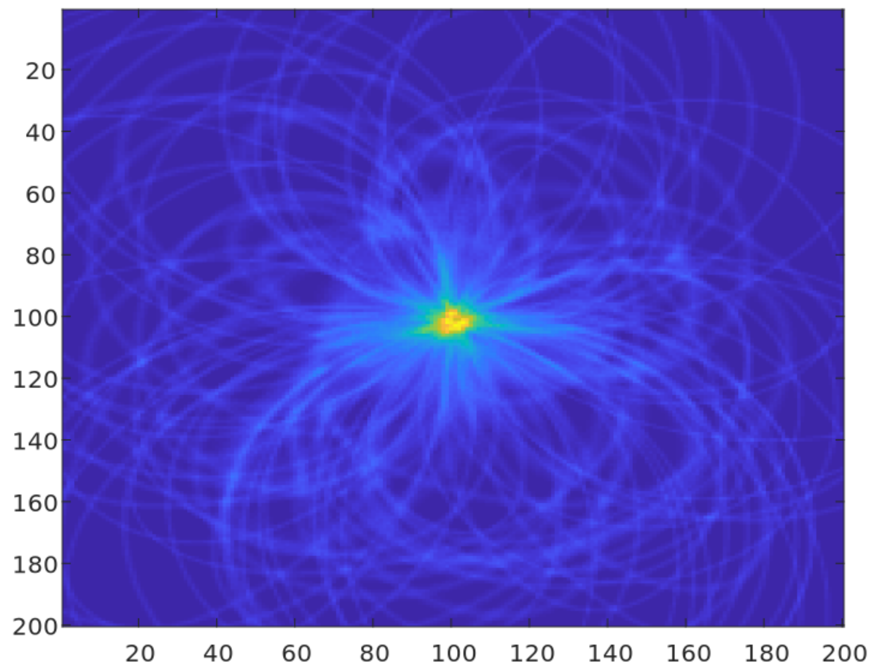
Hector Seminar

KIT
Karlsruher Institut für Technologie

AVX-Implemtierung für CPU-SAFT



Abschlussbericht der Kooperationsphase 2019/20

Durchgeführt am Institut für Prozessdatenverarbeitung und Elektronik (IPE)
des Karlsruher Instituts für Technologie (KIT)
Betreut durch Michael Zapf

Leon von Berg und Paul Schillinger, Kurs KA 14

Inhaltsverzeichnis

1 Einleitung **2**
1.1 Ultraschallmesstechnik zur Brustkrebserkennung . . . 2
1.2 Das USCT-System . . . 3
1.3 A-Scan . . . 4
1.4 Gärtnerkonstruktion . . . 4
1.5 SAFT . . . 5
1.6 SIMD . . . 6
1.7 AVX . . . 7
1.8 Aufgabenstellung . . . 7

2 Material und Methoden **8**
2.1 Verwendete Systeme . . . 8
2.2 Der KIT SAFT-Code . . . 8
2.3 MATLAB . . . 8

3 Architektur **10**
3.1 Branching . . . 10
3.2 Pipelines . . . 10

4 Konvertierung des SSE-Codes zu AVX-Code **11**
4.1 Kombinieren beider Pipelines . . . 11
4.2 Wiedereinführung der zweiten Pipeline . . . 12

5 Profiling **13**
5.1 Version 1 . . . 13
5.2 Version 2 – Probleme mit Abstürzen . . . 13
5.3 Version 3 – Datengenerierung . . . 14
5.4 Version 4 – KIT-SAFT-Code und Bildunterschiede . . . 14

6 Ergebnisse und Analyse **16**
6.1 Cache-Limitierung . . . 16
6.2 Optimierungsmöglichkeiten . . . 16
6.2.1 Dritte Pipeline . . . 16
6.2.2 Single- vs. Double-precision floating-point-Zahlen . . . 16
6.2.3 repmat . . . 18
6.2.4 Weniger Loops in MATLAB . . . 19
6.3 Instruktionen-Abwägung . . . 20
6.4 AVX-Implementierung . . . 20

7 Fazit und Ausblick **21**

8 Danksagung **22**

9 Anhang **23**

10 Literatur **III**

11 Selbstständigkeitserklärung **IV**

Abstract

Breast cancer is the most frequent form of cancer for woman. Traditionally most tumors have been detected through scanning by the patients themselves. This is a major problem because tumors that can be detected by scanning have to be of substantial size already. Therefore it is possible to lower the mortality of breast cancer by as much as 25 % with effective early detection. Up to now mammography is the most common method. Since mammography is X-ray based, the patient is exposed to ionizing radiation. A possible new method that does not increase the risk of cancer is Ultra-Sound Computer Tomography (USCT) which is currently in development at the Karlsruhe Institute of Technology (KIT). This method uses the Synthetic Aperture Focusing Technique (SAFT), which requires a very high computational effort and uses large amounts of data. Therefore it is very important that the code dealing with all the data is as efficient as possible. Since this new technology has been in research for a while, new CPU architectures capable of using newer instruction sets have been established. These new instructions sets, most notably AVX2, allow for a better implementation of Same Intruction Mulitple Data (SIMD) (which aims to increase performance in high throughput situations.) Firstly this work will report about the SAFT approach and secondly about the implementation of AVX intructions into the existing code.

1 Einleitung

In Deutschland erhalten jedes Jahr 69.000 Frauen die Erstdiagnose Brustkrebs.[1] Damit ist Brustkrebs die häufigste Art einer Krebserkrankung bei Frauen (Abb. 1),[2] wobei auch vergleichsweise junge Patientinnen betroffen sind. Fast drei von zehn Betroffenen haben ihr 55. Lebensjahr noch nicht erreicht, wenn sie die Diagnose erhalten.[1] Der Bericht zum Krebsgeschehen in Deutschland 2016 des Robert Koch-Instituts prognostiziert, dass die Zahl der Brustkrebserkrankungen in den kommenden Jahren weiter steigen wird.[3]

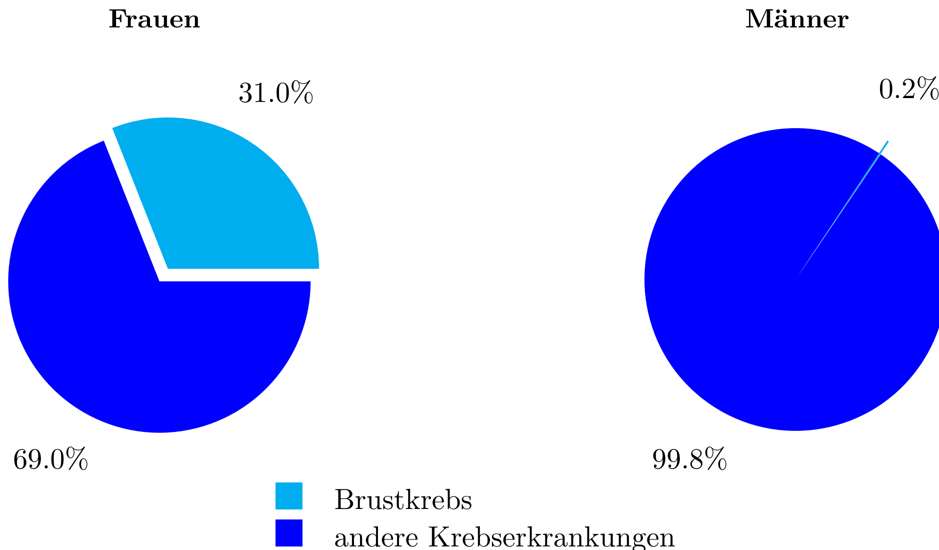


Abbildung 1: Häufigkeit von Brustkrebserkrankungen in Deutschland (2016)[2]

Wesentlich für eine erfolgreiche Behandlung ist eine frühzeitige Erkennung der Krankheit. Die heutzutage übliche Mammografie[1] nutzt ionisierende Röntgenstrahlung zum Screening der Brust. Dabei setzt sich die Patientin bei jeder dieser Untersuchungen einer Dosis schädlicher Strahlung aus. Die durchschnittliche Strahlenbelastung ist mit bis zu 0,6 mSv[4] so hoch, dass die Mammografie nur bei Frauen im Alter von 50 bis 69 Jahren sinnvoll in der Vorsorge eingesetzt werden kann, da sonst die Risiken den Nutzen überwiegen. Eine mögliche Alternative mit niedrigerem Risiko ist der Einsatz von Ultraschall (Mammosonografie). Da die bei der Mammosonografie aufgenommenen Daten noch stark verarbeitet werden müssen, um ein Bild zu erhalten, muss die Datenverarbeitung weiter verbessert werden, um gleiche Ergebnisse schneller, optimalerweise in Echtzeit, zu erhalten.

1.1 Ultraschallmesstechnik zur Brustkrebserkennung

Dank der Entdeckung des piezoelektrischen Effekts durch die Brüder Jacques und Pierre Curie im Jahr 1880 und der Erfindung piezoelektrischer Ultraschallwandler in den 1950er Jahren wurde es möglich, Ultraschallsignale elektrisch zu erzeugen und auch umgekehrt zu detektieren.[5] So wurde der Einsatz der Ultraschallmesstechnik möglich. In der medizinischen Diagnostik (Sonografie) werden Ultraschallpulse von einem Punkt aus in den Körper gesendet. Diese werden im Körper je nach Gewebetyp unterschiedlich reflektiert oder gestreut. Über einen Sensor werden die Pulse beim Austreten wieder detektiert. Werden die Ultraschallpulse nacheinander von vielen Orten aus ausgesendet und anschließend detektiert, kann aus den gewonnenen Daten (A-Scans) die Lage eines Tumors bestimmt werden.

1.2 Das USCT-System

Am Institut für Prozessdatenverarbeitung und Elektronik (IPE) des Karlsruher Instituts für Technologie (KIT) wird derzeit ein Verfahren zur Brustkrebserkennung mit Hilfe von Ultraschall entwickelt. Kernstück der Apparatur (Abb. 2a) ist das Wasserbecken (Abb. 2b). Für eine Messung legt sich die Patientin auf die Auflagefläche (in Abb. 2a hochgeklappt), sodass Ihre Brust in das mit Wasser gefüllte Becken ragt. Um die Brust herum sind 157 Ultraschallwandler angeordnet. (Abb. 2b)

(a) Aufnahme der gesamten Apparatur. Die Auflagefläche ist hochgeklappt.

(b) Nahaufnahme des Wasserbeckens (noch ohne Ultraschallwandler).

(c) einzelner Ultraschallwandler

Abbildung 2

Mit dem USCT-System lassen sich Aufnahmen der Brust erstellen. Mit Hilfe dieser kann man einen Tumor nachweisen, sowie dessen Größe und Position bestimmen (Abb. 3). Erste klinische Tests der Apparatur wurden 2013 durchgeführt.[6]

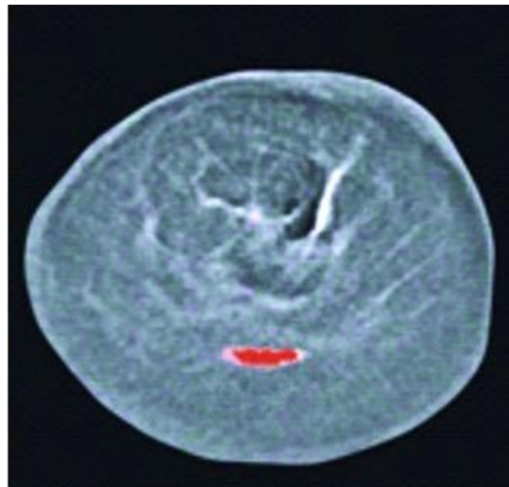


Abbildung 3: Aufnahme einer Brust mit Tumor durch das USCT-System[7]

1.3 A-Scan

Wichtig bei der USCT ist die Laufzeit, sowie die Dämpfung der in der Brust reflektierten/gestreuten Signale. Jeder der 157 Ultraschallwandler sendet nacheinander einen kurzen Ultraschallpuls (Transmissionspuls) aus. Die übrigen 156 zeichnen ab dem Zeitpunkt des Aussendens bis nach dem Eintreffen des Reflexionspulses auf. Die aufgezeichneten Signale werden als A-Scan bezeichnet (Abb. 4).

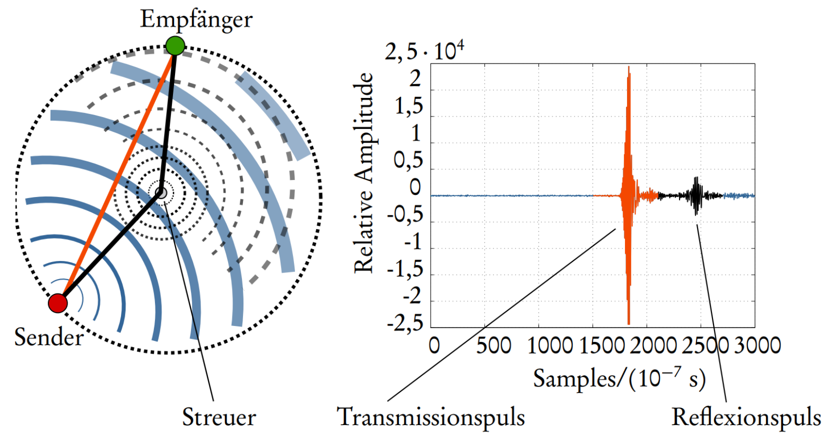


Abbildung 4: Prinzip der Messung (links) und aufgezeichnete Signale (A-Scan, rechts)[8]

1.4 Gärtnerkonstruktion

Aus den aufgezeichneten Daten kann mit Hilfe der Gärtnerkonstruktion die Lage von Tumoren bestimmt werden. Diese ist eine einfache Methode, um Ellipsen zu zeichnen (Abb. 5). Benötigt werden zwei Fixpunkte (F1 und F2), ein Stück Faden, welches länger als die Distanz zwischen den beiden Fixpunkten ist, und ein Stift. Der Faden wird an seinen Enden an den Fixpunkten befestigt und der Stift z. B. im Punkt P im Faden eingespannt. Führt man den Stift um die Fixpunkte herum, entsteht eine Ellipse. Die Form dieser Ellipse variiert je nach Länge des Fadens und Position der Fixpunkte. Im 3-dimensionalen Raum entstehen durch die Gärtnerkonstruktion Ellipsoide.
Bei der USCT fungieren Sender und Empfänger als Fixpunkte. Die Länge des virtuellen Fadens kann über den A-Scan ermittelt werden. Jedem Wert darin lässt sich eine eindeutige Signallaufzeit zuordnen. Da die Schallgeschwindigkeit bekannt ist, lässt sich so die Länge des virtuellen Fadens ermitteln. Man erhält also für jeden A-Scan viele Ellipsoide, die in das Bild eingetragen werden können.

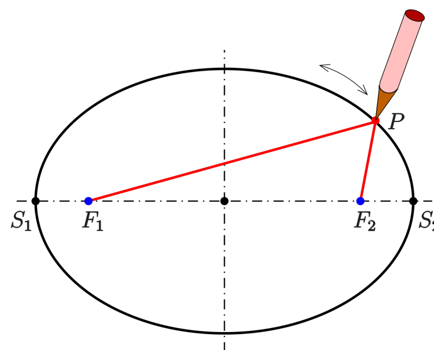
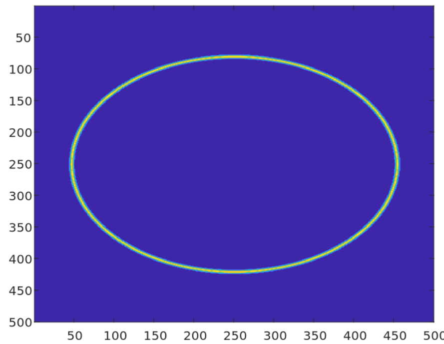


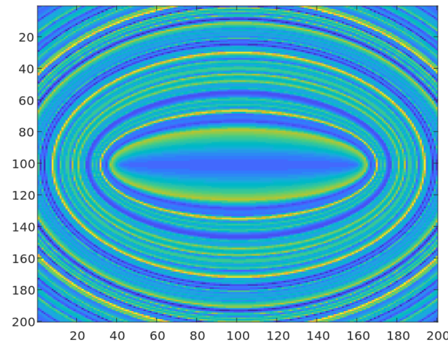
Abbildung 5: Mit Hilfe der Gärtnerkonstruktion gezeichnete Ellipse[]

1.5 SAFT

Die Synthetic Aperture Focusing Technique (SAFT) ist ein am IPE entwickelter Algorithmus, der aus den Messdaten des USCT-Systems ein 3-dimensionales Bild erstellt. Der SAFT-Algorithmus geht umgekehrt vor, wie in 1.3 beschrieben. Statt für jeden Wert im A-Scan ein Ellipsoid zu berechnen, wird für jedes Voxel im Bild ein möglicher Wert im A-Scan berechnet. Dafür wird die Distanz zwischen Sender und Empfänger über das Voxel berechnet. Die Schallgeschwindigkeit und der zeitliche Abstand zwischen Werten im A-Scan sind bekannt, sodass sich jedem Voxel ein eindeutiger Wert im A-Scan zuordnen lässt. Dieser Wert im A-Scan wird dann auf den Wert im Voxel addiert.
Der Transmissionspuls wird nicht nur am Übergang zum Tumorgewebe, sondern teilweise auch im gesunden Gewebe reflektiert. Die an nicht beabsichtigten Stellen reflektierten Signale erscheinen im A-Scan als Rauschen. Zur Verringerung dieses Rauschens werden im A-Scan alle Werte unter einem bestimmten Wert abgeschnitten.
Wenn man also z. B. einen idealisierten A-Scan mit genau einem Ausschlag erstellt, erhält man ein einzelnes Ellipsoid (Abb. 6a). In einem Bild mit einem realistischen A-Scan, können die Störsignale teilweise sogar den gewollten Ausschlag überlagern. (Abb. 6b). Das Rauschen ist hier so stark, dass die Ellipse aus Abb. 6a nicht mehr zu erkennen ist.

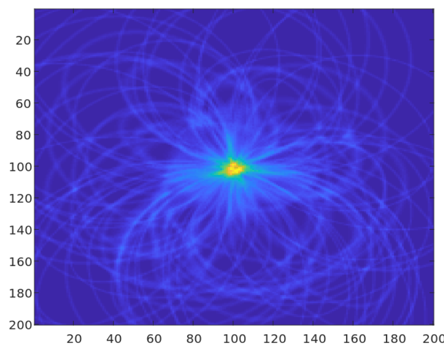


(a) Querschnitt durch ein Ellipsoid, berechnet durch die Eingabe einer Zeit.

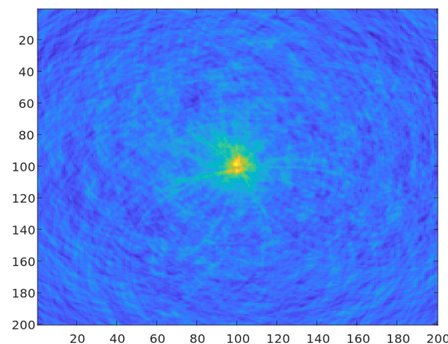


(b) Querschnitt durch viele Ellipsoide, berechnet durch die Eingabe einer Zeit, mit Rauschen im A-Scan.

Abbildung 6



(a) Querschnitt durch ein Ellipsoid, berechnet durch die Eingabe vieler Zeiten.



(b) Querschnitt durch viele Ellipsoide, berechnet durch die Eingabe vieler Zeiten, mit Rauschen im A-Scan.

Abbildung 7

Gibt man die Daten vieler A-Scans ein und ermittelt daraus Ellipsoide, schneiden sich diese z. B. am Übergang zwischen gesundem und Tumorgewebe, oder in einem bestimmten Punkt (Abb. 7a). Es ist also möglich, aus den A-Scans der Ultraschallwandler ein Bild zu rekonstruieren. Auf diese Weise lässt sich auch das Rauschen im A-Scan unterdrücken. Da das Rauschen größtenteils zufällig ist, die Spitzen in den A-Scans aber nicht, ist es trotzdem möglich den Punkt zu finden (Abb. 7b). Das Verhältnis zwischen Rauschen und Signal ist hier das gleiche, wie in Abb. 6b.
Insgesamt entstehen bei einer Untersuchung mit dem USCT-System 24492 A-Scans. Bei einer höheren Anzahl an A-Scans steigt der Berechnungsaufwand linear an. Verdoppelt man die Auflösung an den Achsen, so steigt der Aufwand für die Berechnung der Daten dagegen um das Achtfache an. In der aktuellen Software-Version kann ein Laptop mit einem Intel Core i5-6200u Prozessor und einer Taktfrequenz von 2,30 GHz ca. $2 \cdot 10^8$ Voxel pro Sekunde für einen Ellipsoid berechnen. Legt man eine Bildgröße von $1000 \cdot 1000 \cdot 1000$ Voxeln und die 157 Ultraschallwandler zugrunde, so ergibt sich eine Gesamtrechenzeit von ca.

$$\mathrm{t_{ges}} = \frac{10^9}{2^8 \cdot 1572\ \mathrm{s}} = 34,23\ \mathrm{h}.$$

1.6 SIMD

Eine moderne CPU besitzt mehrere Speicherzellen, sogenannte Register, in denen Daten gespeichert werden. Traditionell waren Register genauso breit, wie die Architektur der CPU. Das bedeutet, dass eine 64-bit CPU auch 64-bit breite Register besitzt. Bis zum Ende der 1990-er Jahre nutzten die damals üblichen Ein-Kern-Prozessoren der Firmen Intel und AMD die Technik „Single Instruction Single Data“, bei der für jede Rechenoperation ein Befehl notwendig ist. Wenn man also einen Satz an Daten, z. B. vier floats, verarbeiten will, ist es nötig, jeden einzelnen Wert einzeln zu verarbeiten (Abb. 8).

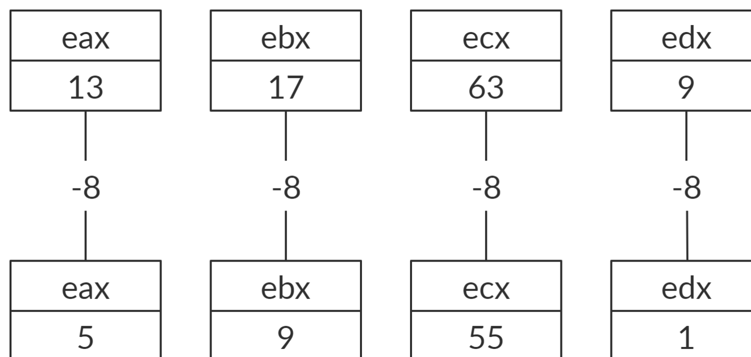


Abbildung 8: Subtraktion von vier floats mit x86-64-Instruktionen

Um die Berechnung großer Datenmengen zu vereinfachen, wurden Ende der 1990-er Jahre die ersten Prozessoren eingeführt, die „Same Instruction Multiple Data“ (SIMD) beherrschen. Der erste SIMD-Standard war Intels MMX von 1996, der in der Lage war, Rechenoperationen an bis zu acht 8-bit Integers gleichzeitig durchzuführen.[10] 1999 wurde MMX durch den neuen SSE-Standard ersetzt, dessen Fokus auf Fließkommazahlen liegt. Mit den neuen 128-bit breiten Registern können bis zu vier Fließkommazahlen gleichzeitig verarbeitet werden[11] (Abb. 9). Ein Codebeispiel, welches x86-64 und SSE vergleicht, findet sich im Anhang (Abb. 22).

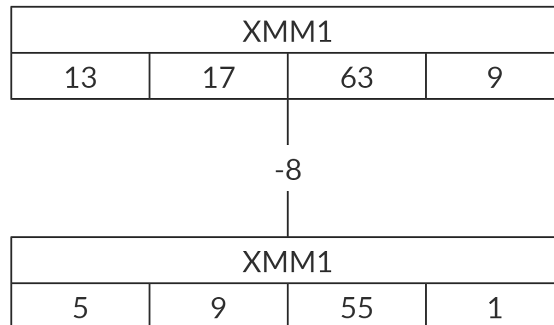


Abbildung 9: Subtraktion von vier floats mit SSE-Instruktionen

1.7 AVX

Anfang des Jahres 2011 erschienen mit Intels Sandy-Bridge-Prozessoren die ersten AVX-fähigen Prozessoren.[12] Im Gegensatz zu SSE besitzt AVX 16 256-bit breite Register. Dadurch ist es möglich, Rechenoperationen an bis zu 8 floats und damit doppelt so vielen wie bei SSE, durchzuführen. In bestimmten Fällen ist daher eine bis zu doppelt so schnelle Datenverarbeitung wie noch bei SSE möglich (Abb. 10). AVX wurde später mit der Markteinführung der Intel Haswell-Prozessoren um einige Befehle im neuen Befehlssatz AVX2 erweitert.[12] Die neueste Erweiterung des AVX-Befehlssatzes ist AVX-512. Dieser hat 32 512-bit breite Register, ist allerdings derzeit nur für High-End-CPUs verfügbar.[12]

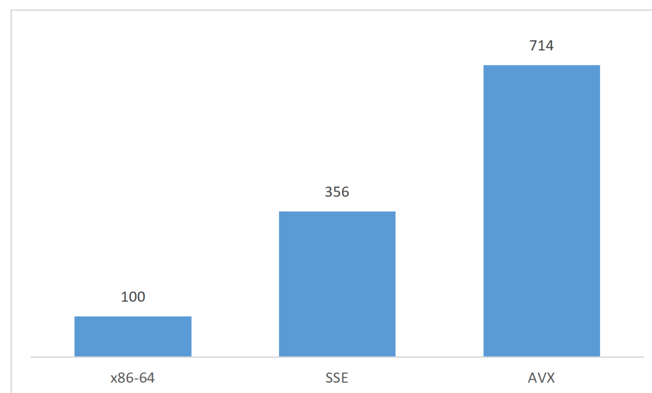


Abbildung 10: Relative Performance von AVX verglichen mit x86-64 und SSE bei der Berechnung einer Mandelbrot-Menge[13]

1.8 Aufgabenstellung

Bei der USCT fallen große Datenmengen an. Damit ein ausreichend genaues Bild entstehen kann, müssen möglichst viele Messergebnisse dem Algorithmus zugeführt werden. Das bedeutet, dass die Durchführungszeit enorm ansteigt. Für die praktische Anwendung des Verarbeitungsalgorithmus, muss die Berechnung in einer vertretbaren Zeit auf durchschnittlicher Computer-Hardware ablaufen können. Noch ist die Bilderstellung bei dem am IPE entwickelten Ultraschall-Computertomographen zu langsam. Ziel der Arbeit ist es, den vorhandenen, in SSE geschriebenen Code zur Bilderzeugung auf AVX zu adaptieren. So soll eine Beschleunigung der Berechnung erreicht werden.

2 Material und Methoden

2.1 Verwendete Systeme

Gearbeitet wurde auf folgenden Systemen:

System	MacBook Pro (mid 2014) (Intel)	Desktop PC (AMD)	Lenovo TP e650
Prozessor	Intel Core i7-4578U	Ryzen 5 1600	Core i5-6200U
Taktfrequenz in GHz	3,00	3,60	2,30
Kerne/Threads	2/4	6/12	2/4
L3-Cache in MB	4	32	3
Arbeitsspeicher in GB	16	16	8

Sofern nicht anders angegeben, sind alle Messdaten auf den ersten beiden System entstanden, die jeweils mit (Intel), oder (AMD) markiert wurden. Alle Prozessoren unterstützen folgende Befehlssätze: x86-64, SSE-SSE4.2, AVX, AVX2.

2.2 Der KIT SAFT-Code

Für die Arbeit wurde der vom IPE entwickelte, auf GitHub frei unter der Open-source Lizenz verfügbare,[14] originale SAFT-Code verwendet. Der Code funktioniert mit 32- und 64-bit Versionen von Linux und Windows. Er ist sowohl in C, als auch in SSE2-Assembler geschrieben. Beide Versionen sind in der Funktion vergleichbar und voll funktionsfähig.

2.3 MATLAB

MATLAB ist eine Entwicklungsumgebung mit dem Fokus auf Datenverarbeitung und Visualisierung. Nachdem MATLAB bereits verwendet wurde, um den ursprünglichen Code zu erstellen, wurde es auch hier eingesetzt.
Es ist möglich, C/C++ Programme direkt in MATLAB zu kompilieren, wobei eine MEX-Datei entsteht. Diese „MATLAB executable"(MEX) bietet eine Schnittstelle zwischen MATLAB und einem C/C++ Programm. Auf dieses kann so in MATLAB wie auf eine Funktion zugegriffen werden, was die Auswertung von Ergebnissen mit MATLAB erleichtert.
Eine weitere Funktion des Programms ist *repmat*. Damit kann man aus einem Array ein neues erstellen, welches beliebig viele Kopien des Ursprungsarrays enthält.
Mit Hilfe der Dokumentation des Originalcodes[14] wurde der Code in MATLAB kompiliert. Dabei wurden die Debug Flag, die Multithreading Flag und die Assembler Flag aktiviert.

Folgende Parameter werden im Programm verwendet:

Variable	Datentyp	Beschreibung
Ascan	Single(1:3000)	Messwerte (siehe Abschnitt 1.2)
pix_Vect	Single(1:3)	Vektor zur Startposition des Bildes im Raum
rec_pos	Single(1:3)	Position des Empfängers im Raum
send_pos	Single(1:3)	Position des Senders im Raum
speed	Single	Schallgeschwindigkeit
res	Single	Auflösung des Bildes (Abstand zwischen den Voxeln)
timeint	Single	Zeitlicher Abstand zwischen Werten im A-Scan
IMAGE_XYZ	UINT32(1:3)	Bildgröße in Voxeln in x-, y- und z-Dimension
IMAGE_SUM	Double(1:end)	zu beschreibendes Bild/ Eingangsbild

Ein A-Scan besteht immer aus genau 3000 einzelnen Messwerten. Wenn man als Test z. B. alle Werte im A-Scan auf null setzt und nur einen Wert auf eins, erhält man ein einzelnes Ellipsoid (vgl. Abb. 6a). Außerdem wurde, damit mögliche Optimierungen leichter erprobt werden können, der SAFT-Code in MATLAB nachprogrammiert. Der MATLAB-Code ist im Anhang zu finden (Abb. 23).

3 Architektur

3.1 Branching

Eine CPU arbeitet normalerweise alle Befehle im Assembler-Programm nacheinander in chronologischer Reihenfolge ab. Ein Loop oder eine *if*-Abfrage ist eine Abweichung von dieser Reihenfolge. Bei einem Loop springt die CPU eine bestimmte Anzahl an Instruktionen zurück. Bei einer *if*-Abfrage kann die CPU anschließend entweder den Code weiter durchlaufen oder nach vorne bzw. nach hinten springen. In Pseudocode lässt sich dies leicht darstellen:

```
b=1
Schleife:
    b+1
    wenn b gleich 10
        Springe zu Ende
    sonst
        Springe zu Schleife
Ende:
Ende des Programms
```

Ein Kern einer modernen CPU besitzt üblicherweise viele Recheneinheiten in mehrfacher Ausführung. Daher sind moderne Prozessoren in der Lage, mehrere Operationen gleichzeitig auszuführen, sofern sie nicht voneinander abhängig sind. Moderne CPUs führen in der Regel spekulativ die nächsten Instruktionen aus. Wenn Instruktionen voneinander abhängig sind, muss das Ergebnis der spekulativen Ausführung verworfen werden. Der Ausgang einer Branching-Operation (z. B. einer *if*-Abfrage) ist meist von dem Ausgang der vorangehenden Operationen abhängig. Wenn Branching oft auftritt, kann durch spekulative Ausführung demnach kein Vorteil in der Performance entstehen.

3.2 Pipelines

Die Berechnung der einzelnen Voxel, bzw. der Voxelgruppen ist voneinander unabhängig. Deshalb ist es möglich, durch geschicktes Programmieren, die CPU besser auszulasten. Da die Berechnung einer Voxelgruppe nur sieben der 16 Register beansprucht, kann parallel eine weitere Voxelgruppe berechnet werden. In der Theorie wird also derselbe Schritt zweimal hintereinander, aber mit unterschiedlichen Voxeln ausgeführt (Abb. 11). In der Praxis erfolgt die Berechnung oft gleichzeitig.

```
;PIPE1
vmovaps ymm1,ymm2
vshufps ymm1,ymm1,254    ;1111 1110b -> S1,S1,S1,S0,S3,S3,S3,S2
vaddps ymm2,ymm2,ymm1    ;t,t,P1,P0,t,t,P3,P2
;PIPE2
vmovaps ymm11,ymm9
vshufps ymm11, ymm11,254
vaddps ymm9,ymm9,ymm11   ;t,t,P5,P4,t,t,P7,P6
```

Abbildung 11: Assembler-Code mit Pipelines

4 Konvertierung des SSE-Codes zu AVX-Code

Da Assembler-Code schwer zu debuggen ist und bei einer Überführung in einen anderen Befehlssatz große Teile des Codes neu geschrieben werden müssen, ist es sinnvoll, in möglichst kleinen Schritten vorzugehen.

4.1 Kombinieren beider Pipelines

In jeder der beiden Pipelines werden im SSE-Code jeweils zwei Voxel berechnet. Ein SSE-Register kann genau vier floats speichern. Es müssen jedoch von jedem Voxel zuerst der Abstand zum Empfänger (Px_R) und zum Sender (Px_S) berechnet werden. Erst dann können beide Werte addiert werden. Ein SSE-Register ist also wie folgt aufgeteilt:

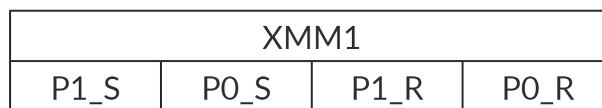


Abbildung 12: Die Abstände zweier Voxel zu Sender und Empfänger im SSE-Register.

In einem AVX-Register können bis zu acht floats gespeichert werden, z. B. nach folgendem Schema:

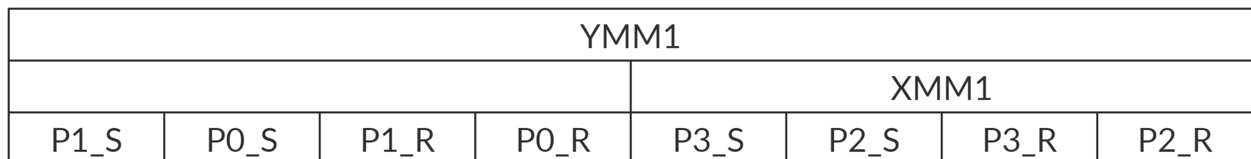


Abbildung 13: Die Abstände von vier Voxeln zu Sender und Empfänger im AVX-Register.

Eine Besonderheit der AVX-Register ist, dass sie nicht komplett neue Register sind, sondern eine Erweiterung der bestehenden SSE-Register (Abb. 13). Das bedeutet, dass es zu jedem AVX-Register ein SSE-Register gibt, welches die unteren 128-bit, also die untere Hälfte des AVX-Registers, ausmacht. Das kann man beim Konvertieren des Codes nutzen.
Im Prinzip besteht der Code aus zwei wichtigen Bestandteilen: Zum einen der Setup-Code, welcher die Register befüllt und initialisiert, zum anderen der zweite Teil, der die tatsächliche Berechnung der Voxel durchführt. Um eine einfache Konvertierung zu ermöglichen, wurde zunächst nur der Berechnungs-Code verändert.
Der Setup-Code initialisiert zwei SSE-Register, die später für die beiden Pipelines verwendet werden. Da ein AVX-Register genau doppelt so breit ist wie ein SSE-Register und dieses die untere Hälfte eines AVX-Registers ausmacht, kann man das zweite SSE-Register, welches vom Setup-Code initialisiert wird, in die obere Hälfte des ersten Registers einsetzen. So erhält man ein volles AVX-Register (Abb. 14). Man verliert jedoch die zweite Pipeline. Die Zahl der berechneten Voxel pro Loop wird nicht erhöht.

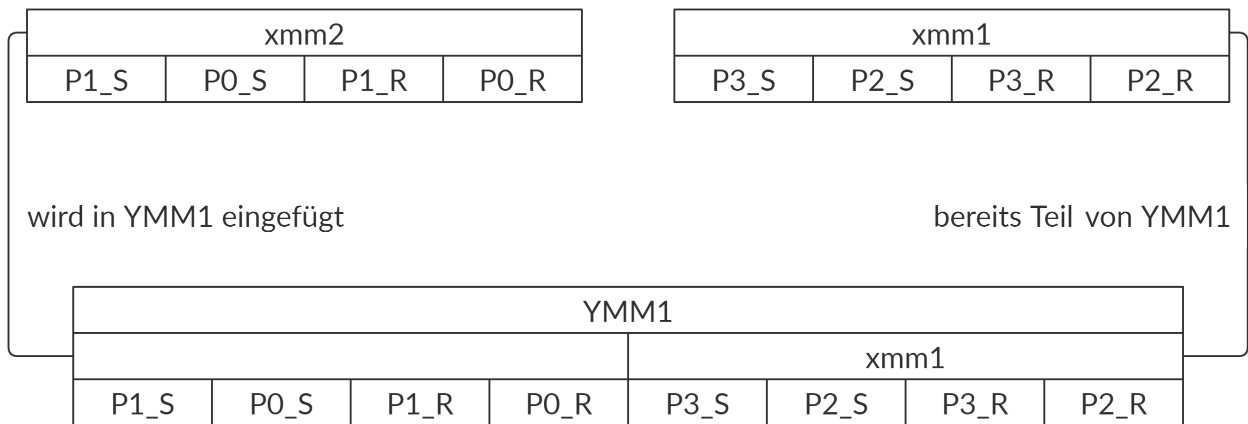


Abbildung 14: Initialisierung eines AVX-Registers aus zwei SSE-Registern

4.2 Wiedereinführung der zweiten Pipeline

Nachdem die Berechnung von 8 Voxeln in einer Pipeline erfolgreich war, wurde erneut eine zweite Pipeline eingeführt. Aufgrund der nun auf 16 erhöhten Zahl an Voxeln, die mit jedem Loop berechnet werden, musste auch der Setup-Code, der die Voxel aus dem Speicher lädt, sowie der nachfolgende Code, welcher die Voxel anschließend wieder abspeichert, verändert werden. Der Code, der nach diesen beiden Schritten entstanden ist, kann im Anhang gefunden werden (Abb. 9).

5 Profiling

Zur Erprobung verschiedener Optimierungsmethoden wurde der KIT-SAFT-Code in MATLAB nachprogrammiert. Dabei wurden z. B. die Auswirkungen bei Verwendung verschiedener Datentypen auf die Performance gemessen. Hierzu wurde eine Profiling-Umgebung in MATLAB geschaffen, die beliebig viele verschiedene Versionen nach Geschwindigkeit und Unterschieden im erstellten Bild auf verschiedenen Bildgrößen vergleicht.

5.1 Version 1

Im Programmkopf der ersten Version wird zuerst der Workspace (MATLAB-Variablen-Speicher, der auch nach Programmende bestehen bleibt) geleert, damit keine Fehler durch mit falschen Werten besetzten Variablen entstehen. Danach werden Variablen initialisiert, die einerseits das Verhalten der darauffolgenden Loops regulieren, sowie Variablen, die in die zu vergleichenden Versionen übergeben werden.
Im Hauptteil, zwei *for*-Loops, erfolgt die rechenaufwendige Arbeit. Der äußere Loop ist für eine wiederholte Durchführung der Versuchsreihe zuständig, damit am Ende die gemessenen Zeiten gemittelt werden können, um ein genaueres Ergebnis zu erhalten. Der innere Loop geht verschiedene Bildgrößen (Kantenlänge des Bildes in Voxeln) durch. Im Inneren werden die unterschiedlich optimierten Programme, die zu aufrufbaren Funktionen umgeschrieben wurden, mit den anfangs initialisierten Variablen, hier in Version 1: Bildgröße und Anzahl an Ellipsoiden, aufgerufen. Die Funktionen geben die benötigte Zeit zur Berechnung des Bildes zurück, die in einem dreidimensionalem Array gespeichert werden. Wobei die erste Dimension die steigenden Bildgrößen, die Zweite die Wiederholungen für die Mittelungen und die Dritte die unterschiedlichen Programme beschreibt. Nach den Loops können mit den gespeicherten Zeiten verschiedene Graphen erstellt und anschließend analysiert werden (Abschnitt 6).

5.2 Version 2 – Probleme mit Abstürzen

In Version 2 wurde ein sogenannter *try-catch* eingebaut, damit bei möglichen Abstürzen der Unterfunktionen nicht das gesamte Programm beendet wird. Außerdem wurde nach jeder Bildgröße der Workspace gespeichert, da ab zu großen Bildern der verfügbare Arbeitsspeicher nicht aussreichte (Out-of-memory) und es zu Problemen mit Swapping kam. Swapping ist eine Funktion vieler neuer Betriebssysteme. Diese lagern Daten aus dem schnellen Arbeitsspeicher auf langsamere, aber größere Datenspeicher, wie z. B. eine Festplatte aus, wenn der Arbeitsspeicher zu voll wird. Bei der Durchführung der optimierten Programme mit wenigen Loops und großen Bildgrößen müssen viele Daten im RAM abgelegt werden. Bei dem AMD-System hatte die Auslagerung auf die Festplatte zu einem sehr instabilen System oder sogar einen kompletten Systemabstürz geführt. Dieses Problem konnte bislang noch nicht behoben werden.

5.3 Version 3 – Datengenerierung

Normalerweise werden die A-Scans mit den zugehörigen Sender- und Empfängerpositionen gemessen und gemeinsam an das Bildgenerierungsprogramm übergeben. Dabei handelt es sich um mehrere Gigabytes an Daten. Beim Profiling der Programme wäre ein ständiges Laden dieser großen Datenmengen sehr zeitintensiv und unhandlich. Deswegen wurden in den vorherigen Versionen simulierte A-Scans, Sender- und Empfängerpositionen von den einzelnen Unterprogrammen generiert. In Version 3 wurde die Generierung in die Profilingumgebung verschoben. Dies garantiert zum einen, dass alle Unterprogramme mit den gleichen Daten arbeiten und zum anderen spart es Rechenzeit. Anstatt dass alle Unterprogramme einzeln simulierte Daten generieren, werden nun nur einmal pro Bildgröße Daten erzeugt und diese anschließend an alle Unterprogramme übergeben.
Ebenfalls wurde ein Funktionspointer implementiert, der es ermöglicht, mehrere Unterfunktionen, hier die unterschiedlich optimierten Programme, in einem Array zu verlinken, um später durch diese zu iterieren. Die erwies sich als sehr praktisch da viele Unterfunktionen mit den gleichen Parametern ausgeführt werden. So können dem Profiling-Programm einfach Programme hinzugefügt oder entfernt werden.

5.4 Version 4 – KIT-SAFT-Code und Bildunterschiede

Zum Abgleich der erzielten Bildqualität und Performance wurden die erstellten MATLAB-Programme mit dem vorhandenen KIT-SAFT-Code verglichen. Dieser Code bietet einige Konfigurationsmöglichkeiten: Es besteht die Wahl zwischen einer Implementierung in C oder in Assembler zur Bildgenerierung. Außerdem gibt es die Option, Multithreading und Debug-Ausgaben zu aktivieren bzw. deaktivieren. Die Konfigurationen sind als Boolean-Flags im C-Code realisiert.
Der bestehende KIT-SAFT-Code muss zunächst in MATLAB kompiliert werden, um dort funktionsfähig zu sein und der Assembler-Teil des Codes muss assembliert werden. Dies wird mit nasm durchgeführt. Durch die Kompilierung sind die eingestellten Konfigurationen fest und unveränderlich. Für eine neue Einstellung ist eine erneute Kompilierung erforderlich. Ebenfalls kann man die Anzahl an verwendeten Cores einstellen, dies erfolgt jedoch als ausführbarer Befehl mit den MEX-Programmen. Für das Profiling bedeutet dies, dass man alle unterschiedlichen Konfigurationen kompiliert und anschließend dem Funktionspointer hinzufügt. Die Eingabeparameter der MEX-Programme stimmen größtenteils mit denen der MATLAB-Programme überein, dennoch sind kleine Anpassungen in der Reihenfolge notwendig.

Bei Betrachtung der generierten Bilder fielen kleine Unterschiede auf. Um diese genauer zu analysieren wurde ein neuer Abschnitt implementiert, der die verschiedenen Unterprogramme auf einer festen Bildgröße vergleicht. Dafür werden die generierten Bilder in einem vierdimensionalen Array gespeichert, wobei ein einzelnes Bild drei Dimensionen benötigt und in die vierte Dimensionen können andere Bilder dahinter gestapelt gespeichert werden. Jedes Bild kann nun mit jedem verglichen werden, indem man diese miteinander subtrahiert. Voxel die im gebildeten Differenzbild (siehe Abb. 19) einen Wert ungleich Null aufweisen, zeigen einen Unterschied zwischen den zwei Bildern an.

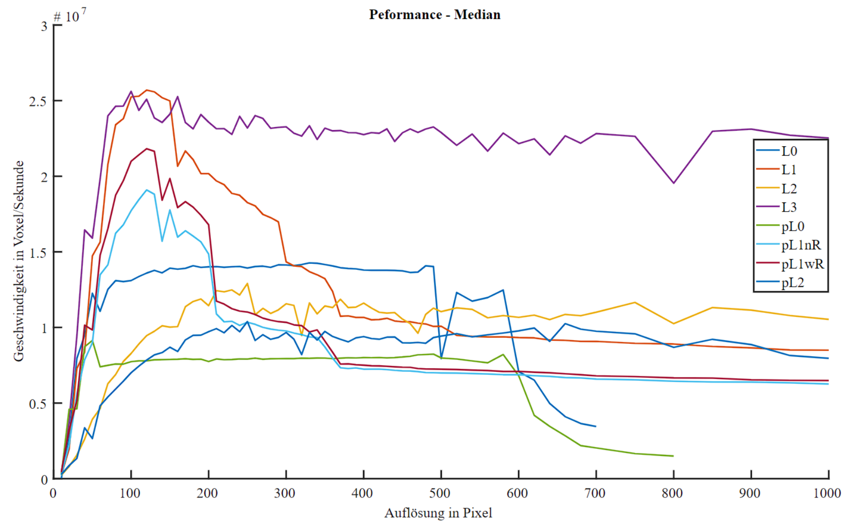


Abbildung 15: Beispiel-Graph mit Performance verschiedener MATLAB-Code-Versionen abhängig von der Bildgröße (Intel)

Der Graph zeigt acht unterschiedliche MATLAB-Codes die in der angelegten Profilingumgebung verglichen wurden. Die y-Achse zeigt die ansteigende Kantenlänge der Bilder in Pixel. Die Ordinatenachse repräsentiert die Geschwindigkeit in Voxel pro Sekunde. Mögliche Erklärung für das jeweilige Verhalten folgen in Abschnitt 6.

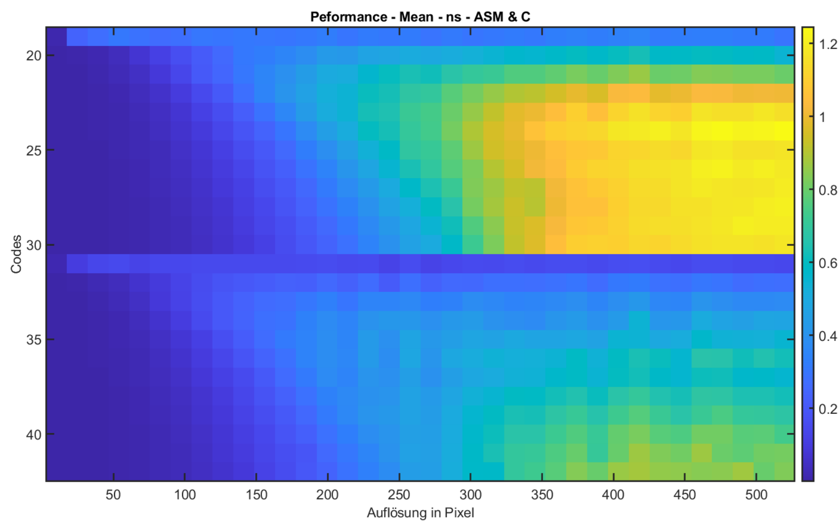


Abbildung 16: Zweidimensionaler-Beispiel-Graph mit unterschiedlichen Einstellungen des CPU-SAFT-Codes (AMD)

Die zweidimensonale Ansicht zeigt die verschiedenen Einstellungen des CPU-SAFT-Codes als Zeilen und die ansteigende Bildgröße als Spalten. Die Nummerierungen an der Ordinatenachse steht jeweils für eine Einstellung, sie beginnt bei 19 da diese Darstellung nur ein Teilausschnitt des durchgeführten Profiling darstellt. Der Farbwert einer bestimmten Zelle zeigt die Performance in Nanosekunden pro Voxel an. Von Zeile 19 bis 30 wurde die Assembler-Implementierung mit ansteigender Core-Anzahl für Multithreading eingestellt. Zeile 31 bis 42 zeigt die C-Implementierung. Man erkennt das die Assmebler-Implementierung mit vielen Multithreading Cores bei großen Bildern erheblich schneller arbeitet als die C-Implementierung. Ebenfalls zeigt sich das die Performance ab einer bestimmten Core-Anzahl nur sehr leicht oder nicht steigt.

6 Ergebnisse und Analyse

6.1 Cache-Limitierung

Ein limitierender Faktor kann die Cachegröße sein. Der Cache befindet sich direkt in der CPU und kann vorab Daten aus langsameren Speichern wie RAM oder Festplatte laden, damit die CPU-Cores nicht auf zu berechnende Daten warten müssen. Dieses Merkmal kann bei ansteigenden Bildgrößen in Verbindung mit zu wenigen Loops problematisch werden, da hier nicht mehr alle Voxel, die parallel berechnet werden sollen, in den Cache passen. Die Voxel werden in den RAM ausgelagert und es kommt zu Verzögerungen.

6.2 Optimierungsmöglichkeiten

6.2.1 Dritte Pipeline

Einige der 16 AVX-Register bleiben auch bei der Nutzung zweier Pipelines ungenutzt. Daher wäre es möglich, eine dritte Pipeline in das Programm einzubauen. Diese würde bei einigen Architekturen mit besonders vielen Recheneinheiten pro Kern, eine erhebliche Steigerung der Effizienz bewirken.

6.2.2 Single- vs. Double-precision floating-point-Zahlen

In vielen Programmiersprachen hat man die Wahl zwischen einer Single-precision floating-point Zahl (Gleitkommazahl mit einfacher Genauigkeit), die standardmäßig 32 Bit an Speicher einnimmt, und einer Double-precision floating-point Zahl (Gleitkommazahl mit doppelter Genauigkeit), die standardmäßig 64 Bit benötigt (Abb. 17). Mit beiden Datentypen kann man die gleichen Instruktionen durchführen, der Unterschied besteht in der Größe und damit verbunden die maximal mögliche Genauigkeit der Zahlenwerte.

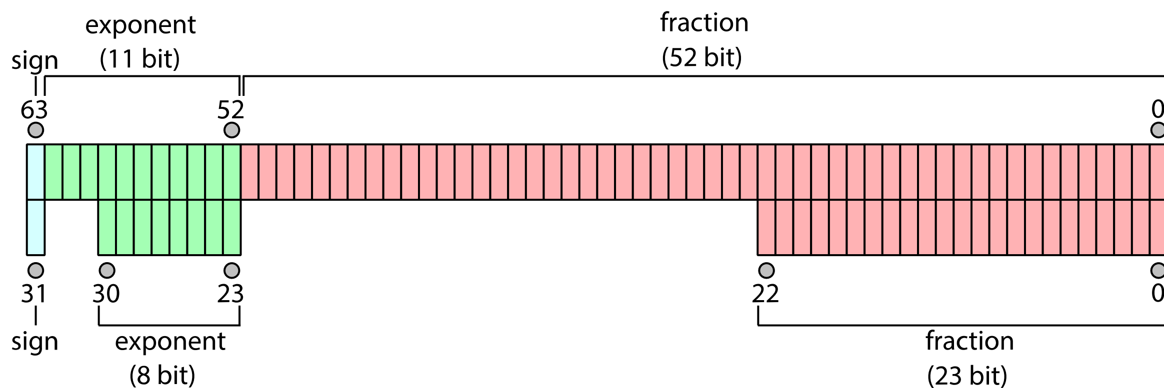


Abbildung 17: Double-[15] vs. Single-precision floating-point-Zahlen[16]

Vergleich der benötigten Bits:

	Single-precision	Double-precision floating-point-Zahlen
Sign Bits	1	1
Exponent Bits	8	11
Fraction Bits	23	53

Die Double-precision floating-point-Zahl bietet eine weit höhere Genauigkeit. Dies bedeutet für den Prozessor mehr Rechenaufwand und nimmt mehr Zeit in Anspruch. Beim Profiling von Versionen, die einen der beiden Datentypen benutzen, zeigt sich zwischen der Verwendung von Singles und Doubles ein deutlicher Unterschied (Abb. 18). Das Programm, das ausschließlich Singles verwendet, arbeitet fast doppelt so schnell.

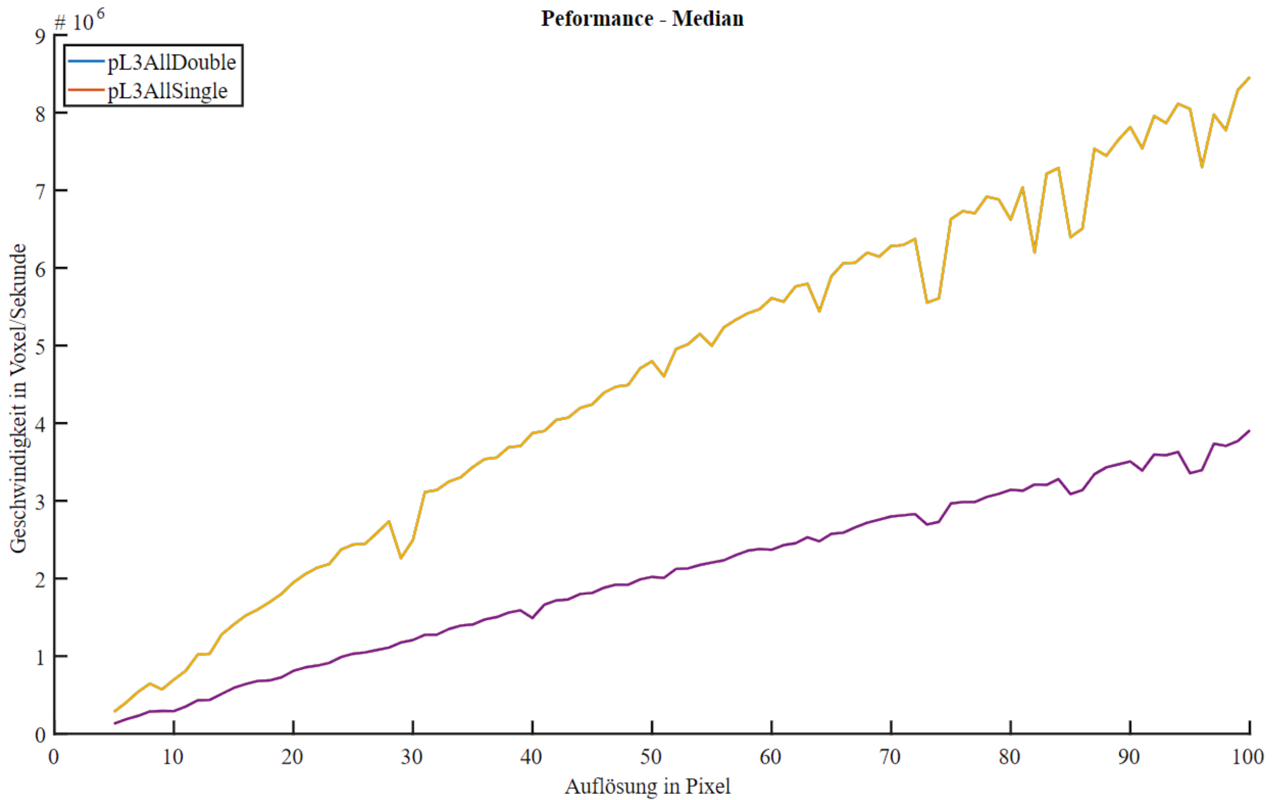


Abbildung 18: Peformancevergleich (Intel): MATLAB-Code mit Singles (gelb) und mit Doubles (lila)

Ebenfalls fallen kleine Bildunterschiede auf, die durch die ungenauere Berechnung mit den Singles im Vergleich zu den Doubles auftreten können (Abb. 19). Diese Abweichungen sind so gering, dass sie selbst in der medizinischen Diagnostik keine Relevanz haben. Daher werden in der optimierten Version aufgrund des deutlichen Vorteils in der Performance Single-precision floating-point-Zahlen eingesetzt.

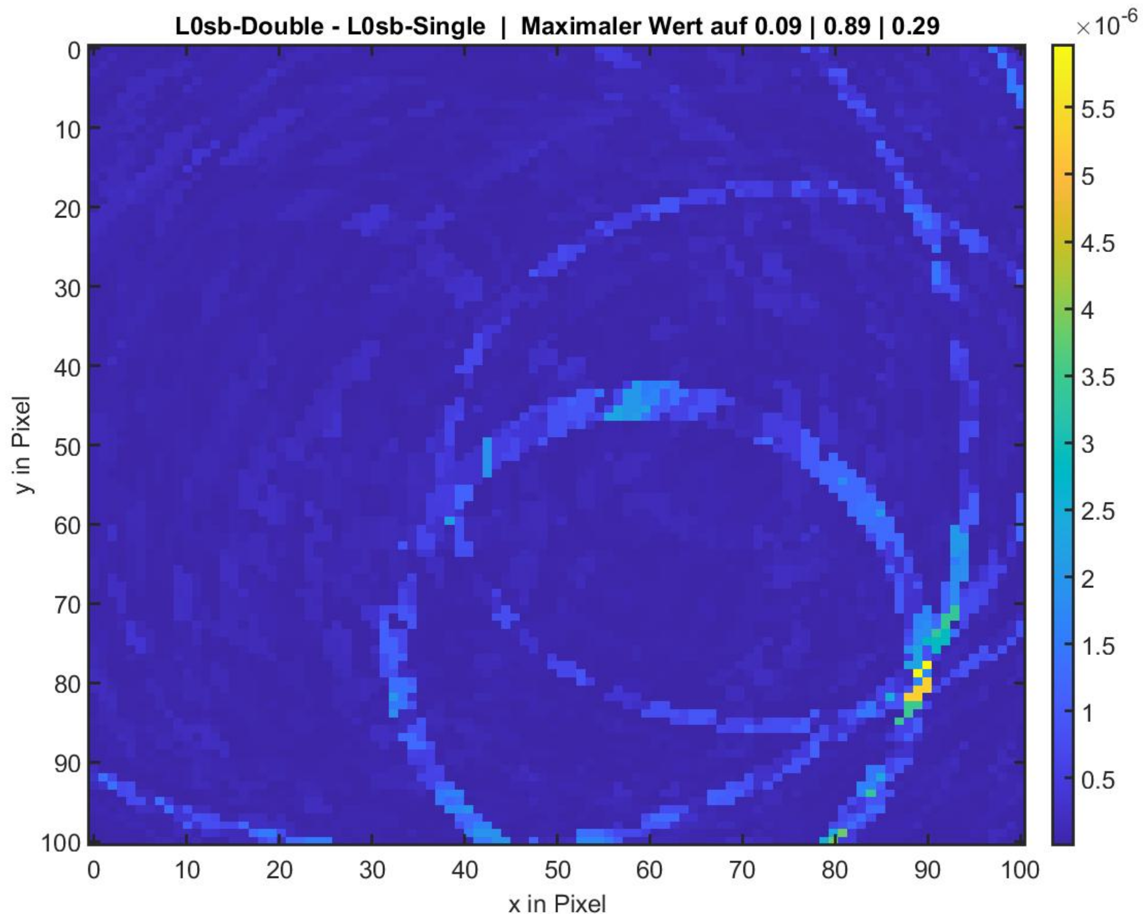


Abbildung 19: Bildunterschiede zwischen Matlab-Code einmal mit Single- und das andere mal mit Double-Parametern (AMD)

Die Graphen-Überschrift zeigt zum einen welche zwei Programme verglichen werden und zum anderen die Position (in Meter) des maximalen Wert der im Bild, hier Differenzbild, vorhanden ist. Rechts befindet sich die Farbpalette die den unterschiedlichen Farben ihre Werte zuordnet. Die $\times 10^{-6}$ darüber beschreibt einen Faktor der noch mit den dargestellten Werten berechnet werden muss.

6.2.3 repmat

MATLAB erlaubt, ähnlich wie SIMD, Rechenoperationen mit Arrays, die jedoch von beliebiger Größe sein können. Anstatt jeden Voxel einzeln zu berechnen, kann man mit Hilfe der *repmat*-Funktion Arrays aus allen wichtigen Variablen erstellen und so z. B. alle Voxel in einer Dimension, also eine ganze Zeile, berechnen. So muss der Code weniger oft geloopt werden. Da weniger Loops in der Regel auch weniger Branching erfordern, ist eine verbesserte Performance zu erwarten. Die Ausgabe des Codes ist ein 3-dimensionales Bild; es gibt demnach drei Loops, die potenziell entfernt werden können.

6.2.4 Weniger Loops in MATLAB

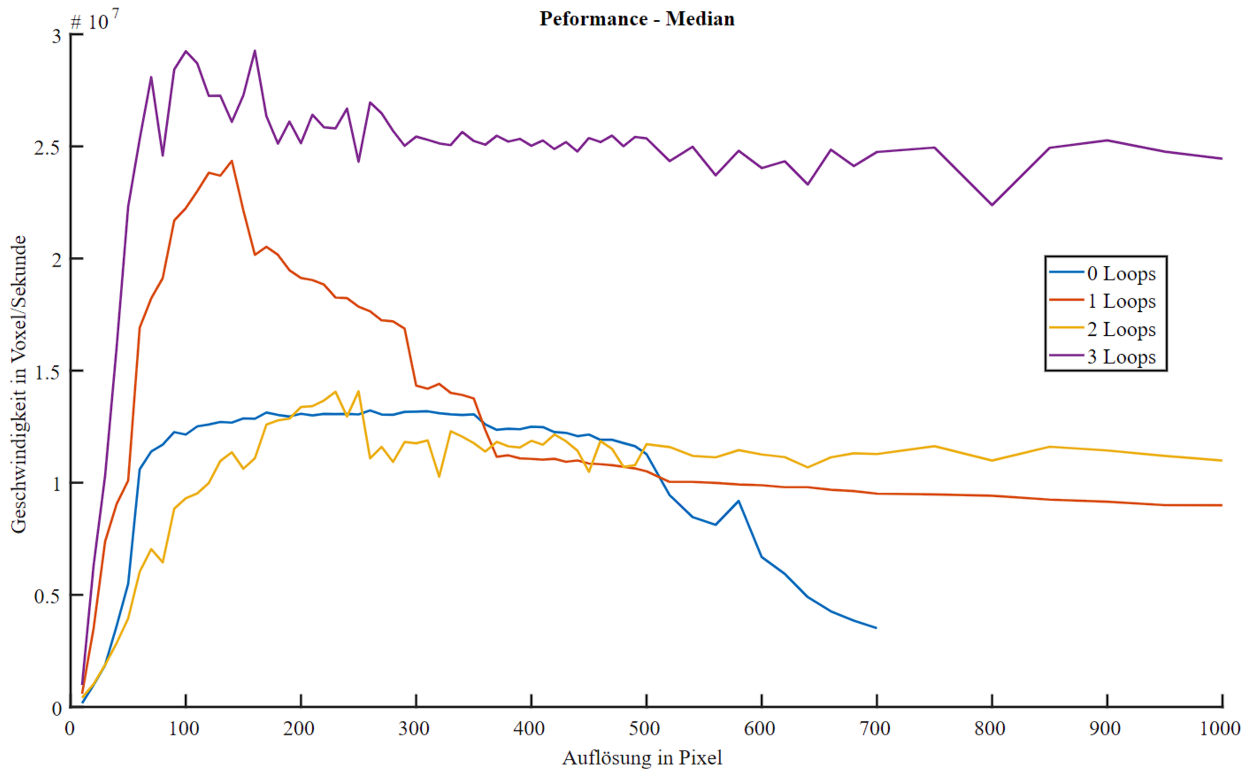


Abbildung 20: Peformancevergleich: MATLAB-Codes mit unterschiedlich vielen Loops (Intel)

Es wurden vier verschiedene Codes verglichen. Der originale MATLAB-Code mit drei Loops, und für jeden weiteren Code wurde eine Loop entfernt. Entgegen der Erwartung steigt die Performance durch die Reduzierung der Loops nicht – im Gegenteil: Sie bricht teilweise stark ein. Dies ist damit zu begründen, dass der Compiler von MATLAB mittlerweile selbstständig SIMD-Instruktionen verwendet. Der Overhead, der durch die *repmat*-Funktion entsteht, tritt deshalb in den Vordergrund. Zudem gibt es eine weitere Limitation: Bei der Implementierung mit null Loops sind alle Variablen 3-dimensional, da das gesamte Bild auf einmal berechnet wird. Je höher die Auflösung wird, desto größer werden diese Variablen. Am Ende übersteigt die Datenmenge den zur Verfügung stehenden Arbeitsspeicher sowie die Auslagerungsdatei. Es kommt zu einem Programmabsturz.

6.3 Instruktionen-Abwägung

AVX-Instruktionen benötigen im Vergleich zu SSE-Instruktionen häufig mehr Takte, um mit den breiteren Registern zu arbeiten, wie nachfolgender Vergleich für einen einfachen Befehl, mit dem Daten aus einem Register verschoben werden, zeigt:

SSE- gegen AVX-Äquivalent:[17]

Instruction	Operands	Ops	Latency	Reciprocal throughput
MOVAPS/D (SSE)	x,x	1	0	0.25
VMOVAPS/D (AVX)	y,y	2	2	0.50

Begriffserklärung:

Operands	Entweder x für 128 Bit SSE-Register oder y für 256 Bit AVX-Register
Ops	Anzahl Mikrooperationen
Latency	Verzögerung der Teilberechnungen bei Pipelines
Reciprocal throughput	wichtig für Pipelines

Durch Verwendung der AVX-Instruktion kann die doppelte Menge an Daten verschoben werden, es werden aber auch doppelt so viele Taktzyklen benötigt. Somit konnte durch die AVX-Implementierung keine Zeit gewonnen werden. Um eine mögliche Steigerung der Performance zu erzielen, sollte man daher nach neuen Instruktionen mit neuen Funktionen suchen, und nicht nur die SSE-Instruktionen mit AVX-Äquivalenten austauschen.

6.4 AVX-Implementierung

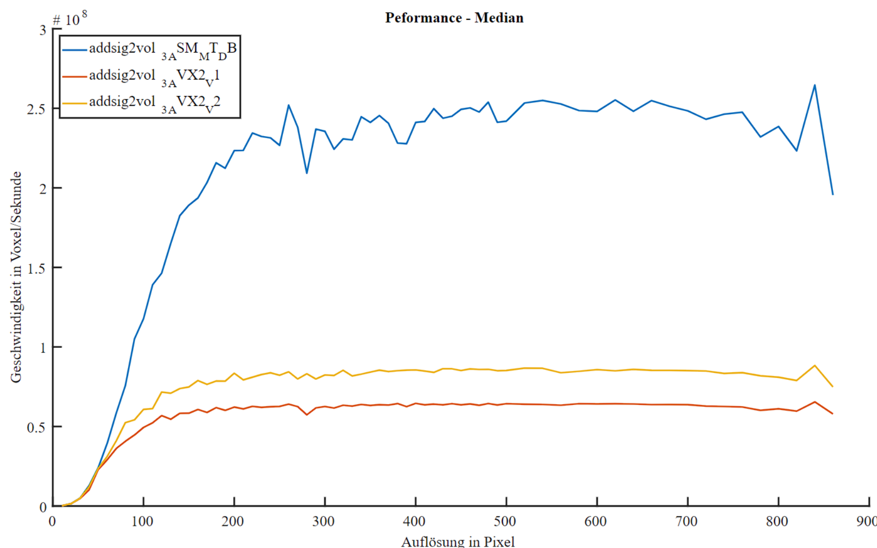


Abbildung 21: Performancevergleich (Intel): SSE-Code (blau), AVX-Code mit einer (rot) und zwei (gelb) Pipelines

Version 1 des AVX-Code (rot) arbeitetet mit ausgetauschten AVX-Instruktionen ohne zweite Pipeline; diese wurde in Version 2 (gelb) implementiert. Die zweite Pipeline verbessert die Performance um fast 50 %. Beide AVX-Versionen erreichen dennoch nicht die Performance des vorhandenen SSE-Codes.

7 Fazit und Ausblick

In der aktuellen Version wurden SSE-Befehle mit AVX-Äquivalenten ausgetauscht, ebenfalls wurden die breiteren Register genutzt. Dennoch konnte bislang keine Steigerung in der Performance erzielt werden (Abb. 21). Das neue AVX-Instruktionsset wurde erfolgreich eingebaut, wobei noch nicht alle bestehenden Optionen ausgetestet wurden. Um den AVX-Prototyp zukünftig effektiver gegenüber den bestehenden, bereits stark optimierten, SSE-Code zu machen, sind weitere Optimierungen und Tests notwendig.
Die meisten AVX-Instruktionen arbeiten mit breiteren Registern, was gegenüber SSE-Instruktionen den Vorteil bringt, mit mehreren Zahlen gleichzeitig rechnen zu können. Zu beachten ist aber dass die AVX-Instruktionen in vielen Fällen gerade dann, wenn sie auch mit SSE-Registern kompatibel sind, mehr Takte benötigen, um mit den breiteren Registern zu arbeiten. Daraus erfolgt keine Zeitersparnis. Die Nutzung von neu eingeführten AVX-Instruktionen in effizient strukturierten Abläufen kann eine Beschleunigung möglich machen. Außerdem sind die unterschiedlichen Prozessor-Architekturen zu beachten, je nach Modell und Hersteller können diese variieren und bringen damit unterschiedliche Realisierungen der AVX-Instruktionen. Folglich bedeutet das eine Optimierung mit einem Prozessor X, nicht unbedingt eine Beschleunigung für andere Prozessoren mit sich bringt. Hier wären spezifische Implementierungen für verschiedene Prozessoren möglich.
Mit Hilfe der zahlreichen und ausführlichen Veröffentlichungen und Analysen zu den AVX-Instruktionen, sollte es möglich sein, den Code weiter zu optimieren.
Der nächste Optimierungsschritt könnte bereits mit der AVX-512 Implementierung geschehen, ein weiteres neues Instruktionsset mit 512 Bit breiten Registern, das aber derzeit nur in modernen High-End-Prozessoren benutzt werden kann.
So könnten zukünftig die bei USCT großen anfallenden Daten in einer vertretbaren Zeit verarbeitet und dargestellt werden und damit eine schonende Alternative zur Mammografie bieten.

8 Danksagung

Unser großer Dank geht an Michael Zapf vom Institut für Prozessdatenverarbeitung und Elektronik (IPE) am Karlsruher Institut für Technologie, der uns diese Arbeit ermöglicht und uns bei inhaltlichen Problemen unterstützt hat. Des Weiteren bedanken wir uns bei unseren Betreuern vom Hector-Seminar Anke Richert, Thomas Herrmann und Paul Bischof für die große Unterstützung über die vergangenen Jahre. Nicht zuletzt danken wir dem Ehepaar Josephine und Dr. Hans-Werner Hector und ihrer Stiftung für die großzügige Unterstützung des Hector-Seminars.

9 Anhang

```
                                                      ;benötigte Taktzyklen:
mov r8d, 8                                                  ;0.25
mov eax, [rbp]           ;eax = 13                    ;0.5
mov ebx, [rbp+8]         ;ebx = 17                    ;0.5
mov ecx, [rbp+16]        ;ecx = 63                    ;0.5
mov edx, [rbp+24]        ;edx = 9                     ;0.5

sub eax, r8d             ;eax = 5                     ;0.25
sub ebx, r8d             ;ebx = 9                     ;0.25
sub ecx, r8d             ;ecx = 55                    ;0.25
sub edx, r8d             ;edx = 1                     ;0.25
                                                      ;gesamt = 3.25


movups xmm1, [rbp]       ;xmm1 = 13; 17; 63; 9        ;0.5
movss xmm2, 8            ;xmm2 = -; -; -; 8           ;0.5
shufps xmm2, xmm2, 0     ;xmm2 = 8; 8; 8; 8           ;0.5
subps xmm1, xmm2         ;xmm1 = 5; 9; 54; 1                 ;0.5
                                                      ;gesamt = 2
```

Abbildung 22: Vergleich der benötigten Taktzyklen, um vier Werte zu subtrahieren (x86-64 vs. SSE)

```
function data = L3sb(res, count, Ascans, speed, int, posSens, posRecs, data, img_start, resint)
  for i=1:count
    ascan = Ascans(:,i);
    sendpos = posSens(i,:);
    recpos = posRecs(i,:);
    pixpos = single(zeros(3,1));
    for z=1:res
      pixpos(3) = img_start(3) + z.*resint;
      for x=1:res
        pixpos(1) = img_start(1) + x.*resint;
        for y=1:res
          pixpos(2) = img_start(2) + y.*resint;
          d_send_pix = sqrt((sendpos(1)-pixpos(1)).^2+(sendpos(2)-pixpos(2)).^2+(sendpos(3)-pixpos(3)).^2);
          d_rec_pix = sqrt((recpos(1)-pixpos(1)).^2+(recpos(2)-pixpos(2)).^2+(recpos(3)-pixpos(3)).^2);
          dges = (d_send_pix + d_rec_pix);
          ascanpos = round((dges/speed)/int);
          data(x,y,z) = data(x,y,z) + ascan(ascanpos);
        end
      end
    end
  end
end
```

Abbildung 23: MATLAB Code zum Erstellen von Bildern aus Messwerten (Ascans)

Aktuelle AVX-Version (nur veränderte Abschnitte abgedruckt):

```
   as2v_complex:
_as2v_complex:


        push rbp
        push rsi
        push rdi
        push rbx         ;+12 (diff:esp+8)

        ;;;save rsp
        ;mov [rel RSP_SAVE],qword rsp
        mov rsi, rsp

        ;;;save XMM6-7 (callee convention WIN64(?))
        ;movups [rel XMM6_SAVE],xmm6
        ;movups [rel XMM7_SAVE],xmm7

        ;transfer via rcx because first parameter on MS-64 convention and 4.
        ;on linux,so pass the pointer as 1. & 4.
        mov rsp,rcx

  ;;Save rsp
  mov [rsp+136], rsi


         fninit                  ;init fpu


    ;init mm7 with zero
    ;mov eax,0           ;
    ;cvtsi2ss xmm7,eax   ; xmm7 = 0
    ;shufps   xmm7,xmm7,0; 0,0,0,0
    ;CVTPS2PI mm7,xmm7   ; mm7  = 0

    ;pixel_pos
    mov rdx,[rsp+32]         ;*pixelpos
    movss  xmm7,[rdx+8]  ;pixelpos      0,0,0,Z
    shufps xmm7,xmm7,0         ;pixelpos      Z,Z,Z,Z
    movlps xmm7,[rdx]         ;pixelpos      Z(TRASH),Z,Y,X

    ;rec_pos
    mov rdx,[rsp+48]         ;*receiverpos
    movss  xmm6,[rdx+8]  ;receiverpos   0,0,0,Z
    shufps xmm6,xmm6,0         ;receiverpos   Z,Z,Z,Z
    movlps xmm6,[rdx]         ;receiverpos   Z(TRASH),Z,Y,X

    ;sender_pos
    mov rdx,[rsp+56]         ;*senderpos
    movss  xmm5,[rdx+8]  ;senderpos     0,0,0,Z
    shufps xmm5,xmm5,0         ;senderpos     Z,Z,Z,Z
    movlps xmm5,[rdx]         ;senderpos     Z(TRASH),Z,Y,X

    ;resolut
    mov rdx,[rsp+72]         ;*resolut
    movss xmm4,[rdx]         ;t,t,t,resolut

    ;factor = 1/(speed*timeinterval)
    mov eax,5                 ; 1 for interp_ratio 1 -> 5 for actual !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
cvtsi2ss xmm3,eax         ;xmm3=1
mov rdx,[rsp+64]         ;*speed
movss xmm2,[rdx]
mov rdx,[rsp+80]         ;*timeinterval
movss xmm1,[rdx]
mulss xmm2,xmm1
divss xmm3,xmm2         ;xmm3 = 1/(speed*timeinterval)
unpcklps xmm4,xmm3         ; t,t,factor,resolut

;interleave in xmm4
mov eax,-1                 ;4294967295
cvtsi2ss xmm3,eax         ;xmm3 = 1

;fixed interpol factor
mov eax,5
cvtsi2ss xmm1,eax         ;xmm1 = 5

mov eax,[rsp+120]         ;n_Ascan
cvtsi2ss xmm2,eax         ;xmm2 = 3000

mulss xmm2,xmm1         ;3000*5 ->15000
cvtss2si eax,xmm2;
mov dword [rsp+128],eax ;15000
unpcklps xmm3,xmm2         ;xmm3 = t,t,15000,-1
shufps xmm4,xmm3,68  ;0100 0100b -> xmm4 = 15000,-1,factor,resolut

;;;create complete factor buffer
movaps xmm8,xmm4
      shufps xmm8,xmm8,85  ;    0101 0101b -> factor,factor,factor,factor
    vinsertf128 ymm8,ymm8,xmm8,1 ; -> factor,factor,factor,factor,factor,factor,factor,factor

;;; create 2*resolut buffer
movss  xmm10,xmm4    ;    t,t,t,resolut
shufps xmm10,xmm10,0 ;    0000 0000b -> resolut,resolut,reolut,resolut
addps xmm10,xmm10    ;    2*resolut,2*resolut,2*resolut,2*resolut
    vinsertf128 ymm10,ymm10,xmm10,1
    ; -> 2*resolut,2*resolut,2*resolut,2*resolut,2*resolut,2*resolut,2*resolut,2*resolut

;;fill up counters
mov edi,[rsp+116]         ; n_pixZ
mov edx,[rsp+112]         ; n_pixY
mov ecx,[rsp+124]         ; n_pixX

;;;fillup pointer to data
mov rsi,[rsp+88]         ; *ascan_complex (if not existent = NULL = 0)

mov r10,[rsp+32]    ;*pixelpos
mov r11,[rsp+24]         ;*buffer_real[index_aktual]              NP
      mov r12,[rsp+96]         ;*image_sum_real
mov r13,[rsp+0]          ;*bild_real


;;delta Counter
mov rbp,0                 ; delta IMAGE

Z_loop:

 Y_loop_init:
 mov edx,[rsp+112]         ; n_pixY
 movlps xmm7,[r10]          ;pixelpos     (t),Z(akt),Y(0),X(0_but_is_notimportant)
```

```
      Y_loop:
        ;pix z,y
        movaps xmm3,xmm7      ;          (trash),z_p,y_p,x_p
        shufps xmm3,xmm3,165 ; 10100101b -> z_p,z_p,y_p,y_p
        ;receiver z,y
        movaps xmm2,xmm6      ;     (trash),z_r,y_r,x_r
        shufps xmm2,xmm2,153 ; 10011001b -> (t),(t),z_r,y_r       psrldq xmm2,4
             ;SSE2 shift right by BYTES not BITS! t,t,z,y
        ;sender  z,y
        movaps xmm1,xmm5      ;     (trash),z_s,y_s,x_s
        shufps xmm1,xmm1,153 ; 10011001b -> (t),(t),z_s,y_s       psrldq xmm1,4
             ;SSE2 shift right by BYTES not BITS! t,t,z,y
        ;interleave
        unpcklps xmm2,xmm1    ;     z_s,z_r,y_s,y_r

        subps xmm3,xmm2             ;     pix-senderpos pix-receiverpos
        mulps xmm3,xmm3             ;     quadrieren

        movaps xmm2,xmm3      ;     z_s,z_r,y_s,y_r
        shufps xmm2,xmm2,238 ;11101110b-> t,t,z_s,z_r            psrldq xmm2,8
             ;SSE2 shift right by BYTES not BITS! t,t,z_s,z_r
        addps  xmm3,xmm2      ;     (t,t,z_s,z_r) + (t,t,y_s,y_r)  -> t,t,S,R
        unpcklps xmm3,xmm3    ;     S,S,R,R  ;!!!!shufps xmm3,xmm3,68  ;     0100 0100b -> S,R,S,R    ;
vinsertf128 ymm3,ymm3,xmm3,1 ; -> S,R,S,R,S,R,S,R

        X_loop_init:
   mov ecx,[rsp+124]      ;    n_pixX
   movss xmm2,[r10]      ;    *pixelpos[3]
         movss xmm7,xmm2       ;    t,Z(akt),Y(akt),X(0)

        X_loop:                      ;     !!!2 PIXEL in parallel!!!
          ;x_p1 & x_p0
          movss        xmm2,xmm7      ;    pixel  t,t,t,x_p
          shufps xmm2,xmm2,0    ;    x_p,x_p,x_p,x_p
          addss        xmm2,xmm4      ;    x_p,x_p,x_p,x_p + t,t,t,resolut -> x_p0,x_p0,x_p0,x_p1
          shufps xmm2,xmm2,17   ;    0001 0001b ->  x_p1,x_p0,x_p1,x_p0
      movaps xmm9,xmm2
      addps   xmm9,xmm10     ;    x_p1,x_p0,x_p1,x_p0+ 2*resolut ->  x_p3,x_p2,x_p3,x_p2

vinsertf128 ymm2,ymm2,xmm9,1 ; x_p0,x_p0,x_p0,x_p1,x_p3,x_p2,x_p3,x_p2

         ;Pipe 2

vmovaps ymm9, ymm2
vaddps ymm9, ymm9, ymm10
vaddps ymm9, ymm9, ymm10        ; x_p1,x_p0,x_p1,x_p0,x_p3,x_p2,x_p3,x_p2 + 4*resolut
                            ;-> x_p5,x_p4,x_p5,x_p4,x_p7,x_p6,x_p7,x_p6


          ;x_r &  x_s
          movss xmm1,xmm6       ;    receiver (trash),z_r,y_r,x_r
          shufps xmm1,xmm5,0    ;    sender; 0000 0000b -> S,S,R,R

          vinsertf128 ymm1,ymm1,xmm1, 1 ; -> S,S,R,R,S,S,R,R

          ;subps xmm2,xmm1       ;    x_p-x_r, x_p-x_s -> S1,S0,R1,R0
      ;PIPE2
      ;subps xmm9,xmm1       ;    x_p-x_r, x_p-x_s -> S3,S2,R3,R2

vsubps ymm2,ymm2,ymm1         ;        x_p-x_r, x_p-x_s -> S1,S0,R1,R0,S3,S2,R3,R2
         ;PIPE2
```

```
vsubps ymm9,ymm9,ymm1 ; x_p-x_r, x_p-x_s -> S1,S0,R1,R0,S3,S2,R3,R2

         ;mulps xmm2,xmm2      ;    quadrieren
      ;PIPE2
      ;mulps xmm9,xmm9      ;    quadrieren

vmulps ymm2,ymm2,ymm2         ;        quadrieren
        ;PIPE2
vmulps ymm9,ymm9,ymm9

         ;addps xmm2,xmm3      ;    S1x,S0x,R1x,R0x + Syz,Syz,Ryz,Ryz
      ;PIPE2
     ; addps xmm9,xmm3      ;    S1x,S0x,R1x,R0x + Syz,Syz,Ryz,Ryz

vaddps ymm2,ymm2,ymm3        ;        S1x,S0x,R1x,R0x,S3x,S2x,R3x,R2x + Syz,Syz,Ryz,Ryz,Syz,Syz,Ryz,Ryz
        ;PIPE2
vaddps ymm9,ymm9,ymm3

         ;sqrtps xmm2,xmm2     ;    sqrt
   ;rsqrtps xmm2,xmm2     ;    reziprok. sqrt approx.
   ;rcpps xmm2,xmm2     ;

      ;PIPE2
     ; sqrtps xmm9,xmm9     ;    sqrt

vsqrtps ymm2,ymm2        ;        sqrt
        ;PIPE 2
vsqrtps ymm9,ymm9

      ;rsqrtps xmm9,xmm9     ;    reziprok. sqrt approx.
      ;rcpps xmm9,xmm9     ;


         ;movaps xmm1,xmm2     ;
vmovaps ymm1,ymm2
        ; shufps xmm1,xmm1,254 ;    1111 1110b -> S1,S1,S1,S0
vshufps ymm1,ymm1,254        ;        1111 1110b -> S1,S1,S1,S0,S3,S3,S3,S2
        ; addps xmm2,xmm1     ;    S1,S0,R1,R0 +  S1,S1,S1,S0 = t,t,P1,P0
vaddps ymm2,ymm2,ymm1 ;         t,t,P1,P0,t,t,P3,P2

       ;PIPE2
     ; movaps xmm11,xmm9
vmovaps ymm11,ymm9
     ;shufps xmm11,xmm11,254 ;    1111 1110b -> S3,S3,S3,S2
vshufps ymm11, ymm11,254
     ; addps xmm9,xmm11     ;    S3,S2,R3,R2 +  S3,S3,S3,S2 = t,t,P3,P2
vaddps ymm9,ymm9,ymm11        ;        t,t,P5,P4,t,t,P7,P6

         ;lauflaenge in t zu index
        ; mulps xmm2,xmm8      ;    t,t,P1,P0*factor,factor,factor,factor = t,t,index1,index0

      ;PIPE2 lauflaenge in t zu index
     ; mulps xmm9,xmm8      ;    t,t,P1,P0*factor,factor,factor,factor = t,t,index1,index0
vmulps ymm2,ymm2,ymm8
        ;PIPE2
vmulps ymm9,ymm9,ymm8

;;; ACHTUNG!!! Der erste Pixel in xmm 2 ist jetzt natürlich P2 (Pixel 3)!!!
;;; Neue Reihenfolge: P2, P3, P0, P1, P6, P7, P4, P5

;time to index for P2
```

```
        cvtss2si eax,xmm2     ;     index_value0

       ;;;;;;;;;;;;;;;;;;;;;;;;
       ;rangecheck pixel 3
        cmp eax,dword [rsp+128]         ;n_Ascan *5    ;cmp eax,[rsp+120]    ;n_Ascan
        jge outrange_xsum3           ;0....2999
        cmp eax,0                   ;
        jl outrange_xsum3

        ;inrange real
        shl eax,3                   ;imul eax,8    ;index-> 64bit (double) addr   NP
        fld qword [r12+rbp+16]         ;*image_sum_real+ delta Image +2pixel
        fld qword [r11+rax]         ;*buffer_real[index_aktual]  fpu double move
        faddp st1,st0

        ;write pixel2!
        fstp qword [r13+rbp+16] ; write pixel to *bild_real[index_aktual]

          ;complex Ascan ?
          cmp rsi,0                   ;dword [rsp+88],0  ;*AScan_complex
          jz out_pixel3

          mov rbx,[rsp+40]          ;*buffer_complex[index_aktual]
          fld qword [rbx+rax]          ;fpu move

          mov rbx,[rsp+104]          ;*image_sum_compl
          fld qword [rbx+rbp+16]   ; delta Image +2pixel
          faddp st1,st0

          mov rbx,[rsp+16]           ;*bild_compl[index_aktual]
          fstp qword [rbx+rbp+16]  ; rbp = *bild_compl[index_aktual]
          jmp out_pixel3

       outrange_xsum3:
        ;real
        fld qword [r12+rbp+16]  ;*image_sum_real + deltabild+ 2pixel

        ;write pixel1!
        fstp qword [r13+rbp+16] ; write pixel to *bild_real[index_aktual]

          ;Complex AScan ?
          cmp rsi,0                   ;dword [rsp+88],0  ;*AScan_complex
          jz out_pixel3         ;jump out if only real

          mov rbx,[rsp+104]          ;*image_sum_compl
          fld qword [rbx+rbp+16]

          mov rbx,[rsp+16]          ;*bild_compl[index_aktual]
          fstp qword [rbx+rbp+16]   ; rbp = *bild_compl[index_aktual]

       ;;;;;;;;;;;
       out_pixel3:


;;;P3(Pixel4)

       rangecheck_pixel_4:

   xor rax,rax           ; clear upper part of RAX for later EAX -> RAX (for delta on memory access)
        shufps xmm2,xmm2,85        ; 0101 0101b -> P1,P1,P1,P1
        cvtss2si eax,xmm2        ; index_value
```

```
        cmp eax,dword [rsp+128]        ;n_Ascan *5   ;cmp eax,[rsp+120]    ;n_Ascan
        jge outrange_xsum4        ;0....2999
        cmp eax,0                ;
        jl outrange_xsum4

        ;inrange real
        shl eax,3                ;imul eax,8    ;index-> 64bit (double) addr   NP
        fld qword [r12+rbp+24]        ;*image_sum_real+ delta Image +3pixel
        fld qword [r11+rax]        ;*buffer_real[index_aktual]  fpu double move
        faddp st1,st0

        ;write pixel3!
        fstp qword [r13+rbp+24] ; write pixel to *bild_real[index_aktual]

          ;complex Ascan ?
          cmp rsi,0                ;dword [rsp+88],0  ;*AScan_complex
          jz out_pixel4

          mov rbx,[rsp+40]          ;*buffer_complex[index_aktual]
          fld qword [rbx+rax]          ;fpu move

          mov rbx,[rsp+104]          ;*image_sum_compl
          fld qword [rbx+rbp+24]  ; delta Image +3pixel
          faddp st1,st0

          mov rbx,[rsp+16]          ;*bild_compl[index_aktual]
          fstp qword [rbx+rbp+24] ; rbp = *bild_compl[index_aktual]
          jmp out_pixel4

       outrange_xsum4:
        ;real
        fld qword [r12+rbp+24] ;*image_sum_real + deltabild+ 1pixel

        ;write pixel1!
        fstp qword [r13+rbp+24] ; write pixel to *bild_real[index_aktual]

          ;Complex AScan ?
          cmp rsi,0                ;dword [rsp+88],0  ;*AScan_complex
          jz out_pixel4          ;jump out if only real

          mov rbx,[rsp+104]          ;*image_sum_compl
          fld qword [rbx+rbp+24]

          mov rbx,[rsp+16]          ;*bild_compl[index_aktual]
          fstp qword [rbx+rbp+24]  ; rbp = *bild_compl[index_aktual]

       ;;;;;;;;;;
       out_pixel4:

       ;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;P0(Pixel 1)
       rangecheck_pixel_1:
vperm2f128 ymm2, ymm2, ymm2, 1

   xor rax,rax          ; clear upper part of RAX for later EAX -> RAX (for delta on memory access)
        cvtss2si eax,xmm2        ; index_value

        cmp eax,dword [rsp+128]         ;n_Ascan *5      ;cmp eax,dword [rsp+120] ;n_Ascan
        jge outrange_xsum1         ;0....2999
```

```
        cmp eax,0                   ;
        jl outrange_xsum1

        ;inrange real
        fld qword [r12+rbp] ; *image_sum_real fpu move
        shl eax,3                  ;imul eax,8    ;index-> 64bit (double) addr   NP
        fld qword [r11+rax]        ;*buffer_real[index_aktual]         fpu double move
        faddp st1,st0

   ;write PIXEL0!
        fstp qword [r13+rbp]        ; write pixel to *bild_real[index_aktual]

          ;complex Ascan ?
          cmp rsi,0                  ;dword [rsp+88],0  ;*AScan_complex
          jz out_pixel1

          mov rbx,[rsp+40]          ;*buffer_complex[index_aktual]
          fld qword [rbx+rax]          ;fpu move

          mov rbx,[rsp+104]          ;*image_sum_compl
          fld qword [rbx+rbp]
          faddp st1,st0

          mov rbx,[rsp+16]          ;*bild_compl[index_aktual]
          fstp qword [rbx+rbp]   ; rbp = *bild_compl[index_aktual]
          jmp out_pixel1

       outrange_xsum1:
        ;real
        fld qword [r12+rbp]  ;*image_sum_real FPU MOVE

        ;;write pixel0
        fstp qword [r13+rbp]         ; write pixel0 to *bild_real[index_aktual]

          ;Complex AScan ?
          cmp rsi,0                  ;dword [rsp+88],0  ;*AScan_complex
          jz out_pixel1 ;jump out if only real

          mov rbx,[rsp+104]          ;*image_sum_compl
          fld qword [rbx+rbp]

          mov rbx,[rsp+16]          ;*bild_compl[index_aktual]
          fstp qword [rbx+rbp]   ; rbp = *bild_compl[index_aktual]

       ;;;;;;;;;;;;;;;;;;;;;;;;;;;
       out_pixel1:

;;;P1(Pixel 2)
       ;;;;;;;;;;;;;;;;;;;;;;;;;;;
       rangecheck_pixel_2:

   xor rax,rax          ; clear upper part of RAX for later EAX -> RAX (for delta on memory access)
        shufps xmm2,xmm2,85        ; 0101 0101b -> P1,P1,P1,P1
        cvtss2si eax,xmm2        ; index_value

        cmp eax,dword [rsp+128]        ;n_Ascan *5   ;cmp eax,[rsp+120]    ;n_Ascan
        jge outrange_xsum2        ;0....2999
        cmp eax,0                  ;
        jl outrange_xsum2

        ;inrange real
```

```
        shl eax,3                  ;imul eax,8    ;index-> 64bit (double) addr   NP
        fld qword [r12+rbp+8]         ;*image_sum_real+ delta Image +1pixel
        fld qword [r11+rax]         ;*buffer_real[index_aktual]  fpu double move
        faddp st1,st0

        ;write pixel1!
        fstp qword [r13+rbp+8] ; write pixel to *bild_real[index_aktual]

          ;complex Ascan ?
          cmp rsi,0                  ;dword [rsp+88],0  ;*AScan_complex
          jz out_pixel2

          mov rbx,[rsp+40]          ;*buffer_complex[index_aktual]
          fld qword [rbx+rax]          ;fpu move

          mov rbx,[rsp+104]           ;*image_sum_compl
          fld qword [rbx+rbp+8]   ; delta Image +1pixel
          faddp st1,st0

          mov rbx,[rsp+16]           ;*bild_compl[index_aktual]
          fstp qword [rbx+rbp+8]  ; rbp = *bild_compl[index_aktual]
          jmp out_pixel2

       outrange_xsum2:
        ;real
        fld qword [r12+rbp+8] ;*image_sum_real + deltabild+ 1pixel

        ;write pixel1!
        fstp qword [r13+rbp+8] ; write pixel to *bild_real[index_aktual]

          ;Complex AScan ?
          cmp rsi,0                  ;dword [rsp+88],0  ;*AScan_complex
          jz out_pixel2          ;jump out if only real

          mov rbx,[rsp+104]           ;*image_sum_compl
          fld qword [rbx+rbp+8]

          mov rbx,[rsp+16]          ;*bild_compl[index_aktual]
          fstp qword [rbx+rbp+8]   ; rbp = *bild_compl[index_aktual]

       ;;;;;;;;;;;
       out_pixel2:

;;;P6(Pixel 7)
                ;;;;;;;;;;;;;;;;;;;;;;;;
rangecheck_pixel_7:

xor rax,rax
cvtss2si eax,xmm9

        cmp eax,dword [rsp+128]        ;n_Ascan *5   ;cmp eax,[rsp+120]    ;n_Ascan
jge outrange_xsum7        ;0....2999
        cmp eax,0                  ;
jl outrange_xsum7

        ;inrange real
        shl eax,3                  ;imul eax,8    ;index-> 64bit (double) addr   NP
 fld qword [r12+rbp+48]        ;*image_sum_real+ delta Image +2pixel
        fld qword [r11+rax]         ;*buffer_real[index_aktual]  fpu double move
        faddp st1,st0
```

```
        ;write pixel2!
        fstp qword [r13+rbp+48] ; write pixel to *bild_real[index_aktual]

          ;complex Ascan ?
          cmp rsi,0                 ;dword [rsp+88],0  ;*AScan_complex
jz out_pixel7

mov rbx,[rsp+40]         ;*buffer_complex[index_aktual]
          fld qword [rbx+rax]         ;fpu move

          mov rbx,[rsp+104]          ;*image_sum_compl
fld qword [rbx+rbp+48]  ; delta Image +2pixel
          faddp st1,st0

          mov rbx,[rsp+16]          ;*bild_compl[index_aktual]
fstp qword [rbx+rbp+48] ; rbp = *bild_compl[index_aktual]
jmp out_pixel7

outrange_xsum7:
        ;real
fld qword [r12+rbp+48] ;*image_sum_real + deltabild+ 2pixel

        ;write pixel1!
fstp qword [r13+rbp+48] ; write pixel to *bild_real[index_aktual]

          ;Complex AScan ?
          cmp rsi,0                 ;dword [rsp+88],0  ;*AScan_complex
jz out_pixel7         ;jump out if only real

          mov rbx,[rsp+104]          ;*image_sum_compl
fld qword [rbx+rbp+48]

          mov rbx,[rsp+16]         ;*bild_compl[index_aktual]
fstp qword [rbx+rbp+48]  ; rbp = *bild_compl[index_aktual]

        ;;;;;;;;;;;
out_pixel7:


;;;P7(Pixel8)
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;
rangecheck_pixel_8:

   xor rax,rax          ; clear upper part of RAX for later EAX -> RAX (for delta on memory access)
        shufps xmm9,xmm9,85        ; 0101 0101b -> P1,P1,P1,P1
        cvtss2si eax,xmm9        ; index_value

        cmp eax,dword [rsp+128]        ;n_Ascan *5   ;cmp eax,[rsp+120]    ;n_Ascan
jge outrange_xsum8        ;0....2999
        cmp eax,0                 ;
jl outrange_xsum8

        ;inrange real
        shl eax,3                 ;imul eax,8    ;index-> 64bit (double) addr   NP
fld qword [r12+rbp+56]        ;*image_sum_real+ delta Image +3pixel
        fld qword [r11+rax]        ;*buffer_real[index_aktual]  fpu double move
        faddp st1,st0

        ;write pixel3!
fstp qword [r13+rbp+56] ; write pixel to *bild_real[index_aktual]
```

```
           ;complex Ascan ?
           cmp rsi,0                  ;dword [rsp+88],0  ;*AScan_complex
jz out_pixel8

           mov rbx,[rsp+40]          ;*buffer_complex[index_aktual]
           fld qword [rbx+rax]          ;fpu move

           mov rbx,[rsp+104]           ;*image_sum_compl
fld qword [rbx+rbp+56]  ; delta Image +3pixel
           faddp st1,st0

           mov rbx,[rsp+16]           ;*bild_compl[index_aktual]
fstp qword [rbx+rbp+56] ; rbp = *bild_compl[index_aktual]
jmp out_pixel8

outrange_xsum8:
         ;real
fld qword [r12+rbp+56] ;*image_sum_real + deltabild+ 1pixel

         ;write pixel1!
fstp qword [r13+rbp+56] ; write pixel to *bild_real[index_aktual]

           ;Complex AScan ?
           cmp rsi,0                  ;dword [rsp+88],0  ;*AScan_complex
jz out_pixel8         ;jump out if only real

           mov rbx,[rsp+104]           ;*image_sum_compl
fld qword [rbx+rbp+56]

           mov rbx,[rsp+16]          ;*bild_compl[index_aktual]
fstp qword [rbx+rbp+56]   ; rbp = *bild_compl[index_aktual]

        ;;;;;;;;;;
        out_pixel8:

;;;P4(Pixel 5)
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;
rangecheck_pixel_5:
        vperm2f128 ymm9, ymm9, ymm9, 1

   xor rax,rax           ; clear upper part of RAX for later EAX -> RAX (for delta on memory access)
         cvtss2si eax,xmm9        ; index_value

         cmp eax,dword [rsp+128]         ;n_Ascan *5      ;cmp eax,dword [rsp+120] ;n_Ascan
jge outrange_xsum5         ;0....2999
         cmp eax,0                   ;
jl outrange_xsum5

         ;inrange real
fld qword [r12+rbp+32] ; *image_sum_real fpu move
         shl eax,3                ;imul eax,8    ;index-> 64bit (double) addr   NP
         fld qword [r11+rax]        ;*buffer_real[index_aktual]         fpu double move
         faddp st1,st0

;write PIXEL5!
fstp qword [r13+rbp+32]        ; write pixel to *bild_real[index_aktual]

           ;complex Ascan ?
           cmp rsi,0                  ;dword [rsp+88],0  ;*AScan_complex
jz rangecheck_pixel_6
```

```
          mov rbx,[rsp+40]         ;*buffer_complex[index_aktual]
          fld qword [rbx+rax]         ;fpu move

          mov rbx,[rsp+104]         ;*image_sum_compl
fld qword [rbx+rbp+32]
          faddp st1,st0

          mov rbx,[rsp+16]         ;*bild_compl[index_aktual]
fstp qword [rbx+rbp+32]   ; rbp = *bild_compl[index_aktual]
jmp rangecheck_pixel_6

outrange_xsum5:
         ;real
fld qword [r12+rbp+32]   ;*image_sum_real FPU MOVE

         ;;write pixel0
fstp qword [r13+rbp+32]         ; write pixel0 to *bild_real[index_aktual]

          ;Complex AScan ?
          cmp rsi,0                 ;dword [rsp+88],0  ;*AScan_complex
jz rangecheck_pixel_6 ;jump out if only real

          mov rbx,[rsp+104]         ;*image_sum_compl
fld qword [rbx+rbp+32]

          mov rbx,[rsp+16]         ;*bild_compl[index_aktual]
fstp qword [rbx+rbp+32]   ; rbp = *bild_compl[index_aktual]

;;;P5(Pixel 6)
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;
rangecheck_pixel_6:

   xor rax,rax           ; clear upper part of RAX for later EAX -> RAX (for delta on memory access)
        shufps xmm9,xmm9,85         ; 0101 0101b -> P1,P1,P1,P1
        cvtss2si eax,xmm9         ; index_value

        cmp eax,dword [rsp+128]         ;n_Ascan *5   ;cmp eax,[rsp+120]    ;n_Ascan
jge outrange_xsum6         ;0....2999
        cmp eax,0                 ;
jl outrange_xsum6

        ;inrange real
        shl eax,3                 ;imul eax,8    ;index-> 64bit (double) addr   NP
fld qword [r12+rbp+40]         ;*image_sum_real+ delta Image +1pixel
        fld qword [r11+rax]         ;*buffer_real[index_aktual]  fpu double move
        faddp st1,st0

        ;write pixel1!
fstp qword [r13+rbp+40] ; write pixel to *bild_real[index_aktual]

          ;complex Ascan ?
          cmp rsi,0                 ;dword [rsp+88],0  ;*AScan_complex
jz out_pixel6

       mov rbx,[rsp+40]         ;*buffer_complex[index_aktual]
          fld qword [rbx+rax]         ;fpu move

          mov rbx,[rsp+104]          ;*image_sum_compl
fld qword [rbx+rbp+40]   ; delta Image +1pixel
          faddp st1,st0
```

```
           mov rbx,[rsp+16]          ;*bild_compl[index_aktual]
fstp qword [rbx+rbp+40] ; rbp = *bild_compl[index_aktual]
jmp out_pixel6

outrange_xsum6:
         ;real
fld qword [r12+rbp+40] ;*image_sum_real + deltabild+ 1pixel

         ;write pixel1!
fstp qword [r13+rbp+40] ; write pixel to *bild_real[index_aktual]

           ;Complex AScan ?
           cmp rsi,0                  ;dword [rsp+88],0  ;*AScan_complex
           jz out_pixel2          ;jump out if only real

           mov rbx,[rsp+104]          ;*image_sum_compl
fld qword [rbx+rbp+40]

           mov rbx,[rsp+16]          ;*bild_compl[index_aktual]
fstp qword [rbx+rbp+40]  ; rbp = *bild_compl[index_aktual]

        ;;;;;;;;;;;
out_pixel6:

        ;add x_p + 8pixel
        addss xmm7,xmm10  ; t,t,t,x_p  + t,t,t,2*resolut
        addss xmm7,xmm10  ; t,t,t,x_p  + t,t,t,2*resolut
addss xmm7,xmm10  ; t,t,t,x_p  + t,t,t,2*resolut
addss xmm7,xmm10  ; t,t,t,x_p  + t,t,t,2*resolut

        ;inc image index (4pixel)
add rbp,64         ;*bild_real[index_aktual] +4pixel[double]
sub  ecx,8

check_ifloop:
cmp  ecx,8
        jge X_loop

        ;mod(ecx,4) ==0???
        cmp  ecx,0
        je exit_X_loop  ;finished!!!
          ;add x size until mod(ecx,4) == 0 -> ecx =4
          add ecx,1
          subss xmm7,xmm4      ; t,t,t,x_p  - t,t,t,resolut
          sub rbp,8                ;*bild_real[index_aktual] -1pixel[double]
          jmp check_ifloop

        exit_X_loop:
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      ;add y_p + 1pixel
      xorps xmm0,xmm0           ; xmm0 = 0
      movss xmm0,xmm4           ; t,t,t,resolut
      shufps xmm0,xmm0,81  ; 01010001b -> 0,0,resolut,0
      addps xmm7,xmm0           ; t,t,y_p,t  + 0,0,resolut,0

      sub  edx,1
      jnz Y_loop
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    ;add z_p + 1pixel
```

```
  xorps xmm0,xmm0           ; xmm0 = 0
  movss xmm0,xmm4           ; t,t,t,resolut
  shufps xmm0,xmm0,69    ; 01000101b -> 0,resolut,0,0
  addps xmm7,xmm0           ; t,z_p,t,t  + 0,resolut,0,0

  sub edi,1
  jnz Z_loop
  jmp exit
  ;;;;end sum_branch;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



  ;;;;;;;;;;;;;;;;;;;;;;;;;;
  exit:
;;;;clean up FPU/MMX
      ;emms              ;reset mmx, ready fpu
      ;FLDCW [esp+4]     ;restore old CTRWORD
      ;buffering in stack (16bit!) + FWAIT to ensure save is complete (FSTCW without wait)


      ;recover XMM6-7 (callee convention WIN64(?))
      ;movups xmm7,[rel XMM7_SAVE]
      ;movups xmm6,[rel XMM6_SAVE]

      ;;;;;clean up
      ;mov rsp, [rel RSP_SAVE]
mov rsp,[rsp+136]

      pop rbx
      pop rdi
      pop rsi
      pop rbp


      ret
      ;retn ; bug return near!!!
```

Neuste Profilingumgebung:

```
clear all
close all

disp('#### START ####');
compStartProgram = tic;

workspaceName = datestr(now,'dd-mm-yy_HH-MM');

%Parameter
vergleichsTest = true;
performanceTest = false;
repeats = 3;
NnImg = single(4);  %Wie viele unterschiedliche Bildananazahlen
resImgMix = [10:10:200,220:20:700];

%Codes that will be compared
leonCodes = {@L0sb,@L1sb,@L2sb,@L3sb};
paulCodes = {@pL0, @pL1, @pL2, @pL3};
addsigCodes = {@addsig2vol_3_ASM, @addsig2vol_3_C};
addsigMtCodes = {@addsig2vol_3_ASM_MT, @addsig2vol_3_C_MT};
maxMt = 12;

ourCodes = [leonCodes, paulCodes];
noMtCodes= [leonCodes, paulCodes, addsigCodes];
allCodes = [leonCodes, paulCodes, addsigCodes, addsigMtCodes];
nNoMtCodes = size(noMtCodes,2);
nMtCodes = size(addsigMtCodes,2);
noMtNames = 2*size(ourCodes,2) + 2;

addsigCodes{1}(1);
addsigCodes{2}(1);
addsigMtCodes{1}(1);
addsigMtCodes{2}(1);

codeNames = string.empty;
for i=1:size(ourCodes,2)
    codeNames(i)=strcat(func2str(ourCodes{i}), "-Double");
end
currentCodeNameSize = size(codeNames,2);
for i=1:size(ourCodes,2)
    codeNames(i + currentCodeNameSize)=strcat(func2str(ourCodes{i}), "-Single");
end
currentCodeNameSize = size(codeNames,2);
for i = size(ourCodes,2)+1:size(noMtCodes,2)
    codeNames(currentCodeNameSize + i - size(ourCodes,2)) = func2str(allCodes{i});
end
currentCodeNameSize = size(codeNames,2);
for i = 0:nMtCodes - 1
    for c = 1:maxMt
        codeNames(currentCodeNameSize + i*maxMt + c) = strcat(func2str(addsigMtCodes{i+1}), " ", int2str(c),"C");
    end
end
for i = 1:size(codeNames,2)
    disp(codeNames(i));
end

codeAufzahlung = string.empty;
for i = 1:repeats
```

```
    codeAufzahlung(i) = num2str(i);
end

%Inputvariabeln
speed = single(1480);               %Geschwindigkeit des Schalls in m/s
addsigSpeed = repmat(speed, 1, NnImg);
resRl = single(1);                  %BildgrÃ¶ÃŸe in Metern


if performanceTest
    times = single(zeros(repeats,size(resImgMix,2),size(codeNames,2))); %Speicher für die erzeilten Zeiten
    for i = 1:size(resImgMix,2)
        %Inputvariablen
        %Bild
        resImg = single(resImgMix(i));                  %BildgrÃ¶ÃŸe in Pixeln
        pixDensity = resRl / resImg;     %Pixeldichte in Meter/Pixel
        resint = pixDensity;
        image = zeros(resImg,resImg,resImg);%Bildmatrix
        try
            imagesforaddsig32 = uint32([resImg,resImg,resImg]);
            imagesforaddsig = double(zeros(resImg,resImg,resImg));
        catch e
        end
        img_start = [0.0,0.0,0.0];
        %Scan
        posObj = [(rand()*resRl),(rand()*resRl),(rand()*resRl)];     %Random Objekt Position
        maxAscan = 2 * sqrt(sqrt(resRl^2+resRl^2)^2+resRl^2);       %m
        minAscan = 2 * sqrt(sqrt(pixDensity^2+pixDensity^2)^2+pixDensity^2);     %m
        maxTime = maxAscan / speed;         %s
        minTime = minAscan / speed;         %s
        intAscan = minTime;                 %s
        samplesAscan = 1 + maxTime / minTime;
        presetAscan = 0:intAscan:maxTime;
        speedAscan = 1 / (speed*intAscan);
        invPixDensity = 1/pixDensity;    %FÃ¼r Indexing bei Bilderstellung
        posSens = zeros(NnImg,3);
        posRecs = zeros(NnImg,3);
        Ascans = zeros(round(2 * samplesAscan),NnImg);
        for x = 1:NnImg
            posRec = [(rand()*resRl),(rand()*resRl),(rand()*resRl)];       %Random Position Reciever festelgen
            posRecs(x,:) = posRec;
            %aRecs(:,:,x) = single(posRec);
            posSen = [(rand()*resRl),(rand()*resRl),(rand()*resRl)];       %Random Position Sender festelgen
            posSens(x,:) = posSen;
            %aSens(:,:,x) = single(posSen);
            %         fprintf('%g\n',posSen)
            disGes = sqrt(sum((posSen - posObj).^2)) + sqrt(sum((posObj - posRec).^2));  %Distanzberechnung
            indexPeak = (disGes / speed) / intAscan;        %Berechnung Peak Index
            Ascan = randn(1,round(2 * samplesAscan));     %Ascan mit Rauschen kreieren
            %Ascan = zeros(1,round(2 * samplesAscan));
            Ascan(round(indexPeak)) = Ascan(round(indexPeak)) + (100/NnImg);
            Ascans(:,x) = Ascan;
        end
        addsigAscans = double(Ascans);

        fprintf('--- Res: %g(%g)/%g(%g) ---\n',resImgMix(i), i, resImgMix(end), size(resImgMix,2));
        for c = 1:size(codeNames,2)
            try
                fprintf(codeNames(c) + "\n");
                for n = 1:repeats
                    if c<=size(leonCodes,2)
```

```
                    tic;
                    allCodes{c}(resImg, NnImg, Ascans, speed, intAscan, posSens, posRecs, image,
                                img_start, pixDensity);
                    times(n, i, c) = toc;
                elseif c<=size(ourCodes,2)
                    tic;
                    allCodes{c}(resImg, NnImg, posRecs, posSens, image, Ascans, speedAscan,
                                pixDensity, invPixDensity);
                    times(n, i, c) = toc;
                elseif c<=size(ourCodes,2) + size(paulCodes,2)
                    tic;
                    allCodes{c-size(ourCodes,2)}(single(resImg), single(NnImg), single(Ascans),
                                single(speed), single(intAscan), single(posSens), single(posRecs),
                                single(image), single(img_start), single(pixDensity));
                    times(n, i, c) = toc;
                elseif c<=2*size(ourCodes,2)
                    tic;
                    allCodes{c-size(ourCodes,2)}(single(resImg), single(NnImg), single(posRecs),
                                single(posSens), single(image), single(Ascans), single(speedAscan),
                                single(pixDensity), single(invPixDensity));
                    times(n, i, c) = toc;
                elseif c<=noMtNames
                    tic;
                    allCodes{c-size(ourCodes,2)}(addsigAscans,single(img_start'),single(posRecs'),
                                single(posSens'),single(speed),single(pixDensity),intAscan,
                                imagesforaddsig32',imagesforaddsig);
                    times(n, i, c) = toc;
                elseif c <= noMtNames + maxMt
                    if n==1
                        allCodes{nNoMtCodes + 1}(c-noMtNames);
                    end
                    tic;
                    allCodes{nNoMtCodes + 1}(addsigAscans,single(img_start'),posRecs',posSens',
                                speed,pixDensity,intAscan,imagesforaddsig32',imagesforaddsig);
                    times(n, i, c) = toc;
                elseif c <= noMtNames + 2*maxMt
                    if n==1
                        allCodes{nNoMtCodes + 2}(c-noMtNames-maxMt);
                    end
                    tic;
                    allCodes{nNoMtCodes + 2}(addsigAscans,single(img_start'),posRecs',posSens',
                                speed,pixDensity,intAscan,imagesforaddsig32',imagesforaddsig);
                    times(n, i, c) = toc;
                end
                fprintf([' in ',num2str(times(n, i, c)),' Sekunden\n']);
            end
        catch e
            times(:, i, c) = NaN;
            fprintf('CRASHED\n');
            fprintf(1,'The identifier was:\n%s',e.identifier);
            fprintf(1,'There was an error! The message was:\n%s\n',e.message);
        end
    end
    save("Doku_" + workspaceName + "Unfertig");
end
compEndProgram = toc(compStartProgram);

nCodes = size(codeNames,2);
meanTimes = reshape(mean(times,1), [size(times,2),size(times,3)])';
medianTimes = reshape(median(times,1), [size(times,2),size(times,3)])';
meanDoubles = sum(meanTimes(1:8,:));
```

```
meanSingles = sum(meanTimes(9:16,:));

for i = 1:size(resImgMix,2)
    voxels(i) = NnImg*(resImgMix(i)^3);
end

CPUSpeed = 3.6e9; %GHz
meanPerformanceClocksPerVoxel = (meanTimes./(1/CPUSpeed))./repmat(voxels,size(meanTimes,1),1);
meanPerformanceClocksPerVoxel2 = (1/CPUSpeed)./(meanTimes./repmat(voxels,size(meanTimes,1),1));
meanPeformance = repmat(voxels,size(meanTimes,1),1)./(meanTimes.*1e9);

matlabIndex = 1:nCodes-2*maxMt-2;
AsmMt = nCodes-2*maxMt+1:nCodes-maxMt;
CMt = nCodes-maxMt+1:nCodes;
AsmCMt = nCodes-2*maxMt+1:nCodes;

startData = 10;
endData = 520;

figure('Name','Clocks/Voxel - Mean','NumberTitle','off');
plot(resImgMix,meanPerformanceClocksPerVoxel2(AsmMt,:));
title('Clocks/Voxel - Mean');
xlabel('Auflösung in Pixel');
ylabel('Clocks/Voxel');
legend(codeNames(AsmMt),'Location','northwest')

figure('Name','Zeit - Mean','NumberTitle','off');
imagesc(resImgMix,[1:nCodes],meanTimes);
%imagesc(meanTimes);
title('Zeit - Mean');
xlabel('Auflösung in Pixel');
ylabel('Codes');
legend(codeNames,'Location','northwest')

figure('Name','Peformance - Mean - ns','NumberTitle','off');
imagesc(resImgMix(find(resImgMix==startData):find(resImgMix==endData)),[1:nCodes],
            meanPeformance(:,find(resImgMix==startData):find(resImgMix==endData)));
title('Peformance - Mean - ns');
xlabel('Auflösung in Pixel');
ylabel('Codes');
%legend(codeNames,'Location','northwest')

figure('Name','Peformance - Mean - ns - ASM','NumberTitle','off');
imagesc(resImgMix(find(resImgMix==startData):find(resImgMix==endData)),[AsmMt],
            meanPeformance(AsmMt,find(resImgMix==startData):find(resImgMix==endData)));
title('Peformance - Mean - ns - ASM');
xlabel('Auflösung in Pixel');
ylabel('Codes');
legend(codeNames,'Location','northwest')

figure('Name','Peformance - Mean - ns - C','NumberTitle','off');
imagesc(resImgMix(find(resImgMix==startData):find(resImgMix==endData)),[CMt],
            meanPeformance(CMt,find(resImgMix==startData):find(resImgMix==endData)));
title('Peformance - Mean - ns - C');
xlabel('Auflösung in Pixel');
ylabel('Codes');
legend(codeNames,'Location','northwest')

figure('Name','Peformance - Mean','NumberTitle','off');
title('Peformance - Mean');
xlabel('Auflösung in Pixel');
```

```
ylabel('Geschwindigkeit in Voxel/Sekunde');
hold on
for y = 1:nCodes
    plot(resImgMix,voxels(1,:)./meanTimes(y,:))
end
legend(codeNames,'Location','northwest')
hold off

figure('Name','Peformance - Mean','NumberTitle','off');
title('Peformance - Mean');
xlabel('Auflösung in Pixel');
ylabel('Geschwindigkeit in Voxel/Sekunde');
plot(resImgMix,voxels./[meanSingles;meanDoubles])
legend(["Single", "Doubles"],'Location','northwest')

figure('Name','Peformance - Mean - MatlabCodes','NumberTitle','off');
title('Peformance - Mean - MatlabCodes');
xlabel('Auflösung in Pixel');
ylabel('Geschwindigkeit in Voxel/Sekunde');
hold on
for y = matlabIndex
    plot(resImgMix,voxels(1,:)./meanTimes(y,:))
end
legend(codeNames(matlabIndex),'Location','northwest')
hold off

for y = 1:8
    figure('Name',['Peformance - Mean - Si vs Db - ', func2str(allCodes{y})],'NumberTitle','off');
    title(['Peformance - Mean - Si vs Db - ', func2str(allCodes{y})]);
    xlabel('Auflösung in Pixel');
    ylabel('Geschwindigkeit in Voxel/Sekunde');
    plot(resImgMix,voxels(1,:)./meanTimes([y,y+8],:))
    legend(codeNames([y,y+8]),'Location','northwest')
end

figure('Name','Peformance - Mean - ASM','NumberTitle','off');
title('Peformance - Mean - ASM');
xlabel('Auflösung in Pixel');
ylabel('Geschwindigkeit in Voxel/Sekunde');
hold on
for y = AsmMt
    plot(resImgMix,voxels(1,:)./meanTimes(y,:))
end
legend(codeNames(AsmMt),'Location','northwest')
hold off

figure('Name','Peformance - Mean - C','NumberTitle','off');
title('Peformance - Mean - C');
xlabel('Auflösung in Pixel');
ylabel('Geschwindigkeit in Voxel/Sekunde');
hold on
for y = CMt
    plot(resImgMix,voxels(1,:)./meanTimes(y,:))
end
legend(codeNames(CMt),'Location','northwest')
hold off

figure('Name','3D Peformance begrenzt','NumberTitle','off')
surf(meanPeformance(:,find(resImgMix==startData):find(resImgMix==endData)));
shading flat
figure('Name','3D Zeiten','NumberTitle','off')
```

```
    surf(meanTimes);
    shading flat
end


if vergleichsTest
    %Inputvariablen
    %Bild
    resImg = 100;               %BildgrÃ¶ÃŸe in Pixeln
    pixDensity = resRl / resImg;    %Pixeldichte in Meter/Pixel
    resint = pixDensity;
    image = zeros(resImg,resImg,resImg);%Bildmatrix
    bilder = zeros(resImg,resImg,resImg,size(codeNames,2)); %Speicher für die erzeilten Zeiten
    try
        imagesforaddsig32 = uint32([resImg,resImg,resImg]);
        imagesforaddsig = double(zeros(resImg,resImg,resImg));
    catch e
    end
    img_start = [0.0,0.0,0.0];
    %Scan
    posObj = [(rand()*resRl),(rand()*resRl),(rand()*resRl)];    %Random Objekt Position
    maxAscan = 2 * sqrt(sqrt(resRl^2+resRl^2)^2+resRl^2);       %m
    minAscan = 2 * sqrt(sqrt(pixDensity^2+pixDensity^2)^2+pixDensity^2);    %m
    maxTime = maxAscan / speed;         %s
    minTime = minAscan / speed;         %s
    intAscan = minTime;                 %s
    samplesAscan = 1 + maxTime / minTime;
    presetAscan = 0:intAscan:maxTime;
    speedAscan = 1 / (speed*intAscan);
    invPixDensity = 1/pixDensity;   %FÃ¼r Indexing bei Bilderstellung
    posSens = zeros(NnImg,3);
    posRecs = zeros(NnImg,3);
    Ascans = zeros(round(2 * samplesAscan),NnImg);
    for x = 1:NnImg+1
        posRec = [(rand()*resRl),(rand()*resRl),(rand()*resRl)];      %Random Position Reciever festelgen
        posRecs(x,:) = posRec;
        %aRecs(:,:,x) = single(posRec);
        posSen = [(rand()*resRl),(rand()*resRl),(rand()*resRl)];      %Random Position Sender festelgen
        posSens(x,:) = posSen;
        %aSens(:,:,x) = single(posSen);
        %         fprintf('%g\n',posSen)
        disGes = sqrt(sum((posSen - posObj).^2)) + sqrt(sum((posObj - posRec).^2));  %Distanzberechnung
        indexPeak = (disGes / speed) / intAscan;        %Berechnung Peak Index
        Ascan = randn(1,round(2 * samplesAscan));    %Ascan mit Rauschen kreieren
        %Ascan = zeros(1,round(2 * samplesAscan));
        Ascan(round(indexPeak)) = Ascan(round(indexPeak)) + (100/NnImg);
        Ascans(:,x) = Ascan;
    end
    addsigAscans = double(Ascans);
    for c = 1:size(codeNames,2)
        try
            fprintf(codeNames(c) + "\n");
                if c<=size(leonCodes,2)
                    bilder(:,:,:,c)= allCodes{c}(resImg, NnImg, Ascans(:,1:NnImg), speed, intAscan,
                                posSens(1:NnImg,:), posRecs(1:NnImg,:), image, img_start, pixDensity);
                elseif c<=size(ourCodes,2)
                    bilder(:,:,:,c)= allCodes{c}(resImg, NnImg, posRecs(1:NnImg,:),posSens(1:NnImg,:),
                                image, Ascans(:,1:NnImg), speedAscan, pixDensity, invPixDensity);
                elseif c<=size(ourCodes,2) + size(paulCodes,2)
                    bilder(:,:,:,c)= allCodes{c-size(ourCodes,2)}(single(resImg), single(NnImg),
                                single(Ascans(:,1:NnImg)), single(speed), single(intAscan),
```

```
                                single(posSens(1:NnImg,:)), single(posRecs(1:NnImg,:)),
                                            single(image), single(img_start), single(pixDensity));
                elseif c<=2*size(ourCodes,2)
                    bilder(:,:,:,c)= allCodes{c-size(ourCodes,2)}(single(resImg), single(NnImg),
                                single(posRecs(1:NnImg,:)), single(posSens(1:NnImg,:)), single(image),
                                single(Ascans(:,1:NnImg)), single(speedAscan),
                                single(pixDensity), single(invPixDensity));
                elseif c<=noMtNames
                    bilder(:,:,:,c)= allCodes{c-size(ourCodes,2)}(addsigAscans,single(img_start'),
                                single(posRecs'),single(posSens'),single(speed),single(pixDensity),
                                intAscan,imagesforaddsig32',imagesforaddsig);
                elseif c <= noMtNames + maxMt
                    allCodes{nNoMtCodes + 1}(c-noMtNames);
                    bilder(:,:,:,c)= allCodes{nNoMtCodes + 1}(addsigAscans,single(img_start'),
                                single(posRecs'),single(posSens'),single(speed),single(pixDensity),
                                intAscan,imagesforaddsig32',imagesforaddsig);
                elseif c <= noMtNames + 2*maxMt
                    allCodes{nNoMtCodes + 2}(c-noMtNames-maxMt);
                    bilder(:,:,:,c)= allCodes{nNoMtCodes + 2}(addsigAscans,single(img_start'),
                                single(posRecs'),single(posSens'),single(speed),single(pixDensity),
                                intAscan,imagesforaddsig32',imagesforaddsig);
                end
        catch e
            fprintf('CRASHED\n');
            fprintf(1,'The identifier was:\n%s',e.identifier);
            fprintf(1,'There was an error! The message was:\n%s\n',e.message);
        end
    end
    for c = 1:size(codeNames,2)
        figure('Name',codeNames(c));
        [M,I] = max(bilder(:,:,:,c));
        [M2,I2] = max(M);
        [M3,I3] = max(M2);
        imagesc([0,100],[0,100],bilder(:,:,I3,c))
        %                  bildsnr = max(data(:))/std(data(:));
        %                  erwsnr= sqrt(count)*2/mean(std(ascan));
        title(['Maximaler Wert auf ', num2str(I3*resint),' | ',num2str(I2(I3)*resint),' |
                    ',num2str(I(I2(I3))*resint),' snr: ',num2str(M3/mean(mean(mean(bilder(:,:,:,c))))) )]);
        xlabel('x in Meter');
        ylabel('y in Meter');
    end
    c=1;
    vergleichsbild = abs(bilder(:,:,:,c)-bilder(:,:,:,c+8));
    figure('Name',[codeNames(c)+' - '+codeNames(c+8)]);
    [M,I] = max(vergleichsbild);
    [M2,I2] = max(M);
    [M3,I3] = max(M2);
    imagesc([0,100],[0,100],vergleichsbild(:,:,I3))
    %                  bildsnr = max(data(:))/std(data(:));
    %                  erwsnr= sqrt(count)*2/mean(std(ascan));
    title([codeNames(c)+' - '+codeNames(c+8)+ '  |  Maximaler Wert auf '+ num2str(I3*resint)+' |
                '+num2str(I2(I3)*resint)+' | '+num2str(I(I2(I3))*resint)]);
    xlabel('x in Pixel');
    ylabel('y in Pixel');
end
```

10 Literatur

[1] DKFZ. Brustkrebs: Diagnose, Therapie und Nachsorge. https://www.krebsinformationsdienst.de/tumorarten/brustkrebs/index.php, Juli 2020.

[2] Mein Krebsratgeber. Brustkrebs. https://www.krebsratgeber.de/krebsarten/brustkrebs, 29. Januar 2020.

[3] Robert Koch Institut. Bericht zum Krebsgeschehen in Deutschland 2016, 2016.

[4] DKFZ. Mammographie zur Früherkennung von Brustkrebs. https://www.krebsinformationsdienst.de/tumorarten/brustkrebs/mammographie-frueherkennung.php, Dezember 2018.

[5] Wikipedia. Piezoelektrizität. https://de.wikipedia.org/wiki/Piezoelektrizität, Juli 2020.

[6] M.; Dapp R.; Hopp T.; Kaiser W.A.; Gemmeke H. Ruiter, N.V.; Zapf. First Results of a Clinical Study with3D Ultrasound Computer Tomography, chapter 2013 IEEE International Ultrasonics Symposium, Prague. 21-25 July 2013.

[7] T.; Zapf M.; Kaiser C.; Ruiter N.V. Gemmeke, H.; Hopp. 3D ultrasound computer tomography: Hardware setup, reconstruction methods and first clinical results, volume 873, chapter Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, pages 59–65. 21 November 2017.

[8] R. Dapp. Abbildungsmethoden für die Brust mit einem 3D-Ultraschall-Computertomographen, chapter Dissertation am Karlsruher Institut für Technologie. 2013.

[9] Ag2gaeh. Ellipse: Gärtnerkonstruktion. https://de.wikipedia.org/wiki/Datei:Ellipse-gaertner-k.svg, April 2020.

[10] Wikipedia. Mmx (instruction set). https://en.wikipedia.org/wiki/MMX_(instruction_set), Juli 2020.

[11] Wikipedia. Streaming SIMD Extensions. https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions, Juli 2020.

[12] Wikipedia. Advanced Vector Extensions. https://en.wikipedia.org/wiki/Advanced_Vector_Extensions#Advanced_Vector_Extensions, Juli 2020.

[13] Chris Lomont. Introduction to Intel® Advanced Vector Extensions. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html, Juni 2011.

[14] Michael Zapf. KIT-3DUSCT/SAFT-CPU. https://github.com/KIT-3DUSCT/SAFT-CPU, 2016.

[15] Wikipedia. Double-precision floating-point format. https://en.wikipedia.org/wiki/Double-precision_floating-point_format, Juli 2020.

[16] Wikipedia. Single-precision floating-point format. https://en.wikipedia.org/wiki/Single-precision_floating-point_format, Juli 2020.

[17] Agner Fog. Technical University of Denmark. Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd,and via cpus. https://www.agner.org/optimize/instruction_tables.pdf, 2019-08-15.

11 Selbstständigkeitserklärung

Hiermit versichern wir, dass wir diese Arbeit unter der Beratung durch Michael Zapf vom Institut für Prozessdatenverarbeitung und Elektronik am Karlsruher Institut für Technologie, Anke Richert und Paul Bischof, Kursleiter am Hector-Seminars, selbstständig verfasst haben und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht wurden.

Paul Schillinger	Leon von Berg
Untergrombach	Ettlingen