Master-Thesis

IP913

Parallel Computer Vision:

Heterogeneous Video Scheduling & TLD Framework

Lang Christian

Matrikelnummer: 08-169-047

Fachhochschule Nordwestschweiz

Advisor:

Prof. Dr. Christoph Stamm, FHNW

Experte:

Christof Sidler, SCS

Abgabedatum: 14.02.2014

Danksagungen

An dieser Stelle möchte ich mich bei all denjenigen bedanken,
die mich während dieser Arbeit unterstützt und motiviert haben.

Projektbetreuer:
Christoph Stamm, FHNW

Experte:
Christof Sidler, Super Computing Systems

Fachliche Unterstützung:
Daniel Kröni, FHNW

Lektorin:
Velin Erdin

Und nicht zuletzt meinen Eltern für ihre
Unterstützung während des gesamten Studiums.

Abstract

Die vorliegende Arbeit befasst sich mit der automatischen Personenerkennung in Live-Videos auf einem handelsüblichen PC mit Grafikbeschleuniger. Die Fragestellung lautet, wie ein rechenaufwendiges Live-Erkennungssystem unter Ausnutzung der verfügbaren Ressourcen realisiert werden kann. Zudem soll ein Weg gefunden werden, wie die Resultate der Erkennungssoftware automatisch auf Korrektheit und Fehlerrate überprüft werden können.

Dazu wird ein Annotationswerkzeug entwickelt, welches es erlaubt, Testvideos mit den korrekten Positionen und Namen der Personen zu vermerken. Zudem wird ein Framework entwickelt, welches das Design von Streaming-Anwendungen durch eine Pipeline ermöglicht. Dieses Framework soll auf zeitliches Verhalten und erreichbare Performance in heterogenen Systemen untersucht und optimiert werden.

Im Rahmen dieser Arbeit wurde ein Pipeline-Framework entwickelt, welches auf einfache Weise erlaubt, alle Ressourcen eines heterogenen Systems auszunutzen. Dazu verwendet es die Pipeline- und Task-Parallelität um einerseits begrenzende Eigenschaften einer typischen Pipeline auszumerzen, andererseits kann dadurch eine Parallelisierung erreicht werden, ohne dass spezifische Task-Implementationen von Hand parallelisiert werden müssen. Durch den Umbau eines bereits bestehenden Kopfdetektions-System wird die Effektivität des Frameworks aufgezeigt.

Inhaltsverzeichnis

1 Einleitung

Da heutzutage in beinahe jedem PC, Smartphone oder Tablet eine Art von Grafikbeschleunigung vorhanden ist, macht es Sinn, diese GPUs nebst der Grafikberechnung für rechenaufwändige allgemeine Aufgaben zu verwenden. Um alle unterschiedlichen Rechengeräte eines Systems anzusprechen, bieten sich heterogene System-Architekturen (HSA) an. Diese ermöglichen zwar, einen Algorithmus einmalig zu schreiben und trotzdem auf unterschiedlichen Geräten auszuführen, bieten aber keine Unterstützung beim Entscheiden, wann welche Art von Gerät für die Ausführung verwendet werden soll.

Für diese Aufgabe wird ein Scheduler benötigt, welcher einerseits über Informationen zu der aktuellen Auslastung des Systems verfügt, andererseits über die unterschiedlichen Rechengeräte und dessen Ausführungsgeschwindigkeit Bescheid weiss. Mittels dieser Informationen sollte er im Stande sein, anstehende Daten auf dem besten aktuell freien Rechengerät zu verarbeiten und somit das System, unter Ausnutzung aller Ressourcen, optimal zu beschleunigen.

Im Kontext von Stream-Anwendungen, wie z. B. Live-Video-Verarbeitung, werden alle einzelnen Verarbeitungsschritte, welche jeder Teil des Streams durchlaufen muss, häufig durch eine Pipeline verkörpert. Dieses Pipeline-Pattern ermöglicht einerseits die gute Strukturierung einer Anwendung, andererseits bietet die implizite Pipeline-Parallelität die Chance, das System durch grobgranulare Parallelität zu beschleunigen.

1.1 Aufgabenstellung

Basierend auf den Projekten 7 [1] und 8 [2] soll ein Hilfswerkzeug erstellt werden, welches es ermöglicht, die Detektionsgenauigkeit der Personenerkennungssoftware zu bestimmen. Dieses Tool soll die manuelle Annotation von Personen in grossen Videostreams ermöglichen und die Positionsdaten als lesbare Dateien persistieren. Zudem soll eine klare und einfache API entwickelt werden, welche es erlaubt, die Daten erneut einzulesen und mit echten Detektionen zu vergleichen.

Es soll ein Scheduler für heterogene Pipeline-Systeme entwickelt werden, um alle vorhandenen Rechengeräte auszulasten und das System bestmöglich zu beschleunigen. Dabei soll eine geeignete Scheduling-Strategie ermittelt werden, welche selbst möglichst wenig Rechenzeit benötigt und ohne Busy-Waiting auskommt. Zudem soll evaluiert werden, wie eine Pipeline-Struktur abstrahiert werden kann, sodass sie die Anforderungen eines nebenläufigen Systems unterstützt. Die entwickelte Scheduling-Strategie soll anhand von Simulationen und Messungen analysiert und optimiert werden.

Die Software aus dem Projekt 8 soll mittels des entwickelten heterogenen Scheduling so umgebaut werden, dass alle Ressourcen ausgenutzt werden. Dabei sollen die bekannten HSAs und kompatiblen Task-Implementationen aus OpenCV verwendet werden. Zudem soll die Personenerkennung mittels dem Tracking-Learning-Detection (TLD) Framework erweitert werden.

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

1.2 Überblick

Für das Annotations-Tool wird eine Klasse geschrieben, welche es erlaubt, Detektionen in Video-Streams zu verwalten und zu persistieren. Dazu verwendet sie eine Speicherlösung von OpenCV. Das Tool und die Detektions-Klasse werden in Kapitel 2 beschrieben.

Für die Verwirklichung eines Pipeline-Systems wird die Asynchronous Agents Library (AAL) von Microsoft verwendet, welche das gut geeignete Aktoren-Modell implementiert. Das Pipeline-Pattern wird theoretisch analysiert, um Schwachstellen und Optimierungspotential zu identifizieren. Auf diesen Erkenntnissen aufbauend, werden drei Scheduling-Strategien entwickelt und mittels zwei unterschiedlichen Simulationen getestet. Kapitel 3 erläutert die Problematik des Pipeline-Schedulings, das Vorgehen der drei Strategien und dessen Vergleich.

Mittels der entwickelten Scheduling-Strategie wird ein Pipeline-Framework implementiert, welches in Kapitel 4 beschrieben wird. Dieses liefert alle Grundbausteine, um eine nutzerspezifische heterogene Pipeline aufzubauen und durch den Scheduler verwalten zu lassen. Das Framework nutzt dabei die implizite grobgranulare Parallelität einer Pipeline aus und ermöglicht dadurch die Ausnutzung aller verfügbaren Ressourcen ohne speziell parallelisierte Task-Implementationen.

Kapitel 5 erläutert die Verwendung des Pipeline-Frameworks in der Personenerkennungs-Software und zeigt zugleich die Einsatzfähigkeit des Frameworks.

2 Annotations-Tool

Für die empirische Analyse der Erkennungsgenauigkeit einer Personenerkennung-Software wird ein Markierungswerkzeug erstellt, welches die Annotation von echten Detektions-Bereichen in Testvideos erlaubt. Damit sollen auf einfache Weise die Köpfe in den Testdaten aus dem Projekt 7 [1] mit dem Personennamen markiert und abgespeichert werden. Diese annotierte Daten werden zur Erstellung von statistischen Werten wie „True Positive Rate“ oder „False Positive Rate“ benötigt und ermöglichen eine präzise Bewertung des Erkennungssystems.

2.1 Benutzer-Schnittstelle

OpenCV bietet bereits in der Grundkompilation ein eigenes Modul für grundlegende grafische Benutzerschnittstellen. Damit lassen sich Bilder einfach in einem Fenster anzeigen und Videos mit einer Fortschrittsanzeige versehen. Das `highgui`-Modul beherrscht aber in Kombination mit der GUI-Bibliothek Qt [3] zusätzliche Funktionen und eine angenehmere Darstellung. Um diese zu aktivieren, wird OpenCV mit Qt neu kompiliert (siehe Anhang B), wobei die grafischen Verbesserungen beim nächsten Applikationsstart ersichtlich sind. Abbildung 2.1 zeigt das GUI des Annotations-Tools.

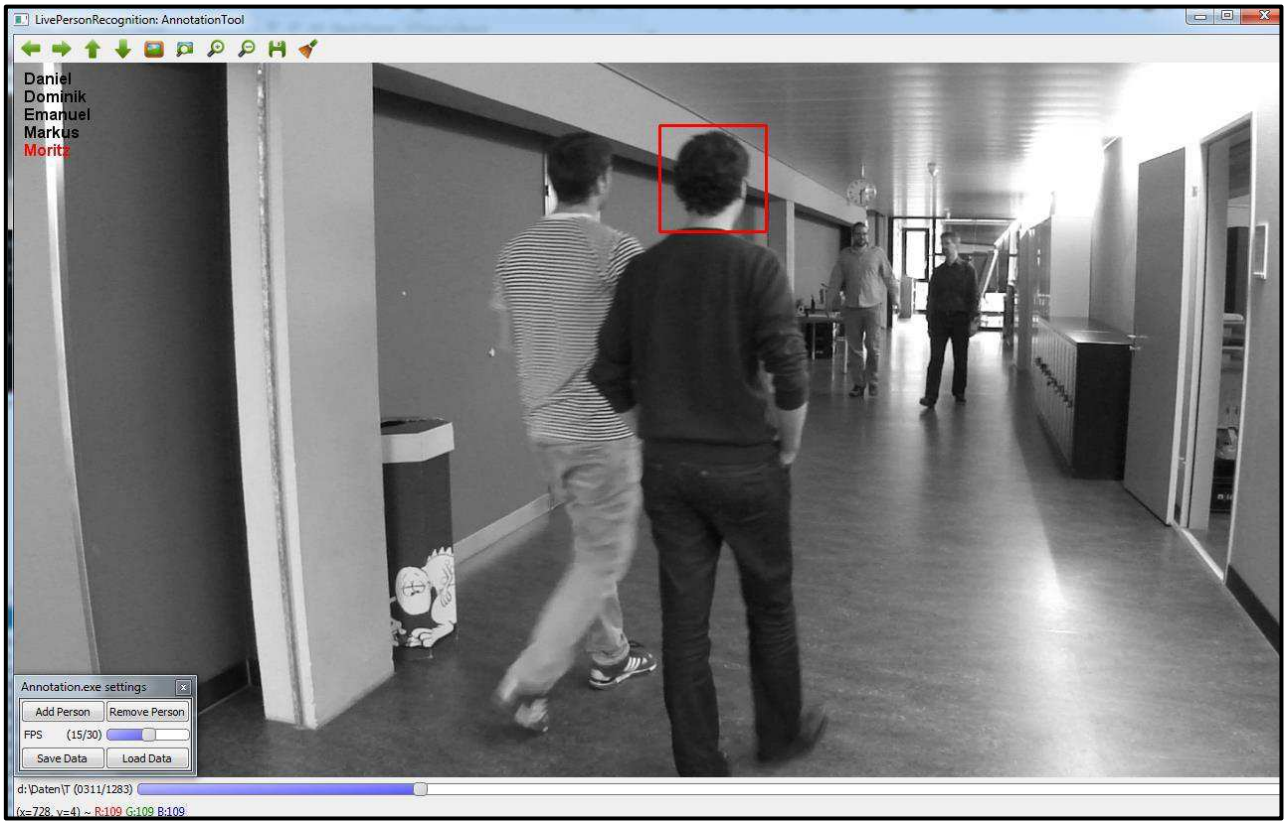


Abbildung 2.1: GUI des Annotations-Tools inklusive Control-Panel zur Verwaltung der Personen, Framerate und Speicherung der Daten. Die Person „Moritz“ ist ausgewählt und dessen aktuelle Kopfposition angezeigt.

Die Erweiterungen des mit Qt kompilierten OpenCV sind die Toolbar mit generellen Werkzeugen und das Control-Panel, welches eigene Buttons und Trackbars enthalten kann. Diese können mit der Funktion `createButton` bzw. `createTrackbar` erstellt werden. Dabei kann jeweils ein Funktions-Zeiger übergeben werden, welcher auf die Callback-Funktion des Bedienelements zeigt. Mit der Funktion `setMouseCallback` kann ein Maus-Handler registriert werden, der alle Maus-Ereignisse registrieren und abhandeln kann. Allerdings werden bestimmte Events standartmässig von OpenCV selbst behandelt, wodurch die

Verwendungsmöglichkeiten eingeschränkt werden. Beispielsweise öffnet ein Rechtsklick standartmässig ein Kontextmenu, weshalb dieses Event nicht für eigene Zwecke verwendet werden kann.

Die Textausgabe im Bild wird mittels der `addText`-Funktion ausgelöst. Diese benötigt ein Schrift-Objekt, welches mittels `fontQt` erstellt wird. Der Text kann auf ein beliebiges Bild in Form eines `Mat`-Objektes gezeichnet und so auf dem Bildschirm ausgegeben werden. Des Weiteren können die Fensterpositionen mittels der Funktionen `saveWindowParameters` und `loadWindowParameters` gespeichert und wiederhergestellt werden.

2.2 Datenverwaltung und Persistierung

Die Verwaltung der Positionsdaten übernimmt die eigens dafür geschriebene `PositionData`-Klasse. Sie ermöglicht einerseits die Organisation der Daten in einer `Map` im Hauptspeicher, andererseits die Persistierung des kompletten Datensatzes in eine YAML-Datei. Dazu wird die Klasse `FileStorage` aus OpenCV benutzt, welche die Formate XML und YAML unterstützt. YAML ist eine an XML und JSON angelehnte Sprache und wird zur Datenserialisierung eingesetzt. Ähnlich wie XML ist sie auch für Menschen gut verständlich.

Das Datenschema besteht aus einer Map `persons`, welche die Datensätze aller Personen enthält. Als Key wird der Name der Person gespeichert. Das Value ist eine weitere Map, welche als Key den Index eines Frames hat und als Value die Grösse und Position des Detektionsbereiches in diesem Frame. Diese Map wird beim Persistieren serialisiert und lesbar abgespeichert. Ein Beispiel einer solchen YAML-Datei ist in Listing 2.1 ersichtlich.

```
%YAML:1.0
videoWidth: 1280
videoHeight: 720
selectedPerson: Dominik
persons:
   -
      name: Daniel
      positions:
         - { frame:273, area:{ x:63, y:13, width:120, height:120 } }
         - { frame:274, area:{ x:71, y:13, width:120, height:120 } }
         - { frame:275, area:{ x:83, y:15, width:120, height:120 } }
   -
      name: Dominik
      positions:
         - { frame:0, area:{ x:957, y:125, width:65, height:65 } }
         - { frame:1, area:{ x:957, y:125, width:65, height:65 } }
         - { frame:2, area:{ x:957, y:126, width:65, height:65 } }
```

Listing 2.1: Beispiel einer YAML-Datei (Flur.MP4.yaml). Enthält die Information über die Grösse des annotierten Videos, die aktuell selektierte Person und die Daten von zwei Personen. Die Zugehörigkeit dieser Daten wird durch den Dateinamen definiert.

Da die Daten des Detektionsbereiches in Pixel abgespeichert werden, ist die Grössenangabe des annotierten Videos notwendig, falls die Daten in einer anderen Skalierung dargestellt werden sollen. Diese Grössenangaben werden beim Instanziieren der `PositionData`-Klasse übergeben. Sollen bestehende Daten geladen werden, wird ein Skalierungsfaktor aus der gespeicherten Videogrösse und der aktuellen Videogrösse berechnet. Dieser Faktor wird beim Importieren der Daten angewendet, damit alle geladenen Detektionsbereiche beim Anzeigen im aktuellen Video die richtige Position und Grösse aufweisen. Bei einem erneuten Speichern werden alle Daten und die Videogrösse mit den neu berechneten Werten überschrieben.

2.3 Verwendung

Nachdem das Tool gestartet und ein entsprechendes Video geöffnet wurde, muss als erstes mindestens eine Person erfasst werden. Das in Abbildung 2.2 ersichtliche Control-Panel, welches alle notwendigen Funktionen enthält, wird mit einem Klick auf den Pinsel (letztes Icon rechts in der Toolbar) geöffnet. Hier können neue Personen hinzugefügt und bestehende gelöscht werden. Zu beachten ist, dass bei den Namen auf Gross/Kleinschreibung geachtet werden muss. Um zwischen verschiedenen Personen umzuschalten, wird die „P“-Taste verwendet.

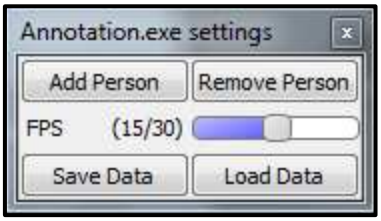


Abbildung 2.2: Control-Panel des Annotation-Tools. Ermöglicht die Verwaltung der Personen, die Einstellung der Abspielgeschwindigkeit und das Speichern und laden der YAML-Dateien.

Mittels Links-Klick ins Bild wird die ausgewählte Person im Bild markiert. Die Grösse des Bereiches lässt sich mit den „A“- und „Y“-Tasten verändern. Um einen Bereich zu verschieben, genügt ein erneutes Klicken an die neue Position im Bild. Bei gedrückter Maustaste kann der Bereich verschoben werden. Um einen fälschlich markierten Bereich zu entfernen, wird die „Q“-Taste verwendet.

Die Navigation durch das Video kann auf mehrere Arten gehandhabt werden. Einerseits lässt sich die Fortschrittsanzeige mit der Maus verschieben, wodurch das gewählte Frame sofort angezeigt wird. Andererseits kann im Pausenmodus mit den Tasten „B“ und „N“ um einzelne Frames nach hinten oder vorne gesprungen werden. Mittels der Leertaste lässt sich das Abspielen starten und pausieren. Die Abspielgeschwindigkeit wird mit der FPS-Bar (Bilder pro Sekunde) im Control-Panel geregelt. Falls diese auf 0 gestellt wird, wird das kontrollierte Abspielen deaktiviert und das Programm spielt das Video so schnell ab, wie es die PC-Hardware zulässt.

Wird beim Starten ein Video geladen, zu welchem bereits eine YAML-Datei im selben Ordner existiert, werden die darin enthaltenen Daten automatisch geladen. Sollen diese Daten zu einem späteren Zeitpunkt erneut geladen werden, also die bis dahin gemachten Änderungen verworfen werden, kann der entsprechende Button im Control-Panel verwendet werden. Durch Drücken des „Save Data“-Buttons werden die Daten gespeichert und eventuell vorhandene Dateien überschrieben.

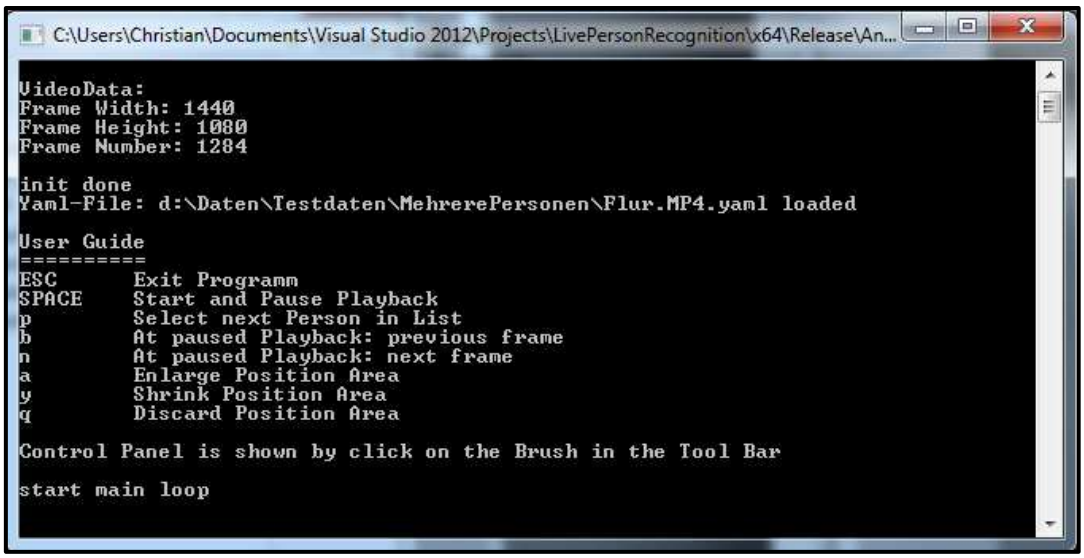


Abbildung 2.3: Konsole mit Informationen zur Video-Datei, den geladenen Annotations-Daten und einem User-Guide.

Eine zusätzliche Übersicht über die Bedienung liefert die beim Starten geöffnete Konsole, welche in Abbildung 2.3 gezeigt ist. Die praktikabelste Variante um effizient Videos zu annotieren ist, als erstes alle Personen zu erfassen, die Abspielgeschwindigkeit auf ca. 5 fps einzustellen und mit gedrückter Maustaste dem Kopf der selektierten Person während dem Abspielen nachzufahren. Währenddessen kann mit der zweiten Hand die Knöpfe für die Grössenregulierung des Bereichs bedient und das Video pausiert werden, falls ein gemachter Fehler korrigiert werden soll. Dazu wird bevorzugterweise manuell durch die einzelnen Frames navigiert und Markierungen verschoben oder gelöscht.

3 Heterogenes Pipeline-Scheduling

Live-Video-Verarbeitung stellt hohe Leistungsansprüche an die verwendete Hard- und Software. Um die verfügbare Hardware eines PCs vollständig ausnutzen zu können, müssen die Ressourcen einerseits über eine geeignete Schnittstelle angesprochen, andererseits effizient verwaltet werden können.

Dazu wird in diesem Kapitel das Aktoren-Modell und die darauf basierende Agenten-Bibliothek von Microsoft vorgestellt, welche den Umgang und das Design von nebenläufigen Systemen vereinfacht. Um eine Streaming-Anwendung einfach aufzubauen und zu parallelisieren, wird das Pipeline-Pattern verwendet. Dieses wird anhand einer allgemeinen Pipeline-Struktur auf Schwachstellen analysiert und darauf aufbauend unterschiedliche Verwaltungs-Strategien implementiert. Diese sollen mit einfachen Methoden die verfügbaren Rechengeräte verwalten und durch Pipeline-Parallelität eine grob-granulare Parallelisierung ermöglichen. Danach werden Grenzwerte für die Leistungsfähigkeit eines solchen Systems definiert und diese mit zwei unterschiedlichen Typen von Simulationen verglichen.

Ein Teil dieses Kapitels wird auch im Paper [4] im Anhang E behandelt.

3.1 Aktorenbasierte Systeme

Das grösste Problem bei parallelisierten Programmen ist die Zugriffsverwaltung der von unterschiedlichen Prozessen gemeinsam genutzten Daten. Die Komplexität dieser expliziten Synchronisierung führt häufig zu Problemen und soll deshalb vermieden werden. Das erstmals 1973 von Hewitt et al. vorgestellte Aktoren-Modell [5] umgeht die Synchronisierung, indem die Kernelemente des Systems nur lokale Speicher zur Verfügung haben und kein geteilter oder globaler Speicher existiert. Die als Aktoren bezeichneten Einheiten dieses prozessorientierten Systems versenden und empfangen Nachrichten, um den Datenaustausch zu ermöglichen. Da jeder Speicher implizit nur vom ihm zugewiesenen Aktor verändert und gelesen werden darf, können alle Aktoren als sequentieller Programmcode betrachtet werden und jegliche Synchronisierung der Daten entfällt. Wie schon angedeutet, werden die Aktoren nicht durch Objekte sondern durch leichtgewichtige Prozesse verkörpert. Jeder Aktor enthält eine eigene Nachrichtenwarteschlange, welche eingehenden Nachrichten sammelt und die Abarbeitung nach dem FIFO-Prinzip ermöglicht. Abbildung 3.1 zeigt eine Übersicht über das Aktoren-Modell.

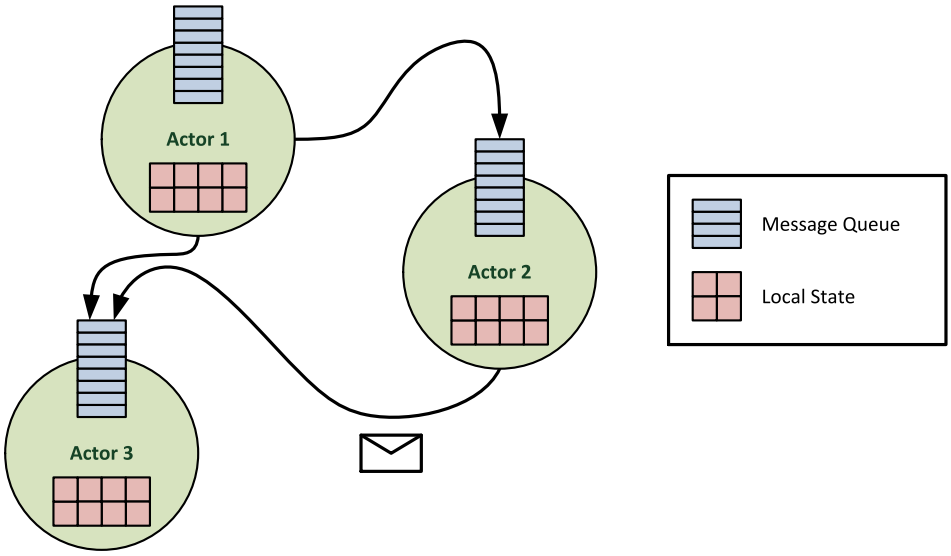


Abbildung 3.1: Aktoren-Modell: Jeder Aktor enthält eine eigene Nachrichtenwarteschlange, in welcher alle eingehenden Nachrichten gesammelt werden. Der Prozess des Aktors verarbeitet die Nachrichten nach dem FIFO-Prinzip und verwendet dazu nur seinen lokalen Speicher. Der gesamte Informationsaustausch zwischen den Aktoren findet über die Nachrichten statt.

Die implizite Vermeidung von Synchronisierungsproblemen mittels Message-Passing-Paradigma und die klare Abtrennung unterschiedlicher Aufgaben durch einzelne Aktoren, prädestiniert das Aktoren-Modell für den Einsatz in einer Videoverarbeitungspipeline.

3.1.1 Agenten in Microsoft C++

Neben der Parallel Pattern Library (PPL) enthält Visual Studio seit Version 2010 ebenfalls die Asynchronous Agents Library (AAL). Diese implementiert das Prinzip des Aktoren-Modells und verwendet die Concurrency Runtime, welche in der C-Runtime (CRT) enthalten ist, als Laufzeitumgebung. Einführungen in AAL [6] und in allgemeine Parallelisierungs-Themen mit Microsoft C++ [7] sind online verfügbar.

Durch die Verwendung von Aktoren und Nachrichten kann das Paradigma des „Control Flow“ durch das des „Data Flow“ ersetzt werden. Dies hat den Vorteil, dass der Programmierer sich im globalen Kontext nicht mit dem Warten auf Daten oder der Synchronisierung beschäftigen muss, sondern sich auf die Struktur des Datenflusses konzentrieren kann. Aus diesem Grund eignet sich das Aktoren-Modell sehr gut für die Konzeptionierung nebenläufiger Programme.

Die Agenten-Bibliothek in der PPL bezeichnet ihre Aktoren als asynchrone Agenten. Diese haben grundsätzlich dieselben Eigenschaften wie Aktoren.

- Agenten können Nachrichten an andere Agenten senden (`send`, `asend`).
- Agenten können Nachrichten für die Verarbeitung empfangen (`receive`, `try_receive`).
- Agenten können neue Agenten erzeugen.

Da die Agenten in C++ als Objekte implementiert sind, ist es nicht notwendig, einen zusätzlichen Weg für die Erzeugung von Objekten zur Verfügung zu stellen. Alle Agenten werden wie normale Objekte instanziert, was sowohl innerhalb eines Agenten oder in einem sonstigen Kontext des Programmes erledigt werden kann. Dies unterscheidet sich erheblich von reinen Aktor-basierten Sprachen wie z. B. Erlang, in welchen es nur den Kontext von Aktoren gibt, da bereits die `main`-Funktion durch einen Aktor dargestellt wird. Daraus ergibt sich als weiteren Unterschied, dass Nachrichten nicht an Adressen von Prozessen (Aktoren) gesendet werden, sondern direkt an das Agenten-Objekt. Betrachtet man diese Eigenschaft genauer, wird die Nachricht allerdings nicht an die Agenten, sondern an das `ITarget`-Objekt gesendet. Typischerweise ist dies ein Puffer-Objekt, welches Nachrichten sammeln kann. Um Nachrichten zu senden, muss das Puffer-Objekt zusätzlich ein `ISource`-Objekt sein. Diese beiden abstrakten Klassen bilden einen der Grundbausteine aller Klassen des Message-Passing-Systems in AAL und sind in Abbildung 3.2 ersichtlich.

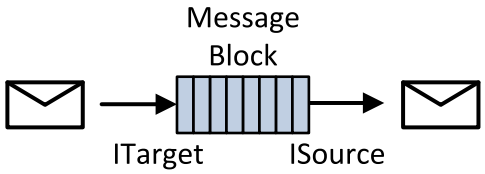


Abbildung 3.2: Der Messageblock kann Nachrichten über die `ITarget`-Schnittstelle empfangen und über die `ISource`-Schnittstelle versenden.

Das Message-Passing wird durch die sogenannten Messageblocks ermöglicht. Diese sind es, welche die eigentliche Asynchronität ins Spiel bringen und dadurch eine Datenfluss-basierte Implementation ermöglichen. Denn sie werden zwar ebenfalls durch Objektinstanzen verkörpert, ähneln den Aktoren aber mehr als die Agenten. Grund dafür ist, dass den Agenten fix ein Taskobjekt (abstrahierter Thread aus der PPL) zugeordnet wird, welches nur für die Ausführung der `run`-Methode des Agenten zuständig ist, in welcher die komplette Logik des Agenten implementiert wird. Diese enthält z. B. `receive`-Befehle, welche bei leerer Eingangswarteschlange warten und den Task somit schlafen legen. Abbildung 3.3 zeigt den Life-Cycle eines Agenten, welcher durch die Tatsache, dass er bei Bedarf schlafen gelegt wird und auf weitere Daten wartet, sich am Kontrollfluss-Pattern orientiert. Weitere Informationen [8] zur Klasse `agent` finden sich auf MSDN.

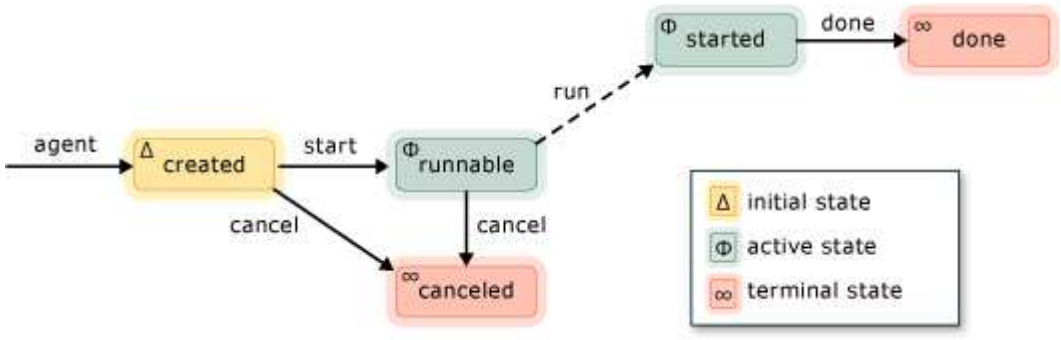


Abbildung 3.3: Life-Cycle eines Agenten aus der AAL. Nachdem der Agent instanziert wurde, wird er mittels `start`-Methode gestartet und läuft bis die `run`-Methode beendet oder die Ausführung von aussen abgebrochen wird. Die durchgezogenen Pfeile werden durch den User-Thread ausgeführt. Die Ausführung der `run`-Methode läuft im zugewiesenen Task, welcher von der CRT bereitgestellt wird. Bild aus [8].

Ein weiterer Unterschied der Agenten aus der AAL zum Aktoren-Modell ist die Tatsache, dass ein Agent keine eigene Nachrichtenwarteschlange besitzt, sondern die bereits erwähnten Messageblocks als Puffer benötigt. Somit muss zwischen zwei Agenten zwingend ein Messageblock sein, um Nachrichten zu versenden. Diese Blocks können allerdings vielseitig eingesetzt werden; sei es für bidirektionale Kommunikation, für Broadcasting oder anderes.

3.1.2 Message Blocks

Die Messageblocks stellen die Grundeinheiten der AAL dar. Sie ermöglichen das asynchrone Senden, Empfangen und Filtern von Nachrichten. Unter den Messageblocks befinden sich auch bereits Kombinationen aus Puffer und Agenten. Diese sogenannten `transformer`- und `call`-Klassen können nicht nur Nachrichten empfangen, sondern zusätzlich verarbeiten und, im Falle der `transformer`-Klasse, sogar weiterleiten. Das Klassendiagramm in Abbildung 3.4 zeigt die Abhängigkeiten der wichtigsten Messageblock-Klassen auf.

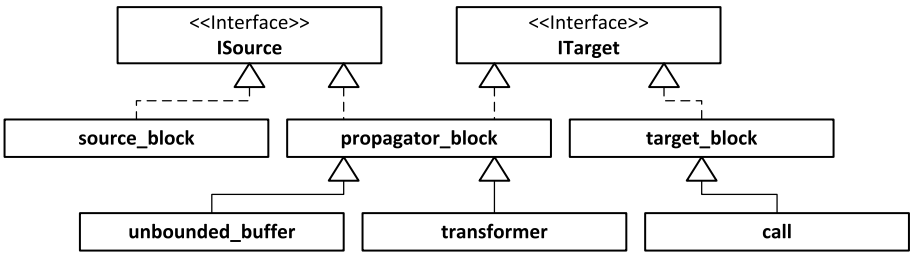


Abbildung 3.4: Übersicht über die relevanten Klassen der Messageblocks. Alle sind entweder von `ISource` oder `ITarget` oder beiden abgeleitet und können über diese Schnittstellen miteinander gekoppelt werden. Die `_block`-Klassen bilden Basisimplementationen, welche für eigene Messageblock-Implementationen verwendet werden können. Eine detailiertere Ansicht ist in Anhang C.1 verfügbar.

Eine Liste aller Messageblocks und auch die jeweils mögliche Anzahl an verbundenen `ISource`- und `ITarget`-Objekten ist in Tabelle 3.1 ersichtlich.

Name	Anzahl Sources	Anzahl Targets
unbounded_buffer	unbegrenzt	unbegrenzt
overwrite_buffer	unbegrenzt	unbegrenzt
single_assignment	unbegrenzt	unbegrenzt
call	unbegrenzt	-
transformer	unbegrenzt	1
choice	10	1
join	unbegrenzt	1
multitype_join	10	1
timer	-	1

Tabelle 3.1: Verfügbare Messageblocks und dessen Anzahl Sources beziehungsweise Targets. Die `call`- und `timer`-Klasse bilden eine Ausnahme, indem sie nur einen Typ der Messageblock-Schnittstellen implementieren. Ein `call` ist nur ein Target, welches allerdings unbegrenzte Sources haben kann. Der `timer` ist dagegen nur eine Source mit maximal einem Target.

3.2 Allgemeine Pipeline

Ein typischer Aufbau einer Videoverarbeitungs-Pipeline könnte wie in Abbildung 3.5 aussehen. In diesem Beispiel wird ein Frame eines Videostreams geladen, in ein Graustufenbild konvertiert, skaliert, ein Filter darauf angewendet und auf dem Bildschirm ausgegeben. Dieser Ablauf ändert sich für keines der Frames des gesamten Videos. Dabei ist es wichtig, dass die Bildreihenfolge in der letzten Stufe gleich wie am Anfang der Pipeline ist. Ob die Bilder innerhalb der Pipeline eine andere Reihenfolge einnehmen, spielt keine Rolle.

Abbildung 3.5: Typische Pipeline einer Videoverarbeitung. Die erste Stufe erzeugt oder lädt Daten. Diese werden in den folgenden Schritten verarbeitet und im letzten Schritt ausgegeben oder gespeichert.

Wird das gezeigte Beispiel verallgemeinert, entsteht Abbildung 3.6. Die Quelle am Anfang der Pipeline ist für die Erzeugung der Datenpakete zuständig, welche beim Durchlaufen der einzelnen Tasks verarbeitet werden. Die Tasks symbolisieren die logischen Einzelteile des Gesamtproblems, welches durch die Pipeline gelöst werden soll. Im Beispiel mit einem Videostream entsprechen die Datenpakete den einzelnen Videoframes, die durch die Tasks in mehreren Stufen verarbeitet werden. Die Senke am Ende jeder Pipeline empfängt das fertig verarbeitete Datenpacket und führt es seiner weiteren Verwendung zu.

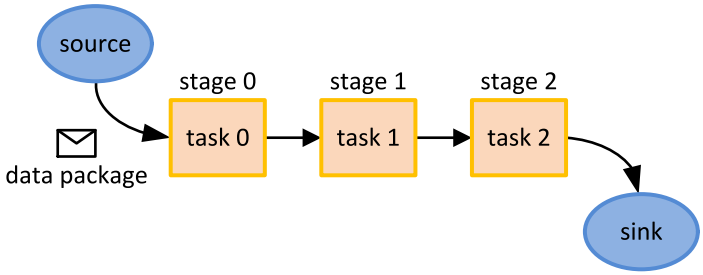


Abbildung 3.6: Allgemeine Struktur einer Pipeline. Diese enthält immer eine Daten-Quelle, ein oder mehrere Tasks und eine Daten-Senke.

3.2.1 Zeitliches Verhalten einer homogenen Pipeline

Um das Verhalten einer Pipeline bewerten zu können, werden in diesem Abschnitt unterschiedliche Eigenschaften einer Pipeline definiert. Genauere Betrachtungen sind im Paper [4] im Anhang E ersichtlich.

Ein Pipeline-System besteht aus n verfügbaren Recheneinheiten, welche durch die m vorhandenen Rechengeräte verwaltet werden. Ein Rechengerät ist z. B. die CPU, welche eine bestimmte Anzahl Recheneinheiten (die CPU-Kerne) zur Verfügung stellt. n beschreibt dabei die Gesamtanzahl aller Recheneinheiten aller Rechengeräte in einem System. Eine Pipeline hat jeweils p Stufen und muss k Datenpakete verarbeiten.

Die interessanten Eigenschaften einer Pipeline sind die Zeit l, die ein Paket beim Durchlaufen der Pipeline benötigt und das Ausgabeintervall x, welches die Zeit zwischen der Ausgabe zweier aufeinanderfolgenden Pakete beschreibt. Die mittlere Paketverzögerung wird mit grossem L angegeben und die individuellen Werte mit kleinem l inklusive Index, z. B. l_0, welches die Verzögerung des ersten Pakets (initiale Verzögerung) bezeichnet. Dasselbe gilt für das Ausgabe-Intervall, wobei $x_{0,1}$ die Zeit zwischen der Ausgabe des ersten und des zweiten Pakets bezeichnet. Die Anzahl Pakete, welche insgesamt verarbeitet werden sollen, wird mit k definiert und das minimale Eingangsintervall, welches von der Datenquelle abhängig ist, wird mit t bezeichnet. Weitere Messwerte der Pipeline sind die Ausgabevarianz σ^2, welche in Formel 3.1 definiert ist, und die Gesamtausführungszeit R des Systems.

$$\sigma^2 = \frac{1}{k-1}\sum_{i=0}^{k-2}(x_{i,i+1} - \mu)^2\,, \qquad \mu = \frac{1}{k}\sum_{i=0}^{k-2} x_{i,i+1}$$

Formel 3.1: Die Berechnung der Ausgangsvarianz σ^2, welche die Fähigkeit des Systems beschreibt, einen regelmässigen Ausgabefluss zu erzeugen. k ist die Anzahl zu verarbeitende Pakete, x die Zeit zwischen der Ausgabe von aufeinanderfolgenden Paketen und μ der Erwartungswert.

Erzeugt die Quelle ein einzelnes Datenpacket, wird dieses beim Durchlaufen der einzelnen Stufen genau so viel Zeit benötigen, wie die Ausführung eines sequentiellen Systems mit derselben Aufgabenstellung wie die Tasks der Pipeline. Definiert man die Dauer, welche die einzelnen Stufen zur Verarbeitung eines Pakets benötigen mit der Menge $S = \{s_0, s_1, s_2\}$ und die Recheneinheiten mit der Menge $C = \{c_0, c_1, \dots, c_{n-1}\}$, ergibt sich eine Verzögerungs- bzw. Intervallzeit von $x_{0,1} = l_0 = \sum_i s_i$. Deshalb gilt auch: $x_{i,i+1} = X = L$. Abbildung 3.7 a) zeigt eine solche Ausführung.

Ein System das mehr als eine Recheneinheit zur Verfügung hat, sollte möglichst alle davon verwenden, um die Ausführung zu beschleunigen. Da bei einer typischen Pipeline mehrere Datenpakete verarbeitet werden müssen, kann das zweite Paket bereits in der ersten Stufe verarbeitet werden, wenn das erste Paket zur zweiten Stufe weitergeleitet wurde. Diese Pipeline-Parallelität ermöglicht es zwar nicht die initiale Wartezeit l_0 zu reduzieren, allerdings kann dadurch das Ausgabeintervall x verkürzt werden. So kann ein System mit gleich vielen Recheneinheiten wie Stufen, also $n = |S|$, in einem Intervall von $X = \max(S)$ fertig bearbeitete Datenpakete ausgeben. In Abbildung 3.7 b) kann erkannt werden, dass l_0 nicht verkürzt werden kann, aber X nur noch der Dauer von s_1 entspricht. Es kann wegen den Abhängigkeiten zwischen den Stufen kein Beschleunigungsfaktor von n erreicht werden. Zudem verändern sich die Verzögerungen der einzelnen Pakete. Da das Ausgabeintervall länger ist als das Eingangsintervall, gilt $l_0 < l_1 < l_2$, wodurch die Verzögerungen gegen unendlich gehen, solange die Quelle nicht künstlich begrenzt wird.

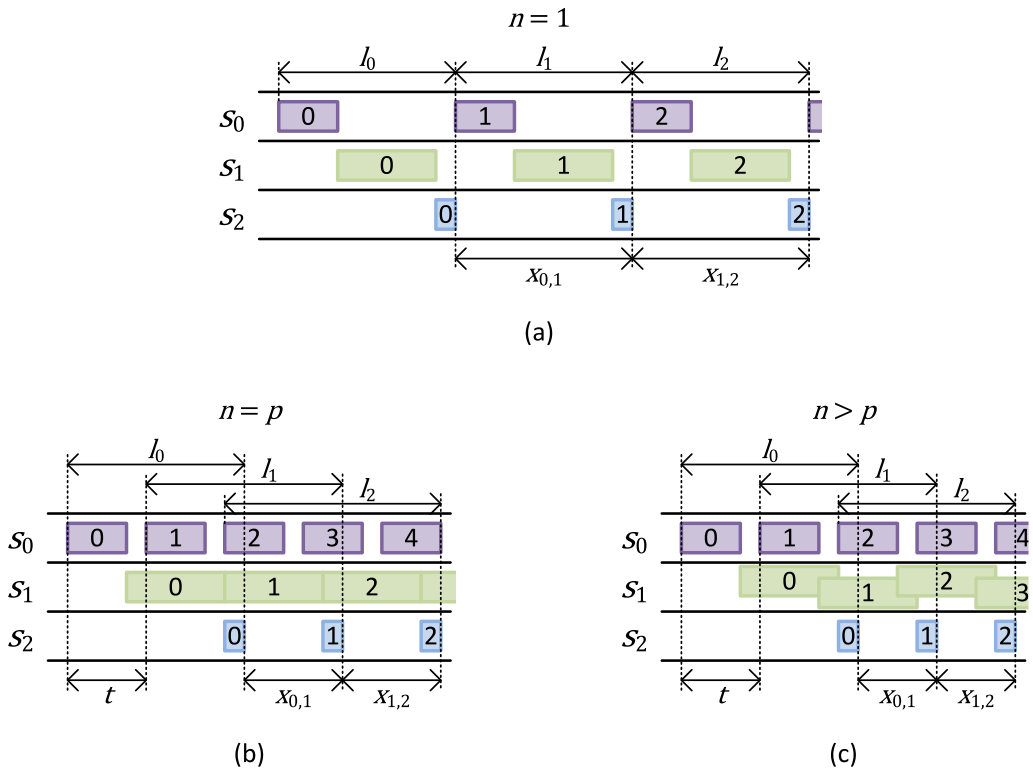


Abbildung 3.7: Ablaufdiagramm bei der Verarbeitung einer Pipeline. (a) Sequentielle Ausführung einer Pipeline. (b) Jede Stufe kann parallel zu den anderen Stufen Daten verarbeiten. (c) Jede Stufe kann zusätzlich mehrere Daten parallel verarbeiten. In diesem Beispiel wird angenommen, dass dem System vier Recheneinheiten zur Verfügung stehen. Die Stufe 0 kann jeweils nicht schneller werden, da sie durch das t der Quelle begrenzt wird.

Um diese Problematik zu beheben und das System weiter zu beschleunigen, müssen die einzelnen Stufen mehrere Datenpakete parallel verarbeiten können. Durch diese Task-Parallelität kann die Begrenzung der Stufe s_1 aufgehoben werden. Wie in Abbildung 3.7 c) ersichtlich ist, kann mit vier Recheneinheiten das Ausgabeintervall auf $t = x_{0,1} = x_{1,2}$ und die Verzögerungen auf $l_0 = l_1 = l_2$ verkürzt werden.

3.2.2 Implementierung einer heterogenen Pipeline

Für die Implementierung einer Pipeline wird die AAL [8] verwendet. Dadurch ist die Abstraktion der Stufen und Datenpakete bereits durch die Messageblocks und die Nachrichten gegeben. Eine Pipeline mit drei Stufen kann wie in Abbildung 3.8 aussehen.

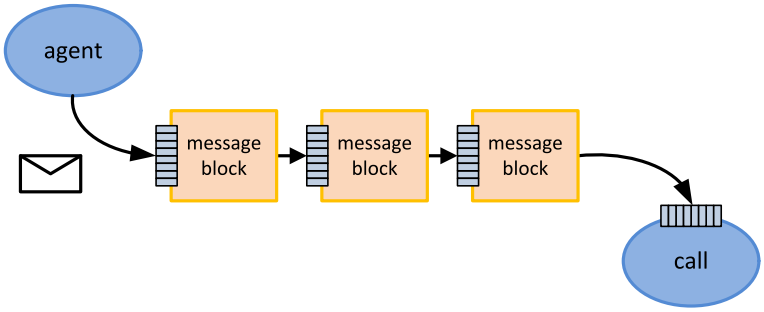


Abbildung 3.8: Pipeline-Implementation mittels AAL. Die `agent`-Klasse übernimmt die Rolle der Quelle, die `call`-Klasse die Rolle der Senke und ein Messageblock mit `ITarget`- und `ISource`-Schnittstelle die Rolle der einzelnen Stufen.

Da die AAL die Concurrency Runtime verwendet, um die Synchronisation und Verwaltung der Threads zu erledigen (siehe Anhang C.2), muss sich darüber keine Gedanken gemacht werden. Um aber ein heterogenes System durch die Pipeline abbilden zu können, müssen die Messageblocks der einzelnen Stufen erweitert werden. In einem Beispiel-System, welchem die zwei Rechengeräte CPU und GPU zur Verfügung stehen, sieht eine heterogene Stufen wie in Abbildung 3.9 aus.

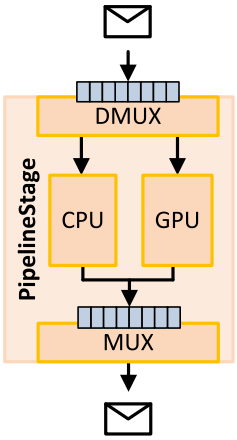


Abbildung 3.9: Heterogene Pipeline-Stufe, die zwei Rechengeräte verwaltet. Der Demultiplexer (DMUX) wählt die Ressource, welche zur Verarbeitung der Nachricht verwendet wird. Der Multiplexer (MUX) sammelt alle fertiggestellten Datenpakete.

Der Demultiplexer spielt bei der Verwaltung der unterschiedlichen Ausführungs-Ressourcen eine wichtige Rolle. Weil dieser das Datenpaket dem entsprechenden Rechengerät weiterleitet, muss er wissen, welches Gerät freie Recheneinheiten hat und die beste Performance für die entsprechende Aufgabe bietet. Diese Entscheidung fällt er jeweils in Kooperation mit einem weiteren Element, welches die Rechengeräte verwaltet (siehe Abschnitt 3.3). Der Multiplexer ist für das Sammeln der bearbeiteten Datenpakete und das freigeben verwendeter Ressourcen zuständig.

Um die Entscheidung des Demultiplexers zu ermöglichen, soll das System ermitteln können, welche Implementierung priorisiert werden soll. Dazu existiert einerseits die Möglichkeit, dass der Nutzer beim Erstellen der Pipeline definiert, welche Variante der Stufe zu priorisieren ist. Andererseits kann die Pipeline die Datenlokalität erzwingen, indem sie versucht, das gleiche Rechengerät wie in der vorherigen Stufe zu verwenden. Die zweite Möglichkeit verhindert unnötiges Umherkopieren der Daten zwischen unterschiedlichen Rechengeräten.

Allgemein soll eine Scheduling-Strategie entwickelt werden, die folgende Ziele verfolgt.

- Non-preemptive.
- Fairness & Durchsatz maximieren.
- Implementierungen priorisierbar.

Die Anforderung für nicht-unterbrechendes Scheduling ist einerseits dadurch begründet, dass es dem Nutzer vereinfacht werden soll, Task-Implementationen für die einzelnen Stufen zu schreiben. Andererseits unterstützt diese Eigenschaft eine Echtzeitausführung, die eng mit einer Live-Stream-Anwendung verwandt ist. Die Fairness und der Durchsatz sollen ermöglichen, dass alle Tasks regelmässig Rechenressourcen erhalten und dass möglichst viele Daten verarbeitet werden können. Die Priorisierung der einzelnen Implementationen pro Stufe stellt sicher, dass wenn möglich das schnellste Rechengerät für die Ausführung verwendet wird.

Mit der Einführung von heterogenen Pipelines entstehen zwei neue Eigenschaften. Eine gleichförmig beschleunigte Pipeline zeichnet sich durch die Tatsache aus, dass ein Rechengerät in allen Stufen die schnellste Ausführung hat, ein weiteres die zweitschnellste und so weiter. Somit wird ein einzelnes Paket optimal nur durch das schnellste Rechengerät verarbeitet. Typischerweise bildet ein System mit CPU und GPU eine gleichförmig beschleunigte Pipeline, da die Tasks, welche eine GPU-Implementation aufweisen,

schneller auf der GPU als auf der CPU ausgeführt werden können. Eine ungleichförmig beschleunigte Pipeline würde somit Stufen enthalten, deren Tasks auf dem Gerät A schneller wären, aber auch Stufen die auf dem Gerät B weniger Zeit für die Ausführung benötigen würden.

Eine Pipeline ist vollständig, wenn in jeder Stufe eine Task-Implementation für jedes Rechengerät existiert. Somit würde eine Pipeline mit acht Stufen und zwei Rechengeräten 16 unterschiedliche Task-Implementationen beinhalten. In einer unvollständigen Pipeline existieren hingegen Stufen, in welchen nicht für alle Rechengeräte eine Task-Implementation vorhanden ist. Das Beispiel mit einer CPU und einer GPU wäre typischerweise eine unvollständige Pipeline, da es nicht bei allen Stufen möglich ist oder Sinn macht, eine GPU-Implementation bereitzustellen. Die allgemeinste Art, welche aber am schwierigsten zu verwalten ist, ist eine ungleichförmige und unvollständige Pipeline.

3.3 Scheduling einer heterogenen Pipeline

Die Task-Verarbeitung eines einzelnen Frames im Videostream wird als Job bezeichnet. Diese Jobs sollen von den vorhandenen Rechengeräten in möglichst kurzer Zeit abgearbeitet werden. Da das System nicht weiss, wie lange die Ausführung eines Jobs auf einem spezifischen Rechengerät dauert und in einer Live-Anwendung nicht alle Jobs beim Starten der Bearbeitung bekannt sind, spricht man von einem Non-Clairvoyant Online-Scheduling. Im HSA-Kontext kommt zusätzlich die Heterogenität der Rechengeräte hinzu.

In den folgenden Abschnitten werden drei unterschiedliche Implementationen beschrieben, welche die in Abschnitt 3.2 beschriebenen Probleme und das heterogene Online-Scheduling unterschiedlich zu lösen versuchen.

3.3.1 Sofortiges Scheduling

Wird die Aufgabenstellung aus der Sicht eines agentenbasierten Systems betrachtet, ergibt sich die einfachste Lösung, indem der Demultiplexer beim Eintreffen eines neuen Datenpakets das beste freie Rechengerät wählt und die Daten sofort dorthin weiterleitet. Dieser Ansatz verhindert das Warten auf eine freie Recheneinheit eines besseren Geräts, bietet aber das Risiko, dass häufig ein schlechteres Rechengerät gewählt wird, obwohl demnächst eine besser geeignete Ressource zur Verfügung stehen würde.

Für die Verwaltung der Rechengeräte wird ein `PriorityGovernor` verwendet. Eine Instanz dieser Klasse kann mehrere Recheneinheiten (Slots) des entsprechenden Geräts verwalten. Um die Recheneinheiten eines solchen Governors anzufordern, kann die Methode `tryAcquireSlot` verwendet werden. Der `PriorityGovernor` implementiert das Observer-Pattern, um auf frei gewordene Slots aufmerksam zu machen. Dabei kann sich ein `IObserver` beim Governor für Benachrichtigungen registrieren. Wenn ein Slot durch Aufrufen der Methode `releaseSlot` freigegeben wird, wird derjenige `IObserver` benachrichtigt, welcher sich als erstes beim Governor registriert hat.

Die Implementation eines Tasks für ein bestimmtes Rechengerät wird als `transformer`-Instanz an das `ExecutionPath`-Objekt übergeben. Dieses wird durch die `PipelineStage_Instant`-Implementation verwaltet. Typischerweise wird für jeden Typ von verfügbaren Rechengeräten ein `ExecutionPath` erstellt, welcher vom jeweiligen `PriorityGovernor` die benötigten Slots anfordern kann. Abbildung 3.10 b) zeigt den Aufbau einer einzelnen Stufe in einem solchen System.

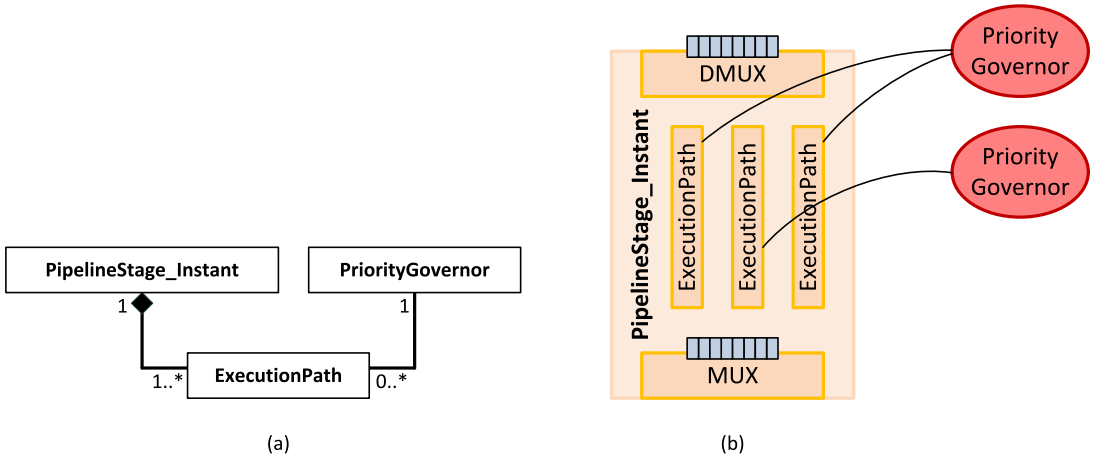


Abbildung 3.10: Aufbau einer Stufe mit der `PipelineStage_Instant`-Implementation. Ein `ExecutionPath` ist genau einem `PriorityGovernor` zugeordnet, wobei einer `PipelineStage` mehrere Pfade desselben Governors hinzugefügt werden können. (a) Abhängigkeiten der einzelnen Objekte. (b) Exemplarische Darstellung einer Stufe.

Wie in Abbildung 3.10 b) ersichtlich ist, können mehrere `ExecutionPaths` desselben Rechengeräts hinzugefügt werden. Dies ermöglicht die mehrfache Ausführung eines Tasks auf demselben Gerät und somit den in Abbildung 3.7 c) beschriebenen Fall. Dieses Vorgehen ist notwendig, da die Tasks durch `transformer`-Objekte implementiert werden und diese jeweils nur eine Nachricht gleichzeitig verarbeiten können. Mittels Mehrfachpfaden können begrenzende Stufen gezielt von Hand optimiert werden. Um jeder Stufe zu ermöglichen, alle verfügbaren Recheneinheiten eines Geräts auszunutzen, können in jeder Stufe so viele `ExecutionPaths` wie Slots pro Rechengerät hinzugefügt werden.

Wird ein Datenpaket vom Demultiplexer empfangen, beginnt das Scheduling und danach die Abarbeitung der Daten. Abbildung 3.11 zeigt den Ablauf, wenn ein verwendeter Pfad frei und sofort wieder belegt wird. Dabei registriert sich die `PipelineStage` als erstes bei allen besetzten Pfaden (`ExecutionPath` 2) und bei allen Governors (`PriorityGovernor` 1), dessen Pfade aktuell frei wären. Wird der besetzte Pfad frei, wird die Benachrichtigung deaktiviert und die `PipelineStage` registriert sich bei dessen Governor (`PriorityGovernor` 2). Da dieser zu diesem Zeitpunkt über einen freien Slot verfügt, benachrichtigt er sofort den ersten registrierten Observer.

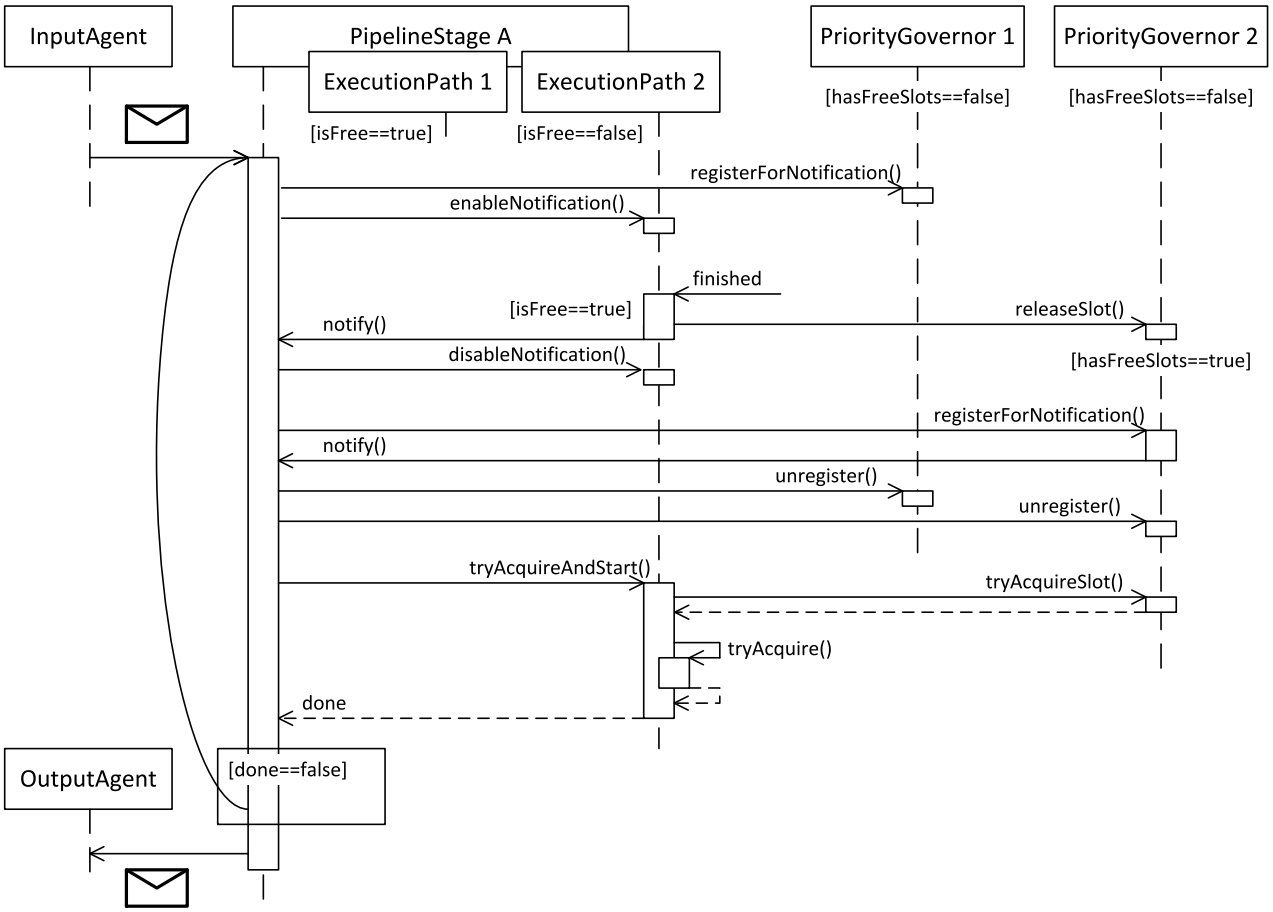


Abbildung 3.11: Sequenzdiagramm des sofortigen Schedulings. Die Stufe registriert sich bei den interessanten Governors und fordert nach der Benachrichtigung einen Slot an. Der `ExecutionPath` 1 bezieht Slots vom `PriorityGovernor` 1, der `ExecutionPath` 2 vom `PriorityGovernor` 2.

Der beschriebene Ablauf kann in einem nebenläufigen System variieren und ergibt deshalb unterschiedliche Bedingungen für den Start der Verarbeitung. Nach der Benachrichtigung für einen freien Slot, prüft die `PipelineStage` deshalb alle anderen Governors auf Slots, die gleichzeitig frei geworden sind.

Die Governors mit freien Slots werden gemäss einer vom Datenpaket definierten Policy sortiert. Dabei werden alle Governors, die in der Policy-Liste vorkommen, an den Anfang der Liste mit möglichen Governors gestellt. Alle anderen werden in derselben Reihenfolge, wie dessen Benachrichtigungen eingetroffen sind, angehängt. Nach dem Anwenden der Policy auf der Liste, wird jeder Governor in der Liste mittels `tryAcquireSlot`-Methode angefragt, bis entweder ein Slot gewährleistet oder die Liste komplett abgearbeitet wurde. Danach wird entweder die Verarbeitung gestartet, oder der komplette Ablauf von neuem begonnen.

Nachdem die Verarbeitung im definierten `transformer` beendet wurde, wird die Nachricht an den Multiplexer der `PipelineStage` weitergeleitet. Dort wird als erstes der Slot des Governors und anschliessend der Pfad der `PipelineStage` freigegeben. Diese Reihenfolge ist zwingend gleich wie beim Anfordern im Demultiplexer zu wählen, da es ansonsten zu Deadlocks kommen kann. Des Weiteren sortiert der Multiplexer bei Bedarf die erhaltenen Nachrichten nach kleinstem Index.

Dass nach dem Erhalten eines Slots geprüft wird, ob der Pfad nicht bereits wieder benutzt wurde, ist allerdings unnötig. Dies ergibt sich einerseits dadurch, dass die Pfade nur vom besitzenden `PipelineStage`-Objekt angesprochen werden können (siehe Abbildung 3.10 a)) und andererseits durch die Tatsache, dass der Demultiplexer nicht nebenläufig implementiert ist. Das Sammeln und Sortieren von möglichen Governors ist dazu gedacht, dass immer das bestmögliche Rechengerät verwendet wird. Da allerdings keine

Wartezeit auf bessere Geräte zugelassen wird und typischerweise mehr wartende Datenpakete als freie Slots existieren, funktioniert dieses System nur in seltenen Fällen.

3.3.2 Zurückhaltendes Scheduling

Um die negativen Eigenschaften der Scheduling-Strategie aus Abschnitt 3.3.1 zu verbessern, soll verhindert werden, dass eine Stufe einen Slot verwendet, welcher eigentlich besser für eine andere Stufe geeignet wäre. Zudem wird der `ExecutionPath` neu implementiert, um direkt mehrere Pakete gleichzeitig verarbeiten zu können. Für die Verwaltung der Rechengeräte kommt der bereits erwähnte `PriorityGovernor` zum Einsatz.

Abbildung 3.12 zeigt schematisch wie die einzelnen Bestandteile zusammenarbeiten. Die wichtigste Änderung ist, dass jeder `PriorityGovernor` nur noch einen `ExecutionPath` pro Stufe haben kann. Dies ist durch die neue Multithreading-Fähigkeit der Task-Implementation begründet. Dabei werden die Task-Implementationen nicht mehr als `transformer`-Objekt, sondern als `function`-Objekt übergeben. Im Vergleich zur `transformer`-Klasse erledigt die `function`-Klasse keine Thread-Verwaltung, sondern stellt nur das Design einer Aufgabe dar. Damit diese Aufgabe ausgeführt werden kann, muss diese durch einen Thread gestartet werden. Dies kann z. B. sehr einfach mit der `task`-Klasse aus der PPL bewerkstelligt werden. Eine Instanz einer solchen Klasse verwendet einen von der Concurrency Runtime [9] angeforderten Thread, um ein mitgeliefertes `function`-Objekt auszuführen.

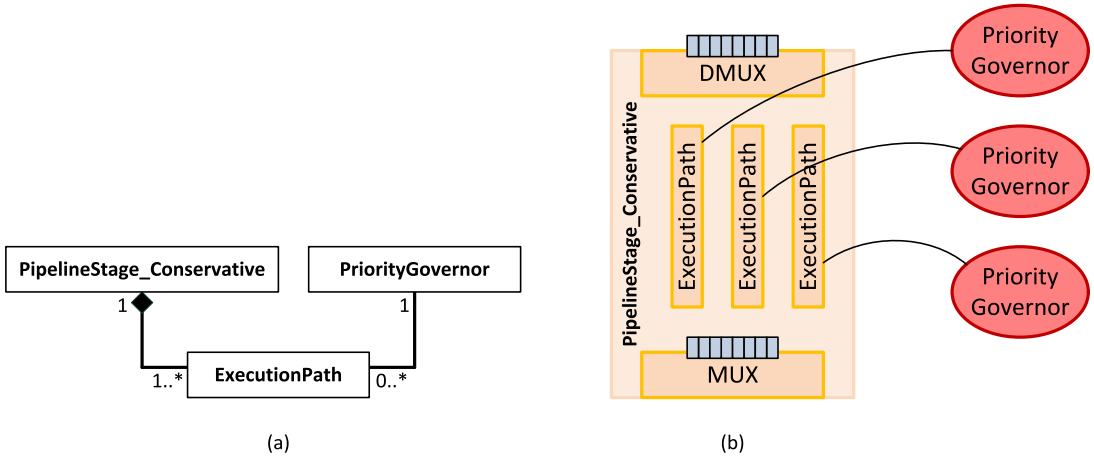


Abbildung 3.12: Aufbau einer Stufe mit der `PipelineStage_Conservative`-Implementation. Ein `ExecutionPath` ist genau einem `PriorityGovernor` zugeordnet, wobei einer `PipelineStage` nur jeweils ein Pfad desselben Governors hinzugefügt werden kann. (a) Abhängigkeiten der einzelnen Objekte. (b) Exemplarische Darstellung einer Stufe.

Das Scheduling ähnelt der zuvor vorgestellten Strategie. Die Stufe registriert sich direkt bei allen Governors, da das angesprochene Überprüfen auf einen freien Pfad wegfällt. In Abbildung 3.13 meldet sich der nicht priorisierter Governor 2 bei der `PipelineStage` und offeriert dieser einen Slot. Da er sich zum ersten Mal meldet, weist die Stufe den Slot zurück und lässt den Governor andere wartende Stufen benachrichtigen. Im Idealfall würde sich ein weiterer, eventuell bevorzugter, Governor bei der Stufe melden, welche den offerierten Slot bereits beim ersten `notify` annehmen würde. Da dies im Beispiel aber nicht der Fall ist und der `PriorityGovernor` 2 den Slot zudem keiner anderen Stufe anbieten kann, benachrichtigt er die Stufe A erneut.

Die `PipelineStage` merkt sich alle abgewiesenen Governors und nimmt einen offerierten Slot an, wenn der nicht priorisierte Governor bereits in dieser Liste enthalten ist. Im Beispiel tritt dieser Fall ein, wenn der Governor 2 die Stufe A zum zweiten Mal benachrichtigt. Da der Slot aber erst beim Startversuch besetzt wird, kann es passieren, dass er bereits von einem andern Teilnehmer belegt wird, bevor die Anfrage abgesetzt wurde.

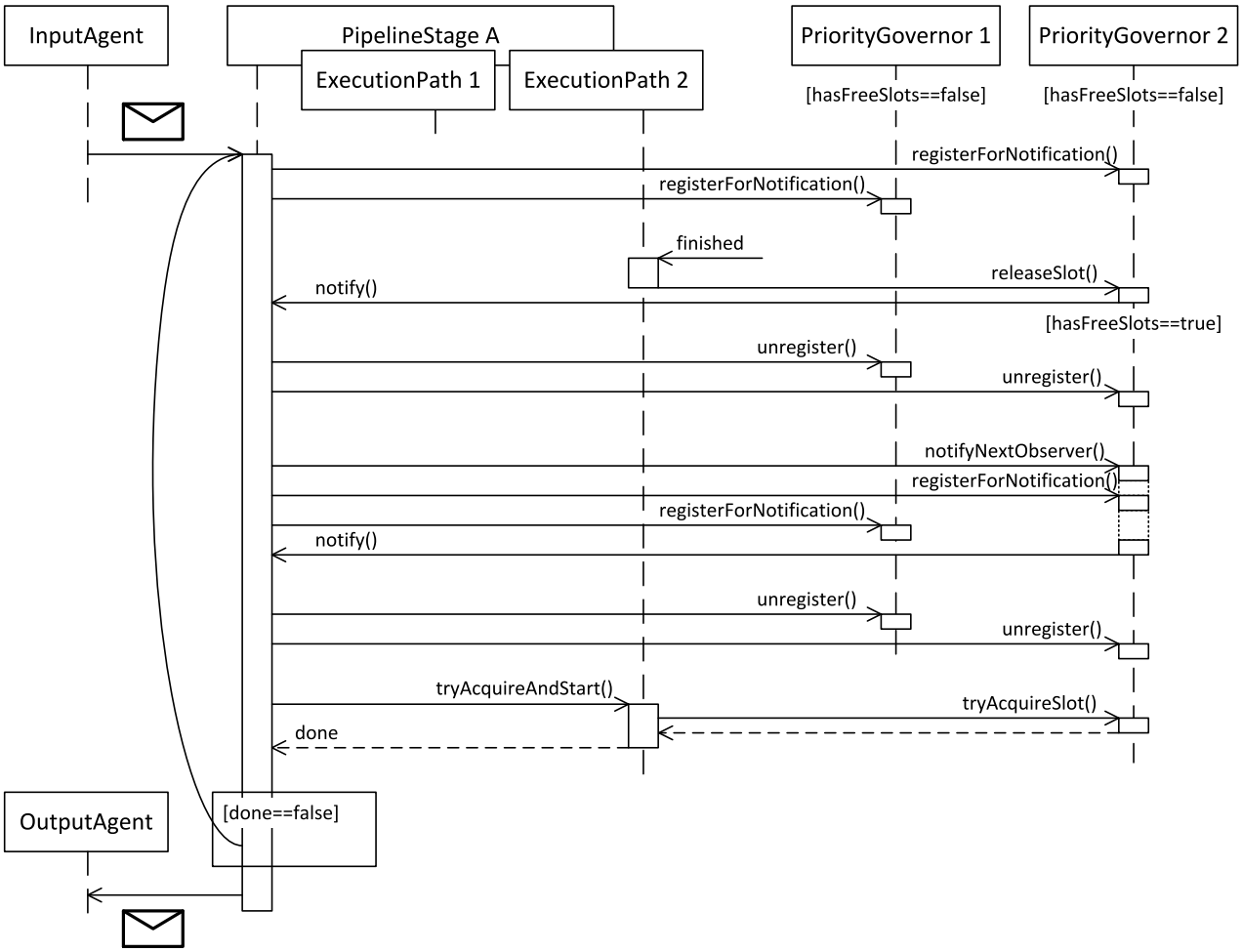


Abbildung 3.13: Sequenzdiagramm des zurückhaltenden Schedulings. Die Stufe registriert sich bei allen Governors und fordert nach der zweiten Benachrichtigung eines nicht bevorzugten Governors oder bei der ersten Benachrichtigung eines priorisierten Governors einen Slot an. Der `ExecutionPath` 1 bezieht Slots vom `PriorityGovernor` 1, der `ExecutionPath` 2 vom `PriorityGovernor` 2.

Diese Scheduling-Strategie ist in der Theorie besser, da sie versucht, nur priorisierte Ressourcen zu verwenden. Da ein Pipelinesystem aber typischerweise mehr wartende Stufen als freie Slots aufweist, tritt der optimale Fall nur sehr selten ein. Zudem verursacht das Ablehnen von Slots, wonach jedes Mal ein neuer Anfragedurchlauf bei allen Governors gestartet werden muss, einen beträchtlichen Overhead im Vergleich zur Strategie des sofortigen Startens.

3.3.3 Warteschlangebasiertes Scheduling

Das Problem, welches bei beiden bisher vorgestellten Strategien besteht, ist, dass nachdem sich eine `PipelineStage` dazu entschieden hat, einen Slot anzunehmen, dieser bereits vergeben sein kann. Hinzu kommt, dass das in Abschnitt 3.3.2 eingeführte Ablehnen von nicht bevorzugten Rechenressourcen nicht zu den gewünschten Ergebnissen führt. Deshalb ist die dritte Scheduling Strategie grundlegend anders gestaltet und verzichtet auf den `PriorityGovernor`.

Daher wird der `DeviceScheduler` für das Management der Ressourcen verwendet. Dieser verwaltet ebenfalls Slots eines einzelnen Rechengeräts, bietet diese aber nicht für Konsumenten an, sondern startet direkt registrierte Jobs in der jeweiligen `PipelineStage`. Wie in Abbildung 3.14 ersichtlich, ändert sich am Zusammenspiel nichts, ausser das die Governors durch die `DeviceScheduler` und die Pfade durch die `StageExecutors` ersetzt werden.

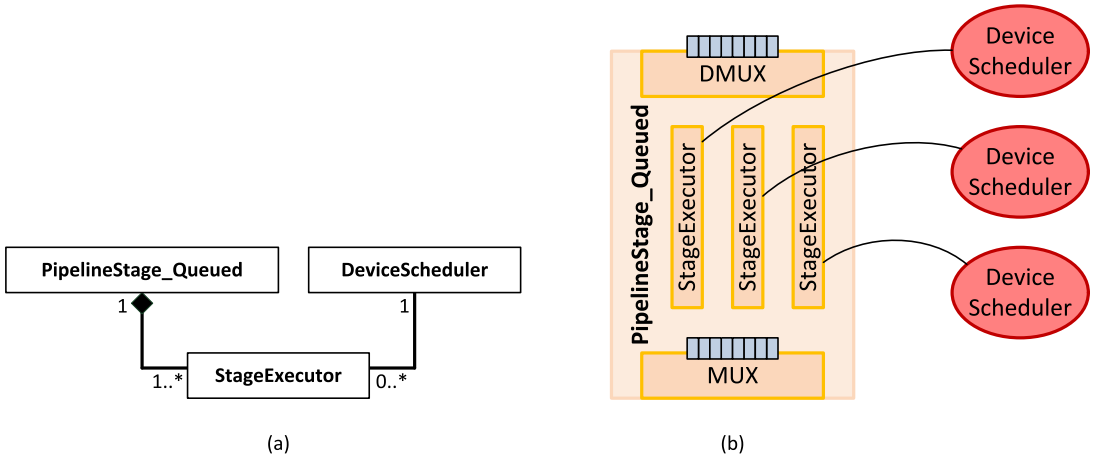


Abbildung 3.14: Aufbau einer Stufe mit der `PipelineStage_Queued`-Implementation. Ein `StageExecutor` ist genau einem `DeviceScheduler` zugeordnet, welcher den Job direkt im Executor startet. (a) Abhängigkeiten der einzelnen Objekte. (b) Exemplarische Darstellung einer Stufe.

Im Gegensatz zum `ExecutorPath` wird im `StageExecutor` nicht versucht, nach einer Benachrichtigung einen Slot zu akquirieren und danach den Task zu starten, sondern der Scheduler startet den Task direkt.

Obwohl der `DeviceScheduler` ebenfalls Slots verwaltet, die den freien Recheneinheiten eines Rechengeräts entsprechen, unterscheidet sich seine Implementation erheblich von derjenigen des `PriorityGovernors`. Um einen Slot für die Verarbeitung von Daten zu erhalten, kann die Stufe über die Methode `scheduleJob` einen Ausführungsjob beim Scheduler registrieren. Dieser wird intern in eine Warteschlange eingereiht und ausgeführt, sobald der Job an erster Stelle und ein Slot verfügbar ist. Für diese Funktionalität wird ebenfalls das Observer-Pattern verwendet, wobei dem Scheduler ein `IJobExecutor`-Objekt übergeben wird, welches die Methode `startJob` besitzt. Diese Schnittstelle wird von der `PipelineStage` implementiert und ermöglicht es dem Scheduler die Verarbeitung des Datenpakets zu starten.

Beim Startvorgang überprüft die Stufe als erstes, ob das Datenpaket noch bearbeitet werden muss und startet in diesem Fall die Verarbeitung mittels der Methode `startAsyncProcessing` des `StageExecutors`. Dieser ist seinerseits für die Freigabe des verwendeten Slots nach der Verarbeitung zuständig, indem er die Methode `releaseSlot` des Schedulers aufruft. Wenn das Datenpaket nicht mehr im Demultiplexer der Stufe vorhanden ist, wurde es bereits verarbeitet und die Methode `startJob` liefert dem Scheduler `false` zurück. Daraufhin löscht dieser den Job aus seiner Warteschlange und versucht den nächsten auszuführen. Diese Vorgänge sind in Abbildung 3.15 illustriert.

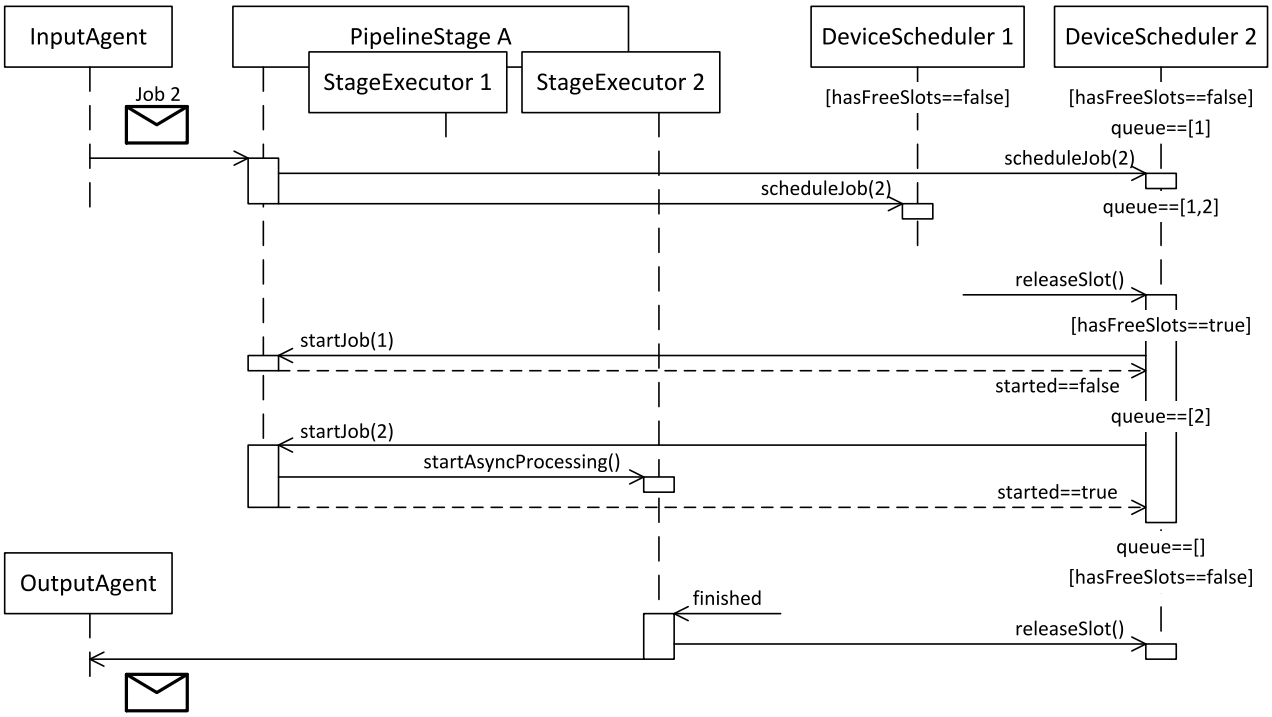


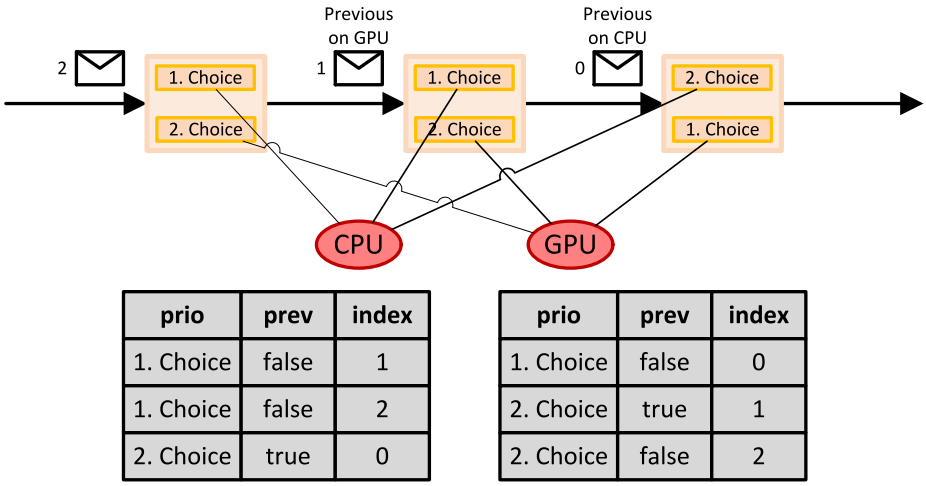
Abbildung 3.15: Sequenzdiagramm des warteschlangebasierten Schedulings. Die Stufe registriert einen anstehenden Job bei allen Schedulern. Ein solcher startet die Datenverarbeitung über die Stufe im entsprechenden Executor. Der `StageExecutor` 1 wird vom `DeviceScheduler` 1, der `StageExecutor` 2 vom `DeviceScheduler` 2 gestartet.

Die entscheidende Neuerung in diesem System ist die Wartschlange des `DeviceSchedulers`. Sie ermöglicht es, die Ziele aus der Einleitung von Abschnitt 3.2.2 durch gezieltes Umordnen der Jobs innerhalb der Warteschlange zu erreichen. Dazu werden zusätzlich folgende Daten beim `scheduleJob`-Aufruf übergeben.

1. Priorität des Executors dieser Stufe.
2. Ob die Daten in der vorherigen Stufe ebenfalls auf dem Rechengerät dieses Schedulers verarbeitet worden sind.
3. Der Index des Datenpakets.

Die Warteschlange wird als erstes nach der Executor-Priorität sortiert. Der Wert 1 bezeichnet dabei den zuerst zu wählenden, der Wert 2 den als zweites zu wählenden Executor, usw. Task-Implementationen die nicht priorisiert werden, erhalten den Maximalwert eines Integers. Alle Jobs, die denselben Priorisierungswert haben, werden auf Datenlokalität optimiert. Wenn die Daten bereits in der vorherigen Stufe auf dem Rechengerät verarbeitet wurden, wird der Job auf diesem Scheduler bevorzugt. Ist dieser Wert bei zwei Jobs ebenfalls der gleiche, wird derjenige mit dem niedrigeren Index priorisiert, da die Datenpakete aufsteigend nummeriert werden und somit ein älteres Paket bevorzugt wird.

Ein Beispiel einer dreistufigen Pipeline ist in Abbildung 3.16 dargestellt. Darin ist ersichtlich, dass in der CPU-Warteschlange die Gerätepriorität des Pakets 2 gegenüber der Datenlokalität des Pakets 0 bevorzugt wird. Des Weiteren wird das Paket 1 dem Paket 2 vorgezogen, da dieses anhand des niedrigeren Index als erstes verarbeitet werden soll. In der GPU-Warteschlange wird das Paket 1 vor dem Paket 2 ausgeführt, da so die Datenlokalität ermöglicht wird. Zufälligerweise sind in diesem Beispiel die Indizes in der GPU-Warteschlange geordnet.



prio	prev	index
1. Choice	false	1
1. Choice	false	2
2. Choice	true	0

prio	prev	index
1. Choice	false	0
2. Choice	true	1
2. Choice	false	2

Abbildung 3.16: Einfaches Beispiel mit drei wartenden Datenpaketen bei drei Stufen mit zwei Rechengeräten. Die Tabellen zeigen die Wartschlangen der `DeviceScheduler` nachdem alle Jobs geplant aber noch nicht ausgeführt wurden. Die Ausführung startet oben in der Tabelle.

Durch die Verwendung einer optimierenden Warteschlange kann die Ausführung der Tasks verbessert werden. Dies vor allem durch die zusätzlichen Informationen die dem Scheduler mitgeteilt werden. Trotzdem existiert ein Kompromiss zwischen einem einfachen Design der Scheduler und einer globalen Warteschlange, welche alle Geräte und alle `StageExecutors` verwalten würde.

Ein Spezialfall einer Ausführung stellt ein nicht belastetes System dar. In einem solchen System sind alle Warteschlangen der Scheduler leer, was dazu führt, dass der Job beim Aufruf der `scheduleJob`-Methode sofort gestartet wird. Dadurch wird verhindert, dass die Ausführung des Jobs durch die Sortierung der Warteschlange auf dem besten geeigneten Gerät gestartet wird. Um dies in diesem Fall trotzdem zu ermöglichen, werden alle `StageExecutors` während dem Einstellen der Prioritäten nach diesen sortiert, wodurch ein neuer Job einer Stufe immer als erstes beim priorisierten `DeviceScheduler` registriert und somit sofort gestartet wird.

3.3.4 Analyse der Implementationen

Die drei Implementationen wurden in der vorgestellten Reihenfolge entwickelt und haben deshalb gewisse historische Abhängigkeiten. Tabelle 3.2 listet die wichtigsten Eigenschaften der unterschiedlichen Systeme auf.

	Instant	Conservative	Queued
Pro Stufe können mehrere Pakete gleichzeitig auf demselben Gerät verarbeitet werden.	Nein	Ja	Ja
Pro Stufe können mehrere Pakete gleichzeitig auf Verarbeitung warten.	Nein	Nein	Ja
Executor-Auswahlkriterien	1. First-Come-First-Serve (2. Policy)	1. Policy 2. Zweite Slot-Offerte	1. Executor-Priorität 2. Vorherige Ausführung 3. Paket-Index
Form der Scheduling Queue	Keine Queue	Globale Queue. Zusammengesetzt aus den Demuxer aller Stufen.	Lokal für jeden Scheduler. Echte Queue.
Verteilung der Recheneinheiten durch Governor/Scheduler	Slot-Offerte an Demuxer.	Slot-Offerte an Demuxer.	Direktes Starten des Executors.

Tabelle 3.2: Eigenschaften der drei Scheduling Strategien. Wichtige Verbesserungen sind die multithreaded Executors, das mehrfache Warten auf die Ausführung und die Art der Executor-Priorisierung.

Eine der wichtigsten Anforderungen an diese Art von Scheduling-System ist das asynchrone Zusammenspiel aller Teilnehmer, um Busy-Waiting zu verhindern. Dies wird in allen Implementationen mithilfe des Observer-Patterns erreicht. Allerdings unterscheiden sich die Instant- und Conservative-Implementierungen von der Queued-Variante. Dort werden die Slots nicht nur an die Stufen offeriert, sondern direkt vom `DeviceScheduler` belegt und die Verarbeitung im `StageExecutor` gestartet. Diese Variante bringt den grossen Vorteil, dass es nicht zu Fällen kommen kann, in welchen der `PriorityGovernor` die `PipelineStage` über freie Slots benachrichtigt und diese aber bereits belegt sind, wenn die Stufe sie anfordern möchte. Dieses Problem entsteht durch die Nebenläufigkeit aller Teilnehmer und ist deshalb schwierig zu kontrollieren. Dies wird versucht, indem immer nur der am längsten wartende Observer vom Governor benachrichtigt wird. Trotzdem kann es vorkommen, dass zwei unterschiedliche Stufen sich bei einem Governor mit freien Slots registrieren und der zweite bereits benachrichtigt wird, bevor der erste den Slot belegen konnte. Dieser Fall ist in Abbildung 3.17 vereinfacht dargestellt und wird durch die Einführung des `DeviceSchedulers` verhindert.

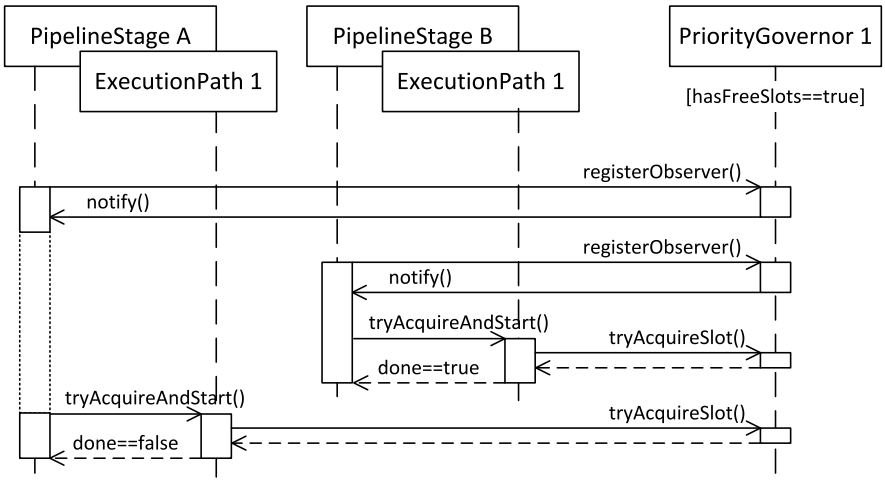


Abbildung 3.17: Durch Nebenläufigkeit kann es zu Problemen kommen, wenn die `PipelineStage` den Slot nicht sofort nach dem `Notify` des `PriorityGovernors` belegt. Es wird auf die Darstellung von diversen, für das Beispiel unwichtige, Methodenaufrufe verzichtet.

Ein Design-Problem, welches in der Instant-Implementierung existiert, ist die Anforderung des `ExecutionPaths` durch die `PipelineStage`. Wie bereits erwähnt (siehe 3.3.1), ist dieser Vorgang unnötig, verkompliziert den Ablauf und erhöht durch das Anfordern von zwei notwendigen Ressourcen die Gefahr von Deadlocks. Zudem wird ein spezielles System benötigt, damit die `PipelineStage` auf zwei unterschiedliche Benachrichtigungen reagieren kann.

Im Weiteren besitzt die Instant-Variante keine automatische Möglichkeit begrenzende Stufen bei Pipelines mit nur einem Rechengerät zu beschleunigen. Dadurch kann es bei bestimmten Pipelines dazu kommen, dass z. B. nur drei der vier verfügbaren CPU-Slots ausgenutzt werden. Dieses Verhalten kann in Abbildung 3.7 b) betrachtet werden und durch das in Abschnitt 3.2 vorgeschlagene Multithreading der Stufe verbessert werden.

Beide beschriebenen Schwächen werden durch die Conservative-Implementation behoben, welche als erste die multithreaded Stufen einführt. Solche Pipelinestufen haben zudem die folgenden Vorteile.

- Kein Auffüllen der Pipeline beim Starten des Systems nötig.
- Vollständige Auslastung der Ressourcen auch bei speziellen Pipelines, mit wenigen oder begrenzenden Stufen.

Durch die Implementierung der Policy im Datenpaket kann die Ausführung auf Datenlokalität, aber nicht auf Verarbeitung auf dem besten geeigneten Gerät optimiert werden. Deshalb fällt bei den Instant- und Conservative-Implementationen auf, dass Daten bei kleiner Systemlast bis zur letzten Stufe auf einem schlechteren Gerät verarbeitet werden, obwohl eine schnellere Ressource freie Slots zur Verfügung hätte. Um in solchen Fällen einen Wechsel der Daten auf das bessere Gerät zu erzwingen, wird eine Priorisierung der Geräte bzw. der Executors benötigt. Diese wird mit dem `DeviceScheduler` eingeführt.

Um die Optimierungsmöglichkeiten durch die Scheduler zu erweitern, muss dafür gesorgt werden, dass viele Jobs auf die wenigen Slots warten. Dadurch füllen sich die Warteschlangen und bieten dem Scheduler mehr Auswahl des am besten geeigneten Jobs. Der Demultiplexer ermöglicht dies, indem er nicht auf die Verarbeitung eines registrierten Jobs wartet, sondern sofort das nächste anstehende Paket beim Scheduler registriert.

3.4 Leistungs-Evaluation

In diesem Abschnitt wird die Performance des entwickelten Pipeline-Schedulers genauer untersucht. Dazu werden zwei unterschiedliche Arten von Simulationen durchgeführt und den jeweils oberen und unteren theoretischen Grenzwert der simulierten Pipeline ermittelt. Der primäre Messwert, welcher die Performance darstellt, ist die Dauer der kompletten Ausführung; also der Zeitraum zwischen dem Starten der Verarbeitung des ersten Pakets bis zum Beenden der Verarbeitung des letzten Pakets.

3.4.1 Grenzwerte

Da bereits das optimale Planen von anfangs bekannten Arbeitsschritten zur Klasse der NP-harten Probleme gehört (siehe Makespan Minimierung [10]), befindet sich auch das hier beschriebene Problem des Online-Schedulings in dieser Schwierigkeitsklasse. Dazu kommt die Heterogenität des Systems, welche dazu führt, dass die Ausführung derselben Aufgabe auf unterschiedlichen Rechengeräten zu anderen Ausführungszeiten führt.

Um die Performance des echten Systems einschätzen zu können, werden Vergleichswerte in Form von oberem und unterem Grenzwert verwendet. Wie erwähnt, existiert für die Berechnung einer minimalen Ausführungsdauer eines solchen Systems keine praktikable exakte Methode, weshalb drei unterschiedliche Heuristiken für die Schätzung einer unteren Grenze verwendet werden. Für die Evaluierung eines oberen Grenzwertes werden zwei unterschiedliche Varianten eingesetzt. Für den unteren Grenzwert α wird jeweils der tiefste Wert der drei Varianten gewählt, für den oberen Grenzwert β wird der grössere Wert der beiden Methoden verwendet. Alle Berechnungen ignorieren etwaige Verzögerungen durch Datentransfers

zwischen den Geräten. Die genauen Eigenschaften dieser Vergleichswerte und deren Berechnung finden sich im Paper [4] im Anhang E.

3.4.2 Simulationen

Es werden zwei Arten von Simulationen durchgeführt, um die Leistung des Systems zu ermitteln. Die erste wird mit statischen Ausführungswerten der Tasks aufgebaut. Dabei wird für jedes Rechengerät definiert, wie viele Millisekunden die Ausführung der einzelnen Stufen dauert. Zudem wird die Dauer der Kopieroperationen zwischen den Geräten definiert. In der Simulation mit dynamischen Tasks werden diese als echte Matrix-Multiplikationen implementiert, welche somit je nach Systemauslastung unterschiedlich viel Zeit für die Verarbeitung benötigen.

Durch die in Abschnitt 3.2.2 vorgestellten Eigenschaften ergeben sich vier Pipeline-Varianten, die alle mit statischen Tasks simuliert werden können. Tabelle 3.3 listet die Zeitaufwände der einzelnen Stufen auf. Als Beschleunigungsfaktor von der CPU zur GPU wurde 0.2 verwendet. Die Datentransfers zwischen den beiden Geräten benötigen jeweils einen Zehntel der GPU-Ausführungszeit. In einem realen System variieren die Ausführungszeiten, ausgelöst durch unterschiedlich grosse Datenmengen oder Systemschwankungen. Die Daten unterscheiden sich im Simulationssystem nicht, allerdings unterliegt dieses ebenfalls belastungsbedingten Schwankungen, weshalb für die Simulation solcher Effekte keine zusätzlichen Massnahmen unternommen werden.

[ms]	vollständig gleichförmig			unvollständig gleichförmig			vollständig ungleichförmig			vollständig ungleichförmig		
Grenzwerte	937	bis	7500	940	bis	6900	1355	bis	7500	1405	bis	7250
Stufe	CPU	GPU	Daten	CPU	GPU	Daten	CPU	GPU	Daten	CPU	GPU	Daten
A	50	10	1	50	10	1	20	10	1	20	10	1
B	10	2	0.2	10			30	15	0.1	30		
C	5	1	0.1	5	1	0.1	20	10	1	20	10	1
D	25	5	0.5	25	5	0.5	10	20	2	10	20	2
E	30	6	0.6	30	6	0.6	15	10	1		10	
F	15	3	0.3		3		5	5	0.5	5	5	0.5
G	5	1	0.1	5	1	0.1	20	10	1	20	10	1
H	10	2	0.2	10	2	0.2	15	20	2	15	20	2
	→ 0.2	→ 0.1		→ 0.2	→ 0.1							

Tabelle 3.3: Ausführungszeiten der statischen Tasks. In den gleichförmig beschleunigten Pipelines unterscheiden sich die Werte zwischen CPU und GPU jeweils um den Faktor 0.2 und zwischen GPU und Datentransfer um den Faktor 0.1.

Für die dynamischen Tasks existiert keine ungleichförmig beschleunigte Variante, da die GPU bei Matrix-Multiplikationen immer die bessere Performance als die CPU hat. Zur Berechnung der Grenzwerte werden die Ausführungszeiten der Task-Implementationen mehrmals gemessen und gemittelt. Die Daten sind in Tabelle 3.4 ersichtlich.

[ms]	vollständig gleichförmig			unvollständig gleichförmig		
Grenzwerte	249	bis	2004	250	bis	1803
Stufe	CPU	GPU	Daten	CPU	GPU	Daten
A	5.01	0.99	0.74	5.01	0.99	0.74
B	5.01	0.99	0.74	5.01		0.74
C	5.01	0.99	0.74	5.01	0.99	0.74
D	5.01	0.99	0.74	5.01	0.99	0.74
E	5.01	0.99	0.74	5.01	0.99	0.74
F	5.01	0.99	0.74		0.99	0.74
G	5.01	0.99	0.74	5.01	0.99	0.74
H	5.01	0.99	0.74	5.01	0.99	0.74

Tabelle 3.4: Ausführungszeiten der dynamischen Tasks. Diese werden vorgängig gemessen und nur für die Evaluierung der Grenzwerte verwendet, wobei dazu nur die Zeiten der Tasks und nicht des Datentransfers einbezogen werden.

3.4.3 Messungen

Die im vorherigen Abschnitt gezeigten Pipelines werden auf dem in Tabelle 3.5 beschriebenen PC simuliert, wobei nur drei der vier verfügbaren CPU-Kerne verwendet werden (siehe Abschnitt 3.4.4). Die Simulation verarbeitet jeweils 50 Pakete und wird fünfmal wiederholt. Alle gemessenen Werte werden danach gemittelt und es wird mittels Standard-Abweichung überprüft, ob der Simulationsdurchlauf keine zu grossen Schwankungen aufweist und wiederholt werden muss.

Betriebssystem	Windows 7 x64 SP1
Motherboard	ASUS Rampage II Extreme
CPU	Intel Core i7 950 3.06 GHz
GPU	ZOTAC GTX-460 AMP 1GB DDR5
Arbeitsspeicher	Mushkin Redline 3x2GB DDR3-1600
HDD	Samsung 2x SATA 500GB als RAID 0

Tabelle 3.5: Hardware des Testsystems. Die CPU besitzt vier identische Kerne und die GPU eine Rechen-Engine.

Gemessen werden die in Abschnitt 3.2 teilweise bereits erwähnten Werte, die im Folgenden mit abnehmender Relevanz aufgelistet werden.

- Gesamtausführungszeit der Pipeline R.
- Ausgabevarianz σ^2.
- Anzahl Datentransfers zwischen CPU und GPU.
- Initiale Verzögerung l_0.

Bezogen auf das Scheduling-Problem entspricht die Optimierungsvorgabe $\min(R)$, weshalb die Gesamtausführung der entscheidende Massstab bei der Bewertung ist. Sie symbolisiert den möglichen Durchsatz des Systems. Die Ausgabevarianz hingegen versucht abzuschätzen, wie gut sich das System für eine Live-Anwendung eignet, in welcher eine regelmässige Datenausgabe verlangt wird. Die Anzahl Datentransfers deuten auf die Effizienz der Scheduling-Strategie hin. Werden wenige Datentransfers für einen guten Wert von R benötigt, impliziert dies eine häufig optimale Wahl des Rechengeräts für die Verarbeitung der Daten. Die initiale Verzögerung ist grundsätzlich nicht relevant für den laufenden Betrieb,

zeigt allerdings die Fähigkeit des Systems, die Pipeline mit Arbeit zu füllen und sich auf früh gestartete Pakete zu konzentrieren. Diese Eigenschaft hängt mit der Ausgabevarianz zusammen.

$$\frac{R}{\alpha}$$

Formel 3.2: Berechnung der relativen Gesamtausführungszeit. R ist die absolute Ausführungszeit, α der untere Grenzwert. Kleine Werte sind besser.

Um die Gesamtausführungszeit in einen geeigneten Kontext zu setzten, wird diese relativ zu dem theoretischen unteren Grenzwert betrachtet. Dazu wird mittels der Formel 3.2 ein Faktor berechnet, welcher die Abweichung vom unteren Grenzwert beschreibt. Ein Wert von 1 entspricht dem unteren Grenzwert.

3.4.4 Resultate

Die in Abschnitt 3.3 entwickelten Scheduling Strategien werden mittels den beschriebenen Simulationen getestet. Das sofortige Scheduling wird als „Simple“ und dessen erweiterte Variante, welche gleich viele `ExecutorPaths` wie Recheneinheiten verwendet, wird als „Multi“ gekennzeichnet. Das zurückhaltende Scheduling wird als „Conservative“ bezeichnet. Die warteschlangebasierte Implementierung, welche die Priorisierung der Rechengeräte verwendet, wird „Prioritized“ und diejenige ohne Executor-Priorisierung „Unprioritized“ genannt.

Die Messungen verwenden nur drei der vier verfügbaren CPU-Kerne für die Task-Ausführungen, weil es sehr wichtig ist, dass das System sofort über abgeschlossene Verarbeitungen informiert wird. Diese Problematik wird im Abschnitt 4.1.2 und im Anhang C.2 erläutert. Durch die Nichtverwendung eines CPU-Kerns, wird dieser für die Betriebssystemverwaltung, andere Anwendungen und vor allem für die Verwaltung der Pipeline durch die Scheduler reserviert. Dadurch bleibt zwar ein gewisser Teil der Rechenressourcen meist unbenutzt, stellt aber sicher, dass die Verwaltung nie in Engpässe läuft und die essentiellen Benachrichtigungen, hauptsächlich von Seite der GPU, sofort empfangen und bearbeitet werden können. Die Messungen mit vier Kernen haben gezeigt, dass sich alle Systeme deutlich besser verhalten, wenn nur die erwähnten drei CPU-Kerne verwendet werden.

Aus den Messresultaten geht deutlich hervor, dass alle heterogenen Implementationen besser sind als homogene Pipelines, egal ob diese nur einen CPU-Kern, alle vier CPU-Kerne oder die GPU verwenden. Zudem zeigt sich, dass sich alle heterogenen Systeme sehr viel näher an der unteren als an der oberen Grenze befinden. Die relative Gesamtausführungszeit beträgt meist Werte um den Faktor 1.8 oder tiefer, wobei die obere Schranke im Mittel einem Faktor von 6.5 entsprechen würde.

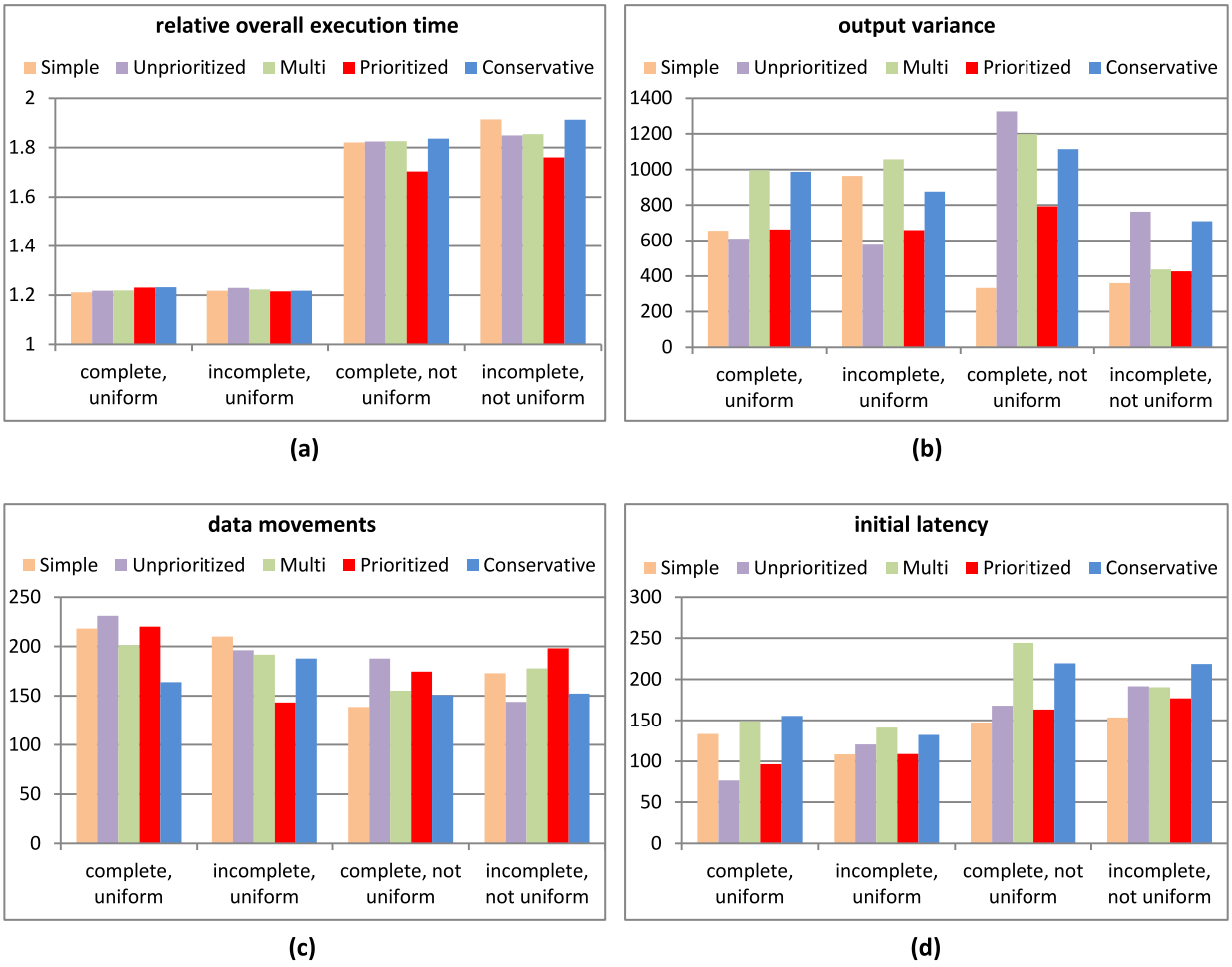


Abbildung 3.18: Messungen der künstlichen Pipeline mit statischen Tasks. Das System verwendet drei CPU-Kerne und eine GPU-Engine. Kleine Werte sind besser. (a) Gesamtausführungszeit relativ zum unteren Grenzwert. (b) Ein kleiner Wert bedeutet ein gleichmässiges Ausgabe-Intervall. (c) Anzahl Datentransfers zwischen CPU und GPU. (d) Die Zeit l_0, welche das erste Paket bis zur Ausgabe benötigt.

Abbildung 3.18 stellt die Messwerte der statischen Simulation grafisch dar. Es kann erkannt werden, dass die „Simple“-Implementation in einer vollständigen und gleichmässig beschleunigten Pipeline die kürzeste Gesamtausführungsdauer (Bild a) erzielt. Gesamthaft zeigt die „Prioritized“-Implementation die beste Performance. Vor allem in nicht gleichmässig beschleunigten Systemen dominiert sie das Bild. Dies ergibt sich aus der fehlenden Möglichkeit der sofortigen und zurückhaltenden Implementation, überhaupt zu wissen, welcher `ExecutionPath` die beste Performance liefert. Sie priorisieren grundsätzlich die GPU-Implementierungen.

Die Ausgabevarianz (Bild b) bietet kein so klares Bild. Im Mittel schneidet die „Simple“-Implementierung am besten ab, da sie pro Stufe nur jeweils ein Paket verarbeiten kann und deshalb automatisch länger in der Pipeline enthaltene Pakete verarbeitet. Dies, weil ein wartendes Paket in einer Stufe alle vorherigen Stufen blockiert. Die Resultate der „Prioritized“-Version sind am regelmässigsten verteilt, was auf eine Unabhängigkeit des Systems gegenüber der Pipelineart hindeutet.

Durch Anzahl Datentransfers (Bild c) und die initiale Verzögerung (Bild d) können keine klaren Aussagen zum besten System gemacht werden. Allerdings bestätigen sich die schlechten Werte der Implementierungen „Multi“ und „Conservative“ in diesen beiden Grafiken.

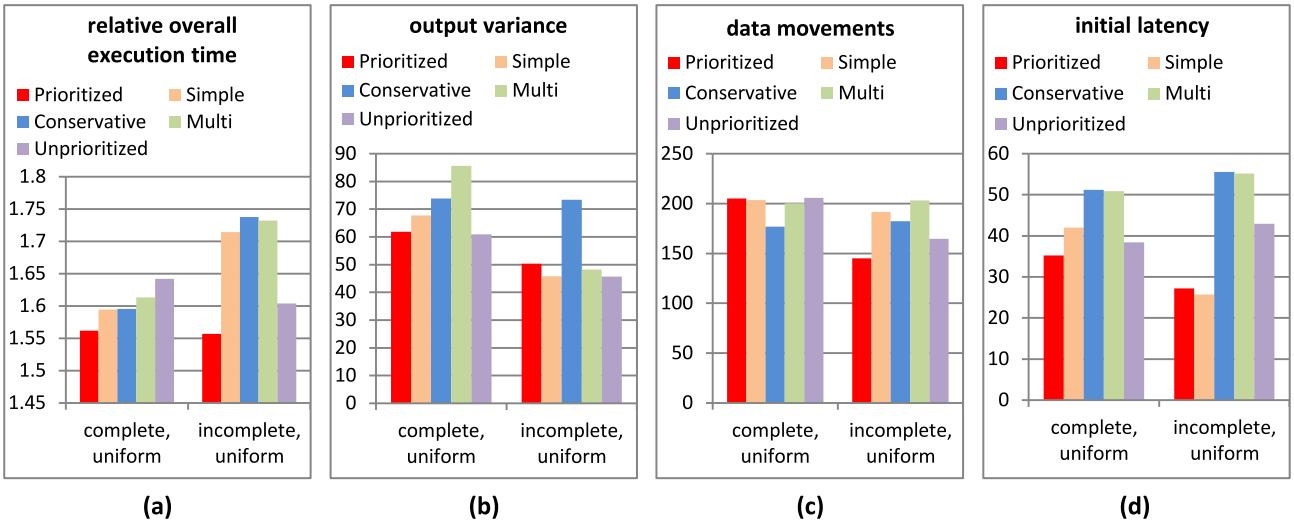


Abbildung 3.19: Messungen einer realistischen Pipeline mit Matrix-Multiplikationen. Da die GPU-Tasks immer schneller sind als die CPU-Tasks, existiert keine ungleichmässig beschleunigte Pipeline. Das System verwendet drei CPU-Kerne und eine GPU-Engine. Kleine Werte sind besser. (a) Gesamtausführungszeit relativ zum unteren Grenzwert. (b) Ein kleiner Wert bedeutet ein gleichmässiges Ausgabe-Intervall. (c) Anzahl Datentransfers zwischen CPU und GPU. (d) Die Zeit l_0, welche das erste Paket bis zur Ausgabe benötigt.

Die Messdaten der Pipeline mit realen Matrix-Multiplikationen sind in Abbildung 3.19 grafisch dargestellt. In dieser Konstellation schneidet in den meisten Fällen die „Prioritized“-Variante am besten ab. Zudem erreicht die „Simple“-Implementation teilweise ähnlich gute Werte, noch besser ist aber die „Unprioritized“-Version. Diese verhält sich in der statischen Simulation deutlich schlechter als in der Simulation eines realitätsnahen Systems. Da nur zwei unterschiedliche Pipelinearten simuliert werden können, erscheinen die Werte im Allgemeinen stabiler.

4 Heterogenes Pipeline-Framework

Neben der Scheduling-Strategie wird parallel dazu ein Pipeline-Framework entwickelt. Dieses bietet alle benötigten Elemente einer typischen Pipeline an. Dieses Kapitel beschreibt die Verwendung und Entwicklung des universell einsetzbaren Frameworks für heterogene Pipeline Architekturen. Das Framework verwendet die warteschlangebasierte Scheduling-Strategie aus dem vorherigen Kapitel und stellt alle benötigten Grundbausteine für eine nutzerspezifische Pipeline zur Verfügung.

4.1 Verwendung des Frameworks

In diesem Abschnitt wird die Verwendung aller Elemente (siehe Tabelle D.1) des entwickelten Frameworks erläutert. Beim Einsatz des Frameworks werden Smartpointer [11], die Move-Semantik [12], Funktions-Objekte, Template-Klassen und diverse andere C++11 Paradigmen eingesetzt. Für den Zugriff auf andere Rechenressourcen können z. B. die in [2] vorgestellten HSAs verwendet werden. In dieser Arbeit wird dafür C++ AMP [13] verwendet. Der komplette Code des Anwendungsbeispiels ist in Anhang D.2 ersichtlich.

4.1.1 Pipeline-Design

Als erstes muss definiert werden, wie die Pipeline aufgebaut ist und welche Daten verarbeitet werden sollen. Dabei kann entweder ein einziger Datentyp als Nachricht durch die Pipeline verarbeitet werden oder jede Stufe hat ihren eigenen Ausgangsdatentyp, welcher dem Eingangsdatentyp der nächsten Stufe entspricht. In einer solchen Variante werden maximal $p + 1$ Datentypen bei p Stufen benötigt. Abbildung 4.1 zeigt ein Beispiel mit drei Pipelinestufen. Neben den Datentypen müssen für jedes Rechengerät die maximal p Task-Implementationen definiert werden. Diese nehmen jeweils eines oder mehrere Datenpakete (siehe Abschnitt 4.1.3) vom Eingangstyp entgegen und liefern die verarbeiteten Daten in Form des Ausgangsdatentyps zurück.

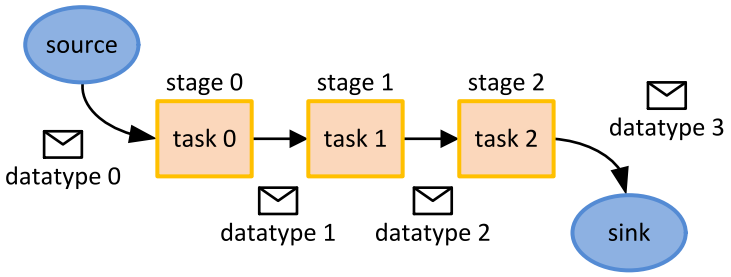


Abbildung 4.1: Beispielpipeline mit drei Stufen. Bei der Verwendung von unterschiedlichen Datentypen werden in diesem Beispiel vier Stück davon benötigt.

Listing 4.1 zeigt eine Beispiel-Implementation eines Datenpakets. Dieser Typ muss von der Klasse `BasePackage` abgeleitet werden, damit er die benötigte Funktionalität für die Verwendung mit dem Pipeline-Framework erhält. Das Framework übernimmt die Verwaltung der Pakete und leitet diese an die Task-Implementationen weiter. Da es sich dabei um Kopien handelt, soll ein Datenpaket keine echten Daten, sondern nur Zeiger auf die Daten beinhalten. Dies verhindert, dass ungewollt tiefe Kopien der echten Daten angelegt werden. Im Beispiel wird die Smartpointer-Klasse `shared_ptr` verwendet. Diese implementiert ein Reference-Counting und kümmert sich so um das Abräumen von nicht mehr verwendeten Objekten auf dem Heap.

```
class MyPackage : public BasePackage {
public:
	shared_ptr<accelerator_view> cpuView;
	shared_ptr<accelerator_view> gpuView;
	shared_ptr<vector<float>> cpuData;
	shared_ptr<array<float, 1>> gpuData;
	shared_ptr<vector<float>> cpuResult;

	void prepareData(bool forGpu) {
		if (!cpuData && !gpuData) throw exception("no data available");
		if (!cpuView || !gpuView) throw exception("not all accelerator_views set");

		if (forGpu) {
			if (!gpuData) {
				gpuData = make_shared<array<float, 1>>(dataSize);
				array<float, 1> stagingArray(gpuData->extent, *cpuView, *gpuView);
				copy_async(cpuData->begin(), cpuData->end(), stagingArray).wait();
				copy_async(stagingArray, *gpuData).wait();
			}
		}
		else {
			if (!cpuData) {
				cpuData = make_shared<vector<float>>(dataSize);
				array<float, 1> stagingArray(gpuData->extent, *cpuView, *gpuView);
				copy_async(*gpuData, stagingArray).wait();
				copy_async(stagingArray, cpuData->begin()).wait();
			}
		}
	}
};
```

Listing 4.1: Beispiel eines Datenpakets für die Verwendung mit dem Pipeline-Framework. Alle Daten liegen auf dem Heap und werden im Datenpaket durch Zeiger angesprochen. Die `prepareData`-Methode macht Daten für das gewünschte Rechengerät verfügbar.

Da jede Task-Implementierung ein Paket erhalten kann, welches die Daten noch nicht auf dem gewünschten Rechengerät verfügbar hat, müssen diese häufig für das gewünschte Gerät vorbereitet werden. Dabei stellt sich die Frage, welche Rechenressourcen für eine Kopieroperation zwischen zwei Geräten notwendig sind. Weil die Datenvorbereitung allerdings erst ausgeführt werden kann, nachdem der Demultiplexer entschieden hat, welches Gerät für die Verarbeitung verwendet werden soll, ist zum Zeitpunkt der Datenvorbereitung bereits ein Slot des ausführenden Gerätes reserviert worden. Grundsätzlich müsste jeweils ein Slot von beiden Geräten für die Kopieroperation angefordert werden, damit die effektive Ressourcenverwendung korrekt im System der `DeviceScheduler` abgebildet werden würde. Da I/O-Operationen jedoch typischerweise über Techniken wie Direct Memory Access (DMA) und separate Daten-Engines auf der GPU ausgeführt werden, ist die Belastung der Rechengeräte vernachlässigbar. Es ist aber wichtig, dass alle notwendigen Operationen asynchron ausgeführt werden, um ein Busy-Waiting des CPU-Threads zu verhindern, während die eigentliche Arbeit vom DMA-Kontroller oder der GPU erledigt wird.

Die Datenvorbereitung wird in Listing 4.1 als `prepareData`-Methode implementiert. Diese kann vor jeder Task-Bearbeitung aufgerufen werden und stellt sicher, dass die Daten auf dem verwendeten Rechengerät verfügbar sind. Im Beispiel wird mittels des expliziten Umwandeln des `shared_ptr` zu Boolean überprüft, ob die gewünschten Daten verfügbar sind. Falls nicht, wird der entsprechende Container erstellt und gefüllt. Das Kopieren wird durch die asynchrone Kopierfunktion `copy_async` gestartet und mittels `wait` auf ihre Fertigstellung gewartet. Um die Operation zu beschleunigen, wird ein Staging-Array verwendet, welches einen Speicherbereich im Hauptspeicher darstellt, der von der GPU (nicht aber der CPU) direkt angesprochen werden kann.

4.1.2 Datenpakete füllen

Die Quelle am Anfang einer Pipeline ist zuständig für die Erzeugung der Datenpakete. Im Pipeline-Framework erledigt dies die Klasse `Source`. Allerdings erzeugt sie leere Pakete, welche nur die Framework-Grundfunktionalität enthalten. Das Erzeugen der anwendungsspezifischen Daten wird durch die erste Stufe in der Pipeline erledigt.

```
void createDataForCpu(MyPackage& in) {
	in.cpuView = make_shared<accelerator_view>(
		accelerator(accelerator::cpu_accelerator).default_view);
	in.gpuView = make_shared<accelerator_view>(
		accelerator(accelerator::default_accelerator).default_view);

	in.cpuData = make_shared<vector<float>>();
	for (int i = 0; i < dataSize; ++i) {
		in.cpuData->push_back(static_cast<float>(i) * 3.14f);
	}
}
```

Listing 4.2: Das Erzeugen von anwendungsspezifischen Daten und Abfüllen in die Datenpakete wird durch die erste Stufe der Pipeline erledigt. Das Beispiel initialisiert die `accelerator_views` der CPU und GPU und erzeugt Daten für die CPU.

Listing 4.2 zeigt die Erzeugung von Daten im CPU-Speicher. Typischerweise werden Daten nur durch ein einziges Rechengerät erzeugt oder geladen. Das Kopieren der Daten auf die Speicher anderer Rechengeräte erfolgt bei Bedarf in der nächsten Stufe durch die `prepareData`-Methode.

4.1.3 Verarbeiten der Daten

Bei der Implementierung des Tasks muss ebenfalls darauf geachtet werden, dass die Ausführung auf nicht-CPU-Geräten asynchron initiiert und asynchron auf dessen Beendigung gewartet wird. Bei der Verwendung von C++ AMP kann der Anwender trotzdem ungewollt ein Busy-Waiting erzeugen, indem er die `wait`-Methode der `accelerator_view` direkt aufruft. Diese wartet nicht asynchron auf die Beendigung des Tasks. Wie in Anhang C.3 ausgeführt wird, muss über Umwege ein `task`-Objekt erstellt werden, welches die `wait`-methode als asynchrones Warten implementiert.

Listing 4.3 zeigt die Implementierung eines GPU-Tasks, welcher alle Elemente des Daten-Containers mit sieben multipliziert. Der Aufruf der `parallel_for_each`-Funktion startet die Ausführung auf der GPU asynchron und weckt den erwähnten `wait`-Aufruf nach der Fertigstellung. Um sicher zu stellen, dass nach diesem Arbeitsschritt keine veralteten Daten im CPU-Speicher verfügbar sind, wird der entsprechende `shared_ptr` geleert.

```
void doTaskOnGpu(MyPackage& in) {
	if (!in.gpuData) throw exception("No GPU data");

	array<float, 1>& data = *in.gpuData;
	parallel_for_each(data.extent, [&](index<1> idx) restrict(amp) {
		data[idx] *= 7;
	});
	in.gpuView->create_marker().to_task().wait();

	in.cpuData.reset();
}
```

Listing 4.3: GPU-Task-Implementation im Detail. Alle Aufrufe der AMP-API sind asynchron. Durch leeren des Smartpointers wird sichergestellt, dass ein eventuell nachfolgender CPU-Task nicht mit alten Daten weiterarbeitet.

Das Pipeline-Framework fordert für jede Task-Ausführung eine einzige Recheneinheit eines Rechengerätes an. Dies wird durch einen Slot des `DeviceSchedulers` abstrahiert. Damit diese Abstraktion richtig funktioniert, muss darauf geachtet werden, dass jede Task-Implementation nur eine Recheneinheit für ihre

Arbeit verwendet. Eine CPU-Implementation, die z. B. mittels Open Multi-Processing (OpenMP) parallelisiert wird, würde diese Voraussetzung verletzen und zu einem überlasteten System führen.

Die Task-Implementationen werden dem Framework als Instanz der Template-Klasse `function` übergeben. Eine einfache Art solche Objekte zu erzeugen, ist die Generierung eines Lambdas, wie in Listing 4.4 gezeigt wird. Das `function`-Objekt erhält vom Pipeline-Framework jeweils einen `vector` mit einem oder mehreren Datensätzen vom definierten Typ des Datenpakets. Als Rückgabewert muss das `function`-Objekt das bearbeitete Datenpacket zurückgeben.

```
auto gpuTask = [](vector<MyPackage> in) -> MyPackage {
      in[0].prepareData(true);
      doTaskOnGpu(in[0]);
      return in[0];
};
```

Listing 4.4: Beispiel einer Taskdefinition. Dieser Task nimmt Daten des Typs `MyPackage` und verarbeitet diese. Wichtig ist, dass die Daten in jeder Task-Implementation zuerst für das entsprechende Rechengerät vorbereitet werden.

Eine Pipeline Stufe kann mehrere Datenpakete erhalten, wenn der Task Daten von mehreren Paketen für die Verarbeitung benötigt. Eine solche Stufe wird als „Multi-Paket-Stufe“ bezeichnet. Ein Beispiel ist die Erstellung eines Differenzbildes zwischen zwei aufeinanderfolgenden Frames eines Videostreams. Wie viele Pakete eine Stufe erhalten soll, wird bei der in Listing 4.5 gezeigten Instanziierung durch den zweiten Parameter definiert. Dieser Wert muss mindestens 1 sein, wodurch sichergestellt wird, dass die Task-Implementation mindestens ein Datenpaket erhält.

```
auto multStage = Stage<MyPackage, MyPackage>::create(
      "Multiplication", 1, cpuScheduler, cpuTask);
multStage->addExecutor(gpuScheduler, gpuTask);
```

Listing 4.5: Erstellung einer Pipeline Stufe. Bei der Instanziierung werden eine erste Task-Implementation und dessen `DeviceScheduler` mitgegeben. Zusätzliche Implementationen werden mit `addExecutor` hinzugefügt. Die Priorisierung mehrerer Implementationen wird mit `setExecutorPriority` definiert.

Um solche Probleme mit Multi-Paket-Stufen (siehe Abschnitt 4.2.3) zu verhindern, muss die Task-Implementation eine klare Trennung der lesend bearbeitenden Daten und der Resultate haben. Listing 4.6 zeigt ein Beispiel eines Tasks, welcher die Summe der Daten von Paket a und Paket b in den Resultatspeicher von Paket a schreibt.

```
void doMultiPackTaskOnCpu(MyPackage& a, MyPackage& b) {
      if (!a.cpuData || !b.cpuData) throw exception("No CPU data");

      a.cpuResult = make_shared<vector<float>>(dataSize);

      for (int i = 0; i < a.cpuData->size(); ++i) {
            a.cpuResult->at(i) = a.cpuData->at(i) + b.cpuData->at(i);
      }
}
```

Listing 4.6: Task-Implementation und Erstellung einer Multi-Paket-Stufe. Paket-Daten dürfen entweder nur beschrieben oder nur gelesen werden.

Im Task-Lambda müssen die Daten aller verwendeten Pakete für das Rechengerät vorbereitet werden. Die Anzahl der Datenpakete wird bei der Instanziierung der Stufe definiert. Listing 4.7 zeigt das Erstellen des Stufen-Objekts für das Beispiel aus Listing 4.6.

```
auto multiPackTask = [](vector<MyPackage> in) -> MyPackage {
      in[0].prepareData(false);
      in[1].prepareData(false);
      doMultiPackTaskOnCpu(in[0], in[1]);
      return in[0];
};

auto multiPackStage = Stage<MyPackage, MyPackage>::create(
      "MultiPack", 2, cpuScheduler, multiPackTask);
```

Listing 4.7: Alle durch die Task-Implementation verwendeten Daten müssen vorbereitet werden. Bei der Instanziierung der Stufe wird die Anzahl der verwendeten Pakete angegeben.

4.1.4 Aufbau einer Pipeline

Nach der Implementation der Tasks und dem Design der Pipeline kann diese aufgebaut werden. Neben den Tasks werden aber noch die Daten-Quelle und -Senke benötigt, welche für die Verwaltung der Pakete innerhalb der Pipeline verantwortlich sind. Dabei wartet die Quelle solange mit der Erstellung neuer Pakete, bis die Senke ein fertiggestelltes Datenpaket aus der Pipeline entfernt.

Die Anzahl solcher „Pipelineslots“ wird beim Erzeugen der Quelle definiert. Die Überwachung der Anzahl Pakete in der Pipeline ist aus Performance-Gründen wichtig. Eine Pipeline mit kleiner Anzahl Slots ermöglicht eine schnelle Verarbeitung eines einzelnen Pakets und somit eine kleine Verzögerung. Eine grosse Anzahl Slots stellt die vollständige Auslastung aller Ressourcen sicher. Deshalb muss die Slotanzahl je nach Anwendung und Pipelinegrösse angepasst werden. Um die Auslastung der Ressourcen zu gewährleisten, sollten mindestens gleich viele Slots wie Recheneinheiten (z. B. 4 CPU-Cores + 1 GPU-Engine = 5) zur Verfügung gestellt werden.

Listing 4.9 zeigt die Erstellung einer Quelle mit acht Pipelineslots. Zudem wird eine Senke erstellt. Von der Senke wird ein `shared_ptr` auf den finalen Puffer abgeholt. Darin werden die fertiggestellten Datenpakete durch die Senke abgelegt und so dem Nutzer zum Abholen angeboten. Beim Verschieben in den `FinalBuffer` gibt die Senke jeweils die Pipelineslots frei.

Für die Verwaltung der Rechengeräte wird je ein `DeviceScheduler` für die CPU und die GPU instanziiert. Da neben dem Betriebssystem auch das Framework selbst einen gewissen Management-Aufwand erledigen muss, macht es Sinn, einen CPU-Kern dafür zu reservieren und deshalb dem CPU-Scheduler z. B. nur drei von vier verfügbaren Kernen zuzuweisen.

```
// create pipeline input and output
auto source = Source<MyPackage>::create(8);
auto sink = Sink<MyPackage>::create(source);
auto buffer = sink->getFinalBuffer();

// create pipeline management
auto cpuScheduler = DeviceScheduler::create(3, "CPU");
auto gpuScheduler = DeviceScheduler::create(1, "GPU");
```

Listing 4.8: Erstellung der Management-Objekte einer Pipeline. Es wird mindestens ein `DeviceScheduler`, eine `Source`, mindestens eine `Stage` und eine `Sink` benötigt.

Die einzelnen Pipelineelemente werden als `unique_ptr` verwaltet und mittels der in Listing 4.9 ersichtlichen `link_target`-Methode zusammengesetzt. Der `unique_ptr` implementiert das Besitz-Paradigma, welches garantiert, dass ein Objekt nur von einem einzigen Zeiger referenziert wird. So erzwingt das Framework, dass nur das vorherige Pipelineelement auf das nächste Zugriff hat und somit das einzige Objekt ist, welches Nachrichten an die nachfolgende Stufe senden kann. Die Verschiebung des Besitzes vom ursprünglichen `unique_ptr`-Objekt zu demjenigen der verlinkten Stufe, muss explizit mittels der `move`-Funktion erlaubt werden. Diese Funktion ist Teil der Move-Semantik in C++11 und erzwingt die Verschiebung durch Übergabe des Objektes als r-Referenz.

```
// link pipeline stages together
multiPackStage->link_target(move(sink));
multStage->link_target(move(multiPackStage));
createStage->link_target(move(multStage));
source->link_target(move(createStage));
```

Listing 4.9: Zusammensetzen aller Pipeline-Elemente. Da die `unique_ptr` bei der expliziten Verschiebung ungültig werden, muss die Pipeline vom Ende zum Anfang hin zusammengebaut werden.

Da vom Hauptprogramm nur die Quelle referenziert wird, muss verhindert werden, dass diese zerstört wird, während in der restlichen Pipeline noch Verarbeitungsprozesse stattfinden. Die Zerstörung der Quelle und dessen `unique_ptr` der ersten Stufe würden die Zerstörung der kompletten Pipeline auslösen und somit zu fehlerhaftem Verhalten während der Bearbeitung führen. Um dies zu verhindern, wartet der Destruktor der Quelle, bis die Pipeline finalisiert ist. Dies ist der Fall, wenn alle Pipelineslots von der Senke freigegeben wurden und das letzte, beim Start der Pipeline definierte, Paket im finalen Puffer abgelegt wurde. Bei der Zerstörung der Pipeline werden alle Stufen, die Quelle und die Senke abgeräumt. Ebenfalls werden die `DeviceScheduler` zerstört, sollten diese nicht anderweitig referenziert werden.

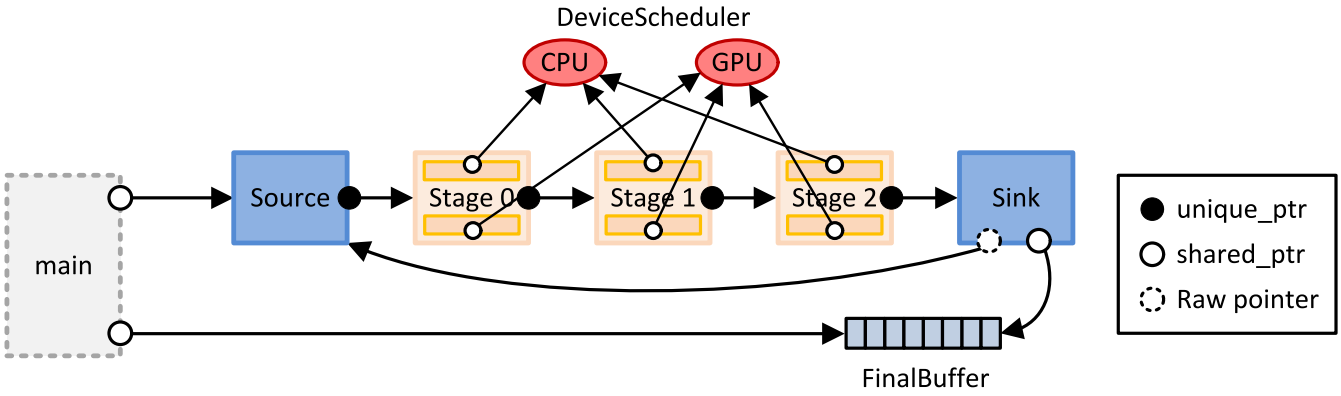


Abbildung 4.2: Objektabhängigkeiten einer Pipeline. Die Quelle ist die Wurzel der gesamten Pipeline und neben dem `FinalBuffer` das einzige Objekt, welches aus dem Hauptprogramm referenziert wird. Nachdem das letzte Paket durch die `Sink` abgeholt wurde, ist die Pipeline finalisiert und kann zerstört werden.

Abbildung 4.2 zeigt den Aufbau eines Pipelinesystems und die verwendeten Smartpointer. Um zyklische Abhängigkeiten zu verhindern, verwendet die Senke einen Roh-Pointer für die Handhabung der Quelle.

4.1.5 Ausführung der Pipeline

Nachdem das komplette Konstrukt aufgebaut wurde, kann die Pipeline gestartet werden. Dies geschieht mittels der `start`-Methode auf zwei Arten. Wenn zum Startzeitpunkt die Anzahl Datenpakete im Stream bekannt sind, wird die begrenzte Ausführung gestartet, indem die Anzahl Pakete angeben werden. Dadurch wird die Quelle genau diese Anzahl Pakete erzeugen und danach auf die Finalisierung warten. Für den Fall, dass anfangs keine Information über die Anzahl Datenpakete zur Verfügung steht, wird die Ausführung unbegrenzt gestartet. Damit die Pipeline trotzdem irgendwann stoppt, muss eine Stufe der Pipeline ein `STOP`-Paket generieren. Dieses stoppt die Generierung neuer Datenpakete in der Quelle. Damit die Quelle entscheiden kann, welches das letzte verarbeitet Datenpaket ist, muss das `STOP`-Paket den Index nach dem letzten Datenpaket haben. Es ist möglich, dass nach dem ersten `STOP`-Paket noch weitere Datenpakete in die Pipeline eintreten. Diese werden ebenfalls zu `STOP`-Paketen umgewandelt und unverarbeitet durch die Pipeline geleitet, da nur Datenpakete durch die Task-Implementationen verarbeitet werden. Listing 4.10 zeigt wie `STOP`-Pakete innerhalb der ersten Pipeline-Stufe generiert werden.

```
auto createTask = [](vector<MyPackage> in) -> MyPackage {
	if (in[0] >= packNum) {
		in[0].type = BasePackage::Type::STOP;
	}
	else {
		createDataForCpu(in[0]);
	}
	return in[0];
};
```

Listing 4.10: Beispiel der ersten Stufe in einer Pipeline. Diese füllt die leeren Datenpakete mit Nutzerdaten. Wenn keine weiteren Daten zur Verfügung stehen, werden alle weiteren eingehenden Datenpakete zu `STOP`-Paketen umgewandelt und weitergeleitet. So wird die Pipeline-Ausführung beendet.

In Listing 4.11 wird die Ausführung der Pipeline gestartet. Dabei wird dem Nutzer zusätzlich mitgeteilt, wie viele Datenpakete die Pipeline nicht mehr verlassen werden, da sie in den Puffern von Multi-Paket-Stufen hängen bleiben. Der Nutzer kann nach dem Start Pakete aus dem finalen Puffer blockierend abholen, bis er das `END`-Paket empfängt. Sollte die Pipeline nach der Ausführung erneut gestartet werden, aber noch nicht alle Daten aus dem Puffer abgeholt worden sein, wird der Puffer nach dem ersten `END`-Paket erneut Daten enthalten.

```
// start unbounded pipeline execution
int offset = source->start(0);

// receive processed data until pipeline execution has finished
MyPackage out;
do {
	out = Concurrency::receive(*buffer);
	if (out.type == BasePackage::Type::DATA) {
		displayData(out);
	}
} while (out.type != BasePackage::Type::END);
```

Listing 4.11: Unbegrenztes Starten der Pipeline und Abholen der verarbeiteten Daten im FinalBuffer. Für jeden Aufruf der `start`-Methode wird genau ein `END`-Paket im Puffer landen, welches das Ende der Verarbeitung des dazugehörigen Ausführungsstarts signalisiert.

Gestartet kann die Pipeline allerdings nur, wenn die vorherige Ausführung komplett beendet und die Pipeline somit finalisiert wurde. Dies kann durch die beiden Methoden `isFinalized` oder `waitUntilFinalized` der Quelle überprüft werden.

4.1.6 Abräumen der Pipeline

Wie bereits erwähnt, wird eine finalisierte Pipeline durch die Zerstörung des `shared_ptr` der Quelle abgeräumt. Um sicherzustellen, dass die Ausführung endet, muss der Start entweder begrenzt initiiert, manuell die `stop`-Methode der Quelle aufgerufen oder `STOP`-Pakete durch die Pipeline erzeugt werden. Sind zum Zeitpunkt der Zerstörung der Pipeline noch nicht alle Daten aus dem finalen Puffer abgeholt worden, muss nur sichergestellt werden, dass dessen `shared_ptr`, und somit der Puffer selbst, nicht zerstört wird.

4.2 Framework-Verhalten im Detail

Dieser Abschnitt geht vertieft in diverse kritische Abläufe innerhalb des Frameworks ein. Dabei handelt es sich um die Verwaltung der gesamten Pipeline oder der einzelnen Task-Implementationen.

4.2.1 Nachrichten innerhalb der Pipeline

Die Nachrichten, welche durch die Pipeline wandern, erben von der Klasse `BasePackage`. Diese Klasse liefert grundlegende Funktionalität für das Framework. Dabei kann ein Paket der Pipeline unterschiedliche Formen annehmen. Diese werden in Tabelle 4.1 beschrieben.

Paket-Typ	`BasePackage::Type`	Beschreibung
Datenpaket	`DATA`	Verwaltet die Daten, welche durch die Tasks der Pipeline verarbeitet werden. Leere Datenpakete werden von der Quelle erzeugt und typischerweise durch die erste Stufe einer Pipeline gefüllt.
Konfigurations-Paket	`CONF`	Wird für die Initialisierung beim Start der Pipeline verwendet. Ermittelt den Offset der Pipeline und setzt den Sortierungsstart in allen Stufen. Soll nicht vom Nutzer erzeugt werden.
END-Paket	`END`	Wird nur durch die Senke erzeugt und in den finalen Puffer geleitet, wenn die Pipeline finalisiert ist. Signalisiert dem Nutzer, dass keine weiteren Daten im finalen Puffer eintreffen werden, solange die Pipeline nicht erneut gestartet wird / wurde.
STOP-Paket	`STOP`	Löst das Beenden der Pipeline-Ausführung aus. Bietet dieselbe Funktion wie die Methode `stop` der Quelle. Wird vom Nutzer innerhalb von Task-Implementationen erzeugt.

Tabelle 4.1: Übersicht über alle verwendeten Arten von Paketen. Vom Nutzer werden nur STOP-Pakete erstellt und es verlassen nur Datenpakete oder END-Pakete die Pipeline.

Die vom Nutzer hauptsächlich verwendete Form ist das Datenpaket. Diese werden von der Quelle erzeugt und müssen vom Nutzer gefüllt und durch die Task-Implementationen verarbeitet werden. Die anderen Formen werden für die Verwaltung der Pipeline verwendet.

4.2.2 Starten und Stoppen einer Pipeline

Eine Pipeline kann sich in mehreren Zuständen befinden. Diese werden in der Quelle implementiert und sind in Abbildung 4.3 dargestellt. Nachdem die Quelle und der Rest der Pipeline erstellt wurden, befindet sich das Konstrukt im Zustand „stopped“ oder „finalized“. In diesem Zustand werden keinerlei Pakete generiert oder in der Pipeline verarbeitet. Dies ist der Zustand, aus welchem eine Verarbeitung initiiert und somit die Pipeline gestartet werden kann. Ebenfalls ist es der einzige Zustand, in welchem die Pipeline zerstört werden kann. Dies, wie in Abschnitt 4.1.4 erwähnt, damit es zu keinen fehlerhaften Datenverarbeitungen in der Pipeline kommen kann.

Durch einen Aufruf der Methode `start` wechselt die Pipeline in den Zustand „running“. Dies startet die Generierung und Verarbeitung von Datenpaketen. Wird die Pipeline begrenzt gestartet, erreicht sie irgendwann die Anzahl zu generierende Pakete und sendet keine weiteren mehr in die Pipeline. Dieser Fall tritt ebenfalls bei einem Aufruf der Methode `stop` oder beim Generieren eines `STOP`-Pakets ein. Danach wartet die Quelle auf die Freigabe des Pipelineslots des zuletzt verarbeiteten Pakets und wechselt nach dieser Finalisierung der Pipeline in den Zustand „stopped“ zurück.

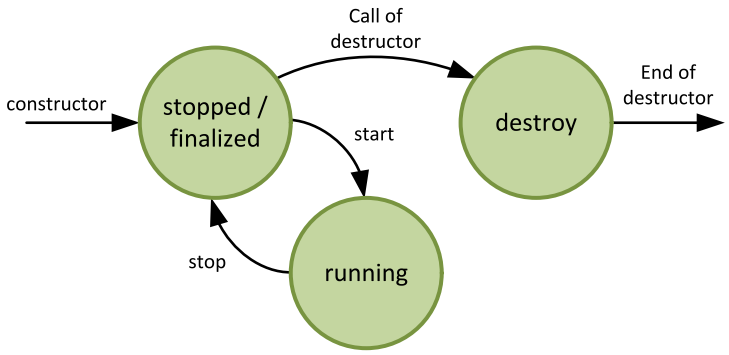


Abbildung 4.3: Die Quelle kennt drei Zustände. Der Zustand „running“ erstellt und versendet anfangs Datenpakete. Nach dem Stoppen der Pipeline oder Versenden des letzten Pakets wartet die Quelle auf die Finalisierung der Pipeline.

Vor jedem Start müssen alle Bestandteile der Pipeline initialisiert werden. Dazu wird von der Quelle ein Konfigurations-Paket erzeugt und durch die Pipeline gesendet. Die Quelle setzt dabei das `index`-Feld auf den Index des ersten zu generierenden Pakets. Dieser Index wird von allen Stufen und der Senke für die Sortierfunktionalität benötigt. Diese muss wissen, welches der niedrigste Index aller möglichen Pakete ist, um die Sortierung zu starten. Die Sortierfunktion wird nur von Multi-Paket-Stufen und von der Senke verwendet. Alle anderen Stufen verwenden diese aus Performance-Gründen nicht.

Ein weiteres Feld, welches vom Konfigurations-Paket verwendet wird, ist der `offset`. Mittels dieses Feldes wird die Anzahl der Pakete ermittelt, welche die Pipeline nach dem Stoppen nicht mehr verlassen können, da durch den Puffer in Multi-Paket-Stufen Pakete zurückgehalten werden. Es können $z - 1$ Pakete einer z-Paket-Stufe nicht verarbeitet werden. Der Offset der gesamten Pipeline berechnet sich aus der Summe aller verlorenen Pakete in Multi-Paket-Stufen. Beim Erhalten des Konfigurations-Pakets setzt die Senke den Offset-Wert bei der Quelle, damit diese beim Freigeben eines Pipelineslot durch die Senke entscheiden kann, ob die Pipeline alle möglichen Pakete verarbeitet hat.

Des Weiteren löst das Konfigurations-Paket die Löschung aller Stufenpuffer aus. Abbildung 4.4 zeigt das Zusammenwirken der Pipeline-Elemente und des Konfigurations-Pakets.

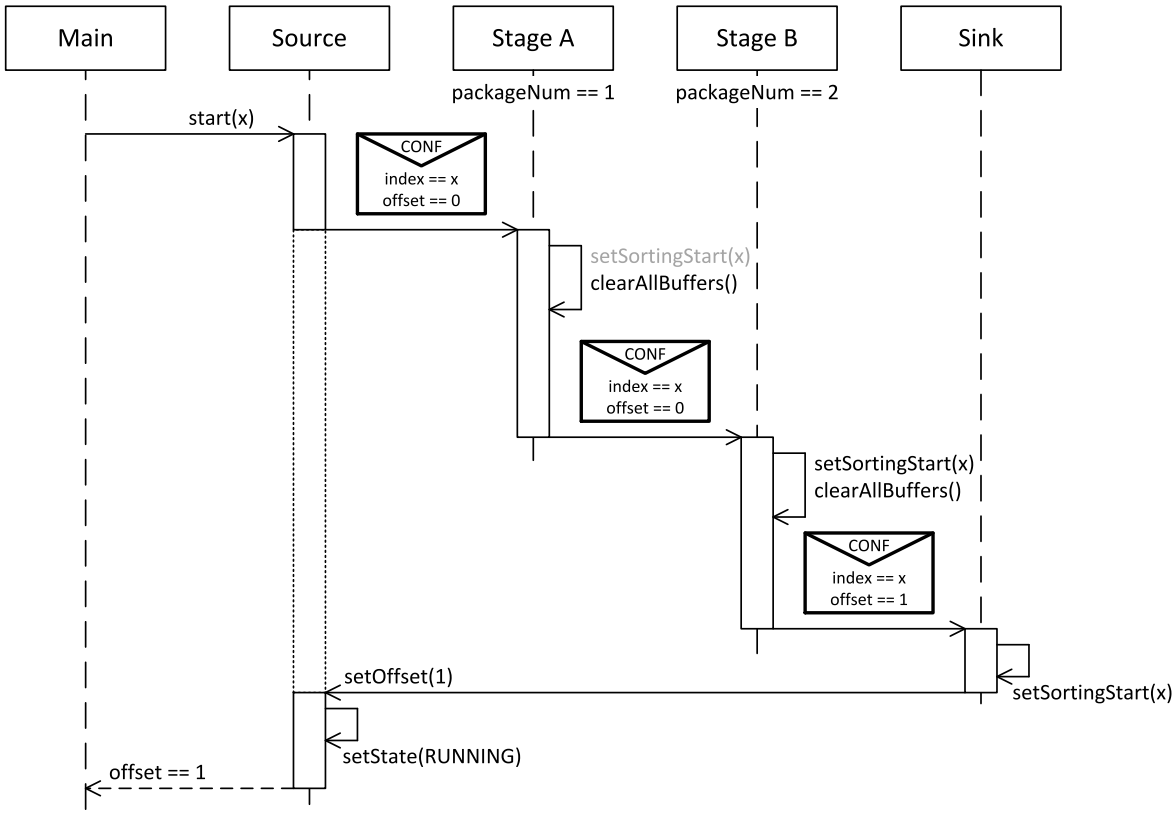


Abbildung 4.4: Starten der Pipeline und initialisieren aller Stufen durch ein Konfigurations-Paket. Dieses Beispiel entspricht der Pipeline aus Anhang D.2. Der ausgegraute Aufruf ist in der Stufe A unnötig, da diese nicht mehrere Pakete benötigt und somit nicht sortiert werden muss.

Das Stoppen der Pipeline wird in allen bereits erwähnten Fällen auf die gleiche Art gelöst. Die Quelle besitzt eine Member-Variable, welche den Index des ersten, nicht mehr verarbeiteten, Pakets enthält. Diese Variable wird bei einem begrenzten Start direkt beim Starten gesetzt. In den beiden anderen Fällen wird sie durch den Aufruf der `stop`-Methode gesetzt. Bei parameterlosem Aufruf, setzt die Methode den Wert auf das nächste Paket, welches gesendet werden müsste. Bei parametrisierten Aufruf wird der Wert des Parameters übernommen. Dieser Parameter wird durch das `STOP`-Paket an die Senke übermittelt, welche den Aufruf der Methode `stop` auslöst. Erreicht der Index des als nächstes zu sendende Pakets in der Quelle den gesetzten End-Index, bricht die Quelle ab und wartet auf die Finalisierung der Pipeline.

4.2.3 Multi-Paket-Stufe

In Multi-Paket-Stufen werden alle eingehenden Pakete nach aufsteigendem Index sortiert und solange gepuffert, bis die gewünschte Anzahl aufeinanderfolgende Pakete verfügbar ist. Alle diese Pakete werden als `vector` der Task-Implementation übergeben, aber nur das erste Paket wird aus dem Puffer entfernt. Die anderen Pakete werden für weitere Task-Ausführungen benötigt.

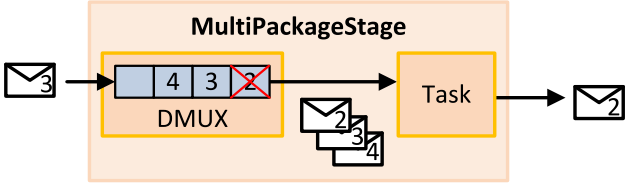


Abbildung 4.5: Ablauf in einer Multi-Paket-Stufe die drei Datenpakete benötigt. Der Demultiplexer leitet die benötigten Pakete zur Task-Implementation. Diese gibt nur das Paket mit dem niedrigsten Index aus.

Abbildung 4.5 zeigt ein Beispiel eines solchen Ablaufs, am Beispiel einer Stufe, welche drei Pakete benötigt. Anfangs enthält der Stufenpuffer die Pakete 2 und 4. Nach dem Eintreffen des Pakets mit dem Index 3, enthält der sortierte Stufenpuffer die benötigte Anzahl von Datenpaketen. Die Pakete mit den Indizes 2, 3 und 4 werden sortiert an die Task-Implementation weitergeleitet. Die Verarbeitung bezieht sich immer auf das Paket mit dem niedrigsten Index, weshalb in diesem Fall das Paket 2 als abgearbeitet betrachtet und aus dem Puffer gelöscht wird. Der Task schreibt seine Resultate in das Paket 2 und reicht es an die nächste Stufe weiter. Dabei muss zwingend das Paket 2 weitergereicht werden. Allerdings könnten die Resultate in einem anderen Paket gespeichert werden. Es muss jedoch sichergestellt werden, dass nur ein einzelnes Paket als Resultatspeicher verwendet wird.

Race-Conditions zwischen unterschiedlichen Stufen werden verhindert, da alle benötigten Pakete im Puffer der Stufe sein müssen, bevor sie zur Verarbeitung verwendet werden können. Allerdings können Probleme zwischen parallelablaufenden Task-Ausführungen innerhalb derselben Stufe auftreten. Diese entstehen, wenn z. B. bereits vier von drei benötigten Paketen bereit sind und somit nicht nur das Paket mit Index 2, sondern auch dasjenige mit Index 3 eine Task-Verarbeitung auslöst.

In Abbildung 4.6 liest die Task-Ausführung 2 die Daten der Pakete 2, 3 und 4. Die Resultate werden in Paket 2 gespeichert, was in diesem Beispiel unkritisch ist. Allerdings wird in der Ausführung 3 das Paket mit Index 3 als Resultatspeicher verwendet, weshalb nicht garantiert ist, dass die Ausführung 2 die korrekten Daten des Pakets 3 liest.

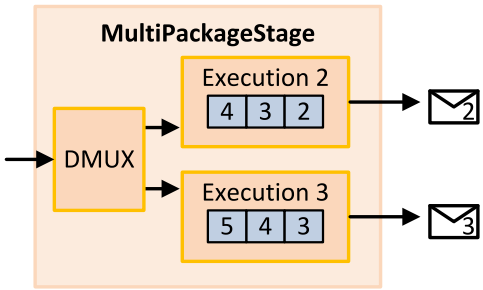


Abbildung 4.6: Nebenläufige Ausführung eines Tasks, welcher gleichzeitig auf dieselben Daten zugreift. Im Beispiel sind die Zugriffe auf Paket 3 lesend (Execution 2 & 3) und schreibend (Execution 3).

Um solche Probleme zu verhindern, muss die Task-Implementation von Multi-Paket-Stufen eine klare Trennung der lesend bearbeitenden Daten und der Resultate haben. Das heisst, dass Daten innerhalb des Pakets in einer Multi-Paket-Stufe entweder gelesen oder beschrieben werden. Zudem soll nur in eines der Datenpakete Resultate abgespeichert werden.

4.2.4 Abholen der verarbeiteten Daten

Die Senke bildet das Ende der Pipeline und zudem die Schnittstelle zum umgebenden Kontext. Diese Schnittstelle wird mittels des finalen Puffers, als Instanz eines `unbounded_buffer` aus der AAL, implementiert. Dadurch wird die Abholung der Daten durch einen Thread ausserhalb des Pipeline-Frameworks ermöglicht. Dies ist wichtig, wenn z. B. für die Darstellung von verarbeiteten Daten zwingend der GUI-Thread verwendet werden muss.

Da ein Datenstrom eine Form von sortierten Daten darstellt, muss die Reihenfolge eingehalten werden. Die Senke sortiert deshalb alle eingehenden Pakete und gibt sie sortiert an den finalen Puffer weiter. Danach gibt sie den Pipelineslot bei der Quelle frei. Weil diese Operation zudem überprüft, ob dieses Paket das letztmögliche Ausgabe-Paket der Pipeline ist, muss die Slot-Freigabe zwingend sortiert durchgeführt werden. Um zirkuläre Abhängigkeiten zu vermeiden, wird in der Senke ein Roh-Zeiger für die Quelle verwendet. Dies ermöglicht zudem den Zugriff auf die Quelle, wenn dessen `shared_ptr` bereits zerstört und die Zerstörung der Quelle initiiert, aber noch nicht durchgeführt wurde.

4.2.5 Priorisieren von Rechengeräten

Bei der Erstellung der Pipeline-Stufen kann nach dem Hinzufügen aller Task-Implementationen eine Priorisierung der Rechengeräte vorgenommen werden. Dies wird mithilfe der Methode `setExecutorPriority` erledigt, welcher ein `vector` von Paaren, bestehend aus dem `DeviceScheduler` und dessen Priorität, übergeben werden muss. Je höher der Wert, desto höher die Priorität. Soll ein Gerät nicht priorisiert werden, kann ein negativer Wert verwendet werden. Die Priorisierung wird im Beispiel in Anhang D.2 vorgeführt.

Die definierten Werte werden zu einer Rangfolge umgewandelt, wobei das beste Rechengerät den Index 1 für die erste Wahl der Stufe erhält. Dies ist nötig, damit die konkurrenzierenden Scheduler vergleichbare Rang-Indizes haben. Die Verwendung dieser Rangfolge kann in Abbildung 3.16 betrachtet werden.

Aufgrund der Tatsache, dass die Stufen einen neuen Job jeweils bei einem `DeviceScheduler` nach dem anderen registrieren, wird der Job in einem nicht belasteten System immer sofort vom ersten `DeviceScheduler` ausgeführt. Um sicherzustellen, dass der erste Scheduler in der Liste immer das am besten geeignete Rechengerät verwaltet, werden diese beim Setzten der Prioritäten sortiert.

Aktuell bietet das Framework keine Möglichkeit, das System selbständig so zu kalibrieren, dass automatisch derjenige `Executor` die beste Priorität erhält, welcher die schnellste Ausführungszeit hat. Dies muss anhand von Testreihen ermittelt werden und über die Methode `setExecutorPriority` jeder Stufe definiert werden. Dieser Automatismus könnte mittels einer Kalibrierungs-Klasse nachgeliefert werden, indem sie die jeweiligen Ausführungszeiten misst und die Priorisierung anhand dieser Werte selbstständig einstellt. Problematisch ist dabei, dass die Ausführungszeiten unterschiedlicher Task-Implementationen nicht bei jeder Ausführung im selben Verhältnis zueinander stehen. Zudem können die Ausführungszeiten von der Grösse oder Komplexität der Daten abhängig sein. Aus diesem Grund wäre z. B. eine Mittelung über alle Messungen und eine regelmässige Aktualisierung der Prioritäten sinnvoll.

4.2.6 Scheduling und Ausführung von Jobs

Wie bereits in Abschnitt 3.3.3 beschrieben, werden neue Jobs durch die Stufe bei allen verwendeten `DeviceSchedulern` registriert. Diese lösen bei genügend freien Ressourcen die Verarbeitung in ihrer zugeordneten Task-Implementation aus. Abbildung 4.7 zeigt den Ablauf vom Eintreffen des Datenpakets bis zum Weitersenden an die nächste Stufe.

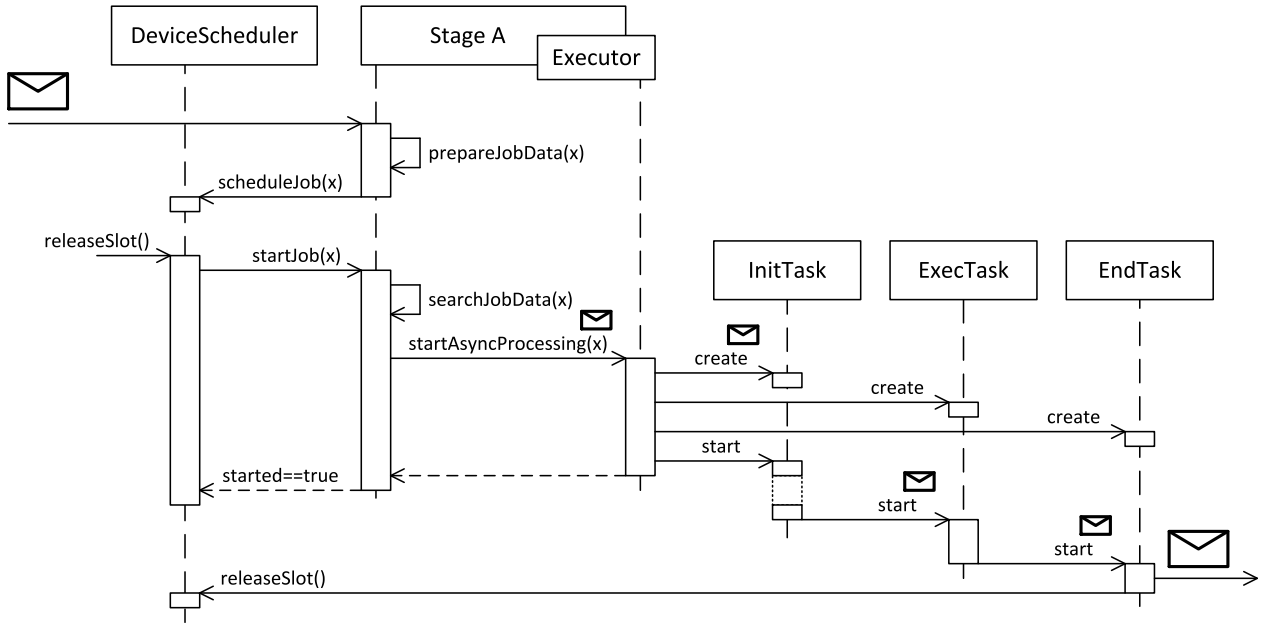


Abbildung 4.7: Ablauf der Initiierung einer Job-Ausführung im entsprechenden `Executor`. Dieses Beispiel vernachlässigt das Vorhandensein anderer `DeviceScheduler` oder Task-Implementationen. Die effektive Bearbeitung der Daten findet im `ExecTask` statt, wobei der `InitTask` für das Bereitstellen des Datenpakets und der `EndTask` für dessen Weitersendung zuständig sind.

Nachdem der `DeviceScheduler` die Ausführung initiiert und die Stufe die zu verarbeitenden Daten gefunden hat, werden diese an den entsprechenden `Executor` gesendet. Dieser Aufruf von `startAsyncProcessing` kreiert drei `task`-Objekte (siehe PPL [7]), die mittels der `then`-Methode zu „Continuations“ verkettet werden.

Der erste Teil dieser Kette ist für die Übergabe der Daten an den `ExecTask` verantwortlich. Dies ist nötig, weil der `ExecTask` mittels des `function`-Objekt der Task-Implementation erstellt wird. Das `function`-Objekt ist durch den Nutzer so definiert, dass es als Parameter das zu verarbeitende Datenpaket erhält. Somit müssen die Daten direkt in der Parameterliste übergeben werden. Diese Übergabe der Daten vom ersten zum zweiten Teil der Verarbeitungskette wird als „value-based continuation“ bezeichnet. Bei einer „value-based continuation“ führt die CRT zuerst den ersten `task` aus und startet den nachfolgenden nur, wenn keine Fehler im ersten `task` aufgetreten sind.

Die Verbindung des zweiten zum dritten Teil der Ausführungskette wird mittels einer „task-based continuation“ implementiert. Diese wird den nachfolgenden `task` immer ausführen und ermöglicht ihm so, auf auftretende Fehler zu reagieren. Dazu wird dem `EndTask` ein `task`-Objekt vom selben Typ wie der `ExecTask` übergeben. Durch dieses Objekt können mittels `get`-Methode die Rückgabedaten abgefragt werden. Bei diesem Aufruf werden gleichzeitig allfällige Exceptions geworfen, welche innerhalb der `ExecTask`-Asuführung generiert worden sind. Wenn keine Exceptions aufgetreten sind, werden die verarbeiteten Daten an die nächste Stufe weitergeleitet und der Slot des `DeviceSchedulers` freigegeben.

5 Personenerkennung mittels Pipeline-Framework

Das entwickelte Framework wird verwendet, um die Software aus dem Projekt 7 und 8 in eine Pipeline-Struktur umzubauen und alle Rechengeräte eines PCs durch den heterogenen Scheduler verwalten zu lassen. Dazu wird in diesem Kapitel beschrieben, welche Bestandteile neu erstellt, umgebaut oder entfernt werden mussten. Dabei wird auf diverse Besonderheiten bei der Task-Implementation in realen Systemen eingegangen, welche typischerweise auf bereits implementierte Lösungen zurückgreifen. Im Fall der bestehenden Software ist damit vor allem OpenCV und insbesondere dessen CUDA-Implementation gemeint.

5.1 Allgemeine System-Struktur

Neben den Änderungen am `MotionDetector` und `HeadDetector`, welche im nächsten Abschnitt beschrieben werden, muss das System vor allem beim Laden und Anzeigen des Video-Streams angepasst werden. Dabei fällt das komplette parallele Puffern von Video-Frames weg, da diese Eigenschaft der Parallelität neu durch das Pipeline-Framework übernommen wird. Der Aufbau der Pipeline ist in Abbildung 5.1 dargestellt.

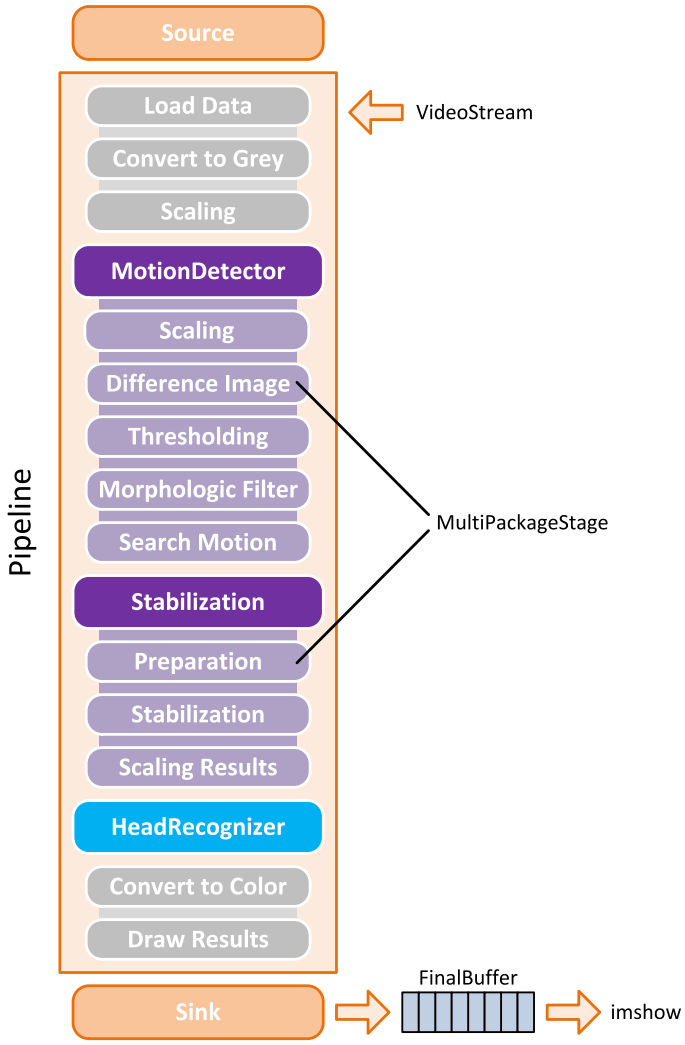


Abbildung 5.1: Finaler Aufbau der in der Personenerkennungs-Software verwendeten Pipeline. Das Laden der Videoframes wird durch die `VideoReader`-Klasse von OpenCV ermöglicht. Die Differenzbild-Stufe benötigt zwei, die Vorbereitungsstufe der Stabilisierung zehn Datenpakete.

Für die Verarbeitung der Daten werden nur noch die Klassen `MotionDetector`, `HeadDetector`, `StageFactory` und die Datenpaket-Klasse `VideoFrame` benötigt. Alle anderen Bestandteile werden durch das Pipeline-Framework zur Verfügung gestellt.

In Abbildung 5.1 ist erkennbar, dass zwei Multi-Paket-Stufen in der Pipeline enthalten sind. Bei deren Implementation wurde darauf geachtet, dass nur die Teil-Aufgabe, welche mehrere Datenpakete benötigt, in solch einer Stufe ausgeführt wird. Daher ist die Stufe „Preparation“ nur dafür zuständig, die benötigten Daten zu sammeln und für die nächste Stufe in einem einzigen Datenpaket vorzubereiten. Dadurch, dass die Stufe klein und einfach bleibt, wird die Gefahr vermindert, dass ungewollt Fehler beim Laden oder Speichern der Daten implementiert werden.

5.2 Umbau bestehender Implementationen

Die Mutation eines bestehenden Systems zu einer Pipeline kann in mehreren Schritten durchgeführt werden. Dabei wird zwischen Mikroblöcken (einzelne Stufen) und Makroblöcken (eine Sub-Pipeline) unterschieden. Folgende Schritte erleichtern den Umbau.

1. Identifizierung logischer und datenbezogener Abhängigkeiten und Definition der einzelnen Makroblöcke und deren Daten-Input und -Output.
2. Verschieben aller klassenlokalen oder globalen Variablen in die Datenpakete.
3. Stufen implementieren und zu gesamter Pipeline zusammenbauen.

Der erste Schritt ist der kritischste. Hier wird eine bestehende Software analysiert und unterteilt. Dabei stellen die Makroblöcke eine grobe Unterteilung des Gesamtproblems dar. Innerhalb eines Makroblocks wird typischerweise nur eine Art von Datenpaket verwendet, welche zwischen den Blöcken aber ändern kann. Im System aus Abbildung 5.1 könnten z. B. folgende Makroblöcke identifiziert werden: Laden der Daten, Bewegungsdetektion, Stabilisierung, Kopfdetektion und Ausgabe. Da es sich dabei nicht um allzu grosse Blöcke handelt und jeweils mit ähnlichen Daten gearbeitet wird, wird das gesamte System nur als ein Makroblock betrachtet. Ein System das mehrere Makroblöcke benötigt, ist deutlich grösser und verarbeitet unterschiedlichere Daten.

Im zweiten Schritt werden die Klassen für die Datenpaket-Typen erstellt und möglichst alle für die verarbeiteten Stream-Elemente spezifischen Instanzvariablen dorthin verschoben. Die Task-Implementationen können somit im optimalen Fall als statische Methoden oder Funktionen implementiert werden. Um die Ressourcen eines heterogenen Systems auszunutzen, müssen die vorhandenen Task-Implementationen in diesem Schritt auf die anderen Rechengeräte portiert werden.

Im letzten Schritt werden die Stufen erstellt und zur Pipeline zusammengefügt. Wenn die Stufen trotzdem noch auf Instanzvariablen zugreifen, macht es Sinn, die Stufen durch Methoden der jeweiligen Klasse generieren zu lassen.

Im Folgenden wird anhand der beiden verwendeten Klassen erklärt, auf was beim Umbau geachtet werden muss. Wie bereits erwähnt, verwenden alle Stufen ein `VideoFrame`-Paket.

5.2.1 MotionDetector

Die Klasse für die Detektion von Bewegungen in einem Video ist bereits gut für die Verwendung im Pipeline-Framework vorbereitet. Die Variablen für die Bilddaten und Detektionsbereiche können direkt in die `VideoFrame`-Klasse verschoben werden. Dadurch, dass Multi-Paket-Stufen über einen Puffer verfügen, kann die Problematik des Bereichsarchivs in der Stabilisierung durch den Puffer in der Stufe übernommen werden.

Die eigentliche Worker-Methode `findMotionRegions` wird so umgebaut, dass sie eine Sub-Pipeline generiert, welche alle bisherigen Verarbeitungsschritte enthält. Dabei muss die Parallelisierung durch OpenMP im „Search Motion“-Teil entfernt werden. Somit nutzen alle Stufen dieses Blocks nur je eine Recheneinheit eines Geräts. Für Stufen, welche Operationen von OpenCV verwenden, können jeweils Task-

Implementationen für CPU und GPU eingerichtet werden, da viele Operationen in OpenCV als CUDA-Varianten verfügbar sind. Dies wären z. B. `absdiff`, `threshold` oder morphologische Operationen. Die Funktion `resize` sollte nicht gleichzeitig durch die CPU und die GPU verwendet werden, da diese im aktuellen OpenCV-Build nicht dieselben Resultate liefern. Dies kann beim Erstellen des Differenzbildes und im restlichen Teil der Pipeline zu Fehlern führen.

5.2.2 HeadDetector

Die Kopfdetektion verwendet die OpenCV-Klasse `CascadeClassifier` und dessen CUDA-Pendent `CascadeClassifier_CUDA`. Diese können direkt als Task-Implementationen verwendet werden. Allerdings sind diese nicht thread-sicher implementiert und dürfen nicht verwendet werden, wenn das Rechengerät der jeweiligen Task-Implementation mehrere Recheneinheiten verwendet.

Eine simple Lösung für dieses Problem, ist das Sperren der Classifier-Methode `detectMultiScale` durch einen Mutex. Dadurch wird verhindert, dass die Stufe mehrfach ausgeführt wird, zerstört so aber die Task-Parallelität innerhalb der Stufe. Eine andere Möglichkeit ist, für jede Task-Ausführung ein neues `CascadeClassifier`-Objekt zu erzeugen und dieses für die Verarbeitung von nur einem Datenpaket zu verwenden. In diesem Fall sind beide Varianten für die CPU-Implementation sehr langsam, weshalb diese Stufe nur die GPU-Implementation verwendet.

Das beschrieben Problem der Thread-Sicherheit existiert auch bei der `VideoReader`-Klasse, die mittels Mutex in der „Load Data“-Stufe verwendet wird.

5.3 Performance Vergleich

Das fertig umgebaute System wird mit der alten Struktur verglichen. Dazu werden unterschiedliche Testvideos (siehe Tabelle 5.1) aus dem Projekt 7 auf dem System in Tabelle 3.5 verarbeitet und dabei Messungen durchgeführt.

Testvideo	Datei	# Frames
A	\Testdaten\Dominik\Kreis.MP4	352
B	\Testdaten\Martin\Gerade_SchalMuetze.MP4	976
C	\Testdaten\Thekla\ZickZack.MP4	1024
D	\Testdaten\Moritz\NachUnten_Muetze.MP4	976
E	\Testdaten\ZweiPersonen\Gerade_Dominik_Moritz.MP4	844
F	\Testdaten\MehrerePersonen\Flur.MP4	1264
G	\Testdaten\MehrerePersonen\VorGebaeudeNord.MP4	8344

Tabelle 5.1: Für die Messungen verwendete Testvideos.

Tabelle 5.2 zeigt die Resultate dieser Messungen, welche für sieben unterschiedliche Testvideos durchgeführt werden. Dabei unterscheiden sich die Testdaten nicht nur in der Anzahl Bilder, sondern auch in der Menge der Aktivität, welche im Video zu sehen ist. Die Verarbeitungsgeschwindigkeit ist dabei von dieser Aktivität, nicht aber von der Bildanzahl abhängig. Ein grösseres Aufkommen von Bewegung deutet auf mehrere Personen im Bild hin und führt deshalb zu mehr Arbeit für den Kopfdetektor. Dadurch entsteht ein grösserer Rechenaufwand, als dies der Fall bei Videos mit wenig Aktivität ist.

Aus den gemessenen Werten ist ersichtlich, dass die Implementation mittels Pipeline-Framework deutlich schneller als das ursprüngliche System ist. Im Mittel benötigt das neue System 1.46-mal weniger Verarbeitungszeit als das alte System.

[ms]	Old System				Pipeline System				
Testvideo	Initial Delay	Processing	Showing	Rate [fps]	Initial Delay	Processing	Showing	Rate [fps]	Speedup
A	24	10176	10150	**34.59**	682	7233	6580	**48.67**	1.41
B	25	22245	22219	**43.88**	665	14354	15127	**67.99**	1.55
C	23	24431	24407	**41.91**	663	16739	16190	**61.17**	1.46
D	23	20315	20290	**48.04**	699	13871	14636	**70.36**	1.46
E	23	20987	20961	**40.22**	693	14228	13914	**59.32**	1.48
F	24	43310	43285	**29.18**	687	32371	31774	**39.05**	1.34
G	21	170692	170670	**48.88**	735	110455	121875	**75.54**	1.55

Tabelle 5.2: Zeitmessungen der beiden Systeme. Das neue System verarbeitet die Daten im Schnitt 1.46-mal schneller als das ursprüngliche System. Die Rate wird bezogen auf die Datenverarbeitungsdauer berechnet.

Die initiale Verzögerung l_0 ist im alten System deutlich geringer, da jeweils nur die Daten verarbeitet werden, die für das Anzeigen eines Bildes notwendig sind, bevor das System die nächsten Daten lädt. Im Pipeline-System erzeugt die längere Initialisierung zudem eine teilweise längere Dauer zwischen Anzeigen des ersten und letzten Bildes. Hauptsächlich entsteht dieser Effekt durch die Trennung zwischen der Datenverarbeitung und Datenausgabe. Eine solche Trennung ist im alten System nicht vorhanden, weshalb der Unterschied zwischen Verarbeitungszeit und Anzeigedauer der Initialisierungsdauer entspricht.

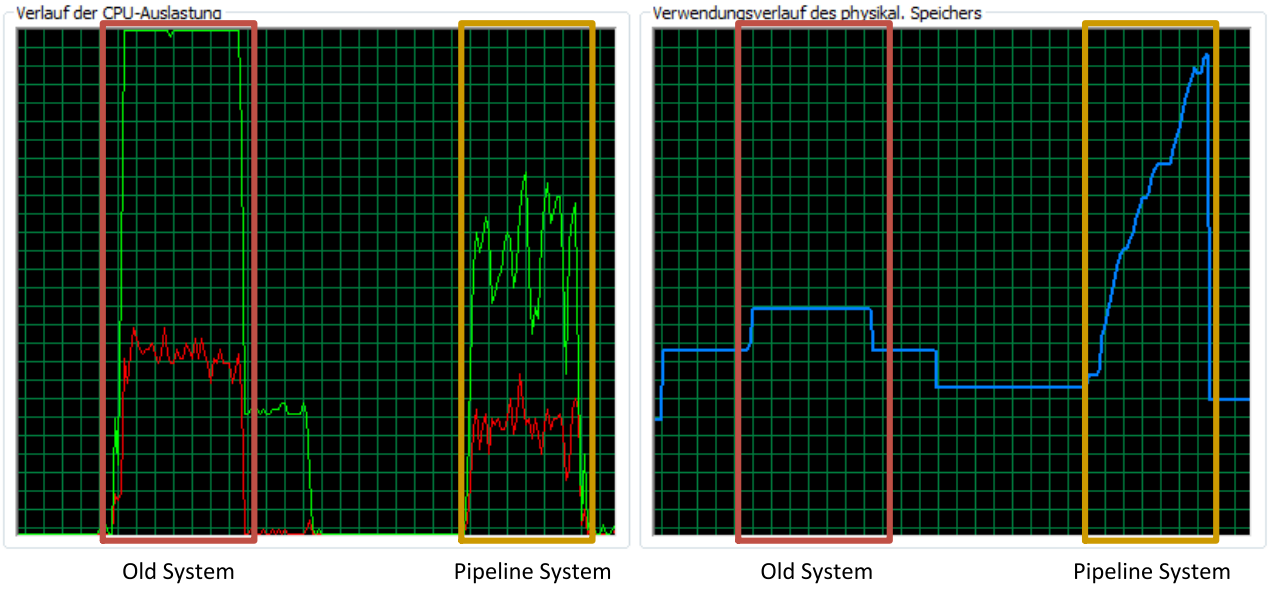


Abbildung 5.2: Betriebssystem-Auslastung während der Ausführung des alten und neuen Systems. Rote Kennlinie: CPU -Auslastung durch das Betriebssystem; Grüne Kennlinie: CPU-Auslastung durch Nutzerprogramme; Blaue Kennlinie: Füllstand des Arbeitsspeichers. Das alte System überlastet die CPU, konsumiert aber eine konstante Menge des Arbeitsspeichers. Das neue System beschränkt die Nutzung der CPU, kann aber bei nicht genügend schnellem Abholen der Daten zu Überlastung des Speichers führen.

Das alte System verwendet OpenMP zur Parallelisierung der Bewegungsdetektion und kann so die Mehrkern-CPU besser ausnutzen. Allerdings führt diese Parallelisierung zu einer Überlastung der CPU, was zu höheren Verwaltungskosten und zu niedriger Effektivität führt. Zudem kann es im Gegensatz zum heterogenen System nicht auf Lastschwankungen reagieren und erzielt deshalb eine deutlich schlechtere Performance. Dieses Lastverhalten beider Systeme ist in Abbildung 5.2 aufgezeichnet.

Darin ist ebenfalls erkennbar, dass das alte System einen konstanten Speicherverbrauch hat, wobei der Verbrauch des Pipeline-Systems schnell anwachsen kann. Dies erklärt sich durch die Verwendung des

finalen Puffers. Dadurch kann die eigentliche Verarbeitung in der Pipeline weiterlaufen, auch wenn die Daten nicht, oder nur langsam, vom Nutzer abgeholt werden. Ist die Abholrate niedriger als der Durchsatz der Pipeline steigt der Speicherbedarf deshalb kontinuierlich an und kann zu Problemen im Betriebssystem führen. Um solche Probleme zu verhindern, sollte der Nutzer die Verarbeitung der Pipeline pausieren, wenn die weitere Verwendung der Daten mehr Zeit benötigt.

6 Diskussion

Diese Master-Thesis befasst sich mit der Detektion und Erkennung von Personen in Live-Videos mithilfe eines handelsüblichen PCs mit GPU. Im ersten Teil soll ein Annotationstool entwickelt werden, welches auf einfache Weise ermöglicht, Köpfe von Personen in Videos zu markieren und mit dem korrekten Namen zu beschriften. Im zweiten Teil soll ein geeignetes Design evaluiert werden, auf welches die Anwendung einer Live-Video-Anwendung abgebildet werden kann. Dieses soll anhand von Parallelisierungsmöglichkeiten und allgemeinen Performanceeigenschaften analysiert werden. Mit den gemachten Erkenntnissen soll ein Framework inklusive heterogenem Scheduler entwickelt werden, welcher alle Rechenressourcen eines PCs ausnutzen kann. Eine bestehende Personendetektions-Anwendung soll mittels dieses Frameworks beschleunigt und zudem mittels TLD zu einer Personenwiedererkennung erweitert werden.

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

6.1 Annotations-Tool

Das entwickelte Annotations-Tool wird mittels OpenCV und Qt implementiert und erfüllt die Anforderungen mit einem minimalen aber angenehmen Bedienkonzept. Trotzdem bleibt der Vorgang der Annotierung eines Videos ein aufwendiger und langsamer Prozess. Mittels der geschriebenen `PositionData`-Klasse können die Daten effizient und lesbar gespeichert werden. Dies bietet den Vorteil, dass auch manuelle Änderungen an den Daten ermöglicht werden, welche im Tool nicht vorgesehen sind, wie z. B. die Ergänzung der Personennamen durch dessen Nachnahmen. Mittels dieser Klasse ist zudem der Grundstein für die Analyse der Detektionssoftware gelegt, da Daten geladen und abgefragt werden können.

6.2 Heterogenes Pipeline-Scheduling und -Framework

Das Pipeline-Pattern eignet sich gut für Streaming-Anwendungen und ermöglicht durch funktionale Parallelität eine grobgranulare Parallelisierung solcher Systeme. Die Pipeline selbst lässt sich gut durch das Aktoren-Modell und dessen Message-Passing-Paradigma abbilden, welches zudem Problemen in nebenläufigen Systemen vorbeugt. Für die Implementierung des Pipeline-Frameworks wird die AAL von Microsoft verwendet und die Pipeline-Stufen mittels Task-Klasse aus der PPL zur Mehrfachausführung befähigt. Zudem werden sie zu heterogenen Stufen erweitert, die für alle verfügbaren Rechengeräte eine Task-Implementation enthalten können. Dadurch entsteht eine heterogene Pipeline, in welcher die Datenpakete unterschiedliche Pfade nehmen und so unterschiedlich schnell zum Ausgang gelangen können. Dieses und weitere auftretende Probleme des heterogenen Systems werden durch einen Scheduler gehandhabt. Dieser verwaltet alle Rechenressourcen des PCs und optimiert die Pfade der Datenpakete, um eine minimale Gesamtrechenzeit, gleichmässige Ausgabeintervalle und grösstmögliche Datenlokalität zu erreichen. Die entwickelte, warteschlangenbasierte Scheduling-Strategie bewährt sich in ausführlichen Messungen mit zwei Simulationsarten gegen zwei andere Strategien und bestätigt die Praxistauglichkeit des entwickelten Pipeline-Frameworks.

Im Kontext von heterogenem Pipeline-Scheduling sind mehrfachausführbare Stufen sinnvoll, da diese ein Blockieren der gesamten Pipeline durch langsame Stufen verhindert. Zudem ist es nützlich die unterschiedlichen Task-Implementationen einer Stufe zu priorisieren, um die Verarbeitung auf dem schnellsten Rechengerät zu fördern. Eine Scheduling-Strategie ist umso effektiver, je mehr Informationen sie vom gesamten System für die Planung zur Verfügung hat. Deshalb erreicht die warteschlangenbasierte Strategie die beste Gesamtperformance.

6.3 Personenerkennung mittels Pipeline-Framework

Durch den Umbau der bestehenden Software zu einem Pipeline-System konnten diverse Schwachstellen des Frameworks erkannt und verbessert werden. Generell ist der Umbau eines bestehenden Systems mit wenig Aufwand verbunden, da das Framework alle wichtigen Teile bereits vorbereitet hat. Die Teilaufgaben der Anwendung müssen so implementiert werden, dass sie die Daten vorzugsweise nur aus den Datenpaketen beziehen. Dies sollte meist möglich sein, da eine Stream-Anwendung gut für eine Pipeline-Struktur geeignet ist. Wichtig ist, dass Task-Implementationen immer asynchron sind und bei der Verwendung anderer Komponenten, diese zwingend multi-threading-fähig sein müssen, sobald sie von mehreren Recheneinheiten gleichzeitig ausgeführt werden.

Durch den Umbau des Systems konnte die Brauchbarkeit des Pipeline-Frameworks und der mögliche Geschwindigkeitszuwachs bestätigt werden. Zudem wurde das Verhalten eines solchen Systems analysiert. Problematisch an einem System mit getrennter Verarbeitungs-Pipeline und Ausgabe ist z. B., dass es zu einer Überlastung des Arbeitsspeichers kommen kann, sollten die Daten nicht genügend schnell durch aus dem finalen Puffer abgeholt werden.

6.4 TLD-Framework

Wie im Anhang A ersichtlich ist, war ursprünglich geplant, die Erkennungssoftware aus dem Projekt 8 durch das TLD-Framework zu erweitern und so das Tracking und Wiedererkennen von Köpfen zu ermöglichen. Da allerdings sehr viel mehr Zeit als geplant in die Verbesserung des Schedulers und des Pipeline-Frameworks investiert wurde, war die Verwirklichung des TLD-Systems im Rahmen dieser Arbeit nicht mehr möglich.

Abkürzungs- und Stichwortverzeichnis

Quellenverzeichnis

[1] C. Lang, «IP712: Parallel Computer Vision: Person Data Extraction,» FHNW, Windisch, 2012.

[2] C. Lang, «IP813: Parallel Computer Vision: Heterogeneous System Architecture & Enhanced Head Tracking,» FHNW, Windisch, 2013.

[3] Qt, «Qt Project,» [Online]. Available: http://qt-project.org/. [Zugriff am 18. Oktober 2013].

[4] C. Lang, «Parallel Computer Vision: Heterogeneous Video Pipeline Framework,» 2014.

[5] C. Hewitt, P. Bishop und R. Steiger, «A Universal Modular Actor Formalism for Artificial Intelligence,» Proceeding IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence, 1973, p. 235–245.

[6] M. Chu und K. Varadarajan, «Actor-Based Programming with the Asynchronous Agents Library,» Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/magazine/ff959205.aspx. [Zugriff am 29. Oktober 2013].

[7] C. Campbell und A. Miller, «Parallel Programming with Microsoft Visual C++,» Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/library/gg675934.aspx. [Zugriff am 29. Oktober 2013].

[8] Microsoft, «Asynchronous Agents,» [Online]. Available: http://msdn.microsoft.com/library/vstudio/dd551463. [Zugriff am 30. Oktober 2013].

[9] Microsoft, «Concurrency Runtime,» [Online]. Available: http://msdn.microsoft.com/en-us/library/dd504870.aspx. [Zugriff am 31. December 2013].

[10] K. Lee, J. Y.-T. Leung und M. L. Pinedo, «Makespan minimization in online scheduling with machine eligibility,» Springer Science+Business Media, New York, 2013.

[11] S. Babu, «C++11 Smart Pointers,» [Online]. Available: http://www.codeproject.com/Articles/541067/Cplusplus11-Smart-Pointers. [Zugriff am 16. Januar 2014].

[12] D. Kalev, «C++11 Tutorial: Introducing the Move Constructor and the Move Assignment Operator,» [Online]. Available: http://blog.smartbear.com/c-plus-plus/c11-tutorial-introducing-the-move-constructor-and-the-move-assignment-operator/. [Zugriff am 16. Januar 2014].

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

[14] Z. Kalal, J. Matas und K. Mikolajczyk, Tracking-Learning-Detection, IEEE Transactions on Pattern Analysis and Machine Learning Intelligence, 2011.

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

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

[17] J.-Y. Bouguet, Pyramidal Implementation of the Lucas Kanade Feature Tracker, Intel Corporation, 2000.

[18] OpenCV, «Video Analysis, Optical Flow,» [Online]. Available: http://docs.opencv.org/master/modules/video/doc/video.html. [Zugriff am 02. Oktober 2013].

[19] G. Farneback, Two-frame motion estimation based on polynomial expansion, Lecture Notes in Computer Science, 2003.

[20] M. Tao, J. Bai, P. Kohli und S. Paris, SimpleFlow: A Non-iterative, Sublinear Optical Flow Algorithm, Computer Graphics Forum, 2012.

[21] C. Zach, T. Pock und H. Bischof, A Duality Based Approach for Realtime TV-L1 Optical Flow, Heidelberg, Germany: In Proceedings of Pattern Recognition (DAGM), 2007.

[22] J. Sanchez, E. Meinhardt-Llopis und G. Facciolo, TV-L1 Optical Flow Estimation, Image Processing On Line, 2012.

[23] F. Pernici, FaceHugger: The ALIEN Tracker Applied to Faces, European Conference on Computer Vision, 2012.

[24] G. Nebehay, «OpenTLD,» [Online]. Available: http://gnebehay.github.com/OpenTLD/. [Zugriff am 06. März 2013].

[25] E. Gravdal, «Android Store: OpenTLD,» [Online]. Available: https://play.google.com/store/apps/details?id=com.gravdal.opentld&hl=de. [Zugriff am 03. Oktober 2103].

[26] Microsoft, «Asynchronous Message Blocks,» [Online]. Available: http://msdn.microsoft.com/en-us/library/vstudio/dd504833%28v=vs.100%29.aspx#. [Zugriff am 31. Oktober 2013].

[27] C. Lang, «Non busy waiting for completion of parallel_for_each,» [Online]. Available: http://social.msdn.microsoft.com/Forums/en-US/e167bdb3-6541-4efa-ac64-62c3ad1d8350/non-busy-waiting-for-completion-of-parallelforeach?forum=parallelcppnative. [Zugriff am 16. Januar 2014].

[28] A. Vermeulen, G. Beged-Dov und P. Thompson, «The Pipeline Design Pattern,» Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems, 1995.

[29] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad und M. Stal, «Pipes and Filters,» in *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, Siemens AG, Germany, Wiley, 1996, pp. 53-70.

[30] A. Navarro, R. Asenjo, S. Tabik und C. Cascaval, «Analytical Modeling of Pipeline Parallelism,» Parallel Architectures and Compilation Techniques, Raleigh, NC USA, 2009.

[31] W.-K. Liao, A. Choudhary, D. Weiner und P. Varshney, «Performance Evaluation of a Parallel Pipeline Computational Model for Space-Time Adaptive Processing,» The Journal of Supercomputing, 2005.

[32] F. Bower, D. Sorin und L. Cox, «The impact of dynamically heterogeneous multicore processors on thread scheduling,» Micro, IEEE, 2008.

[33] F. Glover und M. Laguna, Tabu Search, Boston: KluwerAcademic Publishers, 1997.

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, 14.02.2014

Unterschrift

Lang Christian

Anhang

A. Klärung

Dieser Anhang enthält die Klärung für die Master-Thesis. Sie enthält die Aufgabenstellung, die Motivation und gewisse Vorschläge für das Vorgehen.

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

Im dritten Teil der Arbeit soll die Personendetektion zu einer Personenerkennung (Recognition) in Videos ausgebaut werden. Dabei sollen erkannte Personen in Bildsequenzen verfolgt und die dabei anfallenden Personenbilddaten für das Lernen der verbesserten Klassifizierungsalgorithmen verwendet werden. Dabei ist den Fragen nachzugehen, wie ein solches System realisiert werden kann, wie sichergestellt wird, dass alle Recheneinheiten gleichzeitig im Einsatz sind und wie überprüft werden kann, ob das System gut funktioniert.

3. Organisation

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

Experte: Christof Sidler
Super Computing Systems (SCS)

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

4. Rückblick

Im Projekt 7 und 8 wurden Grundlagen der Bildverarbeitung, heterogenen Systemarchitekturen (HSA) und Machine-Learning (ML) im Bereich der Personenerkennung erarbeitet. Dazu wurden Technologien wie OpenCV, OpenMP, C++ AMP und OpenCL verwendet. Zudem wurde ein Videoverarbeitungssystem aufgebaut, welches Videostreams effizient verarbeiten kann und als Grundlage für die Personenerkennungsanwendung verwendet wird. Darauf wurde ein Bewegungsdetektor aufgebaut und mithilfe von Rejection-Cascades ein Kopfdetektor trainiert, welcher Köpfe aus allen Blickrichtungen erkennen kann. Diese Kopfansichten sind notwendig für das Erstellen eines Kopfmodells der jeweiligen Person.

5. Ziele

Folgende Ziele sollen erreicht werden:

1. Es soll eine Zusatzsoftware entwickelt werden, um die Testvideos mit den korrekten Annotationen der aufgenommenen Personen zu versehen. Diese Annotationen sollen gespeichert und während dem Ausführen der eigentlichen Erkennungssoftware für statistische Auswertungen verwendet werden.
2. Das weiterentwickelte System soll die CPU und GPU gleichermassen auslasten. Dies wird mithilfe einer Bearbeitungspipeline erreicht, in welcher jede Teilaufgabe entweder auf CPU oder GPU erledigt werden kann. So kann ein Scheduler jeweils entscheiden, auf welcher Ressource ein anstehender Task abgearbeitet werden soll. Dieser Scheduler soll implementiert werden.
3. Die vorhandene Software soll weiterentwickelt werden, um die erkannten Personen zu verfolgen und wiederzuerkennen. Dazu soll sie ähnlich dem Tracking-Learning-Detection (TLD) Framework aufgebaut werden. Dabei soll das Tracking und die Erkennung der Köpfe implementiert werden.

6. Teilaufgaben

Der gesamte Projektumfang beträgt 27 ECTS und somit 810 Arbeitsstunden. Die Arbeit gliedert sich in die Teilaufgaben, die in den folgenden Unterkapiteln beschrieben werden.

6.a) Annotationstool

Durch Vergleichen der erzielten Detektionen mit dem echten Auftreten der Personen in einem Testvideo können Statistiken erstellt werden, welche Aussagen über die Präzision der Personenerkennung zulassen. Dazu werden die Daten der echten Positionen benötigt, welche mittels eines Annotationstools aufgenommen werden sollen.

Dieses Tool soll es ermöglichen, durch simples Zeigen mit der Maus auf einen Kopf, während das Video läuft, die Positionsdaten zu erfassen. Dabei soll die Ablaufgeschwindigkeit des Videos einstellbar sein und der Nutzer soll das Video jederzeit pausieren und erneut starten können. Das Video kann beliebig oft abgespielt und der Startpunkt über eine Fortschrittsleiste gewählt werden. Beim laufenden Film kann durch Links-Klicken ins Bild die Position der gewählten Person definiert werden. Aktuelle Detektionspositionen sollen auch wieder entfernt werden können und die Grösse des Detektionsfensters einstellbar sein. Da mehrere Personen im Bild erscheinen können, soll eine Liste mit deren Namen gefüllt werden können. Es wird jeweils nur der Positionsverlauf der markierten Person bearbeitet.

Um dieses Tool zu implementieren soll OpenCV mit Qt-Support und dessen YAML-Persistenz (Klasse `FileStorage`) verwendet werden.

6.b) Heterogene Bearbeitungspipeline

Die Fähigkeiten der HSAs soll eingesetzt werden, um die Video-Verarbeitung in Echtzeit zu ermöglichen. Eine Bearbeitungspipeline eines einzelnen Video-Frames soll so strukturiert werden, dass Abhängigkeiten und mögliche Parallelisierungen ersichtlich werden. Dadurch wird es möglich, einen eigenen Scheduler zu implementieren, welcher die Tasks mittels HSAs auf die einzelnen Rechenwerke verteilt. Dieser Scheduler soll sicherstellen, dass die Vorteile eines Videostreams (gleichzeitige Verarbeitung mehrerer Frames) ausgenutzt und möglichst immer alle Rechenwerke ausgelastet werden.

i) Asynchrone Ausführung

Die Voraussetzung für ein solches System ist, die Möglichkeit Aufgaben asynchron an die Prozessoren zu verteilen. Dies wird in OpenCL durch die Methode `enqueueNDRangeKernel` und dem mitgelieferten `event` ermöglicht. Das `event`-Objekt ermöglicht eine Statusabfrage der Aufgabe. In OpenCV wird die `Stream`-Klasse für asynchrone Ausführung auf der GPU verwendet. Allerdings ist der Funktionsumfang dieser Klasse sehr beschränkt. Unter AMP wird die Funktion `parallel_for` aus der PPL verwendet.

Grundsätzlich kann die Asynchronität durch einzelne Threads für jeden blockierenden Aufruf implementiert werden. Dabei erzeugt ein Scheduler-Thread für jeden Task, der asynchron ausgeführt werden soll, einen separaten Task-Thread, welcher die parallele Ausführung auf dem Beschleuniger initiiert und sich beim Beenden der Aufgabe mit einem Callback zurückmeldet. Nachteil dieser Variante ist, dass dadurch in jedem Fall ein Thread und dessen CPU-Prozess blockiert werden.

ii) Bearbeitungspipeline

Die Bearbeitungspipeline beschreibt alle Schritte des Systems, welche während der Verarbeitung eines einzelnen Frames des Videostreams abgehandelt werden. Sie erlaubt die Analyse der Abhängigkeiten der sogenannten Pipeline-Stufen und allfällige Parallelisierungsmöglichkeiten. Im Kontext der Videoverarbeitung dient sie dazu, mögliche Optimierungen durch das gleichzeitige Bearbeiten von mehreren Frames zu erkennen und zu ermöglichen. Dazu sollen die Stufen der Pipeline auf CPU oder GPU ausführbar sein, um so eine ähnliche Optimierung wie bei einer Befehlspipeline des Prozessors zu erhalten. Durch das parallele Bearbeiten von mehreren Frames (z.B. Stufe X von Frame 1 wird auf CPU und Stufe Y von Frame 2 auf GPU verarbeitet) wird zwar die Ausführungsdauer der kompletten Pipeline eines einzelnen Frames verlängert, da aber mehrere Frames gleichzeitig bearbeitet werden, steigt die Durchsatzrate insgesamt an.

Ein Entwurf der Bearbeitungspipeline ist in Abbildung A.1 ersichtlich. Die Funktionalität der einzelnen Stufen wird im Unterkapitel 6.c) behandelt.

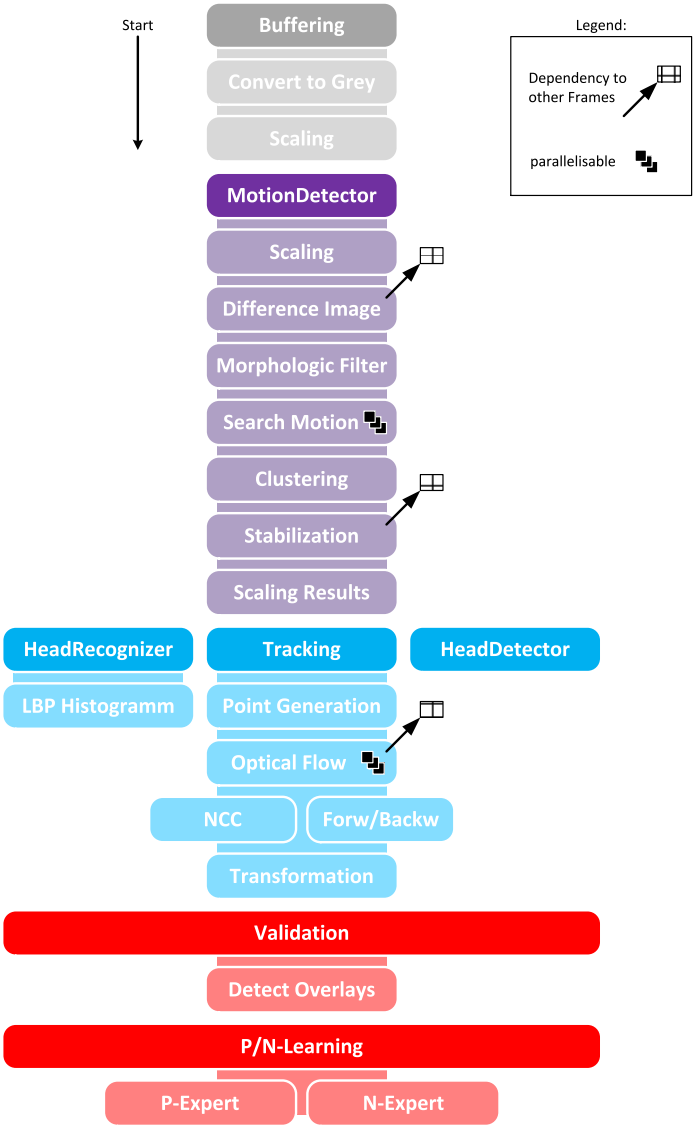


Abbildung A.1: Entwurf der Bearbeitungspipeline. Die Abarbeitung erfolgt von oben nach unten. Nebeneinanderliegende Stufen können unabhängig voneinander parallel ausgeführt werden. Alle Stufen sind implizit von der oberen Stufe abhängig. Die markierten Stufen hängen zusätzlich von vorhergehenden Frames ab.

Wie stark sich die Abarbeitung der Stufen durch Pipelining optimieren lässt, hängt dabei einerseits davon ab, ob die einzelnen Stufen für beide Prozessortypen implementierbar sind, andererseits von den Abhängigkeiten zu vorhergehenden Frames.

Um die Stufen zu verwalten, soll eine Task-Klasse erstellt werden, welche jeweils den Index des dazugehörigen Bildes und die unterstützten Rechenwerke enthält. Zudem soll direkt über dieses Task-Objekt die Ausführung asynchron gestartet und überwacht werden können. Wie erwähnt, muss zudem die Abhängigkeit zu Tasks anderer Bilder darin vermerkt werden können.

iii) HSA-Scheduler

Die zuvor beschriebenen Task-Objekte werden vom HSA-Scheduler verwendet, um die einzelnen Stufen optimal auf den Rechenwerken abarbeiten zu lassen. Der Scheduler soll so implementiert werden, dass er dynamisch entscheidet, welcher Task auf welchem Prozessor bearbeitet wird. Dazu analysiert er die aktuelle Auslastung der Recheneinheiten und entscheidet fortlaufend, in welche Warteschlange der Task eingereiht wird.

Bei der effektiven Ausführung soll die Parallelität durch das Pipelining von mehreren Video-Frames entstehen. Es soll also auf jeder Recheneinheit irgendeine Teilaufgabe eines Frames berechnet werden. Dabei werden die Tasks in drei Ausführungskategorien unterteilt: CPU, GPU oder beides. Das bedeutet es wird Tasks geben, welche nur auf der CPU ausgeführt werden können, solche die nur auf der GPU laufen und solche, die für beide Plattformen implementiert sind. Der Scheduler wird entscheiden, welcher Task als nächstes ausgeführt werden darf, wenn eine Ressource frei wird. Eine mögliche Variante wäre, dass jeweils ein Task des Bildes mit dem niedrigsten Index priorisiert wird. Als zweites Kriterium wird beachtet, ob der nächste Task dieses Bildes auf der frei gewordenen Recheneinheit ausgeführt werden kann. Falls nicht, wird ein Task des nächsten Bildes gewählt. Die grafische Darstellung eines solchen Ablaufes ist in Abbildung A.2 ersichtlich.

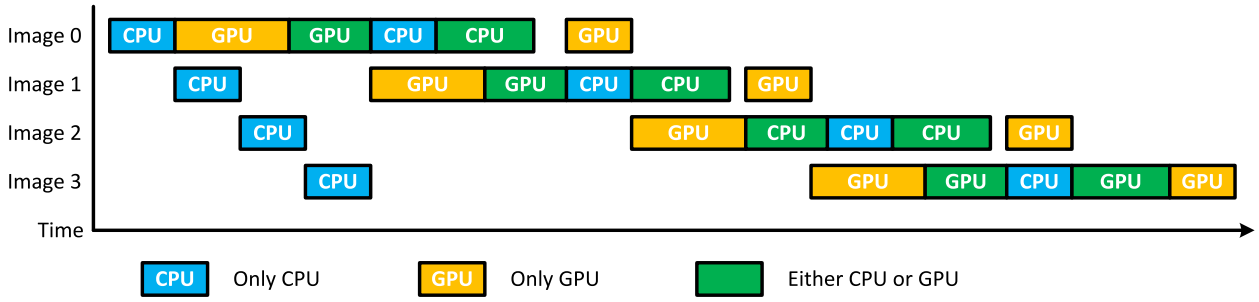


Abbildung A.2: Pipelining auf zwei unterschiedlichen Ressourcen: CPU und GPU.

Es wird jeweils das Bild mit dem niedrigsten Index bevorzugt. Im Beispiel ist dies das Bild 0. Der Scheduler wird jeweils, nachdem eine Ressource frei geworden ist, entscheiden, welcher Task als nächstes darauf ausgeführt wird. Somit sind im Idealfall immer alle Rechenwerke beschäftigt. Da es aber möglich ist, dass alle wartenden Tasks nur auf ein und derselben Art von Rechenwerk ausgeführt werden können, kann es zu Wartezeiten kommen, in welchen nicht alle Ressourcen ausgenutzt werden. Um dies zu verhindern, ist es wichtig, dass möglichst alle Tasks auf allen verfügbaren Rechenwerken ausgeführt werden können.

Ein Problem, das in Abschnitt 6.b) ii) erwähnt wurde, sind die Abhängigkeiten einzelner Tasks von anderen Bildern. Z. B. der in Abbildung A.1 markierte Task „Difference Image“ benötigt zwingend das skalierte Graustufenbild des vorherigen Frames um ausgeführt zu werden. Diese Abhängigkeiten müssen ebenfalls vom Scheduler beachtet werden. Zudem erzwingen solche „Inter-Frame-Dependencies“ eine Art Concurrent-Buffering der Frames, welche ermöglicht, dass alle Tasks auf eventuell benötigte Resultate anderer Frame-Tasks zugreifen können.

Die Entwicklung des HSA-Schedulers soll unabhängig vom TLD-Framework stattfinden, um unnötige Komplikationen beider Teilbereiche auszuschliessen. Anstatt der echten Tasks sollen deshalb „Dummy“-Tasks erstellt werden, welche bei der Ausführung für eine bestimmte Dauer Rechenleistung einer entsprechenden Recheneinheit verbrauchen. Dies kann z.B. durch eine rechenintensive Kalkulation oder eine simple Schleife mit Zähler implementiert werden. Mittels dieser Vereinfachung kann sichergestellt werden, dass auftretende Probleme vom Scheduler her rühren und nicht von den spezifischen Task-Implementationen. Ebenfalls soll die Performance-Analyse mit Hilfe des Concurrency-Visualizers vereinfacht werden.

6.c) TLD-Framework

Die durch den Head-Detector erkannten Köpfe dienen als Ausgangspunkt für die Verfolgung und Wiedererkennung der Personen. Sie sollen für die Initialisierung eines Langzeit-Tracking-Systems dienen, welches sich an dem TLD-Framework von Kalal et al. [14] orientiert. Abbildung A.3 zeigt ein solches System, welches mit dem Motion- und Head-Detector bereits existierende Teile enthält.

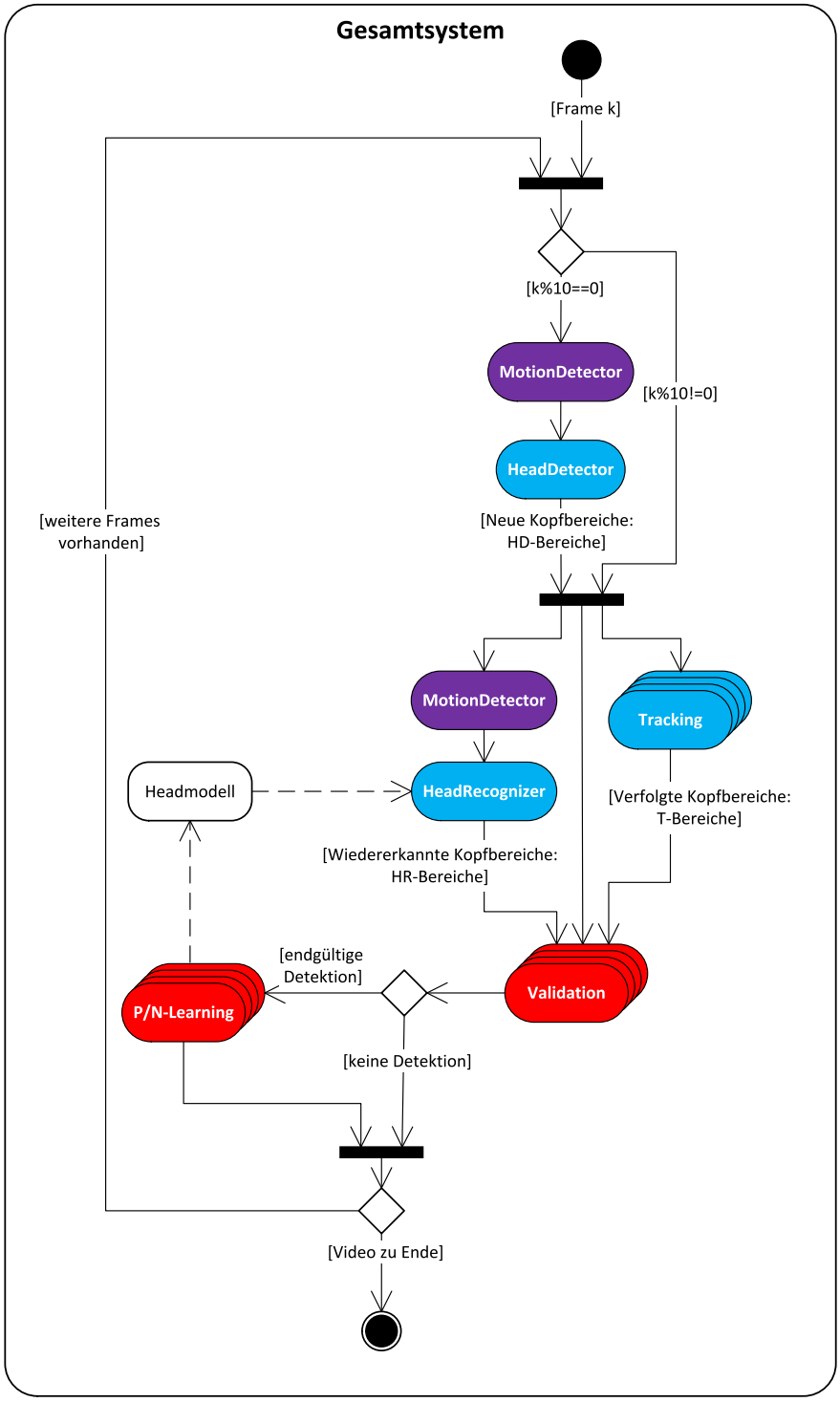


Abbildung A.3: UML-Aktivitätsdiagramm des zu realisierenden Frameworks, welches sich an TLD orientiert. Erweiterungen im Vergleich zu TLD sind der Head-Recognizer, der Motion- und der Head-Detector. Es wird grafisch hervorgehoben, dass die Module Tracking, Validation und P/N-Learning für jede Person separat und somit mehrfach pro Frame ausgeführt werden.

Die folgenden Abschnitte beschreiben die noch zu implementierenden Module der bestehenden Software, wobei jeweils alle Vorgänge für jede Person separat ausgeführt werden. Genauer bedeutet dies, dass der Head-Detector und -Recognizer das einzelne Frame nur einmal verarbeiten, das Tracking, die Validierung und das P/N-Learning jedoch für jede aktuell verfolgte oder bekannte Person einzeln das Bild verarbeitet.

Die wichtigste übernommene Eigenschaft von TLD ist die Kombination des Kurzzeit-Trackings mit dem Head-Recognizer zu einem sich gegenseitig kontrollierenden System. Dadurch wird das Online-Trainieren des Head-Recognizers mittels dem in Abschnitt 6.c) iv) beschriebenen P/N-Learning [15] ermöglicht. Somit wird das „Supervised-Learning-System“, welches vom Nutzer alle annotierte Daten benötigt, um zu funktionieren, zu einem „Semi-Supervised-Learning-System“, welches nur einmalig eine Initialisierung mit der gewünschten Art Daten benötigt. Zudem kommt im Vergleich zur Version von Kalal et al. der Head-Detector hinzu, welcher die Initialisierung übernimmt und so das System autonom arbeiten lässt. Das endgültige System ist deshalb „unsupervised“.

i) Kurzzeit-Tracking

Das Tracking ist eine der neuen Problematiken, die gelöst werden müssen. Dabei soll ein vom Head-Detector zuvor erkannter Kopf möglichst lange weiterverfolgt werden, ohne eine neue Suche von Köpfen zu starten. Die dadurch dazugewonnenen Ansichten desselben Kopfes werden benötigt, um das Kopfmodell und somit auch die Wiedererkennung zu verbessern. Kalal et al. [16, 14] verwenden in ihren Implementationen von TLD jeweils den Algorithmus von Lucas und Kanade [17] für die Berechnung des optischen Flusses. Der optische Fluss wird ermittelt, indem einzelne Punkte des zu verfolgenden Bereichs im neuen Bild wiedererkannt werden und die Verschiebung zwischen dem Ursprungspunkt und dem neuen Punkt berechnet wird. OpenCV bietet für die Berechnung des optischen Flusses unterschiedliche Varianten [18] an:

- Lucas und Kanade [17]
- Gunnar Farneback [19]
- Simple Flow [20]
- Dual TV L1 [21, 22]

Nach der Berechnung der neuen Punkte, wird evaluiert, welche Punkte zuverlässig verfolgt wurden. Dazu wird einerseits der „Normalized Correlation Coefficient“ (NCC) verwendet, welcher die Ähnlichkeit zweier Bildbereiche angibt, andererseits wird die Technik des „Forward-Backward-Error“ [16] eingesetzt, um die Punktverschiebung zu validieren. Abbildung A.4 erklärt diese Technik übersichtlich.

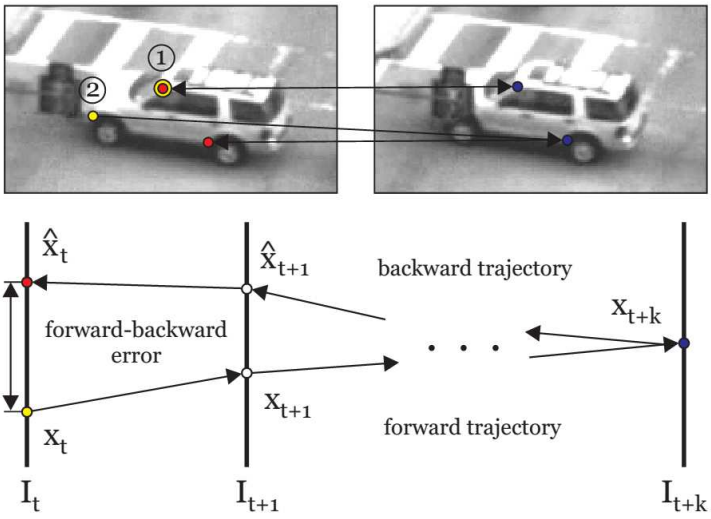


Abbildung A.4: Darstellung des Forward-Backward-Errors. Als erstes werden die Punkte 1 und 2 aus dem linken Bild im rechten gesucht. Die gefundene Verschiebung wird mittels umgekehrtem Tracking (vom rechten zum linken Bild) validiert, wobei der Punkt 1 korrekt und der Punkt 2 falsch abgebildet wird. Bild aus [16].

Beim Forward-Backward-Error wird die Tatsache ausgenutzt, dass ein Punkt in beide Richtungen der Zeitachse verfolgt werden kann, was immer zur selben Trajektorie führt. Ein Punkt x zum Zeitpunkt t wird als x_t bezeichnet. Wird dieser Punkt in den nachfolgenden Bildern eines Videostreams verfolgt, befindet er sich nach k Schritten an der Stelle x_{t+k}. Wurde der Punkt korrekt verfolgt, sollte es möglich sein, denn Punkt x_{t+k} in umgekehrter Richtung zu verfolgen und wieder im Ursprungspunkt x_t anzukommen. Dies ist der Fall beim Punkt 1 aus der Abbildung A.4, was die Korrektheit des Trackings bestätigt. Bei Punkt 2 ist dies allerdings nicht der Fall, da die gesuchte Stelle des Radkastens im rechten Bild verdeckt ist. Somit führt das Backward-Tracking nicht zum Ursprungspunkt und erkennt dadurch eine fehlerhafte Verfolgung.

Abbildung A.5 zeigt den Ablauf des gesamten Tracking-Vorgangs. Nachdem in der zu verfolgenden Bounding-Box ein regelmässiges Gitter von Punkten erzeugt wurde, werden diese Punkte mittels optischem Fluss verfolgt und mittels NCC und Forward-Backward-Error jeweils nur 50% aller Punkte als präzise verfolgbar ausgewählt.

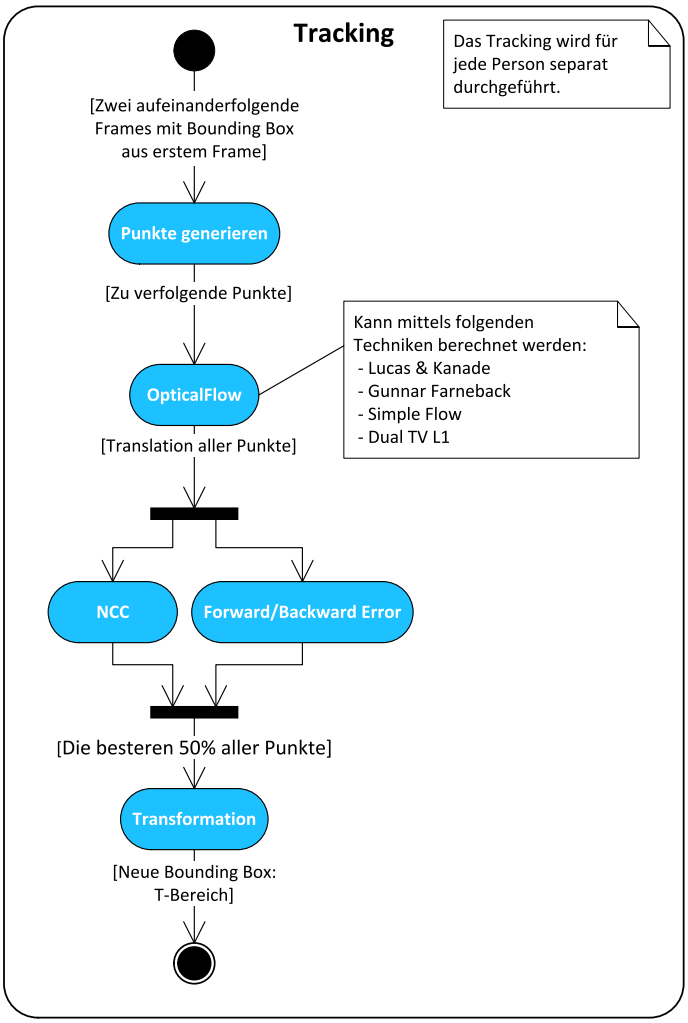


Abbildung A.5: Ablauf des Trackings. Nach der Initialisierung der Punkte werden diese mittels optischem Fluss im folgenden Bild verfolgt und danach die 50% der besten Punkte zur effektiven Berechnung der neuen Bounding-Box verwendet.

Die neue Bounding-Box wird mittels den besten Punkte des Trackings berechnet. Um die Verschiebung und Verzerrung der Box zu ermitteln, wird das „Median Flow“ [16] Verfahren verwendet. Dabei wird zur Berechnung der Verschiebung der Box der Median aller Einzelverschiebungen pro räumliche Dimension benutzt. Die Verzerrung der Box wird ebenfalls für beide Dimensionen einzeln ermittelt, indem der Median aller Distanzveränderungen zwischen allen Punktepaaren verwendet wird. Schlussendlich ergibt sich aus den beiden Faktoren (jeweils zweidimensional) die Grösse und Position der neuen Bounding-Box, welche als T-Bereich bezeichnet wird.

ii) Head-Recognizer

Um Personen zu erkennen, wenn sie erneut detektiert werden, wird ein Mehrklassen-Classifier benötigt, der Köpfe von Personen unterscheiden kann. Dazu muss er mittels Bildern von den zu erkennenden Köpfen trainiert werden. Die Implementation eines solchen Recognizers kann einerseits auf dem template-basierten Ansatz von Kalal aufbauen oder ein abstrakteres System wie das des „Local Binary Pattern Histogramms“ (LBPH) verwenden, welches durch die Klasse `FaceRecognizer` in OpenCV implementiert wird. Diese Klasse bietet neben LBPH auch Eigenfaces und Fisherfaces als Features an. Allerdings unterstützt sie die für das Online-Training benötigte Update-Funktion nur in Kombination mit LBPH. Ein Handicap dieser Klasse ist, dass nur positive Samples (Bilder von zu erkennenden Köpfen) für das Training verwendet werden. Um die negativen Samples (Hintergrundbilder) trotzdem zur Verbessern des Classifiers verwenden zu können, könnten diese mit einem eigenen Label für Hintergrundbereiche benutzt werden. Eine weitere Variante wäre, alle Samples abzuspeichern und den Classifier nach jeder Änderung an diesen Templates komplett neu zu trainieren. Dadurch könnten einerseits die negativen Samples dazu verwendet werden, um zuvor als positiv markierte Samples zu entfernen, andererseits könnten so auch die Eigenface- und Fisherface-Implementationen der Klasse `FaceRecognizer` verwendet werden.

iii) Validation

Der Validierungsvorgang wird zur Auswahl der definitiven Bounding-Box benötigt, da die drei vorhergehenden Systeme: Tracking (T-Bereich), Head-Recognizer (HR-Bereiche) und Head-Detector (HD-Bereiche) mehrere Bereiche als gefundene Köpfe liefern. Abbildung A.6 zeigt die Evaluation der zu verfolgenden Kopfposition.

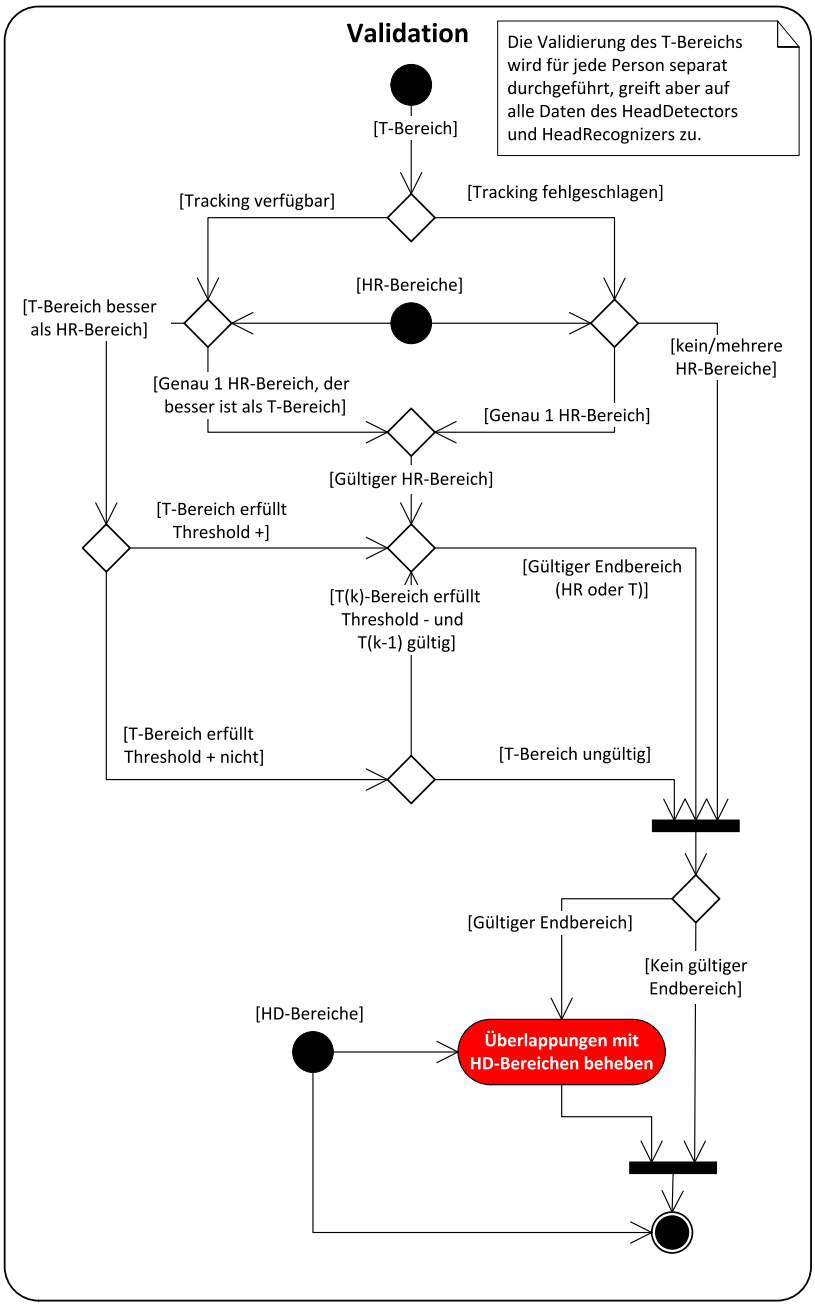


Abbildung A.6: Evaluation aller gefundenen Kopfbereiche. Es wird entweder die beste oder keine Kopfposition zurückgegeben. Zusätzlich werden alle neuen Kopfpositionen des HeadDetectors (HD-Bereiche), die keiner Person zugeordnet werden können, zurückgegeben.

Um die neue Kopfposition (Endbereich) einer zuvor erkannten Person zu finden, gibt es die folgenden Möglichkeiten. Falls das Tracking keine Position und der Head-Recognizer nur genau einen HR-Bereich liefert, wird dieser als Endbereich verwendet und das Tracking im nächsten Frame neu initialisiert. Falls das Tracking erfolgreich ist, aber einen schlechteren Vertrauens-Wert (Confidence) als das allenfalls einzige Resultat des Head-Recognizers, wird der HR-Bereich gewählt. Falls dieser ein schlechteres Resultat als das Tracking, keines oder mehrere liefert, wird der T-Bereich weiter evaluiert. Er wird akzeptiert, falls dessen Vertrauen höher als ein „Threshold Plus“ ist. Falls dieser schlechter ist, kann das Resultat trotzdem akzeptiert werden, wenn es mindestens einen Vertrauens-Wert des „Threshold Minus“ erreicht und der T-Bereich des vorherigen Bildes ($T_{(k-1)}$) ebenfalls gültig war. In allen anderen Fällen wird kein Endbereich des Kopfes zurückgeliefert, was als fehlgeschlagene Detektion/Tracking interpretiert wird.

Für die Iterationen, in welchen zusätzliche HD-Bereiche vorhanden sind, wird überprüft ob solche sich mit einem allfälligen gültigen T-Bereich decken. Alle anderen werden als neue Kopfbereiche anderer Personen zurückgegeben.

iv) P/N-Learning

Das von Kalal et al. vorgeschlagenen P/N-Learning [15] ermöglicht nicht nur die als gültig validierten Endbereiche für die Verbesserung des Head-Recognizers zu verwenden, sondern auch alle anderen HR-Bereiche. Dazu werden zwei Experten-Systeme aufgebaut, welche die ausgeschlossenen positiven und negativen Samples erkennen. Abbildung A.7 zeigt das Vorgehen der P- und N-Experten.

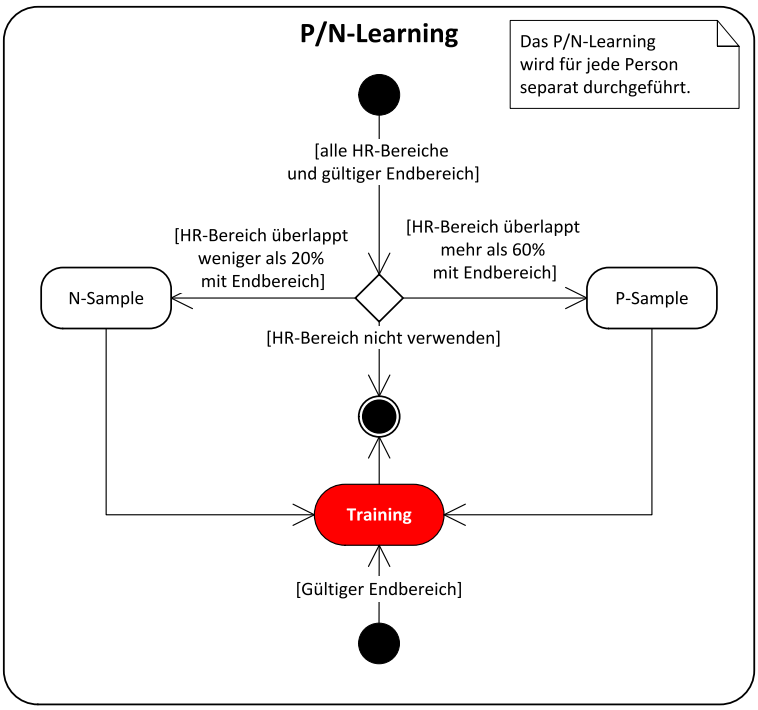


Abbildung A.7: Trainieren des Head-Recognizers mit den als Hintergrund oder Kopf erkannten Bildern. Diese werden aus denn nicht verwendeten HR-Bereichen durch die P- und N-Experten gewonnen. Der gültige Endbereich wird immer für das Training verwendet.

Die HR-Bereiche, welche durch die Validierung nicht als Endbereich erkannt werden, können mittels den Experten überprüft werden. Dadurch lassen sich diese nicht annotierte Daten deklarieren und ebenfalls für das Training des Head-Recognizers verwenden. Der P-Experte geht davon aus, dass sich ein gefundener Kopf immer auf einer Trajektorie durch den Bildbereich bewegt und korrekte Erkennungen nur in der Nähe dieser Trajektorie auftauchen. Der dazu gehörige P-Constraint besagt also, dass ein Objekt im ersten Bild keine willkürliche Position im zweiten Bild einnehmen kann. Vorausgesetzt die Aufnahmegeschwindigkeit ist auf die maximale Bewegungsgeschwindigkeit des Objektes abgestimmt. Ein Überschallflugzeug mit einer Rate von 30 fps zu filmen, würde z. B. keinen Sinn und den Constraint hinfällig machen. In unserem Fall jedoch, kann der P-Experte alle Bereiche die sich genügend stark mit dem gültigen T-Bereich überlappen als positives Sample deklarieren.

Der N-Experte hingegen nimmt an, dass ein detektierter Kopf immer nur einmal im Bild auftauchen kann. Dadurch kann ein N-Constraint formuliert werden, welcher die Tatsache beschreibt, dass ein einmaliges Objekt immer nur einmal im selben Bild auftauchen kann. Somit kann der N-Experte davon ausgehen, dass alle gefundenen Bereiche, die genügend Abstand zum gültigen Trackingbereich haben, False-Positives sind. Diese Bereiche werden als negative Samples deklariert und können ebenfalls für das Online-Training verwendet werden. Alle Bereiche die nicht von den Experten deklariert werden, bleiben ohne Annotation

und werden verworfen. Durch dieses Vorgehen können die Fehler vom Head-Recognizer erkannt und durch das Training ausgemerzt werden.

6.d) Weitere Aufgaben / Informationen

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

- ALIEN-Tracker [23]: Ähnlich TLD (Predator).
- Folgende Technologien sollen im Auge behalten werden: OpenVX, Soft-Cascades und das OCL-Modul in OpenCV.

7. Produkte

Als Produkte dieser Masterthesis sind die Dokumentation und die aus mehreren Modulen bestehende Software zu betrachten. Die Software soll automatisch Personen in Echtzeit-Videos erkennen, verfolgen und bei erneutem Erscheinen wiedererkennen. Dazu wird ein Langzeit-Tracking bestehend aus dem Head-Recognizer und dem Tracking-Modul implementiert. Der Recognizer wird dabei durch ein P/N-Learning online trainiert und so kontinuierlich verbessert. Die notwendigen annotierten Daten werden dabei gemeinsam durch das P/N-Learning und Validierungs-Modul geliefert. Die grösste Neuerung im Vergleich zu den bestehenden TLD-Implementationen [24, 25] ist die Tatsache, dass mehrere Personen gleichzeitig detektiert und wiedererkannt werden können. Zudem benötigt TLD jeweils eine manuell durch den Nutzer durchgeführte Markierung des zu verfolgenden Objektes.

Um die Software echtzeit-fähig zu machen, werden die einzelnen Module so vorbereitet, dass sie möglichst auf der CPU oder der GPU ausführbar sind. So wird es dem implementierten HSA-Scheduler ermöglicht, eine Verarbeitungspipeline zu verwalten, welche alle Rechenwerke eines PCs möglichst vollständig auslastet. Dazu soll der Scheduler mit den im P8 erarbeiteten Erkenntnissen über die verfügbaren HSAs optimal aufgebaut werden.

Damit die Detektionsgenauigkeit der Anwendung empirisch ermittelt werden kann, wird ein Hilfswerkzeug entwickelt, welches es auf einfache Weise ermöglicht, die vorhandenen Testdaten mit den echten Positionen zu annotieren.

Zusätzlich wird zu den Thematiken des P9 ein wissenschaftliches Paper entworfen und zur Veröffentlichung eingereicht.

8. Semesterplanung

Das Semester ist wie in Abbildung A.8 gezeigt verplant. Woche 52 und 1 beinhalten die Weihnachtsferien. Die Woche 48 wird teilweise für die Vorbereitung auf das CS-Seminar in der Woche 49 verwendet.

Aktuelle Position bei Planung

Termine				PK						CsS			Ferien									
Kalenderwoche		40	41	42	43	44	45	46	47	48	49	50	51	52	1	2	3	4	5	6	7	8	9
Thema	**MS**																						
Klärung	**82**	32	50																				
Paper	**86**			10				30	10				20		16								
Annotationstool	**60**			40	20																		
Bearbeitungspipeline	**210**																						
Scheduler	60				30	30																	
Impl. Task-Blöcke	110					20	40				10		10				10	20					
Refactoring	40						10											10	20				
TLD-Framework	**250**																						
Tracking	100								40	20						10	20	10					
HeadRecognizer	40										30					10							
Validation	110											50	20			10	20	10					
Dokumentation	**170**							20								20			30	50	50		
Präsentation	**50**																					50	
Total:	**908**																						

Klärung — Projektabgabe — Präsentation

Abbildung A.8: Projektplanung

B. OpenCV mit Qt kompilieren

Um OpenCV mit Qt [3] zu kompilieren, muss dieses als erstes installiert und dann OpenCV mit der `With_QT`-Variablen in CMake kompilieren. Tabelle B.1 zeigt die benötigten Variablen, welche bereits in [2] beschrieben sind. Tabelle B.2 gibt einen Hinweis, wo die benötigten Dateien zu finden sind.

Aktivieren / Ausfüllen	Deaktivieren
WITH_OPENGL	CUDA_ATTACH_VS_BUILD_RULE_TO_CUDA_FILE
WITH_OPENCL	BUILD_TEST
WITH_NVCUVID	BUILD_PERF_TEST
WITH_QT	BUILD_DOCS
WITH_CUBLAS	BUILD_EXAMPLES
WITH_EIGEN	
EIGEN_INCLUDE_PATH	
CUDA_FAST_MATH	

Tabelle B.1: Einstellungen in CMake für das erfolgreiche Kompilieren von OpenCV mit CUDA 5.5, Qt 5.2, OpenGL, OpenCL und Visual Studio 2012 Compiler vc11.

Name	Wert
QT_QMAKE_EXECUTABLE	D:/Programme/Qt/5.2.0/msvc2012_64_opengl/bin/qmake.exe
CMAKE_LIBRARY_PATH	C:/Program Files (x86)/Microsoft SDKs/Windows/v7.1a/Lib
Qt5Concurrent_DIR	D:/Programme/Qt/5.2.0/msvc2012_64_opengl/lib/cmake/Qt5Concurrent
Qt5Core_DIR	D:/Programme/Qt/5.2.0/msvc2012_64_opengl/lib/cmake/Qt5Core
Qt5Gui_DIR	D:/Programme/Qt/5.2.0/msvc2012_64_opengl/lib/cmake/Qt5Gui
Qt5OpenGL_DIR	D:/Programme/Qt/5.2.0/msvc2012_64_opengl/lib/cmake/Qt5OpenGL
Qt5Test_DIR	D:/Programme/Qt/5.2.0/msvc2012_64_opengl/lib/cmake/Qt5Test
Qt5Widgets_DIR	D:/Programme/Qt/5.2.0/msvc2012_64_opengl/lib/cmake/Qt5Widgets

Tabelle B.2: Variablen aus CMake mit ersichtlichen Pfaden zu den einzelnen benötigten Qt-Dateien.

Soll OpenCV in Kombination mit OpenGL verwendet werden, muss darauf geachtet werden, dass die OpenCV-Datei `cvConfig.h` inkludiert wird, auf den SDK-Pfad von Windows (z. B. `c:\Program Files (x86)\Microsoft SDKs\Windows\v7.1A\Lib\x64`), in welchem sich die `lib`-Dateien von OpenGL befinden, als `Library`-Pfad verwiesen wird und dass die Dateien `glu32.lib` und `opengl32.lib` als `AdditionalDependencies` angegeben werden.

C. Microsoft Visual Studio Bibliotheken

Dieser Anhang enthält zusätzliche Daten und Abbildungen in Bezug auf die Bibliotheken aus Visual Studio ab Version 2010.

1. Klassendiagramm der Messageblocks aus der AAL

Detailansicht der wichtigsten Klassen aus der AAL. Die Daten stammen aus MSDN, z. B. von hier [26].

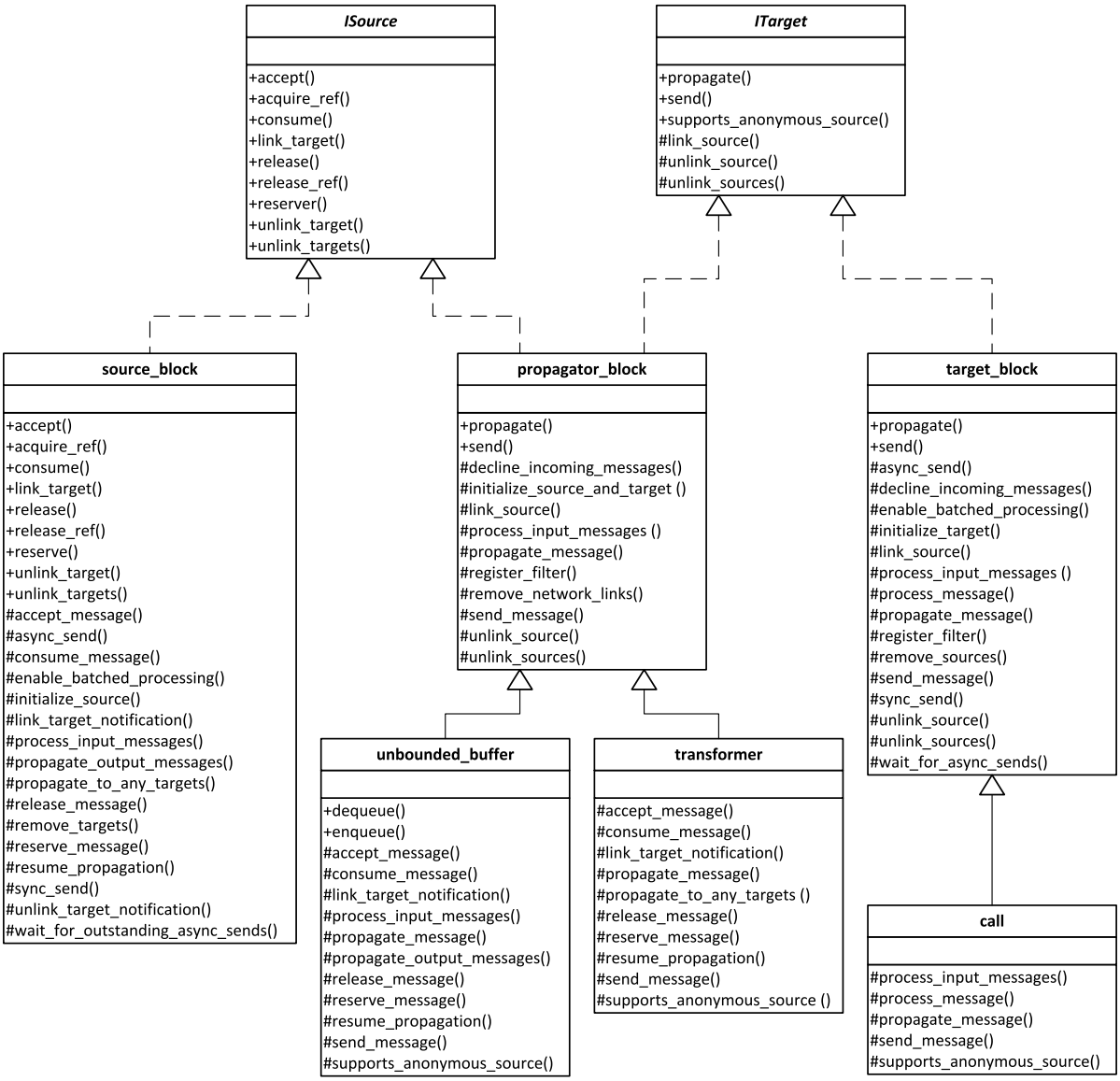


Abbildung C.1: Klassendiagramm der Messageblocks inkl. aller `public` und `protected` Methoden.

2. Bibliotheks-Übersicht Concurrency Runtime

Die Abbildung C.2 erklärt die Zusammenarbeit der einzelnen nebenläufigen Bibliotheken aus Microsoft Visual Studio.

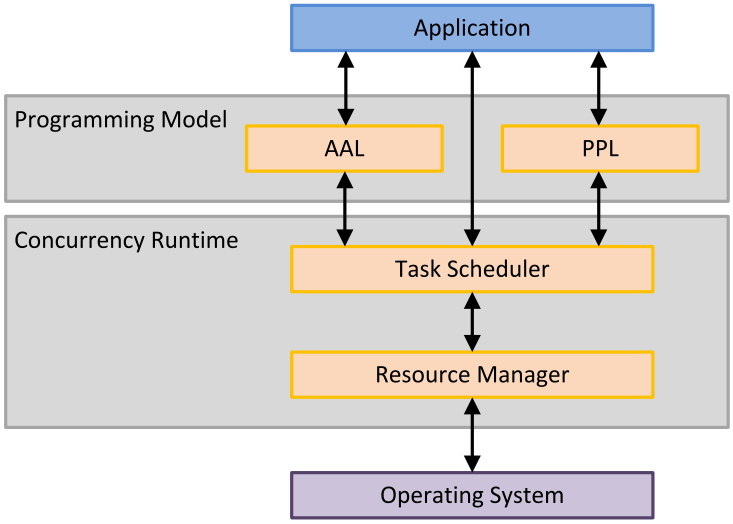


Abbildung C.2: Die AAL und PPL bauen beide auf der Concurrency Runtime [9] auf, welche die Verwaltung der Systemressourcen übernimmt.

3. Busy-Waiting in C++ AMP

Um die Synchronisierung zwischen CPU und GPU bei der Verwendung von C++ AMP [13] besser zu verstehen, wurde im MSDN Forum [27] nachgefragt. Dabei ging es um das Problem, dass zeitlich abgestimmte Ausführungen unter Volllast des Systems auf der GPU viel zu lange brauchen, um zu enden.

```
Lang Christian, 7.11.13

Hi, I am Master Student at the FHNW in Windisch (Switzerland) and am using C++ AMP for
the parallelization on the GPU. Right now I am developing a heterogeneous image pipeline
which uses all CPU Cores and GPU engines simultaniously. To do this, I depend on that all
tasks of the pipeline can be executedasynchronously.

My Question is, how to correctly wait for the completion of the execution of a
parallel_for_each which runs on the GPU? The goal is, that the waiting thread sleeps
after the call of the wait method and wakes up until the execution has finished.

My first approach was to use the accelerator_view and call the accelerator_view::wait()
method. This waits until the end of the execution on GPU, but seems to do this with busy
waiting. (Abbildung C.3)

After many hours of searching and testing other ways, I found a way to do non busy
waiting: myAccView.create_marker().to_task().wait(); (Abbildung C.4)

Attached are two screenshots of the ConcurrencyVisualizer. Pay attention to the idle time
of the CPU in the second version, against the full utilisation of the CPU in the first
version. Why is this?
```

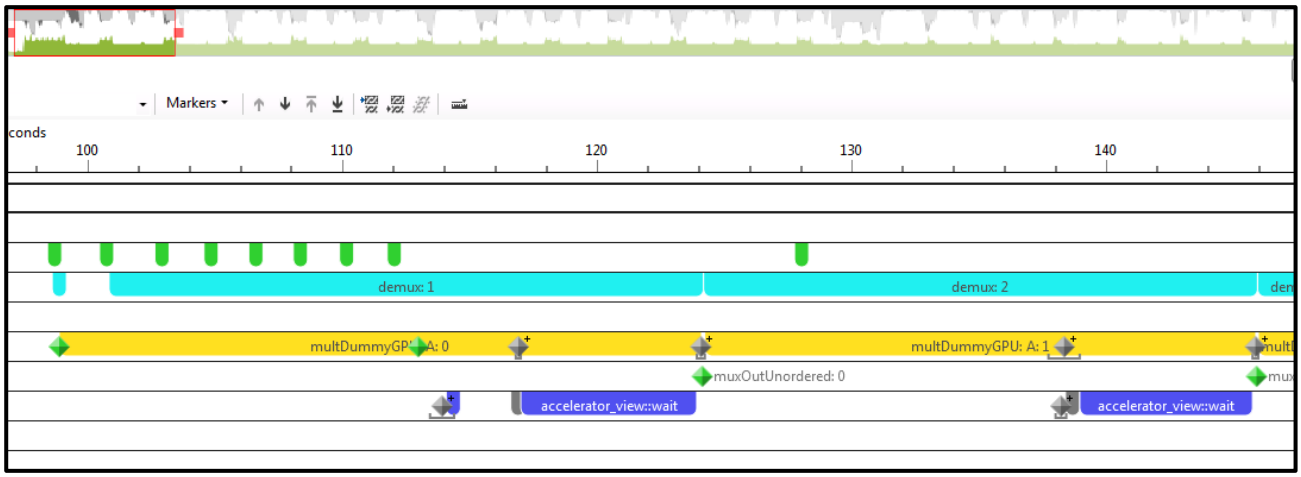


Abbildung C.3: Warten auf die Beendigung des AMP-Auftrages mittels `accelerator_view.wait()`. Das Warten lastet einen CPU-Kern vollständig aus.

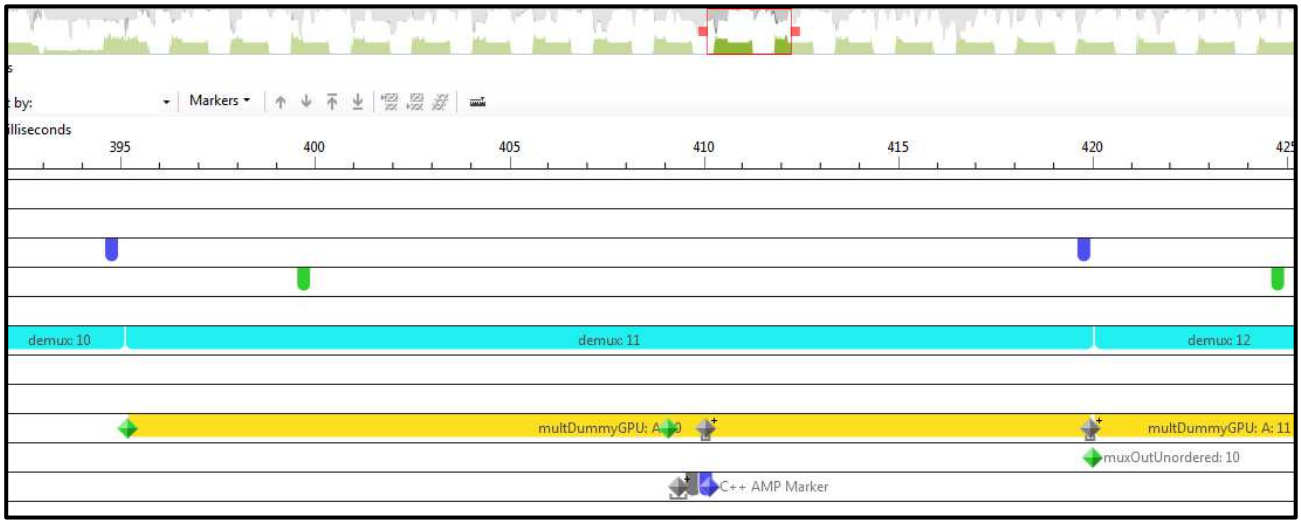


Abbildung C.4: Warten auf die Beendigung des AMP-Auftrages mittels `accelerator_view.create_marker().to_task().wait()`. Das Warten benötigt keine zusätzlichen CPU-Ressourcen.

Hasibur Rahman, (MSFT) Microsoft, 21.11.13

Hi Lang,
The difference you are seeing is because accelerator_view::wait() does busy-waiting where as to_task().wait() does cooperative waiting. The completion_future::to_task() function returns a concurrency::task<void> object and waiting on concurrency::task<> is cooperative waiting. Cooperative waiting allows the CPU to be available for other threads to use.

Also accelerator_view::wait() function starting with Windows 8 (and higher versions) does cooperative waiting and you should see similar behavior for the two cases above on Windows 8 (higher version).

Please feel free to ask if you have any further questions.

Lang Christian, 21.11.13

Hy Rahman
Thank you for your answer. You confirmed my assumptions. My System is a Windows 7.

I have done many testing since the posted question and have seen another weird phenomenon. When my system is fully loaded, the return of the wait() method (the one of the completion_future) takes much longer, then the execution of the task on the GPU. In my case i do some matrix multiplication (64x64) on the GPU and want to repeat that until I reach 50 milliseconds. I do the same thing on CPU until I reach 100ms. A test with only one CPU core loaded is shown in the graph (Abbildung C.5).

It is one task on the CPU shown, that requires approximately 100ms. The other task runs on GPU and takes approx. 50ms. You can see that the GPU task repeats the matrix multiplication several times, where the load of the GPU is shown too.

In the second graph the GPU task needs as much time as the CPU task but repeats the matrix multiplication only two times. The execution on GPU does not need more time than in the first example. (Abbildung C.6)

Is this behavior a consequence of the fact, that the system is to loaded to communicate with the GPU? Or how can I improve that?

Below you can see the relevant code of the GPU task (I know that GetTickCount() is not that precise, but it should not be the problem):

```
const int start = static_cast<int>(GetTickCount());
do {
      mSeries->write_flag(_TwithInt("GPU starting: ", input));
      parallel_for_each(av_c.extent, [&](index<2> idx) restrict(amp) {
            // do mat mult
      });
      mSeries->write_flag(_TwithInt("GPU started: ", input));
      mGpuAccView.create_marker().to_task().wait();
      mSeries->write_flag(_TwithInt("GPU finished: ", input));
} while (static_cast<int>(GetTickCount()) - start < waitTimeMs);
```

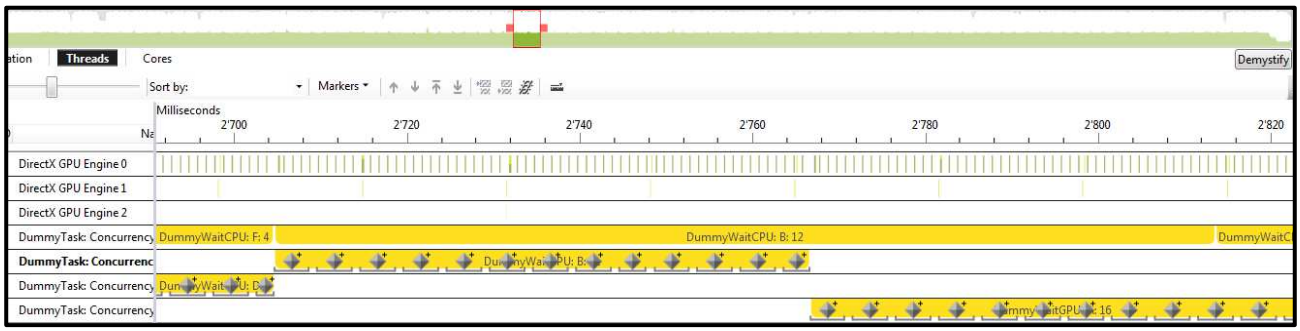


Abbildung C.5: Regelmässige GPU-Berechnungen um die GPU auszulasten. System ist nicht ausgelastet, nur ein CPU-Kern wird für Berechnungen verwendet.

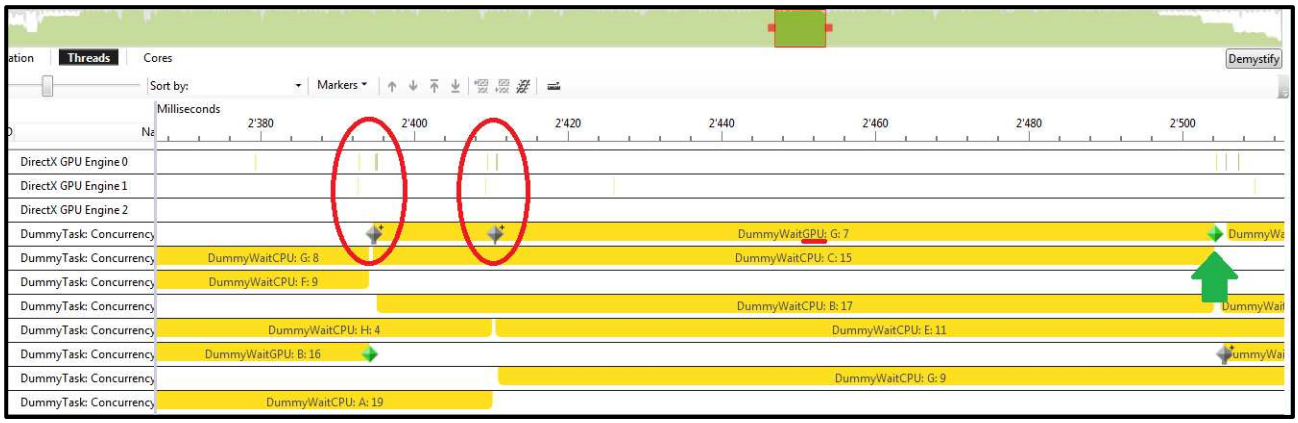


Abbildung C.6: System ist ausgelastet. Alle vier CPU-Kerne führen Berechnungen durch. Zugleich wird GPU mittels denselben Berechnungen ausgelastet. Die Ausführung und Beendigung dieser GPU-Ausführungen dauern enorm lange und stimmen nicht mit GPU-Auslastung überein.

Lukasz Mendakiewicz, Microsoft (MSFT), 30.11.13

Hi Lang,
Much of the below is guesswork given the limited amount of information I have about your software and hardware stack, so please correct me if I'm wrong.

I believe your CPU has 4 logical cores, thus capable of running 4 threads simultaneously. In the second graph in the above post, there are 5 threads running, 4 of which are labelled "CPU" and in my understanding are saturating all the cores with computation, making the 5th thread - the GPU work submitter - preempted (waiting) for an extended amount of time. You can verify that by unhiding the thread activity lanes in Concurrency Visualizer - green segments are execution and mean active work being performed in the given thread, while yellow segments are preemption designating waiting.

Since your GPU task is fully synchronizing CPU and GPU execution with the "wait" call, it causes a starvation on the GPU.

There are couple of approaches you can try to improve the situation, depending on the architecture of the rest of your application:

- submitting larger work items to GPU would lower the relative overhead caused by the synchronization and the submission cost itself
- synchronizing CPU and GPU on a lower frequency (e.g. every couple of p_f_e executions, instead after every one)
- lowering the load on the other CPU threads in order to give the GPU submitter thread chance to run

Another important thing to note is that your work submission frequency is on the fringe of Windows scheduler granularity. The default thread quantum in client version of Windows 7 is approximately 30 ms, and is much longer in server editions. This may introduce some noise to your experiments.

D. Pipeline-Framework

Dieser Anhang umfasst eine Übersicht über alle Elemente des Pipeline-Frameworks und ein simples Anwendungsbeispiel.

1. Building-Blocks

Bezeichnung / Zweck	Implementierung	Beschreibung
Stufe	`Stage`	Einzelne Stufe der Pipeline die mehrere Task-Implementationen enthalten kann. Kann mehrere Pakete sortiert für einen Multi-Paket-Task zur Verfügung stellen. Wichtige Methoden: create, addStageExecutor, setExecutorPriority, link_target
Task-Implementierung	`function`-Objekt	Eine Implementierung eines einzelnen Tasks. Kann als Lambda, Funktor oder Funktionszeiger instanziert werden. Wird als Kopie der Stufe übergeben. Nimmt einen `vector` vom Typ der Datenpakete als Parameter entgegen und gibt ein Paket zurück.
Pipelinepaket	Ableiten von `BasePackage`	Die Klasse für die Pipeline-Pakete. Beinhaltet die Daten für die einzelnen Stufen und die grundlegenden Funktionen für die Pipeline. Daten sollen auf dem Heap liegen und per Smartpointer angesprochen werden. Kann direkt zu einem `Integer` gecastet werden um die Index-Nummer zu erhalten.
Quelle	`Source`	Stellt die Wurzel der Pipeline dar. Ist neben dem `SinkConnector` das einzige Objekt, welches vom Hauptprogramm referenziert wird. Die Zerstörung der Quelle impliziert das Abräumen aller Stufen der Pipeline und der Senke. Überwacht die Anzahl Pakete in der Pipeline (Pipelineslots). Wichtige Methoden: create, start, stop, isFinalized, waitUntilFinalized, link_target
Senke	`Sink`	Stellt das Ende der Pipeline dar. Empfängt alle verarbeiteten Pakete und stellt sie über den `FinalBuffer` zur Verfügung. Hilft bei der Kalibrierung der Pipeline. Wichtige Methoden: create, getFinalBuffer
Rechengeräte-Verwaltung	`DeviceScheduler`	Verwaltet die Recheneinheiten eines einzelnen Rechengeräts. Initiiert die Ausführung eines Jobs beim `Executor` über die `IJobExecutor`-Schnittstelle der Stufe. Wichtige Methoden: scheduleJob, releaseSlot

Tabelle D.1: Alle benötigten Elemente einer Pipeline. Diese sind ebenfalls in Abbildung 4.2 grafisch ersichtlich.

2. Anwendungsbeispiel

In Listing D.6.1 wird eine Pipeline mit zwei Stufen aufgebaut und gestartet. Die Pipeline verarbeitet 20 Pakete und wird nach der Ausführung abgeräumt. Dieser Code befindet sich in der Datei „FrameworkTest.cpp“.

```
static const int dataSize = 10;
static const int packNum = 20;

class MyPackage : public BasePackage {
public:
	shared_ptr<accelerator_view> cpuView;
	shared_ptr<accelerator_view> gpuView;
	shared_ptr<vector<float>> cpuData;
	shared_ptr<array<float, 1>> gpuData;
	shared_ptr<vector<float>> cpuResult;

	void prepareData(bool forGpu) {
		if (!cpuData && !gpuData) throw exception("no data available");
		if (!cpuView || !gpuView) throw exception("not all accelerator_views set");

		if (forGpu) {
			if (!gpuData) {
				gpuData = make_shared<array<float, 1>>(dataSize);
				array<float, 1> stagingArray(gpuData->extent, *cpuView, *gpuView);
				copy_async(cpuData->begin(), cpuData->end(), stagingArray).wait();
				copy_async(stagingArray, *gpuData).wait();
			}
		}
		else {
			if (!cpuData) {
				cpuData = make_shared<vector<float>>(dataSize);
				array<float, 1> stagingArray(gpuData->extent, *cpuView, *gpuView);
				copy_async(*gpuData, stagingArray).wait();
				copy_async(stagingArray, cpuData->begin()).wait();
			}
		}
	}
};


void createDataForCpu(MyPackage& in) {
	in.cpuView = make_shared<accelerator_view>(
		accelerator(accelerator::cpu_accelerator).default_view);
	in.gpuView = make_shared<accelerator_view>(
		accelerator(accelerator::default_accelerator).default_view);

	in.cpuData = make_shared<vector<float>>();
	for (int i = 0; i < dataSize; ++i) {
		in.cpuData->push_back(static_cast<float>(i) * 3.14f);
	}
}


void doTaskOnCpu(MyPackage& in) {
	if (!in.cpuData) throw exception("No CPU data");

	for (float& data : *in.cpuData) {
		data *= 7;
	}

	in.gpuData.reset();
}
```

```
void doTaskOnGpu(MyPackage& in) {
	if (!in.gpuData) throw exception("No GPU data");

	array<float, 1>& data = *in.gpuData;
	parallel_for_each(data.extent, [&](index<1> idx) restrict(amp) {
		data[idx] *= 7;
	});
	in.gpuView->create_marker().to_task().wait();

	in.cpuData.reset();
}


void doMultiPackTaskOnCpu(MyPackage& a, MyPackage& b) {
	if (!a.cpuData || !b.cpuData) throw exception("No CPU data");

	a.cpuResult = make_shared<vector<float>>(dataSize);

	for (int i = 0; i < a.cpuData->size(); ++i) {
		a.cpuResult->at(i) = a.cpuData->at(i) + b.cpuData->at(i);
	}
}


void displayData(MyPackage& in) {
	for (float& data : *in.cpuResult) {
		cout << data << ", ";
	}
	cout << endl;
}


void doFrameworkTest() {
	cout << "start framework tests" << endl;

	// create pipeline management
	auto cpuScheduler = DeviceScheduler::create(3, "CPU");
	auto gpuScheduler = DeviceScheduler::create(1, "GPU");

	// create pipeline input and output
	auto source = Source<MyPackage>::create(8);
	auto sink = Sink<MyPackage>::create(source);
	auto buffer = sink->getFinalBuffer();

	// create tasks
	auto createTask = [](vector<MyPackage> in) -> MyPackage {
		if (in[0] >= packNum) {
			in[0].type = BasePackage::Type::STOP;
		}
		else {
			createDataForCpu(in[0]);
		}
		return in[0];
	};
	auto cpuTask = [](vector<MyPackage> in) -> MyPackage {
		in[0].prepareData(false);
		doTaskOnCpu(in[0]);
		return in[0];
	};
	auto gpuTask = [](vector<MyPackage> in) -> MyPackage {
		in[0].prepareData(true);
		doTaskOnGpu(in[0]);
		return in[0];
	};
	auto multiPackTask = [](vector<MyPackage> in) -> MyPackage {
		in[0].prepareData(false);
		in[1].prepareData(false);
		doMultiPackTaskOnCpu(in[0], in[1]);
		return in[0];
```

```
	};

	// create stages
	auto createStage = Stage<MyPackage, MyPackage>::create(
		"Create", 1, cpuScheduler, createTask);
	auto multStage = Stage<MyPackage, MyPackage>::create(
		"Multiplication", 1, cpuScheduler, cpuTask);
	multStage->addExecutor(gpuScheduler, gpuTask);
	auto multiPackStage = Stage<MyPackage, MyPackage>::create(
		"MultiPack", 2, cpuScheduler, multiPackTask);

	// set priority index of multiplication stage
	vector<pair<shared_ptr<DeviceScheduler>, int>> prioList;
	prioList.push_back(make_pair(gpuScheduler, 50));
	prioList.push_back(make_pair(cpuScheduler, 10));
	multStage->setExecutorPriority(prioList);

	// link pipeline stages together
	multiPackStage->link_target(move(sink));
	multStage->link_target(move(multiPackStage));
	createStage->link_target(move(multStage));
	source->link_target(move(createStage));

	// start unbounded pipeline execution
	int offset = source->start(0);

	// receive processed data until pipeline execution has finished
	MyPackage out;
	do {
		out = Concurrency::receive(*buffer);
		if (out.type == BasePackage::Type::DATA) {
			displayData(out);
		}
	} while (out.type != BasePackage::Type::END);

	cout << "finished framework tests" << endl << endl;
}
```

Listing D.6.1: Komplette Beispielanwendung des Pipeline-Frameworks.

E. Paper Draft

Im Rahmen der Master-Thesis wurde folgender Entwurf eines wissenschaftlichen Papers erstellt.

Parallel Computer Vision: Heterogeneous Video Pipeline Framework

Lang Christian
Institute of Mobile and Distributed Systems
University of Applied Sciences Northwestern Switzerland
Bahnhofstrasse 6, Sektor 5.2B
5210 Windisch, Switzerland
christian.lang1@students.fhnw.ch

Draft
v0.7

Abstract

In the context of live video processing the main problem is to accomplish a sufficient frame rate. One way to achieve this is to use all available computational resources of a typical personal computer. But the scheduling of the dependent jobs in such a heterogeneous video pipeline is not a trivial task. In this paper we develop a framework that provides a simple way to create a heterogeneous pipeline and helps with the management of the available resources.

To understand the bottlenecks in a streaming application we analyze the timing characteristics of the pipeline pattern and show how to improve the exit rate with pipeline and task parallelism. We further extend the pipeline pattern with the concept of a stage executor which abstracts the task implementations on different compute devices. We propose a simple scheduling strategy to manage the available computational resources that can be used by the developed pipeline framework.

The feasibility of our system is proven by two types of simulations which are analyzed on their potential performance measured in relative overall execution time, output variance, data movement and initial latency.

Keywords

live video processing, heterogeneous system architecture, online scheduling, coarse grained parallelism, actor model, message passing, data flow, lower bound, pipeline parallelism

1 Introduction

Detecting and recognizing of known people in a high resolution live video stream is a very expensive task that cannot be done by a sequential program on a typical *personal computer* (PC). Today, modern PCs do usually contain several multicore processors of different types that can process many processes or threads in parallel. Examples are multicore CPUs or GPUs which all should be used to support an expensive task like live video processing in such *heterogeneous system architecture* (HSA).

Consider for example a live video observation system that not only records the overlooked scene but additionally detects and recognizes people and alerts the security personnel if an ominous person has entered the area. This task consists of several dependent subtasks that have to be processed for each frame in the live stream. The critical requirement that the system has to meet is the frame output frequency that should be as high as the input frequency.

The *pipeline pattern* [1, 2] is well suited to map the structure of subsequent subtasks of such an application. It allows to design the implicit dependencies between subtasks of the same frame processing and to use the coarse-grained implicit *functional pipeline parallelism* provided by the pattern. This is caused by the independence of different frames that allows a MIMD[1] system to concurrently process different subtasks of multiple frames. This approach is often used in instruction, graphics or audio processing where plenty of data need to be processed and a coarse-grained parallelism should be used. No fine-grained parallelism can be achieved because not all subtasks can be simply converted into a concurrent version.

In a live processing system, where not the whole input data is available at the beginning, an online scheduling of jobs, generated by subtasks of the pipeline, is needed. The scheduling of a heterogeneous system has to meet the requirements of (i) minimization the mean frame latency and (ii) guarantee that the exit frame rate does not fall behind the arrival frame rate. These requirements should be achieved by the usage of all available resources and the distribution of the jobs to the best suited computational resource.

In Section 2 we describe a generic pipeline structure and analyze the timing characteristics of such a pipeline. This shows the possible performance improvements by the functional pipelining parallelism and an additional task parallelism. Section 3 defines the requirements of a scheduling that is needed to take advantage of the mentioned improvements in a heterogeneous multicore system. In Section 4 we define the structure of a heterogeneous pipeline and develop on this basis a scheduling strategy. Therefore we show the parts of our system and their responsibilities. In Section 5 we define several theoretical bounds to estimate the performance of our system, describe two types of simulations and show the measured benchmarks.

1.1 Related Work

The analysis of the pipeline parallelism by Navarro et al. [3] that extends the paper of Liao et al. [4] discusses the performance optimization by assigning multiple computational resources to a single subtask of a pipeline. This is shown by comparing two different implementations of this approach and the analytical model based on the queueing theory. The description of a parallel pipelining model is delivered by [4]. These analysis and concepts are used to categorize our use case and are enhanced by our own analysis in Section 2.

The Asynchronous Agents Library (AAL) [5, 6] implements the actor model which cope with the problem of shared memory and race conditions in concurrent systems. The library names his actors *agents* and uses the *message passing paradigm* to enable communication between agents. The management of the lightweight task, that are used by the agents to process their work, is done by the Microsoft Concurrency Runtime for C++ [7]. The actor model is well suited to implement a pipeline with functional parallelism and is used to build our simulation system used in Section 5.

A survey of online scheduling is given by Lee et al. [8]. It defines precisely all types of job scheduling in parallel machine environments with machine eligibility constraints, delivers lower bounds for the *makespan minimization problem* and describes two basic online scheduling paradigms. The minimization of a makespan, the time a system needs to complete some defined tasks, is also known as *job shop problem* and is a generalization of the *traveling salesman problem*. This and the next paper give an overview of scheduling problems in the context of HSAs.

The emerging of HSAs opens new ways of processing but also generates new complex problems. Bower et al. [9] give an overview of the structure of such systems and justifies their existence. The paper does that in the context of job scheduling on a modern operating system.

[1] System that can process multiple instructions on multiple data concurrently. Definition by Michael J. Flynn (1966).

The three different HSA implementation frameworks OpenCL, OpenACC and C++AMP are reviewed by us in [10]. We give a short introduction in each technology, show common problems and perform several tests to compare the different implementations. To implement the HSA tasks of the simulation we use C++ AMP.

2 Pipeline Pattern

The pipeline pattern and the *data flow pattern* can be used if a stream of data has to be processed. In this Section we define the structure of a pipeline and analyze the timing characteristics. The structure of a generic pipeline as Buschmann et al. describes it, contains one or several processing elements that are connected to its neighbors to build a pipeline (Figure 1). These elements are called *pipeline stage*. The stage is responsible for the execution of a subtask, or simply one *task*, of the pipeline. All these tasks enrich, refine or transform the data that is floating through the pipeline, where the data is held by the *data package*. A pipeline always needs a data source at the head which is responsible for the generation of data packages that are consecutively sent through each stage and consumed by the sink at the end of the pipeline.

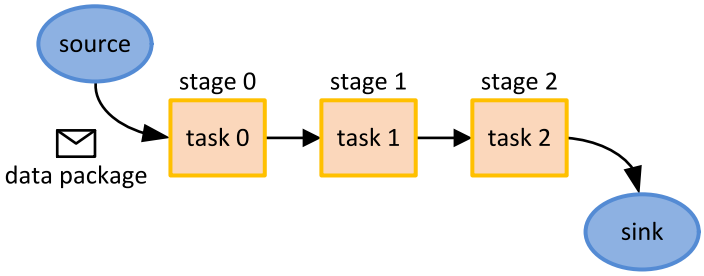


Figure 1: Structure of a generic pipeline. It contains a source as an input and a sink as an output of the data packages. The subtasks of the overall problem are processed at the individual stages.

The pipes that connect the source, all the stages and the sink are responsible for saving the received data packages and forwarding them to the next processing element of the pipeline. Through this *queueing* the connected elements are synchronized. As Vermeulen et al. mentioned there are different flow types possible in a pipeline structure. In this paper we concentrate on a push-pull hybrid, in which a processing element actively sends a package to the passive queue and the next element actively requests the package from the queue.

2.1 Timing Characteristics

In order to be able to improve the performance of a pipelined system we have to understand how parallelization can be done and which impact it has on the timing characteristics of such a system.

A system itself is defined by the number of available compute units n and the number of compute devices m that hold individual numbers of the n compute units. A typical compute unit is one core of a CPU or a GPU engine, whereas a compute device is a whole CPU or GPU. Therefore, d_j $(0 \leq j < m)$ is the number of compute units of the j-th device. The number of pipeline stages is denoted as p and k defines the number of data packages to process.

The interesting properties of a pipeline are (i) the latency l, which describes the timespan between the arrival of a single package until it exits the system and (ii) the exit interval x, which is the timespan between the output of two consecutive data packages. We denote the mean latency with an uppercase L and individual latencies with a lowercase l_i including the package index, e. g. the latency of the first package (initial latency) is denoted as l_0. The same is done with the exit interval, where $x_{0,1}$ denotes the timespan between the output of the first and the second package. The minimal arrival timespan that the data source can achieve is denoted as t. Additional benchmarks are the output variance σ^2 (Equation 1) and the overall execution time R of the system.

$$\sigma^2 = \frac{1}{k-1}\sum_{i=0}^{k-2}(x_{i,i+1} - \mu)^2, \qquad \mu = \frac{1}{k}\sum_{i=0}^{k-2} x_{i,i+1}$$

Equation 1: Calculation of the output variance σ^2 that describes the system's ability to produce a regular output stream. k is the number of data packages and x the exit interval between consecutive packages. μ is the expected value.

In Figure 2 we show an example with stage execution times out of $S = \{s_0, s_1, s_2\}$ where $p = |S| = 3$ and n compute units $C = \{c_0, c_1, \dots, c_{n-1}\}$. The exit interval of all packages in a primitive sequential implementation with $n = 1$ would be $x_{0,1} = l_0 = \sum_i s_i$. Therefore, all latencies and exit intervals are equal to their mean values $x_{i,i+1} = X = L$. If we take advantage of the pipeline parallelism, we can improve the exit interval with a system that has $n = p$ compute units to a value of $x_{i,i+1} = \max(S)$. Because of dependencies between each stage, no improvement factor of n can be achieved. The interval can only be reduced to the time the slowest stage, in this case s_1, needs. In addition each package has individual latency values. In the example the exit rate is slower than the arrival rate because the limiting stage s_1 is longer than the arrival interval t. Therefore, $l_0 < l_1 < l_2$ is true and the latency will increase to infinity if the input is not limited at some point.

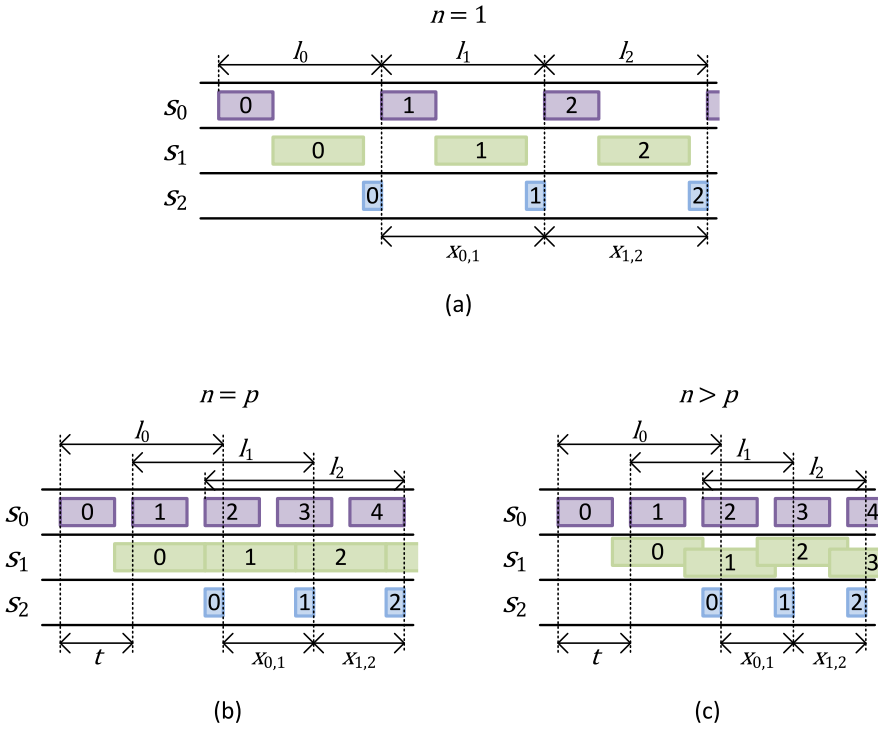


Figure 2: The exit interval x can be improved through the utilization of pipeline parallelism. (a) Sequential execution of a pipeline. (b) Each stage can only process one data package at the time, but all stages can be concurrently executed. (c) Each stage can use multiple computational resources to process multiple data packages. In this example it is assumed that the system has four compute units. The stage s_0 cannot be faster because of t that depends on the arrival rate.

To overcome the handicap of limiting stages, a type of task parallelism can be used in systems with $n > p$ compute units. Therefore, the stage's tasks have to be able of parallel processing of multiple data packages. In this case they can use the available computational resources to execute the task of a slow stage multiple times and are so able to reduce the limiting effect of the slowest stage. If our example has now $n = 4$ available compute units, the limiting stage s_1 can process multiple data packages in parallel and the exit interval x can be reduced until reaching the arrival interval t. That is defined by the arrival rate which depends on the data source. In our example this interval is reached and cannot be further improved if the arrival rate cannot be speeded up. Therefore, $l_0 = l_1 = l_2$ and $t = x_{0,1} = x_{1,2}$ are true. Of course the implicit latency l between of each data package cannot be made smaller than l_0 by any of these coarse-grained methods.

The observations we made show that with the pipeline and task parallelism we are able to enhance the pipeline paradigm in such a way that we can parallelize individual stages and overcome the limiting effect of slow stages. The only restrictions that persist are the arrival rate and the limited number of available compute units. The scheduling of a limited amount of computational resources is discussed in Section 3 and depending on that, further performance potential is discussed in Section 5.

3 Heterogeneous Pipeline Scheduling

In Section 2 we showed a suitable way to design a live stream processing system that can benefit from all computational resources a PC may be offering. To manage all these resources and offer them to the pipeline system an online

scheduler is needed. In this Section we analyze the fixed specifications of our system and describe the requirements a scheduler for such a system has to meet. Because a heterogeneous system implies that different implementations of any task can exist we expand the pipeline pattern. In Section 4 we propose a simple approach for a scheduling strategy that is evaluated in Section 5.

3.1 Scheduling Requirements

The specifications of our system can be described by the types of Lee et al. Because no information about the jobs is available before the start of the processing, we need an (i) online scheduling strategy. The online scheduling itself is (ii) non-clairvoyant because no further information about the job becomes available after its arrival. Only after the task execution the needed effort (the execution time) becomes visible. The fact that the different compute devices have different speed but for themselves need the same amount of time for the same task makes them to (iii) uniform machines. In addition only some of the compute devices can execute a defined task and therefore our scheduling depends on (iv) machine eligibility constraints.

The requirements of the system described in the introduction conduct several decisions for the design of the scheduler. We chose to focus on the following: (i) Non-preemptive scheduling, to support the real time and live stream requirements and simplify the delivery of task implementations by the user; (ii) Throughput, which is implicit connected to fastest execution and shortest delay; (iii) Fairness, which prevents starvation and supports execution on the best matching computational resource. By the best matching computational resource we mean the fastest compute device that has a free compute unit to process another data package.

3.2 Heterogeneous Pipeline Stage

In the context of HSA an additional paradigm is needed to map distinct task implementations to their compute device and to enable the benefit that comes with a heterogeneous system. For that purpose the pipeline paradigm is extended by the concept of a *task executor*. An executor out of the set E is the explicit implementation of a task for a defined compute device out of the set D. In Figure 3 the executor $e_{0,1} \in E$ is an implementation of the task of stage $s_0 \in S$ which will use compute units of the compute device $d_1 \in D$.

These executors create several possible paths through the pipeline. The example pipeline contains $m^p = 4$ possible paths which do not all have the same overall execution time. Therefore, it is important that the scheduler chooses a path, which minimizes the execution time. The scheduler strategy that is used to make this optimal choice is described in Section 4. Since the different compute devices do not have to share the same memory, there could be caused overhead by the data movement between devices.

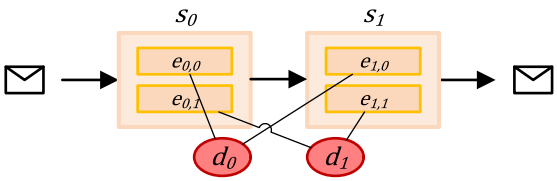


Figure 3: Each stage contains at least one executor. Typically the number of such executors is equal to the number of different compute devices.

The introduction of a heterogeneous pipeline adds two new properties to the pipeline. A pipeline is (i) complete if it has task implementations for each device on each stage. A pipeline is (ii) uniformly accelerated if the speed hierarchy of the devices is the same in each stage. A typical pipeline with a CPU and a GPU will be incomplete but uniformly accelerated, because not each task will be feasible or useful to implement on GPU, but these tasks that are implemented will be faster on the GPU than on the CPU. The most general case is an incomplete and not uniformly accelerated pipeline.

4 Developing of our Scheduling Strategy

A high speed processing system relies on minimum reaction time between an event and the subsequent action. In our case this event is a finished task which releases a compute unit that can be assigned to another waiting task. Based on the chosen cooperative scheduling we design the pipeline on the aspect of the message passing paradigm and therefore view the scheduling process from the perspective of a stage that has data to process. To show all building blocks of our system an example is shown in Figure 4.

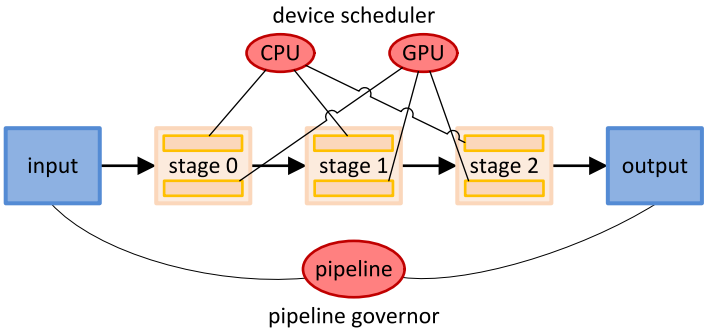


Figure 4: An example of a system with three stages and two different compute devices. Both devices (CPU, GPU) are managed by individual device schedulers. The number of data packages in the whole pipeline is managed by a pipeline governor.

After the receiving of a message the stage wants to process the corresponding data on one of its executors. To decide which executor's compute device has usable resources we introduce the *device scheduler*. The device scheduler is responsible for managing all compute units of a single device. He is the one that decides which requesting executor can use his compute units for task processing.

The whole scheduling protocol consists of three steps: (i) A stage receives a message and registers its job at each known device scheduler. This depends on for which devices a task implementation is available. If only a CPU executor exists for a task, the stage only knows the corresponding CPU device scheduler. After this step the stage can receive the next data package. The user is able to prioritize the executors of a stage. (ii) A job consists of a priority index, a flag which indicates if the data is previously processed on the scheduled device and a data package index is added to a priority queue of the scheduler. (iii) If the scheduler has free compute units or one of the used ones becomes free, the scheduler takes the job with the highest priority and tries to execute it. Because the job can be added to multiple scheduler queues, the risk is high, that its execution has already been started by another compute device. In this case the next job in the queue is executed. The scheduler waits for further jobs if the queue is empty and compute units are free.

The prioritization of the job queue is used to optimize the path a data package takes through the pipeline. The following list defines the order of the prioritization constraints.

1. Preferred device of this stage: A job is executed first if the evaluating device is the preferred one of this stage. This constraint ensures that always the fastest available device is used to process a task. In a uniformly accelerated pipeline this is always the same device.
2. Device of previous stage: The second point enforces data locality by preferring the same device that has processed the data in the preceding stage and thus minimize overhead through data movement.
3. Lowest package index: The third constraint boosts the observing of the package sequence and at the same time provides equal distributed pipeline outputs.

With the described protocol and prioritization we ensure that the best suited available device is always chosen. In addition we prevent the starvation of poorly rated jobs through a *pipeline governor* (see Figure 4) that limits the number of data packages inside the pipeline. The limit of this governor should be individually adjusted to the used pipeline. A small limit ensures a small latency between the first input and output; on the other hand a high limit ensures that all available compute units can be fully loaded.

4.1 Variation of Scheduling Strategies

Along with the strategy described above, which is named "Prioritized", four others are developed. By comparing these variants we gain general knowledge of the scheduling problem and can optimize the proposed strategy.

The first implementation variant is called "Simple" which does not use a queue, but governors for each compute device, that grant free compute units to the first requesting pipeline stage. Because this implementation cannot process multiple data packages per stage, "Multi" uses the same strategy as "Simple" but can request multiple compute units of the same device per stage. The implementation "Conservative" also uses governors for the management of the compute units, but considers that the first requesting stage is not always best executor for the inquired device. Therefore, an offered compute unit is only used at the first time if it is the preferred/previous device. Otherwise it refuses the compute unit and waits for another. The variant "Unprioritized" does not use the stage prioritization information of the "Prioritized" implementation and therefore always priorities the previous executor.

5 Evaluation

The performance evaluation of an online scheduler is not a simple task because already the scheduling optimization of well-known jobs is in the class of NP-hard problems. In the context of HSA we have several different implementations of a task and therefore different execution times for one problem. Furthermore, we have implicit pipeline dependencies which have to be guaranteed.

In this Section we estimate lower and upper bounds of a heterogeneous pipeline with dependent and independent tasks. Further, we perform two types of simulations to evaluate different implementations of our system and compare these with the theoretical bounds to gain performance data.

5.1 Bounds

As already mentioned the problem of online scheduling cannot be solved in polynomial time. Therefore, we develop three heuristic approaches to evaluate a possible lower bound and two approaches to calculate an upper bound of our system. The objective function of the lower bound is the minimization of the makespan and of the upper bound the maximization of the makespan. All these estimations ignore the possible overhead of data movement between different compute devices. These are unimportant in the upper bound estimation, because the upper bound is typically the execution on a single device and hence no data movement is happening. In the case of the lower bound the disregarding of data movement ensures that no unneeded data movement distorts the value.

The first approach α_1 is to compute the execution time of a data package processing if this package could be split and each part processed on different compute units. This can be achieved by calculating the speed v_i of each stage i and then sum up the execution time of one package per stage to the overall execution time of the pipeline. This value is then multiplied by k, the number of data packages the system has to process (Equation 2). As already defined p is the number of stages and $e_{i,j}$ is the time the executor of stage i on the compute device j needs. d_j is equal to the number of data packages the j-th device can process in parallel.

$$\alpha_1 = k \cdot \sum_{i=0}^{p-1} \frac{1}{v_i}, \qquad v_i = \sum_{j=0}^{m-1} \frac{d_j}{e_{i,j}}$$

Equation 2: Calculation of the lower bound by estimate the mean execution time of one package. This calculation supposes the impossible fact that a single data package can be processed at the same time by several compute units. However, it does not violate the stage dependencies.

The next two approaches use a kind of optimization algorithm. The function $t(q_x)$ returns the time that the queue of the compute unit x needs for the execution of all scheduled tasks and $t(x, y)$ is the time a job y needs to execute on the compute unit x.

The first optimization algorithm α_2 tries to evenly distribute the jobs over the job queues q of all available compute units (Listing 1). This is done by evaluating the least loaded queue after adding the pending job. If two queues results in the same load after the scheduling of the job, the queue with the faster compute device is chosen. The lower bound is the execution time of the most loaded queue. This approach does conform to the stage dependencies.

Input: The sorted list of $Jobs$ to process. The different compute units CU that process the jobs.
Output: The time the system needs to finish all jobs.

1. **for** (job $j : Jobs$)
2. $\quad c_{tmp} \leftarrow \underset{x \in CU}{argmin} \left(t(q_x) + t(x, j)\right)$
3. $\quad c \leftarrow \underset{x \in c_{tmp}}{argmin} \left(t(x, j)\right)$
4. $\quad$ add job j to job queue q_c
5. $c \leftarrow \underset{x \in CU}{argmin} \left(t(q_x)\right)$
6. **return** $t(q_c)$

Listing 1: Pseudo code of the heuristic approach α_2 to evaluate the lower bound. It tries to evenly distribute the jobs over all available compute units.

The second optimization approach β_3 uses backtracking and a tabu list [11] to improve an initial generated solution (Listing 2). The initial solution is generated by scheduling each job on its fastest compute device. If a device contains multiple compute units these are loaded uniformly. After the initialization, the algorithm tries to distribute jobs from the most loaded queue q_m to the least loaded one q_l. It searches for the best matching job for such a move. This depends on

the smallest speed improvement factor (see line 9 in Listing 2), defined by the time the job needs on device l divided through the time the job needs on device m. The algorithm stops if no improving moves are available and the resulting lower bound is the execution time of the most loaded queue. This algorithm violates the constraint of the stage dependencies but is not greedy, compared to algorithm α_2.

Input: The sorted list of $Jobs$ to process. The different compute units CU that process the jobs.
Output: The time the system needs to finish all jobs.

1. **for** (job $j : Jobs$)
2. $c_{tmp} \leftarrow \underset{x \in CU}{argmin} \left(t(x, j)\right)$
3. $c \leftarrow \underset{x \in c_{tmp}}{argmin} \left(t(q_x)\right)$
4. add job j to job queue q_c
5. **while** (true)
6. $m \leftarrow \underset{x \in CU}{argmax} \left(t(q_x)\right)$
7. $l \leftarrow \underset{x \in CU}{argmin} \left(t(q_x)\right)$
8. $t_{old} \leftarrow t(q_m)$
9. $j \leftarrow \underset{y \in (q_m \setminus z)}{argmin} \left(\frac{t(l,y)}{t(m,y)}\right)$
10. remove job j from job queue q_m
11. **if** $(t(q_l) + t(l, j) < t_{old})$
12. **then** add job j to job queue q_l
13. $z \leftarrow \{\}$
14. **else** add job j to tabu list z
15. re-add job j to job queue q_m
16. **if** $z = q_m$
17. **then break**
18. $c \leftarrow \underset{x \in CU}{argmin} \left(t(q_x)\right)$
19. **return** $t(q_c)$

Listing 2: Pseudo code of the heuristic optimization approach α_3 to evaluate the lower bound. z is the tabu list that contains all jobs that were unsuccessful tried to move from the longest to the shortest queue. It is always the job chosen that has the smallest speed improvement factor.

Two approaches β_1 and β_2 are used to evaluate different upper bounds of the problem. β_1 computes the sequential execution time of the slowest device and β_2 of the longest path (chose the slowest device of each stage independently). This has to be distinguished because there may not uniformly accelerated pipelines exist where the slowest path does not use only one device.

Definition 1 shows the bounds that are used at the analysis in the next Section.

$$upper\ bound\ \beta = \max(\beta_1, \beta_2)$$
$$lower\ bound\ \alpha = \min(\alpha_1, \alpha_2, \alpha_3)$$

Definition 1

5.2 Simulations and Results

Comparing the overall execution time with the lower bound α, the measured time is noted as a relative value (Figure 5 (a) and Figure 6 (a)). The relative value is calculated with R/α and denotes the factor the measured system performs worse than the lower bound. A value of 1 means the system reaches the lower bound. A lower value is better.

The test system that runs the described simulations consists of a four core CPU and a GPU with one engine. The simulated pipelines have to process 50 data packages five times in a row to gain average measuring data. To simulate some fluctuations of execution times of a real system, no further effort has to be made because our test system has such variations too.

In order to gain performance data of our framework we defined two types of dummy tasks that can be executed in a pipeline. The static type (i) is some kind of busy waiting that stresses the needed resource until a previously defined time span has reached. With this method we are able to define a pipeline with fixed execution times for each implementation of each stage. The dynamic type (ii) performs a matrix multiplication on the data packet that is received

by the stage. Typically such a task also takes the same amount of time at each execution and therefore an initially measured execution time can be used to estimate the bounds of the simulation. On the other hand, this task implementation can perform differently if the system is under heavy load and therefore better corresponds to a real system.

The simulations are done by building up a pipeline with eight stages and possible CPU and GPU implementations. In the static task simulation the execution time of each implementation on each stage is defined so that the GPU is a factor five faster than the CPU if the simulated pipeline is uniformly accelerated. The not uniformly accelerated test pipelines are defined with different random times and the incomplete pipelines misses at least one task implementation per compute device. If data has to be copied from one device to another, a data movement has to be executed that consumes one-tenth of the time the GPU would need to process the data. Because the matrix multiplication in the dynamic task simulation is always faster on GPU than CPU, there are only uniformly accelerated test pipelines.

To benchmark our system, we compare the scheduling strategy of Section 4 with the described variations in a dynamic task simulation (Figure 5) and a static task simulation (Figure 6) and observe several system behaviors. These are analyzed to improve our understanding of the online scheduling problem.

The static task simulations showed that the developed HSA systems always outperform the homogeneous parallel CPU implementation or the sequential GPU implementation. Its relative overall execution time is typically at 1.8 or lower whereas the upper bound has an average value of 6.5. The described system "Prioritized" (marked in red) achieves most of the time the best performance where not only the overall execution time but also the output variance, the number of data movements and the initial latency were considered. The simulation of a real world pipeline (Figure 5) shows that the "Prioritized" system achieves the best results at the most important metrics (overall time, output variance).

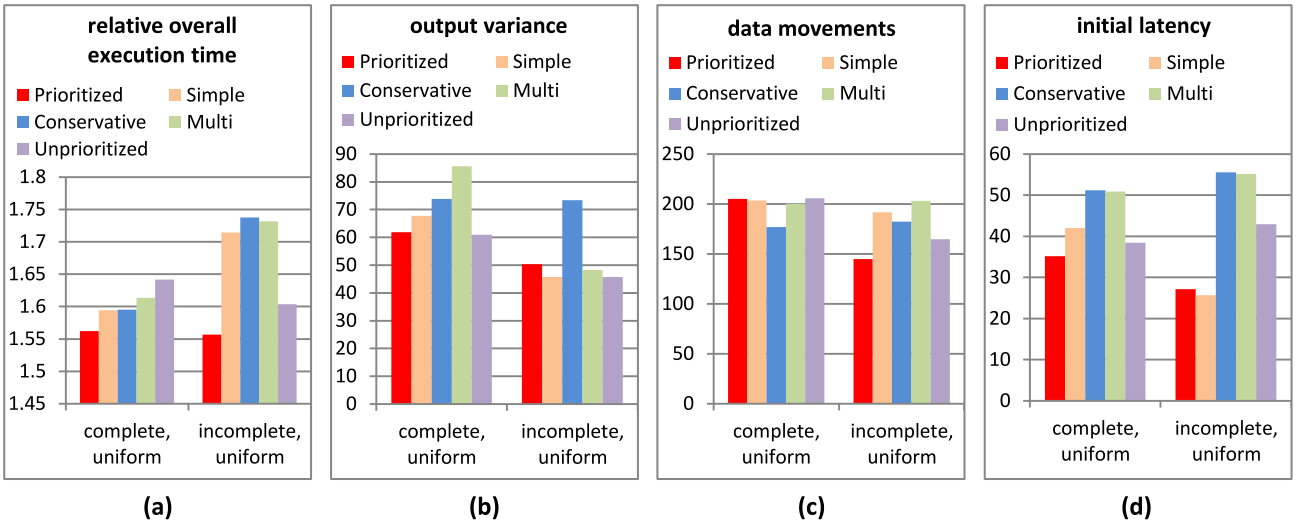


Figure 5: Measuring of a real pipeline of matrix multiplications. Because matrix multiplications are always faster on GPU than on CPU, there are only uniform accelerated pipelines. The systems use three CPU cores and one GPU engine. Short bars are better. (a) This benchmark shows the time factor the system needs, compared to the lower bound. A value of 1 means lower bound. (b) A small value means evenly distributed data output over time. (c) Number of all up- and downloads to and from the GPU. (d) The time l_0 in milliseconds the first data package needs until output.

The analysis of the described scheduling strategy shows that the "Prioritized" implementation outperforms all others and achieves good performance compared to the theoretical lower bound. In addition all measured benchmarks stay stable over all different systems and therefore implicate the practical suitability.

5.3 Conclusion

We analyzed all the implementations from Section 4 with several different pipeline types and made the following findings: (i) It is more practical to use an implementation that can use multiple compute units of the same device per stage, because this prevents that one slow stage limits the whole pipeline. (ii) It is useful to let the user prioritize the different executors at instantiation of the pipeline because this ensures that the system always takes the fastest available compute device. (iii) The described scheduling strategy is more flexible because its execution performance does not depend on the limit of the pipeline governor. (iv) A global queue implementation does know all participants of the system and can distribute the jobs better than independent conservative users that requests computational resources. (v)

Because the management of the devices and the operating system also needs some computational resources, it arises that our system performs best with only $d_{CPU} - 1$ CPU cores used.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, "Pipes and Filters," in *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, Siemens AG, Germany, Wiley, 1996, pp. 53-70.

[2] A. Vermeulen, G. Beged-Dov and P. Thompson, "The Pipeline Design Pattern," Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems, 1995.

[3] A. Navarro, R. Asenjo, S. Tabik and C. Cascaval, "Analytical Modeling of Pipeline Parallelism," Parallel Architectures and Compilation Techniques, Raleigh, NC USA, 2009.

[4] W.-K. Liao, A. Choudhary, D. Weiner and P. Varshney, "Performance Evaluation of a Parallel Pipeline Computational Model for Space-Time Adaptive Processing," The Journal of Supercomputing, 2005.

[5] Microsoft, "Asynchronous Agents," [Online]. Available: http://msdn.microsoft.com/library/vstudio/dd551463. [Accessed 30. October 2013].

[6] Microsoft, "Asynchronous Message Blocks," [Online]. Available: http://msdn.microsoft.com/en-us/library/vstudio/dd504833%28v=vs.100%29.aspx#. [Accessed 31. October 2013].

[7] Microsoft, "Concurrency Runtime," [Online]. Available: http://msdn.microsoft.com/en-us/library/dd504870.aspx. [Accessed 31. December 2013].

[8] K. Lee, J. Y.-T. Leung and M. L. Pinedo, "Makespan minimization in online scheduling with machine eligibility," Springer Science+Business Media, New York, 2013.

[9] F. Bower, D. Sorin and L. Cox, "The impact of dynamically heterogeneous multicore processors on thread scheduling," Micro, IEEE, 2008.

[10] C. Lang, "IP813: Parallel Computer Vision: Heterogeneous System Architecture & Enhanced Head Tracking," FHNW, Windisch, 2013.

[11] F. Glover and M. Laguna, Tabu Search, Boston: KluwerAcademic Publishers, 1997.

Appendix

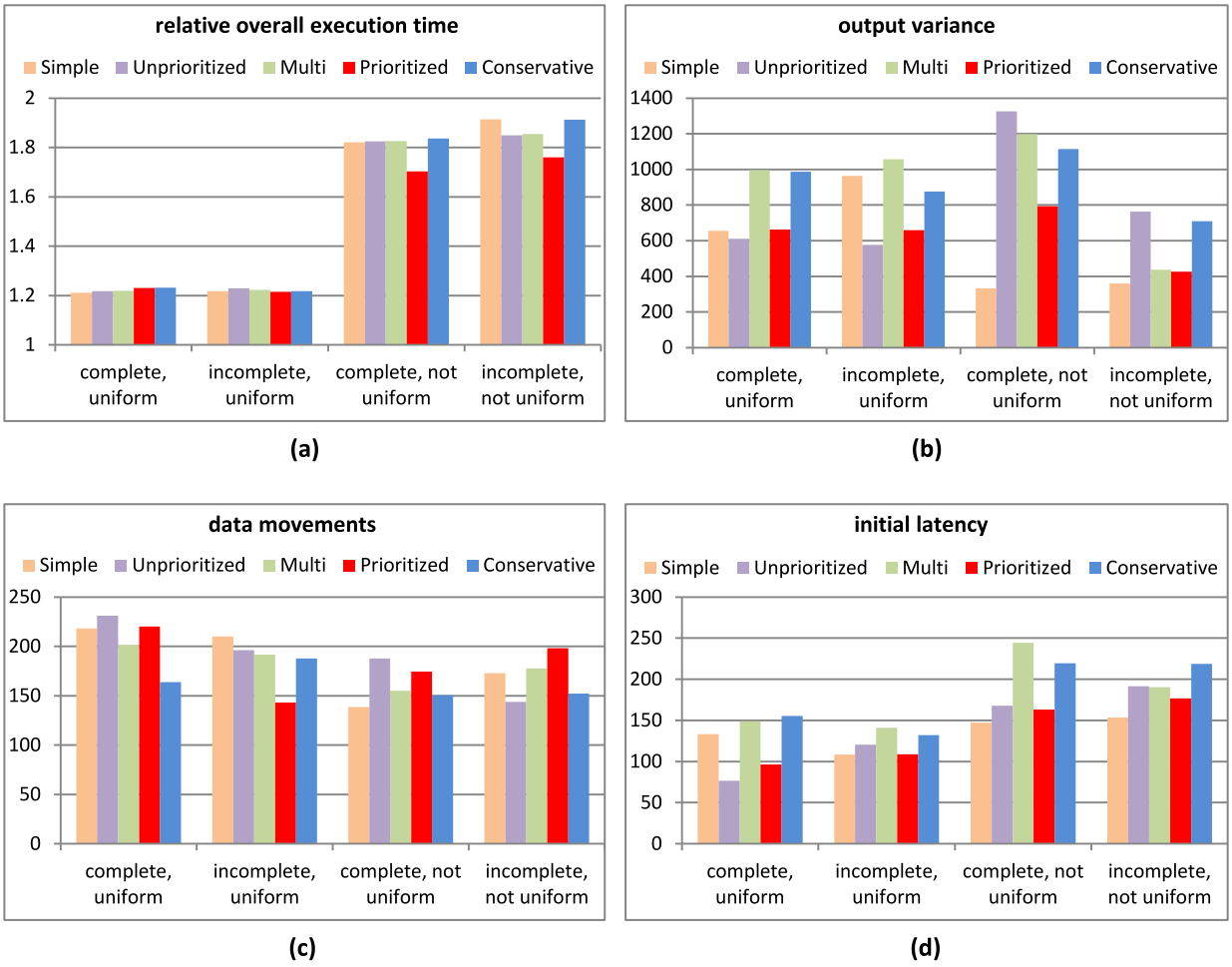


Figure 6: Measuring of a synthetic pipeline with static stage execution times. The systems use three CPU cores and one GPU engine. Short bars are better. (a) This benchmark shows the time factor the system needs compared to the lower bound. A value of 1 means lower bound. (b) A small value means evenly distributed data output over time. (c) Number of all up- and downloads to and from the GPU. (d) The time l_0 in milliseconds the first data package needs until output.