

# Optimizing Python codes

Google Colab

Mahendra Verma  
IIT Kanpur

[mkv@iitk.ac.in](mailto:mkv@iitk.ac.in)

# Python codes

On MacPro (16", 2019) Big Sur

Google Colab

Intel(R) Xeon(R) CPU @ 2.20GHz

(!cat /proc/cpuinfo)

TIME in seconds

~  
~  
Fluctuations around 10% due to other jobs



## Table of contents



Strong and Weak Scaling



Python Optimization



MPI



+ Section



+ Code + Text

✓ RAM   
Disk 

```
[ ] 1 !cat /proc/cpuinfo
```

## Strong and Weak Scaling

```
[2] 1 import numpy as np
    2 import matplotlib.pyplot as plt
    3 import matplotlib.colors as colors
    4 from google.colab import files
    5
    6 def strong_scale(N):
    7     return 1/(1-f+f/N)
    8
    9 def Gustafson_scale(N):
   10     return 1-f+f*N
   11
   12 N = np.linspace(1,100,100)
   13
   14 fig, ax = plt.subplots(figsize = (3,2.5))
   15
   16 f = 0.5
```


# Data size

$$A = 10^8$$

$$A = [10^4, 10^4]$$

# Vectorization

```
1
2
3 from datetime import datetime
4 import numpy as np
5 a = np.random.random(10**8)
6 b = np.random.random(10**8)
7 c = np.empty(10**8)
8
9 # with loop
10 t1 = datetime.now()
11 for i in range(10**8):
12     c[i] = a[i]*b[i]
13 t2 = datetime.now()
14 print ("for loop, time = ", t2-t1)
15
16
17 t1 = datetime.now()
18 c=(a*b)
19 t2 = datetime.now()
20 print ("for vectorised ops, time = ", t2-t1)
21
```


$$c_i = a_i \cdot b_i$$
$$2 \quad 3$$
$$\dot{v} = -v^2$$

for  $i$  in range(1000):  
 $v(t+\Delta t) = v(t) + \Delta t \dot{v}$

MAC

Loop: T = 39 sec

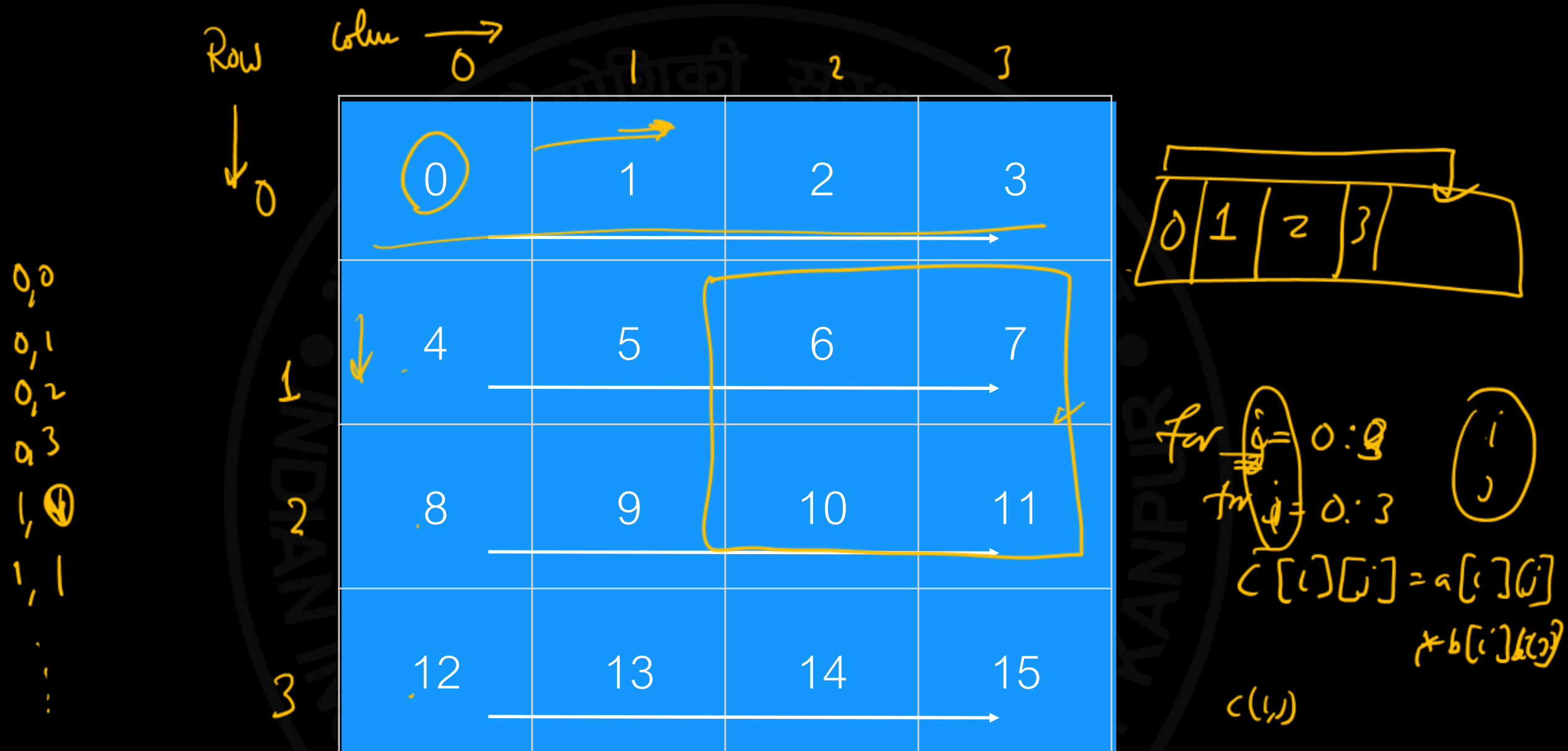
Vector: T = 0.11 sec

Colab

Loop: T = 48 sec

Vector: T = 0.36 sec

# Storage of Arrays



C: Column index varies faster than row index (i,j)

Row major  
C, Python

Column major  
Fortran

# Loop

```
36
37 t1 = datetime.now()
38 for i in range(N):
39     for j in range(N):
40         c2[i,j] = a2[i,j]*b2[i,j]
41
42 t2 = datetime.now()
43 print ("for loop_i_j, time = ", t2-t1)
44 #for loop_i_j, time = 0:00:55.346258
45
46
47 t1 = datetime.now()
48 for j in range(N):
49     for i in range(N):
50         c2[i,j] = a2[i,j]*b2[i,j]
51
52 t2 = datetime.now()
53 print ("for loop_j_i, time = ", t2-t1)
54 #for loop_j_i, time = 0:01:06.284808
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

MAC

Loop i\_j: T = 53 sec

Loop j\_i: T = 1:12 sec

Colab

Loop i\_j: T = 1:16 76

Loop j\_i: T = 1:53 113

# 2D vectorized

```
a = np.random.random((10**4,10**4))
b = np.random.random((10**4,10**4))

t1 = datetime.now()

c = a*b

t2 = datetime.now()
print ("for 2D array, time = ", t2-t1)
```

MAC

T = 0.16 sec

Colab

T = 0.38 sec



# Loop with if

```
t1 = datetime.now()
for i in range(N):
    for j in range(N):
        if (a2[i,j] > 0.5):
            c2[i,j] = a2[i,j]*b2[i,j]
        else:
            c2[i,j] = -a2[i,j]*b2[i,j]

t2 = datetime.now()
print ("for loop_i_j, time = ", t2-t1)
# for loop_i_j, time = 0:01:23.087140
```

MAC

Colab

Loop\_i\_j with if: T = 1:23 sec

Loop\_i\_j with if: T = 1:44 sec

# Optimizing C++ codes

Mahendra Verma  
IIT Kanpur

[mkv@iitk.ac.in](mailto:mkv@iitk.ac.in)

# Using Google Colab

The screenshot shows a Google Colab notebook interface. At the top, the title bar displays 'HPC\_C++.ipynb' with a star icon, and a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. A status bar indicates 'All changes saved'. On the right, there are icons for 'Comment', 'Share', and a user profile. Below the title bar, a toolbar includes a 'Table of contents' icon, a '+ Code' button, and a '+ Text' button. A tooltip above the '+ Code' button reads 'Insert code cell below ⌘/Ctrl+M B'. To the right of the toolbar, there are indicators for 'RAM' and 'Disk' usage. The left sidebar contains a search bar and a list of sections: 'Install Blitz++', 'C++ code Optimization', 'Multiplying 1D random Carrays', 'Multiplying 1D random Blitz arrays', 'Multiply 2D C++ array', 'Multiply two 2D Blitz++ arrays', 'C++ cache', and 'Blitz++ cache'. The main area displays three code cells. The first cell, labeled '[5]', contains Python code to mount Google Drive: 

```
1 from google.colab import drive
2 drive.mount('/content/drive')
```

 Below the code, it says 'Mounted at /content/drive'. The second cell, labeled '[6]', contains a shell command to clone a GitHub repository: 

```
1 !git clone https://github.com/blitzpp/blitz.git
```

 The third cell, labeled '[7]', contains a series of shell commands to build and install Blitz++: 

```
1 %%script bash
2 /* DONOT RUN THIS IF BLITZ is INSTALLED */
3 cd blitz
4 rm -rf build
5 mkdir build
6 cd build
7 cmake -DCMAKE_INSTALL_PREFIX=/content/drive/MyDrive/lib/bl
8 /* Put your google drive path after PREFIX= */
9 make lib
10 make install
```

 Several parts of the code in the third cell are highlighted with yellow circles and underlines: 'cmake', the path '/content/drive/MyDrive/lib/bl', 'make lib', and 'make install'. A yellow arrow points from the word 'github' written in the right margin to the URL in the second code cell.

Table of contents

- Install Blitz++
- C++ code Optimization
- Multiplying 1D random Carrays
- Multiplying 1D random Blitz arrays
- Multiply 2D C++ array
- Multiply two 2D Blitz++ arrays
- C++ cache
- Blitz++ cache

Insert code cell below ⌘/Ctrl+M B

[5] 1 from google.colab import drive  
2 drive.mount('/content/drive')

Mounted at /content/drive

[6] 1 !git clone https://github.com/blitzpp/blitz.git

[7] 1 %%script bash  
2 /\* DONOT RUN THIS IF BLITZ is INSTALLED \*/  
3 cd blitz  
4 rm -rf build  
5 mkdir build  
6 cd build  
7 cmake -DCMAKE\_INSTALL\_PREFIX=/content/drive/MyDrive/lib/bl  
8 /\* Put your google drive path after PREFIX= \*/  
9 make lib  
10 make install

# Using gcc

On Macbook Pro (16", 2019)  
Big Sur

Fluctuations around 10% due to other jobs

# Install gcc in mac

\$ brew install gcc

\$gcc --version

# Install gcc in ubuntu

\$ sudo apt update

~~\$ sudo apt install build-essential~~

\$ sudo apt install build-essential

\$ gcc --version

# Timing using

```
#include <chrono>  
using namespace std::chrono;
```

```
auto start = high_resolution_clock::now();  
auto stop = high_resolution_clock::now();  
auto duration = duration_cast<microseconds>(stop - start)/1e6;  
cout << duration.count() << endl;
```

1e3

TIME in seconds



# Using compiler options

-O1 -O2 -O3

- Compilers have many optimisation options.
- It tries to generate faster codes for advanced options. Compiling takes longer for such options.
- Try -O3 or -Ofast, which could yield speedup of the order of 10.
- Study these options for other compilers.



# Loop

```
#define N 100000000
```

```
for (int i=0; i<N; i++) {  
    a[i] = rand();  
    b[i] = rand();  
}
```

```
for (int i=0; i<N; i++) {  
    c[i] = a[i]*b[i];  
}
```

Without any option: dt = 675 ms

Uncertainty 10%

With -O1: T = 79.7 ms

With -O2: T = 0

With -O3: T = 0

With -Ofast: T = 0

# 2D C++ array

```
#define N 10000
```

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

Without any option:  $T = 1.49$  s

With -O3:  $T = 0$

j-i loop: Without any option:  $T = 12.4$  s

With -O3:  $T = 0$



Using blitz++

# Blitz++

Blitz++ utilizes advanced C++ template metaprogramming techniques, including expression templates,<sup>[1]</sup> to provide speed-optimized mathematical operations on sequences of data without sacrificing the natural syntax provided by other mathematical programming systems. Indeed, it has been recognized as a pioneer in the area of C++ template metaprogramming.<sup>[2]</sup>

Taken from Wikipedia

# With blitz array

#define N 100000000

```
int main()
{
    Uniform<float> x;
    x.seed((unsigned int)time(0));
    Array<double, 1> a(N), b(N), c(N);
    auto start = high_resolution_clock::now();
    for (int i=0; i<N; i++) {
        a(i) = x.random();
        b(i) = x.random();
        c(i) = a(i)*b(i);
    }
}
```

$c = a * b$

Without any option:

$T = 5.39$

Without any option:  $T = 2.69$

With -O3:  $T = 0.52$

With -O3:  $T = 0.24$

# Blitz++ 2D array

```
#define N 10000
```

```
for (int i=0; i<N; i++)  
  for (int j=0; j<N; j++) {  
    a(i,j) = x.random();  
    b(i,j) = x.random();  
    c(i,j) = a(i,j)*b(i,j);  
  }
```

$c = a * b$

$T = 5.49$

With -O3

$T(ij) = 3.43$

$T(ij) = 0.21$

$T(ji) = 38.7$

$T(ji) = 10$

With -O3:  $dt = 0.22$

# Summary

$$A = 10^8$$

$$A = [10^4, 10^4]$$

# Time in seconds

Structure	Python	C++	C++ (-O3 opt)	Blitz	Blitz (-O3)
1D loop	48	0.675	0	2.7	0.505
1D vector	0.35			5.39	0.239
2D loop (i,j)	1.16	1.48	0	3.43	0.213
2D loop (j, i)	1.53	12.37	0	38.8	10.096
2D vector	0.38			5.49	0.223



# Optimizing Python codes

Mahendra Verma  
IIT Kanpur

[mkv@iitk.ac.in](mailto:mkv@iitk.ac.in)

# Python codes

Google Colab  
Intel(R) Xeon(R) CPU @ 2.20GHz  
(!cat /proc/cpuinfo)



TIME in milliseconds (ms)

Fluctuations around 10% due to other jobs



HPC\_python.ipynb ☆

File Edit View Insert Runtime Tools Help

All changes saved

Comment

Share



Table of contents



Strong and Weak Scaling



Python Optimization



MPI



+ Section



+ Code + Text

✓ RAM   
Disk 

```
[ ] 1 !cat /proc/cpuinfo
```

## Strong and Weak Scaling

```
[2] 1 import numpy as np
    2 import matplotlib.pyplot as plt
    3 import matplotlib.colors as colors
    4 from google.colab import files
    5
    6 def strong_scale(N):
    7     return 1/(1-f+f/N)
    8
    9 def Gustafson_scale(N):
   10     return 1-f+f*N
   11
   12 N = np.linspace(1,100,100)
   13
   14 fig, ax = plt.subplots(figsize = (3,2.5))
   15
   16 f = 0.5
```

# Data size

$$A = 10^8$$

$$A = [10^4, 10^4]$$

*sum(a\*b)*

*sum(sin(a))*

# 1D array sum(a\*b)

```
7 # with loop
8 t1 = datetime.now()
9 tot = 0
10 for i in range(10**8):
11     tot += a[i]*b[i]
12 t2 = datetime.now()
13 print ("for loop, time, tot = ", t2-t1, tot)
14 ## 1D vectorized
15 t1 = datetime.now()
16 tot = np.sum(a*b)
17 t2 = datetime.now()
18 print ("for 1D vectorised a*b, time, tot = ", t2-t1, tot)
```


*import numpy as np*

*01:2:3.001 GMT*

*loop*

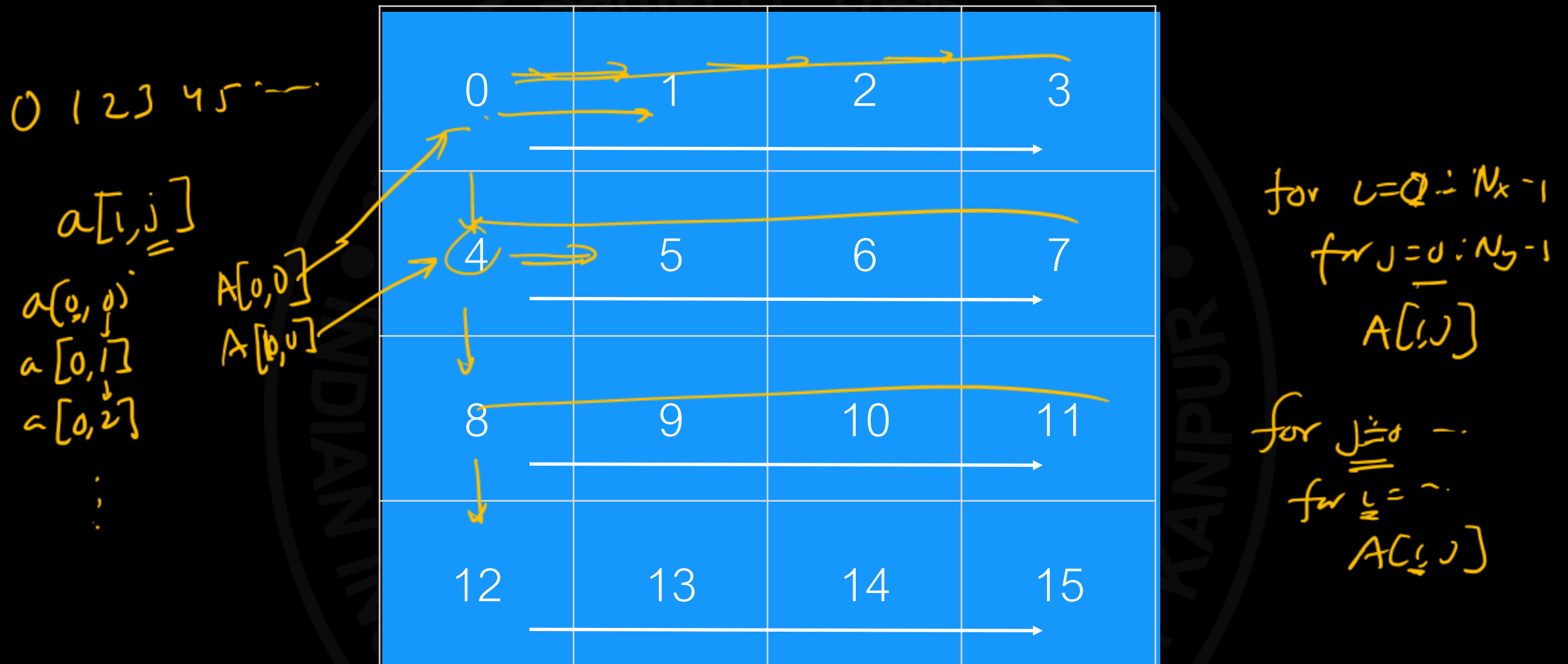
*a\*b*

*80 times faster!!*



	<u>None</u>	-O1	-O2	-O3	-Ofast
C++	361	357	352	261	<u>155</u>
Loop	<u>30910</u>				
Vector	<u>473</u>				

# Storage of Arrays



C: Column index varies faster than row index (i,j)

Row major  
C, Python

Column major  
Fortran

# Loop

```
26 t1 = datetime.now()
27 tot = 0
28 for i in range(N):
29     for j in range(N):
30         tot += a2[i,j]*b2[i,j]
31 t2 = datetime.now()
32 print ("for loop_i_j, time, tot = ", t2-t1, tot)
33
34 t1 = datetime.now()
35 tot = 0
36 for j in range(N):
37     for i in range(N):
38         tot += a2[i,j]*b2[i,j]
39 t2 = datetime.now()
40 print ("for loop_j_i, time, tot = ", t2-t1, tot)
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

sum(a\*b)

	None	-O1	-O2	-O3	-Ofast
C++ ij	359	350	421	289	133
C++ ji	5753	5826	6205	5910	143
Py ij	37920				
Py ji	45400				
Py Vector	746				

# Loop with if

```
45 for i in range(N):
46     for j in range(N):
47         if (a2[i,j] > 0.5):
48             c2[i,j] = a2[i,j]*b2[i,j]
49         else: for tot
50             c2[i,j] = -1.098*a2[i,j]*b2[i,j]
51
52 t2 = datetime.now()
53 print ("for loop_i_j_with_if, time, tot = ", t2-t1, tot)
```

Loop\_i\_j with if: T = 62230 sec



# sum(sin(a))

```
for i in range(10**8):  
    tot += np.sin(a[i])
```

*loop*

```
tot = np.sum(np.sin(a))
```

*vector*

	None	-O1	-O2	-O3	-Ofast
C++	1727	2293	1351	1353	653
Py Loop	92150				
Py Vector	1313				

# 2D sum(sin(a))

exp  
log

	None	-O1	-O2	-O3	-Ofast
C++ ij	1775	2149	2165	1794	836
C++ ji	9750	6624	6217	6272	652
Py ij	90390				
Py ji	102400				
Py Vector	1141				

# Optimizing C++ codes

Mahendra Verma  
IIT Kanpur

[mkv@iitk.ac.in](mailto:mkv@iitk.ac.in)

# Using Google Colab

The screenshot shows a Google Colab notebook interface. At the top, the title bar reads "HPC\_C++.ipynb" with a star icon. Below it is a menu bar with "File", "Edit", "View", "Insert", "Runtime", "Tools", and "Help". To the right of the menu bar are icons for "Comment", "Share", and a settings gear. A status bar at the top right indicates "All changes saved".

On the left side, there is a "Table of contents" panel with a search icon. It lists several sections: "Install Blitz++", "C++ code Optimization", "Multiplying 1D random Carrays", "Multiplying 1D random Blitz arrays", "Multiply 2D C++ array", "Multiply two 2D Blitz++ arrays", "C++ cache", and "Blitz++ cache". At the bottom of this panel is a "+ Section" button.

The main area of the notebook contains three code cells. The first cell is a text cell with the text "Insert code cell below" and a keyboard shortcut "⌘/Ctrl+M B". The second cell is a code cell with the following code:

```
[5] 1 from google.colab import drive
     2 drive.mount('/content/drive')
```

Below the code, it says "Mounted at /content/drive". The third cell is a code cell with the following code:

```
[7] 1 %%script bash
     2 /* DONOT RUN THIS IF BLITZ is INSTALLED */
     3 cd blitz
     4 rm -rf build
     5 mkdir build
     6 cd build
     7 cmake -DCMAKE_INSTALL_PREFIX=/content/drive/MyDrive/lib/bl
     8 /* Put your google drive path after PREFIX= */
     9 make lib
    10 make install
```

A red bracket is drawn on the left side of the third code cell, spanning from line 3 to line 10. A yellow circle is drawn around the word "blitz" in line 3. The notebook interface also shows RAM and Disk usage indicators at the top right.

# Install gcc in mac

```
$ brew install gcc
```

```
$ gcc --version
```

# Install gcc in ubuntu

```
$ sudo apt update
```

```
$sudo apt install build-essential
```

```
$sudo apt install build-essential
```

```
$gcc --version
```

# Timing using

```
#include <chrono>  
using namespace std::chrono;
```

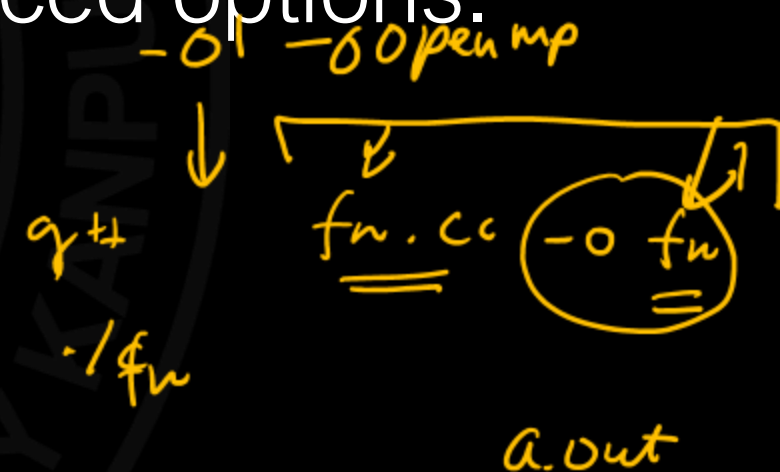
```
auto start = high_resolution_clock::now();  
auto stop = high_resolution_clock::now();  
auto duration = duration_cast<microseconds>(stop - start)/1e6;  
cout << duration.count() << endl;
```

TIME in milliseconds (ms)



# Using compiler options

- Compilers have many optimisation options.
- It tries to generate faster codes for advanced options. Compiling takes longer for such options.
- Options: ~~-O1, -O2, -O3, -Ofast~~
- -Ofast yields the most efficient code among the above.
- Study these options for other compilers.





# sum(a\*b)

#define N 100000000

```
for (int i=0; i<N; i++) {  
    a[i] = (float)(rand()) / (float)(RAND_MAX);  
    b[i] = (float)(rand()) / (float)(RAND_MAX);  
}
```

```
for (int i=0; i<N; i++) {  
    sum += a[i]*b[i];  
}
```

None	-O1	-O2	-O3	-Ofast
361	357	352	261	155

Uncertainty 10%

# 2D C++ array sum(a\*b)

```
#define N 10000
```

```
for (int j=0; j<N; j++)  
    for (int i=0; i<N; i++) {  
        sum += a[i][j] * b[i][j] ;  
    }
```

	None	-O1	-O2	-O3	-Ofast
ij	359	350	421	289	133
Ji	5753	5826	6205	5910	143
ij (with if)	4681	4147	4142	4256	4228

# sum(sin(A))

```
#define N 1000000000
```

```
for (int i=0; i<N; i++) {  
    a[i] = rand();  
    b[i] = rand();  
}
```

```
for (int i=0; i<N; i++) {  
    sum += a[i]*b[i];  
}
```

*sum(a[i])*

None	-O1	-O2	-O3	-Ofast
<u>1727</u>	2293	1351	1353	<u>653</u>

# 2D C++ array sum(sin(a))

```
#define N 10000
```

```
for (int i=0; i<N; i++)  
    for (int j=0; j<N; j++) {  
        sum += sin(a[i][j]) ;  
    }
```

	None	-O1	-O2	-O3	-Ofast
ij	1775	2149	2165	1794	836
Ji	9750	6624	6217	6272	652



Using blitz++

# Blitz++

Blitz++ utilizes advanced C++ template metaprogramming techniques, including expression templates,<sup>[1]</sup> to provide speed-optimized mathematical operations on sequences of data without sacrificing the natural syntax provided by other mathematical programming systems. Indeed, it has been recognized as a pioneer in the area of C++ template metaprogramming.<sup>[2]</sup>

[99]

Taken from Wikipedia

# sum(a\*b) for 1D blitz array

#define N 1000000000

```
for (int i=0; i<N; i++) {  
    a(i) = x.random();  
    b(i) = x.random();  
}
```

```
for (int i=0; i<N; i++)  
    sum_arr += a(i)*b(i);
```

~~sum(a\*b)~~

sum(c)

	None	-O1	-O2	-O3	-Ofast
C++	1727	2293	1351	1353	155
Loop	2020	354	353	270	135
Vector	5778	669	790	537	454

blitz++



# sum(a\*b) for 2D blitz array

#define N 1000

	None	-O1	-O2	-O3	-Ofast
C++ ij	359	350	421	289	133
C++ ji	5753	5826	6205	5910	143
Blitz ij	2239	350	352	263	185
Blitz ji	10542	5203	5045	5010	5061
Vector	5051	317	327	258	273



# sum(sin(a)) blitz array

#define N 1000000000

```
for (int i=0; i<N; i++)  
    sum_arr += sin(a(i));
```

c=sin(a)  
sum(c)

	None	-O1	-O2	-O3	-Ofast
<u>C++</u>	1727	2293	1351	1353	<u>653</u>
Loop	3409	1790	1336	1352	677
Vector	5501	5779	2303	1818	1230

# sum(sin(a)) for 2D Blitz array

#define N 100000000

	None	-O1	-O2	-O3	-Ofast
C++ ij	1775	2149	2165	1794	836
C++ ji	9750	6624	6217	6272	652
Blitz ij	2846	1386	1325	1315	667
Blitz ji	8690	6421	6534	6885	3464
Vector	5404	1439	1445	1497	777

# Summary

$$A = 10^8$$

$$A = [10^4, 10^4]$$

sum(a\*b): Time in mili sec

Structure	Python	C++ (-Ofast)	Blitz (-Ofast)
<u>1D loop</u>	30013	155	135
1D vector	473		454
2D loop (i,j)	37920	133	185
2D loop <u>(j, i)</u>	45400	143	5061
2D vector	746		273

sin(a): Time in mili sec

Structure	Python	C++ (-Ofast)	Blitz (-Ofast)
1D loop	92150	653	677
1D vector	1313		1230
2D loop (i,j)	90390	836	667
2D loop (j,i)	102400	652	3464
2D vector	1141		777

# Optimizing C++ codes

Mahendra Verma  
IIT Kanpur

[mkv@iitk.ac.in](mailto:mkv@iitk.ac.in)



# Cache locality

Timings with -O3 & -Ofast on Colab

<https://www.youtube.com/watch?v=...>

$$C[i,j] = \sum_k A[i,k] * B[k,j]$$

$$C = A * B$$

N = 2048

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

-O3: 53.1812 s

-Ofast: 6.9823 s

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

-O3: 6.4822 s

-Ofast: 6.5144 s



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

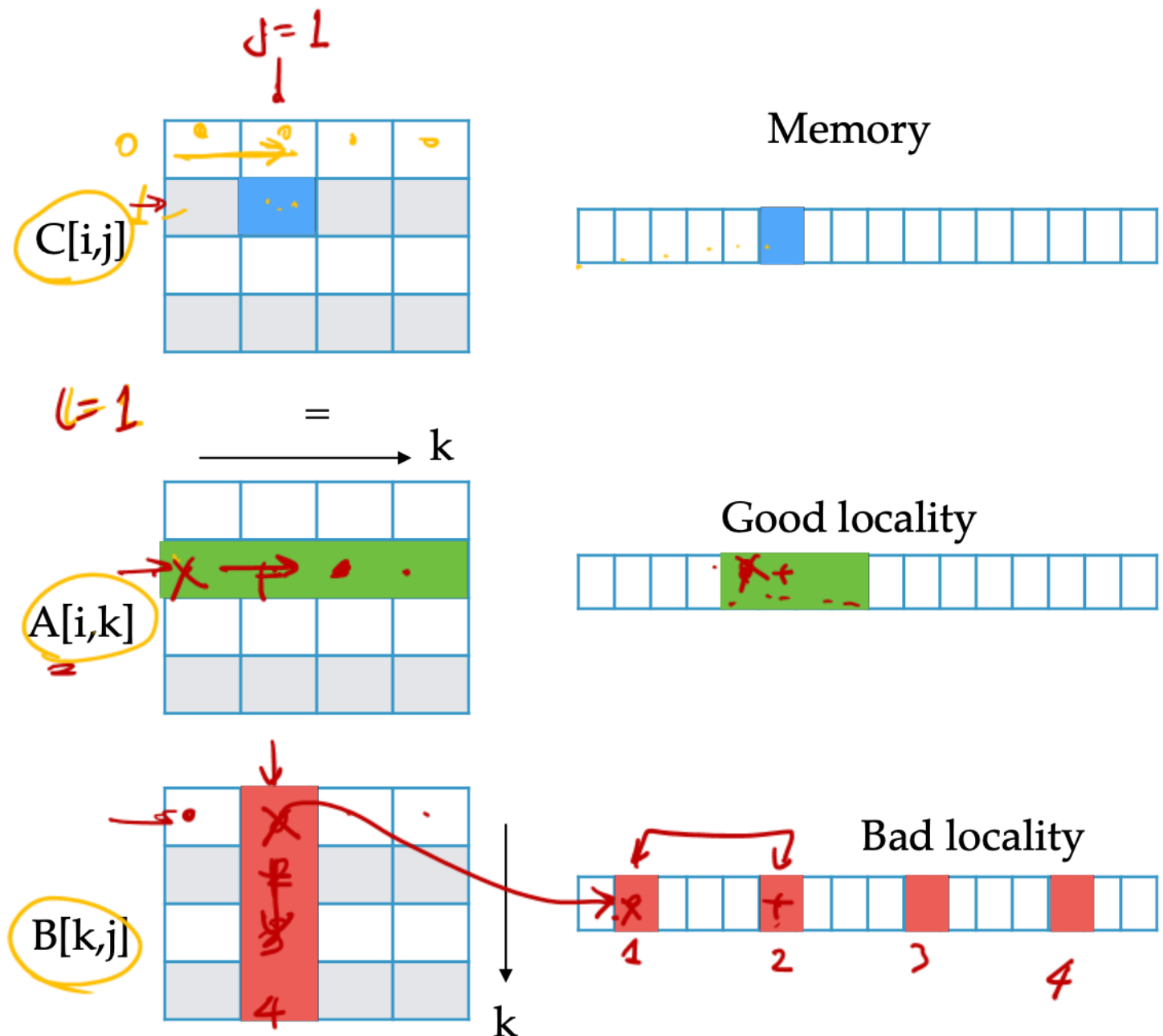
-O3: 6.9082 s

-Ofast: 6.8868 s

```

for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    for (int k=0; k<N; k++)
    {
      C(i,j) += A(i,k)*B(k,j);
    }

```

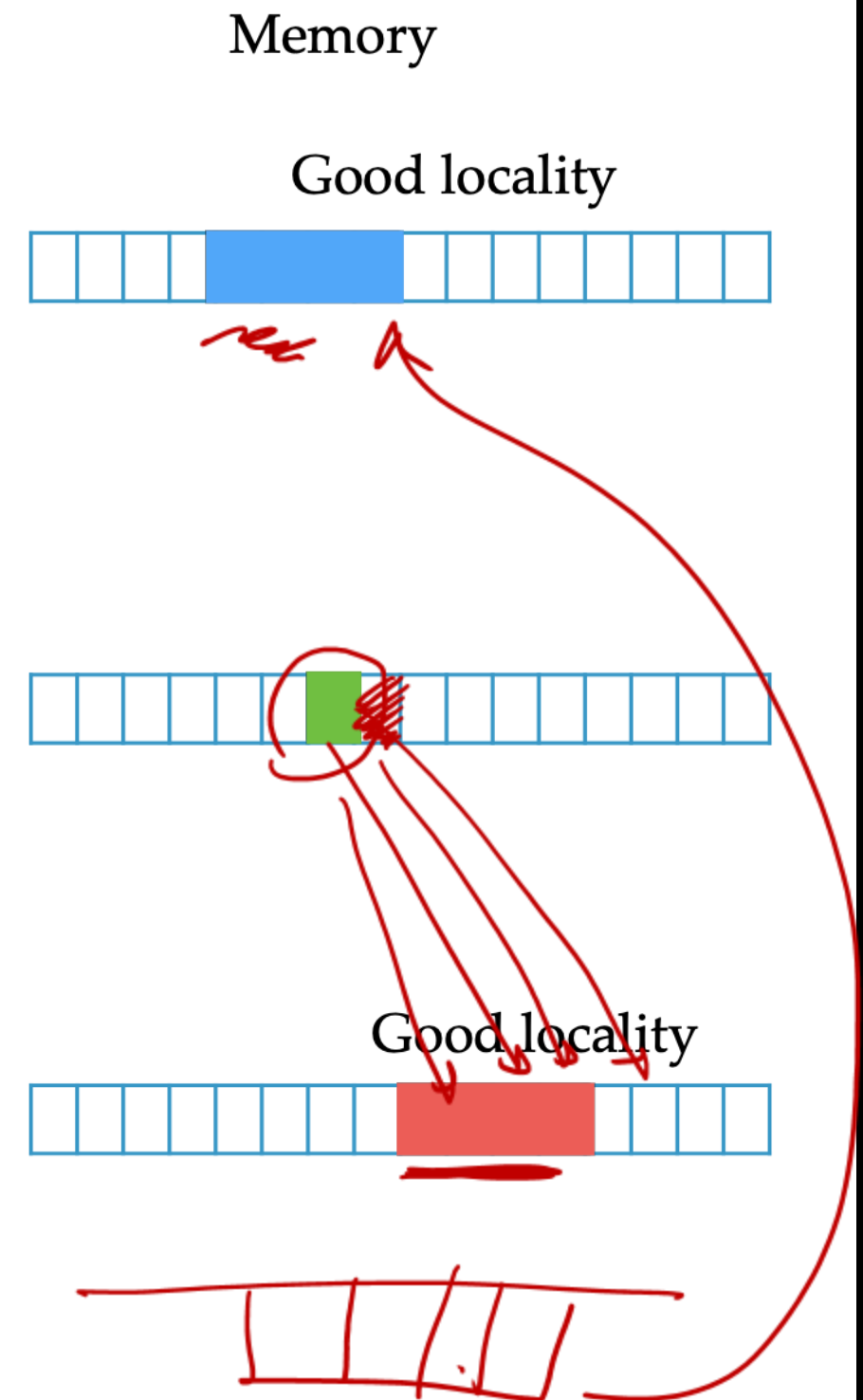
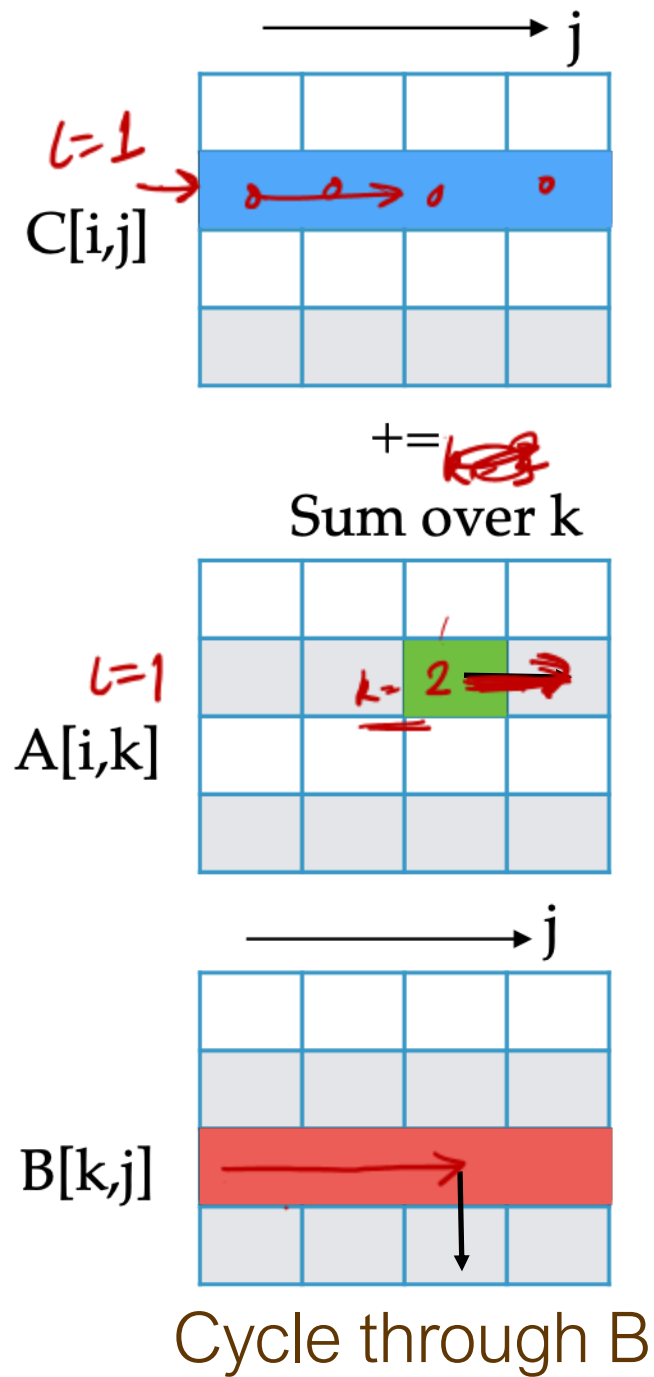


Bad locality of B leads to inefficiency

```

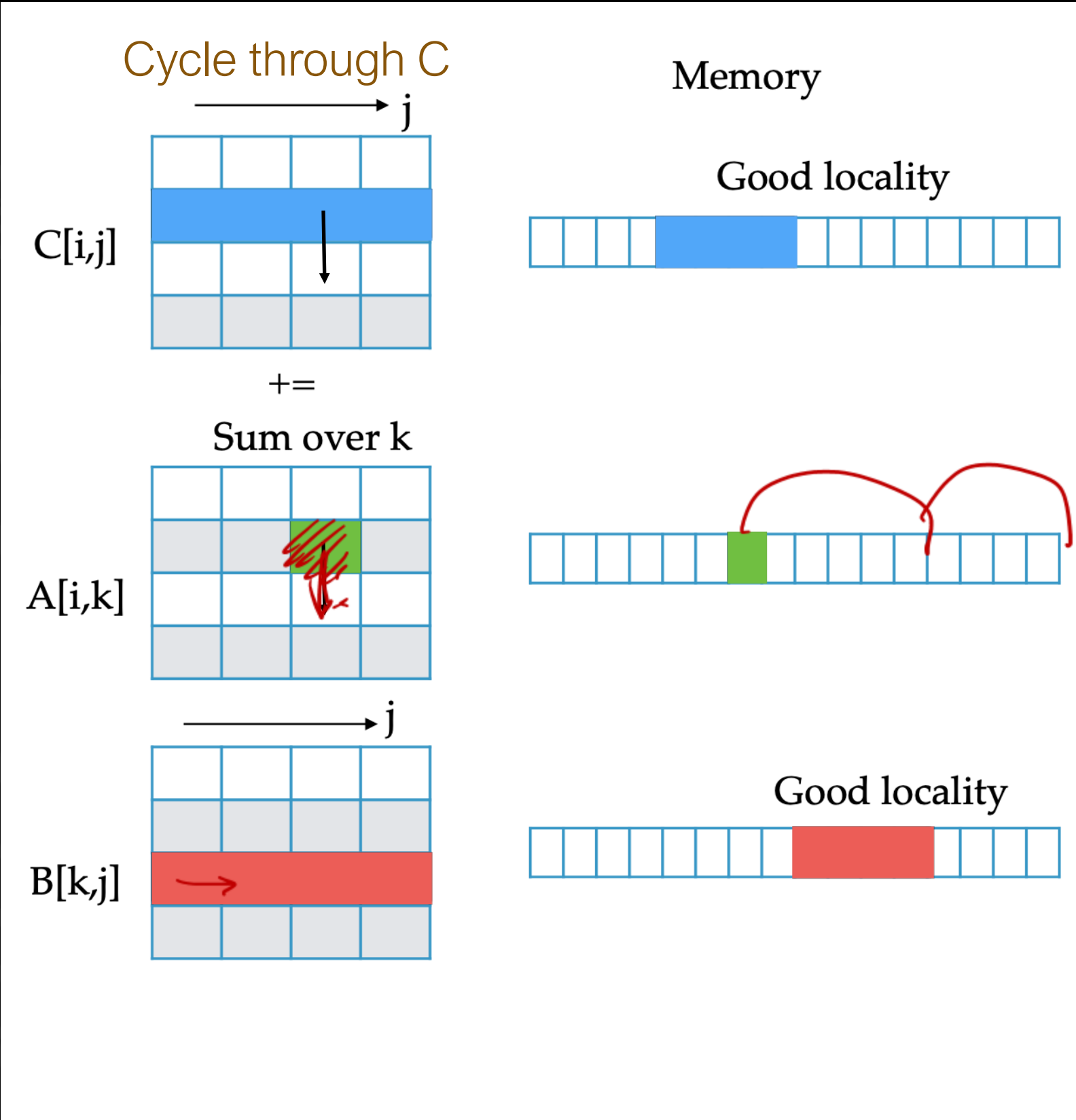
for (int i=0; i<N; i++)
  for (int k=0; k<N; k++)
    for (int j=0; j<N; j++)
    {
      C(i,j) += A(i,k)*B(k,j);
    }

```



Good locality of B & C leads to efficient code

```
for (int k=0; k<N; k++)
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
    {
      C(i,j) += A(i,k)*B(k,j);
    }
```



Good locality of B & C leads to efficient code

# The impact of cache locality on performance in C through matrix multiplication



Gunavaran Brihadiswaran

Follow

Jul 29, 2020 · 5 min read ★



## The Experiment

I changed the order of the nested loops and performed three iterations for each combination. The results are as follows.

	Elapsed Time (seconds)					
Iteration	i,j,k	i,k,j	j,i,k	j,k,i	k,i,j	k,j,i
1	195.92	34.78	183.97	274.35	35.53	286.1
2	198.37	34.86	189.94	278.82	35.29	278.64
3	196.41	34.71	183.9	280.99	35.18	274.59
Average	196.90	34.78	185.94	278.05	35.33	279.78

# Speeding up using Numba

Mahendra Verma  
IIT Kanpur

[mkv@iitk.ac.in](mailto:mkv@iitk.ac.in)

# Numba

- Numba translates Python functions to optimized machine code at runtime using LLVM lib.
- JIT: *Just in time* compiler
- Performance comparable to C or Fortran codes.
- Apply Numba decorators on python functions.
- <https://numba.pydata.org/>



```
from numba import jit, njit, prange
```

```
@jit
```

```
def my_log_nocache(a):
```

```
    M, N = a.shape
```

```
    tot = 0
```

```
    for i in prange(M):
```

```
        for j in prange(N):
```

```
            tot += np.log(a[i, j])
```

```
    return tot
```



# Cache option

9a x.a - 0 x.0

- `@njit(cache=True)`
- This decorator (command) instructs Python to save the compiled code in a cache.

```
from numba import jit, njit, prange

@jit(cache = True)
def my_log_cache(a):
    M, N = a.shape
    tot = 0
    for i in prange(M):
        for j in prange(N):
            tot += np.log(a[i,j]) + np.sin(a[i,j])
    return tot
```

# Timing in Python

```
from numba import jit, njit, prange

@njit
def dummy():
    return None

@jit Cache
def my_log_nocache(a):
    M, N = a.shape
    tot = 0
    for i in prange(M):
        for j in prange(N):
            tot += np.log(a[i,j]) + np.sin(a[i,j])
    return tot
```

*Sum(log(a))*

Dummy timing 0.036316633224487305

No cache 1st timing 2.4474663734436035 tot= -54026001.04944232

No cache 2nd timing 1.6440708637237549 tot= -54026087.339905575

Cache 1st timing 1.7453806400299072 tot= -54024569.87402726

Cache 2nd timing 1.6321403980255127 tot= -54034193.91120762

Cache 3rd timing 1.658315896987915 tot= -54028264.60840158

# Timings in ms

C++ (-Ofast)	653
Py Loop	92150
Py Vector	1313
Numba	1171

# nopython mode

- @jit(nopython=True)
- Or @njit
- This mode generates fast non-python code.

# Parallel option

- @njit(parallel=True)
- This mode generates parallel code if possible
- Use prange instead of range

```
from numba import jit, njit, prange
n
@jit(parallel = True)
def my_log_nocache(a):
    M, N = a.shape
    tot = 0
    — for i in prange(M):
        for j in prange(N):
            tot += np.log(a[i,j]) + np.sin(a[i,j])
    return tot
```

Handwritten annotations:

- A yellow circle around `@jit(parallel = True)` with a small 'n' above it.
- A yellow circle around `prange(M)` with `2048` written next to it.
- A yellow circle around `prange(N)` with `2048` written next to it.
- A yellow circle around the entire loop body with `256 x 2048` written next to it.



On MacBook Pro 16" (2019): 8 cores

### Without parallel

```
(base) codes/codes23 $python numba_p11.py
Dummy timing 0.08803677558898926
No cache 1st timing 1.0817480087280273 tot= -54012060.12899187
No cache 2nd timing 1.0117287635803223 tot= -54026917.40798862
Cache 1st timing 1.1753530502319336 tot= -54015353.38920122
Cache 2nd timing 1.0465059280395508 tot= -54032690.54539104
Cache 3rd timing 1.076902151107788 tot= -54024366.320906855
```

### With parallel

```
Dummy timing 0.16474199295043945
No cache 1st timing 0.607647180557251 tot= -54030760.91180665
No cache 2nd timing 0.1352088451385498 tot= -54042526.14654334
Cache 1st timing 0.6361870765686035 tot= -54047140.20137752
Cache 2nd timing 0.13567590713500977 tot= -54034882.26079227
Cache 3rd timing 0.13421010971069336 tot= -54036017.213262975
```

# Advanced features

- nogil = True
- The @vectorize decorator
- The @guvectorize decorator

# No speedup with Numba

```
@jit
start = time.time()
tot = np.sum(np.log(a))
end = time.time()
print(f'No Python sum timing {end - start}', f'tot= {tot}')
```

```
start = time.time()
tot = my_sum(a)
end = time.time()
print(f'No Jit sum 1st timing {end - start}', f'tot= {tot}')
```

```
@jit
def my_sum(c)
    return np.sum(a)
```

np.sum(a) is already optimized.

Hence, Numba does not speedup the code.



# Summary

- Numba yields C/Fortran like high performance in Python.
- Numba: Cheap way to speedup!
- No need to make changes in the code. We just need to put Numba decorators ahead of functions.