Writing cache friendly code

Two major rules:

- Repeated references to data are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)

Locality Example (1)

Claim: **Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.**

Question: **Which of these functions has good locality?**

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

Locality Example (1)

Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

Question: Which of these functions has good locality?

Cold cache, 4-byte words, 4-word cache blocks(16B), miss rate = ???

```
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 = ¼ = 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%

Layout of C Arrays in Memory

C arrays allocated in contiguous memory locations with addresses ascending with the array index:

```
int32_t A[10] = {0, 1, 2, 3, 4, ..., 8, 9};
```

7FFF99702320	0
7FFF99702324	1
7FFF99702328	2
7FFF9970232C	3
7FFF99702330	4
...	...
7FFF99702340	8
7FFF99702344	9

Two-dimensional Arrays in C

In C, a two-dimensional array is an array of arrays:

```
A[0]
A[1]
A[2]
```

```
int32_t A[3][5] = {
    { 0,  1,  2,  3,  4},
    {10, 11, 12, 13, 14},
    {20, 21, 22, 23, 24}
};
```

4*5=20

In fact, if we print the values as pointers, we see something like this:

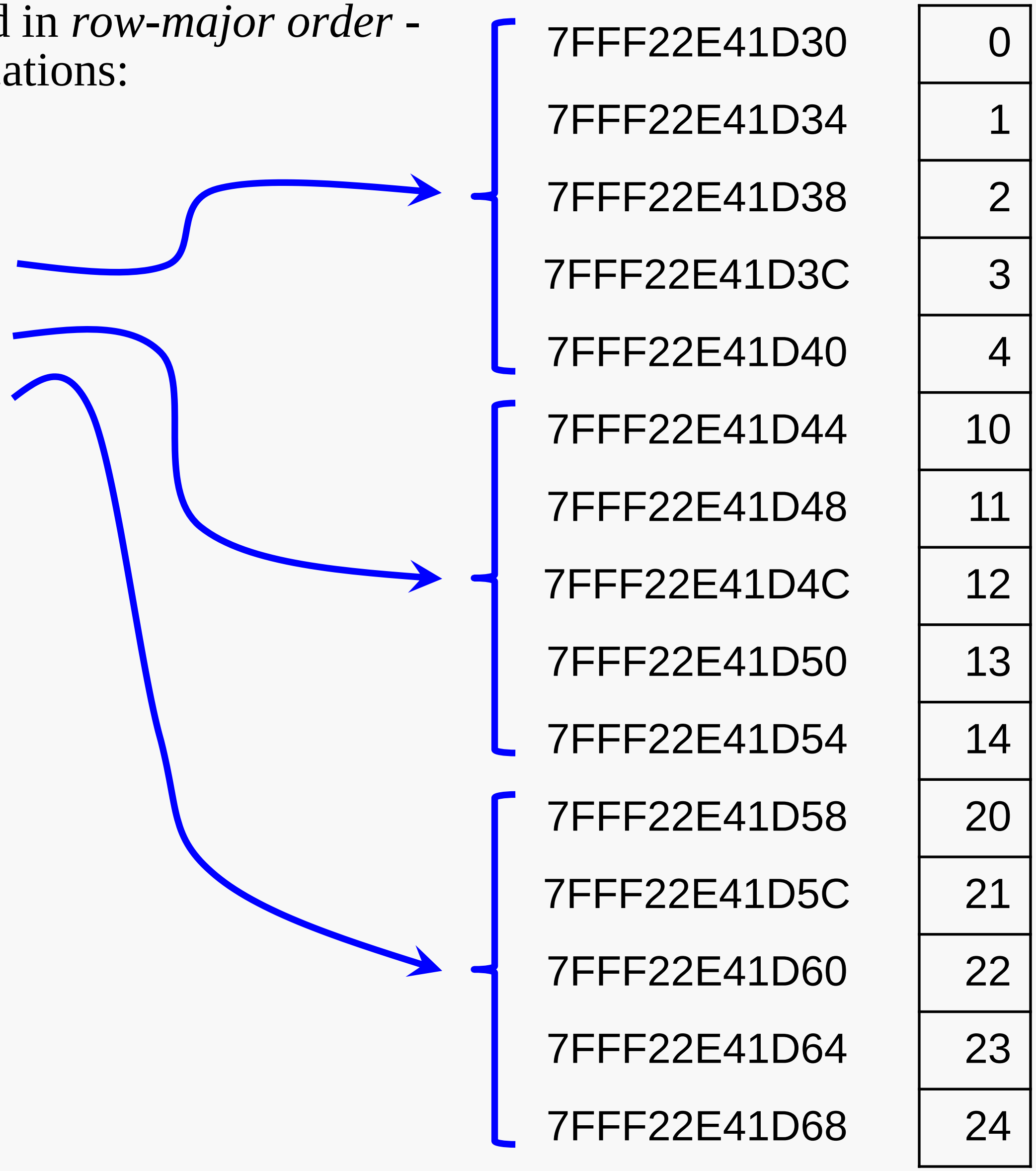
```
A:      0x7fff22e41d30

A[0]:   0x7fff22e41d30
                         0x14
A[1]:   0x7fff22e41d44
                         0x14
A[2]:   0x7fff22e41d58
```

Layout of C Arrays in Memory

Two-dimensional C arrays allocated in *row-major order* - each row in contiguous memory locations:

```
int32_t A[3][5] =
 { { 0,  1,  2,  3,  4},
   {10, 11, 12, 13, 14},
   {20, 21, 22, 23, 24}
 };
```



Layout of C Arrays in Memory


```
int32_t A[3][5] =
 { { 0,  1,  2,  3,  4},
   {10, 11, 12, 13, 14},
   {20, 21, 22, 23, 24},
 };
```

Stepping through columns in one row:

```
    for (i = 0; i < 3; i++)
        for (j = 0; j < 5; j++)
            sum += A[i][j];
```

- accesses successive elements in memory
- if cache block size B > 4 bytes, exploit spatial locality compulsory miss rate = 4 bytes / B

	7FFF22E41D30	0
	7FFF22E41D34	1
i = 0	7FFF22E41D38	2
	7FFF22E41D3C	3
	7FFF22E41D40	4
	7FFF22E41D44	10
	7FFF22E41D48	11
i = 1	7FFF22E41D4C	12
	7FFF22E41D50	13
	7FFF22E41D54	14
	7FFF22E41D58	20
	7FFF22E41D5C	21
i = 2	7FFF22E41D60	22
	7FFF22E41D64	23
	7FFF22E41D68	24

Layout of C Arrays in Memory

```
int32_t A[3][5] =
 { { 0,  1,  2,  3,  4},
   {10, 11, 12, 13, 14},
   {20, 21, 22, 23, 24},
 };
```

j = 0

j = 1

7FFF22E41D30	0
7FFF22E41D34	1
7FFF22E41D38	2
7FFF22E41D3C	3
7FFF22E41D40	4
7FFF22E41D44	10
7FFF22E41D48	11
7FFF22E41D4C	12
7FFF22E41D50	13
7FFF22E41D54	14
7FFF22E41D58	20
7FFF22E41D5C	21
7FFF22E41D60	22
7FFF22E41D64	23
7FFF22E41D68	24

Stepping through rows in one column:

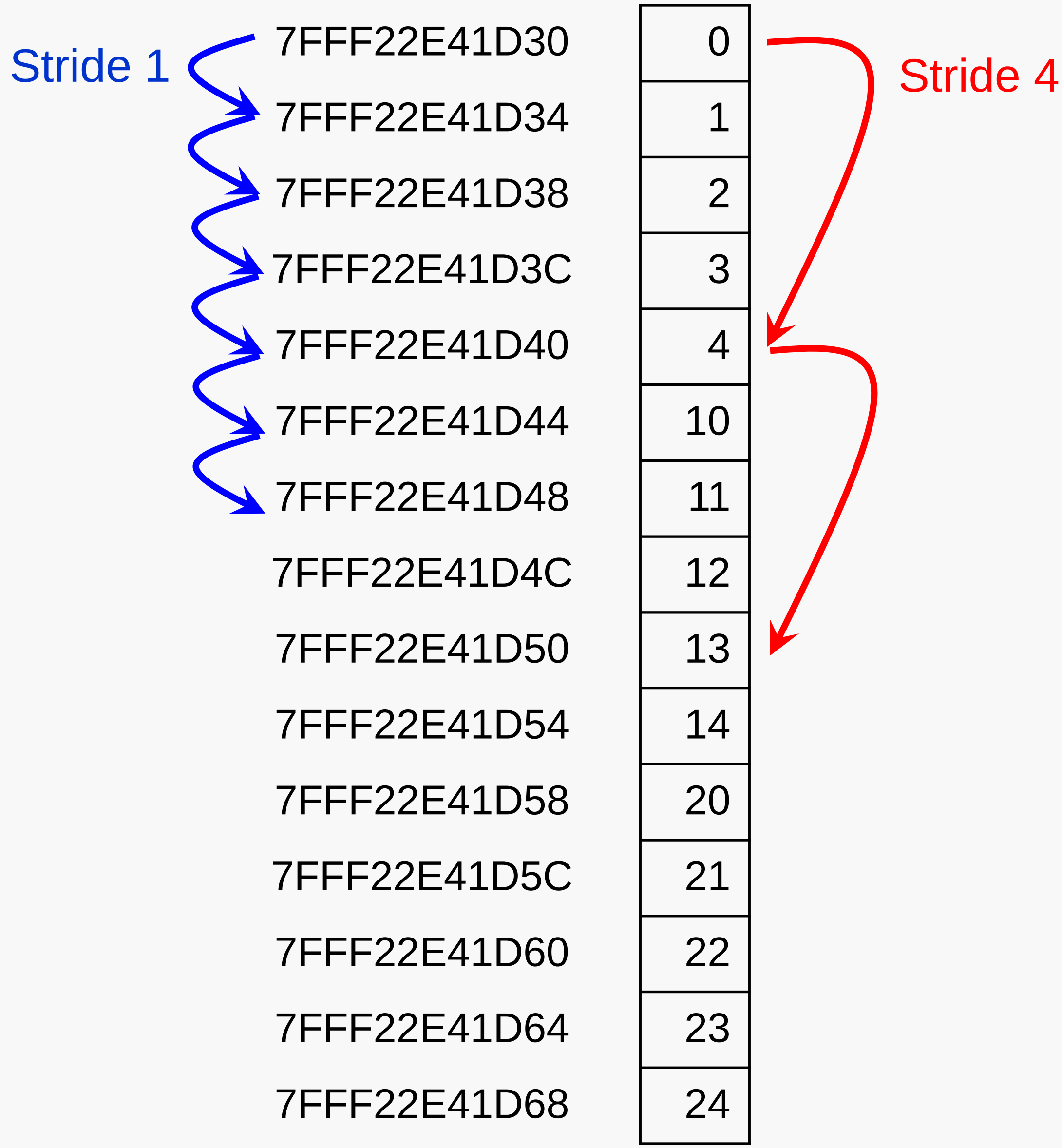
```
for (j = 0; i < 5; i++)
    for (i = 0; i < 3; i++)
        sum += a[i][j];
```

accesses distant elements

no spatial locality!

compulsory miss rate = 1 (i.e. 100%)

Stride and Array Accesses



Writing Cache Friendly Code

Repeated references to variables are good (temporal locality)

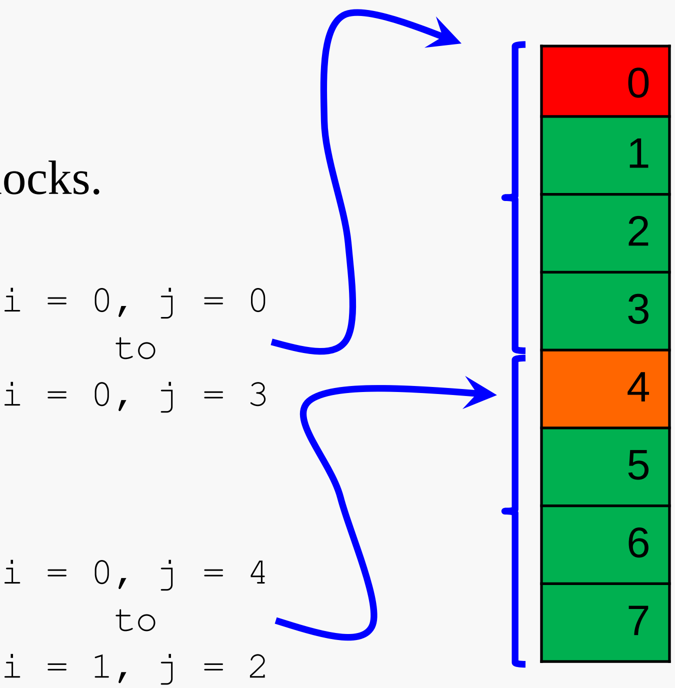
Stride-1 reference patterns are good (spatial locality)

Assume an initially-empty cache with 16-byte cache blocks.

```
int sumarrayrows(int a[M][N]) {

  int row, col, sum = 0;

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



Miss rate = 1/4 = 25%

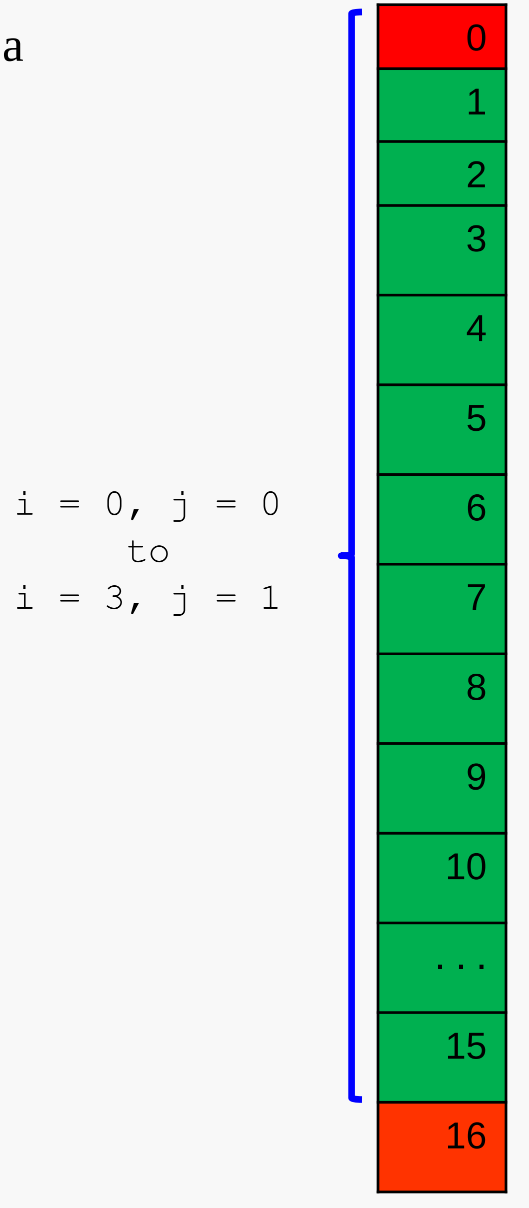
Writing Cache Friendly Code

Consider the previous slide, but assume that the cache uses a block size of 64 bytes instead of 16 bytes..

```
int sumarrayrows(int a[M][N]) {

   int row, col, sum = 0;

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



Miss rate = 1/16 = 6.25%

Writing Cache Friendly Code

"Skipping" accesses down the rows of a column do not provide good locality:

```
int sumarraycols(int a[M][N]) {

   int row, col, sum = 0;

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

Miss rate = 100%

(That's actually somewhat pessimistic... depending on cache geometry.)

Layout of C Arrays in Memory

It's easy to write an array traversal and see the addresses at which the array elements are stored:

```
int A[5] = {0, 1, 2, 3, 4};

for (i = 0; i < 5; i++)
   printf("%d:  %p\n",
            i,  &A[i]);
```

We see there that for a 1D array, the index varies in a stride-1 pattern.

```
i    address
-----------
0:    28ABE0
1:    28ABE4
2:    28ABE8
3:    28ABEC
4:    28ABF0
```

stride-1 : addresses differ by the size of an array cell (4 bytes, here)

Layout of C Arrays in Memory


```
int B[3][5] = { ... };
for (i = 0; i < 3; i++)
    for (j = 0; j < 5; j++)
        printf("%d %3d:  %p\n",
                     i, j,   &B[i][j]);
```

We see that for a 2D array, the <u>second</u> index varies in a stride-1 pattern.

```
i-j order:

i   j     address
----------------
0   0:    28ABA4
0   1:    28ABA8
0   2:    28ABAC
0   3:    28ABB0
0   4:    28ABB4
1   0:    28ABB8
1   1:    28ABBC
1   2:    28ABC0
```

stride-1

But the <u>first</u> index does not vary in a stride-1 pattern.

```
j-i order:

i   j     address
----------------
0   0:    28ABA4
1   0:    28ABB8
2   0:    28ABCC
0   1:    28ABA8
1   1:    28ABBC
2   1:    28ABD0
0   2:    28ABAC
1   2:    28ABC0
```

stride-5 (0x14/4)

3D Arrays in C


```
int32_t A[2][3][5] = {
    { {   0,    1,    2,    3,    4},
      {  10,   11,   12,   13,   14},
      {  20,   21,   22,   23,   24}},
    { {   0,    1,    2,    3,    4},
      {110, 111, 112, 113, 114},
      {220, 221, 222, 223, 224}}
   };
```

1,0,0	1,0,1	1,0,2	1,0,3	1,0,4
1,1,0	1,1,1	1,1,2	1,1,3	1,1,4
1,2,0	1,2,1	1,2,2	1,2,3	1,2,4

pages

rows

0,0,0	0,0,1	0,0,2	0,0,3	0,0,4
0,1,0	0,1,1	0,1,2	0,1,3	0,1,4
0,2,0	0,2,1	0,2,2	0,2,3	0,2,4

columns

Locality Example (2)

Question: Can you permute the loops so that the function scans the 3D array `a[][][]` with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sumarray3d(int a[N][N][N]) {

    int row, col, page, sum = 0;

    for (row = 0; row < N; row++)
        for (col = 0; col < N; col++)
            for (page = 0; page < N; page++)
                sum += a[page][row][col];
    return sum;
}
```

Layout of C Arrays in Memory

```
int C[2][3][5] = { ... };

for (i = 0; i < 2; i++)
   for (j = 0; j < 3; j++)
      for (k = 0; k < 5; k++)
         printf("%3d  %3d  %3d: %p\n",
                    i, j, k, &C[i][j][k]);
```

We see that for a 3D array, the <u>third</u> index varies in a stride-1 pattern:

```
i-j-k order:

i  j  k     address
------------------
0  0  0:    28CC24
0  0  1:    28CC28
0  0  2:    28CC2C
0  0  3:    28CC30
0  0  4:    28CC34
0  1  0:    28CC38
0  1  1:    28CC3C
0  1  2:    28CC40
```

0x4
0x4
0x4

Layout of C Arrays in Memory

```
int C[2][3][5] = { ... };

for (i = 0; i < 2; i++)
   for (j = 0; j < 3; j++)
      for (k = 0; k < 5; k++)
         printf("%3d  %3d  %3d: %p\n",
                  i, j, k, &C[i][j][k]);
```

We see that for a 3D array, the <u>third</u> index varies in a stride-1 pattern:

```
i-j-k order:

i  j  k     address
------------------
0  0  0:    28CC24 } 0x4
0  0  1:    28CC28 } 0x4
0  0  2:    28CC2C } 0x4
0  0  3:    28CC30
0  0  4:    28CC34
0  1  0:    28CC38
0  1  1:    28CC3C
0  1  2:    28CC40
```

But… if we change the order of access, we no longer have a stride-1 pattern:

```
k-j-i order:

i  j  k     address
------------------
0  0  0:    28CC24 } 0x3C
1  0  0:    28CC60 } 0x28
0  1  0:    28CC38 } 0x3C
1  1  0:    28CC74
0  2  0:    28CC4C
1  2  0:    28CC88
0  0  1:    28CC28
1  0  1:    28CC64
```

Layout of C Arrays in Memory

```
int C[2][3][5] = { ... };

for (i = 0; i < 2; i++)
   for (j = 0; j < 3; j++)
      for (k = 0; k < 5; k++)
         printf("%3d  %3d  %3d: %p\n",
                   i, j, k, &C[i][j][k]);
```

We see that for a 3D array, the <u>third</u> index varies in a stride-1 pattern:

But… if we change the order of access, we no longer have a stride-1 pattern:

```
i-j-k order:

i  j  k     address
------------------
0  0  0:    28CC24   0x4
0  0  1:    28CC28   0x4
0  0  2:    28CC2C   0x4
0  0  3:    28CC30
0  0  4:    28CC34
0  1  0:    28CC38
0  1  1:    28CC3C
0  1  2:    28CC40
```

```
k-j-i order:

i  j  k     address
------------------
0  0  0:    28CC24   0x3C
1  0  0:    28CC60   0x28
0  1  0:    28CC38   0x3C
1  1  0:    28CC74
0  2  0:    28CC4C
1  2  0:    28CC88
0  0  1:    28CC28
1  0  1:    28CC64
```

```
i-k-j order:

i  j  k     address
------------------
0  0  0:    28CC24   0x14
0  1  0:    28CC38   0x14
0  2  0:    28CC4C   0x14
0  0  1:    28CC28
0  1  1:    28CC3C
0  2  1:    28CC50
0  0  2:    28CC2C
0  1  2:    28CC40
```

Locality Example (2)

Question: Can you permute the loops so that the function scans the 3D array `a[]` with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sumarray3d(int a[N][N][N]) {

    int i, j, k, sum = 0;

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

This code does not yield good locality at all.

The inner loop is varying the <u>first</u> index, worst case!

Locality Example (3)

Question: Which of these two exhibits better spatial locality?

```
// struct of arrays
struct soa {
  float *x;
  float *y;
  float *z;
  float *r;
};

compute_r(struct soa s) {
  for (i = 0; …) {
    s.r[i] = s.x[i] * s.x[i]
           + s.y[i] * s.y[i]
           + s.z[i] * s.z[i];
  }
}
```

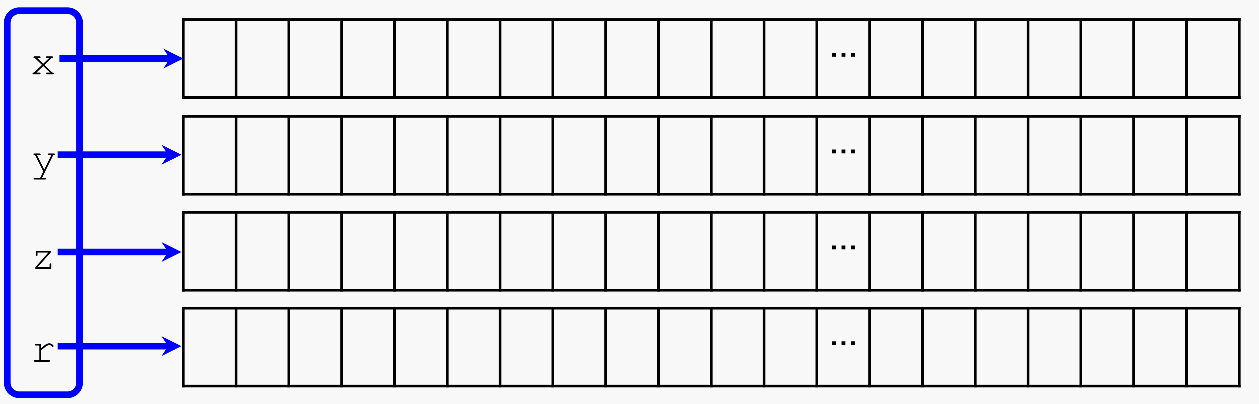
```
// array of structs
struct aos {
  float x;
  float y;
  float z;
  float r;
};

compute_r(struct aos *s) {
  for (i = 0; …) {
    s[i].r = s[i].x * s[i].x
           + s[i].y * s[i].y
           + s[i].z * s[i].z;
  }
}
```

For the following discussions assume a cache block size of 32 bytes, and that the cache is not capable of holding all the blocks of the relevant structure at once.

Locality Example (3)


```
// struct of arrays
struct soa {
  float *x;
  float *y;
  float *z;
  float *r;
};
struct soa s;
s.x = malloc(1000 * sizeof(float));
...
```

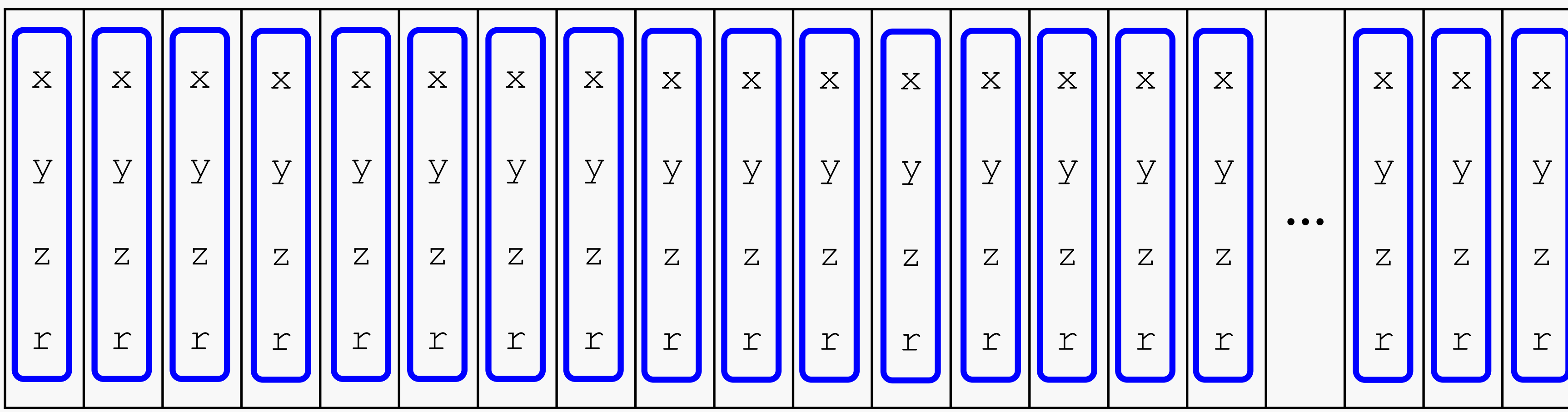


16 bytes **4 bytes per cell, 1000 cells per array**

Locality Example (3)

```
// array of structs
struct aos {
  float x;
  float y;
  float z;
  float r;
};

struct aos s[1000];
```

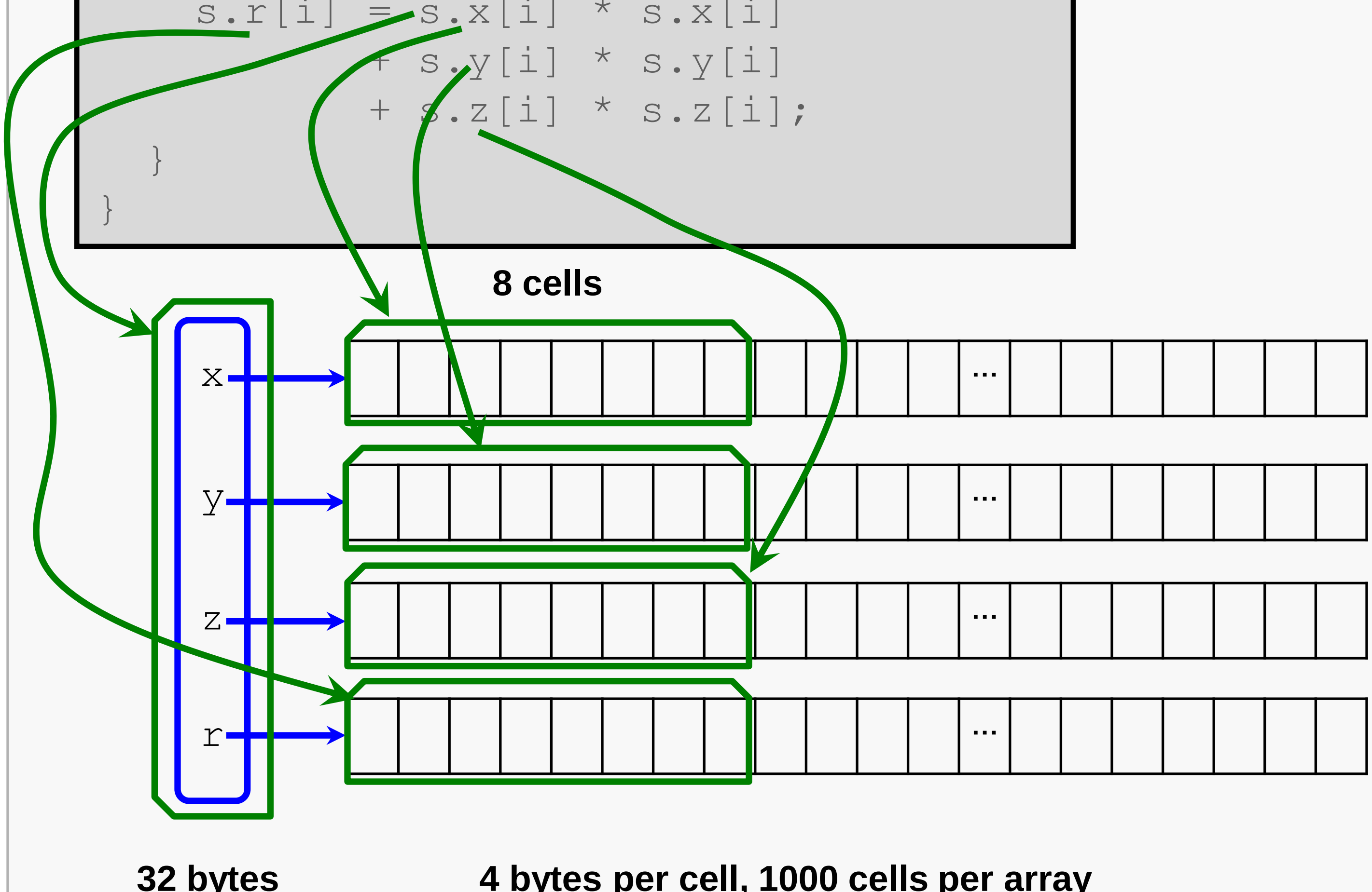


16 bytes per cell, 1000 cells

Locality Example (3)

Describe the locality exhibited by this algorithm:

```
// struct of arrays
compute_r(struct soa s) {
  for (int i = 0; i < 1000; i++) {
    s.r[i] = s.x[i] * s.x[i]
           + s.y[i] * s.y[i]
           + s.z[i] * s.z[i];
  }
}
```




```
s.x[0]   miss
s.y[0]   miss
s.z[0]   miss
s.r[0]   miss
s.x[1]   hit
s.y[1]   hit
s.z[1]   hit
s.r[1]   hit
. . .
s.x[7]   hit
s.y[7]   hit
s.z[7]   hit
s.r[7]   hit
s.x[8]   miss
s.y[8]   miss
s.z[8]   miss
s.r[8]   miss
```

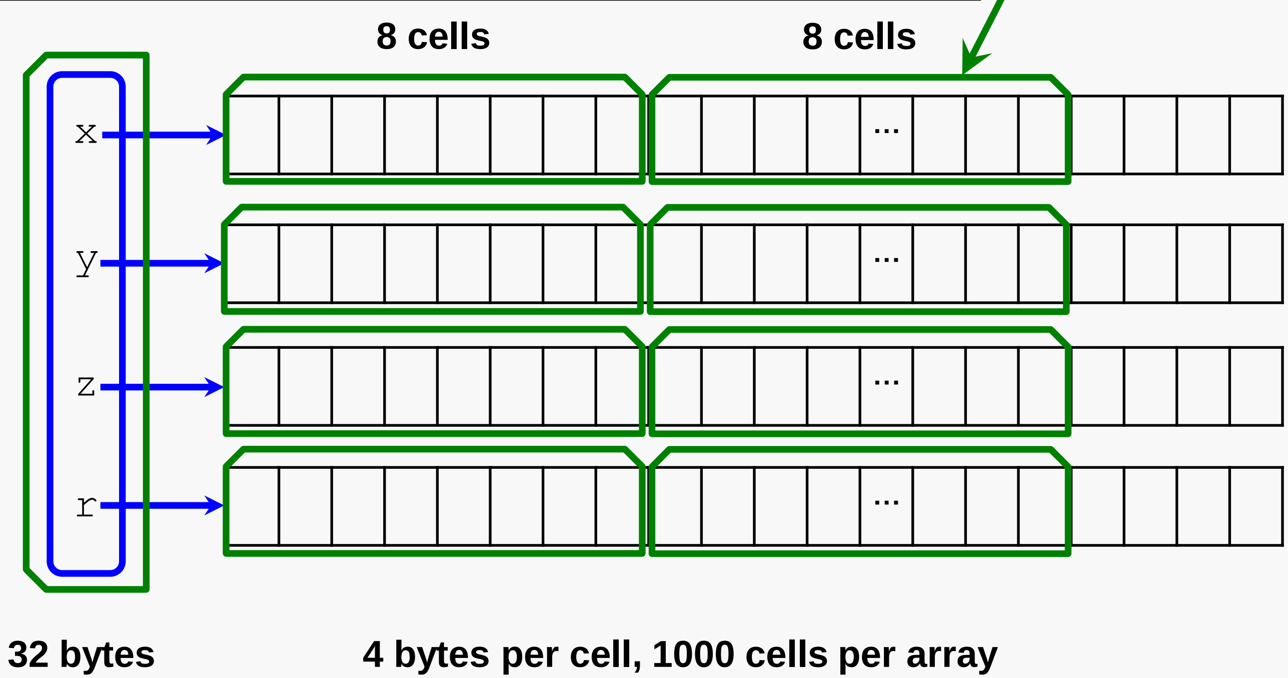
Locality Example (3)

Describe the locality exhibited by this algorithm:

```
// struct of arrays
compute_r(struct soa s) {
  for (int i = 0; i < 1000; i++) {
    s.r[i] = s.x[i] * s.x[i]
           + s.y[i] * s.y[i]
           + s.z[i] * s.z[i];
  }
}
```

s.x[8]	miss
s.y[8]	miss
s.z[8]	miss
s.r[8]	miss
s.x[9]	hit
s.y[9]	hit
s.z[9]	hit
s.r[9]	hit

. . .



For the arrays:

Misses = 4*1*125

Hits = 4*7*125

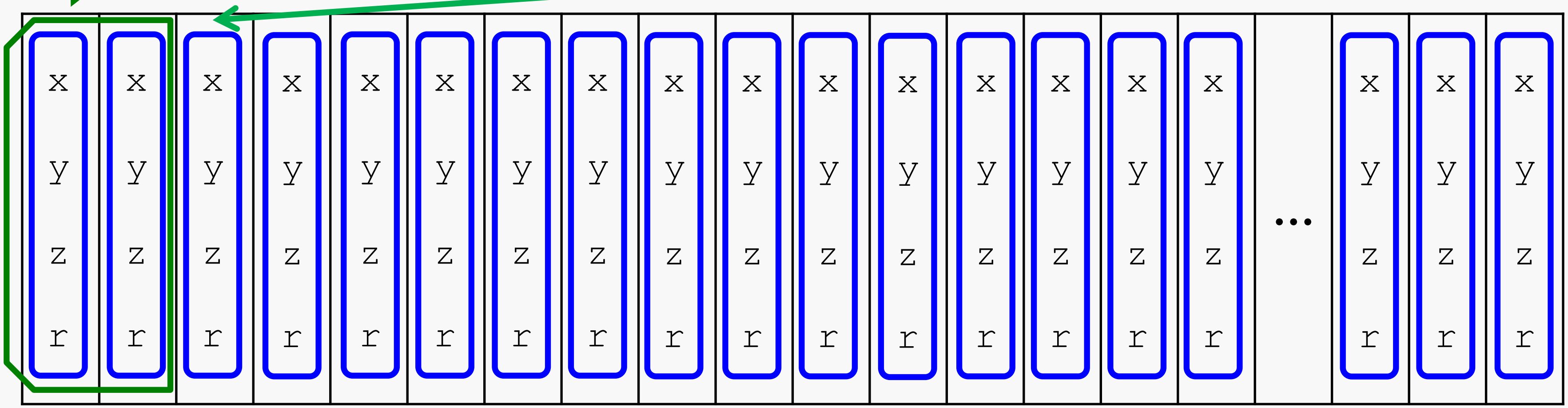
Hit rate = 87.5%

Locality Example (3)

Describe the locality exhibited by this algorithm:

```
// array of structs
compute_r(struct aos *s) {
  for (int i = 0; i < 1000; i++) {
    s[i].r = s[i].x * s[i].x
           + s[i].y * s[i].y
           + s[i].z * s[i].z;
  }
}
```

s[0].x	miss
s[0].y	hit
s[0].z	hit
s[0].r	hit
s[1].x	hit
s[1].y	hit
s[1].z	hit
s[1].r	hit
s[2].x	miss

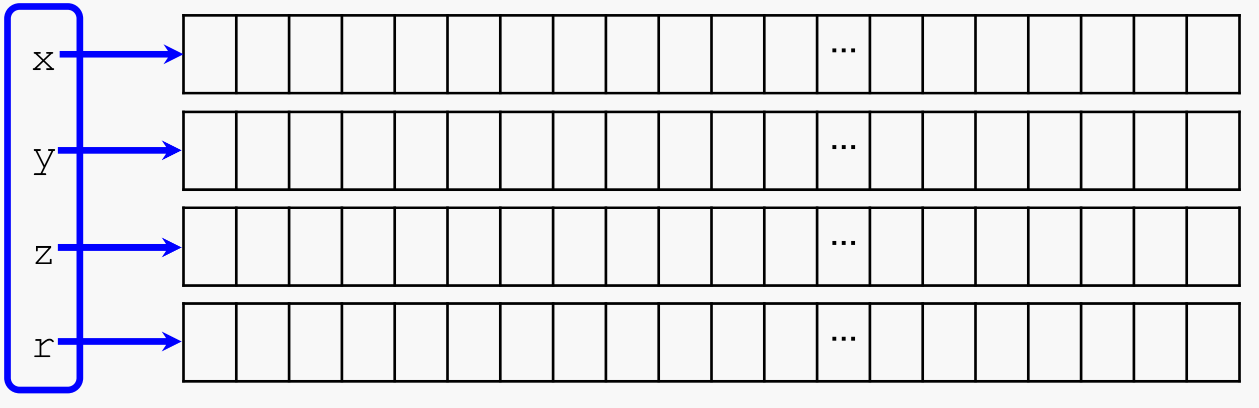


Hit rate: 7/8 or 87.5%

Locality Example (4)

Describe the locality exhibited by this algorithm:

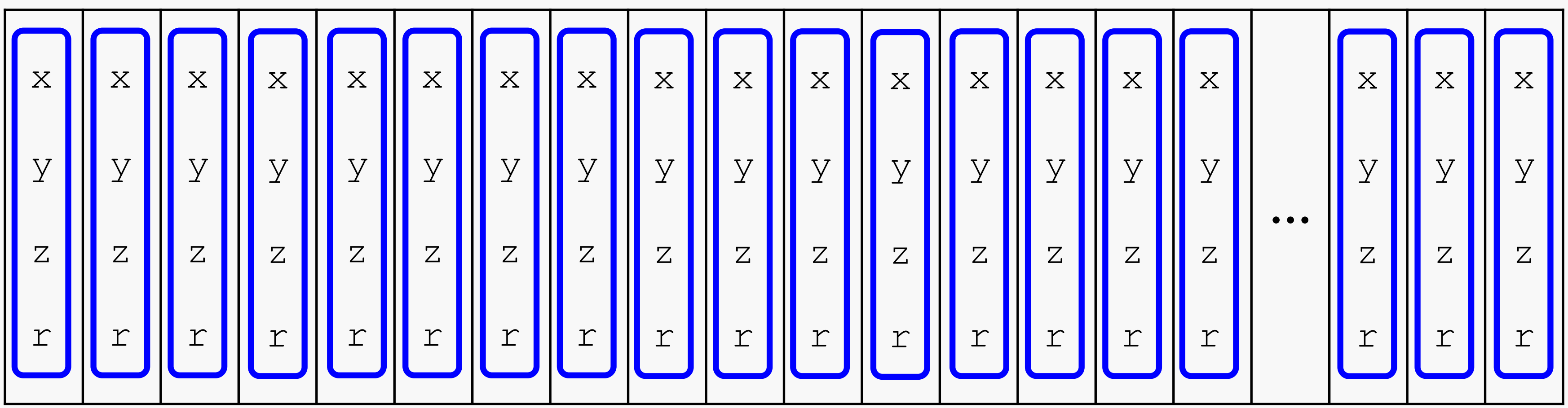
```
// struct of arrays
sum_r(struct soa s) {
  sum = 0;
  for (int i = 0; i < 1000; i++) {
    sum += s.r[i];
  }
}
```



Locality Example (4)

Describe the locality exhibited by this algorithm:

```
// array of structs
sum_r(struct aos *s) {
  sum = 0;
  for (int i = 0; i < 1000; i++) {
    sum += s[i].r;
  }
}
```



Writing Cache Friendly Code

Make the common case 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: Our qualitative notion of locality is quantified through our understanding of cache memories.

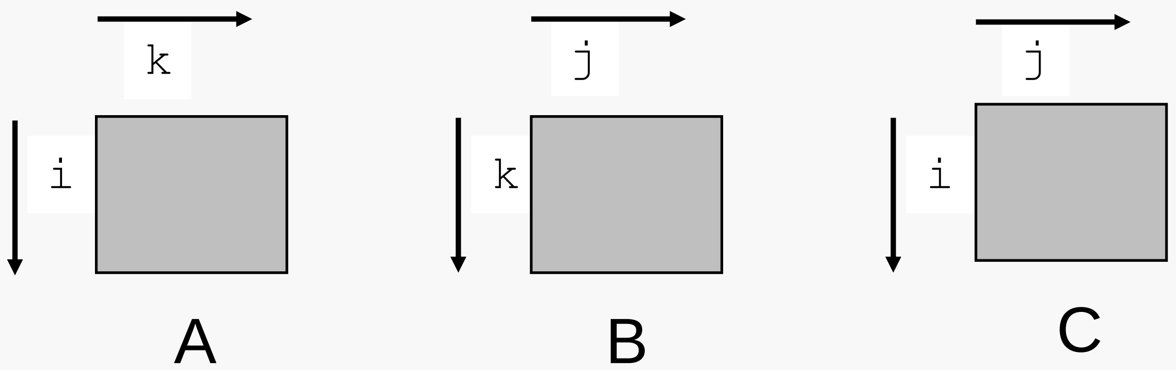
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32B (big enough for four 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 Example

Description:

- Multiply N x N matrices
- O(N^3) total operations
- N reads per source element
- N values summed per destination

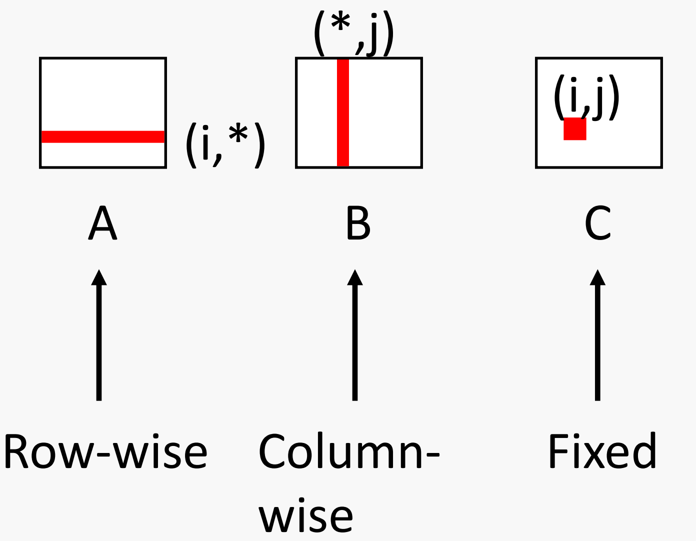
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable `sum` *held in register*

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:



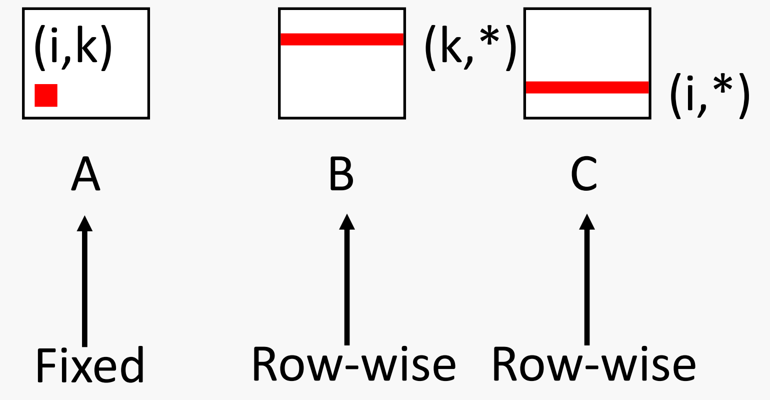
Misses per inner loop iteration:

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:



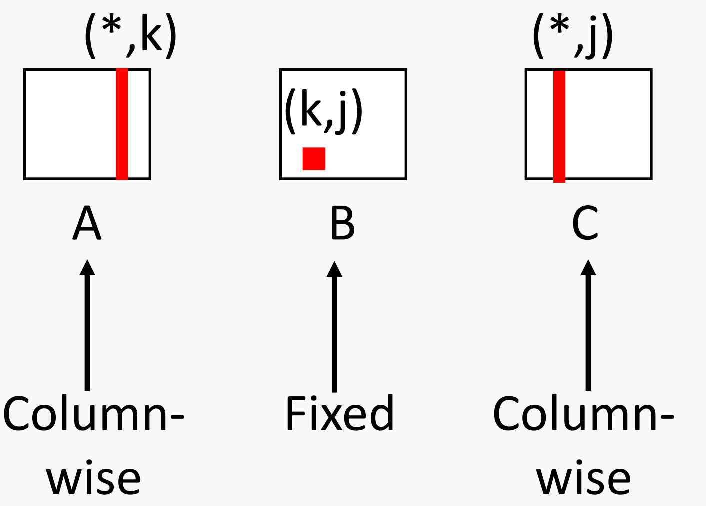
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (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:



Misses per inner loop iteration:

A	B	C
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):

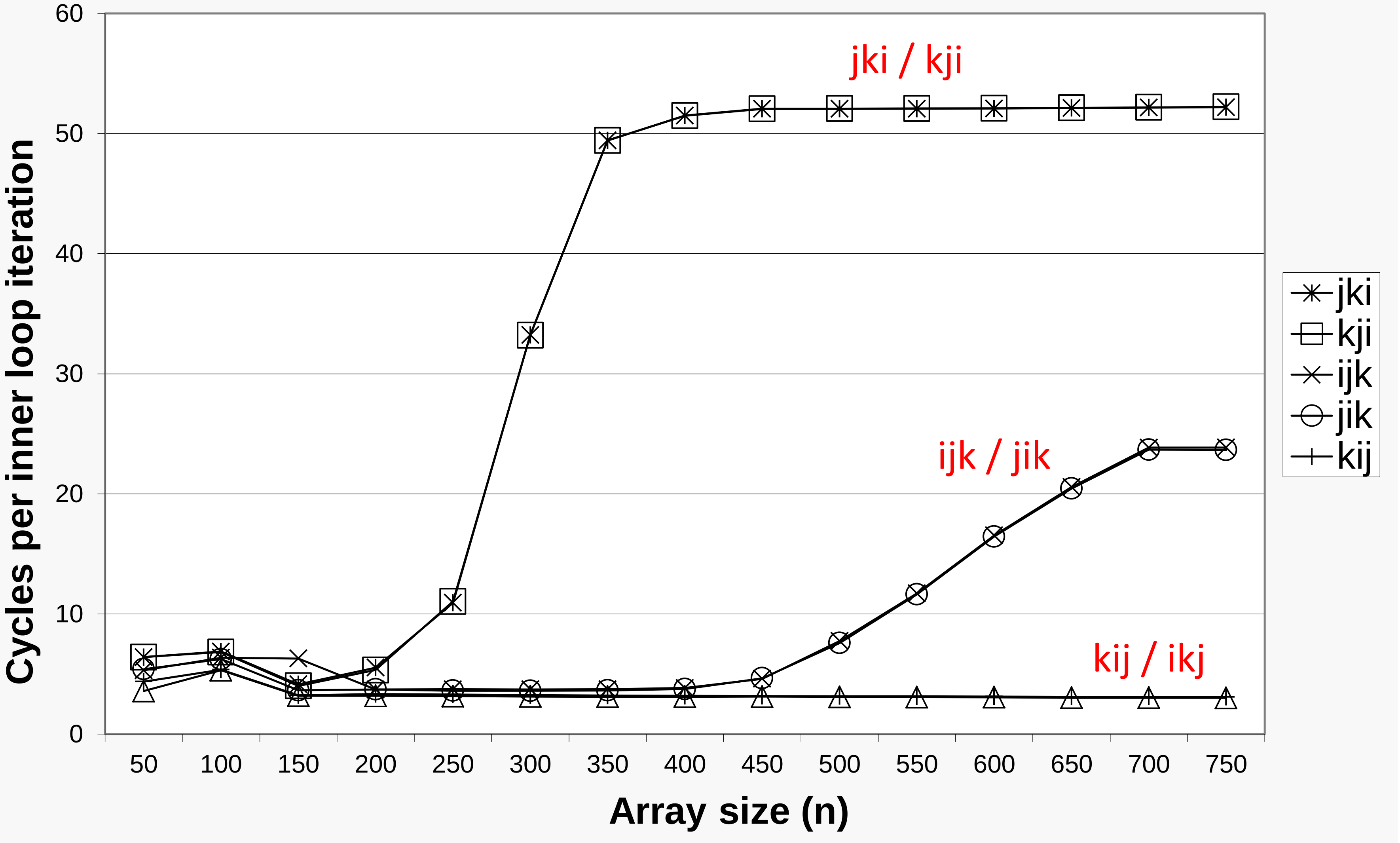
- 2 loads, 1 store
- misses/iter = 0.5

```
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

Core i7 Matrix Multiply Performance



Concluding Observations

- 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.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)