Optimierung von Programmen

Programmebene

Christian Pape

Hochschule Karlsruhe

28. März 2019

Inhalt

Begriffe

Sprunganweisungen

Vergleiche und dynamisches Binden führt in Maschinencode zu Sprunganweisungen.

- **Bedingte Sprünge**: besitzen eine Bedingung im Maschinenbefehl (z.B. BEQ #100).
- **Unbedingte Sprünge**: immer ausgeführt mit absoluter Zieladresse (z.B. JSR $1000).
- **Indirekten Sprünge**: Zieladresse erst aus einer Speicherzelle oder Register holen (z.b. JMP $(1000)).

Eine **Sprungvorhersage** (innerhalb Pipeline):

1. Bei bedingten Sprüngen herausfinden, ob der Sprung durchgeführt wird und
2. die korrekte Zieladresse zu bestimmen.

Begriffe

Pipeline

- Befehle der Hochsprache werden in ein oder mehrere Maschinenbefehle übersetzt.
- Maschinencode wird oft in Microcodebefehle zerlegt (bei x86).
- Microcode wird auf Pipelines verteilt.
- Pipeline arbeitet stufenweise Befehle ab.

Begriffe

Pipeline

- Nicht jede Pipeline kann jeden Befehl ausführen, z.B. keine Gleitkommabefehle.
- Jede Pipeline kann in der Regel sehr einfache Befehle ausführen, z.B. Verschiebeoperation.
- Vorhersage bei Sprüngen und Laden von Werten kann fehlschlagen: Hohe Kosten

Ein korrekt vorhergesagte Verzweigung kostet ca. 0–2 Taktzyklen, während eine fehlgeschlagene Vorhersage ca. 12–25 Taktzyklen verursacht. [Fog, 2017, Abschnitt 7.12]

Programmverbesserung

Ziele

CPU insbesondere die Pipelinestufen vollständig auslasten.

- Unnötige Befehle einsparen.
- Komplexe (langsame) Befehle vermeiden.
- Sprünge vermeiden oder reduzieren.
- Speicherzugriffe reduzieren oder auf 1st-level-cache hin optimieren.
- Spezialbefehle oder zusätzliche Register verwenden (Compiler).

Programmverbesserung

Befehle einsparen

Arithmetische Umformungen bei Variablen durchführen.
Beispiel Distributivgesetz:

```
a * x + a * y = a * ( x + y )
```

Es wird eine Multiplikation weniger ausgeführt.
Funktion wie sqrt, log, cos kosten viele Taktzyklen.
Anzahl Aufrufe durch Umformungen reduzieren:

```
log(a) + log(b) = log(a * b)
log( pow(a, 8.0) ) = 8.0 * log(a)
pow(a, 2.0) / pow(b, 2.0) = pow( (a*b), 2.0)
            =  a * a * b * b
```

Programmverbesserung

Befehle einsparen

Oft gebrauchte Zwischenergebnisse bei Datenstrukturen vorberechnen und zwischenspeichern.
Beispiel Schnitt Sehstrahl mit Dreick ABC:

- A,B,C Punkte im euklidischen Raum.
- Algorithmus benötigt $\overrightarrow{AB} = B - A$, $\overrightarrow{AC} = C - A$.
- Beide Vektoren bei Dreieck zwischenspeichern.
- Nachteil: Speicherverbrauch steigt (Cache).

Programmverbesserung

Befehle einsparen

Beispiele Algorithmen und Datenstrukturen.
Größte Sparpotential.
Es werden z.B. Vergleiche eingespart.

- Binäre Suche statt sequentieller Suche (Sortieren nötig).
- $O(n \log n)$ Sortierverfahren statt $O(n^2)$ (je nach Fall).
- Raumunterteilung beim Raytracing, z.B. k-d-Bäume (Rechneraufgabe 3).
- Konstanter Faktor teilweise relevant.

Programmverbesserung

Befehle einsparen

Beispiele aus Aufgaben:

- Rechneraufgabe 1: Schnittpunktberechnung früher abbrechen, wenn Schnittpunkt weiter entfernt ist als bis gefundener.
- Rechneraufgabe 1: Anzahl sqrt Aufrufe von 3 auf 2 reduzieren.
- Taylorreihe e^x: Fakultät und Potenzen nicht immer wieder vollständig neu berechnen.

Programmverbesserung

Befehle einsparen

Beispiel: Sortieren durch direktes Einfügen.

- **Wächterelement** verwenden, um Sonderfall einzusparen.
- Erste Element mit Minimum der Folge vertauschen.
- Zusätzliche Kosten: $O(n)$
- Einsparpotenial: $O(n^2)$ im schlimmsten und durchschnittlichen Fall.

Programmverbesserung

Komplexe Befehle vermeiden

Tipps für C++

- Datentypen: Ganzzahlig statt Gleitkomma. Nicht mischen.
  int x = i ∗ 0.5 − j ∗ 0.5;
- Gleitkommadivision vermeiden: 0.5 ∗ a statt a / 2.0.
- **unsigned int** am schnellsten oder **int**
- **unsigned int** Division schneller als bei **int**
- ++i statt i++ bei Iteratoren in Ausdrücken.
- ∗(p++) (analog a[i++]) schneller als ∗(++p) (a[++i]).
- STL-Funktionen statt eigener verwenden: Iteratoren und
  #include <algorithm>
- Templates

Viele Websites mit Tipps und Tricks.

Programmverbesserung

Komplexe Befehle vermeiden

std : swap, std : min, std : max, std : sort , std : rotate , std : find

Beispiel: Direktes Einfügen

- Einzufügenden Wert nicht durchrutschen lassen (viele komplexe Vertauschungen)
- Stelle zum Einfügen suchen (viele weniger komplexe Vergleiche).
- Verschieben aller Werte auf einmal nach rechts und
- Wert an richtige Stelle einfügen (ein "Mega"tausch)

Programmverbesserung

Komplexe Befehle vermeiden

Logische statt ganzzahlige arithmetischen Operationen verwenden:

```
i * 4 = i << 2
i / 2 = i >> 1
2.0 * a = a + a
i * 12 = i * 8 + i * 4 = i << 3 + i << 2
```

Diese Umformungen sind nur bei unsigned Datetypen (<<) oder positiven Werten (>>) korrekt (wieso?)
Macht in der Regel schon der C/C++-Compiler.

Programmverbesserung

Komplexe Befehle vermeiden

Quadratwurzelberechnung $\sqrt{a}$ (Rechneraufgabe 2).
Newton-Verfahren:

$$x_{n+1} = 0,5 \cdot (x_n - \frac{a}{x_n})$$

- Geeigneten Startwert sollte größenordnungsmäßig stimmen.
- Idee: $\sqrt{2^n} \geq \sqrt{2^{\lfloor \frac{n}{2} \rfloor}}$
- float: Exponent n ist binär codiert. Rechtverschiebung entspricht $2^{\lfloor \frac{n}{2} \rfloor}$.
- Fehlerkorrektor mit Und-Verknüpfung (Charakteristik berücksichtigen).
- Startwert kann mit logischen, ganzzahligen Verknüpfungen ermittelt werden.

Programmverbesserung

Komplexe Befehle vermeiden

```
// ap sei ein Zeiger auf den float Wert a
// rp sei ein Zeiger auf den Wert der Wurzel
int *ai = reinterpret_cast<int *>(ap);
int *initial = reinterpret_cast<int *>(rp);
*initial =  (1 << 29) + (*ai >> 1)
          - (1 << 22) - 0x4C000;
// initial zurueck casten zu float *
// und Newton Verfahren durchfuehren
```

- Alternative: Tabelle mit Startwerten, a wird zu "Hashwert" reduziert.
- Intel-Prozessor: fsqrt als Maschinenbefehl, AVX-Befehl vsqrts 2x schneller

Programmverbesserung

Sprünge vermeiden

- Nur verbessern, falls Sprungvorhersage zu oft versagt.
- Messen!
- Sprünge in Code:
  - Kontrollanweisungen if, switch, while, do-while, for
  - Operatoren wie &&, ?:
  - Funktionsaufrufe und -Ende (return, throw).

Programmverbesserung

Sprünge vermeiden

Beispiel: Euklidischer Algorithmus

```
#include <iostream>

long long getGGT(long long a, long long b) {
  while (a != b) {
    if (a > b) {
      a = a - b;
    } else {
      b = b - a;
    }
  }
  return a;
}
```

Sprünge mit `valgrind --tool=cachegrind --branch-sim=yes` untersucht mit Beispiel
getGGT$(2 * 5 * 11 * 11 * 17, 3 * 2 * 7 * 11 * 13 * 19)$
Annotierter Quelltext mit
`cg_annotate <cachegrind.out.XXXXX> <source>`.
Quelltext mit absolutem Pfad angeben (Unix Bug)

Programmverbesserung

Sprünge vermeiden

- Bc: conditional branches counted
- Bcm: conditional branches mispredicted
- Bi: indirect branches counted
- Bim : indirect branches mispredicted

```
Bc    Bcm   Bi Bim
36    6     0    0     while (a != b) {
108   19    0    0       if (a > b) {
```

Programmverbesserung

Speicherzugriffe reduzieren

Versuche if-else in if umzuwandeln.
Idee: Anweisung des else-Zweig immer durchführen. Fehler im if-Fall korrigieren.

```
long long getGGT(long long a, long long b) {
  while (a != b) {
    b = b - a;
    if (b < 0) { // a > b (b wurde negativ)
      b = b + a; // Fehler rückgängig machen
      a = a - b;
    }
  }
  return a;
}
```

Programmverbesserung

Sprünge vermeiden

- Bc: conditional branches counted
- Bcm: conditional branches mispredicted
- Bi: indirect branches counted
- Bim : indirect branches mispredicted

```
Bc    Bcm   Bi Bim
36    6     0    0     while (a != b) {
108   19    0    0       if (a > b) {
```

Programmverbesserung

Sprünge vermeiden

- Bc: conditional branches counted
- Bcm: conditional branches mispredicted
- Bi: indirect branches counted
- Bim: indirect branches mispredicted

```
Bc   Bcm   Bi Bim
36    12    0   0     while (a != b) {
 0     0    0   0       b = b - a;
36     6    0   0       if (b < 0) { // a > b (b wurde negativ)
```

Anzahl Sprünge reduziert. Prozentual mehr misspredictions. Absolut weniger.
Zusätzliche Anweisungen.

Programmverbesserung

Sprünge vermeiden

- Bitweise Verknüpfung & statt Boolesche Kurzschlussoperator &&.
- Nur möglich, wenn Operanden 0 oder 1 sind.
- Wert eine Booleschen Operation immer 0 oder 1 (&&, >)

Beispiel: Punkt in AABB (axis alined bounding box).

Programmverbesserung

Sprünge vermeiden

```
bool pointInRectangle1(double a, double b,
        double x, double y, double w, double h) {
  return (a >= x) && (a <= x + w)
          && (b >= y) && (b <= y + h);
}

bool pointInRectangle2(double a, double b,
        double x, double y, double w, double h) {
  return (a >= x) & (a <= x + w)
          & (b >= y) & (b <= y + h);
}
```

Messung: kein Unterschied. Bei ganzen Zahlen ggf. anders.

Programmverbesserung

Speicherzugriffe verbessern

C/C++-Speichermodell
Variablen anlegen und zerstören kostet Zeit

- Laufzeitkeller (call stack): stark begrenzt
- Statischer Speicherbereich (static memory)
- Dynamischer Speicherbereich (Heap)
- thread_local: objekt pro Thread

(Zeit nimmt von oben nach unten zu)

Programmverbesserung

Speicherzugriffe verbessern

Flaschenhals Hauptspeicher, Daten-Cache.
Funktionsweise CPU (Zugriffszeiten von oben nach unten nehmen ab)

- RAM (GB)
- 3rd, 2nd (MB)
- 1st (32K)
- CPU-Register

Bis Faktor 100 Latenzzeiten, wenn Daten von RAM in CPU-Register geladen werden müssen.
Instruction-Cache: Schleifenrümpfe etc. nicht zu groß

Programmverbesserung

Speicherzugriffe verbessern

Daten für häufige Berechnung im Cache behalten.

- Prefetch (Hardware, Software)
- Aufteilung Datenstrukturen in "heisse" und "kalte" Daten.
- Datenstrukturengrösse an cache-line Grösse anpassen (64 bytes).

Programmverbesserung

Speicherzugriffe verbessern

Beispiel: Dreick im Raytracer

- Drei Koordinaten (3 x 3 x sizeof(float) = 9 x 4 = 36 bytes), Drei Normalenvektoren (36 bytes)
- Koordinaten werden häufig verwendet (heiss)
- Normalenvektoren erst beim Shading (kalt).

Derzeit:

```
class Triangle {
public:
  Vector<float, 3> p1, p2, p3; // edges
  Vector<float, 3> n1, n2, n3; // normals
..
}
```

Programmverbesserung

Speicherzugriffe verbessern

Offensichtlich Aufspaltung:

```
class TriangleNormals {
public:
  Vector<float, 3> n1, n2, n3; // normals
..
}

class Triangle {
public:
  Vector<float, 3> p1, p2, p3; // edges
  TriangleNormals *normals; // normals
..
}
```

sizeof(Triangle) = 36 + 8 = 44 byte

Nachteil: 2 Triangle passen nicht vollständig in eine Cache-line

Bei 2-dim-Vektoren: 2 Triangle passen in eine Cache-line

Programmverbesserung

Speicherzugriffe verbessern

Zeiger einsparen.

Aufspaltung mit zwei getrennten Listen.

```
class TriangleNormals {
public:
  Vector<float, 3> n1, n2, n3; // normals
..
};

class Triangle {
public:
  Vector<float, 3> p1, p2, p3; // edges
};

vector<Triangle> triangles;
vector<TriangleNormals> triangleNormals;
// Zusammenhang ueber Index
```

Programmverbesserung

Speicherzugriffe verbessern

Tricks, um Daten (auf cache-line-Grösse) zu reduzieren.

- Bit fields nutzen.
- Unbenutzte Bits in Variablen anderweitig verwenden.
- Komprimieren.

Programmverbesserung

Speicherzugriffe verbessern

```
struct { // bit fields
  unsigned int a : 16;
  unsigned int b : 6;
  unsigned int c : 4;
  unsigned int d : 2;
  unsgined int e : 12;
};
```

- Wertebereich durch Anzahl Bits eingeschränkt $0 \leq d \leq 3 = 2^2$
- Keine Padding, wenn alle Bits in Datentyp der vorherigen Variable passen.
- a bis d verbrauchen höchstens 32-Bit.
- Zugriffszeiten verlangsamen sich.

Programmverbesserung

Speicherzugriffe verbessern

Unbenutzte Bits anderweitig verwenden.
Beispiel: Rot-Schwarz-Baum und Zeiger

- Ein Rot-Schwarz-Baum ist ein binärer Suchbaum.
- Knoten können schwarz oder rot sein.
- Schwarze Knoten bilden einen vollständig balancierten Suchbaum.
- Es können zusätzlich rote Knoten darin vorkommen.
- Höchstens so viele rote wie schwarze Knoten auf einem Pfad.
- 1 Bit Zusatzinformation nötig.

Programmverbesserung

Speicherzugriffe verbessern

Beispiel: Rot-Schwarz-Baum und Zeiger

Annahme: Hardware hat bei Suchdaten Alignment auf gerade Adressen

Niederwertigeste Bit im Zeiger auf die Daten verwenden.

```
struct Knoten {
  void *daten;
  Knoten *links, *rechts;

  bool isRot() {
    return (daten & 0x1);
  }

  void *getDaten() {
    return (daten & ~0x1);
  }
};
```

Programmverbesserung

Speicherzugriffe verbessern

Mehr Bits verwenden:

- new überschreiben (für bestimmten Datentyp)
- Grösseren Speicherblock reservieren
- Alignment selbst berechnen

Nachteil: Speicherverbrauch auf dem Heap wächst

Programmverbesserung

Spezialbefehle verwenden (SIMD)

- Streaming SIMD Extensions (SSE) ist eine Befehlserweiterung von Intel.
- Entworfen, um die vorangehenden MMX Erweiterungen für integer-Berechnungen zu erweitern.
- Als Reaktion auf AMDs 3DNow! Technologie für die Intel Pentium III Prozessorserie eingeführt
- Vektorbefehle für 3D-Grafik, Physik oder mathematische Anwendungen
- Erweiterungen SSE2, SSE3/SSE3S und SSE4, AVX (Advanced Vector Extensions), AVX-2 erweitert.
- Aktueller Standard ist AVX-512 für Serverprozessoren.

Programmverbesserung

Spezialbefehle verwenden (SIMD)

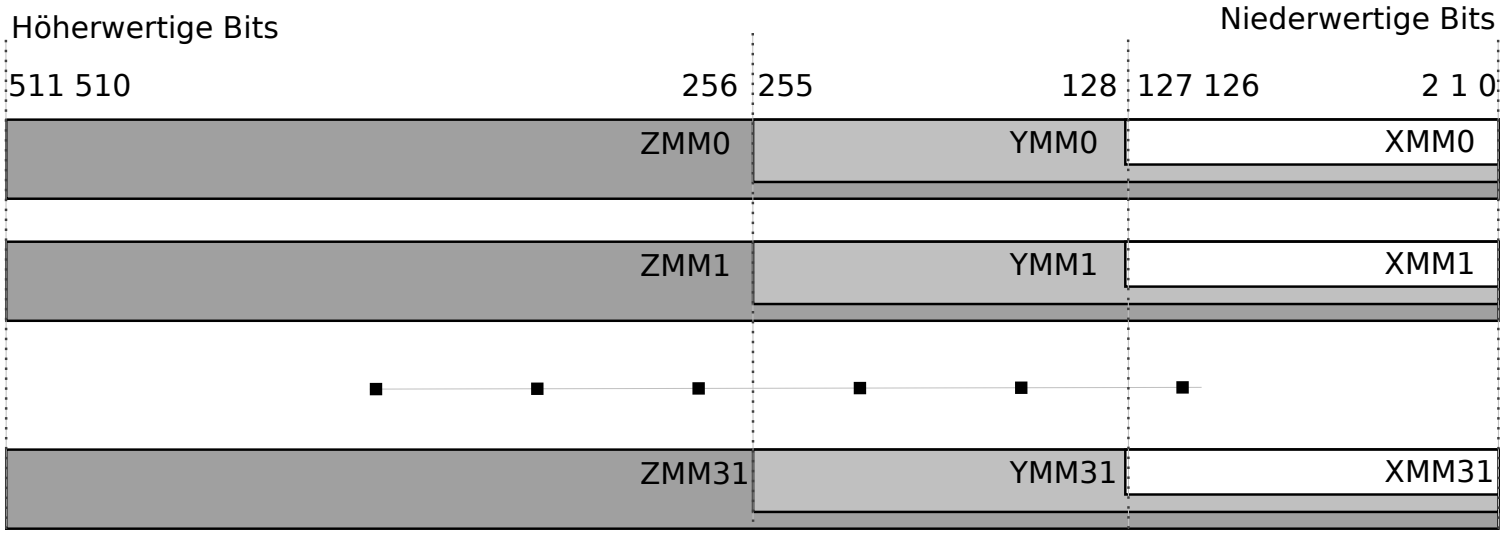
- SSE kann 4-float-Werte auf einmal mit der gleichen Operation verarbeiten.
- SSE2 erweitert dies um 2-double-Werte.
- SSE erweitert ie CPU-Architektur um 16 neue 128-Bit Register xmm0 bis xmm15.
- Ein Register kann vier float-Werte oder zwei double-Werte speichern.
- Die SSE-Befehle für diese Register verarbeiten die enthaltenen Werte parallel.
- AVX (Advanced Vector eXtension) erweitert die Register auf 256-Bit (seit 2008).
- AVX-512 noch einmal auf 512-Bit.

Programmverbesserung

Spezialbefehle verwenden (SIMD)

AVX-512 (in E203 AVX-2)

- 32 512-Bit breite SIMD-Register (ZMM0 bis ZMM31).
- Bis acht double Operationen auf einmal.
- Zur Abwärtskompatibilität auch im 64-Bit (XMM) oder 128-Bit (YMM) Modus arbeiten.
- Im 32-Bit Modus sind nur die acht Register XMM0 bis XMM7 verfügbar.



Programmverbesserung

Spezialbefehle verwenden (SIMD)

Beispiel addieren:

```
addps %xmm0 %xmm1
```

- addps: addiere packed single precision (alle vier float werden addiert)
- addss: addiere scalar single precistion (nur float in LSB wird addiert)
- addpd, addsd: für double
- vaddps: (AVX) 256-Bit Register packed single precision
- Postfix i statt s,d für integer.

`http://softpixel.com/~cwright/programming/simd/sse.php`
Ziel der Optimierung: Der Compiler soll packed Befehle erzeugen!

Programmverbesserung

Spezialbefehle verwenden (SIMD)

Daten müssen in Register geladen werden.

- Bis AVX-2: sehr striktes Memory-Alignment
- Daten müssen auf 64, 128, 256 Bit (8, 16, 32 byte) Grenze liegen.
- Programm stürzt andernfalls ab (Speicherzugriffsverletzung)
- AVX-512: weniger strikt durch neue Lade/Speicherbefehle (langsamer)
- Compiler legt die Daten in der Regel nicht so ab.

C++-Schlüsselwort alignas verwenden.

Programmverbesserung

Spezialbefehle verwenden (SIMD)

Beispiel (aus CPP-Referenz):

```
// every object of type sse_t will be aligned to 16-byte boundary
struct alignas(16) sse_t
{
  float sse_data[4];
};

// the array "cacheline" will be aligned to 128-byte boundary
alignas(128) char cacheline[128];
```

- Maximaler unterstützter alignmen-Wert ist Compiler-abhängig.
- Standard: etwa 8?
- GCC: höher
- Nur für Hardware gültige Werte erlaubt, 3 z.b. in der Regel nicht.
- Keine alignment für new möglich.

Programmverbesserung

Spezialbefehle verwenden (SIMD)

Verwendung SIMD-Befehle in GCC:

- x86 Built-in Functions oder
- Compiler-Erweiterungen (für Aufgaben)

Beispiel: Prototyp built-in functions für addps

```
v4sf __builtin_ia32_addps (v4sf, v4sf)
```

v4sf Datentyp für MMX-Register (vordefiniert bzw. selbst definieren)
Kein Assemblercode: Compiler hat weiterhin Kontrolle über Parameter und Rückgabewert.
SEE, AVX, Versionen: Passender Assembler wird von Compiler erzeugt.

Programmverbesserung

Spezialbefehle verwenden (SIMD)

GCC Optionen, um SSE/AVX vollständig anzuschalten:

```
-march=native -mavx2 -O3
```

`-O3` für Schleifen ausrollen.
`-march=native` schaltet alle Anweisungen, auch AVX, für CPU an, auf dem der Compiler ausgeführt wird.
Nur `-mavx2` schaltet auch die GCC-built-ins für AVX2 an.

```
-ffast-math
```

Weitere Beschleunigung, aber keine exakte ISO konforme Fließkommaberechnung mehr.

Programmverbesserung

Spezialbefehle verwenden (SIMD)

Mit Compiler-Erweiterungen mit speziellen Datentypen:

- Datentyp wie v4sf definieren.
- Verhält sich wie ein array mit 2,4, 8 Werten.
- Operatoren wie `+`, `*`, `[]`, `=` sind für SIMD-Befehle überladen.

```
v4sf x = {1.0, 2.0, 3.0, 0.0};
v4sf y = {-1.0, 2.0, 2.0, 0.0};
v4sf z = x + y; // SIMD Addition
std::cout << z[2] << std::endl; // 5.0
```

Beispiel: `add_sse.cc`.

Programmverbesserung

Spezialbefehle verwenden (SIMD)

Mit Compiler-Erweiterungen ohne speziellen Datentypen:

- Normale Datentypen verwenden, z.B. array, struct
- Operatoren wie `+`, `*`, `[]`, `=` werden vom Compiler für SIMD genutzt.
- Befehle müssen unabhängig voneinander sein.

```
float x[4] = {1.0, 2.0, 3.0, 0.0};
float y[4] = {-1.0, 2.0, 2.0, 0.0};
float z[4];
for (int i = 0; i < 4; i++) {
  z[i] = x[i] + y[i];
}
std::cout << z[2] << std::endl; // 5.0
```

Schleifen ausrollen (`-O2`) nötig.

Beispiel: `simple_sse.cc`.

Programmverbesserung

Spezialbefehle verwenden (SIMD)

Mit Compiler-Erweiterungen ohne speziellen Datentypen:

- Ganz normal programmieren.
- Daten so organisieren, dass float, double, int in 2, 4, 8, ... Gruppen zusammenstehen.
- z.B. Schnittpunktberechnung mit 4 Sehstrahlen und 1 Dreieck programmieren oder
- (Rechneraufgabe 2) vier Quadratwurzeln auf einmal berechnen.
- Erzeugten Assembler prüfen.

Beispiel: `sse_vector.cc`.

Programmverbesserung

Spezialbefehle verwenden (SIMD)

Aufgabe: Matrixmultiplikation 4 x 4 Matrix.

- Ohne SSE-Erweiterung übersetzen.
- Mit SEE-Erweiterung übersetzen.
- Assemblerausgabe vergleichen.

Programmverbesserung

Spezialbefehle verwenden (SIMD)

```
void multiplizieren(a float[4][4], b float[4][4], c & float[4][4]) {
  // "c = a * b" programmieren mit zwei Schleifen
}

void addieren(a float[4][4], b float[4][4], c & float[4][4]) {
  // "c = a + b" programmieren mit zwei Schleifen
}

// Main-Methode mit Aufrufen. ggf. alignment 16 bytes
// g++ -Wall -pedantic -mfpmath=sse -march=native -mavx2 -O3
```

Literaturverzeichnis


Fog, A. (2017).
Optimizing software in C++. An optimization guide for Windows Linux and Mac platforms.
http://www.agner.org/optimize/optimizing_cpp.pdf.