Attention 大作业报告

袁一木 522030910149

1 摘要

注意力机制是当下在自然语言处理和大模型中最重要的部分之一。随着模型的体量不断增大，需要处理的数据越来越多，对于硬件层面的计算要求也越来越高，适应注意力机制的专用加速器也越来越受到关注和重视。本次大作业实现了一个简单的自注意力机制硬件层面仿真，实现了注意力机制中的注意力汇聚 K^TQ，简单量化操作，简化后的 softmax 操作以及最后的注意力分数计算 $A'V$，通过软件随机生成数据进行比对来验证硬件设计的正确性。为了提升硬件的计算速度，采用了简单的**折叠**机制，通过复用**脉动阵列**和**寄存器**来减少硬件开销，实现了简单的注意力机制模块。随后，利用 vivado 进行综合，分析关键路径并且进行了简单的优化。项目已经在 github 上发布：attention_verilog
(https://github.com/mumupika/attention_verilog)

2 软件设计

为了更好地验证我们的硬件计算和中间结果，在位层面上模拟真实的乘法，我们采用 **C++** 语言编写软件部分作为 golden model 与硬件进行对照，来验证硬件层面的数据正确性，并采用 clang+makefile 进行编译运行。

2.1 文件架构

我们的软件部分主要包含以下部分。其中各个文件主要实现了以下功能：

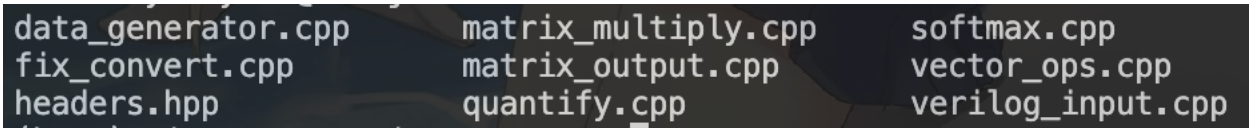
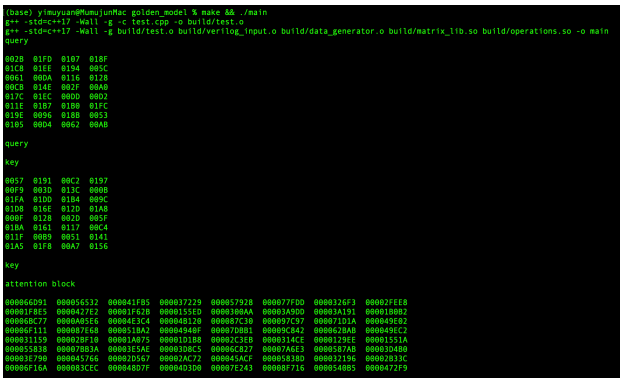


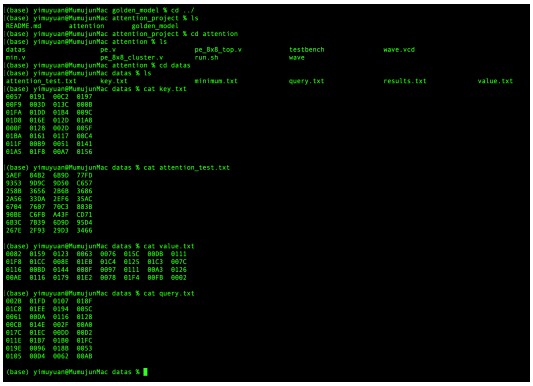
Figure 1: 软件部分含有的程序

- data_generator.cpp: 采用 C++ 17 STL 中随机数生成器实现了随机生成数据的功能。可以生成在一定范围内均匀分布的浮点数，最后返回一个随机生成的维度为 4×8 的矩阵。
- matrix_multiply.cpp: 实现了朴素矩阵乘法。其中主要有两种类型的矩阵乘法，一种返回的是浮点数，另一种返回的是该浮点数的对应长度的二进制表示。其中，为了验证矩阵二进制乘法软件编写的正确性，我们编写了采用基础浮点数乘法的矩阵乘法，与采用二进制乘法的矩阵乘法进行对比，来验证软件乘法的正确性。
- softmax.cpp: 实现了朴素的 softmax 算法。同 matrix_multiply.cpp 一样，也实现了基础计算库文件和二进制运算文件来验证后者的正确性。
- fix_convert.cpp: 浮点数与规定长度的二进制之间的相互转换文件。
- matrix_output.cpp: 继承标准 I/O 模块规范浮点数矩阵输出。
- vector_ops.cpp: 二进制的加减乘运算实现。在内部实现了对二进制加减乘三个运算的实现，来更好地验证硬件运算的正确性。
- headers.hpp: 用于构建文件系统的头文件。
- quantify.cpp: 用于量化操作，更好地验证硬件运算。
- verilog_input.cpp: 用于将数据转换成 verilog 仿真时能够读取的十六进制文件，和读取 verilog 仿真后输出的结果，并通过标准 I/O 显示在屏幕上。

通过简单的 `make && ./main` 即可运行得到我们的硬件输入数据和对比文件。这个程序会生成四个文件：`key.txt`，`value.txt`，`query.txt`,`attention_test.txt` 做好了这些，我们就可以准备进入硬件部分的设计了。



(a) 终端输出结果



(b) verilog 文件输入

3 硬件设计

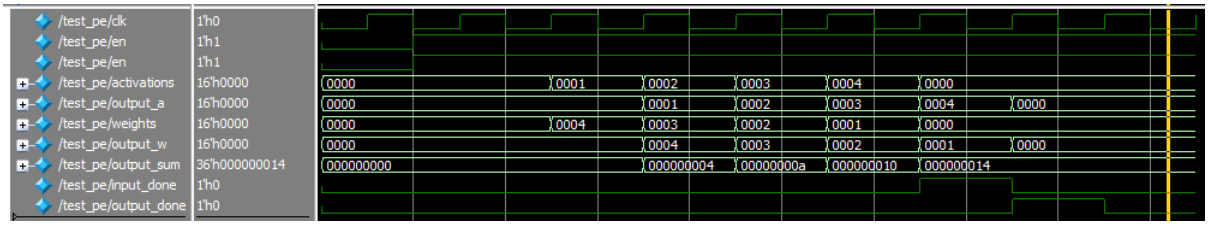
我们硬件的主要组成部分有：

- 单元设计部分：pe.v, min.v, pe_8x8_cluster.v, pe_8x8_top.v。
- 测试激励部分：min_tb.v, test_pe.v, test_pe_8x8.v, test_pe_8x8_top.v
- 数据部分：attention_test.txt, minimum.txt, results.txt, small_2.txt, key.txt, query.txt, small_1.txt, value.txt

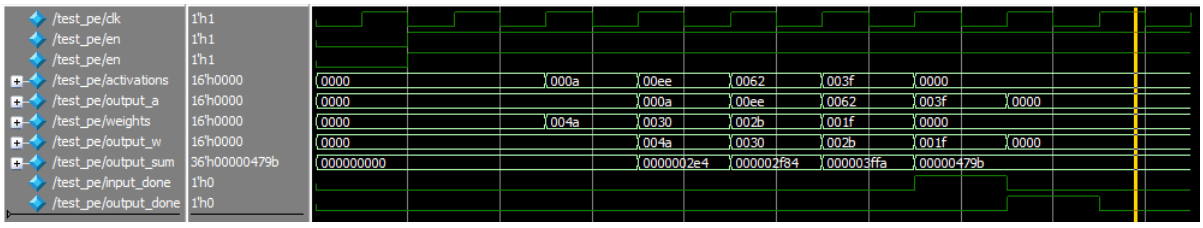
接下来逐个进行介绍。

3.1 pe 单元

在脉动阵列中，每个脉动阵列是通过一个乘累加单元组合而成的。乘累加单元是脉动阵列的最基础的组成部分。我们在 `pe.v` 中实现了基础的乘累加单元，通过 `small_1.txt, small_2.txt, test_pe.v` 进行了仿真验证。在该单元中，我们不断接受输入进来的激励和权重，并且进行乘累加计算。计算完成后，最后输出结果并提示计算完成。并且可以将我们的输入数据传出运算单元，为后面的 pe 单元设计最好准备。我们的仿真如下。可以看到我们基本完成了我们的期望功能。



(c) 乘累加单元仿真 1

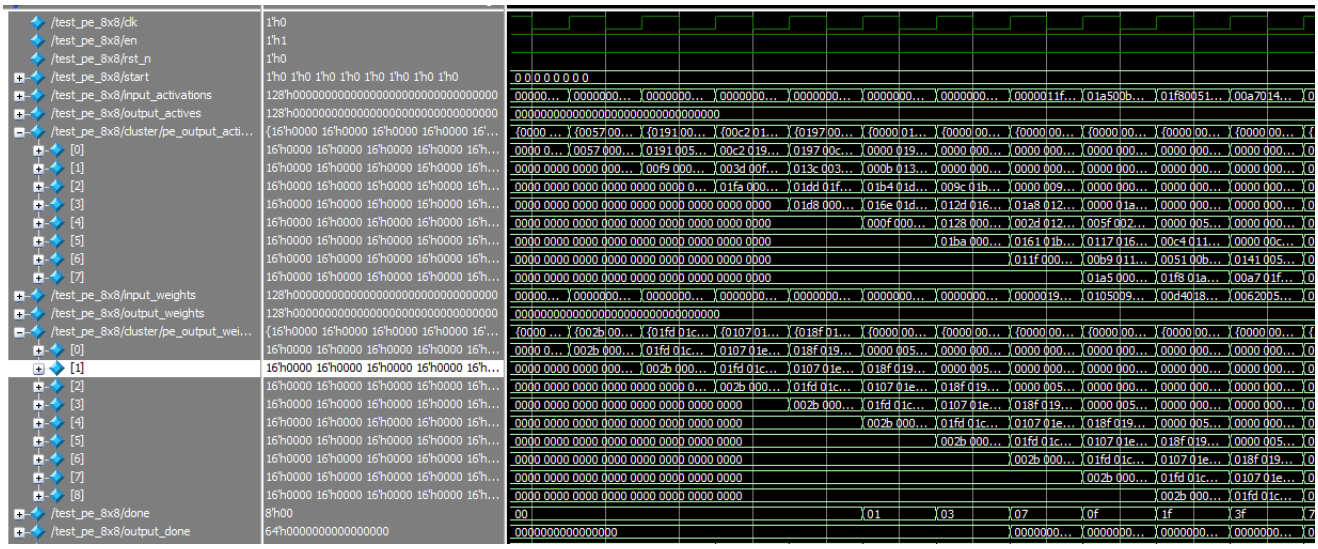


(d) 乘累加单元仿真 2

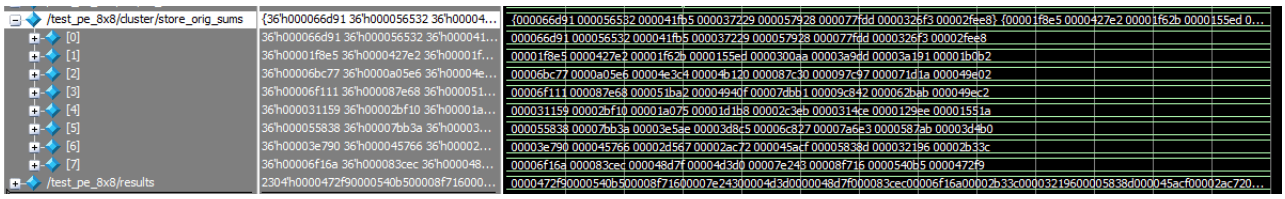
Figure 2: 乘累加单元仿真

3.2 8×8 乘累加集群（脉动阵列）

接下来需要将多个 pe 脉动阵列组装起来，形成能够计算我们任务要求的脉动阵列。这里我们选择形成 8×8 的脉动阵列，位于 `pe_8x8_cluster.v` 中。我们生成了 64 个 pe 单元实例，并通过线网和寄存器相互连接，并在集群内部采用了 64 个位宽为 36 位的结果寄存器来存储结果后输出。进行简单的测试如下，测试激励为 `key.txt, query.txt, test_pe_8x8.v`.



(a) 集群单元仿真 1



(b) 集群单元仿真 2

```
attention block

000066D91  000056532  000041FB5  000037229  000057928  000077FDD  0000326F3  00002FEE8
00001F8E5  0000427E2  00001F62B  0000155ED  0000300AA  00003A9DD  00003A191  00001B0B2
00006BC77  0000A05E6  00004E3C4  00004B120  000087C30  000097C97  000071D1A  000049E02
00006F111  000087E68  000051BA2  00004940F  00007DBB1  00009C842  000062BAB  000049EC2
000031159  00002BF10  00001A075  00001D1B8  00002C3EB  0000314CE  0000129EE  00001551A
000055838  00007BB3A  00003E5AE  00003D8C5  00006C827  00007A6E3  0000587AB  00003D4B0
00003E790  000045766  00002D567  00002AC72  000045ACF  00005838D  000032196  00002B33C
00006F16A  000083CEC  000048D7F  00004D3D0  00007E243  00008F716  0000540B5  0000472F9
```

(c) 真实结果对照

从上图二和上图三的结果对照来看，我们基本完成了一次矩阵相乘功能，输出的结果是正确的。该乘累加运算单元能够用于运算 $8 \times N$ 与 $N \times 8$ 的矩阵相乘，再经过至少 $N+7$ 个时钟周期进行计算，最终得到运算结果。从上图图一可以看到，矩阵运算的每一行/列按照流水线的方式流入，后一行比前一行延迟一个时钟周期。在输入结束后，会输入一个结束信号，来表示当前行已经输入完成的消息。该消息会进入乘累加单元，并在计算完成后输出，以告诉外界计算已经完成。

3.3 最小值选择器

接下来我们需要设计在 softmax 过程中需要使用的最小值选择器，从矩阵运算结果的一行中选择出最小的一个值。在这个最小值模块中，我们使用了 16 个寄存器来在至少 3 个时钟周期中选择出八个数字中最小的一个数字。我们的设计位于 `min.v` 中。通过 `min_tb.v, minimum.txt` 来验证其正确性。我们有如下的测试结果：

3.4 整体注意力控制模块

3.4.1 基本流程

最为复杂并且是整个注意力计算的核心位于该部分。我们将任务主要划分成了 6 个阶段：

- 计算 $K^T Q$ 部分进行注意力汇聚。
- 对计算结果进行量化。
- 寻找最小值并且进行 softmax 计算。
- 再次对计算结果进行量化。

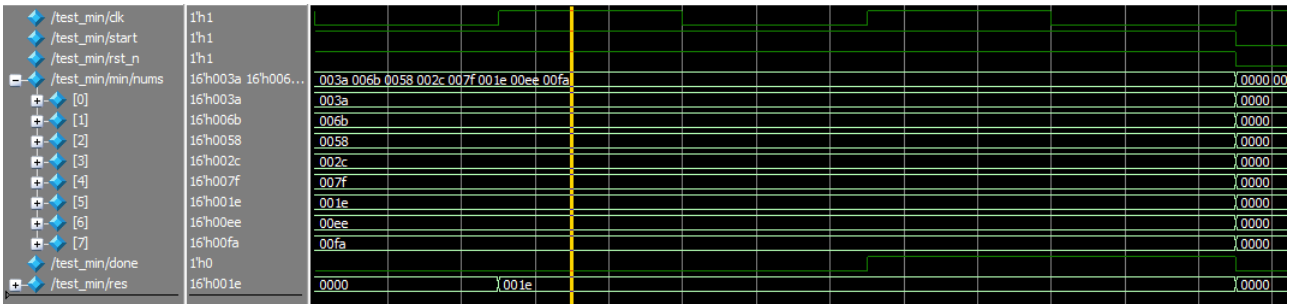
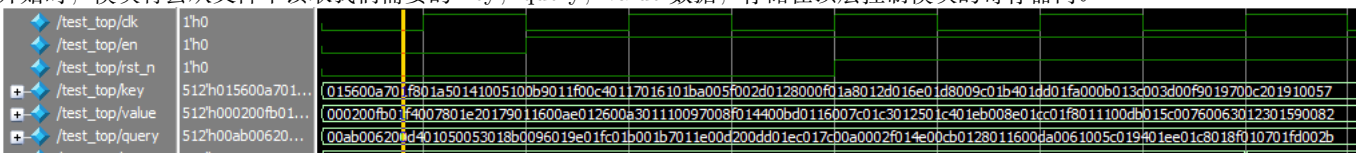


Figure 3: 最小值模块仿真

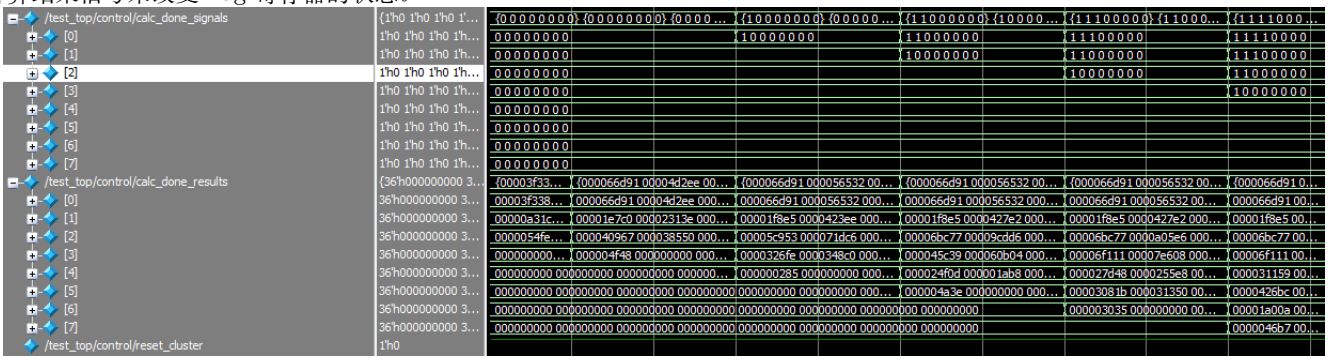
- 计算 VA' 注意力分数。
- 量化输出结果。

为了尽可能地加快速度，我们在前四个阶段和后两个阶段都分别采用流水线的方式进行了运算。在重新进行矩阵运算时，为了方式输入数据的紊乱和时序问题，我们并没有采用流水线方式从第四阶段转移到第五阶段。

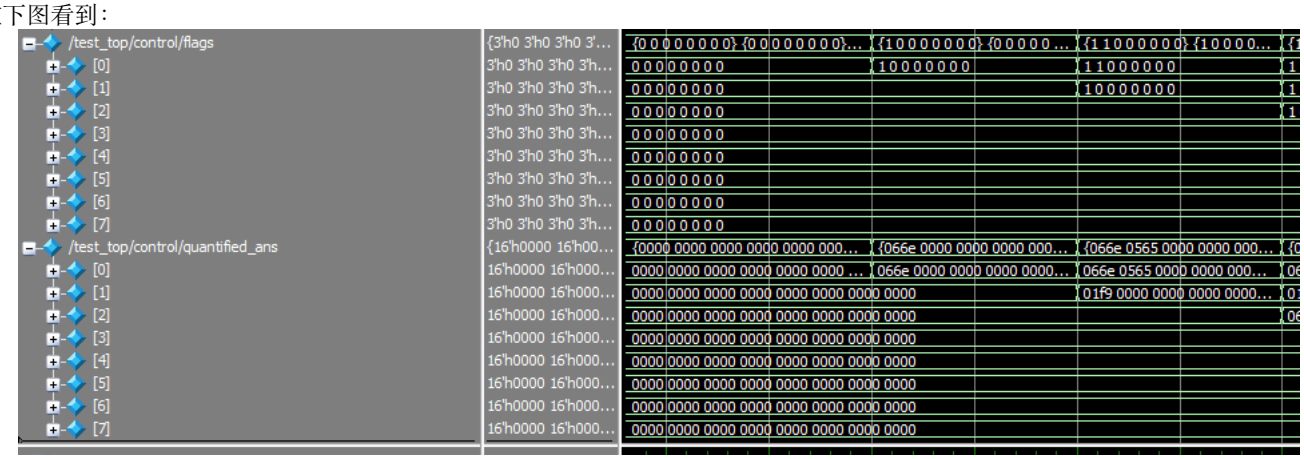
开始时，模块将会从文件中读取我们需要的 key，query，value 数据，存储在顶层控制模块的寄存器内。



接着，将数据传入 8x8 数据集群中进行计算。我们采用了一个 1bit 位宽的 attention_flag 来记录我们执行的是注意力汇聚阶段还是注意力分数计算阶段，采用 3 位宽的 8x8 的 flag 寄存器来判断每个计算单元所处于的阶段，并且通过每个计算单元输出的计算结束信号来改变 flag 寄存器的状态。



当读取到计算结束信号时，综合通过判断 flag 来进行判断是否需要进行量化。在该模块量化完成后，flag 才会更新。因此我们可以在下图看到：

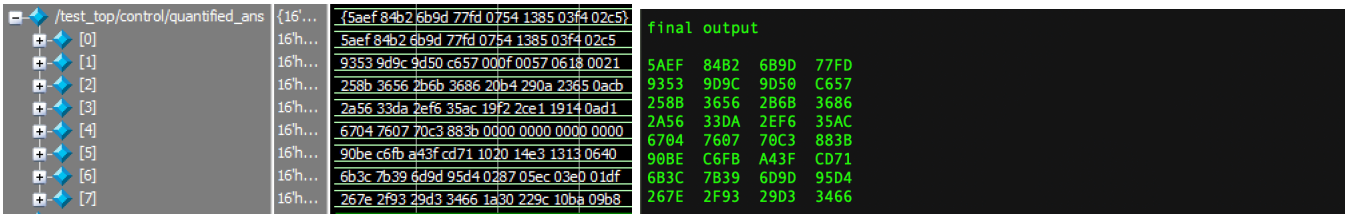


在完成了第一阶段的量化后，我们需要先通过最小值模块寻找到最小值，然后进行 softmax 运算。为了实现在每一列完成运算后立刻开始寻找该列的最小值，我们采用 7 个 127 位的寄存器作为最小值模块的输入，采用 7 个寄存器控制最小值模块进行运算。在得到该列最小值后，该列的 flag 更新，随后将其应用于该列的 softmax 运算，随后再次完成量化操作，然后等待直到最后一个计算单元完成上述的操作。将 attention_flag 更新，表示完成注意力的汇聚操作。

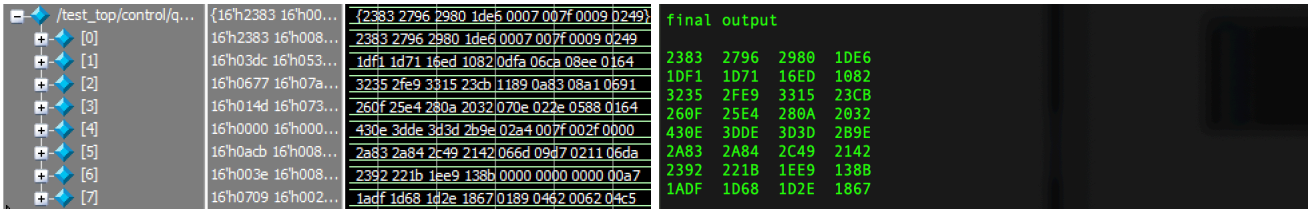
随后重复第一第二阶段的操作，不同的是此时 attention_map 为 1，因此我们的数据输入流将会与前面有所不同。其余的操作与上述操作没有太多的不同。至少在 54 个周期后我们将会完成我们的运算。

3.4.2 设计验证

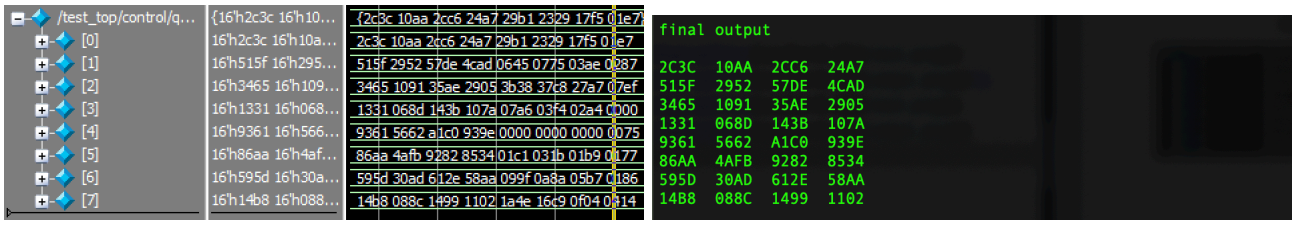
我们将会逐个验证我们的运算过程中的结果是否正确，根据我们的 golden_model 提供的结果。我们利用随机数生成机器，进行了三次验证。验证结果如下：



(a) 测试结果 1 (b) 真实结果 1



(c) 测试结果 2 (d) 真实结果 2



(e) 测试结果 3 (f) 真实结果 3

```
# All clear! Well done!
#
# ** Note: $stop    : Y:/Documents/codes/verilog/attention_project/attention/testbench/test_pe_8x8_top.v(76)
#    Time: 605 ns  Iteration: 1  Instance: /test_top
# Break in Module test_top at Y:/Documents/codes/verilog/attention_project/attention/testbench/test_pe_8x8_top.v line 76
```

(g) testbench

```
# Error! Error at 00000010, Ground truth = 6704, your results = 0038
#
# ** Note: $stop    : Y:/Documents/codes/verilog/attention_project/attention/testbench/test_pe_8x8_top.v(64)
#    Time: 595 ns  Iteration: 1  Instance: /test_top
```

(h) testbench 错误对照

Figure 4: 结果对照

我们编写了 testbench 可以将结果写入文件中 **result.txt**，并且可以对照输出结果，输出完全正确或者是计算错误的值。如上图的两个结果所示。

4 讨论

4.1 关键路径分析

我们首先对我们实现的硬件采用 vivado 进行综合。首先建立时钟约束。我们的时钟约束设定如下：

clock.xdc (C:/Mac/Home/Desktop/project_1/project_1.srcs/constrs_1/new/clock.xdc)

```
1  create_clock -period 10.000 -name clk1 -waveform {0.000 5.000} -add [get_ports clk]
2  set_input_delay -clock [get_clocks *] -add_delay 1.5 [get_ports -filter { NAME =~  "*" && DIR
3  set_output_delay -clock [get_clocks *] -add_delay 1.5 [get_ports -filter { NAME =~  "*" && DI
```

我们设定时钟的周期为 10ns。占空比 50%。输入延迟和输出延迟设定为 1.5ns。进行综合后，我们可以看到时序报告如下，符合时序要求。

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.011 ns	Worst Hold Slack (WHS):	0.061 ns	Worst Pulse Width Slack (WPWS):	4.650 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	43630	Total Number of Endpoints:	43630	Total Number of Endpoints:	12948

接下来我们来查看相关的关键路径。关键路径出现在从量化结果到 softmax 后得出的结果这一条路径上，原因是在进行 softmax 操作时，我们采用了两个减法器和一个乘法器进行操作，这样导致组合逻辑延时过长。我们有综合后的路径和延时如下：

Name	Path 1
Slack	3.011ns
Source	quantified_ans_reg[0][0][1]/C (rising edge-triggered cell FDRE clocked by clk1 {rise@0.000ns fall@5.000ns period=10.000ns})
Destination	softmax_res_reg[0][0]__0/PCIN[0] (rising edge-triggered cell DSP48E1 clocked by clk1 {rise@0.000ns fall@5.000ns period=10.000ns})
Path Group	clk1
Path Type	Setup (Max at Slow Process Corner)
Requirement	10.000ns (clk1 rise@10.000ns - clk1 rise@0.000ns)
Data P...Delay	5.622ns (logic 4.602ns (81.863%) route 1.020ns (18.137%))
Logic Levels	6 (CARRY4=4 DSP48E1=1 LUT2=1)
Clock ... Skew	-0.145ns
Clock U...tainty	0.035ns

显示时钟路由偏差公式给 -0.145ns

(a) 关键路径报告

```
    if(minimum_done[col] == 1) begin
        for(row = 0; row < 8; row = row + 1) begin
            softmax_res[row][col] <= (quantified_ans[row][col] - minimum_results[col]) * (quantified_ans[row][col] - minimum_results[col]);
            flags[row][col] <= flags[row][col] + 1;
            minimum_start[col] <= 0;
        end
    end
end
```

(b) 关键路径部分

因此我们可以在减法和乘法之间插入寄存器进行优化。进行优化后我们可以发现关键路径发生了转移。经过优化后我们有如下所示：

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.192 ns	Worst Hold Slack (WHS):	0.061 ns	Worst Pulse Width Slack (WPWS):	4.650 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	49134	Total Number of Endpoints:	49134	Total Number of Endpoints:	12948

(c) 关键路径报告

```
180                 if(minimum_done[col] == 1) begin
181                     for(row = 0; row < 8; row = row + 1) begin
182                         mid_res[row][col] <= quantified_ans[row][col] - minimum_results[col];
183                         softmax_res[row][col] <= mid_res[row][col] * mid_res[row][col];
184                         flags[row][col] <= flags[row][col] + 1;
185                         if(flags[row][col] == 3)
186                             minimum_start[col] <= 0;
187                     end
188                 end      You, 5天前 • Add simple control code.
189             end
```

(d) 关键路径部分

4.2 理论计算延迟和吞吐率

我们从输入结果到输出结果，根据仿真和计算可以得到，需要 53 个时钟周期，也就是 530ns（10ns 一个时钟周期）。接下来我们计算我们设计的吞吐率。根据计算我们发现，每次运算需要进行两次矩阵运算，总体 PE 运算次数为 384 次，每次进行注意力计算需要至少 530ns。因此我们有：

$$\text{throughput} = (4 \times 64 + 4 \times 32)/(530ns) = 724.53MFLOPS$$

。

根据 vivado 综合结果，

4.3 运用情况分析

Slice LUTs (303600)	Slice Registers (607200)	F7 Muxes (151800)	DSPs (2800)	Bonded IOB (600)	BUFGCTRL (32)
8392	12819	112	320	2052	1
2178	6080	0	64	0	0

Figure 5: 利用情况

根据上图的利用情况来看，我们的设计超出了其允许的最大线网数目，这一点可以通过再设计数据读写逻辑和顶层模块来进行控制。使用了 8392 个查找表（用于组合逻辑），12819 个寄存器，320 个数字信号处理器。

5 python 的 round 函数处理

我们可以将数值先扩大到需要舍入位数的前一位，进行 round() 操作后再进行还原（但是本项目采用的是 C++）。例如我们可以：

```
# print(round(1.3735,3))
print(round(1.3735*1000)/1000)
```

这样就可以得到正确的四舍五入值。