung zwischen CPU und GPU benötigt.

Alle anderen Typen von Devices sind effektive Beschleuniger, wie z.B. GPUs, und können für die Ausführung verwendet werden. Um den zu verwendenden Beschleuniger auszuwählen, kann den Datenobjekten jeweils eine `accelerator_view` mitgegeben werden, welche definiert, auf welchen Beschleuniger die

Daten transferiert und bearbeitet werden. Da AMP die Beschleuniger über DirectCompute anspricht, werden die gängigen Modelle aller Hersteller unterstützt.

Da der WARP-Emulator als echter Fallback von GPU-Code dient und durch den grossen Overhead nicht wirklich als Alternative zu echtem parallelem CPU-Code angesehen werden kann, wird von Microsoft explizit die Verwendung von PPL vorgeschlagen. Um die Handhabung von AMP und PPL zu vereinfachen, kann wie in [33, 34] ein einfacher Wrapper verwendet werden.

Für eine maximale Flexibilität bei der Hardwareunterstützung verwendet AMP ein JIT-Compiler für die GPU-Codegenerierung, weshalb darauf geachtet werden muss, dass der Code vor der Zeitmessung bereits kompiliert wurde. Dies erreicht man durch vorgängiges Ausführen des entsprechenden Codeteils. Zudem wird dadurch sichergestellt, dass die AMP-Laufzeitumgebung und die GPU vollständig initialisiert werden.

Insgesamt bietet AMP eine sehr einsteigerfreundliche Lösung in die Welt der HSAs, wobei sie trotzdem die Flexibilität und Mächtigkeit erhalten kann. Durch die hohe Abstraktion durch C++ und die Verwendung von Lambdas, die es ermöglicht, den Code dort zu platzieren, wo er ausgeführt wird, ist das Schreiben von HSA-Code sehr intuitiv. Trotz allem führt auch hier kein Weg daran vorbei, das Speichermodel und andere Low-Level-Konzepte genauer zu betrachten, wenn man den Code speziell auf die entsprechende Aufgabenstellung optimieren möchte.

2.8.2 OpenCL

Der etwas länger bekannte Standard OpenCL hat bereits diverse Verbesserungen hinter sich und befindet sich aktuell in Version 1.2, welche von allen grossen Hardwareherstellern unterstützt wird. OpenCL setzt sich als Ziel, alle möglichen Parallelrecheneinheiten eines Systems anzusprechen und den in OpenCL-C geschriebenen Code auf einer beliebigen Recheneinheiten ausführen zu können, ohne den Code anpassen zu müssen.

Die Einrichtung von OpenCL gestaltet sich relativ einfach. Einerseits benötigt jeder anzusprechende Beschleuniger im System den entsprechenden OpenCL-Treiber, wobei der in jedem Treiber integrierte ICD-Loader die Verwaltung der unterschiedlichen Geräte übernimmt. Andererseits muss im jeweiligen Projekt auf die Include- und Library-Verzeichnisse verwiesen und die API-Funktionen verwendet werden. Zusätzlich können IDE-Erweiterungen z.B. für Visual Studio installiert werden, um das Arbeiten mit OpenCL-C zu vereinfachen. Ebenfalls kann die C++-Header-Datei eingebunden werden, wodurch das komplette API über C++-Klassen angesprochen werden kann.

Code, der beschleunigt werden soll, muss als Kernel in OpenCL-C geschrieben werden und wird meistens als separate Datei mit der Endung `.cl` abgespeichert. Dabei muss vor allem auf die korrekte Annotation des Speicherbereichs der Daten geachtet werden, was mit den Schlüsselwörtern `__global`, `__constant`, `__local` und `__private` erledigt wird. Zudem werden Funktionen wie `get_global_id` verwendet, um die aktuell zu berechnende Position im Speicher zu ermitteln. Um den geschriebenen Kernel auszuführen, werden einige Vorkehrungen benötigt: Das Einlesen und Kompilieren des Kernel-Codes, die Vorbereitung der Daten und die Übergabe der Kernelparameter. Danach kann der Kernel mittels `enqueueNDRangeKernel` ausgeführt werden. Zuvor muss jedoch der entsprechende Kontext für die Ausführung erstellt werden. Für diesen wählt man als erstes die gewünschte Plattform (z.B. nVidia) und darin den entsprechende Beschleuniger (z.B. mit Attribut `CL_DEVICE_TYPE_GPU`). Danach kann ein Kontext erstellt werden, in welchem der Kernel kompiliert, eine Ausführungswarteschlange eingerichtet und der Code ausgeführt wird.

Um den Kernel mithilfe einer manuell definierten Work-Group-Grösse zu optimieren, muss ein zusätzlicher Parameter beim Aufruf der `enqueueNDRangeKernel` Funktion angegeben werden. Zusätzlich wird dann eine Abfrage im Kernel nötig, welche überprüft, ob sich der aktuelle Index im definierten Speicherbereich befindet. Das notwendige Padding der Daten, welches die erwähnte Prüfung nötig macht, muss bereits bei der Angabe des `globalRange` einberechnet werden.

Der Auto-Vektorisierer, welcher bei CPU-Ausführung aktiv wird, sobald eine Work-Group-Grösse von mehr als 1 gewählt wird, kann unter Umständen zu unnötigem Overhead führen. Deshalb kann dieser mir einem

Kompiler-Hinweis im Kernel (`__attribute__((vec_type_hint(float3)))`) auf einen bestimmten Typ, hier `float3`, eingestellt oder ganz deaktiviert werden, wenn der angegeben Typ gar nie verwendet wird.

Die Ausführung des Codes auf unterschiedlichen Devices wird durch das Erstellen unterschiedlicher Kontexte bewältigt. Der Kernel wird in diesen Kontexten kompiliert und ausgeführt. Da die Kompilation explizit aufgerufen wird, kann diese bereits bei der Initialisierung stattfinden und so vom Aufwand des effektiven Algorithmus getrennt werden.

OpenCL benötigt eine gewisse Einarbeitungszeit, bis die eigene Sprache, die Speicherplatzkennzeichner und die Einrichtung des Host-Codes verstanden sind und funktionieren. Dies führt zu einer steilen Lernkurve am Anfang, welche danach abflacht. Trotzdem hat man danach nicht ausgelernt und ein vertieftes Verständnis für das Speicherkonzept ist notwendig, um den Code optimieren zu können. Allerdings erscheint OpenCL sehr transparent, da alle Vorgänge explizit ausgelöst werden. Dadurch ergeben sich vor allem bei der Auslagerung von Initialisierungsvorgängen mehr Möglichkeiten.

2.8.3 OpenACC

Der als Vorgänger des OpenMP 4.0 Standards betrachtete OpenACC Standard wird momentan nur von wenigen Herstellern implementiert und unterstützt im herkömmlichen PC-Bereich nur CUDA-GPUs von nVidia. Sein hohes Ziel, Code für GPUs oder andere Beschleuniger durch simple Annotationen zu parallelisieren und nutzbar zu machen, scheint im Moment noch fern. Trotzdem vermag er eine Vorschau zu liefern, wie die nächste Generation von HSAs aussehen könnte und wie mit einfachen Annotationen Code für GPUs nutzbar gemacht wird.

PGI stellt dabei ein Paket aus Compiler und Kommandozeilenumgebung für Windows zur Verfügung. Nach der simplen Installation folgt ein komplizierter Registrationsprozess. Auch nach diversen Versuchen mit unterschiedlichen Lizenzen, Softwareversionen und Testprogrammen kann kein lauffähiger Code erzeugt werden. Ob der Compiler wegen Lizenzproblemen nicht funktioniert oder ob die Installation und Einrichtung der Software nicht korrekt geklappt hat, ist ungewiss.

Um den CAPS Compiler unter Ubuntu einzurichten, sind diverse Vorbereitungen nötigt. Neben dem Installieren des korrekten CUDA-Treibers, des C++-Kompilers und anderer Tools, müssen auch mehrere Umgebungsvariablen im System und dem Terminal eingerichtet werden. Leider kann dabei sehr viel schief oder vergessen gehen, weshalb sehr schnell viel Zeit für die Installation der OpenACC-Entwicklungsumgebung verloren geht, zumal es sich nicht um eine gewohnte IDE handelt, da keinerlei Syntax- oder andere Unterstützung geliefert wird. Zudem muss der gesamte Build-Prozess eigenhändig mit einem `Makefile` eingerichtet werden.

Die Anpassung bestehenden Codes an OpenACC gestaltet sich als sehr einfach. Durch das Kapseln der äussersten For-Schleife mit `#pragma acc kernels loop` wird der Code analysiert und automatisch für ein CUDA-Gerät vorbereitet. Einzig die Definition der Kopiervorgänge setzt ein Minimum an Theoriekenntnis für die Technologie voraus. Um jedoch manuelle Optimierungen vorzunehmen, wird ein vertieftes Verständnis des Speichermodells und der CUDA-Infrastruktur benötigt. Der Aufwand um den Code anzupassen, bleibt aber gering, da auch in diesem Fall nur wenige Annotationen benötigt werden.

Da der CAPS-Kompiler direkt CUDA-Code erzeugt, kann er nicht als echte HSA betrachtet werden, da eine Parallelisierung auf einer CPU so nicht möglich ist. Auch die Verwaltung von unterschiedlicher GPUs in einem System lässt sich nur als Kompilerparameter bewältigen, was die Funktionalität stark einschränkt. In Bezug auf OpenMP 4.0 ist eine echte HSA-Funktionalität erkennbar, da der alte OpenMP-Teil die Parallelisierung auf CPUs und der OpenACC-Teil diese auf GPUs übernehmen wird.

Die Beschleunigung simpler Schleifen gestaltet sich unter OpenACC als sehr einfach. Trotzdem steigt die Komplexität schnell an, wenn manuelle Optimierungen gemacht werden sollen. Zusätzlich können Probleme wegen Inkompatibilitäten auftauchen, wenn komplexere Code-Fragmente parallelisiert werden sollen und/oder bereits andere Libraries verwendet werden.

2.8.4 Vergleich

In diesem Abschnitt werden alle getesteten HSAs bezüglich Handhabung miteinander verglichen. Dabei unterscheidet sich vor allem der benötigte Lernaufwand, welcher für die gewünschte Problemlösung erbracht werden muss. Abbildung 2.31 zeigt eine Darstellung der gefühlten Lernkurve. AMP zeigt darin die am ehesten lineare Steigung, was dieses zu einer sehr einsteigerfreundlichen HSA macht. OpenCL stellt für den Benutzer die steilste Hürde dar, bei welchem es sich allerdings um ein sehr transparentes und modifizierbares System handelt. OpenACC hingegen bietet sich für kurze Testversuche oder simple Probleme an, welche auf GPU parallelisiert werden sollen. Für den realen Einsatz mit starker Optimierung in grossen Systemen ist es nicht geeignet.

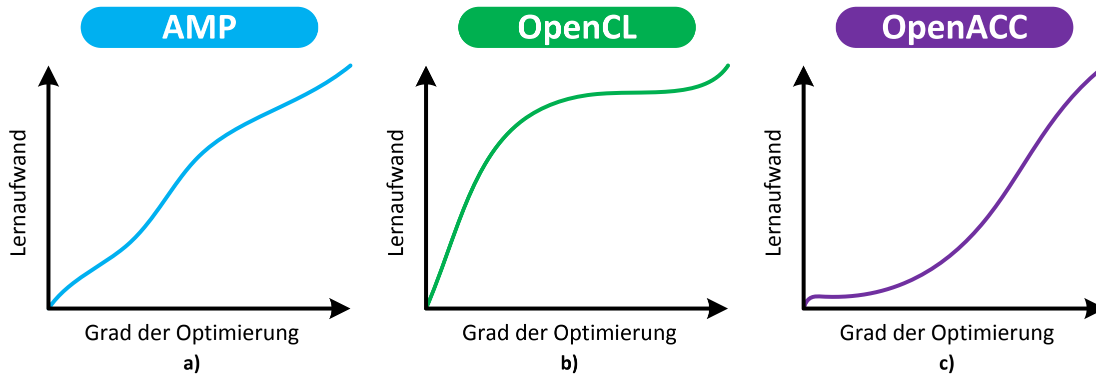


Abbildung 2.31: Vergleich der getesteten HSAs in Bezug auf den benötigten Lernaufwand. a) AMP verlangt einen regelmässigen Wissenszuwachs. b) OpenCL benötigt setzt vor dem Einstieg viel Theoriestudium voraus. c) OpenACC wird bei hohen Anforderungen lernintensiv.

Des Weiteren ist die Installation ein entscheidender Faktor für die Brauchbarkeit einer Technologie. Hier schneiden AMP und OpenCL sehr gut ab, da sehr wenige Handgriffe notwendig sind, um das System einzurichten. OpenACC kann dagegen nicht überzeugen, da sich die Einrichtung als mühsam erweist, es nicht in eine vollwertige IDE eingebettet ist und wenige spezialisierte Implementationen vorhanden sind, welche zudem zusätzliche Kosten verursachen. Weiter ist OpenACC nur für CUDA-Beschleuniger einsetzbar, wogegen AMP und OpenCL alle Typen von GPUs als auch CPUs unterstützen und somit echte HSAs darstellen.

Bei Betrachtung der Codelänge sticht OpenACC klar hervor, da es weniger als fünf Zeilen Code in Form von Pragmas benötigt, um eine verschachtelte for-Schleife auf der GPU auszuführen. Auch jegliche Art von Padding oder Umgang mit mehrdimensionalen Indizes bleiben dem Benutzer erspart. Hingegen werden solche Paradigmen bei AMP und OpenCL zwingend benötigt, was die Komplexität und Fehleranfälligkeit erhöht. Zudem wird die Verwaltung von Host und Device nicht automatisch, wie bei OpenACC, erledigt, was allerdings die hohe Anpassbarkeit dieser Systeme ausmacht.

Eine Zusammenfassung aller Aussagen zur Handhabung ist in Tabelle 2.10 ersichtlich.

	AMP	OpenCL	OpenACC (CAPS Compiler)
Installation	Einfach	Einfach	Mittel
Verständnisaufbau für simple Implementierung	Kurz	Mittel	Kurz
Zusätzlicher Verständnisaufbau für optimierte Implementierung	Mittel	Kurz	Lang
Code einer simplen Implementierung	Mittel	Lang	Kurz
Zusätzlicher Code einer optimierten Implementierung	Mittel	Kurz	Kurz
Handhabung unterschiedlicher Devices	Einfach	Einfach	Nicht möglich
Kompatibilität zu unterschiedlichen Beschleunigern	nVidia, AMD, Intel	nVidia, AMD, Intel	nVidia
Vorgehen bei Messungen (Warm Up)	JIT-Compiling, Runtime-Init	JIT-Compiling, Runtime-Init	`#pragma`-Aufsplittung

Tabelle 2.10: Vergleich der getesteten HSAs in Bezug auf die Handhabung.

2.9 Empfehlung

Die Messungen und Erfahrungen, welche mit den drei HSAs gemacht wurden, lassen unterschiedliche Aussagen zu. Klar ist, dass OpenACC noch in den Kinderschuhen steckt, was die Kompatibilität und Performance angeht. In diesem Bereich gibt es viel Nachholbedarf, wobei abzuwarten ist, ob sich die Zahl der Anbieter von OpenACC-Implementationen und somit auch deren Kompatibilität zu unterschiedlicher Hardware erweitern werden.

Bei der Betrachtung von AMP und OpenCL ist die Entscheidung schwieriger. Beide Systeme haben ihre Vor- und Nachteile, gesamthaft schlägt sich OpenCL aber besser. Dies primär wegen der Überlegenheit bezüglich Performance, im Bereich GPU und CPU, gegenüber den anderen HSAs, aber auch durch die hohe Anpassbarkeit und Transparenz. Was einem die Suppe ein wenig versalzen mag, ist die steile Lernkurve bei der Einarbeitung und die aufwendige Initialisierung durch den Host. Diese beiden Merkmale schneiden bei AMP deutlich besser ab, was dem Benutzer den Einstieg wiederum sehr erleichtert, ihn aber trotzdem zu vertieftem Eintauchen ermutigt. Zudem gelingt die Handhabung von Kernel-Code deutlich einfacher durch die Verwendung von Lambdas und das unterstützende Framework. Um die Performancefrage zu entschärfen, kann im CPU-Bereich die Microsoft PPL verwendet werden, welche sich ähnlich wie AMP handhaben lässt.

Abschliessend lässt sich sagen, dass sich OpenCL empfiehlt, sobald Wert auf Performance gelegt wird oder komplexere Problemstellungen gelöst werden müssen. AMP ist hingegen für die Einführung in die Thematik von HSAs im schulischen Umfeld sehr geeignet oder bei sehr kleinen Problemen, die nicht optimal beschleunigt werden müssen.

3 Kopfdetektion

Das Ziel, Personen in einem Video-Livestream zu detektieren und wiederzuerkennen, benötigt eine Möglichkeit zu ermitteln, wo im Bild sich eine Person und deren Kopf befinden. Der Kopf und vor allem das Gesicht beinhalten die zur Wiedererkennung notwendigen Informationen Für die Generierung eines Datenmodells werden diese für jede erkannte Person benötigt.

Der im Projekt 7 entwickelte Bewegungsdetektor erledigt dabei die Einschränkung des Bildes in Bereiche, in welchen sich bewegende Personen befinden. Um die Köpfe in diesen Bereichen zu finden, wird ein System benötigt, welches diese erkennen kann. Dazu wird ein so genannter „Cascade Classifier“ verwendet, welcher speziell darauf trainiert ist, Köpfe zu detektieren.

3.1 Machine Learning

Ein grosser und bedeutender Bereich der Computerwissenschaften ist Machine Learning. Diese Disziplin beschäftigt sich mit der Frage, ob und wie eine Maschine selbständig lernen kann. Um diese Frage zu klären, werden Systeme entwickelt, welche Muster aus Daten identifizieren und die dadurch gewonnenen Erkenntnisse dazu verwenden, ein allgemeines Modell der Daten zu erstellen. Diese Generalisierung der „erlebten“ Muster ist ein zentraler Bestandteil solcher Systeme und beeinflusst deren Qualität erheblich.

Typischerweise wird der Vorgang des Trainierens, welcher den Lernprozess beschreibt, durch die Analyse von Daten definiert, welche die gewünschte Eigenschaft enthalten (Positive Samples) oder nicht enthalten (Negative Samples). In diesem Fall lernt die Maschine durch die annotierten Daten, welcher Teil davon zu dem gewünschten Modell gehört und welcher nicht. Die Information wird später verwendet, um unbekannte Samples zu klassifizieren. Diese Art des Lernens wird als „Supervised Learning“ bezeichnet und ist Bestandteil dieses Kapitels. Das Pendant dazu ist das „Unsupervised Learning“, welches keinerlei Begleitinformation neben den Daten benötigt, um allfällige Muster zu erkennen. Diese Art des Machine Learnings siedelt sich eher im Bereich „Data Mining“ an.

In den folgenden Abschnitten werden die gebräuchlichsten Theorien und Technologien zur Mustererkennung in der Bildverarbeitung vorgestellt und einige davon im Unterkapitel 3.2 implementiert.

3.1.1 Classifier

Ein Hauptmerkmal von Intelligenz ist die Fähigkeit, verschiedene Dinge zu unterscheiden und zu klassifizieren. Diese Art von künstlicher Intelligenz wird durch Supervised-Learning-Algorithmen implementiert. Dabei spielt es keine Rolle, ob der Klassifizierer (Classifier) die Objekte in zwei oder mehr Klassen einordnen kann. Die Variante des binären Classifiers wird vor allem in der Detektion bestimmter Objekttypen verwendet, wobei ein Multiklassen Classifier z.B. für die Zuordnung der Namen zu einem Gesicht verwendet werden kann. Wenn ein Classifier einen kontinuierlichen Wert, wie das Alter einer Person zurückgeben soll, spricht man von regressiven Classifiern.

Ein Classifier kann auf verschiedene Arten implementiert werden: Decision Trees, Support Vector Machines, Neural Networks, etc. Allen gemeinsam ist der Ablauf deren Verwendung. Die erste Phase beinhaltet das Trainieren des Classifiers mit entsprechenden Samples. Diese Trainingsphase kann sehr unterschiedlich ausfallen und ist implementationsabhängig. Nach Abschluss dieser Phase existiert ein einsatzbereiter Classifier, welcher aber nur genau für den trainierten Zweck eingesetzt werden kann. Bei gewissen Arten von Classifier-Implementationen ist es zudem möglich, den Classifier später mit neuen Samples weiter zu trainieren und dessen Genauigkeit dadurch zu verbessern.

Die zweite Phase der Verwendung ist der Einsatz des Classifiers im jeweiligen Anwendungsbereich. In dieser Phase wird dem Classifier jeweils ein neues, unbekanntes Sample eingegeben, worauf dieser dessen Klasse als Resultat liefert. Ob diese Klasse korrekt ist oder nicht, kann nur durch eine höhere Intelligenz, meist durch einen Menschen, ermittelt werden. Damit der Anwender dennoch die Zuverlässigkeit des Classifiers beurteilen kann, wird dieser nach dem Trainieren üblicherweise mit zusätzlichen annotierten Testdaten

ausführlich getestet. Dabei lassen sich statistische Werte ermitteln, welche in Tabelle 3.1 ausgelistet sind und mit welchen die Qualität eines Classifiers genauer beschrieben werden kann. Eine Übersicht über alle Beurteilungswerte bietet Wikipedia [35].

Name	Synonym	Beschreibung	Formel
True Positive (TP)	Hit	Der Test erkennt eine korrekte Zugehörigkeit zur getesteten Klasse.	
False Positive (FP)	False Alarm, Typ 1 Fehler	Der Test erkennt eine Zugehörigkeit, welche aber falsch ist.	
True Negative (TN)	Correct Rejection	Der Test erkennt korrekt, dass das getestete Sample nicht in die entsprechende Klasse gehört.	
False Negative (FN)	Miss, Typ 2 Fehler	Der Test erkennt keine Zugehörigkeit zur Klasse, obwohl diese vorhanden ist.	
Positive Predictive Value (PPV)	Precision	Verhältnis zwischen richtig erkannten Samples zu allen erkannten Samples.	$PPV = \frac{TP}{TP + FP}$
Negative Predictive value (NPV)		Verhältnis zwischen richtig abgewiesenen Samples zu allen abgewiesenen Samples.	$NPV = \frac{TN}{TN + FN}$
True Positive Rate (TPR)	Recall, Sensitivity, Hit Rate	Verhältnis zwischen richtig erkannten Samples zu allen effektiv zu erkennenden Samples.	$TPR = \frac{TP}{TP + FN}$
True Negative Rate (TNR)	Specificity	Verhältnis zwischen richtig abgewiesenen Samples zu allen effektiv abzuweisenden Samples.	$TNR = \frac{TN}{TN + FP}$
False Positive Rate (FPR)	False Alarm Rate	Verhältnis zwischen falsch erkannten Samples zu allen effektiv abzuweisenden Samples.	$FPR = \frac{FP}{FP + TN}$
False Negative Rate (FNR)		Verhältnis zwischen falsch abgewiesenen Samples zu allen effektiv zu erkennenden Samples.	$FNR = \frac{FN}{FN + TP}$
F-Score	F-Mass	Das F-Mass gibt das Verhältnis zwischen „Precision" und „Recall" an und kann deshalb als alleinige Qualitätsbeschreibung verwendet werden. Es berechnet dabei das harmonische Mittel aus beiden Werten, kann aber mit der Variablen β unterschiedlich gewichtet werden.	$F_\beta = (1 + \beta^2) \cdot \frac{PPV \cdot TPR}{(\beta^2 \cdot PPV) + TPR}$

Tabelle 3.1: Beschreibung der gebräuchlichsten statistischen Werte bei binären Classifiern.

Die in Tabelle 3.1 beschriebenen Werte können beispielsweise in einer Konfusionsmatrix sehr übersichtlich dargestellt werden. Eine solche Matrix ist in Abbildung 3.1 gezeigt. Innerhalb der Matrix werden die Anzahl gefundener Werte eingetragen, mit welchen dann die erwähnten statistischen Werte berechnet werden können. Z.B. die Sensitivity, welche sich aus $\frac{TruePositive}{In\ Wirklichkeit\ Positiv}$ zusammensetzt.

	In Wirklichkeit Positiv	In Wirklichkeit Negativ	
Getestet Positiv	TruePositive	FalseNegative (Typ 1 Fehler)	Positive Predictive Value
Getestet Negativ	FalsePositive (Typ 2 Fehler)	TrueNegative	Negative Predictive Value
	Sensitivity	Specificity	

Abbildung 3.1: Konfusionsmatrix mit den jeweils dazugehörigen statistischen Werten.

Eine weitere Darstellungsmöglichkeit der Qualität eines Classifiers ist die „Receiver Operating Characteristics“ (ROC) Kurve [36]. Eine solche ist in Abbildung 3.2 a) dargestellt. Die ROC Kurve beschreibt das Verhältnis zwischen der „True Positive Rate“ (TPR) und der „False Positive Rate“ (FPR) eines Systems. Grundsätzlich soll ein System aus Classifiern die TPR maximieren und gleichzeitig die FPR minimieren. Da diese beiden Werte allerdings voneinander abhängig sind, ist dies nicht ohne weiteres möglich.

Um eine ROC Kurve zu erzeugen, ist ein Classifier notwendig, welcher zwar nur zwei Klassen erkennt, aber einen Wahrscheinlichkeitswert für jedes Sample zurückgibt. Diese Wahrscheinlichkeit kann dann mit unterschiedlichen Grenzwerten ausgewertet werden. So erzeugt jeder Schwellwert einen anderen Punkt in der Grafik, woraus sich schlussendlich eine Kurve ergibt. Bei einem binären Classifier, welcher effektiv nur die detektierte Klasse zurückgibt, müssen mehrere Classifier mit unterschiedlichen Parametern trainiert werden, um die einzelnen Messpunkte in der Grafik zu erhalten. Zu beachten ist, dass bei beiden Verfahren das System oder die Technologie des Classifiers gemessen wird, welche sich grundsätzlich nur verändert, wenn ein schwerwiegender Parameter (z.B. Anzahl Stufen einer Kaskade), der Algorithmus oder das Dataset ausgetauscht werden.

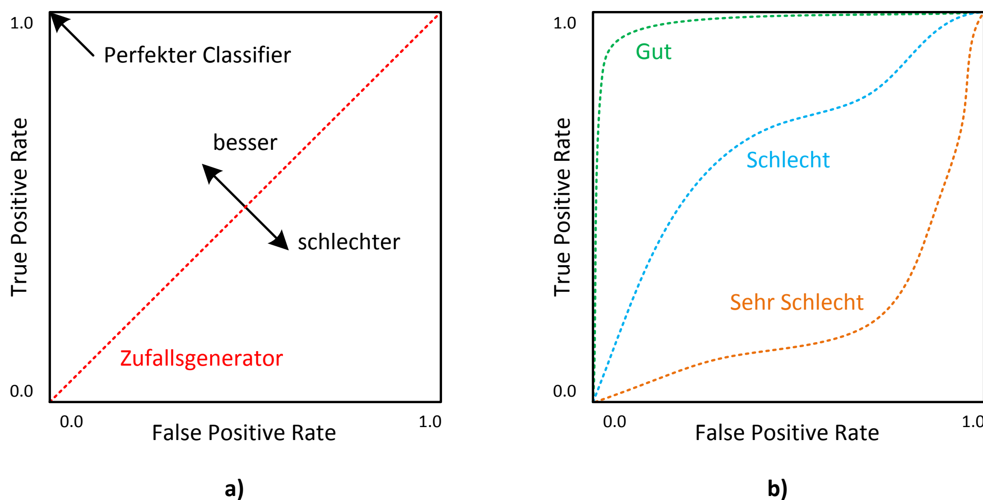


Abbildung 3.2: a) ROC Raum in welchem ein Classifier beschrieben wird.
b) Beispiele von unterschiedlich guten Classifiern. Bild adaptiert von [36].

In Abbildung 3.2 b) sind Beispielkurven mit unterschiedlicher Qualität ersichtlich. Dabei entspricht die grüne Kurve einem typischen System von Classifiern mit akzeptabler Genauigkeit, die blaue Kurve hingegen einem System, welches nur knapp besser ist als ein zufälliges Raten der Klasse. Eine solche Kurve würde z.B. einem Weak Classifier (siehe Abschnitt 3.1.3) entsprechen. Eine orange Kurve sollte in der Praxis nie verwendet werden, da ein solcher Classifier schlechter als ein Zufallsgenerator ist. Dies ist meist auf eine Falschinterpretation der Daten zurückzuführen.

Zusätzlich zu der Möglichkeit Classifier optisch zu unterscheiden, bietet die ROC Kurve die Variante der „Area under an ROC Curve“ (AUC). Dieser skalare Wert kann als alleiniges Qualitätsmerkmal des Classifiers verwendet werden. Allerdings ist dabei Vorsicht geboten, da sich dieser Wert auf den gesamten Parameterraum bezieht, der Classifier im Betrieb aber typischerweise bei fixierten Parametern läuft.

3.1.2 Features

Damit eine Classifier überhaupt funktionstüchtig ist, benötigt er eine Definition der zu beachtenden Merkmale der Daten. Im Bereich Computer-Vision sind dies optische Merkmale wie z.B. die Lichtverhältnisse oder die Textur in einem bestimmten Bildbereich. Solche Merkmale oder Eigenschaften werden als Features bezeichnet. Für die Objekterkennung sind vor allem diejenigen relevant, welche auf den folgenden Seiten beschrieben werden.

Die „Haar-like Features“ wurden ursprünglich von Viola et al. [37] entwickelt und von Lienhart et al. [38] verbessert. Sie sind mit den Haar-Wavelets verwandt, welche 1909 von Alfréd Haar definiert wurden. Haar-like-Features, wie sie in Abbildung 3.3 zu sehen sind, stellen die Helligkeitswerte benachbarter Bildbereiche in Relation und beschreiben so Übergänge von hellen zu dunklen Bereichen und umgekehrt.

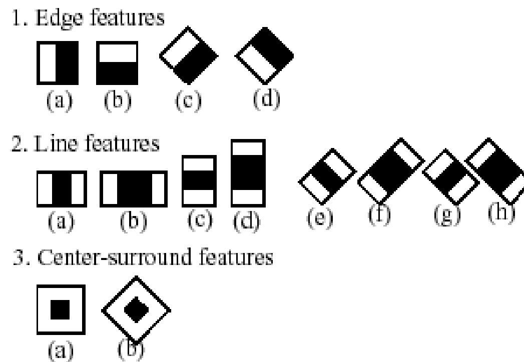


Abbildung 3.3: Haar-like-Features die in OpenCV verwendet werden. Bild von [39].

Dies wird bewerkstelligt, indem jeweils die Summe aller Helligkeitswerte eines Bereiches berechnet und die einzelnen Bereichssummen voneinander subtrahiert werden. Beispielsweise wird in 1.a) aus Abbildung 3.3 die Summe des schwarzen Bereiches vom weissen Bereich abgezogen. Allerdings kann sich dieses Vorgehen bei anderen Formen der Features unterscheiden. Neben den gezeigten Two-Rectangle-Features (Zeile 1), Three-Rectangle-Features (Zeile 2) und Center-Surrounded-Features existiert auch die Variante der Four-Rectangle-Features.

Um diese Features effizient zu berechnen, wird das sogenannte „Integral Image“ verwendet, wie es von Viola in [40] erstmals vorgestellt wurde und eine Art von Lookup-Table darstellt. In dieser wird für jeden Punkt im Bild die Summe aller Helligkeitswerte der links und oberhalb liegenden Punkte addiert und abgespeichert. Das resultierende Integral-Image enthält also im Punkt (0,0) genau die Helligkeitswerte des ursprünglichen Punktes (0,0) und im letzten Punkt in der unteren rechten Ecke die Summe aller Pixel des gesamten Bildes.

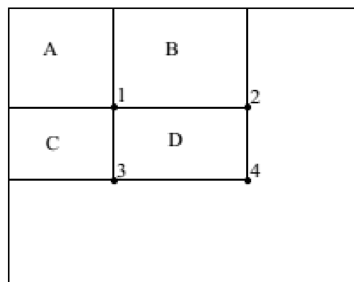


Abbildung 3.4: Berechnung unterschiedlicher Bereiche in einem Integral Image.
Bild aus [40].

Durch diese Information lässt sich die Summe jedes beliebigen Bereiches des Bildes in konstanter Zeit berechnen. Dies wird in Abbildung 3.4 verdeutlicht, indem die Summe des Bereichs D durch die Formel: $D = 4 - (2 + 3) + 1$ berechnet wird. Eine Erweiterung von Lienhart sind die um 45° gedrehten Features, welche ebenfalls in Abbildung 3.3 ersichtlich sind. Um diese effizient zu berechnen, wird jeweils ein separates Integral-Image benötigt.

Eine zweite häufig verwendete Variante von Features bei der Objektdetektion in Bildern sind die „Local Binary Pattern“ (LBP) Features. Sie wurden von Ojala et al. [41] vorgestellt und von Liao et al. [42] zu „Multi-Scale Block Local Binary Pattern“ (MB-LBP) weiterentwickelt. LBP-Features beschreiben die lokale Struktur eines Bildes, also die sogenannte „Textur“, in einem bestimmten Bereich. Dazu wird der Wert eines zentralen Pixels mit den Werten der acht umliegenden Pixeln verglichen und in einem Byte abgespeichert, welcher der beiden Werte jeweils grösser ist. So wird in Abbildung 3.5 a) der Wert 4 des

zentralen Pixels mit den Nachbarwerten verglichen. Wenn der Nachbarpixel den grösseren oder gleichen Wert enthält, wird dafür eine binäre 1 ansonsten eine 0 abgespeichert. Dieser binäre String stellt das LBP-Feature dieses Bereiches dar.

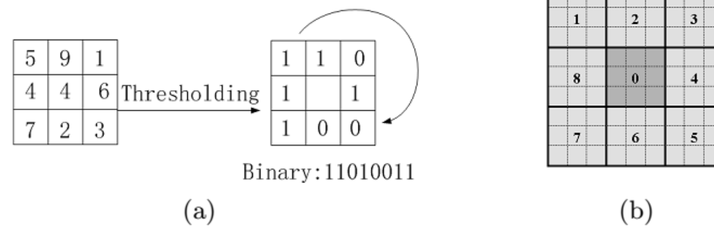


Abbildung 3.5: Berechnung von LBP und MB-LBP Features. Bild aus [42].
a) Basic LBP Operator. b) 9x9 MB-LBP Operator mit Mittelwerten aller Sub-Bereiche.

Da dieses Verfahren immer nur neun Pixel eines Bildes analysiert, spricht man hierbei von einem Deskriptor für Mikrostrukturen. Um grössere Bereiche durch einen LBP-Wert zu beschreiben, benötigt man eine flexiblere Definition des Deskriptors, welche Liao in Form des MB-LBP liefert. Dieser bezieht sich nicht auf einzelne Pixel, sondern auf Bereiche, welche den Mittelwert der betrachteten Pixel erhalten. Dadurch kann dieser auf beliebig grosse Bereiche (auch Makrostrukturen) angewendet werden wie z.B. in Abbildung 3.5 b) auf 3x3 Sub-Bereiche, was zu einem 9x9 MB-LBP-Operator führt. Ein 3x3 MB-LBP-Operator entspricht dementsprechend dem ursprünglichen LBP-Operator.

Da mit diesem System sehr viele Informationen für unterschiedliche Features im Bild wiederverwendet werden können, wird auch bei MB-LBP das Integral-Image als Datenstruktur verwendet. Darin ist jeweils die benötigte Summe der Sub-Bereiche gespeichert, woraus der Mittelwert mithilfe der vorgegebenen Sub-Bereichsgrösse berechnet werden kann.

3.1.3 Boosting

Ein typisches Problem bei schnellen Classifiern ist, dass sie trotz intensivem Trainieren nur schlechte Erkennungsraten erreichen. Wenn der Classifier aber mit sehr vielen Features und somit hoher Präzision trainiert wird, ist einerseits das Training sehr zeitaufwendig, andererseits wird die Auswertung und Detektion sehr langsam. Diese Problematik wird durch das so genannte „Boosting" adressiert, welches von Freund und Schapire [43] entwickelt wurde. Deren Implementation nennt sich heute AdaBoost [44], wobei diese neben anderen wie LogitBoost, TotalBoost oder LPBoost die bekannteste ist.

Wie bereits erwähnt, wird ein Classifier mit sehr vielen Beispieldaten trainiert, wodurch sie sehr viele unterschiedliche Features zur Auswertung erhalten. Um einen starken Classifier zu generieren, welcher zusätzlich alle wichtigen Features beachtet, werden beim Boosting zuerst mehrere schwache Classifier trainiert und diese am Schluss zu einem Grossen zusammengebaut. Diese „Weak Classifier" betrachten dabei nur sehr wenige Features, teilweise sogar nur eines. Typischerweise werden sie deshalb als Entscheidungsstümpfe (Decision Stumps) implementiert, welche eine sehr einfache Variante eines Entscheidungsbaumes darstellen. Durch diese Einschränkung sind sie zwar sehr schnell, erreichen allerdings nur eine schwache Präzision. Um durch die Kombination mehrere Weak-Classifer einen genaueren Gesamtklassifizierer zu erzeugen, müssen die Weak-Classifier nur leicht präziser als ein Zufallsgenerator (Recall > 0.5) sein.

Damit der resultierende Classifier die schwachen Classifier optimal einsetzen kann, werden diese nach Präzision gewichtet. So wird ein präziserer Classifier einen stärkeren Einfluss auf die Endklassifizierung haben, als einer mit geringerer Präzision. Zudem wird beim Trainieren der Weak-Classifier eine Gewichtung der Trainingsdaten eingesetzt, um unterschiedlich präzise Classifier zu erzeugen. Abbildung 3.6 symbolisiert den Ablauf und die Beziehungen der einzelnen Teile während dem Trainieren.

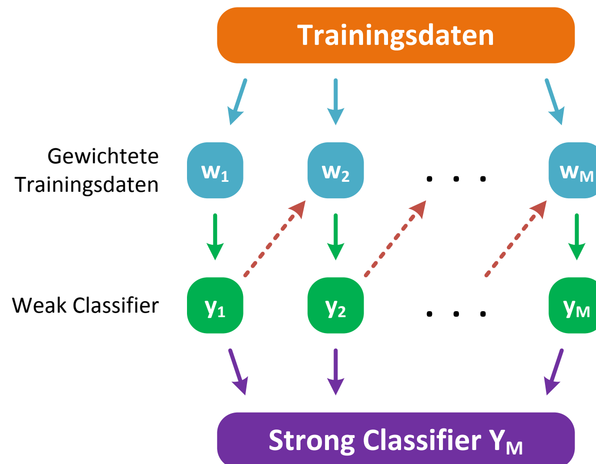


Abbildung 3.6: Erstellen eines Strong Classifier mittels Boosting. Die grünen Pfeile symbolisieren das Trainieren der Weak Classifier mittels der unterschiedlich gewichteten Trainingsdaten. Diese Gewichtung wird jeweils nach dem Trainieren des vorherigen Classifiers angepasst. Der resultierende Strong Classifier besteht aus M Weak Classifiers. Grafik aus [45, p. 658] adaptiert.

Ähnlich wie die Gewichtung am Schluss werden die Trainingsdaten nach jedem Training eines Weak Classifiers bewertet. Diese Gewichtung wirkt sich auf das Training des nächsten Weak-Classifiers aus, indem es schlecht erkannte Samples von vorherigen Classifier stärker gewichtet. Dadurch wird erzwungen, dass der neue Classifier das vorherig schlecht klassifizierte Sample besser erkennt. So wird eine gleichmässig gute Erkennungsrate des Strong Classifier über alle Samples, und folglich über alle Features, erreicht.

Der daraus resultierende Classifier $Y_M(x)$ wird durch die Formel 3.1 definiert, wobei α_m die Gewichtung der einzelnen Classifier und $y_m(x)$ die Classifier selbst darstellt.

$$Y_M(x) = sign\left(\sum_{m=1}^{M} \alpha_m y_m(x)\right)$$

Formel 3.1: Resultierender Classifier nach Boosting. Die Funktion sign() Rundet auf -1 oder 1 und verbleibt bei exaktem 0-Wert. Formel aus [45].

Das hier beschriebene Boosting bezieht sich nur auf binäre Classifier, es existieren aber Varianten, die auf Multiklassen- oder Regressions-Classifier angewendet werden können. Dass solche realisierbar sind, lässt sich mithilfe der Überlegung zeigen, dass sich ein Multiklassenproblem auf einzelne binäre Klassifizierungsprobleme aufteilen lässt. Ausführlichere Erläuterungen zu Erweiterungen finden sich in [43].

3.1.4 Rejection Cascades

Ein ähnlicher Ansatz wie beim Boosting wird bei den „Rejection Cascades" von Viola et al. [40] gewählt, indem ebenfalls mehrere unterschiedliche Classifier kombiniert werden, um die Verarbeitungsgeschwindigkeit bei der Auswertung von Samples zu erhöhen. Allerdings arbeiten diese Classifier nicht wie beim Boosting parallel, sondern sequenziell. Dabei ist der erste auswertende Classifier relativ simple und dadurch schnell, um erste negative Samples als solche zu erkennen. Nur diejenigen Samples, welche im ersten Classifier nicht ausgeschlossen werden, werden vom nachfolgenden analysiert. Dieser Ablauf ist in Abbildung 3.7 dargestellt. Dabei wird ein Sample nur als positiv erkannt, wenn es alle Stufen einer Rejection-Cascade durchlaufen hat und somit als letztes vom Y_M akzeptiert wurde.

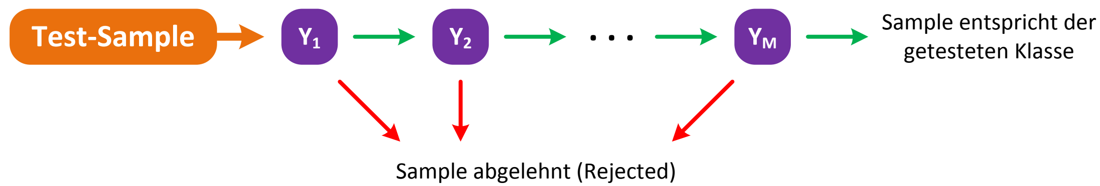


Abbildung 3.7: Ablauf einer Auswertung in einer Rejection Cascade mit M Stufen. In den ersten Stufen werden sehr viele Samples verarbeitet und so viel als mögliche negative Samples ausgeschlossen. Samples, die bis zur letzten Stufe vordringen, sind sehr selten.

Diese Technik ermöglicht es, negative Samples sehr früh im Evaluationsprozess auszuschliessen und so einen weiteren Verarbeitungsaufwand zu verhindern. Dadurch wird die Analyse einer grossen Anzahl Samples stark beschleunigt, was wiederum ermöglicht, gegen Ende der Kaskade stärker trainierte Classifier zu verwenden, die zwar mehr Rechenaufwand benötigen, aber dadurch präzisere Resultate liefern.

Somit ergibt sich der typische Aufbau eines Cascade-Classifiers, welcher mit relativ schwachen und wenige Features umfassenden Classifiern beginnt (Y_1) und mit starken und viele Features umfassenden Classifiern (Y_M) endet. Diese Taktik liefert allerdings nur dann gute Ergebnisse, wenn sehr viele der zu testenden Samples nicht zur testenden Klasse gehören und somit früh abgelehnt werden.

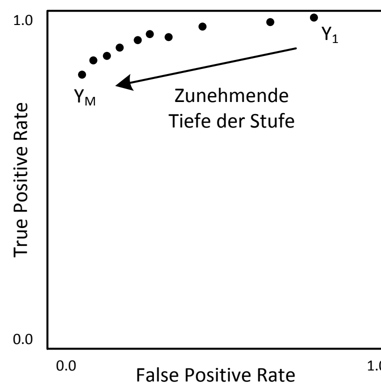


Abbildung 3.8: ROC Grafik der Classifier innerhalb eines Cascade Classifiers.

Werden die ROC-Grafiken der einzelnen Classifier in Abbildung 3.8 betrachtet, kann erkannt werden, dass frühe Stufen eher schlechte Fehlalarmraten aber gute Erkennungsraten haben, spätere Stufen dagegen versuchen, beide Werte zu optimieren.

3.1.5 Soft-Cascade

Um das Vorgehen der Rejection-Cascades anpassungsfähiger zu machen und trotzdem seine Geschwindigkeit und Präzision beizubehalten, schlägt Bourdev et al. [46] die so genannten „Soft Cascades“ vor. Diese sollen mehrere Schwächen von den bisherigen Cascade-Classifiern beheben: Die Tatsache, dass die Erkennungswahrscheinlichkeit einer vorherigen Stufe nicht in die Evaluation der nächsten Stufe einfliesst, das Problem, dass Multi-View-Detektoren meist nicht mit einem einzelnen Classifier verwirklicht werden können und die Problematik der Kalibrierung, welche nur durch den langwierigen Prozess des Neutrainierens erreicht wird.

Um dies zu erreichen, werden die einzelnen Stufen der Kaskade zu Weak-Classifiern mit kontinuierlichem Ausgabewert vereinheitlicht. Jeder dieser Weak-Classifier ist gewichtet und bewertet nur ein einzelnes Feature. Da solche Classifier sehr stark denen des Boostings gleichen, wird genau dieselbe Technik verwendet, um sie zu generieren. Der gesamte Classifier besteht aus allen diesen Ein-Feature-Classifiern, die nacheinander ausgewertet werden. Bei dieser Auswertung erzeugt jede Stufe einen Resultatwert, der zu den vorherigen Werten addiert die „Cumulative Sum“ ergibt. Diese Gesamtsumme wird jeweils mit einem Grenzwert verglichen, welcher bestimmt, ob das Sample abgelehnt oder weiter ausgewertet wird.

Abbildung 3.9 a) zeigt Beispiele solcher Gesamtsummen-Kurven, welche von positiven und negativen Samples in Bezug auf Gesichtserkennung stammen. Es lässt sich der generell unterschiedliche Kurvenverlauf der beiden Klassen erkennen. Vor allem der steile Anstieg des Wertes bei den ersten Feature-Classifiern (y_0 und folgende) grenzt die positiven Samples von den negativen ab.

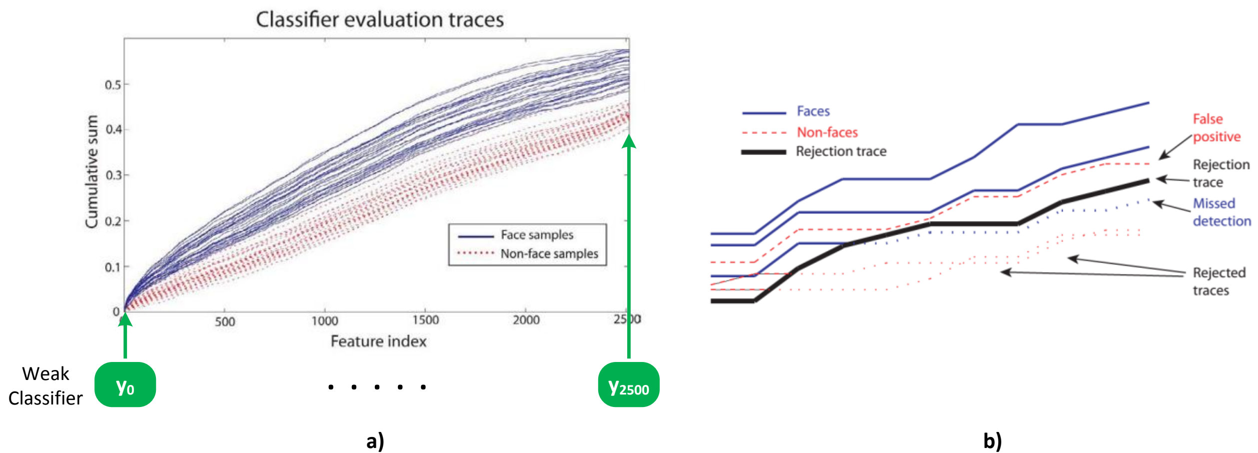


Abbildung 3.9: Funktionsprinzip eines Soft Cascade Classifier. Erweiterte Bilder aus [46]. a) Beispielkurven der Cumulated Sum von negativen und positiven Samples. Die Abszisse symbolisiert das Fortschreiten der Evaluation in der Kaskade aus Feature-Classifiern. b) Darstellung des Verhaltens der Grenzwertfunktion.

Bei der effektiven Evaluation wird der errechnete Wert nach jeder Stufe (einzelner Feature-Classifier) mit dem zur Stufe zugehörigen Grenzwert verglichen. Dies ist in Abbildung 3.9 b) ersichtlich, in welcher die schwarze Kurve den Grenzwert symbolisiert. Um die Geschwindigkeit von „Hard Cascade“ Classifiern (Rejection-Cascade-Classifier, siehe Kapitel 3.1.4) zu erreichen, werden alle Samples von der weiteren Evaluation ausgeschlossen, sobald ihre Gesamtsumme unter den verlangten Grenzwert fällt. Obwohl dies dem Verfahren der Hard-Cascades ähnlich scheint, ist es weniger restriktiv und erlaubt einem Sample bei einzelnen Feature-Classifiern einen schlechten Wert zu erzielen, ohne direkt abgelehnt zu werden. Dies wird durch die Verwendung der kumulierten Summe als Vergleichswert mit dem Grenzwert ermöglicht, wodurch ein schlechter Wert durch einen guten Wert bei vorherigen Features ausgeglichen werden kann.

Die wichtigste Eigenschaft der Soft-Cascades beruht ebenfalls auf der Verwendung eines Grenzwertes. Dabei kann die Erkennungsrate, genauer die ROC-Kurve, jederzeit durch die Veränderung der Grenzwerte in der Grenzwertfunktion angepasst werden. Dadurch lässt sich zudem die Geschwindigkeit des Classifiers verändern, weshalb Bourdev bei der Kalibrierung des Classifiers den Begriff „ROC-Surface“ verwendet, welcher die Erweiterung der ROC-Kurve mit der Dimension der Auswertungsgeschwindigkeit beschreibt. In diesem neuen dreidimensionalen Raum lässt sich für die verlangten Spezifikationen eine optimale Einstellung der Grenzwertfunktion finden, wobei sich dieses Optimum jeweils auf der ROC-Oberfläche befindet.

3.1.6 Einsatz und Beispiel eines Classifiers

Als Beispiel eines kompletten Classifiers soll hierfür der Viola-Jones-Detektor genauer betrachtet werden. Abbildung 3.10 zeigt die grobe Struktur und Funktionsweise dieses häufig eingesetzten Systems.

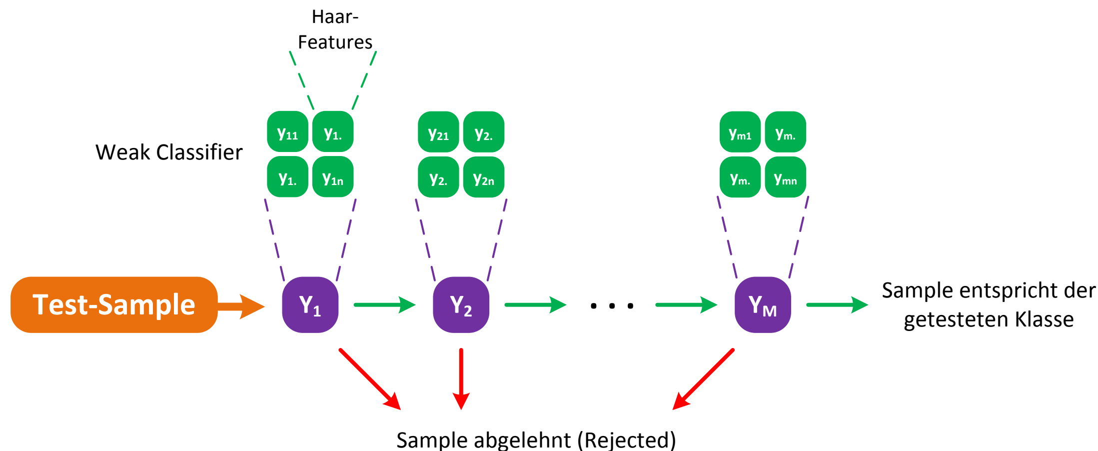


Abbildung 3.10: Aufbau des Viola-Jones-Detektors. Dieser besteht aus M durch Boosting generierte Classifier, welche zu einer Kaskade mit M Stufen zusammengebaut werden. Als Vergleichsmerkmal werden Haar-Features verwendet.

Es werden Haar-like-Features verwendet, um mittels Boosting mehrere Weak-Classifier zu starken Classifiern zu kombinieren. Diese Decision-Stumps werden dann zu einer Detektionskaskade zusammengefügt, welche die meisten aller negativen Samples bereits in den ersten Stufen ablehnt.

Wie diese neuen, zu verarbeitenden Samples entstehen wurde bis jetzt allerdings nicht besprochen. Dazu wird das Gesamtbild mit einem Skalierungsfaktor (meist zwischen 1.1 bis 1.5) mehrfach verkleinert. Die einzelnen Skalierungen dieser so genannten „Multi-Scale Pyramid“ werden dann in die durch den Classifier definierte „Window Size“ unterteilt. Diese unterschiedlich skalierten Ausschnitte entsprechen der Grösse der Trainings-Samples und können nun mit dem Detektor klassifiziert werden. Dieser Detektionsablauf wird in OpenCV mittels der Methode `detectMultiScale` der Klasse `CascadeClassifier` durchgeführt.

3.2 Trainieren eines Classifiers mit OpenCV

Um die Köpfe, und nicht nur die Gesichter, der Personen im Video erkennen zu können, wird ein neuer Cascade-Classifier trainiert. Dieser soll die Erkennung in einem einzigen Arbeitsschritt ermöglichen und muss deshalb mit einem geeigneten Dataset, welches Kopfansichten aus allen Richtungen beinhaltet, trainiert werden. Weiter soll evaluiert werden, ob der entsprechende Classifier mit den üblichen Haar-Features oder den neuen und schnelleren LBP-Features bessere Resultate erzielt. Die Anleitung dazu findet sich in der OpenCV-Dokumentation [47] und im Tutorial [48]. Die dazu benötigten Tools werden leider nicht mitgeliefert und müssen durch eine eigene Kompilation von OpenCV erzeugt werden.

3.2.1 Einrichten OpenCV

Während der Verwendung von OpenCV (GIT 2.4.9) mit GPU-Unterstützung im Projekt 7 zeigte sich, dass das CUDA-Toolkit 5.0 inkompatibel mit dem Visual Studio 2012 Compiler ist und OpenCV deshalb nicht für die für AMP notwendige Compiler-Version und gleichzeitig CUDA kompiliert werden kann. Diverse im Internet verfügbare Workarounds führen leider nicht zum Erfolg.

Mit der neuen Version des CUDA-Compilers nvcc im Toolkit 5.5 und der stabilen Version 2.4.6.1 oder der GIT-Version 2.9.0.0 von OpenCV lässt sich eine gewünschte Version mit GPU-Unterstützung für den vc11-Compiler erstellen. Dazu werden, wie in [49] beschrieben, die Quelldateien via GIT heruntergeladen und

das Projekt mit CMake erstellt. Dazu sollten mindestens die Einstellungen aus Tabelle 3.2 in CMake übernommen werden. Gewisse Hinweise auf diese Einstellungen findet man auf [50].

Aktivieren / Ausfüllen	Deaktivieren
WITH_EIGEN	CUDA_ATTACH_VS_BUILD_RULE_TO_CUDA_FILE
EIGEN_INCLUDE_PATH	BUILD_TEST
WITH_CUBLAS	BUILD_PERF_TEST
WITH_TBB	BUILD_DOCS
TBB_INCLUDE_DIRS	BUILD_EXAMPLES
TBB_STDDEF_PATH	
TBB_LIB_DIR	
ANT_EXECUTABLE	
WITH_OPENCL	
WITH_NVCUVID	
CUDA_FAST_MATH	

Tabelle 3.2: Einstellungen in CMake für das erfolgreiche Kompilieren von OpenCV mit CUDA 5.5 und Visual Studio 2012 Compiler vc11.

3.2.2 Datasets / Samples

Als Trainingsdaten wird das Dataset UcoHead [51] mit über 9000 Trainingsbildern von insgesamt zehn Personen verwendet, welche aus unterschiedlichen Winkeln und mit diversen Positionen abgelichtet wurden. Die 9442 Trainings- und 4604 Test-Bilder sind als 40x40 Pixel grosse 8bit-Graustufenbilder im pgm-Format abgelegt und können direkt so verwendet werden. Ergänzend zum Dataset wird auf das Paper von Muñoz-Salinas et al. [52] verwiesen.

Weiter wird das CoffeeBreak Head Orientation Dataset [53] mit 9522 Trainings- und 8595 Testbildern von einer grossen Anzahl unterschiedlicher Personen verwendet. Die Bilder sind als 50x50 Pixel grosse Farbbilder im jpg- und png-Format vorhanden und werden direkt so verwendet. Ergänzend zum Dataset wird auf das Paper von Tosato et al. [54] verwiesen.

Da für das Trainieren eines Classifiers zusätzlich Hintergrundbilder (Negatives) nötig sind, werden die 3019 Bilder von Naotoshi [48] verwendet, welche unter „Downloads" z.B. mit dem JDownloader heruntergeladen werden können. Diese sind als Graustufenbilder im jpg-Format mit einer Grösse von 640x480 Pixel abgespeichert.

Um diese Bilder für das Training verwenden zu können, müssen sie entsprechend vorbereitet werden. Als erstes wird eine Textdatei benötigt, in welcher alle Dateien aufgelistet werden. Diese Datei befindet sich am einfachsten im selben Verzeichnis wie die Bilder selber. Die Datei für die Hintergrundbilder sieht dann z.B. wie in Listing 3.1 aus.

```
neg_0001.jpg
neg_0002.jpg
neg_0003.jpg
neg_0004.jpg
etc.
```

Listing 3.1: Auszug aus der Datei für die Hintergrundbilder: _negatives.txt

Für die Trainingsbilder wird eine erweiterte Datei benötigt, in welcher jeweils für jedes Objekt auf dem Bild die Position und der Bereich angegeben werden muss. Da sich auf den Bildern des UcoHead-Datasets pro Bild immer nur ein Objekt, also der Kopf, befindet und das Bild entsprechend zugeschnitten ist, kann nach jedem Dateiname folgender String hinzugefügt werden: `1 0 0 40 40`. Die `1` nummeriert das erste und

einzige Objekt, die beiden `0` die Ursprungsposition des Bereichs und die beiden `40` die Bereichsgrösse von 40x40 Pixel. Somit entspricht die Datei für die Trainingsdaten etwa dem Listing 3.2.

```
image0000.pgm 1 0 0 40 40
image0001.pgm 1 0 0 40 40
image0002.pgm 1 0 0 40 40
image0003.pgm 1 0 0 40 40
etc.
```

Listing 3.2: Auszug aus der Datei für die Trainingsbilder: _train.txt

Als nächstes werden alle Trainingsbilder zu einer vec-Datei zusammengefasst, welche nur die entsprechenden Bereiche der definierten Objekte enthält. Diese Sample-Datei wird schliesslich zum Trainieren benötigt und mit dem Tool `opencv_createsamples.exe` erzeugt. Dazu wird eine kleine cmd-Datei geschrieben, welche die benötigten Befehls-Parameter enthält. Listing 3.3 zeigt die verwendeten Parameter und Befehle. `num` definiert dabei die Anzahl der positiven Samples, welche generiert werden sollen. Im Fall, dass mehrere Objekte pro Bild vorkommen, kann diese grösser als die Anzahl der Trainingsbilder sein. `vec` bezeichnet die Zieldatei und `info` die Quelldatei der Trainingsbilder, welche im vorherigen Schritt erstellt wurde. Die Parameter `w` und `h` definieren die endgültige Skalierung der Samples, damit alle die gleichen Abmessungen haben. Da im UcoHead Dataset bereits alle Samples auf 40x40 Pixel skaliert sind, kann man diesen Wert belassen.

```
opencv_createsamples.exe -num 9442 -vec Samples\train.vec
     -info Datasets\UcoHead\train\_train.txt -w 40 -h 40
pause
```

Listing 3.3: Batch-Datei (create_train1.cmd) für die Erzeugung der Sample-Datei: train.vec

Alle Pfad- oder Dateiangaben sind relativ, können aber auch durch absolute Angaben ersetzt werden. Weitere Parameter finden sich in [47] und können z.B. für die Generierung von rotierten Samples verwendet werden. Dadurch lassen sich aus wenigen Trainingsbildern viele unterschiedliche Samples erzeugen. Um etwaige Hinweis- oder Abbruchtexte nach der Ausführung lesen zu können, empfiehlt es sich, den `pause`-Befehl am Ende der Batch-Datei zu verwenden.

Die Verwendung des Tools mit nur den Parametern `vec`, `w` und `h`, wie in Listing 3.4, ermöglicht das Betrachten der erzeugten Samples der vec-Datei.

```
opencv_createsamples.exe -vec Samples\train.vec -w 40 -h 40
```

Listing 3.4: Batch-Datei (show_train.cmd) für die Betrachtung der erzeugten Samples.

Der komplette Vorgang der Samples-Erzeugung wird ebenfalls für das CoffeeBreak Dataset durchgeführt.

3.2.3 Performance-Test

Die Hintergrundbilder werden primär während des Trainiervorgangs benötigt, können aber zusätzlich für Performancetests nach dem Trainieren zum Einsatz kommen. Dazu wird eines der Testbilder verwendet, welches auf unterschiedlichen Hintergründen erkannt werden soll. Wie in Listing 3.5 ersichtlich, wird ein spezifisches Bild und alle vorhandenen Hintergrundbilder kombiniert, um neue Test-Samples zu erstellen. Um sicherzustellen, dass sich noch keine Testbilder im Zielordner befinden, wird dieser vorgängig komplett geleert.

```
del Samples\test\* /Q
opencv_createsamples.exe -img Datasets\UcoHead\test\image1779.pgm -num 3019
     -bg Datasets\Negatives\_negatives.txt -info Samples\test\_test.txt
     -maxxangle 0.6 -maxyangle 0 -maxzangle 0.3 -maxidev 100
     -bgcolor 0 -bgthresh 0 -w 40 -h 40
pause
```

Listing 3.5: Batch-Datei (create_test.cmd) für die Generierung unterschiedlicher Test-Samples.

Bei Aufruf von `opencv_createsamples.exe` definiert der Parameter `img` das zu erkennende Testbild, `num` die Anzahl Hintergründe, `bg` die Quelldatei der Hintergründe und `info` die Zieldatei der neuen Samples. Es handelt sich dabei um keine vec-Datei mehr, sondern um eine ähnlich, wie bei den Hintergrundbildern aufgebaute, Textdatei. Die effektiven Bilder werden in denselben Ordner wie die Textdatei gespeichert. Die weiteren Parameter definieren die auf das Testbild anzuwendenden Transformationen, welche das Bild variieren lassen.

Nach der Generierung der Test-Samples, kann ein existierender Haar-Classifier getestet werden. In der aktuellen OpenCV-Version 2.9.0 funktioniert dies leider noch nicht mit LBP-Classifier. Listing 3.6 zeigt die Anwendung des Performance-Tools `opencv_performance.exe`.

```
opencv_performance.exe -data Cascades\haar\haar.xml -info Samples\test\_test.txt
     -w 40 -h 40 -ni
pause
```

Listing 3.6: Batch-Datei (test_haar.cmd) für den Performancetest des Haar-Classifiers.

Der Parameter `data` definiert die Konfiguration des Classifiers und `info` die Eingabedaten in Form der Test-Samples. Die Option `ni` verhindert die Speicherung der gefundenen Bildbereiche während der Detektion.

3.2.4 Trainieren

Da das Trainieren mit LBP Features deutlich schneller ist als mit Haar Features, werden Classifier hauptsächlich mit diesen trainiert. Ausgewählte Einstellungen werden dann für Trainings mit Haar eingesetzt, damit die beiden Varianten miteinander verglichen werden können. Grundsätzlich wird das Tool `opencv_traincascade.exe` verwendet. Da es aber noch Probleme mit Haar-Trainings hat, wird für diese die alte Variante des Tools `opencv_haartraining.exe` verwendet. Die jeweiligen Aufrufe sind in Listing 3.7 und Listing 3.8 aufgezeigt. Beide enthalten dieselben Parametereinstellungen, soweit der jeweilige Parameter in beiden Varianten existiert.

```
opencv_traincascade.exe -data Cascades\lbp\ -vec Samples\train.vec
     -bg Datasets\Negatives\_negatives.txt -numPos 6000 -numNeg 3000
     -precalcValBufSize 1512 -precalcIdxBufSize 1512 -numStages 18
     -featureType LBP -w 40 -h 40 -minHitRate 0.9999 -maxFalseAlarmRate 0.3
     -weightTrimRate 0.95 -maxDepth 1 -mode ALL -bt GAB
pause
```

Listing 3.7: Batch-Datei (train_lbp.cmd) für das Trainieren von Classifiern mit LBP-Features.

```
opencv_haartraining.exe -data Cascades\haar\ -vec Samples\train.vec
     -bg Datasets\Negatives\_negatives.txt -npos 6000 -nneg 3000 -mem 512
     -nstages 18 -w 40 -h 40 -minhitrate 0.9999 -maxfalsealarm 0.3
     -weighttrimming 0.95 -mode ALL
pause
```

Listing 3.8: Batch-Datei (train_haar_haartraining.cmd) für das Trainieren von Classifiern mit Haar-Features.

Eine Übersicht über die Parameter ist in Tabelle 3.3 gezeigt, die genaue Auflistung ist unter [47] verfügbar.

Parameter: opencv_traincascade	**Parameter: opencv_haartraining**	**Beschreibung**
-data	-data	Pfadangabe für die finale xml-Datei, welche die finalen Parameter des Classifiers enthält.
-vec	-vec	Dateiangabe zur train.vec-Datei, welche alle positiven Trainingsdaten enthält und in Kapitel 3.2.2 erzeugt wurde.
-bg	-bg	Dateiangabe zur _negatives.txt-Datei, welche alle negativen Trainingsdaten enthält und in Kapitel 3.2.2 erzeugt wurde.
-numPos	-npos	Anzahl der positiven Samples, die für das Training jeder Stufe verwendet werden. Muss mit der Formel 3.2 berechnet werden.
-numNeg	-nneg	Anzahl zu verwendende negative Samples. Entspricht der Anzahl der Dateien in _negatives.txt.
-precalcValBufSize	-mem	Menge des zu verwendenden Arbeitsspeichers für die Vorberechnungen. Sollte möglichst gross gewählt werden, um schnellste Ausführung zu erreichen. Bei zu grossen Werten stürzt das Tool ab.
-precalcIdxBufSize		Wie oben.
-numStages	-nstages	Anzahl Stufen der Kaskade. Typischer Wert: 20.
-featureType		Die Art der zu verwendenden Features. Mögliche Werte: HAAR, LBP.
-w	-w	Die Breite der verwendeten positiven Bilder. Typischer Wert bei Köpfen: 25 Pixel.
-h	-h	Die Höhe der verwendeten positiven Bilder. Typischer Wert bei Köpfen: 25 Pixel.
-minHitRate	-minhitrate	Die gewünschte minimale Trefferrate pro Stufe. Je grösser, desto aufwendiger der Trainingsprozess, desto präziser die Resultate des Classifiers. Die gesamte Trefferrate beträgt ungefähr: $minHitRate^{numStages}$. Typischer Wert: 0.999.
-maxFalseAlarmRate	-maxfalsealarm	Die gewünschte maximale Fehlalarmrate pro Stufe. Je kleiner, desto aufwendiger der Trainingsprozess, desto weniger Fehlalarme. Die gesamte Fehlalarmrate beträgt ungefähr: $maxFalseAlarmRate^{numStages}$. Typischer Wert: 0.5.
-weightTrimRate	-weighttrimming	Prozentzahl des Gewichts, welches bestimmt, ob Äste in den Decision Trees der Weak Classifier weggelassen werden. Typischer Wert: 0.95.
-maxDepth		Maximale Tiefe der Decision Trees der Weak Classifier. Ein typischer Wert ist 1, was den erwähnten Decision Stumps entspricht.
-mode	-mode	Bestimmt die Wahl der verwendeten Haar-Features, welche in Abbildung 3.3 gezeigt sind. Mögliche Werte: BASIC, CORE, ALL.
-bt		Art des Boostings. Mögliche Werte: GAB, DAB, RAB, LB. Typischerweise wird GAB (Gentle AdaBoost) verwendet.

Tabelle 3.3: Wichtige Parameter für das Trainieren von Cascade-Classifiern.

Speziell ist der Parameter `numPos`, welcher nicht der Anzahl positiver Bilder entspricht. Stattdessen muss er mithilfe der Formel 3.2 berechnet werden. Um diese Parameter nicht neu definieren zu müssen, wenn z.B. die Anzahl Stufen oder die Datasets geändert werden, wird `S=3000` und `#vec=9000` gewählt, was zu einem universell einsetzbaren Wert für `numPos=6000` führt.

$$\#vec \geq numPos + ((numStages - 1) \cdot (1 - minHitRate) \cdot \#numPos) + S$$

Formel 3.2: Wird benötigt, um den Parameter numPos für das Training von Cascade-Classifiern zu bestimmen. #vec entspricht der Anzahl positiver Samples in der vec-Datei. S entspricht der Anzahl der negativen Samples.

Zu beachten ist, dass ein solcher Trainingsvorgang teilweise sehr lange dauern kann. Typische Trainings mit LBP-Features dauern ca. zwei Stunden und mit Haar-Features ca. zwei Tage. Einer der offensichtlichen

Gründe für diesen Unterschied ist, dass nur die neuere Variante `opencv_traincascade.exe` mehrere Cores der CPU ausnutzt. Allerdings speichern beide Trainings-Tools ihre Zwischenergebnisse nach jeder erfolgreichen Stufe ab und ermöglichen so das Fortfahren des Trainings, falls es unterbrochen wurde. Die Zwischenresultate werden im Verzeichnis, welches mit dem Parameter `-data` definiert wird, abgespeichert.

3.2.5 Auswertung

Alle Trainings und Testläufe der Classifier werden auf dem Windows System durchgeführt, welches bereits in Tabelle 2.9 erwähnt wurde. Um die trainierten Classifier zu bewerten, werden die folgenden Videosequenzen der Testdaten aus Projekt 7 verwendet.

- \Testdaten\Moritz\ZickZack.MP4
- \Testdaten\Dominik\ZickZack.MP4
- \Testdaten\Thekla\ZickZack.MP4
- \Testdaten\Martin\ZickZack.MP4
- \Testdaten\MehrerePersonen\Flur.MP4

Da leider nur die mit Haar-Features trainierte Classifier mit dem Performance-Test bewertet werden können, wird eine subjektive Einschätzung der Präzision und Fehlerrate vorgenommen und mit einer Note zwischen 1 und 10 bewertet. Um genauer erwägen zu können, wie die Note zustande kommt, wird sie jeweils mit einer kurzen Bemerkung versehen. Alle Details der Trainings und Resultate können dem Anhang D.2 entnommen werden.

Der erste Classifier `cascade_lbp_head_130806.xml` wird mit den Parametern aus Abbildung 3.11 trainiert. Diese führen zu einem funktionierenden, aber erst mässig guten Ergebnis. Als erstes wird evaluiert, welche Grösse der positiven Samples optimal ist. Ab dem Classifier `cascade_lbp_head_130819.xml` ist klar, dass 40x40 Pixel die besten Ergebnisse erzielen.

Datum	06.08.2013
featureType	LBP
numPos	6000
numNeg	3000
precalcValBuffSize	1512
precalcIdxBufSize	1512
numStages	20
w	40
h	40
minHitRate	0.999
maxFalseAlarmRate	0.5
weightTrimRate	0.95
maxDepth	1
mode	ALL
Boosting	GAB
Datasets	1 + 2

Abbildung 3.11: Parameter des ersten Classifiers cascade_lbp_head_130806.xml mit LBP-Features.

Vergleicht man diese Zahl mit den 20x20 bis 25x25 Pixel, welche bei den in OpenCV mitgelieferten Gesichtsdetektoren verwendet werden und betrachtet den in Abbildung 3.12 ersichtlichen grösseren Rand um den Kopf, erscheint diese Zahl sinnvoll. Zudem wird dadurch eine Qualitätsverminderung der Bilder durch Skalieren verhindert, da die letzlich verwendeten Bilder des Datasets UcoHead bereits 40x40 Pixel gross sind.

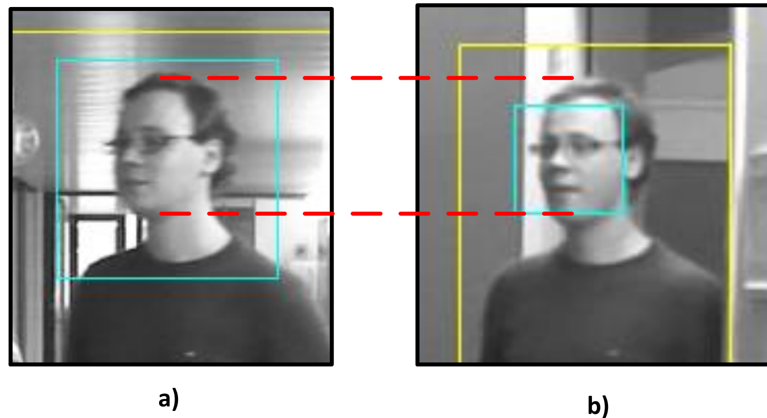


Abbildung 3.12: Vergleich der Detektionsfensters (Cyan) des in a) eigenen Kopfdetektors (cascade_lbp_head_130823_2.xml) und des in b) Gesichtsdetektors von OpenCV (haarcascade_frontalface_alt.xml). Der gefundene Bereich des Bewegungsdetektors ist gelb markiert.

Als nächstes werden in den aufgelisteten Trainings die unterschiedlichen Arten des Boostings getestet.

- `cascade_lbp_head_130816.xml`
- `cascade_lbp_head_130819.xml`
- `cascade_lbp_head_130819_3.xml`
- `cascade_lbp_head_130820.xml`

Die Varianten mit „LogitBoost“ (LB) und „Discrete AdaBoost“ (DAB) schneiden dabei schlecht ab, da sie sehr viele Bereiche detektieren, welche keinen Kopf enthalten, weshalb sie zwar eine hohe TPR, jedoch auch eine sehr hohe FPR haben. „Real AdaBoost“ (RAB) hingegen findet nur sehr wenig und leider sind die wenigen detektierten Bereiche meist keine Köpfe. Dadurch ergibt sich nicht nur eine hohe FPR, sondern zudem eine schlechte TPR. „Gentle AdaBoost“ (GAB) schneidet am besten ab und liefert eine akzeptable TPR mit der kleinsten gemessenen FPR (siehe Kapitel 3.1.1 für Erklärungen der Statistiken).

Der nächste Schritt ist das Kalibrieren der Anzahl Stufen der Kaskade. Die häufig verwendete Stufenzahl 20 liefert bereits akzeptable Resultate. Die im Folgenden aufgelisteten Tests liefern die Vergleichswerte von unterschiedlichen Stufenzahlen.

- `cascade_lbp_head_130819.xml`
- `cascade_lbp_head_130820_2.xml`
- `cascade_lbp_head_130821.xml`
- `cascade_lbp_head_130821_2.xml`

Bei einer Erhöhung der Stufenzahl von 20 auf 24 steigt einerseits die Trainingsdauer von ca. zwei auf vier Stunden. Dies lässt sich mit der zusätzlichen Anzahl Classifiern, die trainiert werden müssen, erklären. Da diese hinzugefügten Classifier zusätzlich stärker sein müssen, benötigen sie mehr Trainingszeit als die vorhergehenden. Andererseits wirkt sich die Stufenzahl auf die Statistiken aus. Eine höhere Anzahl Stufen hat zwar eine kleinere FPR, aber auch eine schlechtere TPR zur Folge. Der Test mit der Stufenzahl 18 erzielt dabei die besten Resultate. Erstens wird eine kurze Trainingszeit und eine schnelle Detektion erreicht und zweitens eine akzeptable TPR.

Dem erhöhten Wert der FPR wird in den anschliessend aufgelisteten Tests durch die Herabsetzung der `maxFalseAlarmRate` begegnet.

- `cascade_lbp_head_130821_2.xml`
- `cascade_lbp_head_130821_3.xml`
- `cascade_lbp_head_130821_4.xml`
- `cascade_lbp_head_130823_2.xml`

Dadurch kann die Fehlerkennung von nicht-Köpfen stark verringert werden. Allerdings steigt die Trainingsdauer leicht an und beträgt bei einer `maxFalseAlarmRate` von 0.25 immer noch akzeptable zwei Stunden. Um die Erkennungsrate zu verbessern, wird im Test `cascade_lbp_head_130822_2.xml` der Wert `minHitRate` von 0.999 auf 0.9999 erhöht, was eine Steigerung der Qualität des Classifiers mit sich bringt.

Als letzter Schritt werden die zwei Datesets in den folgenden Tests miteinander verglichen.

- `cascade_lbp_head_130819.xml`
- `cascade_lbp_head_130819_2.xml`
- `cascade_lbp_head_130822_2.xml`
- `cascade_lbp_head_130822_3.xml`

Das UcoHead Dataset erweist sich dabei als effektiver als das CoffeeBreak Dataset. Gründe dafür sind die unterschiedlichen Bedingungen, die während den Aufnahmen herrschten. So stammen die Samples des CoffeeBreak Datasets aus einer Überwachungskamera, bei welcher mit einem Gesichtsdetektor die entsprechenden Frames detektiert und ausgerichtet wurden. Allerdings enthalten die Bilder teilweise nicht nur den Kopf, sondern auch andere Dinge im Vorder- oder Hintergrund und teilweise wird der eigentliche Kopf sogar verdeckt. Zudem sind nur Köpfe mit Vorder- oder Seitenansicht enthalten, weshalb ein trainierter Classifier nur auf solche Ansichten anspricht. Anders verhält sich ein Classifier mit dem UcoHead Dataset, welcher sehr stark auf Hinter- und Seitenansichten, aber auch gut auf Frontansichten von Köpfen reagiert. Hier macht sich der grosse Aufwand bezahlt, welcher betrieben wurde, um die Köpfe gleichmässig aus allen Blickwinkeln und bei unterschiedlichen Lichtbedingungen abzulichten. Weiter werden durch die guten Laborbedingungen Verdeckungen des Kopfes oder das Auftauchen anderer Objekte im Bild verhindert.

Der Vollständigkeit halber wird der Parameter `maxDepth` von 1 auf 2 gesetzt und der Classifier `cascade_lbp_head_130823.xml` trainiert. Dies funktioniert ohne Probleme, soll der Classifier jedoch verwendet werden, stürzt OpenCV mit der Fehlermeldung „Not a Stump“ ab. Dies deutet darauf hin, dass die Implementation nur Classifier mit Decision-Stumps unterstützt oder für die Verwendung tieferer Classifier eine andere OpenCV-Klasse verwendet werden müsste.

Der finale LBP-Classifier `cascade_lbp_head_130823_2.xml` wird mit den Parametern aus Abbildung 3.13 trainiert und liefert gute Ergebnisse. Allerdings muss erwähnt werden, dass nicht alle Personen gleich gut erkannt werden, insbesondere die Frontalansicht einiger Personen wird nur sehr selten detektiert.

Datum	23.08.2013
featureType	LBP
numPos	6000
numNeg	3000
precalcValBuffSize	1512
precalcIdxBufSize	1512
numStages	18
w	40
h	40
minHitRate	0.9999
maxFalseAlarmRate	0.25
weightTrimRate	0.95
maxDepth	1
mode	ALL
Boosting	GAB
Datasets	1 + 2

Abbildung 3.13: Parameter des finalen Classifiers cascade_lbp_head_130823_2.xml mit LBP-Features.

Um die Unterschiede von Haar-Features gegenüber den besprochenen LBP-Features zu betrachten, wird ein solcher Classifier im Test `cascade_haar_head_130809.xml` trainiert. Da dieser mit den vorgeschlagenen Standardwerten generiert wird, erreicht er keine gute Detektionsleistung. Dennoch lassen sich Vergleiche mit dem entsprechenden LBP-Classifier `cascade_lbp_head_130808.xml` anstellen. Als erstes ist die Trainingsdauer zu erwähnen, welche mit Haar-Features zwei Tage dauert, im Vergleich zu 20 Minuten bei LBP. Zusätzlich ist die Detektionsgeschwindigkeit beträchtlich schlechter. Bezüglich der Erkennungsqualität

schneidet der Haar-Classifier geringfügig besser ab, obwohl dieser eine sehr hohe FPR aufweist, die bei LBP-Features geringer ausfällt. Beachtet man die Parameter genauer, fällt vor allem die verwendete Grösse der Samples auf, welche in diesen Tests mit 20x20 Pixel gewählt wurde. Während die LBP-Features mit solch kleinen Samples Probleme haben, scheinen Haar-Features unter denselben Umständen bessere Ergebnisse zu erzielen. Tabelle 3.4 fasst die wichtigsten Merkmale beider Varianten kurz zusammen.

	LBP-Features	Haar-Features
Trainingsdauer	+	-
Detektionsgeschwindigkeit	+	-
True Positive Rate	-	+
False Positive Rate	+	-
Optimale Samplegrösse	40 Pixel	20 Pixel

Tabelle 3.4: Vergleich zwischen Classifiern mit LBP- und Haar-Features.

Einige Beispieldetektionen des finalen LBP-Classifiers im Live-Video finden sich im Anhang D.3.

3.2.6 Trainieren einer Soft Cascade

Ab der neusten OpenCV Version 2.9.0 befindet sich eine Klasse zum Trainieren einer Soft-Cascade in der API-Beschreibung des Frameworks. Bei der eigenen Kompilierung wird sogar das entsprechende Tool opencv_trainsoftcascade.exe erstellt und im Binary-Ordner abgelegt. Leider befindet sich dieser Teil von OpenCV noch stark in der Entwicklung und es existiert deshalb nur wenig Dokumentation zum Thema. Da Soft-Cascades aber eine interessante Möglichkeit für schnelle und flexible Classifier bieten, wird versucht einen solchen zu trainieren. Anders als bei den bisher vorgestellten Tools von OpenCV werden die Parameter nicht direkt beim Aufruf, sondern in einer zusätzlichen xml-Datei angegeben. Deshalb benötigt der Aufruf in Listing 3.9 nur einen Parameter.

```
opencv_trainsoftcascade.exe -config trainsoft_config.xml
pause
```

Listing 3.9: Batch-Datei (trainsoft.cmd) für das Trainieren von Soft-Classifiern.

Die eigentlichen Einstellungen werden in der angegebenen `trainsoft_config.xml` definiert und könnten in etwa wie in Listing 3.10 aussehen.

```
<trainPath>trainPath</trainPath>
<testPath>testPath</testPath>
<modelWinSize>modelWinSize</modelWinSize>
<offset>offset</offset>
<octaves>octaves</octaves>
<positives>positives</positives>
<negatives>negatives</negatives>
<btpNegatives>btpNegatives</btpNegatives>
<shrinkage>shrinkage</shrinkage>
<treeDepth>treeDepth</treeDepth>
<weaks>weaks</weaks>
<poolSize>poolSize</poolSize>
<cascadeName>cascadeName</cascadeName>
<outXmlPath>outXmlPath</outXmlPath>
<seed>seed</seed>
<featureType>featureType</featureType>
```

Listing 3.10: Vorlage einer möglichen config.xml.

Leider scheint diese Implementation noch nicht einsatzbereit zu sein, da ein Ausführen mit den gezeigten Einstellungen immer zu einem Abbruch führt. Die Ursache dafür ist, dass die exe-Datei die angegebene config-Datei nicht findet oder nicht korrekt einlesen kann.

4 Diskussion

Diese Masterarbeit befasst sich mit der Echtzeit-Detektion und -Erkennung von Personen in Videos mithilfe eines handelsüblichen PCs mit GPU.

Der zweite von drei Teilen (Projekt 8) beschäftigt sich mit drei der bekanntesten HSAs und analysiert einerseits dessen Fähigkeit, vorhandenen CPU-Code auf einer GPU zu beschleunigen. Andererseits wird die Flexibilität betrachtet, welche es ermöglicht, für GPUs optimierten Code einfach und effizient auf einer CPU auszuführen. Zusätzlich soll die Kompatibilität und die Handhabung der verwendeten Systeme bewertet werden.

Um die in Projekt 7 erstellte Software zur Personenerkennung weiterzuentwickeln, werden im zweiten Teil des Projekt 8 die notwendigen Theoriekenntnisse erarbeitet und ein Classifier zur Kopfdetektion trainiert. Dabei werden die Möglichkeiten von OpenCVs Machine-Learning-Funktionen analysiert und die bestmögliche Variante evaluiert. Zusätzlich soll eine automatisierte Testapplikation entwickelt werden, mit welcher statistische Tests an der Erkennungssoftware durchgeführt werden können.

Die komplette Aufgabenstellung ist in der Klärung in Anhang A ersichtlich.

4.1 Heterogene Systeme

Im ersten Teil dieser Arbeit wurden die drei folgenden HSAs ausführlich getestet: AMP, OpenCL und OpenACC. Dabei wurde einerseits Wert auf die mögliche Performance, andererseits auf die Handhabung gelegt. Während den Tests und manuellen Optimierungsversuchen der HSAs musste immer mehr in das Thema der Hardware- und Beschleunigungstechnologie eingestiegen werden. Deshalb beschäftigt sich dieser Teil verstärkt mit der allgemeinen CPU- und GPU-Hardware, der nVidia CUDA-Hardware, dem Pipelining durch SSE-Module und diversen weiteren Problematiken, welche im Kontext von HSAs auftauchen.

Bei der Auswertung der erstellten HSA-Messungen konnten deren Stärken und Schwächen ermittelt werden. Eine grosse Schwäche weisst der noch junge OpenACC-Standard auf, da dessen Implementationen nur Zwischenkompiler darstellen, welche nVidia-spezifischen CUDA-Code erzeugen. Nicht nur betreffend der Kompatibilität kann OpenACC nicht überzeugen, auch die erreichten Beschleunigungswerte kommen nicht an optimale Ergebnisse der anderen HSAs heran. Trotzdem besteht Hoffnung, dass die Fusion von CPU- und GPU-Beschleunigung in OpenMP 4.0 durch einfache Annotationen des Codes, zu einem Durchbruch führt.

Die Tests von OpenCL und Microsoft AMP liefern ganz andere Ergebnisse. Einerseits bieten beide eine parallele Ausführung für GPU-Code auf CPUs an, andererseits erreichen die GPU-Implementationen gute Beschleunigungswerte. Klarer Favorit ist jedoch keiner der beiden. Während AMP vor allem wegen seiner starken Abstraktion und schnell zu erlernenden Konzepte für einen einfachen Einstieg sorgt, bietet OpenCL die kürzeren Ausführungszeiten und höhere Konfigurierbarkeit. Allerdings muss darauf hingewiesen werden, dass AMP bereits mit der simpelsten Implementation eine gute Performance erreicht. Im Bereich CPU-Ausführung ist OpenCL jedoch absoluter Spitzenreiter und vermag mittels Vektorisierung teilweise sogar OpenMP zu übertreffen. Um mit Microsoft-Technologien dennoch eine akzeptable CPU-Parallelisierung zu erreichen, kann PPL sinnvoll mit AMP kombiniert werden.

Abschliessend wird für performancekritische Anwendungen OpenCL empfohlen. Im Bereich der Bildverarbeitung mittels OpenCV bietet sich dies neuerdings besonders an, da OpenCV die meisten Funktionen, die ursprünglich nur CPU- und CUDA-Varianten hatten, nach OpenCL portiert hat. Zudem kann bei eigenen Implementierungen von der Abstraktion des neuen ocl-Moduls profitiert werden.

4.2 Kopfdetektion

Die Kopfdetektion im zweiten Teil dieser Arbeit wurde mittels der Tools von OpenCV implementiert und getestet. Diese können mit einer eigenen Kompilation von OpenCV generiert werden. Nebenbei wurde OpenCV auf die neuste Version 2.9.0 aktualisiert, welches in Kombination mit CUDA 5.5 und Visual Studio 2012 endlich ein für AMP brauchbares Kompilat erzeugt.

Um Problematiken und Testergebnisse der Kopfdetektoren besser zu verstehen, fand als erstes eine vertiefte Auseinandersetzung mit den Theorien von Machine-Learning und Cascade-Classifiers statt. Die beiden Varianten für verwendete Features in solch einem Classifier erschienen zwar als wichtige Vergleichsmerkmale, LBP etablierte sich allerdings als viel praktikableren Ansatz, insbesondere wegen den erheblich kürzeren Trainingszeiten. Ein weiterer wichtiger Faktor ist die Wahl des Datasets. Die Wahl fiel schliesslich auf UcoHead, welches zu merklich besseren Classifiern führte, dank der gleichmässigen Bilder unter Laborbedingungen, gegenüber den Überwachungskamerabilder aus dem CoffeeBreak Dataset.

Der finale LBP-Classifier mit 18 Stufen erzielt gute, mit dem Gesichtsdetektor von OpenCV vergleichbare, Ergebnisse und kann so verwendet werden. Um der Schwäche beim Erkennen von Gesichtern entgegenzuwirken, kann er mit dem Gesichtsdetektor kombiniert werden. Um die fälschlicherweise detektierten Bereiche, wie z.B. in Abbildung D.2 h) und j) auszuschliessen, kann darin nach Augen oder Ohren gesucht werden.

Leider konnten keine präzisen statistischen Auswertungen der Classifier erstellt werden, da durch die ausführliche Bearbeitung des HSA-Themas keine Zeit mehr für die Implementation einer Testanwendung blieb. Um die Nutzbarkeit des Classifiers zu beweisen, ist eine solche Anwendung allerdings notwendig und könnte deshalb im Projekt 9 entwickelt werden.

4.3 Ausblick

Vorausschauend auf Projekt 9 bieten sich diverse interessante Themen an. Als erstes soll die automatische Testanwendung nochmals erwähnt werden, welche für die statistische Belegbarkeit der eigenen Detektionsresultate notwendig ist. Um die Detektion der Köpfe weiter zu verbessern und sich im Bereich von Cascade-Classifiern weiter auf dem Laufenden zu halten, lohnt sich die Beobachtung der Soft-Cascade-Funktionalität von OpenCV. Diese scheint aktuell nicht vollständig zu sein, könnte aber demnächst verfügbar werden. Ein weiterer Punkt betreffend OpenCV ist das neue ocl-Modul, welches eine interessante Variante zur Beschleunigung mittels OpenCL darstellt. Der neue OpenVX-Standard könnte in Zukunft ebenfalls eine wichtige Rolle im Bereich Computer-Vision auf Beschleunigern einnehmen, da er die Abstraktion von Hardware speziell für Bildbearbeitungsanwendungen zum Ziel hat.

Damit die Entwicklung der eigene Applikation weiter vorankommt, soll sich das Projekt 9 verstärkt mit TLD auseinandersetzten, welches für das Tracking der erkannten Köpfe zuständig sein soll. Dank diesem System soll es möglich werden, die Personen im Bild zu verfolgen und kontinuierlich neue Ansichten des Kopfes zu erhalten. Diese werden dann zur Wiedererkennung der Personen in der Datenbank verwendet und bei erfolgreichem Erkennungsvorgang zur Verbesserung des abgespeicherten Kopfmodells benötigt.

Abkürzungs- und Stichwortverzeichnis

Accelerated Massive Parallelism (AMP) 14
Area under an ROC Curve (AUC)........................46
Boosting..48, 49, 50, 52
Branch-Prediction.......................................5, 8, 9
Cache 4
Cascade Classifier 44
Compute Unified Device Architecture (CUDA).....5
Compute-Unit..3, 6, 9
Cumulative Sum...50
Data Locality 5
Decision Stumps 48, 52
Device 2
False Positive Rate (FPR)........................45, 46, 58
Fermi...6, 8, 23
GigaThread 6, 7
Haar-like Features 47, 52
Hard Cascade...51
Heterogene System Architektur (HSA) 2
Host...2
Installable Compiler Driver Loader (ICD Loader) 17
Integral Image..47, 48
Just in Time Compiling (JIT) 11, 39
Kernel...2, 6, 14, 17
Lambda 14
Latency-Hiding..8
Local Binary Pattern (LBP) 47, 52
Loop unrolling...11
Machine Learning 44
Multi-Scale Block Local Binary Pattern (MB-LBP) 47, 48
Multi-Scale Pyramid..52
OpenCV...13, 24, 52
OpenMP 20
Padding 15, 19, 39, 40
Parallel Pattern Library (PPL) 15, 39
Pipelining 5, 7
Rang 15
Receiver Operating Characteristics (ROC) 46, 50, 51
Recheneinheit.. 3
Rechenwerk 3
Rejection Cascades 49
ROC-Surface... 51
Row Major 12
Single Instruction Multiple Threads (SIMT) 9
Soft Cascades 50
SSE..3, 11
Steuerwerk.. 3
Streaming Multiprocessor (SM)......................... 6
Streaming Processor (SP)................................. 6
Streaming-Memory... 13
Supervised Learning.. 44
Terminologie... 8
Thread-Block... 6
Tiling.. 15
True Positive Rate (TPR) 45, 46, 58
Vektorisierung 11
Warp 7, 10
Warp-Scheduler 6, 7
Weak Classifier...............................46, 48, 50, 52
Window Size 52
Windows Advanced Rasterization Platform (WARP)......................................13, 14, 15, 39

Quellenverzeichnis

[1] S. Baxter, «Modern GPU,» [Online]. Available: http://web.archive.org/web/20111225004726/http://www.moderngpu.com/intro/intro.html. [Zugriff am 7. Juni 2013].

[2] nVidia, «Whitepaper Fermi Architecture,» [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. [Zugriff am 6. Juni 2013].

[3] nVidia, «Whitepaper Kepler Architecture,» [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. [Zugriff am 6. Juni 2013].

[4] nVidia, «CUDA C Programming Guide,» [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Zugriff am 10. Juni 2013].

[5] Wikipedia, «Instruction Pipeline,» [Online]. Available: http://en.wikipedia.org/wiki/Instruction_pipeline. [Zugriff am 10. Juni 2013].

[6] Y. Zhang, M. I. Sinclair und A. A. Chien, «Improving Performance Portability in OpenCL Programs,» University of Chicago, Chicago, 2013.

[7] M. Sabahi, «A Guide to Auto-vectorization with Intel® C++ Compilers,» Intel, [Online]. Available: http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers. [Zugriff am 12. Juni 2013].

[8] Hayes Technologies, «Technologie: SSE,» [Online]. Available: http://www.hayestechnologies.com/de/techsimd.htm. [Zugriff am 12. Juni 2013].

[9] J. Shen, J. Fang, H. Sips und A. L. Varbanescu, «Performance Gap between OpenMP and OpenCL for Multi-Core CPUs,» International Conference on Parallel Processing Workshops, 2012.

[10] B. George, «C++ AMP GPU debugging now available on Windows 7,» [Online]. Available: http://blogs.msdn.com/b/nativeconcurrency/archive/2013/01/25/c-amp-gpu-debugging-now-available-on-windows-7.aspx. [Zugriff am 20. März 2013].

[11] MSDN, «C++ AMP Overview,» Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/library/vstudio/hh265137.aspx. [Zugriff am 14. Februar 2013].

[12] D. Moth, «C++ AMP in a nutshell,» [Online]. Available: http://blogs.msdn.com/b/nativeconcurrency/archive/2011/09/13/c-amp-in-a-nutshell.aspx. [Zugriff am 20. März 2013].

[13] D. Callahan, «C++ AMP Performance Guidance,» [Online]. Available: http://blogs.msdn.com/b/nativeconcurrency/archive/2012/07/26/performance-guidance-for-c-amp.aspx. [Zugriff am 20. März 2013].

[14] D. Moth, «Microsoft's C++ AMP Unveiled,» [Online]. Available: http://www.drdobbs.com/windows/microsofts-c-amp-unveiled/231600761. [Zugriff am 20. März 2013].

[15] K. Gregory und A. Miller, C++ AMP, Accelerated Massive Parallelism with Microsoft Visual C++, Sebastopol, California 95472: O'Reilly Media, Inc., 2012.

[16] MSDN, «Windows Advanced Rasterization Platform (WARP) Guide,» [Online]. Available: http://msdn.microsoft.com/en-us/library/gg615082%28v=VS.85%29.aspx. [Zugriff am 21. Mai 2013].

[17] Intel, «SDK for OpenCL Applications,» [Online]. Available: http://software.intel.com/en-us/vcsource/tools/opencl-sdk. [Zugriff am 13. Mai 2013].

[18] nVidia, «CUDA Downloads,» [Online]. Available: https://developer.nvidia.com/cuda-downloads. [Zugriff am 13.

Mai 2013].

[19] Khronos, «OpenCL C++-Header 1.2,» [Online]. Available: http://www.khronos.org/registry/cl/api/1.2/cl.hpp. [Zugriff am 13. Mai 2013].

[20] nVidia, «Nsight Visual Studio Edition,» [Online]. Available: https://developer.nvidia.com/nvidia-nsight-visual-studio-edition. [Zugriff am 13. Mai 2013].

[21] Khronos, «OpenCL Overview,» [Online]. Available: http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf. [Zugriff am 13. Mai 2013].

[22] D. R. Berlich, «OpenCL-Tutorial, Teil 1: Einstieg in die OpenCL-Programmierung,» *Heise iX,* Nr. 3, pp. 138-143, 2013.

[23] Khronos, «OpenCL Quick Reference Card,» [Online]. Available: http://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf. [Zugriff am 13. Mai 2013].

[24] The Portland Group, «PGI Accelerator Trial,» [Online]. Available: https://www.pgroup.com/account/register.php?openacc_user. [Zugriff am 18. April 2013].

[25] nVidia, «CAPS Compiler Trial,» [Online]. Available: http://www.nvidia.de/object/openacc-gpu-directives-de.html. [Zugriff am 18. April 2013].

[26] R Tutorial, «Installing CUDA Toolkit 5.0 on Ubuntu 11.10 Linux,» [Online]. Available: http://www.r-tutor.com/gpu-computing/cuda-installation/cuda5.0-ubuntu. [Zugriff am 18. April 2013].

[27] CAPS Entreprise, «How to activate and generate your trial license,» [Online]. Available: http://kb.caps-entreprise.com/how-to-activate-and-generate-your-trial-license/. [Zugriff am 18. April 2013].

[28] CAPS Entreprise, «Runtime Error: hmpperr::DeviceError,» [Online]. Available: http://kb.caps-entreprise.com/i-get-this-runtime-error-message-hmpperrdeviceerror-what-cumoduleloaddataex-no-binary-for-gpu/. [Zugriff am 18. April 2013].

[29] OpenACC Corporation , «OpenACC,» [Online]. Available: http://openacc.org/. [Zugriff am 25. April 2013].

[30] R. Farber, «Dr. Dobbs: Easy GPU Parallelism with OpenACC,» [Online]. Available: http://www.drdobbs.com/parallel/easy-gpu-parallelism-with-openacc/240001776?pgno=1. [Zugriff am 25. April 2013].

[31] R. Farber, «Dr. Dobbs: Creating and Using Libraries with OpenACC,» [Online]. Available: http://www.drdobbs.com/parallel/creating-and-using-libraries-with-openac/240012502. [Zugriff am 25. April 2013].

[32] nVidia, «Profiling and Tuning OpenACC Code,» [Online]. Available: http://on-demand.gputechconf.com/gtc/2012/presentations/S0517B-Monday-Programming-GPUs-OpenACC.pdf. [Zugriff am 21. Mai 2013].

[33] D. Griffing, «Simplifying single-dimensional C++ AMP Code,» [Online]. Available: http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/24/simplifying-single-dimensional-c-amp-code.aspx. [Zugriff am 17. Juni 2013].

[34] TwoPointSevenOh, « C++ AMP, general 1D custom CPU fallback to PPL,» [Online]. Available: http://social.msdn.microsoft.com/Forums/en-US/parallelcppnative/thread/5bec9298-299d-42a5-a972-0464c2dc9e68/#adfeaebb-1793-4470-a5f9-1a478ffcd0ee. [Zugriff am 17. Juni 2013].

[35] Wikipedia, «Beurteilung eines Klassifikators,» [Online]. Available: https://de.wikipedia.org/wiki/Beurteilung_eines_Klassifikators#Anwendung_im_Information_Retrieval. [Zugriff am 19. August 2013].

[36] Wikipedia, «Receiver operating characteristic,» [Online]. Available: http://en.wikipedia.org/wiki/Receiver_operating_characteristic. [Zugriff am 21. August 2013].

[37] P. Viola und M. J. Jones, «Rapid Object Detection using a Boosted Cascade of Simple Features,» IEEE CVPR, 2001.

[38] R. Lienhart und J. Maydt, «An Extended Set of Haar-like Features for Rapid Object Detection,» IEEE ICIP, 2002.

[39] OpenCV, «Cascade Classification,» [Online]. Available:

http://docs.opencv.org/master/modules/objdetect/doc/cascade_classification.html. [Zugriff am 20. August 2013].

[40] P. Viola und M. Jones, «Robust Real-time Object Detection,» Second international Workshop on statistical and computional theories of Vision-Modeling, Learning, Computing and Sampling, Vancouver, Canada, 2001.

[41] T. Ojala, M. Pietikainen und M. Maenpaa, «A comparative study of texture measures with classification based on feature distributions,» Pattern Recognition 29, 1996.

[42] S. Liao, X. Zhu, Z. Lei, L. Zhang und S. Z. Li, «Learning Multi-scale Block Local Binary Patterns for Face Recognition,» International Conference on Biometrics (ICB), 2007.

[43] Y. Freund und R. E. Schapire, «A decision-theoretic generalization of on-line learning and an application,» Springer, Computational Learning Theory: Eurocolt, 1995.

[44] Wikipedia, «AdaBoost,» [Online]. Available: http://en.wikipedia.org/wiki/Adaboost. [Zugriff am 16. August 2013].

[45] C. M. Bishop, Pattern Recognition and Machine Learning, USA: Springer, 2009.

[46] L. Bourdev und J. Brandt, «Robust Object Detection via Soft Cascade,» IEEE CVPR, 2005.

[47] OpenCV, «Cascade Classifier Training,» [Online]. Available: http://docs.opencv.org/doc/user_guide/ug_traincascade.html. [Zugriff am 05. März 2013].

[48] N. Seo, «Tutorial: OpenCV haartraining,» [Online]. Available: http://note.sonots.com/SciSoftware/haartraining.html. [Zugriff am 05. März 2013].

[49] OpenCV, «Installation in Windows,» [Online]. Available: http://docs.opencv.org/trunk/doc/tutorials/introduction/windows_install/windows_install.html. [Zugriff am 6. August 2013].

[50] CUDA Tech News, «CUDA 5.5 + OpenCV 2.4.9 + Visual Studio 2012,» [Online]. Available: http://cudatechnews.blogspot.ch/2013/07/update-cuda-55-opencv-249-visual-studio.html. [Zugriff am 6. August 2013].

[51] AVA, «UcoHead: a data set for multi-view head estimation in low resolution images,» [Online]. Available: http://www.uco.es/investiga/grupos/ava/node/27. [Zugriff am 05. März 2013].

[52] R. Muñoz-Salinas, E. Yeguas, A. Saffiotti und R. Medina-Carnicer, «Multi-Camera Head Pose Estimation,» Machine Vision and Applications, Volume 23, Number 3, S.479-490, 2012.

[53] University of Verona - VIPS lab, «CoffeeBreak Head Orientation Dataset,» [Online]. Available: https://sites.google.com/site/diegotosato/ARCO/coffeebreak. [Zugriff am 23. August 2013].

[54] D. Tosato, M. Spera, M. Cristani und V. Murino, «Characterizing humans on Riemannian manifolds,» IEEE Transactions on Pattern Analysis and Machine Intelligence, 2012.

[55] B. George, «MSDN Blogs: Parallel Programming in Native Code,» Microsoft, [Online]. Available: http://blogs.msdn.com/b/nativeconcurrency/archive/2013/01/25/c-amp-gpu-debugging-now-available-on-windows-7.aspx. [Zugriff am 07. März 2013].

[56] Khronos Group, «OpenCL,» [Online]. Available: http://www.khronos.org/opencl/. [Zugriff am 07. März 2013].

[57] OpenMP ARB, «OpenMP API,» [Online]. Available: http://openmp.org/wp/. [Zugriff am 07. März 2013].

[58] nVidia, «OpenACC,» [Online]. Available: https://developer.nvidia.com/openacc. [Zugriff am 07. März 2013].

[59] nVidia, «OpenACC: Directives for GPUs,» [Online]. Available: https://developer.nvidia.com/content/openacc-directives-gpus. [Zugriff am 07. März 2013].

[60] nVidia, «OpenACC: Example (Part 1),» [Online]. Available: https://developer.nvidia.com/content/openacc-example-part-1. [Zugriff am 07. März 2013].

[61] nVidia, «OpenACC: Example (Part 2),» [Online]. Available: https://developer.nvidia.com/content/openacc-example-part-2. [Zugriff am 07. März 2013].

[62] nVidia, «OpenACC: Developing with GPUs,» [Online]. Available: http://www.nvidia.com/object/openacc-gpu-directives.html. [Zugriff am 07. März 2013].

[63] nVidia, «OpenACC: Six Ways to SAXPY,» [Online]. Available: https://developer.nvidia.com/content/six-ways-saxpy. [Zugriff am 07. März 2013].

[64] N. Dalal und B. Triggs, «Histogram of oriented Gradients for Human Detection,» 2005.

[65] G. Nebehay, «Robust Object Tracking Based on Tracking-Learning-Detection,» Fakultät für Informatik der Technischen Universität Wien, 2012.

[66] Z. Kalal, J. Matas und K. Mikolajczyk, «Online learning of robust object detectors during unstable tracking,» IEEE ICCV, Kyoto, 2009.

[67] Z. Kalal, J. Matas und K. Mikolajczyk, «P-N Learning: Bootstrapping Binary Classifiers by Structural Constraints,» IEEE CVPR, San Francisco, 2010.

[68] Z. Kalal, J. Matas und K. Mikolajczyk, «Face-TLD: Tracking-Learning-Detection applied to Faces,» IEEE ICIP, 2010.

[69] Z. Kalal, J. Matas und K. Mikolajczyk, «Forward-Backward Error: Automatic Detection of Tracking Failures,» IEEE ICPR, Istambul, 2010.

[70] Z. Kalal, «TLD,» [Online]. Available: http://info.ee.surrey.ac.uk/Personal/Z.Kalal/tld.html. [Zugriff am 07. März 2013].

[71] G. Nebehay, «OpenTLD,» [Online]. Available: http://gnebehay.github.com/OpenTLD/. [Zugriff am 07. März 2013].

[72] S. Stalder, H. Grabner und L. v. Gool., «Beyond semi-supervised tracking: Tracking,» IEEE International Conference on Computer Vision Workshops, 2009.

[73] OpenCV, «FaceRecognizer,» [Online]. Available: http://docs.opencv.org/modules/contrib/doc/facerec/index.html. [Zugriff am 07. März 2013].

[74] C. Lang, «Performance gap on small filters at convolution with OpenCL on CPU,» [Online]. Available: http://software.intel.com/en-us/comment/1739995. [Zugriff am 17. Juni 2013].

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig angefertigt habe. Alle Stellen, die wörtlich oder sinngemäss anderen Quellen entnommen sind, habe ich als solche gekennzeichnet.

Ort, Datum

Windisch, 30.08.2013

Unterschrift

Lang Christian

Anhang

A. Klärung **74**
1. Ausgangslage 74
2. Aufgabenstellung 74
3. Organisation 74
4. Rückblick 74
5. Ziele 75
6. Teilaufgaben 75
7. Produkt 82
8. Semesterplanung 82

B. nVidia Architektur **84**
1. Fermi Architektur 84
2. Kepler Architektur 86

C. Messreihen HSA **88**
1. Testbilder 88
2. Zusätzliche Grafiken AMP 89
3. Zusätzliche Grafiken OpenCL 91
4. Zusätzliche Grafiken HSA-Vergleich 92
5. Forum-Anfrage OpenCL 94

D. Kopfdetektion mit OpenCV **96**
1. CMake Konfigurations-Protokoll von OpenCV 2.9.0 Build 96
2. Bewertungstabelle trainierter Classifier 98
3. Beispielbilder echter Detektionen 98

A. Klärung

Dieser Anhang enthält die Klärung für das Projekt 8. Sie enthält die Aufgabenstellung, die Motivation und gewisse Vorschläge für das Vorgehen. Die Planung wurde durch die aktualisierte Version vom 31. Mai 2013 ersetzt.

1. Ausgangslage

Im Rahmen des gesamten Masterstudiums sollen Erfahrungen mit Parallelisierung und Auslagerung von Algorithmen für die Personenerkennung auf GPU gemacht werden. Dazu sollen Technologien zur massiven Parallelisierung von Computer Vision Aufgaben evaluiert und zudem Algorithmen und Vorgehen für die Personenerkennung in Videos gesucht, implementiert und parallelisiert werden.

2. Aufgabenstellung

Mit welchen Tools können alle verfügbaren Rechenkerne möglichst optimal eingesetzt werden? Welche Anforderungen werden dabei an den Programmierer gestellt? Diese Fragen sollen nun beantwortet und mit konkreten Beispielen aus der Videoverarbeitung untermauert werden. Die eigentliche Videoverarbeitung soll vorangetrieben werden, so dass aus den zu extrahierenden Personendaten mehrdimensionale Personenmodelle erstellt werden können, welche später für die Wiedererkennung eingesetzt werden sollen.

3. Organisation

Auftraggeber: Prof. Dr. Christoph Stamm
Institut für Mobile und Verteilte Systeme, FHNW

Student: Christian Lang
christian.lang1@students.fhnw.ch

4. Rückblick

Im Projekt 7 wurde der Grundstein für das weitere Vorgehen gelegt. Dazu gehört einerseits das Einarbeiten in die Problematik, andererseits der grundsätzliche Aufbau eines Videoverarbeitungssystems. Dieses kann Videostreams effizient verarbeiten und Personen darin durch Erkennung von Bewegungen detektieren. Wichtig sind auch die verwendeten Technologien, wie OpenCV oder OpenMP. Zudem wurden weitere Technologien gefunden, welche verwandte Vorgehensweisen verwenden, um Software durch sogenannte „Beschleuniger“ auf die gesamte Hardware eines PCs zu verteilen.

Da die Materie und die verwendeten Tools Neuland waren, wurde verhältnismässig viel Zeit in Fachgrundlagen investiert, was nun wettgemacht werden soll.

5. Ziele

Folgende Ziele sollen erreicht werden:

1. Einrichten, Testen und Analysieren der folgenden drei Technologien für heterogene Systeme: Microsoft AMP, OpenCL, OpenMP (OpenACC). Dies unter Verwendung eines Computervision-Problems, welches mit allen Systemen gelöst werden kann und für Performance-Messungen geeignet ist.
2. Auswahl geeigneter Algorithmen für die Detektion, das Tracking und das Wiedererkennen eines Kopfes im gefundenen Personenbereich des Bewegungsdetektors. Es soll die beste Variante gewählt werden, in Bezug auf Performance, Detektionsgenauigkeit und Erweiterbarkeit. Um diese Anforderungen zu erreichen, muss auf eine geeignete Modellierung der Personendaten Wert gelegt werden.
3. Entwickeln einer Testumgebung, welche automatisiert statistische Tests mit den aufgezeichneten Testvideos durchführen kann und dadurch eine Bewertung der Genauigkeit der verwendeten Algorithmen ermöglicht. Das Testwerkzeug soll zudem die manuelle Annotation der Referenz-Sequenzen unterstützten.

6. Teilaufgaben

Der gesamte Projektumfang beträgt 16 ECTS und somit 480 Arbeitsstunden. Die Arbeit gliedert sich in die folgenden Unterkapitel:

6.a) Technologien für Heterogene Systeme

Das Projekt 7 hat gezeigt, dass es schwierig ist, alle gewünschten Technologien und Bibliotheken miteinander zu verwenden. Besonders die Kombination vom Microsoft Visual Studio 2012 Compiler und nVidia CUDA gestaltet sich umständlich. Des Weiteren konnte ein Teil der Systeme nur am Rand betrachtet werden, welche nun im Detail getestet werden sollen.

Das Ziel dieses Teils ist den aktuellen Trend zu heterogene Systemarchitekturen (HSA) mit zu verfolgen und die drei vielversprechendsten zum Laufen zu bringen und ausführlich auf Performance und Usability zu testen. Diese werden in den folgenden Unterkapiteln kurz vorgestellt.

i) Microsoft AMP

Microsoft C++ Accelerated Massive Parallelism (AMP) [11] erweitert den C++11 Standard um eine Abstraktion des Multithreadings auf CPUs mithilfe von Tasks und ermöglicht weiter eine einfache Nutzung von sogenannten „Beschleunigern“, welche in den häufigsten Fällen GPUs sind. Die beste Tool-Unterstützung hat man in Visual Studio 2012, welches sich wieder mehr den C++ Entwicklern verschrieben hat. Es bietet ein Debugging von GPU-Code, welches ursprünglich nur unter Win8 möglich war, dank dem Update KB2670838 [55] aber nun auch unter Win7 unterstützt wird.

AMP setzt auf Microsoft DirectCompute auf, welches direkt auf Direct3D fähige Grafikkarten zugreifen kann. Da AMP ein offener Standard ist, können auch andere Hersteller eigene Compiler bauen, welche dann über einen anderen Weg, die GPU oder andere Beschleuniger ansprechen. Z.B. nVidia könnte einen AMP kompatiblen Compiler bauen, welcher dann über CUDA die GPU anspricht. Aktuell ist erst der Compiler von Microsoft verfügbar, mit welchem man jedoch auf alle gängigen Grafikbeschleuniger zugreifen kann. Diverse Blogs für einen guten Einstieg sind unter [12, 14] verfügbar.

ii) Khronos Group OpenCL

Die Open Computing Language (OpenCL) [56] Spezifikation, welche ursprünglich von Apple initiiert wurde, nun aber von der Khronos Group verwaltet und weiterentwickelt wird, stellt eine einheitliche Schnittstelle für heterogene Rechensysteme zur Verfügung. OpenCL ist eine Untermenge von ISO C99 mit Erweiterungen und ebenfalls ein offener Standard. Die Hardwareabstraktion erfolgt dadurch, dass alle Rechengeräte (CPUs, GPUs) als „Devices“ betrachtet werden, welchen Aufgaben in Form von „Kernels“ übergeben werden.

Anders als bei der AMP-Spezifikation, gibt es bereits mehrere Hersteller, die OpenCL implementieren. Diese sind: AMD, nVidia und Intel, wobei die aktuelle Spezifikation die Versionsnummer 1.2 trägt. Um in die OpenCL-Programmierung auf CUDA-Geräten einzusteigen, kann der übliche nVidia-Treiber in Kombination mit dem Nsight Entwicklungs-Addon für Visual Studio [20] verwendet werden.

iii) OpenMP 4.0 (OpenACC)

Die Open Multiprocessing (OpenMP) [57] Spezifikation wird vom OpenMP Architecture Review Board (OpenMP ARB) verwaltet und arbeitet mit sehr vielen Firmen wie AMD, Intel, IBM, Microsoft, nVidia, etc. zusammen. Sie bietet Compiler-Direktiven für C/C++ und Fortran an, welche den annotierten Code dann parallel auf der CPU laufen lassen. Die aktuelle Version trägt die Nummer 3.1, die Version 4.0 ist jedoch bezogen auf heterogene Systeme interessanter. Denn in dieser Version wird OpenMP mit OpenACC [29] fusioniert, wodurch dann auch OpenMP eine Abstraktion von Grafik- und anderen Beschleunigern bietet. OpenACC und OpenMP sind beides offene Spezifikationen und können bzw. werden von unterschiedlichen Herstellern implementiert.

Da OpenMP noch nicht in der Version 4.0 spezifiziert, geschweige denn implementiert ist, bietet OpenACC die einzige Möglichkeit die zukünftigen Funktionalitäten auszuprobieren. Da nVidia auch an der Entwicklung von OpenACC beteiligt war, bieten sie einige kleine Blogs [58, 59, 60, 61] und einen Testzugang zum PGI Accelerator [62]. Weiter gibt es einen Eintrag [63], welcher unterschiedliche Varianten aufzeigt, wie man auf CUDA-Geräten programmieren kann.

iv) Übersicht HSA

Die Abbildung A.1 zeigt den grundsätzlichen Schichten-Aufbau der drei betrachteten HSA Varianten. Man erkennt, dass sowohl GPUs als auch CPUs jeweils einen Treiber und ein Betriebssystem benötigen. Darüber setzten nun die erwähnten Technologien auf, um die Grafikbeschleuniger zu nutzen. CUDA ist ein Beispiel in Kombination mit nVidia-Beschleuniger und DirectCompute (oder Direc3D, oder DirectX) für eine Variante, welche mit allen DirectX-kompatiblen Geräten arbeiten kann. Die drei HSA-Varianten darüber können nun als High-Level Technologie verwendet werden. Wobei die bereits erwähnte nVidia-Variante von OpenCL auf CUDA aufsetzt oder AMP die DirectCompute Schicht verwendet. OpenACC und OpenMP sind noch getrennt visualisiert, da diese erst in naher Zukunft zu einem neuen Ganzen verwachsen werden. Jedoch ist die Verschmelzung durch die schwach hinterlegte Verbindung angedeutet. OpenMP liefert dabei die Parallelisierung auf CPUs und der OpenACC-Teil ermöglicht die Nutzung von GPUs und anderen Beschleunigern.

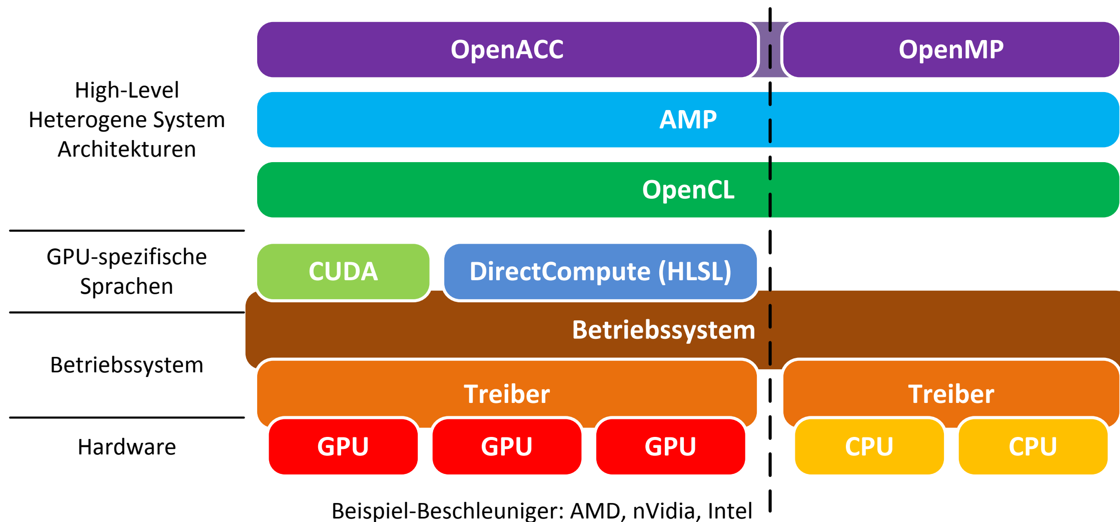


Abbildung A.1: Übersicht HSA.

v) Vorgehen beim Vergleichen

Um die unterschiedlichen HSA Technologien auf Performance und Usability zu untersuchen, muss ein geeignetes Test-Problem verwendet werden, welches einerseits einfach und kurz zu programmieren ist, andererseits möglichst viele Unterschiede der Systeme hervorhebt.

Dazu wird das Problem der Faltung, im Kontext der Bildverarbeitung, implementiert. Dieses Problem wurde gewählt, da es einerseits kurz und einfach zu parallelisieren ist, andererseits mit dem Erhöhen der Bildgrösse auf einfache Weise sehr rechenaufwendig gemacht werden kann, um sinnvolle Performance-Messungen durchführen zu können. Bezogen auf die Usability ist es jedoch schwieriger zu sagen, ob sich mit diesem Algorithmus alle wichtigen Unterschiede auffinden lassen. Für den Anfang sollte dies jedoch genügen. Falls trotzdem verfügbare Konzepte der Systeme vernachlässigt werden, könnten diese eventuell künstlich eingebaut oder das Problem erweitert werden.

Sollte Bedarf an einem anderen Test-Problem sein und Zeit zur Verfügung stehen, könnte zusätzlich eine Testanwendung aus dem nVidia SDK ausgewählt und für alle Systeme portiert werden.

6.b) Kopfdetektion und -wiedererkennung

Nachdem im Projekt 7 ein Algorithmus zur Detektion von Personen mithilfe von Bewegungserkennung entwickelt wurde, muss im nächsten Schritt die Kopfposition evaluiert werden. Zusätzlich soll ein geeignetes Kopfmodell für die Personenerkennung gewählt werden. Da die Erkennung eines Kopfes und die Zuweisung dieses Kopfes zu einer bereits bekannten Person ähnlich sind, müssen sich auch Gedanken zur Personenerkennung gemacht und das System soweit es geht darauf vorbereitet werden.

i) Kopfdetektion mit Classifier

Die bereits im Bericht vom Projekt 7 vorgeschlagene Detektion von Augen, Mund, Ohren etc. wäre eine mögliche Variante um den Kopf präzise zu erfassen. Auch die Erkennung der Ausrichtung des Kopfes wäre dadurch möglich. OpenCV bietet folgende Funktionen, um Objekte wie z.B. Ohren zu detektieren:

- `HOGDescriptor` nach Dalal et al. [64]
- `CascadeClassifier` nach Viola/Jones [37] und Lienhart et al. [38]

Wobei der `CascadeClassifier` speziell für zeitkritische Anwendungen ausgelegt wurde und somit besser geeignet ist. Zusätzlich bietet OpenCV bereits vortrainierte Modelle in Form von XML-Dateien dafür an. Darunter sind solche für Gesichter, Augen, Profilansicht des Kopfes oder auch die obere Hälfte des

menschlichen Körpers. Jedoch würden diese vermutlich nicht reichen, um den Kopf aus allen Richtungen erkennen zu können. Also müsste z.B. die Rückansicht eines Kopfes trainiert werden.

Da aus diesem Grund ein neuer Classifier trainiert werden muss, kann dieser auf eine allgemeinere Nutzung ausgelegt werden, nämlich auf das Erkennen eines Kopfes in irgendeiner Ausrichtung. Somit hätte man die Möglichkeit, alle Köpfe mit nur einem Classifier zu detektieren. Dies würde für die weiteren Schritte ausreichen, da die Information der Kopfausrichtung zur Personenerkennung nicht unbedingt notwendig ist. Falls nach dieser Stufe die genaue Ausrichtung des Kopfes trotzdem berechnet werden soll, um z.B. diese anzuzeigen oder für eine genauere Personenerkennung mitzuliefern, könnte diese wie vorgeschlagen mit der Detektion der Nase, Augen und Ohren erreicht werden. Alternativ könnten mehrere neue Classifier trainiert werden, welche dann jeweils Köpfe aus nur einer bestimmten Richtung erkennen.

Um einen neuen Classifier zu trainieren, bietet OpenCV bereits alle notwendigen Tools an. Eine Einführung und kurze Anleitung findet man in dessen Wiki [47]. Zusätzlich kann das ausführliche Tutorial von Seo [48] verwendet werden. Darin werden bereits diverse Quellen für geeignete Datasets erwähnt. Das Dataset von AVA [51], welches ca. 10'000 Kopfbilder von 10 Personen aus allen möglichen Blickwinkeln enthält, wäre ein weiteres, welches gut geeignet wäre. Ergänzend soll noch hervorgehoben werden, dass die von OpenCV mitgelieferten Classifier Haar-Feature-basiert sind, ab OpenCV 2.0 aber auch Classifier auf Basis von Local Binary Patterns (LBP) trainiert werden können. Diese Variante wird im Paper von Liao et al. [42] beschrieben und soll zusätzlich zum Ansatz mit Haar-Features getestet werden. Interessant sind LBPs weil ihnen ein allgemeineres Konzept unterliegt, welches eventuell besser geeignet ist und zusätzlich häufig zu kürzeren Erkennungszeiten führt. Beim Erzeugen neuer Classifier ist zu beachten, dass das Trainieren unter Umständen sehr lange dauern kann. Deshalb soll sehr früh damit begonnen werden, um während den Trainingsdurchläufen einer anderen Arbeit nachgehen zu können.

ii) „One-Shot“ Gesamtsystem

Mit den bis jetzt vorgestellten Methoden ergibt sich ein Gesamtsystem wie in Abbildung A.2 gezeigt. Hier wird für jedes Bild jeweils die gesamte Prozesskette komplett durchlaufen. Somit hat der Detektor genau genommen nur ein Bild zur Verfügung um die Person zu erkennen. Dies beginnt beim Detektieren von Personen, gefolgt vom neuen `HeadDetector`, welcher die Kopfpositionen erkennt. Danach könnten die erkannten Köpfe mit einer internen Datenbank von bekannten Personen verglichen und eine allfällig gefundene Identität ausgegeben werden. Als letzter Schritt würde der gefundene Bildausschnitt evaluiert und wenn möglich dem Modell der Person hinzugefügt werden.

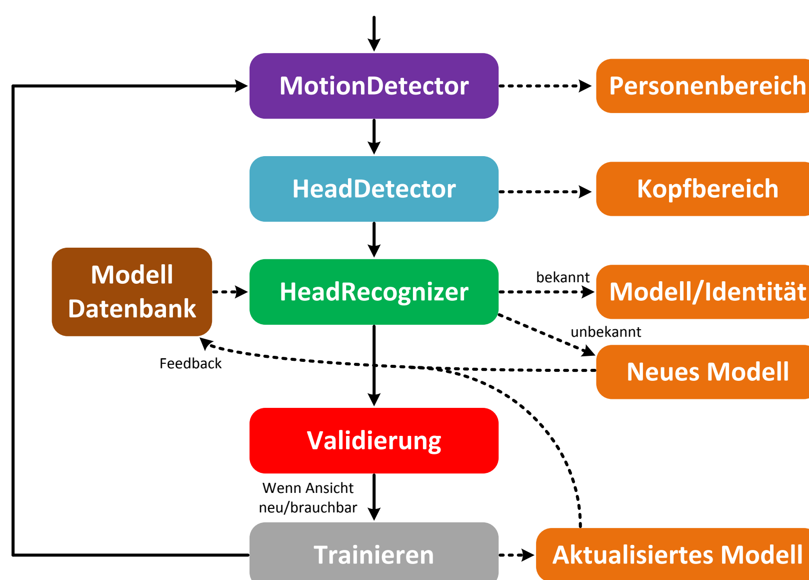


Abbildung A.2: „One-Shot“ Gesamtsystem.

Diese Variante lässt sich einfach verstehen und aufbauen, da die einzelnen Prozesse immer geordnet nacheinander ablaufen. Jedoch ergeben sich dadurch Probleme, da es zu sprunghaftem Verhalten kommen kann, wenn die Kopferkennung nicht sehr genau funktioniert. Es kann z.B. vorkommen, dass im ersten Bild die Person korrekt erkannt wird, im nächsten gar nicht und im dritten Bild eine andere Person detektiert wird. Dieses Problem könnte durch ein Tracking des Kopfes, welches im nächsten Kapitel beschrieben wird, vermindert werden. Um anfangs grundsätzliche Erfahrungen mit der Kopferkennung und Verwaltung des Kopfmodells zu sammeln, soll diese Variante im ersten Ansatz verwendet werden.

iii) Kopfmodell (Personenerkennung) und Tracking

Da nicht nur das Gesicht zur Personenerkennung verwendet werden soll, sondern auch z.B. der Hinterkopf, ergibt sich ein Problem betreffend der Zuordnung der einzelnen Ansichten zu einem gemeinsamen Datensatz. Das heisst, es reicht nicht aus, dass der `HeadDetector` im optimalen Fall alle Positionen der Köpfe detektieren kann. Angenommen die Person blickt im ersten Bild in die Richtung der Kamera und im zweiten Bild in die entgegengesetzte Richtung, kann trotzdem nicht erkannt werden, dass beide Kopfansichten zur selben Person gehören, da die Unterschiede der beiden Bildausschnitte zu gross sind.

Deshalb soll ein Tracking der gefundenen Köpfe stattfinden, womit sich ein drehender Kopf verfolgt und somit alle veränderten Ansichten aufgezeichnet werden können. Mit solchen Problemen beschäftigen sich die Arbeiten von Nebehay [65], bzw. die Papers von Kalal et al. [66, 67, 68, 69] zum Thema Tracking-Learning-Detection (TLD) [70], wobei Nebehay eine Open-Source [71] Implementation von TLD auf GitHub zur Verfügung stellt.

Diese Arbeiten zeigen eine robuste Art von Verfolgung eines bestimmten Objektes. Dazu wird ein Ansatz ähnlich dem Semi-Supervised Tracking [72] verwendet. Bei diesem wird gleichzeitig ein Tracking mit Feature-Points und ein Template Matching durchgeführt, wie es in Abbildung A.3 oben gezeigt ist. Die Resultate beider Vorgänge werden danach zusammengeführt und ausgewertet und in bestimmten Fällen das aktuelle Bild zu den Mustern des Template Matchings hinzugefügt. So wird einerseits ein Abdriften des Trackings durch das Template Matching verhindert, andererseits kann die Mustersammlung des Template Matchings durch gute Resultate des Trackings mit neuen Ansichten des Objektes ergänzt werden. Die Betonung liegt hier auf „gute Resultate“, denn die Sammlung der Bilder soll nicht unkontrolliert wachsen, sondern nur Bilder mit relevanten, neuen Ansichten verwenden. Auch sollen nur solche gespeichert werden, welche mit einer genügend hohen Genauigkeit erkannt worden sind. Um das Trackingresultat zu bewerten, wird die ebenfalls von Kalal beschriebene Methode des Forward-Backward Error [69] verwendet.

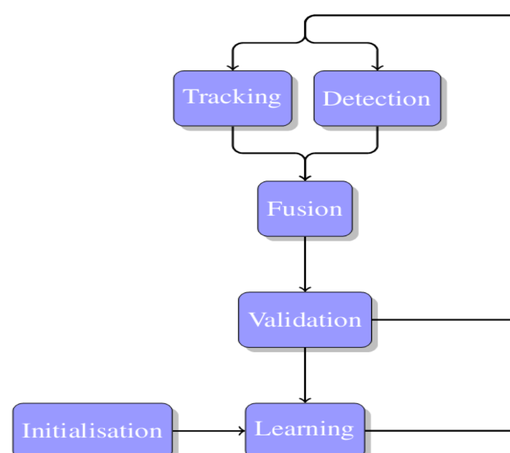


Abbildung A.3: Prinzip OpenTLD, Bild aus [65].

Bei der Arbeit von Nebehay [65] besteht der `Detector` zwar nicht nur aus einem Template Matching, jedoch ist dies der interessante Bestandteil, da dieses Template Matching bereits ein Modell des Objektes erstellt. Dieses besteht aus einer undefinierten Anzahl von normalisierten Bildausschnitten des sichtbaren Objekts und Bildern, auf welchen das Objekt nicht zu sehen ist. Diese beiden Arten von Muster werden als

„Positive Patch" und „Negative Patch" bezeichnet und können für das Trainieren einer Personenerkennung verwendet werden.

Um nun ein Personenmodell zu erstellen, welches die Person aber nicht nur mit den aktuellen Bedingungen erkennen kann, sollen diese „Patches" verwendet werden, um ein flexibleres Modell zu generieren. Dazu soll der `FaceRecognizer` [73] zum Einsatz kommen, welcher seit OpenCV 2.4 verfügbar ist. Diese Klasse bietet die Verwendung von folgenden Gesichtserkennungsmodellen an:

- Eigenfaces
- Fisherfaces
- Local Binary Patterns Histograms

Wie die Namen schon andeuten, sind Eigen- und Fisherfaces nur für die Erkennung von Gesichtern geeignet. OpenCV bietet jedoch auch hier das bereits bei der Kopfdetektion erwähnte System von LBP an. Dieses arbeitet mit einem Histogramm von Local Binary Patterns (LBPH) und ist allgemeiner verwendbar, da es keine Annahmen über die Eigenschaften von Gesichtern beinhaltet und nicht skalierungsabhängig ist. Zudem bietet nur diese Implementation die Möglichkeit, ein bereits trainiertes Modell mit einer neuen Ansicht zu erweitern und somit das Modell zu verbessern, ohne es komplett neu zu trainieren.

iv) „Continuous Tracking" Gesamtsystem

Mit den zusätzlichen Ansätzen des Kopfmodells und des Trackings, ergibt sich eine komplexere Struktur des Systems, welche in Abbildung A.4 gezeigt ist. In diesem wird nach der ersten Erkennung der Kopfbereiche das Tracking verwendet, um diese Bereiche zu verfolgen. Der jeweils aktualisierte Kopfbereich und die „Patches" können nun zum Erweitern des Modells verwendet werden. Danach wird der Kopfbereich vorläufig nur noch mithilfe des Trackings aktualisiert. Dadurch ist garantiert, dass falls der Kopfbereich verfolgt werden kann, sich immer noch dieselbe Person darin befindet. So sollte es möglich sein, auch sehr unterschiedliche Varianten der Kopfansicht der korrekten Person zuzuordnen.

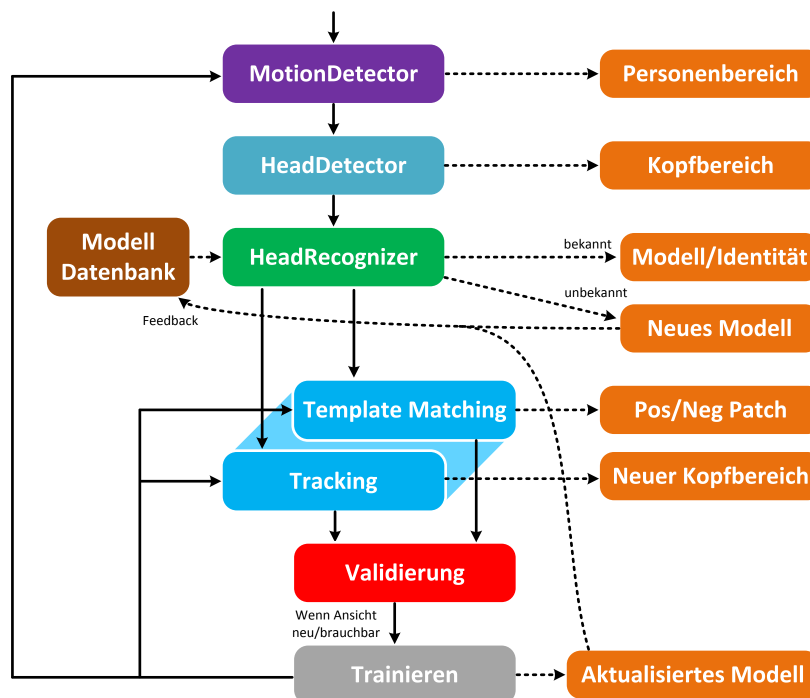


Abbildung A.4: „Continuous Tracking" Gesamtsystem.

Damit aber auch neu im Bild erscheinende Personen detektiert werden, muss in regelmässigen Abständen trotzdem ein Gesamtdurchlauf gestartet werden. Bereiche die dann gleichzeitig vom Tracking und dem `HeadDetector` gefunden werden, sollen weiterhin der vorherigen Person zugeordnet werden.

Da das neue System aufwendiger und komplexer ist, sollen beim `HeadRecognizer` nur die Grundvoraussetzungen gelegt werden. Umso mehr Aufmerksamkeit soll jedoch das Tracking erhalten, da es ein zentraler Bestandteil des ganzen Systems ist.

v) Eingabebilderskalierung

Um die Kopfdetektion und Personenerkennung zu beschleunigen, würde es Sinn machen, die Eingabebilder jeweils auf eine einheitliche Grösse zu skalieren. Somit sollen die gefundenen Personenbereiche vor der Kopfdetektion auf eine Grösse skaliert werden, welche so gewählt wird, dass mögliche Köpfe darin etwa der minimal erkennbaren Pixelgrösse entsprechen. Dasselbe vor der Personenerkennung. Die minimale Grösse hängt jeweils von der Grösse der Templates und des verwendeten Algorithmus/Modell ab. Zusätzlich kann so ein von der Skalierung abhängiger Algorithmus, z.B. für die Personenerkennung, trotzdem verwendet werden.

6.c) Vorgehen

Die im Kapitel 6.b) beschriebenen Methoden sollen zu dem System in Abbildung A.4 zusammengebaut werden. Dieses wird zwar bereits für die Erkennung von bekannten Personen vorbereitet, wie die Zieldefinition (Kapitel 0) jedoch verdeutlicht, soll der Schwerpunkt der Arbeit auf dem Erkennen und Verfolgen von Köpfen, sowie auf dem Personenmodell liegen. Deshalb werden folgende Arbeiten im Projekt 8 geplant:

1. Bauen eines `HeadDetectors` mithilfe von trainierten Classifiern. Evaluieren der beiden Varianten: Haar-Features und LBP-Features aus OpenCV
2. Vorbereiten eines einfachen `HeadRecognizer` (z.B. als Interface mit Stub-Implementierung)
3. Aufbauen eines Trackings mit Template Matching und dazugehörigem Modell für die Pos./Neg. Patchs unter Zuhilfenahme des unter GPL stehenden OpenTLD

Ergänzend ist zu erwähnen, dass die komplette Systemarchitektur auch im Fokus des GPGPU-Ansatzes steht und für die Einbeziehung von HSA-Technologien vorbereitet werden muss.

6.d) Testumgebung

Um das entwickelte System testen und bewerten zu können, soll das Programm Testfunktionen enthalten, welche nummerische Werte über Erfolgsraten der Detektion liefern. Damit kann einerseits die Qualität der Erkennung, aber auch der Einfluss der unterschiedlichen Testdaten, gemessen werden. Es sollen folgende Faktoren gemessen werden:

- Genauigkeit der erkannten Personenbereiche
- Genauigkeit der erkannten Kopfbereiche
- Genauigkeit der erkannten Personen

Damit die automatischen Tests überhaupt durchgeführt werden können, müssen die Testdaten mit den korrekten Werten annotiert werden. Somit sollen auftretende Ereignisse dem jeweiligen Frame zugeordnet werden können und beim Testen der Applikation die detektierten Ereignisse mit den annotierten verglichen werden. Um die manuelle Annotation zu unterstützen, soll ein Tool entwickelt werden, welches es erlaubt, eine Testvideo langsam laufen zu lassen, bei einem Ereignis zu stoppen und dann das gewünschte Tag zu dieser Frame-Nummer hinzuzufügen. Es sollen folgende Ereignisse annotiert werden:

- Person betritt Bildbereich
- Person verlässt Bildbereich
- Kopf der Person wird verdeckt
- Kopf der Person wird sichtbar

Bei allen Tags soll der definierte Name der Person und die ungefähre Position mitgegeben werden. Es soll eingestellt werden können, wie stark der gefundene Bereich vom getagten Bereich abweichen darf. Bei Bedarf sollen später weitere Tags definiert werden können.

Die Applikation soll ein geeignetes GUI erhalten und alle Funktionen (Live-Erkennung, Debug-Modi, Annotations-Werkzeuge und Test-Modi) vereinen. Am Ende des Projekts sollen diese Funktionen benutzt werden, um die Testdaten zu annotieren und eine Testreihe durchzuführen, welche die Qualität der Kopf-Detektion bewertet.

6.e) Weitere Aufgaben / Informationen

Die folgenden Themen werden in dieser Arbeit nur am Rande behandelt oder zeigen eventuelle Alternativen zu dem behandelten Vorgehen.

- OpenCV mit GPU-Unterstützung (CUDA) mit dem Microsoft vc11-Compiler kompilieren
- Das Personenmodell soll möglichst einfach erweiterbar (z.B. auf 3D-Modelle) gestaltet werden
- ArrayFire: Framework für Parallelisierung und GPGPU-Processing
- Prinzip der Vektorisierung von Schleifen: SSE etc.
- Active Appearance Model

6.f) Ausblick

Projekt 9 bildet den Abschluss der gesamten Arbeit um die situationsunabhängige Personenerkennung. Dabei sollen das bis dahin entwickelte System für die im Projekt 8 als beste evaluierte HSA Technologie angepasst und somit beschleunigt werden. Zudem sollen die gemachten Erfahrungen mit Verfolgung, Personenmodell und Erkennung genutzt werden, um das System zu verbessern und die noch fehlende Personenerkennung zu implementieren.

Die in Echtzeit laufende und selbständig lernende Personenerkennung ist das Ziel der Master-Thesis.

7. Produkt

Als Produkt des Projekts 8 ist die Dokumentation und der geschrieben Code zu betrachten. Während sich die Dokumentation mit der Analyse der getesteten Technologien für Heterogenen Systeme und den Problematiken der Modelfindung und der Kopf-Verfolgung beschäftigt, soll der Code praktische Vergleichsbeispiele für das Testen der Technologien und die Implementierung der Kopfdetektion und das Tracking liefern. Zudem sollen die erwähnten Testwerkzeuge implementiert und eine Testreihe durchgeführt werden.

8. Semesterplanung

Das Semester ist wie in Abbildung A.5 gezeigt verplant. Da im ersten Semester wegen den 6 Unterrichtsmodulen weniger Zeit für das Projekt blieb, werden diese ungefähr 60 Stunden nun nachgeholt.

Woche 14 und 21 beinhalten Feiertage oder anderweitige Ausfälle. Die Woche 28 wird für die Vorbereitung auf das Blockmodul in Woche 29 verwendet. Woche 27 ist für Ferien eingeplant.

Aktuelle Position bei Planung

Semesterwoche		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15				MSP			BM						
Kalenderwoche		8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
Thema	**MS**																												
Analyse / Verbessern P7	**16**	16																											
Klärung	**60**	8	24	24													4												
Heterogene Systeme	**200**																												
Einrichten	96				24	8	24		24	8	8																		
Testen	72					16				16	16			24															
CUDA	32																32												
Umbau auf C++11	0																												
Kopfdetektion	**208**																												
HeadDetector	112																	40	16					40	16				
HeadRecognizer	0																												
Tracking (TLD)	96																								24	40	32		
Testumgebung	**0**																												
Annotationstool	0																												
Testtool	0																												
Tests durchführen	0																												
Dokumentation	**180**											24	24		16	24	4										8	40	40
Total:	**664**																												

Klärung

Projektabgabe

Abbildung A.5: Projektplanung

B. nVidia Architektur

Dieser Anhang enthält zusätzliche Daten und Abbildungen in Bezug auf die vorgestellte nVidia Fermi und Kepler Architektur. Die Informationen dazu stammen hauptsächlich aus [2, 3, 4].

Architektur	CUDA Cores	Warp Schedulers	Dispatch Units	SFUs	Shared Memory	L1 Cache	L2 Cache
G80 / G92	8	1	1	2	16 kB	Keiner	Keiner
GT200 (Tesla)	8	1	1	2	16 kB	Keiner	Keiner
GF100, GF104 (Fermi)	32/48	2	2	4	16 / 48 kB	16 / 48 kB	768 kB
GK104, GK110 (Kepler)	192	4	8	32	16 / 32 / 48 kB	16 / 32 / 48 kB	1536 kB

Tabelle B.1: Detailangaben zu den CUDA-Architekturen. Werte sind immer pro SM / SMX, wobei dies Richtwerte für die jeweilige Architektur sind und nicht für alle Grafikkarten dieser Serie exakt übereinstimmen müssen.

1. Fermi Architektur

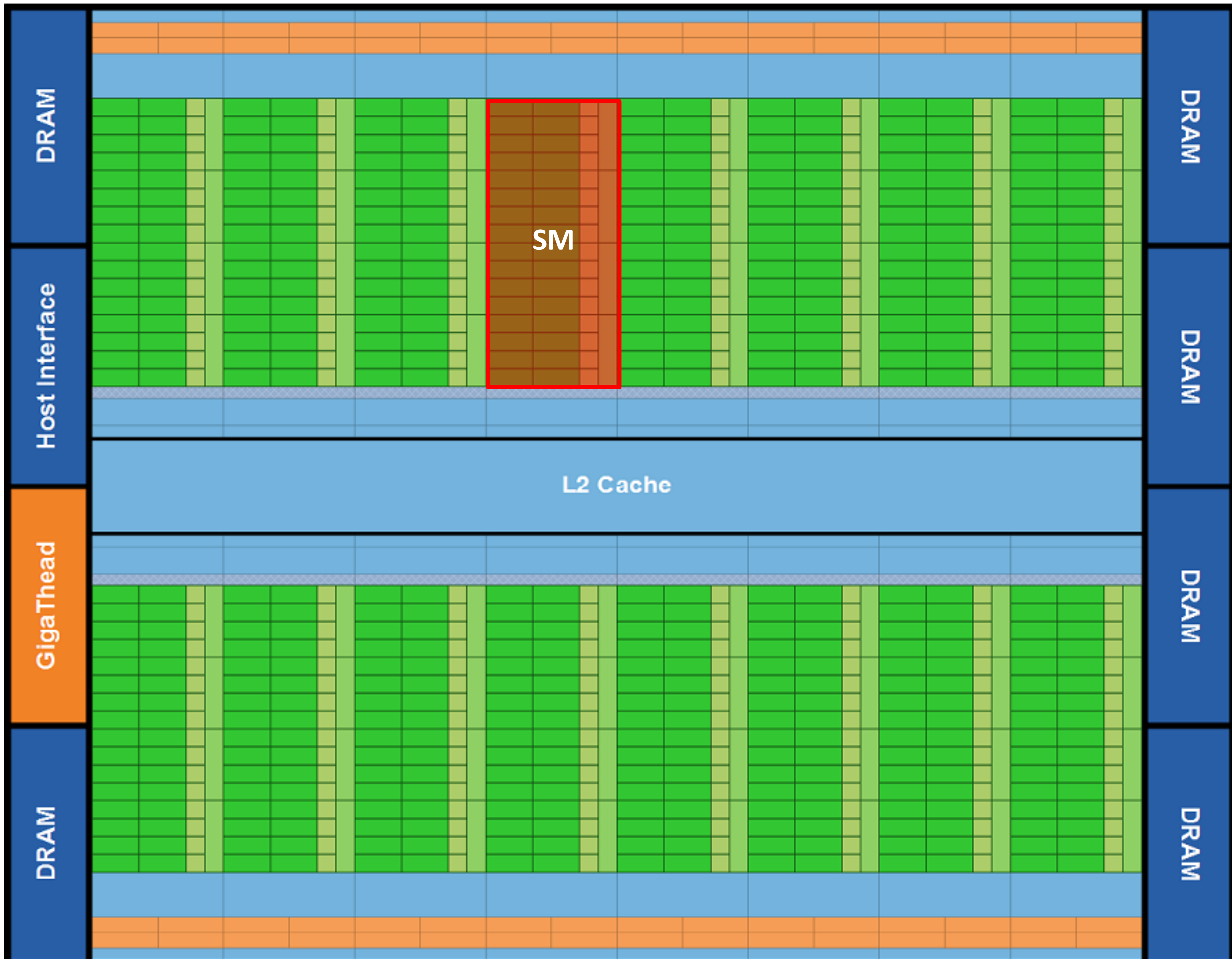


Abbildung B.1: Gesamtübersicht über einen nVidia Chip mit Fermi Architektur. Bild aus [2].

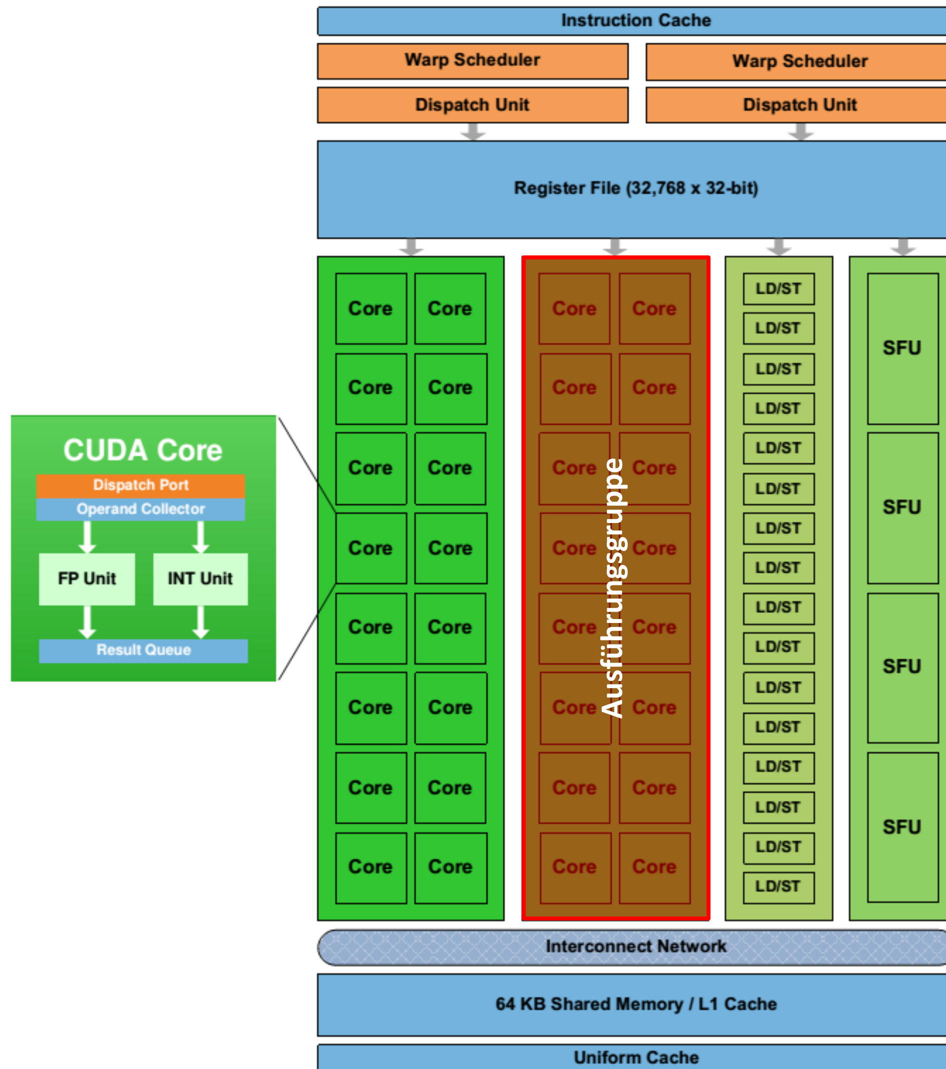


Abbildung B.2: Aufbau eines Streaming Multiprocessors der Fermi Architektur. Bild aus [2].

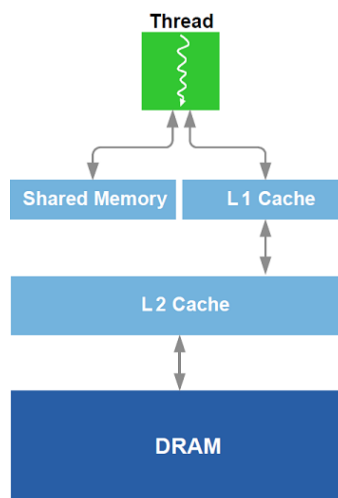


Abbildung B.3: Speicheranbindung eines einzelnen Threads unter Fermi. Bild aus [2].

2. Kepler Architektur

Abbildung B.4: Gesamtübersicht über einen nVidia Chip mit Kepler Architektur. Bild aus [3].

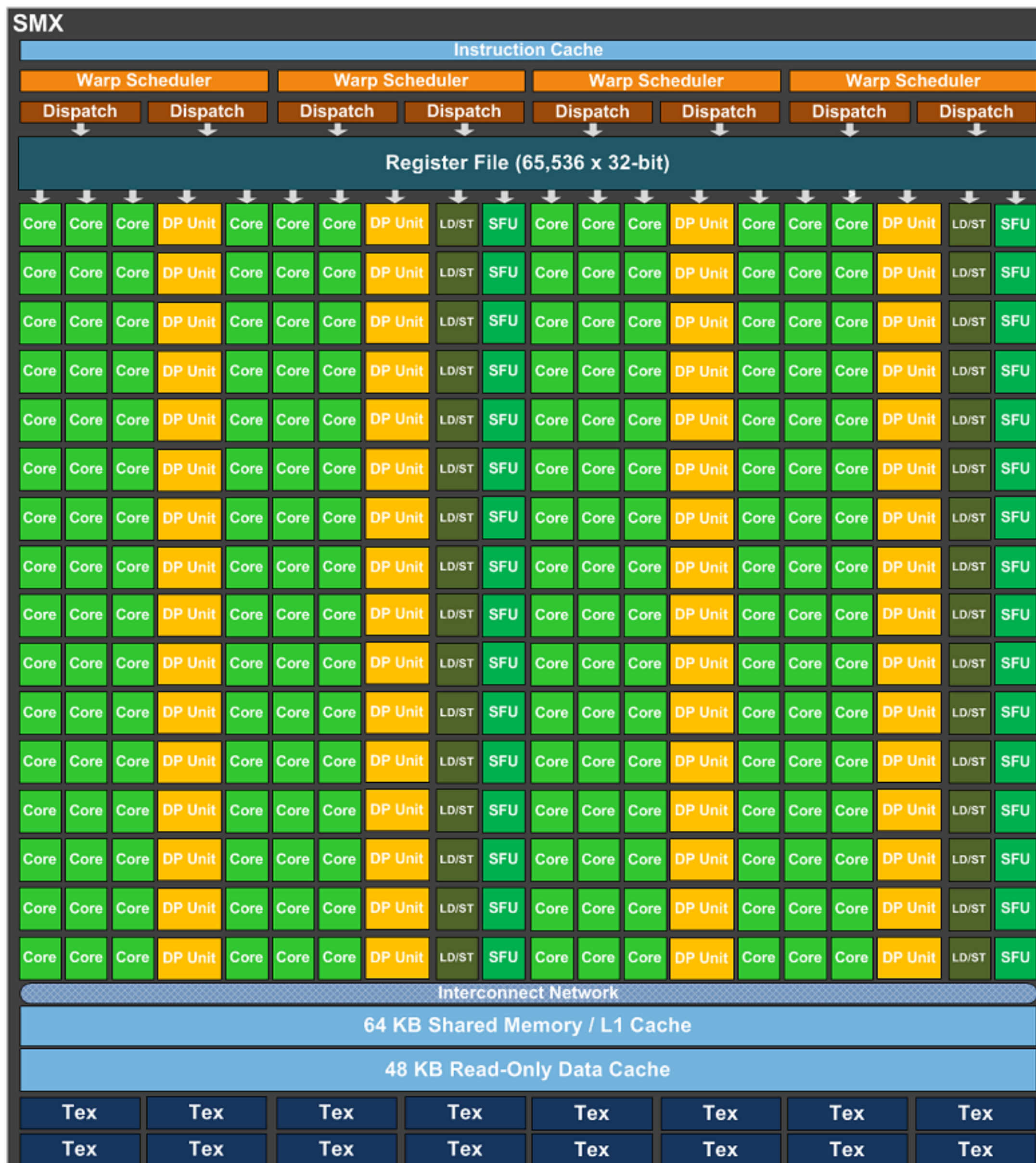


Abbildung B.5: Aufbau eines Streaming Multiprocessors der Kepler Architektur. Bild aus [3].

C. Messreihen HSA

Dieser Anhang enthält die für die Testreihen verwendeten Eingangsbilder und zusätzliche Diagramme bezogen auf die HSA-Messreihen. Alle Darstellungen finden sich in den originalen Excel-Dateien unter „Bericht\Resultate“.

1. Testbilder

kleines Bild: 128x128 Pixel

`lena.bmp`

mittleres Bild: 591x591 Pixel

`schweizerkreuz.bmp`

grosses Bild: 2272x1704 Pixel

`quadbike.bmp`

2. Zusätzliche Grafiken AMP

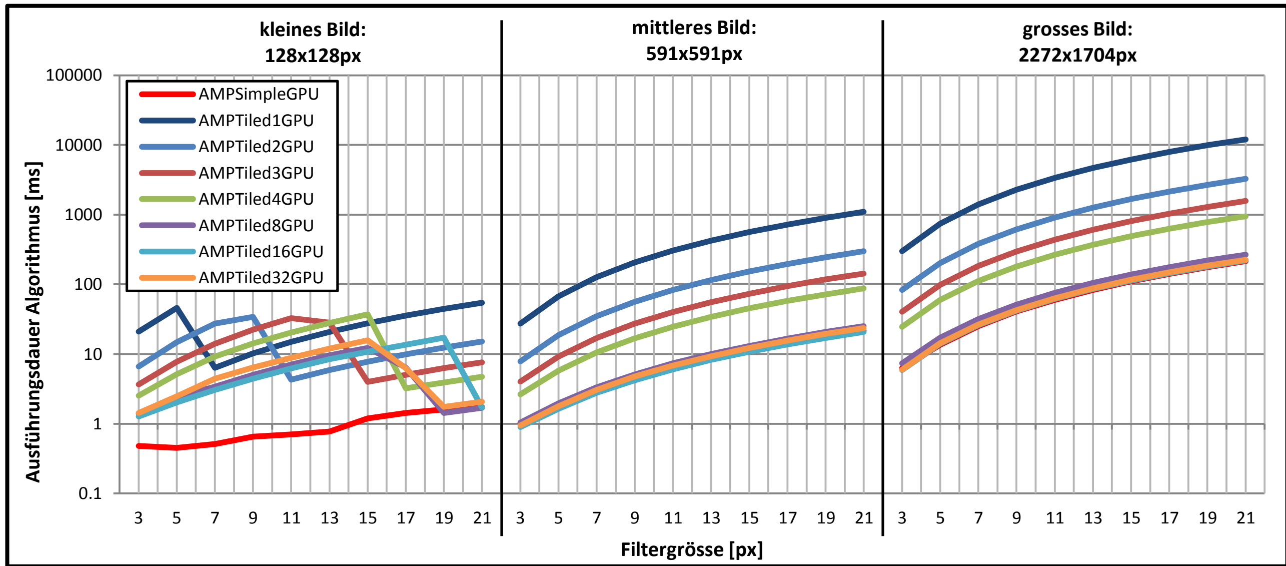


Abbildung C.1: Zeitaufwand des Algorithmus aller AMP-Varianten auf GPU.

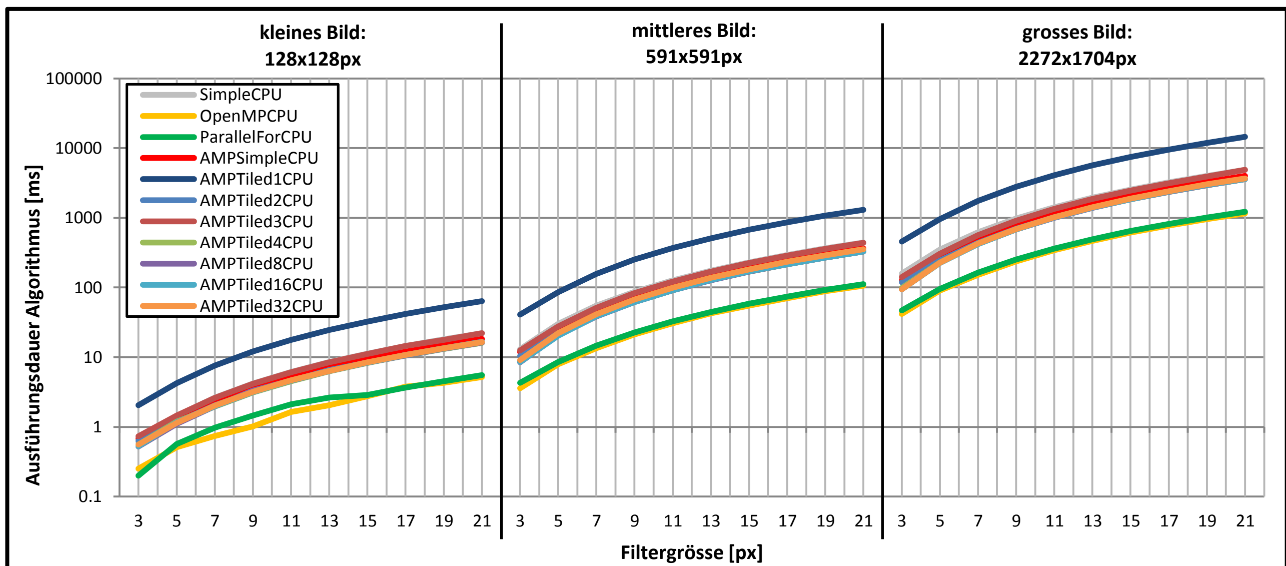


Abbildung C.2: Zeitaufwand des Algorithmus aller AMP-Varianten auf CPU.

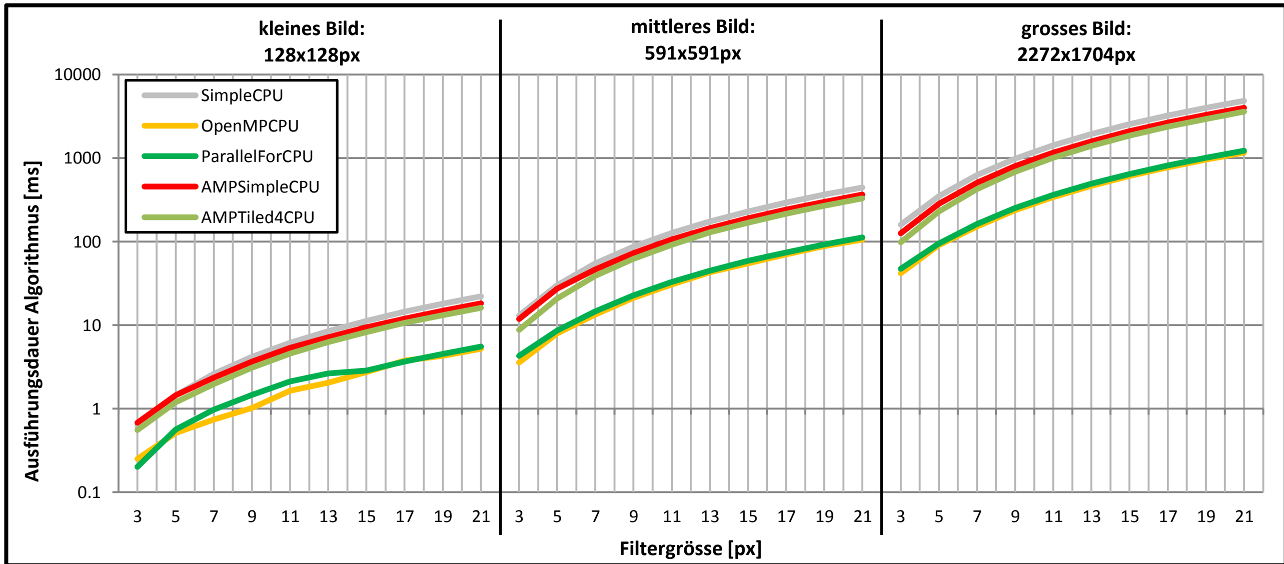


Abbildung C.3: Zeitaufwand des Algorithmus der besten AMP-Varianten auf CPU.

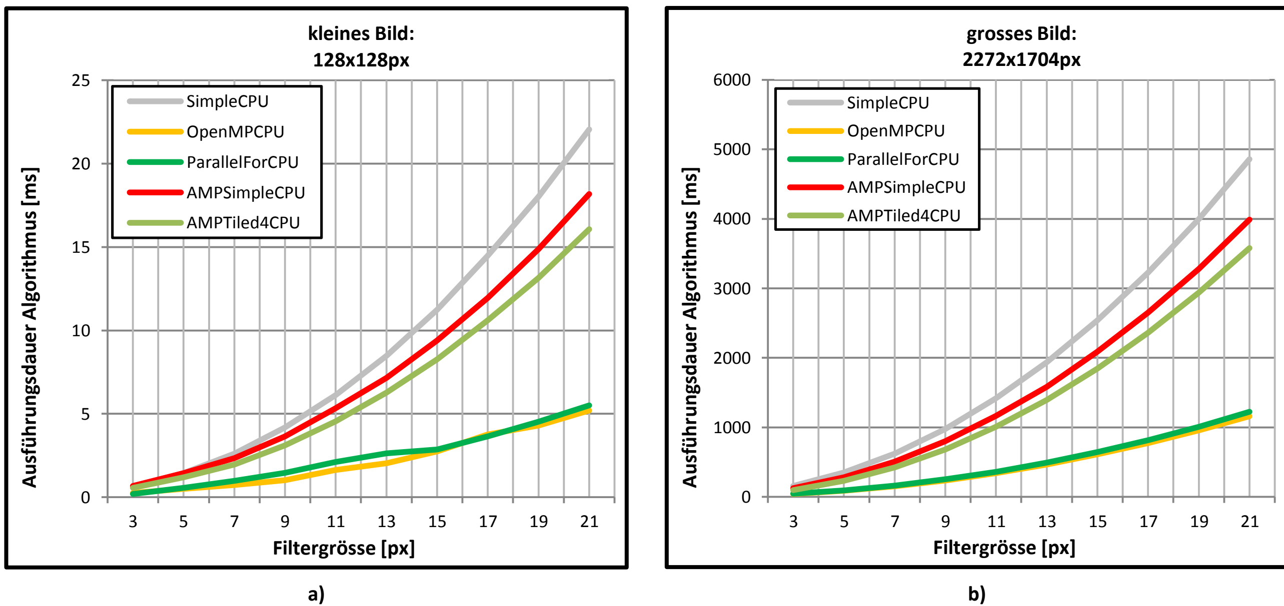


Abbildung C.4: Zeitaufwand des Algorithmus der besten AMP-Varianten auf CPU im kleinen und grossen Problembereich.

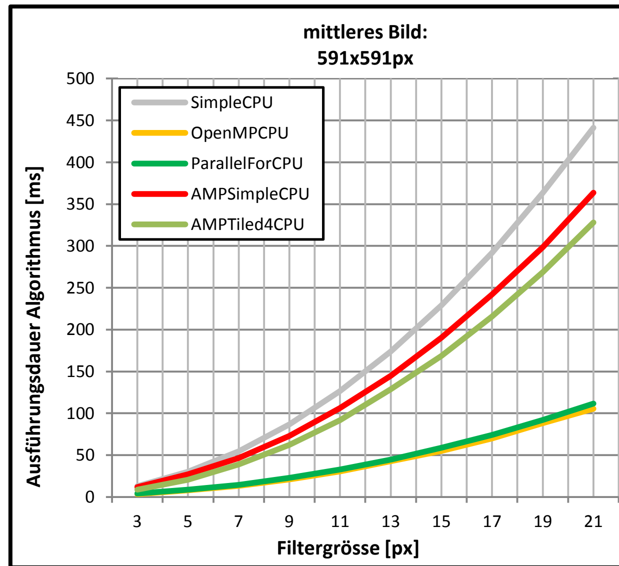


Abbildung C.5: Zeitaufwand des Algorithmus der besten AMP-Varianten auf CPU im mittleren Problembereich.

3. Zusätzliche Grafiken OpenCL

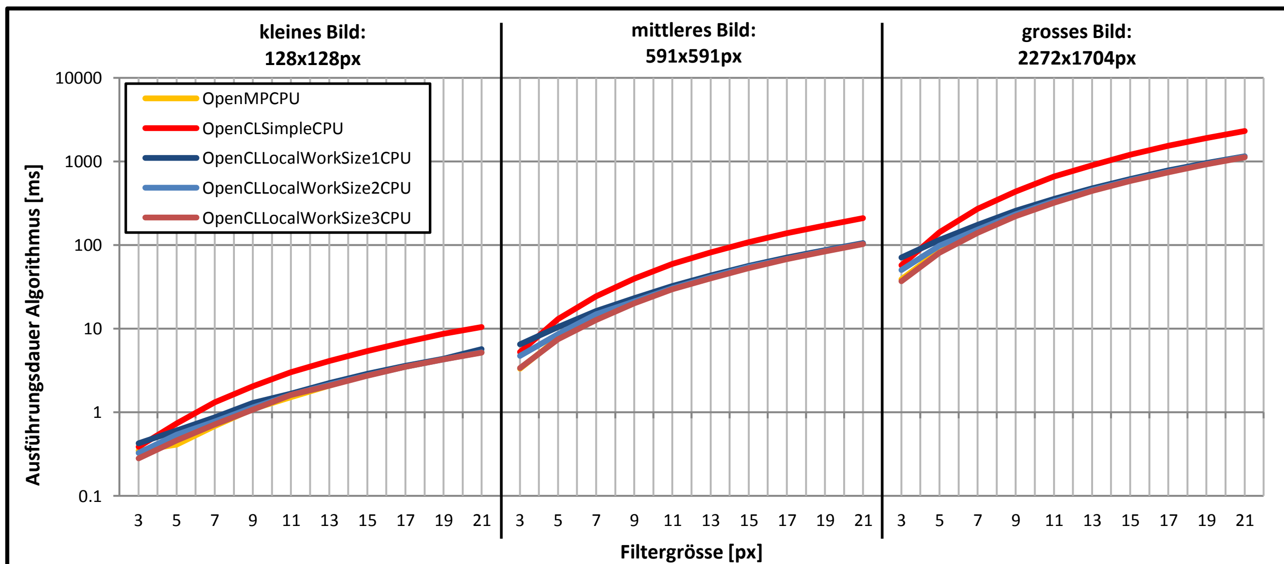


Abbildung C.6: Zeitaufwand des Algorithmus der relevanten OpenCL-Varianten mit Auto-Vektorisierer auf CPU.

4. Zusätzliche Grafiken HSA-Vergleich

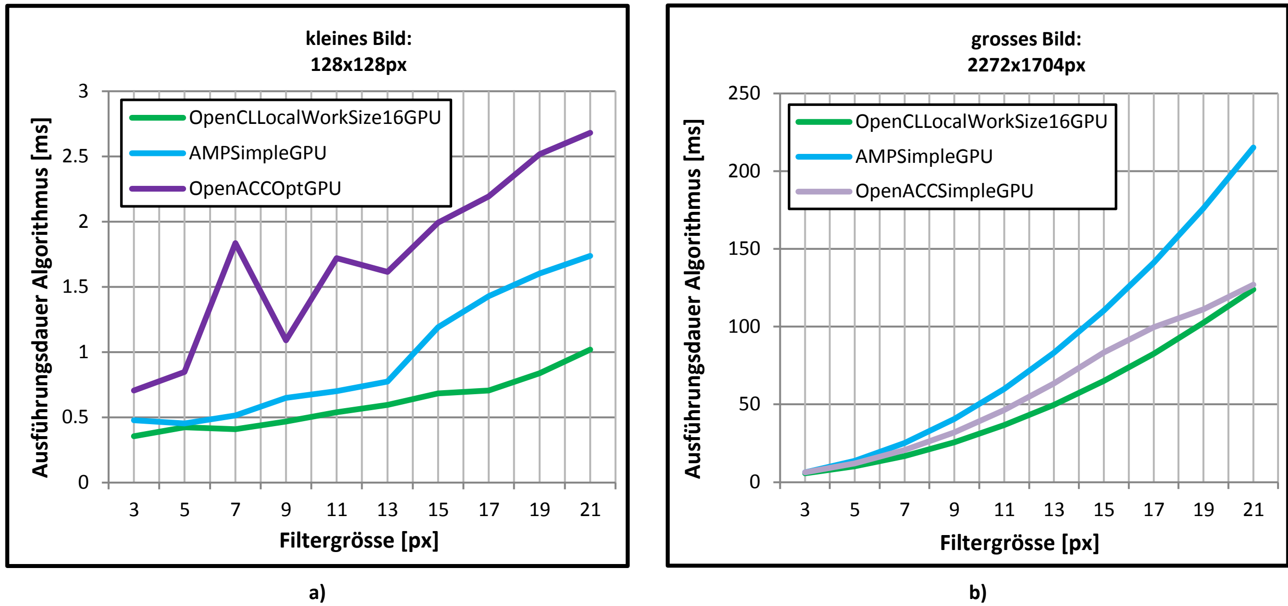


Abbildung C.7: Ausführungsdauer des Algorithmus auf der GPU der jeweils besten Messreihe. a) Kleine Problemgrösse und b) Grosse Problemgrösse mit jeweils bestem Ergebnis der optimierten OpenCL-Variante.

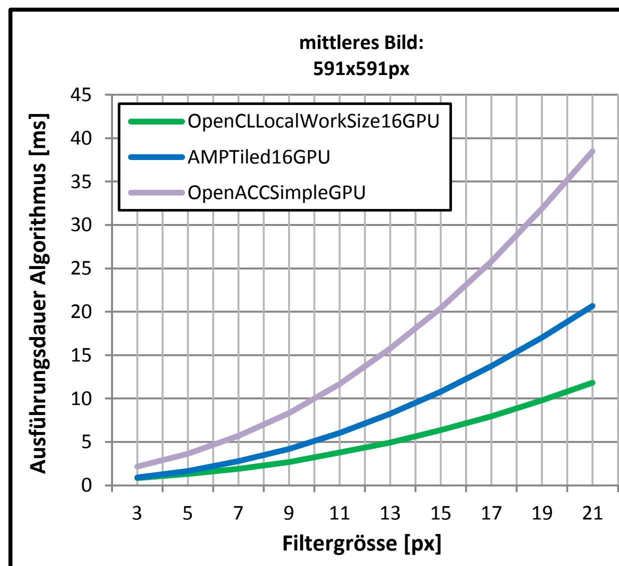


Abbildung C.8: Ausführungsdauer des Algorithmus auf der GPU der jeweils besten Messreihe. Mittlere Problemgrösse mit bestem Ergebnis der optimierten OpenCL-Variante.

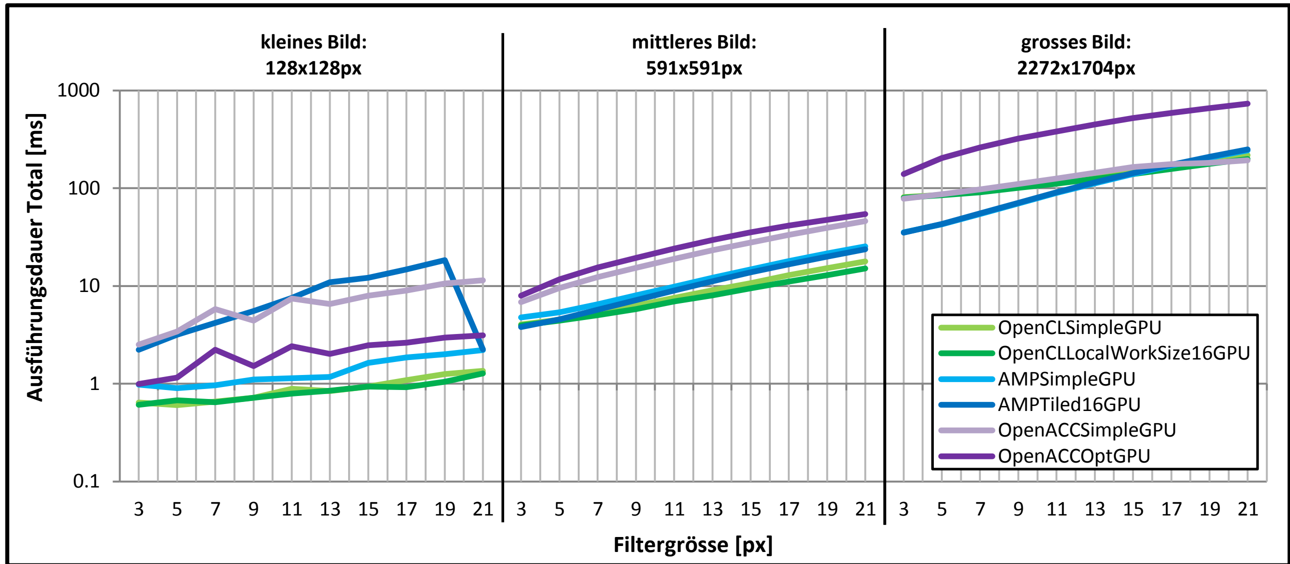
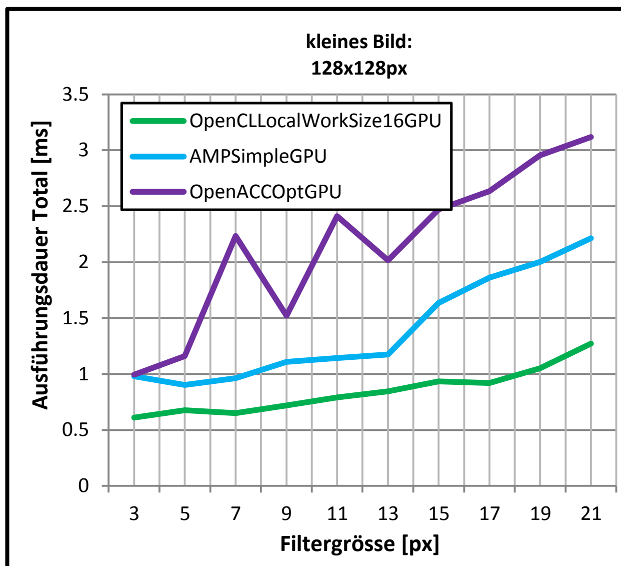
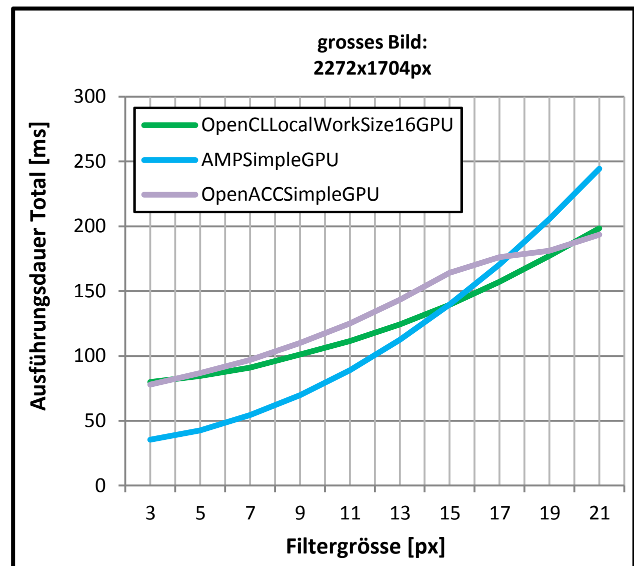


Abbildung C.9: Gesamtausführungsdauer auf der GPU der jeweils besten simplen und optimierten Messreihe aller HSAs.



a)



b)

Abbildung C.10: Gesamtausführungsdauer auf der GPU der jeweils besten Messreihe aller HSAs. a) Kleine Problemgrösse. b) Grosse Problemgrösse.

5. Forum-Anfrage OpenCL

Um die Ausführung von OpenCL-Code auf CPU besser zu verstehen, wurde im Intel-Forum [74] nachgefragt. Im Folgenden ist die komplette Konversation abgedruckt.

```
Christian Lang, 11.06.2013

Hi, I am a Master Student at the FHNW in Windisch (Switzerland) and I am comparing three
Heterogeneous System Architectures (HSA), namely Microsoft C++ AMP, OpenCL and OpenACC.
In this work I measure not only GPU performance but also CPU performance in the scope of
image convolution on three different images (128x128, 591x591, 2272x1704 pixel) and 10
different filter sizes (3x3, 5x5,..., 21x21). In the CPU case I compare the HSAs with a
simple OpenMP implementation and discovered that my OpenCL implementation, that runs on
an Intel Core i7 950 3.06 GHz, has similar performance as OpenMP. As you can see in
"Performance_OpenCL_Alg.xls" (Abbildung C.11) OpenCL reaches the performance of OpenMP
only at relatively big filter sizes.

Now my question is, how can this be explained or fixed.I know that a bigger Work Group
Size would activate vectorization and result in a better performance, but OpenMP does not
use that as well, so I would like to know, why OpenCL do not fully use all 4 Cores at
small filters.

I added the source code of the OpenMP and OpenCL Implementation. As you can see I use the
class "Mat" of OpenCV to store my images and the filter. To measure execution time I used
the OpenMP function "omp_get_wtime()".

I would be very pleased, if you could point me to the right direction.
```

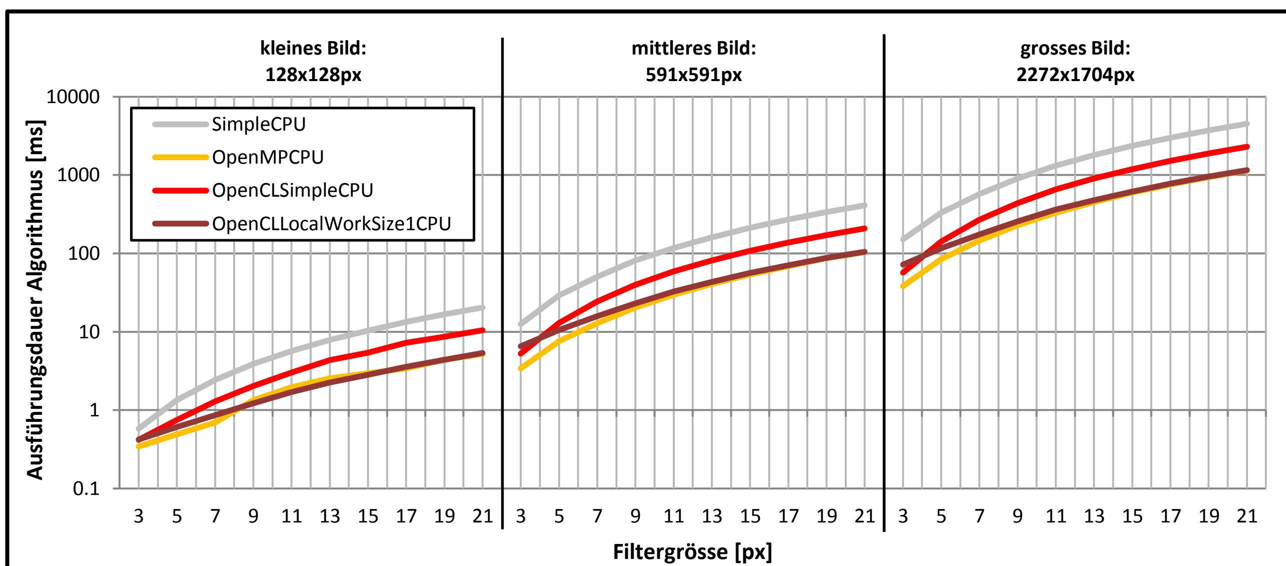


Abbildung C.11: Performance_OpenCL_Alg.xls

```
Maxim Shevtsov (Intel), 13.06.2013

Hi, from your charts I can conclude (pls correct me) that scalarized version of the code
(i.e. with local size of 1) is faster than auto-vectorized code (OpenCLLocalWorkSize1CPU
vs OpenCLSimpleCPU). This is often the case for code in (u)chars, since using these data
types requires packing/unpacking for vectorization which is currently performed using
SSE/AVX integer instructions.

So first experiment is to disable vectorizer via vec_type_hint, which should improve the
perf of the original OpenCLSimpleCPU version (since running scalar ver of the kernel via
specifying the local size of 1 is too overheady, because of excessive number of the
workgroups produced). I'm seeing a padding in your code, so hopefully global size for the
NDRange is good (e.g. dividable by 8), otherwise (for example for the uneven size in the
dimension zero of the NDRange) the scalar version of the kernel would run anyway.
```

Secondly, to get a better results, I would suggest to operate on (at least) 8 pixels in a workitem. This can be done thru uchar8 data type (notice that number of kernel calls will shrink accordingly and reduce overhead on workitems scheduling). Actually this means the manual vectorization, yet you can turn the vectorizer on and off and re-access the perf for the new ver.

Christian Lang, 17.06.2013

Hi. Thank you for your answer. It helped me to perform some more tests, especially with auto-vectorization turned off. In addition, I made some tests with auto-vectorization on and work group sizes bigger than 1x1. Finally, a work group size of 3x3 with auto-vectorization beats the OpenMP implementation.

But my actual question has not be answered. As you can see in "Performance_OpenCL_CPU_Usage.png" (Abbildung C.12), the OpenMP implementation uses fully all 4 cores of CPU however OpenCL uses the whole CPU ressources only at bigger filter and image sizes.

My question now is: Does this effect appear because of the compilation of the device code each time it is executet (I run each kernel 5 times in a row)? Or are the calculations of the small images and filters such inexpensive that the CPU does not reach full load bevor the kernel is finished?

Thank you in advance.

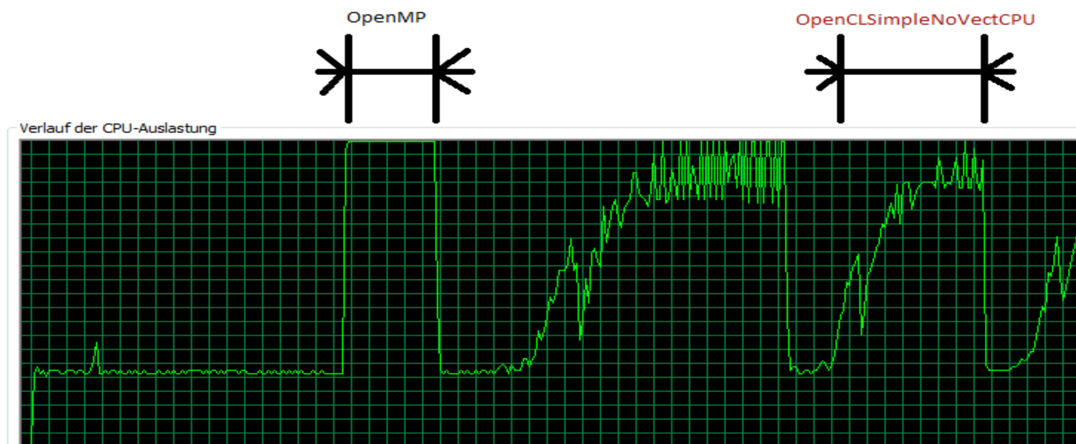


Abbildung C.12: Performance_OpenCL_CPU_Usage.png

Maxim Shevtsov (Intel), 17.06.2013

Hi again, be careful with plain "CPU cores utilization" being the metric to optimize. As you know a simple infinite loop in each thread might occupy the cores completely, wasting time and watts, despite potentially perfect utilization.

If the performance of say 2 threading models is similar, yet the utilization is not, it might indicate different appoaches to task stealing and to wait-loops in the models. Intel OpenCL for CPU relies on the TBB which is generally more adaptive to the load. Especially for the cases when there are no much workgroups (workgroups finest granularity for therading in our implementation) to saturate all the threads and facilate the task-stealing.

On the OpenMP saturating the cores completely, I recollect that when compiling the same native openmp app with Visual Studio 2008 you will see the uneven cores saturation (just like with Intel OpenCL), yet the same app compiled with VS 2010 demostartes full utilization while the same perf as VS2008.

D. Kopfdetektion mit OpenCV

1. CMake Konfigurations-Protokoll von OpenCV 2.9.0 Build

```
CUDA detected: 5.5
CUDA NVCC target flags: -gencode;arch=compute_11,code=sm_11;-
gencode;arch=compute_12,code=sm_12;-gencode;arch=compute_13,code=sm_13;-
gencode;arch=compute_20,code=sm_20;-gencode;arch=compute_20,code=sm_21;-
gencode;arch=compute_30,code=sm_30;-gencode;arch=compute_35,code=sm_35;-
gencode;arch=compute_30,code=compute_30
Could NOT find PythonInterp (missing: PYTHON_EXECUTABLE) (Required is at least version
"2.0")
Found apache ant 1.8.4: D:/Programme/apache-ant-1.8.4/bin/ant.bat
Could NOT find JNI (missing: JAVA_AWT_LIBRARY JAVA_JVM_LIBRARY JAVA_INCLUDE_PATH
JAVA_INCLUDE_PATH2 JAVA_AWT_INCLUDE_PATH)

General configuration for OpenCV 2.9.0 =====================================
Version control: unknown

Platform:
Host: Windows 6.1 AMD64
CMake: 2.8.11.2
CMake generator: Visual Studio 11 Win64
CMake build tool: C:/PROGRA~2/MICROS~1.0/Common7/IDE/devenv.com
MSVC: 1700

C/C++:
Built as dynamic libs?: YES
C++ Compiler: C:/Program Files (x86)/Microsoft Visual Studio 11.0/VC/bin/x86_amd64/cl.exe
(ver 17.0.51106.1)
C++ flags (Release): /DWIN32 /D_WINDOWS /W4 /GR /EHa /D _CRT_SECURE_NO_DEPRECATE /D
_CRT_NONSTDC_NO_DEPRECATE /D _SCL_SECURE_NO_WARNINGS /Gy /bigobj /Oi /wd4251 /MD /O2 /Ob2
/D NDEBUG /Zi
C++ flags (Debug): /DWIN32 /D_WINDOWS /W4 /GR /EHa /D _CRT_SECURE_NO_DEPRECATE /D
_CRT_NONSTDC_NO_DEPRECATE /D _SCL_SECURE_NO_WARNINGS /Gy /bigobj /Oi /wd4251 /D_DEBUG
/MDd /Zi /Ob0 /Od /RTC1
C Compiler: C:/Program Files (x86)/Microsoft Visual Studio 11.0/VC/bin/x86_amd64/cl.exe
C flags (Release): /DWIN32 /D_WINDOWS /W3 /D _CRT_SECURE_NO_DEPRECATE /D
_CRT_NONSTDC_NO_DEPRECATE /D _SCL_SECURE_NO_WARNINGS /Gy /bigobj /Oi /MD /O2 /Ob2 /D
NDEBUG /Zi
C flags (Debug): /DWIN32 /D_WINDOWS /W3 /D _CRT_SECURE_NO_DEPRECATE /D
_CRT_NONSTDC_NO_DEPRECATE /D _SCL_SECURE_NO_WARNINGS /Gy /bigobj /Oi /D_DEBUG /MDd /Zi
/Ob0 /Od /RTC1
Linker flags (Release): /machine:x64 /INCREMENTAL:NO /debug
Linker flags (Debug): /machine:x64 /debug /INCREMENTAL
Precompiled headers: YES

OpenCV modules:
To be built: core imgproc highgui bioinspired flann features2d calib3d ml video objdetect
contrib cudev gpulegacy gpuarithm gpuwarping gpu legacy gpufilters gpuimgproc gpubgsegm
gpucodec gpufeatures2d gpuoptflow gpustereo ocl nonfree optim photo softcascade stitching
superres ts videostab
Disabled: world
Disabled by dependency: -
Unavailable: androidcamera java python

GUI:
QT: NO
Win32 UI: YES
OpenGL support: NO

Media I/O:
ZLib: build (ver 1.2.7)
JPEG: build (ver 90)
WEBP: build (ver 0.3.1)
```

```
PNG: build (ver 1.5.12)
TIFF: build (ver 42 - 4.0.2)
JPEG 2000: build (ver 1.900.1)
OpenEXR: build (ver 1.7.1)

Video I/O:
Video for Windows: YES
DC1394 1.x: NO
DC1394 2.x: NO
FFMPEG: YES (prebuilt binaries)
codec: YES (ver 53.61.100)
format: YES (ver 53.32.100)
util: YES (ver 51.35.100)
swscale: YES (ver 2.1.100)
gentoo-style: YES
OpenNI: NO
OpenNI PrimeSensor Modules: NO
PvAPI: NO
GigEVisionSDK: NO
DirectShow: YES
Media Foundation: NO
XIMEA: NO

Other third-party libraries:
Use IPP: NO
Use Eigen: YES (ver 3.2.0)
Use TBB: YES (ver 4.1 interface 6105)
Use OpenMP: NO
Use GCD NO
Use Concurrency NO
Use C=: NO
Use Cuda: YES (ver 5.5)
Use OpenCL: YES

NVIDIA CUDA
Use CUFFT: YES
Use CUBLAS: YES
USE NVCUVID: YES
NVIDIA GPU arch: 11 12 13 20 21 30 35
NVIDIA PTX archs: 30
Use fast math: YES

OpenCL
Include path: C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v5.5/include
libraries: C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v5.5/lib/x64/OpenCL.lib
Use AMDFFT: NO
Use AMDBLAS: NO

Python:
Interpreter: NO

Java:
ant: D:/Programme/apache-ant-1.8.4/bin/ant.bat (ver 1.8.4)
JNI: NO
Java tests: NO

Tests and samples:
Tests: NO
Performance tests: NO
C/C++ Examples: NO

Install path: D:/Programme/OpenCV/build_vc11/install

cvconfig.h is in: D:/Programme/OpenCV/build_vc11

-----------------------------------------------------------------
Configuring done
Generating done
```

2. Bewertungstabelle trainierter Classifier

Datum	06.08.2013	08.08.2013	09.08.2013	12.08.2013	13.08.2013	14.08.2013	15.08.2013	16.08.2013	19.08.2013	19.08.2013	19.08.2013	20.08.2013
featureType	LBP	LBP	HAAR	LBP	LBP	LBP	LBP	LBP	LBP	LBP	LBP	LBP
numPos	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000
numNeg	3000	3000	3000	3000	3000	3000	3000	3000	3000	3000	3000	3000
precalcValBuffSize	1512	1512	512	1512	1512	1512	1512	1512	1512	1512	1512	1512
precalcIdxBufSize	1512	1512	512	1512	1512	1512	1512	1512	1512	1512	1512	1512
numStages	20	20	20	20	20	20	19	20	20	20	20	20
w	40	20	20	20	50	50	50	50	40	40	40	40
h	40	20	20	20	50	50	50	50	40	40	40	40
minHitRate	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999
maxFalseAlarmRate	0.5	0.5	0.5	0.5	0.5	0.7	0.3	0.45	0.45	0.45	0.45	0.45
weightTrimRate	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95
maxDepth	1	1	1	1	1	1	1	1	1	1	1	1
mode	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL
Boosting	GAB	GAB	GAB	RAB	GAB	GAB	GAB	RAB	GAB	GAB	DAB	LB
Datasets	1 + 2	1 + 2	1 + 2	1 + 2	2 + 4	2 + 4	2 + 4	2 + 4	1 + 2	2 + 4	1 + 2	1 + 2
Trainingszeit	2h	20min	2d	20min	2.5h	2.5h	1.5d	1d	2h	2h	4h	2h
Name	cascade_lbp_head _130806.xml	cascade_lbp_head _130808.xml	cascade_haar_head _130809.xml	cascade_lbp_head _130812.xml	cascade_lbp_head _130813.xml	cascade_lbp_head _130814.xml	cascade_lbp_head _130815.xml	cascade_lbp_head _130816.xml	cascade_lbp_head _130819.xml	cascade_lbp_head _130819_2.xml	cascade_lbp_head _130819_3.xml	cascade_lbp_head _130820.xml
Akzeptanzrate						3000 : 0.00491852				3000 : 0.000048456		
subjektive Präz. [1-10]	4	3	4	1	5	2	3	3	5	3	1	1
gemessene Präzision			Hits:1247 Miss:1772 FP:9417									
Bemerkung	Nur Hinterköpfe Findet viel anderes	Nur Hinterköpfe	Findet viel anderes	Nur Hinterköpfe	Findet viel anderes	Findet viel anderes	Findet sehr wenig	Findet sehr wenig Findet viel anderes	Fast nur Hinterköpfe	Fast nur Hinterköpfe Findet viel anderes	Findet sehr viel anderes	Findet sehr viel anderes

Datum	20.08.2013	20.08.2013	21.08.2013	21.08.2013	21.08.2013	21.08.2013	22.08.2013	22.08.2013	22.08.2013	23.08.2013	23.08.2013	
featureType	LBP	LBP	LBP	LBP	LBP	LBP	LBP	LBP	LBP	LBP	LBP	
numPos	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000	
numNeg	3000	3000	3000	3000	3000	3000	3000	3000	3000	3000	3000	
precalcValBuffSize	1512	1512	1512	1512	1512	1512	1512	1512	1512	1512	1512	
precalcIdxBufSize	1512	1512	1512	1512	1512	1512	1512	1512	1512	1512	1512	
numStages	24	24	16	18	18	18	18	18	18	18	18	
w	40	40	40	40	40	40	25	40	40	40	40	
h	40	40	40	40	40	40	25	40	40	40	40	
minHitRate	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.9999	0.9999	0.9999	0.9999	
maxFalseAlarmRate	0.45	0.55	0.4	0.4	0.35	0.3	0.3	0.3	0.3	0.3	0.25	
weightTrimRate	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	
maxDepth	1	1	1	1	1	1	1	1	1	2	1	
mode	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL	ALL	
Boosting	GAB	GAB	GAB	GAB	GAB	GAB	GAB	GAB	GAB	GAB	GAB	
Datasets	1 + 2	1 + 2	1 + 2	1 + 2	1 + 2	1 + 2	1 + 2	1 + 2	2 + 4	1 + 2	1 + 2	
Trainingszeit	4h	4h	1h	1.5h	2h	2h	2h	2h	3h	2h	3h	
Name	cascade_lbp_head _130820_2.xml	cascade_lbp_head _130820_3.xml	cascade_lbp_head _130821.xml	cascade_lbp_head _130821_2.xml	cascade_lbp_head _130821_3.xml	cascade_lbp_head _130821_4.xml	cascade_lbp_head _130822.xml	cascade_lbp_head _130822_2.xml	cascade_lbp_head _130822_3.xml	cascade_lbp_head _130823.xml	cascade_lbp_head _130823_2.xml	
Akzeptanzrate	3000 : 0.00000693883		3000 : 0.0000607453	3000 : 0.0000285235			3000 : 0.0000042781	3000 : 0.0000104466		3000 : 0.0000210026		
subjektive Präz. [1-10]	4	3	3	4	6	6.5	6	7	6	0	7.5	
gemessene Präzision												
Bemerkung	Findet wenig	Findet viel anderes (Körper)	Findet viel anderes schneller als mit 20 Stufen	Findet anderes (Körper)	gut Findet anderes (Körper)	gut Findet anderes (Körper)	nicht ganz so zuverlässig	gut Erkennt mehr als 130821_4, aber mehr false alarm	Dataset 4 ist schlechter als Dataset 1	funktioniert nicht (not a stump)	gut erkennt weniger als 130822_2, aber weniger false alarm	

Datasets:
1 UcoHead
2 Negative Samples von Naotoshi
3 M.I.T cbcl
4 Coffeebreak

Abbildung D.1: Bewertungstabellen der trainierten Classifier. Gelb markierte Felder enthalten Parameter, die sich zum vorherigen Test verändert haben.

3. Beispielbilder echter Detektionen

Die in Abbildung D.2 gezeigten Bilder sind Ausschnitte echter Detektionen im Live-Video. Sie wurden alle mit den finalen Einstellungen des LBP-Classifiers (`cascade_lbp_head_130823_2.xml`) erstellt. Tabelle D.1 zeigt die dazu verwendeten Testdaten.

Bilder	Testvideo
a) und b)	\Testdaten\Moritz\ZickZack.MP4
c) und d)	\Testdaten\Dominik\ZickZack.MP4
e) bis h)	\Testdaten\Thekla\ZickZack.MP4
i) und j)	\Testdaten\Martin\ZickZack.MP4
k) bis m)	\Testdaten\MehrerePersonen\Flur.MP4

Tabelle D.1: Verwendete Testvideos in Abbildung D.2.

Abbildung D.2: Beispielbilder des finalen Classifiers. h) und j) enthalten je eine Fehldetektion.