CASPER

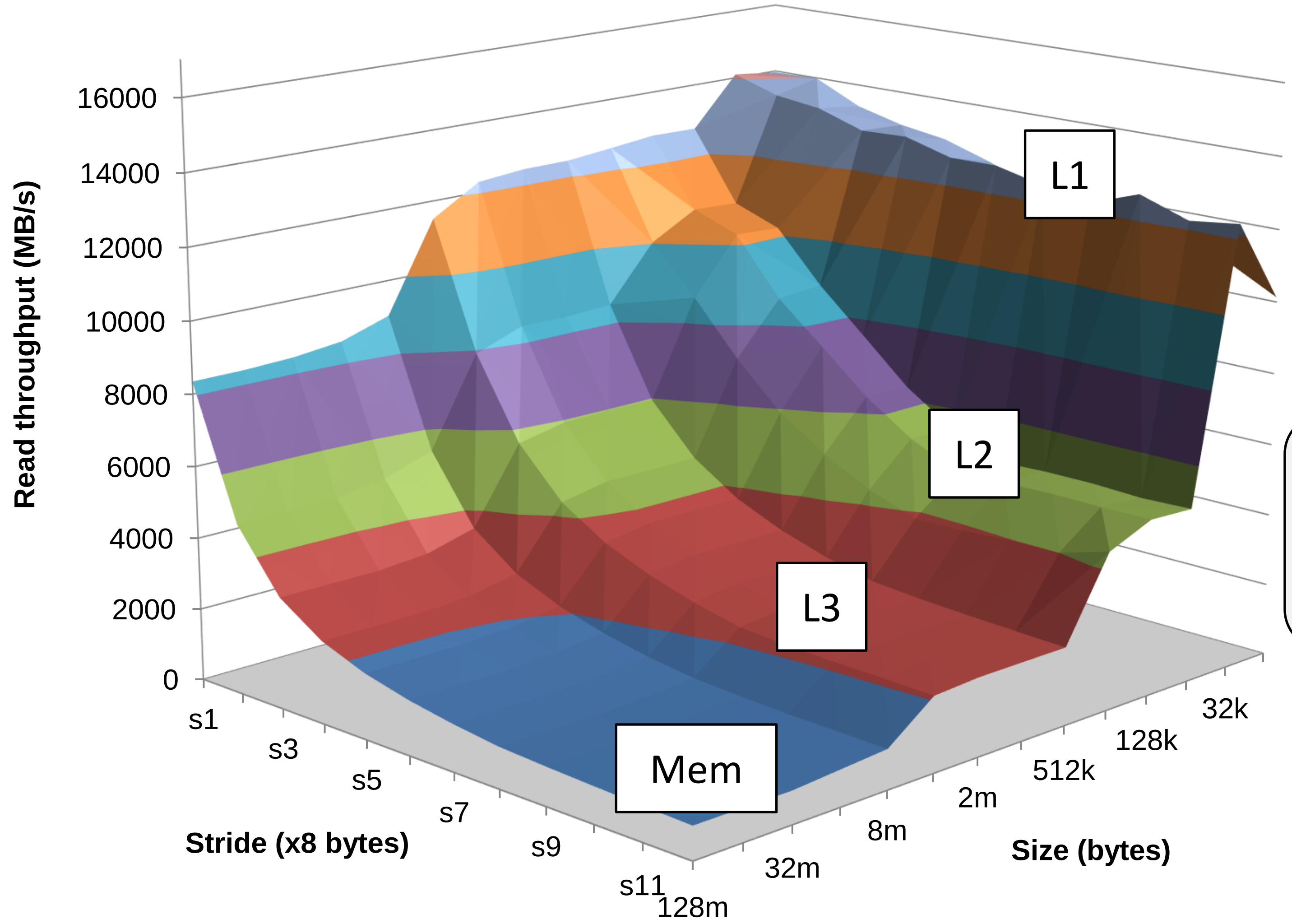
CS305: Computer Architecture

Caches-V

https://www.cse.iitb.ac.in/~biswa/courses/CS305/main.html

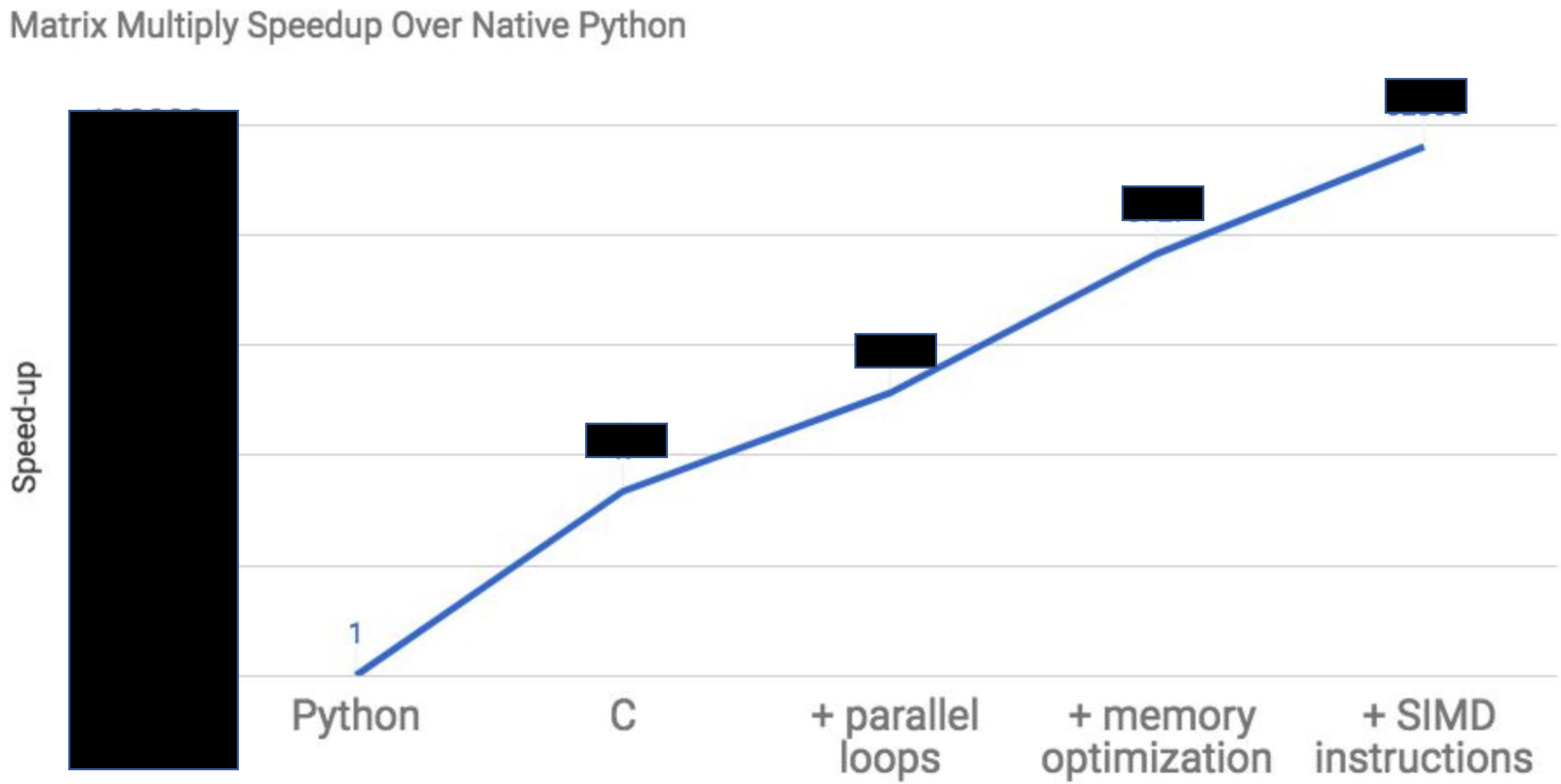
https://www.cse.iitb.ac.in/~biswa/

Memory Mountain

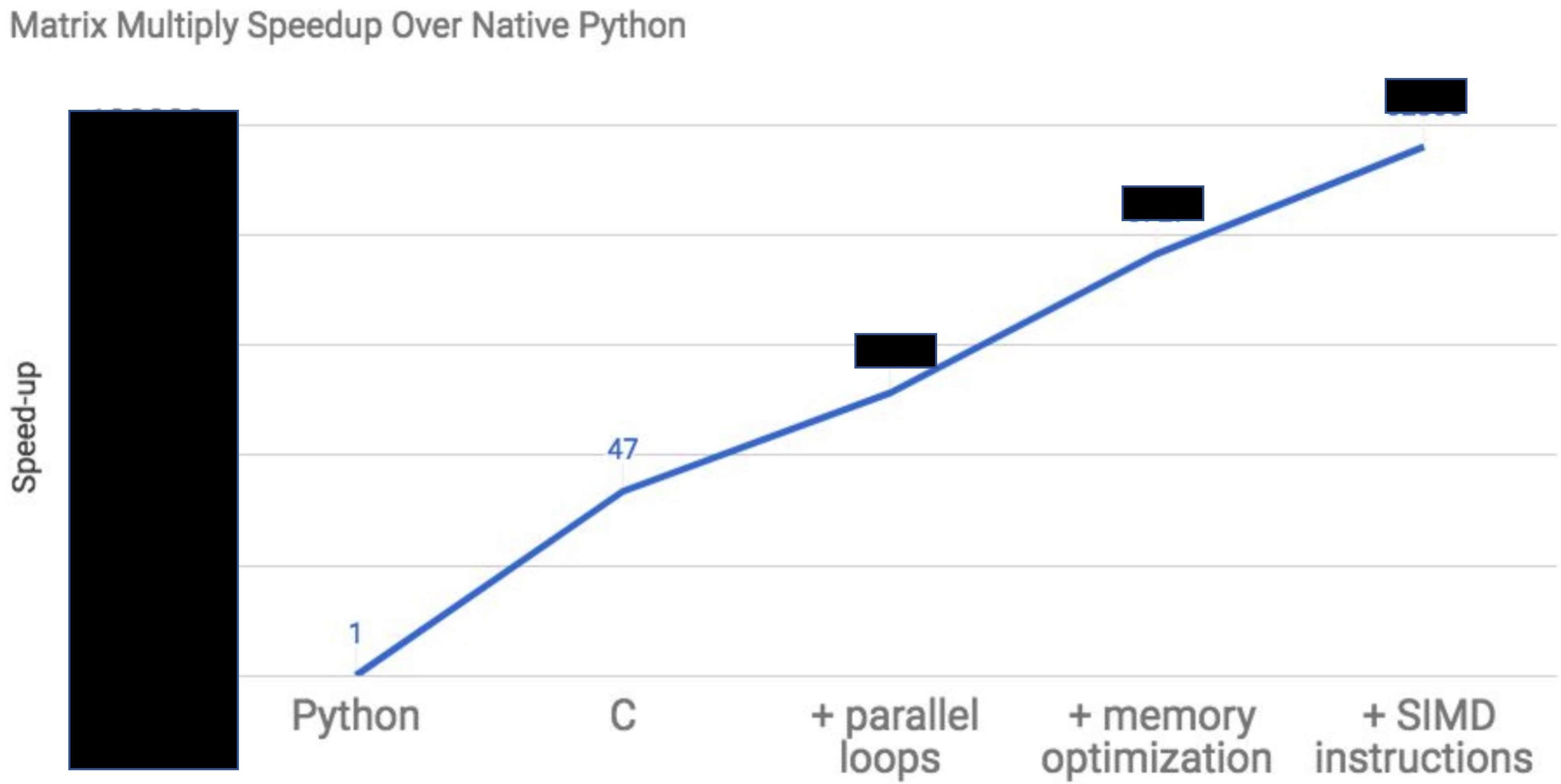


Think about it, before you write your program

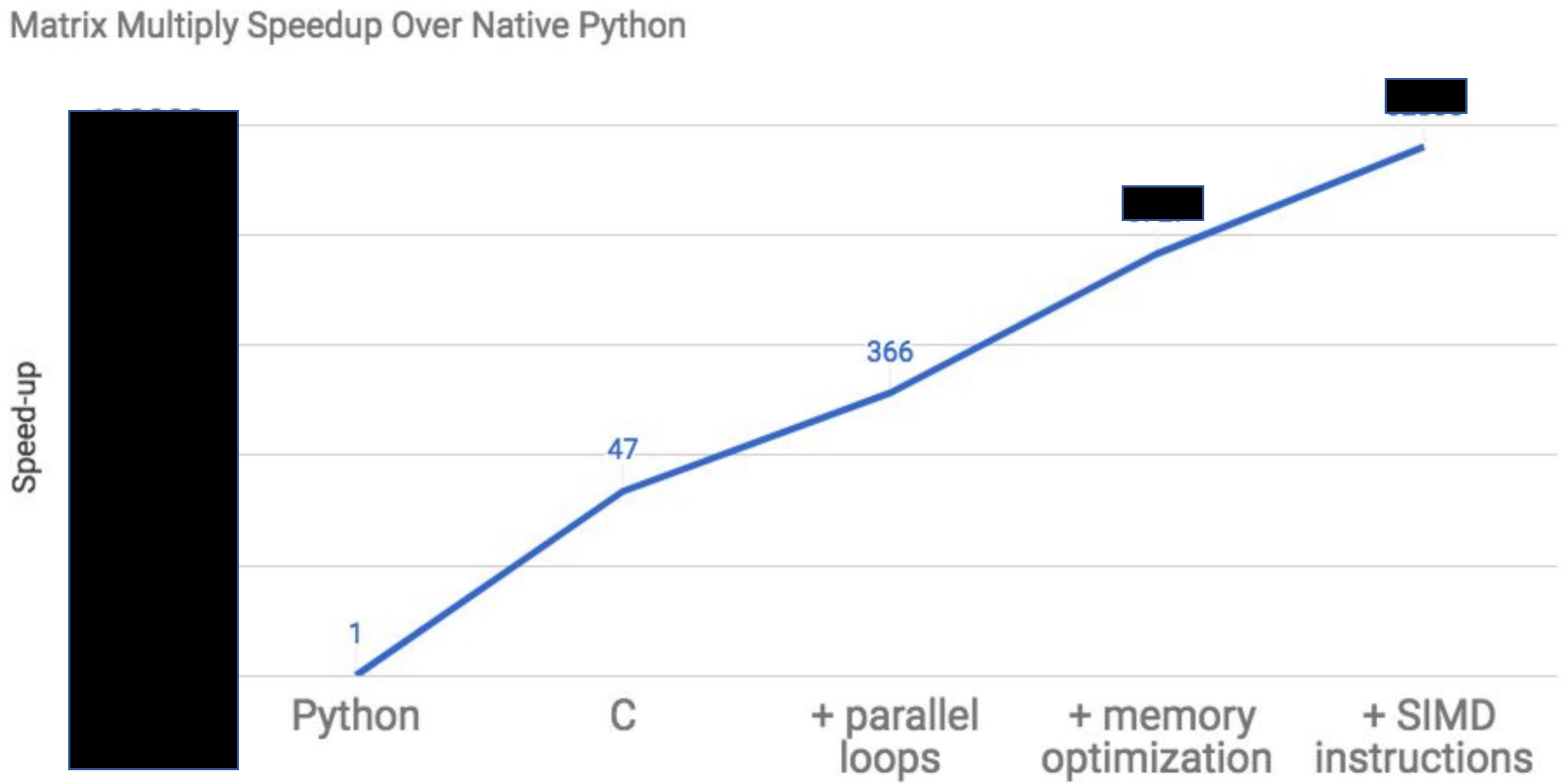
Does programming languages matter?



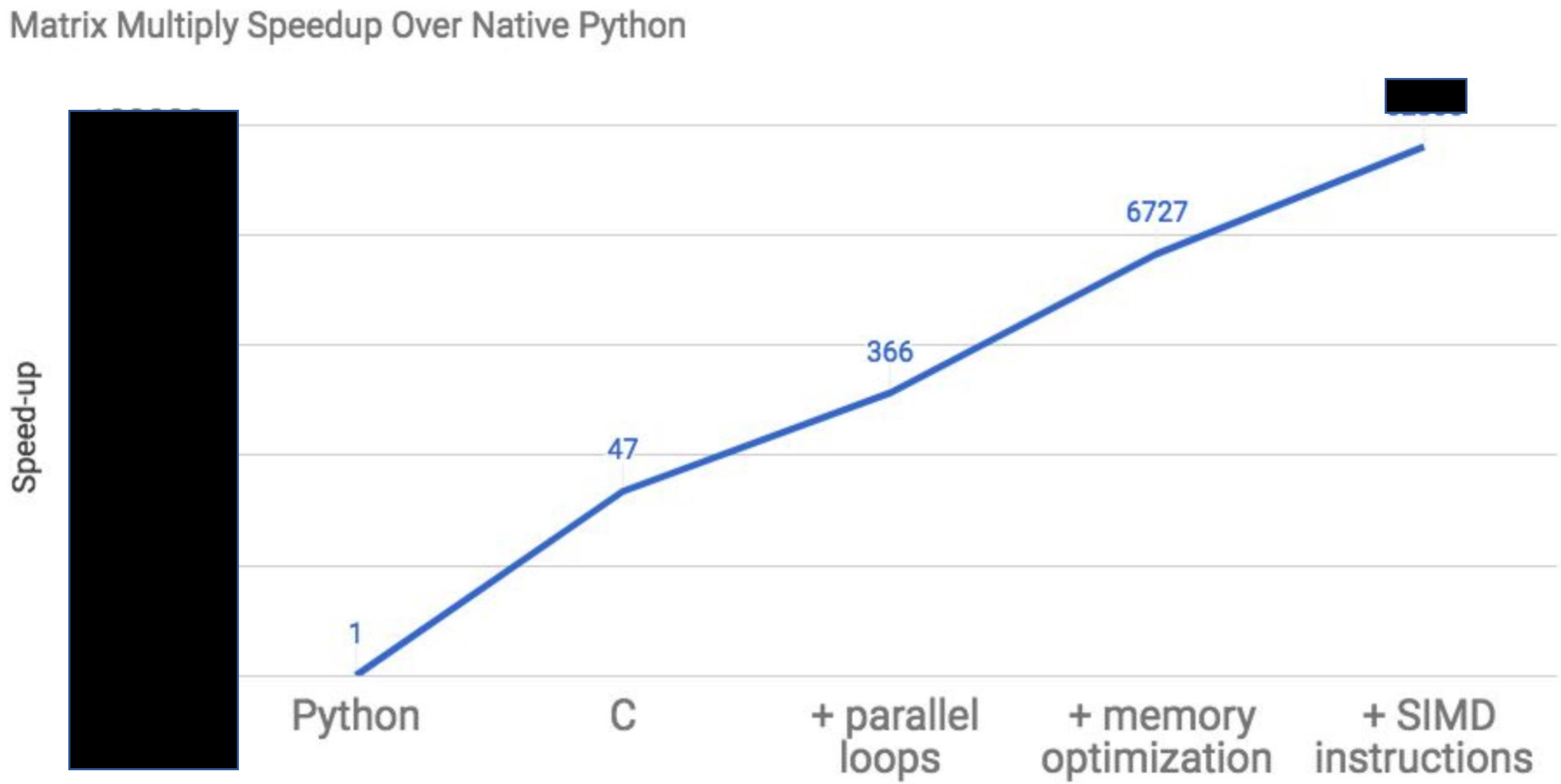
Seriously?


Matrix Multiply Speedup Over Native Python
Speed-up
[redacted]
1
47
[redacted]
[redacted]
[redacted]
Python
C
+ parallel loops
+ memory optimization
+ SIMD instructions

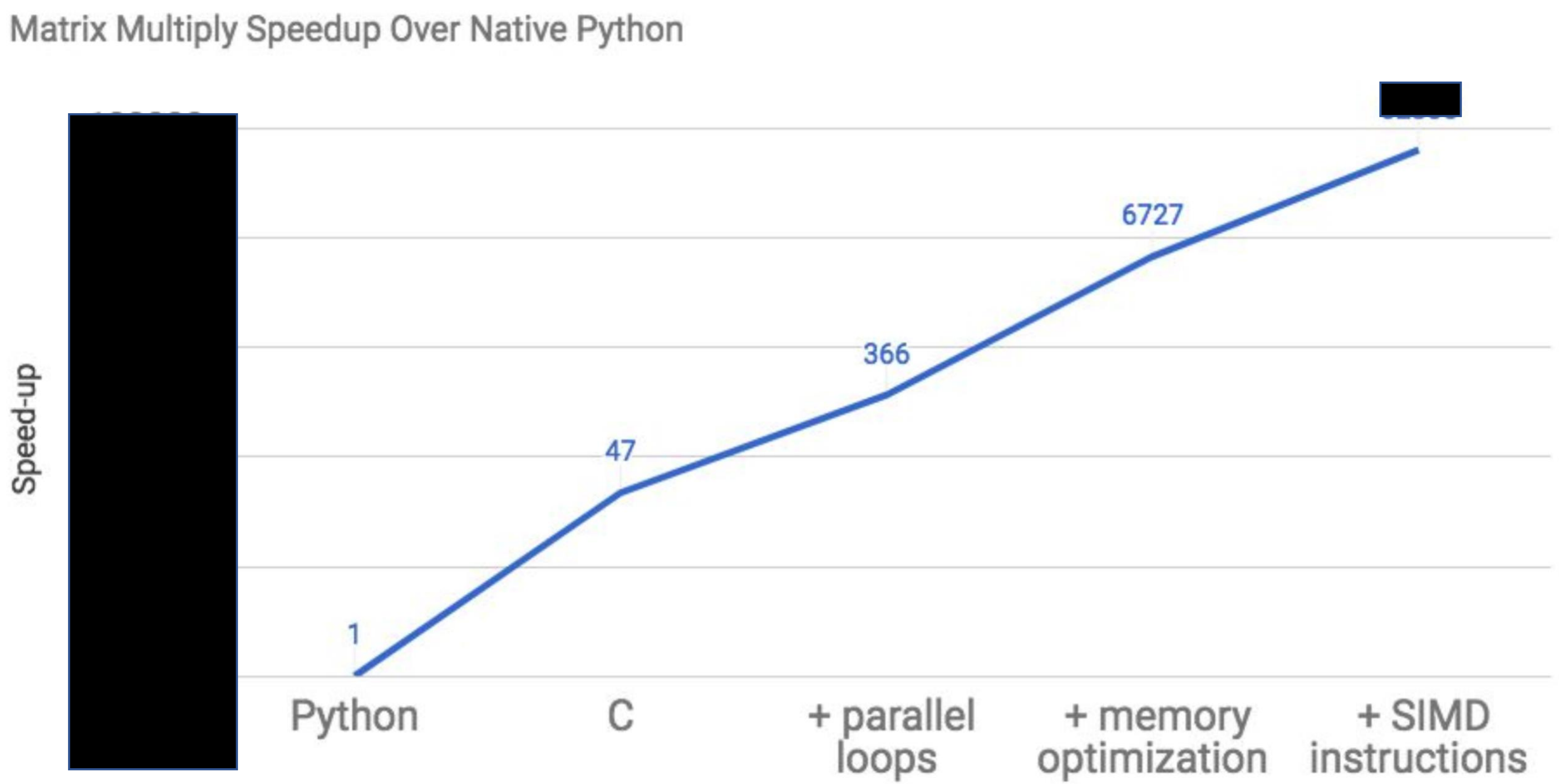
What?


Matrix Multiply Speedup Over Native Python
[redacted]
Speed-up
1
47
366
[redacted]
[redacted]
Python
C
+ parallel loops
+ memory optimization
+ SIMD instructions

Insane

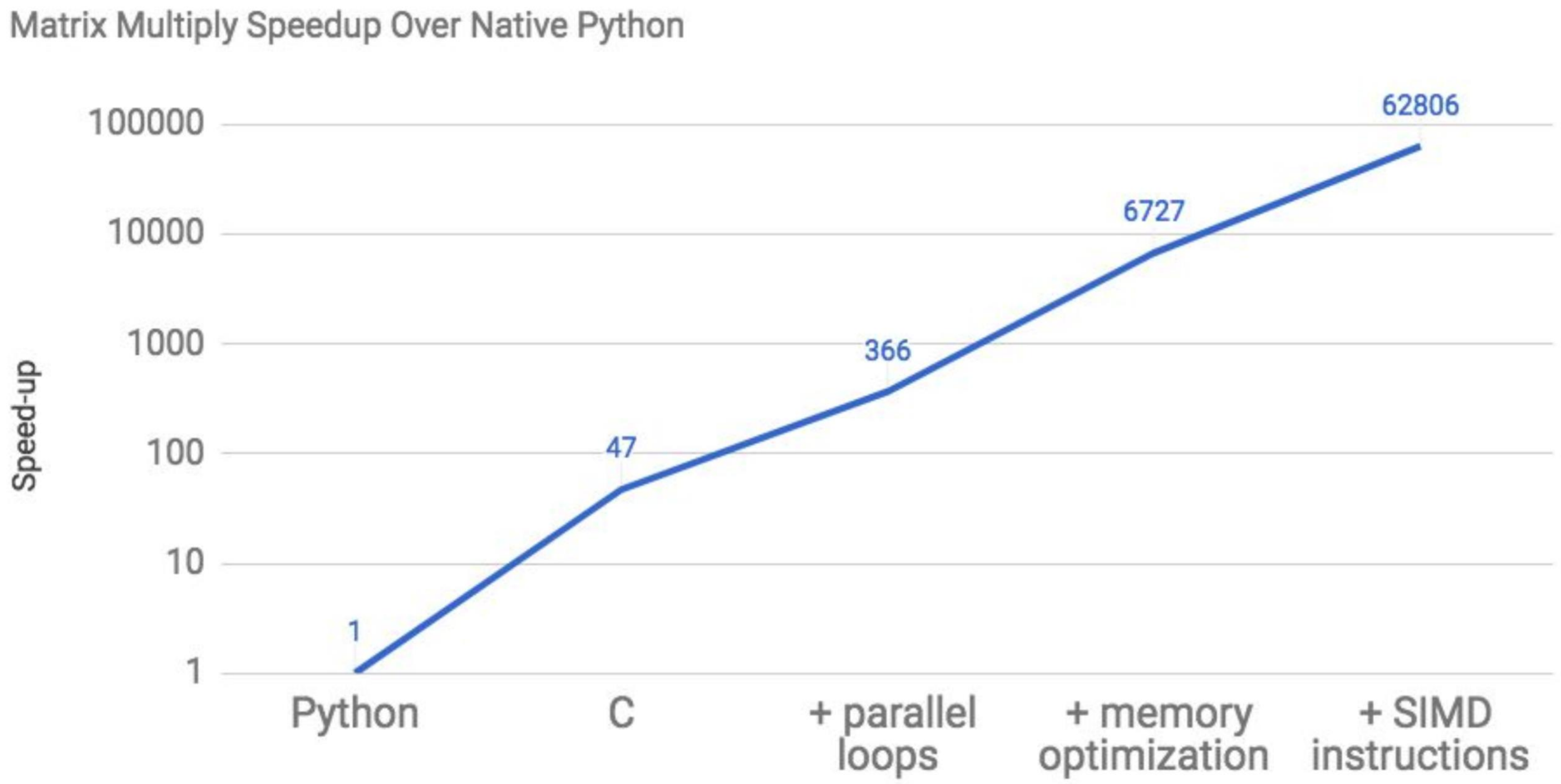

Matrix Multiply Speedup Over Native Python
[redacted]
[redacted]
6727
366
47
1
Speed-up
Python
C
+ parallel loops
+ memory optimization
+ SIMD instructions

Still?


Matrix Multiply Speedup Over Native Python
[redacted]
Speed-up
[redacted]
6727
366
47
1
Python
C
+ parallel loops
+ memory optimization
+ SIMD instructions

62,806X

Ohhhhh!!


Matrix Multiply Speedup Over Native Python
Speed-up
100000
10000
1000
100
10
1
62806
6727
366
47
1
Python
C
+ parallel loops
+ memory optimization
+ SIMD instructions

Can Compilers/programmers exploit locality?

Matrix Multiplication: 101

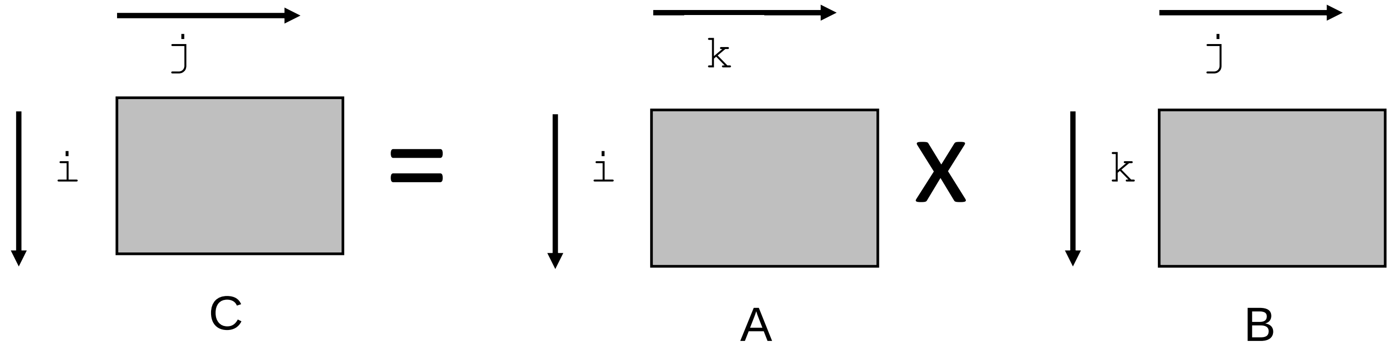
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

$$4\times3 \ + \ 2\times2 \ + \ 7\times5 = 51$$

$$\begin{bmatrix} 4 & 2 & 7 \\ 1 & 8 & 2 \\ 6 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 3 & 0 & 1 \\ 2 & 4 & 5 \\ 5 & 9 & 1 \end{bmatrix} = \begin{bmatrix} 51 & & \\ & & \\ & & \end{bmatrix}$$

Miss Rate analysis

- Assume:
  - Block size = 32B (big enough for four doubles)
  - 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



Effect of Cache Layout

C arrays allocated in row-major order

- each row in contiguous memory locations

Stepping through columns in one row:

- **for (i = 0; i < N; i++) sum += a[0][i];**
- accesses successive elements
- if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
  - miss rate = sizeof(a_{ij}) / B

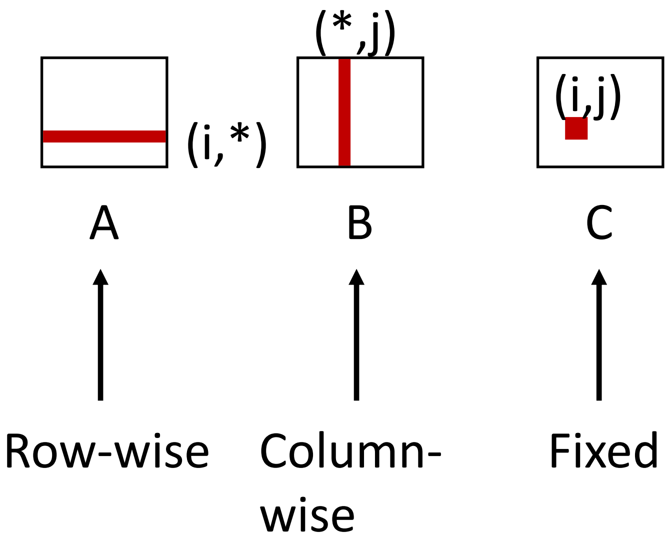
Stepping through rows in one column:

- **for (i = 0; i < N; i++) sum += a[i][0];**
- accesses distant elements
- no spatial locality!
  - miss rate = 1 (i.e. 100%)

Effect of loop order (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:



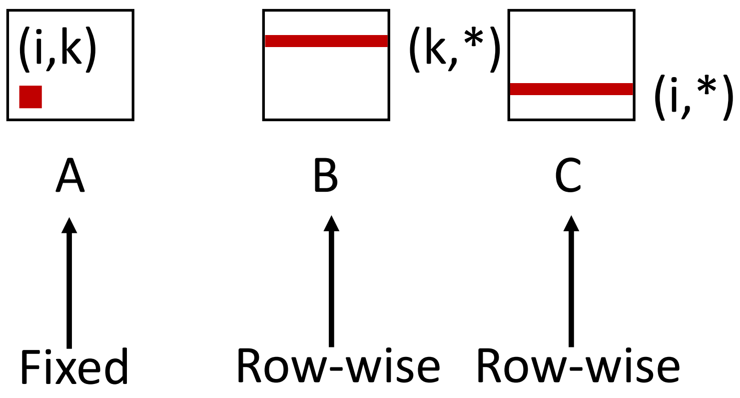
Miss rate for inner loop iterations:

A	B	C
0.25	1.0	0.0

Effect of loops (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:



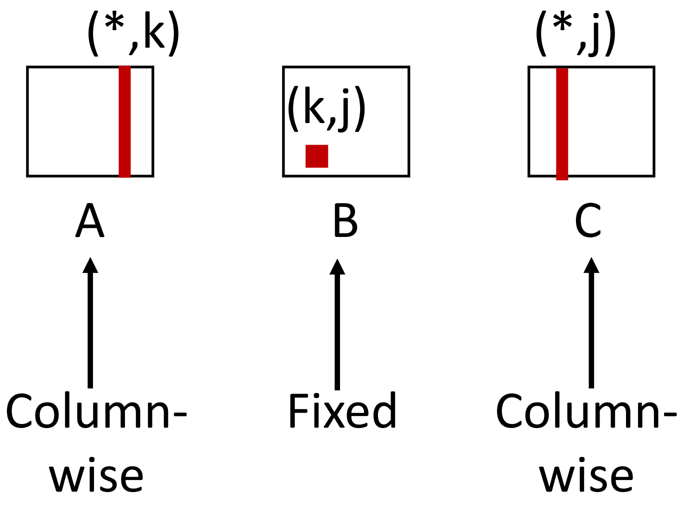
<u>Miss rate for inner loop iterations:</u>

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

Effect of loops (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;
  }
}
```

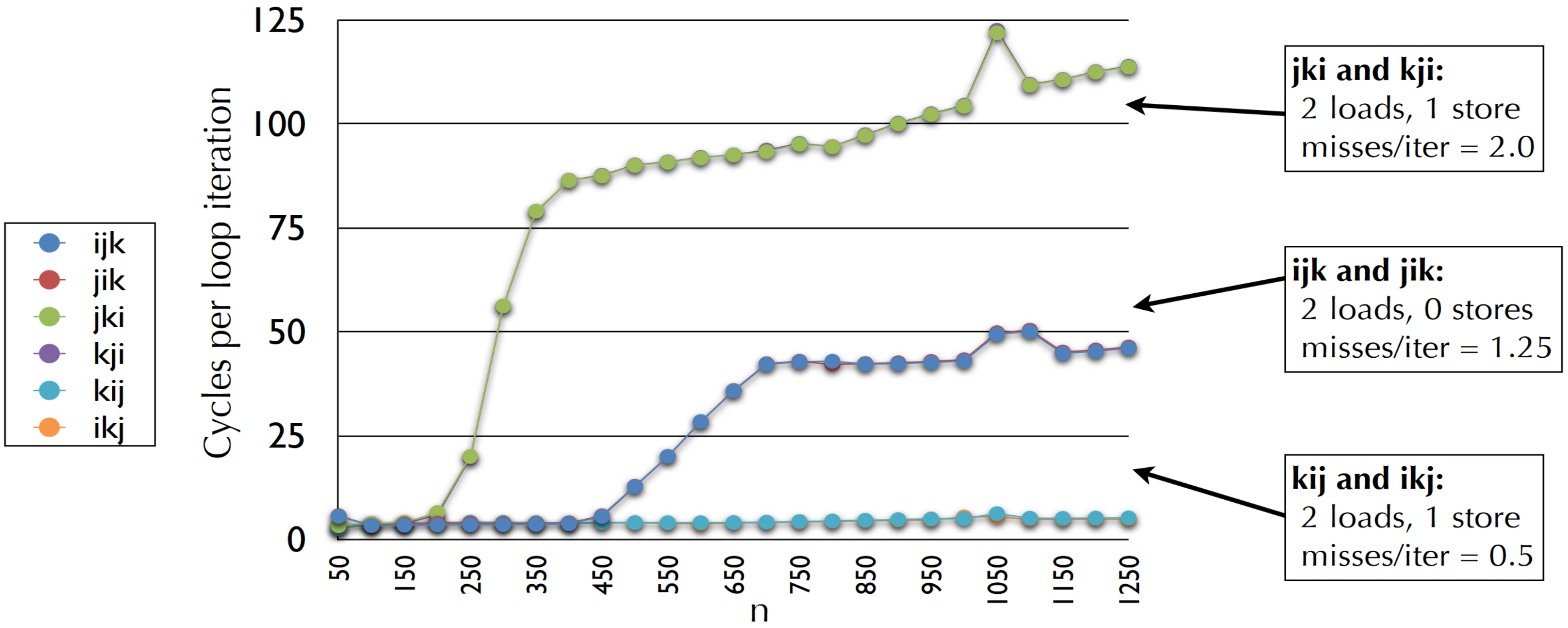
Inner loop:



Miss rate for inner loop iterations:

A	B	C
1.0	0.0	1.0

Effect of loops



- Miss rate better predictor or performance than number of mem. accesses!
- For large N, kij and ikj performance almost constant.
  Due to **hardware prefetching**, able to recognize stride-1 patterns.

Few Linux commands of interest

perf:
https://perf.wiki.kernel.org/index.php/Tutorial#Counting_with_perf_stat

dmidecode

/proc/cpuinfo

Vd'aka