

به نام خدا



دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

پروژه درس ساختار و زبان کامپیوتر

استاد: دکتر امیرحسین جهانگیر

نام و نام خانوادگی: اشکان تارپوردی

شماره دانشجویی: ۴۰۱۱۰۵۷۵۳

**پروژه بخش اول:** برنامه اسمبلی ضرب دو ماتریس  $n \times n$  اعداد ممیز شناور ۳۲ 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 80x86 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.

**پاسخ بخش اول:** ضرب خارجی دو ماتریس  $a$  و  $b$  را بصورت زیر تعریف می‌کنیم:  
اگر  $a = (a_1, a_2, a_3) = a_1 i + a_2 j + a_3 k$  و  $b = (b_1, b_2, b_3) = b_1 i + b_2 j + b_3 k$  ضرب خارجی دو ماتریس  $a \times b$  بصورت زیر است:

$$a \times b = (a_1 i + a_2 j + a_3 k) \times (b_1 i + b_2 j + b_3 k) = a_1 b_1 (i \times i) + a_1 b_2 (i \times j) + a_1 b_3 (i \times k) + a_2 b_1 (j \times i) + a_2 b_2 (j \times j) + a_2 b_3 (j \times k) + a_3 b_1 (k \times i) + a_3 b_2 (k \times j) + a_3 b_3 (k \times k)$$

برای محاسبه ضرب خارجی بردارهای یک‌به‌روشن زیر عمل می‌کنیم:

$$i \times j = k, j \times i = -k$$

$$j \times k = i, k \times j = -i$$

$$k \times i = j, i \times k = -j$$

ابتدا به کمک زبان اسمبلی پردازنده 80x86، برنامه‌ای برای ضرب خارجی (Cross Product) دو ماتریس  $n \times n$  پیاده‌سازی کرده‌ایم. در تصویر ذیل، برای دو حالت  $n=3$  و  $n=5$ ، ورودی و خروجی برنامه را مشاهده می‌کنید.  
ابتدا ضرب خارجی دو ماتریس را به روش معمولی و با استفاده از رجیسترهای 80x86 پیاده‌سازی کرده‌ایم. برای نشان دادن تفاوت دو برنامه معمولی و موازی، چون مقیاس ما در حد نانو و میکرو ثانیه است، برنامه را  $10^5$  دفعه اجرا کرده و زمان اجرای کل تعداد دفعات را با یکدیگر جمع کرده و خروجی نهایی، زمان اجرای برنامه بر حسب میلی ثانیه است.

```
ubuntu@ubuntu:~/Desktop/ConvolutionProject/x86_templates$ ./normal.sh < testCross3.txt
/usr/bin/ld: warning: asm_io.o: missing .note.GNU-stack section implies executable stack
/usr/bin/ld: NOTE: This behaviour is deprecated and will be removed in a future version of the linker
4
```

تصویر ۱ زمان اجرای ضرب عادی به ازای  $n=3$

```
ashkan@LAPTOP-PRAUBDAM:/mnt/d/Programming/Assembly/ConvolutionProject/x86_template$ ./normal.sh < testCross5.txt
60
```

تصویر ۲ زمان اجرای ضرب عادی به ازای  $n=5$

اکنون ضرب خارجی دو ماتریس را با استفاده از دستورات برداری (موازی) پردازنده پیاده‌سازی کرده‌ایم. دو تصویر بعدی ضرب به روش موازی را برای دو حالت  $n=3$  و  $n=5$  نشان می‌دهد. مانند قسمت قبل، برنامه را  $10^5$  دفعه اجرا کرده و میانگین می‌گیریم.

```

ubuntu@ubuntu:~/Desktop/ConvolutionProject/x86_template$ ./parallel.sh < testCross3.txt
/usr/bin/ld: warning: asm_io.o: missing .note.GNU-stack section implies executable stack
/usr/bin/ld: NOTE: This behaviour is deprecated and will be removed in a future version of the linker
2

```

تصویر ۳ زمان اجرای ضرب موازی به ازای  $n = 3$

```

ashkan@LAPTOP-PRAUBDAM:/mnt/d/Programming/Assembly/ConvolutionProject/x86_template$ ./parallel.sh < testCross5.txt
10

```

تصویر ۴ زمان اجرای ضرب موازی به ازای  $n = 5$

به ازای  $n = 3$  زمان اجرای هر دو برنامه معمولی و موازی را بدست آورده‌ایم. همانطور که در تصاویر بالا مشهود است، زمان اجرای ضرب عادی تقریباً دو برابر زمان اجرای ضرب موازی در  $10^5$  دفعه تکرار است. اکنون ضرب خارجی دو ماتریس  $n \times n$  را با زبان سطح بالا (High Level Language) جاوا (Java) پیاده‌سازی کرده‌ایم. این برنامه را نیز در دو حالت  $n = 3$  و  $n = 5$  آزمایش کرده و نتیجه‌ی آن در دو تصویر پیش‌رو قابل مشاهده است. همانند دو قسمت قبل، این برنامه را نیز  $10^5$  دفعه اجرا کرده و میانگین زمان اجرا را محاسبه کرده‌ایم.

```

ashkan@LAPTOP-PRAUBDAM:/mnt/d/Programming/Assembly/ConvolutionProject/x86_template$ java crossProductRandom
Enter the size of the matrices (n): 3
Result matrix saved to JavaMatrixProductOutput.txt
Total execution time: 536 milliseconds

```

تصویر ۵ زمان اجرای ضرب به ازای  $n = 3$  در جاوا

```

ashkan@LAPTOP-PRAUBDAM:/mnt/d/Programming/Assembly/ConvolutionProject/x86_template$ java crossProductRandom
Enter the size of the matrices (n): 5
Result matrix saved to JavaMatrixProductOutput.txt
Total execution time: 1010 milliseconds

```

تصویر ۶ زمان اجرای ضرب به ازای  $n = 5$  در جاوا

همانطور که مشاهده می‌کنید، به ازای مقادیر کوچک  $n$ ، برنامه به زبان اسمبلی و به زبان سطح بالا تفاوت چندانی در زمان اجرا ندارند. اما اگر مقدار  $n$  را بطور قابل توجهی زیاد کنیم، تفاوت زمان اجرای این دو برنامه مشهود خواهد شد. تصاویر بعدی مقایسه زمان اجرای سه برنامه ضرب خارجی دو ماتریس به روش زبان اسمبلی با محاسبه معمولی، روش زبان اسمبلی با کمک دستورات پردازش موازی و به روش پیاده‌سازی با زبان سطح بالا را در مقادیر بزرگ از  $n$  نشان می‌دهد.

```

ashkan@LAPTOP-PRAUBDAM:/mnt/d/Programming/Assembly/ConvolutionProject/x86_template$ java crossProductRandom
n: 1000
Result matrix saved to JavaMatrixProductOutput.txt
Total execution time: 5163 milliseconds

```

تصویر ۷ زمان اجرای ضرب توسط زبان سطح بالا به ازای  $n = 1000$

```

ashkan@LAPTOP-PRAUBDAM:/mnt/d/Programming/Assembly/ConvolutionProject/x86_template$ ./normal.sh < testMatrix.txt
3361

```

تصویر ۸ زمان اجرای ضرب معمولی به ازای  $n = 1000$

```
ashkan@LAPTOP-PRAUBDAM:/mnt/d/Programming/Assembly/ConvolutionProject/x86_template$ ./parallel.sh < testMatrix.txt
```

635

تصویر ۹ زمان اجرای ضرب موازی به ازای  $n=1000$

**پروژه بخش دوم:** 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 برنامه‌های 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  $5 \times 5$  و دیگری را  $3 \times 3$  در نظر بگیرید.

**پاسخ بخش دوم:** فرمول صریح محاسبه Convolution-2D بصورت زیر است:

$$g(x, y) = w * f(x, y) = \sum_{j=-a}^a \sum_{i=-b}^b w(i, j) \cdot f(x-i, y-j)$$

که  $g(x, y)$  تابع تغییر یافته و  $f(x, y)$  تابع اصلی است. به  $w$  kernel گفته می‌شود و همچنین هر عنصر از kernel بصورت  $-a \leq i \leq a$  و  $-b \leq j \leq b$  است.

ابتدا به کمک زبان اسمبلی پردازنده  $80x86$ ، یک ماتریس  $n \times n$  را در ورودی گرفته و سپس در دو برنامه جدا، که یکی دارای  $3 \times 3$  kernel و دیگری دارای  $5 \times 5$  kernel است، ماتریس kernel را ورودی می‌گیریم. در برنامه اول عمل Convolution-2D را با  $3 \times 3$  kernel انجام داده و در برنامه دوم همان عمل را با  $5 \times 5$  kernel انجام می‌دهیم.

```
ashkan@LAPTOP-PRAUBDAM:/mnt/d/Programming/Assembly/ConvolutionProject/x86_template$ time ./convolution3x3.sh < test
Conv.txt > o

real    0m3.753s
user    0m0.530s
sys     0m0.675s
```

تصویر ۱۰ زمان اجرای Convolution  $3 \times 3$  با زبان اسمبلی

```
ashkan@LAPTOP-PRAUBDAM:/mnt/d/Programming/Assembly/ConvolutionProject/x86_template$ time ./convolution5x5.sh < test
Conv5.txt > o

real    0m4.994s
user    0m0.819s
sys     0m0.702s
```


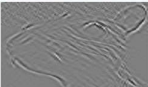
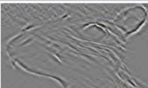


تصویر ۱۱ زمان اجرای Convolution  $5 \times 5$  با زبان اسمبلی

اکنون برای محاسبه عمل Convolution-2D، از زبان سطح بالای جاوا استفاده می‌کنیم. مانند قسمت قبل، دو برنامه با  $3 \times 3$  kernel و دیگری با  $5 \times 5$  kernel پیاده‌سازی می‌کنیم.

تصویر ۱۲ زمان اجرای  $3 \times 3$  Convolution با زبان جاوا

تصویر ۱۳ زمان اجرای 5 x 5 Convolution با زبان جاوا

**پاسخ بخش سوم:** در این بخش به کمک برنامه اسمبلی محاسبه Convolution و استفاده از زبان سطح بالای جاوا، کاربرد عمل Convolution را در پردازش تصویر بررسی می‌کنیم.

Operation	Kernel $\omega$	Image result $g(x,y)$
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Ridge or edge detection	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

تصویر ۱۴ مثالی از پردازش تصویر به کمک Convolution

4

اکنون می‌خواهیم به کمک یک زبان سطح بالا، ابتدا تصویر ورودی را به یک ماتریس تبدیل کرده و سپس این ماتریس را به عنوان ورودی به تابع محاسبه Convolution که با زبان اسمبلی طراحی کرده‌ایم ورودی دهیم، و در نهایت ماتریس خروجی برنامه زبان اسمبلی را به یک برنامه دیگر که با زبان سطح بالای جاوا پیاده‌سازی شده است، ورودی دهیم و تصویر نهایی را با تصویر اولیه مقایسه کنیم.

تصویر اولیه مانند تصویر زیر است:



تصویر ۱۵ تصویر اصلی

پس از تبدیل این تصویر به ماتریس متناظر آن و سپس ورودی دادن این ماتریس به تابع محاسبه Convolution در زبان اسمبلی، ماتریس خروجی برنامه به زبان اسمبلی را به برنامه تبدیل ماتریس به تصویر می‌دهیم و نتیجه آن بصورت زیر است:



تصویر ۱۶ تصویر پس از اعمال Convolution با یک kernel خاص



اگر روی همین تصویر با یک kernel دیگر عمل Convolution را انجام دهیم، خروجی به صورت زیر می‌شود:



تصویر ۱۷ تصویر پس از اعمال Convolution با یک kernel دیگر

Kernel فوق باعث می‌شود که تنها خطوط حاشیه‌ای تصویر باقی بماند.