Cache Optimizations

2024 Fall ECE454: Computer Systems Programming — Lecture 9

Jon Eyolfson — 1.0.0

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

Write Code That Has Locaility

Temporal locality:
  Recently referenced items are likely to be referenced soon after

Spatial locality:
  Items with nearby addresses tend to be referenced close together

How to achieve locality?
  Proper choice of algorithm
  Loop transformations

Row-Major Order: Elements of a Row are Beside Each Other

We *could* represent a 2D array using a plain 1D array

Assuming we have an `int` array, `table`, we can index elements:

```
table[rowIndex * NUM_COLS + colIndex]
```

The following is equivalent:

```
table[0][0] → table[0]
table[0][1] → table[1]
table[0][2] → table[2]
table[1][0] → table[3]
table[1][1] → table[4]
table[1][2] → table[5]
```

`&table[i][j]` is also the same as `table + i * NUM_COLS + j`

Let’s Assume a Simple Cache

The cache block size (line size) is 8 bytes
Therefore, it could hold 2 `int`s

It’s 2-way set associative (2 blocks per set),
with a single set (in this case it’s the same as fully associative)

The total size is two blocks, and we’ll assume LRU
(least recently used) replacement policy

Questions We Should Ask Ourselves

How many elements are there per block?

Does the data structure fit in the cache?

Do I reuse blocks over time?

In what order am I accessing blocks?

Let’s Start with a Simple Array

```
#define N 4
int A[N];
for (int i = 0; i < N; ++i) {
  /* ... = */ A[i];
}
```

What’s the miss rate in this scenario? (assuming the array is cache aligned)

Miss rate = # misses / # accesses

Let’s Start with a Simple Array

```
#define N 4
int A[N];
for (int i = 0; i < N; ++i) {
  /* ... = */ A[i];
}
```

What’s the miss rate in this scenario? (assuming the array is cache aligned)
  Miss rate = # misses / # accesses

In this case: `A[0]` is a miss, which brings in `A[1]`,
  and `A[2]` is a miss, which brings in `A[3]`, miss rate = 50%

In general for this scenario the miss rate is $\frac{N/2}{N} = \frac{1}{2} = 50\%$

Let's Add Another Loop

```
#define N 4
int A[N];
for (int k = 0; k < P; ++k) {
  for (int i = 0; i < N; ++i) {
    /* ... = */ A[i];
  }
}
```

What's our miss rate now?

Let's Add Another Loop

```
#define N 4
int A[N];
for (int k = 0; k < P; ++k) {
  for (int i = 0; i < N; ++i) {
    /* ... = */ A[i];
  }
}
```

What's our miss rate now?

Since the array fits in cache, the miss rate is:

$$\frac{N/2}{N \times P} = \frac{1}{2 \times P}$$

For sequential accesses with re-use, if data fits in the cache, the first visit suffers all the misses

What If the Data Does Not Fit in Cache?

```
#define N 8
int A[N];
for (int k = 0; k < P; ++k) {
  for (int i = 0; i < N; ++i) {
    /* ... = */ A[i];
  }
}
```

What’s our miss rate now?

It’s the same as the code that only uses each element once:

$$\frac{N/2}{N} = \frac{1}{2} = 50\%$$

Now, What About a 2D Array?

```
#define N 2
int A[N];
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j) {
    /* ... = */ A[i][j];
  }
}
```

What's the miss rate?

Now, What About a 2D Array?

```
#define N 2
int A[N];
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j) {
    /* ... = */ A[i][j];
  }
}
```

What’s the miss rate?

It’s exactly the same as accessing a 1D array, 50%

It doesn’t matter if the entire array fits in cache or not

Let's Access the Array in Column Order

```
#define N 2
int A[N];
for (int j = 0; j < N; ++j) {
  for (int i = 0; i < N; ++i) {
    /* ... = */ A[i][j];
  }
}
```

What's the miss rate now?

In this case: `A[0]` is a miss, which brings in `A[1]`,
the next access to `A[2]` is a miss, which brings in `A[3]`,
after we access `A[1]` and `A[3]` which are hits

Miss rate is 50% if the 2D array fits in cache (same as sequential)

What Happens if the 2D Array Does Not Fit in Cache?

```
#define N 4
int A[N];
for (int j = 0; j < N; ++j) {
  for (int i = 0; i < N; ++i) {
    /* ... = */ A[i][j];
  }
}
```

What's the miss rate now? Converting to a 1D array we access:
`A[0]`, `A[4]`, `A[8]`, `A[12]`, `A[1]`, etc.

What Happens if the 2D Array Does Not Fit in Cache?

```
#define N 4
int A[N];
for (int j = 0; j < N; ++j) {
  for (int i = 0; i < N; ++i) {
    /* ... = */ A[i][j];
  }
}
```

What's the miss rate now? Converting to a 1D array we access:
`A[0]`, `A[4]`, `A[8]`, `A[12]`, `A[1]`, etc.

The miss rate is now 100%!

This is Why Loop Interchange is So Important

Changing

```
#define N 4
int A[N];
for (int j = 0; j < N; ++j) {
  for (int i = 0; i < N; ++i) {
    /* ... = */ A[i][j];
  }
}
```

to

```
#define N 4
int A[N];
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j) {
    /* ... = */ A[i][j];
  }
}
```

changes our miss rate of 100% to 50%

What About Matrix Multiplication?

```
for (int i = 0; i<N; i++) {
  for (int j = 0; j<N; j++) {
    for (int k = 0; k<N; k++) {
      /* ... = */ A[i][k] * B[k][j];
    }
  }
}
```

We can argue about this when `i = 1`

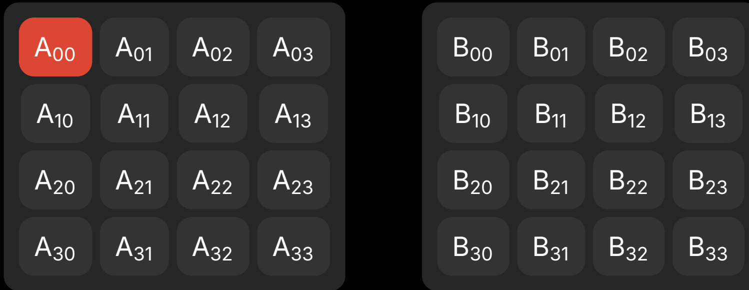
Miss Rate When `i = 1`

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

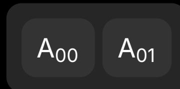
B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Cache:

Miss Rate When `i = 1`



Cache:



Miss Rate When `i = 1`

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Cache:

A_{00}	A_{01}

B_{00}	B_{01}

Miss Rate When `i = 1`

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Cache:

A_{00}	A_{01}

B_{00}	B_{01}

Miss Rate When `i = 1`

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Cache:

A_{00}	A_{01}
B_{10}	B_{11}

Miss Rate When `i = 1`

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Cache:

A_{02}	A_{03}
B_{10}	B_{11}

Miss Rate When `i = 1`

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Cache:

A_{02}	A_{03}

B_{20}	B_{21}

Miss Rate When `i = 1`

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Cache:

A_{02}	A_{03}

B_{20}	B_{21}

Miss Rate When `i = 1`

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Cache:

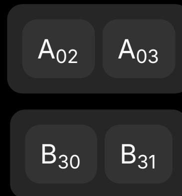
A_{02}	A_{03}
B_{30}	B_{31}

Miss Rate When `i = 1`

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Cache:



Miss rate: 75%

Generalizing the Previous Example for arrays A and B

Assuming a row does not fit in cache, arrays are cache aligned, and we can fit D elements in the cache

We'll have $\frac{1}{D} \times N + N$ misses out of $N + N$ accesses, therefore our miss rate is: $\frac{1}{2 \times D} + \frac{1}{2}$

Improving Cache Reuse

- Misses are expensive
  - L1 cache reference: 1-4 ns (L1 cache size: 32 KB)
  - Main memory reference: 100 ns (memory size: 4-256 GBs)
- Matrix multiplication has lots of data re-use
  - Key idea: Try to use entire cache block once it is loaded
    - Challenge: We need to work with both rows and columns

Improving Cache Reuse

- Misses are expensive
  - L1 cache reference: 1-4 ns (L1 cache size: 32 KB)
  - Main memory reference: 100 ns (memory size: 4-256 GBs)
- Matrix multiplication has lots of data re-use
  - Key idea: Try to use entire cache block once it is loaded
    - Challenge: We need to work with both rows and columns
- Solution:
  - Operate in sub-squares of the matrices
    - One sub-square per matrix should fit in cache simultaneously
    - Heavily re-use the sub-squares before loading new ones
  - Called Tiling or Blocking (a sub-square is a tile)

Implementation for Tiled Matrix Multiplication

Where T is the number of elements we can fit on a cache line

```
for (int i = 0; i < MATRIX_N; i += T)
  for (int j = 0; j < MATRIX_N; j += T)
    for (int k = 0; k < MATRIX_N; k += T)
      for (int i1 = i; i1 < i+T; i1++)
        for (int j1 = j; j1 < j+T; j1++)
          for (int k1 = k; k1 < k+T; k1++)
            c[i1][j1] += a[i1][k1]*b[k1][j1];
```

Miss Analysis for Tiled Matrix Multiply

Assuming we can fit each title in cache (cache size is more than $3 \times T^3$), our miss rate for matrices A_T and B_T are:

$\frac{2}{T} \times N$ misses out of $2N$ accesses, giving us a miss rate of $\frac{1}{T}$

A massive difference from before!