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 angegben 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~~ 32,635s
/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

2.1.3 Aufgabe 1 - Die Beobachtungen aus den Report-Dateien

Bei der "Default-Optimierung", "Aggressive-Optimierung" und "Prozessorspezifische Optimierung" wird weder Inlining oder Vektorisierung vom Compiler betrieben.

Bei der "Interprozeduale Optimierung (multi-File)" haben wir dann das erste mal Inlining der Funktionen Work(double*, double*), Serier1(int), Serier2(int) und AddY(double, int) aber auch keine Vektorisierung der Schleifen. Bei dem fast Model haben wir hingegen Inlining und auch Vektorisierung von Serier1(int) und Serier2(int).

Bei der "Interprozeduale + Prozessorspezifische Optimierung" kommt es darauf an ob man es mit Multi-File-Betrachtung laufenlässt oder ohne. Bei mit Multi-File-Betrachtung verhält es sich wie bei der "Interprozeduale Optimierung (multi-File)" und ohne die Betrachtung verhält es sich wie bei den ersten drei beschriebenen Optimierungen.

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.

3 Aufgabe 3 - Skalare Optimierung

Erstellen einer Baseline

Table 2: Laufzeitmessung

Average Runtime	Average Performance
10.5561s	0.5085 GFLOPS/s

Compilereinstellungen

```
#ICC Copilation
CXX_ICC = icpc
CXXFLAGS_ICC = -qopenmp -fp-model=precise
OPTFLAGS_ICC = -O2 -qopt-report=5 -qopt-report-file=$@.optrpt
LDFLAGS_ICC = -qopenmp
```

Wir haben uns für diese Compilereinstellungen entschieden, da wir es für sinnvoll fanden das der Compiler nur mit der Default-Optimierung läuft. Für diese haben wir uns entschieden da wir nicht wollten das der Compiler unseren gesamten Code von sich aus schon Optimiert und wir wollten vermeiden das der Compiler Sachen weg Optimiert die vielleicht wichtig sind.

Bei dem Model haben wir uns für "precise" entscheiden da wir der Meinung waren das es sinnvoll wäre, wenn der Compiler die Reihenfolge des Programmes beibehält. Wir wollten das das Programm in der vorgegebenen Reihenfolge aus geführt wird damit wir später besser sehen können ob unsere Optimierungen der Berechnungen etwas gebracht haben oder nicht.

Skalare Optimierung

Table 3: Ergebnisse des Benchmarkings

Benchmark	Time (ns)	CPU Iterations
BM_New_Process_Ref	64.3	1000000
BM_New_Process_Opt	32.1	2000000
BM_New_Process_Opt	16.5	3000000
BM_New_Process_Opt	3.11	224955341

Table 4: Laufzeitmessung

Average Runtime	Average Performance
2.9314 Seconds	1.8313 GFLOPS/s

Die Änderungen am Quellcode

Vorher:

```
for (int j = 0; j < nr_Particles; j++) {
const float dx = partikel[j].x - partikel[i].x;
const float dy = partikel[j].y - partikel[i].y;
const float dz = partikel[j].z - partikel[i].z;
...
}
```

Nachher

```
Particle partikelI = partikel[i];
...
for (int j = 0; j < nr_Particles; j++) {
const float dx = partikel[j].x - partikelI.x;
const float dy = partikel[j].y - partikelI.y;
const float dz = partikel[j].z - partikelI.z;
...
}
```

Vorher

```
const float drPower32 = pow(drSquared, 3.0 / 2.0);
```

Nachher

```
const float drPower32 = drSquared * sqrt(drSquared);
```

Vorher

```
Fx += dx / drPower32;
Fy += dy / drPower32;
Fz += dz / drPower32;
```

Nachher

```
Fx += dx * invDrPower32;
Fy += dy * invDrPower32;
Fz += dz * invDrPower32;
```