Matrix Multiplication

CPSC 424/524
Spring 2017

Writing Cache Friendly Code

- Make the common cases go fast
  - Focus on the inner loops of the core functions
- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Programmers must have a quantitative understanding of cache memories in order to reduce to practice their intuitive qualitative notions of locality.

Matrix Multiplication Example

- Description:
  - Multiply N x N matrices
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination element
  - Registers may be used to hold scalars

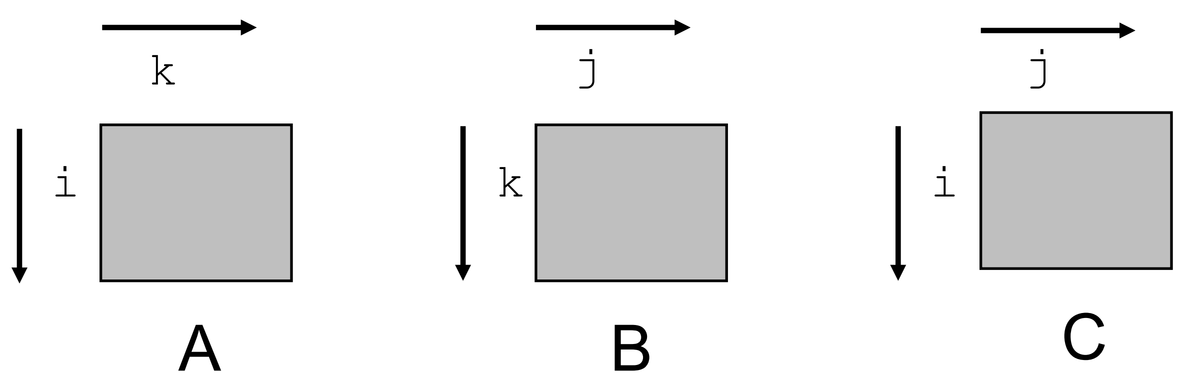
Variable `sum` *held in register*

```
/* 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;
  }
}
```

C = A * B

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Single-level cache with line size L = 32 bytes (four 64-bit words)
  - Double precision matrix where dimension (N) is very large
    - Think of 1/N as ≈ 0.0
  - Cache is not big enough to hold multiple rows/columns
  - Registers used to hold scalars (partial sum, index variables, etc.)
- Analysis Method:
  - Look at access pattern of inner loop



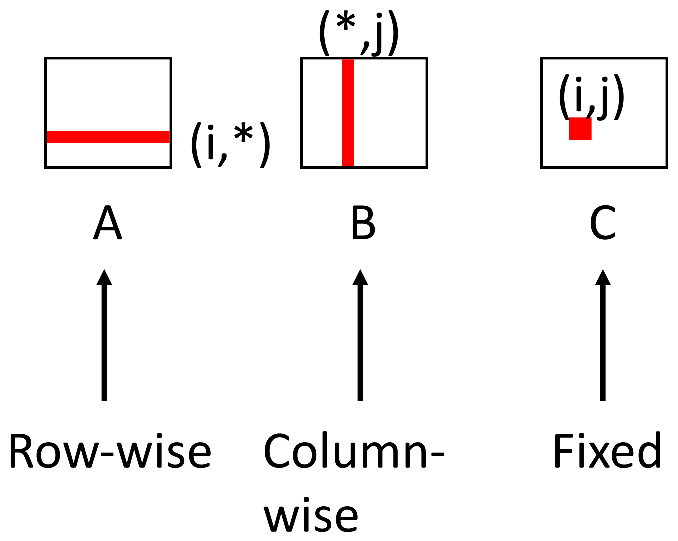
Layout of C Arrays in Memory

- C arrays allocated in row-major order
  - Each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (j = 0; j < N; j++) sum += A[i][j];`
  - Accesses successive elements
  - If cache line size (L) > 8 bytes, exploit spatial locality
    - "Compulsory Miss Rate" = 8 bytes / L for a single line
- Stepping through rows in one column:
  - `for (i = 0; i < N; i++) sum += A[i][j];`
  - Accesses distant elements
  - No spatial locality!
    - Compulsory Miss Rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* 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;
  }
}
```

Inner loop:

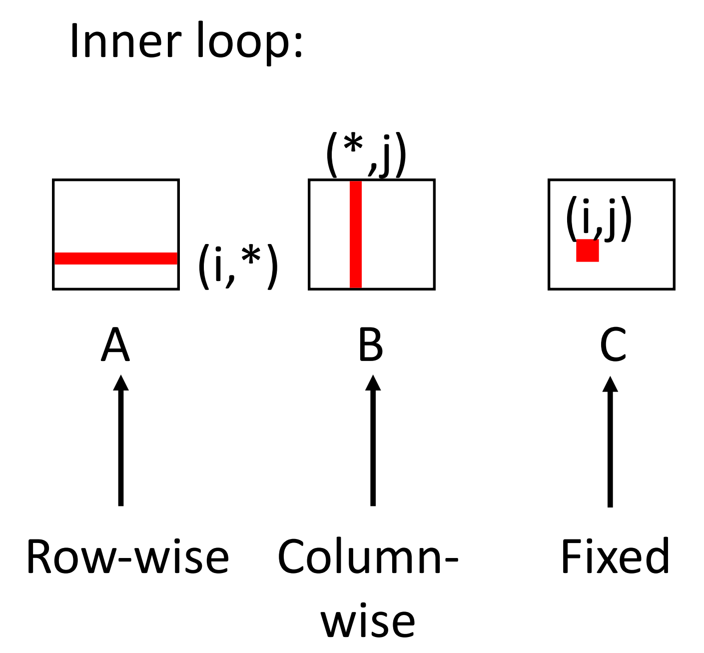


<u>Accesses & Misses per inner loop iteration:</u>

Loads	Stores	A	B	C
2	0	0.25	1.0	0.0

Matrix Multiplication (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
  }
}
```



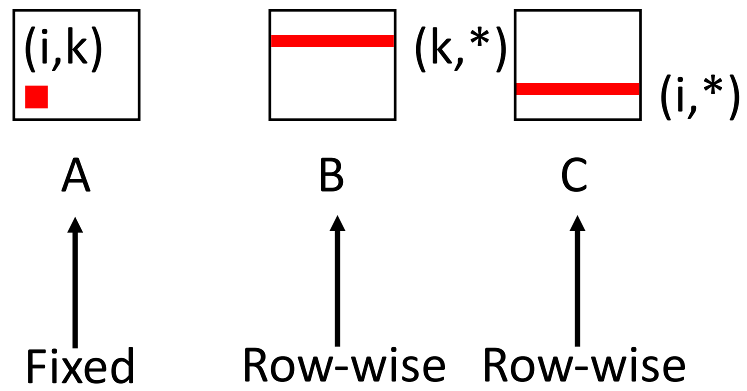
Accesses & Misses per inner loop iteration:

Loads	Stores	A	B	C
2	0	0.25	1.0	0.0

Matrix Multiplication (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];
  }
}
```

Inner loop:

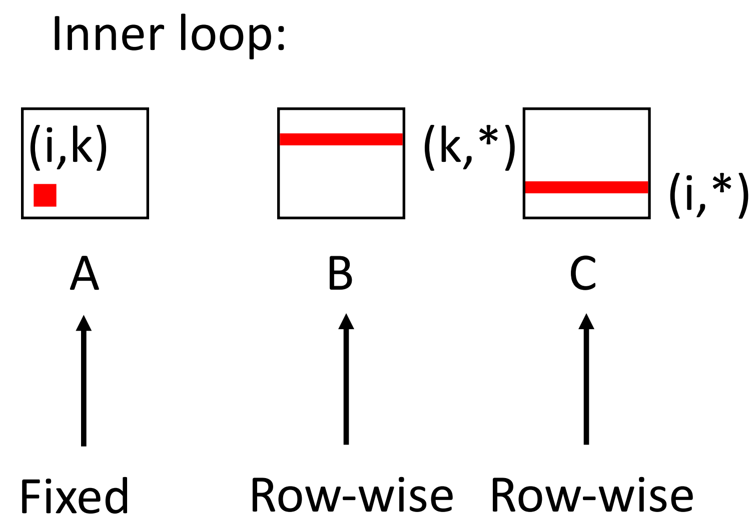


Accesses & Misses per inner loop iteration:

Loads	Stores	A	B	C
2	1	0	0.25	0.25

Matrix Multiplication (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];
  }
}
```



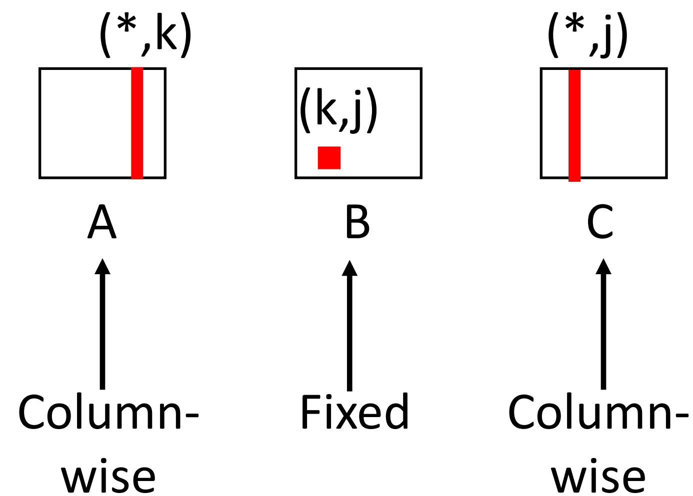
Accesses & Misses per inner loop iteration:

Loads	Stores	A	B	C
2	1	0	0.25	0.25

Matrix Multiplication (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;
  }
}
```

Inner loop:



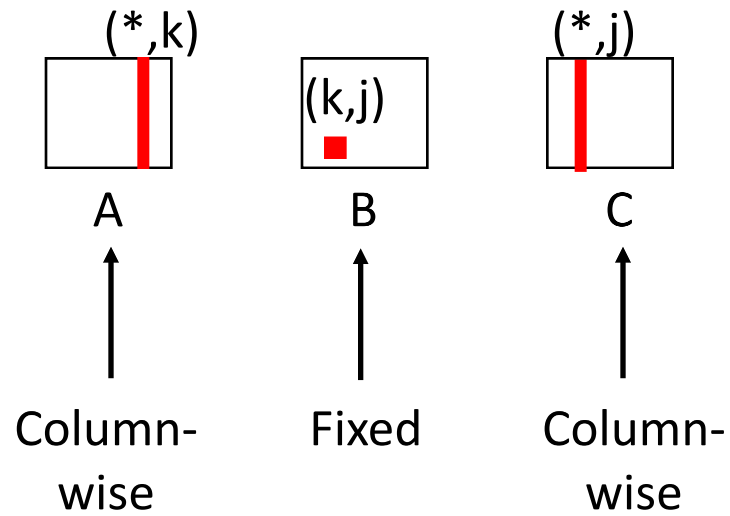
Accesses & Misses per inner loop iteration:

Loads	Stores	A	B	C
2	1	1.0	0.0	1.0

Matrix Multiplication (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:



Accesses & Misses per inner loop iteration:

Loads	Stores	A	B	C
2	1	1.0	0.0	1.0

Summary of Matrix Multiplication

```
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 (& jik):
- 2 loads, 0 stores
- misses/iter = 1.25

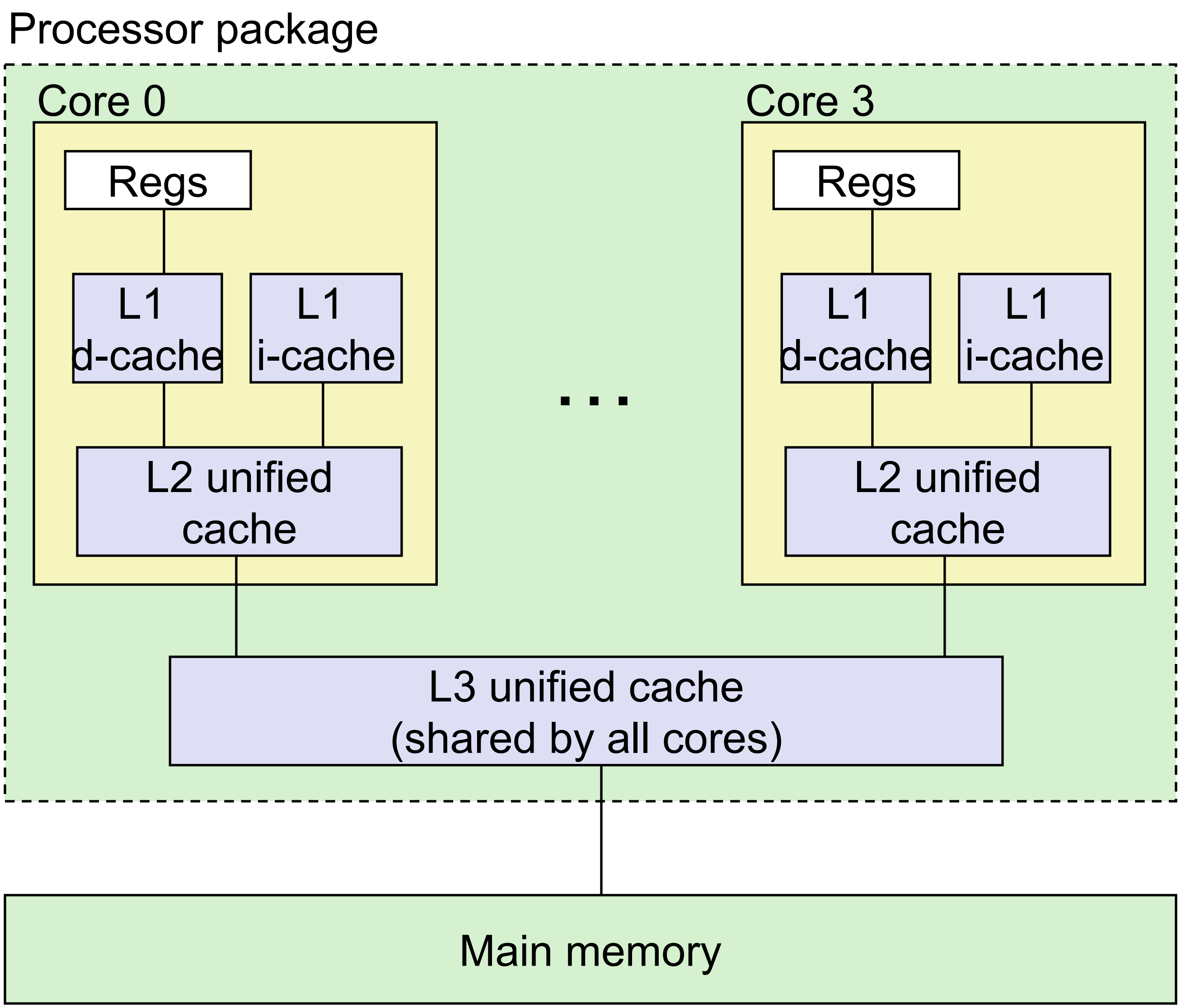
```
for (k=0; k<N; k++) {
 for (i=0; i<N; i++) {
  r = A[i][k];
  for (j=0; j<N; j++)
   C[i][j] += r * B[k][j];
 }
}
```

kij (& 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 (& kji):
- 2 loads, 1 store
- misses/iter = 2.0

Intel Core i7 Cache Hierarchy



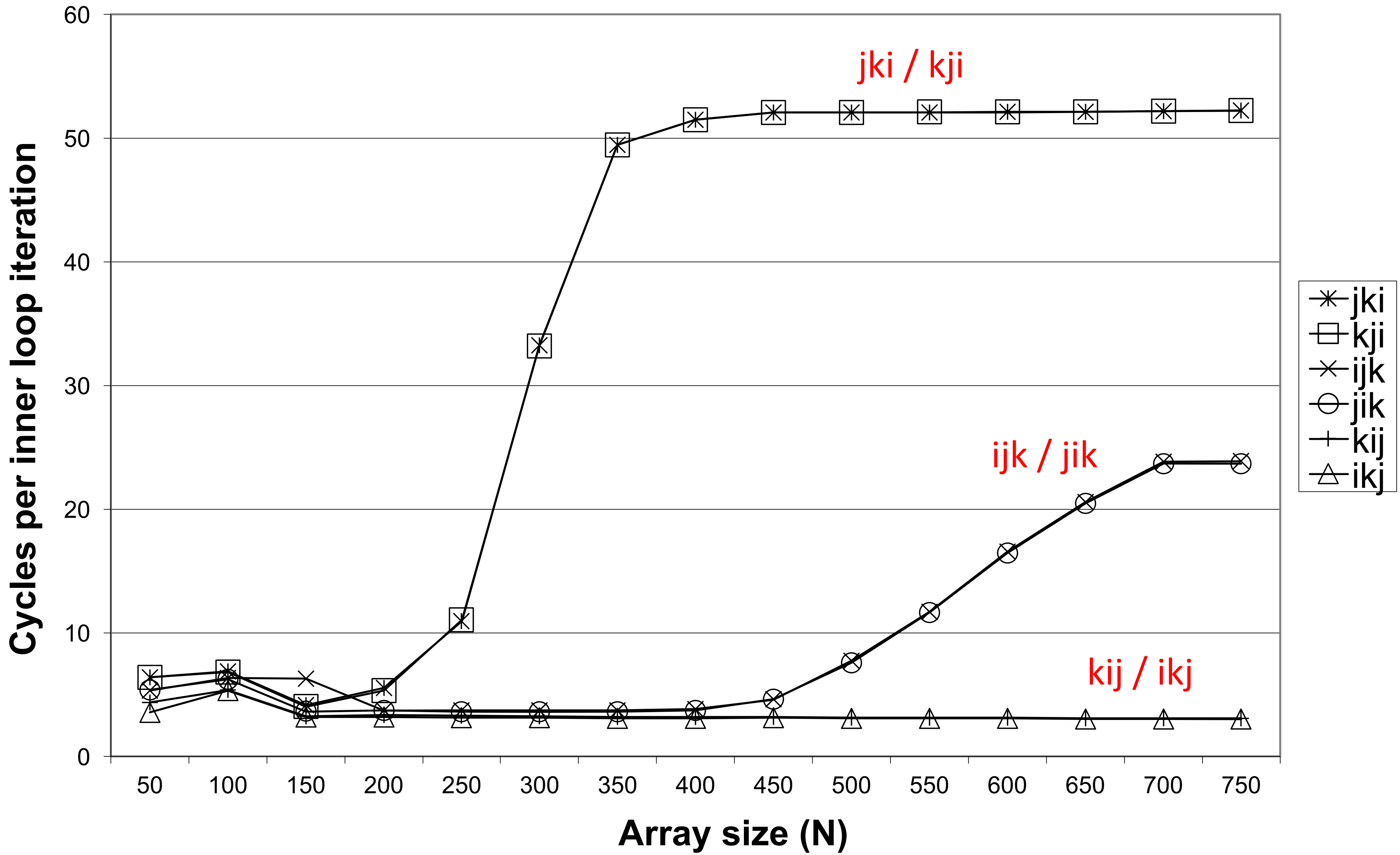
L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

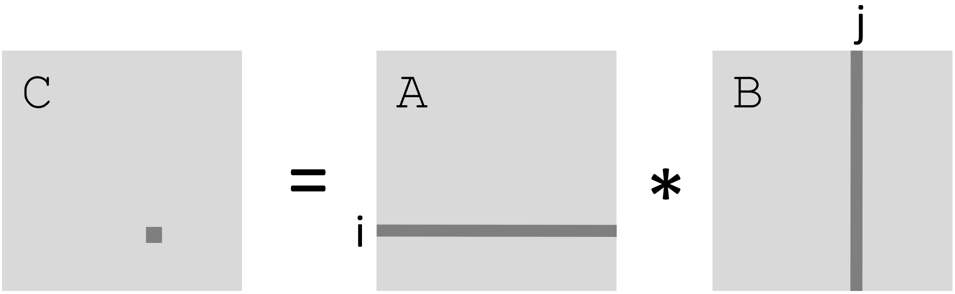
Block size: 64 bytes for all caches.

Core i7 Matrix Multiply Performance



Blocking to improve spatial locality

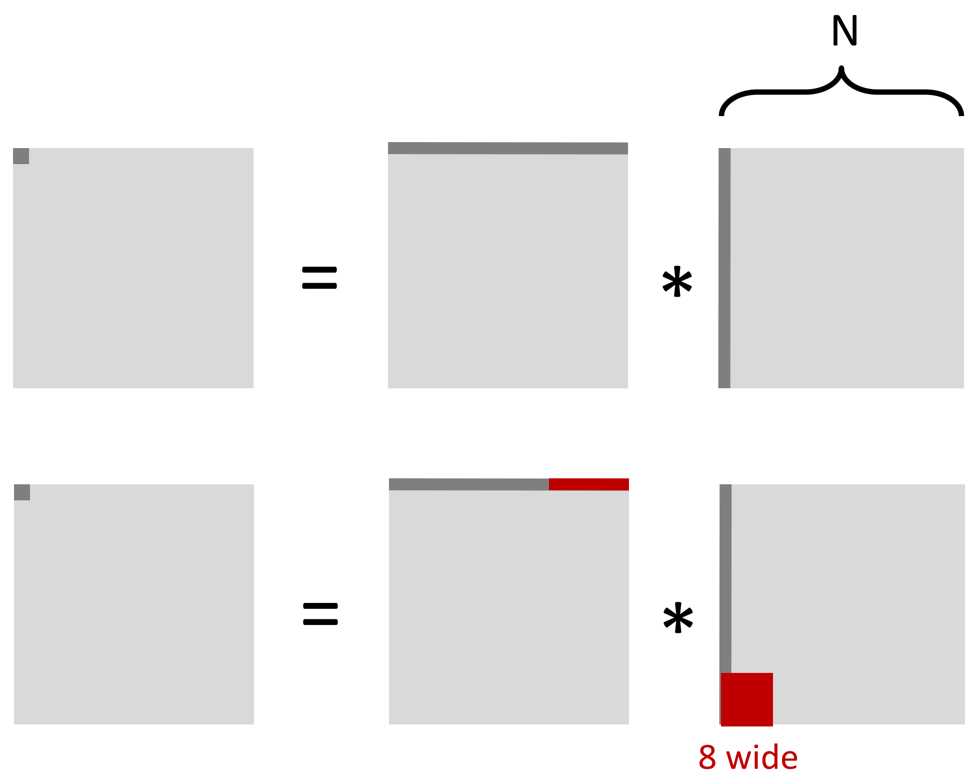
```
C = (double *) calloc(sizeof(double), N*N);

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



Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache line L = 64 bytes (8 doubles)
  - Cache size M << N (much smaller than N)

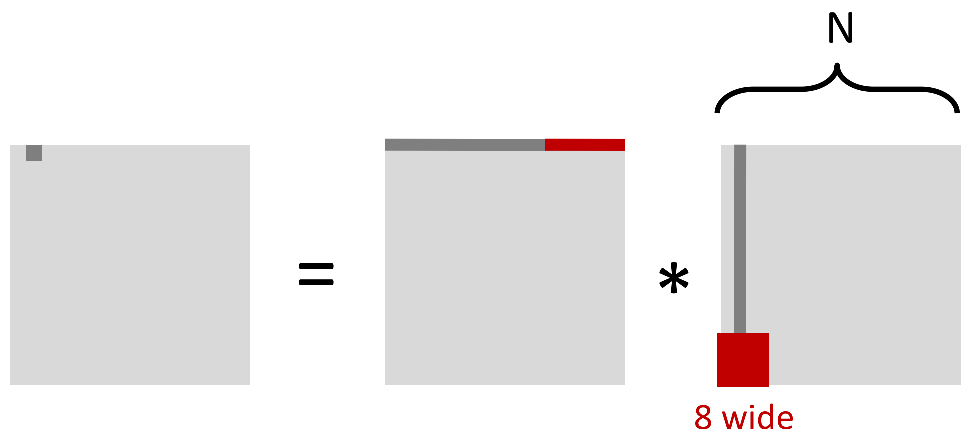
- First iteration:
  - N/8 + N = 9N/8 misses
  - Afterwards in cache: (schematic)



Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 64 bytes (8 doubles)
  - Cache size M << N (much smaller than N)

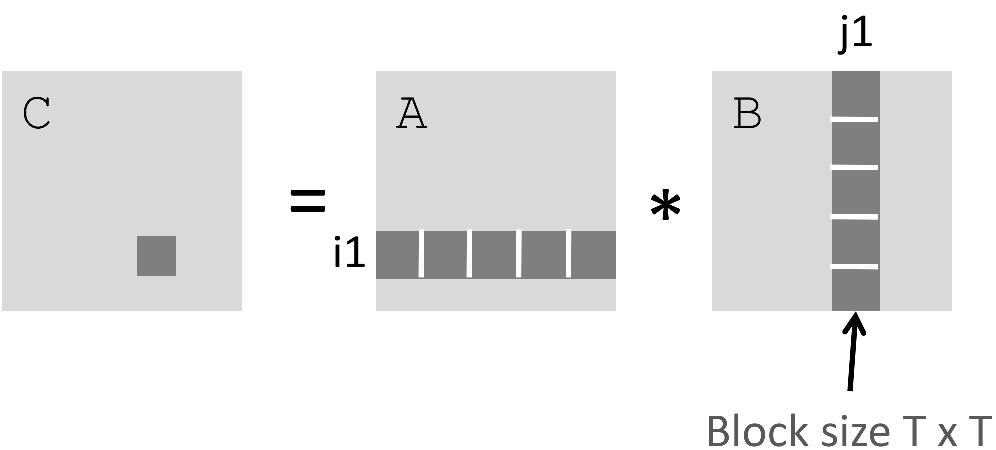
- Second iteration:
  - Again:
    N/8 + N = 9N/8 misses



- Total misses:
  - $9N/8 * N^2 = (9/8) * N^3$

Blocked Matrix Multiplication

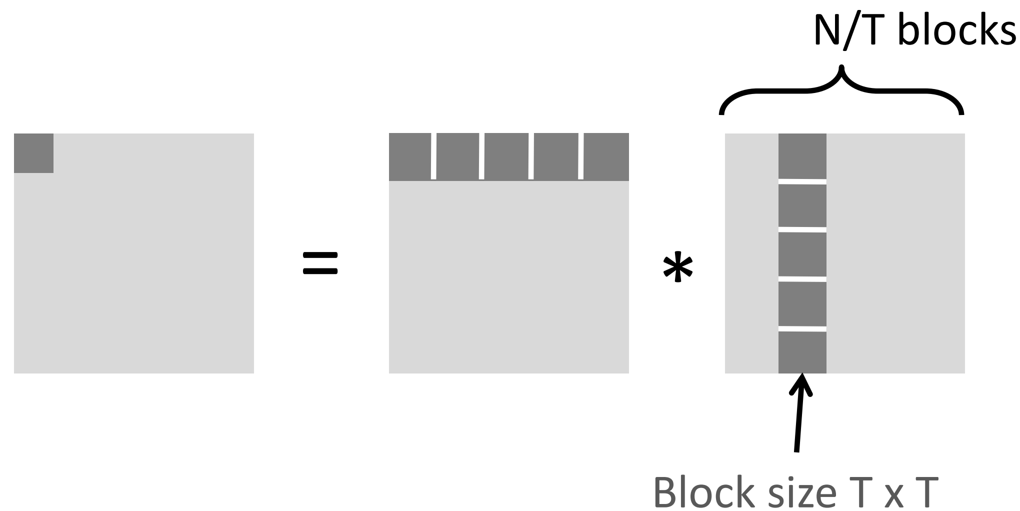
```
C = (double *) calloc(sizeof(double), N*N);

/* Multiply N x N matrices A and B  */
void mmm(double *A, double *B, double *C, int N) {
    int i, j, k;
    for (i = 0; i < N; i+=T)
        for (j = 0; j < N; j+=T)
            for (k = 0; k < N; k+=T)
                /* T x T mini matrix multiplications */
                for (i1 = i; i1 < i+T; i++)
                    for (j1 = j; j1 < j+T; j++)
                        for (k1 = k; k1 < k+T; k++)
                            C[i1*N+j1] += A[i1*N + k1]*B[k1*N + j1];
}
```



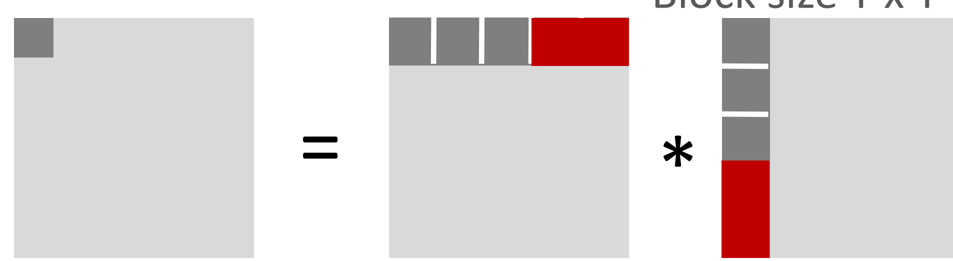
Cache Miss Analysis

- Assume:
  - Cache block = 64 bytes (8 doubles)
  - Cache size M << N (much smaller than N)
  - Three blocks fit into cache: $3T^2 < M$

- First (block) iteration:
  - $T^2/8$ misses for each block
  - $2N/T * T^2/8 = NT/4$ (omitting matrix C)

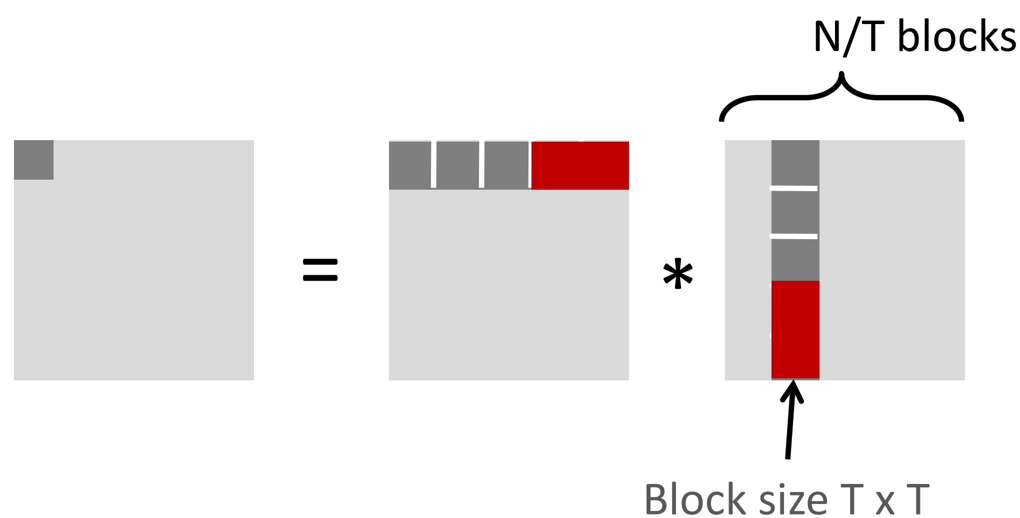


  - Afterwards in cache (schematic)



Cache Miss Analysis

- Assume:
  - Cache block = 64 bytes (8 doubles)
  - Cache size $M \ll N$ (much smaller than N)
  - Three blocks fit into cache: $3T^2 < M$
- Second (block) iteration:
  - Same as first iteration
  - $2N/T * T^2/8 = NT/4$



- Total misses:
  - $NT/4 * (N/T)^2 = N^3/(4T)$

Summary

- No blocking: $(9/8) * N^3$
- Blocking: $1/(4T) * N^3$

- Suggests using largest possible block size T, but limit $3T^2 < M$!

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3N^2$, computation $2N^3$
    - Every array elements used O(N) times!
  - But program has to be written properly

Recursion: Cache Oblivious Algorithms

- The blocked algorithm requires finding a good block size, essentially requiring knowledge of M.

- Cache Oblivious Algorithms offer an alternative:
  - Treat NxN matrix multiply recursively, decomposing it into a set of smaller and smaller problems by dividing the largest dimension
  - Eventually, these will fit in cache (more subdivision is okay!)

- Cases for A (nxk) * B (kxm)
  - Case1: n $\geq$ max{k,m}: split rows of A
  - Case 2: k $\geq$ max{n,m}: split columns of A and rows of B
  - Case 3: m $\geq$ max{n,k}: split columns of B

$$\boxed{\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}}$$

Case 1

$$\boxed{(A_1, A_2)\begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = (A_1 B_1 + A_2 B_2)}$$

Case 2

$$\boxed{A(B_1, B_2) = (AB_1, AB_2)}$$

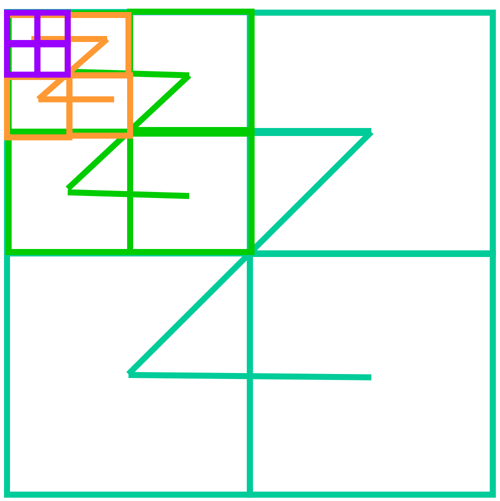
Case 3

Experience with Cache-Oblivious Algorithms

- In practice, need to cut off recursion well before 1x1 blocks
  - 16x16 blocks often used
- Implementing a high-performance Cache-Oblivious code isn't easy
  - Careful attention to micro-kernel is needed
- Recursive Micro-Kernels yield less performance than iterative ones using same scheduling techniques
  - Using fully recursive approach with highly optimized recursive micro-kernel, Pingali et al report never getting more than 2/3 of peak.
  - Pre-fetching is needed to compete with best code: not well-understood in the context of cache oblivious codes

Recursive Data Layouts

- A related idea is to use a recursive data structure for the matrix
  - Improve locality with machine-independent data structure
  - Can minimize latency with multiple levels of memory hierarchy
- Several possible recursive decompositions depending on the order of the sub-blocks
- This figure shows Z-Morton Ordering (“space filling curve”)
- For more info: Gustavson, Kagstrom, et al, SIAM Review, 2004



Advantage: recursive layout works well for any cache size

Disadvantages:
- Expensive index calculations to find A[i,j]
- May need to switch layouts for small sizes

Concluding Remarks

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor “cache friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
    - “Cache Oblivious” Algorithms may help
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)