

به نام خدا



دانشکده مهندسی کامپیوتر

پروژه پایانی درس ساختار زبان کامپیوتر

استاد : دکتر جهانگیر

دانشجو : مهدی بهرامیان شماره دانشجویی : ۴۰۱۱۷۱۵۹۳

۱۴۰۲ بهمن

فهرست مطالب

۲ فهرست مطالب
۳ فهرست مطالب
۴ توضیح ساختار پروژه
۴ پوشه src
۴ فایل main.c
۴ فایل asm_io.S
۴ فایل matmul.S
۵ فایل matmul_sisd.S
۵ فایل matmul_dotprod.S
۵ فایل matmul_dotprod_sisd.S
۵ فایل matmul_normalalgo.c
۵ فایل matmul_samealgo.c
۵ پوشه benchmarking
۶ فایل benchmark.sh
۶ فایل diffchecker.c
۶ فایل testgen.py
۷ پوشه Src
۷ فایل main.c
۷ فایل asm_io.S
۸ فایل convolution.S
۸ فایل convolution_sisd.S
۸ فایل convolution_matmul_simd.py
۸ فایل convolution_matmul_sisd.py
۸ پوشه benchmarking
۹ فایل benchmark.sh
۹ فایل diffchecker.c
۹ فایل testgen.py
۹ پوشه image_processing
۹ فایل imagereader.py
۹ فایل imagewriter.py
۹ فایل process.sh
۱۰ ضرب ماتریسی
۱۰ توضیح بخش های کلی و تکرار شده کد
۱۳ روش اول - استفاده از ضرب داخلی بردار ها
۱۳ توضیح روش

۱۳	توضیح روش برداری سازی.....
۱۴	توضیح بخش برداری سازی شده.....
۱۰	روش دوم - ۳ حلقه تو در تو.....
۱۰	توضیح روش.....
۱۷	توضیح روش برداری سازی.....
۱۸	توضیح بخش برداری سازی شده.....
۱۹	نتایج آزمایش.....
۲۱	کانولوشن.....
۲۱	توضیح بخش های کلی و تکرار شده کد.....
۲۲	روش اول - استفاده از ضرب خارجی ماتریسی (ضرب عادی ماتریس).....
۲۲	توضیح روش.....
۲۴	توضیح روش برداری سازی.....
۲۵	توضیح بخش برداری سازی شده.....
۲۵	روش دوم - استفاده از ضرب داخلی ماتریسی.....
۲۵	توضیح روش.....
۲۷	توضیح روش برداری سازی.....
۲۷	توضیح بخش برداری سازی شده.....
۲۷	نتایج آزمایش.....
۲۷	آزمون سرعت.....
۲۸	پردازش تصویر.....
۲۸	کرنل های ما
۲۹	خروجی پردازش.....
۳۴	نتیجه گیری.....

توضیح ساختار پروژه

پوشش **src**

فایل main.c

این یک فایل C است که به عنوان واسطه میان دستورات printf و scanf عمل میکند و توابعی مانند write_float، read_float را شامل میشود.

فایل asm_io.S

این فایل یک کد اسembly است که توابع main.c را صدا میزند و رجیستر های rbx، rcx، rsi را در استک پوش و پاپ میکند تا دسترسی به توابع C مورد نیاز ما در کد اسembly اصلی راحت تر باشد و نیاز نباشد در هر بار صدا زدن توابع ما نیاز به پوش و پاپ دستی قبل و بعد از صدا زدن توابع باشیم که باعث تمیز تر شدن کد میشود.

فایل matmul.S

این کد ضرب ماتریسی را با استفاده از الگوریتم سه حلقه تو در تو محاسبه میکند و از نظر الگوریتمی مانند SIMD است و از دستورات ymm و رجیستر های SIMD بهره میبرد.

فایل matmul_sisd.S

این کد نسخه matmul.S ی Single Instruction Single Data است.

فایل matmul_dotprod.S

این کد ضرب ماتریسی را با استفاده از الگوریتم ضرب داخلی بردار های سطر های A و B^T را در هم ضرب میکند و در ماتریس خروجی قرار میدهد. این کد از نظر الگوریتمی مانند matmul_samealgo.c است و از دستورات SIMD و رجیستر های ymm بهره میبرد.

فایل matmul_dotprod_sisd.S

این کد نسخه matmul_dotprod.S ی Single Instruction Single Data است

فایل matmul_normalalgo.c

```
for (int i = 0; i < n; i++) {  
    for (int k = 0; k < n; k++) {  
        for (int j = 0; j < n; j++) {  
            mat3[i][j] += mat1[i][k] * mat2[k][j];  
        }  
    }  
}
```

این کد ضرب ماتریسی را به صورت عادی انجام میدهد منتها با این تفاوت که برای سرعت بالاتر به جای این که ترتیب حلقه ها k, j, i باشد برای تسريع در دسترسی به حافظه و عدم cache miss ترتیب اندیس ها به i, k, j تغییر یافته است.

فایل matmul_samealgo.c

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        for (int k = 0; k < n; k++) {  
            mat3[i][j] += mat1[i][k] * mat2[j][k];  
        }  
    }  
}
```

این کد mat1 را به عنوان A ورودی میگیرد و mat2 را به عنوان B^T ورودی میگیرد و سپس همانطور که دیده میشود $\sum_k mat1[i][k]*mat2[j][k]=\langle mat1[i], mat2[j] \rangle$ برابر میشود با mat3[i][j] که همانطور که میبینید این عبارت ضرب نقطه ای را حساب میکند.

پوشه benchmarking

این پوشه شامل کلیه فرایند های آزمون پیاده سازی های مختلف برای ضرب ماتریسی است

benchmark.sh فایل

این کد مسئولیت آزمون انواع روش های پیاده شده برای انجام ضرب ماتریسی را بر عهده دارد.

diffchecker.c فایل

این کد مسئول آن است که بین ۲ پیاده سازی بررسی میکند که آیا خروجی آنان تفاوت محسوسی دارند یا خیر.

testgen.py فایل

این کد مسئول آن است که ۶ تست تصادفی در بازه های متفاوت اندازه ماتریس تولید میکند.

```

 convolution
 └── benchmarking
     ├── benchmark.sh
     ├── diffchecker.c
     └── testgen.py
 ├── image_processing
     ├── dstimg
     ├── kernels
     ├── srcimg
     ├── asm_exec
     ├── convolution
     ├── convolution.o
     ├── imagereader.py
     ├── imagewriter.py
     └── process.sh
 └── src
     ├── asm_io.S
     ├── convolution.S
     ├── convolution.c
     ├── convolution_matmul_simd.S
     ├── convolution_matmul_sisd.S
     ├── convolution_sisd.S
     ├── main.c
     ├── main.rs
     └── playground.S
 └── Cargo.lock
 └── Cargo.toml
 └── build.sh

```

پوشه SRC

فایل main.c

این یک فایل c است که به عنوان واسطه میان دستورات printf و scanf عمل میکند و توابعی مانند write_float این را شامل میشود.

فایل asm_io.S

این فایل یک کد اسملی است که توابع main.c را صدا میزند و رجیستر های rbx, rcx, rsi را در استک پوش و پاپ میکند تا دسترسی به توابع c مورد نیاز ما در کد اسملی اصلی راحت تر باشد و نیاز نباشد در هر بار صدا زدن توابع ما نیاز به پوش و پاپ دستی قبل و بعد از صدا زدن توابع باشیم که باعث تمیز تر شدن کد میشود.

فایل convolution.S

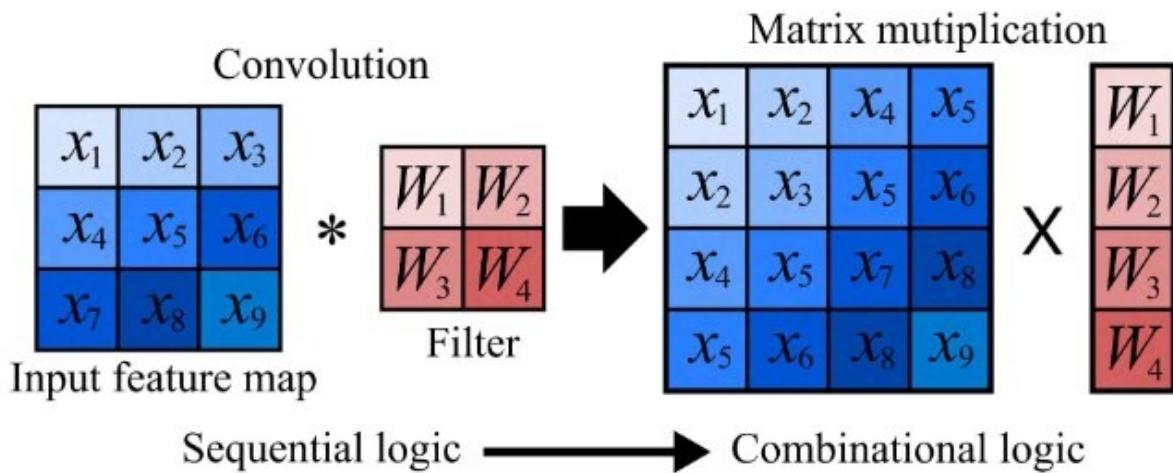
این کد کانولوشن را به روش ضرب نقطه ای ماتریس ها انجام میدهد و از دستورات SIMD و رجیستر های YMM برای تسریع الگوریتم استفاده میکند.

فایل convolution_sisd.S

این کد نسخه convolution.S است که Single Instruction Single Data است.

فایل convolution_matmul_simd.py

این کد کانولوشن را به روش ضرب ماتریسی به صورت مشابه زیر انجام میدهد.



در واقع محاسبات مورد نیاز به صورت یک به یک به این فرم تبدیل میشود و برای انجام این روش ابتدا ماتریس آبی سمت راست را از روی ماتریس ورودی میسازیم و سپس آنرا به الگوریتم ضرب ماتریسی میدهیم و خروجی آنرا چاپ میکنیم و در این کد از دستورات SIMD و رجیستر های YMM برای تسریع استفاده شده است.

فایل convolution_matmul_sisd.py

این کد نسخه convolution_matmul_sisd.py است که Single Instruction Single Data است.

پوشه benchmarking

این پوشه شامل کلیه فرایند های آزمون پیاده سازی های مختلف ضرب ماتریسی است

benchmark.sh فایل

این کد مسئولیت آزمون انواع روش های پیاده شده برای انجام ضرب ماتریسی را بر عهده دارد

diffchecker.c فایل

این کد مسئول آن است که بین ۲ پیاده سازی بررسی میکند که آیا خروجی آنان تفاوت محسوسی دارند یا خیر

testgen.py فایل

این کد مسئول آن است که ۶ تست تصادفی در بازه های متفاوت اندازه ماتریس تولید میکند.

image_processing پوشه

این پوشه شامل کدهای لازم برای انجام عملیات پردازش تصویر است

imagereader.py فایل

این کد آدرس یک تصویر و یک کرنل را از آرگومان های شل ورودی میگیرد و در نهایت ماتریس پیکسل های سیاه و سفید تصویر و مقادیر کرنل را خروجی میدهد.

imagewriter.py فایل

این کد آدرس مقصد را از آرگومان های شل، و ماتریس نهایی بعد از پردازش را از ورودی میگیرد و در نهایت ماتریس ورودی داده شده را به صورت یک عکس در مقصد داده شده میسازد.

process.sh فایل

این یک کد شل است که به ازای هر تصویر در پوشه kernels و به ازای هر کرنل در پوشه srcimg آن تصویر را توسط این کرنل با استفاده از دستورات pipeline شل پردازش میکند و در پوشه dstimg قرار میدهد.

ضرب ماتریسی

توضیح بخش های کلی و تکرار شده کد

```
global asm_main
%include "../src/asm_io.S"
```

این بخش تابع `asm_main` را به صورت `global` تعریف میکند تا لینکر بتواند اشاره به این تابع را متوجه شود و بعد از آن فایل `asm_io.S` را اینکلود میکند (در واقع محتویات فایل با این خط جایگزین میشوند) که در نتیجه میتوانیم از توابع نوشته شده درون آن استفاده کنیم.

```
ReadDIM:
call    asm_read_uint
        cmp    rax, 512
        jle    OkDIM
        mov    rax, wrong_dim_error
        call   asm_write_str
        jmp    ReadDIM

OkDIM:
        mov    qword[n], rax
        mov    bl, al
        shr    rax, 3
        and    bl, 0x07
        jz    OkNV
        inc    rax
OkNV:
        mov    qword[nv], rax
```

این بخش مسئول ورودی گرفتن n (اندازه ماتریس ها) است. و اگر ابعاد ماتریس داده شده درست نباشد ($n \not\leq 512$) برقرار نباشد خطای مناسبی را چاپ میکند و به مرحله خواندن باز میگردد و در غیر این صورت به `OkDIM` میرود که به این مفهوم است که n را به درستی ورودی گرفته است. سپس برای بخش برداری سازی نیاز است که $\lceil \frac{n}{8} \rceil$ را نیز محاسبه کنیم زیرا به

جای این که به ازای $i = 0, 1, \dots, n-1$ بخواهیم یک تک عملیات انجام دهیم باید به ازای $\lceil \frac{n}{8} \rceil - 1$ عملیات مواری انجام دهیم که به یک نتیجه منتها در زمان سریعتر میرسیم. و برای این که این عبارت را محاسبه کنیم میتوانیم

n را ۳ بار به راست شیفت دهیم و چک کنیم که اگر آن ۳ بیت حذف شده ناصرف بودند بنابر این باید یک واحد به مقدار `rax` اضافه کنیم که مقدار `rax` به عبارت مورد نظر تبدیل شود و در نهایت آنرا درون `nV` ذخیره میکنیم.

```
mov         rcx, qword[n]
xor         rsi, rsi
MAT1_i: ; rbx
    mov         rbx, rcx
    mov         rcx, qword[n]
MAT1_j: ; rcx
    asm_read_float
    movss      [mat1 + rsi * 4], xmm0
    inc         rsi
    loop       MAT1_j
    dec         rsi
    and         si, 0xFE00
    add         rsi, 0x0200
    mov         rcx, rbx
    loop       MAT1_i
```

در این قطعه کد با استفاده از یک حلقه تو در تو ماتریس mat1 را ورودی میگیریم و از rsi به عنوان اندیس حافظه mat1 رفتار میکند و داریم از ۹ بیت سمت راست rsi به عنوان ۰ استفاده میکنیم و از ۹ بیت دوم آن به عنوان ۱ استفاده میکنیم. بنابراین بعد از اتمام حلقه داخلی (حلقه j باید j را ۰ کنیم که این موضوع معادل این است که ۹ بیت سمت راست rsi را ۰ کنیم که این کار معادل این است که $rsi = 0x00FE00$ اند کنیم و بعد از این مورد باید ۱ را یکی افزایش دهیم که این کار معادل این است که $rsi = 0x00000001$ باشد. جمع کنیم (این عدد در واقع بیت ۱۰ است و منظاروی ۹ بیت اول هیچ اثری نمیگذارد و اثر آن روی ۹ بیت دوم معادل $+1$ است که در واقع این کار باعث افزایش ۱ به اندازه ۱ میشود) و در هر حلقه مقدار ورودی گرفته شده را که درون $xmm0$ ورودی گرفته ایم را درون $[rsi][mat1]$ میریزیم (ضرب در ۴ به علت این که rsi ما اندیس بر مبنای تعداد float است بنابراین ما باید آنرا به اندیس بر مبنای byte تبدیل کنیم) پس بنابراین این قطعه کد ماتریس mat1 را ورودی میگیرد.

```
mov        rcx, qword[n]
xor        rsi, rsi
MAT2_i: ; rbx
    mov        rbx, rcx
    mov        rcx, qword[n]
    MAT2_j: ; rcx
        |    asm_read_float
        |    movss    [mat2 + rsi * 4], xmm0
        |    add      rsi, 0x200
        |    loop    MAT2_j
    and      rsi, 0x1FF
    inc      rsi
    mov      rcx, rbx
    loop    MAT2_i
```

در این قطعه کد ماتریس $mat2$ را به صورت ترانهاده ورودی میگیریم و این که به این صورت کار میکند که rsi به عنوان اندیس روی ماتریس $mat2$ کار میکند و مانند قسمت قبلی ۹ بیت اول آن ز است و ۹ بیت دوم آن α است و چون داریم ماتریس را به صورت ترانهاده ورودی میگیریم در حلقه داخلی باید $1 + \alpha$ شود (برای این کار کافیست مانند قسمت قبل $0x200$ را به $rsi + rsi$ شود) و در حلقه بیرونی باید $1 + \beta$ شود. (برای این کار کافیست $1 + rsi + rsi$ شود) و $\beta = \alpha$ شود (برای این کار کافیست rsi را با $0x1FF$ برابر کنیم) توجه کنید که اگر بخواهیم ماتریس $mat2$ را به صورت عادی ورودی بگیریم مانند بخش قبل عمل میکنیم.

```

    mov        rbx, qword[n]
    xor        rsi, rsi
PRINT_i: ; rbx
    mov        rcx, qword[n]
PRINT_j: ; rcx
    movss    xmm0, [mat3 + rsi * 4]
    asm_write_float
    mov        rax, ''
    asm_write_char
    inc        rsi
    loop     PRINT_j
dec        rsi
and        si, 0xFE00
add        rsi, 0x0200
mov        rax, 10
asm_write_char
dec        rbx
jnz        PRINT_i

```

این قطعه کد وظیفه چاپ کردن mat^3 را دارد و از نظر عملکردی بسیار مشابه ورودی گرفتن $mat1$ است با این تفاوت که این کد به جای ورودی گرفتن، $xmm0$ را با $[r1 * 4]$ پر میکنیم و آنرا چاپ میکنیم و بین خروجی های یک خط کارکتر فاصله چاپ میکنیم و بین خطوط کارکتر اینتر را چاپ میکنیم که معادل ۱۰ در اسکی است.

```

section .data align=64
mat1:
| dd 0x40000 DUP(0.0)
mat2:
| dd 0x40000 DUP(0.0)
mat3:
| dd 0x40000 DUP(0.0)
tmp_v1:
| dd 8 DUP(0.0)
tmp_v2:
| dd 8 DUP(0.0)
zero_v:
| dd 8 DUP(0.0)
one_v:
| dd 8 DUP(1.0)

n: dq 0
nv: dq 0

section .rodata
input_dim_message: db "input dim of matrix (0 <= n <= 512) : " , 0
wrong_dim_error: db "input a valid n (0 <= n <= 512)" , 10 , 0

```

در این قطعه از کد حافظه مورد نظر برای ماتریس ها را میگیریم و این حافظه را با ۶۴ alignment میسازیم زیرا در هندگام موازی سازی و استفاده از دستور `movaps` این دستور با آدرس های حافظه که آدرس های آنها مضرب ۳۲ باشد کار میکند و در غیر این صورت باید از `movups` استفاده کنیم که بسیار کند تر از `movaps` است بنابر این حافظه به این صورت قرار گرفته که از حداقل توان سخت افزار بتوانیم استفاده کنیم.

روش اول - استفاده از ضرب داخلی بردار ها

توضیح روش

$$(A \times B)[i][j] = \sum_k A[i][k]B[k][j] = \sum_k A[i][k]B^T[j][k] = \langle A[i], B^T[j] \rangle$$

بنابر این در این روش ما با استفاده از ماتریس A و ترانهاده ماتریس B میتوانیم به سادگی خانه های ماتریس $A \times B$ را با استفاده از ضرب داخلی سطر های ماتریس های A و B بدست بیاوریم که درون حافظه پشت سر هم قرار دارند بنابر این از نظر دسترسی به حافظه و استفاده بهینه از حافظه نهان مناسب هستند.

```
mov      rcx, qword[n]
xor      rax, rax
xor      rsi, rsi
MAT3_i: ; rbx
    mov      rbx, rcx
    mov      rcx, qword[n]
    xor      rdi, rdi
    MAT3_j: ; r8
        mov      r8, rcx
        mov      rcx, qword[n]
        calc_dot
        mov      rcx, r8
        movss   [mat3 + rax * 4], xmm0
        inc      rax
        and      di, 0xFE00
        and      si, 0xFE00
        add      rdi, 0x0200
        loop    MAT3_j
        dec      rax
        and      ax, 0xFE00
        add      rax, 0x0200
        mov      rcx, rbx
        add      rsi, 0x0200
        dec      rcx
        jnz     MAT3_i
```

در این قطعه کد ما با استفاده از حلقه تو در تو در هر مرحله سعی در محاسبه $[A \times B][i][j]$ داریم و برای انجام این کار باید از استفاده کنیم که rcx و rsi و rdi را به عنوان ورودی میگیرد و به این شکل کار میکند که rcx باید برابر با ابعاد بردار در حال ضرب باشد که در اینجا برابر با n میشود و rsi و rdi هر کدام باید اندیسی به اولین خانه سطر مورد نظر به ترتیب در mat_1 و mat_2 باشند بنابر این چون حلقه بیرونی مسئول آ و حلقه درونی مسئول ز است و چون rsi باید اندیس به خانه $[mat_1[i][j]]$ باشد و rdi باید اندیس به خانه $[mat_2[j][i]]$ باشد بنابر این rdi باید به ازای هر حلقه داخلی $+ 0x200$ شود و قبل از شروع حلقه داخلی باید rsi و rdi باید به ازای هر حلقه بیرونی $+ 0x200$ شود و قبل از شروع حلقه 0 شود و البته چون خود تابع `calc_dot` روی rsi و rdi اثر میگذارد باید آنان را به حالت اولیه خود باز گردانیم که برای این کار کافیست مانند گذشته^۹ بیت سمت راست آنرا 0 کنیم که این کار با اندگرفتن si و di با $0x00$ اتفاق میفتد. و بقیه بخش های این قطعه کد صرفا مسئولیت کنترل حلقه هارا به عهده دارند.

```
%macro calc_dot 0 ;calculate [rsi] dot [rdi] as vector size : rcx return xmm0
    movss    xmm0, [zero_v]
    calc_dot_loop:
        movss    xmm1, [mat1 + rsi * 4]
        movss    xmm2, [mat2 + rdi * 4]
        mulss    xmm1, xmm2
        addss    xmm0, xmm1
        inc      rsi
        inc      rdi
        loop    calc_dot_loop
    dec      rsi
    dec      rdi
%endmacro
```

و در نهایت به این شکل ضرب نقطه ای محاسبه میشود. همانطور که در کامنت نوشته شده، این کد `rsi` و `rdi` را به عنوان اندیس خانه اول سطر های مورد نظر به ترتیب از `mat1` و `mat2` میگیرد و آنها را به عنوان بردار هایی به اندازه `rcx` در نظر گرفته، ضرب داخلیشان را درون `xmm0` محاسبه میکند. برای محاسبه این عبارت حلقه ای ایجاد کرده ایم که به علت این که از دستور `loop` استفاده میکنیم به اندازه `rcx` اولیه اتفاق میفتد و در هر مرحله از این حلقه $[rsi*4]$ را در `xmm1` و $[rdi*4]$ را در `mat2[rdi*4]` میریزیم و سپس آنها را در هم ضرب کرده و $[xmm0]$ را با حاصل ضرب جمع میکنیم و در نهایت `rsi` و `rdi` را یکی زیاد میکنیم. بنابر این بعد از اولین حلقه $xmm0 = mat1[rsi*4]mat2[rdi*4]$

و بعد از حلقه دوم $xmm0 = mat1[rsi*4+4]mat2[rdi*4+4] + mat1[rsi*4]mat2[rdi*4]$ و همانطور ادامه پیدا میکند و بعد از اتمام حلقه $xmm0 = \sum_k mat1[(rsi+k)*4]mat2[(rdi+k)*4]$ خواهد بود و چون در واقع `rsi` و `rdi` در ابتدا در واقع برابر با خانه ابتدایی سطر خود بودند و از طرفی ماتریس ها را طوری در حافظه نگه میداریم که سطرهای پشت سر هم قرار داشته باشند بنابر این با این عملیات عبارت $\sum_{k=0}^{k=rcx-1} mat1[i][k]mat2[j][k]$ را محاسبه میکنیم و با توجه به این $A = A^T$ و $mat2 = B^T$ و $mat1 = n$ بنابر این این عبارت برابر با $\sum_k A[i][k]B^T[j][k]$ است.

بنابر این کد اسمبلي ما همانطور که توضیح داده شد رفتار میکند پس به درستی ضرب ماتریسی را پیاده سازی میکند.

توضیح روش برداری سازی

برای تسریع این الگوریتم ما باید بخش `calc_dot` را تسریع کنیم و برای این تسریع باید محاسبه ای که انجام میشود را بررسی کنیم و نکته ای که وجود دارد این است که حلقه های متفاوت از یکدیگر مستقل هستند و اگر بتوانیم همزمان $xmm0$ را هم $+ mat1[rsi*4+4]*mat2[rdi*4+4]$ و هم $+ mat1[rsi*4+8]*mat2[rdi*4+8]$ کنیم در این صورت میتوانیم در مراحل کمتری $xmm0$ را حساب کنیم. بنابر این روش ما به این صورت است :

$$xmm0 = xmm0 + \begin{bmatrix} mat1[rsi*4] & mat2[rdi*4] \\ mat1[rsi*4+4] & mat2[rdi*4+4] \\ \dots & \dots \\ mat1[rsi*4+28] & mat2[rdi*4+28] \end{bmatrix}$$

و با این حرکت ما ۸ عملیات ضرب و تعدادی عملیات جمع را به صورت همزمان انجام می‌شود و این رابطه را با استفاده از دستورات `vdpps` و `movaps` و `vperm2f128` انجام داد.

توضیح بخش برداری سازی شده

```
%macro calc_dot 0 ;calculate [rsi] dot [rdi] as vector size : rcx*8 return xmm0
    movss      xmm0, [zero_v]
    calc_dot_loop:
        vmovaps   ymm1, [mat1 + rsi * 4]
        vmovaps   ymm2, [mat2 + rdi * 4]
        vdpps     ymm1, ymm1, ymm2, 0xF1
        vperm2f128 ymm2, ymm1, ymm1, 0x01
        addss     xmm1, xmm2
        addss     xmm0, xmm1
        add       rsi, 8
        add       rdi, 8
        loop      calc_dot_loop
    sub       rsi, 8
    sub       rdi, 8
%endmacro
```

این قطعه کد بسیار شبیه به حالت غیر برداری خود است با این تفاوت که به جای این که با استفاده از دستور `movss` یک خانه از ماتریس‌ها را بخواند، با استفاده از دستور `movaps` یک قطعه ۸ تایی از ماتریس مورد نظر را می‌خواند و در رجیستر `ymm` مورد نظر میریزد. و به جای دستور `mulss` و `addss` از دستور `vperm2f128` و `vdpps` استفاده شده که دستور `add` اول `ymm`‌های داده شده را به دو تیکه بالا و پایین تبدیل می‌کند و بعد هر بخش را با توجه به عدد هگزی که بعد از آن داده شده است ضرب داخلی می‌کند و در همان بخش میگذارد بنابر این باید بعد از انجام ضرب داخلی دو بخش را با یکدیگر دوباره تلفیق کنیم که این کار با استفاده از ۲ دستور بعدی یعنی `vperm2f128` و `addss` این تلفیق را انجام میدهیم. و در نهایت خروجی ضرب داخلی انجام شده را با `xmm0` جمع می‌کنیم و از بیرون تقریباً هیچ تفاوتی نمی‌کند به جز این که `rcx` باید به جای این که n باشد باید nv باشد (n در واقع $\frac{n}{8}$ است) و `rsi` و `rdi` به جای این که هر مرحله $+1$ شوند باید $+8$ شوند

(چون روی ماتریس‌ها با فاصله ۸ تایی پرش می‌کنیم) و این تمام تغییر است.

روش دوم - ۳ حلقه تو در تو

توضیح روش

در این روش ما با ترتیب j, k, i حلقه می‌زنیم و ضرب ماتریسی محاسبه می‌کنیم.

```

; rsi {i , k}
; rdx {i , j}
; rdi {k , j}
xor    rsi,rsi
xor    rdx,rdx
mov    r9, qword[n]
i:
xor    rdi,rdi
mov    r8, qword[n]
k:
movss      xmm0, [mat1 + rsi * 4] ; load mat1[i][k]
inc     si
mov    rcx, qword[n]
movss      xmm2, [mat2 + rdi * 4] ; load first mat2 cell(mat2[k][0])
movss      xmm3, [mat3 + rdx * 4] ; load first mat3 cell(mat3[i][0])
j:
vmulss    xmm1, xmm0 , xmm2; xmm2 := mat1[i][k] * mat2[k][j]
movss      xmm2, [mat2 + rdi * 4 + 4] ; load next mat2 cell
addss    xmm1, xmm3 ; xmm2 := mat3[i][j] + mat1[i][k] * mat2[k][j]
movss      xmm3, [mat3 + rdx * 4 + 4] ; load next mat3 cell
movss      [mat3 + rdx * 4], xmm1 ; rdx = i * 0x0200 + j
inc     di
inc     dx
loop    j
dec     dx
and    dx, 0xFE00
dec     di
and    di, 0xFE00
add    rdi,0x0200
dec    r8
jg     k
dec     si
and    si, 0xFE00
add    rsi,0x0200
add    rdx,0x0200
dec    r9
jg     i

```

در قطعه کد همانطور که در کامنت ها نوشته ایم `rsi`, `rdi`, `rsi`, `rdx` معادل با اندیس هایی روی ماتریس ها هستند و چون برای این روش ما به $mat_1[i][k]$ و $mat_2[j][j]$ نیاز داریم پس به ترتیب دسترسی های ما به حافظه میشود: $[mat_1[rsi * 4 + i][k]]$ و $[mat_2[rdi * 4 + j][j]]$ و برای بروز رسانی $mat_3[rdx * 4 + i][j]$ کافیست به طور مشابه قبل موقع تغییر یافتن شمارنده های مورد نظر آن اندیس، آنرا بروز رسانی کنیم.

و برای محاسبه اصلی کافیست $[mat_1[rsi * 4 + i][k]]$ را به ازای k های مختلف در xmm ذخیره کنیم و به ازای هر j باید $[mat_2[rdi * 4 + j][j]]$ را در xmm ضرب کنیم، آنرا با $[mat_3[rdx * 4 + i][j]]$ جمع کنیم و در نهایت درون $[mat_3[rdx * 4 + i][j]]$ ذخیره کنیم. ولی کدی که در بالا مشاهده میکنید این موضوع را کمی به جلو برد و برای استفاده بهینه‌تر از سیستم pipelining پردازنده در هر حلقه به جای اینکه حافظه این مرحله را بگیرد، در اولین فرصت حافظه مرحله بعدی را میگیرد و طبق آزمایش من این کار افزایش حداقل ۵ درصدی سرعت را به همراه دارد.

توضیح روش برداری سازی

مانند قبل، ما باید داخلی ترین حلقه را، که حلقه j است، موازی سازی کنیم. و برای این کار کافیست به طور همزمان $\text{mat}^3[i][j+7]$ و $\text{mat}^3[i][j+1]$ و ... و $\text{mat}^3[i][j]$ به طور همزمان بروز رسانی شوند.

$$\begin{bmatrix} \text{mat}^3[i][j] \\ \text{mat}^3[i][j+1] \\ \dots \\ \text{mat}^3[i][j+7] \end{bmatrix} = \begin{bmatrix} \text{mat}^3[i][j] \\ \text{mat}^3[i][j+1] \\ \dots \\ \text{mat}^3[i][j+7] \end{bmatrix} + \begin{bmatrix} \text{mat}^1[i][k] \\ \text{mat}^1[i][k] \\ \dots \\ \text{mat}^1[i][k] \end{bmatrix} * \begin{bmatrix} \text{mat}^2[k][j] \\ \text{mat}^2[k][j+1] \\ \dots \\ \text{mat}^2[k][j+7] \end{bmatrix}$$

و درواقع کافیست رابطه ریاضی بالا را به زبان اسembly بیان کنیم که کافیست از دستورات `vmovaps` و `vbroadcastss` و `vmulps` و `vaddps` استفاده کنیم.

برای ساختن ماتریس دوم از سمت راست، `vmulps` برای محاسبه ضرب درایه به درایه، `vaddps` برای `vbroadcastss` محاسبه جمع درایه به درایه و `vmovaps` برای انتقال درایه های بردار ها به صورت یکجا.

توضیح بخش برداری سازی شده

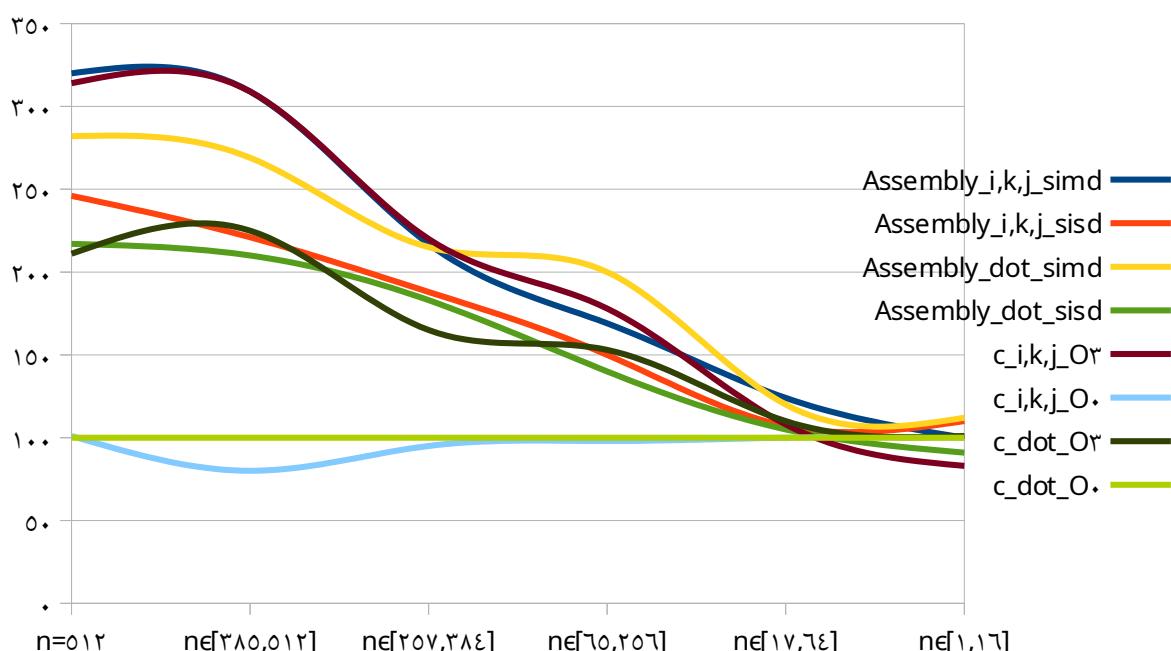
```
; rsi {i , k}
; rdx {i , j}
; rdi {k , j}
xor    rsi,rsi
xor    rdx,rdx
mov    r9, qword[n]
i:
xor    rdi,rdi
mov    r8, qword[n]
k:
vbroadcastss  ymm0, [mat1 + rsi * 4]; load mat1[i][k] into all floats of ymm0
inc    si
mov    rcx, qword[nv]
vmovaps   ymm2, [mat2 + rdi * 4]; load first mat2 cells(mat2[k][0],mat2[k][1], ...,mat2[k][7])
vmovaps   ymm3, [mat3 + rdx * 4]; load first mat3 cells(mat3[i][0],mat3[i][1], ...,mat3[i][7])
j:
vmulps   ymm1, ymm0 , ymm2; ymm2 := mat1[i][k] * mat2[k][j+0 ... j+7]
vmovaps   ymm2, [mat2 + rdi * 4 + 32]; load next set of mat2 cells
vaddps   ymm1, ymm3 ; ymm2 := mat3[i][j+0 ... j+7] + mat1[i][k] * mat2[k][j+0 ... j+7]
vmovaps   ymm3, [mat3 + rdx * 4 + 32]; load next set of mat3 cells
vmovaps   [mat3 + rdx * 4], ymm1 ; rdx = i * 0x0200 + j
add    di, 8
add    dx, 8
loop   j
sub    dx, 8
and    dx, 0xFE00
sub    di, 8
and    di, 0xFE00
add    rdi,0x0200
dec    r8
jg     k
dec    si
and    si, 0xFE00
add    rsi,0x0200
add    rdx,0x0200
dec    r9
jg     i
```

همانطور که میبینید میتوان به سادگی بین دو حالت مقایسه انجام نمود که $mat1[rsi] * 4$ را به $ymm0$ و $mat1[rsi]$ را با استفاده از $vbroadcastss$ درون تمامی درایه های $ymm0$ میگذارد و در پایین تر بازگذاری و ذخیره اطلاعات درون $mat2$ و $mat3$ به جای استفاده از $vmovaps$ از $vmovss$ استفاده میشود تا عملکرد معادل با توضیح ریاضیاتی بالا داشته باشد و به طور موازی ۸ درایه از ماتریس ها را انتقال دهد. در نهایت برای انجام محاسبه به جای $addss$ از $vmulps$ و به جای $vaddps$ از $addss$ استفاده میشود تا به طور موازی محاسبه مورد نیاز را انجام دهد.

نتایج آزمایش

.../matmul/benchmarking master ! v13.2.1 v3.11.6 19:33 source benchmark.sh						
TEST 1	1 ≤ n ≤ 16	16 < n ≤ 64	64 < n ≤ 256	256 < n ≤ 384	384 < n ≤ 512	n=512
assembly(normal simd)	.055371868	.063169281	.392525405	.919592876	1.634013691	2.095706294
assembly(normal sisd)	.050007890	.073645486	.440744851	1.053154834	2.287169158	2.718104760
assembly(dotprod simd)	.049525999	.065627270	.330802993	.926089210	1.887837279	2.375178660
assembly(dotprod sisd)	.060478004	.074395976	.472203715	1.080666148	2.4075677907	3.075622705
c(normal gcc -O3)	.066677767	.072889281	.377536356	.901096419	1.637259448	2.134315422
c(normal gcc -O0)	.055008587	.078342465	.672483972	2.088710385	6.256579234	6.562344226
c(dotprod gcc -O3)	.054391793	.071087812	.436376257	1.202886597	2.249977396	3.174027052
c(dotprod gcc -O0)	.055553079	.078446101	.663811889	1.989223636	5.051765722	6.691491977

جدول درصد بهبود نسبت به حالت پایه (الگوریتم ضرب داخلی زبان C بدون بهینه سازی کامپایلر)



nε[1,16]	nε[17,64]	nε[65,207]	nε[208,384]	nε[385,512]	n=512	Code
1.00	124	179	217	309	220	Assembly_i,k,j_simd
110	107	100	188	221	247	Assembly_i,k,j_sisd
112	120	200	210	269	282	Assembly_dot_simd
91	100	140	183	210	217	Assembly_dot_sisd
83	108	178	220	309	214	c_i,k,j_O2
100	100	98	90	80	101	c_i,k,j_O0
101	110	103	170	220	211	c_dot_O2
100	100	100	100	100	100	c_dot_O0

همانطور که در آزمایش میبینیم بین کد های اسmbلی بهترین عملکرد را حالت ۳ حلقه تو در تو SIMD داشته بعد از آن کد ضرب نقطه ای SIMD است و بعد به طور مشابه ۳ حلقه تو در تو SIMD و در نهایت ضرب نقطه ای SIMD است.

و تقریباً به طور مشابه در بین حالت های مختلف کد C نیز همین موضوع را با قدرت بیشتری میبینیم و این به دلیل تفاوت در دسترسی به حافظه است چون ضرب ماتریسی بسیار وابسته به پهنای باند حافظه است و تغییر کوچک در آن هم میتواند اثر گذار باشد.

اما نکته جالبی که اینجا دیده میشود این است که سریع ترین حالت کد C ما هم به اندازه خیلی خوبی بهینه سازی شده است و تقریباً معادل با سریع ترین نسخه کد اسmbلی ماست.

کانولوشن

توضیح بخش های کلی و تکرار شده کد

در این بخش تنها قسمت های تغییر یافته نسبت به ضرب ماتریسی آمده است.

```
ReadDIM:  
call      asm_read_uint  
mov       rbx, rax  
call      asm_read_uint  
  
cmp       rbx, 1024  
jg        WRONG  
cmp       rax, rbx  
jg        WRONG  
jmp       OkDIM  
  
WRONG:  
mov       rax, wrong_dim_error  
call      asm_write_str  
jmp       ReadDIM  
  
OkDIM:  
mov       qword[n], rbx  
mov       qword[m], rax  
mov       bl, al  
shr       rax, 3  
and       bl, 0x07  
jz        OkMV  
inc       rax  
OkMV:  
mov       qword[mv], rax  
  
mov       rbx, qword[n]  
sub       rbx, qword[m]  
inc       rbx  
mov       qword[r], rbx
```

کانولوشن ما به این صورت کار میکند که ابتدا n و m (ابعاد ماتریس اصلی و ماتریس کرنل) را از ورودی میگیرد و اگر ابعاد ماتریس های داده شده درست نباشد ($0 \leq m \leq n \leq 1024$ برقرار نباشد) خطای مناسبی را چاپ میکند و به مرحله خواندن باز میگردد و در غیر این صورت به OkDIM میرود که به این مفهوم است که n و m را به درستی ورودی گرفته است. سپس برای بخش برداری سازی نیاز است که $\frac{m}{8}$ را نیز محاسبه کنیم زیرا به جای این که به ازای $i = 0, 1, \dots, m-1$ بخواهیم

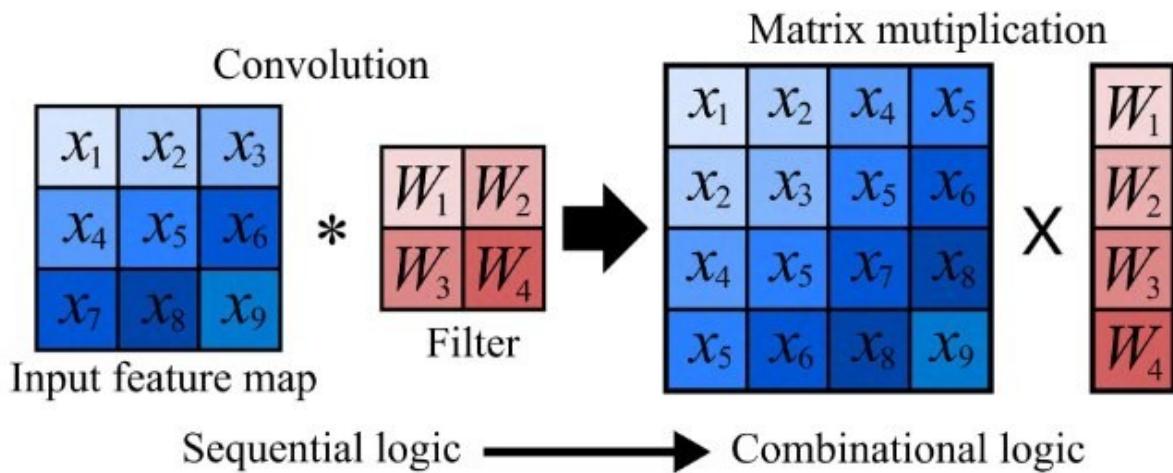
یک تک عملیات انجام دهیم باید به ازای $i = 0, 8, \dots, 8.(\lfloor \frac{m}{8} \rfloor - 1)$ 8 عملیات موازی انجام دهیم که به یک نتیجه منتها

در زمان سریعتر میرسیم. و برای این که این عبارت را محاسبه کنیم میتوانیم m را ۳ بار به راست شیفت دهیم و چک کنیم که اگر آن ۳ بیت حذف شده ناصفر بودند بنابر این باید یک واحد به مقدار rax اضافه کنیم که مقدار rax به عبارت مورد نظر تبدیل شود و در نهایت آنرا درون MV ذخیره میکنیم. و در نهایت برای این که خروجی ماتریسی به ابعاد $n-m+1$ است، به این عدد نیز برای سادگی کار نیاز داریم و آنرا به عنوان ۲ ذخیره میکنیم.

روش اول - استفاده از ضرب خارجی ماتریسی (ضرب عادی ماتریس)

توضیح روش

در این روش ما به شکل زیر کانولوشن را حساب میکنیم.



در واقع محاسبات مورد نیاز به صورت یک به یک به این فرم تبدیل میشود و برای انجام این روش ابتدا ماتریس آبی سمت راست را از روی ماتریس ورودی میسازیم و سپس آنرا به الگوریتم ضرب ماتریسی میدهیم و خروجی آنرا چاپ میکنیم. برای این کار باید از روی ماتریس $n \times n$ داده شده یک ماتریس $(n-m+1)^2 \times m^2$ بسازیم که هر خانه آن درواقع با ۴ اندیس مشخص میشود () به شکل زیر تعریف میشود.

$$\begin{aligned} mat2[(k, l)][0] &= B[k][l] & mat1[(i, j)][(k, l)] &= A[i+k][j+l] \\ \Rightarrow (mat1 \times mat2)[(i, j)][0] &= \sum_{(k, l)} mat1[(i, j)][(k, l)] mat2[(k, l)][0] = \sum_{(k, l)} A[i+k][j+l]B[k][l] \end{aligned}$$

و به این شکل این رابطه دقیقاً همان رابطه کانولوشن را تشکلی میدهد.

برای اینکه بتوانیم به این روش کانولوشن را محاسبه کنیم، باید ابتدا ماتریس $mat1$ را بسازیم و سپس محاسبه اصلی را با استفاده از الگوریتم ضرب ماتریس در بردار حساب کنیم.

```

xor    r8, r8
xor    r12, r12
i: ; r8
    xor    r9, r9
    j: ; r9
        xor    r10, r10
        xor    r13, r13
        k: ; r10
            xor    r11, r11
            l: ; r11
                mov    rax, r12
                shl    rax, 10
                or     rax, r13
                ;; rax = (ai , aj)
                mov    rbx, r8
                add    rbx, r10
                shl    rbx, 5
                or     rbx, r9
                add    rbx, r11
                ;; rbx = (i+k , j+l)
                movss xmm0, [mat1 + rbx * 4]
                movss [Amat + rax * 4], xmm0

                inc    r13
                inc    r11
                cmp    r11, qword[m]
                jl     l
                inc    r10
                cmp    r10, qword[m]
                jl     k
                inc    r12
                inc    r9
                cmp    r9, qword[r]
                jl     j
                inc    r8
                cmp    r8, qword[r]
                jl     i

```

این قطعه کد مسئولیت ساختن ماتریس $mat1$ توضیح داده شده در بالاتر و در کد ما $Amat$ را بر عهده دارد و به طور مشابه بالا کار میکند. با استفاده از ۴ حلقه تو با تو با اندیس های i, j, k, l ، سعی در انجام مرحله به مرحله $Amat[(i, j)][(k, l)] = mat1[i+k][j+l]$ داریم، بنابر این نیاز به اندیس های $(i, j), (k, l)$ و $(i+k, j+l)$ داریم که آنها را به ترتیب در رजیستر های rax و rbx محاسبه و بروزرسانی میکنیم و در نهایت مقدار $A[i+k][j+l]$ را درون $Amat[(i, j)][(k, l)]$ انتقال دهیم. و بعد از اینکه این ماتریس ساخته شد باید آنرا در نسخه برداری B که آنرا در کد $Bmat$ مینامیم به شکل زیر ذخیره کنیم.

```

    mov      rcx, qword[m]
    xor      rsi, rsi
    MAT2_i: ; rbx
        mov      rbx, rcx
        mov      rcx, qword[m]
    MAT2_j: ; rcx
        asm_read_float
        movss   [Bmat + rsi * 4], xmm0
        inc     rsi
        loop   MAT2_j
    mov      rcx, rbx
    loop   MAT2_i

```

این قطعه کد تقریباً مشابه ورودی گرفتن عادی ماتریس است با این تفاوت که `Bmat` را به صورت یک بردار ذخیره میکنیم بنابر این برای رفتن به خانه بعدی `Bmat` و ورودی گرفتن آن کافی است هر بار صرفاً اندیس را یکی زیاد کنیم.

و در نهایت بعد از اینکه ماتریس `Amat` و بردار `Bmat` تشکیل شد، باید آنها را در یکدیگر ضرب کنیم. برای ضرب نیز از کدی مشابه بخش قبل استفاده میکنیم.

```

xor    rsi, rsi
xor    rax, rax
mov   rbx, qword[r2]
matmul_i:
    xor    rdi, rdi
    mov   rcx, qword[m2]
    movss xmm0, [zero_v]
matmul_k:
    movss xmm1, [Amat + rsi]
    mulss xmm1, [Bmat + rdi]
    addss xmm0, xmm1
    add   rsi, 4 ; rsi : (i , k)
    add   rdi, 4 ; rdi : (k)
    loop matmul_k
    movss [Cmat + rax * 4], xmm0
    sub   rsi, 4
    and   si, 0xF000
    add   rsi, 0x1000
    inc   rax ; rax : (i)
    dec   rbx
    jnz   matmul_i

```

و با توجه به اینکه ماتریس دوم یک بردار است، بنابر این این الگوریتم نیاز به حلقه زدن برای \sum_k ندارد چون \sum_k همیشه \circ است. بنابر این ساختار این کد به این شکل است که صرفاً نیاز به یک حلقه تو در تو ساده با اندیس های i, k داریم که $(Amat \times Bmat)[i] = \sum_k Amat[i][k]B[k]$ نیاز داریم که آنها را به ترتیب در رजیستر های `rsi` و `rdi` و `rax` نگهداری و بروزرسانی میکنیم و از آنها برای دسترسی به ماتریس ها استفاده میکنیم. تنها نکته ای که باقی میماند این است که `rsi` و `rdi` اندیس بایت مورد نظر را نگه میدارند بنا بر این برای جلو رفتن $+ 4$ می شوند و در هنگام دسترسی میتوان مستقیماً `Amat[rsi][rdi]` و یا `Bmat[rdi]` را استفاده کرد.

توضیح روش برداری سازی

برای برداری سازی این راه کافیست قسمت ضرب ماتریسی را برداری سازی نمود که برای این کار کافیست موقعی که میخواهیم به ازای k های متفاوت $(Amat \times Bmat)[i] = \sum_k Amat[i][k]B[k]$ این عبارت را حساب کنیم، به طور موازی چند

k متوالی را به صورت موازی حساب کنیم و برای این کار کافیست k ها را به ۸ دسته با توجه به باقیمانده شان به ۸ تقسیم

کنیم و در هر مرحله یک محاسبه از هر دسته را همزمان انجام دهیم (بنابر این ۸ محاسبه را به صورت همزمان انجام خواهیم داد) و در نهایت وقتی ۸ دسته محاسبه شدند، در نهایت نتیجه ۸ دسته را با یکدیگر جمع کنیم و به عنوان مقدار $(Amat \times Bmat)[i]$ ذخیره کنیم.

توضیح بخش برداری سازی شده

```

xor    rsi, rsi
xor    rax, rax
mov    rbx, qword[r2]
matmul_i:
    xor    rdi, rdi
    mov    rcx, qword[m2v]
    vmovaps ymm0, [zero_v]
    matmul_k:
        vmovaps ymm1, [Amat + rsi]
        vmulps ymm1, [Bmat + rdi]
        vaddps ymm0, ymm1
        add    rsi, 32 ; rsi : (i , k)
        add    rdi, 32 ; rdi : (k)
        loop   matmul_k
        vdpps ymm0, ymm0, [one_v], 0xF1
        vperm2f128 ymm1, ymm0, ymm0, 0x01
        addss  xmm0, xmm1
        movss  [Cmat + rax * 4], xmm0
        sub    rsi, 4
        and    si, 0xF000
        add    rsi, 0x1000
        inc    rax ; rax : (i)
        dec    rbx
        jnz    matmul_i

```

همانطور که در قطعه کد بالا میتوانید ببینید، ما در حلقه داخلی درواقع در حال استفاده از رجیستر های ymm به عنوان ۸ متغیر float هستیم و به جای دستور movss از mulss از vmulps و به جای دستور addss از vaddps استفاده شده که این دستورها به جای اینکه به صورت تکی با رجیستر برخورد کنند، رجیستر را به عنوان ۸ مقدار float به صورت موازی در نظر میگیرد بنا بر این وقتی حلقه داخلی به اتمام میرسد در نهایت مقدار ۸ دسته بندی که در بالا توضیح داده شده است درون ymm^۰ قرار میگیرد بنابر این در نهایت باید ۸ مقدار ذخیره شده درون آنرا با یکدیگر جمع نمود و درون حافظه $[Cmat[rax * 4]]$ ریخت. برای این کار از دستور vdpps یا ضرب نقطه‌ای استفاده میکنیم و ایده ما این است که اگر ymm^۰ را با بردار $[1, 1, \dots, 1]$ ضرب نقطه‌ای کنیم در نهایت حاصل مقدار جمع اعداد ذخیره شده درون ymm می‌شود منتها دستور vdpps با ymm ها به صورت ۲ بخش ۱۲۸ بیتی مستقل برخورد میکند بنابر این برای استخراج نتیجه باید بعد از انجام vdpps آنرا وارونه کنیم و با خودش جمع کنیم و این بخش را دستور vperm2f128 addss انجام میدهند.

روش دوم - استفاده از ضرب داخلی ماتریسی

توضیح روش

در این روش کافیست ما ماتریس کرنل را روی ماتریس اصلی تکان دهیم و در هر مرحله مقدار خانه (j, i) ماتریس خروجی را برابر با ضرب نقطه‌ای ماتریس اصلی در ماتریس کرنل قرار دهیم. بنابر این در این روش نیازمند یکتابع محاسبه کننده ضرب

نقشه‌ای شیفت خورده دو ماتریس هستیم که مقدار حرکت را به عنوان ورودی بگیرد و در نهایت ضرب نقطه‌ای را به عنوان خروجی تولید کند و ما در بیرون از تابع، مرحله به مرحله ماتریس خروجی را محاسبه کنیم.

```
%macro mat_dot 0 ; takes rax as the index to top left location of mat1 matrix to be dotproducted with mat2 matrix and puts result in xmm0
    vmovss    xmm0 , [zero_v]
    xor      rsi, rsi ; rsi : (k , l) = k * 0x0400 + l
; rax : (i , j) = i * 0x0400 + j => rsi + rax = 0x0400 * (i+k) + j+l : (i+k , j+l)
    mov      r10, qword[m]
    k:
        mov      rcx, qword[m]
        lv:
            movss    xmm1 , [mat2 + rsi]; mat2[k][l] ; rsi = k * 0x0400 + l
            mulss    xmm1 , [mat1 + rsi + rax]; mat1[i+k][j+l]
            addss    xmm0 , xmm1
            add      si , 4
            loop
            lv
            sub      si , 4
            and      si, 0xF000
            add      rsi,0x1000
            dec      r10
            jnz      k
%endmacro
```

همانطور که در کامنت‌های این قطعه کد مشاهده می‌کنید، این قطعه کد مسئولیت محاسبه کردن ضرب نقطه‌ای ماتریس کرنل را در زیر ماتریسی از ماتریس اصلی، که نقطه بالا راست آن با اندیس rax تعیین می‌شود، را دارد.

این کد در واقع مسئول محاسبه عبارت ریاضی $\sum_{(k,l)} mat1[i+k][j+l]mat2[k][l]$ است و اندیس $\{i, j\}$ را به عنوان رجیستر rax ورودی می‌گیرد و اندیس $\{l, k\}$ را در رجیستر rsi نگهداری کرده و به ازای l, k ‌های متفاوت بروز نگه میدارد. اما همانطور که در این رابطه ریاضی می‌بینید به اندیس $\{i+k, j+l\}$ نیز نیاز داریم که آنرا به سادگی با استفاده از جمع rsi و rax محاسبه می‌کنیم چرا که $rsi = k * 1024 + l$ و $rax = i * 1024 + j$ در این صورت $rsi + rax = (i+k) * 1024 + (j+l)$ می‌باشد. در این صورت صرفاً کافیست در هر مرحله از حلقه مان، $xmm0$ را با معادل با اندیس $rsi + rax$ جمع کنیم. و در نهایت به این صورت ضرب داخلی را حساب می‌کنیم.

```
xor      rax, rax ; rax : (i , j)
mov      r8, qword[r]
i:
;
    mov      r9, qword[r]
    j:
        mat_dot
        movss    [mat3 + rax] , xmm0 ; mat3[i][j]
        add      ax, 4
        dec      r9
        jnz      j
        sub      ax, 4
        and      ax, 0xF000
        add      rax,0x1000
        dec      r8
        jnz      i
```

و در نهایت با استفاده از قطعه کد بالا، به ازای j, i ‌های متفاوت rax معادل را حساب کرده و mat_dot را صدا می‌زنیم که برای ما ضرب نقطه‌ای مورد نظر را حساب می‌کند و در نهایت حاصل را درون $mat3[rsi]$ میریزیم.

توضیح روش برداری سازی

برای برداری سازی این الگوریتم کافیست بخش mat_dot را برداری سازی کنیم و برای برداری سازی آن کافیست چند اتفاقاً را به صورت همزمان محاسبه کنیم. برای این کار به طور مشابه قبل کار میکنیم، اما با توجه به باقیمانده به ۸ به دسته تقسیم میکنیم و تلاش میکنیم در هر مرحله یکی از اندیس‌های هر دسته را به صورت موازی محاسبه کنیم. و در نهایت وقتی همه ۸ دسته محاسبه شدند کافیست نتیجه نهایی دسته‌ها را با یکدیگر جمع کنیم و xmm۰ را به عنوان خروجی آماده کنیم.

توضیح بخش برداری سازی شده

```
%macro matdot 0 ; takes rax as the index to top left location of mat1 matrix to be dotproducted with mat2 matrix and puts result in xmm0
    vmovaps    ymm0 , [zero_v]
    xor        rsi, rsi
    mov        r10, qword[m]
    k:
    mov        rcx, qword[mv]
    lv:
    |   vmovaps    ymm1 , [mat2 + rsi]; mat2[k][l] ; rsi = k * 0x0400 + l
    |   vmulps    ymm1 , [mat1 + rsi + rax]; mat1[i+k][j+l] ;
    |   vaddps    ymm0 , ymm1
    |   add         si , 32
    |   loop        lv
    |
    |   sub         si, 32
    |   and         si, 0xF000
    |   add         rsi,0x1000
    |
    |   dec         r10
    jnz        k
    vdpps      ymm0 , ymm0 , [one_v] , 0xF1
    vperm2f128 ymm1 , ymm0 ; ymm0 , 0x01
    addss      xmm0 , ymm1
%endmacro
```

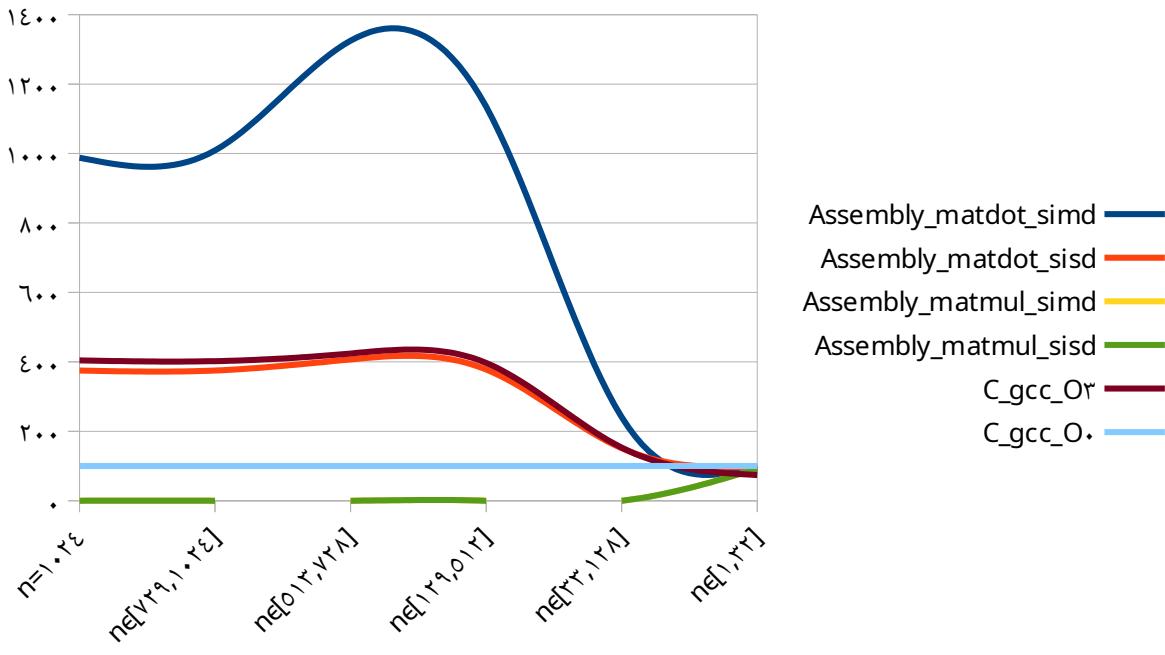
همانطور که در قطعه کد بالا مشاهده میکنید، این کد نیز دقیقاً از ایده‌های قسمت قبل کمک گرفته است و از ymm۰ به عنوان حاصل ۸ دسته بندی استفاده میکند و در نهایت با ایده vperm2f128 و vdpps نتیجه مورد نظر ما که جمع ۸ عدد موازی ذخیره شده درون ymm۰ است را حساب میکند.

نتایج آزمایش

آزمون سرعت

(.venv) ➜ .../convolution/benchmarking ➜ master ✘!?						
TEST 1	TEST 2	TEST 3	TEST 4	TEST 5	TEST 6	TEST 7
Code	1 ≤ n ≤ 32	32 < n ≤ 128	128 < n ≤ 512	512 < n ≤ 768	768 < n ≤ 1024	n=1024
assembly(normal)	.056981347	.099955336	1.405671367	16.008316992	94.199046245	124.040142497
assembly(normal sisd)	.057262346	.158751987	4.265461249	52.514102020	253.153018304	327.121484638
assembly(matmul)	.060958421	DNF	DNF	DNF	DNF	DNF
assembly(matmul sisd)	.060994141	DNF	DNF	DNF	DNF	DNF
c(normal gcc -O3)	.074111745	.156572607	4.017220905	50.553952136	236.263498924	303.696541717
c(normal gcc -O0)	.055078115	.239197873	15.911035506	212.881640497	949.383193588	1225.566595640

جدول درصد بهبود نسبت به حالت پایه (زبان C بدون بهینه سازی کامپایلر)



$n \in [1, 32]$	$n \in [32, 128]$	$n \in [128, 512]$	$n \in [512, 728]$	$n \in [728, 1024]$	$n = 1024$	Code
98	241	1130	1320	1009	988	Assembly_matdot_simd
97	101	378	407	370	370	Assembly_matdot_sisd
92	•	•	•	•	•	Assembly_matmul_simd
92	•	•	•	•	•	Assembly_matmul_sisd
74	103	397	424	402	404	C_gcc_O2
100	100	100	100	100	100	C_gcc_O0

همانطور که میبینید، ضرب نقطه ای با دستور های برداری با اختلاف سریع ترین گزینه برای انجام کاتولوشن است و ضرب خارجی به علت گرفتن حافظه بسیار بالا ($O(n^4)$) توانایی پردازش n های بالای ۳۲ را ندارد و در همان کلاس کمتر مساوی ۳۲ نیز از ضرب نقطه ای کند تر است بنابر این ضرب داخلی با دستور های **simd** مناسب ترین گزینه برای انجام پردازش تصویر هستند.

پردازش تصویر

برای انجام پردازش تصویر از ۳ تصویر و ۵ کرنل متفاوت استفاده میکنیم.

کرنل های ما

$$\text{Box Blur} \begin{bmatrix} 0.111, 0.111, 0.111 \\ 0.111, 0.111, 0.111 \\ 0.111, 0.111, 0.111 \end{bmatrix}$$

Gaussian Blur

$$\begin{bmatrix} 0.003765, 0.015019, 0.023792, 0.015019, 0.003765 \\ 0.015019, 0.059912, 0.094907, 0.059912, 0.015019 \\ 0.023792, 0.094907, 0.150342, 0.094907, 0.023792 \\ 0.015019, 0.059912, 0.094907, 0.059912, 0.015019 \\ 0.003765, 0.015019, 0.023792, 0.015019, 0.003765 \end{bmatrix}$$

Unsharp

$$\begin{bmatrix} -0.00391, -0.01563, -0.02344, -0.01563, -0.00391 \\ -0.01563, -0.06250, -0.09375, -0.06250, -0.01563 \\ -0.02344, -0.09375, 1.85980, -0.09375, -0.02344 \\ -0.01563, -0.06250, -0.09375, -0.06250, -0.01563 \\ -0.00391, -0.01563, -0.02344, -0.01563, -0.00391 \end{bmatrix}$$

Sobel Edge(X)

$$\begin{bmatrix} -1, 0, 1 \\ -2, 0, 2 \\ -1, 0, 1 \end{bmatrix}$$

Sobel Edge(Y)

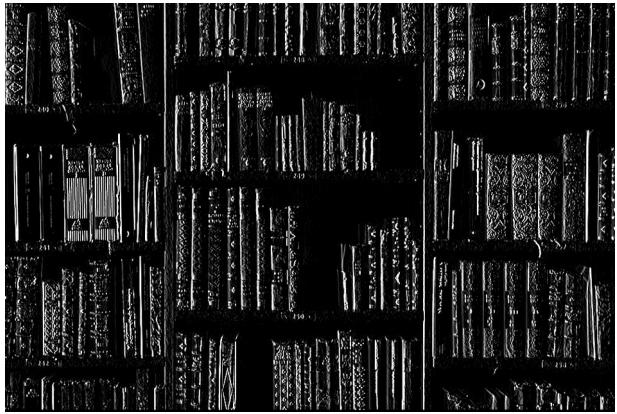
$$\begin{bmatrix} -1, -2, -1 \\ 0, 0, 0 \\ 1, 2, 1 \end{bmatrix}$$

خروجی پردازش

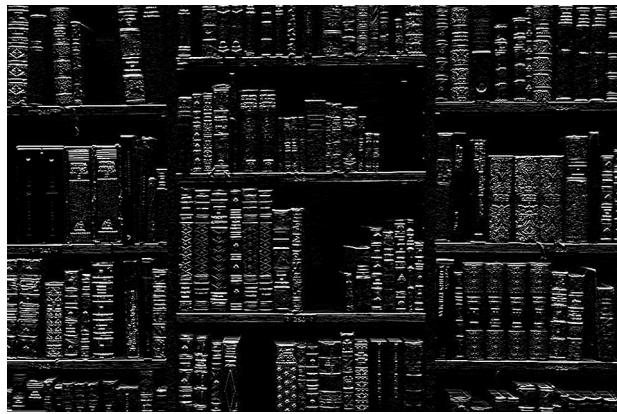


BoxBlur

GaussianBlur



Sobel Edge(X)

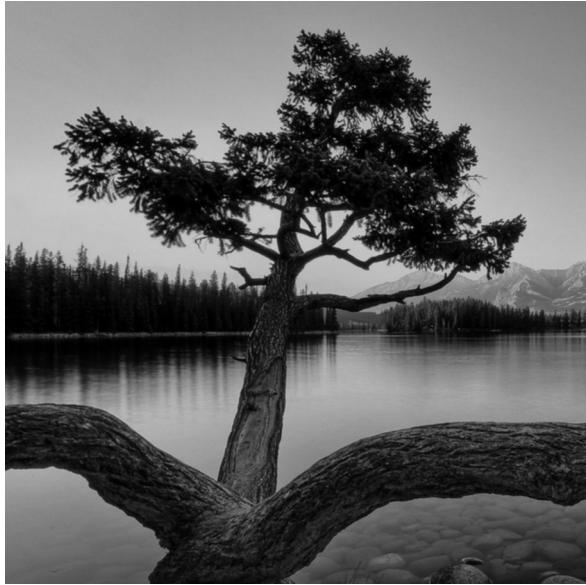


Sobel Edge(Y)

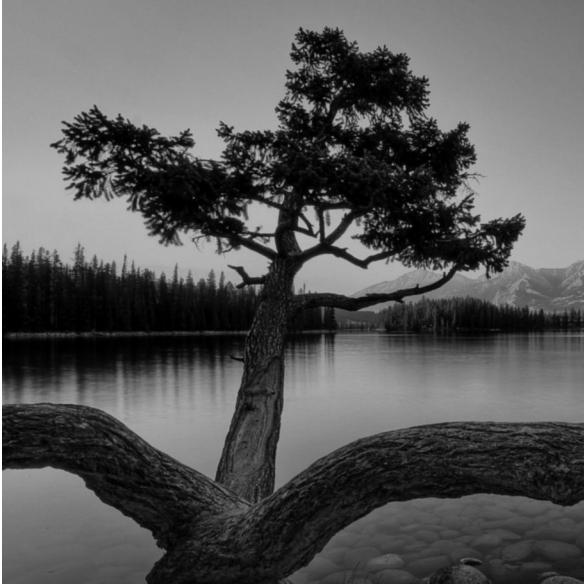


Unsharp

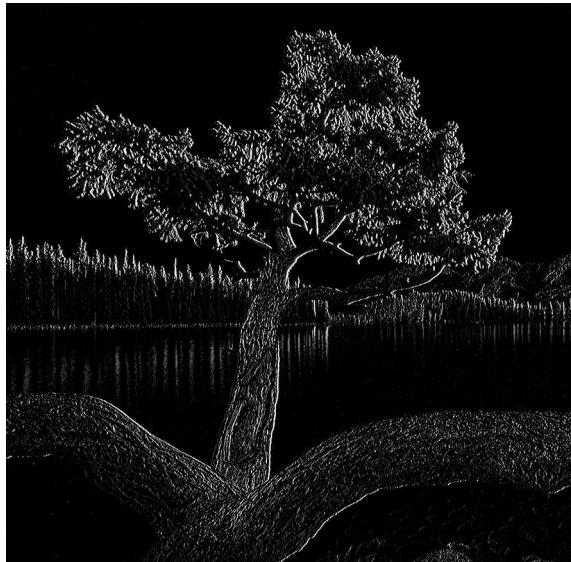




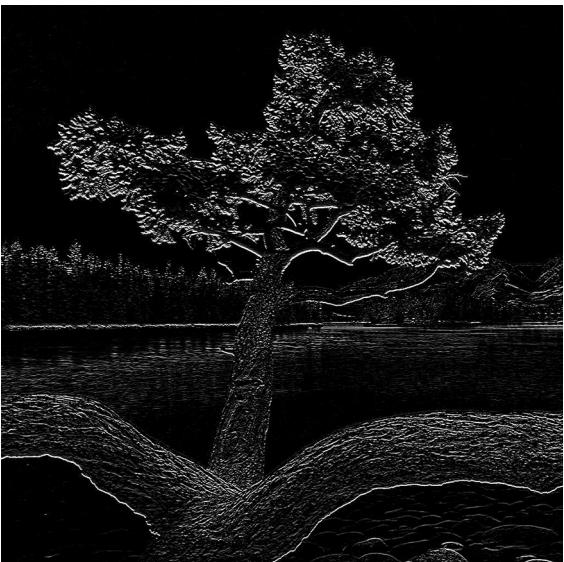
BoxBlur



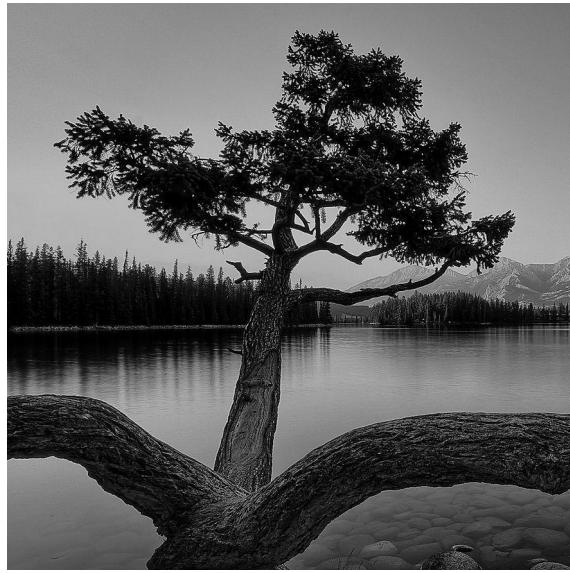
GaussianBlur



Sobel Edge(X)



Sobel Edge(Y)



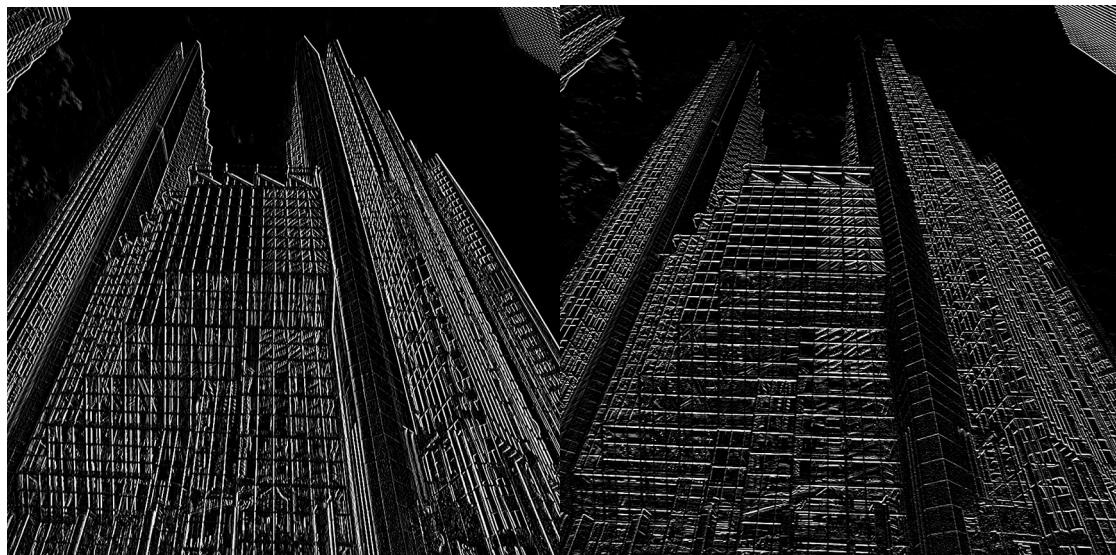
Unsharp





BoxBlur

GaussianBlur



Sobel Edge(X)

Sobel Edge(Y)



Unsharp

نتیجه‌گیری

با انجام این پروژه به این نتیجه رسیدیم که با برنامه نویسی اسembلی میتوانیم در مسائل خاصی (مانند کانولوشن) تا ۵ برابر سریع‌تر از بهینه سازی کامپایلر عمل کنیم اما گاهی مسائل خاص پرکاربرد نیز با استفاده از بهینه سازی کامپایلر به حالت بهینه خود میرسند (مانند ضرب ماتریسی) و نیازی به برنامه نویسی اسembلی نیست اما حتی اگر میدانیم که چگونه کدهای سطح بالای مؤثر تری بسازیم به طور مثال اگر اندازه ماتریس یا آرایه خود را توان دو قرار دهیم میتوانیم به سادگی بین سطر های مختلف ماتریس حرکت کنیم بنابر این کامپایلر نیز با استفاده از این نکته میتواند به صورت بهینه روی آرایه ها حرکت کند که باعث تسريع برنامه های ما میشود.