The memory hierarchy: Cache friendly code–loop interchange, cache blocking, cache oblivious algorithms

Yipeng Huang

Rutgers University

April 14, 2022

Table of contents

Announcements

Class session plan

- 4/19, 4/21, 4/26: Diving deeper: Digital logic. (CS:APP Chapter 4.2) (Recommended reading: Patterson & Hennessy, Computer organization and design, appendix on "The Basics of Logic Design." Available online via Rutgers Libraries)
- 4/28: Survey of advanced topics in computer architecture.

Recap of Tuesday

For a given total capacity of cache (capacity = $S \times E \times B$),

- How to improve support for spatial locality? B
- How to improve support for temporal locality? E

Table of contents

Direct-mapped cache

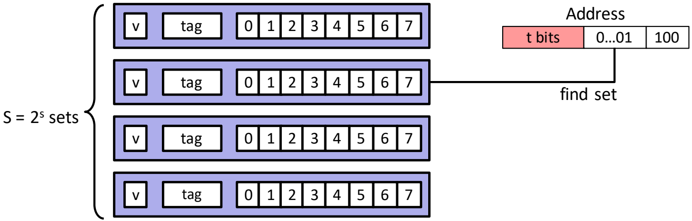

Figure: Direct-mapped cache. Image credit CS:APP

No need for replacement policy

- The number of sets in cache is $S = 2^s = 2^2 = 4$.
- A hash function that limits exactly where a block can go.
- In direct-mapped cache, blocks can go into only one of $E = 1$ way.
- No cache replacement policy is needed.

Associative caches

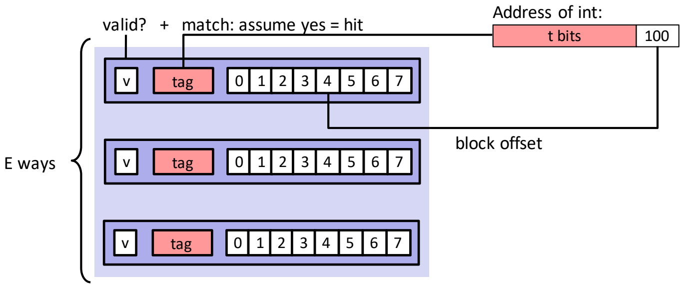

Figure: Fully associative cache. Image credit CS:APP

Needs replacement policy

- Blocks can go into any of E ways
- Here, $E = 3$
- Good for capturing temporal locality.
- If all ways/lines/blocks are occupied, and a cache miss happens, which way/line/block will be the victim and get evicted for replacement?

Cache replacement policies for associative caches

FIFO: First-in, first-out

- Evict the cache line that was placed the longest ago.
- Each cache set essentially becomes limited-capcity queue.

LRU: Least Recently Used

- Evict the cache line that was last accessed longest ago.
- Needs a counter on each cache line, and/or a global counter (e.g., program counter).

Table of contents

Cache-friendly code

Algorithms can be written so that they work well with caches

- Maximize hit rate
- Minimize miss rate
- Minimize eviction counts

Do so by:

- Increasing spatial locality.
- Increasing temporal locality.

Advanced optimizing compilers can automatically make such optimizations

- GCC optimizations
- `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`
- `-floop-interchange`
- `-floop-block`

Loop interchange

Refer to textbook slides on "Rearranging loops to improve spatial locality"

- Loop interchange increases spatial locality.
- In PA5, fourth part "cacheBlocking" you can explore the impact of this on matrix multiplication.
- In practice, programmers do not have to worry about this optimization.
- Optimized automatically in GCC by compiler flag `-floop-interchange` and `-O3`.

Cache blocking

Refer to textbook slides on "Using blocking to improve temporal locality"

- Cache blocking increases temporal locality.
- In PA5, fourth part "cacheBlocking" you can explore the impact of this on matrix multiplication.
- In practice, programmers do not have to worry about this optimization.
- Optimized automatically in GCC by compiler flag `-floop-block`. But it is not part of default optimizations such as `-O3` so you have to remember to set it.

Table of contents

Multilevel cache hierarchies

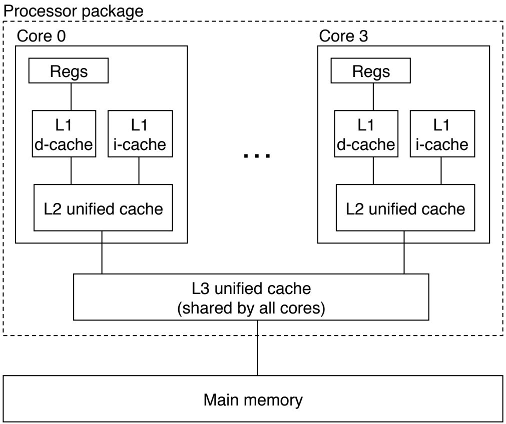

Figure: Intel Core i7 cache hierarchy. Image credit CS:APP

Small fast caches nested inside large slow caches

- L1 data and instruction cache: 32KB, 64 set, 8-way associative, 64B block, 4 cycle latency.
- L2 cache: 256KB, 512 set, 8-way associative, 64B block, 10 cycle latency.
- L3 cache: 8MB, 8192 set, 16-way associative, 64B block, 40-75 cycle latency.

Notice how latency cost increases as E-way associativity increases.

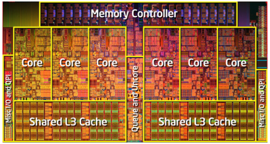

Figure: Intel 2020 Gulftown die shot. Image credit AnandTech

Cache oblivious algorithms

The challenge in writing code / compiling programs to take advantage of caches:

- Programmers do not easily have information about target machine.
- Compiling binaries for every envisioned target machine is costly.

Matrix transpose baseline algorithm: iteration

$$\mathbf{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

$$\mathbf{B} = \mathbf{A}^\intercal = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} \\ a_{0,1} & a_{1,1} & a_{2,1} & a_{3,1} \\ a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} \end{bmatrix}$$

```
for ( size_t i=0; i<n; i++ ) {
  for ( size_t j=0; j<n; j++ ) {
    B[ j*n + i ] = A[ i*n + j ];
  }
}
```

Matrix transpose via recursion

$$\mathbf{A} = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} = \left[\begin{array}{cc|cc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{array}\right]$$

$$\mathbf{B} = \mathbf{A}^\mathsf{T} = \begin{bmatrix} A^\mathsf{T}_{0,0} & A^\mathsf{T}_{1,0} \\ A^\mathsf{T}_{0,1} & A^\mathsf{T}_{1,1} \end{bmatrix} = \left[\begin{array}{cc|cc} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} \\ a_{0,1} & a_{1,1} & a_{2,1} & a_{3,1} \\ \hline a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} \end{array}\right]$$

Strategy:

- Divide and conquer large matrix to transpose into smaller transpositions.
- After some recursion, problem will fit well inside cache capacity.
- Once enough locality exists withing subroutine, switch to plain iterative approach.

Advantages:

- No need to know about cache capacity and parameters beforehand.
- Works well with deep multilevel cache hierarchies: different amounts of locality for each cache level.

Table of contents

Memory hierarchy implications for software-hardware abstraction

It is not entirely true the architecture can hide details of microarchitecture

Even less true going forward. What to do?

Application level recommendations

- Use industrial strength, optimized libraries compiled for target machine.
- Lapack, Linpack, Matlab, Python SciPy, NumPy...
- `https://people.inf.ethz.ch/markusp/teaching/263-2300-ETH-spring11/slides/class08.pdf`

Algorithm level recommendations

Deploy cache-oblivious algorithm implementations.

Compiler level recommendations

- Enable compiler optimizations—*e.g.*, `-O3`, `-floop-interchange`, `-floop-block`.
- `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`