Lecture 10: Memory Hierarchy
-- Memory Technology and Principal of Locality

Locality of Matrix Multiplication and Two Cache Optimization Algorithms

CSCE 513 Computer Architecture

Department of Computer Science and Engineering

Yonghong Yan

yanyh@cse.sc.edu

https://passlab.github.io/CSCE513

Sources of locality

- Temporal locality
  - Code within a loop
  - Same instructions fetched repeatedly
- Spatial locality
  - Data arrays
  - Local variables in stack
  - Data allocated in chunks (contiguous bytes)

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

Writing Cache Friendly Code

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- Examples:
  - cold cache, 4-byte words, 4-word cache blocks

```
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;
}
```

Miss rate = 1/4 = 25%

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 100%

Matrix Multiplication Example

- Major cache effects to consider
  - Total cache size
    - Exploit temporal locality and blocking)
  - Block size
    - Exploit spatial locality

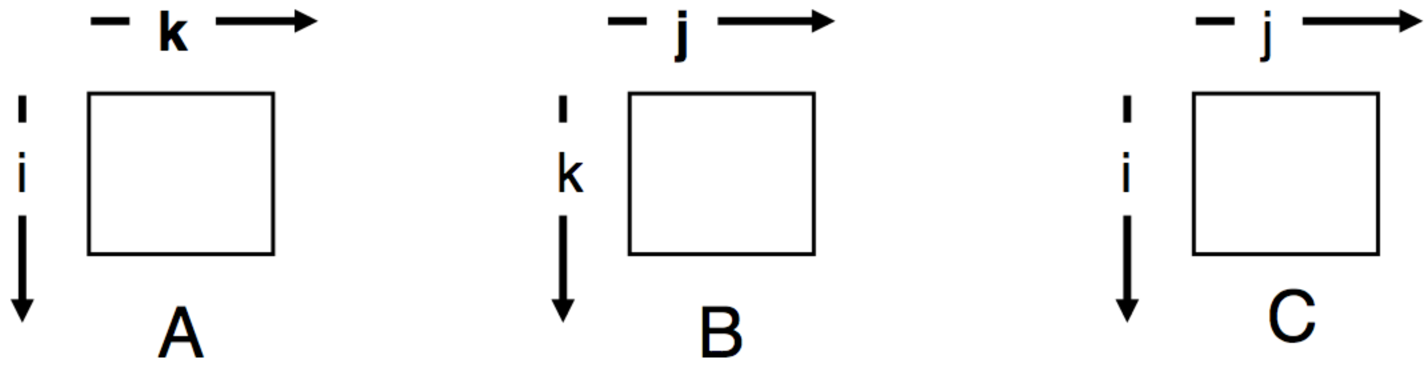
```
/* ijk */
for (i=0; i<n; i++)   {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable `sum` *held in register*

- Description:
  - Multiply N x N matrices
  - $O(N^3)$ total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

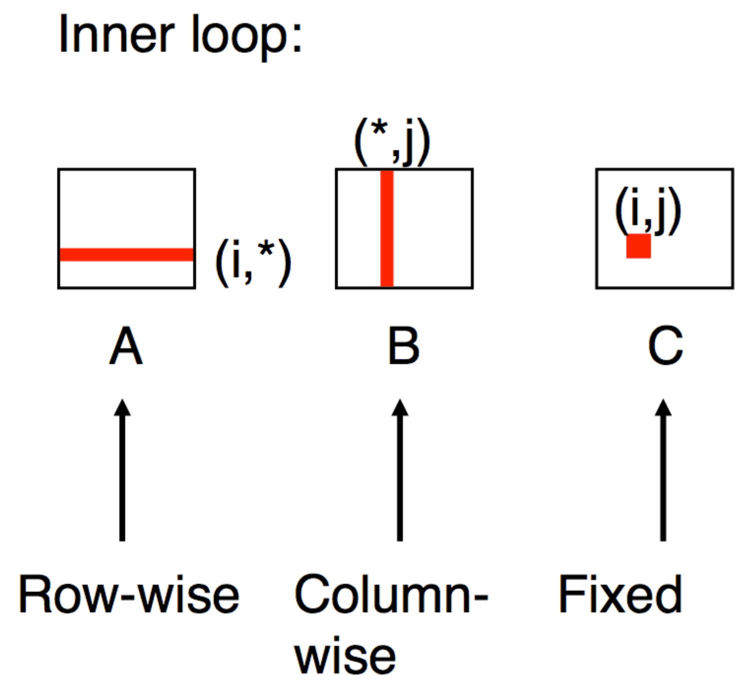
Miss Rate Analysis for Matrix Multiply

- Assume:
  - Cache line size = 32 Bytes (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis method:
  - Look at access pattern of inner loop



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;
  }
}
```



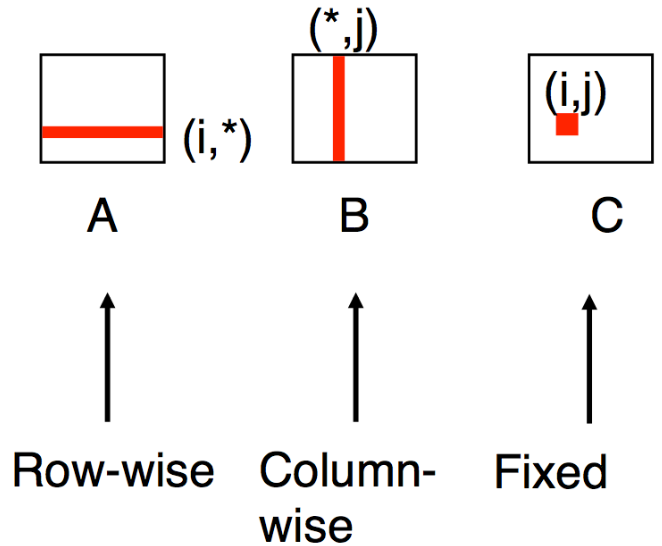
- Misses per Inner Loop Iteration:

A	B	C
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
  }
}
```

Inner loop:



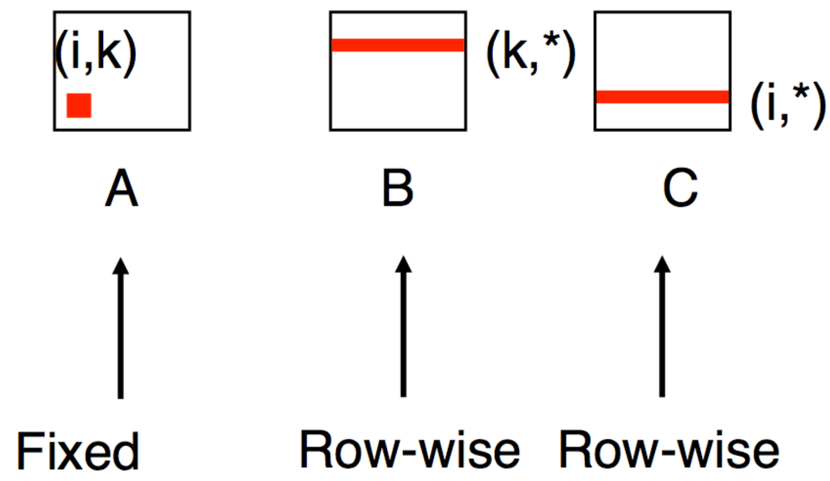
- <u>Misses per Inner Loop Iteration:</u>

A	B	C
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:

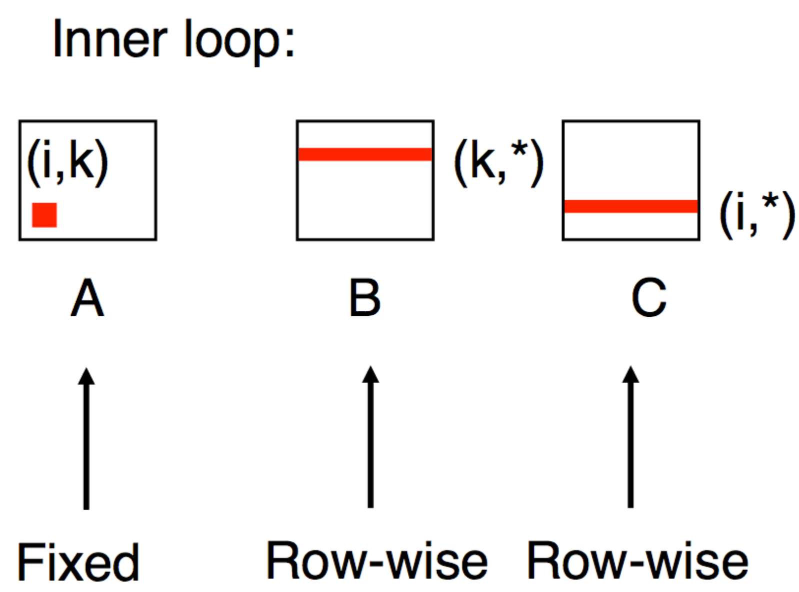


- Misses per Inner Loop Iteration:

A	B	C
0.0	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];
  }
}
```

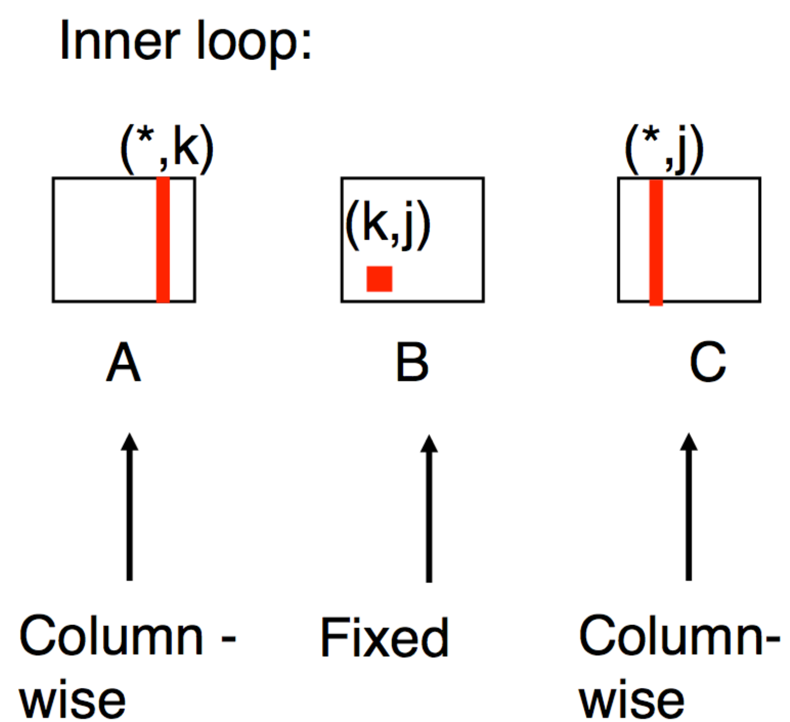


- Misses per Inner Loop Iteration:

A	B	C
0.0	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;
  }
}
```

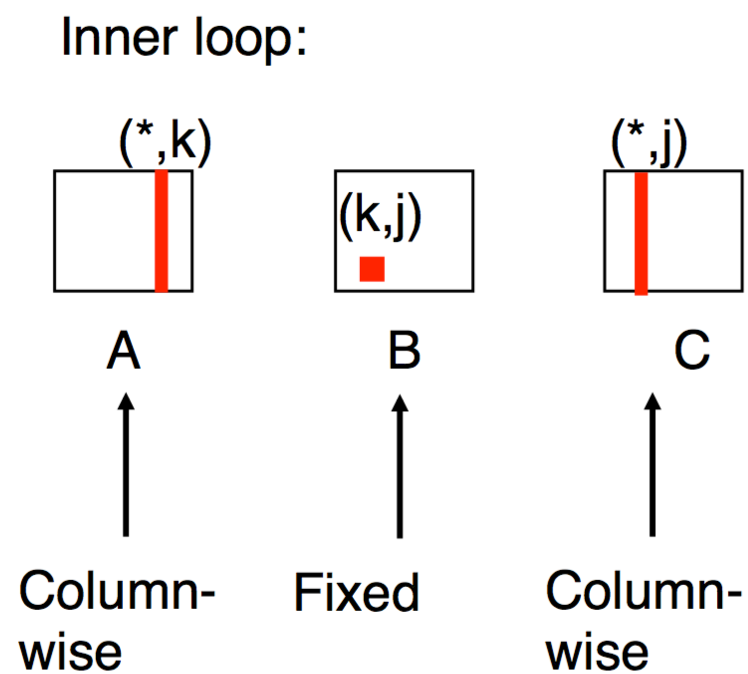


- Misses per Inner Loop Iteration:

A	B	C
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;
  }
}
```



- Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Summary of Misses of Matrix Multiplication

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

```
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;
   }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

```
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];
   }
}
```

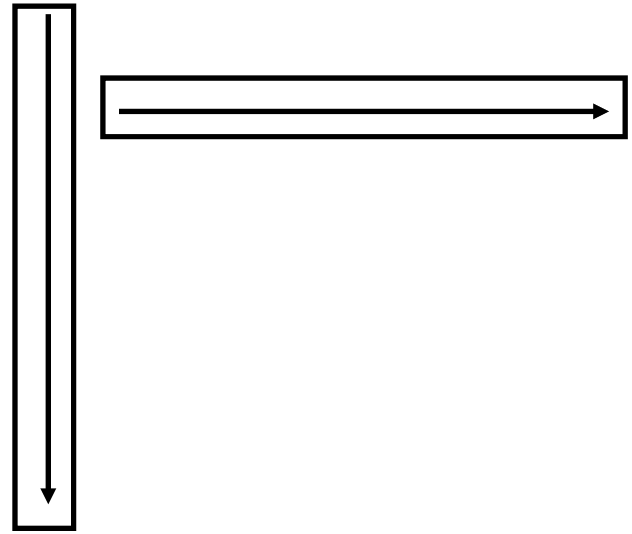
jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

```
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;
   }
}
```

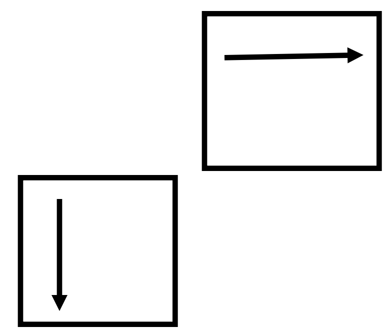
Two Cache Optimization Algorithms:

1. Blocking (Tiling)
2. Cache Oblivious Algorithm

Blocking Example



```
/* Before */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
      {r = 0;
       for (k = 0; k < N; k = k+1){
         r = r + y[i][k]*z[k][j];};
       x[i][j] = r;
      };
```



- Two inner loops:
  – Read all NxN elements of z[]
  – Read N elements of 1 row of y[] repeatedly
  – Write N elements of 1 row of x[]
- Capacity misses a function of N & Cache Size:
  – $2N^3 + N^2$ => (assuming no conflict; otherwise …)
- Idea: compute on BxB submatrix that fits

Array Access in Matrix Multiplication

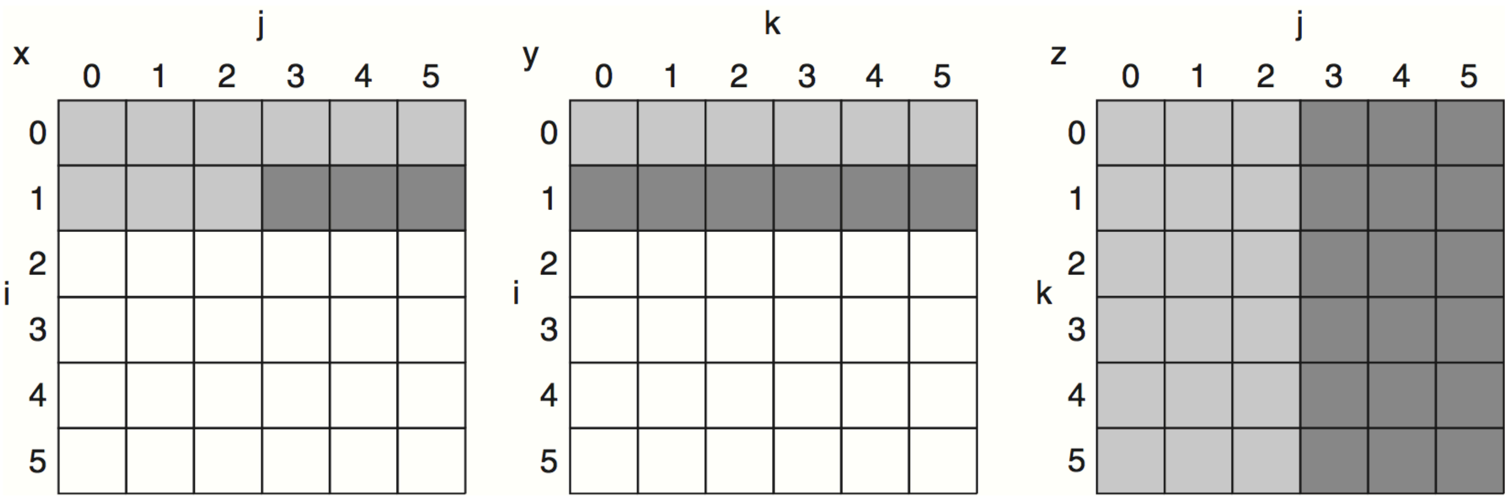


Figure 2.8 A snapshot of the three arrays x, y, and z when $N = 6$ and i = 1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. Compared to Figure 2.9, elements of y and z are read repeatedly to calculate new elements of x. The variables i, j, and k are shown along the rows or columns used to access the arrays.

Array Access for Blocking/Tiling Transformation

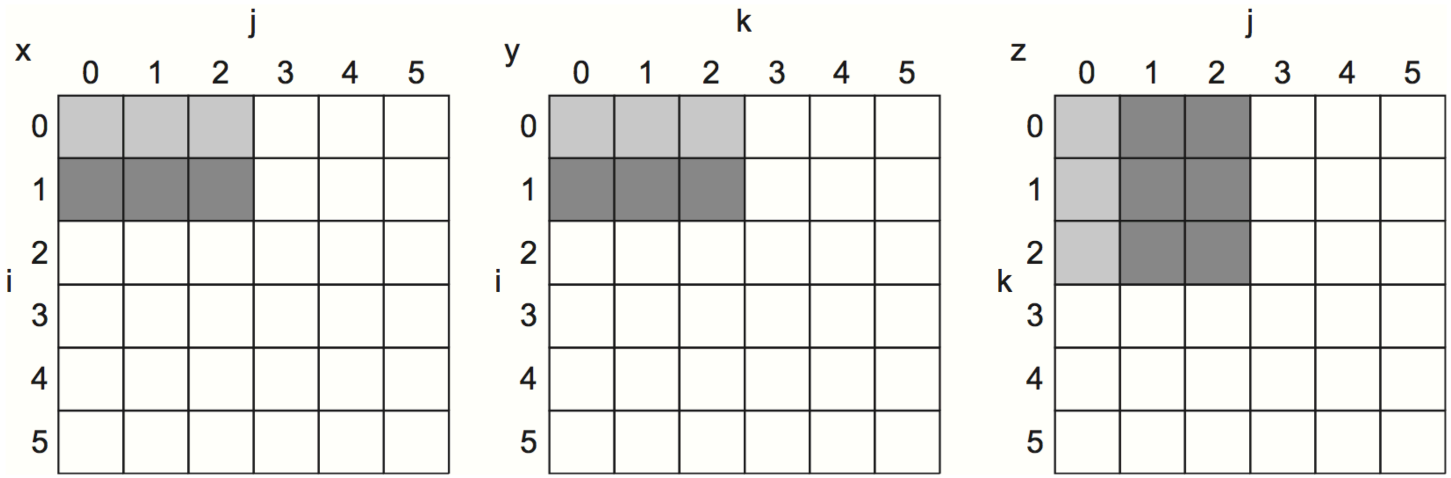


Figure 2.9 The age of accesses to the arrays x, y, and z when *B* = 3. Note that, in contrast to Figure 2.8, a smaller number of elements is accessed.

- https://en.wikipedia.org/wiki/Loop_nest_optimization
- SC17 Invited Talks: Michael Wolfe, Test of Time Award Winner, **https://www.youtube.com/watch?v=oVY8BvFao3M, an ~1 hour talk without slide**

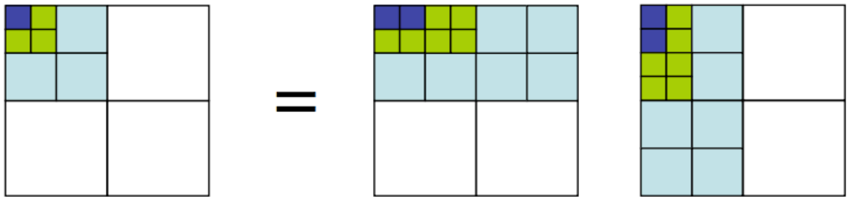
Blocking Example

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
     for (i = 0; i < N; i = i+1)
         for (j = jj; j < min(jj+B-1,N); j = j+1)
          {r = 0;
           for (k = kk; k < min(kk+B-1,N); k = k+1) {
             r = r + y[i][k]*z[k][j];};
           x[i][j] = x[i][j] + r;
          };
```

- B called *Blocking Factor*
- Capacity misses from $2N^3 + N^2$ to $2N^3/B + N^2$
- Reduce conflict misses too?

Cache Oblivious Algorithm

(optimal) Cache-Oblivious Matrix Multiply



C $m \times p$ $\quad$ A $m \times n$ $\quad$ B $n \times p$

divide and conquer:

divide C into 4 blocks

compute block multiply recursively

achieves optimal $\Theta(n^3/\sqrt{Z})$ cache complexity

https://en.wikipedia.org/wiki/Cache-oblivious_algorithm

Cache-Oblivious Algorithms: http://supertech.csail.mit.edu/papers/FrigoLePr99.pdf

C Implementation of Cache Oblivious Algorithm

```c
/* C = C + AB, where A is m x n, B is n x p, and C is m x p, in
   row-major order.  Actually, the physical size of A, B, and C
   are m x fdA, n x fdB, and m x fdC, but only the first n/p/p
   columns are used, respectively. */
void add_matmul_rec(const double *A, const double *B, double *C,
                    int m, int n, int p, int fdA, int fdB, int fdC)
{
     if (m+n+p <= 48) { /* <= 16x16 matrices "on average" */
          int i, j, k;
          for (i = 0; i < m; ++i)
               for (k = 0; k < p; ++k) {
                    double sum = 0;
                    for (j = 0; j < n; ++j)
                         sum += A[i*fdA +j] * B[j*fdB + k];
                    C[i*fdC + k] += sum;
               }
     }
     else { /* divide and conquer */
          int m2 = m/2, n2 = n/2, p2 = p/2;

          add_matmul_rec(A, B, C, m2, n2, p2, fdA, fdB, fdC);
          add_matmul_rec(A+n2, B+n2*fdB, C, m2, n-n2, p2, fdA, fdB, fdC);

          add_matmul_rec(A, B+p2, C+p2, m2, n2, p-p2, fdA, fdB, fdC);
          add_matmul_rec(A+n2, B+p2+n2*fdB, C, m2, n-n2, p-p2, fdA, fdB, fdC);

          add_matmul_rec(A+m2*fdA, B, C+m2*fdC, m-m2, n2, p2, fdA, fdB, fdC);
          add_matmul_rec(A+m2*fdA+n2, B+n2*fdB, C+m2*fdC, m-m2, n-n2, p2, fdA, fdB, fdC);

          add_matmul_rec(A+m2*fdA, B+p2, C+m2*fdC+p2, m-m2, n2, p-p2, fdA, fdB, fdC);
          add_matmul_rec(A+m2*fdA+n2, B+p2+n2*fdB, C+m2*fdC, m-m2, n-n2, p-p2, fdA, fdB, fdC);
     }
}

void matmul_rec(const double *A, const double *B, double *C,
                int m, int n, int p)
{
     memset(C, 0, sizeof(double) * m*p);
     add_matmul_rec(A, B, C, m, n, p, n, p, p);
}
```

note: base case is ~16×16

recursing down to 1×1
would kill performance
(1 function call per element,
no register re-use)

dividing *C* into 4
— note that, instead, for very non-square matrices, we might want to divide *C* in 2 along longest axis