High-Level Synthesis Lab – CNN Inference Accelerator

FPGA Ignite 2023 Summer School

31 July – 4 August, 2023, Heidelberg University, Heidelberg, Germany

Jason Anderson

1 Introduction

This lab will introduce you to high-level synthesis (HLS) concepts using the LegUp HLS tool. You will apply HLS to synthesize a hardware implementation of a key machine learning computation from software written in the `C` programming language. Specifically, you will synthesize a circuit that performs a convolution operation, as in a convolutional neural network (CNN). By changing the HLS code and constraints, you will generate several circuits of the circuit, with successively better speed performance.

Figure 1 illustrates the computation you will implement in hardware in this lab via high-level synthesis. In the figure, there are three input feature maps (on left), and six filters (in the middle). Each filter has a depth of three (same as the number of input feature maps). The computational work is to compute the output feature maps (on the right). Each output feature map is produced by convolving a filter with the input feature maps. That is, the number of filters is equal to the number of output feature maps. To compute an output feature map, a filter is "swept across" the input feature maps, and at each position, the weights of the filter are multiplied by the input feature map weights. These products are accumulated to become a value in the output feature map at a particular position. In an actual CNN, a bias is usually added to the sum-of-products, and then a non-linear function is applied, however, these steps are insignificant to the main computational work, so we ignore them in this lab.

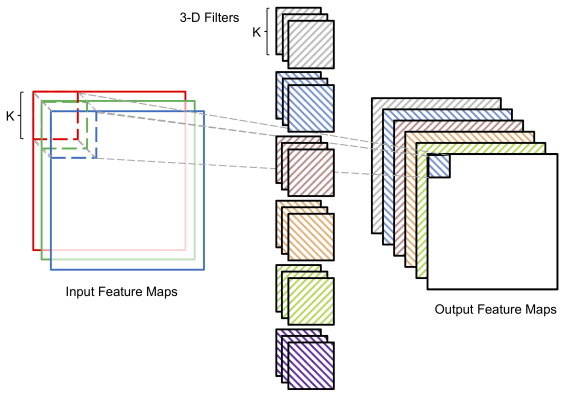

Figure 1: Convolution operation.

2 Initial Implementation

In this section, we will compile the CNN layer to hardware, without any modifications to the `C` code.

This labs assumes the installation instructions have been executed on the computer you are using and a Docker image and container have been created. Start up the Docker container and attach to it. To do this, you can find the list of containers using:

```
docker ps -a
```

Once you find the container ID, you can attach to it:

```
docker start <container ID>
docker attach <container ID>
```

Enter the part 1 directory:

```
cd /tmp/legup/legup-master/examples/ignite
cd part1
```

For part 1, use a text editor (`vi` or `emacs`) to open `cnn.c` and browse through the code. In the `convolution` function, you should see six nested loops that walk over the output feature maps, then rows and columns of the output maps. For each row, column location of an output feature map, we convolve the corresponding filter across the input feature maps (three inner-most loops). Note that the inner-most loop is across the input feature maps. The `main` function here simply calls the `convolution` function, and performs error checking, which compares the computed image against the golden output.

Open `cnn.h` to see that for this lab, we have 16 input feature maps, 8 filters/output feature maps, the filter dimensions are 3×3, and the input feature map dimensions are 30×30, to produce output feaure maps that are 28×28.

Before compiling to hardware, verify that the `C` program is correct by compiling and running the software. This is typical of HLS design, where the designer will verify that the design is functionally correct in software before compiling it to hardware. Type:

```
gcc cnn.c -o cnn
./cnn
```

You should see the message *PASS* appear.

Once you are familiar with the `C` code and its behaviour, compile the convolution program into hardware using LegUp by typing:

```
make
```


This command tells LegUp to compile the entire C program into hardware. Several report files and a Verilog file called cnn.v will be generated. You will see the LegUp HLS tool's output. By default, we have set the target device to be the Altera (Intel) Cyclone V FPGA, and set the target clock frequency to 100 MHz. Open and scroll through cnn.v. You will notice that the HLS-generated Verilog file appears to be very difficult to follow and debug. This is another reason we want to first verify the design in software before compiling to hardware.

Near the bottom of cnn.v, you will find a Verilog module called main_tb. This is the testbench used for the simulation of the main Verilog module, top, that implements the convolution circuit. main_tb simulates the top module with a clock, a start and a reset signal, then it waits for a finish signal from top to signify the completion of the convolution work. Navigate to around line 260 to find the FSM that LegUp generated in order to control the state of execution in the hardware circuit. This is the main structure that enables the circuit to honour the data dependencies and control flow from the sequential software program.

As you peruse the Verilog produced by LegUp, notice that the LLVM instructions that correspond to the RTL have been annotated into the Verilog as comments. This is helpful for finding the correspondence between the input C code, the LLVM, and the RTL.

You should also open up the file scheduling.legup.rpt to view how LegUp HLS has scheduled the operations into FSM states. You can see the LLVM instructions and the state in which they are scheduled. For example, here's a state in the main function, where you can see a load and integer compare have been scheduled:

```
state: LEGUP_F_main_BB__0_3
   ...
   %3 = load i32* @total, align 4, !MSB !3, !LSB !2, !extendFrom !3 (endState: LEGUP_F_main_BB__0_3)
   %4 = icmp eq i32 %3, -18870, !MSB !2, !LSB !2, !extendFrom !2 (endState: LEGUP_F_main_BB__0_3)
   br i1 %4, label %5, label %7, !MSB !1, !LSB !2, !extendFrom !1
   Transition: if (%4): LEGUP_F_main_BB__5_4 default: LEGUP_F_main_BB__7_5
```

Now let's simulate the Verilog RTL hardware with ModelSim to find out the actual number of cycles needed – the *cycle latency*. To do this, type:

```
make v
```

It may take a few minutes to simulate. We want to focus on the message near the end which appears something like this:

```
...
# TOTAL:      -18870
# PASS
# At t=         60722510000 clk=1 finish=1 return_val=         0
# Cycles:              3036123
# ** Note: $finish    : cnn.v(1800)
#    Time: 60722510 ns  Iteration: 4  Instance: /main_tb
```

We see that the simulation took 3,036,123 cycles. Also observe that simulation printed "PASS", which is the same message we got when the software version passed the built-in error-checking functionality. This means that the LegUp generated hardware produced the same results as the software version.

At the end of this handout, you will find a blank table. Fill in the number of cycles in this part of the lab under the column labelled "Basic".

The simulation above is called a functional simulation since it simulates the logic without mapping it to the Altera FPGA. After synthesizing the Verilog onto the Altera Cyclone V FPGA, we obtain information such as the resource usage and the Fmax of this design (i.e. the clock period). It will take too long to do this for this lab, so I pasted some sample results below.

The design used 336 Adaptive Logic Modules (ALMs), 656 Registers, 1 DSP block, and 18 RAM blocks.

```
Family : Cyclone V
Device : 5CSEMA5F31C6
Timing Models : Final
```



```
Logic utilization (in ALMs) : 336 / 32,070 ( 1 % )
Total registers : 656
Total pins : 0 / 457 ( 0 % )
Total virtual pins : 36
Total block memory bits : 124,416 / 4,065,280 ( 3 % )
Total RAM Blocks : 18 / 397 ( 5 % )
Total DSP Blocks : 1 / 87 ( 1 % )
```

You can also view the speed performance of the design, after its complete implementation on the Cyclone V, in the same report, and shown below. This circuit can operate at about 146MHz on the Cyclone V.

```
+-----------------------------------------------------+
; Slow 1100mV 85C Model Fmax Summary                  ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 146.5 MHz ; 146.5 MHz       ; clk        ;      ;
+------------+-----------------+------------+------+
```

Wall-clock time is the key performance metric for HLS, computed as the product of the cycle latency and the clock period. In this case, our cycle latency was 3,036,123 and the clock period was 6.8ns. The wall-clock time of our implementation is therefore $3{,}036{,}123 \times 6.8 = 20.7ms$.

Fill in the results for this part of the lab under the column labelled "Basic" in the table in Section 6.

3 Loop Unrolling and Pipelining

In this section, you will use loop unrolling and pipelining to improve the throughput of the hardware generated by LegUp. Loop unrolling refers to modifying a loop by replicating its loop body and then reducing its trip count (# of loop iterations) correspondingly. For example, "unrolling by a factor of 2" means to duplicate the loop body and halve the loop trip count. Loop unrolling may expose parallelism to the HLS compiler, allowing a high-performance circuit to be generated

Loop pipelining allows a new iteration of the loop to be started before the current iteration has finished. By allowing the execution of the loop iterations to be overlapped, higher throughput can be achieved. The amount of overlap is indicated by the initiation interval (II). The II indicates how many cycles are required before starting the next loop iteration. Thus, an II of 1 means a new loop iteration can be started every clock cycle, which is the best one can achieve. The II needs to be larger than 1 in other cases, when there is resource contention or when there are loop-carried dependencies. Figure 2 shows an example of loop pipelining. Figure 2(b) shows the sequential loop, where the II = 3, and it takes 9 clock cycles for the 3 loop iterations before the final write is performed. Figure 2(c) shows the pipelined loop. In this example, there are no resource contentions or data dependencies. Hence, the II = 1, and it takes 5 clock cycles before the final write is performed. You can see that loop pipelining can significantly improve the performance of your circuit, especially when there are no data dependencies or resource contentions.

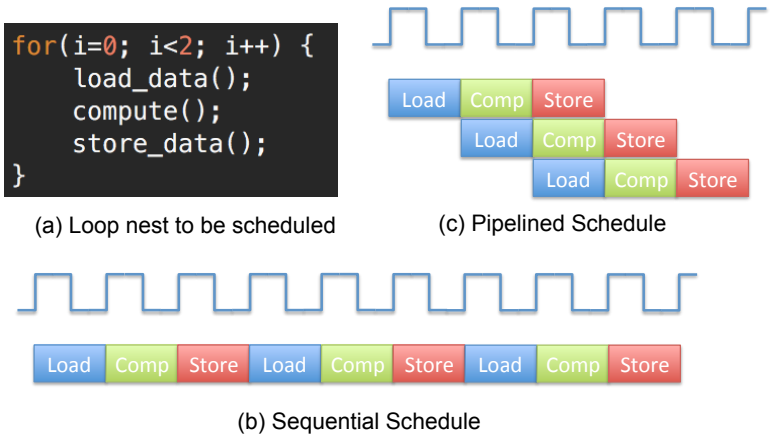

Figure 2: Loop Pipelining Example

Change directory into `part2` of this lab. `cnn.h` is identical to that used in the prior step. `cnn.c` has a small change in the inner loop of the `convolution` function, namely:

```
for (im = 0; im < NUM_INPUT_MAPS; im+=2) {
     // two MACs per loop iteration
     output_fm_value += weights[om][i][j][im] * input_fms[row+i][col+j][im];
     output_fm_value += weights[om][i][j][im+1] * input_fms[row+i][col+j][im+1];
}
```

Observe that the inner loop has been unrolled by a factor of two. There are two accesses to the `weights` array and two accesses to the `input_fms` array every loop iteration. As the RAMs on FPGAs are dual-ported, these two memory operations per array can execute concurrently (in parallel). The main reason to unroll is to expose such parallelism to the HLS compiler. The hope here is that, while the number of clock cycles per loop iteration may increase in the unrolled versus original case, it will not double, leading to an overall reduction in cycle latency and improved performance.

Compile the project in software and execute the software using `gcc` as you did in the previous step. You should see `PASS` in the terminal window. Synthesize the software to hardware using LegUp HLS and simulate the hardware with ModelSim (type `make` followed by `make v`), you should see the following:

```
...
# TOTAL:      -18870
# PASS
# At t=         42659150000 clk=1 finish=1 return_val=         0
# Cycles:              2132955
# ** Note: $finish    : cnn.v(1977)
#    Time: 42659150 ns  Iteration: 4  Instance: /main_tb
```

The circuit now completes its work in about 2.1 million cycles – a huge reduction over the 3 million cycles required for part 1 of this lab above.

Now, modify `cnn.c` to perform further unrolling, where the original loop will be unrolled by a factor of four. Change the code in `cnn.c` so the inner loop in the `convolution` function looks like this:

```
for (im = 0; im < NUM_INPUT_MAPS; im+=4) {
     // four MACs per loop iteration
     output_fm_value += weights[om][i][j][im] * input_fms[row+i][col+j][im];
     output_fm_value += weights[om][i][j][im+1] * input_fms[row+i][col+j][im+1];
     output_fm_value += weights[om][i][j][im+2] * input_fms[row+i][col+j][im+2];
     output_fm_value += weights[om][i][j][im+3] * input_fms[row+i][col+j][im+3];
}
```

Save your code changes! Compile in software and execute in software to verify correctness (using `gcc`). Then, synthesize to hardware and simulate the hardware (use `make` then `make v`). You should see results similar to this:

```
...
# TOTAL:      -18870
# PASS
# At t=         29111630000 clk=1 finish=1 return_val=         0
# Cycles:              1455579
# ** Note: $finish    : cnn.v(2309)
```

With unrolling by four, cycle latency is reduced further to 1.46 million cycles. Now turn on loop pipelining by adding a label, `loop:` to the inner-most loop, as follows:

```
loop: for (im = 0; im < NUM_INPUT_MAPS; im+=4) {
```

Don't forget to save your code changes! In the `part2` directory, open up the `config.tcl` file and uncomment the directive to turn on loop pipelining for the loop with the label you added. This tells LegUp HLS to try to pipeline the specific loop mentioned in the directive.

Having made those modifications, you can now synthesize the design and simulate it using `make` followed by `make v`. You should see something about loop pipelining in the LegUp HLS output. Here, you can see that the II of the loop is 2. Open up the LegUp pipelining report – the file name is `pipelining.legup.rpt`. At the bottom of the report, you can see a 6-stage pipeline was generated for the loop, with an II=2.

Take a moment to think about what are some potential reasons why the II cannot be 1 in this case. Are there any loop-carried dependencies? What about resource contentions?

Now, simulate the design in ModelSim by clicking the *simulate HW* icon on the toolbar. You should see results similar to this:

```
...
# TOTAL:      -18870
# PASS
# At t=         20079950000 clk=1 finish=1 return_val=         0
# Cycles:              1003995
```

Observe that loop pipelining has dramatically improved the cycle latency for the design, and it is now about 1 million clock cycles. When this design is implemented on Cyclone V, the area and performance results are shown below. Fill in the column "Unroll/Pipelining" in the results table at the end of this handout. Can you think of ways to improve the results further?

```
Family : Cyclone V
Device : 5CSEMA5F31C6
Timing Models : Final
Logic utilization (in ALMs) : 350 / 32,070 ( 1 % )
Total registers : 585
Total pins : 0 / 457 ( 0 % )
Total virtual pins : 36
Total block memory bits : 124,416 / 4,065,280 ( 3 % )
Total RAM Blocks : 18 / 397 ( 5 % )
Total DSP Blocks : 4 / 87 ( 5 % )

+-----------------------------------------------------+
; Slow 1100mV 85C Model Fmax Summary                  ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 142.88 MHz ; 142.88 MHz      ; clk        ;      ;
+------------+-----------------+------------+------+
```

4 Memory Partitioning

In the prior part, we unrolled the inner loop manually, and turned on loop-pipelining. Performance improvements were attained with each of these optimizations. One of the limiters of performance, however, is the number of memory ports on RAMs in the FPGA. Generally, commercial FPGAs contain dual-port RAMs. In this part, the weights and input feature maps are *partitioned* into multiple separate arrays. Each array is implemented as a separate RAM in the FPGA, and thus, each array can be accessed independently. Array partitioning is a key way to improve memory-access parallelism in HLS, which can significantly help performance.

Change directories to access part 3 of this lab (`cd ../part3`). First look at `cnn.h`. You will see that instead of there being a single `weights` array and a single `input_fms` array, these arrays have each been split into two. We now have `weights0` and `weights1`, for example. The partitioning has been done at the *last* array dimension, which is now 8 instead of 16, as it was before. Now open `cnn.c` and look at the `convolution` function, and specifically, at the inner-most loop. Observe how the partitioned arrays are being accessed. Note that modern HLS tools, including LegUp, have the ability to *automatically* partition arrays, based on user constraints. Here, however, the partitioning has been done manually.

Compile and run the software (using `gcc`). Compile the software to hardware using LegUp HLS, and run ModelSim, using `make` followed by `make v`. You should see results similar to that below, a `PASS` and a cycle latency of 728,026 (versus about 1 million cycles for the last part of this exercise).

```
# TOTAL:      -18870
# PASS
# At t=         14560570000 clk=1 finish=1 return_val=         0
# Cycles:               728026
```

Notice the reduced cycle count of 728K with the memory partitioning and loop pipelining! In the LegUp HLS output, observe that the report II=1. Why is the II reduced to 1 in this case?

The results for the Cyclone V implementation are shown below:

```
Family : Cyclone V
Device : 5CSEMA5F31C6
Timing Models : Final
Logic utilization (in ALMs) : 329 / 32,070 ( 1 % )
Total registers : 519
Total pins : 0 / 457 ( 0 % )
Total virtual pins : 36
Total block memory bits : 124,416 / 4,065,280 ( 3 % )
Total RAM Blocks : 18 / 397 ( 5 % )
Total DSP Blocks : 3 / 87 ( 3 % )

+-------------------------------------------------+
; Slow 1100mV 85C Model Fmax Summary               ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 129.77 MHz ; 129.77 MHz      ; clk        ;      ;
+------------+-----------------+------------+------+
```

Fill in the results for the "Partitioning" column in the results table at the end of this handout.

5 Synthesizing Parallel Hardware with Pthreads

In this section, you will implement a multi-threaded version of the convolution circuit using Pthreads. Through this approach, a software engineer without hardware skills can exploit spatial parallelism on an FPGA. Each software thread will translate to an instance of the hardware accelerator kernel module. In this case, we will use two threads. A

first thread responsible the top half of rows of output feature maps; a second thread responsible for the bottom half of rows.

Change directories to access the final part of this lab, part 4. There are no differences in the `cnn.h` file versus the previous part – this part also uses the partitioned memories. The only differences are in `cnn.c`. In `cnn.c` look at the definition of the `pthread_arg` struct, which has two fields, `low` and `high`.

```
typedef struct {
  int low;
  int high;
} pthread_arg;
```

These represent the range of output-feature-map row indices a thread is to be responsible for computing. Now look at the `convolution` function, which is nearly identical to the previous part of this exercise. However, in this case, the thread argument is used to determine loop bounds related to the output feature maps. At the bottom of the function, you will see the update to `total` is in a *critical section*, as multiple threads update this value – thus, a `mutex` is required. The changes to the `main` function are shown below. Two threads are forked using `pthread_create`. Each thread handles half of the output feature map rows.

```
int main(void) {

  pthread_arg arg1, arg2;
  arg1.low = 0;
  arg1.high = 4;
  arg2.low = 4;
  arg2.high = 8;

  pthread_t t1, t2;
  pthread_create(&t1, NULL, convolution, &arg1);
  pthread_create(&t2, NULL, convolution, &arg2);
```

Compile in software and run the software – you may need to add the `-lpthread` directive when you run `gcc`. This tells the compiler to link in the Pthreads library. When you run the program, you should see a successful execution and `PASS` in the terminal window.

Then run LegUp HLS and simulate the generated Verilog (use `make` and `make v` as you have done already many times!). You should observe results similar to that shown below, with total cycles being 636,858 and a `PASS`. We observe yet another significant reduction in cycle latency versus the prior parts of the exercise. In fact, relative to part 1 of this exercise, we have achieved a nearly 5× reduction in cycle latency with relatively small code changes and some HLS constraints!

```
# TOTAL:      -18870
# PASS
# At t=         12737210000 clk=1 finish=1 return_val=         0
# Cycles:               636858
# ** Note: $finish    : cnn.v(4055)
```

Results for the Cyclone V implementation are shown below. Notice that a large increase in ALMs and DSPs is observed versus the prior case. Why is such an area increase observed?

```
Family : Cyclone V
Device : 5CSEMA5F31C6
Timing Models : Final
Logic utilization (in ALMs) : 1,008 / 32,070 ( 3 % )
Total registers : 1758
Total pins : 0 / 457 ( 0 % )
Total virtual pins : 36
Total block memory bits : 124,416 / 4,065,280 ( 3 % )
```



```
Total RAM Blocks : 18 / 397 ( 5 % )
Total DSP Blocks : 6 / 87 ( 7 % )

+-----------------------------------------------------+
; Slow 1100mV 85C Model Fmax Summary                  ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 123.69 MHz ; 123.69 MHz      ; clk        ;      ;
+------------+-----------------+------------+------+
```

Fill in the “Pthreads” column of the results table at the end of this handout.

6 Performance and Area Results

Table 1: Results worksheet.

	Basic	Unroll/Pipelining	Memory Partitioning	Pthreads
Cycles				
FMax				
Clock period				
Wall-clock time				
ALMs				
DSP blocks				

7 Questions for Class Discussion

1. High-level synthesis (HLS) is becoming more popular as a digital circuit design methodology. Name three advantages of HLS.

2. Give an example of an application for which HLS would most likely *not* be a suitable design methodology?

3. What are the limitations to parallelizing in hardware? How do they compare to parallelization in software?

4. Do you think this convolution benchmark is a suitable application for multithreaded hardware? Why or why not? List some other examples of applications that may be ideal for multithreaded hardware.

5. Do you think the coding style of the C source code plays a large role in determining the quality of results of the final synthesized design? If so, can you think of some coding strategies / guidelines for each category of benchmarks?

6. In this lab, we showed several ways to parallelize the hardware, can you think of any others? How do they compare with the methods mentioned in this lab?

8 Summary

High-level synthesis is gaining traction as a design methodology, as it allows hardware to be designed at a higher level of abstraction, lowering design time and cost. In this lab, you have gained experience with several key high-level synthesis concepts, including loop pipelining, exploiting spatial parallelism, and pipelining functionality, as applied to a practical example: convolution in hardware. Several key research challenges for HLS remain: 1) continued efforts to raise the quality of HLS hardware to bring it as close as possible to human-crafted hardware quality; 2) development of automated transformations to eliminate the need for engineers to "tune" their coding style to meet tool expections; and 3) the development of software-like debugging and profiling tools for HLS-generated hardware.

