汇报：MLP 实现情况

黄宗乐　7/25

1　短暂回顾上次代码的问题

上次的代码中，我试图一次处理 16 个 batch 的数据，因此虽然使用了 DATAFLOW，overlap 的部分非常有限，整体的延迟还是比较大的。经过学姐的指导，我理解到 DATAFLOW 外面还有循环，应该通过循环处理各个 batch 的数据。学姐画的两张图反应了上述状况：

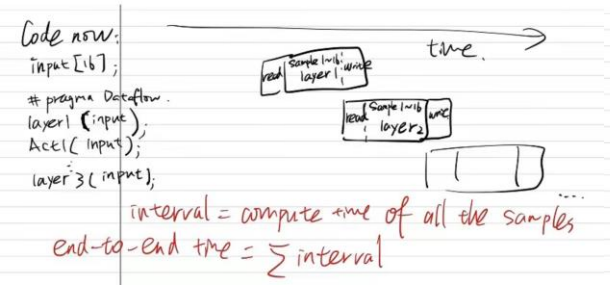

图 1：上次我的代码

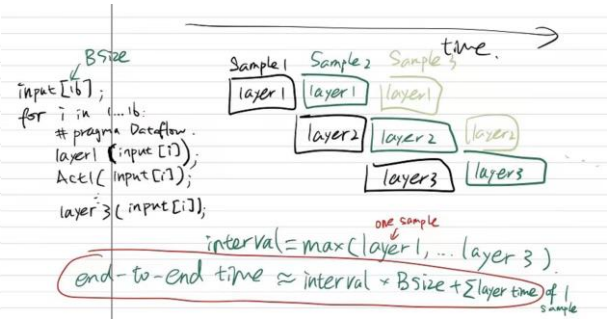

图 2：学姐的指导

下面的报告分为三个部分：

- Stream 实现：直接将 BSIZE 改成 top 中的循环，使整体结构成为图 2；
- 优化：在 Stream 实现的基础上做了一些优化，包括但不限于 unroll factor 的调整等；
- 其他问题：咨询其他可以优化的空间。

网络的结构为上次学姐指定的：

- Batch_size = 16
- Input_dimension = 2
- Hidden_layer_dimension = 128
- Output_dimension = 1

计算过程中的尺寸变化为：(16,2)×(2,128) → (16,128)×(128,1) →(16,1)

2　Stream 实现

在上次的代码中，各模块之间的 inpipe/outpipe 数据类型是 **hls::stream<blockvec>**，而 blockvec 本质上是一个长度为 batch_size 的数组。因此为了得到图 2 的结构，inpipe/outpipe 改成 **hls::stream<float>**，并且在外部进行 batch_size 次循环，计算矩阵乘法的循环也做了相应的调整。具体情况如下所示：

```
hls::stream<blockvec> inpipe;
hls::stream<blockvec> outpipe[6];

#pragma HLS DATAFLOW
loadIn(A, inpipe, L1);
blockmatmul(inpipe, w1bram, outpipe[0], L1,L2);
activation(outpipe[0], bias1, outpipe[1],L2);
blockmatmul2(outpipe[1], w2bram, outpipe[2],L2,L3);
activation(outpipe[2], bias2, outpipe[3],L3);
blockmatmul3(outpipe[3], w3bram, outpipe[4], L3,L4);
activation(outpipe[4], bias3, outpipe[5],L4);
storeDDR(C, outpipe[5], L4);
```

图 3：上次的 dataflow

```
hls::stream<float> inpipe;
hls::stream<float> outpipe[4];

for(int i = 0; i < BSIZE; i++){
#pragma HLS DATAFLOW
    loadIn(&input[i * L0], inpipe);
    matmul1(inpipe, weight1, outpipe[0]);
    activation(outpipe[0], bias1, L1, outpipe[1]);
    matmul2(outpipe[1], weight2, outpipe[2]);
    activation(outpipe[2], bias2, L2, outpipe[3]);
    storeDDR(outpipe[3], &output[i * L2]);
```

图 4：Stream 实现的 dataflow

综合之后，得到 vitis 的仿真结果如下：

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)
top		5365	1.788E4		5366		no	5	~0	87	1
dataflow_parent_loop_proc		5095	1.698E4		5095		no	0	0	87	1
dataflow_in_loop_VITIS_LOOP_121_5		383	1.277E3		314		dataflow	0	0	87	1
matmul1_1		313	1.043E3		313		no	0	0	80	1
Loop 1		128	427.000	1	1	128	yes				
stream_loop		52	173.000	26		2	no				
block_loop		22	73.326	16	1	8	yes				
write_out_VITIS_LOOP_42_1		128	427.000	2	1	128	yes				
matmul2		139	463.000		139		no	0	0	3	~0
stream_loop		137	457.000	11	1	128	yes				
activation		140	467.000		140		no	0	0	2	~0
VITIS_LOOP_84_1		138	460.000	12	1	128	yes				
activation_1		10	33.330		1		yes	0	0	2	~0
loadIn_124		75	250.000		75		no	0	0	0	0
loop_load		3	9.999	3	1	2	yes				
storeDDR_1		70	233.000		70		no	0	0	0	0
VITIS_LOOP_121_5		5094	1.698E4	5094		16	no				
VITIS_LOOP_111_1		268	893.000	134		2	no				
VITIS_LOOP_112_2		130	433.000	4	1	128	yes				

可以看到，图 4 中循环的 latency=5095，而每次循环的 interval = 314，latency = 383。已知 batch_size = 16，理论上总用时为 314 × 15 + 383 = 5093，和综合结果匹配。其中，interval 的瓶颈为模块 matmul1，其 latency = 313。因此下面的部分说明了我对上述模块进行的优化。

3 优化

观察上面的 report，matmul1 模块的主要问题不在于计算的循环（stream_loop），而在于初始化的循环（128 cycles）和 write_out 循环（128）。虽然使用 PIPELINE 之后能把 interval 减小到 1，但是由于数据量较大，整体还是不可接受的。

初始化的循环很容易解决了，只要显示使用 UNROLL 和 PIPELINE 指令即可（PIPELINE 内侧循环自动 UNROLL）：

```
float tmp[L1 / P1][P1] = {0};
#pragma HLS ARRAY_PARTITION variable=tmp dim=2 complete
```

图 5：Stream 的初始化

```
float tmp[L1 / P1][P1];
#pragma HLS ARRAY_PARTITION variable=tmp dim=2 complete

init:
for(int i = 0; i < L1 / P1;i++){
#pragma HLS PIPELINE
for(int ii = 0; ii < P1; ii++)
        tmp[i][ii] = 0;
}
```

图 6：优化之后的初始化

write_out 则不这么容易：由于 hls::stream<float>的 FIFO 特性，无法使用 UNROLL（强行使用会带来较大的 interval）。因此我重写了 inpipe/outpipe 的数据类型为 hls::stream<VecL1>等，其中 VecL0 表示某个 batch 特征维度为 L0 的向量。在计算时，使用 UNROLL 并行地给 VecL1 的变量赋值（下面没有显式写 UNROLL，根据 nested loop 规则，PIPELINE 内部循环自动 UNROLL），最后再将 VecL1 写入到 stream 中。这样的问题在于函数的通用性不是很好：各层传输的 pipe 的特征维度都不同，因此每个函数的定义都要重新写：

```
write_out:
for(int i = 0; i < L1 / P1; i++){
for(int ii = 0; ii < P1; ii++){
#pragma HLS PIPELINE
        outStream.write(tmp[i][ii]);
    }
}
```

图 7：Stream 的输出部分

```
VecL1 tmpC;
#pragma HLS ARRAY_PARTITION variable=tmpC.a cyclic factor=8 //P1

write_out:
for(int i = 0; i < L1 / P1; i++){
#pragma HLS PIPELINE
    for(int ii = 0; ii < P1; ii++){
        tmpC.a[i * P1 + ii] = tmp[i][ii];
    }
}
outStream.write(tmpC);
```

图 8：优化之后的输出部分

```
void loadIn(float* input, hls::stream<VecL0> &outStream);
void matmul1(hls::stream<VecL0> &inStream, VecL1* weight1, hls::stream<VecL1> &outStream);
void matmul2(hls::stream<VecL1> &inStream, VecL2* weight2, hls::stream<VecL2> &outStream);
void act1(hls::stream<VecL1> &inStream, const float* bias1, hls::stream<VecL1> &outStream);
void act2(hls::stream<VecL2> &inStream, const float* bias2, hls::stream<VecL2> &outStream);
void storeDDR(hls::stream<VecL2> &inStream, float* output);
void top(float* input, float* output);
```

图 9：优化之后的函数声明（虽然可以使用 template 将功能相似的函数融合在一起，但是 hls::stream 本身就是 template，保险起见还是分开写了。）

此外，修改了数据类型之后，读入 stream 也可以一次性完成，从而也减小了一点计算的开销：

```
stream_loop:
for (int k = 0; k < L0; k++){

    float tmpA = inStream.read();
    VecL1 tmpB = weight1[k];
    #pragma HLS aggregate variable=tmpB

    block_loop:
    for (int i = 0; i < L1 / P1; i++){
    #pragma HLS PIPELINE
    #pragma HLS dependence variable=tmp inter false
        ele_loop:
        for (int ii = 0; ii < P1; ii++){
    #pragma HLS UNROLL
            tmp[i][ii] += tmpA * tmpB.a[i * P1 + ii];
        }
    }
}
```

图 10：优化前：需要不断从读入 inStream 中读入

```
VecL1 tmpA = inStream.read();
#pragma HLS aggregate variable=tmpA

load_weight:
for (int k = 0; k < L1; k++){
//#pragma HLS UNROLL fator=L0

    VecL2 tmpB = weight2[k];
    #pragma HLS aggregate variable=tmpB

    block_loop:
    for (int i = 0; i < L2 / P2; i++){
    #pragma HLS PIPELINE
    #pragma HLS dependence variable=tmp inter false

        ele_loop:
        for (int ii = 0; ii < P2; ii++){
        #pragma HLS UNROLL
            tmp[i][ii] += tmpA.a[k] * tmpB.a[i * P2 + ii];
        }
    }
}
```

图 11：优化后：从 inStream 中读入一次即可

得到新的综合结果如下：

Name	Issue Type	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP	DSP (%)
top		2376	7.919E3		2377		no	5	~0	61	~0
dataflow_parent_loop_proc		2375	7.916E3		2375		no	1	~0	61	~0
dataflow_in_loop_VITIS_LOOP_192_5		243	810.000		142		dataflow	1	~0	61	~0
matmul1_1		93	310.000		93		no	1	~0	40	~0
init		16	53.328	1	1	16	yes				
load_weight_block_loop		47	157.000	17	1	32	yes				
write_out		16	53.328	2	1	16	yes				
act1		38	127.000		38		no	0	0	16	~0
VITIS_LOOP_126_1		28	93.324	14	1	16	yes				
matmul2		141	470.000		141		no	0	0	3	~0
load_weight		139	463.000	13	1	128	yes				
loadIn_145		78	260.000		78		no	0	0	0	0
VITIS_LOOP_14_1		7	23.331	3		2	no				
act2		10	33.330		10		no	0	0	2	~0
storeDDR_1		70	233.000		70		no	0	0	0	0
VITIS_LOOP_192_5		2374	7.913E3	2374		16	no				

可以看到此时 latency 已经从 5095 降到 2376，瓶颈成为 matmul2 这个模块。但是这个模块的 128 次循环是由 weight 读入限制的，因此我还没想到好的办法优化（其他问题中详细说明）。还值得注意的是，原本 matmul1 中的 stream_loop 中还有一个子循环，但是现在子循环已经被 flatten 到外层，成为一个单独的 load_weight_block_loop，这也是因为不需要在循环中 read stream, 所以原来的 imperfect loop 被优化了。UNROLL factor 选择了使得 matmul1 不成为瓶颈的最大值，从而减少资源占用。

4 其他问题

这一部分主要想讨论能不能进一步优化 matmul2 的延时。和 matmul1 不同，这部分延时主要是计算时的最外层循环带来的（如下图）。Weight2 的输入形状是(128,1)，为了不重复访问外部 weight，需要在最外层循环中载入一行数据（此处第二维度为 1，所以优化不明显，matmul1 中第二维度为 128，就很有意义了）。

但是这就意味着如果 weight2 有 128 行，则最外层循环就有 128 次。我试图通过 UNROLL 的方式来并行操作，但是对于这种 nested loop，内侧循环的 PIPELINE 会直接让外侧也 PIPELINE，哪怕对最外层加上 UNROLL 也没有用。

```
load_weight:
for (int k = 0; k < L1; k++){
//#pragma HLS UNROLL fator=L0

    VecL2 tmpB = weight2[k];
    #pragma HLS aggregate variable=tmpB

    block_loop:
    for (int i = 0; i < L2 / P2; i++){
    #pragma HLS PIPELINE
    #pragma HLS dependence variable=tmp inter false

        ele_loop:
        for (int ii = 0; ii < P2; ii++){
        #pragma HLS UNROLL
            tmp[i][ii] += tmpA.a[k] * tmpB.a[i * P2 + ii];
        }
    }
}
```

我还想过把最外层循环拆成两个循环（外层为 k，内层为 kk，weight index＝k*P+kk），然后将内存循环放入 PIPELINE 内部（上图中 block_loop 以内），根据 nested loop 的规则，可以达到 unroll factor=P。但是这样就要重复 load weight，感觉开销更大。

因此这一部分还没有想到比较好的优化方法，不知道学姐有没有什么建议？