به نام خدا

محمد فرحان بهرامی

401105729

نام استاد : دکتر جهانگیر

پاسخ تمرین ساختار زبان

پروژه برنامه نویسی اسمبلی

حافظا علم و ادب ورز که در مجلس شاه
هر که را نیست ادب لایق صحبت نبود

فاز اول

1- برنامه اسمبلی ضرب دو ماتریس nxn اعداد ممیز شناور 32 بیتی به روش معمولی، یعنی ضرب و جمع متوالی درایه‌های متناظر دو ماتریس و مقایسه سرعت اجرای این برنامه با استفاده از دستورات برداری (موازی) پردازنده را برای پردازنده x8680 در دو حالت n=3 و n=5 بنویسید و با هم مقایسه کنید. همین طور با زمان اجرای برنامه‌ی ضرب ماتریس به زبان سطح بالا مقایسه و نقد کنید. (کتاب Modern X86 Assembly Language Programming آقای Daniel Kusswurm مرجع خوبی است و مثالهای مشابه دارد).

ورود مقادیر هر ماتریس را میتوانید به کمک یک زبان سطح بالاتر انجام دهید و برنامه اسمبلی خود را که داخل آن تعبیه کرده‌اید فرابخوانید. خروجی برنامه را نیز میتوانید با همان زبان سطح بالا نشان دهید و با حاصل یک برنامه سطح بالا مقایسه کنید و نمایش دهید تا از صحت برنامه شما بتوان اطمینان حاصل کرد.

در این قسمت ، در 3 حالت مختلف ضرب خارجی یک ماتریس مورد مقایسه قرار میگیرد.

در فایل Multiply.asm ضرب خارجی به زبان اسمبلی در حالت عادی نوشته شده است که در قسمت کامنت ها کاملا کارکرد و روند کد مشخص شده است، در فایل ParallelMul.asm این ضرب بصورت موازی یعنی از طریق دستورات SIMD : Single Instruction Multiple Data ضرب انجام میشود که همانند فایل قبلی توسط کامنت ها مشخص شده است. و در نهایت این ضرب به زبان سطح بالایی مانند پایتون در فایل HLMul.py نوشته شده است که هدف اصلی ما نشان دادن تفاوت بارز بین زبان های سطح پایین و بالا در حالت هایی که مقادیر یا تکرر کد زیاد است، این تفاوت بارز است.

در فایل HLMul.py بصورت بدیهی ورودی سایز ماتریس n*n را می دهیم در نهایت 2 ماتریس 2بعدی n*n میدهیم که خروجی آن نتیجه ضرب ماتریس و زمان مصرفی میباشد. ما عملیت ضرب را به اندازه 10^6 بار انجام می دهیم تا بتوانیم تایم را بهتر نمایش دهیم.

```python
import time

start = time.time()
n = int(input())
A = []
B = []
max = 1000000
for i in range(n):
    row = list(map(float, input().split()))
    A.append(row)

for i in range(n):
    row = list(map(float, input().split()))
    B.append(row)

C = []
for _ in range(n):
    C.append(([0]*n))
for _ in range(max):
    for i in range(n):
        for j in range(n):
            temp = 0
            for k in range(n):
                temp += A[i][k] * B[k][j]
            C[i][j] = temp

for i in range(n):
    for j in range(n):
        print("%.2f" % C[i][j], end=" ")
    print()
end = time.time()
print(end - start)
```

اما در زبان اسمبلی آرایه 2 بعدی نداریم، اما بجای این کار ، یکار آرایه به سایز n*n*2 میگیریم ، و همانطور که میدانیم در ماتریس A اگر درایه a[i][j] را بخواهیم میتوان از طریق یک ارایه یک بعدی به صورت a[n*i + j] به آن دسترسی پیدا کرد و همچنین میدانیم که شیفت دادن یک عدد بسیار راحت و سریع تر از ضرب است، در نتیجه ابعاد ماتریس را بصورت توان هایی از 2 در نظر میگیریم که از سایز n بیشتر باشد، و هنگام دسترسی به درایه ها با شیفت دادن به درایه مورد نظر میرسیم.

برای ورودی و خروجی گرفتن در اسمبلی بعد از اسمبل کردن کد از طریق nasm با استفاده از gcc آن را لینک میکنیم تا بتوانیم ورودی و خروجی را مانند زبان C از تابع های printf & scanf & ... استفاده کنیم.

در عکس زیر نحوه ورودی و خروجی دادن به کمک gcc میباشد.

```
read_float: ; for get input as float
    sub rsp, 8

    mov rsi, rsp
    mov rdi, read_float_format ; move float format to read in rdi
    mov rax, 1
    call scanf ; scan input
    movss xmm0,[rsp] ; move input into xmm0
    add rsp, 8
    ret

 print_float: ; for print the result as flaot
   sub rsp, 8
   cvtss2sd xmm0, xmm0 ; convert the result that stored in xmm0 to be printable
   mov rdi,print_float_format ; move float format to print in rdi
   mov rax, 1
   call printf
   add rsp, 8
   ret
```

در نهایت در فایل Multiply.asm بعد از ورودی گرفتن سایز ماتریس ما یک loop 2 بعدی میزنیم تا ماتریس اول و بار دیگر ماتریس دوم را بگیریم و در حافظه ذخیره کنیم و چون قرار بود 32 بیتی باشد، از رجیستر ها ، eax, ebx & ... استفاده میکنیم تا بتوانیم مقادیر 32 استفاده کنیم و در نهایت یک لوپ 3 بعدی میزنیم تا ضرب ماتریسی را انجام دهیم و بطور موقت در حافظه ذخیره کنیم و بعد از محاسبه آن را پرینت کنیم.

در عکس زیر داخلی ترین لوپ ضرب مشخص است.

```
    xor r14d,r14d ; inc j = 0
MUL_LOOP2: ; this loop is for j
    mov eax,[n] ; copy size of matrix in eax
    cmp eax,r14d ; compare eax , r14d to jump if needed
    jle MUL_ENDLOOP2 ; if n<=j break and run rest of code

    xor r15d, r15d ; int k = 0
    pxor xmm0, xmm0 ; define temp sum for each element of C[i][j] as 0 to sum all multiplys
MUL_LOOP3: ; this loop is for k
    mov eax,[n] ; copy size of matrix in eax
    cmp eax,r15d ; compare eax , r15d to jump if needed
    jle MUL_ENDLOOP3 ;if n<=j break

    ;to load the value of A[i][k]
    mov ecx, r13d
    sal ecx, 3 ; i << 3 : to find the value A[8*i + k]
    add ecx, r15d
    sal ecx, 2 ; ecx << 2: because our values are dword and 32 bit so we need to skip 4 byte of memory to store each value
    movss xmm1, A[ecx] ; load the value of A[i][k] into xmm1

    ;to load the value of B[k][j] and multiply it by A[i][k]
    mov ecx, r15d
    sal ecx, 3 ; k << 3 : to find the value B[k*8 + j]
    add ecx, r14d
    sal ecx, 2 ; ecx << 2: because our values are dword and 32 bit so we need to skip 4 byte of memory to store each value
    mulss xmm1,B[ecx] ; multiply B[k][j] by A[i][k] and store the result in xmm1
    addss xmm0, xmm1 ; add xmm1 to last sum of multiplys in xmm0

    inc r15d ; k++
    jmp MUL_LOOP3 ; does loop again
MUL_ENDLOOP3:
```

در فایل ParallelMul.asm همین روند انجام شده است ، فقط بجای اینکه از دستورات ضرب و انتقال و جمع تکی، بصورت موازی استفاده شده و اعداد را 4تا 4تا ضرب و جمع میکنیم، چون رجیستر xmm 128 بیت است و اعداد ما 32 بیتی اند، در نتیجه ما میتوانیم تعداد را 4تا 4تا محاسبه کنیم.

همانطور که در تصویر زیر مشخص است بجای دستوراتی ساده مانند mov, mul & ... از دستوراتی که موازی کار میکنند مانند movups, mulps & ... استفاده میکنیم.

```
MUL_LOOP3: ; this loop is for k
    mov eax,[n] ; copy size of matrix in eax
    cmp eax,r15d ; compare eax , r15d to jump if needed
    jle MUL_BREAKLOOP3 ;if n<=j break

    ;to load the 4 element value of A[i] : A[i][k:k+3]
    mov   ecx, r14d   ; ecx = i
    sal ecx, 3    ; i << 3 : to find the value A[8*i + k]
    add ecx, r15d    ; ecx = i + j
    sal ecx, 2    ; ecx << 2: because our values are dword and 32 bit so we need to skip 4 byte of memory to store each value
    movups xmm1, A[ecx] ; load the value of A[i][k:k+3] into xmm1, because xmm is 128 bit and our value has 32 bit so xmm can store 4 v

    ;to load the 4 element value of B[][j] : A[k:k+3][j]
    mov   ecx, r13d    ; ecx = j
    sal ecx, 3    ; k << 3 , mul 2^3
    add ecx, r15d     ; ecx = i + j
    sal ecx, 2    ; ecx << 2: because our values are dword and 32 bit so we need to skip 4 byte of memory to store each value
    movups xmm2, B[ecx] ; load the value of A[k:k+3][j] into xmm1, because xmm is 128 bit and our value has 32 bit so xmm can store 4 v


    mulps xmm1, xmm2 ; multiply 4 element row of A and col of B
    addps xmm0, xmm1 ; sum them then store in xmm0

    add r15d, 4  ; k++
    jmp MUL_LOOP3 ; does loop again
```

همچنین خروجی ها را بصورت زیر پرینت میکنیم.

```
; to print the result of A @ B = C
xor r14d, r14d ; int i = 0
OUTPUT_LOOP1:
    mov eax,[n] ; copy value of n into eax to compare later
    cmp eax,r14d ; do eax - r14d to make a jump if needed
    jle OUTPUT_ENDLOOP1 ;if n<=i break
    xor r13d,r13d ; int j=0
OUTPUT_LOOP2:
    mov eax,[n] ; copy value of n into eax to compare later
    cmp eax,r13d ;  do eax - r13d to make a jump if needed
    jle OUTPUT_ENDLOOP2 ;if n<=j break

    mov ecx, r14d ;
    sal ecx, 3 ; i << 3 : to find the value of C[i*8 + j] to print it
    add ecx, r13d
    sal ecx, 2 ; ; ecx << 2: because our values are dword and 32 bit so we need to skip 4 byte of memory to store each value
    movss xmm0, C[ecx] ; load and get C[i][j] into xmm0 to print it
    call print_float ; print each element of matrix as float


    inc r13d ; j++
    jmp OUTPUT_LOOP2
OUTPUT_ENDLOOP2:
    mov edi, 10 ; move the ASCII code of \n into edi
    call putchar ; print \n
    inc r14d ; i++
    jmp OUTPUT_LOOP1
```

و در قسمت text. باید فانکشن های C را extern کنیم تا بتوانیم استفاده کنیم. و در قسمت data. متغیر های خود را تعریف میکنیم.

```
A resd 128      ; float A[8][8]
B resd 128      ; float B[8][8]
C resd 128      ; float c[8][8]
n dd 0          ; define variable n as int 32
max dd 60000000 ; its for computing the time of multiply in order 10e7
read_int_format:    db "%d"   , 0 ; format to scan
print_int_format:   db "%d"   , 0 ; format to print
read_float_format:  db "%f"   , 0 ; format to scan
print_float_format: db "%.2f ", 0 ; format to print

segment .text ; extern some C function like printf, scanf and ...
    global main
    extern printf
    extern putchar
    extern puts
    extern scanf
    extern getchar
```

و در نهایت برای حالت 3*3 ماتریس زیر را ورودی می دهیم :

```
3
23.24 9823.2 984.2
289.3 89.233 78.2
12.32 98.32 7898
12.111 98.32 6734.2
2398.23 923.2 7832.2
45.34 349.23 877.3
```

و نتیجه خروجی 3 فایل به صورت زیر میباشد که برای پایتون با اینکه دفعات کمتری محاسبه میشود، زمان بیشتری طول میکشد:

```
O ( n = 6 * 10^7 )
Normal
23603200.00 9414776.00 77957216.00
221050.58 138133.67 2715699.50
594038.50 2850199.00 7781943.00

t = 2.358606729s
==================================
Parallel
23603200.00 9414776.00 77957216.00
221050.56 138133.66 2715699.50
594038.50 2850199.00 7781943.00

t = 1.575404655s
==================================
High Level Python : O ( n = 10 ^ 6 )
23603198.02 9414775.36 77957208.51
221050.56 138133.67 2715699.62
594038.50 2850198.87 7781942.65
23603198.02 9414775.36 77957208.51
221050.56 138133.67 2715699.62
594038.50 2850198.87 7781942.65

t 6.625543594360352s
```

و در حالتی که ماتریس ورودی 5*5 است داریم :

```
5
2332.98 243.23 873.23 982.2 993.3
23.45 982 923.2 874 23.111
232.53 9834.2 93.202 982 23.5
798.4 56.4 346. 654. 24.6 767.3
23.54 57.5 233.32 784.3 34.2
23.54 575.5 23.2 78.4 45.3
46.2 895.5 32.4 67.4 245.4
34.33 46.22 546.43 34.2 324.3
982 24 234  235 93
23.43 545.3 647.3 24.8 24.2
```

که نتیجه خروجی آن بصورت زیر است:

```
Normal Mul : O ( n = 6 * 10 ^ 7 )
716001.19 1083927.00 2166022.50 1411962.75 484614.66
557427.38 936423.38 969125.25 756300.75 305561.88
796961.62 1427888.12 8981038.00 619950.69 915595.81
329148.59 676082.56 555087.94 378374.59 232529.20
190024.09 268620.16 898597.38 612258.06 137990.27
t = 9.063031418s
=====================================================
Parallel Mul
716001.19 1083927.00 2166022.50 1411962.75 484614.66
557427.38 936423.31 969125.19 756300.75 305561.88
796961.69 1427888.00 8981037.00 619950.75 915595.81
329148.59 676082.56 555087.94 378374.62 232529.20
190024.09 268620.16 898597.31 612258.06 137990.27
t = 5.340821332s
=====================================================
High level Python O ( n = 10 ^ 6 )
1083926.98 2166022.44 1411962.75 484614.64 563943.58
936423.36 969125.21 756300.77 305561.87 623280.13
1427888.03 8981037.46 619950.69 915595.74 2545966.40
676082.57 555087.90 378374.60 232529.20 223633.20
782204.41 113295.03 335566.04 198859.24 164610.08
t = 20.028716802597046s
```

نتیجه مهم : زبان اسمبلی بشدت سرعت بالاتری نسبت به زبان های سطح بالا دارد و میتوان حتی در مقادیر چند برابر بزرگتر و بیشتر، در اسمبلی سریعتر نتیجه را دریافت کنیم.

فاز 2

سپس برنامه خود را برای محاسبه یک تابع Convolution-2D دلخواه (مثلاً برای یک کار پردازش تصویر یا هوش مصنوعی)به کار ببرید و زمان اجرای برنامه کامل را با برنامه(های) مشابه موجود یا خودتان مقایسه کنید. توضیح اینکه عمل Convolution (به زبان ساده) یک تابع را بر روی محور افقی از روی یک تابع دیگر عبور می‌دهد و در هر نقطه برخورد (یا در نقاط گسسته) حاصل ضرب این دو تابع را محاسبه و ثبت می‌کند.(تعداد قابل توجهی از پردازشهای تصویر و اخیراً مدل‌های یادگیری ماشین، بر اساس تابع Convolution پیاده‌سازی شده است.) میتوانید یک تابع را با ماتریس 5 در 5 ودیگری را 3 در 3 در نظر بگیرید.

توجه کنید که هدف این است که عمل Convolution مبتنی بر ضرب ماتریس بخش قبل را پیاده کنید و میزان تسریع آن را در یک کاربرد پر استفاده یا جالب (که شما انتخاب کرده‌اید) نسبت به اجرای بدون اسمبلی نشان دهید. قسمت ورود و خروج اطلاعات را می‌توانید با یک زبان سطح بالاتر انجام دهید و از صحت نتایج نیز مطمئن شوید.

در این فاز یکبار به زبان اسمبلی و در حالت عادی و بدون موازی کانوولوشن را در فایل Convolution.asm انجام می دهیم، و بار دیگر در زبان سطح بالایی مانند پایتون این کار در فایل HLConvo.py انجام می دهیم.

برای کانوولوشن یک ماتریس ورودی n*n و یک کرنل که معمولا 3*3 میباشد، با روش های مختلف فرایند انجتم میشود و تغییراتی روی عکس صورت میگیرد.

در فایل toMatrix.py عکس را گرفته و به سیاه سفید تبدیل میکنیم چون عکس رنگی 3 ماتریسی 3 بعدی است، در نهایت آن را به مینیموم سایز ابعاد به صورت مربعی ریسایز میکنیم.

```
import numpy as np
from PIL import Image

image = Image.open('lion.jpg').convert('L')
n = 700
image = image.resize((n, n))
matrix = np.array(image)
n = matrix.shape[0]
print(n)
for row in matrix:
    for el in row:
        print(el, end=' ')
    print()
```

در فایل toImage.py ماتریس را در حالت سیاه سفید گرفته و به عکس تبدیل و در نهایت نتیجه را ذخیره میکنیم.

```python
import numpy as np
import cv2

n = int(input())
data = np.array([list(map(float, input().split())) for _ in range(n)])
cv2.imwrite('output1.jpg', data)
```

در فایل HLConvo.py ما کانوولوشن را در حالت زبان سطح بالا انجام می دهیم که یک سایز و یک ماتریس ورودی برای عکس و یک ورودی عددی سایز و ماتریس دیگر برای کرنل میگیریم و در نهایت فرایند کانوولوشن را انجام می دهیم :

```python
import numpy as np
import time

start = time.time()
def dot_product(vec1, vec2):
    return sum(x * y for x, y in zip(vec1, vec2))

n = int(input())
mat1 = []
mat2 = []

for _ in range(n):
    mat1.append(list(map(float, input().split())))

m = int(input())
for _ in range(m):
    mat2.append(list(map(float, input().split())))

mat1 = np.array(mat1)
mat2 = np.array(mat2)
mat3 = np.zeros((n - m + 1, n - m + 1))

for i in range(n - m + 1):
    for j in range(n - m + 1):
        for k in range(m):
            mat3[i][j] += dot_product(mat1[i + k][j:j+m], mat2[k])
# print(mat3.shape[0])
# for row in mat3:
#     print(' '.join(map(str, row)))
end = time.time()
print(end - start)
```

و در فایل Convolution.asm همین فرایند را در زبان اسمبلی انجام می دهیم و در نهایت خروجی این 2 فایل یک عدد ک نشان دهنده سایز ماتریس نهایی است که از طریق

n-m+1 بدست میاید که n سایز عکس ورودی و m سایز کرنل است، به همراه ماتریس عکس نهایی میباشد.

در نهایت ماتریس خروجی را به فایل toImage می دهیم و عکس تبدیل میشود.

مقایسه زمانی کانوولوشن بصورت زیر است :

```
Convo in Asm :
0.062335351
====================
Convo in Python :
4.151884078979492
```

نتیجه مهم : میتوان پردازش تصاویر ور دیگر پردازش ها را به کمک اسمبلی چندین برابر سریع تر انجام داد.

و در عکس ورودی ما بطور مثال :

که بعد از کانوولوشن با کرنل شارپ کردن :

```
3
0 -1 0
-1 5 -1
0 -1 0
```

به عکس زیر تبدیل میشود :