PPG18

Optimise cache accesses to improve performance

Jannek Squar
squar@informatik.uni-hamburg.de

Scientific Computing
Department of Informatics
Universität Hamburg
https://wr.informatik.uni-hamburg.de

2018-04-26

Hierarchie der Storage Tiers

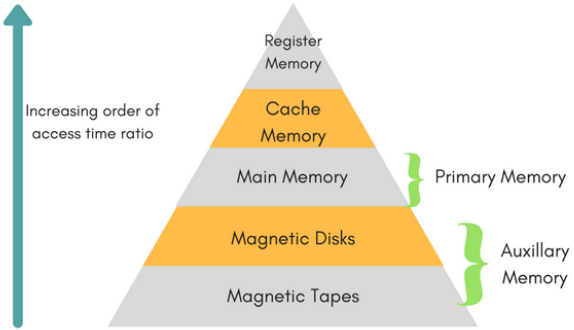


Figure: Storage Tiers Schema [7]

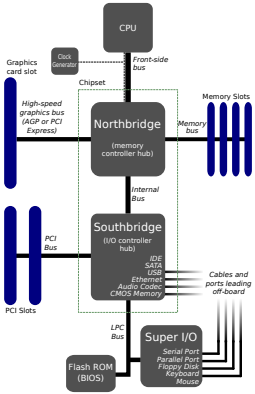
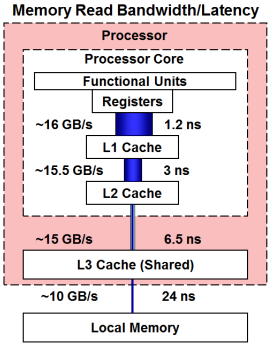


Figure: Motherboard Schema [10]

Laden eines Datums in CPU-Recheneinheit



1. Datum in Register?
2. Datum in L1?
3. Datum in L2?
4. Datum in L3?
5. Northbridge-Speichercontroller
   $\rightarrow$ Datum in Memory?
6. Southbridge-Speichercontroller
   $\rightarrow$ Datum in Speichermedium?

Figure: Sandy Bridge Latenz und Bandbreite [6, 11]

Architektur

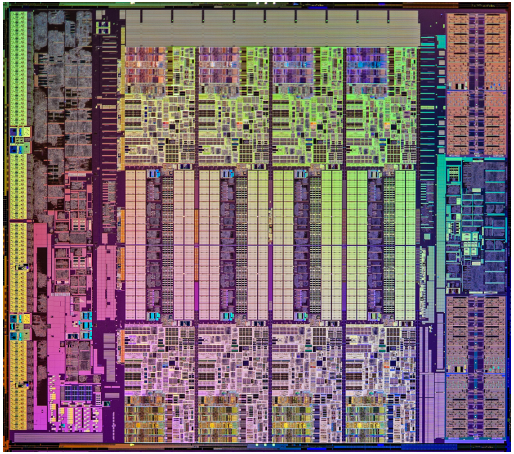


Figure: Haswell die [1]

Problemstellung

- Laden von Daten zeitintensiv (hohe Latenz)
  - Cache liegt auf CPU Die
    → schneller Datenzugriff
  - Cache-Speicher teuer
    → Aufbau einer Speicher-Hierarchie
- Ausnutzen von Lokalitätseigenschaften
  - Zeitliche Lokalität:
    → Verdrängung unwichtigerer Daten
  - Räumliche Lokalität:
    → Read-ahead auf Cache-Line ($\sim$ 64 Byte)

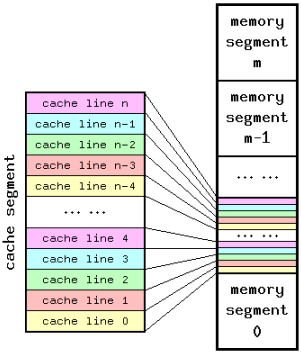


Figure: Unterteilung in Cache-Lines [3]

Hitting the cache

Bewertung der Cache-Performance mittels Hit-Ratio:

$$Hit\ Ratio = \frac{Hit}{Hit + Miss}$$

Durchschnittliche Speicherzugriffszeit:

$$t_{avg} = p \times t_c + (1 - p) \times t_m$$

p Wahrscheinlichkeit für Cache-Hit ($\approx$ Hit-Ratio)

t_c Zugriffszeit Cache

t_m Zugriffszeit Memory

mit $t_c \ll t_m$

Auswirkung auf Laufzeit

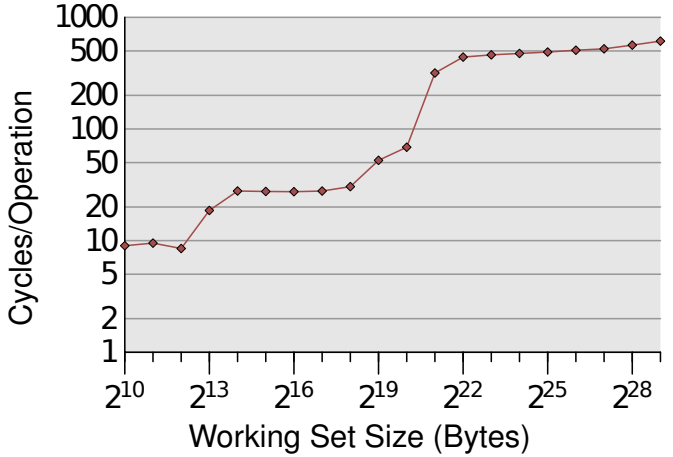


Figure: Laufzeit unterschiedlicher Storage Tiers (L1, L2, main memory) [9]

Optimierungsansatz

- Ausnutzen der Datenlokalität
  → weniger Datenbewegung in langsamen Storage Tiers
- Kein direkter Einfluss auf Cache-Funktionalität
  → indirekte Optimierungen
- Evaluation einer Optimierung:
  - Cache-Kapazität: `lscpu`
  - Cache-Miss-Zähler: `perf stats -e cache-misses ./APPLICATION`
  - Zeitmessung: `time ./APPLICATION`

Column Order

Values as stored in Memory:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Written down as

Column major: $\begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$ Row major: $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$

Figure: Datenzugriffsmuster [5]

Column Order - Demo

Demo

Loop Blocking

Unterteilung großer Matrizen in Chunks, die komplett in den Cache passen

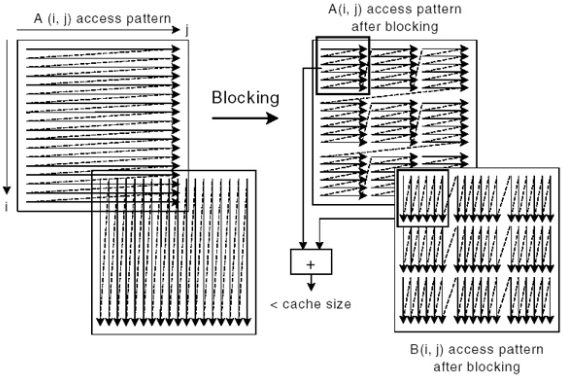


Figure: Unterteilung der Matrix in Chunks [4]

Loop Blocking - Code example

```
1 float A[MAX, MAX], B[MAX, MAX];
2 for (i=0; i< MAX; i++)
3   for (j=0; j< MAX; j++)
4     A[i,j] = A[i,j] + B[j, i];
```

Listing 1: Originales Zugriffsmuster [4]

```
1 float A[MAX, MAX], B[MAX, MAX];
2 for (i=0; i< MAX; i+=block_size)
3   for (j=0; j< MAX; j+=block_size)
4     for (ii=i; ii<i+block_size; ii++)
5       for (jj=j; jj<j+block_size; jj++)
6         A[ii,jj] = A[ii,jj] + B[jj, ii];
```

Listing 2: Verbessertes Zugriffsmuster [4]

Datentypen

- Möglichst kleine Datentypen
  → Mehr Daten pro Cache-Line
- Padding:
  - "Leere" Speicherblöcke (Padding) für Alignment
    → Weniger Lesevorgänge auf Segmenten
  - Erhöhter Speicherplatzbedarf
  - Neuordnung der Daten reduziert Anzahl an Paddings
    → Mehr Daten pro Cache-Line

Figure: Datenanordnung mit/ohne Padding [12]

Datentypen - ungeordneter Struct

```
1  struct s1
2  {
3      char a;
4      short a1;
5      char b1;
6      float b;
7      int c;
8      char e;
9      double f;
10 };
11 // Size of Struct s1 = 32
```

Listing 3: Ungeordneter Struct, 11 Padding-Bytes [2]

Figure: Ungeordneter Struct, 11 Padding-Bytes [2]

Datentypen - geordneter Struct

```
1  struct s2
2  {
3      double f;
4      float b;
5      int c;
6      short a1;
7      char a,b1,e;
8  };
9  // Size of Struct s2 = 24
```

Listing 4: Geordneter Struct, 3 Padding-Bytes [2]

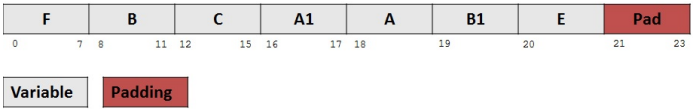


Figure: Geordneter Struct, 3 Padding-Bytes [2]

Optimierung - SoA vs. AoS

Besonders interessant für Vektorisierung, erlaubt aber auch Ausnutzung von räumlicher Lokalität

```
1 struct {
2   uint r, g, b, w;
3 } MyAoS[N];
```

Listing 5: Array of structs [8]

```
1 struct {
2   uint r[N];
3   uint g[N];
4   uint b[N];
5   uint w[N];
6 } MySoA;
```

Listing 6: Struct of arrays [8]

Ergänzung: False sharing

- Schreiben auf Cache-Line *kann* Neuladen erzwingen
- Nebenläufiges Lesen
  → unnötiger Cache-Miss
- Finden und beheben mittels Tools und Compiler-Instruktionen

```
1  struct foo {
2      int x;
3      int y;
4  };
5  static struct foo f;
6  /* sum_a und inc_b laufen nebenlaeufig */
7  int sum_a(void)
8  {
9      int s = 0;
10     int i;
11     for (i = 0; i < 1000000; ++i)
12         s += f.x;
13     return s;
14 }
15 void inc_b(void)
16 {
17     int i;
18     for (i = 0; i < 1000000; ++i)
19         ++f.y;
20 }
```

Listing 7: Beispiel für false sharing

References I

[1] http://www.extremetech.com/wp-content/uploads/2014/08/haswell-e-die-shot-high-res.jpg.

[2] Coding for performance: Data alignment and structures. https://software.intel.com/en-us/articles/coding-for-performance-data-alignment-and-structures.

[3] Functional principles of cache memory. http://alasir.com/articles/cache_principles/cache_line_tag_index.html.

[4] How to use loop blocking to optimize memory use on 32-bit intel architecture. https://software.intel.com/en-us/articles/how-to-use-loop-blocking-to-optimize-memory-use-on-32-bit-intel-architecture.

[5] Left handed, right handed, up vector, column vector, row vector, column major, row major, what? http://mabulous.com/left-handed-right-handed-up-vector-column-vector-row-vector-column-major-row-major-wh

References II

[6] Memory access times. https://cvw.cac.cornell.edu/codeopt/memtimes.

[7] Memory organization. https://www.studytonight.com/computer-architecture/memory-organization.

[8] Putting your data and code in order: Data and layout - part 2. https://software.intel.com/en-us/articles/putting-your-data-and-code-in-order-data-and-layout-part-2.

[9] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11:2007, 2007.

[10] Moxfyre. Motherboard diagram. https://commons.wikimedia.org/wiki/File:Motherboard_diagram.svg.

[11] Subhash Saini, Johnny Chang, and Haoqiang Jin. Performance evaluation of the intel sandy bridge based nasa pleiades using scientific and engineering applications. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 25–51. Springer, 2013.

References III

[12] Tim Wischof. Memory management and optimizations. techreport, Universität Hamburg, 2017.