Architektur und Programmierung von Grafik- und Koprozessoren

Performanz von Computerprogrammen

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

Lernziele

1. **Systematischer Performanzbegriff** - die Studierenden sollen ein Instrumentarium erlernen, um die Performanz eines Computerprogramms, das auf einem bestimmten Prozessor ausgeführt wird, einschätzen und messen zu können.
2. **Hochparallele Architekturen** - die Studierenden verstehen die historische Entwicklung, die in den letzten zehn Jahren zu Multi-Core und Many-Core Architekturen geführt hat.
3. **Arten von Nebenläufigkeit** - die Studierenden lernen, auf welchen Ebenen Prozessoren Nebenläufigkeit exponieren, und wie dies beim Programmieren berücksichtigt werden muss.
4. **Theoretische Schranken für Parallele Programme** - die Studierenden können das Parallelisierungspotential eines Computerprogramms auf einem bestimmten, parallelen Prozessor einschätzen.

Transistoren und Performanz

Mooresches Gesetz

- Gordon Earle Moore
- Geboren am 3. Jan. 1929
- Mitbegründer der Intel Corp.

Abbildung: Gordon E. Moore, ©Steve Jurvetson, 2004, CC BY 2.0 Lizenz

Mooresches Gesetz

- **Verdopplung der Anzahl Schaltelemente (Transistordichte)** pro Integriertem Schaltkreis...
- ...**alle zwölf** ("Cramming More Components Onto Integrated Circuits" (1965), G.E. Moore, IEEE Electronics)
- ...**bis 24 Monate** ("Progress In Digital Integrated Electronics" (1975), G.E. Moore, IEEE International Electron Devices Meeting).
- Schaltelement: Bauteil, das umschaltet, wenn elektrische Spannung anliegt

Mooresches Gesetz

- Mooresches Gesetz anfangs nur mit Zeithorizont von 10 Jahren, wurde später im Rahmen der *International Technology Roadmap for Semiconductors* (ITRS) aufgegriffen und auf *Einheitsfläche* eines Integrierten Schaltkreises (IC) / Fertigungsdichte übertragen.
  - Daher wird i. d. R. heute stattdessen auf die *Strukturgröße* Bezug genommen (14nm, 10nm etc.)
- Bemerkenswert: Mooresches Gesetz bezog sich ursprünglich auf ca. 35, heute auf Milliarden von Transistoren.

Mooresches Gesetz

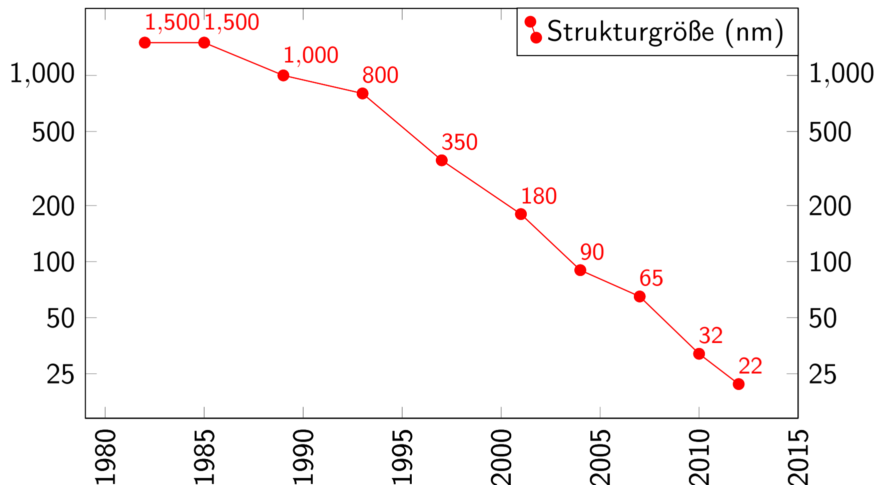

Abbildung: Historische Entwicklung Strukturgröße von Intel Prozessoren.

Bemerkungen

- Schaltelemente können verschiedensten Zwecken zugeführt werden:
  - Bauteile zur Durchführung von Berechnungen: Logikgatter, Arithmetik/Logikeinheiten (ALUs), Prozessorkerne, Erweiterungseinheiten (z. B. MMX,SSE... als “Multimedia Extensions”)
  - Speicherbauteile: Flip-Flops, Register, Caches
  - Interne Leiterbahnen (“Fabrique”)
  - Weitere Infrastruktur wie Multiplexer, Matrix Switches, etc.
- Zum besseren Verständnis der letztgenannten Maßnahmen ist ein systematischer Performanzbegriff notwendig

Was ist Performanz?

Beispiel Passagierflugzeuge

Das Flugzeug mit der höchsten Fluggeschwindigkeit war bis 2003 die Concorde mit 1350 mph. Die Concorde konnte 132 Passagiere befördern. Bei etwa gleicher Reichweite hat die Boeing 747 eine Fluggeschwindigkeit von 470 mph und ein maximales Beförderungsvolumen von 450 Passagieren.

Definiert man Performanz als proportional zur Höchstgeschwindigkeit, ist die Concorde der klare Sieger. Man kann Performanz aber auch bzgl. des *Durchsatzes* (Geschwindigkeit $\times$ Passagierzahl) definieren, dann ist die Boeing 747 überlegen.

Was ist Performanz?

Computer

Ähnlich verhält es sich mit Computern.

- Computer X ist performanter als Computer Y, wenn Programm P auf X schneller läuft als auf Y.
- Data Center Sicht: Server ist am performantesten, der möglichst viele Benutzerjobs an einem Tag abarbeiten kann (Durchsatz).
- Sicht als Individuum: Computer ist am performantesten, der die Antwortzeit, also die Zeit zwischen Initiierung und Abarbeitung einer Aufgabe, minimiert.

Was ist Performanz?

Computerprogramme

An den Beispielen sieht man, dass die Performanz eines Computers offensichtlich mit der Performanz eines oder mehrerer Computerprogramme zusammenhängt, die auf dem Computer ausgeführt werden.

I. allg. verstehen wir unter einem Computerprogramme eine Abfolge von *Befehlen* I, die sich auf damit assoziierte Daten D beziehen.

Performanz

- Wir wollen im folgenden einen systematischen Performanzbegriff erarbeiten.
- Performanz $\neq$ Komplexität.
- Wir gehen davon aus, dass die algorithmische Komplexität wohlverstanden ist und dass Algorithmen gewählt werden, die mit Bezug auf die Eingabedaten bzgl. ihrer algorithmischen Komplexität vorteilhaft sind.
- Dies kann aber bspw. auch bedeuten, dass wir aufgrund der erwarteten Menge an Eingabedaten ein Verfahren mit asymptotisch höherer Komplexität, aber z. B. mit einfacheren Speicherzugriffsmustern, ggü. dem Verfahren mit geringerer Komplexität vorziehen.
- Achtung: auf parallelen Architekturen sollte eine parallele Maschine (z. B. PRAM) zur Komplexitätsbestimmung herangezogen werden.

Performanz

- Wir interessieren uns i. Allg. für die **Performanz eines Computerprogramms**, das auf verschiedenen Prozessoren ausgeführt wird.
- Diese lässt sich u. a. mittels der absoluten Zeit, die das Programm unter Verwendung des jeweiligen Prozessors *zur Abarbeitung all seiner Befehle bzgl. eines spezifischen Inputs* benötigt, quantifizieren.
- Anmerkung: ggf. ist das schwierig, z. B. bei Multi-Tasking Betriebssystemen.
- Die so bestimmte Größe nennt man "wall clock time" bzw. "CPU execution time".

CPU Execution Time Definition (1)

CPU execution time

- Unter der CPU execution time (syn.: "wall clock time", "CPU time") versteht man die Zeit, die ein Programm auf einem Prozessor zur Abarbeitung all seiner eingabeabhängigen Instruktionen benötigt.
- Problem hierbei: welcher Teil der insgesamt verstrichenen Zeit wurde bei der Ausführung unseres Programms / des Programmbestandteils von Interesse verbracht, und wieviel Zeit wurde für Overhead, Kommunikation, ggf. andere Prozesse etc. verwendet?

CPU Execution Time Definition (2)

Die CPU execution time errechnet sich als das Produkt

$$CPU\ time = Anzahl\ Taktzyklen(I, D) \times Zeit\ pro\ Taktzyklus, \quad (1)$$

wobei I (=Instruktionen) und D (=Daten) der Input für den spezifischen Programmlauf sind. Wegen der Identität $Zeit\ pro\ Taktyklus := \frac{1}{Taktfrequenz}$ ergibt sich äquivalent:

$$CPU\ time = \frac{Anzahl\ Taktzyklen(I, D)}{Taktfrequenz}. \quad (2)$$

CPU Execution Time Definition (3)

Es gilt ferner:

$$Anzahl\ Taktzyklen(I, D) = \#I \times CPI. \tag{3}$$

CPI bezeichnet die *durchschnittliche* Instruktionszeit des Computerprogramms bzgl. des Inputs ("clock cycles per instruction").

(S)AXPY

ANSI-C Beispiel, das wir immer wieder verwenden werden:

```
void saxpy(float* s, float a, float* x, float* y, int N) {
    for (int i = 0; i < N; ++i)
        s[i] = a * x[i] + y[i];
}
```

oder

```
void saxpy(float a, float* x, float* y, int N) {
    for (int i = 0; i < N; ++i)
        y[i] = a * x[i] + y[i];
}
```

oder Varianten.

CPI (1)

Ihr Compiler übersetzt die folgende ANSI-C Funktion:

```
void saxpy(float* a, float* x, float* y) {
    for (int i = 0; i < 2; ++i)
        y[i] = a[i] * x[i] + y[i];
}
```

in folgenden x86 Assembly Code[1]:

```
movss xmm1, dword ptr [rdi]   ;Lade x[0] aus Speicher
mulss xmm1, xmm0              ;Mul mit a aus Register xmm0
addss xmm1, dword ptr [rsi]   ;Add y[0] zu Registerinhalt
movss dword ptr [rsi], xmm1   ;Speichere xmm1 in y[0]

mulss xmm0, dword ptr [rdi+4];Mul direkt in xmm0 (a)
addss xmm0, dword ptr [rsi+4];Add direct in xmm0
movss dword ptr[rsi+4], xmm0 ;Speichere xmm0 in y[1]
```

[1] godbolt.org/, gcc x86-64, -O3

CPI (2)

```
movss xmm1, dword ptr [rdi]   ;Lade x[0] aus Speicher
mulss xmm1, xmm0              ;Mul mit a aus Register xmm0
addss xmm1, dword ptr [rsi]   ;Add y[0] zu Registerinhalt
movss dword ptr [rsi], xmm1   ;Speichere xmm1 in y[0]

mulss xmm0, dword ptr [rdi+4];Mul direkt in xmm0 (a)
addss xmm0, dword ptr [rsi+4];Add direct in xmm0
movss dword ptr[rsi+4], xmm0 ;Speichere xmm0 in y[1]
```

Nehmen wir an, dass Speicherzugriffslatenz $:= 100$ Taktzyklen, und dass Latenz für arithmetische Operationen $:= 5$ Taktzyklen. Dann ist CPI für diese Subroutine $:= (6 \times 100 + 4 \times 5) * 1/7$. [2]

[2] Tatsächliche Latenzen weichen ab.

CPI (3)

Die Realität sieht aber meist dynamischer aus:

```
void saxpy(float a, float* x, float* y, int N) {
    for (int i = 0; i < N; ++i)
            y[i] = a * x[i] + y[i];
}
```

Der Compiler kann die Schleife nicht mehr einfach “unrollen”. CPI ist nun eingabeabhängig. (Übung: analysieren Sie die Subroutine *saxpy* () mithilfe des Compiler Explorer Tools unter `https://godbolt.org`. Verwenden Sie verschiedene Compiler und verschiedene Optimierungslevel.)

CPI (4)

Im Beispiel haben wir auch angenommen, dass Speicherzugriffe eine Latenz von 100 Taktzyklen mit sich bringen. In der Realität werden Speicherbewegungen jedoch durch die Cache Hierarchie ausgeführt, die in der Praxis zu erwartende Latenz weicht sicher stark ab.

CPI (5)

CPI wird daher in der Realität mittels eines Profilers und "Sampling" bestimmt. Dies ist eine in den Prozessor eingebaute Funktionalität, die, falls aktiviert, die *Instruktionspipeline* zu definierten Zeitpunkten anhält und analysiert, welche Instruktionen sich in der Pipeline befinden. Entsprechend oft ausgeführt erhält man so Aufschluss darüber, welche Instruktionen im Durchschnitt wie häufig ausgeführt werden. Der Profiler kann somit CPI statistisch bestimmen.

Performanz

Wir kennen also nun alle Faktoren, um die CPU execution time als Maß für die Performanz zu berechnen:

$$CPU\ time = \frac{Anzahl\ Taktzyklen(I, D)}{Taktfrequenz},$$

wobei

$$Anzahl\ Taktzyklen(I, D) = \#I \times CPI.$$

Jetzt wollen wir uns noch anschauen, was Instruktionen eigentlich sind.

Instruktionen

- *Assembler Befehle* (wie z. B. x86 aus dem Beispiel zuvor) entsprechen i. d. R. recht eindeutig entsprechenden Maschineninstruktionen. Es gibt Ausnahmen (sog. *Pseudoinstruktionen*), dann wird ein Muster von Instruktionen unter einem Assemblerbefehl zusammengefasst.
- Maschineninstruktionen sind i. d. R. *Bitmuster*, häufig entspricht die Anzahl der Bits der Wortbreite des Prozessors.
- *OP-Code* beschreibt die Funktion, die die Instruktion erfüllt (z. B. ADD,MUL,MOV,etc.). Bei 32-bit Befehlssatz können z. B. 5 Bit für OP-Code reserviert sein.
- Weitere Bits addressieren z. B. Register (bei 32-bit Befehlssatz z. B. 5 Bit, sodass 32 Register adressierbar sind) oder erlauben Speicherzugriffe via Registerindirektion.

Beispiel: MIPS Instruktion

Einfaches Additions Statement in C:

```
int a = b + c;
```

Compiler generiert daraus den MIPS Assembly Befehl:

```
add $t0,$s1,$s2
```

Lese: “Addiere die Inhalte der Register `$s1` und `$s2`, schreibe das Ergebnis in Register `$t0`.”

Beispiel: MIPS Instruktion

```
add $t0,$s1,$s2
```

Die zugehörige Maschineninstruktion lautet:

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
000000 (0_{10})	10001 (17_{10})	10010 (18_{10})	01000 (8_{10})	00000 (0_{10})	100000 (32_{10})

Tabelle: Vgl. Patterson/Henessy Computer Organization and Design, 5th edition, 2014, p. 80

Beispiel: MIPS Instruktion

```
add $t0,$s1,$s2
```

Die zugehörige Maschineninstruktion lautet:

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
000000 (0_{10})	10001 (17_{10})	10010 (18_{10})	01000 (8_{10})	00000 (0_{10})	100000 (32_{10})

- Feld 1 & Feld 6: führe Addition durch.
- Feld 2: erster "Source Operand" kommt aus Register `$s1`.
- Feld 3: zweiter "Source Operand" kommt aus Register `$s2`.
- Feld 4: Ergebnisregister ist `$t0`.
- Feld 5: wird für diese Instruktion nicht verwendet.

Beispiel: MIPS Instruktion

Generell (MIPS):

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Tabelle: Vgl. Patterson/Henessy Computer Organization and Design, 5th edition, 2014, p. 82

- Feld 1: Basisoperation ("opcode").
- Feld 2: erster "Source Operand".
- Feld 3: zweiter "Source Operand".
- Feld 3: "Destination Operand".
- Feld 5: Shift Amount: Spezialcode für Shift Operationen.
- Feld 6: spezifiert opcode Variante (z. B. "immediate").

Beispiel: MIPS Instruktion

Generell (MIPS):

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Grundprinzip: alle zur Durchführung der Operation notwendigen Informationen werden in `Wortbreite` bits (hier: 32) untergebracht.
- Compiler Backend übersetzt in konkreten Maschinenbefehlssatz.
- CPU dekodiert opcode, erkennt Binäroperation, lädt Registerinhalte, führt Addition durch, speichert Ergebnis.
- Feldgröße $\Rightarrow$ Anzahl Register

Performanz, Recap

Kurz zusammenfassend:

- Bestimme wall clock time als das Produkt $\#I \times CPI \times \mathit{Zeit\ pro\ Taktzyklus}$.
- $\#I$ und CPI werden i. d. R. statistisch mit Profiler bestimmt.
- Einzelne Instruktionen leiten sich aus Assembly Befehlen ab und kodieren Operation und Operanden in kurzer Binärdarstellung.

Maßnahmen zur Erhöhung der Performanz

Es gibt also folgende Mittel, um die Performanz eines Computerprogramms zu verbessern, das auf einem Prozessor ausgeführt wird:

- Verwende Instruktionen, die *im Durchschnitt* schneller sind. Braucht eine Speicherzugriffsinstruktion z. B. 100 Taktzyklen, um den Inhalt einer Speicherzelle in ein Register zu laden, und eine arithmetische Instruktion benötigt nur zwei Taktzyklen, kann es sinnvoll sein, Berechnungen öfter zu wiederholen, als Ergebnisse zwischenzuspeichern.
- Verwende Algorithmen und Hochsprachen Statements, die die Anzahl an Instruktionen reduzieren.
- Erhöhe die Taktfrequenz des Prozessors.

Performanz Einflussfaktoren (1)

Komponente	**Hat Einfluss auf**	**Wie**
Algorithmus	Instruktionszahl, CPI	Algorithmus bestimmt Art der Instruktionen. Manche Algorithmen favorisieren außerdem bestimmte Arten von Instruktionen und beeinflussen dadurch CPI.
Programmiersprache	Instruktionszahl, CPI	Direkte Übersetzung von *Statements* in Instruktionen beeinflusst Instruktionszahl. Programmiersprachen mit hohem Abstraktionsniveau benötigen oft Instruktionen mit höherem CPI.

Tabelle: Vgl. Patterson/Henessy Computer Organization and Design, 5th edition, 2014, p. 39

Performanz Einflussfaktoren (2)

Komponente	Hat Einfluss auf	Wie
Compiler	Instruktionszahl, CPI	Compiler hat Schlüsselrolle, balanziert das Verhältnis zwischen opt. Instruktionen für Prozessor und CPI, Kompilierzeiten, Größe des Binary Outputs, etc.
Befehlssatz (ISA)	Instruktionszahl, CPI, Taktfrequenz	Einziger Faktor, der auch Taktfrequenz beeinflusst, etwa RISC vs. CISC Befehlssatz.

Tabelle: Vgl. Patterson/Henessy Computer Organization and Design, 5th edition, 2014, p. 39

Befehlssatz

Befehlssatz

Der Befehlssatz eines Prozessors und die dem Befehlssatz zu Grunde liegende Befehlssatzarchitektur (engl.: “instruction set architecture (ISA)”) beeinflussen CPI und Taktfrequenz. Architekturen mit einfachen, generellen Instruktionen (“reduced instruction sets”) erlauben niedrigere CPI, komplexere Befehlssätze mit special purpose Instruktionen (“complex instruction sets”) werden zumeist durch höhere CPI realisiert.

Leistungsaufnahme (1)

- Mooresches Gesetz wurde lange Zeit (landläufig) missinterpretiert als: alle 18 Monate verdoppelt sich die Performanz.
  - Rührt daher, dass über ca. 30 Jahre entsprechend Taktfrequenz erhöht wurde.
  - Möglich, weil *Leistungsaufnahme* nicht mit der gleichen Rate wächst, wie die Anzahl Transistoren.
- Leistungsaufnahme hängt von Eingangsspannung ab. Diese ist proportional zur Transistorgröße: kleinerer Transistor $\Rightarrow$ niedrigere Eingangsspannung.

Leistungsaufnahme (2)

Energieverbrauch von ICs hängt von *dynamischer Energie* ab. Diese ist proportional zur kapazitiven Ladung C sowie zur Eingangsspannung V_{dd}. Energieverbrauch für Zustandsübergang $0 \rightarrow 1 \rightarrow 0$:

$$E = C \times V_{dd}^2. \tag{4}$$

$\Rightarrow$ Energieverbrauch für einfachen Schaltvorgang

$$E = \frac{1}{2} \times C \times V_{dd}^2. \tag{5}$$

Auf die Zeit bezogen ergibt sich die dynamische Leistungsaufnahme pro Transistor

$$P \simeq E \times Taktfrequenz \Leftrightarrow \frac{1}{2} \times C \times V_{dd}^2 \times Taktfrequenz. \tag{6}$$

Leistungsaufnahme (3)

$$P \simeq \frac{1}{2} \times C \times V_{dd}^2 \times Taktfrequenz.$$

- Dennards Skalierungsgesetz: V_{dd} ist proportional zur Transistorgröße. Die *Spannungsdichte* bleibt gleich.
- D. h. wenn Transistoren kleiner werden (Mooresches Gesetz), wird auch die Leistungsaufnahme des einzelnen Transistors geringer.
- Da V_{dd} quadratisch eingeht, schrumpft die Eingangsspannung sogar annähernd proportional zur Geschwindigkeit, mit der Transistoren "vermehrt" werden.
  - Intel von etwa Mitte der 1990'er Jahre bis Mitte der 2000'er Jahre: Leistungsaufnahme 30× Wachstum, gleichzeitig Taktfrequenz 1000×.

Historische Leistungsaufnahme von Intelprozessoren

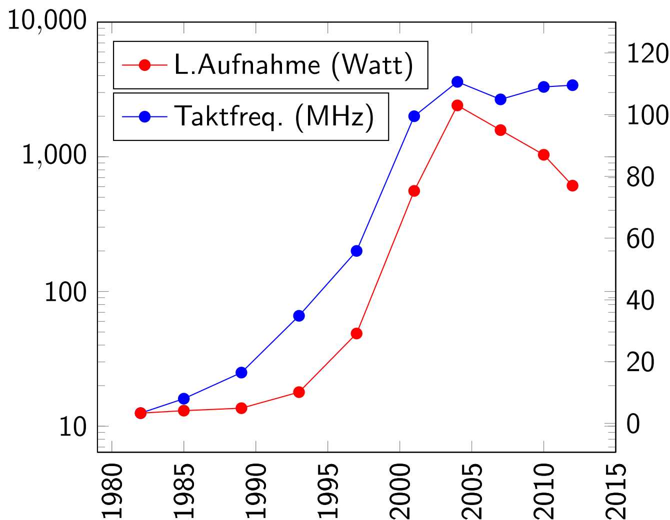

Abbildung: Quelle: Patterson and Hennessy, Computer Organization and Design (2014).

Powerwall

- Transistor, der mit Stromzufuhr verbunden ist, unterliegt "Leakage". Ähnlich wie tropfender Wasserhahn: es wird etwas Energie verbraucht, egal ob Spannung anliegt (Wasserhahn offen) oder nicht (Wasserhahn zu).
- Problem: Leakage Spannung V_l unterliegt nicht Dennards Skalierungsgesetz, daher wird Leakage immer problematischer, je mehr Transistoren auf dem Chip.
- Mooresches Gesetz $\Rightarrow$ mehr Transistoren, aber wir können sie nicht nutzen.

Powerwall

- "Red Brick Wall": über lange Jahre war absehbar, dass deshalb Erhöhung der Taktfrequenz nicht auf Dauer möglich ist.
- Intels Pentium 4 war 2004 die letzte Prozessorgeneration, bei der die Taktfrequenz noch bedenkenlos erhöht werden konnte.
- Herb Sutter von Microsoft (2005): "The Free Lunch Is Over".
- Keine *triviale Performanzsteigerung* durch regelmäßige Anhebung der Taktfrequenz mehr möglich.

Powerwall

Kein “Free Lunch” mehr. Aber das Mooresche Gesetz gilt immer noch (noch). Was also tun mit der immer weiter wachsenden Anzahl an Transistoren?

Pragmatische Lösung der Prozessorhersteller: verbaue Chip Elemente, die in inaktivem Zustand keinen Strom verbrauchen:

- Prozessoren mit mehreren Kernen, die nur Spannung bekommen, wenn gerechnet wird.
- Größere Caches.
- Special purpose Register, die nicht immer genutzt werden.

Powerwall

Software Sicht (Herb Sutter): “The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency.” “The free lunch is over.”

Problem: serielle Programme werden nicht mehr einfach mit jeder Prozessorgeneration schneller, sondern nur, wenn sie gemäß der veränderten Bedingungen skalieren.

Bemerkung

Vgl. Tabelle “Performanz Einflussfaktoren”: Entscheidungen zur Verbesserung der Performanz verschieben sich vom Hardware Hersteller zum Anwendungsentwickler!

- Parallelisierung für Multi-Core.
- SIMD freundliches Datenlayout.
- Cache-kohährente Speicherzugriffe.