

به نام خدا



گزارش پروژه نهایی درس ساختار و زبان کامپیوتر

دکتر امیرحسین جهانگیر

نیما موزن (۴۰۱۱۰۶۵۹۹)

زمستان ۱۴۰۲













## مقدمه و معرفی کلی

پروژه ما به طور کلی دارای دو فاز است. در فاز اول با استفاده از اسمبلی ۸۰۸۶، ابتدا ضرب دو ماتریس  $n \times n$  و سپس convolution آنها را با روش ضرب نظیر به نظیر انجام می دهیم. هر دو عملیات را یک بار به روش عادی و یک بار به روش موازی انجام داده و زمان انجام آنها را با هم مقایسه می کنیم. هدف از انجام این پروژه نشان دادن این است که روش موازی حداقل دوبرابر سریعتر از روش عادی است.

در فاز دوم اما، باید با استفاده از یک تابع convolution متمایز با قبلی، یک کاربرد آن را در دنیای واقعی نشان دهیم. انتخاب من برای این کاربرد پردازش تصویر است. همچنین باید این کاربرد را یک بار با استفاده از کد اسمبلی خود و یک بار با استفاده از کد زبان سطح بالا انجام دهیم و نشان دهیم کد اسمبلی ما سریعتر است و خریدار نسبت به خریدن این برنامه مشتاق تر است. ساختار کلی این گزارش به این شکل است که هر فاز به صورت جداگانه ابتدا نحوه استفاده از برنامه توضیح داده شده و سپس مقداری وارد جزئیات نیز می شود.

# فاز اول

برای استفاده از این بخش ابتدا وارد پوشه phase1 می شویم.

Name	Date modified	Type	Size
 asm_io.asm	1/30/2024 10:19 PM	Assembler source ...	2 KB
 asm_io.inc	1/30/2024 10:29 PM	INC File	1 KB
 asm_io.o	2/11/2024 1:40 AM	O File	2 KB
 code	2/11/2024 1:40 AM	File	23,456 KB
 code.asm	2/10/2024 4:43 PM	Assembler source ...	14 KB
 code.o	2/11/2024 1:40 AM	O File	23,444 KB
 driver.c	12/18/2023 12:14 AM	C Source File	1 KB
 driver.o	2/11/2024 1:40 AM	O File	2 KB
 result.txt	2/11/2024 1:40 AM	Text Document	25,393 KB
 run.sh	12/18/2023 12:14 AM	Shell Script	1 KB
 test_case_maker.py	2/11/2024 1:38 AM	Python File	1 KB
 test-case.txt	2/11/2024 1:38 AM	Text Document	7,813 KB

برنامه پایتونی با اسم test-case-maker.py را می بینیم. آن را باز کرده و اجرا می کنیم. این برنامه به ما تست کیس مناسبی برای استفاده در کد اسمبلی ما می دهد. خروجی این برنامه دو ماتریس با اندازه n و همچنین ماتریس حاصل ضرب آنهاست که به چک کردن جواب خودمان کمک می کند. این برنامه دو ماتریس را به فرمت ورودی برنامه اسمبلی ما وارد فایل test-case.txt می کند و ما با ورودی دادن این فایل به برنامه و ریختن خروجی در result.txt از برنامه استفاده می کنیم.

```

test_case_maker.py > ...
1  # this code makes two n*n matrices and writes them in test-case.txt in our input format
2  # after that it writes the real answer of matrix1 * matrix2 which we must see in result.txt
3
4  import numpy as np
5
6  file = open('phase1\\test-case.txt', "a")
7
8  n = 1000
9  A = np.random.randint(10, size=(n,n))
10 B = np.random.randint(10, size=(n,n))
11
12
13
14 print(A)
15 print(B)
16
17 # print(n)
18 file.write(f"{n}\n")
19 for a in A:
20     for b in a :
21         file.write(f"{b} ")
22 file.write("\n")
23 for b in B:
24     for a in b :
25         file.write(f"{a} ")
26 print()
27 C = np.dot(A,B)
28 file.close()
29 print(C)
30

```

```

[8 2 6 ... 8 0 8]
[7 6 8 ... 9 1 3]
[5 0 6 ... 1 3 4]
...
[1 3 2 ... 8 5 3]
[9 2 6 ... 4 6 6]
[2 9 1 ... 2 4 1]]
[[9 5 3 ... 7 2 5]
[0 8 1 ... 8 2 1]
[0 9 2 ... 8 8 3]
...
[9 8 7 ... 6 8 7]
[6 4 2 ... 5 9 4]
[6 6 0 ... 3 6 1]]

[[20515 20023 19883 ... 21460 19966 20303]
[20736 19917 19534 ... 20931 19772 20078]
[20976 20211 20681 ... 20713 20127 21415]
...
[19841 19713 19373 ... 20808 19547 19953]
[20065 19986 19150 ... 20654 19106 20456]
[20187 19557 19443 ... 20607 19349 20125]]

```



```
result.txt
993 9525.000000 9793.000000 9834.000000 10005.000000 9465.000000 9374.000000 9529.000000 9051.000000 9430.000000 10318.000000 10008.0
994 9986.000000 10069.000000 10230.000000 10170.000000 9808.000000 9747.000000 10518.000000 9927.000000 9921.000000 10561.000000 1011
995 9491.000000 9869.000000 9761.000000 10038.000000 9435.000000 8988.000000 9704.000000 9239.000000 9550.000000 10051.000000 9796.00
996 10303.000000 10394.000000 10038.000000 10307.000000 10005.000000 9762.000000 10144.000000 9614.000000 10148.000000 10420.000000 1
997 9869.000000 10162.000000 10013.000000 9962.000000 9753.000000 8899.000000 10117.000000 9852.000000 9749.000000 10361.000000 10226
998 9995.000000 10502.000000 10328.000000 10692.000000 10410.000000 10002.000000 10720.000000 10285.000000 9999.000000 10685.000000 1
999 9917.000000 10296.000000 10547.000000 10128.000000 10034.000000 8872.000000 9988.000000 9381.000000 9854.000000 10604.000000 9938
1000 10415.000000 10883.000000 10627.000000 10550.000000 10550.000000 10171.000000 11277.000000 10305.000000 10645.000000 11024.000000
1001 10126.000000 10450.000000 10873.000000 10545.000000 10199.000000 9885.000000 10644.000000 9789.000000 10115.000000 10626.000000 1
1002
1003 389674012
1004 271815261
1005
1006 5077530.000000
1007 5077530.000000
1008
1009 793891
1010 385235
1011
1012 |

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS
nimanm7@DESKTOP-VS22VKG: /mnt/c/Users/ASUS/Documents/University/term3/language_model/final_project/phase1$ time ./run.sh code < test-case
.txt > result.txt
real    0m1.172s
user    0m0.625s
sys     0m0.391s
nimanm7@DESKTOP-VS22VKG: /mnt/c/Users/ASUS/Documents/University/term3/language_model/final_project/phase1$
```

همانطور که می بینید از آنجا که  $n$  ما ۵۰۰ است در ۱۰۰۱ خط اول دو ماتریس نوشته شده است. اولی حاصل ضرب دو ماتریس به روش عادی و دومی حاصل ضرب دو ماتریس به روش موازی است که می توان دید هر دو ماتریس با هم و با ضرب به دست آمده با کتابخانه های آماده پایتون سازگارند.




















در خطوط بعدی دو عدد آمده که زمان اجرای کد عادی و سپس موازی است (تعداد کلاک های بین شروع و پایان عملیات) که مشخص است کد موازی از سرعت بیشتری بهره می برد. سپس در دو خط بعدی نتیجه convolution دو ماتریس به روش عادی و موازی نوشته شده که مقادیر برابرند و بعد از آن نیز مثل ضرب، زمان اجرای دو برنامه نوشته شده که باز هم حالت موازی بسیار سریعتر است و در کاربرد های واقعی این اختلاف سرعت به چشم می آید و باعث موفقیت بیشتر برنامه از نظر اجرا و فروش می شود.

## نحوه پیاده سازی

کد نوشته شده برای این بخش در code.asm در دسترس است. برای بخش ضرب عادی، الگوریتمی که همیشه برای ضرب ماتریس استفاده می کنیم استفاده شده است. یعنی سطر های ماتریس اول و ستون های ماتریس دوم باهم ضرب برداری می شوند و جواب این ضرب برداری در خانه متناظر در ماتریس سوم ذخیره می شود. برای بخش موازی نیز همین عملیات را داریم، با این تفاوت که در این بخش ضرب برداری گفته شده به روش موازی انجام می گیرد، یعنی هر چهار عنصر بردار همزمان در چهار عنصر بردار دیگر ضرب می شوند و این موازی سازی که همواره از اصول افزایش سرعت برنامه ها بوده، نقش بسزایی در سریعتر شدن برنامه مان دارد. برای محاسبه convolution نیز از ضرب نظیر به نظیر استفاده شده است. در بخش عادی عناصر ماتریس را دانه دانه مثل کاری که انسان هم قادر به انجامش است انجام می دهیم و سپس در بخش موازی عناصر را چهارتا چهارتا در ثبات های برداری می ریزیم و ضرب برداری را انجام داده و حاصل آنها را جمع می کنیم. لازم به ذکر است برای پیاده سازی هر دو فاز از تابع های آماده ای که برای مسابقه asm masters توسط آقای میرزایی پیاده سازی شده بود استفاده شده است. کامنت های لازم در کنار همه تکه کدهای code.asm نوشته شده است و با مطالعه آنها می توان درک کاملی از چگونگی انجام شدن برنامه پیدا کرد.

## فاز دوم

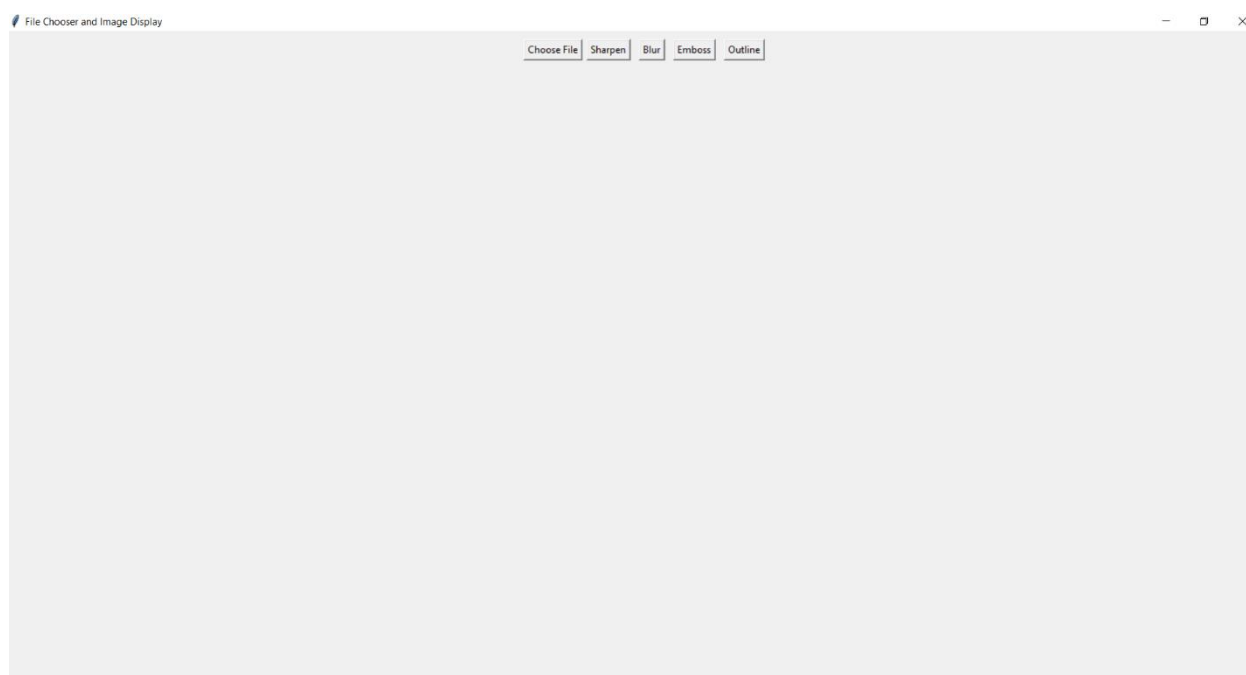
برای استفاده از این بخش وارد پوشه phase2 می شویم.

documents > University > term3 > language model > final_project > phase2			
Name	Date modified	Type	Size
 asm_io.asm	1/30/2024 10:19 PM	Assembler source ...	2 KB
 asm_io.inc	1/30/2024 10:29 PM	INC File	1 KB
 asm_io.o	2/11/2024 1:21 AM	O File	2 KB
 driver.c	12/18/2023 12:14 AM	C Source File	1 KB
 driver.o	2/11/2024 1:21 AM	O File	2 KB
 gui.py	2/11/2024 1:22 AM	Python File	6 KB
 image_assembly.jpg	2/11/2024 1:21 AM	JPG File	598 KB
 image_assembly.txt	2/11/2024 1:21 AM	Text Document	9,644 KB
 image_python.jpg	2/11/2024 1:21 AM	JPG File	598 KB
 image_python.txt	2/11/2024 1:21 AM	Text Document	4,781 KB
 image1.jpg	12/15/2022 11:54 PM	JPG File	211 KB
 image1.txt	2/11/2024 1:21 AM	Text Document	3,604 KB
 image2.txt	2/10/2024 5:39 PM	Text Document	13,269 KB
 logic.py	2/11/2024 12:05 AM	Python File	4 KB
 main_code	2/11/2024 1:21 AM	File	7,831 KB
 main_code.asm	2/10/2024 11:36 PM	Assembler source ...	5 KB
 main_code.o	2/11/2024 1:21 AM	O File	7,817 KB
 run.sh	12/18/2023 12:14 AM	Shell Script	1 KB
 tempCodeRunnerFile.py	2/11/2024 1:20 AM	Python File	0 KB

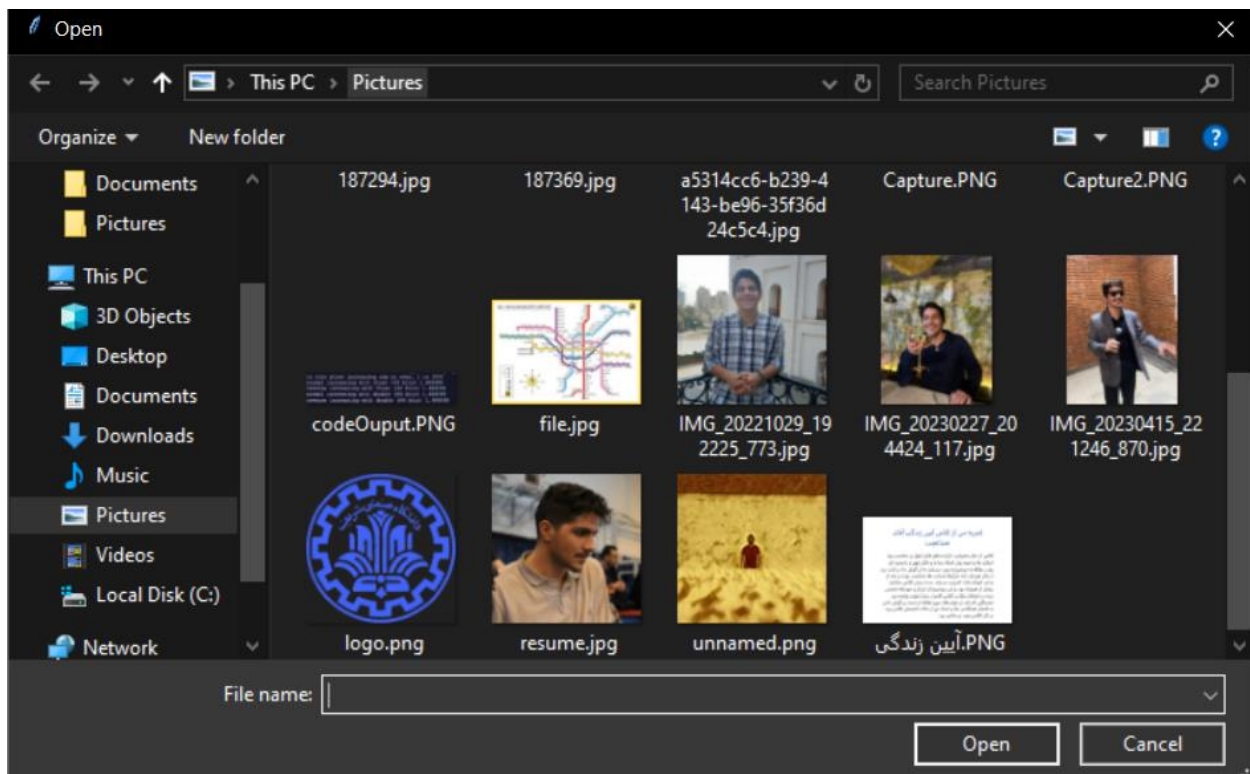
در این پوشه کد main\_code.asm دارای کد اسمبلی اصلی است که یک  $n$  و یک ماتریس به سباز  $n$  که همان تصویر ماست و یک ماتریس  $3$  در  $3$  که پردازشگر ماست و مشخص می کند تصویر ما به چه حالتی پردازش شود به عنوان ورودی می گیرد. این ورودی باید در فایل image1.txt ذخیره شود.



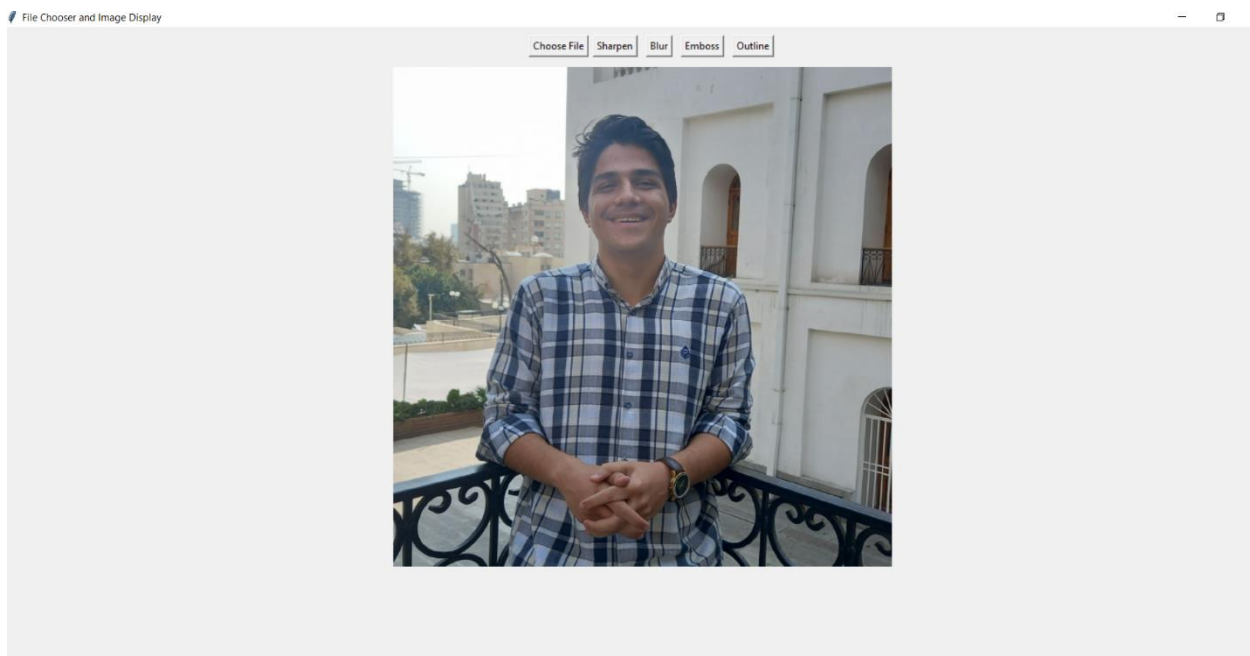
سپس این کد با استفاده از convolution ماتریس ۹۹۸ در ۹۹۸ تصویر پردازش شده را در image\_assembly.txt ذخیره می کند. اما برای استفاده کلی ما از برنامه ، اینترفیس گرافیکی ساده ای تهیه شده تا تمام فعالیت های لازم با چند کلیک قابل انجام باشند. برنامه gui.py همان ورژن گرافیکی برنامه logic.py است که هردو در پوشه دیده می شوند. آموزش تصویری کار با آن در ادامه آمده است.



شکل کلی اپلیکیشن ما به این صورت است. با کلیک روی دکمه choose file می توانیم تصویر مورد نظرمان را از حافظه انتخاب کنیم.



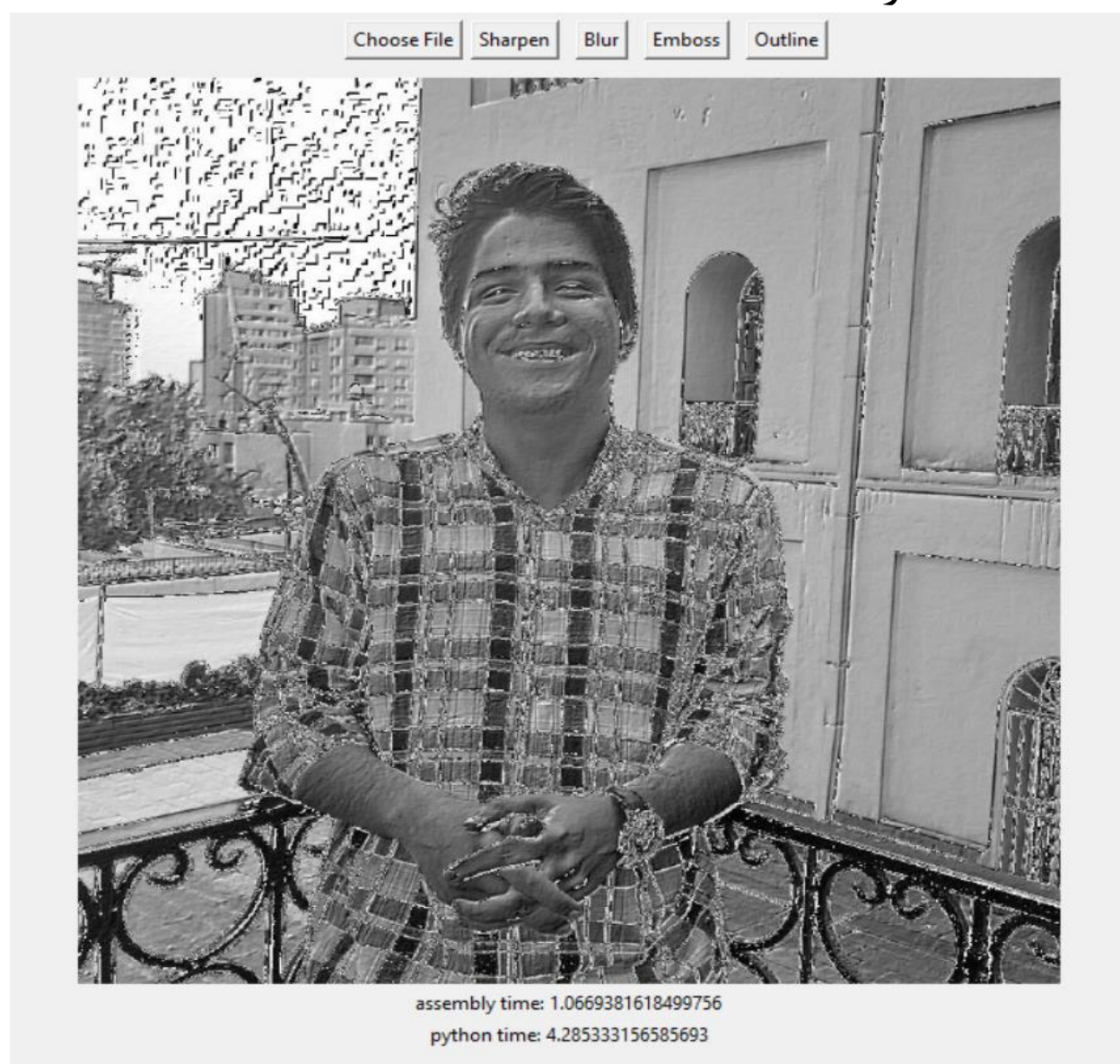
سپس تصویر مورد نظرمان را انتخاب کرده و آن را در برنامه مشاهده می کنیم.



حال که تصویرمان را انتخاب کردیم می توانیم با استفاده از چهار دکمه ای که قرار داده شده، ماتریس پردازشگر را انتخاب کنیم. با کلیک روی هرکدام از آنها بعد از تقریباً پنج ثانیه تصویر پردازش شده جای تصویر اصلی را می گیرد و در پایین آن هم زمان پردازش آن با کد اسمبلی و کد پایتون نوشته می شود تا بتوان راحت تر مقایسه کرد. برای مثال در صورت کلیک بر روی گزینه sharpen:



و علاوه بر نشان دادن تصویر، تصاویر و ماتریس های ساخته شده در فایل های image\_python.jpg، image\_assembly.txt، image\_assembly.jpg و image\_python.txt نیز ذخیره می شوند. بدیتهای ماتریس تصویر اصلی نیز در فایل image1.txt ذخیره می شود و می توان گفت اینترفیس گرافیکی ما تمام کارهای لازم که در بخش توضیح پیاده سازی بیشتر به آن می پردازیم را انجام می دهد. در پایین تصاویری از دیگر مدل های پردازش یعنی به ترتیب blur، emboss و outline آمده است:



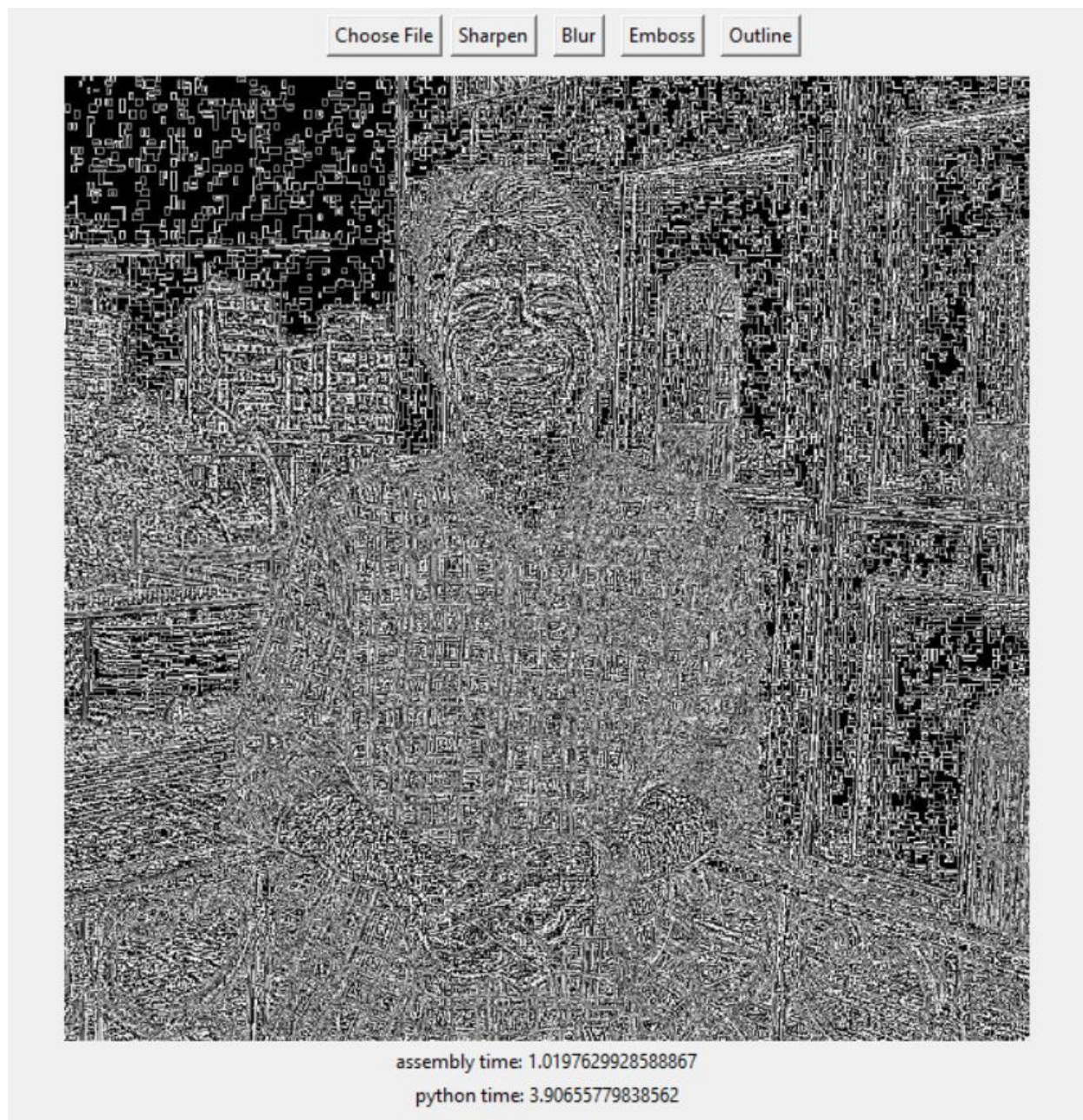


Choose File Sharpen Blur Emboss Outline



assembly time: 1.1639511585235596

python time: 4.004271984100342



که در همگی آنها همانطور که دیده می شود برنامه اسمبلی در حدود یک ثانیه تمام عملیات ورودی گرفتن و خروجی دادن و پردازش تصویر را انجام می دهد (که حدود ۰/۲ ثانیه آن عملیات پردازش تصویر است که سرعت وحشتناکی است) در حالیکه کد پایتون حداقل چهار ثانیه طول می کشد. زمانیکه کاربران واقعی بخواهند از برنامه استفاده کنند همین اختلاف زمانی در سطح رضایتشان



از برنامه بسیار موثر است. برای اینکه تفاوت تجربه استفاده از اسمبلی و کد سطح بالا را نشان بدهیم دکمه luck را اضافه می کنیم که یکی از پردازشگرها را به طور تصادفی انتخاب می کند و سپس فقط فرایند کد اسمبلی را انجام می دهد. این پروسه با وجود کندی هایی که گرفتن عکس، قرار دادن عکس و کار با فایل پایتون دارد، با سرعت خیلی بیشتر عکس پردازش شده را تحویل ما می دهد که تجربه خوشایندتری است و هر کاربری کار با این دکمه را به دیگر دکمه ها ترجیح می دهد.



ذخیره شدن عکس ها و ماتریس ها نیز در دو شکل زیر نشان داده شده است.

[illegible]



همانطور که گفته شد می توان با استفاده از کد اسمبلی در ترمینال نیز بخش اسمبلی را انجام داد و با دادن تصویر در فرمت فایل txt فایلی با فرمت txt شامل تصویر پردازش شده تحویل گرفت. البته ورودی و خروجی های کوچک را در ترمینال نیز می توان انجام داد.

```
nimanm7@DESKTOP-VS22VKG:/mnt/c/Users/ASUS/Documents/University/term3/language_model/final_project/phase2$ ./run.sh main_code
4
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 1 1 0 0 0 0 1 1
27.000000 32.000000
47.000000 52.000000
nimanm7@DESKTOP-VS22VKG:/mnt/c/Users/ASUS/Documents/University/term3/language_model/final_project/phase2$ ./run.sh main_code <image1.txt
> image_assembly.txt
nimanm7@DESKTOP-VS22VKG:/mnt/c/Users/ASUS/Documents/University/term3/language_model/final_project/phase2$ time ./run.sh main_code <image
1.txt> image_assembly.txt

real    0m0.932s
user    0m0.453s
sys     0m0.375s
nimanm7@DESKTOP-VS22VKG:/mnt/c/Users/ASUS/Documents/University/term3/language_model/final_project/phase2$
```

همچنین می توان با استفاده از فایل logic.py تمام عملیات را با نوشتن دستی ماتریس پردازشگر و انتخاب عکس از حافظه انجام داد. این کد خروجی ای به شکل زیر دارد که به وسیله آن می توان تصاویر ساخته شده توسط کد اسمبلی و همچنین سرعت آنها را باهم مقایسه کرد.

```
python image matrix:
[[253. 253. 253. ... 158. 158. 158.]
 [253. 253. 253. ... 155. 155. 155.]
 [253. 253. 253. ... 157. 157. 157.]
 ...
 [139. 144. 147. ... 86. 85. 91.]
 [144. 145. 145. ... 87. 89. 95.]
 [146. 148. 142. ... 84. 88. 94.]]

assembly image matrix:
[[253. 253. 253. ... 158. 158. 158.]
 [253. 253. 253. ... 155. 155. 155.]
 [253. 253. 253. ... 157. 157. 157.]
 ...
 [139. 144. 147. ... 86. 85. 91.]
 [144. 145. 145. ... 87. 89. 95.]
 [146. 148. 142. ... 84. 88. 94.]]

assembly run time: 0.6764321327209473
python run time: 5.0598978996276855
```

## نحوه پیاده سازی

پیاده سازی تابع convolution لازم برای پردازش تصویر مهم ترین بخش این فاز است. ایده کلی آن از سایت [/https://setosa.io/ev/image-kernels](https://setosa.io/ev/image-kernels) گرفته شده است. به این گونه که ما برای تمام عناصر ماتریس عکس اصلی، یک ماتریس ۳ در ۳ حولشان در نظر می گیریم. سپس convolution این ماتریس ها را با ماتریس ۳ در ۳ مشخص شده توسط خودمان، که آن را پردازشگر صدا می زنیم، حساب می کنیم و در نقطه نظیر نقطه اول در ماتریس تصویر نهایی قرار می دهیم. برای افزایش سرعت در زبان اسمبلی، این ضرب نظیر به نظیر نقاط دو ماتریس را با استفاده از روش موازی پیاده سازی می کنیم. به گونه ای که برای هر سطر ماتریس ها یک ضرب برداری موازی انجام می دهیم و سپس جمع می کنیم که حتی پیاده سازی ساده تری نسبت به حالت عادی دارد. ماتریس نهایی در فایل image\_assembly.txt ذخیره می شود و می توان با استفاده از زبان های سطح بالایی مثل پایتون آن را به تصویر تبدیل کرد، همانگونه که ابتدا برعکس این فرایند را با استفاده از همین زبان انجام داده بودیم و تصویر را به ماتریس ۱۰۰۰ در ۱۰۰۰ تبدیل و سپس در فایل image1.txt ریخته بودیم. مهم ترین بخش کد اسمبلی همان تابعی است که ذکر شد که تصویرش هم آمده است و بقیه بخش های کد نیز به فاز اول تشابه دارد و با وجود کامنت های فراوان می توان درکش کرد.

```

calculate_convolution:
    mov ebp, 0 ; this is the output of the subroutine
    movd xmm2, ebp
    mov r14, 0
convolution_loop:
    ; access matrix1[r12+r14-1][r13-1] which is the first element of each row of 3x3 matrix and put it in xmm0
    mov rbx, r12
    add rbx, r14
    dec rbx
    imul rbx, [n]
    add rbx, r13
    dec rbx
    movups xmm0, matrix1[rbx*4]

    ; access matrix2[r14][0], the first elements of each row of image processor matrix and put it in xmm1
    mov rbx, r14
    imul rbx, 3
    movups xmm1, matrix2[rbx*4]

    ; xmm0[0] = xmm0[0] * xmm1[0] + xmm0[1] * xmm1[1] + xmm0[2] * xmm1[2]
    dpps xmm0, xmm1, 0x71

    ; add every row result to the main answer
    addss xmm2, xmm0

    ; increment counter and do other loop-related stuff
    inc r14
    cmp r14, 2
    jle convolution_loop

; mov xmm2 to our main output
movd ebp, xmm2
ret

```

برنامه اصلی زبان سطح بالای ما نیز، ابتدا یک تصویر و یک ماتریس پردازشگر از ما می گیرد، آن را به شکل فایل متنی ذخیره می کند و سپس فرایندی مشابه زبان اسمبلی را با زبان پایتون انجام می دهد. ماتریس به دست آمده از کد اسمبلی و پایتون در فایل های متنی به نام های `image_assembly.txt` و `image_python.txt` ذخیره می شوند و سپس با استفاده از کد های پایتون لازم، این ماتریس ها به تصویر تبدیل شده و در فایل های تصویری `image_python.jpg` و `image_assembly.jpg` ذخیره می شوند.