HAW HAMBURG

Forschungswerkstatt 1

Moritz Höwer

Laufzeitoptimierung heterogener Multiprozessorsysteme im Automotive Bereich

Betreuung durch: Prof. Dr. Franz-Josef Korf
Eingereicht am: 15. Februar 2021

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Zielsetzung und Motivation 2

2 Grundlagen **2**
2.1 Rechnerarchitekturen 3
2.2 Digitaler Signalprozessor (DSP) 5
2.3 Functional Safety and Road Vehicle Standard (ISO 26262) 6

3 Herausforderungen und verwandte Arbeiten **7**
3.1 Architecture Mapping & Offloading 7
3.2 Laufzeitoptimierung 9
3.3 Zertifizierung nach ISO 26262 11

4 Fazit & Ausblick **13**

Literatur **14**

Selbstständigkeitserklärung **17**

Abstract:

In dieser Ausarbeitung wird zunächst eine Einführung in das Themengebiet der Laufzeitoptimierung auf heterogenen Multiprozessorsystemen gegeben, wobei grundlegende Rechnerarchitekturen wie *Very Long Instruction Word* (VLIW) und *Single Instruction Multiple Data* (SIMD) eingeführt werden. Es folgt ein Einstieg in die Anforderungen an Sicherheit und Zuverlässigkeit im Automotive Bereich – verdeutlicht am Beispiel einer Fahrspurerkennung, welche als Grundlage für *Advanced Driver Assistance Systems* (ADAS) dienen kann und den *Automotive Safety Integrity Level* (ASIL) der Klasse B erfüllen muss. Zuletzt werden anhand einer Literaturrecherche verwandter Arbeiten vorgestellt sowie deren Anwendbarkeit im Bezug auf den Automotive Bereich evaluiert.

Keywords: Automotive, ASIL, DSP, heterogene Multiprozessorsysteme, ISO 26262, Laufzeitoptimierung, SIMD, VLIW

1 Einleitung

Im Jahr 1975 formulierte Gordon Moore das nach ihm benannte „Moore'sche Gesetz", wonach sich die Anzahl der Transistoren in integrierten Schaltungen alle zwei Jahre verdoppelt[16]. Dieses exponentielle Wachstum galt lange Zeit auch für die Taktraten, sodass Programmierer darauf setzen konnten, dass ihre Programme jedes Jahr schneller wurden, ohne dafür etwas am Code ändern zu müssen.[22] Dieser Trend endete jedoch um das Jahr 2003, als man auf physikalische Grenzen stieß. Die Hersteller begannen stattdessen auf Mehrkernprozessoren umzuschwenken, was die Programmierer dazu zwang ihre Programme zu parallelisieren, um deren Geschwindigkeit weiter zu steigern.[22]

In 2011 folgte mit der *Accelerated Processing Unit* (APU) von AMD das erste heterogene Multiprozessorsystem, welches eine *Central Processing Unit* (CPU) mit einer *Graphics Processing Unit* (GPU) in einem Chip integrierte. Dieser Ansatz verspricht einen schnelleren Datenaustausch zwischen den beiden Prozessoren.[7] Heutzutage integrieren moderne Chips wie der M1 von Apple[1] mehrere Prozessoren unterschiedlicher Leistungsklassen mit GPU und speziellen Hardwarebeschleunigern um möglichst viel Leistung bei möglichst geringem Stromverbrauch zu erzielen.

[1] https://www.apple.com/de/newsroom/2020/11/apple-unleashes-m1/ – Abgerufen am 11.02.21

1.1 Zielsetzung und Motivation

Im Verlauf des Masters soll – verteilt auf die Forschungswerkstätten 1&2, das Grund- und Hauptprojekt sowie die Masterarbeit – eine Fahrspurerkennung auf der TDA4VM[2]-Plattform von Texas Instruments umgesetzt werden.

Der TDA4VM ist ein heterogenes Multiprozessorsystem mit mehreren verschiedenen *Reduced Instruction Set* (RISC)-Prozessoren, Digitalen Signalprozessoren (DSPs) und Beschleunigern. Da zwei der RISC-Prozessoren in einer „MCU-Safety-Island-Domain“ – also mit eigener Spannungsversorgung, getrennt und unabhängig von der „Main-Domain“ – organisiert sind, plant der Hersteller eine Zertifizierung gemäß des *Functional Safety and Road Vehicle Standard* (ISO 26262) zu ermöglichen, was eine Voraussetzung für die Nutzung im Automotive Bereich ist. Die Plattform eignet sich gut für die Serienproduktion, da sie sowohl kostengünstig als auch stromsparend ist und somit passiv gekühlt werden kann. Daraus ergeben sich jedoch Begrenzungen hinsichtlich der verfügbaren Rechenleistung im Vergleich zu typischen Desktop oder Laptop Computern.

Es stellen sich zahlreiche Fragen hinsichtlich der Laufzeitoptimierung sowie auch der Zertifizierung einer Fahrspurerkennung auf dieser Plattform. Im Folgenden wird mittels einer Literaturrecherche der Grundstein zu deren Beantwortung gelegt. Zunächst erklärt Abschnitt 2 die Grundlagen, auf denen dann im Abschnitt 3 die Herausforderungen aufbauen. Ebenfalls in Abschnitt 3 werden verwandte Arbeiten vorgestellt, bevor Abschnitt 4 die Ausarbeitung mit einem Fazit und Ausblick abschließt.

2 Grundlagen

In diesem Abschnitt werden zunächst einige grundlegende Begriffe und Konzepte erläutert, die für den weiteren Verlauf der Ausarbeitung relevant sind. Dabei bezeichnet „nebenläufig“ die Unabhängigkeit von der konkreten Ausführungsreihenfolge, wohingegen „parallel“ den Spezialfall der gleichzeitigen Ausführung (auf mehreren Prozessor(kern)en) bezeichnet.[3]

[2] https://www.ti.com/product/TDA4VM – Abgerufen: 24.01.2021
[3] http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/SeminarDidaktik/Nebenlaeufigkeit/nebenlaeufig.htm – Abgerufen am 09.02.2021

2.1 Rechnerarchitekturen

Wie in Abschnitt 1 beschrieben müssen Programmierer zunehmend auf Nebenläufigkeit setzen, um die Ausführungszeit ihrer Programme zu verkürzen. Diese kann auf drei Ebenen erreicht werden:

- Function Parallelism
- Data Parallelism
- Instruction Level Parallelism

Function Parallelism (auch Task Parallelism) bezeichnet Parallelität auf der Ebene von Funktionen. Das Programm wird dazu in mehrere Ausführungszweige („*threads*“) aufgeteilt, die dann gleichzeitig auf den unterschiedlichen Kernen eines Mehrkernprozessors ausgeführt werden können. Auch das Verteilen von Rechenaufgaben auf mehrere Prozessoren fällt unter den Begriff des Function Parallelism. Dabei können die Prozessoren baugleich (homogenes Multiprozessorsystem), oder unterschiedlich (heterogenes Multiprozessorsystem) sein. Eine Herausforderung beim Function Parallelism ist die Synchronisation der einzelnen threads beim Zugriff auf geteilte Ressourcen. Beachtet man dies nicht kommt es zu Wettlaufsituationen („*race-conditions*“) und in der Folge zu schwer auffindbaren, nichtdeterministischen Programmfehlern.

Beim Data Parallelism (auch als *Single Instruction Multiple Data* (SIMD) oder Vektorisierung bezeichnet) wird die reguläre Struktur der Eingabedaten ausgenutzt um die Anzahl der zur Berechnung nötigen Instruktionen zu verringern. Voraussetzung dazu ist ein spezieller Prozessor, welcher SIMD-Instruktionen unterstützt. Verdeutlichen lässt sich das Prinzip von Data Parallelism an der Addition zweier Vektoren: Klassischerweise würde man diese in einer Schleife elementweise addieren. Dabei werden wiederholt dieselben Instruktionen ausgeführt, jedoch mit unterschiedlichen Eingabedaten. Da der einzelne Schleifendurchlauf (mit Ausnahme des Zählers) keine Abhängigkeiten zu den anderen Durchläufen hat ist eine solche Schleife „vektorisierbar“ und kann durch eine SIMD-Instruktion ersetzt werden.[20] Diese ermöglicht beispielsweise das gleichzeitige Addieren von bis zu 16 `float`-Werten[4] mit nur einer Instruktion, wodurch sich die Ausführung signifikant beschleunigen lässt. Falls die Unabhängigkeit der einzelnen Schleifendurchläufe voneinander nicht gegeben ist (z.B. bei der Berechnung einer Summe), kann jedoch nicht ohne Weiteres mittels SIMD beschleunigt werden.

[4] Beispielhafter Wert auf Basis der Intel Advanced Vector Extensions 512 (AVX-512)

Als Instruction Level Parallellism bezeichnet man die parallele Ausführung mehrerer Instruktionen. Dies erfordert, dass der Prozessor mehrere Ausführungseinheiten (Functional Units) hat, welche unabhängig voneinander arbeiten können. Ein solcher Prozessor wird als „Superskalar“ bezeichnet. Für Instruction Level Parallelism muss – anders als bei Data und Function Parallelism – der Programmcode nicht verändert werden.

Bei *Very Long Instruction Word* (VLIW)-Prozessoren obliegt es dem Compiler für jeden Taktzyklus und jede Functional Unit eine Instruktion zu generieren. Gelingt dies nicht müssen *No Operation* (`NOP`)-Instruktionen eingefügt werden, denn um die Korrektheit der Ausführung sicherzustellen kann der Compiler immer nur diejenigen Instruktionen verwenden, deren Abhängigkeiten bereits ausgeführt wurden[5].[9] Dazu ist viel Wissen über den genauen internen Aufbau des Prozessors nötig, wodurch VLIW-Compiler sehr komplex werden. Ein Vorteil dieses Ansatzes ist, dass VLIW-Hardware vergleichsweise simpel gestaltet werden kann. Nachteilig ist jedoch, dass das Kompilat nur von dem einen Prozessor ausgeführt werden kann, für den es erstellt wurde. Bei einem Wechsel des Prozessors muss neu kompiliert werden.[8]

Ein Alternativer Ansatz sind *In-Order*-Prozessoren. Deren Aufbau ist komplexer, da sie Logik zur Prüfung der Abhängigkeiten zwischen den Instruktionen besitzen. Dies ermöglicht es dem Prozessor die Instruktionen dynamisch zur Ausführungszeit auf die Functional Units zu verteilen und somit mehrere unabhängige Instruktionen gleichzeitig auszuführen. Das *In-Order* bedeutet dabei, dass die Reihenfolge der Instruktionen nicht verändert wird. Kann eine Instruktion aufgrund ihrer Abhängigkeiten noch nicht ausgeführt werden muss gewartet werden. Das versuchen *Out-of-Order*-Prozessoren zu verbessern, indem sie die Reihenfolge der Instruktionen verändern. Um trotzdem die korrekte Ausführung des Programms zu garantieren wird jedoch nochmals komplexere Hardware benötigt. Durch Verlagerung der aufwändigen Abhängigkeitsprüfung in die Hardware wird sowohl bei *In-* als auch *Out-of-Order* Prozessoren die Komplexität des Compilers verringert. Zudem kann ein Kompilat für einen solchen Prozessor auch auf einen anderen Prozessor mit gleichem Instruktionssatz übertragen werden, ohne dass ein erneutes Kompilieren notwendig wird.

[5] Beispielsweise kann einen Addition erst nach erfolgreichem Laden der Summanden aus dem Hauptspeicher ausgeführt werden.

2.2 Digitaler Signalprozessor (DSP)

Als DSPs bezeichnet man spezialisierte Prozessoren, welche für die Anforderungen der digitalen Signalverarbeitung entwickelt wurden.

Bei der Signalverarbeitung werden auf einem Eingangssignal – grundsätzlich ein kontinuierlicher Datenstrom (z.B. Audiosignal eines Mikrofons, Bild einer Kamera, …) – wiederkehrend immer dieselben Rechenoperationen ausgeführt. Eine typische Anwendung ist das FIR-Filter[13]. Dabei werden in jedem Zeitschritt der aktuelle Eingangswert und die vorherigen $N - 1$ Eingangswerte jeweils mit einem Koeffizienten multipliziert und dann zum Ausgabewert aufsummiert. Da im nächsten Zeitschritt bereits ein neuer Eingabewert vorliegt gelten Echtzeitanforderungen im Bezug auf die maximale Laufzeit der Filteroperation.

Aufgrund dieser Anforderungen sind DSPs anders aufgebaut als *General Purpose Processor*s (GPPs). Da wiederholt dieselben Rechenoperationen ausgeführt werden und es wenig Kontrollfluss gibt wird auf *branch prediction* verzichtet. Die in jedem Zeitschritt neuen Eingabedaten erfordern eine hohe Speicherbandbreite und machen Caches auf dem Datenpfad unnötig. Stattdessen haben DSPs üblicherweise schnelles *On-Chip-Memory* für Zwischenergebnisse.[8] Beides führt außerdem zu einer besseren Vorhersagbarkeit der Ausführungszeit (keine *cache-misses* oder *missprediction*) und vereinfacht so die Abschätzung der Maximallaufzeit. Das zuvor erwähnte Multiplizieren mit anschließendem Aufsummieren (Akkumulieren) wird als *Multiply-Accumulate* (MAC) bezeichnet und kommt in der digitalen Signalverarbeitung sehr häufig vor. Daher implementieren DSPs oftmals die Multiplikation in nur einem Taktzyklus und bieten breitere Register zum Akkumulieren an, um *overflows* zu vermeiden. Häufig enthält der Instruktionssatz auch eine spezielle MAC-Instruktion.[8]

Grundsätzlich ist die Programmierung eines DSP komplizierter als die eines GPP – ein typisches DSP-Programm ist stark auf den speziellen Prozessor angepasst und somit nicht ohne weiteres auf andere Prozessoren übertragbar. Die Entwicklungskosten eines DSP-Programms sind daher hoch. Aufgrund des einfacheren Aufbaus ist die DSP-Hardware jedoch günstiger und auch stromsparender als GPPs mit vergleichbarer Rechenleistung, sodass sich DSPs gut für die Serienproduktion eignen.

2.3 Functional Safety and Road Vehicle Standard (ISO 26262)

Der ISO 26262 ist eine Spezialisierung der Sicherheitsnorm „Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme“ (IEC 61508) für den Automobilbereich. Er definiert den nach Produktsicherheitsgesetz einzuhaltenden „Stand der Wissenschaft und Technik“ und muss daher von allen Herstellern und Zulieferern im Automobilbereich eingehalten werden.[11]

Der ISO 26262 regelt in 10 Teilen den Produktsicherheitslebenszyklus von der Entwicklung bis zur Außerbetriebnahme.[11] Relevant für diese Arbeit ist hauptsächlich der Teil 6 zur Produktentwicklung auf Softwareebene. Dort werden Vorgaben in folgenden Bereichen gemacht:

- Software Sicherheitsanforderungen
- Architektur der Gesamtsoftware
- Design und Implementierung der einzelnen Module
- Verifikation der Module
- Verifikation und Testen der Gesamtsoftware

Der Geltungsbereich des Standards ist die Verhinderung von Gefahren für Mensch und Maschine durch sporadisch im Betrieb auftretende Hardwarefehler, sowie die Vermeidung von systematischen Fehlern bei der Entwicklung der Software. Nicht betrachtet werden böswillige Manipulationen[21] oder die Erfüllung der Kundenanforderung. Dazu wird, basierend auf einer Risiko- und Gefährdungsanalyse ein *Automotive Safety Integrity Level* (ASIL) von A-D für die zu entwickelnde Software zugewiesen. ASIL-D steht dabei für das höchste Gefährdungspotential und erfordert strenge Vorgaben, wohingegen bei geringem Gefährdungspotential auf einen ASIL verzichtet werden kann und das Unternehmenseigene Qualitätsmanagement (QM) ausreicht.[11] Ein solcher ASIL gilt ebenfalls für die zur Ausführung der Software eingesetzte Hardware (geregelt in Teil 5 des ISO 26262), was maßgeblichen Einfluss auf die in Unterabschnitt 1.1 beschriebene Auswahl der Plattform hatte.

Um die strengen Vorgaben zu reduzieren, die aus einer ASIL-Einstufung folgen, gibt es unter anderem die Möglichkeit der *ASIL-decomposition*. Unter Einhaltung gewisser Rahmenbedingungen erlaubt diese eine Funktionalität mit hohem ASIL (z.B. ASIL-D) in zwei Funktionalitäten mit niedrigeren ASILs (z.B. 2x ASIL-B) aufzuteilen, wenn letztere redundant dieselbe Funktionalität bereitstellen und außerdem unabhängig sind. Unabhängig meint dabei, dass ein Fehler in einem System nicht das andere beeinflusst. Dies

wird auch *freedom from interference* genannt und ermöglicht außerdem das Ausführen von Komponenten unterschiedlicher Kritikalitäten, bis hin zur Ausführung von Komponenten ohne jegliche Sicherheitseinstufung auf demselben System, ohne dass dabei alle Komponenten den höchsten ASIL erben. Beides kann den Entwicklungsaufwand erheblich verringern, da der nach funktionalen Sicherheitsanforderungen zu entwickelnde Code auf das notwendige Minimum beschränkt wird.

3 Herausforderungen und verwandte Arbeiten

Ein Algorithmus zur Fahrspurerkennung wurde in [12] bereits entwickelt und auf einem Jetson AGX Xavier[6] optimiert. Dieser ist – wie der TDA4VM – ebenfalls ein heterogenes Multiprozessorsystem, jedoch mit einer GPU anstelle von DSPs und Beschleunigern. Der Algorithmus soll nun, wie in Unterabschnitt 1.1 beschrieben, weiterentwickelt werden.

Die sich dabei ergebenden Herausforderungen lassen sich in drei Kategorien einteilen – Architecture Mapping & Offloading, Laufzeitoptimierung und die Zertifizierung nach ISO 26262 – welche in den folgenden Unterabschnitten erläutert werden.

3.1 Architecture Mapping & Offloading

Für die Fahrspurerkennung zur Verfügung stehen auf dem TDA4VM konkret zwei *Out-of-order* Rechenkerne mit SIMD-Instruktionen[7], drei DSPs mit VLIW-Architektur[8] sowie Spezialbeschleuniger zur Matrixmultiplikation bzw. Kameradatenverarbeitung. Definiert werden muss hier somit zunächst das Architecture Mapping, also die Zuordnung welche Teile des Algorithmus auf welchem Rechenkern laufen. Um diese Zuordnung dann umzusetzen muss das Hauptprogramm zur Laufzeit einzelne Berechnungen auslagern („*Offloading*“).

Grundsätzlich kann das Architecture Mapping statisch oder dynamisch sein – aufgrund der Echtzeitanforderungen ist für die vorliegende Anwendung eine statisches Zuordnung zu bevorzugen. Das Offloading erfordert immer eine Kommunikation und Synchronisation zwischen verschiedenen Rechenkernen, wodurch ein Overhead entsteht. Dieser muss

[6] https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-agx-xavier/ – Abgerufen am 24.01.21
[7] Konkret handelt es sich um einen Dual Core ARM Cortex-A72
[8] Konkret handelt es sich dabei um 2x C66x & 1x C71x

in die Laufzeit der ausgelagerten Berechnung einbezogen werden und somit auch in die Entscheidung, ob sich ein Offloading für eine bestimmte Berechnung lohnt.

In [4] wird eine Abstraktionsschicht entwickelt, um die Entwicklung paralleler Programme auf heterogenen Multiprozessorsystemen zu unterstützen. Der Fokus liegt dabei auf Portabilität, welche in der vorliegenden Anwendung nicht gefordert ist, und darauf, Vorhersagen über das Laufzeit- und Skalierungsverhalten zu treffen. In [2] hingegen beschreiben die Autoren einen Ansatz für automatisierten Parallelisierung und Offloading in heterogenen Multiprozessorsystemen. Der vorgestellte Algorithmus basiert auf clang/LLVM[9] und dem in OpenMP 4.0[10] eingeführten *accelerator model*. Basierend auf einem Modell des Systems, welches die Eigenschaften der verschiedenen Prozessoren sowie der Kommunikation dazwischen beschreibt, wird der Quellcode analysiert. Dabei identifiziert der vorgestellte Algorithmus Codeabschnitte, die parallelisiert oder ausgelagert werden können, und annotiert den Ursprungsquelltext dann mit den entsprechenden OpenMP-Direktiven.

Im Kontext der Serienentwicklung für den Automotive Bereich sind beide Ansätze problematisch, da sie generisch und damit komplex sind. Dies erschwert eine Zertifizierung nach ISO 26262 oder macht diese sogar gänzlich unmöglich. Eine Alternative zum Offloading kann darin bestehen, ein separates und eigenständiges Programm für jeden Co-Prozessor zu schreiben. Diese Programme können dann je nach Plattform über gemeinsamen geteilten Speicher, Mailboxen oder ähnliches mit dem Hauptprogramm kommunizieren. Letzteres kann darüber dann einen ebenfalls im Programm für den Co-Prozessor fest enthaltenen Algorithmus anstoßen und nach Beendigung das Ergebnis abfragen. Dieser Ansatz ist zwar nicht generisch oder portabel, dafür aber sehr einfach und somit besser zertifizierbar.

[9] Das LLVM-Projekt ist eine Sammlung modularer und wiederverwendbarer Compiler und Toolchain Technologien. (https://llvm.org/ – Abgerufen am 13.02.21)

[10] OpenMP ist eine Schnittstellenspezifikation für Parallele Programmierung mit Unterstützung in gängigen Compilern. (https://www.openmp.org/ – Abgerufen am 13.02.21)

3.2 Laufzeitoptimierung

Eng verzahnt mit den Fragen des Architecture Mapping sind die Fragen nach den Optimierungsmöglichkeiten der verschiedenen Rechenkerne, denn um trotz der geringen Leistungsaufnahme von unter 20W[11] für den gesamten TDA4VM echtzeitfähig zu bleiben müssen alle vorhandenen Ressourcen ausgenutzt werden.

Herausforderungen für gute Vektorisierbarkeit mit SIMD sind Speicherausrichtung, Vektorisierungsverhältnis[12] und Cache-Effizienz.[15] Ideal sind dabei Zugriffe auf einen kontinuierlichen Speicherblock. Zwar bieten moderne SIMD-Instruktionssätze auch Operationen um nicht-kontinuierlichen Speicher in das Vektorregister zu laden („gather“) oder zu speichern („scatter“)[15]. Diese haben jedoch höhere Latenzen als die kontinuierlichen Varianten und erzeugen durch die fehlende Lokalität im Speicher zudem vermehrte *cache-misses*.[15] Unbedingt vermeiden sollte man den Zugriff über Wortgrenzen hinweg („*unaligned*“), da dieser jeweils zwei Speicherzugriffe erzeugt und somit nochmals deutlich langsamer ist. Um das ideale Vektorisierungsverhältnis von 1 zu erreichen sollte möglichst die gesamte Länge des Vektorregisters genutzt werden, da dieses ansonsten mit *padding* aufgefüllt werden muss und Rechenleistung ungenutzt bleibt.[15]

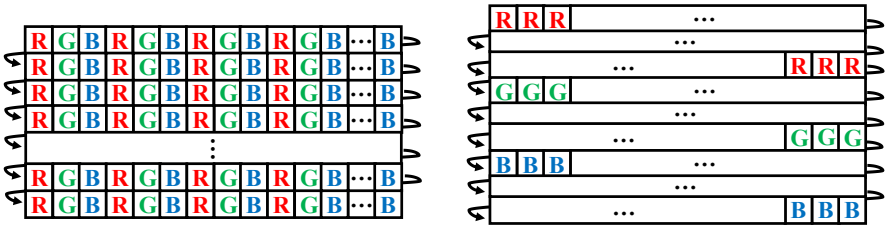

Abbildung 1: *Array of Structure* (AoS) ↔ *Structure of Array* (SoA) – Quelle: [15]

Aufgrund des hohen Einflusses des Speicherlayouts auf die Vektorisierbarkeit kann es sich lohnen, den Speicher vor der Berechnung anders zu organisieren. Abbildung 1 zeigt zwei Möglichkeiten ein Farbbild im Speicher darzustellen – bei der *Array of Structure* (AoS)-Repräsentation werden die Rot, Grün und Blau Werte der Pixel hintereinander gespeichert, während bei der *Structure of Array* (SoA)-Repräsentation erst alle Roten, dann alle Grünen und dann alle Blauen Pixel abgelegt werden. Je nach Zugriffsmuster

[11] https://news.ti.com/new-low-power-high-performance-ti-jacinto-7-processors-enable-mass-market-adoption-automotive-adas-and-gateway-technology – Abgerufen am 02.02.2021
[12] Verhältnis der genutzten Einträge im Vektorregister zur Länge

der Berechnung sollten die Daten zuvor in die passendere Repräsentation transformiert werden.[15]

Weitere Möglichkeiten die Vektorisierung zu verbessern sind das Ausrollen von Schleifen („*loop-unrolling*“) oder das Ausblenden von Vektorelementen mithilfe von Vektormasken („*predication*“). Beides reduziert die Anzahl der bedingten Sprünge und ermöglicht somit kürzere Folgen von Instruktionen.

Da nahezu alle modernen Prozessoren über SIMD-Instruktionen verfügen, unterstützen GCC[13][17] und clang/LLVM[6] sowie auch andere Compiler heutzutage *auto-vectorization*. Dabei werden beispielsweise anhand von Datenabhängigkeitsanalyse vektorisierbare Schleifen ermittelt und automatisch durch SIMD-Instruktionen ersetzt. Da die Compiler jedoch korrekten Code auf jeder unterstützten Plattform garantieren müssen, und die SIMD-Funktionalitäten sehr plattformspezifisch sind, ist auto-vectorization üblicherweise konservativer und somit nicht so performant wie handoptimierter Code.[17]

Bei VLIW-Prozessoren ergibt sich eine ähnliche Problematik wie die des Vektorisierungsverhältnisses. Anders als bei SIMD müssen hier jedoch nicht ausreichend unabhängige Daten pro Taktzyklus zur Verfügung stehen, sondern ausreichend unabhängige Instruktionen, um alle Functional Units auszulasten. Diese Problemstellung ist deutlich komplexer als die der Vektorisierung und mit steigender Länge des VLIW wird es für den Programmierer zunehmend schwerer den Überblick zu behalten. Wie in Abschnitt 2 beschrieben übernehmen daher die VLIW-Compiler diese Aufgabe, beispielsweise mittels *trace scheduling*[9]. Da ein VLIW-Compiler nur Code für eine einzelne Architektur generieren muss, und alle benötigten Informationen zu dieser kennt, gibt es anders als bei der auto-vectorization keine Portabilitätsprobleme und es kann sehr effizienter Code generiert werden. Allerdings profitieren auch VLIW-Compiler davon, wenn der Programmierer gut parallelisierbaren Code schreibt. Die zuvor dargelegten Überlegungen zum Speicherlayout sind daher auch für VLIW-Architekturen relevant.

Die genannten Rahmenbedingungen finden sich ebenfalls im Konzept des „Stream Programming“[1] wieder. Dabei geht es darum, große Datenströme, deren Elemente individuell bearbeitet werden können, mithilfe unabhängiger Filter (auch genannt „*kernels*“) in immer gleicher, wiederkehrender Reihenfolge zu bearbeiten. Üblicherweise in Verbindung mit Echtzeitanforderungen[23]. Wie in Unterabschnitt 2.2 dargestellt ist dies der zugrunde liegende Anwendungsfall für DSPs, das Stream Programming eignet sich

[13] Die GNU Compiler Collection (https://gcc.gnu.org/ – Abgerufen am 13.02.2021)

jedoch auch für SIMD-Prozessoren. Zwar gibt es eigene Programmiersprachen für des Stream Programming (z.B. StreamIt[23]), jedoch können die Ansätze mit Kompromissen auch auf andere Programmiersprachen übertragen werden. Die Autoren von [24] nutzen beispielsweise Ansätze aus dem Stream Programming um Data Parallelism auf einem clustered VLIW-Prozessor[14] umzusetzen.

3.3 Zertifizierung nach ISO 26262

Aufgrund des geplanten Einsatzes in einem Serienprodukt im Automotive Bereiche sind, wie in Unterabschnitt 2.3 beschrieben, die Vorgaben des ISO 26262 einzuhalten. Dafür muss zunächst ein ASIL für die Funktionalität festgelegt werden. Da der dafür nötige Prozess den Rahmen dieser Ausarbeitung sprengen würde werden im Folgenden verkürzt nur die Kernaspekte dargestellt: Ausgangspunkt ist das gesamte *Advanced Driver Assistance System* (ADAS) – beispielsweise ein Auto- oder Staupilot. Dieses muss zur Realisierung seiner Aufgabe in die Fahrfunktionen des Fahrzeugs eingreifen, wodurch Fehler ein hohes Gefährdungspotential haben. Daher erhält das ADAS insgesamt eine ASIL-D-Einstufung. Der Algorithmus zur Fahrspurerkennung ist allerdings nur ein Teil des ADAS. Es wird zudem angenommen, dass mehrere Algorithmen diese Information auf Basis unterschiedlicher Eingangsdaten liefern (z.B. basierend auf LiDAR-Sensoren [12] oder Kameras [14]). Über die in Unterabschnitt 2.3 beschriebene *ASIL decomposition* kann daher der ASIL für die beiden einzelnen Algorithmen auf ASIL-B gesenkt werden.

Angewendet auf die vorliegende Problemstellung ergeben sich aus dem ASIL-B unter anderem folgende Anforderungen:

- Die Komplexität sollte so gering wie möglich gehalten werden
- Beschränkung auf ein als sicher geltendes Subset der verfügbaren Sprachfeatures (z.B. MISRA-C++[15])
- Keine bedingungslosen Sprunganweisungen (`goto`)
- Keine Nutzung von dynamischem Speicher ohne Überprüfung zur Laufzeit
- Keine impliziten Typumwandlungen
- Statische Analysen auf dem Code müssen durchgeführt werden
- Tests müssen auf niedergeschriebenen Anforderungen basieren

[14] Bei einem clustered VLIW-Prozessor sind die Functional Units nicht nur einem langen Instruktionswort untergeordnet, sondern in Clustern mit jeweils eigenen, kürzeren, Instruktionswort organisiert.
[15] https://www.misra.org.uk/Activities/MISRAC/tabid/171/Default.aspx – Abgerufen 14.02.2021

- Tests müssen Äquivalenzklassen abdecken
- Es muss Statement und Branch coverage ausgewertet werden[16]
- Es müssen *Hardware in the Loop* (HiL)-Tests durchgeführt werden

In [10] wird exemplarisch eine Umsetzung des ISO 26262 am Beispiel der Systemarchitektur eines Steuergeräts für das Kombiinstrument im Auto dargestellt. Dabei wird vor allem dargelegt, wie die in Unterabschnitt 2.3 beschriebene *freedom from interference* erreicht wird, um die ASIL-A-klassifizierten sicherheitskritischen Meldungen gemeinsam mit nicht sicherheitskritischen Meldungen anzuzeigen.

Die Autoren von [5] stellen einen Software Stack (HERCULES) vor, mit dem Ziel ein System aus *commercial-off-the-shelf* (COTS) Produkten mit den Safety- und Echtzeitanforderungen des ISO 26262 zu verbinden. Dabei sollen nach ASIL-B oder ASIL-D zertifizierte Komponenten neben *high-performance computing* (HPC) Anwendungen ohne Sicherheits- und Echtzeitanforderungen auf auf derselben Hardware laufen. Um die für das Gesamtsystem notwendige Zertifizierung zu ermöglichen wird *freedom from interference* dabei durch Virtualisierung erreicht. Bei der Auswahl des Hypervisors wurde auf geringe Komplexität und kleine Codebase geachtet.

Beide Ansätze lösen jedoch nicht die in Unterabschnitt 1.1 beschriebene Problemstellung, dass eine ISO 26262-Zertifizierung auch für die HPC-Anteile benötigt wird.

Näher an der Problemstellung sind hingegen die Autoren von [18], die Sicherheits- und Echtzeitanforderungen auf GPU-beschleunigte ADAS betrachten. In [19] erreichen die Autoren Echtzeitfähigkeit auf COTS-Systemen, indem unter anderem vor längeren Berechnungen alle benötigten Daten in den Cache geladen werden, sodass die Berechnung dann weitgehend ohne *cache-misses* und Zugriffe auf den Hauptspeicher laufen kann und somit vorhersagbar wird. Jedoch werden dafür außerdem noch eigens von den Autoren entwickelte Hardware Bausteine („*Real-Time Bridges*“) benötigt, um die Peripheriegeräte zu kontrollieren. In [3] schlagen die Autoren eine Softwarelösung vor, um auf einem COTS-Multicoresystem mithilfe von Diversität in der Ausführung sogenannte *common cause failure* (CCF) erkennen zu können – eine Voraussetzung für ASIL-D-Systeme. Dabei wird die Berechnung redundant auf zwei unterschiedliche Kernen ausgelagert, einem *HEAD* und einem verzögert gestarteten *TAIL*. Ein separater Monitor überwacht dabei periodisch die Instruktionszähler der beiden Kerne und hält den *TAIL* kurzzeitig an, wenn dieser im nächsten Zeitintervall droht den *HEAD* einzuholen. Durch die zeitliche

[16] Das dafür genutzte Tool muss nach ISO 26262 qualifiziert sein

Separation betrifft ein CCF die beiden Berechnungen immer an unterschiedlichen Stellen und tritt somit beim abschließenden Vergleich des Ergebnisses in Form einer Abweichung zutage.

4 Fazit & Ausblick

In dieser Ausarbeitung wurde ein Einstieg in die Herausforderungen bei der Optimierung einer Fahrspurerkennung auf einem heterogenen Multiprozessorsystem im Hinblick auf eine Zertifizierung nach ISO 26262 gegeben. Dabei stellt letzteres die größte Hürde dar. Zwar finden sich in der Literatur verschiedene Ansätze zur teils automatisierten Beschleunigung mit SIMD oder zur Auslagerung von Programmteilen auf Co-Prozessoren, jedoch sind diese Ansätze häufig generisch und damit nicht oder nur mit unverhältnismäßig hohem Aufwand zertifizierbar. Für die Umsetzung einer eigenen Lösung können jedoch die Ideen hinter den Ansätzen verwendet werden. Ebenfalls möglich ist das Erstellen von Prototypen, beispielsweise mit OpenMP, um diese dann zu analysieren und die Erkenntnisse in eine eigene, plattformspezifische und somit zertifizierbare Umsetzung einfließen zu lassen.

Als nächstes soll im Rahmen des Grundprojekt die geplante Zielplattform, der TDA4VM, genauer untersucht werden. Die Erkenntnisse daraus fließen dann zusammen mit der hier ermittelten Literatur in die Forschungswerkstatt 2 und das Hauptprojekt, wo es um eine erste konkrete Implementierung der Fahrspurerkennung gehen soll. Diese soll dann schließlich in der Masterarbeit verfeinert und auf die Zertifizierung vorbereitet werden.

Literatur

[1] Abelson, Harold ; Sussman, Gerald J.: *Structure and interpretation of computer programs*. The MIT Press, 1996

[2] Aguilar, M. A. ; Leupers, R. ; Ascheid, G. ; Murillo, L. G.: Automatic parallelization and accelerator offloading for embedded applications on heterogeneous MPSoCs. In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, S. 1–6

[3] Alcaide, S. ; Kosmidis, L. ; Hernandez, C. ; Abella, J.: Software-only based Diverse Redundancy for ASIL-D Automotive Applications on Embedded HPC Platforms. In: *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2020, S. 1–4

[4] Arndt, Oliver J.: *Modellierung von paralleler Software für homogene und heterogene Multiprozessor-Systeme*, Hannover: Institutionelles Repositorium der Leibniz Universität Hannover, Dissertation, 2020

[5] Burgio, Paolo ; Bertogna, Marko ; Capodieci, Nicola ; Cavicchioli, Roberto ; Sojka, Michal ; Houdek, Přemysl ; Marongiu, Andrea ; Gai, Paolo ; Scordino, Claudio ; Morelli, Bruno: A software stack for next-generation automotive systems on many-core heterogeneous platforms. In: *Microprocessors and Microsystems* 52 (2017), S. 299–311

[6] Chen, Kuan-Hsu ; Shen, Bor-Yeh ; Yang, Wuu: An automatic superword vectorization in LLVM. In: *16th Workshop on Compiler Techniques for High-Performance and Embedded Computing*, 2010, S. 19–27

[7] Daga, M. ; Aji, A. M. ; Feng, W.: On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In: *2011 Symposium on Application Accelerators in High-Performance Computing*, 2011, S. 141–149

[8] Eyre, Jennifer ; Bier, Jeff: DSP processors hit the mainstream. In: *Computer* 31 (1998), Nr. 8, S. 51–59

[9] Fisher, Joseph A.: Very long instruction word architectures and the ELI-512. In: *Proceedings of the 10th annual international symposium on Computer architecture*, 1983, S. 140–150

[10] Gieraths, Antje: Umsetzung der Anforderungen aus der ISO 26262 bei der Entwicklung eines Steuergeräts aus dem Fahrerinformationsbereich. In: *Automotive-Safety & Security 2014* (2015)

[11] Hillenbrand, Martin: *Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik/Elektronik Architekturen von Fahrzeugen*. KIT Scientific Publishing, 2012

[12] Höwer, Moritz: *Effiziente GPU-basierte Klassifizierung von Fahrspuren auf eingebetteten Echtzeitsystemen*, Hamburg: Hochschule für Angewandte Wissenschaften Hamburg, Bachelorarbeit, Juni 2019

[13] Kutil, Rade: *Digitale Signalprozessoren*. 2004

[14] Low, C. Y. ; Zamzuri, H. ; Mazlan, S. A.: Simple robust road lane detection algorithm. In: *2014 5th International Conference on Intelligent and Advanced Systems (ICIAS)*, Juni 2014, S. 1–4

[15] Maeda, Yoshihiro ; Fukushima, Norishige ; Matsuo, Hiroshi: Taxonomy of Vectorization Patterns of Programming for FIR Image Filters Using Kernel Subsampling and New One. In: *Applied Sciences* 8 (2018), Nr. 8. – URL https://www.mdpi.com/2076-3417/8/8/1235. – ISSN 2076-3417

[16] Moore, G. E.: Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]. In: *IEEE Solid-State Circuits Society Newsletter* 11 (2006), Nr. 3, S. 36–37

[17] Naishlos, Dorit: Autovectorization in GCC. In: *Proceedings of the 2004 GCC Developers Summit*, 2004, S. 105–118

[18] Olmedo, I. S. ; Capodieci, N. ; Cavicchioli, R.: A Perspective on Safety and Real-Time Issues for GPU Accelerated ADAS. In: *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, 2018, S. 4071–4077

[19] Pellizzoni, R. ; Betti, E. ; Bak, S. ; Yao, G. ; Criswell, J. ; Caccamo, M. ; Kegley, R.: A Predictable Execution Model for COTS-Based Embedded Systems. In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, S. 269–279

[20] Ren, G. ; Wu, P. ; Padua, D.: An empirical study on the vectorization of multimedia applications for multimedia extensions. In: *19th IEEE International Parallel and Distributed Processing Symposium*, 2005, S. 10 pp.–

[21] Robinson-Mallett, Christopher: Berührungspunkte und übergreifende Risiken des Security- und Safety Engineering durch die Vernetzung elektronischer Automobilsysteme mit Internet-Konnektivität, 09 2013

[22] Sutter, Herb: The free lunch is over: A fundamental turn toward concurrency in software. In: *Dr. Dobb's journal* 30 (2005), Nr. 3, S. 202–210

[23] Thies, William ; Karczmarek, Michal ; Amarasinghe, Saman: StreamIt: A Language for Streaming Applications. In: Horspool, R. N. (Hrsg.): *Compiler Construction*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2002, S. 179–196. – ISBN 978-3-540-45937-8

[24] Yan, Shan ; Lin, Bill: Stream execution on embedded wide-issue clustered vliw architectures. In: *EURASIP Journal on Embedded Systems* 2008 (2009), S. 1–9

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Forschungswerkstatt 1 – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Laufzeitoptimierung heterogener Multiprozessorsysteme im Automotive Bereich

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original