

Operating Systems
Assignment 2
Matrix Multiplication using
multi-threading

Submitted by:
Patrick Georges Kromil
20010383

Submitted to:
Dr.Noha Adly
Eng.Sajed Hassan

1- Code Organization:

1 – Read arguments then decide which files to read from and store to.

2 – Begin by performing all three methods (A thread per Matrix, A thread per Row, and A thread per element)

3 – For a thread per matrix, I did not create a thread to perform the procedure, instead I chose to let the main process perform it.

4 – For a thread per row, first I looped over the rows of the resultant matrix which is equal to the number of rows of the first matrix and in each iteration, I created a thread with `pthread_create()` and passing the number of rows to the thread function and accessing the matrices globally.

After each thread was created, we called `pthread_join()` so the main thread (or main process) waits for each worker thread to terminate, so no need to call `pthread_exit()` inside each thread.

5 – For a thread per element, first I looped over each element of the resultant matrix which are equal to the product of the number of rows of the first matrix times the number of columns of the second one, in each iteration I created a thread and passed to its function a pointer to a struct holding the row number and the column number of the calculated element. Then, we called `pthread_join()` to wait for all worker threads to terminate.

2-Main Functions:

1-`pthread_create()` : it takes the thread id and creates the corresponding thread which calls the specified function

2 – `pthread_join()`: it takes the thread id as an argument and makes the main thread wait for this thread to terminate execution before proceeding.

3 – `fopen()`: to open a txt file and read from it or write to it.

4 – `malloc()`: This method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

3-How to Compile and run the code:

A.To compile the code to have an executable file:

```
$ gcc -pthread -o main main.c
```

B.To run the code :

with arguments :

```
$ ./main a b c
```

without arguments :

```
$ ./main
```

4-Sample runs :

without arguments :

```
patrick@patrick-ASUS-TUF-Gaming-F15-FX506HE-TUF506HE:~/Desktop/Matrix Multiplication$ ./main
NO ARG
First Matrix Displayed:
1 2 3
6 7 8
11 12 13

Second Matrix Displayed:
1 2 3
8 9 10
13 14 15

A thread per matrix
56 62 68
166 187 208
276 312 348

Seconds taken: 0
MicroSeconds taken: 1

A thread per row
56 62 68
166 187 208
276 312 348

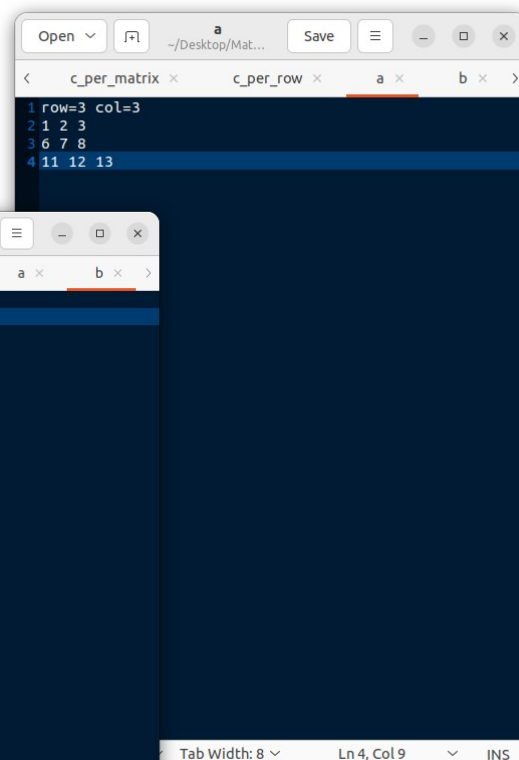
Seconds taken: 0
MicroSeconds taken: 282

A thread per element
56 62 68
166 187 208
276 312 348

Seconds taken: 0
MicroSeconds taken: 880
```

Files content :

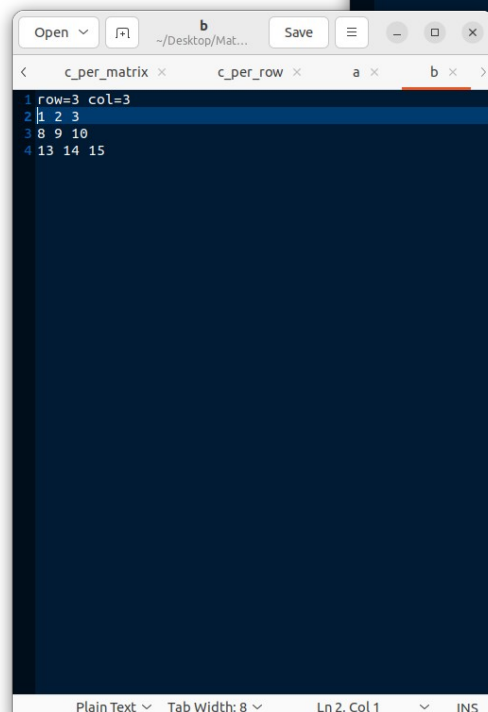
a.txt



The screenshot shows a text editor window with the following content:

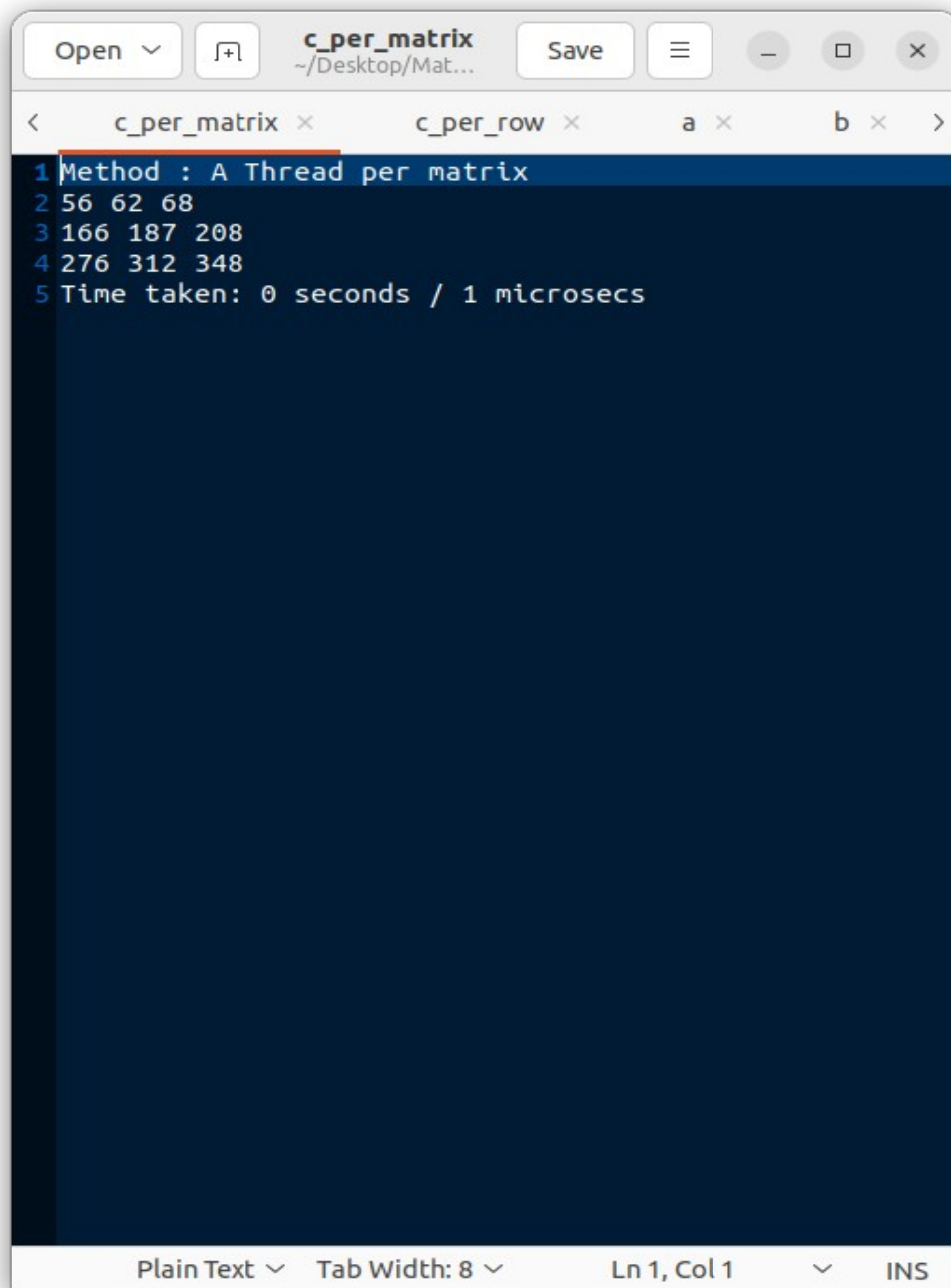
```
1 row=3 col=3
2 1 2 3
3 6 7 8
4 11 12 13
```

b.txt



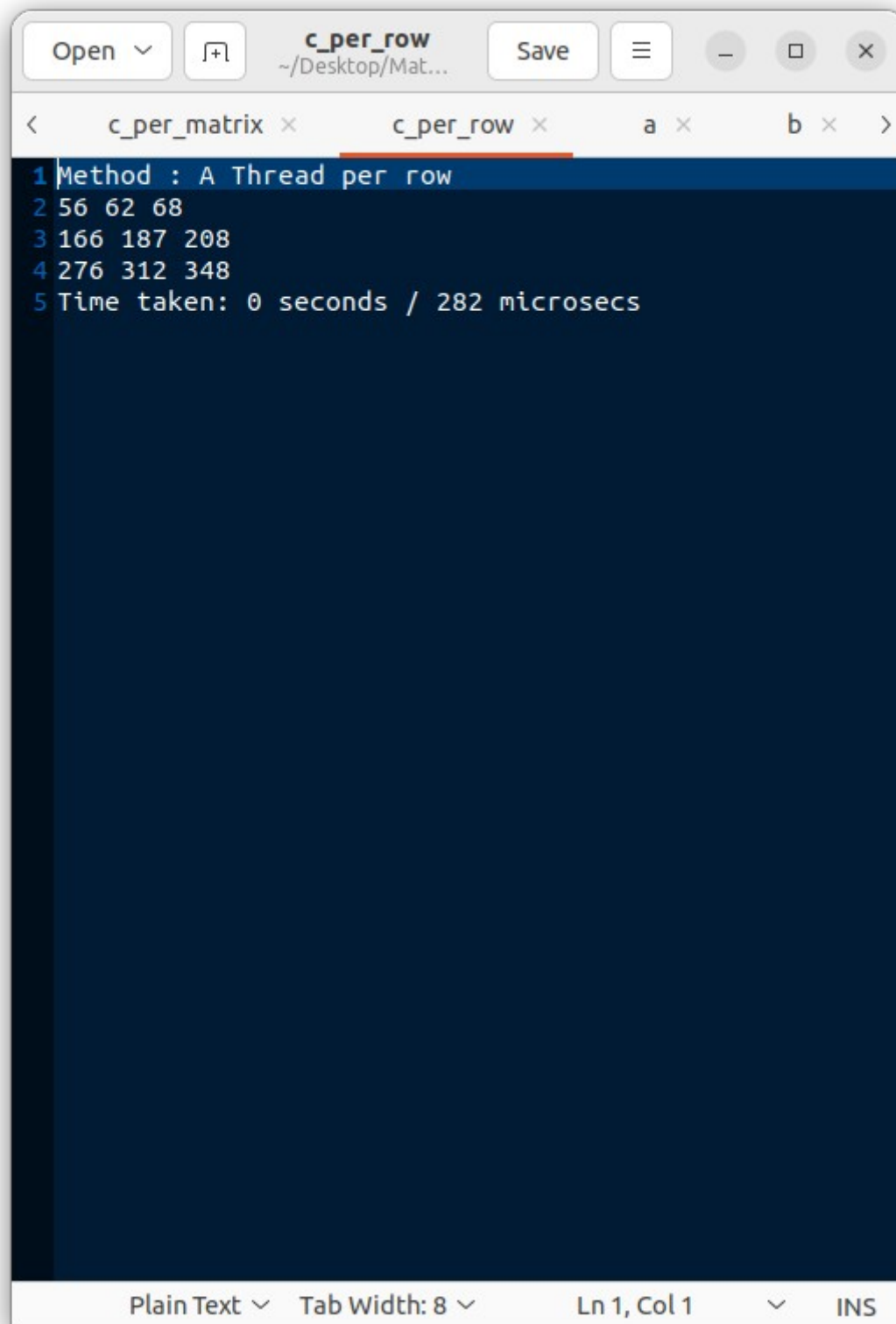
The screenshot shows a text editor window with the following content:

```
1 row=3 col=3
2 1 2 3
3 8 9 10
4 13 14 15
```

c_per_matrix.txt

The image shows a text editor window with a dark blue background. The window has a title bar with the text "c_per_matrix" and a path "~/Desktop/Mat...". Below the title bar, there are tabs for "c_per_matrix", "c_per_row", "a", and "b". The "c_per_matrix" tab is selected. The editor contains five lines of text, with the first line highlighted in blue. The status bar at the bottom indicates "Plain Text", "Tab Width: 8", "Ln 1, Col 1", and "INS".

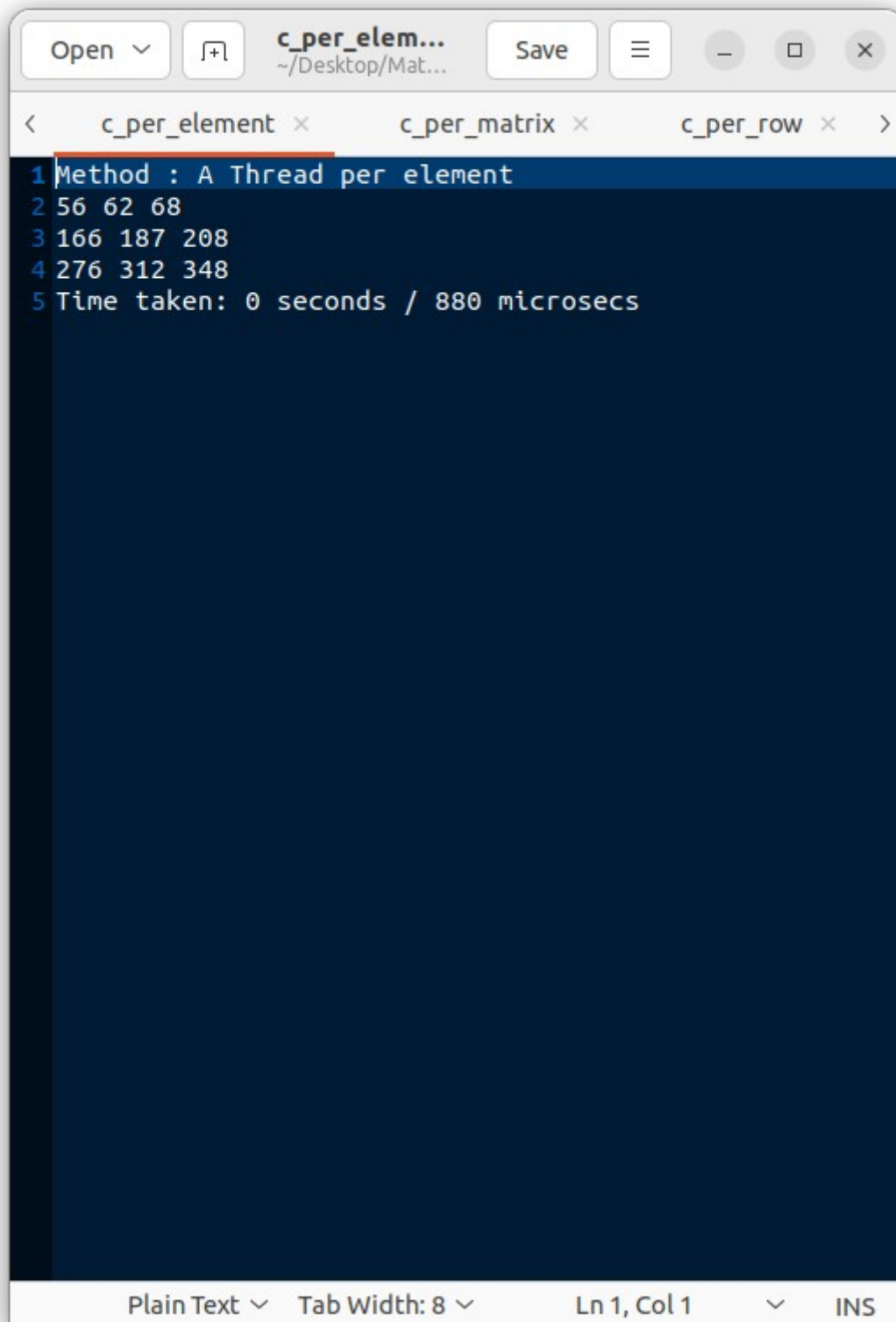
```
1 Method : A Thread per matrix
2 56 62 68
3 166 187 208
4 276 312 348
5 Time taken: 0 seconds / 1 microsecs
```

c_per_row.txt

The screenshot shows a code editor window with the title bar 'c_per_row' and a file path '~/.Desktop/Mat...'. The editor has tabs for 'c_per_matrix', 'c_per_row', 'a', and 'b'. The 'c_per_row' tab is active and contains the following text:

```
1 Method : A Thread per row
2 56 62 68
3 166 187 208
4 276 312 348
5 Time taken: 0 seconds / 282 microsecs
```

The status bar at the bottom indicates 'Plain Text', 'Tab Width: 8', 'Ln 1, Col 1', and 'INS'.

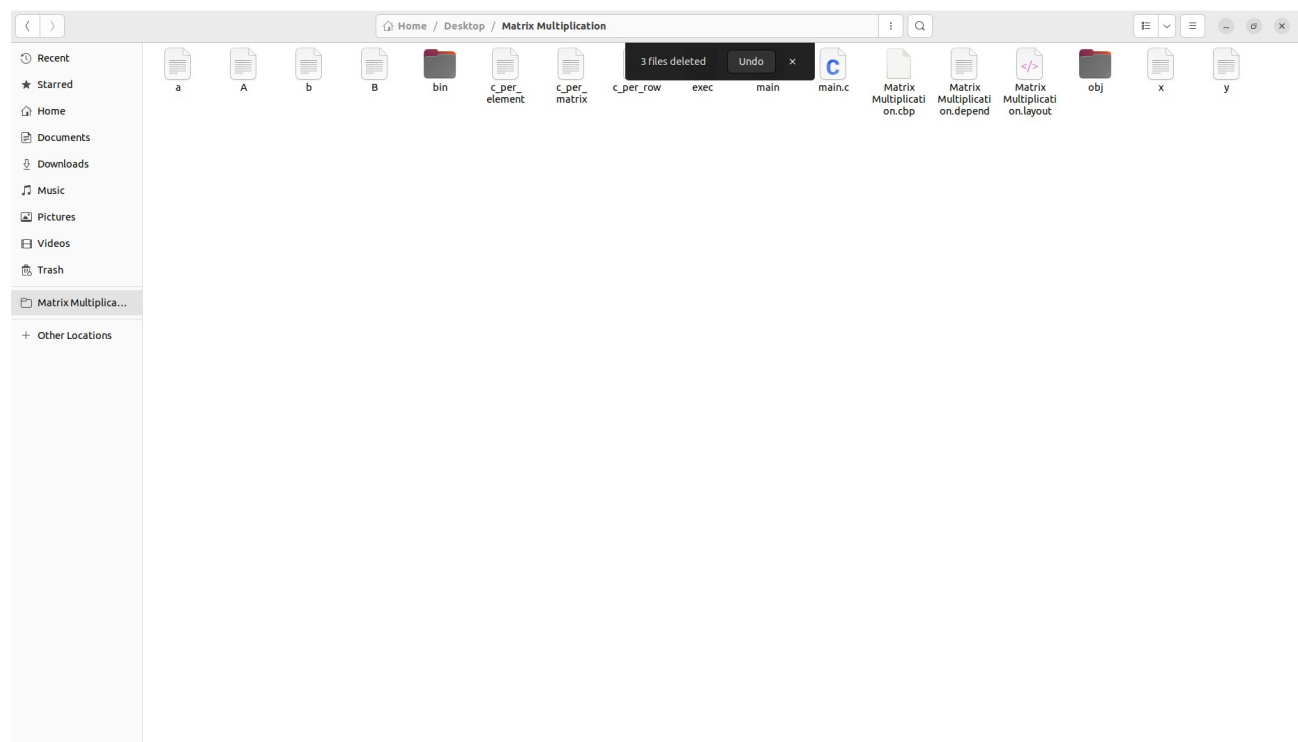
c_per_element.txt

The image shows a code editor window with a dark blue background. The window has a title bar with the text "c_per_elem..." and a file path "~/Desktop/Mat...". Below the title bar, there are three tabs: "c_per_element", "c_per_matrix", and "c_per_row". The "c_per_element" tab is selected. The editor contains five lines of text:

```
1 Method : A Thread per element
2 56 62 68
3 166 187 208
4 276 312 348
5 Time taken: 0 seconds / 880 microsecs
```

The status bar at the bottom of the window shows "Plain Text", "Tab Width: 8", "Ln 1, Col 1", and "INS".

With Arguments



```
patrick@patrick-ASUS-TUF-Gaming-F15-FX506HE-TUF506HE:~/Desktop/Matrix Multiplication$ ./main x y z
First Matrix Displayed:
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15

Second Matrix Displayed:
1 6 11
2 7 12
3 8 13
4 9 14
5 10 15

A thread per matrix
55 130 205
130 330 530
205 530 855

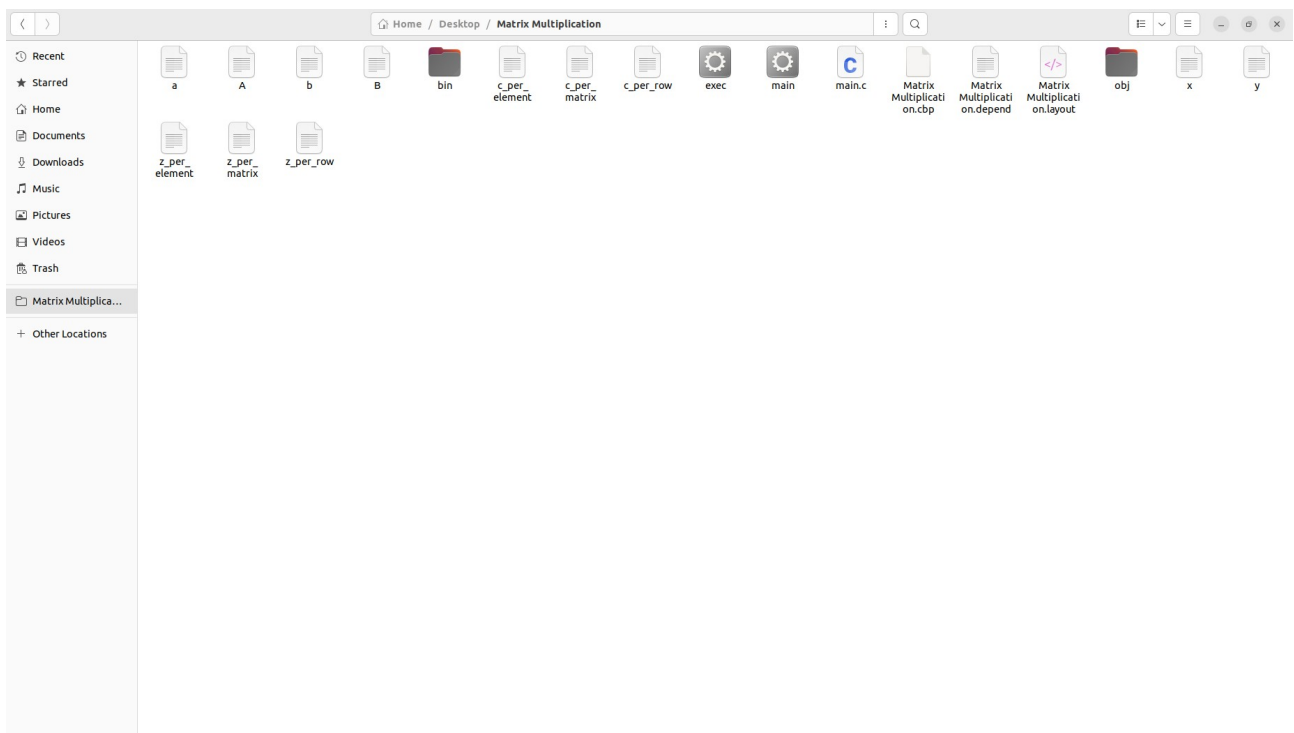
Seconds taken: 0
MicroSeconds taken: 1

A thread per row
55 130 205
130 330 530
205 530 855

Seconds taken: 0
MicroSeconds taken: 265

A thread per element
55 130 205
130 330 530
205 530 855

Seconds taken: 0
MicroSeconds taken: 713
```



Files Content

Open x ~/Desktop/Mat... Save

x x z_per_element x z_per_matrix x

1 row=3 col=5
2 1 2 3 4 5
3 6 7 8 9 10
4 11 12 13 14 15

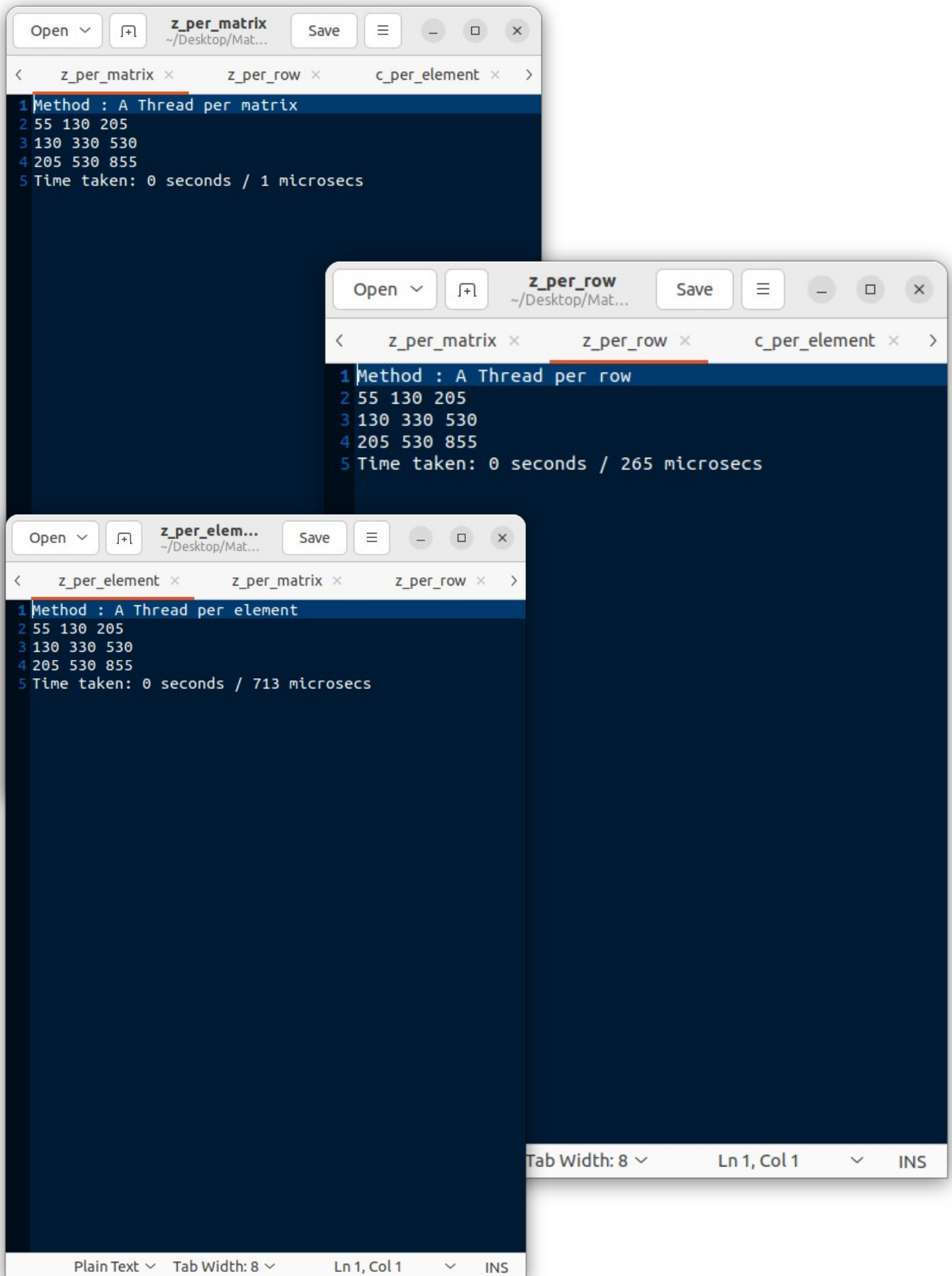
Plain Text Tab Width: 8 Ln 4, Col 1 INS

Open y ~/Desktop/Mat... Save

y

1 row=5 col=3
2 1 6 11
3 2 7 12
4 3 8 13
5 4 9 14
6 5 10 15
7

Loa... Plain Text Tab Width: 8 Ln 6, Col 3 INS



5-A comparison between the three methods of matrix multiplication :

<p>A thread per matrix</p> <p>This was the fastest method among the 3 methods. The idea is, creating and handling threads requires extra overhead and lots of computation, so, it's not always the ideal solution to go for multi-threading, and there's always a trade off.</p> <p>1 thread created for the whole matrix.</p>	<p>A thread per row</p> <p>It is an intermediate solution between the two methods. In case of large matrices It would be better to perform it using threads per row instead of performing the whole computation in one thread sequentially. It will be better to compute each row in parallel, but not to create a thread for each element because there will be much time consumed in creating and handling each thread. So we will be wasting the time saved due to parallelism in handling threads.</p> <p>N threads created $N = \# \text{ Rows of the resultant matrix} = \# \text{ Rows of the first matrix}$</p>	<p>A thread per element</p> <p>This method is the least efficient because of the overhead of creating and handling threads and with larger matrices the number of threads multiplies and we would lose the advantage of parallelism due to threads in exchange with the overhead of creating this number of threads. This method will be somehow efficient with small matrices.</p> <p>$N \times M$ threads created $N = \# \text{ Rows of the resultant matrix} = \# \text{ Rows of the first matrix}$ $M = \# \text{ Cols of the resultant matrix} = \# \text{ Cols of the second matrix}$ $N \times M = \# \text{ of elements of the resultant matrix}$</p>
--	--	---

6 – Link to video :

[Test Cases Video](#)