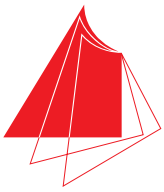


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Embedded Software Labor
Implementierung eines CDMA-Decoders

von
Tim Hänlein

bei Prof. Dr. Dirk W. Hoffmann

Hochschule Karlsruhe
2021

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Dieser Bericht dokumentiert die Implementierung eines CDMA-Decoders in den Sprachen C und C++ sowie die Laufzeitoptimierung in C. Dies wurde im Wintersemester 2020/21 an der Hochschule Karlsruhe im Rahmen des vorlesungsbegleitenden Labors „Embedded-Software“ umgesetzt.

Code Division Multiple Access (CDMA) ist ein Codemultiplexverfahren, welches die zeitgleiche Datenübermittlung mehrerer Übertragungen auf derselben Frequenz ermöglicht. Dabei wird zwischen synchronem und asynchronem CDMA unterschieden, wobei sich diese Umsetzung auf synchrones CDMA beschränkt, das bei der Codierung von Chipsequenzen des GPS zum Einsatz kommt.

Um eine portable und leichtgewichtige Lösung zu garantieren, kamen keine Bibliotheken außer der Standard Template Library (STL) zum Einsatz. Die Implementierung setzt auf ein CMake-Projekt auf. Entwickelt wurde unter Windows mit MSVC, getestet unter Linux mit gcc. Das Programm soll eine in einer Datei gespeicherten Chipsequenz einlesen, decodieren und anschließend die Ergebnisse auf der Konsole mit den Informationen, welcher Satellit mit welchem Versatz welches Bit gesendet hat, ausgeben.

2 Implementierung

2.1 C++

Die Implementierung des Decoders in C++ geschieht sprachbedingt objektorientiert. Für die Verwaltung von Feldern und ähnlichen Datenstrukturen wurde die STL-Library verwendet. Somit konnte komplett auf die manuelle Allokation von Heap-Speicher verzichtet und somit Speicherlecks präveniert werden. Die Zeitmessung geschieht mittels `std::chrono`, um eine portable Lösung zu gewährleisten. Bei dem verwendeten C++-Standard handelt es sich um C++11, welcher aufgrund der Verwendung von `std::move`, der Chrono-Bibliothek und Listeninitialisierung benötigt wird. Der C++-Code gliedert sich essenziell in 6 Segmente:

1. Einlesen der Chipsequenz
2. Starten der Zeitmessung
3. Erstellen der Sequenzgeneratoren
4. Instanziieren des Decoders und Ausführung der Decodierung
5. Stoppen der Zeitmessung
6. Ausgabe der berechneten Ergebnisse und benötigen Zeit

Mittels `std::ifstream` und `std::istream_iterator` wird eine als Programmargument angegebene Textdatei eingelesen und zeichenweise in einem `std::vector` als Integer abgespeichert. Nach dem „starten“ des Timers werden die Sequenzgeneratoren mit den jeweiligen Indizes der Muttersequenzen erstellt und in einem `std::vector` abgelegt. Die Goldsequenzen werden an dieser Stelle noch nicht generiert. Zum Instanziieren des Decoders ist die eingelesene Chipsequenz zu übergeben. Da es sich hier um eine größere Datenmenge (1KB) handelt und sie nach der Decodierung in der Main-Funktion nicht mehr benötigt wird, verschiebt der Konstruktor die Sequenz mittels

`std::move` in ein Attribut des Objekts. Anschließend führt der Decoder die Decodierung des Signals aus und gibt die Ergebnisse zurück, die anschließend zusammen mit der gemessenen Zeit an den „Standard Output Stream“ übergeben werden.

SequenceGenerator

Der SequenceGenerator führt die Generierung der Goldsequenzen aus, die benötigt werden, um die Kreuzkorrelation durchzuführen. Dazu sind die öffentliche Methode `generate()` und die private Methode `shiftMotherSequence()` bereitgestellt. `shiftMotherSequence()` modelliert die Verschiebung der Muttersequenzen und das Einfügen des neuen Elements am Anfang der Liste. `generate()` generiert die Goldsequenz mit den angegebenen Indizes. Dafür wird für die Länge der resultierenden Sequenz elementweise aus den Muttersequenzen gebildet und in einem `std::vector` fixer Länge gespeichert. Die zwei Muttersequenzen von jedem Generator sind durch die STL-Struktur `std::deque` modelliert, da diese konstante Laufzeit für das Einfügen von Elementen am Anfang garantieren. Nach der Erzeugung der Goldfolge, wird der `std::vector` zurückgegeben.

Decoder

Ähnlich wie die Klasse SequenceGenerator hat die Klasse Decoder eine öffentliche Methode `decode()` und eine private Methode `correlate()`. `decode()` nimmt dabei als Parameter den zuvor erzeugten `std::vector` mit den Sequenzgeneratoren. Für jeden Generator wird die zugehörige Sequenz mittels `SequenceGenerator::generate()` erzeugt und in die Methode `correlate()` gegeben. Zusätzlich werden der Peak, die aktuelle Satelliten-ID und einen Zeiger auf den Outparameter in die private Methode übergeben. Am Ende wird ein `std::vector` mit den gefundenen Korrelationen zurückgegeben. `correlate()` kreuzkorreliert die Goldfolge und akkumuliert die jeweiligen Einträge in der Chipsequenz auf. Ist der absolute Wert der Akkumulation höher als der gegebene Peak, wurde eine Übereinstimmung gefunden. Die Satelliten-ID, das Offset sowie das gesendete Bit bilden dann eine Struct, die an den Outparameter angehängt wird. An dieser Stelle kann die Korrelation abgebrochen werden.

2.2 C

Basierend auf der C++-Implementierung wurde die C-Version geschrieben. Dabei ist zu beachten, dass C im Gegensatz zu C++ keine Namensräume und Objektorientierung kennt. Das bedeutet, dass die Objektattribute aus SequenceGenerator und Decoder entweder als Parameter übergeben oder wenn möglich als Modulvariable deklariert werden müssen. Somit wurde der Namespace „CDMA“ entfernt und den öffentlichen Methoden als Namenspräfix angehängt, wobei die privaten Methoden nun als statisch bezeichnet sind und mit einem Unterstrich beginnen. Durch die genannten Gegebenheiten müssen sämtliche Vektoren in Arrays umgeschrieben werden. Da nun keine SequenceGenerator-Objekte mehr erstellt werden können, sind die Goldfolgen vorher zu generieren und anschließend an die decode-Methode zu übergeben. Dafür wird ein Stack-Array mit 24 Einträgen erstellt, das für jeden Satelliten einen Pointer auf seine Goldfolge speichert. Die Decodierung und Ausgabe der Ergebnisse erfolgt dann analog zur C++-Implementierung. Zusätzlich muss noch der mit `malloc()` allokierte Spei-

cher, also die Goldsequenzen und die Korrelationsergebnisse mit `free()` freigegeben werden, um Speicherlecks vorzubeugen.

SequenceGenerator

Der Sequenzgenerator ist nun keine Klasse mehr, sondern ein Modul aus einer „öffentlichen" Funktion `CDMA_GenerateSequence()` und einer statischen Funktion `_shiftMotherSequence`. Das Modul ist damit ähnlich aufgebaut wie die C++-Klasse. Allerdings werden keine Generatorobjekte erzeugt, sondern die Goldsequenz mit entsprechendem Index von erster Funktion generiert und in einen Outparameter geschrieben. Die Muttersequenzen sind hierbei durch Stackarrays statt `std::deque` modelliert, was eine Verschiebung aller Elemente darin erfordert, um eines am Anfang anzufügen.

Decoder

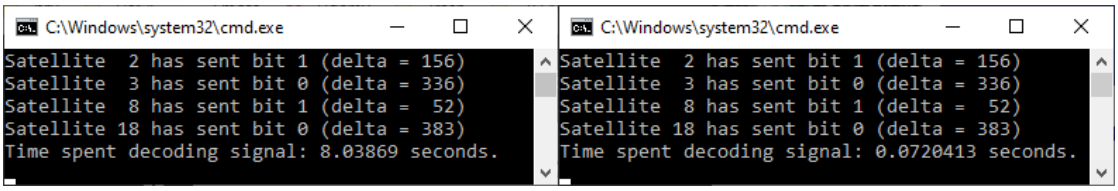
Im Modul „Decoder" finden sich wie auch in der C++-Implementierung zwei Funktionen wieder: `CDMA_decode()` und `_correlate()`. Erstere wird in `main()` aufgerufen und startet die Korrelation für jede Goldfolge. Zweitere führt die Kreuzkorrelation anhand der übergebenen Sequenz aus und akkumuliert die entsprechenden Stellen der Chipsequenz. Wie in C++ wird ein Outparameter verwendet, um die Korrelationsergebnisse zu speichern. Allerdings muss nun zusätzlich ein boolscher Wert zurückgegeben werden, um Information darüber zu führen, wie viele Ergebnisse bereits gefunden wurden. Für die Ergebnisse wird so viel Speicher reserviert, wie es sendende Satelliten gibt.

3 Laufzeitanalyse

Um die Laufzeitanalyse durchzuführen, wurde unter C++ die Bibliothek `std::chrono` verwendet. Unter C kam die etwas ungenauere Funktion `clock()` zum Einsatz. Bei dem Testsystem handelt es sich um eine x64 AMD-Plattform der Zen 2 Generation mit 4,00 GHz und 3600 MHz DDR4-Speicher. Die Ergebnisse sind somit nicht mit anderen Systemen zu vergleichen, da diese stark plattformabhängig sind. Die Zeitmessungen wurden nach dem Einlesen der Chipsequenz gestartet und unmittelbar nach der Decodierung beendet. Somit beschreibt die Zeitspanne ausschließlich die Decodierung selbst und die Schritte, welche dafür nötig sind, wie zum Beispiel die Generierung der Goldsequenzen.

Nachfolgend ist die Betrachtung für C++, die unoptimierte, sowie die optimierte C-Implementierung aufgeführt mit jeweils dem Debug- und Release-Build. Im Debug-Build ist die Compilerseite Optimierung ausgeschaltet und Debuginformationen sind zugelassen. Der Release-Build speichert keinerlei Debuginformationen, verwendet die höchste verfügbare Optimierungsstufe mit Priorisierung von Geschwindigkeit (O3 bei GCC, O2 bei MSVC) und lässt Funktionsinlining zu.

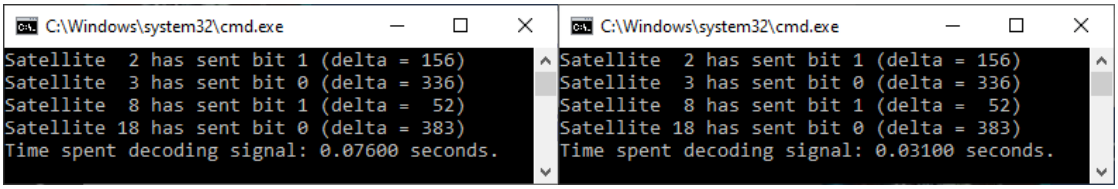
3.1 C++



(a) Debug-Build. (b) Release-Build.

Abbildung 1: Die Konsolenausgaben der jeweiligen Builds der C++-Implementierung. Links: der Debug-Build. Rechts: der Release-Build. Die Geschwindigkeitszunahme bei Anschalten der Compileroptimierungen ist enorm. Die Laufzeit von etwa 8 Sekunden im Debug sanken auf etwa 72 Millisekunden im Release. Das ergibt eine Geschwindigkeitssteigerung um den Faktor 111.

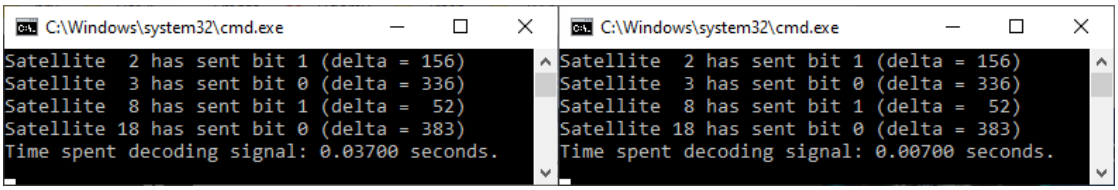
3.2 C unoptimiert



(a) Debug-Build. (b) Release-Build.

Abbildung 2: Die Konsolenausgaben jeweiliger Builds der unoptimierten C-Implementierung. Links: der Debug-Build. Rechts: der Release-Build. Der Geschwindigkeitszuwachs im Release ist mit Halbierung der Laufzeit vergleichsweise zur C++-Implementierung nicht ansatzweise so groß.

3.3 C optimiert



(a) Debug-Build. (b) Release-Build.

Abbildung 3: Die jeweiligen Build-Ausgaben der optimierten C-Implementierung. Links: der Debug-Build. Rechts: der Release-Build. Der Geschwindigkeitszuwachs von Debug zu Release ist auch hier nicht so groß wie in C++, aber dennoch größer als in der unoptimierten C Implementierung. Der Release ist etwa 5 Mal schneller als der Debug-Build.

Um die Laufzeit des C-Programms zu verringern, sind zunächst triviale Optimierungen der Sprache vorgenommen worden. Darunter fallen:

1. Konstante Werte wurden durch Präprozessordefinitionen ersetzt.

2. Die Funktion `_correlate()` in dem Modul `Decoder` wurde in die Funktion `CDMA_decode()` eingefügt.

3. Zählervariabeln erhielten das Schlüsselwort `register`.

4. Es wurden Prä- statt Postinkrements verwendet.

5. Bei Iteration über Pointer wurde auf Indexzugriff verzichtet und stattdessen ein Zeiger inkrementiert.

Das alles sind Optimierungen, die der Compiler ohnehin vorgenommen hätte. Sie verringern allerdings bereits die Zeit im Debug-Build um einige Millisekunden. Weitere programmspezifische Verbesserungen erzielten die größten Zeiteinsparungen und verringerten die Ausführungszeit nochmals um wenige 10 Millisekunden:

1. Ein „Early-Exit“ bricht die Korrelation ab, wenn bereits alle gesendeten Bits gefunden wurden. Diese Abfrage erzielt vor allem bei Bitsendungen von niedrig indexierten Satelliten große Verbesserung. Im schlimmsten Fall, das heißt der 24. Satellit sendet, läuft das Programm minimal langsamer.

2. Da die Goldsequenzen immer gleich sind, müssen sie nicht vor jeder Decodierung neu berechnet werden. Es reicht, diese einmal zu generieren und dann wiederzuverwenden. Ein zweidimensionales Stack-Array speichert die Goldfolgen als lokale Variable in der decode-Funktion, auf welche dann nur noch lesend zugegriffen wird.

3. Der Modulo-Operator, welcher zur Begrenzung des Index zum Zugriff auf die Goldsequenz dient, befindet sich in der inneren Schleife, wird oft ausgeführt und ist damit sehr teuer. Um darauf zu verzichten, wurde zunächst eine Methode der Bitmanipulation eingesetzt [1], welche aber durch die Duplizierung der Goldsequenzen ersetzt wurde. Somit konnte auf Kosten der Speicherlast auf den Modulo-Operator komplett verzichtet und die Laufzeit um einiges verbessert werden.

4. Die letzte Optimierung wurde ebenfalls in der inneren Schleife angewandt, um auf den ternären Operator sowie auf den unären Negationsoperator bei der Akkumulation zu verzichten. Dabei sind die Goldfolgen nun nicht länger als eine Menge aus 1 und 0 repräsentiert. Anstelle der 0 wird -1 gespeichert, welche mit dem entsprechenden Eintrag in der Chipsequenz verrechnet werden kann, um das gewünschte Ergebnis zu erzielen.

4 Werkzeuge

Entwickelt wurde unter Windows 10 mit Visual Studio und MSVC, wobei das Projekt auf eine CMake-Umgebung aufsetzt. Anschließend wurde noch unter Fedora mit GCC getestet. Dabei umfasst eine mitgelieferte Makefile die Build-Konfiguration des Release-Builds für GCC. Alternativ lässt sich mit CMake eine Konfiguration für die eigene Plattform selbst generieren. Zur Leistungsanalyse wurde die CPU-Profilerstellung des Visual-Studio-Diagnosetools aktiviert. So konnten die CPU-Zeit intensivsten Programmpfade ausfindig gemacht werden, um gezielt diesen Code zu verbessern.

[1] https://graphics.stanford.edu/ seander/bithacks.html#ModulusDivisionParallel