به نام خدا

درس طراحی سیستم های دیجیتال

طراحی واحد ضرب کننده ی ماتریس ها

استاد درس: دکتر بهاروند

نام اعضای گروه:

سارا آذرنوش، علی جوانمرد، رویا قوامی عادل، داریوش امیری

دانشگاه صنعتی شریف

تابستان 1400


بسم الله الرحمن الرحيم

فهرست مطالب

چکیده

ماتریس‌ها کاربردهای زیادی در محاسبات شاخه‌های مختلف دانش دارند برای مثال در تمامی شاخه‌های فیزیک، شامل مکانیک کلاسیک، نورشناسی، الکترومغناطیس، مکانیک کوانتوم و الکترودینامیک کوانتومی، ماتریس برای مطالعه‌ی پدیده‌های فیزیکی به کار می‌رود. ضرب معمولی ماتریس‌ها رایج‌ترین نوع ضرب در ماتریس‌ها است. این نوع ضرب تنها زمانی تعریف می‌شود که تعداد ستون‌های ماتریس اول با تعداد سطرهای ماتریس دوم برابر باشد. لازم به ذکر است که محاسبات ماتریسی، در سیستمهای پردازش تصویر، در سیستمهای مخابراتی MIMO، در سیستم های مخابراتی که از روش OFDM برای ارسال اطلاعات استفاده می کنند و واحد ضرب ماتریس در برنامه های پردازش سیگنال دیجیتال مانند تصویربرداری دیجیتال، پردازش سیگنال، گرافیک رایانه ای و چند رسانه ای استفاده می شود. بنابراین بسیار مهم است که مساحت کمتری اشغال کند، سریع کار کند و انرژی کمتری مصرف کند. در این پروژه می خواهیم واحد ضرب کننده با کمک دو ماتریس با سایز دلخواه و نمایش نقطه شناور طراحی کنیم.

مقدمه

همانطور که در مقدمه ی کتاب محاسبات ماتریسی[1] آمده است، مطالعه ی محاسبات مربوط به ماتریس، با مطالعه مسئله ضرب ماتریس-ماتریس آغاز می شود. اگرچه این مسئله از نقطه نظر ریاضی ساده است، ولی در علوم محاسبات و کامپیوتر بسیار حائز اهمیت می باشد.

کاربردهای ضرب ماتریس در بسیاری از زمینه‌های مختلف همچون علم محاسبه و بازشناخت الگو و حتی زمینه‌های به ظاهر بی‌ربط مانند شمردن تعداد گشت‌ها در یک گراف دیده می‌شود. الگوریتم‌های بسیاری برای این‌کار روی سیستم‌های رایانش موازی طراحی شده‌است که در آن چند هسته به صورت همزمان و موازی عملیات را انجام می‌دهند.

به طور کل، اگر ماتریس ساختاری خاص داشته باشد ، معمولاً امکان بهره برداری از آن وجود دارد. مثلا یک ماتریس متقارن می تواند در نیمی از فضای حافظه و به عنوان یک ماتریس معمولی ذخیره شود؛ یا ضرب یک بردار در ماتریسی که تعداد زیادی از درایه هایش صفر است ممکن است زمان اجرای بسیار کمتری نسبت به ضرب همان بردار در ماتریسی کامل داشته باشد.

۱-بررسی الگوریتم های موجود

در ابتدا باید توجه کرد که محاسبات ماتریس بر اساس سلسله مراتبی از عملیات جبری خطی ساخته شده است.:

- ضرب نقطه ای، شامل عملیات ضرب و جمع اسکالر است.
- ضرب ماتریس-بردار تشکیل شده از عملیات های ضرب نقطه ای می باشد.
- ضرب ماتریس-ماتریس شامل مجموعه ای از حاصل ضرب های ماتریس-بردار است.

همه این عملیات را می توان در فرم الگوریتمی و یا به زبان جبر خطی توصیف کرد. از طرفی، در اینجا برای ارزیابی مرتبه ی زمانی می توانیم Big-Oh استفاده کنیم. ضرب نقطه ای از مرتبه زمانی(O(n ، ضرب ماتریس-بردار از مرتبه زمانی (O(n ^2 و ضرب ماتریس-ماتریس(O(n ^3 می باشد. در نتیجه، برای ایجاد الگوریتمی که شامل ترکیبی از این عملیات ها باشد، تمرکز ما باید بر روی کاهش بیشترین مرتبه ی زمانی یعنی همان (O(n ^3 باشد.

حال با گفته شدن مطالب بالا؛ خوب است نکات ظریفی جلب کنیم. مسئله ی ضرب دو ماتریس 2*2 زیر را در نظر بگیرید.

روش اول نمایش: در فرمولاسیون ضرب نقطه ای، هر درایه را میتوان به صورت ضرب نقطه ای درایه های مربوط به آن به دست آورد:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}=\begin{bmatrix} 1\cdot 5+2\cdot 7 & 1\cdot 6+2\cdot 8 \\ 3\cdot 5+4\cdot 7 & 3\cdot 6+4\cdot 8 \end{bmatrix}.$$

نکته ی جالب توجه این است که همین ضرب را میتوان به دو شکل دیگر نیز نمایش داد.

روش دوم نمایش: یکی از این ورژن های نمایش دادن مسئله ی ضرب صفحه ی قبل، نمایش saxpy است. در این نمایش، هر ستون در حاصل ضرب نهایی، درواقع ترکیبی خطی از ستون های ماتریس دوم است:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}=\left[\, 5\begin{bmatrix} 1 \\ 3 \end{bmatrix}+7\begin{bmatrix} 2 \\ 4 \end{bmatrix},\ \ 6\begin{bmatrix} 1 \\ 3 \end{bmatrix}+8\begin{bmatrix} 2 \\ 4 \end{bmatrix}\right].$$

روش سوم نمایش: نمایش که outer product نام دارد، می توان نشان داد که ضرب نهایی، برابر است با ضرب خارجی عبارات داده شده:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}=\begin{bmatrix} 1 \\ 3 \end{bmatrix}\begin{bmatrix} 5 & 6 \end{bmatrix}+\begin{bmatrix} 2 \\ 4 \end{bmatrix}\begin{bmatrix} 7 & 8 \end{bmatrix}.$$

گرچه سه روش بالا از لحاظ ریاضی کاملا با هم یکی هستند، ممکن است در زمان پیاده سازی performance کاملا متفاوتی نسبت به یکدیگر نشان دهند که این موضوع به علت تاثیر memory traffic می باشد که در اینجا به توضیح بیشتر این موضوع نمی پردازیم.

اکنون، پیش از اینکه به توضیح الگوریتم های موجود بپردازیم، باید اشاره کنیم که تحقیقاتی که بر روی محاسبات ماتریس در بسیاری از زمینه های کاربردی انجام میشود، بستگی به توسعه الگوریتم های موازی ای که scale می شوند دارند. چنین الگوریتم هایی این ویژگی را دارند که با افزایش سایز مسئله، تعداد پردازنده های درگیر نیز بیشتر می شوند.

اگرچه زبان های برنامه نویسی جدید و system tool های مربوط، پروسه ی پیاده سازی موازی محاسبات ماتریسی را راحت تر کرده اند؛ همچنان توانایی "فکر کردن به طور موازی" بسیار مهم است. چنین موضوعی مستلزم این است که درک درستی از load balancing، communication overhead و processor synchronization داشته باشیم.

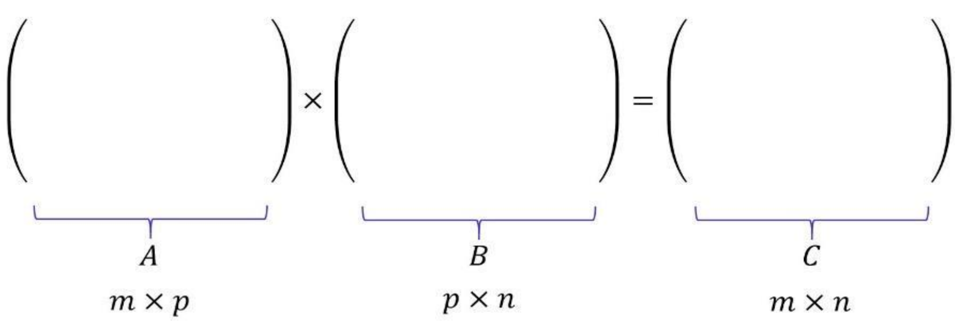
در الگوریتم های زیر، توجه داشته باشید که تمرکز خود را بر روی به روزرسانی مرحله به مرحله ی خروجی میگذاریم یعنی:

$$C = C + AB, \qquad C \in \mathbb{R}^{m\times n},\ A \in \mathbb{R}^{m\times r},\ B \in \mathbb{R}^{r\times n}.$$

در واقع به روز رسانی C = C + AB در نظر گرفته شده است نه C = AB چرا که درعمل، این وضعیت معمول تر است.

الگوریتم 1 – ijk Matrix Multiplication

با داشتن ماتریس های A و B و C که در بالا تعریف شده اند، ساده ترین روش، این است که در هر مرحله حاصل C + AB روی C، بازنویسی یا overwrite شود. شبه کد مربوطه در ادامه آمده است.



```
for i = 1:m
    for j = 1:n
        for k = 1:r
            C(i,j) = C(i,j) + A(i,k)·B(k,j)
        end
    end
end
```

کد c مربوط به این الگوریتم:

```
void multiply(int A[][N], int B[][N], int C[][N])

{

        for (int i = 0; i < N; i++)

        {

                for (int j = 0; j < N; j++)

                {

                        C[i][j] = 0;

                        for (int k = 0; k < N; k++)

                        {

                                C[i][j] += A[i][k]*B[k][j];

                        }

                })

        }

}
```

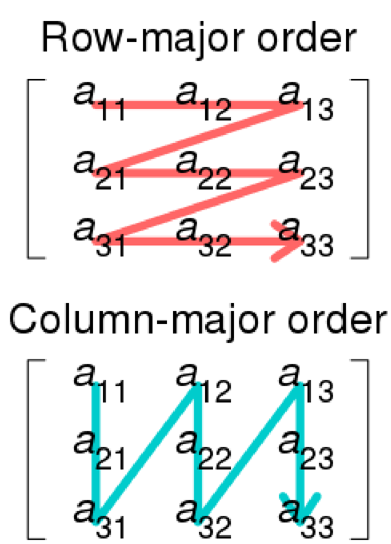
که عملگر جمع و ضرب، عملگر های معمولی جمع و ضرب نیستند بلکه با توجه به IEEE754 طراحی شده اند. لازم به ذکر است که در این الگوریتم، نیاز به حافظه ی اضافی نیست. اردر حافظه mn خواهد بود و پیچیدگی زمانی این محاسبه O(nmp) است.

در این روش، AB را به عنوان مجموعه ای از ضرب های نقطه در نظر می گیریم که یکبار به ترتیب از چپ به راست، بالا به پایین محاسبه می شود.

بررسی حافظه ی نهان در این الگوریتم:

در الگوریتم بالا ترتیب حلقه‌ها را می‌توانیم جابه‌جا کنیم. اگر چه این جابه‌جایی در مدت زمان اجرای الگوریتم تأثیری نخواهد داشت اما این ترتیب در بحث‌های مربوط به دسترسی حافظه (access pattern)و مسائل مربوط به استفاده از حافظه نهان پردازنده مهم است. مثلاً اینکه ماتریس‌ها به ترتیب سطری یا ستونی (یا ترکیبی از این دو) ذخیره شوند در زمان حافظهٔ نهان پردازنده تأثیر گذار خواهد بود.

حتی اگر حالت بهینه را در نظر بگیریم که حافظهٔ کش شرکت پذیر کامل با M سطر حافظهٔ b بیتی باشد و ماتریس‌های A و B به صورت سطری ذخیره شده‌باشند، این الگوریتم بهینه نخواهد بود. هنگامی که n>$\frac{M}{b}$ از آنجایی که ماتریس‌ها به صورت سطری ذخیره شده‌اند هر پیمایش حلقهٔ داخلی در الگوریتم (یک پیمایش روی سطر ماتریس اول و ستون ماتریس دوم) یک خطای کش به هنگام دسترسی به خانه‌ی‌های ماتریس دوم به همراه خواهد داشت؛ و این به این معناست که الگوریتم در بدترین حالت حاوی O(n^3) خطای کش خواهد بود. امروزه -یعنی از سال ۲۰۱۰- خطای کش‌ها به نسبت اعمال پردازنده تأثیر بیشتری روی زمان اجرا می‌گذارند بنابراین بهتر است با روشی این خطای کش‌ها را کاهش دهیم.



الگوریتم 2- Dot Product Matrix Multiplication

با داده شدن ماتریس های A و B و C، به روزرسانی عبارت C + AB را میتوان به فرم زیر نوشت:

$$
\begin{aligned}
&\textbf{for } i = 1{:}m \\
&\quad \textbf{for } j = 1{:}n \\
&\qquad C(i,j) = C(i,j) + A(i,:)\cdot B(:,j) \\
&\quad \textbf{end} \\
&\textbf{end}
\end{aligned}
$$

در بالا مشاهده می کنیم که دو حلقه داخلی، درواقع نوع الگوریتم 1 عه یک gaxpy (که generalize شده ی فرم نمایش ضرب ماتریس saxpy که بالاتر توضیح داده شده بود) جابجا شده را در عمل نشان می دهد.

الگوریتم 3- Saxpy Matrix Multiplication

با داده شدن ماتریس های A و B و C، به روزرسانی عبارت C + AB را میتوان به فرم زیر نوشت:

$$
\begin{aligned}
&\textbf{for } j = 1{:}n \\
&\quad \textbf{for } k = 1{:}r \\
&\qquad C(:,j) = C(:,j) + A(:,k)\cdot B(k,j) \\
&\quad \textbf{end} \\
&\textbf{end}
\end{aligned}
$$

توجه داشته باشید که پیش از توضیح الگوریتم ها، روش دوم نمایش ضرب دو ماتریس همان نمایش saxpy بود و در اینجا کد مربوط به ضرب با ان نمایش پیاده سازی شده است.

الگوریتم 4- Outer Product Matrix Multiplication

با داده شدن ماتریس های A و B و C، به روزرسانی عبارت C + AB را میتوان به فرم زیر نوشت:

$$
\begin{aligned}
&\textbf{for } k = 1{:}r \\
&\quad C = C + A(:,k)\cdot B(k,:) \\
&\textbf{end}
\end{aligned}
$$

همانطور که مشاهده می شود، ضرب توضیح داده شده در این الگوریتم مربوط به روش سوم نمایش ضرب دو ماتریس است که همانطور که آنجا هم گفتیم، حاصل، به صورت جمع ضرب های خارجی تر محاسبه می شود.

در ادامه ، الگوریتم های مطرح میشود که رویکردی کاملا متفاوت تر نسبت به الگوریتم های گفته شده برای حل مسئله ی ضرب دو ماتریس دارند

الگوریتم 5- divide and conquer

ابتدا به هریک از ماتریس های ورودی،به دید یک ماتریس ۲ × ۲ نگاه میکنیم بدین صورت که اگر ماتریس ما $n \times n$ باشد، آن را ماتریسی ۲ × ۲ ای درنظر میگیریم که درایه های آن ماتریس ها (یا درواقع بلوک هایی) به ابعاد n/2 × n/2 باشد.

$$B = \begin{bmatrix} e & f \\ g & h \end{bmatrix} \qquad A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

بنابراین ماتریس C = AB به صورت زیر نوشته می شود:

$$C = \begin{bmatrix} \mathrm{a} \times \mathrm{e} + \mathrm{b} \times \mathrm{g} & \mathrm{a} \times \mathrm{f} + \mathrm{b} \times \mathrm{h} \\ \mathrm{c} \times \mathrm{e} + \mathrm{d} \times \mathrm{g} & \mathrm{c} \times \mathrm{f} + \mathrm{d} \times \mathrm{h} \end{bmatrix}$$

همانطور که مشاهده می شود، مراحل زیر به ترتیب باید طی شود:

1) A و B را به 4 زیر ماتریس تقسیم میکنیم -> divide

2) 8 ضرب مشخص شده را محاسبه میکنیم -> conquer

3) با استفاده از عبارات ضرب بالا، مقادیر C11 , C12, C21, C22 را حساب میکنیم

بنابراین order زمانی این الگوریتم را می توان به شکل زیر محاسبه کرد:

$T_{(n)} = 8 \times T_{(n/2)} + O_{(n^2)}$

مطابق قضیه اصلی در حالت اول که :

$$\text{If } f(n) = O(n^{\log_b a - \varepsilon}) \text{ for } \varepsilon > 0 \text{ then: } T(n) = \theta(n^{\log_b a})$$

بنابراین خواهیم داشت $a = 8, b = 2$ و بنابراین:

$$T(n) = O(n^3)$$

حال حافظه اشغالی را بررسی میکنیم :

$$M_{(n)} = 4M_{(n/2)} \text{ so there for : } M_{(n)} = c\times 4^{\log_2 n-1}$$

در نتیجه : $M_{(n)} = O(n^2)$

شبهه کد این الگوریتم:

```
Mult (A, B, n)
If n=1 Output A×B
Else
Compute A11, B11, . . ., A22, B22
X1 ← Mult (A11, B11, n/2)
X2 ← Mult (A12, B21, n/2)
X3 ← Mult (A11, B12, n/2)
X4 ← Mult (A12, B22, n/2)
X5 ← Mult (A21, B11, n/2)
X6 ← Mult (A22, B21, n/2)
X7 ← Mult (A21, B12, n/2)
X8 ← Mult (A22, B22, n/2)
C11←X1+X2
C12←X3+X4
C21←X5+X6
C22←X7+X8
Output C
End If
```

کد پایتون این الگوریتم

```
import numpy as np

def split(matrix):
        row, col = matrix.shape
        row2, col2 = row//2, col//2
```

```
        return matrix [:row2, :col2], matrix[:row2, col2:], matrix[row2:, :col2], matrix[row2:, col2:]

def divideconquer (x, y):

        if len(x) == 1:
                return x * y
        a, b, c, d = split(x)
        e, f, g, h = split(y)

        c11 = a×e + b×g
        c12 = a×f + b×h
        c21 = c×e + d×g
        c22 = c×f + d×h
        c = np.vstack((np.hstack((c11, c12)), np.hstack((c21, c22))))
        return c
```

کارکرد این روش برای ماتریس های غیر مربعی:

این الگوریتم در عمل برای ماتریس های غیر مربعی سریع تر عمل میکند. کافیست به جای تقسیم کردن هردو ماتریس به چهار تکه یکی از ماتریس ها را به دو تکه ی برابر (یا تقریبا برابر) با تقسیم کردن سطرها یا ستون ها تقسیم کنیم. در زیر می توانید الگوریتم چنین کاری را ببینید:

Inputs: matrices A of size $n \times m$, B of size $m \times p$.

Base case: if max(n, m, p) is below some threshold, use an unrolled version ofthe iterative algorithm.

Recursive cases:

If max(n, m, p) = n, split A horizontally:

$$C = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

Else, **If** max(n, m, p) = p, split B vertically:

$$C = A\begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix}$$

Otherwise, max(n, m, p) = m. Split A vertically and B horizontally:

$$C = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2$$

بررسی حافظه ی نهان در این الگوریتم:

الگوریتم گفته‌شده در این بخش تقسیم‌بندی را تا جایی می‌تواند ادامه دهد که کل ماتریس در حافظهٔ کش جا شوند و بنابراین از نظر تعداد خطاهای کش مانند روش تقسیم‌بندی بلوکی عمل می‌کند. با این تفاوت که در آن الگوریتم خود پیاده‌سازی الگوریتم با توجه به اندازهٔ کش پردازندهٔ هدف انجام می‌شود
(پارامتر O(sqrtM) ای را باید در خود متن الگوریتم تعیین کنیم) در حالیکه این الگوریتم برای کش‌های پویا با اندازه‌های مختلف بهینه‌تر عمل خواهد کرد.

تعداد خطاهای کش در این الگوریتم با M خط حافظهٔ کش که هر خط b بیت دارد به صورت زیر خواهد بود:

$$\Theta\left(m+n+p+\frac{mn+np+mp}{b}+\frac{mnp}{b\sqrt{M}}\right)$$

از معایب این روش میتوان به این موضوع اشاره کرد که همانطور که گفته شد، محاسبه ی ضرب از $O(n^3)$ می‌باشد و برتری خاصی به دیگر الگوریتم ها ندارد و چه بسا که به دلیل اشغال حافظه اضافی، سخت افزار پیچیده تر و نیاز به مربعی کردن ماتریس وجود دارد بنابراین نسبت به الگوریتم محاسبه ی عادی ماتریس، دارای محدودیت های بیشتری هنگام پیاده سازی سخت افزاری است.

الگوریتم Strassen Matrix Multiplication -6

الگوریتم هایی وجود دارند که زمان اجرای بهتری از الگوریتم های فوق دارند. اولین الگوریتم کشف شده که اینگونه بود الگوریتم استراسن بود که در سال 1969 توسط وولکر استراسن(Volker Strassen) کشف شد. این الگوریتم به «الگوریتم سریع ضرب ماتریس» نیز معروف است. این الگوریتم بر مبنای ضرب دو ماتریس 2*2 با 7 عملیات ضرب است که در عوض، تعداد بیشتری جمع و عملیات ریاضی این چنینی لازم دارد.

در ابتدای بحث، به حاصل ضرب دو بلوک 2*2 می پردازیم که هر بلوک، ماتریسی مربعی است. طبق الگوریتم -1، که همان ضرب عادی دو ماتریس است، هر درایه به صورت Cij = Ai1B11 + Ai2B21 محاسبه می شود بنابراین 8 ضرب و 4 عمل جمع برای ضرب دو بلوک زیر نیاز داریم:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

همانطور که گفتیم، Strassen نشان داد که میتوانیم ماتریس C را با تنها محاسبه ی 7 ضرب و 18 جمع، به صورت زیر محاسبه کرد:

$$
\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}),\\
P_2 &= (A_{21} + A_{22})B_{11},\\
P_3 &= A_{11}(B_{12} - B_{22}),\\
P_4 &= A_{22}(B_{21} - B_{11}),\\
P_5 &= (A_{11} + A_{12})B_{22},\\
P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}),\\
P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}),\\
C_{11} &= P_1 + P_4 - P_5 + P_7,\\
C_{12} &= P_3 + P_5,\\
C_{21} &= P_2 + P_4,\\
C_{22} &= P_1 + P_3 - P_2 + P_6.
\end{aligned}
$$

به راحتی می توان با جایگزینی معادلات بالا، صحت ادعا را ثابت کرد. حال به سراغ بررسی حالت کلی می رویم:

فرض کنید n = 2m است. در نتیجه، بلوک ها m در m هستند. بنابراین میتوان چنین گفت که در ضرب C = AB، شامل 3^(2m) ضرب و 2(2m) – 3^(2m) جمع می باشد. حال اگر الگوریتم استراسن را در نظر بگیریم، یعنی عملیات ضرب معمولی ماتریس ها در سطح بلوک انجام شود، آنگاه به تعداد 7m^3 ضرب به 7m^3 + 11m^2 جمع لازم داریم. پس میتوان گفت اگر m>>1 باشد، آنگاه الگوریتم استراسن تقریبا 7/8 الگوریتم ضرب معمولی عمل میکند.

الگوریتم استراسن قابل بهبود یافتن است. درواقع، اگر این الگوریتم را بر روی هر یک از بلوک های ضرب مرتبط با Pi که سایزشان نصف ماتریس اصلی است پیاده کنیم. بنابراین، درواقع اگر A و B ماتریس های n*n باشند و n = 2^q، آنگاه میتوانیم بار ها و به صورت تکرار شونده، الگوریتم استراسن را اجرا کنیم تا زمانی که به حالت پایه ی 1*1 برسیم. البته واضح است که هیچ نیازی برای رساندن تا n = 1 نداریم. زمانی که سایز بلوک به اندازه ی کافی کوچک شد ($n <= n_{min}$)، آنگاه معقول تر است تا از ضرب معمولی دو ماتریس استفاده کنیم تا Pi را به دست آوریم. به همین خاطر، اغلب میبینیم که ماتریس پایه را 2*2 در نظر میگیرند.

پس در نهایت، اگر فرض کنیم n = 2^q و $A \in R^{n\times n}$ و $B \in R^{n\times n}$، اگر n_{min} = 2^d باشد به طوری که d<=q، الگوریتم زیر مقدار حاصل ضرب C = AB را با اجرا کردن الگوریتم استراسن و به صورت بازگشتی محاسبه میکند.

توجه داشته باشید که الگوریتم استراسن برخلاف الگوریتم های پیشین بازگشتی است و درواقع از روش تقسیم و غلبه بهره گرفته شده است. تابع این الگوریتم در ادامه که با متلب نوشته شده است آمده است:

function $C = \text{strass}(A, B, n, n_{\min})$
    if $n \le n_{\min}$
        $C = AB$ (conventionally computed)
    else
        $m = n/2;\ u = 1{:}m;\ v = m+1{:}n$
        $P_1 = \text{strass}(A(u,u) + A(v,v), B(u,u) + B(v,v), m, n_{\min})$
        $P_2 = \text{strass}(A(v,u) + A(v,v), B(u,u), m, n_{\min})$
        $P_3 = \text{strass}(A(u,u), B(u,v) - B(v,v), m, n_{\min})$
        $P_4 = \text{strass}(A(v,v), B(v,u) - B(u,u), m, n_{\min})$
        $P_5 = \text{strass}(A(u,u) + A(u,v), B(v,v), m, n_{\min})$
        $P_6 = \text{strass}(A(v,u) - A(u,u), B(u,u) + B(u,v), m, n_{\min})$
        $P_7 = \text{strass}(A(u,v) - A(v,v), B(v,u) + B(v,v), m, n_{\min})$
        $C(u,u) = P_1 + P_4 - P_5 + P_7$
        $C(u,v) = P_3 + P_5$
        $C(v,u) = P_2 + P_4$
        $C(v,v) = P_1 + P_3 - P_2 + P_6$
end

اگر $n_{min} >> 1$ باشد، آنگاه کافیست تعداد ضرب هارا بشماریم چراکه تعداد جمع ها تقریبا یکسان است. و اگر تنها تعداد ضرب هارا بشماریم، کافی است داخلی ترین سطح بازگشت (deepest level of the recursion) را که تمام ضرب ها در آن اتفاق می افتد بررسی کنیم. در استراسن، به تعداد q-d زیر مجموعه (زیربلوک) داریم و بنابرین به تعداد 7^(q-d) تا ضرب معمولی داریم که باید انجام شوند. این ضرب ها همگی سایز n_{min} دارند و بنابراین استراسن به تعداد $(2^d)^3\ 7^{q-d}$ ضرب دارد. در مقایسه با ضرب معمولی که به تعداد $(2^d)^3$ تا ضرب دارد، داریم:

$$\frac{s}{c} = \left(\frac{2^d}{2^q}\right)^3 7^{q-d} = \left(\frac{7}{8}\right)^{q-d}.$$

پس اگر $d = 0$ باشد، یعنی آنقدر الگوریتم را اجرا کنیم تا به ماتریس 1*1 برسیم، خواهیم داشت:

$$s = (7/8)^q\, c = 7^q = n^{\log_2 7} \approx n^{2.807}$$

بنابراین نتیجه میگیریم تعداد عملیات ضرب در الگوریتم استراسن از مرتبه ی $O(n^{2.807})$ می باشد. با این حال، باید دقت کرد که تعداد عملیات های جمع نیز (نسبت به تعداد ضرب ها) زمانی که n_{min} کوچکتر می شود، حائز اهمیت خواهد شد.

معایب این الگوریتم به شرح زیر است:

۱- در این الگوریتم، تعداد ثابت هایی که از آن ها استفاده کرده ایم زیاد است و برای کاربرد های معمول، بهینه نیست.

۲- برای (Sparse matrices) روش های بهتری ویژه این نوع ماتریس ها وجود دارد.

۳- از آنجایی که محاسبات کامپیوتری دقت محدودی روی مقادیر غیر صحیح دارند الگوریتم ساده نسبت به الگوریتم استراسن، خطاهای بزرگ کمتری دارد.

۴- چون این الگوریتم بازگشتی است، زیر ماتریس ها فضای اضافه میگیرند و حافظه بیشتری اشغال میشود.

دیگر الگوریتم ها:

الگوریتم کوپراسمیت-وینوگارد: سریع ترین الگوریتم با $O(n^k)$ الگوریتمی است که از تعمیم الگوریتم کوپراسمیت-وینوگارد به دست آمده و از نظر زمانی $O(n^{2.3728939})$ می باشد. این الگوریتم توسط François Le Gall کشف شد و به قدری ضرایب زیادی دارد و سربار الگوریتم بالاست که تنها برای ماتریس های بسیار بزرگی که هم اکنون در علوم کامپیوتر کاربردی ندارند، کارامد خواهد بود. درواقع، این الگوریتم اغلب به عنوان یک بلاک سازنده برای تئوری اثبات حد زمانی در بقیه الگوریتم ها استفاده می شود و همانطور که گفتیم، برخلاف الگوریتم استراسن در عمل استفاده نمی شود زیرا تنها مزیتی برای ماتریس های بسیار بزرگ فراهم میکند که نمی توانند توسط سخت افزارهای کنونی پردازش شوند.

الگوریتم فریوالد: الگوریتم فریوالد یک الگوریتم تصادفی احتمالاتی در زمینه ی ضرب ماتریس هاست. همانطور که می دانیم، ضرب دو ماتریس $n \times n$ به روش معمول از مرتبه زمانی $O(n^3)$ است زیرا هر یک از n^2 درایه ماتریس اول در n درایه از ماتریس دوم ضرب می شوند. یکی از نیازهای رایج در اعمال ریاضی مختلف در ماتریس ها، بررسی این موضوع است که اگر سه ماتریس A و B و C داشته باشیم آیا $A \times B = C$ یک راه ساده ی محاسبه ی $A \times B$ با روش رایج و چک کردن برابری درایه به درایه آن با C است. این روش، قطعی است و خطایی ندارد اما برای n های بزرگ بسیار زمان بر است.

الگوریتم فریوالد، به وسیله ی یک فرایند تصادفی، مرتبه زمانی را تا $O(n^3)$ کاهش می دهد و در k بار اجرای الگوریتم، احتمال خطای آن کم تر از 2^{-k} است که این برای k های بزرگ ناچیز و قابل قبول است.

جمع بندی...

با توجه به این که باید حداقل روی همه اعضای دو ماتریس $n \times n$ پیمایش انجام بشود (n^2)∩برای الگوریتم های ضرب ماتریس وجود دارد. راز (Raz) ثابت کرد که کران پایین $(n^2 \log n_n)$∩ نیز برای هر الگوریتم ضرب ماتریس نیز وجود دارد.

ممیز شناور و استاندارد IEEE-754

پیش از اینکه به توضیح این دو سیستم بپردازیم، خوب است به شرح توضیحات بیشتری پیرامون IEEE-754 میپردازیم که استاندارد مربوط به اعداد ممیز شناور می باشد

به لحاظ تاریخی، کامپیوترهای گوناگون انتخاب‌های متفاوتی در تعیین مبنا، کران‌های نما و ارقام مانتیسِ نمایش ممیز شناور داشته‌اند. در سال 1985 با تلاش‌های گروهی متشکل از ریاضیدانان، دانشمندان علوم کامپیوتر و شرکت‌های تولید سخت‌افزار به سرپرستی ویلیام کاهان از دانشگاه کالیفرنیا، استانداردی برای نمایش اعداد ممیز شناور تحت عنوان IEEE 754 به سازندگان سخت‌افزارها عرضه شد. هم‌اکنون در بیشتر کامپیوترها از این استاندارد استفاده می‌شود. استاندارد IEEE754، چند قالب کلی با دقت‌های مختلف از جمله دقت معمولی، دقت مضاعف و دقت‌های معمولی و مضاعف توسعه یافته برای نمایش اعداد ارائه می‌کند. در این‌جا به منظور آشنایی بیشتر با شیوه‌ی نمایش اعداد در این استاندارد، نحوه‌ی نمایش در دقت معمولی و مضاعف را شرح می‌دهیم.

مبنای در نظر گرفته شده در این استاندارد β=2 است. مطابق این استاندارد، در دقت معمولی از 32 بیت و در دقت مضاعف از 64 بیت برای نمایش یک عدد استفاده می‌شود. هر نمایش از سه بخش تشکیل می‌شود که عبارتند از علامت (s)، نمای تعدیل یافته (c) و قسمت کسری مانتیس نرمال شده (f).

این سه بخش با استفاده از روابط زیر مشخص می‌شوند به‌صورت (s|c|f) در کنار هم قرار می‌گیرند:

دقت معمولی:

$$x=\pm(1.f)2\times2e=(-1)s(1.f)2\times2c-127$$

دقت مضاعف:

$$x=\pm(1.f)2\times2e=(-1)s(1.f)2\times2c-1023$$

در دقت معمولی، از 32 بیت اختصاص داده شده برای نگهداری عدد، یک بیت برای علامت (s) استفاده می‌شود به‌طوری که s=0 برای علامت مثبت و s=1 برای علامت منفی به‌کار می‌رود. از 31 بیت باقیمانده، 8 بیت آن برای نگهداری نمای تعدیل یافته (c) و 23 بیت آن برای قسمت کسری مانتیس نرمال شده (f) استفاده می‌شود. در دقت مضاعف، از 64 بیت اختصاص داده شده برای نگهداری عدد، یک بیت برای علامت (s) و از 63 بیت باقیمانده، 11 بیت آن برای نگهداری نمای تعدیل یافته (c) و 52 بیت آن برای قسمت کسری مانتیس نرمال شده (f) استفاده می‌شود.

همان‌طور که ملاحظه می‌کنید شکل کلی قالب‌های ذکر شده در دقت‌های معمولی و مضاعف، کمی شبیه به نمایش ممیز شناور نرمال است. فقط باید توجه داشت که در استاندارد IEEE مانتیس x به صورت $(1.f)_2$ نرمال و تنها قسمتی از مانتیس که با f نشان داده شده است، نمایش داده می‌شود. در واقع، چون اولین بیت مانتیس نرمال شده همواره برابر با 1 است نیازی به ذخیره‌سازی آن نیست. در عوض، این بیت برای نمایش نما مورد استفاده قرار می‌گیرد.

مثال: عدد $x=-45.75$ را در نظر بگیرید. می‌خواهیم این عدد را در استاندارد IEEE با دقت معمولی نمایش دهیم. برای این منظور، ابتدا نمایش دودویی آن را می‌یابیم. داریم $x=-(101101.11)$. حال باید این عدد را به فرم $(1.f)_2\times 2^e$ درآوریم:

$$x=-(1.0110111)\times 2^5$$

اکنون از این تساوی باید مقادیر f،s و c را بیابیم. با توجه به قالب کلی دقت ساده داریم:

$$e=5=c-127\quad f=0110111,\quad s=1,$$

در نتیجه $c=132=(10000100)_2$ و بنابراین:

$$x=1|10000100|01101110000000000000000$$

مثال: عدد زیر در استاندارد IEEE با دقت معمولی نمایش داده شده است.

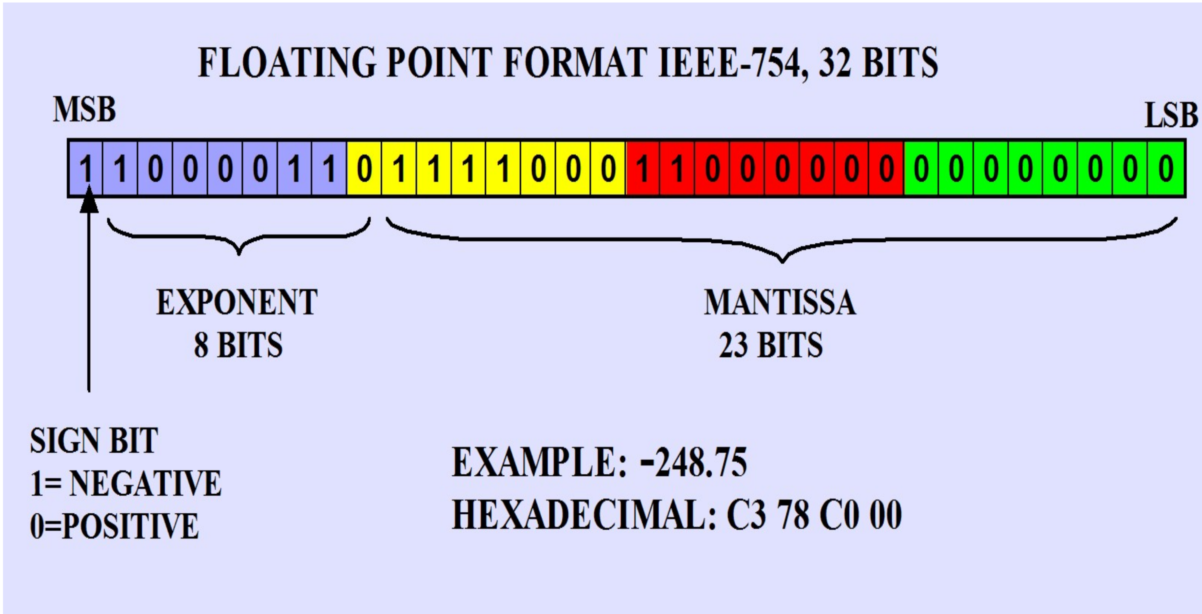
$$y=0|10000001|10011000000000000000000$$

می‌خواهیم نمایش اعشاری آن را بیابیم. با توجه به نمایش فوق داریم:

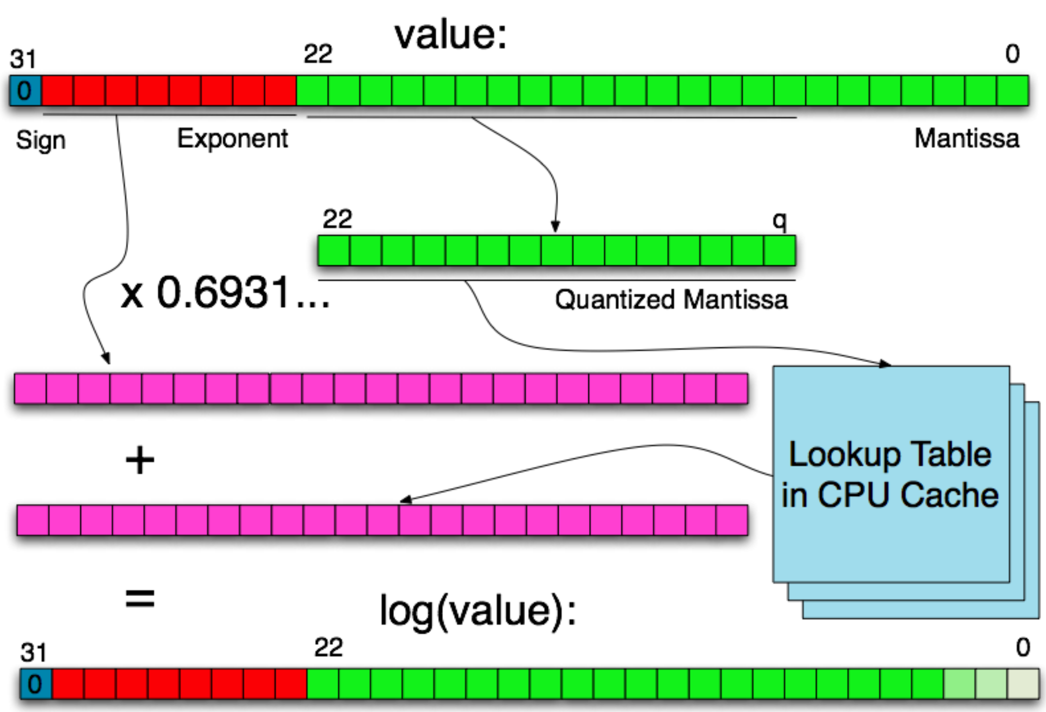
$$f=10011\quad c=(10000001)_2=129,\quad s=0,$$

بنابراین y عددی مثبت است و $e=c-127=129-127=2$. در نتیجه:

$$y=+(1.f)_2\times 2^e=(1.10011)_2\times 2^2=(110.011)_2=6.375$$



شکل۱-ممیز شناور در استاندارد IEEE754



شکل۲-نحوه کار با ممیز شناور در استاندارد IEEE754

جدول زیر خلاصه ای از کوچکترین قالبهای این استاندارد است.

جدول۱- خلاصه ای از کوچکترین قالبهای استانداردIEEE754

نام	اسم معمول	مبنا	تعداد بیت‌ها/ارقام ضریب علمی	ارقام دەدهی	بیت های توان	بایاس توان	نمای بیشنیه	نمای کمینه
باینری۱۶	Half precision	2	11	۳.۳۱	5	$2^4 - 1 = 15$	15	-14
باینری۳۲	Single precision	2	24	۷.۲۲	8	$2^7 - 1 = 127$	127	-126
باینری۶۴	Double precision	2	53	۱۵.۹۵	11	$2^{10} - 1 = 1023$	1023	-1022
باینری۱۲۸	Quadruple precision	2	113	۳۴.۰۲	15	$2^{14} - 1 = 16383$	16383	-16382
باینری۲۵۶	Octuple precision	2	237	۷۱.۳۴	19	$2^{18} - 1 = 262143$	262143	-262142
دهدهی۳۲		۱۰	۷	۷	۷.۵۸		96	-95
دهدهی۶۴		۱۰	۱۶	۱۶	۹.۵۸		384	-383
دهدهی۱۲۸		۱۰	۳۴	۳۴	۱۳.۵۸		6144	-6143

ارائه ی راه حل سخت افزاری برای واحد ضرب کننده ی دو ماتریس

با بررسی اعضای گروه در راستای یافتن الگوریتم مناسبی که کد سخت افزار بر اساس آن زده شود، در نهایت دو معماری در نظر گرفته شد. در معماری اول، که کد سخت افزاری آن زده شده است؛ مسیری طی شد که زمان بهتری نسبت به معماری دوم دارد و تا حدی هم میتوان گفت مشابه آن است.
معماری دوم، مربوط به الگوریتم استراسن و یا استفاده ازتقسیم و غلبه بود که در ادامه به الزامات چگونگی پیاده سازی آن و گام هایی که لازم است طی شود میپردازیم.

شایان به ذکر است که علت استفاده ی ما از معماری اول، بیشتر به علت کاستی الگوریتم مربوط به معماری دوم بود. چراکه الگوریتم تقسیم و غلبه همانطور که شرح داده شد از لحاظ زمانی تفاوتی با الگوریتم ساده ی ضرب ماتریس ندارد. تنها تفاوت آن، این است که قابلیت بهبود دارد ولی پیاده سازی این قابلیت بهبود و محاسبات لازمه، از لحاظ زمانی و سخت افزاری به صرفه نخواهد بود. از طرفی، الگوریتم استراسن اگرچه از لحاظ زمانی بهینه تر از روش عادی ضرب بود، ولی از آنجایی که این الگوریتم مختص به ماتریس های مربعی است در این پروژه کارایی ندارد. میتوانستیم این محدودیت مربعی نبودن را نیز جبران کنیم اما حتی اگر ماتریس غیر مربعی را با روش هایی مانند اضافه کردن سطرها و ستون های صفر به مربع تبدیل کنیم و ضرب را انجام دهیم و سپس آن را به ابعاد مورد نظر برسانیم، الگوریتم مزیت زمانی خود را از دست خواهد داد. چه بسا این اتفاق برای الگوریتم تقسیم و غلبه نیز خواهد افتاد.

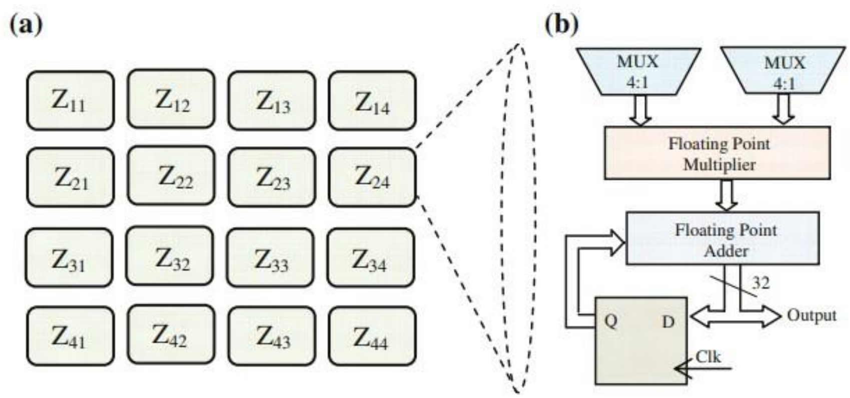
معماری اول

مشخصات کلی

این معماری شامل دو ماژول اصلی حافظه ی RAM و matrix mult می باشد که matrix mult به کمک دو ماژولی که توسط استاد داده شد، یعنی single multiplier و adder، واحد ضرب کننده ماتریسی برای اعداد ممیز شناور به فرمت IEEE 754 را تشکیل می دهند. بنابراین ،هر بار، سطر و ستونی از دو ماتریس از حافظه select می شوند که درایه های آن اعداد به فرمت مذکور می باشند. سپس به صورت ترتیبی (با توجه به کلاک سیستم)، به شکلی که از نظر سخت افزاری، سرعت و مساحت بهینه ای داشته باشد، آن سطر و ستون خوانده شده در هم ضرب شده و یک درایه ی مربوط به ماتریس را خروجی تشکیل میشود. سپس، دوباره به سراغ حافظ می رویم تا سطرها و ستون های بعد select شوند.

پیش از توضیح ماژول های این معماری، به بررسی آن و مقایسه ی آن با داک دریافت شده می پردازیم. در این طراحی، توانستیم در نهایت، به واحد ضرب کننده ای همانند واحد ضرب کننده ای که در داک داده شده بود، دست بیابیم با این تفاوت که توانستیم آن را به صورتی بهینه تر، بنویسیم.

در واقع، طرح داده شده به شکل زیر بود:



که از طرح بالا متوجه می شویم که هرکدام از MUX ها وظیفه ی انتخاب بین 4 بلوک یا زیر ماتریس ماتریسZ را دارند. در واقع ماتریس 4*4 بالا، به 4 زیرماتریس یا بلوک 2*2 تقسیم می شود.

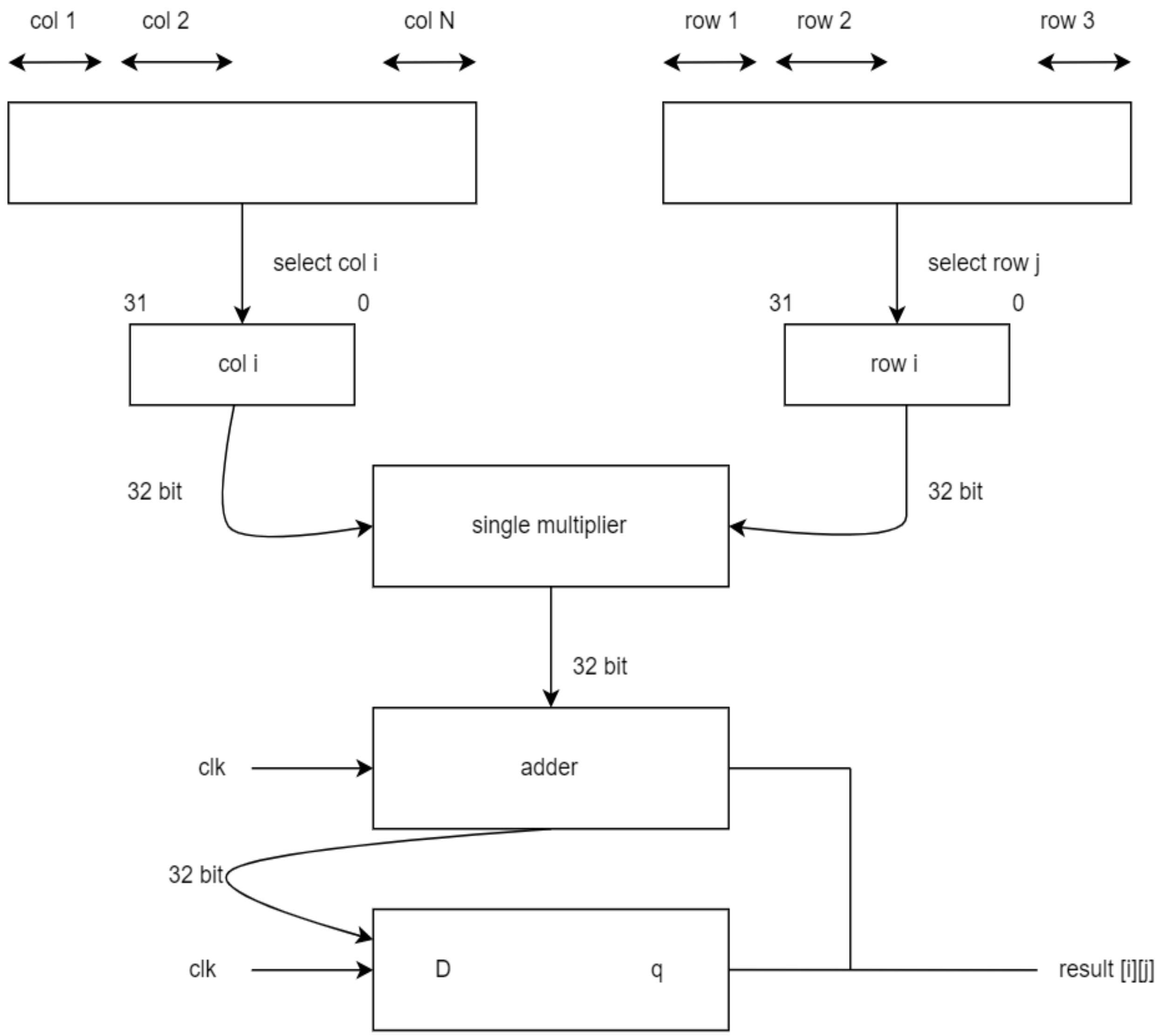
حال، سه تفاوتی که بین طرح ما و این طرح است به شرح زیر است و مابقی تفاوت چندانی ندارد:

1) ورودی های ما، به صورت ماتریس هایی در یک بعد (به صورت آرایه) هستند. به طور مثال، ماتریس 2*2 ی In1 به صورت زیر نوشته می شود:

mat1 [127:96]   : درایه اول   //سطر اول ماتریس 1

mat1 [95:64] : درایه دوم // سطر اول ماتریس 1
mat1 [63:32] : درایه سوم // سطر دوم ماتریس 1
mat1 [31:0] : درایه چهارم // سطر دوم ماتریس 1

و در نهایت، آنچه به عنوان ورودی داده می شود، [127:0] mat1 می باشد.
2) تفاوت دوم (که مشهودتر و به دنبال مورد اول است)، این می باشد که در طرح ما، هر بار یک سطر و یک ستون از هر ماتریس انتخاب می شوند و نه به صورت بلوک بلوک. ولی این روش با توجه به شیوه ای که برای نمایش و ورودی دادن هر یک از ورودی ها (به صورت یک آرایه به جای نمایش ماتریس 2*2) قرار دادیم مناسب تر است و منطق آن فرقی ندارد. از طرفی، بلوک بلوک کردن و درواقع استفاده از الگوریتم استراسن معایبی در پی دارد که هم در بخش توضیح الگوریتم مطرح شد و هم در ادامه ی این گزارش باری دیگر اشاره خواهیم کرد.
3) برای جلوگیری از کار بیهوده، ماژول ماکس را در طرحمان به صورت درونی داخل matrixmult پیاده سازی کردیم و از زدن آن به عنوان ماژولی مستقل خودداری کردیم چراکه هم لزومی نداشت و هم همانطور که در مورد دوم توضیح داده شد، سایز ورودی های MUX و به دنبال آن سیگنال کنترل کننده وابسته به اندازه ی ستون ورودی اول یا سطر ورودی دوم می باشد.
پس درون ماژول matrixmult که عملیات اصلی را انجام می دهد، واقعه ای همانند شکل زیر درحال رخ دادن است:



State machine:

برای طراحی state machine بدینصورت عمل می‌کنیم؛
در ابتدا اگر ابعاد هر یک از ماتریس های ورودی فرد بود آن را extend کرده و با اضافه کردن سطر یا ستون صفر، ابعاد را زوج میکنیم.
به عنوان مثال اگر ماتریس ما ۳*۲ بود، آن گاه یک سطر صفر به از پایین به ماتریس اضافه می کنیم یا اگر ۴*۳ بود، یک ستون صفر از سمت راست به ماتریس اضافه می‌کنیم. بدیهی است اگر ابعاد ماتریس، زوج بود تغییری در آن ایجاد نمیکنیم.
علت extend کردن: از آنجایی که ماتریس را به بلوک های ۲*۲ تقسیم می‌کنیم، بنابراین نیاز است تا ابعاد ماتریس زوج باشد و بدیهی است اضافه کردن سطر یا ستون صفر در حاصل تاثیری نمیگذارد.

سپس ۲ تا counter در نظر می گیریم.
سپس آدرس ها را تعیین کرده تا بر اساس آن به بتوانیم بلوک های ماتریس اول و بلوک های متناظر در ماتریس دوم را بخوانیم و در یکدیگر ضرب کنیم.
در مرحله بعد جمع های داخلی برای ضرب بلوک ها را انجام میدهیم.
سپس حاصل را در موری write می‌کنیم.
نکته حائز توجه در اینجا این است که ما بعد از عملیات بالا یک clock صبر میکنیم تا در حافظه نوشته شود و به اصطلاح stable شود.
سپس counter هایی که برای خواندن از حافظه در نظر گرفته شده بودند را افزایش میدهیم.
در نهایت به پایان یک چرخه یا همان reset میرسیم.

ورودی های بخش مربوط به قسمت ضرب کننده

در ابتدا باید به سیستم دو ماتریس با ابعاد n * p و m * n را دریافت میکند که هر یک از درایه های این ماتریس ها ۳۲ بیتی و مطابق با استاندارد IEEE754 است. با دقت به ضرب معمولی دو ماتریس متوجه میشویم که در هر مرحله، سطرهای ماتریس اول در ستون های ماتریس دوم ضرب می شوند و سپس با جمع مقادیر حاصله، پاسخ مربوط به درایه ای از خروجی که مربوط به آن سطر و ستون بود به دست می آید. عملیات ضرب درایه ها در یکدیگر را به کمک ماژول ضرب کننده، عملیات جمع کردن پاسخ های ماژول ضرب کننده را به کمک ماژول جمع کننده انجام میدهیم و انجام ضرب کلی دو ماتریس به عهده ی ماژول matrixmult می باشد. از طرفی یک ماژول موری هم در سیستم تعبیه شده که در ادامه بیشتر به آن می پردازیم.

خروجی های ضرب کننده

با توجه به اینکه ورودی هایمان دو ماتریس با ابعاد n * p و m * n بودند، پس خروجی یک ماتریس m در p خواهد بود. همچنین لازم به ذکر است با توجه به شیفت بیت های mantissa و افزایش exponent اعداد، ممکن است برخی بیت ها به علت نبودن دقت کافی برای نمایششان از دست بروند و محاسبه با اندکی خطا مواجه شود.

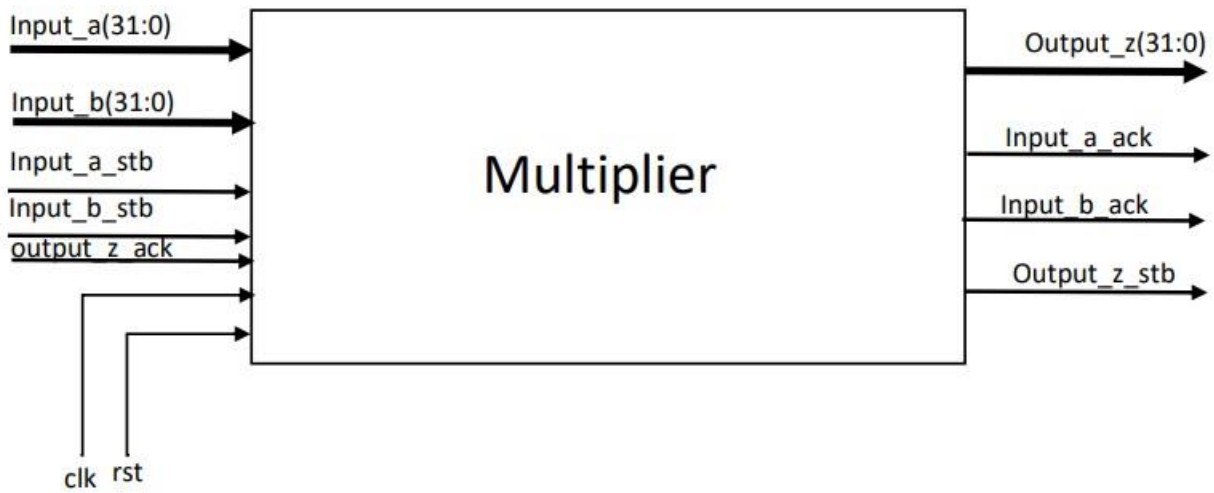
ماژول single_multiplier:

این ماژول و تست بنچ مربوط به آن، توسط خود استاد داده شد بنابراین به شرح خلاصه ای از کارکرد آن می پردازیم. واحد ضرب کننده ای که در اختیار ما قرار داده شد به صورت ترتیبی عمل می کند و حاصل ضرب را بعد از چند کلاک خروجی می دهد.

وظیفه: گرفتن دو عدد اعشاری و ضرب کردن آنها در یکدیگر و دادن خروجی نرمال شده طبق استاندارد IEEEE-754

ورودی ها: input_a و input_b(ورودی هایی که در هم ضرب می شوند)، input_a_stb و input_b_stb(سیگنال هایی هستند که نشان میدهد ورودی ها stable هستند، output_z_ack(سیگنالی است که نشان می دهد خروجی توسط ماژول بالاتر خوانده شده یا نه تا پیش از آن خروجی را تغییر ندهیم)، clk، reset

خروجی ها: output_z (خروجی حاصل ضرب)، input_a_ack و input_b_ack (سیگنال هایی که نشان میدهند تا زمانی که ماژول بالاتر آنها را دریافت نکرده خروجی نباید تغییر کند)، output_z_stb (نشان میدهد خروجی آماده است)



شکل۵- ماژول (ضرب کننده) MULTIPLIER

طرز کار:

این ماژول با یک machine state دارای 11 حالت طراحی شده است.

حالتهای a_get و b_get: هنگامی که stb_a_input فعال باشد ورودی a را ذخیره می کند و ack_a_input را فعال میکند به b_get میرود و برای b همان کار را انجام می دهد و سپس به unpack میرود.

حالت unpack: مقدار عالمت، توان و اندازه عدد را جدا می کند.

حالت cases_special: حالت های استثنا مثل ضرب بینهایت در صفر، بینهایت در بینهایت یا صفر در صفر را چک می کند و اگر این حالت ها بود جواب استثنا را تولید می کند و به z_put می رود. اگر نه به a_normalise می رود.

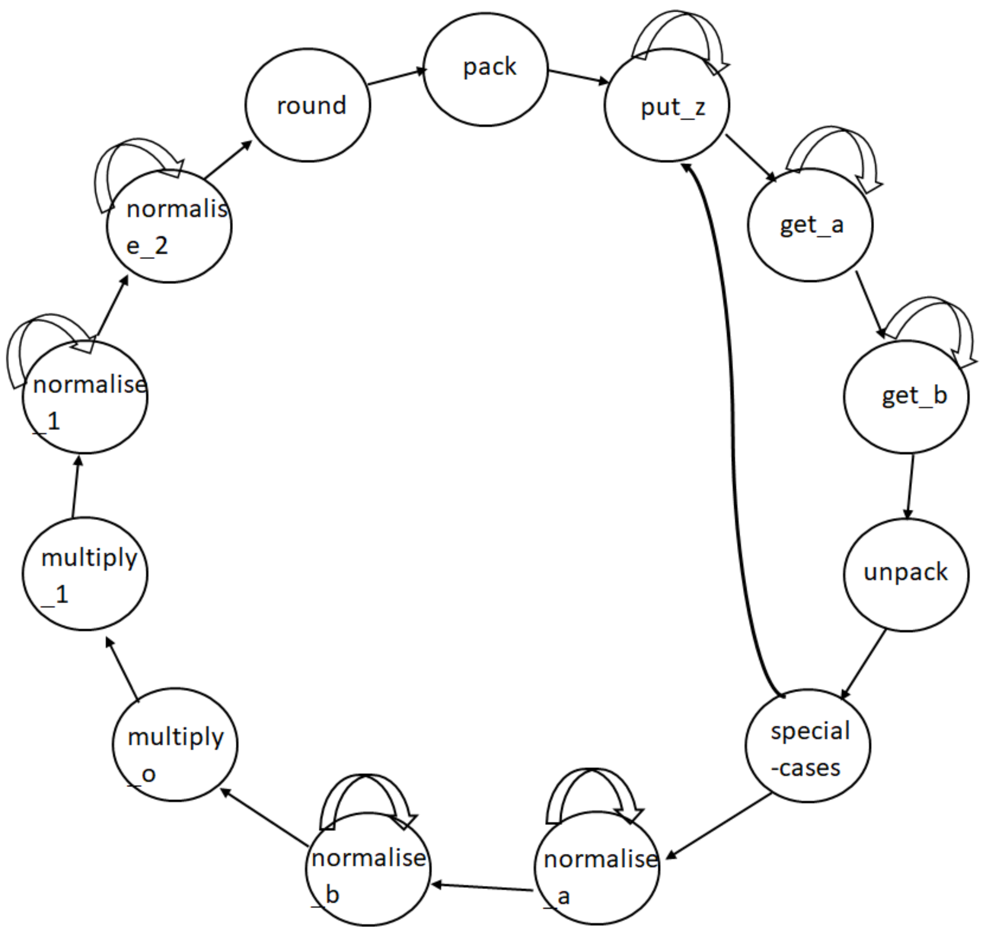
حالتهای normilize_a: بر اساس استاندارد IEEE 754 مقدار a را نرمالایز می کند (شیفت می دهد تا رقم سمت چپ ۱ شود). و به normilize_b میرود و همین کار را برای b می کند.

حالتهای multiply/1_multiply_0 : اندازه های هر دو عدد را در هم ضرب می کند.

حالتهای normalise/0_normalise/round_1: حاصل را نرمالایز می کنند.

حالت pack: علامت، اندازه و توان به دست آمده برای جواب را به صورت استاندارد IEEE 754 کنار هم قرار می دهد.

حالت put_z : مقدار حاصل ضرب را در خروجی output_z قرار میدهد و stb_z_output را فعال می کند و منتظر دریافت ack_z_output می شود. پس از دریافت آن به استیت ابتدایی a_get می رود برای ضرب بعدی.



FSM of multiplier-شکل۳

ماژول adder:

این ماژول نیز، توسط استاد داده شده است و در ادامه به طور خلاصه آن را توضیح میدهیم. به کمک این ماژول میتوانیم جمع دو عدد point floating precisions را به دست آوریم و به منظور بهینه کردن فرآیند، از یک واحد جمع کننده اعداد ممیز شناور استفاده میکنیم، در ابتدا هم باید به علامت ها توجه کنیم چنانچه هم علامت بودن جمع کرده و در غیر این صورت باید تفریق کنیم و مطابق با IEEE 754 کار

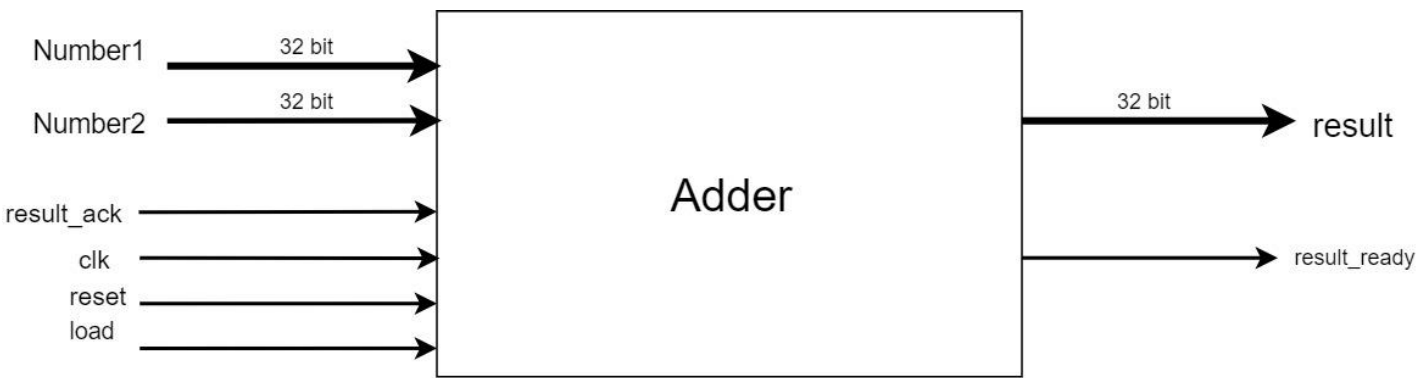
را پیش ببریم. لازم به ذکر است که در این ماژول اعداد هم رده می شوند و نمای آن ها برابر شده سپس به وسیله جمع کننده وریلاگ اعداد را با هم جمع می زنیم.

وظیفه: این ماژول یک واحد جمع کننده ی اعداد با ممیز شناور می باشد

ورودی ها: number1 و number2(ورودی هایی که با هم جمع می شوند)، result_ack(به کمک این سیگنال، تا زمانی که خروجی توسط ماژول بالاتر خوانده نشود تغییر نمیابد)، load(برای مقداردهی اولیه)، clk، reset

خروجی ها: result(خروجی)، result_ready(به کمک این سیگنال متوجه میشویم نتیجه بر روی خروجی قرار داده شده)

طرز کار:
فرآیند این واحد بدین شکل است که ابتدا نمای اعداد با یکدیگر برابر میشوند. لذا mantissa عدد با نمای کوچکتر را به راست شیفت می دهیم و به نمای آن یک واحد می افزاییم. سپس mantissa دو عدد جمع شده و عادی سازی (normalize)را انجام میدهیم. overflow را چک کرده و نتیجه را به خروجی واحد مورد نظر منتقل می کنیم



ماژول matrixmult:

وظیفه: این ماژول وظیفه ی اصلی را در مدار ما ایفا میکند. در هر مرحله ورودی های ماتریس که به شکل آرایه ورودی داده میشوند وارد ماژول می شوند؛ سپس هر بار بخشی از هر آرایه ی ورودی انتخاب می شوند که در واقع مانند فرمتی که بالاتر توضیح داده شد، این بخش ها هر یک سطری یا ستونی از ماتریس هایمان هستند(مشابه مثال پایین). سپس عمل ضرب برای آن ها صورت میگیرد و سپس به سراغ بخش های دیگر وکتورها میرویم.

به طور مثال، در ماتریس 2*2 مثال زده شده در بخش مشخصات کلی، [127:96] mat1 و mat1 [95:64] درایه های سطر اول هستند در ابتدا انتخاب می شوند. از طرفی از ماتریس دوم، درایه های mat2 [127:96] و mat2 [63:32] که درایه های ستون اول ماتریس دوم هستند برداشته می شوند.

در مرحله ای بعد، به طور موازی و همزمان و در چند کلاک میتوان ضرب این سطر و ستون انتخاب شده را به پایان رساند. به این صورت که به صورت همزمان روی درایه های سطر و ستون به ترتیب حرکت میکنیم و هر بار، درایه های متناظر با هم انتخاب می شوند تا به مرحله ی بعد بروند. برای اینکار هربار از selectRow و selectCol استفاده میکنیم. همانطور که از نام گذاری آن ها مشخص است، از این دو باس برای select کردن درایه های مربوطه در هرمرحله از هر سطر و ستون استفاده می شود.

در مرحله ی بعد، درایه های select شده وارد ضرب کننده ی ساده می شوند تا همانطور که در توضیحات آن ماژول گفته شده بود، در هم ضرب و طبق استاندارد نرمالسازی، خروجی نرمال می شوند.

پس همانطور که گفته شد، مرحله به مرحله درایه های سطرهای ماتریس اول را در درایه های ستون ماتریس دوم ضرب کرده و نتیجه نهایی را در هر مرحله با adderMat[x][y] مربوط به آن درایه جمع میکنیم و این کار را تا زمانی ادامه می دهیم تا selectRow و selectCol به انتهای آرایه ها برسند.

ورودی ها: mat1, mat2 (که ورودی ها هستند و ماتریس هایی هستند که به صورت آرایه درآوردیم)، start (مقادیر اولیه ورودی داده می شوند) ، rst، clk، show (درواقع این سیگنال نشان میدهد تا زمانی که ماژول بالاتر خروجی را دریافت نکرده باید آن را نشان دهیم و عوض نکنیم)

خروجی ها: finish (که خبر میدهد خروجی اماده است)، result (خروجی)

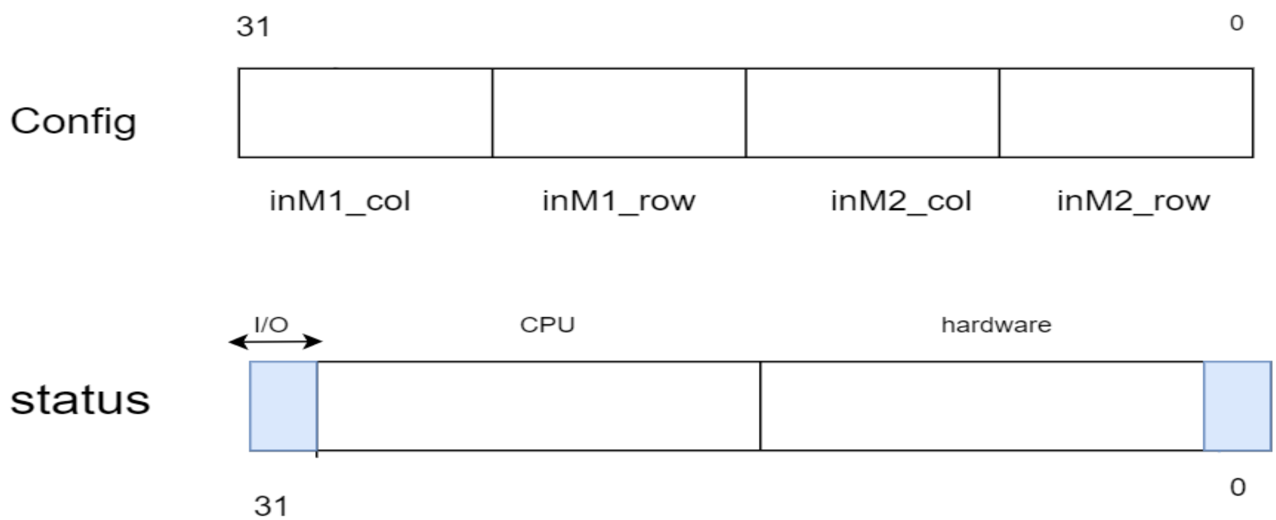
شکل۶- ماتریس ضرب کننده (MATRIX MULTIPLIER)

ماژول Memory:

از آنجایی که ماتریس های ورودی باید در جایی از سیستم ذخیره شوند، هدف از زدن memory به طور کل در چنین ساختاری به این صورت است که ورودی های ماتریس از فایل درون ram ریخته شوند و در دو word اول این حافظه نیز، دو ایندکس صفرم و یکم بخش هایی به عنوان config و starus در نظر گرفته می شود.

در ایندکسی که مربوط به confing است، اطلاعاتی از قبیل تعداد سطر و ستون ماتریس اول و تعداد سطر و ستون ماتریس دوم قرار دارد که هر یک از این 4 تا، در 8 بیت از این word عه 32 بیتی ذخیره می شوند.

در ایندکسی که مربوط به status است، اطلاعات و بیت هایی (طبق فرمتی خاص که مشخص باشد) قرار دارد که روابط بین حافظه و cpu را تحقق می بخشد. در واقع گویی یک حافظه داریم که matrix mult و cpu به آن دسترسی دارند.

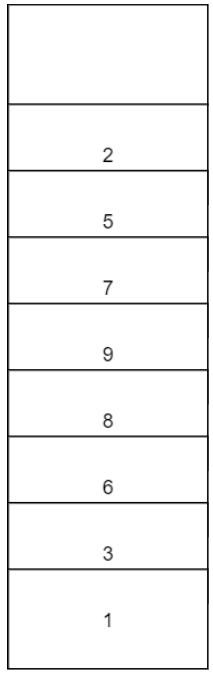


نکته ی دیگری که باید به آن توجه شود، نحوه ی ذخیره سازی ماتریس ها در حافظه است. به طور مثال، برای دو ماتریس زیر، چنین در حافظه قرار میگیرند:

ماتریس اول $\begin{matrix}2 & 5\\ 7 & 9\end{matrix}$

ماتریس دوم $\begin{matrix}8 & 6\\ 3 & 1\end{matrix}$

شکل حافظه:



در زمان پیاده سازی، همانطور که در کلاس گفته شد، باید به این موضوع توجه شود که برای عملیات write در حافظه، همانموقع که ادرس را قرار میدهیم، نوشته می شود و آن تاخیر اندک بین این دو، خود در زمان سنتز حل و فصل می شود اما برای عملیات read چنین نیست و بعد از یک تاخیر میتوانیم به داده دسترسی پیدا کنیم چراکه مدتی طول میکشد تا آن داده آماده شود.

بررسی تست بنچ و نتایج به دست آمده ی معماری اول:

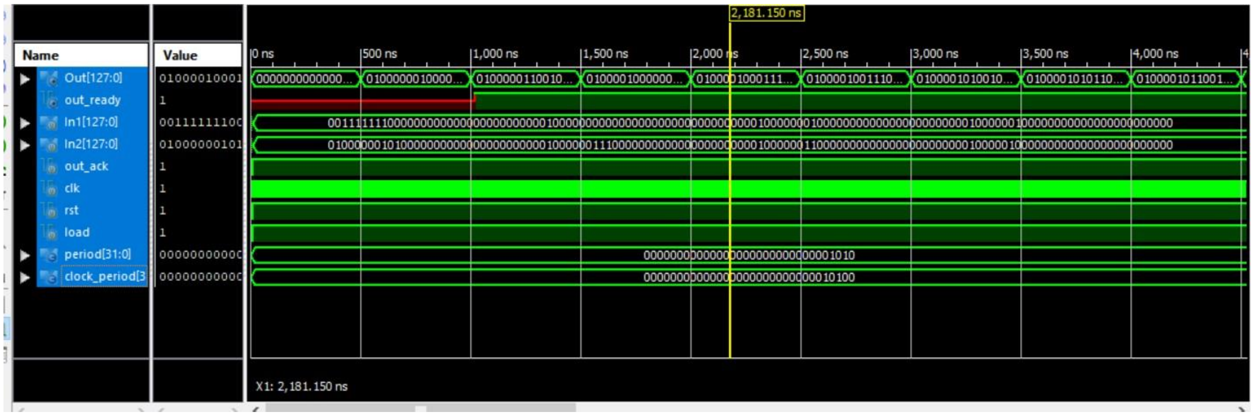
در قسمت تست این کد، دو نوع ورودی و خروجی دادیم که هرکدام ماتریس های 2*2 می باشند.(یکی کامنت شده است)

- $\begin{matrix} 2 & 1 \\ 4 & 3 \end{matrix}$ و $\begin{matrix} 6 & 5 \\ 8 & 7 \end{matrix}$ که حاصل ضرب آنها برابر است با: $\begin{matrix} 22 & 19 \\ 50 & 43 \end{matrix}$

- $\begin{matrix} 3.01 & 2.75 \\ 8 & 4.023 \end{matrix}$ و $\begin{matrix} 11.3 & 5.011 \\ 2.04 & 1.06 \end{matrix}$: $\begin{matrix} 37.2154 & 16.97085 \\ 61.7799 & 28.639253 \end{matrix}$

البته از آنجایی که نمایش اعشاری که اعداد فرم بسته ای ندارد، و مثلا مقدار 16.97085 به صورت $1,69708499908447265625000000... \times 10^1$ می نویستند، می توان گفت هر کدام از درایه های ورودی نیز چنین خطایی دارند و در نتیجه عدد خروجی از مدار، ممکن است با درصد خطای کمی، نزدیک به عددی باشد که انتظار داریم از ضرب به دست بیاوریم.

شکل موج برای مثال اول:




```
result[1][1] = 01000001011011000000010001111110
result[1][2] = 01000010000100010010100101011101
result[2][1] = 01000001110000111101011100001010
result[2][2] = 01000010010000010100011110101110
result[1][1] : confirmed
result[1][2] : confirmed
result[2][1] : confirmed
result[2][2] : confirmed
```

نسبت طلایی

کد نسبت طلایی به زبان جاوا نوشته شده است و در فایل زیپ پروژه قرار گرفته است. در در این کد، دو ماتریس ورودی از فایل خواند ه می شوند و سپس حاصل ضرب آن ها محاسبه میشود که میتوان جواب را با جواب کد سخت افزاری مقایسه کرد.

```
import java.io.*;

class MatricesMultiplier {
    private static final int n = 3;
    private static final int m = 4;
    private static final int k = 3;
    private static final String pattern1 = "/Users/Dariush/Desktop/matrix1.txt";
    private static final String pattern2 = "/Users/Dariush/Desktop/matrix2.txt";

    public static void main(String[] args) throws IOException {
        String[][] firstMatrix = readMatrixFromFile(pattern1, n, m);
        String[][] secondMatrix = readMatrixFromFile(pattern2, m, k);
        String[][] mul = matrixMultiplier(firstMatrix, secondMatrix, n, m, k);
        printMatrix(mul, n, k);
    }

    private static String[][] readMatrixFromFile(String pattern, int row, int column) throws IOException {
        String[][] matrix = new String[row][column];
        int rowCount = 0;
```

```
        int columnCount = 0;
        File file = new File(pattern);
        BufferedReader br = new BufferedReader(new FileReader(file));

        String line;
        while ((line = br.readLine()) != null) {
            if (column == columnCount) {
                columnCount = 0;
                rowCount++;
            }
            matrix[rowCount][columnCount] = line;
            columnCount++;
        }
        br.close();

        return matrix;
    }

    private static String[][] matrixMultiplier(String[][] firstMatrix, String[][] secondMatrix, int n, int m, int k) {

        float[][] firstMatrixFloat = hexMatrixToFloatMatrix(firstMatrix, n, m);
        float[][] secondMatrixFloat = hexMatrixToFloatMatrix(secondMatrix, m, k);

        float[][] answerMatrix = new float[n][k];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < k; j++) {
                answerMatrix[i][j] = 0;
                for (int l = 0; l < m; l++) {
                    answerMatrix[i][j] += firstMatrixFloat[i][l] * secondMatrixFloat[l][j];
                }
            }
        }
        return FloatMatrixToHexMatrix(answerMatrix, n, k);
    }

    private static String[][] FloatMatrixToHexMatrix(float[][] floatMatrix, int n, int m) {
        String[][] hexMatrix = new String[n][m];

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                hexMatrix[i][j] = floatToHexConverter(floatMatrix[i][j]);
            }
        }
        return hexMatrix;
    }

    private static String floatToHexConverter(float number) {

        float abs = ((number < 0) ? -1 * number : number);

        int intBits = Float.floatToIntBits(abs);
        String binary = Integer.toBinaryString(intBits);
        String hexString = Integer.toString(Integer.parseInt(binary, 2), 16);
        if (number >= 0) {
            return hexString;
        } else {
            String firstPart = "";
            int firstPartInt = 0;
```

```
        int remain = binary.length() % 4;
        firstPart += ("1" + binary.substring(0, remain));
        for (int i = 0; i < firstPart.length(); i++) {
            firstPartInt += (power(2, firstPart.length() - i - 1)) * Integer.parseInt(firstPart.split("")[i]);
        }
        String negative = (intToHexConverter(firstPartInt) + hexString.substring(1));
        return negative;
    }
}

private static String intToHexConverter(int num) {
    switch (num) {
        case 0:
            return "0";
        case 1:
            return "1";
        case 2:
            return "2";
        case 3:
            return "3";
        case 4:
            return "4";
        case 5:
            return "5";
        case 6:
            return "6";
        case 7:
            return "7";
        case 8:
            return "8";
        case 9:
            return "9";
        case 10:
            return "a";
        case 11:
            return "b";
        case 12:
            return "c";
        case 13:
            return "d";
        case 14:
            return "e";
        case 15:
            return "f";
        default:
            return "";
    }
}

private static float[][] hexMatrixToFloatMatrix(String[][] hexMatrix, int n, int m) {
    float[][] floatMatrix = new float[n][m];

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            floatMatrix[i][j] = hexToFloatConverter(hexMatrix[i][j]);
        }
    }
```

```
        return floatMatrix;
    }

    private static float hexToFloatConverter(String number) {
        Long i = Long.parseLong(number, 16);
        Float f = Float.intBitsToFloat(i.intValue());
        return f;
    }

    private static void printMatrix(String[][] answer, int row, int column) {
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < column; j++) {
                System.out.print(answer[i][j] + " ");
            }
            System.out.println("");
        }
    }

    static int power(int N, int P) {
        int pow = 1;
        for (int i = 1; i <= P; i++)
            pow *= N;
        return pow;
    }
}
```

تصویر زیر شامل ورودی ماتریس اول، ورودی ماتریس دوم، و ماتریس حاصل ضرب است که به کمک کد نسبت طلایی پرینت کردیم:

```
d6b54772 2eea184c 3a1b72ce 7973c76e
a21c118c 072adc8e 77a21ee4 ed37e65c
f46e88d0 670caa74 8df238d4 d64ba368

c22df4de a53634ae 38326c6c
21daefa8 293e815c 71e302fa
65996fb6 4dde1c0c a99b0dc6
4c010640 15b4cbca fbf4ee0a

7f800000 4fac2c64 ff800000
7f800000 7f800000 7f800000
7722169e 5a29c674 7f800000
```

نتایج سنتز

تصاویر مربوط به سنتز در ادامه آمده است و گزارشات نیز در فایل زیپ قرار دارد.

matrixMulti Project Status (07/11/2021 - 11:50:54)			
Project File:	dsdprj.xise	**Parser Errors:**	No Errors
Module Name:	matrixMulti	**Implementation State:**	Placed and Routed
Target Device:	xc3sd3400a-4fg676	• **Errors:**	
Product Version:	ISE 14.7	• **Warnings:**	
Design Goal:	Balanced	• **Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	
Environment:	System Settings	• **Final Timing Score:**	

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	1,845	47,744	3%	
Number used as Flip Flops	1,833			
Number used as Latches	12			
Number of 4 input LUTs	4,201	47,744	8%	
Number of occupied Slices	2,584	23,872	10%	
Number of Slices containing only related logic	2,584	2,584	100%	
Number of Slices containing unrelated logic	0	2,584	0%	
Total Number of 4 input LUTs	4,293	47,744	8%	
Number used as logic	4,201			
Number used as a route-thru	92			
Number of bonded IOBs	389	469	82%	
Number of BUFGMUXs	1	24	4%	
Number of DSP48As	16	126	12%	
Average Fanout of Non-Clock Nets	3.34			

```
=========================================================================
*                          Design Summary                               *
=========================================================================

Top Level Output File Name         : matrixMulti.ngc

Primitive and Black Box Usage:
------------------------------
# BELS                             : 5950
#      GND                         : 1
#      INV                         : 11
#      LUT1                        : 92
#      LUT2                        : 853
#      LUT2_D                      : 16
#      LUT3                        : 1175
#      LUT3_D                      : 175
#      LUT3_L                      : 39
#      LUT4                        : 1689
#      LUT4_D                      : 180
#      LUT4_L                      : 52
#      MULT_AND                    : 36
#      MUXCY                       : 688
#      MUXF5                       : 518
#      VCC                         : 1
#      XORCY                       : 424
# FlipFlops/Latches                : 1845
#      FDC                         : 11
#      FDCE                        : 652
#      FDE                         : 1141
#      FDPE                        : 1
#      FDR                         : 16
#      FDRE                        : 12
#      LDE                         : 12
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 388
#      IBUF                        : 259
#      OBUF                        : 129
# DSPs                             : 16
#      DSP48A                      : 16
```

```
Device utilization summary:
---------------------------

Selected Device : 3sd3400afg676-4

 Number of Slices:                     2597  out of  23872    10%
 Number of Slice Flip Flops:           1845  out of  47744     3%
 Number of 4 input LUTs:               4282  out of  47744     8%
 Number of IOs:                         389
 Number of bonded IOBs:                 389  out of    469    82%
 Number of GCLKs:                         1  out of     24     4%
 Number of DSP48s:                       16  out of    126    12%

---------------------------
Partition Resource Summary:
---------------------------

  No Partitions were found in this design.

---------------------------
```

```
TABLE OF CONTENTS
  1) Synthesis Options Summary
  2) HDL Parsing
  3) HDL Elaboration
  4) HDL Synthesis
       4.1) HDL Synthesis Report
  5) Advanced HDL Synthesis
       5.1) Advanced HDL Synthesis Report
  6) Low Level Synthesis
  7) Partition Report
  8) Design Summary
       8.1) Primitive and Black Box Usage
       8.2) Device utilization summary
       8.3) Partition Resource Summary
       8.4) Timing Report
            8.4.1) Clock Information
            8.4.2) Asynchronous Control Signals Information
            8.4.3) Timing Summary
            8.4.4) Timing Details
            8.4.5) Cross Clock Domains Report
```

معماری دوم

در معماری دوم، هدف این است که به کمک ایده گرفتن از الگوریتم استراسن، بتوانیم یک ضرب کنند ه ی ماتریس پیاده سازی کنیم. برای مطالعه ی الگوریتم، لطفا به بخش بررسی الگوریتم های موجود مراجعه کنید. در ادامه به شرح مواردی که در صورت پیاده سازی الگوریتم به فرم سخت افزاری مهم است می پردازیم و از چالش های این موضوع بیشتر میگوییم.

ماتریس بلوک، ماتریسی است که ورودی های آن، خودشان ماتریس هستند. به طور کلی، الگوریتم هایی که در سطح بلوک طراحی شده اند، معمولا در ضرب ماتریس-ماتریس بسیار کارآمد می باشند چراکه دارای عملکرد انتخابی هستند و بنابراین، در بسیاری از محاسبات کامپیوتری عملکرد بالا یا در واقع high performance دارند. اما؛ باید توجه داشت که به ازای n های بزرگ، دیتا لوکالیتی(data locality) نسبت به حجم محاسبات اصلی، تاثیر بیشتری در مفید واقع شدن الگوریتم یا همان efficiency دارد. بنابراین زمانی که از کامپیوترهای پرسرعت با حافظه ی سلسله مراتبی (براساس چندین سطح حافظه ی نهان) در کارهای محاسباتی بزرگ استفاده می شود، سازماندهی مناسب محاسبات ماتریس اهمیت بیشتری پیدا میکند و داشتن توانایی استدلال در مورد سلسله مراتب حافظه و محاسبات پردازنده های چند هسته ای، ضروری می شود.

علاوه بر مواردی که در بالا اشاره شد، نکته ی مهم دیگر این است که تحقیقاتی که بر روی محاسبات ماتریس در بسیاری از زمینه های کاربردی انجام میشود، بستگی به توسعه الگوریتم های موازی ای که scale می شوند دارند. چنین الگوریتم هایی این ویژگی را دارند که با افزایش سایز مسئله، تعداد پردازنده های درگیر نیز بیشتر می شوند.

اگرچه زبان های برنامه نویسی جدید و system tool های مربوط، پروسه ی پیاده سازی موازی محاسبات ماتریسی را راحت تر کرده اند؛ همچنان توانایی "فکر کردن به طور موازی" بسیار مهم هست. چنین موضوعی مستلزم این است که درک درستی از load balancing ، communication overhead و processor synchronization داشته باشیم.

موازی سازی محاسبات ماتریسی و موارد سخت افزاری که باید تعبیه شوند:

برای شرح دادن ایده ی اصلی مرتبط با موازی سازی محاسبات ماتریس، مدل محاسبه ی زیر را در نظر می گیریم:

با داده شدن $A \in R^{m\times r}$ و $B \in R^{r\times n}$ و $C \in R^{m\times n}$، می توان به طور موثر حاصل ضرب را با به روزرسانی کردن مقدار $C = C + AB$ محاسبه کرد. به این صورت که p پردازنده در دسترس است و هر پردازنده حافظه ی محلی خودش را دارد و همچنین برنامه ی محلی(یا همان local program) خودش را اجرا کند.

می توان این ادعا را داشت که ایده ی سخت افزاری موازی پیاده سازی کردن به روزرسانی حاصل ضرب، ایده ی مفیدی است چراکه به صورت ذاتی، خود محاسبه به طور موازی شکل میگیرد و همچنین این راه در بسیاری از الگوریتم ها نیز کاربرد دارد. طراحی یک فرایند یا procedure موازی با شکستن مسئله ی داده شده به مسئله های کوچکتر(بلوک های کوچکتر) است که از یکدیگر مستقل هستند. در مسئله ی ما فرض کنید بلوک های زیر را داریم:

$$C = \begin{bmatrix} C_{11} & \cdots & C_{1N} \\ \vdots & \ddots & \vdots \\ C_{M1} & \cdots & C_{MN} \end{bmatrix}, \; A = \begin{bmatrix} A_{11} & \cdots & A_{1R} \\ \vdots & \ddots & \vdots \\ A_{M1} & \cdots & A_{MR} \end{bmatrix}, \; B = \begin{bmatrix} B_{11} & \cdots & B_{1N} \\ \vdots & \ddots & \vdots \\ B_{R1} & \cdots & B_{RN} \end{bmatrix},$$

$$m = m_1 M, \qquad r = r_1 R, \qquad n = n_1 N$$

که $Aij \in R^{m1\times r1}$ و $Bij \in R^{r1\times n1}$و $Cij \in R^{m1\times n1}$ هستند. پس نتیجه میگیریم که آپدیت شدن $C = C + AB$ می تواند به MN زیر مسئله ی کوچکتر تبدیل شود:

$$\text{Task}(i,j): \qquad C_{ij} = C_{ij} + \sum_{k=1}^{R} A_{ik}B_{kj}.$$

Load balancing و تقسیم کار بین پردازنده ها

توجه داشته باشید که حاصل ضرب های بلوک-بلوک $A_{ik}B_{kj}$ همگی سایز یکسانی دارند. یک برنامه موازی سازی، کار را به طور یکسان بین پردازنده ها تقسیم می کند. دو استراتژی تقسیم برای این مدل سازی به ذهن می آید. که block distribution of tasks و block-cyclic distribution of tasks نام دارند. مثالی از هریک در شکل های زیر آورده شده است که در هر یک، Proc(µ, T)، وظیفه ی به روز رسانی C_{ij} برای M: Prow :i = µ و N: Pcol :j = T را دارد. برای مثال زیر، هر کدام از استراتژی ها 12 تا Cij را به هر پردازنده می دهند و و هر آپدیت شامل R ضرب بلاک-بلاک است. از طرفی، میتوان گفت در هر دو استراتژی میزان محاسبه اختصاص داده شده به هر پردازنده یکسان است.

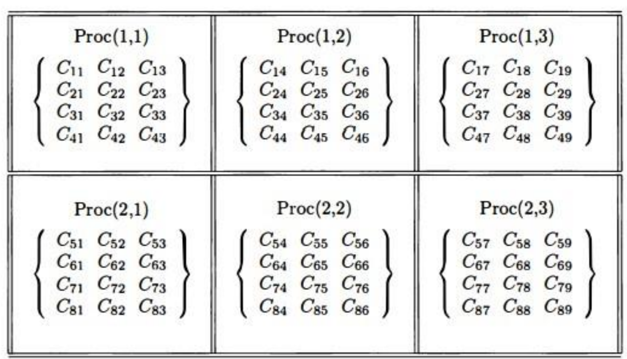


Figure 1.6.1. *The block distribution of tasks*
($M = 8$, $p_{\text{row}} = 2$, $N = 9$, and $p_{\text{col}} = 3$).

Proc(1,1)	Proc(1,2)	Proc(1,3)
$\begin{Bmatrix} C_{11} & C_{14} & C_{17} \\ C_{31} & C_{34} & C_{37} \\ C_{51} & C_{54} & C_{57} \\ C_{71} & C_{74} & C_{77} \end{Bmatrix}$	$\begin{Bmatrix} C_{12} & C_{15} & C_{18} \\ C_{32} & C_{35} & C_{38} \\ C_{52} & C_{55} & C_{58} \\ C_{72} & C_{75} & C_{78} \end{Bmatrix}$	$\begin{Bmatrix} C_{13} & C_{16} & C_{19} \\ C_{33} & C_{36} & C_{39} \\ C_{53} & C_{56} & C_{59} \\ C_{73} & C_{76} & C_{79} \end{Bmatrix}$
Proc(2,1)	**Proc(2,2)**	**Proc(2,3)**
$\begin{Bmatrix} C_{21} & C_{24} & C_{27} \\ C_{41} & C_{44} & C_{47} \\ C_{61} & C_{64} & C_{67} \\ C_{81} & C_{84} & C_{87} \end{Bmatrix}$	$\begin{Bmatrix} C_{22} & C_{25} & C_{28} \\ C_{42} & C_{45} & C_{48} \\ C_{62} & C_{65} & C_{68} \\ C_{82} & C_{85} & C_{88} \end{Bmatrix}$	$\begin{Bmatrix} C_{23} & C_{26} & C_{29} \\ C_{43} & C_{46} & C_{49} \\ C_{63} & C_{66} & C_{69} \\ C_{83} & C_{86} & C_{89} \end{Bmatrix}$

Figure 1.6.2. *The block-cyclic distribution of tasks* ($M = 8$, $p_{\text{row}} = 2$, $N = 9$, *and* $p_{\text{col}} = 3$).

اگر M مضربی از P_{row} نباشد یا Nمضربی از P_{col} نباشد، توزیع کار بین پردازنده ها دیگر متعادل نخواهد بود. در واقع ، اگر

$$M = \alpha_1 p_{\text{row}} + \beta_1, \qquad 0 \le \beta_1 < p_{\text{row}},$$
$$N = \alpha_2 p_{\text{col}} + \beta_2, \qquad 0 \le \beta_2 < p_{\text{col}},$$

آنگاه تعداد ضرب بلاک-بلاک برای هر پردازنده می تواند بین $\propto 1 \propto 2\ R$ تا $R(\propto 1 + 1)(\propto 2 +1)$ باشد. البته، این موضوع زمانی که محاسبات بزرگ در حال انجام است بی اهمیت خواهد بود. در واقع زمانی که $N >> P_{col}$ و $M >> P_{row}$ داریم:

$$\frac{(\alpha_1+1)(\alpha_2+1)R}{(\alpha_1\alpha_2)R} = 1 + O\left(\frac{p_{\text{row}}}{M} + \frac{p_{\text{col}}}{N}\right)$$

می توان نتیجه گرفت هر دو نوع استراتژی تقسیم کار بین پردازنده ها، برای به روز رسانی C+AB، load balance هستند. و می توانیم چنین استدلال کنیم که تخصیص کار بین پردازنده ها با افزایش اندازه مسئله، به طور فزاینده ای متعادل می شود.

حرکت داده و تاخیرهای مربوط به پیاده سازی معماری با چند پردازنده:

موضوع دیگری که باید مورد توجه قرار گیرد، اهمیت زمانی است که صرف هماهنگی پردازنده ها و حرکت داده در مدار است. زمانی که بحث محاسبه ی اطلاعات به کمک چند پردازنده می باشد، داده ای که یک پردازنده نیاز دارند ممکن است دور باشد و اگر این موضوعی باشد که زیاد اتفاق بیافتد به احتمال زیاد مزیت استفاده از چند پردازنده را از دست خواهیم داد. جدا از این ها، زمانی که هر پردازنده باید صبر کند تا محاسبات پردازنده ی دیگر تمام شود نیز زمان از دست رفته به حساب می آید.

همه ی این موضوعات، موجب میشود، مدل سازی عملکرد یا درواقع مدل سازی performance دشوار باشد، خصوصاً اینکه در صورت استفاده از تنها یک پردازنده، محاسبات و ارتباطات می توانند همزمان رخ دهند و مشکل های بالا پیش نمی آید.

پیاده سازی حافظه در این معماری:

فرض ما بر این است که ماتریس ها درون فایلی قرار دارند، بنابراین دو ماژول طراحی می‌شود:

وظیفه ماژول اول خواندن از فایل است و وظیفه ماژول دوم به شرح ذیل میباشد؛

در ابتدا باید توجه داشته باشیم که استاندارد IEEE ۱ بعدی است. از طرفی ماتریس هم دو بعد دارد، بنابراین طراحی باید سه بعدی باشد، اما از آنجاییکه سه بعدی تنها در سیستم وریلاگ امکان پذیر است و در وریلاگ نمیتوان طراحی کرد، مجبور می‌شویم به صورت دو بعدی طراحی کنیم بدینصورت که عرض (width) مموری را ۳۲ بیت در نظر گرفته و طول (height) آن را نیز به صورت پارامتری در اوردر چند هزار بیت تعریف میکنیم.

در هر خانه مموری یک عدد floating point قرار میگیرد.

در ادامه نحوه قرار گیری ماتریس ها را در مموری شرح می‌دهیم؛

مطابق خواست پروژه، خانه اول مموری، status قرار میگیرد. در واقع اندازه های دو ماتریس که فرض میشود یکی m*n و دیگری n*P میباشد، در آن قرار دارند.

حال از خانه دوم مموری به ترتیب سطر اول ماتریس را در مموری قرار می‌دهیم یعنی درایه اول از سطر اول در خانه دوم مموری، سپس درایه دوم از سطر دوم مموری در خانه سوم مموری و به همین شکل در مموری قرار می دهیم. سپس درایه‌های سطر دوم ماتریس را در ادامه مشابه سطر در مموری قرار می‌دهیم.

هنگامی که ماتریس اول به صورت کامل در مموری به روش فوق، قرار داده شد، یه گپ در حدود ۱۰-۲۰ خانه در مموری ایجاد کرده و سپس ماتریس دوم را مشابه ماتریس اول در مموری قرار می‌دهیم.

نکته حائز توجه این است که ما بسته به تعداد input address هایی که داریم، می توانیم همزمان به همان تعداد در مموری درایه های ماتریس را قرار دهیم به عنوان مثال اگر دو تا input address داشته باشیم میتوان دو تا دو تا، درایه های ماتریس ها را در هر عملیات در دو خانه مموری قرار داد.

در ادامه یک Write هم در کد ما قرار دارد برای مواقعی که نیاز به نوشتن در مموری داریم.

یک wrtite_enable که در واقعی flag ای میباشد برای اینکه قابلیت نوشتن فعال شود یا غیر فعال.

یک write_address که نشان میدهد در چه آدرسی از مموری باید دیتای مورد نظر نوشته شود.

و یک write_data که دیتای مورد نظر در آن قرار داشته و قرار است در مموری نوشته شود.

به عنوان مثال شکل زیر نجوه پر شدن مموری و قرار گیری دو ماتریس مورد نظر را در آن نشان می‌دهد.

ماتریس اول: A_{2*3} ماتریس دوم: B_{3*1}

status
a_{11}
a_{12}
a_{13}
a_{21}
a_{22}
a_{23}

b_{11}
b_{12}

b 13

پیاده سازی

ماتریس های A و B را در نظر بگیرید که هردو مربعی و n*n هستند و هدف، پیاده سازی ضرب آن هاست. گام های پیاده سازی به شرح زیر است:

- **گام اول:** هر کدام از ماتریس ها را به ماتریس های مربعی با ابعاد p تقسیم میکنیم که p همان تعداد پردازنده های در دسترس می باشد.
- **گام دوم:** یک ماتریس از پروسسور ها میسازیم که سایز آن $p^{1/2} \times p^{1/2}$ است. بنابراین، هر پروسه میتواند یک بلوک از ماتریس A و یک بلاک از ماتریس B را پوشش دهد.
- **گام سوم:** هر بلوک، به هریک از پردازنده ها فرستاده می شود، و زیربلوک های کپی شده درهم ضرب می شوند و نتیجه، به نتیجه ی نهایی و زیربلاک مرتبط آن (زیر بلوک مرتبط با آن ضرب ها در C) اضافه می شود.
- **گام چهارم:** زیر گراف های A، یک گام به چپ چرخانده می شوند و زیر بلوک های B یک گام به جهت بالا چرخانده می شوند.
- گام پنجم: گام های بالا به تعداد sqrt(p) 3 y 4 بار تکرار می شود.

Let us first assume that $M = N = R = p_{\text{row}} = p_{\text{col}} = 2$ and that the C, A, and B matrices are distributed as follows:

Proc(1,1)
C_{11}, A_{11}, B_{11}

Proc(1,2)
C_{12}, A_{12}, B_{12}

Proc(2,1)
C_{21}, A_{21}, B_{21}

Proc(2,2)
C_{22}, A_{22}, B_{22}

Proc(1,1)
Send a copy of A_{11} to Proc(1,2)
Receive a copy of A_{12} from Proc(1,2)
Send a copy of B_{11} to Proc(2,1)
Receive a copy of B_{21} from Proc(2,1)
$C_{11} = C_{11} + A_{11}B_{11} + A_{12}B_{21}$

Proc(1,2)
Send a copy of A_{12} to Proc(1,1)
Receive a copy of A_{11} from Proc(1,1)
Send a copy of B_{12} to Proc(2,2)
Receive a copy of B_{22} from Proc(2,2)
$C_{12} = C_{12} + A_{11}B_{12} + A_{12}B_{22}$

Proc(2,1)
Send a copy of A_{21} to Proc(2,2)
Receive a copy of A_{22} from Proc(2,2)
Send a copy of B_{21} to Proc(1,1)
Receive a copy of B_{11} from Proc(1,1)
$C_{21} = C_{21} + A_{21}B_{11} + A_{22}B_{21}$

Proc(2,2)
Send a copy of A_{22} to Proc(2,1)
Receive a copy of A_{21} from Proc(2,1)
Send a copy of B_{22} to Proc(1,2)
Receive a copy of B_{12} from Proc(1,2)
$C_{22} = C_{22} + A_{21}B_{12} + A_{22}B_{22}$

در نهایت، تست معماری ای که با ملاحظات بالا طراحی می شود به شرح زیر است:

هدف: تجزیه و تحلیل سرعت به دست آمده توسط الگوریتم و موازی سازی محاسبات انجام شده طبق الگوریتم استراسن زمانی که اندازه ی ورودی ها و تعداد پردازنده های معماری را افزارش میدهیم.

محدودیت ها:

- تعداد پردازنده های استفاده شده باید ریشه ی دوم داشته باشند
- باید قابلیت تقسیم داده ها به صورت متوازن بین پردازنده ها وجود داشته باشد(البته در غیر این، صورت می توان از نتیجه ی قسمت load balancing استفاده کرد)

به عنوان نکته آخر، می توان اشاره کرد که مزیت استفاده از این معماری همانطور که در ابتدای توضیحات گفته شد، flexibility آن می باشد.

۱۲-کد پایتون

```
import numpy as np
```

```
def split(matrix):
```

```
"""
Splits a given matrix into quarters.
Input: nxn matrix
Output: tuple containing 4 n/2 x n/2 matrices corresponding to a, b, c, d
"""
row, col = matrix.shape
row2, col2 = row//2, col//2
return matrix[:row2, :col2], matrix[:row2, col2:], matrix[row2:, :col2],
matrix[row2:, col2:]


def strassen(x, y):
"""
Computes matrix product by divide and conquer approach, recursively.
Input: nxn matrices x and y
Output: nxn matrix, product of x and y
"""


# Base case when size of matrices is 1x1
if len(x) == 1:
return x * y


# Splitting the matrices into quadrants. This will be done recursively
```

```
# untill the base case is reached.
a, b, c, d = split(x)
e, f, g, h = split(y)

# Computing the 7 products, recursively (p1, p2...p7)
p1 = strassen(a, f - h)
p2 = strassen(a + b, h)
p3 = strassen(c + d, e)
p4 = strassen(d, g - e)
p5 = strassen(a + d, e + h)
p6 = strassen(b - d, g + h)
p7 = strassen(a - c, e + f)

# Computing the values of the 4 quadrants of the final matrix c
c11 = p5 + p4 - p2 + p6
c12 = p1 + p2
c21 = p3 + p4
c22 = p1 + p5 - p3 - p7

# Combining the 4 quadrants into a single matrix by stacking horizontally
and vertically.
c = np.vstack((np.hstack((c11, c12)), np.hstack((c21, c22))))
```

```
return c
```

مراجع

1. *Matrix computations 4th*
2. *Writing for Science*
3. https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm
4. https://www.geeksforgeeks.org/strassens-matrix-multiplication/
5. https://www.geeksforgeeks.org/strassens-matrix-multiplication-algorithm-implementation/