Palo Praktikum 2 - Writeup

Kacper Cömlek

06.05.2024

Contents

1 Aufgabenstellung

1.1 Aufgabe 1

- Beispielprogramm: 020_06_CompilerExercise
  - In den Home Ordner kopieren
  - Programm mit verschiedenen Compileroptionen aufrufen und Laufzeit notieren
- Das Programm hat mehrere Durchläufe, berechne die mittlere gerundete Laufzeit
  - Außreißer wie z.B. der erste Durchlauf werden ignoriert

- Die Ergebnisse haben eine Checksum und Total, die überprüft werden müssen. Wenn die Optimierung die Checksum verändert, ist das Ergebnis falsch. In der Tabelle kann dann *n.a. (nicht anwendbar)* eingetragen werden.
- Sofern nicht anders angegeben verwenden wir Floating-Point Modell precise. Ggf. explizit einstellen.
- In den Einzelnen Schritten Report Dateien anlegen und versuchen anhand dessen die Beobachtungen zu erklären.

1.2 Aufgabe 2

Es sollen folgende Unterpunkte in eigenen Worten erklärt werden:

- Contant folding und constant propagation
- Loop unrolling
- Scheduling
- Algebraic reduction

Welche Optimierung der Compiler jeweils vornimmt und welche Vorteile und ggf. Nachteile die Optimierung haben kann. Falls angegeben beschreiben Sie bitte auch was den Compiler ggf. daran hindern kann die jeweilige Optimierung vorzunehmen.

2 Aufgabenlösung

2.1 Aufgabe 1 Lösung

Meine Lösung für Aufgabe 1 ist in Tabelle 1 dargestellt...

2.1.1 Aufgabe 1 - Compileroptionen

Meine Compileroptionen & Checksum

```
#ICC Compilation
CXX_ICC = icpc
CXXFLAGS_ICC = -fopenmp -fp-model=precise -O0
OPTFLAGS_ICC = -qopt-report=5 -qopt-report-file=$@.optrpt
LDFLAGS_ICC =  -qopenmp -qopt-report-file=ipo_test.optrpt
               -qopt-report=5

Total = 19590.488262 Check Sum = 5001000000

#ICC Compilation für ipo
CXX_ICC = icpc
```

```
CXXFLAGS_ICC = -fopenmp -fp-model=precise -O3 -ipo
OPTFLAGS_ICC = -qopt-report=5
LDFLAGS_ICC = -qopenmp -qopt-report-file=ipo_test.optrpt
              -qopt-report=5
```

2.1.2 Aufgabe 1 - Laufzeiten

Es folgen die Laufzeiten für die verschiedenen Optimierungen

Table 1: Optimization Options

Optionen	**Mittlere Laufzeit**
/00 (keine Optimierung)	4,1786s
/02 (default Optimierung)	4,1032s
/03 (aggressive Optimierung)	4,1929s
Wählen Sie für die folgenden Schritte das Setting /03	
Prozessorspezifische Optimierung -xHost	4,0935s
Interprozeduale Optimierung (multi-File) -ipo	3,5788s
Interprozeduale + Prozessorspezifische Optimierung	3,5753s
Prozessorspezifische Optimierung + Interprozeduale Optimierung (multi-File) + Floating Point Model /fast	2,4554s

2.2 Aufgabe 2 Lösung

Moderne Compiler führen viele Veränderungen auf dem Code Durch, um die Laufzeit zu verbessern. Für den Entwickler ist es äußerst nützlich zu wissen, was der Compiler kann und was er nicht kann.

2.2.1 Material

- optimizing_cpp.pdf Kapitel 8.1

2.2.2 Constant Folding und Constant Propagation

In diesem Optimierungsverfahren werden mathematische Ausdrücke in Konstanten umgewandelt. So muss das Programm nicht in jedem Durchlauf die Berechnung erneut durchführen. Sondern der Compiler macht die berechnung einmal vorher und speichert das Ergebnis.
Beispiel:
$int\ a = 5 * 2;$
$int\ b = 2 + 3;$
$int\ c = b/a;$
wird zu

$int\ a = 10;$
$int\ b = 5;$
$int\ c = 0.5;$

Genau so würde auch eine Funktion ggf. in eine inline Konstante umgewandelt werden. Desweiteren wird empfohlen die Ausdrücke mit Klammern zu umschließen, um die Reihenfolge der Berechnung zu garantieren und somit die Konstantenbildung zu ermöglichen.
Eine Limitierung stellen jedoch komplexere Funktionen dar wie zum Beispiel die Sinusfunktion. Diese kann in der Regel nicht in eine Konstante umgewandelt werden. Manche Compiler schaffen *sqrt* oder *pow* Funktionen zu optimieren, jedoch ist dies nicht immer der Fall.

2.2.3 Loop Unrolling

In dieser Optimierungsmethode werden Schleifen durch Inline-Code ersetzt. Dies wird insbesonders bei kleinen Schleifen durchgeführt. **Beispiel:**

```
for (int i = 0; i < 4; i++) {
  a[i] = b[i] + c[i];
}
```

wird zu

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
```

Leider "unrollen" manche Compiler zu viel, dies führt dazu dass der code cache, micro-op cache und der loop buffer überlastet wird. Dies führt zu einer schlechteren Performance. Bei aggressiveren Optimierung wird eher unrolled, man muss also aufpassen ob wirklich eine Verbesserung stattfindet.

2.2.4 Scheduling

Der Compiler versucht die Reihenfolge der Befehle zu optimieren. Dies wird insbesonders bei Pipelining und Superskalarprozessoren wichtig. Der Compiler versucht die Befehle so anzuordnen, dass die Pipelines nicht unterbrochen werden.
Moderne CPUs können auch ohne die Hilfe des Compilers die Befehle umordnen, jedoch ist es für den CPU leichter, wenn der Compiler dort mithilft.

2.2.5 Algebraic Reduction

Die meisten Compiler können einfache mathematische Ausdrücke verinfachen wie etwa:

$$-(-a) = a$$

Entwickler würden eher unwahrscheinlich solch einen Ausdruck schreiben, jedoch kann es sein dass solch ein Ausdruck infolge anderer Optimierungen entsteht wie durch zum Beispiel Inlining.

Entwickler schreiben dennoch Ausdrücke welche verinfacht werden können. Das kann aufgrund der Lesbarkeit beispielsweise passieren. Beispiel:

```
if(!a && !b) => if(!(a || b))
```

Der Linke Ausdruck ist leichter zu lesen auch wenn er eine extra Operation braucht. Der Compiler optimiert das nach rechts.
Da es in der Algebra sehr viele Regeln für die Umformung und Vereinfachung gibt, gibt es keinen Compiler welcher alle Regeln umsetzen kann. In der Regel implementieren die Compiler unterschiedliche Regeln und sind verschieden gut im verinfachen von leichteren oder komplexeren Ausdrücken.
Außerdem sind die Compiler besser da drin Integer zu vereinfachen auch wenn für Floats die selben Algebra-Regeln gelten. Das hat den Hintergrund dass bei Floats leicher ungewollte Nebeneffekte auftreten können. Speziell der Verlust der Präzision ist ein Risiko. Grundsätzlich ist es am sichersten algebraische Optimierungen händisch / manuel auszuführen.

2.3 Aufgabe 3 Lösung

Erster Durchlauf ohne Veränderungen:

```
CXXFLAGS_ICC = -qopenmp -fp-model=precise
kc2170s@fb05s-es-cont01:~/dev/Vorlage$ make run-icc
./01_Original_icc
#### Runtime Measurements Particle Simulation  ###
Run 0: Runtime: 10.54146103, GLOPS/s: 0.50926401, Not in Average
Run 1: Runtime: 10.54218103, GLOPS/s: 0.50922901, Not in Average
Run 2: Runtime: 10.54181003, GLOPS/s: 0.50924701, Not in Average
Run 3: Runtime: 10.54113603, GLOPS/s: 0.50927901,
Run 4: Runtime: 10.54284603, GLOPS/s: 0.50919701,
Run 5: Runtime: 10.54302903, GLOPS/s: 0.50918801,
Run 6: Runtime: 10.54333003, GLOPS/s: 0.50917301,
Run 7: Runtime: 10.54104103, GLOPS/s: 0.50928401,
Run 8: Runtime: 10.54194603, GLOPS/s: 0.50924001,
Run 9: Runtime: 10.54319703, GLOPS/s: 0.50918001,

####### Average Performance #########
Average Runtime: 10.54236103 +- 0.000370 Seconds
Average Performance: 0.50922003 +- 0.00001803 GFLOPS/s
#####################################


## Ohne Flags
CXXFLAGS_ICC =
```

```
#### Runtime Measurements Particle Simulation  ###
Run 0: Runtime: 0.75828603, GLOPS/s: 7.07962601, Not in Average
Run 1: Runtime: 0.75850603, GLOPS/s: 7.07757201, Not in Average
Run 2: Runtime: 0.75819003, GLOPS/s: 7.08052301, Not in Average
Run 3: Runtime: 0.75825103, GLOPS/s: 7.07995101,
Run 4: Runtime: 0.75822003, GLOPS/s: 7.08024101,
Run 5: Runtime: 0.75836703, GLOPS/s: 7.07887001,
Run 6: Runtime: 0.75840703, GLOPS/s: 7.07849601,
Run 7: Runtime: 0.75811503, GLOPS/s: 7.08122501,
Run 8: Runtime: 0.75824703, GLOPS/s: 7.07999101,
Run 9: Runtime: 0.75814103, GLOPS/s: 7.08098001,

####### Average Performance #########
Average Runtime: 0.75825003 +- 0.000041 Seconds
Average Performance: 7.07996503 +- 0.00038003 GFLOPS/s
#####################################
```

2.3.1 a) Erstellen der Baseline

Die Baseline wurde ohne Optimierungen erstellt, um die Performance vergleichbar zu messen.

```
CXXFLAGS_ICC = -O0
#### Runtime Measurements Particle Simulation  ###
Run 0: Runtime: 13.49689803, GLOPS/s: 0.39774901, Not in Average
Run 1: Runtime: 13.48814203, GLOPS/s: 0.39800701, Not in Average
Run 2: Runtime: 13.49010003, GLOPS/s: 0.39795001, Not in Average
Run 3: Runtime: 13.49346803, GLOPS/s: 0.39785001,
Run 4: Runtime: 13.47982303, GLOPS/s: 0.39825301,
Run 5: Runtime: 13.49495703, GLOPS/s: 0.39780601,
Run 6: Runtime: 13.49897003, GLOPS/s: 0.39768801,
Run 7: Runtime: 13.49980603, GLOPS/s: 0.39766401,
Run 8: Runtime: 13.49966103, GLOPS/s: 0.39766801,
Run 9: Runtime: 13.49636003, GLOPS/s: 0.39776501,

####### Average Performance #########
Average Runtime: 13.49472103 +- 0.002648 Seconds
Average Performance: 0.39781403 +- 0.00007803 GFLOPS/s
#####################################
```

Um die Performance zu Micro-Benchmarken werden folgende Flaggen verwendet:

```
LDFLAGS_ICC = -qopenmp -lbenchmark -lpfm

Particle partikelI = partikel[i];
Run 4: Runtime: 12.93897903,     GLOPS/s: 0.41490001,
```

```
Run 5: Runtime: 12.93852903,     GLOPS/s: 0.41491401,

softening außerhalb der Schleifen
Run 4: Runtime: 12.88132003,     GLOPS/s: 0.41675701,
Run 5: Runtime: 12.88253703,     GLOPS/s: 0.41671801,

Final:
kc2170s@fb05s-es-cont01:~/dev/Vorlage$ make run-icc
./01_Original_icc
#### Runtime Measurements Particle Simulation  ###
Run 0: Runtime: 5.35824803, GLOPS/s: 1.00189101, Not in Average
Run 1: Runtime: 5.35637103, GLOPS/s: 1.00224201, Not in Average
Run 2: Runtime: 5.35758803, GLOPS/s: 1.00201501, Not in Average
Run 3: Runtime: 5.35826003, GLOPS/s: 1.00188901,
Run 4: Runtime: 5.35855003, GLOPS/s: 1.00183501,
Run 5: Runtime: 5.35833003, GLOPS/s: 1.00187601,
Run 6: Runtime: 5.35620103, GLOPS/s: 1.00227401,
Run 7: Runtime: 5.81265103, GLOPS/s: 0.92356801,
Run 8: Runtime: 5.35562303, GLOPS/s: 1.00238201,
Run 9: Runtime: 5.36150303, GLOPS/s: 1.00128301,

####### Average Performance #########
Average Runtime: 5.42301703 +- 0.064943 Seconds
Average Performance: 0.99073003 +- 0.01119403 GFLOPS/s
#####################################
```