پروژه درس ساختار و زبان کامپیوتر

رادین شاه دائی ۴۰۱۱۰۶۰۹۶

استاد: دکتر امیرحسین جهانگیر

دانشکده مهندسی کامپیوتر شریف

ترم ۱۴۰۲۱

فهرست مطالب

توضیح کلی پروژه

این پروژه شامل دو بخش ضرب ماتریس‌های مربعی و `2D-Convolution` می‌باشد که با زبان اسمبلی `x86` نوشته شده است. برای نشان دادن درستی بخش `2D-Convolution` پردازش تصویر انجام دادم که در ادامه به آن می‌پردازم.

چالش‌ها

با توجه به اینکه لپ‌تاپ من، `Macbook M1 Pro` می‌باشد که پردازنده‌ی آن `Arm` می‌باشد، مجبور به استفاده از ماشین مجازی شدم که راه اندازی آن کمی مشکل بود. با استفاده از نرم افزار `UTM` و ابزار `Qemu` پردازنده `x86` مجازی‌سازی یا `Emulate` کردم. پردازنده مجازی‌سازی شده پردازنده استاندارد `Haswell` استفاده کردم. این سیستم `4GB` رم داشته و فرکانس پردازنده‌ی آن نیز به تقریب `1GHz` می‌باشد.

- اطلاعات پردازنده با استفاده از دستور `lscpu`

```
radinshd@radinvm:~$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         40 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  4
  On-line CPU(s) list:   0-3
Vendor ID:               GenuineIntel
  Model name:            Intel Core Processor (Haswell)
    CPU family:          6
    Model:               60
    Thread(s) per core:  1
    Core(s) per socket:  4
    Socket(s):           1
    Stepping:            4
    BogoMIPS:            1997.63
```

- اطلاعات رم با استفاده از دستور `free -m`

```
radinshd@radinvm:~$ free -m
               total        used        free      shared  buff/cache   available
Mem:            3908         177        3379           5         350        3498
Swap:           3907           0        3907
```

- اطلاعات `CPU Frequency` با استفاده از دستور `cat /proc/cpuinfo | grep Hz`

```
radinshd@radinvm:~$ cat /proc/cpuinfo | grep "MHz"
cpu MHz         : 998.815
cpu MHz         : 998.815
cpu MHz         : 998.815
cpu MHz         : 998.815
```

برای پیدا کردن دستورهای مناسب نیز از سایت `felixcloutier.com` استفاده کردم. به عنوان مثال برای پیدا کردن دستورات پردازش موازی کلیدواژه‌ی `packed` را سرچ می‌کردم و بعنوان مثال، دستور `dpps` که دو بردار را در هم ضرب داخلی می‌کند را پیدا کردم.

ورودی و خروجی

برای ورودی گرفتن و خروجی دادن، از فایل `asm_io.asm` که در پیوست قرار گرفته استفاده کردم. در این فایل توابعی نظیر `read_float`، `read_int`، `print_float` و غیره پیاده سازی شده اند که کارهای `IO` را با کمک توابع آماده زبان `C` نظیر `puts`، `printf`، `scanf` و غیره انجام می‌دهد. فایل `asm_io.asm` درون پوشه‌های `src-asm` قرار گرفته است و در برنامه‌های اسمبلی خود بعنوان کتابخانه از آن استفاده کرده‌ام.

ران کردن برنامه‌ها

در پیوست، فایل اسکریپت `run.sh` قرار دارد که مراحل اسمبل کردن را با استفاده از `nasm` انجام داده و سپس برنامه و کتابخانه‌های خارجی استفاده شده از زبان `C` را لینک کرده و سپس فایل قابل اجرای برنامه ساخته می‌شود. بعد از آن برنامه اجرا می‌شود و پس از اجرای برنامه، `object-file` ها پاک می‌شوند.

```
#!/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 &&
./$1

rm -r asm_io.o driver.o $1.o
```

MATRIX MULTIPLICATION

فایل اسمبلی

برای این بخش از پروژه، دو فایل `multiply-normal.asm` و `multiply-packed.asm` دارم که به ترتیب ضرب ماتریس‌های مربعی با اندازه کمترمساوی ۸ را پیاده سازی کرده‌ام. این دو فایل در قسمت ورودی گرفتن با هم تفاوتی ندارند. ماتریس اول را به صورت عادی و با تابع `input_first` و ماتریس دوم را به صورت ترانهاده یا `Transposed` با استفاده از تابع `input_second` انجام داده‌ام. نحوه‌ی اجرای این توابع به طور دقیق با کامنت گذاری مناسب در فایل‌های اسمبلی توضیح داده شده است. سپس ضرب دو ماتریس را انجام دادم که دلیل نگهداری ماتریس دوم به صورت ترانهاده در همین تابع ضرب مشخص است. کد اسمبلی اینگونه بسیار سریعتر و تمیزتر خواهد بود و خوانایی بیشتری خواهد داشت. در هر دوم فایل تک تک مراحل تابع `multiply` به وضوح با کامنت گذاری های مناسب در فایل های اسمبلی توضیح داده شده است. در بخش آخر یا ماتریس `product` را با استفاده از تابع `print_matrix` و یا زمان اجرای برنامه با استفاده از دستور `rdtsc`، که کلاک `CPU` را درون `EDX:EAX` ذخیره سازی میکند را خروجی می‌دهم. توضیح مفیدی که در مورد تابع ضرب `multiply-packed` باید بدهم این است که با توجه با اینکه دستورات موازی سازی شده را با ثبات های `xmm` که هر کدام `128bit` هستند و بردارهای ۴ تایی از مقادیر `32bit floating-point` در آن‌ها ذخیره می‌شود، اگر سایز ماتریس از ۴ بیشتر بود، دو بار ثبات های `xmm` را با مقادیر یک سطر ماتریس اول و مقادیر یک ستون ماتریس دوم پر میکنم، و سپس از دستور `dpps` که ضرب داخلی است استفاده می‌کنم. توضیحات در کامنت‌های فایل اسمبلی نیز گفته شده است.

صحت اجرای برنامه

- برنامه‌ی `multiply-normal`

```
radinshd@radinvm:~/VM/templates/x86_template/Mulitply/src-asm$ ./run.sh multiply-normal
3

1 2 3
4 5 6
7 8 9

1.5 2.5 3.5
3.5 2.5 1.5
4.5 4.5 4.5
22.000000 21.000000 20.000000
50.500000 49.500000 48.500000
79.000000 78.000000 77.000000
radinshd@radinvm:~/VM/templates/x86_template/Mulitply/src-asm$ ./run.sh multiply-normal
5

1 2 3 4 5
1 2 3 4 5
6 6 6 6 6
5 4 3 2 1
5 4 3 2 1

1.1 2.2 3.3 4.4 5.5
1.1 2.2 3.3 4.4 5.5
6.6 6.6 6.6 6.6 6.6
5.5 4.4 3.3 2.2 1.1
5.5 4.4 3.3 2.2 1.1
72.599998 66.000000 59.399998 52.799999 46.200001
72.599998 66.000000 59.399998 52.799999 46.200001
118.800003 118.800003 118.800003 118.800003 118.799995
46.199997 52.799999 59.399998 66.000000 72.599998
46.199997 52.799999 59.399998 66.000000 72.599998
```

- برنامه‌ی `multiply-packed`

```
radinshd@radinvm:~/VM/templates/x86_template/Mulitply/src-asm$ ./run.sh multiply-packed
3

1 2 3
4 5 6
7 8 9

1.5 2.5 3.5
3.5 2.5 1.5
4.5 4.5 4.5
22.000000 21.000000 20.000000
50.500000 49.500000 48.500000
79.000000 78.000000 77.000000
radinshd@radinvm:~/VM/templates/x86_template/Mulitply/src-asm$ ./run.sh multiply-packed
5

1 2 3 4 5
1 2 3 4 5
6 6 6 6 6
5 4 3 2 1
5 4 3 2 1

1.1 2.2 3.3 4.4 5.5
1.1 2.2 3.3 4.4 5.5
6.6 6.6 6.6 6.6 6.6
5.5 4.4 3.3 2.2 1.1
5.5 4.4 3.3 2.2 1.1
72.599998 66.000000 59.400002 52.799999 46.199997
72.599998 66.000000 59.400002 52.799999 46.199997
118.800003 118.800003 118.800003 118.800003 118.799995
46.199997 52.799999 59.399998 65.999992 72.599998
46.199997 52.799999 59.399998 65.999992 72.599998
```

مقایسه سرعت اجرا

برای مقایسه سرعت اجرا، از یک اسکریپت `bash` به نام `benchmark.sh` استفاده کردم. در این اسکریپت، می‌توان به تعداد دلخواه برنامه‌ی را اجرا کرد و زمان اجرای این برنامه‌ها بررسی می شود. با توجه به کوچک بودن سایز ماتریس در برنامه‌هایی که نوشتیم، نمیتوانیم از سرعت ورودی گرفتن ماتریس ها صرف نظر کنیم، به همین علت قبل و بعد از اجرای توابع `multiply` که ضرب ماتریس را پیاده سازی کرده، اند، در برنامه‌ی اسمبلی از دستور `rdtsc` و در برنامه‌ی سطح بالا جاوا از دستور `System.nanoTime()` استفاده کردم. در برنامه‌های اسمبلی اختلاف کلاک `CPU` با فرکانس `1GHz` خروجی داده می‌شود، که عملا مقدار زمان سپری شده در مقیاس نانوثانیه است. در برنامه جاوا نیز اختلاف زمانی دو برنامه قبل و بعد از اجرای تابع `multiply` اندازه گیری می‌شود. با توجه به کوتاه بودن این زمان‌ها و بودن `noise` و `error` در سیستم، بهتر است هر کدام از برنامه‌ها را `10` الی `20` بار اجرا کنیم و میانگین بگیریم تا اختلاف زمانی دقیق اجرای این برنامه‌ها را داشته باشیم.

نمونه‌ی استفاده از دستور `System.nanoTime()` در برنامه‌ی جاوا:

```java
long startTime = System.nanoTime();
float[] product = multiply(matrix1, matrix2);
long endTime = System.nanoTime();
long duration = (endTime - startTime);
```

نمونه‌ی استفاده از دستور `rdtsc` در برنامه‌های اسمبلی:

```
rdtsc
mov dword [t1], eax     ; save lower bits in t1
call multiply_normal    ; call main multiply function
rdtsc
xor rdi, rdi
sub eax, dword [t1]     ; save difference of lower bits in eax
```

اجرای برنامه‌ی `benchmark.sh` با تعداد تکرار ۲۰

```
radinshd@radinvm:~/VM/templates/x86_template/Mulitply$ ./benchmark.sh  20
Program         | Average Time (ms)
x86 normal 3×3  | 68.25
x86 normal 5×5  | 71.25
x86 packed 3×3  | 52.75
x86 packed 5×5  | 69.20
Java 3×3        | 82.00
Java 5×5        | 351.85
```

ابتدا متوجه می‌شویم که در کل زمان اجرای تابع `multiply` در زمان اسمبلی مخصوصا در `n` بزرگ بسیار سریعتر است. واضحا زمان اجرای برنامه‌های پردازش موازی که با `packed` مشخص کردم از `normal` سریعتر بوده‌است. نکته‌ی حائز اهمیت، تفاوت کمتر زمان اجرا در ضرب `5x5` برای `packed` و `normal` می‌باشد که به این دلیل است که ضرب موازی در فایل اسمبلی `multiply-packed` به صورت ۴ تا ۴ تا انجام می‌شود و انگار برای ماتریس `5x5`، ۳ ضرب اضافی انجام می‌دهد که باز هم از ضرب عادی سریعتر می‌باشد.

تابع اجرای `benchmark` درون فایل `benchmark.sh`

```
benchmark() {
    local program_command="$1"
    local intput="$2"
    local nano=0

    local num_times="$3"

    for ((i=1; i<=num_times; i++))
    do
        output=$($program_command < "$2")

        value=$(echo $output | grep -oE '[0-9]+')

        nano=$((nano + value))
    done

    micro=$(bc <<< "scale=2; $nano / 1000")
    average=$(bc <<< "scale=2; $micro / $num_times")

    echo "$average"
}
```

نتیجه‌ای که می‌گیریم این است که با همان دانش الگوریتمی، نوشتن برنامه به زبان اسمبلی زمان اجرای برنامه را به شدت کاهش میدهد و باعث بهنیه‌سازی برنامه میشود.

2D-CONVOLUTION

فایل اسمبلی

در این برنامه، ابتدا `kernel` از کاربر ورودی گرفته می‌شود که یک ماتریس مربعی با سایز کمترمساوی ۵ می‌باشد. در ادامه یک ماتریس بزرگ از کاربر ورودی گرفته می‌شود که تا عکس‌هایی با کیفیت `Full HD` را نیز پشتیبانی می‌کند . در واقعیت این ورودی ها با استفاده از یک فایل `input.txt` و خروجی برنامه نیز در فایل `output.txt` ذخیره می‌شود که در اسکریپت `bash` به توضیح آن بخش می‌پردازم. برای تابع `convolution` از روش `edge-extension` استفاده کردم و این کار را با تابع `correct_index` انجام دادم. برای اجرای `convolution` نیز روی ماتریس بزرگ عکس پیمایش کرده، ماتریس‌های کوچکی حول هر پیکسل ساخته و آن ماتریس کوچک را در ماتریس `kernel` ضرب داخلی می‌کنم که این را با توابع `normal_dot` یا `packed_dot` انجام دادم. در نهایت پس از ساخت هر پیکسل عکس خروجی آن را به `Integer` تبدیل می‌کنم و خروجی را چاپ می‌کنم، که در واقع درون فایل `output.txt` ذخیره می شود.

فایلهای جاوا

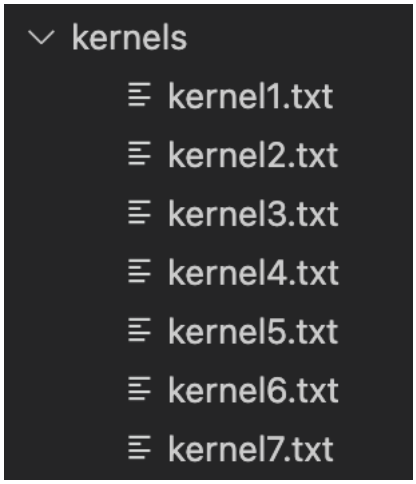
در این برنامه ۲ فایل جاوا به نام‌های `Main` و `Convertor` استفاده می‌شوند. فایل `Main` یک عکس `PNG` را به یک فایل تکست که شامل سایز عکس و مقدار هر پیکسل که عددی بین `0` تا `255` است تبدیل میکند و فایل `Convertor` دقیقا برعکس این کار را انجام می‌دهد. این فایلها را برای این ساخته‌ام که کاربرد `Convolution` که در پردازش تصویر دیده می شود را به خوبی نمایش دهم.

فایل Bash

این فایل که با دستور `./convolution.sh` اجرا می‌شود، از کاربر می‌خواهد عکسی که به دنبال انجام پردازش تصویر روی آن است را انجام دهد را انتحاب کرده، سپس `kernel` های موجود را به کاربر نشان می‌دهد و از کاربر می‌خواهد یک کدام را انتخاب کند، و سپس عکس خروجی را در فایل `output.png` ذخیره میکند. تبدیل عکس به فایل تکست و برعکس با کمک برنامه‌های جاوا بوده است و خود انجام `convolution` با برنامه‌ی اسمبلی صورت گرفته است. این برنامه به اصطلاح `full-proof` است، یعنی چک میکند که عکسی که کاربر ورودی میدهد واقعا وجود داشته باشد و یا از `kernel` های موجود در فولدر `kernels` فقط انتخاب میکند. در واقع هنگام دادن محصول به مشتری، تنها اجازه‌ی اجرای همین فایل برای کاربر داده می‌شود و کاربر خریدار نمیتواند باقی فایلها را بخواند و بنویسد.

فولدر kernels

در این فولدر نیز `kernel` های متنوعی برای پردازش تصویر قرار گرفته است.




```
1: Identity
2: 3x3 Blur
3: 5x5 Blur
4: Sharpen
5: 3x3 Gaussian Blur
6: 5x5 Gaussian Blur
7: Edge detection
```

نمونه‌ای از اجرای فایل Bash

۱) شروع برنامه:

۲) انتخاب عکس، در اینجا انتخاب اشتباه عکس را نیز آورده شده است.

```
Please enter the name of your PNG file: No
Invalid file. Please enter the name of a valid PNG file.

Please enter the name of your PNG file: cat.png
Running Java on your image...
```

۳) انتخاب `kernel`:

```
Kernels:
  1: Identity
  2: 3x3 Blur
  3: 5x5 Blur
  4: Sharpen
  5: 3x3 Gaussian Blur
  6: 5x5 Gaussian Blur
  7: Edge detection
Please pick a kernel:
```

۴) ذخیره کردن عکس خروجی:

```
Creating output image as "output.png"...
PNG image created successfully.
```

نمونه‌هایی از اجرای پردازش تصویر

۱) تغییر `cat.png` با استفاده از `5x5 Gussian Blur`

عکس ورودی:

عکس خروجی:

۲) تغییر عکس `radin.png` با استفاده از `edge detection`

عکس ورودی:

عکس خروجی:

۳) تغییر عکس `doctor.png` با استفاده از `sharpen`

عکس ورودی:

عکس خروجی:

مقایسه سرعت با زبان سطح بالا

برای مقایسه سرعت، دقیقا همان الگوریتمی که در برنامه اسمبلی پیاده سازی کرده بودم را با برنامه‌ی جاوا پیاده سازی کردم. با استفاده از فایل `benchmark.sh` می‌توانیم تفاوات سرعت پردازش تابع `convolution` این دو برنامه را برای عکس `cat.png` ببینیم.

```
radinshd@radinvm:~/VM/templates/x86_template/Convolution$ ./benchmark.sh
|  Program    | Exec. Time  |
|-------------|-------------|
| x86 packed  | 2.55920     |
|    java     | 21.28447    |
```

همانطور که مشخص است، برنامه اسمبلی `2.5s` زمان برده در حالی که برنامه جاوا `21.3s` که تقریبا `8x` کندتر می باشد. همین باعث می‌شود که خریدار از محصول ما نسبت به همان محصول با زبان های سطح بالا راضی تر باشد و محصول ما را خریداری کند!

فایل `benchmark.sh`

```bash
#!/bin/bash

cd src-java
output=$(java Main <<< "cat.png")
cd ..

cat kernels/kernel6.txt src-java/output.txt >> input.txt


input_file="../input.txt"


benchmark() {
    local program=$1
    local input_file=$2

    start_time=$(date +%s.%N)
    $program < $input_file > /dev/null
    end_time=$(date +%s.%N)
    execution_time=$(echo "$end_time - $start_time" | bc)
    printf "%0.5f\t  " "$execution_time"
}

# Print table header
echo "|  Program    | Exec. Time  |"
echo "|-------------|-------------|"

cd src-asm
echo "| x86 packed | $(benchmark "./run.sh convolution-packed" "$input_file") |"
cd ../src-java
echo "|    java    | $(benchmark "java Convolution" "$input_file") |"
cd ..



rm input.txt src-java/output.txt
```