



سیستم های قابل بازپیکربندی

دکتر صاحب الزمانی

شایان نقی زاده

402131043

گزارش پروژه

1. نرم افزاری

1.1 ایجاد شبکه و آموزش نرم افزاری

در ابتدا شبکه رو نرم افزاری با پیکربندی که در پروژه خواسته شده ایجاد می کنیم که در ادامه آمده است شبکه آموزش داده شده و وزن ها و پارامترهای مورد نیاز از آن به دست می آید و آن ها را ذخیره می کنیم.

```
[69]
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 3, kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28 * 28 * 3, 14)
        self.fc2 = nn.Linear(14, 12)
        self.fc3 = nn.Linear(12, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x

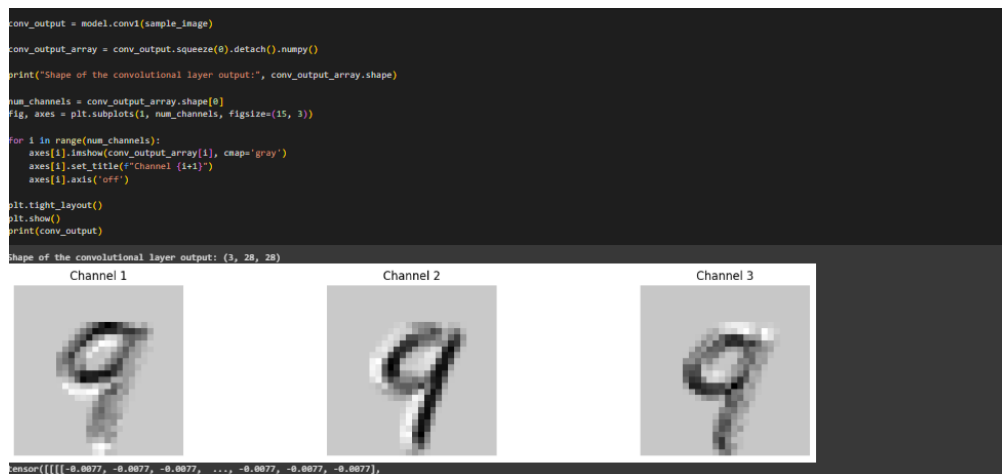
model = CNN()

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

معماری نرم افزاری و معماری پیشنهاد شده در شرح پروژه که شامل یک لایه کانولوشن 3 کاناله است که به ما ۸ در ۲۸ در ۲۸ مقدار می دهد که بعد از flatten شدن به لایه های کاملن متصل وارد می شود و جلو می رود در ادامه ۳ لایه کاملن متصل وجود دارد که به صورت 12 14 و لایه آخر که 10 نورون دارد است تابع فعال ساز relu استفاده شده است.

2.1 خروجی لایه های نرم افزاری

خروجی لایه کانولوشن



خروجی کانولوشن به لایه کاملن متصل اول که 14 نورون دارد می رود و خروجی زیر تولید می شود(این خروجی بدون relu است).

```
fc1_output = model.fc1(model.flatten(model.relu(model.conv1(sample_image))))
print("Outputs after the first fully connected layer (fc1):")
fc1_output
```

Outputs after the first fully connected layer (fc1):

tensor([[[47.7745, 86.1894, -97.4630, 46.6618, -141.3577, 54.7832,
54.1561, 114.1214, -66.5749, -57.7125, 60.5681, -138.2976,
106.8612, -30.4745]], grad_fn=<AddmmBackward0>])

خروجی لایه دوم که 12 نورون دارد

```
fc2_output = model.fc2(model.relu(model.fc1(model.flatten(model.relu(model.conv1(sample_image))))))
print("Outputs after the second fully connected layer (fc2):")
fc2_output
```

Outputs after the second fully connected layer (fc2):

tensor([[[-41.5766, -35.8263, 20.7333, 67.4531, -52.2863, -19.4050, 0.9667,
-10.4274, 6.4679, 14.4211, 39.6023, -60.3128]], grad_fn=<AddmmBackward0>])

[148] fc2_output.shape

torch.Size([1, 12])

خروجی 10 نورونی لایه آخر (این خروجی ها متناظر با عدد 9 در ورودی است)

```
[149] fc3_output = model.fc3(model.relu(model.fc2(model.relu(model.fc1(model.flatten(model.relu(model.conv1(sample_image))))))))
      print("Outputs after the third fully connected layer (fc3):")
      print(fc3_output)
```

Outputs after the third fully connected layer (fc3):
 tensor([[-8.2090, -8.3009, -11.2110, -0.3989, 11.1993, -3.1878, -23.1433, 4.8206, 5.1284, 23.0423]], grad_fn=<AddmmBackward0>)

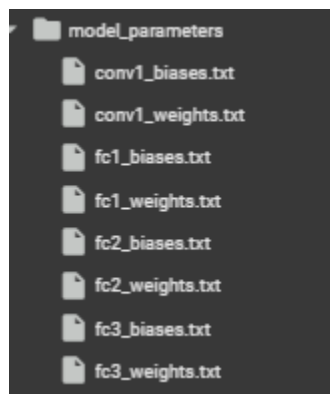
```
fc3_output = model.fc3(model.relu(model.fc2(model.relu(model.fc1(model.flatten(model.relu(model.conv1(sample_image))))))))
print("Outputs after the third fully connected layer (fc3):")
print(fc3_output)
```

Outputs after the third fully connected layer (fc3):
 tensor([[[-18.6617, -8.6383, -17.2400, 1.2680, 0.4183, -0.3688, -17.5510, 1.5488, -0.7065, 24.7345]], grad_fn=<AddmmBackward0>)

```
[64] fc3_output.shape
```

torch.Size([1, 10])

پارامترها رو که به صورت آرایه هستند به فرمت **cpp** تبدیل می کنیم و در **testbench** قرار می دهیم چون قرار است به عنوان ورودی به شبکه داده شود.



وزن های و پارامترهای به دست آمده با دقت خیلی بالایی در نرم افزار ذخیره می شوند که ما نیاز به آن نداریم پس برای بهبود پارامترها را فقط تا 5 رقم بعد از ممیز ذخیره می کنیم که نشان داده می شود دقت هنوز بالا 90 درصد یعنی 91 است. برای اینکار از تابع زیر استفاده می کنیم و تمام و پارامترها رو تبدیل می کنیم.

```

In [527]: def truncate_(tensor):
           return torch.floor(tensor * (2**5)) / (2**5)

In [528]: conv1_truncated = truncate_(truncated_model.conv1.weight)

In [580]: truncated_model.conv1.weight = torch.nn.Parameter(conv1_truncated)

In [530]: conv1_bias_truncated = truncate_(truncated_model.conv1.bias)

In [531]: truncated_model.conv1.bias = torch.nn.Parameter(conv1_bias_truncated)

In [532]: truncated_model.conv1.bias.tolist()
Out[532]: [-0.03125, -0.25, -0.03125]

In [533]: x = model.conv1.bias

In [534]: fc1_weight_truncated = truncate_(truncated_model.fc1.weight)

In [535]: truncated_model.fc1.weight = torch.nn.Parameter(fc1_weight_truncated)

In [536]: fc1_bias_truncated = truncate_(truncated_model.fc1.bias)

In [537]: truncated_model.fc1.bias = torch.nn.Parameter(fc1_bias_truncated)

In [538]: truncated_model.fc1.bias.tolist()
Out[538]: Show hidden output

In [539]: model.fc1.bias.tolist()
Out[539]: Show hidden output

```

```

In [540]: fc2_weight_truncated = truncate_(truncated_model.fc2.weight)

In [541]: truncated_model.fc2.weight = torch.nn.Parameter(fc2_weight_truncated)

In [542]: fc2_bias_truncated = truncate_(truncated_model.fc2.bias)

In [543]: truncated_model.fc2.bias = torch.nn.Parameter(fc2_bias_truncated)

In [544]: truncated_model.fc3.bias.tolist()
Out[544]: [-0.25,
            1.0625,
            -0.40625,
            0.71875,
            -0.21875,
            -0.09375,
            -1.53125,
            -0.09375,
            0.1875,
            -0.0625]

In [581]: model.fc3.bias.tolist()
Out[581]: [-0.24971696734428406,
            1.0750676393508911,
            -0.3813229501247406,
            0.739291787147522,
            -0.1910531371831894,
            -0.08148735761642456,
            -1.5209436416625977,
            -0.08819516748189926,
            0.20864646136760712,
            -0.058155059814453125]

```

```
[542] fc2_bias_trncated = truncate_(truncated_model.fc2.bias)

[543] truncated_model.fc2.bias = torch.nn.Parameter(fc2_bias_trncated)

[582] truncated_model.fc3.bias.tolist()

[-0.25,
 1.0625,
-0.40625,
 0.71875,
-0.21875,
-0.09375,
-1.53125,
-0.09375,
 0.1875,
-0.0625]

[581] model.fc3.bias.tolist()

[-0.24971696734428406,
 1.0750676393508911,
-0.3813229501247406,
 0.739291787147522,
-0.1910531371831894,
-0.08148735761642456,
-1.5209436416625977,
-0.08819516748189926,
 0.20864646136760712,
-0.058155059814453125]

[545] fc3_weight_trncated = truncate_(truncated_model.fc3.weight)

[546] truncated_model.fc3.weight = torch.nn.Parameter(fc3_weight_trncated)

[547] fc3_bias_trncated = truncate_(truncated_model.fc3.bias)

[548] truncated_model.fc3.bias = torch.nn.Parameter(fc3_bias_trncated)
```

بعد از ذخیره دقت را بررسی می کنیم

```
[583] evaluate_model(model, test_loader)

Test Accuracy: 96.90%

evaluate_model(truncated_model, test_loader)

Test Accuracy: 91.98%
```

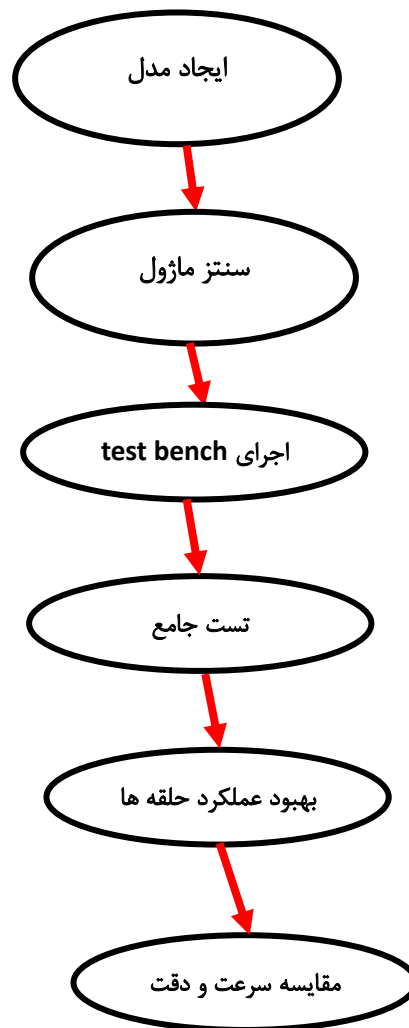
که کاهش دقت داشتیم ولی با این کار حجم حافظه مصرفی را بهبود دادیم و سرعت محاسبات خیلی بالاتر می شود.

مدل را با شبکه truncated شده با ورودی 9 تست می کنیم.

```
[569] truncated_model(sample_image)
→ tensor([[ -4.1825, -1.9946, -6.0270, -1.7381,  4.0342, -0.1354, -8.7225,  2.2501,
           1.2658,  6.9322]], grad_fn=<AddmmBackward0>)
```

حاصل عدد 9 است که د حقیقت همان ورودی اعتبارسنجی ما است.

2. سخت افزاری



2.1 ایجاد مدل

بعد از استخراج پارامترها به محیط hls می رویم و معماری شبکه را پیاده سازی می کنیم که در ادامه آمده است.

```
#include <hls_stream.h>
#include <ap_fixed.h>
typedef ap_fixed<16,10> fixed_type;

void cnn(fixed_type input[28][28],
         fixed_type kernel[3][3][3],
         fixed_type bias[3],
         fixed_type W1[14][2352],
         fixed_type b1[14],
         fixed_type W2[12][14],
         fixed_type b2[12],
         fixed_type W3[10][12],
         fixed_type b3[10],
         fixed_type output_fc3[10])
{
    fixed_type local_output_fc3[10];
    fixed_type local_input[30][30];

    for (int i = 0; i < 28; i++) {
        for (int j = 0; j < 28; j++) {
            local_input[i + 1][j + 1] = input[i][j];
        }
    }

    for (int i = 0; i < 30; i++) {
        for (int j = 0; j < 30; j++) {
            if (i == 0 || i == 29 || j == 0 || j == 29) {
                local_input[i][j] = 0;
            }
        }
    }

    //conv layer
    fixed_type output_conv[3][28][28];

    for (int c = 0; c < 3; c++) {
        fixed_type (*current_kernel)[3] = kernel[c];
        fixed_type current_bias = bias[c];
```

پیاده سازی ماژول سخت افزاری ما مشابه با معماری نرم افزاری با همان تعداد لایه و پارامترها است این ماژول به عنوان ورودی تمام پارامترهای شبکه از جمله وزن های لایه های مختلف و بایاس ها و همچنین 3 کرنل لایه کانولوشن را به عنوان ورودی می گیرد و همچنین یک تصویر که به صورت یک آرایه 28 در 28 است و خروجی ما احتمال بودن هر نورون یعنی هر عدد است که بیشترین مقدار به عنوان جواب در نظر گرفته می شود.

این پیاده سازی باعث می شود که حافظه تراشه ما یعنی BRAM و Distributed RAM ها اشغال نشوند ولی مشکل تاخیر ایجاد می کند چون تمام پارامترها باید از حافظه خارجی که برای تراشه ما نیست وارد شود ولی می توان وزن ها و پارامترها را در کد سخت افزاری قرار داد تا speed up بیشتری گرفت ولی منابع مصرفی زیاد خواهد شد.

کد سخت افزاری را سنتز می کنیم ولی باید توجه داشت که ما چون به سرعت بالا نیاز داریم بهتر است همانطور که توصیه شده از fixed point استفاده کنیم برای اینکار از fixed point به صورت 16,10 است یعنی 6 بیت عدد صحیح و 10 بیت برای fraction انتخاب کردیم تا سرعت و منابع برای استفاده بهتر از float شود.

```
typedef ap_fixed<16,10> fixed_type;
```

2.2 سنتز ماژول

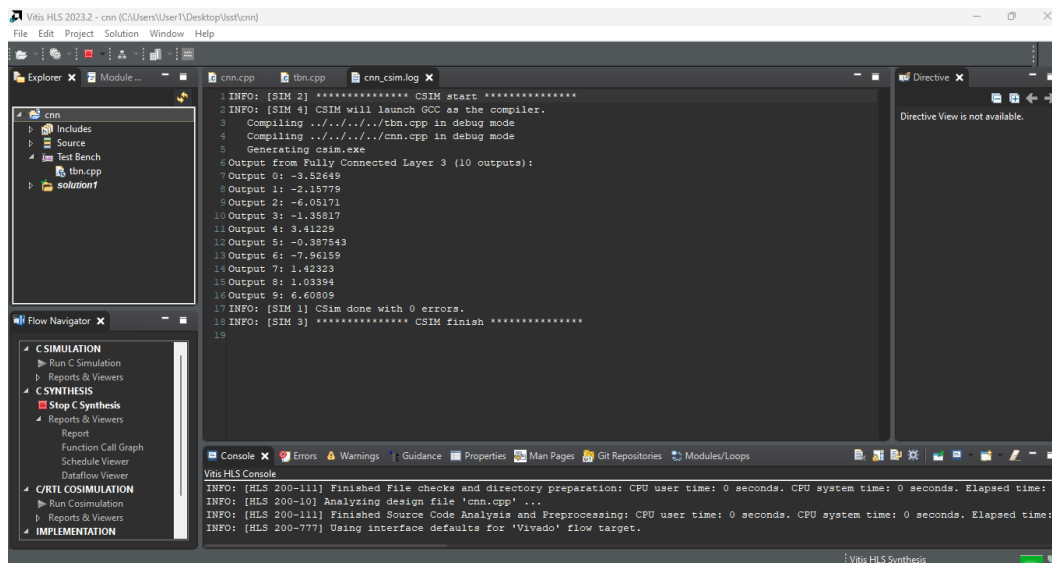
بعد از مشخص کردن معماری و data type کد را سنتز می کنیم تا فرکانس و منابع مصرفی را ببینیم

The screenshot shows the Vitis HLS 2023.2 interface. The main window displays the 'Performance & Resource Estimates' for the 'cnn' module. The table lists various modules and loops, including 'cnn_Pipeline_VITIS_LOOP_21_1_VITIS_LOOP_22_2', 'cnn_Pipeline_VITIS_LOOP_28_3_VITIS_LOOP_29_4', and 'cnn_Pipeline_VITIS_LOOP_63_10_VITIS_LOOP_64_11_VITIS_LOOP_65_12'. The table columns include 'Issue Type', 'Violation Type', 'Distance', 'Slack', 'Latency(cycles)', 'Latency(ns)', 'Iteration Latency', and 'Interval'. The console shows the synthesis results, including the estimated Fmax of 235.35 MHz and the total elapsed time of 36.083 seconds.

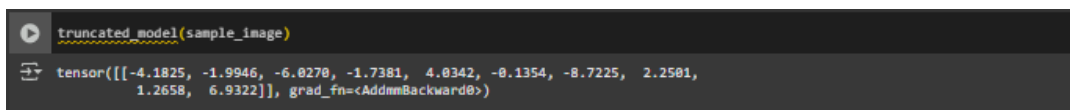
The screenshot shows the Vitis HLS 2023.2 interface. The main window displays the 'Performance & Resource Estimates' for the 'cnn' module. The table lists various modules and loops, including 'cnn_Pipeline_VITIS_LOOP_21_1_VITIS_LOOP_22_2', 'cnn_Pipeline_VITIS_LOOP_28_3_VITIS_LOOP_29_4', and 'cnn_Pipeline_VITIS_LOOP_63_10_VITIS_LOOP_64_11_VITIS_LOOP_65_12'. The table columns include 'Issue Type', 'Violation Type', 'Distance', 'Slack', 'Latency(cycles)', 'Latency(ns)', 'Iteration Latency', 'Interval', 'Trip Count', 'Pipelined', 'BRAM', 'DSP', 'FF', 'LUT', and 'URAM'. The console shows the synthesis results, including the estimated Fmax of 235.35 MHz and the total elapsed time of 36.083 seconds.

2.3 اجرای test bench

بعد از اینکه سنتز انجام شد حالا برای درست بودن یک تست پنج برای شبکه می توسیم و وزن ها و پارامترهای نرم افزاری به دست آمده را به عنوان ورودی به شبکه می دهیم همچنین ورودی ۲۸ در ۲۸ که داده MNIST ما است و برای اعتبارسنجی استفاده می کنیم.



تست پنج را شبیه سازی می کنیم و خروجی را مشاهده می کنیم که متناظر با بزرگترین عدد یعنی عدد ۹ است که مشخص کننده جواب پیشبینی شبکه ما است و خروجی همانند نرم افزاری Truncated شده است.



4.2 تست جامع

برای تست جامع 400 تصویر که به صورت 400 آرایه 28 در 28 و لیبل های متناظر است را در یک حلقه به عنوان ورودی با بقیه پارامترهای شبکه به ماژول می دهیم و مقایسه میکنیم بین حدس شبکه و لیبل های و تعداد درست ها را محاسبه می کنیم.

```
fixed_type inputs_test[28][28];
fixed_type output_fc3[10];
int counter=0;
fixed_type x;

for (int i= 0 ;i<400;i++){
    for (int j =0;j<28;j++){
        for (int k =0; k < 28; k++){
            inputs_test[j][k] = digits[i][j][k];
        }
    }
    cnn(inputs_test,kernel,bias_conv1,fc1_weight,fc1_bias,fc2_weights,fc2_bias,fc3_weights,fc3_bias,output_fc3);
    x = maximum(output_fc3);
    if (x == labels[i]){
        counter=counter+1;
    }
}
std::cout<<counter;
```

در تصویر زیر به همراه فایل پیوست یک آرایه 400 در 28 داریم که مشخص کننده 400 تصویر ورودی ما است و در انتهای و فایل آرایه متناظر با لیبیل ها است که در تصویر زیر مشخص است.

[illegible]

خروجی این حالت یعنی با تعداد تست بیشتر به شکل زیر است.

```
1 INFO: [SIM 2] ***** CSIM start *****
2 INFO: [SIM 4] CSIM will launch GCC as the compiler.
3     Compiling ../../../../cnn.cpp in debug mode
4     Generating csim.exe
5 364
6 INFO: [SIM 1] CSim done with 0 errors.
7 INFO: [SIM 3] ***** CSIM finish *****
8
```

که همیشه متوجه شد که از بین 400 تصویر ورودی 364 تا تصویر را درست تشخیص داده شده توسط پیاده سازی سخت افزاری ما که به ما دقت $91\% = \frac{364}{400}$ است که افت دقت نسبت به نرم افزاری برای این است که ما طول بیت کمتر یعنی 16 بیت نسبت به نرم افزاری داشتیم.

2.5 بهبود عملکرد حلقه ها

بعد به سمت بهینه سازی سخت افزاری با استفاده از `pragma` می رویم به نحوی که حلقه های کوچک را باز می کنیم تا به صورت موازی اجرا و جلو بروند در حقیقت `pragma unroll` برای باز کردن حلقه است تا به صورت ترکیبی جلو برویم و عملیات های که وابستگی ندارند را در صورت داشتن منابع روی تراشه به صورت موازی جلو ببریم که بهبود سرعت داشته باشیم ولی مشخصا منابع مصرفی افزایش می یابد به طور خلاصه این کار منابع مصرفی ما رو زیاد می کند ولی به جهت موازی سازی که انجام می شود سرعت زیاد می شود که در ادامه قابل مشاهده است.

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipeline
cn					35999	2.160E3	-	36000	-	-
cn_Pipeline_VITIS_LOOP_23_2				-	16	96.000	-	16	-	-
cn_Pipeline_VITIS_LOOP_23_21				-	16	96.000	-	16	-	-
cn_Pipeline_VITIS_LOOP_23_22				-	16	96.000	-	16	-	-
cn_Pipeline_VITIS_LOOP_23_23				-	16	96.000	-	16	-	-
cn_Pipeline_VITIS_LOOP_23_24				-	16	96.000	-	16	-	-
cn_Pipeline_VITIS_LOOP_23_25				-	16	96.000	-	16	-	-
cn_Pipeline_VITIS_LOOP_23_26				-	16	96.000	-	16	-	-
cn_Pipeline_VITIS_LOOP_23_27				-	16	96.000	-	16	-	-
cn_Pipeline_VITIS_LOOP_23_28				-	16	96.000	-	16	-	-
cn_Pipeline_VITIS_LOOP_23_29				-	16	96.000	-	16	-	-
cn_Pipeline_VITIS_LOOP_23_310				-	16	96.000	-	16	-	-
cn_Pipeline_VITIS_LOOP_23_311				-	16	96.000	-	16	-	-

Vitis HLS Console
INFO: [HLS 200-790] **** Loop Constraint Status: All loop constraints were satisfied.
INFO: [HLS 200-789] **** Estimated Pmax: 237.36 MHz
INFO: [HLS 200-111] Finished Command cnynth_design CPU user time: 52 seconds, CPU system time: 7 seconds, Elapsed time: 132.542 seconds;
INFO: [HLS 200-112] Total CPU user time: 53 seconds, Total CPU system time: 7 seconds, Total elapsed time: 134.813 seconds; peak allocate:
Finished C synthesis.

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipeline	BRAM	DSP	FF	LUT	URAM	
cn					35999	2.160E3	-	36000	-	-	no	28	284	30753	80065	0
cn_Pipeline_VITIS_LOOP_23_2				-	16	96.000	-	16	-	-	no	0	0	13	67	0
cn_Pipeline_VITIS_LOOP_23_21				-	16	96.000	-	16	-	-	no	0	0	13	93	0
cn_Pipeline_VITIS_LOOP_23_22				-	16	96.000	-	16	-	-	no	0	0	13	95	0
cn_Pipeline_VITIS_LOOP_23_23				-	16	96.000	-	16	-	-	no	0	0	13	95	0
cn_Pipeline_VITIS_LOOP_23_24				-	16	96.000	-	16	-	-	no	0	0	13	97	0
cn_Pipeline_VITIS_LOOP_23_25				-	16	96.000	-	16	-	-	no	0	0	13	97	0
cn_Pipeline_VITIS_LOOP_23_26				-	16	96.000	-	16	-	-	no	0	0	13	95	0
cn_Pipeline_VITIS_LOOP_23_27				-	16	96.000	-	16	-	-	no	0	0	13	67	0
cn_Pipeline_VITIS_LOOP_23_28				-	16	96.000	-	16	-	-	no	0	0	13	99	0
cn_Pipeline_VITIS_LOOP_23_29				-	16	96.000	-	16	-	-	no	0	0	13	99	0
cn_Pipeline_VITIS_LOOP_23_310				-	16	96.000	-	16	-	-	no	0	0	13	99	0
cn_Pipeline_VITIS_LOOP_23_311				-	16	96.000	-	16	-	-	no	0	0	13	99	0

Vitis HLS Console
INFO: [HLS 200-790] **** Loop Constraint Status: All loop constraints were satisfied.
INFO: [HLS 200-789] **** Estimated Pmax: 237.36 MHz
INFO: [HLS 200-111] Finished Command cnynth_design CPU user time: 52 seconds, CPU system time: 7 seconds, Elapsed time: 132.542 seconds;
INFO: [HLS 200-112] Total CPU user time: 53 seconds, Total CPU system time: 7 seconds, Total elapsed time: 134.813 seconds; peak allocate:
Finished C synthesis.

قابل مشاهده است که زمان نسبت به حالت بدون بهبود بهتر شده است ولی منابع مصرفی ما رشد چشم گیری داشته است از جهت این که موازی سازی زیادی انجام شده و بیشتر حلقه ها را باز کردیم و از آنجایی که برای نشان دادن این ویژگی از تراشه **virtex** استفاده شده منابع ما کم نیامده است ولی می توان از تراشه های با منابع کمتر استفاده کرده و مقدار کمتری موازی سازی نیز انجام داد که باز نسبت به حالت ترکیبی جلو رفتن بهتر است

در جدول مشخص است که به جهت باز کردن حلقه ها و انجام عملیات موازی ما تعداد **lut** ها و **dsp** ها رو افزایش دادیم این به این جهت است که تراشه چون نیاز به عملیات ضرب و جمع (MAC) دارد خیلی از این عملیات را به این **Hard core** ها تخصیص می دهد تا باز هم افزایش سرعت داشته باشد نسبت به اجرا با **lut** ها

مدل	CYCLE	TIME	LUT	DSP	FF	BRAM
سخت افزاری	40805	2.450E5	7830	128	6132	32
بهبود یافته سخت افزاری	35999	2.16.E5	80000	284	30753	28

2.6 مقایسه سرعت و دقت

در حالت نرم افزاری زمان اجرای با دقت پایین و بالا با هم تفاوتی زیادی ندارند چون عملیات به صورت ممیز شناور انجام می شود که با طول ثابت که بزرگتر از چیزی که ما نیاز داریم انجام می شود پس بهبودی در سرعت نخواهیم دید .

```

[590] start_time = time.time()
      with torch.no_grad():
          output = model(sample_image)
      end_time = time.time()
      inference_time = end_time - start_time

[591] inference_time
      0.0020737648010253906


[593] start_time = time.time()
      with torch.no_grad():
          output = truncated_model(sample_image)
      end_time = time.time()
      inference_time = end_time - start_time

[594] inference_time
      0.0024445056915283203

```

نرم افزاری	نرم افزاری با دقت کم	سخت افزاری	
0.0020737648010253906	0.0024445056915283203	0.000216	سرعت
96.90%	91.98%	91 %	دقت

ولی در اجرای سخت افزاری ما **Speedup ≈ 11.32** داریم به جهت انتخاب طول بیت کمتر برای محاسبات و انجام محاسبات با fixed point البته این مقایسه عادلانه نیست به جهت توان مصرفی هر کدام از این دو واحد محاسباتی ولی صرفاً به جهت مقایسه هر چند نادقیق انجام شده است.

Synthesis Summary Report of 'cnn'			
General Information			
Date:	Mon Jan 25 18:41:53 2010	Solution:	solution1 (Vivado IP Flow Target)
Version:	2023.2 (Build 4023990 on Oct 11 2023)	Product family:	virtexuplus
Project:	cnn	Target device:	xcvu11p-flga2577-1-e
Timing Estimate			
			
Target	Estimated	Uncertainty	
6.00 ns	4.213 ns	1.62 ns	

فرکانس 166.6MHz

فایل های نرم افزاری و سخت افزاری در پیوست قرار گرفته است.

باتشکر