Caches: PA5 part 2, cache friendly code, digital logic

Yipeng Huang

Rutgers University

April 15, 2021

Table of contents

Looking ahead

Class plan

1. PA5 now out. Due Monday, 4/26.
2. Short quiz 8 now out. Due Monday, 4/19.
3. Digital logic. Reading assignment: 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.

Table of contents

PA5: Optimizing programs for caches

Optimize some code for better cache performance

1. cacheBlocking
2. cacheOblivious

PA5: Optimizing programs for caches

A tour of files in the package

- Baseline implementations: matMul, matTrans.
- Your optimized implementations: cacheBlocking, cacheOblivious.
- What the autograder.py does:
  1. Testing for correctness.
  2. Getting the memory trace.
  3. Comparing your performance against the baseline.

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.

A few specific techniques:

- Loop interchange.
- Cache blocking.
- Cache-oblivious algorithm implementation.

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.

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

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

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

32x32 transpose -> 4 separate 16x16 transpose tasks

16x16 -> 4 separate 8x8

8x8 -> 4 separate 4x4

4x4 -> 4 separate 2x2

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`