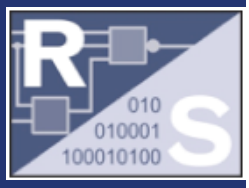
Rechnersysteme 2
Praktikum 2
AMIDAR

Seminararbeit vorgelegt von Marcel Mann und Marcel Humm
Abgabe: 26. Februar 2018
Institut für Datentechnik | Fachgebiet Rechnersysteme | Prof. Dr.-Ing. Christian Hochberger

TECHNISCHE
UNIVERSITÄT
DARMSTADT



1 Aufgabenstellung

Im Rahmen dieses Praktikums galt es einen gegebenen Algorithmus zur Akquisition von GPS Signalen für die Ausführung in einem auf dem AMIDAR-Modell basierenden Java-Prozessor zu implementieren und hinsichtlich zweier unabhängiger Aspekte zu optimieren: Energieverbrauch und benötigte Zeitschritte. Dabei sollte neben konkreten Implementierungsdetails auch die verwendete CGRA-Konfiguration untersucht und angepasst werden, um zwei eigenständige Umsetzungen zu erarbeiten. Den Teilnehmern des Praktikums wurde dazu ein AMIDAR-Simulator zur Verfügung gestellt, der die Auswertung einer ausgeführten Implementierung hinsichtlich der genannten Aspekte ermöglicht.

2 Grundlagen

2.1 Akquisitions-Algorithmus

Das zu implementierende Verfahren wird in GPS-Empfängern zur Erfassung verfügbarer Satelliten verwendet. Gegeben ist dabei eine statisch definierte Menge diskreter Frequenzwerte sowie ein Vektor von (L1 C/A) Satellitencodes.

Das eingehende Signal liegt als Vektor komplexer Werte vor. Im ersten Schritt wird eine e-Funktion verwendet, um daraus eine Menge von Vektoren für jeden Frequenzwert zu bilden. Deren diskrete Fourier-Transformationen (DFT) werden im zweiten Schritt jeweils mit der DFT der Satellitencodes multipliziert und schließlich deren jeweilige inverse DFT gebildet.

In der daraus resultierenden Ergebnismatrix wird im letzten Schritt der normalisierte Maximalwert ermittelt. Überschreitet dieser Wert einen bestimmten Grenzwert, kann das empfangene Eingangssignal als authentisches Satellitensignal bestätigt werden.

2.2 Coarse Grained Reconfigurable Array (CGRA)

Bei einem CGRA handelt es sich um eine Menge vernetzter Verarbeitungseinheiten, auch Processing Elements (PE) genannt. Die Konfiguration aller PEs als auch deren Verbindungen untereinander lassen sich mit nur wenigen Bit als sogenannte Kontexte speichern. Dadurch ist es möglich den Kontext während der Laufzeit mehrfach zu ändern und so eine Vielzahl unterschiedlicher Funktionen abzubilden. Da ein entsprechendes CGRA sehr viel Rechenleistung bereitstellen kann, werden darauf üblicherweise kontrollarme aber rechenintensive Teile von Anwendungen als sogenannte Kernels ausgeführt.

2.3 Adaptive Micro-Instruction Driven Architecture (AMIDAR)

Das AMIDAR-Modell beschreibt eine Prozessorarchitektur mit einer speziellen Art der Befehlsabarbeitung. Auszuführende Instruktionen werden in einzelne Operationen zerlegt und als sogenannte Tokens an eine Reihe verschiedener Funktionseinheiten verteilt. Ein Token enthält dabei neben der auszuführenden Operation auch Informationen über etwaige Abhängigkeiten zu den Ergebnissen anderer Tokens oder an welche Funktionseinheiten die berechneten Ergebnisse weiterzuleiten sind.

Im Rahmen dieses Praktikums wird ein Simulator eines auf dem AMIDAR-Modell basierenden Java-Prozessors verwendet. Zur Laufzeit wird hier mit Profiling-Mechanismen analysiert, inwiefern sich häufig wiederholende Programmteile in Hardware synthetisieren lassen, um deren Ausführung zu beschleunigen. Die daraus entstehenden Funktionseinheiten werden schließlich auf einer vorher definierten CGRA-Struktur abgebildet. Die Struktur des auszuführenden Programmcodes beeinflusst die Effizienz dieses Verfahrens maßgeblich. Daher werden der erreichte Synthesegrad, die in der Simulation benötigte Zeit sowie die verbrauchte Energie nach Abschluss der Ausführung ermittelt und dem Anwender zur weiteren Optimierung des Programmcodes zur Verfügung gestellt.

3 Durchführung

3.1 Einarbeitung und erste Implementierung

Vor der eigentlichen Umsetzung des Akquisitions-Algorithmus galt es zunächst den zur Verfügung gestellten AMIDAR-Simulator auf den von uns verwendeten Entwicklungssystemen zu installieren und sich in dessen Verwendung einzuarbeiten. An dieser Stelle sei zu erwähnen, dass sich der Simulator zum Zeitpunkt dieses Praktikums noch in der Entwicklung befand und dessen Verwendung kaum dokumentiert wurde. Erste Implementierungsarbeiten wurden dadurch bis in die zweite Praktikumswoche verzögert, als schließlich eine Anleitung der durchzuführenden Schritte bereitgestellt und einige Anpassungen am Simulator durchgeführt wurden.

Der darauffolgende Schritt bestand aus dem Verständnis des Algorithmus und dessen Zerlegung in implementierbare Teilschritte. Den Grundprinzipien objektorientierter Programmierung folgend, suchten wir zunächst nach hilfreichen Bausteinen verwendbarer Java-Bibliotheken und bildeten einige Klassen, beispielsweise zur Repräsentation und Verarbeitung komplexer Zahlen. Das Priorisieren einer korrekten Verarbeitung der Eingabedaten und des Abgleichs der Ausgabedaten mit den Sollwerten führte zunächst zu einer sehr schlechten Synthetisierbarkeit unserer Umsetzung.

3.2 Verbesserung der Synthetisierbarkeit

Die Akquisition von GPS-Signalen umfasst die Verarbeitung großer Datensätze in Schleifen. Die Auslagerung solcher rechenintensiven Programmteile auf CGRAs kann zur Reduktion der zur Ausführung benötigten Zeit- und Energie beitragen. Unser nächster Schritt bestand dementsprechend aus der Restrukturierung unseres Programmcodes mit dem Ziel, eine möglichst hohe Synthetisierbarkeit zu erreichen.

Aus der zweiten Kick-off-Veranstaltung des Praktikums wurde bekannt, dass die Erstellung neuer Objekte und Arrays nicht synthetisierbar sei. Aus einem Gespräch mit den Betreuern des Praktikums ging weiterhin hervor, dass die Verwendung primitiver Datenstrukturen gegenüber eigener Klassen effizienter ist.

In ersten Optimierungen entfernten wir dementsprechend unsere eigenen Klassen wieder und beschränkten uns auf die Nutzung mehrdimensionaler Float-Arrays zur Darstellung komplexer Zahlen. Weiterhin bündelten wir die Erstellung aller notwendigen Datenstrukturen in einer anfänglichen Initialisierungsphase. Durch diese Anpassungen ließ sich nun bereits ein Großteil unserer Implementierung synthetisieren.

Bei der Ausführung mit entsprechenden Parametern protokolliert der AMIDAR-Simulator ebenfalls die Synthetisierbarkeit des ausgeführten Programms. Besagtes Protokoll bezieht sich jedoch nicht auf den entwickelten high-level Code sondern den daraus generierten Java-Bytecode. Aufgrund fehlender Dokumentation und mangelnder Erfahrung im Umgang mit Bytecode scheiterten wir leider an der ausführlichen Interpretation der gelieferten Protokolle. Die Simulation mit *01_inputcodes.dat* als Eingabewert resultierte zu diesem Zeitpunkt in ca. 280 Millionen benötigten Zeitschritten und einem Verbrauch von 1,7 Milliarden Energieeinheiten.

3.3 Verbesserung der Performanz

Nachdem nun ein Großteil unserer Implementierung synthetisierbar war, konzentrierten wir uns in den nächsten Schritten auf die Reduktion der notwendigen Zeiteinheiten.

In den Konfigurationsdateien des AMIDAR-Simulators finden sich unter anderem auch Informationen über die benötigten Ressourcen einzelner Operationen. Unser erster Ansatz bestand dementsprechend im Identifizieren und Reduzieren teurer und/oder unnötiger Berechnungen und Speicherzugriffe.

Durch geschicktes Umformen, Zusammenführen oder auch Aufbrechen von Berechnungen auf ineinander verschachtelte Schleifen war es beispielsweise möglich teure Divisionen durch günstigere Multiplikationen zu ersetzen oder Vielzahl von Operationen gänzlich einzusparen.

Listing 3.1: Ohne Umformungen

```
for (int k = 0; k < nSamp; k++) {
 for (int t = 0; t < nSamp; t++)
  fac = -2 * t * k * PI;
}
```

Listing 3.2: Optimierte Berechnungen

```
fac1 = -6.2831855f / nSamp;
for (int k = 0; k < nSamp; k++) {
 fac2 = fac1 * k;
 for (int t = 0; t < nSamps; t++)
  fac3 = t * fac2;
}
```

Auf Kosten der Lesbarkeit, Nachvollziehbarkeit und entgegen dem sonst in der Programmierung dogmatisch praktizierten DRY-Prinzip („*Don't repeat yourself*“) favorisierten wir zudem die Verwendung statischer Zahlenwerte gegenüber Variablen, verzichteten auf wiederverwendbare Methoden und duplizierten Codefragmente. Das ermöglichte es auf globalen Datenstrukturen zu arbeiten und so die Anzahl notwendiger Speicherzugriffe weiter zu reduzieren.

Die genannten Maßnahmen in Kombination mit einem auf 16 erhöhten Unroll-Faktor verringerten dabei die notwendigen Zeiteinheiten um fast 27% auf ca. 205 Millionen bei einer Reduktion des Energieverbrauchs um 23% auf ca. 1,3 Milliarden Einheiten.

Unser nächstes Ziel war eine Verbesserung der Nebenläufigkeit durch das Vermeiden von Abhängigkeiten im Datenfluss. Die bisher mehrdimensionalen Arrays zur Speicherung komplexer Vektoren und Matrizen wurden dafür zunächst auf mehrere, jeweils eindimensionale Arrays aufgeteilt. Um eine effizientere Speicherauslastung anzuregen, haben wir weiterhin Größe und Anzahl der verwendeten Datenstrukturen reduziert und diese mehrfach verwendet, sofern aktuell gespeicherte Werte nicht mehr benötigt wurden.

Durch die genannten Optimierungen konnte die Anzahl der benötigten Zeiteinheiten erneut um 41% auf 119 Millionen und der Energieverbrauch um fast 20% auf ca. 1 Milliarde Einheiten reduziert werden.

Der AMIDAR-Simulator liefert weiterhin ein integriertes Debugging-Tool. Während des Synthesevorgangs werden von diesem automatisch Log-Dateien generiert, aus denen schließlich mit Hilfe eines mitgelieferten Skripts PDF-Dateien erstellt werden können. Diese liefern eine taktgenaue Übersicht über die Verwendung einzelner PEs, wodurch auch das Erkennen von Abhängigkeiten in der Ausführung des synthetisierten Bytecodes ermöglicht wird.

In der Praxis kam es bei der Verwendung des mitgelieferten Skripts meistens zu Fehlern („*dimension too large*“). Dadurch konnten wir leider nur wenige Teile unserer Implementierung analysieren. Auch das

Erkennen von Abhängigkeiten in der Ausführung des synthetisierten Bytecodes gestaltete sich aufgrund der hohen Komplexität als nur wenig effizient.

3.4 Verbesserung des Energievebrauchs

Erfreulicherweise brachten alle bisherigen Optimierungen hinsichtlich Performanz auch eine Reduktion der benötigten Energieeinheiten mit sich. Der simulierte Energieverbrauch des Java-Prozessors wird weiterhin maßgeblich durch die Größe und Struktur des verwendeten CGRAs beeinflusst.

Der simulierte Energieverbrauch des Java-Prozessors wird maßgeblich durch die Größe und Struktur des verwendeten CGRAs beeinflusst. Unser nächster Ansatz konzentrierte sich dementsprechend auf das Experimentieren mit unterschiedlichen CGRA-Konfigurationen. Bisher hatten wir ausschließlich den mitgelieferten CGRA mit 16 PEs („*CGRA_16.json*“) in Verwendung. In der Analyse der vom Debugging-Tool erstellten Dokumente war dabei ersichtlich, dass nicht alle 16 PEs voll ausgelastet wurden.

Zunächst entfernten wir daher sukzessive PEs und erreichten damit bereits einen reduzierten Energieverbrauch von bis zu 60% auf 390 Millionen Einheiten. In vielen Fällen traten jedoch auch Fehler bei der Synthese auf, die nur teilweise durch Anpassungen an der PE-Vernetzung gelöst werden konnten. Da uns nur wenige Informationen über das Design von CGRA-Konfigurationen vorlag, konnten wir über den Sinn und die Fehlerträchtigkeit verschiedener Änderungen leider in den meisten Fällen nur spekulieren.

Eine Verkleinerung der PE-Anzahl resultierte in unseren Versuchen immer in einem stark reduzierten Energieverbrauch gegenüber einem mäßigen Anstieg der benötigten Zeitschritte. Eine mächtigere Vernetzung der verwendeten PEs erhöhte den Energieverbrauch entgegen unserer Erwartungen in nicht nennenswertem Maße, reduzierte allerdings die Anzahl der benötigten Zeitschritte und das generelle das Auftreten von Fehlern bei der Synthese.

4 Zusammenfassung

Erfreulicherweise brachten alle Optimierungen an unserer Implementierung des Akquisitions-Algorithmus hinsichtlich Performanz auch eine Reduktion der benötigten Energieeinheiten mit sich. Die dieser Ausarbeitung beigefügten Abgaben enthalten dementsprechend den gleichen Quelltext („Akquisition.java“) sowohl in der leistungsstarken als auch der energiesparenden Variante.

Die insgesamt niedrigste Anzahl benötigter Zeiteinheiten erreichten wir in unseren Versuchen unter Verwendung des mitgelieferten CGRAs mit 16 PEs und einem Unroll-Faktor von 16. Die PEs waren hierbei mit ihren unmittelbaren Nachbarn verbunden (PE0 mit PE1 und PE4, PE1 mit PE0, PE2 und PE5, usw.). Bei 111.911.879 benötigten Zeiteinheiten belief sich der Energieverbrauch dieser Konfiguration auf 886.024.210,065 Einheiten.

Den niedrigsten Energieverbrauch erreichten wir dagegen unter Verwendung einer eigenen CGRA-Konfiguration. Mit lediglich 3 vollvermaschten PEs und einem Unroll-Faktor von 8 erreichten wir in unseren Versuchen ein Minimum von 390.292.076,185 Energieeinheiten und verbesserten uns damit um fast 56% gegenüber der leistungsstarken Variante. Die Anzahl benötigter Zeiteinheiten stieg dabei jedoch um fast 53% auf 171.152.320.

Ein Unroll-Faktor von mehr als 16 führte in unseren Versuchen zu unverhältnismäßig langen Synthesezeiten und wurde daher nicht näher betrachtet. Die Vermutung liegt nahe, dass dieser Umstand mit dem verfügbaren Arbeitsspeicher der ausführenden Maschine zusammenhängt.

Großes Verbesserungspotential sehen wir weiterhin in den verwendeten CGRA-Konfigurationen zu deren Design wir leider keine zufriedenstellende Literatur finden konnten.