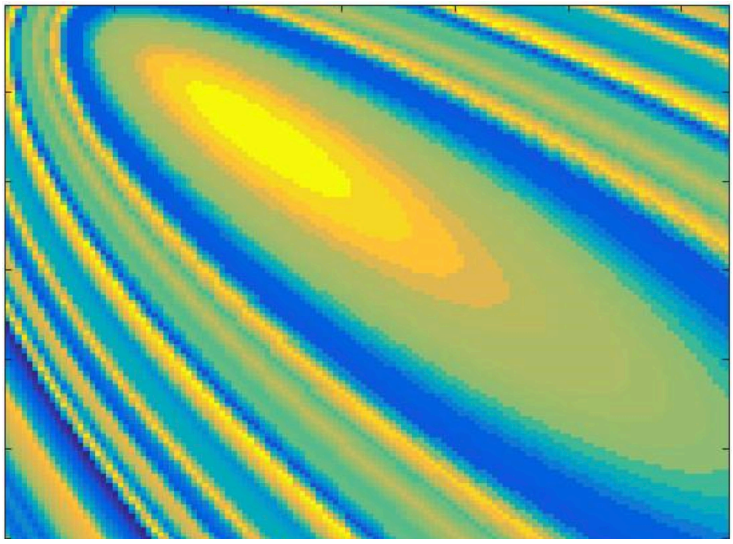
Hector Seminar

Karlsruher Institut für Technologie

SAFT CPU-Cacheoptimierung



Abschlussbericht der Kooperationsphase 2018/19

Durchgeführt am Institut für Prozessdatenverarbeitung und Elektronik (IPE)
des Karlsruher Instituts für Technologie (KIT)

Betreuer: Michael Zapf

Peter Zwick

Dresdener Str. 3

76676 Graben-Neudorf

Bastian Eckhardt

Im Unterfeld 39

76297 Stutensee

Inhaltsverzeichnis

Abstract

Breast cancer is the most frequent form of cancer for women. Most important for a successful treatment is the early detection of tumor cells. The common diagnosis method uses mammography, which is an X-ray based method that exposes the patient to ionizing radiation. Therefore, a different method would be preferred, which does not increase the risk for cancer. One option is the Ultra-Sound Computer Tomography (USCT), which recently gained increased interest worldwide due to availability of new technologies like compact ultrasound transducers and high-power computing resources. The KIT performs research in this field as well, based on the Synthetic Aperture Focusing Technique (SAFT), which requires a very high computational effort and uses large amounts of data. In this work, methods to optimize cache usage have been developed and tested to reduce the computational time of the software. To achieve this goal the picture that needs to be calculated is split into small parts that individually fit into the cache. To optimize performance, the software utilizes multithreading. Since creating new threads is quite heavy on performance we had the different threads take new packages from a global list instead of initializing them with a single package. This also allows for much better partitioning because the faster a core calculates the data the more packages it will work on. The code developed in this work is running in Matlab and can be used for further optimization.

1 Einleitung

In Deutschland erhalten jedes Jahr 69.000 Frauen die Erstdiagnose Brustkrebs [DKFZ]. Damit ist Brustkrebs die häufigste Art einer Krebserkrankung bei Frauen [KREB]. Fast drei von zehn betroffenen Frauen haben ihr 55. Lebensjahr noch nicht erreicht, wenn sie die Diagnose erhalten [DFKZ]. Der Bericht zum Krebsgeschehen in Deutschland 2016 des Robert Koch Instituts zeigt auf, dass die Zahl der Brustkrebserkrankungen in den kommenden Jahren weiter steigen wird [RKI16].

Wesentlich für eine erfolgreiche Behandlung ist eine frühzeitige Erkennung der Krankheit. Zur Diagnose wird heute vor allem die Mammographie verwendet [DKH]. Da dazu ein Röntgengerät eingesetzt wird, setzt sich die Patientin bei jeder dieser Untersuchungen einer Dosis schädlicher Röntgenstrahlung aus. Um dieses Problem zu beseitigen, wird seit langem nach Alternativen gesucht. Eine Möglichkeit ist der Einsatz von Ultraschall (Mammosonographie). Das dabei genutzte Verfahren muss allerdings noch verbessert werden, um gleiche Ergebnisse schneller, optimalerweise in Echtzeit, zu bekommen.

1.1 Ultraschallmesstechnik zur Brustkrebserkennung

Die Entdeckung des piezoelektrischen Effekts durch die Brüder Jacques und Pierre Curie im Jahr 1880 und die Erfindung piezoelektrischer Ultraschallwandler in den 1950er Jahren machte es möglich, Ultraschallsignale elektrisch zu erzeugen und auch umgekehrt zu detektieren [Wik1]. Dies ermöglichte den Einsatz der Ultraschallmesstechnik in vielen Gebieten. Bei der Ultraschallmesstechnik in der Medizin werden Ultraschallpulse von einem Punkt aus in den Körper gesendet. Über einen Sensor werden die Pulse beim späteren Austreten aus dem Körper wieder detektiert. Gemessen wird die Laufzeit sowie die Dämpfung der Ultraschallsignale. Da Tumore eine etwas andere Dichte aufweisen als das gesunde Brustgewebe, kommt es an den Übergängen zur Reflexion und Streuung der Ultraschallwellen. Aus den gewonnenen Daten kann über komplexe Rechenverfahren die Position auch kleiner Tumore bestimmt werden. Dazu muss man allerdings die Ultraschallpulse nacheinander von vielen unterschiedlichen Stellen aus auf das Objekt strahlen und die Signale an vielen verschiedenen Stellen abtasten. Die Idee des Verfahrens ist alt, sodass die Berechnungsmethoden für die Ultraschallbilder schon 1979 veröffentlicht wurden [Nor79]. Die Herstellung der Sender und Empfänger war damals jedoch noch sehr kompliziert, sodass z. B. vorgeschlagen wurde, nur einen Sender und einen Empfänger zu verwenden und das Objekt um die eigene Achse zu drehen [Sch78]. Eine ähnliche Beschreibung findet sich in [Kak88]. Moderne Systeme arbeiten mit einer Vielzahl von Sendern und Empfängern, die elektronisch gesteuert werden können, was die Aufnahme von einer großen Zahl an Messwerten in kurzer Zeit ermöglicht. Die Verarbeitung dieser großen Datenmengen ist damit allerdings zum Problem geworden.

1.2 Das USCT-System

Im Projekt USCT (engl. für Ultra-Sound Computer Tomography) am Institut für Prozessdatenverarbeitung und Elektronik des Karlsruher Instituts für Technologie wurde ein Verfahren zur Brustkrebserkennung mithilfe von Ultraschall entwickelt. Das

Untersuchungsgerät, das am IPE gebaut wurde, ist in Abbildung 1 in [Gem17] in zu sehen.

Es besteht aus 157 Ultraschallwandlern, die um die Brust herum angeordnet sind. Diese senden nacheinander jeweils einen Ultraschallpuls aus. Die reflektierten und gedämpften Signale werden von den anderen, umgebenden Wandlern nach dem Durchtritt durch das Gewebe gemessen. Dadurch bekommt man für jedes mögliche Paar von Wandlern eine Messung, den so genannten A-Scan (Abbildung 2 in [Sto05]).

Um aus den gewonnenen Messdaten ein dreidimensionales Bild zu berechnen, werden im Prinzip die "Spitzen des A-Scans", d. h. die positiven Maximalwerte der Amplitude, verwendet. Zu diesen Zeitpunkten wurde besonders viel Leistung des Pulses empfangen, d. h. es muss z. B. über Reflexion, einen Weg zwischen Sender und Empfänger geben, der dieser Signallaufzeit entspricht (Abbildung 2.3 in [Dap13]).

Da man die Ausbreitungsgeschwindigkeit des Schalls kennt, kann man daraus ein Ellipsoid generieren, auf dem sich der Reflexionspunkt befinden muss. Dabei nutzt man aus, dass meist genau eine Reflexion des Ultraschalls im Gewebe stattfindet (Abbildung 3.8 in [Dap13]). Aufgrund der großen Zahl von Sendern und Empfängern gibt es am Ende eine große Zahl an Ellipsoiden. Berechnet man die Punkte im Raum, an denen sich besonders viele dieser Ellipsoide schneiden, hat man die Reflexionsstellen lokalisiert. Da der A-Scan viele Peaks enthält und das Ergebnis der Bildgebung ein gerastertes Bild ist, werden die Schnittpunkte der Ellipsoide nicht über analytische Formeln berechnet. Stattdessen geht man wie folgt vor:

Zuerst wird festgelegt, wie genau man das Bild auflösen möchte. Man erhält dann, da das Bild 3-dimensional ist, einen Quader, bei dem jeder Voxel (3D-Bildpunkt) einen Wert hat, wie bei einem normalen digitalen Bild (Abbildung 1). Nun wird für jeden Punkt des A-Scans über einem Schwellwert ein Ellipsoid berechnet und alle Voxel, die dieser Ellipsoid „berührt“ werden mit dem Wert des A-Scans „gefüllt“, d. h. dieser Wert wird zu dem vorhandenen Wert des Voxels addiert. Führt man dies für einen A-Scan mit einem deutlichen Peak durch und legt dann einen Querschnitt durch das Bild, erkennt man dort deutlich die Ellipsen. Führt man diese Berechnung nun für alle A-Scans durch erscheinen die Schnittpunkte der Ellipsoide, was den Veränderungen im Gewebe entspricht.

Je mehr Messwerte bei der Bildgebung benutzt und je größer die Auflösung des Bildes wird, desto mehr muss logischerweise auch berechnet werden. Der am IPE entwickelte Bildgebungsalgorithmus nennt sich SAFT (Synthetic Aperture Focusing Technique). Durch die 3-dimensionale Darstellung der Bilder, erhöht sich der Rechenaufwand, wenn man die Seiten verdoppelt auf das 8-fache. Hier wird dann der Cache zu klein, um alle Daten speichern zu können. Zur Einordnung: Die erstellten Bilder haben Größen bis zu mehreren GB, wohingegen selbst in den größten, aber langsamsten Cache nur wenige MB passen. Das Bild ist also bis zu 1000 Mal größer als der Cache. Der Cache ist ein kleiner prozessornaher Speicher auf dem gleichen Chip wie der Prozessor selbst, aus dem Daten sehr viel schneller geladen und bearbeitet werden können. Dadurch kann die Geschwindigkeit der Rechnungen selbst deutlich erhöht werden.

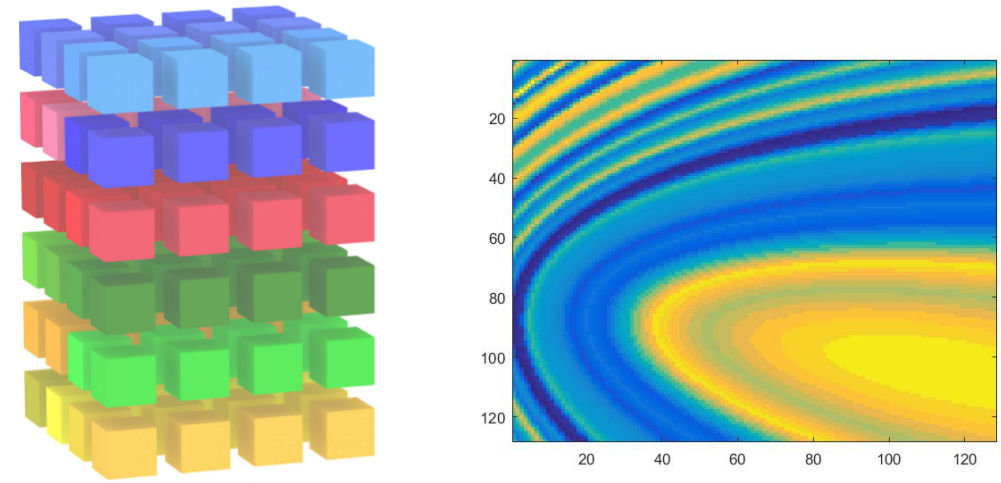


Abbildung 1: Beispiel eines 3D-Bildes, d. h. Quader mit Farbwerten in jedem Voxel (links) und Schnitt durch ein 3D-Bild mit Ellipsoid (rechts)

Aus den berechneten 3D-Bildern kann ein Arzt dann Schlüsse auf die Existenz, Größe und den Ort von Tumoren ziehen. Eine Messung mit dem USCT des KIT an einer Patientin mit Brustkrebs im Vergleich zu einer Aufnahme mit Magnetresonanztomographie ist in Abbildung 2 in [Gem17] zu sehen. Mit der Maschine wurden 2013 auch schon erste klinische Tests durchgeführt [Rui13].

Das IPE arbeitet bereits seit längerem an der Geschwindigkeitsoptimierung durch andere Methoden [Bir16]. Die 3D-Bilder des Geräts haben Größen von bis zu mehreren GB und werden bisher als Ganzes im Speicher geladen und berechnet. Selbst der größte, aber langsamste Cache ist dann viel kleiner als die Bilder, die berechnet werden. Deshalb müssen sie im normalen Speicher angelegt werden was ein Grund für die lange Rechenzeit ist. Die jeweils berechneten Bildpunkte müssen viele Male in den Cache geladen werden, da sie meist bevor sie wiederverwendet werden, von anderen Bildpunkten überschrieben werden.
In der jetzigen Software-Version schafft ein Laptop mit einem Intel Core i5-6200u Prozessor und einer Taktfrequenz von 2,30 GHz ca. $2*10^8$ Voxel pro Sekunde für einen Ellipsoid. Legt man eine Bildgröße von 1000x1000x1000 Voxeln zugrunde und die 157 Ultraschallwandler so ergibt sich eine Gesamtrechenzeit von ca. $1e9/2e8*157^2$ s = 34,23 h.

1.3 Aufgabenbeschreibung

Bei der vollständigen Messung d. h. der Messung aller Kombination von jedem Empfänger mit jeweils jedem Sender entstehen sehr große Datenmengen, welche verarbeitet und in Bilder umgewandelt werden müssen. Damit der Verarbeitungsalgorithmus praktisch angewandt werden kann, 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, die 3D-Bilder so zu unterteilen, dass die gerade berechneten Teile in einen Cache passen und die Berechnung damit beschleunigt wird. Dies ist grundsätzlich möglich, da die Berechnungen der einzelnen Voxel der Bilder problemlos nacheinander durchgeführt werden können.

2 Material und Methoden

Ausgehend von einem bereits bestehenden Programm, wurden verschiedene Software-Tools zur Optimierung eingesetzt. Das Hauptproblem war, dass die Berechnungssoftware des IPE schon in vielen Teilen aus unterschiedlichen Programmiersprachen bestand. Die Software musste in unterschiedlichen Konfigurationen auf dem PC funktionieren damit die Rechenzeit gemessen und die Ergebnisse verglichen werden können. Die ursprünglich eingesetzte Software Octave erwies sich als ungeeignet, da sie nicht mit der am IPE verwendeten Berechnungs-Software funktionierte. Im Folgenden wurde daher auf Matlab eingesetzt.

Auf Basis von Multi-Threading und Cache-Optimierung wurden im Projekt eigene Methoden zur Optimierung der vorhandenen Software entwickelt und getestet.

Im dritten und wichtigsten Teil des vorliegenden Kapitels erklären wir die speziellen, von uns entwickelten und getesteten Methoden zur Optimierung der IPE-Software.

2.1 Verwendeter Computer

Für das Projekt wurde vom IPE folgendes Gerät bereitgestellt:

lenovo tp e560 mit einem intel core i5-6200u cpu @ 2.30ghz

Damit können 4 Threads gleichzeitig bearbeitet werden, der Level 3 Cache ist 3 MB groß und er hat 8 GB RAM.

2.2 Ansteuerung und Auswertung mit Octave und GNU Tools

Als Entwicklungsumgebung wurde Octave gewählt, da es vielfältige Anwendungs-möglichkeiten bietet und auf beliebiger Hardware und Betriebssystemen läuft. Octave funktioniert ähnlich wie das kommerzielle Matlab, ist jedoch ein freies Programm zur Visualisierung und Verarbeitung von Daten. Mithilfe eines zweizeiligen Codes kann man aus einer Liste mit Daten einfach Graphen erstellen. Diese ähneln einem Diagramm aus einer Excel Tabelle, nur dass die Liste der Daten automatisch vom Programm gefüllt wird, z. B. durch das Speichern von Zeitmessungen, und als Graph angezeigt wird.

Octave bietet auch weitere Möglichkeiten, welche die Nutzung vereinfachen. So können selbst geschriebene Programme als Funktionen im Code ausführen und die Ausgabe weiterverwenden. In dem Projekt sollte der Algorithmus ausgeführt werden. Damit kann man die Korrektheit der erstellten Bilder zwischen verschiedenen Versionen des Algorithmus überprüfen und die benötigte Rechenzeit zweier Versionen vergleichen. Dadurch kann man die Veränderung der Performance messen.

Trotz der beschriebenen Vorteile konnte Octave nicht weiterverwendet werden, da das vorhandene Berechnungsprogramm des IPE in Octave nicht kompiliert werden kann. Die Dokumentation des Projekts, die erklärt, wie man dieses kompiliert, ist für die Kompilation unter Linux und Octave mehr als 5 Jahre alt und funktioniert nicht mehr. Bestimmte Flags und Zusatzinfos, die bei der Kompilation angegeben werden müssen, werden nicht erkannt oder sind nicht mehr vorhanden. Aufgrund der bestehenden Probleme wurde der Code in

der Kommandozeile mit dem GNU-Compiler gcc kompiliert. Auch dieser Ansatz hatte nur begrenzten Erfolg. Es ist eine kompilierte Datei entstanden, die jedoch nicht nutzbar war. Es gab keinen entry point, d. h. wenn es ausführt wurde passiert nichts, da der Computer nicht wusste, wo das Programm beginnt und es so nicht starten konnte.

Das Problem konnte auch mithilfe des GNU-Debuggers gdb nicht gelöst werden. Ohne entry point startete der Code erst gar nicht, sodass nichts über die Probleme in Erfahrung gebracht werden konnte. Sogar Octave während des Kompilierens zu debuggen war nicht möglich, da innerhalb von Octave mehrere Prozesse, die man nicht gleichzeitig beobachten konnte, ablaufen.

2.3 Ansteuerung und Auswertung mit Matlab

Wie Octave, ist auch Matlab ein Programm zur Verarbeitung und Visualisierung von Daten. Dieses wurde jedoch auch bei der ursprünglichen Erstellung des Algorithmus verwendet.

2.3.1 Kompilieren in Matlab

In der projekteigenen Dokumentation zur Kompilation des C-Codes gibt es, wie für Linux und Octave, eine Beschreibung für Windows und Matlab. In diesem Fall lief alles problemlos. Das Kommando konnte direkt in Matlab kopiert und das Programm kompiliert werden.

2.3.2 Aufruf des Codes in Matlab

Zu Beginn wurden erste Tests der Bildgebung durchgeführt, um dabei Matlab und die Parameter der Bildgebung zu verstehen. Beim Programmaufruf müssen die folgenden 10 Parameter nacheinander eingegeben werden:

1) Ascan: Messwerte (siehe Abschnitt 1.2)
2) Pix_vect: Vektor zur Starposition des Bildes im Raum
3) rec_pos: relative Position des Receiver im Raum
4) send_pos: relative Position des Transmitter im Raum
5) speed: Ultraschall Geschwindigkeit im Medium
6) res: Auflösung des Bildes (Abstand der Pixel/Voxel)
7) timeint: nicht relevant für Projekt
8) IMAGE_XYZ: x-y-z-Dimensionen Pixelzahlen
9) IMAGE_SUM: zu beschreibendes Bild / Eingangsbild
10) Cache Size: Cachegröße in Bit

Hierbei kann man auch einzelne Ellipsoide erstellen, indem man anstatt eines normalen A-Scans nur eine einzelne Zeit eingibt. Abbildung 2 zeigt den Querschnitt durch ein solches Ellipsoid. Zur Berechnung eines gesamten 3D-Bilds muss das obige Programm für alle A-Scans, d. h. alle Paarungen der Ultraschall-Wandler ausgeführt werden. Zum Evaluieren der Cache-Optimierung war allerdings ein einzelner Aufruf ausreichend.

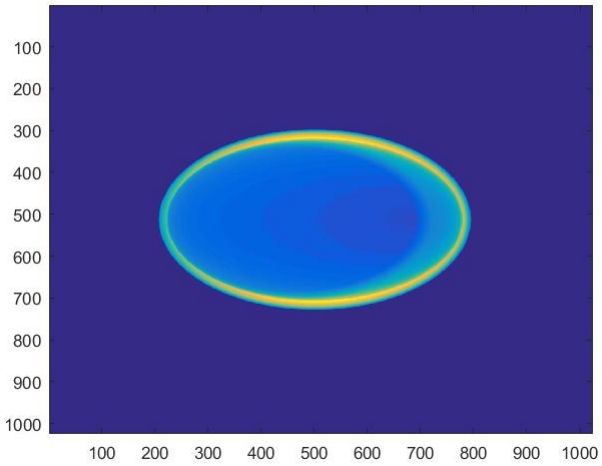


Abbildung 2: Querschnitt durch Ellipsoid, berechnet durch Eingabe einer einzelnen Zeit

2.4 Multithreading

Als Multithreading bezeichnet man es, wenn ein Prozess in kleinere Teile aufteilt und diese Threads (Ausführungsstränge) innerhalb eines Prozesses gleichzeitig bearbeitet werden. Heutige Multicoresysteme sind tatsächlich in der Lage, mehrere Threads gleichzeitig zu bearbeiten [Wik2]. Hier laufen die Threads üblicherweise auf den einzelnen Kernen des Prozessors parallel ab.

2.5 Cache-Optimierung

Die Optimierung des Cache ist eine wichtige Methode zur Beschleunigung von Software insbesondere im Zusammenhang mit Multithreading. Der Cache ist ein sehr kleiner Speicher im Computer der allerdings extrem schnell vom Prozessor ausgelesen oder beschrieben werden kann. Daten, die bei einer Berechnung benötigt werden, werden deshalb immer erst vom Hauptspeicher in den Cache geladen und dann verarbeitet (Abbildung 3). Bei der Cache-Optimierung versucht man nun, diesen Vorgang zu optimieren, da der PC den Cache von sich aus nicht unbedingt optimal nutzt. Bezogen auf das Programm des IPE ist eine Möglichkeit, das Programm so zu optimieren, dass die Bildpunkte möglichst selten in den Cache geladen werden und dann immer gleich alle Berechnungen dazu durchgeführt werden.

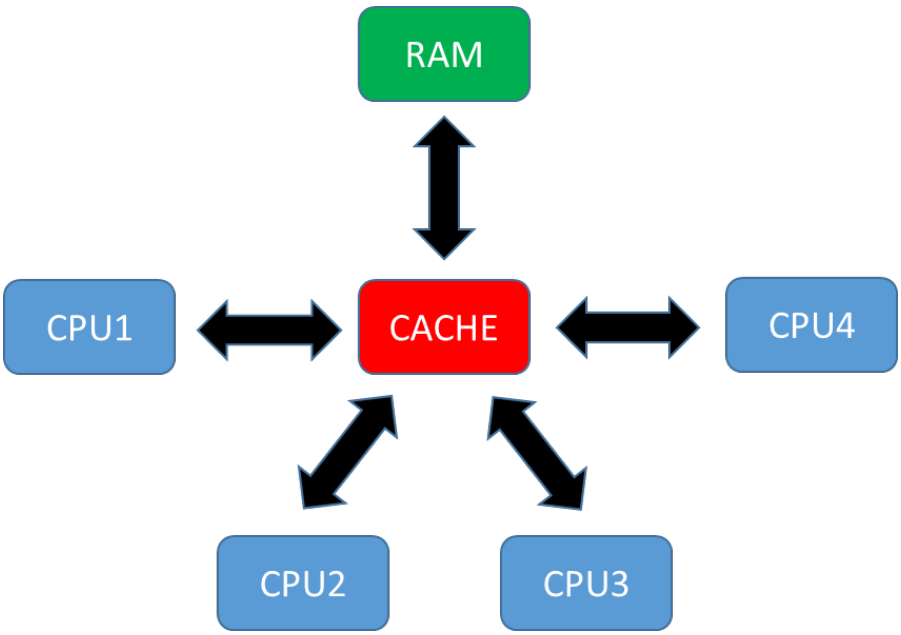


Abbildung 3: Multicoresystem mit gemeinsamem Cache

2.6 Verbesserte Remainder Distribution

Der erste Ansatz war es, die Ebenen des zu erstellenden Bildes besser auf die einzelnen Prozessorkerne aufzuteilen. In der alten Version wurde das Bild in gleichgroße Stücke geteilt und auf die Kerne verteilt. Die übriggebliebenen Ebenen hat ein Kern zusätzlich zu den bereits zugeteilten berechnet. In einer ersten Verbesserung wurden diese bei der Aufteilung übriggebliebenen Ebenen einzeln auf die verschiedenen Kerne aufgeteilt. Wenn nämlich mehrere Kerne die gleiche Menge an Daten berechnen wie ein einzelner Kern, dann wird die Laufzeit durch die Anzahl der Kerne dividiert.

Sofern die Bilder sich gut auf die einzelnen Kerne aufteilen lassen und kaum ein Rest bleibt, kommt es nur zu einer geringen Optimierung. Wenn die Anzahl der Ebenen sich jedoch schlecht auf die Kerne verteilen lässt und ein großer Rest bleibt, so ist die Verbesserung deutlich zu erkennen. Je größer die Ebenen der Bilder werden, umso deutlicher ist der Unterschied, da es dann auch sehr viel länger braucht, um eine einzelne Ebene zu berechnen. Bei der alten Version des Codes würde beispielsweise ein Bild aus sieben Ebenen so auf vier Kerne verteilt werden, dass einer der Kerne vier Ebenen berechnen müsste und die restlichen jeweils nur eine Ebene (Abbildung 4, links). Bei der neuen Version des Codes würden diese Ebenen gleichmäßiger verteilt werden (Abbildung 4, rechts), was in diesem Beispiel zu einer Halbierung der Berechnungsdauer führen würde.

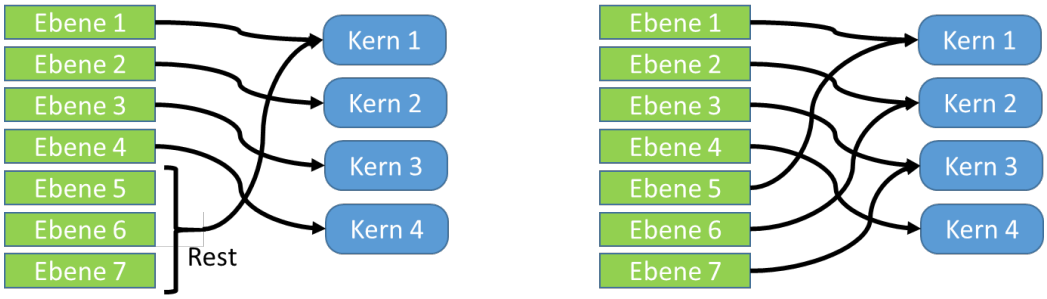


Abbildung 4: Ursprüngliche Methode (links) im Vergleich zur Methode der verbesserten Remainder Distribution (rechts)

2.7 Partitionierung über Z in Cachegröße

Für die erste Cacheoptimierung wurde das Bild über die Ebenen in kleine Teile unterteilt, die einzeln vollständig in den Cache passen. Diese Teile wurden mit dem ursprünglichen Algorithmus berechnet, der die einzelnen Bildteile auf die Kerne aufteilte. Der bestehende, „alte“ Algorithmus wurde demnach mehrmals ausgeführt und die, zu berechnenden Bilder passten alle in den Cache. Problematisch war, dass die Threads nach jedem Bildteil terminiert und danach neu initialisiert werden mussten. Das ist ein recht langsamer Prozess, wodurch sich die Performance deutlich verschlechterte. Außerdem werden nicht alle Daten berechnet, da der Mainthread nicht terminiert werden kann, bevor das Bild vollständig zusammengesetzt ist. Da er jedoch bei jedem Bildteil neue Daten berechnet, gehen die vorhergehenden Daten verloren und das resultierende Gesamtbild wird verfälscht. Aus diesem Grund wurde der Ansatz nicht weiterverfolgt.

2.8 Auslagerung der Aufteilung und einmaliges Erstellen der Threads

Bei der finalen Version der Cacheoptimierung werden die Threads nur einmal initialisiert und terminiert. Da sie aber immer noch nur Bildteile in Cachegröße berechnen sollen, speichern die Threads ihre Ergebnisse selbst und „nehmen“ sich selbstständig ein neues „Paket“. Dafür werden vor der Thread-Erstellung Pakete generiert, die höchstens so groß sind, wie die Cachegröße geteilt durch die Anzahl der Kerne. Die variablen Daten, welche zur Bilderstellung benötigt werden sind in einer globalen Liste abgespeichert auf welche die einzelnen Threads zugreifen können. Damit keine Pakete doppelt bearbeitet werden, zählen die Threads einen globalen Counter hoch, der angibt, welches Paket als nächstes berechnet werden muss. Um zu verhindern, dass zwei Threads im selben Moment auf den Counter zugreifen und dadurch vielleicht das selbe Paket bearbeiten wollen, ist dieser Teil der Thread-Funktion durch ein sogenanntes Mutex Lock gesichert. Durch diesen Algorithmus ist auch das Multithreading viel leichter geworden, da die einzelnen Bildteile keinem Prozessorkern zugeordnet werden müssen. Jeder Thread „nimmt“ sich, sobald er den vorherigen Prozess terminiert hat, ein neues Paket aus der Liste. So können auch einer oder wenige Kerne mehr Pakete bearbeiten als die anderen Kerne, wenn diese durch andere Prozesse verlangsamt werden. Außerdem wird das Bild jetzt nicht nur über die Z-Dimension aufgeteilt. Wenn eine einzelne Ebene größer als der Anteil des Caches ist, der jedem Kern zusteht, werden die Ebenen in sich auch noch über die X- und Y-Dimension aufgeteilt (Abbildung 5). Durch die „selbstständigen“ Threads wird das Bild sehr viel gerechter auf die Prozessorkerne verteilt.

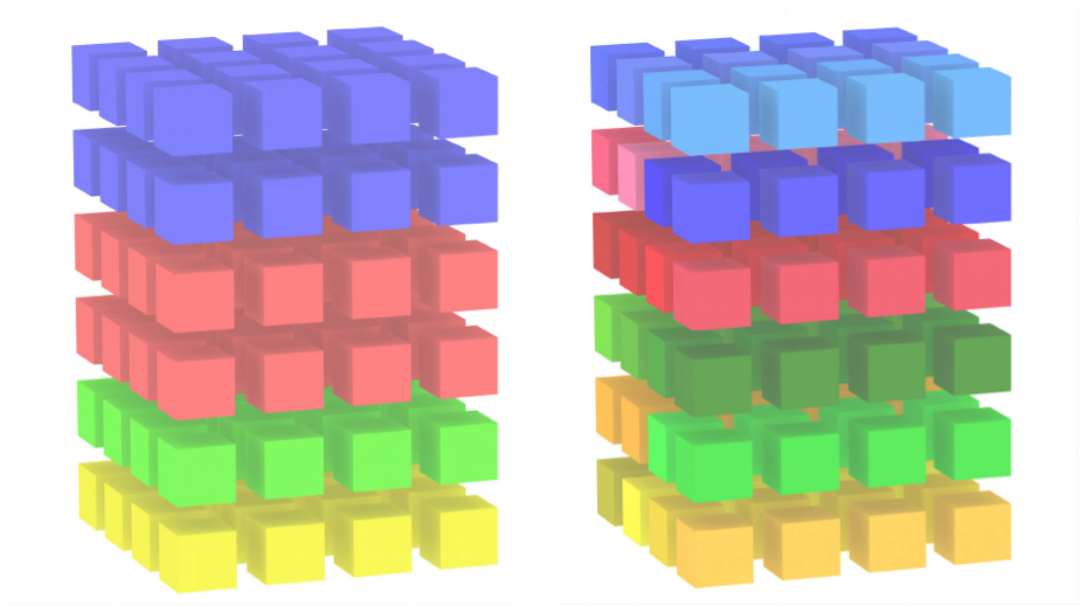


Abbildung 5: Aufteilung eines Bildes nach verbesserter Remainder Distribution (links) und nach der endgültig optimierten Version (rechts). Die Farben kennzeichnen die unterschiedlichen Kerne und die Schattierungen der Farben die unterschiedlichen Arbeitspakete

In der ursprünglichen Version hat jeder Thread zu Beginn ein großes Paket bekommen. In der optimierten Variante berechnet jeder Kern nicht nur ein einzelnes Paket, sondern mehrere kleine Pakete (P1 - P4) nacheinander (Abbildung 6). Diese sind genau so groß, dass sie in den Teil des Caches passen, der für sie vorgesehen ist, in diesem Beispiel ein Viertel der tatsächlichen Cachegröße. In der alten Version müssten die Daten über den langsameren Hauptspeicher geladen werden.

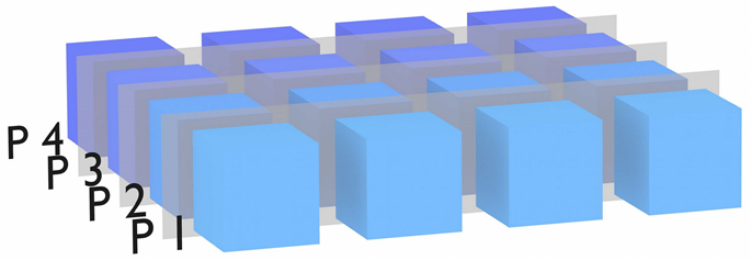


Abbildung 6: Aufteilung von 4 Paketen auf einen Kern

3 Ergebnisse und Diskussion

3.1 Test-Cases zum Erstellen von Bildern und Messung der Performance

Um testen zu können, wie gut die Verbesserungen sind, wurden Testfälle (engl. Test-Cases) generiert, mit denen die Rechengeschwindigkeit für das Erzeugen von Bildern mit dem SAFT-Algorithmus ermittelt wurde. Ein Problem ist, dass Computer während der Berechnung eines Programms noch viele andere Kleinigkeiten wie z. B. Programme zum Virenschutz laufen, die bei jeder Ausführung des Programms variieren können. Das bedeutet, dass man selbst dann jedes Mal ein anderes Ergebnis bekommt, wenn man die Rechenzeit der identischen Berechnung misst. Um dieses Problem zu beseitigen wurde jede Messung mehrfach durchgeführt, unter der Annahme, dass die Zeiten im Mittel gleich sein werden.

In der ersten Version des entwickelten Test-Cases wird jeweils ein Bild mit dem neuen und eines mit dem ursprünglichen Code erstellt. Die Zeiten, die der Code zum Erstellen der Bilder benötigt, werden in eine Liste geschrieben und am Ende in einer Grafik ausgegeben. Dies wird wiederholt durchgeführt und dann gemittelt, um zufällige Abweichungen zu beheben. Der Vorgang wird mit aufsteigenden Bildgrößen wiederholt, um verschiedene Graphen zu erhalten, jeweils für die neue Version und für die alte. Um die Genauigkeit der Ergebnisse zu überprüfen, bildet der Test-Case einmal den Median der Werte für jede Dimension und einmal den Mittelwert. Der Test-Case speichert außerdem jeweils die beiden erstellten Bilder ab und bildet die Differenz beider. Ist das Ergebnis ungleich Null, gibt der Test-Case eine Fehlermeldung zurück. In diesem Fall wären die Bilder nicht identisch, d. h. die optimierte Berechnung würde fehlerhafte Bilder erzeugen.

3.1.1 Verbesserter Test-Case ohne Verlangsamung durch den RAM

In der ersten Version des Test-Cases wurde versäumt, zu berücksichtigen, dass das Bild des neuen Algorithmus den RAM belegt und somit den alten Algorithmus verlangsamt, der danach ausgeführt wird. Dies wurde behoben, indem die Bilder nicht gespeichert, sondern erstellt und direkt wieder gelöscht wurden. Da dabei nicht mehr überprüft werden kann, ob die erstellten Bilder des alten und des neuen Codes noch identisch sind, wurde eine zweite Option implementiert, bei der immer abwechselnd der neue und der alte Code zuerst ausgeführt werden. Bei ausreichend hoher Zahl an Tests pro Dimension, ist das Ergebnis immer noch hinreichend genau. Mithilfe dieses Test-Cases lassen sich Graphiken erstellen, bei denen die erzielte Performance gegen die Bildgröße aufgetragen ist (Abbildung 7). Die vier Graphen zeigen jeweils die Mittelwerte und Median-Werte der beiden untersuchten Versionen des Algorithmus. Für steigende Bildgrößen weichen Mittelwert und Median voneinander ab, da für große Bilder mehr Simulationen nötig sind, um hinreichend genaue Ergebnisse zu erhalten.

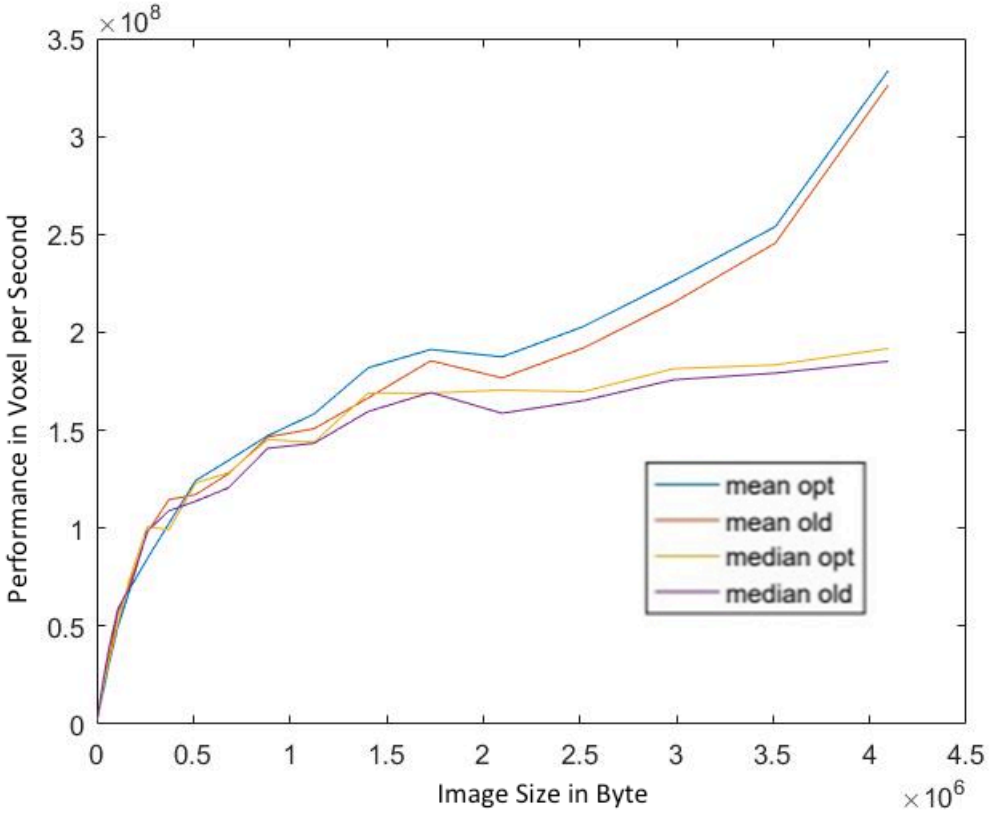


Abbildung 7: Performance-Beispiel für Test-Case ohne Verlangsamung durch RAM

3.1.2 Test-Case über maximal genutzter Cachegröße

Da im Cache während der Berechnung auch noch andere Daten neben dem zu erstellenden Bildausschnitt gespeichert werden, muss der für den Bildausschnitt nutzbare Anteil ermittelt werden. Dieser wird im Folgenden mit MGBC (Maximal Größe des für das Bild genutzten Cache) abgekürzt. Um diesen zu visualisieren, wurde ein zusätzlicher Test-Case geschrieben, der ähnlich wie der alte Test-Case funktioniert. Er lässt den Algorithmus jedoch verschiedene MGBC annehmen, ohne die gesamte Bildgröße zu verändern. Im alten Algorithmus hat dies keine Auswirkung auf die Performance. Der neu entwickelte Algorithmus hingegen sollte bei der tatsächlich verfügbaren Cachegröße des Geräts die beste Performance haben.

3.1.3 Test-Case über MGBC und Bildgröße

Für einen umfassenden Perfomancevergleich wurde der Test-Case über die Bildgröße mit dem über die MGBC vereint. So entstand ein Test-Case, der Bilder mit variabler Bildgröße und MGBC berechnen kann. Um die resultierenden Werte vollständig zu visualisieren, wurden die Differenzen der Mittelwerte der jeweiligen Rechenzeiten der neuen zu den alten Versionen gebildet und in einer Tabelle über die MGBC und Bildgröße dargestellt. Dies kann dann in einer farbigen Grafik (Abbildung 8) dargestellt werden. In dem vorliegenden Fall sieht man allerdings, dass hier das neue Programm grundsätzlich schlechter war als das alte.

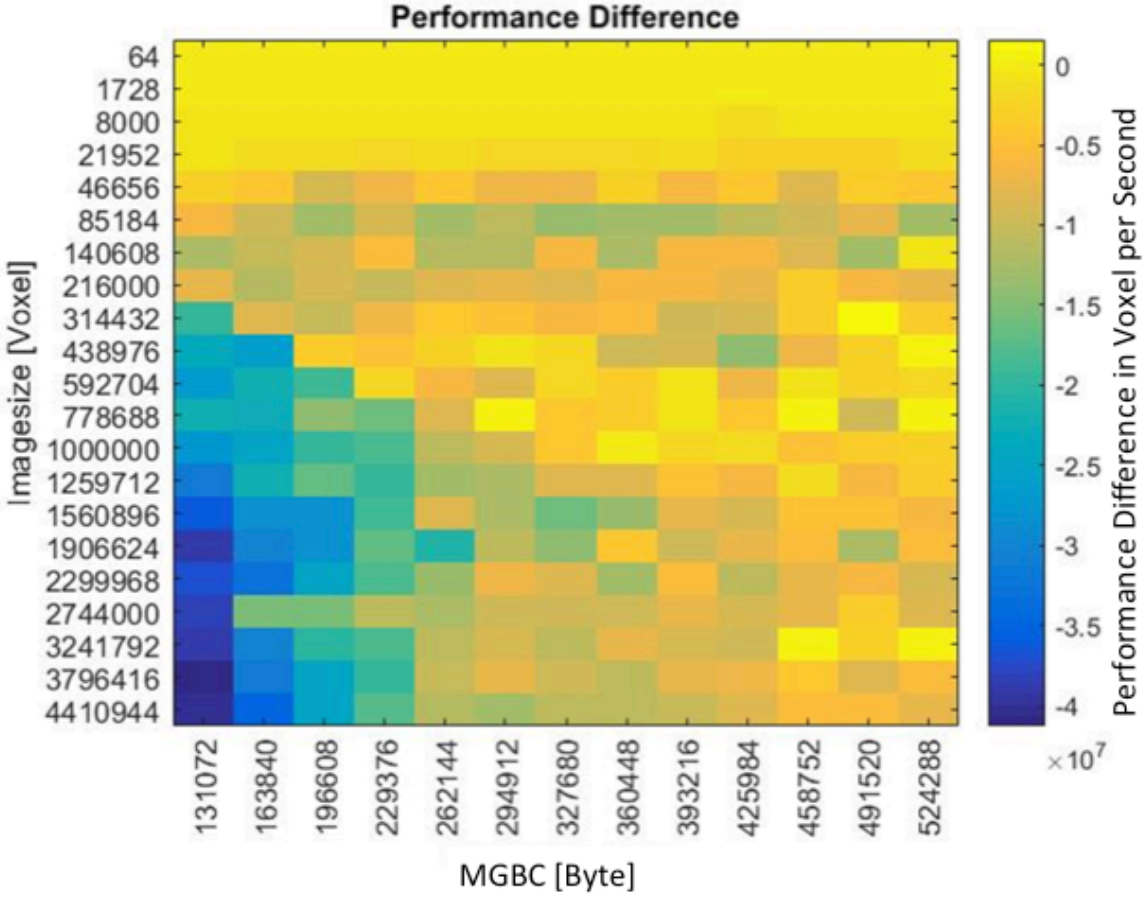


Abbildung 8: Beispiel für Test-Case für unterschiedliche MGBC und unterschiedliche Bildgrößen

3.2 Ergebnisse zur verbesserten Remainder Distribution

Durch die Verbesserung der Remainder Distribution hat sich in bestimmten Fällen die Performance deutlich verbessert. Wenn z. B. sechs Z-Ebenen genommen werden, steigt die Performance erheblich (Abbildung 9). Nimmt man mehr Z-Ebenen, ist dieser Unterschied nicht so deutlich bemerkbar, da nur der Remainder besser verteilt wird. Bei vielen Z-Ebenen ist der Remainder im Verhältnis zu den normal verteilten Ebenen sehr klein und benötigt so auch nur einen kleinen Teil der Rechenzeit.

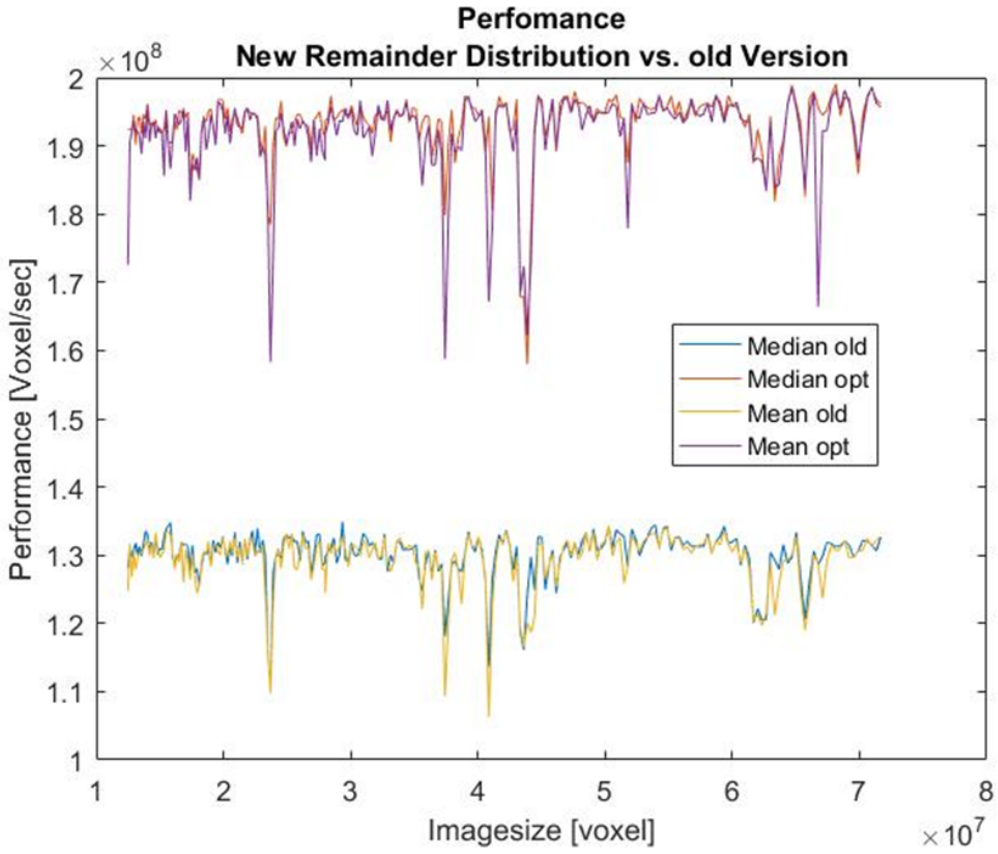


Abbildung 9: Testergebnis der verbesserten Remainder Distribution mit 6 Z-Ebenen

3.3 Ergebnisse der Cache-Optimierung

Durch die Cacheoptimierung wird das zu erstellende Bild in Teile unterteilt, die in den Cache passen. Das ermöglicht einen deutlich schnelleren Zugriff auf diese Daten, sorgt aber für einen größeren Overhead. Die ganzen Pakete müssen nämlich zuerst einmal „geschnürt“ werden und in eine Liste geschrieben werden. Dieser Prozess dauert seine Zeit.

Der in der Arbeit verwendete Cache war ca. 3 MB, d. h. 3,15e6 Byte groß (Abschnitt 2.1). Da im Cache auch immer noch andere Daten von z. B. dem Betriebssystem liegen, zeigt der neue Algorithmus kurz unter diesem Wert seine beste Performance (Abbildung 10). Die Performance der alten Version verändert sich bei den verschiedenen MGBC natürlich nicht. Die Variation der Werte zeigt nur die in Abschnitt 2 beschriebene Statistik der Messungen. Die neue Version ist in fast allen Fällen besser als die alte. Im Fall optimaler Cachenutzung konnte die Performance im vorliegenden Beispiel um bis zu 25 % erhöht werden. Der tatsächliche Prozentsatz ist immer von Bildgröße und Cachegröße abhängig. Im Folgenden wird die Bildgröße variiert und wieder die MGBC auf die vorhandene Cachegröße optimiert.

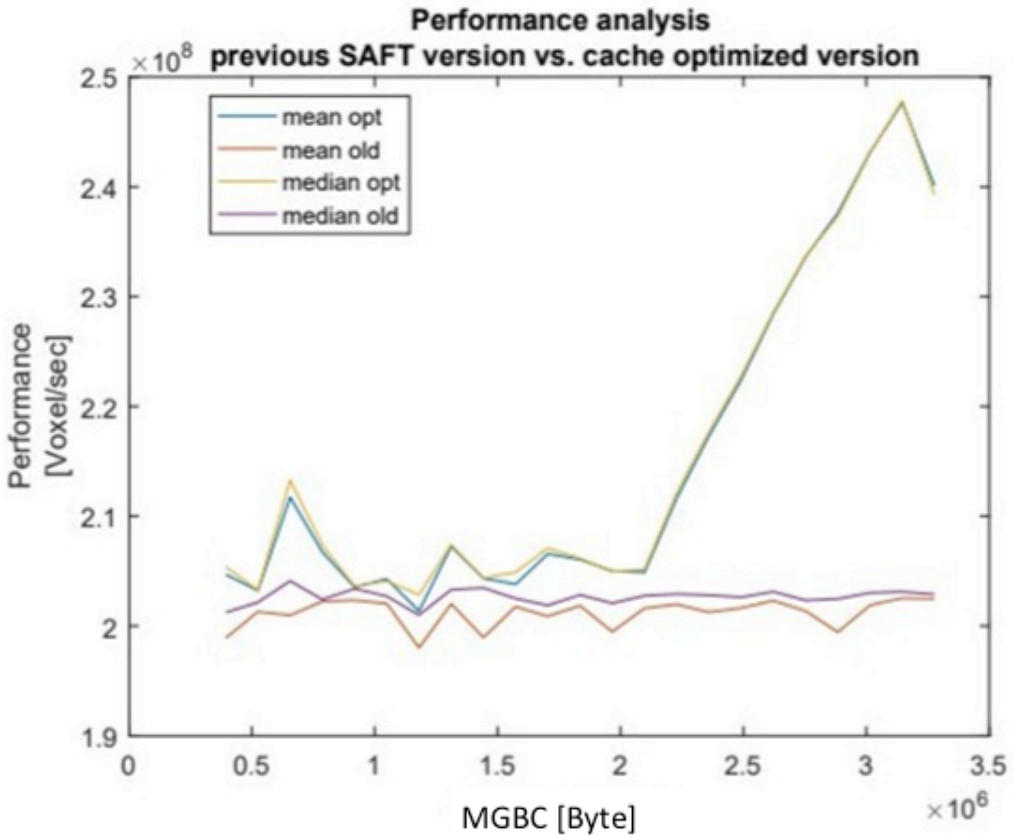


Abbildung 10: Performance bei unterschiedlichen MGBC

Betrachtet man eine Testreihe über MGBC und Bildgröße (Abbildung 11), erkennt man, dass bei kleinen MGBC und Bildgrößen die alte Version schneller ist als die cacheoptimierte. Das liegt daran, dass die cacheoptimierte Version durch die bessere Aufteilung einen größeren Overhead hat, der bei kleinen Bildern einen höheren Anteil ausmacht. Bei sehr kleinen MGBC ist die Performance schlechter, da die neue Version des Codes sehr viele kleine Pakete erstellt, wodurch der Overhead stark zunimmt. Beide Fälle sind allerdings hier irrelevant, da das Ziel die Performanceoptimierung großer Bilder war und dem Programm eine optimale MGBC mitgegeben wird. Bei optimalem MGBC und großen Bildern ist eine Verbesserung von ca. 10% erkennbar (2e7 Voxel/Sekunde Verbesserung gegenüber 2e8 Voxel/Sekunde Ausgangsperformance).

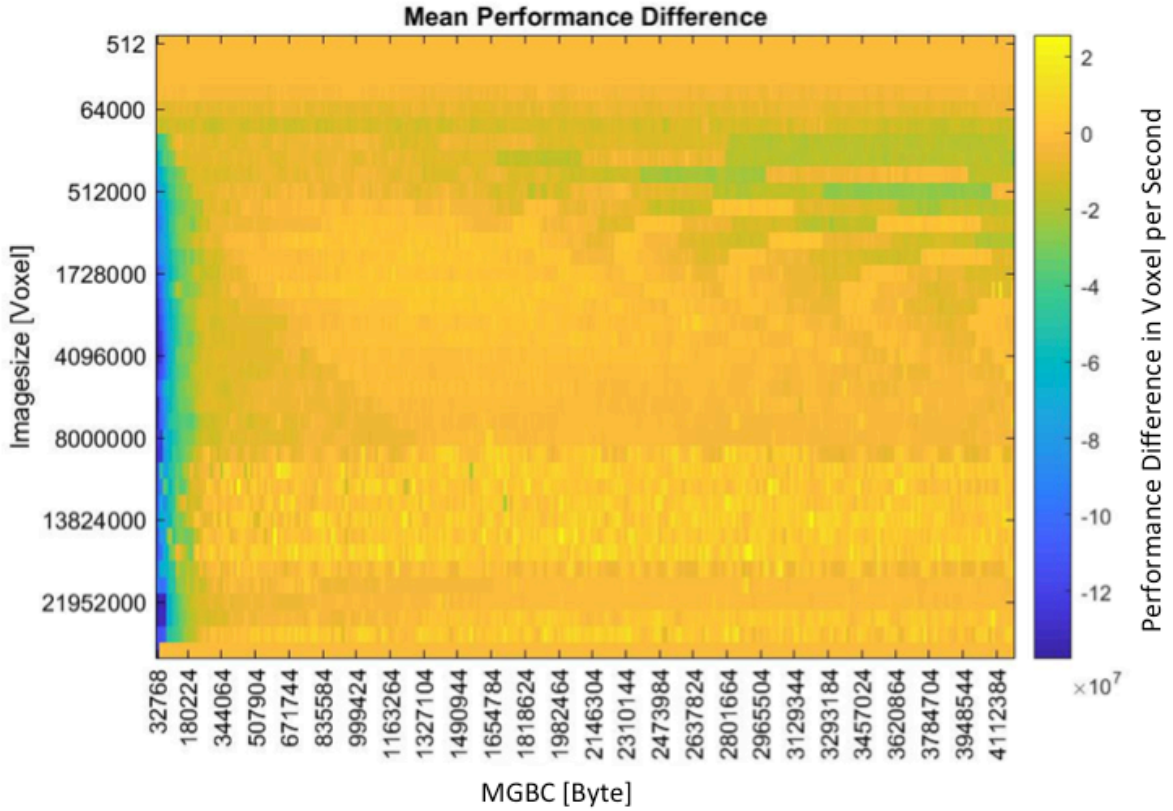


Abbildung 11: Performance bei unterschiedlichen MGBC und Bildgrößen

3.4 Fehlerdiskussion

Das Einrichten einer neuen Entwicklungsumgebung ist sehr komplex und wird umso schwerer, je komplexer das ursprüngliche Projekt. Ohne Vorkenntnisse mit der gegebenen Entwicklungsumgebung ist dies schwer möglich. Eine längere Einarbeitung wäre nötig, um mit Octave arbeiten zu können, aber da unsere Arbeitszeit begrenzt war, mussten wir auf eine andere Entwicklungsumgebung umsteigen.

Den Mainthread nicht für die tatsächliche Berechnung zu verwenden, sondern nur für das Zusammensetzen des Bildes, erleichterte uns sehr die Aufteilung auf viele kleine Pakete. Es löste auch einige Probleme, bei denen das berechnete Bild unvollständig oder falsch war.

Es hat sich als extrem hilfreich herausgestellt, immer wieder aufs Neue die Ergebnisbilder zu überprüfen. Wir sind in unserem Projekt oft in Sackgassen gelandet, da wir nicht gleich bemerkt hatten, dass unser Algorithmus keine vollkommen korrekten Bilder erstellte.

4 Fazit und Ausblick

4.1 Zusammenfassung

Um das Problem zu lösen, haben wir zuerst einen Test-Case angefertigt, um die Performance der verschiedenen Versionen zu vergleichen und um zu überprüfen, ob bei der Aufteilung Daten verloren gehen oder beschädigt werden.

Für die Aufteilung selbst haben wir eine Liste erstellt, in der wir die einzelnen Teile des Bildes klar definiert haben. Da das Programm auch Multithreading verwendet, lassen wir die einzelnen Threads sich selbstständig neue Elemente aus dieser Liste nehmen, bis sie durchgearbeitet ist. Als eine weitere Verbesserung kann das Programm das Bild nicht mehr nur über einzelne Ebenen, sondern über x-, y- und z-Dimensionen aufteilen. Dies hilft bei sehr großen Bildern kleinere Teile in Cachegröße zu erstellen. Mit diesem Optimierungsansatz haben wir die Performance um bis zu 25% verbessert. Der wirkliche Prozentsatz ist immer von Bildgröße und Cachegröße abhängig. Es konnte allerdings gezeigt werden, dass gerade bei sehr großen Bildern die Cacheoptimierung eindeutig eine Verbesserung der Rechenzeit erzielt. Dabei muss jedoch der für das Bild nutzbare Cache gegenüber dem maximal verfügbaren Cache ermittelt werden.

4.2 Ausblick

Um das Ergebnis weiter zu verbessern, könnte die Cachegröße des Gerätes in einem autoadaptiven Benchmark ermittelt werden, ähnlich der Ermittlung der Anzahl der Prozessorkerne. So müsste man diese nicht erst manuell bestimmen und beim Ausführen des Codes angeben. Eine weitere Möglichkeit wäre, die Threads nicht zu terminieren, wenn das Programm mehrmals hintereinander ausgeführt wird. Da in der Anwendung immer einige Messungen nacheinander durchgeführt werden, würde das die Performance weiter optimieren. Diese Verbesserung ist allerdings mit der pthreads.h-Bibliothek, welche momentan für das Multithreading verwendet wird, schlecht möglich. Hierzu müssten große Teile des Codes, wie z. B. die komplette thread function, neu geschrieben werden.

5 Danksagung

Unser großer Dank geht an Michael Zapf vom 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 und Dietmar Gruber für die große Unterstützung über die vergangenen Jahre. Nicht zuletzt danken wir dem Ehepaar Hector und ihrer Stiftung für die großzügige Unterstützung des Hector-Seminars.

6 Quellen

[ONCO] https://www.oncology-guide.com/ursachen/krebsstatistik/, Juli 2019.

[KREB] https://www.krebsratgeber.de/artikel/die-haeufigste-krebsart-bei-frauen, Juli 2019

[DKFZ] https://www.krebsinformationsdienst.de/tumorarten/brustkrebs/index.php, Juli 2019.

[DKH] https://www.krebshilfe.de/informieren/ueber-krebs/haeufige-krebsarten/brustkrebs/, Juli 2019.

[RKI16] Robert Koch Institut, „Bericht zum Krebsgeschehen in Deutschland 2016“, 2016.

[Nor79] Norton, S.J., Linzer, M., “Ultrasonic Reflectivity Tomography: Reconstruction with Circular Transducer Arrays”, Ultrasonic Imaging 1, 154-185, 1979.

[Wik1] https://de.wikipedia.org/wiki/Piezoelektrizität, Juli 2019.

[Sch78] Schomberg, H., „An improved approach to reconstructive ultrasound tomography“, J. Phys. D: Appl. Phys., Vol. 11, 1978.

[Kak88] Kak, A.; Slaney, M., “Principles of Computerized Tomographic Imaging”, Chapter 4.3, IEEE Press, ISBN 978-0898714944.

[Wik2] https://de.wikipedia.org/wiki/Multithreading, Juli 2019.

[Gem17] Gemmeke, H.; Hopp, T.; Zapf, M.; Kaiser, C.; Ruiter, N.V., „3D ultrasound computer tomography: Hardware setup, reconstruction methods and first clinical results”, in Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, Volume 873, 21 November 2017, Pages 59-65.

[Bir16] Birk, M.; Kretzek, E.; Figuli, P.; Weber, M.; Becker, J.: Ruiter, N.V., „High-Speed Medical Imaging in 3D Ultrasound Computer Tomography”, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 27, NO. 2, FEBRUARY 2016, pp. 455-467.

[Rui13] Ruiter, N.V.; Zapf, M.; Dapp, R.; Hopp, T.; Kaiser, W.A.; Gemmeke, H., “First Results of a Clinical Study with3D Ultrasound Computer Tomography”, 2013 IEEE International Ultrasonics Symposium, Prague, Czech Republic, 21-25 July 2013.

[Sto05] Stotzka R.; Ruiter, N.V.; Mueller, T.O.; Liu, R.; Gemmeke, H., “High resolution image reconstruction in ultrasound computer tomography using deconvolution”, in Proceedings of SPIE, Medical Imaging, 2005, San Diego, California, United States.

[Dap13] Dapp, R., “Abbildungsmethoden für die Brust mit einem 3D-Ultraschall-Computertomographen”, Dissertation am Karlsruher Institut für Technologie, 2013.

7 Anhang

Im Folgenden finden sich die erstellten Software-Pakete.

Partitioning:

```
//START OF CHANGES

unsigned int imageSize = n_Zz*n_Yz*n_Xz;
cacheSize = cacheSize / 8; //Divide because of bit/byte conversion
unsigned int yPackages = 1;
unsigned int xPackages = 1;
unsigned int zPackages = (nCores*imageSize + cacheSize - 1)/cacheSize;


if(zPackages <= n_Zz){
// ONLY Z-DIM PARTITIONING
    #ifdef addsig2vol_debug
        mexPrintf("Z-Dim partitioning\n");
    #endif
    if(zPackages < nCores && nCores < n_Zz) zPackages = nCores;
}
else{
    // Y-DIM PARTITIONING
    zPackages = n_Zz;
    yPackages = (nCores*n_Yz*n_Xz*8+cacheSize-1)/cacheSize;


    if(yPackages <= n_Yz){
        #ifdef addsig2vol_debug
            mexPrintf("Y-Dim partitioning\n");
        #endif
    }
    else{
        // X-DIM PARTITIONING
        yPackages = n_Yz;
        xPackages = (nCores*n_Xz*8+cacheSize-1)/cacheSize;
        #ifdef addsig2vol_debug
            mexPrintf("X-Dim partitioning\n");
        #endif
    }
}

glPackages = zPackages*yPackages*xPackages;

unsigned int *finalPackagesStartPixIndex = malloc(sizeof(unsigned int)*glPackages);
finalPackagesStartPixPointer = &finalPackagesStartPixIndex;

unsigned int *finalPackagesNXzNum = malloc(sizeof(unsigned int)*glPackages);
finalPackagesNXzNumPointer = &finalPackagesNXzNum;

unsigned int *finalPackagesNYzNum = malloc(sizeof(unsigned int)*glPackages);
finalPackagesNYzNumPointer = &finalPackagesNYzNum;

unsigned int *finalPackagesNZzNum = malloc(sizeof(unsigned int)*glPackages);
finalPackagesNZzNumPointer = &finalPackagesNZzNum;

float *finalPackagesPixVectz0 = malloc(sizeof(float)*glPackages);
finalPackagesPixVectzPointer0 = &finalPackagesPixVectz0;

float *finalPackagesPixVectz1 = malloc(sizeof(float)*glPackages);
finalPackagesPixVectzPointer1 = &finalPackagesPixVectz1;

float *finalPackagesPixVectz2 = malloc(sizeof(float)*glPackages);
finalPackagesPixVectzPointer2 = &finalPackagesPixVectz2;

int zLayerCounter = 0;
unsigned int startPixIndex = 0;

// Z-Layer multithreading, use partitioning if small enough Z-layers imaged
unsigned int zRemainder = 0;
unsigned int remainingZLayers = n_Zz % zPackages;
int package = 0;
for(unsigned int zzzzzz = 0; zzzzzz<zPackages; zzzzzz++){
```

```
    int additionalZNum = 0;
    if(zRemainder < remainingZLayers){
        zRemainder ++;
        additionalZNum = 1;
    }
    unsigned int n_Zz_num = (unsigned int)(n_Zz/zPackages) + additionalZNum;
    // 1 if not z-partitioning

    // Y-Layer multithreading, use partitioning if small enough Y-layers imaged
    unsigned int yRemainder = 0;
    int yLayerCounter = 0;
    unsigned int remainingYLayers = n_Yz % yPackages;
    for(unsigned int yyyyyy = 0; yyyyyy < yPackages; yyyyyy++){
        int additionalYNum = 0;
        if(yRemainder < remainingYLayers){
            yRemainder ++;
            additionalYNum = 1;
        }
        unsigned int n_Yz_num = (unsigned int)(n_Yz/yPackages) + additionalYNum;

        //X-Layer multithreading, use partitioning if small enough X-layers imaged
        unsigned int xRemainder = 0;
        int xLayerCounter = 0;
        unsigned int remainingXLayers = n_Xz % xPackages;
        for(unsigned int xxxxxx = 0; xxxxxx < xPackages; xxxxxx++){
            int additionalXNum = 0;

            // Remainder Distribution

            if(xRemainder < remainingXLayers){
                xRemainder ++;
                additionalXNum = 1;
            }
            unsigned int n_Xz_num = (unsigned int)(n_Xz/xPackages) + additionalXNum;


            unsigned int startPixIndex = zLayerCounter*n_Yz*n_Xz + yLayerCounter*n_Xz +
            xLayerCounter;

            float pixVecBufferI0 = (*(pix_vectz+0))+ ((xLayerCounter)* (*resz));
            float pixVecBufferI1 = (*(pix_vectz+1))+ ((yLayerCounter)* (*resz));
            float pixVecBufferI2 = (*(pix_vectz+2))+ ((zLayerCounter)* (*resz));



            finalPackagesStartPixIndex[package] = startPixIndex;
            finalPackagesNXzNum[package] = n_Xz_num;
            finalPackagesNYzNum[package] = n_Yz_num;
            finalPackagesNZzNum[package] = n_Zz_num;
            finalPackagesPixVectz0[package] = pixVecBufferI0;
            finalPackagesPixVectz1[package] = pixVecBufferI1;
            finalPackagesPixVectz2[package] = pixVecBufferI2;
            package++;

            xLayerCounter = xLayerCounter + n_Xz_num;
        }
        yLayerCounter = yLayerCounter + n_Yz_num;
    }
    zLayerCounter = zLayerCounter + n_Zz_num;
}
```

Releasing Threads:

```
//imaging

pthread_mutex_init (&mutex, NULL);

//release threads
packageCounter = 0; // already defined

for (i=0;i<nCores-1+1;i++)
{
    pix_vecz_buffer[i][0]= *pix_vectz;
    pix_vecz_buffer[i][1]= *(pix_vectz+1);
    pix_vecz_buffer[i][2]= *(pix_vectz+2);
```

```
    //fill parameter struct
    threadArg[i].outz=outz;//
    threadArg[i].AScanz=AScanz;
    threadArg[i].n_AScanz=n_AScanz;
    threadArg[i].bufferz=bufferz;
    threadArg[i].pix_vectz=&(pix_vecz_buffer[i][0]);/*mxGetPr(pix_vect)*/
    threadArg[i].n_Xz=n_Xz;
    threadArg[i].rec_posz=rec_posz;
    threadArg[i].send_posz=send_posz;
    threadArg[i].speedz=speedz;
    threadArg[i].resz=resz;
    threadArg[i].timeintz=timeintz;
    threadArg[i].AScan_complexz=AScan_complexz;
    threadArg[i].buffer_complexz=buffer_complexz;
    threadArg[i].out_complexz=out_complexz;//
    threadArg[i].n_Yz=n_Yz;
    threadArg[i].n_Zz=n_Zz;/*n_Zz*/
    threadArg[i].IMAGE_SUMz=IMAGE_SUMz;//
    threadArg[i].IMAGE_SUM_complexz=IMAGE_SUM_complexz;

}

#ifdef C_CODE
  xsum_c(&threadArg[0],&threadArg[0],&threadArg[0],&threadArg[0]);
#else
  xsum_complex(&threadArg[0],&threadArg[0],&threadArg[0],&threadArg[0]);
#endif

 for (i=0;i<nCores-1+1;i++)
{
#ifdef p_threads
    rc = pthread_create( &(mythread[i]), NULL, thread_function, (void*)&(threadArg[i]));
    if (rc) { mexPrintf("ERROR: return code from pthread_create() is %d\n", rc); return;}
#endif
}

//catches threads again
for (i=0;i<nCores-1+1;i++)
{
#ifdef p_threads
    rc = pthread_join ( mythread[i], NULL );
    if (rc) { mexPrintf("ERROR: return code from pthread_join() is %d\n", rc); return;}
#endif
}
```

Thread Function:

```
void *thread_function(void *arg)
{
    unsigned int usedPackage = 0;
    Addsig2vol_param threadArg = *((Addsig2vol_param*)arg);

    double*ownOutz = threadArg.outz;
    double*ownOut_complexz = threadArg.out_complexz;
    double*ownIMAGE_SUMz = threadArg.IMAGE_SUMz;
    double*ownIMAGE_SUM_complexz = threadArg.IMAGE_SUM_complexz;

    float ownPix_vectz[3] = {0,0,0};
    ownPix_vectz[0] = threadArg.pix_vectz[0];
    ownPix_vectz[1] = threadArg.pix_vectz[1];
    ownPix_vectz[2] = threadArg.pix_vectz[2];
    threadArg.pix_vectz = &ownPix_vectz[0];

    int packages = 0;


    pthread_mutex_lock (&mutex);
    usedPackage = packageCounter++;
    pthread_mutex_unlock (&mutex);

    while(usedPackage < glPackages){
        unsigned int startPixIndex = (*finalPackagesStartPixPointer)[usedPackage];
        threadArg.outz = ownOutz + startPixIndex;
        threadArg.out_complexz = ownOut_complexz + startPixIndex;
```

```
        threadArg.n_Xz = (*finalPackagesNXzNumPointer)[usedPackage];
        threadArg.n_Yz = (*finalPackagesNYzNumPointer)[usedPackage];
        threadArg.n_Zz = (*finalPackagesNZzNumPointer)[usedPackage];
        threadArg.IMAGE_SUMz = ownIMAGE_SUMz + startPixIndex;
        threadArg.IMAGE_SUM_complexz = ownIMAGE_SUM_complexz  + startPixIndex;
        ownPix_vectz[0] = (*finalPackagesPixVectzPointer0)[usedPackage];
        ownPix_vectz[1] = (*finalPackagesPixVectzPointer1)[usedPackage];
        ownPix_vectz[2] = (*finalPackagesPixVectzPointer2)[usedPackage];



        #ifdef C_CODE
          as2v_c(&threadArg,&threadArg,&threadArg,&threadArg); //compatible win64 & linxu64
        function-call
        #else
        if (addsig2vol_mode==0) as2v_complex(&threadArg,&threadArg,&threadArg,&threadArg);
        if (addsig2vol_mode==2) as2v_complex_sm(&threadArg,&threadArg,&threadArg,&threadArg);

        #endif

        pthread_mutex_lock (&mutex);
        usedPackage = packageCounter++;
        pthread_mutex_unlock (&mutex);
    }
    return NULL;
}
```

Test Case:

```
clear all;

%params of benchmark
numberOfScans = 5000; %Amount of data used for every test
coreNumber = 4; %Number of cores used for testing

lowestDimension = 1;
highestDimension = 96;
lowestCacheSize = 128;
highestCacheSize = 32*128;
stepSizeC = 32; %number of cache sizes per step (has to be whole integer)
stepSizeD = 1; %number of dimensions per step (has to be whole integer)
testsPerDimension = 50;

%Different Methods for Calculating the XYZ size of a dimension
%case 0: 2^dimension
%case 1: 2^dimension -1
%case 2: 4
%case 3: 8*dimension
%case 4: dimension^2
%case 5: 7
%case 6: 3
%case 7: 16
%case 8: 8
%case 9: 10
%case 10: 14
%case 11: 4*dimension
%case 12: dimension
xCalcType = 11;
yCalcType = 11;
zCalcType = 11;

%Different Methods for Calculating the Cache size
%0 : 2^cacheSize
%1 : cacheSize*1024*1024
%2 : cacheSize*1024
cacheType = 2;

%used variables
numberOfTests = 0; %variable calculated from testsPerDimension
ascan = []; %random, non-changing inputs for testing
xSize = 0; %dynamic calculation of x-dimension
ySize = 0; %dynamic calculation of y-dimension
zSize = 0; %dynamic calculation of z-dimension
```

```
%results of benchmark
times=[]; %times it took the algorithm to process an image
meanTimes=[]; %mean of testtimes
medianTimes=[]; %median of testtimes
stdTimes=[]; %std of testtimes
performance = []; %performance of testtimes
Voxel = []; %Voxels per dimension
results = []; %time and comparison result of both images
testIfEqual = false; %interferes with performance but compares if results are equal

%to avoid bugs:
pathFolder = '';

%Begin of benchmark

addsig2vol_3(coreNumber);
addsig2vol_3_alt(coreNumber);


testedDimensions = ceil((highestDimension-lowestDimension) / stepSizeD) + 1;
testedSizes = ceil((highestCacheSize-lowestCacheSize) / stepSizeC) + 1;
ascan=rand([1 numberOfScans])';

dimAxisTicksLabel = strings;
cacheAxisTicksLabel = strings;
times=zeros(testedDimensions, testedSizes, testsPerDimension, 2);
meanTimes=zeros(testedDimensions, testedSizes, 2);
medianTimes=zeros(testedDimensions, testedSizes, 2);
performance = zeros(testedDimensions, testedSizes, 4);
logPerformance = zeros(testedDimensions, testedSizes, 4);
Voxel = zeros(testedDimensions, 1);

for dimension = 1:testedDimensions

    dimension

    xSize = imageSize((dimension - 1)*stepSizeD + lowestDimension, xCalcType);
    ySize = imageSize((dimension - 1)*stepSizeD + lowestDimension, yCalcType);
    zSize = imageSize((dimension - 1)*stepSizeD + lowestDimension, zCalcType);

    Voxel(dimension) = xSize*ySize*zSize;

    for cache = 1:testedSizes

        cache

        actualCacheSize = cacheSizeFunc((cache - 1)*stepSizeC + lowestCacheSize, cacheType);
        for test = 1:testsPerDimension
            if testIfEqual == false
                times(dimension, cache, test, mod(test,2)+1) = addsigCommand(mod(test,2),
                    ascan, xSize, ySize, zSize, actualCacheSize, pathFolder);
                times(dimension, cache, test, mod(test+1,2)+1) = addsigCommand(mod(test+1,2),
                    ascan, xSize, ySize, zSize, actualCacheSize, pathFolder);
            else
                results = addsigCommandCompare(mod(test+1,2), ascan, xSize, ySize, zSize,
                    actualCacheSize, pathFolder);
                times(dimension, cache, test,1) = results(1);
                times(dimension, cache, test,2) = results(2);
                if results(3) == false % if not equal
                    fprintf(['failure beep boop we need to see this\n' num2str(test)]);
                    fprintf(['\n' num2str(dimension)]);
                    fprintf(['\n']);
                end
            end
        end

        cd logdateien
        save(['timesAt__cOpt4_C' num2str(coreNumber) '_V' num2str(testsPerDimension) '_Ld'
            num2str(lowestDimension) '_Hd' num2str(highestDimension) '_cType'
            num2str(xCalcType) '' num2str(yCalcType) '' num2str(zCalcType) '_S'
            num2str(stepSizeD) '_cSl' num2str(lowestCacheSize) '_cSh' num2str(lowestCacheSize)
            '_cSs' num2str(stepSizeC) '_cSc' num2str(cacheType) '_dim' num2str(dimension)]);
        cd ..

        meanTimes(dimension, cache, 1) = mean(times(dimension, cache,:,1));
        medianTimes(dimension, cache, 1) = median(times(dimension, cache,:,1));
        performance(dimension, cache, 1) = xSize*ySize*zSize / meanTimes(dimension, cache,1);
```

```
        performance(dimension, cache, 3) = xSize*ySize*zSize / medianTimes(dimension,
            cache,1);
        meanTimes(dimension, cache, 2) = mean(times(dimension, cache,:,2));
        medianTimes(dimension, cache, 2) = median(times(dimension, cache,:,2));
        performance(dimension, cache, 2) = xSize*ySize*zSize / meanTimes(dimension, cache,2);
        performance(dimension, cache, 4) = xSize*ySize*zSize / medianTimes(dimension,
            cache,2);
        logPerformance(dimension, cache, 1) = log10(performance(dimension, cache, 1));
        logPerformance(dimension, cache, 2) = log10(performance(dimension, cache, 2));
        logPerformance(dimension, cache, 3) = log10(performance(dimension, cache, 3));
        logPerformance(dimension, cache, 4) = log10(performance(dimension, cache, 4));
        processedPerformanceMean(dimension, cache) = performance(dimension, cache, 1) -
            performance(dimension, cache, 2);
        processedPerformanceMedian(dimension, cache) = performance(dimension, cache, 3) -
            performance(dimension, cache, 4);
        processedPerformanceMeanLg(dimension, cache) = logPerformance(dimension, cache, 1) -
            logPerformance(dimension, cache, 2);
        processedPerformanceMedianLg(dimension, cache) = logPerformance(dimension, cache, 3) -
            logPerformance(dimension, cache, 4);
        processedPerformanceMeanPercent(dimension, cache) = performance(dimension, cache, 1) /
            performance(dimension, cache, 2);
        processedPerformanceMedianPercent(dimension, cache) = performance(dimension, cache, 3)
            / performance(dimension, cache, 4);

        dimAxisTicks(dimension) = double(dimension);
        cacheAxisTicks(cache) = double(cache);
        dimAxisTicksLabel(dimension, 1) = num2str(xSize*ySize*zSize);
        cacheAxisTicksLabel(cache, 1) = num2str(actualCacheSize);
    end
end

figure;
performanceN = squeeze(performance(:,1,:));
plot(Voxel(:,1)', performanceN(:,:));

f1 = figure;
imagesc(processedPerformanceMean);
title('Mean Performance Difference');
ax = gca;
ax.XTick = cacheAxisTicks;
ax.YTick = dimAxisTicks;
ax.YTickLabel = dimAxisTicksLabel;
ax.XTickLabel = cacheAxisTicksLabel;
xtickangle(90);
xlabel('Cachesize [Byte]');
ylabel('Imagesize [Voxel]');

f2 = figure;
imagesc(processedPerformanceMedian);
title('Median Performance Difference');
ax = gca;
ax.XTick = cacheAxisTicks;
ax.YTick = dimAxisTicks;
ax.YTickLabel = dimAxisTicksLabel;
ax.XTickLabel = cacheAxisTicksLabel;
xtickangle(90);
xlabel('Cachesize [Byte]');
ylabel('Imagesize [Voxel]');

f3 = figure;
imagesc(processedPerformanceMeanLg);
title('log Mean Performance Difference');
ax = gca;
ax.XTick = cacheAxisTicks;
ax.YTick = dimAxisTicks;
ax.YTickLabel = dimAxisTicksLabel;
ax.XTickLabel = cacheAxisTicksLabel;
xtickangle(90);
xlabel('Cachesize [Byte]');
ylabel('Imagesize [Voxel]');

f4 = figure;
imagesc(processedPerformanceMedianLg);
title('log Median Performance Difference');
ax = gca;
ax.XTick = cacheAxisTicks;
ax.YTick = dimAxisTicks;
```

```
ax.YTickLabel = dimAxisTicksLabel;
ax.XTickLabel = cacheAxisTicksLabel;
xtickangle(90);
xlabel('Cachesize [Byte]');
ylabel('Imagesize [Voxel]');
```

cacheSizeFunc:

```
function ret = cacheSizeFunc(n, type)
switch type
    case 0
        ret = 2^n;
    case 1
        ret = 1024*1024*n;
    case 2
        ret = 1024*n;
end
```

imageSize:

```
function ret = imageSize(n, type)
switch type
    case 0
        ret = 2^n;
    case 1
        if n <= 2
            ret = 4;
        else
            ret = 2^n -1;
        end
    case 2
        ret = 4;
    case 3
        ret = 8*n;
    case 4
        ret = n^2;
    case 5
        ret = 7;
    case 6
        ret = 3;
    case 7
        ret = 16;
    case 8
        ret = 8;
    case 9
        ret = 10;
    case 10
        ret = 14;
    case 11
        ret = 4*n;
    case 12
        ret = n;
    case 13
        ret = 256;
    case 14
        ret = 1;
end
```

addsigCommand:

```
function ret = addsigCommand(whichTest, ascan, xSize, ySize, zSize, cache, AlteMex)
image = zeros(xSize, ySize, zSize);
rand('seed',0);
if whichTest == 0
    tic();
    addsig2vol_3(ascan,single([1 2 3]),10.*ones([3 1],'single'),400.*ones([3
        1],'single'),ones([1 1],'single'),ones([1 1],'single'),ones([1
        1],'single'),uint32([xSize ySize zSize]),image, cache);
    ret = toc();
else
    tic();
    addsig2vol_3_alt(ascan,single([1 2 3]),10.*ones([3 1],'single'),400.*ones([3
```

```
        1],'single'),ones([1 1],'single'),ones([1 1],'single'),ones([1
        1],'single'),uint32([xSize ySize zSize]),image);
    ret = toc();
end
end
```

addsigCommandCompare:

```
function ret = addsigCommandCompare(whichTest, ascan, xSize, ySize, zSize, cache, AlteMex)
image = zeros(xSize, ySize, zSize);
times = zeros(2, 1);
equal = true;
rand('seed',0);

if whichTest == 0
    tic();
    image1=addsig2vol_3(ascan,single([1 2 3]),10.*ones([3 1],'single'),400.*ones([3
        1],'single'),ones([1 1],'single'),ones([1 1],'single'),ones([1
        1],'single'),uint32([xSize ySize zSize]),image, cache);
    times(1) = toc();
else
    %cd(AlteMex)
    tic();
    image2=addsig2vol_3_alt(ascan,single([1 2 3]),10.*ones([3 1],'single'),400.*ones([3
        1],'single'),ones([1 1],'single'),ones([1 1],'single'),ones([1
        1],'single'),uint32([xSize ySize zSize]),image);
    times(2) = toc();
end

rand('seed',0);

if whichTest == 1
    tic();
    image1=addsig2vol_3(ascan,single([1 2 3]),10.*ones([3 1],'single'),400.*ones([3
        1],'single'),ones([1 1],'single'),ones([1 1],'single'),ones([1
        1],'single'),uint32([xSize ySize zSize]),image);
    times(1) = toc();
else
    tic();
    image2=addsig2vol_3_alt(ascan,single([1 2 3]),10.*ones([3 1],'single'),400.*ones([3
        1],'single'),ones([1 1],'single'),ones([1 1],'single'),ones([1
        1],'single'),uint32([xSize ySize zSize]),image);
    times(2) = toc();
end
ret = [times(1), times(2), equal];
end
```

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 selbstständig verfasst haben und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden, sowie Zitate kenntlich gemacht wurden.

Peter Zwick
Graben-Neudorf

Bastian Eckhardt
Stutensee