به نام خدا

پروژه نهایی درس ساختار و زبان کامپیوتر

استاد: دکتر امیرحسین جهانگیر

گردآوری: سید علیرضا میررکنی

شماره دانشجویی: ۴۰۱۱۰۶۶۱۷

دانشکده مهندسی کامپیوتر دانشگاه صنعتی شریف – زمستان ۱۴۰۲

فهرست مطالب

شرح کلی پروژه

پروژه ای که پیش روی شما قرار گرفته، شامل دو بخش می باشد:

- در بخش اول به پیاده سازی ضرب ماتریس ها به دو روش (یکی بدون استفاده از پردازش موازی و به شکل عادی و دیگری با استفاده از موازی سازی و simd) با استفاده از زبان اسمبلی x86 nasm پرداختیم. همچنین برنامه ای با استفاده از زبان سطح بالای java نیز برای ضرب ماتریس ها نوشتیم و سرعت انجام محاسبات و اجرای دستورات را در این سه حالت مقایسه کردیم.
- در بخش دوم به پیاده سازی 2D-convolution برای ماتریس ها به دو روش (یکی با استفاده از dot-product و دیگری با استفاده از matrix multiplication که در بخش قبل نوشته بودیم) با استفاده از زبان اسمبلی x86 nasm پرداختیم. در این بخش نیز برنامه ای با استفاده از زبان سطح بالای java نوشتیم که عملکردی مشابه برنامه نوشته شده با زبان اسمبلی دارد و سرعت آنها را با هم مقایسه کردیم. در نهایت از برنامه اسمبلی برای پردازش تصاویر استفاده کردیم. (یعنی 2D-convolution را با استفاده از برنامه اسمبلی انجام دادیم)

در ادامه پس از توضیح نحوه گرفتن ورودی و خروجی دادن به کاربر و اجرای برنامه، به تفصیل به شرح هر یک از این دو بخش می پردازیم.

نحوه گرفتن ورودی و خروجی دادن

برای گرفتن ورودی از کاربر و نمایش خروجی های مطلوب در ترمینال، از فایل asm.io استفاده شده. در این فایل توابع ورودی گرفتن و خروجی دادن در ترمینال با استفاده از توابع آماده زبان C (printf, scanf, puts, gets, putc, getc) پیاده سازی شده اند، توابعی نظیر read_int, print_int, read_float, print_float و غیره. این فایل در پوشه های main_project و CV قرار گرفته و از آن به عنوان کتابخانه در فایل های template.asm استفاده کرده ایم.

نحوه اجرا کردن برنامه های اسمبلی

در هر کدام از پوشه های main_project و CV یک فایل بش به نام run.sh قرار گرفته که شامل دستورات مورد نیاز برای اجرای فایل template.asm می باشد. برای اجرای فایل template.asm در هر کدام از این پوشه ها، کافی است در ترمینال وارد پوشه مدنظر شوید و کامند زیر را اجرا کنید:

```
./run.sh template
```

برنامه اسمبلی مدنظر اجرا خواهد شد.

دقت کنید که به طور کلی تمامی فایل های مربوط به آن بخشی از پروژه که مربوط به پردازش تصویر می باشند، در پوشه CV قرار گرفته اند و سایر فایل ها در پوشه main_project قرار دارند.

نحوه محاسبه زمان اجرای برنامه های اسمبلی

برای محاسبه زمان اجرای یک زیربرنامه در زبان اسمبلی به این صورت عمل می کنیم:

- در ابتدا کلاک فعلی کامپیوتر را با فراخوانی زیربرنامه save_start_time در دو متغیر start_time_l و start_time_h ذخیره می کنیم. در این زیربرنامه، از دستور rdtsc استفاده می کنیم که کلاک فعلی را در دو رجیستر rdx و rax ذخیره می کند.
- سپس زیربرنامه مدنظر -که می خواهیم مدت زمان اجرای آن را اندازه بگیریم- را فراخوانی می کنیم. حال با فراخوانی زیربرنامه calculate_execution_time ، مدت زمان اجرای آن زیربرنامه را محاسبه می کنیم. در این زیربرنامه، در ابتدا با استفاده از دستور rdtsc کلاک نهایی کامپیوتر را در متغیر های end_time_l و end_time_h ذخیره می کنیم. حال با تفاضل دو کلاک می توانیم تعداد clock cycle هایی که برنامه برای اجرا شدن نیاز داشت را به دست آوریم. همچنین دقت کنید که در سیستم اجرایی سرعت CPU به طور متوسط برابر 3GHz می باشد (که البته این مقدار در کامپیوتر های مختلف متفاوت است). پس می توان طبق رابطه زیر، زمان اجرای زیر برنامه را محاسبه کرد:

$$t = \frac{total\ clock\ cycles}{cpu\ clock\ per\ second\ (cpu\ speed)}$$

کد توابع مورد استفاده در زیر آمده است: (کامنت گذاری و توضیحات مفصل در فایل template.asm)

```
save_start_time:
    rdtsc
    mov [start_time_h],rdx
    mov [start_time_l],rax
calculate_executin_time:

    mov [end_time_h], rdx
    mov [end_time_l],
    mov rbx, [end_time_h]
    sub rbx, [start_time_h]
    shl rbx, 32
    add rbx, [end_time_l]
    sub rbx, [start_time_l]
    mov rax, rbx
    imul rax, 10
    mov rbx, 30
    xor rdx, rdx
    idiv rbx
    mov [execution_time],rax
```

بخش اول پروژه

همانطور که در قسمت شرح پروژه اشاره گردید، در بخش اول برنامه ضرب دو ماتریس را با استفاده از زبان اسمبلی x86 nasm و به دو روش استفاده از پردازش موازی (simd) و بدون استفاده از پردازش موازی (روش معمولی) پیاده سازی کرده ایم. همچنین برنامه ای با استفاده از زبان java برای ضرب دو ماتریس نوشته ایم که در ادامه به مقایسه سرعت این سه روش میپردازیم.

با استفاده از این برنامه ها، شما قادر خواهید بود حاصل ضرب دو ماتریس شامل درایه های حقیقی با ابعاد حداکثر ۱۰۰ (نه لزوما مربعی) را محاسبه نمایید.

با اجرای فایل template.asm در پوشه main_project به روش گفته شده در بخش "نحوه اجرا کردن برنامه های اسمبلی" به پروژه اصلی دسترسی خواهید داشت. با انتخاب گزینه matrix multiplication و در ادامه وارد کردن ابعاد و درایه های ماتریس ها، می توانید ضرب دو ماتریس را محاسبه کنید.

همچنین با انتخاب گزینه set multiplication method می توانید روش انجام ضرب دو ماتریس (با استفاده از پردازش موازی یا به روش عادی) را انتخاب نمایید.

توضیحاتی مختصر درباره کد برنامه

در خود فایل template.asm در پوشه main_project با کامنت گذاری های مفصل هر خط از برنامه به طور کامل توضیح و شرح داده شده است. در اینجا به صورت مختصر به توضیح کد های نوشته شده می پردازیم.

در تابع multiply_matrices ضرب دو ماتریس به روش معمولی و بدون استفاده از پردازش موازی (simd) پیاده سازی شده، به این صورت که از همان روش سنتی سه حلقه تو در تو برای محاسبه درایه های ماتریس حاصل ضرب استفاده کرده ایم و در هر operation یک درایه از matrix1 را در یک درایه از matrix2 ضرب می کنیم و آن را به یک متغیر کمکی (که در اینجا sum نام دارد) اضافه می کنیم و به این شکل هر درایه از ماتریس حاصل ضرب محاسبه می شود.

در تابع multiply_matrices_packed نیز ضرب دو ماتریس را با استفاده از simd پیاده سازی کرده ایم، به این شکل که در ابتدا ترانهاده matrix2 را با صدا زدن زیربرنامه calculate_matrix2_transpose محاسبه کرده ایم و آن را در matrix2_transpose ذخیره کرده ایم. سپس با صدا زدن زیربرنامه append_zeros تعداد ستوان های matrix1 و matrix2_transpose را با اضافه کردن تعدادی صفر به هر سطر، مضرب ۴ کردیم (تا در ادامه بتوانیم درایه های هر سطر را در قالب تعدادی بردار ۴ مولفه ای در هم ضرب داخلی کنیم) و ماتریس های حاصل را به ترتیب در temp_matrix1 و temp_matrix2_transpose ذخیره کردیم. سپس بجای اینکه در هر operation یک درایه از matrix1 را در یک درایه از matrix2_transpose ضرب کنیم، ۴ درایه از temp_matrix1 را در ۴ درایه از temp_matrix2_transpose در قالب دو بردار ۴ تایی ضرب داخلی می کنیم و آن را به یک متغیر کمکی (که در اینجا sum_packed نام دارد) اضافه می کنیم و به این شکل هر درایه از ماتریس حاصل ضرب محاسبه می شود.

دقت کنید که در این دو تابع از رجیستر های xmm1 و xmm2 برای انجام عملیات های جبری روی اعداد اعشاری استفاده می کنیم. این دو رجیستر برای کار با اعداد اعشاری سازگار شده اند و رجیستر هایی ۱۲۸ بیتی هستند، یعنی می توانند ۴ عدد اعشاری از جنس float که ۳۲ بیتی هستند را به صورت همزمان در خود نگهداری کنند.

دستورات کار با این رجیستر ها کمی با دستورات کار با رجیستر های معمولی متفاوند. هر کدام از این دستورات به همراه عملکردشان در جدول زیر آورده شده اند:

عملکرد	دستور
محتویات موجود در خانه address حافظه را به رجیستر xmm1 منتقل می کند.	movss xmm1, address
محتویات موجود در رجیستر xmm1 را به خانه address حافظه منتقل می کند.	movss address, xmm1
محتویات موجود در خانه address حافظه را به رجیستر xmm1 اضافه می کند.	addss xmm1, address
رجیستر xmm1 را با خودش xor می کند (برای صفر کردن xmm1)	pxor xmm1, xmm1
با شروع از خانه address حافظه، ۴ عدد اعشاری (معادل ۱۲۸ بیت) می خواند و آنها را به صورت یک بردار با ۴ مولفه در رجیستر xmm0 ذخیره می کند.	movups xmm0, address
بردار ۴ مولفه ای ذخیره شده در رجیستر xmm1 را در بردار ۴ مولفه ای ذخیره شده در رجیستر xmm0 ضرب داخلی می کند و حاصل را در xmm1 ذخیره می کند.	dpps xmm1, xmm0, 0xF1

در ادامه تصاویری از نحوه عملکرد برنامه که روی دو ماتریس تصادفی اجرا شده آمده است.

```
1- matrix multiplication
2- calculate convolution using dot product
3- calculate convolution using matrix multiplication
4- set multiplication method
5- exit
1
```

انتخاب matrix multiplication در منوی برنامه

```
enter the number of rows for the first matrix (a number between 1 and 100):
3

enter the number of columns for the first matrix (a number between 1 and 100):
4

please enter the entries of the first matrix:
1 2 3 4
5 6 7 8
9 10 11 12

enter the number of columns for the second matrix (a number between 1 and 100):
4

please enter the entries of the second matrix:
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
```

وارد کردن ابعاد و درای های ماتریس های ورودی

```
result:
90.000000 100.000000 110.000000 120.000000
202.000000 228.000000 254.000000 280.000000
314.000000 356.000000 398.000000 440.000000

execution time in nano seconds:
906
```

خروجی برنامه (ماتریس حاصلضرب)

```
please enter the number that represents your next request (between 1 and 4):
1- matrix multiplication
2- calculate convolution using dot product
3- calculate convolution using matrix multiplication
4- set multiplication method
5- exit
4

please enter the number of your desired multiplication method (1 or 2):
1. regular method (not using simd and parallel proccessing)
2. using simd (parallel proccessing)
2

multiplication method has been set successfully!
```

انتخاب set multiplication method در منوی برنامه و انتخاب روش ضرب ماتریس با استفاده از پردازش موازی

مقایسه سرعت روش های مختلف

برای مقایسه سرعت عملکرد سه روش ضرب دو ماتریس (روش عادی در اسمبلی، استفاده از simd در اسمبلی و ضرب ماتریس با زبان سطح بالای java) از دو ماتریس مربعی با ابعاد ۳۰ با درایه های تصادفی استفاده می کنیم. این ماتریس ها در فایل multiplication_input.txt در پوشه main_project قرار گرفته اند. این دو ماتریس را با استفاده از برنامه multiplication_input_generator.py (نوشته شده به زبان پایتون) تولید کرده ایم که می توان با اجرای مجدد برنامه و تغییر پارامتر های موجود در آن، ماتریس های دیگری نیز تولید کرد و از آن ها برای مقایسه سرعت استفاده نمود.پس از دادن این ورودی به برنامه ها، نتایج زیر را مشاهده می کنیم:

```
execution time in nano seconds:
33764
```

برنامه اسمبلی (روش عادی بدون simd)

```
execution time in nano seconds:
13046
```

برنامه اسمبلی (با استفاده از پردازش موازی و simd)

```
execution time in nanoseconds:
 6598200
```

برنامه java

همانطور که مشاهده می شود، مدت زمان اجرای برنامه های اسمبلی بسیار بسیار کمتر از برنامه جاوا می باشد (سرعت برنامه اسمبلی به روش عادی حدود ۱۹۵ برابر سرعت برنامه جاوا است). همچنین برنامه اسمبلی با استفاده از پردازش موازی حدود ۲.۵ برابر سریع تر از برنامه اسمبلی به روش عادی می باشد. علت اینکه سرعت برنامه simd ۴ برابر سرعت برنامه به روش معمولی نمی باشد، این است که ابعاد ماتریس هایی که در برنامه simd ضرب می شوند بزرگ تر از ابعاد ماتریس هایی می باشد که در برنامه به روش عادی در هم ضرب می شوند (در برنامه simd ماتریس های مربعی به ابعاد ۳۰ را به ماتریس هایی با ۳۰ سطر و ۳۲ ستون تبدیل می کنیم)

همچنین زمان اجرای این برنامه ها برای ماتریس های دیگر با سایز های مختلف در جدول زیر آورده شده است:

زمان اجرای برنامه جاوا (نانوثانیه)	زمان اجرای برنامه اسمبلی با پردازش موازی (نانوثانیه)	زمان اجرای برنامه اسمبلی به روش عادی (نانوثانیه)	ابعاد ماتریس های ورودی
۶۸۴۰۰	۲۸۸	۳۳۸	۵
۳۷۶۵۰۱	۱۰۵۲	۲۰۳۰	۱۰
۲۷۴۳۴۰۰	۷۹۱۲	۱۳۶۰۲	۲۰

لازم به ذکر است برنامه جاوا در کلاس MultiplyMatrices در پوشه high_level_programming_language قرار گرفته است.

بخش دوم پروژه

در بخش دوم پروژه به پیاده سازی 2D_convolution برای دو ماتریس (کرنل و ماتریس زمینه) با استفاده از زبان اسمبلی x86 nasm پرداخته ایم. با استفاده از دو روش (یکی استفاده از ضرب نقطه ای ماتریس ها و دیگری استفاده از ضرب معمولی ماتریس ها که در بخش اول آن را پیاده سازی کردیم)، 2D_convolution را پیاده سازی کرده ایم. در این بخش نیز کدی به زبان سطح بالای java با عملکردی مشابه نوشته ایم که در ادامه سرعت آن را با برنامه اسمبلی مقایسه خواهیم نمود.

با اجرای فایل template.asm در پوشه main_project به روش گفته شده در بخش "نحوه اجرا کردن برنامه های اسمبلی"، سپس انتخاب گزینه calculate convolution using dot product، در ادامه وارد کردن ابعاد و درایه های کرنل و همچنین ماتریس زمینه، و در نهایت انتخاب نوع کانولوشن، می توانید حاصل 2D_convolution این دو ماتریس را با استفاده از ضرب نقطه ای مشاهده کنید. همچنین با انتخاب گزینه calculate convolution using matrix multiplication و سپس وارد کردن ابعاد و درایه های کرنل و ماتریس زمینه می توانید حاصل کانولوشن با استفاده از ضرب عادی ماتریس ها را مشاهده نمایید.

توضیحاتی مختصر درباره کد برنامه

در این قسمت نیز به علت وجود کامنت گذاری و توضیحات مفصل در کد برنامه، به توضیحی مختصر درباره کد نوشته شده به زبان اسمبلی بسنده می کنیم.

در تابع regular_convolution، حاصل 2D_convolution کرنل و ماتریس زمینه را به صورت بدون حاشیه و با استفاده از ضرب نقطه ای ماتریس ها محاسبه کرده ایم. در این تابع، در ابتدا درایه های کرنل را به matrix1 منتقل و آن درایه هایی از ماتریس زمینه را که با کرنل همپوشانی دارند در matrix2 ذخیره می کنیم. سپس زیربرنامه های multiply_entries و multiply_entries_packed (با توجه به روش ضرب انتخابی توسط کاربر در قسمت set multiplication method) را فراخوانی می کنیم که اولی به روش عادی و بدون موازی سازی دو ماتریس را ضرب نقطه ای می کند و دومی از پردازش موازی برای ضرب نقطه ای دو ماتریس استفاده می کند. هر دوی این زیربرنامه ها، دو ماتریس matrix1 و matrix2 را ضرب نقطه ای می کنند و حاصل را به ترتیب در متغیر های sum و sum_packed ذخیره می کنند. دقت کنید که در regular_convolution، همواره کرنل را به گونه ای روی ماتریس زمینه قرار می دهیم که کامل در داخل آن قرار بگیرد و اصطلاحا بیرون زدگی اتفاق نیفتد. بنابر این در این روش، اگر سایز کرنل k و سایز ماتریس زمینه n باشد، سایز ماتریس نهایی $n - k + 1$ خواهد بود.

در تابع edge_handling_convolution نیز حاصل 2D_convolution کرنل و ماتریس زمینه را به صورت حاشیه دار محاسبه می کنیم. ۳ حالت مختلف برای حاشیه گذاری ماتریس زمینه پیاده سازی کرده ایم که عبارتند از:

۱) Edge extended
۲) Edge mirrored
۳) Zero edge

در تمامی این روش ها، با اضافه کردن حاشیه به ماتریس زمینه ابعاد آن را افزایش می دهیم و امکان بیرون زدگی کرنل از ماتریس زمینه را فراهم می کنیم. به عبارت دیگر، با افزودن تعدادی سطر و ستون به ماتریس زمینه به گونه ای ابعاد آن را افزایش می دهیم تا ابعاد ماتریس نهایی با ابعاد ماتریس اولیه برابر باشد، یعنی در این روش ها اگر سایز ماتریس زمینه n باشد، سایز ماتریس نهایی نیز n خواهد بود. توضیحات بیشتر درباره نحوه عملکرد این روش های حاشیه گذاری و تفاوت آنها با یکدیگر را می توانید در این لینک مطالعه بفرمایید.

در تابع edge_handling_convolution، تعدادی زیربرنامه فراخوانی می کنیم که عملکرد هر کدام از این زیربرنامه ها را به اختصار توضیح می دهیم:

- build_kernel_for_convolution : درایه های ماتریس kernel را برای فراخوانی توابع ضرب نقطه ای در matrix1 ذخیره می کند. (مشترک با زیربرنامه regular_convolution)
- calculate_result_matrix_size : ابعاد ماتریس نهایی را با توجه به convolution_type طبق آنچه که بالاتر بیان کردیم محاسبه می کند. (مشترک با زیربرنامه regular_convolution)
- build_matrix2_for_convolution : درایه هایی از ماتریس زمینه را که با کرنل هم پوشانی دارند در هر مرحله به matrix2 منتقل می کند تا در ادامه با فراخوانی یکی از زیربرنامه های multiply_entries یا multiply_entries_packed، matrix1 و matrix2 را در هم ضرب نقطه ای کنیم. (مشترک با زیربرنامه regular_convolution)
- adjust_coordinates : برای پیدا کردن درایه ای از ماتریس زمینه که باید در هر مرحله خوانده شود (وقتی که مختصات فعلی که در داخل آن هستیم، خارج از ماتریس زمینه باشد) این زیربرنامه را فراخوانی می کنیم که با توجه به convolution_type انتخابی توسط کاربر، مختصات فعلی را به یک مختصات معتبر تغییر می دهد.

در تابع convolution_using_matrix_multiplication حاصل 2D_convolution کرنل و ماتریس زمینه را با استفاده از ضرب عادی ماتریس ها محاسبه می کنیم. نحوه انجام این کار را می توانید در این لینک مطالعه بفرمایید. با استفاده از زیربرنامه های construct_M و construct_v به ترتیب ماتریس M و بردار v توضیح داده شده در لینک بالا را می سازیم و با فراخوانی زیربرنامه multiply_matrices آن ها را در هم ضرب می کنیم و به این شکل حاصل به دست می آید. دقت کنید که در اینجا نیز حاشیه گذاری نداریم و ماتریس نهایی از ماتریس زمینه کوچک تر است.

در ادامه تصاویری از عملکرد برنامه را روی یک کرنل و ماتریس زمینه تصادفی ضمیمه می کنیم.

```
please enter the number that represents your next request (between 1 and 4):
1- matrix multiplication
2- calculate convolution using dot product
3- calculate convolution using matrix multiplication
4- set multiplication method
5- exit
2
```

انتخاب calculate convolution using dot product در منوی برنامه

```
please enter the size of the kernel (a number between 1 and 100):
4

please enter the entries of the kernel:
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

please enter the size of the base matrix (a number between 1 and 100):
5

please enter the entries of the base matrix:
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25

please enter the number of your desired convolution type (between 1 and 4):
1. regular
2. edge extended
3. edge mirrored
4. zero edge
1
```

وارد کردن ابعاد و درایه های کرنل و ماتریس زمینه و انتخاب نوع کانولوشن

```
the result of 2-D convolution of the given base matrix and kernel:
40.000000 44.000000
60.000000 64.000000

execution time in nano seconds:
1125
```

خروجی برنامه (regular)

```
the result of 2-D convolution of the given base matrix and kernel:
10.000000 12.000000 15.000000 19.000000 22.000000
20.000000 22.000000 25.000000 29.000000 32.000000
35.000000 37.000000 40.000000 44.000000 47.000000
55.000000 57.000000 60.000000 64.000000 67.000000
70.000000 72.000000 75.000000 79.000000 82.000000

execution time in nano seconds:
6169
```

خروجی برنامه (edge extended)

```
the result of 2-D convolution of the given base matrix and kernel:
16.000000 17.000000 20.000000 24.000000 27.000000
21.000000 22.000000 25.000000 29.000000 32.000000
36.000000 37.000000 40.000000 44.000000 47.000000
56.000000 57.000000 60.000000 64.000000 67.000000
71.000000 72.000000 75.000000 79.000000 82.000000

execution time in nano seconds:
3714
```

خروجی برنامه (edge mirrored)

```
the result of 2-D convolution of the given base matrix and kernel:
8.000000 10.000000 12.000000 14.000000 5.000000
18.000000 21.000000 24.000000 27.000000 14.000000
28.000000 36.000000 40.000000 44.000000 27.000000
38.000000 51.000000 60.000000 64.000000 42.000000
21.000000 38.000000 51.000000 54.000000 57.000000

execution time in nano seconds:
6234
```

خروجی برنامه (zero edge)

در ادامه تصاویری از کار با کانولوشن با استفاده از ضرب عادی ماتریس ها را مشاهده خواهید نمود.

```
please enter the number that represents your next request (between 1 and 4):
1- matrix multiplication
2- calculate convolution using dot product
3- calculate convolution using matrix multiplication
4- set multiplication method
5- exit
3
```

انتخاب calculate convolution using matrix multiplication در منوی برنامه

```
please enter the size of the kernel (a number between 1 and 100):
4

please enter the entries of the kernel:
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

please enter the size of the base matrix (a number between 1 and 100):
5

please enter the entries of the base matrix:
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

وارد کردن ابعاد و درایه های کرنل و ماتریس زمینه

```
the result of 2-D convolution of the given base matrix and kernel:
40.000000 44.000000
60.000000 64.000000

execution time in nano seconds:
1021
```

خروجی برنامه

مقایسه سرعت برنامه ها

برای مقایسه سرعت عملکرد برنامه های 2D_convolution با استفاده از ضرب داخلی ماتریس ها در زبان اسمبلی و زبان سطح بالای جاوا، از یک کرنل تصادفی با ابعاد ۵ و یک ماتریس زمینه مربعی با ابعاد ۳۰ با درایه های تصادفی استفاده می کنیم. این ماتریس ها در فایل convolution_using_dot_product_input.txt در پوشه main_project قرار گرفته اند.

این دو ماتریس را با استفاده از برنامه convolution_using_dot_product_input_generator.py (نوشته شده به زبان پایتون) تولید کرده ایم که می توان با اجرای مجدد برنامه و تغییر پارامتر های موجود در آن، ماتریس های دیگری نیز تولید کرد و از آن ها برای مقایسه سرعت این برنامه ها استفاده نمود. در صفحه بعد، نتایج حاصل از دادن این ورودی ها به برنامه های اسمبلی و جاوا را مشاهده می کنید.

	اسمبلی	جاوا
regular	execution time in nano seconds: 85694	execution time in nanoseconds: 2765200
edge extended	execution time in nano seconds: 117683	execution time in nanoseconds: 6496500
edge mirrored	execution time in nano seconds: 114298	execution time in nanoseconds: 7611400
zero edge	execution time in nano seconds: 147044	execution time in nanoseconds: 7743500

همانطور که مشاهده می شود، برنامه به زبان اسمبلی بسیار سریع تر از برنامه به زبان جاوا می باشد (حتی در برخی موارد سرعت کد اسمبلی حدود ۵۰ برابر کد جاوا می باشد). در جدول زیر، زمان اجرای برنامه 2D_convolution با استفاده از ضرب معمولی ماتریس ها را در زبان های اسمبلی و جاوا (با استفاده از کرنل و ماتریس زمینه موجود در فایل convolution_using_matrix_multiplication_input.txt که توسط برنامه نوشته شده به زبان پایتون convolution_using_matrix_multiplication_input_generator.py تولید شده است) مقایسه کرده ایم:

	اسمبلی	جاوا
Convolution using matrix multiplication (regular)	execution time in nano seconds: 33724	execution time in nanoseconds: 786600

در اینجا نیز می توان مشاهده کرد که سرعت برنامه اسمبلی به مراتب بیشتر از برنامه جاوا می باشد، به طوری که حدود ۲۳ برابر سریع تر حاصل کانولوشن کرنل و ماتریس زمینه را محاسبه کرده است.

لازم به ذکر است که برنامه های جاوای مربوطه، به ترتیب در کلاس های ConvolutionUsingDotProduct و ConvolutionUsingMatrixMultiplication در پوشه high_level_programming_language قرار گرفته اند.

پردازش تصویر

در این بخش می خواهیم با استفاده از برنامه 2D_convolution که در بخش دوم پیاده سازی کردیم، تعدادی تصویر را با استفاده از کرنل های مختلف پردازش کنیم. تمامی فایل ها و مستندات مربوط به این بخش در پوشه CV قرار گرفته اند.

برای تبدیل فایل های تصویری (jpg.) به یک تنسور شامل اعداد حقیقی و بالعکس، از دو برنامه in.py و out.py استفاده می کنیم. این دو برنامه به زبان سطح بالای پایتون نوشته شده اند و به ترتیب عملیات های تبدیل فایل تصویری به تنسور و تبدیل یک تنسور از اعداد به یک فایل تصویری را انجام می دهند. برای پردازش یک تصویر دلخواه، در ابتدا فایل آن تصویر را در پوشه input_images قرار دهید، سپس در فایل های in.py و out.py مقدار متغیر image_name را برابر نام فایل تصویری قرار دهید. سپس از بین کرنل های موجود در فایل in.py کرنل مدنظر خود را انتخاب کنید و متغیر kernel را برابر آن قرار دهید. حال کافی است وارد پوشه CV شوید و فایل run.sh را اجرا نمایید.

فایل run.sh در پوشه CV به صورت زیر تغییر یافته است:

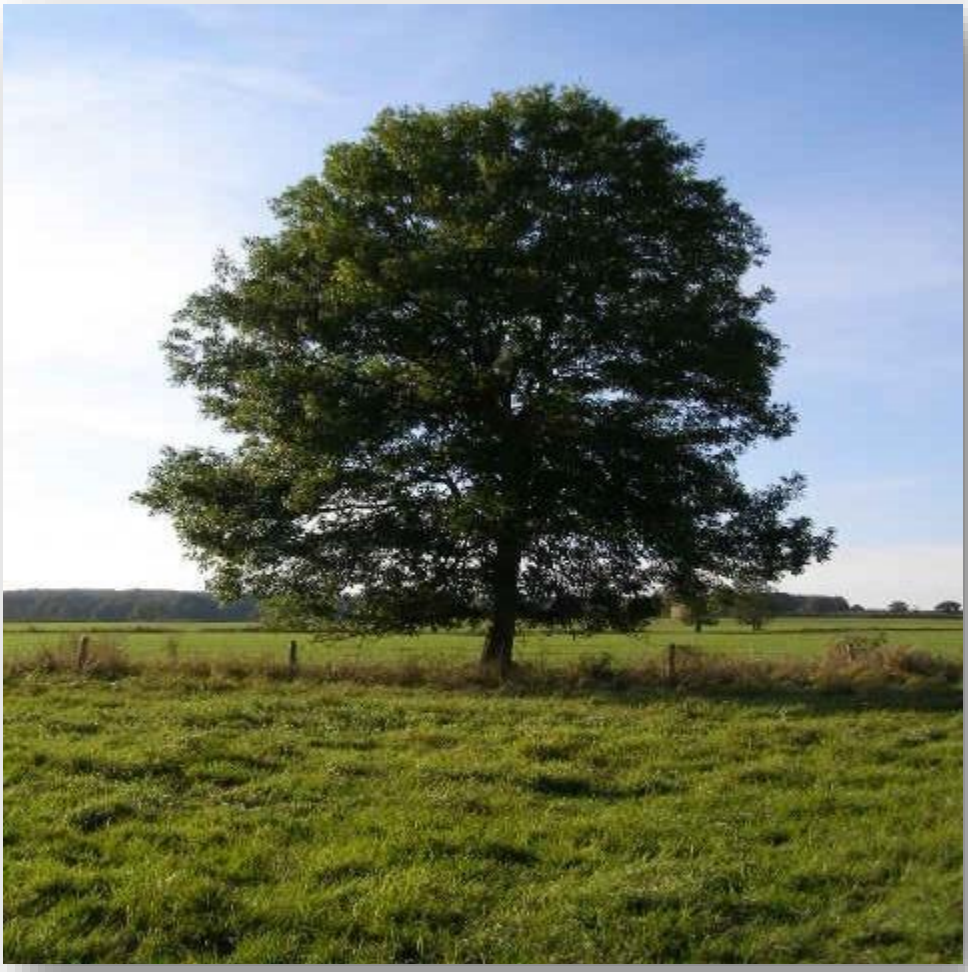
```
#!/bin/bash
nasm -f elf64 asm_io.asm &&
gcc -m64 -no-pie -std=c17 -c driver.c
nasm -f elf64 $1.asm &&
gcc -m64 -no-pie -std=c17 -o $1 driver.c $1.o asm_io.o &&
python3 in.py | ./template | python3 out.py &&
./$1
```

دستور اضافه شده در خط آخر، به این صورت عمل می کند که خروجی به دست آمده از اجرای برنامه in.py را که تنسور متناظر به فایل تصویری است، به عنوان ورودی به برنامه template.asm (که برنامه اسمبلی برای اجرای کانولوشن روی تصاویر است) می دهد و خروجی این برنامه را نیز به عنوان ورودی به برنامه out.py می دهد تا دوباره به فایل تصویری تبدیل بشود.

برنامه نوشته شده در فایل template.asm، به این صورت عمل می کند که تنسور ورودی را در قالب سه ماتریس r (برای اعدد مربوط به رنگ قرمز)، g (برای اعدد مربوط به رنگ سبز) و b (برای اعداد مربوط به رنگ آبی) ذخیره می کند. سپس روی هر کدام این ماتریس ها به صورت جداگانه با استفاده از کرنل داده شده در ورودی، عملیات convolution را اجرا می کند. خروجی های به دست آمده از این سه convolution را به ترتیب در r_result، g_result و b_result ذخیره می کند و آنها را به شکل یک تنسور خروجی می دهد.

در ادامه نمونه هایی از پردازش تصویر روی تصاویر متفاوت با کرنل های متفاوت را می آوریم.

نمونه هایی از پردازش تصویر



tree_original

تصویر بالا تصویر اصلی (original) می باشد. در ادامه، با استفاده از ۵ نوع کرنل مختلف عملیات 2D_convolution را روی آن انجام می دهیم. در هر قسمت ضمن معرفی کرنل مربوطه، توضیحاتی مختصر درباره نحوه عملکرد آن کرنل و اینکه چه کاربردی دارد ارائه خواهیم نمود.

دقت کنید که برای دسترسی به فایل این تصاویر و برخی تصاویر دیگر که روی آنها پردازش صورت گرفته، می توانید به زیر پوشه output_images واقع در پوشه CV مراجعه نمایید.

کرنل *identity*

tree_identity

$$identity\_kernel = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

همانطور که از نام این کرنل می توان فهمید، کرنل همانی همان عکس اولیه را دقیقا با همان جزئیات و مشخصات حفظ می کند. در واقع تصویر بالا با تصویر original هیچ تفاوتی ندارد.

کرنل *blur*

tree_blur

$$blur\_kernel = \begin{bmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{bmatrix}$$

این کرنل در واقع میانگین وزن دار پیکسل های موجود در اطراف یک پیکسل (به همراه خود آن پیکسل) را با خود آن پیکسل جایگزین می کند. در نتیجه در نقاطی که تیزی یا اختلاف رنگ وجود دارد، مقدار عددی پیکسل میانگین وزن دار پیکسل های مجاورش می شود و این باعث می شود که رنگ این پیکسل ها به پیکسل های مجاور شبیه تر شده و تصویر شفافیت کمتری پیدا بکند و مات شود.

کرنل *edge_detection*

tree_edge_detection

$$edge\_detection\_kernel = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

این کرنل در مورد پیکسل هایی که تفاوت رنگ قابل توجهی با پیکسل های مجاور خود دارند (به اصطلاح لبه هستند)، موجب تشدید رنگ آنها می گردد. دقت کنید که چون مجموع درایه ها در این کرنل برابر صفر می باشد، اگر یک پیکسل لبه نباشد و شباهت رنگی زیادی با پیکسل های مجاور خود داشته باشد، وقتی این کرنل روی آن پیکسل اعمال شود، حاصل تقریبا برابر صفر خواهد بود و آن پیکسل در تصویر نهایی تیره خواهد بود.

کرنل *emboss*

tree_emboss

$$emboss\_kernel = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

تمرکز اصلی این کرنل بر روی تغییر رنگ پیکسل هایی است که پیکسل های مجاورشان، قرینه نیستند. اگر به درایه های این کرنل دقت کنید، متوجه می شوید که اگر پیکسل های مجاور یک پیکسل کاملا نسبت به مرکزیت آن پیکسل قرینه باشند، تاثیر یکدیگر را خنثی می کنند و رنگ این پیکسل تغییر نخواهد کرد. اما اگر قرینگی برای پیکسل های مجاور یک پیکسل وجود نداشته باشد، رنگ آن پیکسل تغییر خواهد نمود و هر چقدر میزان این ناقرینگی بیشتر باشد، تغییر رنگ شدیدتری رخ خواهد داد.

کرنل *sharpen*

tree_sharpen

$$sharpen\_kernel = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

از این کرنل برای تیز کردن لبه ها (نقاطی که اختلاف رنگ قابل توجهی با پیکسل های بالا، پایین، چپ و راست خود دارند) استفاده می شود. با کمی دقت متوجه می شویم که اگر این پیکسل های مجاور کاملا مشابه خود پیکسل اصلی باشند، رنگ آن پیکسل تغییر نخواهد کرد. اما هر چقدر تفاوت رنگی این پیکسل ها بیشتر باشد، تغییر رنگ بیشتری برای پیکسل وسط خواهیم داشت.