

CS 610: Writing Cache-Friendly Code

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2026-27-I



Is this function cache friendly?

```
int sumvec(int v[N]) {  
    int sum = 0;  
    for (int i = 0; i < N; i++) {  
        sum += v[i];  
    }  
    return sum;  
}
```

Suppose v is block-aligned, words are 4 bytes, cache blocks are 4 words, and the cache is initially empty.

What can you say about the locality of variables i , sum , and elements of v ?

Is this function cache friendly?

```
int sumvec(int v[N]) {  
    int sum = 0;  
    for (int i = 0; i < N; i++) {  
        sum += v[i];  
    }  
    return sum;  
}
```

Iteration	0	1	2	3	4	5
Addr	v[0]	v[0]+4	v[0]+8	v[0]+12	v[0]+16	v[0]+20
Hit/Miss	Miss	Hit	Hit	Hit	Miss	Hit

Compare the two programs

```
for (int i = 0; i < n; i++) {  
    z[i] = x[i] - y[i];  
    z[i] = z[i] * z[i];  
}
```

```
for (int i = 0; i < n; i++) {  
    z[i] = x[i] - y[i];  
}  
for (int i = 0; i < n; i++) {  
    z[i] = z[i] * z[i];  
}
```

Which version is more efficient if we have large arrays?

Data Locality

Parallelism and data locality go hand-in-hand

- + Stride-1 reference patterns are good—spatial locality
- + Repeated references to memory locations or variables are good—temporal locality

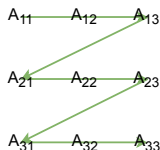
Always focus on optimizing the common case (e.g., inner loops)!

Layout of C Arrays in Memory

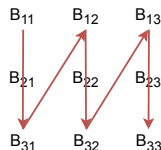
- C arrays are allocated in row-major order
 - + Stepping through columns in one row exploits spatial locality if block size $B > 4$ bytes
 - Stepping through rows in one column accesses distant elements—no spatial locality!

```
int A[N][N], B[N][N];  
  
// row major  
for (i = 0; i < N; i++)  
    for (j = 0; j < N; j++)  
        sum += A[i][j];  
  
// column major  
for (j = 0; j < N; j++)  
    for (i = 0; i < N; i++)  
        sum += A[i][j];
```

Row-major



Column-major



Compare Access Strides

Assume words are 4 bytes, cache blocks are 4 words, and the cache is initially empty.

```
int sumarrayrows(int A[M][N]) {  
    int i, j, sum = 0;  
    for (i = 0; i < M; i++)  
        for (j = 0; j < N; j++)  
            sum += A[i][j];  
    return sum;  
}
```

```
int sumarraycols(int A[M][N]) {  
    int i, j, sum = 0;  
    for (j = 0; j < M; j++)  
        for (i = 0; i < N; i++)  
            sum += A[i][j];  
    return sum;  
}
```

What are the miss rates per iteration if the array A (i) fits in cache and (ii) does not fit in cache?

Miss Rate Analysis for Matrix-Matrix Multiply

- Matrix-Vector multiply and Matrix-Matrix multiply are important kernels
 - ▶ Building blocks across different domains (i.e., ML and computational science applications)
- Multiply $N \times N$ matrices with $\mathcal{O}(N^3)$ operations
 - ▶ N reads per source element and N values summed per destination
 - ▶ sum will be stored in a register
- Algorithm is computation-bound
 - ▶ $3N^2$ memory locations and $\mathcal{O}(N^3)$ operations
 - ▶ Memory accesses should not constitute a bottleneck

```
/* ijk */  
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        sum = 0.0;  
        for (k=0; k<n; k++) {  
            sum += A[i][k] * B[k][j];  
        }  
        C[i][j] = sum;  
    }  
}
```

Cache Model Assumptions

- Consider cold and capacity misses only and ignore conflict misses for now
- **Large cache** model: only cold misses
- **Small cache** model: both cold and capacity misses
 - ▶ Matrix dimensions are so large that even one matrix row overflows the cache

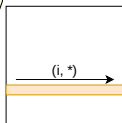
Miss Rate Analysis for Matrix Multiply (ijk)

Analysis method: look at access patterns of the innermost loop

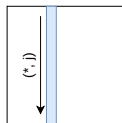
```
/* ijk */  
for (i=0; i<n; i++) {  
  for (j=0; j<n; j++) {  
    sum = 0.0;  
    for (k=0; k<n; k++) {  
      sum += A[i][k] * B[k][j];  
    }  
    C[i][j] = sum;  
  }  
}
```

two loads,
zero
stores

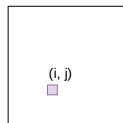
Inner loop



A



B



C

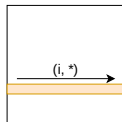
Misses per inner loop iteration

A	B	C
0.25	1.0	0.0

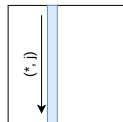
Miss Rate Analysis for Matrix Multiply (jik)

```
/* jik */  
for (j=0; j<n; j++) {  
    for (i=0; i<n; i++) {  
        sum = 0.0;  
        for (k=0; k<n; k++) {  
            sum += A[i][k] * B[k][j];  
        }  
        C[i][j] = sum;  
    }  
}
```

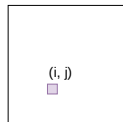
Inner loop



A



B



C

Misses per inner loop iteration

A	B	C
0.25	1.0	0.0

same behavior as
ijk

Miss Rate Analysis for Matrix Multiply (kij)

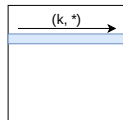
```
/* kij */  
for (k=0; k<n; k++) {  
  for (i=0; i<n; i++) {  
    r = A[i][k];  
    for (j=0; j<n; j++) {  
      C[i][j] += r * B[k][j];  
    }  
  }  
}
```

two loads,
one store

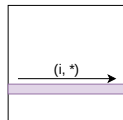
Inner loop



A



B



C

Misses per inner loop iteration

A	B	C
0.0	0.25	0.25

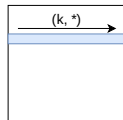
Miss Rate Analysis for Matrix Multiply (ikj)

```
/* ikj */  
for (i=0; i<n; i++) {  
    for (k=0; k<n; k++) {  
        r = A[i][k];  
        for (j=0; j<n; j++) {  
            C[i][j] += r * B[k][j];  
        }  
    }  
}
```

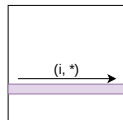
Inner loop



A



B



C

Misses per inner loop iteration

A	B	C
0.0	0.25	0.25

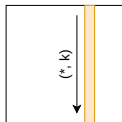
same behavior as
kij

Miss Rate Analysis for Matrix Multiply (jki)

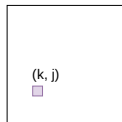
```
/* jki */  
for (j=0; j<n; j++) {  
    for (k=0; k<n; k++) {  
        r = B[k][j];  
        for (i=0; i<n; i++) {  
            C[i][j] += A[i][k] * r;  
        }  
    }  
}
```

two loads,
one store

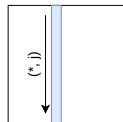
Inner loop



A



B



C

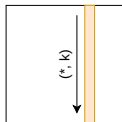
Misses per inner loop iteration

A	B	C
1.0	0.0	1.0

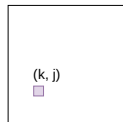
Miss Rate Analysis for Matrix Multiply (kji)

```
/* kji */  
for (k=0; k<n; k++) {  
    for (j=0; j<n; j++) {  
        r = B[k][j];  
        for (i=0; i<n; i++) {  
            C[i][j] += A[i][k] * r;  
        }  
    }  
}
```

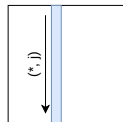
Inner loop



A



B



C

Misses per inner loop iteration

A	B	C
1.0	0.0	1.0

same behavior as
jki

Summary of Misses Per Inner Loop Iteration

```
for (i=0; i<n; i++) {  
  for (j=0; j<n; j++) {  
    sum = 0.0;  
    for (k=0; k<n; k++) {  
      sum += A[i][k]*B[k][j];  
    }  
    C[i][j] = sum;  
  }  
}
```

ijk and jik

- 2 loads, 0 store
- misses/iter = 1.25

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

kij and ikj

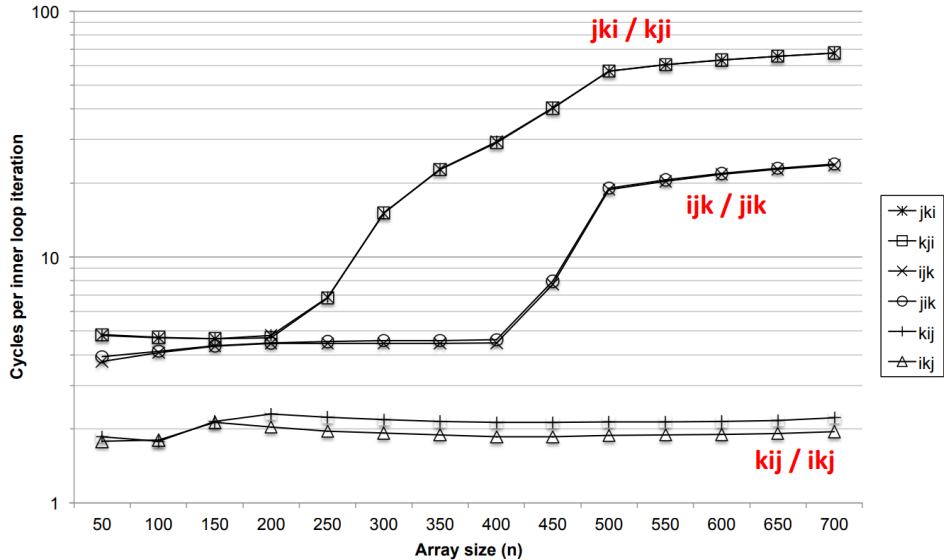
- 2 loads, 1 store
- misses/iter = 0.5

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

jki and kji

- 2 loads, 1 store
- misses/iter = 2.0

Matrix Multiply Performance on Core i7



Total Cache Misses (ijk)

```
for (i=0; i<n; i++) {  
  for (j=0; j<n; j++) {  
    sum = 0.0;  
    for (k=0; k<n; k++)  
      sum += A[i][k] * B[k][j];  
    C[i][j] = sum;  
  }  
}
```

Matrices are very
large compared
to cache size

	A	B	C
i	?	?	?
j	?	?	?
k	?	?	?
	?	?	?

BL is the number of elements that
fit in a cache block

Total Cache Misses (ijk)

```
for (i=0; i<n; i++) {  
  for (j=0; j<n; j++) {  
    sum = 0.0;  
    for (k=0; k<n; k++)  
      sum += A[i][k] * B[k][j];  
    C[i][j] = sum;  
  }  
}
```

BL is the number of elements that fit in a cache block

Matrices are very large compared to cache size

	A	B	C
i	n	n	n
j	n	n	$\frac{n}{BL}$
k	$\frac{n}{BL}$	n	1
	$\frac{n^3}{BL}$	n^3	$\frac{n^2}{BL}$

Total Cache Misses (jki)

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

Matrices are very
large compared
to cache size

	A	B	C
j	?	?	?
k	?	?	?
i	?	?	?
	?	?	?

Cache Miss Analysis for MVM

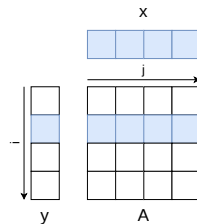
```
for (i = 0; i < N; i++) {  
    for (j = 0; j < N; j++) {  
        y[i] += A[i][j]*x[j];  
    }  
}
```

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{bmatrix} \times \begin{bmatrix} 2 \\ 5 \\ 1 \\ 8 \end{bmatrix} = \begin{bmatrix} 4 \\ 47 \\ 5 \\ 68 \end{bmatrix}$$

- Number of memory locations: $N^2 + 2N$
- Number of operations: $\mathcal{O}(N^2)$
- MVM is limited by memory bandwidth unlike matrix multiplication

MVM (ij)

```
for (i = 0; i < N; i++) {  
    for (j = 0; j < N; j++) {  
        y[i] += A[i][j]*x[j];  
    }  
}
```



Large Cache Model

- Misses:

A : ?
x : ?
y : ?

- Total: ?

Small Cache Model

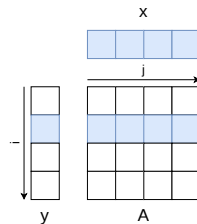
- Misses:

A : ?
x : ?
y : ?

- Total: ?

MVM (ji)

```
for (j = 0; j < N; j++) {  
    for (i = 0; i < N; i++) {  
        y[i] += A[i][j]*x[j];  
    }  
}
```



Large Cache Model

- Misses:

A : ?
x : ?
y : ?

- Total: ?

Small Cache Model

- Misses:

A : ?
x : ?
y : ?

- Total: ?

Cache Miss Estimation Example

Consider a cache of size 64K words and a block of size 8 words. Perform cache miss analysis considering (i) direct-mapped and (ii) fully associative caches.

We will ignore interference among the three matrices but will now consider conflict misses.

```
#define N 512
double A[N][N], B[N][N], C[N][N];
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            C[i][j] += A[i][k]*B[k][j];
```

Cache Miss Estimation Example

```
#define N 512
double A[N][N], B[N][N], C[N][N];
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            C[i][j] += A[i][k]*B[k][j];
```

Direct-Mapped Cache

	A	B	C
i	?	?	?
j	?	?	?
k	?	?	?
	?	?	?

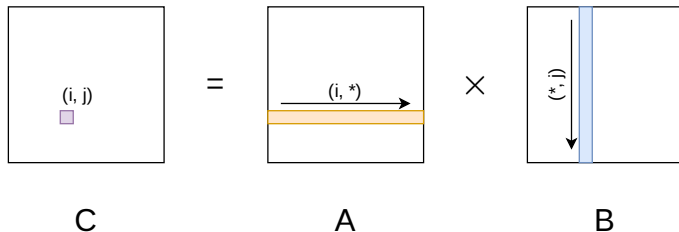
Fully-associative Cache

	A	B	C
i	?	?	?
j	?	?	?
k	?	?	?
	?	?	?

Using Blocking to Improve Temporal Locality

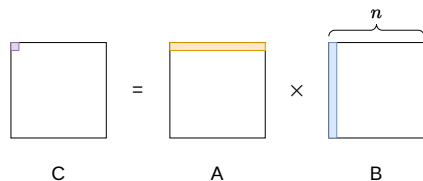
Revisiting Matrix Multiplication

```
/* Multiply N×N matrices A and B */  
void mmm(double *A, double *B, double *C, int N) {  
    for (int i = 0; i < N; i++)  
        for (int j = 0; j < N; j++)  
            for (int k = 0; k < N; k++)  
                C[i*N+j] += A[i*N+k]*B[k*N+j];  
}
```

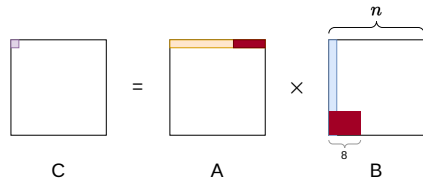


Cache Miss Analysis

- Assumptions
 - ▶ Matrix elements are doubles
 - ▶ Cache block is of eight doubles
 - ▶ Cache size $\ll n$ (i.e., much smaller than n)
- First (i, j) iteration: $\frac{n}{8} + n = \frac{9n}{8}$ misses



- Afterward in cache (schematic):

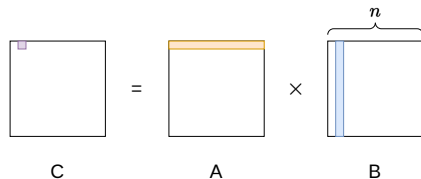


Cache Miss Analysis

- Assumptions

- ▶ Matrix elements are doubles
- ▶ Cache block is of eight doubles
- ▶ Cache size $\ll n$ (i.e., much smaller than n)

- Second (i, j) iteration: $\frac{n}{8} + n = \frac{9n}{8}$ misses



- Total misses: $\frac{9n}{8} * n^2 = \frac{9n^3}{8}$

Cache Blocking

Improve data reuse by chunking the data into smaller blocks

- A block is supposed to fit in the cache

```
for (i = 0; i < N; i++) {  
    ...  
}
```

```
for (j = 0; j < N; j += BLK) {  
    for (i = j; i < min(N, j+BLK); i++) {  
        ...  
    }  
}
```

```
for (i = 0; i < NBODIES; i++) {  
    for (j = 0; j < NBODIES; j++) {  
        OUT[i] += compute(i,j);  
    }  
}
```

```
for (j = 0; j < NBODIES; j += BLK) {  
    for (i = 0; i < NBODIES; i++) {  
        for (jj = 0; jj < BLK; jj++)  
            OUT[i] += compute(i,j+jj);  
    }  
}
```

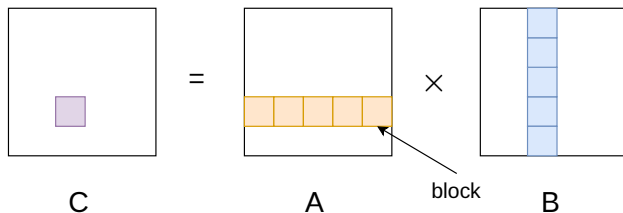
MVM with 2×2 Blocking

```
int i, j;
int N = 100;
int A[N][N], b[N], c[N];
for (i = 0; i < N; i++) {
    c[i] = 0;
    for (j = 0; j < N; j++) {
        c[i] = c[i] + A[i][j] * b[j];
    }
}
```

```
int i, j, ii, jj;
int N = 100;
int A[N][N], b[N], c[N];
for (i = 0; i < N; i += 2) {
    c[i] = 0; c[i + 1] = 0;
    for (j = 0; j < N; j += 2) {
        for (ii = i; ii < min(i + 2, N); ii++) {
            for (jj = j; jj < min(j + 2, N); jj++)
                c[ii] = c[ii] + A[ii][jj] * b[jj];
        }
    }
}
```

Blocked Matrix Multiplication

```
/* Multiply N×N matrices A and B */
void mmm(double *A, double *B, double *C, int N) {
    for (int i = 0; i < N; i += BLK)
        for (int j = 0; j < N; j += BLK)
            for (int k = 0; k < N; k += BLK)
                /* B×B mini-matrix (block) multiplications */
                for (int ii = i; ii < i+BLK; ii++)
                    for (int jj = j; jj < j+BLK; jj++)
                        for (int kk = k; kk < k+BLK; kk++)
                            C[ii*N+jj] += A[ii*N+kk]*B[kk*N+jj];
}
```



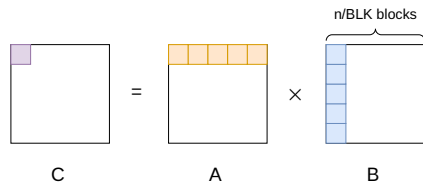
Cache Miss Analysis with Blocking

- Assumptions

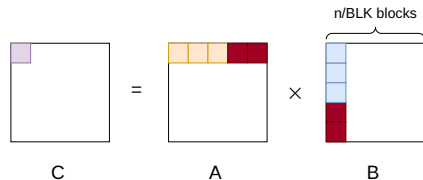
- ▶ Matrix elements are doubles
- ▶ Cache block is of eight doubles
- ▶ Cache size $\ll n$ (i.e., much smaller than n)
- ▶ Three $BLK \times BLK$ blocks fit in the cache, i.e., $3BLK^2 < C$

- First block iteration:

- ▶ $\frac{BLK^2}{8}$ misses for each block
- ▶ Total (ignoring matrix C):
$$2 * \frac{n}{BLK} * \frac{BLK^2}{8} = \frac{n * BLK}{4}$$



- Afterward in cache (schematic):



Cache Miss Analysis with Blocking

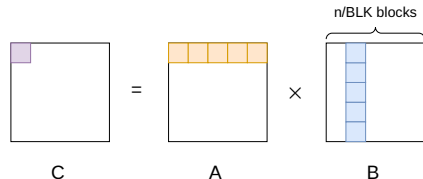
- Assumptions

- ▶ Matrix elements are doubles
- ▶ Cache block is of eight doubles
- ▶ Cache size $\ll n$ (i.e., much smaller than n)
- ▶ Three $BLK \times BLK$ blocks fit in the cache, i.e., $3BLK^2 < C$

- Second block iteration:

- ▶ Same as the first
- ▶ Total (ignoring matrix C):

$$2 * \frac{n}{BLK} * \frac{BLK^2}{8} = \frac{n * BLK}{4}$$



- Total misses: $\frac{n * BLK}{4} * \left(\frac{n}{BLK}\right)^2 = \frac{n^3}{4 * BLK}$

Summary

Without blocking	With blocking
$\frac{9}{8}n^3$	$\frac{1}{4BLK}n^3$

- Find the largest possible block size BLK , but limit $3BLK^2 < C$!
- The reason for the dramatic difference is that matrix multiplication has inherent temporal locality
 - ▶ Input data is $3n^2$, computation is $2n^3$, and every array element is used $\mathcal{O}(n)$ times!
 - ▶ But the program has to be written properly (choose good variant and BLK)

Pointers to Exploit Locality in your Code

Focus on the more frequently executed parts of the code (aka common case) (e.g., inner loops)

Maximize spatial locality with low strides (preferably 1)

Maximize temporal locality by reusing the data as much as possible

References



R. Bryant and D. O'Hallaron. Computer Systems: A Programmer's Perspective. Sections 6.5–6.6, 3rd edition, Pearson Education.



A. Aho et al. Compilers: Principles, Techniques, and Tools. Section 11.2, 2nd edition, Pearson Education.