0.1. Applied strategies

- Loop unrolling: unroll 2 inner loop entirely and 32 for the loop of w in `cnn`

```
for (int i1 = 0; i1 < 16; i1++) {
  for (int h = 0; h < 16 * 14; h++) { // Note: 16*14 equals 224 (the
    for (int w = 0; w < 224; w++) {
      #pragma HLS unroll factor = 32
      for (int p = 0; p < 5; p++) {
        #pragma HLS unroll
        for (int q = 0; q < 5; q++) {
          #pragma HLS unroll

          /**
           * Compute the effective output channel index.
           * i0 selects the outer tile and i1 selects the sub-channel
           * within the tile.
           */
          int i = i0 * 16 + i1;
          output[i1][h][w] +=
              weight[i1][j][p][q] * input[0][h + p][w + q];
        }
      }
    }
```

- Array partition:
  - since I unroll p and q entirely, I can partition dimension 3-4 of weight completely to match 1-1
  - Moreover, since w is unrolled using a factor of 32, I partition it in 32 to match the unrolling

```
#pragma HLS ARRAY_PARTITION variable = input cyclic factor = 1 dim = 1
#pragma HLS ARRAY_PARTITION variable = input cyclic factor = 1 dim = 2
#pragma HLS ARRAY_PARTITION variable = input cyclic factor = 32 dim = 3

#pragma HLS ARRAY_PARTITION variable = output cyclic factor = 1 dim = 1
#pragma HLS ARRAY_PARTITION variable = output cyclic factor = 1 dim = 2
#pragma HLS ARRAY_PARTITION variable = output cyclic factor = 32 dim = 3

#pragma HLS ARRAY_PARTITION variable = weight cyclic factor = 1 dim = 1
#pragma HLS ARRAY_PARTITION variable = weight cyclic factor = 32 dim = 2
#pragma HLS ARRAY_PARTITION variable = weight complete dim = 3
#pragma HLS ARRAY_PARTITION variable = weight complete dim = 4
```

- Pipelining: the compiler automatically added

0.2. Applied techniques

- Unrolling two most inner loop of kernel matrix, which creates 25 parallel multiply-accumulate units per processing element → throughput increases.

```
for (int p = 0; p < 5; p++) {
  #pragma HLS unroll
  for (int q = 0; q < 5; q++) {
    #pragma HLS unroll
```

- Unrolling the next outer loop by a factor of 32 (which is divisible by 224). This enables 32 simultaneous memory accesses → thus throughput increases.

```
for (int w = 0; w < 224; w++) {
  #pragma HLS unroll factor = 32
  for (int p = 0; p < 5; p++) {
```

- Array Partitioning:
  - partition dim 3,4 of weight matrix completely, so that it matches the unrolled factor above
  - set the cyclic factor for input, output, and weight to match the unrolled factor 32 above
  - This techniques allow array access simultaneously, so each parallel operation accesses a different memory bank. Thus, it increases performance

```
#pragma HLS ARRAY_PARTITION variable = input cyclic factor = 1 dim = 1
#pragma HLS ARRAY_PARTITION variable = input cyclic factor = 1 dim = 2
#pragma HLS ARRAY_PARTITION variable = input cyclic factor = 32 dim = 3

#pragma HLS ARRAY_PARTITION variable = output cyclic factor = 1 dim = 1
#pragma HLS ARRAY_PARTITION variable = output cyclic factor = 1 dim = 2
#pragma HLS ARRAY_PARTITION variable = output cyclic factor = 32 dim = 3

#pragma HLS ARRAY_PARTITION variable = weight cyclic factor = 1 dim = 1
#pragma HLS ARRAY_PARTITION variable = weight cyclic factor = 32 dim = 2
#pragma HLS ARRAY_PARTITION variable = weight complete dim = 3
#pragma HLS ARRAY_PARTITION variable = weight complete dim = 4
```

0.3. Difference between lab 3 and lab 4

In Lab 3, our GPU approach leveraged SIMT parallelism and shared-memory tiling. Each thread handled one output pixel using threads within a warp. We used 16×16 thread blocks and explicit shared memory tiles to optimize global memory access dynamically.

On the other hand, Lab 4 on the FPGA employed spatial parallelism through HLS. Instead of thread loops, we created deeply pipelined hardware. Loop are unrolled and pipelined to allow executing code in parallel. To avoid memory bank conflicts, buffers are partitioned into parallel BRAM banks accordingly

The core difference lies in their execution models: GPUs dynamically schedule thousands of threads, using warps to hide memory latency. In contrast, the FPGA design statically allocates dedicated DSPs and BRAMs to form fine-grained hardware pipelines. This FPGA approach results in deterministic, low latency and power-efficient throughput but demands explicit programmer control through directives for loop unrolling, pipelining, and memory partitioning.

0.4. Resources usages

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2	-
FIFO	-	-	-	-	-
Instance	40	4004	450109	243810	0
Memory	2160	-	0	0	25
Multiplexer	-	-	-	104	-
Register	-	-	199	-	-
Total	2200	4004	450308	243916	25
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	152	175	57	61	7
Available	4320	6840	2364480	1182240	960
Utilization (%)	50	58	19	20	2

- DSP is used the most as of 58%

0.5. Bonus

```
for (int w = 0; w < 224; w++) {
  #pragma HLS unroll factor = 32
  for (int p = 0; p < 5; p++) {
    #pragma HLS unroll
    for (int q = 0; q < 5; q++) {
      #pragma HLS unroll

      /**
       * Compute the effective output channel index.
       * i0 selects the outer tile and i1 selects the sub-channel
       * within the tile.
       */
      output[i1][h][w] +=
          weight[i1][j][p][q] * input[0][h + p][w + q];
    }
  }
```

- the output operation takes $2 + 3 = 5$ DSPs
- We unroll p,q fully, and w by factor of 32, then DSP = $5 * 5 * 5 * 32 = 4000$ which is close to the real result 4004
- `float input[1][228][228];` $\rightarrow (228 * 228)$ elements
- `float output[16][224][224];` $\rightarrow (16 * 224 * 224)$ elements
- `float weight[16][256][5][5];` $\rightarrow (16 * 256 * 5 * 5)$ elements
- total memory = $[(228 * 228) + (16 * 224 * 224) + (16 * 256 * 5 * 5)] \times 4B/(18 * 1024/8)B$ = = 1661 BRAM_18K

The reason why there is a great difference between computed (1661) vs actual (2200) is due computation overhead for simultaneous operations, or padding for data alignment or avoid conflict