HLS Implementation of CNN Accelerator

Garima Modi (2018CSZ8010) Samiksha Agrawal(2018SIY7505)
Department of Computer Science and Engineering
IIT Delhi

June 22, 2020

Abstract

In Section 1, the given algorithm is discussed. Section 2 provides the analysis of the algorithm after applying various transformations. In Section 3 , the test-bench developed to verify the algorithm functionality is discussed. Section 4 provides the detailed description of the proposed algorithm along with its Implementation in Vivado HLS. The experimental results are discussed in the same section. Section 5 discusses some of the non-obvious observations during the course of our project.

1 Overview

The algorithm given for analysis is analogous to the convolution layer of the CNN algorithm. This section provides brief overview of the given algorithm. The pictorial representation of the algorithm is given in Figure 1.

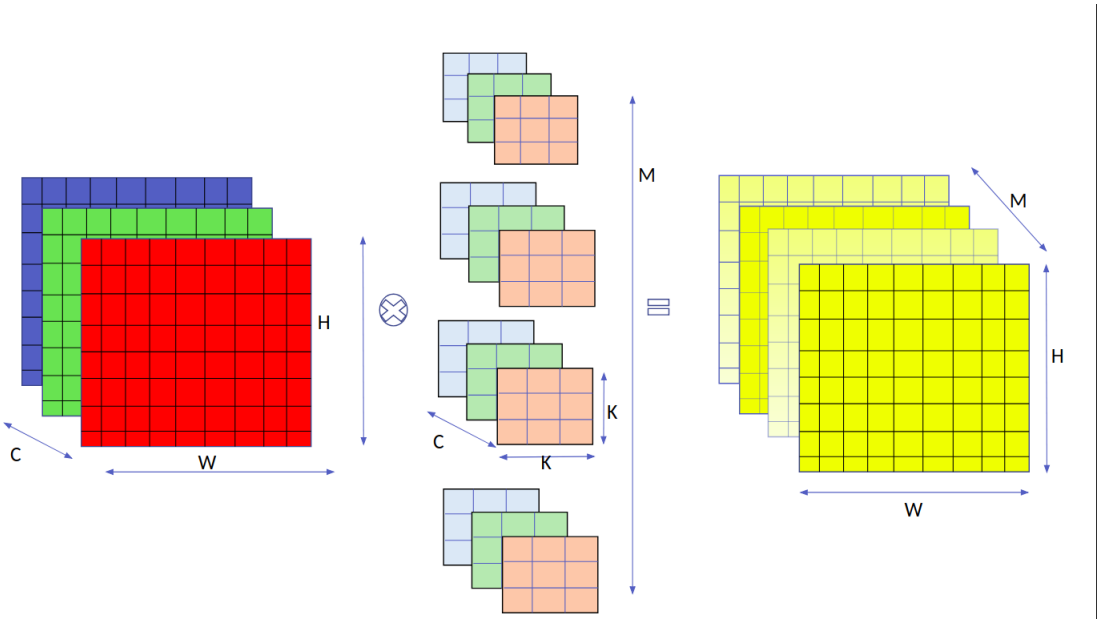


Figure 1: Overview of the given algorithm

1.1 Inputs to the algorithm

The algorithm takes two inputs:

1. **Image:**
    - **Input Form:** 3D Matrix
    - **Dimensions:** CxHxW

- **Description:** Each cell value ranges from 0-255 (pixel values). The image has C channels, each of Height, H, and Width, W.

2. **Kernel:**

   - **Input Form:** 4D Matrix
   - **Dimensions:** MxKxKxC
   - **Description:** There are M different Kernels each having C channels of size KxK. The kernel matrix is generally much smaller than the image matrix.

1.2 Output of the algorithm

Each kernel is convoluted with the image to produce one output layer. Hence, M kernels results in M output layers. The output is a 3-D matrix, having **M layers each of size HxW**.

1.3 Algorithm

The algorithm provided for the analysis is analogous to the convolution layer of CNN algorithm. The algorithm (with padding) is given in Algorithm 1:

Algorithm 1: Baseline Algorithm

```
Result: output[M][H][W]
input[C][H][W], kernel[M][K][K][C];
LH: for h 0 to H do
    LW: for w 0 to W do
        LM: for m 0 to M do
            int sum = 0;
            LK1:for x 0 to K do
                LK2: for y 0 to K do
                    LC: for c 0 to C do
                        sum += (h+x>H|| w+y>W)? 0 : input[i][h+x][w+y]
                         *kernel[m][x][y][i];
                    end
                end
            end
            output[m][h][w] = sum;
        end
    end
end
```

The algorithm runs as follows:

- Each kernel is convoluted (sum of products) with KxKxC elements of the image at a time and with the stride of 1 moves left to right and then top to bottom.
- Convolution of one kernel produces 1 layer of HxW dimensions. Hence, M kernels result in M different layers each of size HXW.

Few changes needed to be done to the given algorithm, due to anomalies or inconsistencies in desired behaviour. The changes done to the given algorithm are:

- The given algorithm had dimensions mismatch, which allowed the computations to be performed on the non-existing elements in the array i.e., accessed out of bounds indexes in input image and output arrays.
- Hence, the algorithm is converted to the original convolution layer, which was the intention of the paper [5] from which the algorithm was obtained.
- To obtain the output of size as same as of the input image, zero padding is done on the right side and bottom of the image of size K-1.

2 Analysis of Algorithm on Vivado HLS

In this section, we discuss the timing and resource utilization analysis of the algorithm after applying various Vivado directives and loop transformations.

2.1 Configuration used for analysis

1. **Input Image Size:** 3x256x256
2. **Input Kernel Size**: 20x3x3x3
3. **Output Size:** 20x256x256
4. **Clock period:** 10 ns
5. **FPGA:** Kintex 7

   Resource Availability:

   - **BRAM:** 650
   - **DSP:** 600
   - **FF:** 202800
   - **LUT:** 101400

2.2 AXI Port

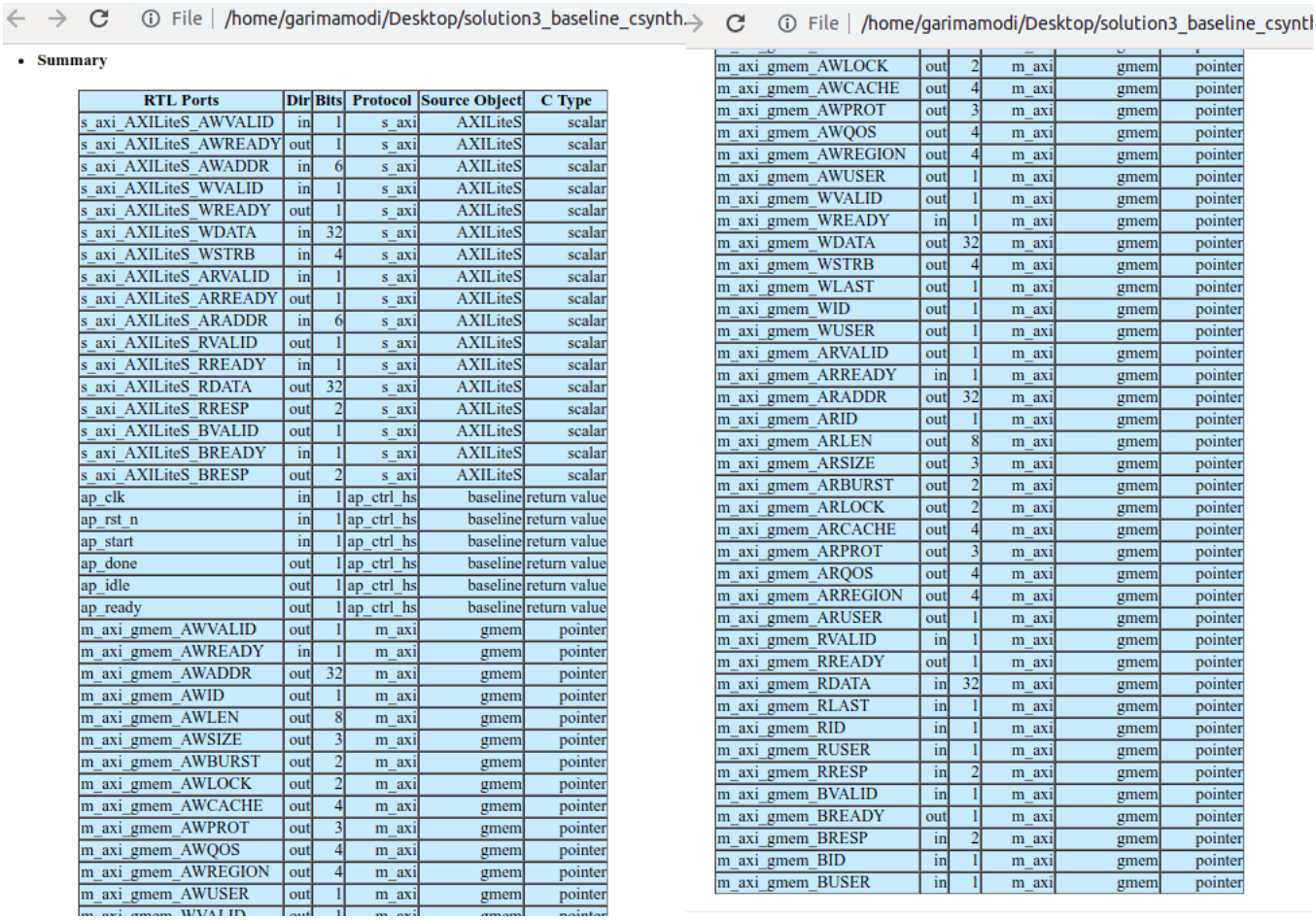
To simulate the off-chip memory, axi port is used. The following Vivado pragma is used:

#pragma HLS INTERFACE m_axi

2.2.1 IP Ports

The Figure 2 shows the interface summary reported in Vivado synthesis report.

As can be seen from the Figure 2, s_axi (off-chip master ports) have two data ports, one each for input and output. The reported m_axi (slave ports) are added to have the connection of input image and kernel with the read data port of axi and output with the write data port of axi.



- **Summary**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_AXILiteS_AWVALID	in	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_AWREADY	out	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_AWADDR	in	6	s_axi	AXILiteS	scalar
s_axi_AXILiteS_WVALID	in	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_WREADY	out	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_WDATA	in	32	s_axi	AXILiteS	scalar
s_axi_AXILiteS_WSTRB	in	4	s_axi	AXILiteS	scalar
s_axi_AXILiteS_ARVALID	in	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_ARREADY	out	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_ARADDR	in	6	s_axi	AXILiteS	scalar
s_axi_AXILiteS_RVALID	out	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_RREADY	in	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_RDATA	out	32	s_axi	AXILiteS	scalar
s_axi_AXILiteS_RRESP	out	2	s_axi	AXILiteS	scalar
s_axi_AXILiteS_BVALID	out	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_BREADY	in	1	s_axi	AXILiteS	scalar
s_axi_AXILiteS_BRESP	out	2	s_axi	AXILiteS	scalar
ap_clk	in	1	ap_ctrl_hs	baseline	return value
ap_rst_n	in	1	ap_ctrl_hs	baseline	return value
ap_start	in	1	ap_ctrl_hs	baseline	return value
ap_done	out	1	ap_ctrl_hs	baseline	return value
ap_idle	out	1	ap_ctrl_hs	baseline	return value
ap_ready	out	1	ap_ctrl_hs	baseline	return value
m_axi_gmem_AWVALID	out	1	m_axi	gmem	pointer
m_axi_gmem_AWREADY	in	1	m_axi	gmem	pointer
m_axi_gmem_AWADDR	out	32	m_axi	gmem	pointer
m_axi_gmem_AWID	out	1	m_axi	gmem	pointer
m_axi_gmem_AWLEN	out	8	m_axi	gmem	pointer
m_axi_gmem_AWSIZE	out	3	m_axi	gmem	pointer
m_axi_gmem_AWBURST	out	2	m_axi	gmem	pointer
m_axi_gmem_AWLOCK	out	2	m_axi	gmem	pointer
m_axi_gmem_AWCACHE	out	4	m_axi	gmem	pointer
m_axi_gmem_AWPROT	out	3	m_axi	gmem	pointer
m_axi_gmem_AWQOS	out	4	m_axi	gmem	pointer
m_axi_gmem_AWREGION	out	4	m_axi	gmem	pointer
m_axi_gmem_AWUSER	out	1	m_axi	gmem	pointer
m_axi_gmem_WVALID	out	1	m_axi	gmem	pointer
m_axi_gmem_WREADY	in	1	m_axi	gmem	pointer
m_axi_gmem_WDATA	out	32	m_axi	gmem	pointer
m_axi_gmem_WSTRB	out	4	m_axi	gmem	pointer
m_axi_gmem_WLAST	out	1	m_axi	gmem	pointer
m_axi_gmem_WID	out	1	m_axi	gmem	pointer
m_axi_gmem_WUSER	out	1	m_axi	gmem	pointer
m_axi_gmem_ARVALID	out	1	m_axi	gmem	pointer
m_axi_gmem_ARREADY	in	1	m_axi	gmem	pointer
m_axi_gmem_ARADDR	out	32	m_axi	gmem	pointer
m_axi_gmem_ARID	out	1	m_axi	gmem	pointer
m_axi_gmem_ARLEN	out	8	m_axi	gmem	pointer
m_axi_gmem_ARSIZE	out	3	m_axi	gmem	pointer
m_axi_gmem_ARBURST	out	2	m_axi	gmem	pointer
m_axi_gmem_ARLOCK	out	2	m_axi	gmem	pointer
m_axi_gmem_ARCACHE	out	4	m_axi	gmem	pointer
m_axi_gmem_ARPROT	out	3	m_axi	gmem	pointer
m_axi_gmem_ARQOS	out	4	m_axi	gmem	pointer
m_axi_gmem_ARREGION	out	4	m_axi	gmem	pointer
m_axi_gmem_ARUSER	out	1	m_axi	gmem	pointer
m_axi_gmem_RVALID	in	1	m_axi	gmem	pointer
m_axi_gmem_RREADY	out	1	m_axi	gmem	pointer
m_axi_gmem_RDATA	in	32	m_axi	gmem	pointer
m_axi_gmem_RLAST	in	1	m_axi	gmem	pointer
m_axi_gmem_RID	in	1	m_axi	gmem	pointer
m_axi_gmem_RUSER	in	1	m_axi	gmem	pointer
m_axi_gmem_RRESP	in	2	m_axi	gmem	pointer
m_axi_gmem_BVALID	in	1	m_axi	gmem	pointer
m_axi_gmem_BREADY	out	1	m_axi	gmem	pointer
m_axi_gmem_BRESP	in	2	m_axi	gmem	pointer
m_axi_gmem_BID	in	1	m_axi	gmem	pointer
m_axi_gmem_BUSER	in	1	m_axi	gmem	pointer

Figure 2: Interface Summary reported in Vivado synthesis report

2.2.2 Timing analysis of AXI port

In general, to read (or write) data using AXI port, at first request is send (7 clock cycles), then the data is read (1 cycle) and if to be stored in local (on-chip) memory, then takes one more cycle.

Figure 3, shows the timing analysis when data is read from off-chip memory and directly used in the algorithm. (In the given figure, there are two 7 cycles request, each for input image and kernel, the next two cycles corresponds to each read. Hence, one read request takes 8 cycles). To avoid the 7 cycle utilization for every read or write request, the read / write of consecutive memory locations should be done, if possible. This is being achieved through the **Burst Mode** of the AXI port inferred.

2.3 On-chip storage

Requesting data again and again from off-chip memory is an expensive operation (8 memory cycles). Hence, if the same data is going to be used again, it should be stored

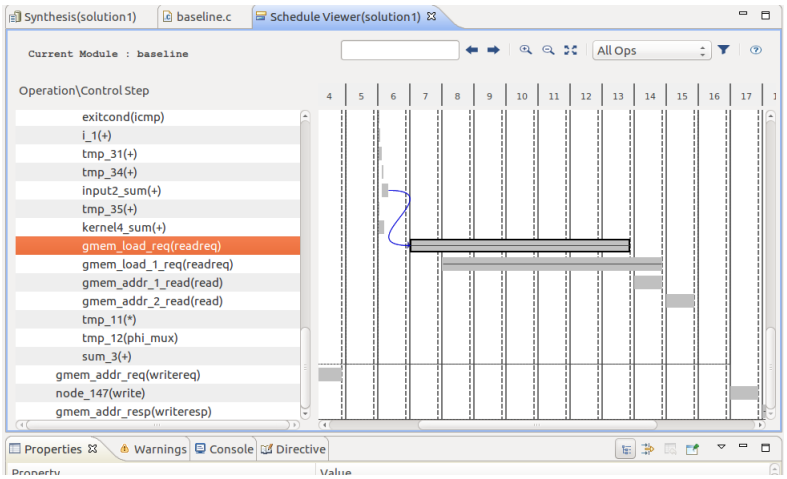


Figure 3: Timing analysis using AXI port

in the on chip memory. This can be achieved by declaring the local variables inside the C program, as shown in Figure 4.

```
#include "baseline.h"

void baseline(int input[C][H][W], int kernel[M][K][K][C], int output[M
{

    int local_input[C][H][W], local_kernel[M][K][K][C];


    int p, q, r, s;
    for(p=0;p<C;p++)
    {
        for(q=0;q<H;q++)
        {
            for(r=0;r<W;r++)
            {
                local_input[p][q][r]=input[p][q][r];
            }
        }
    }
```

Figure 4: Local variables declaration to emulate on-chip memory

2.3.1 Trade-off between on-chip storage and off-chip storage

The trade-off between accessing data from off-chip memory and storing entire data on-chip (one-time off-chip access) is given in Table 1. From Table 1, we can see that

Resource	off-chip	on-chip
Cycle	424,644,821	147,509,677
BRAM	0	514
DSP	3	3
FF	1250	1513
LUT	1831	2510

Table 1: Trade-off between on-chip and off-chip storage

the on-chip storage decreases the latency by 2.87%. But, the resource requirements

increases. Taking latency into account, further analysis is done on the on-chip storage architecture. The resource requirement can be balanced by storing limited data on-chip, rather than complete data. These types of designs are evaluated in further sections.

2.4 Loop Interchange

There are 6 nested loops, labelled as LH, LW, LM, LK1, LK2 and LC, as given in Algorithm 1. This section discusses the timing and resource analysis after interchanging various loops.

2.4.1 Changing the position of loop LM

The kernel data access per instruction will be more than the image data, as justified in [2]. The reason is that each kernel value is convoluted with all image data cells, so each kernel value is called HxW times, whereas, for one kernel, each image data is called at max K times. Hence, convoluting one kernel at a time with the entire image will reduce the memory access cycles. Hence, M should be the outermost loop, as can be observed in Table 2. Hence, LM is made the outermost loop.

Resource/Loop Positioning	**HWMXYC**	**MHWXYC**
Cycles	139,887,097	139,767,757
BRAM	514	514
DSP	3	3
FF	1499	1484
LUT	2348	2348

Table 2: LM loop interchange

2.4.2 LC Loop Interchange

Changing the position of loop LC does not change latency. The primary reason is that the storage of the image and kernel is in row major order. For the image the last changing dimension is C, whereas in case of kernel, the frequently changed dimension is C. But, the change in resource requirement of FF and LUT is observed by changing the position of C loop, as shown in Table 3. Thus, we decide to keep LC loop before LK1 as it shows minor improvement in resource consumption.

Resource	Innermost LC	LC before LK1
Cycles	139,767,757	139,767,757
FF	1497	1405
LUT	2362	2258

Table 3: LC loop interchange

2.5 Loop Unrolling

Vivado HLS provides a pragma for loop unrolling:

#pragma HLS UNROLL skip_exit_check factor=unroll_factor_value region

The skip_exit_check pragma can only be used if partial unrolling is specified, i.e, unroll factor is less than size, and the size is a multiple of unroll factor and presence of region leads to all inside loops to be unrolled except the outside one, whereas absence means just unroll the specified outside loop.

2.5.1 Unroll LM loop

Every kernel convolution is independent of each other. Therefore, multiple kernels can convolute with same image pixels parallely. To exploit this feature, we first try to see the effect of unrolling on the LM loop.

Table 4, tabulates the results of unrolling LM without region with different unroll factors. From Table 4, it can be observed that the increase in unroll factor decreases

Resource	Factor 2	Factor 5	Factor 10	Factor 15	Factor 20
Latency	139,767,747	139,767,741	139,767,739	209,450,114	139,767,736
BRAM Utilization %	~ 79	~ 79	~ 79	~ 79	~ 79
DSP Utilization %	~ 0	~ 0	~ 0	~ 0	~ 0
FF Utilization %	~ 0	~ 1	~ 1	~ 2	~ 2
LUT Utilization %	~ 3	~ 4	~ 7	~ 11	~ 11

Table 4: LM loop unrolling without region

the latency, but the effect is not so much, as the region disabled unrolls only outer loop. In case of factor 15, we observed increase in latency as skip_exit_check is disabled as 15 is not the factor of M=20, as considered for analysis.

Table 5 reports the unrolling LM latency and resource utilization of kintex-7 with region enabled. In Table 5, NA means that the time/resource requirement exceeded w.r.t the one provided by the FPGA used in analysis (reported by Vivado tool) . Hence, the synthesis could not be performed on unroll factor 15 and 20. Due to shooting up of resource requirements, we decide to unroll LM with region enabled.

Resource	Factor 2	Factor 5	Factor 10	Factor 15	Factor 20
Latency	207,507,017	9,994,737	8,232,237	NA	NA
BRAM Utilization %	~ 79	~ 79	~ 79	~ 79	~ 79
DSP Utilization %	~ 0	~ 148	~ 307	NA	NA
FF Utilization %	~ 2	~ 6	~ 18	NA	NA
LUT Utilization %	~ 11	~ 34	~ 100	NA	NA

Table 5: LM loop unrolling with region

2.5.2 LC loop unroll

Table 6, shows the latency and resource utilization percentage with and without unroll (unroll factor= K).

Resource	No unroll	Unroll without region	Unroll with Region	Manual Unroll
Latency	139,761,121	134,599,841	31,374,241	31,374,241
BRAM %	~ 79	~ 79	~ 79	~ 79
DSP %	~ 0	~ 0	~ 0	~ 0
FF %	~ 0	~ 0	~ 0	~ 0
LUT %	~ 1	~ 1	~ 2	~ 2

Table 6: LC unroll

From Table 6, the unroll with region is similar to manual unroll, as region unrolls all the innermost loops. It is observed that unrolling of LC decreases the latency.

2.5.3 Loop Tiling

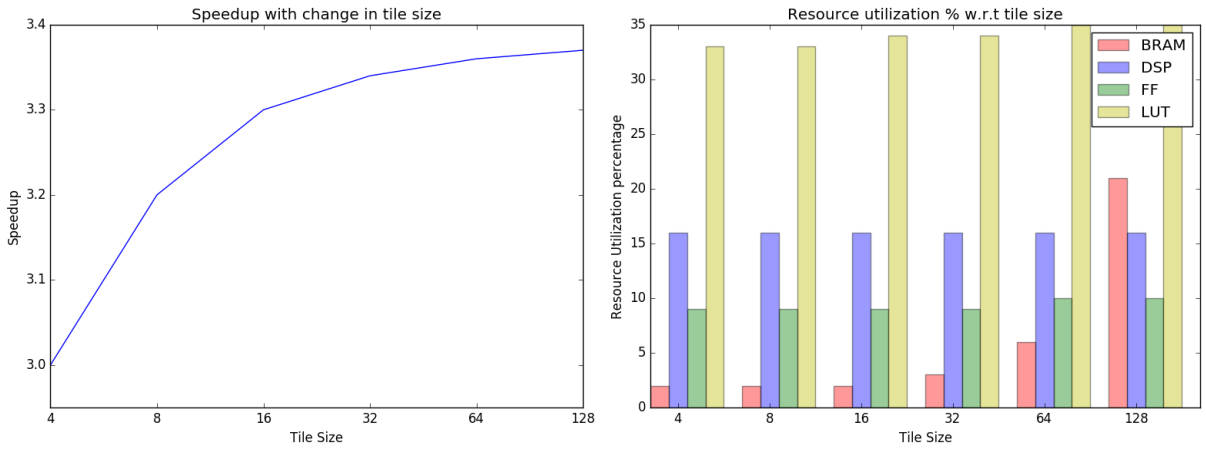
We applied loop tiling transformation on LH and LW loops (loops traversing image height and width). Once all the computation is done on the image tile (size ThxTw for each channel), the tile data is not required again for the convolution with the kernel. Along with tiling, we applied Loop unrolling on LM (without region) with unroll factor 5, such that 5 kernels can convolute together with same tile. Loop unrolling is also applied on LC (with region) with unroll factor of K.

The speedup obtained in loop tiling (Th=Tw=x=axis value) w.r.t. the code without loop tiling and unrolling is shown in Figure 5a.

The resource utilization percentage is shown in Figure 5b.

Observations:

- Increase in tile size decreases latency.
- Increase in tile size increases BRAM utilization.



(a) Speedup w.r.t tile size

(b) Resource utilization w.r.t tile size

Figure 5: Loop tiling transformation

3 Test-bench generation

3.1 Input Dataset generation step

To generate an input dataset, a c program (generate_data.c, uploaded on moodle) is written, that takes the size of image and kernel as user inputs and randomly generates the data values and store them in .dat files of image and kernel.

3.2 Expected output generation

A C program (primitive_algo.c, uploaded on moodle) is written that reads the .dat file of image and kernel and then applies the given function and generates the output. The generated output is written in the file, which gives the expected output values for the verification process.

3.3 Test Bench

The test-bench is written as C program (testbench.c), that reads the input .dat files, that are used to generate the expected output file and then runs the function that is synthesized in Vivado. The output generated by the function is written back to file and compared against the expected output using ”diff” utility. If the output matches the expected output, the simulation passes else fails.

After applying any transformation on the function, we always ran the C simulation to verify if the generated output matches the expected output or not.

4 Hardware conceptualization and its implementation using HLS

4.1 On-chip Storage

4.1.1 Proposed hardware

From above analysis done using Vivado HLS, it was clear that the concept of data-reuse for both Kernel and image has to be exploited to the optimum extent. Since the kernel size is small and the same kernel is going to be accessed multiple number of times, the proposal was to store the kernel data in the local memory (registers) of the Processing Elements(PE) array. Since the kernel was of height K bits, we decided to store the first K rows of the input image (all the channels) in the on-chip Block RAMS. Similarly, each output row will be stored in 1 BRAM.

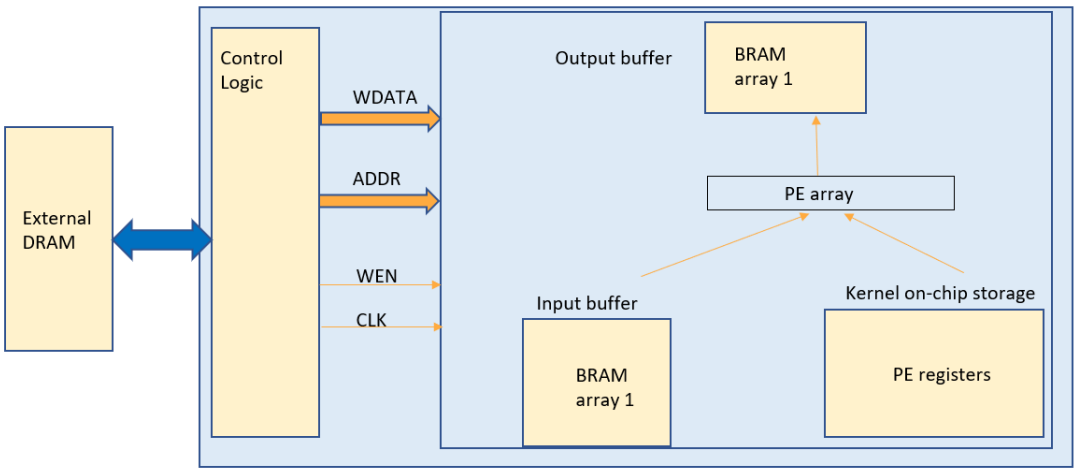


Figure 6: Block diagram of the architecture proposal

The Figure 6 shows the top-level block diagram of the hardware.

4.1.2 Vivado Implementation

In order to infer BRAMs for input image and output image for local on-chip storage, these were declared as variables and appropriate pragma was used.

The pragmas used are shown in Figure 7. As can be seen from this figure,

1. The input image has been partitioned across channel number dimension. Thus, a separate BRAM shall be inferred for each channel of the image.Further, for every channel, there are 4 BRAMs to store 4 rows of the image. Although the

```
set_directive_array_partition -type block -factor 3 -dim 1 "baseline" local_image
set_directive_array_partition -type block -factor 4 -dim 2 "baseline" local_image
set_directive_array_partition -type complete -dim 0 "baseline" local_kernel
set_directive_array_partition -type block -factor 5 -dim 1 "baseline" local_output
```

Figure 7: Pragmas used to infer appropriate on-chip storage for image, kernel and output

computation will make use of 3 rows at a time, we are prefetching 1 extra row, so as to hide its latency with the current computation.

2. Since we do not intend to store Kernel data in BRAMs, we completely partition the kernel across all dimensions (dim 0 means all dimensions).

3. Since we are convolving the image with 5 kernels at a time (unroll factor = 5),we need to store 5 output rows. Thus, partitioning output into 5 BRAMs across the 1st dimension.

4.1.3 Results

Figure 8: Resource diagram to illustrate BRAMs inferred

Figure 8 clearly illustrate that 4 image BRAMs (3 + 1 prefetched) and 5 output BRAMs (as unroll factor = 5).

4.2 Computation phase

4.2.1 Proposed Hardware

While deciding an optimum hardware for the computation of output, two factors were given importance:

1. There is a lot of scope of image data-reuse in the process of convolution. Figure 7 shows that the red area of input feature map will be used repeatedly between 2 consecutive convolutions. Thus, we store the first 3 columns from the Image BRAMS onto registers inside PE array. In the next computation, the pixels of Column1 are shifted out of this memory, the second and third column of previous computation are shifted-left and the 4th column of the image become the new 3rd column.

2. The computation of an output pixel completes only when all the 3 channels of the input image is convolved with the kernel. Thus, the computation in the PE array happens parallelly for different channels of image, so that we need not store the partial sums of output image separately.

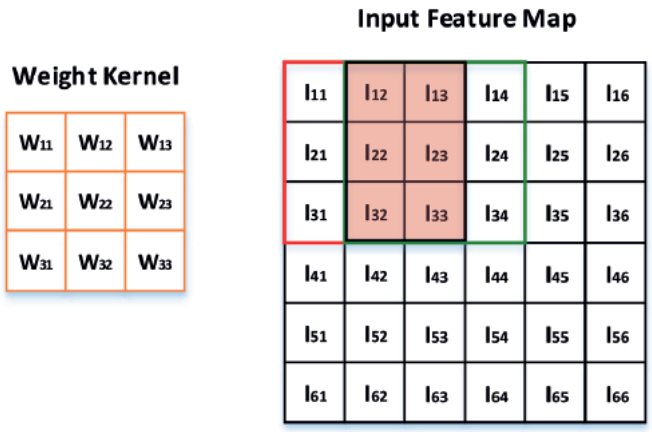


Figure 9: Image data REUSE in the process of convolution

Figure 10 shows the structures inferred using the PE registers. For illustration purpose, we have shown 1 kernel of size 2x2 and 1 channel of image of size 2x4. Image has been padded with one extra column (5th column) and initialized with zeroes. This has been done to ensure that the size of the final output does not reduce in width.

As can be seen from Figure 10, all the stages involved in the computation of an output row, have been pipelined. Thus, after a initial pipeline filling latency of 4, we are able to get throughput=1 and a new output is generated every cycle.

Further the LOAD operation for Set2, Set3 and Set4 are each comprised of 3 sub-operations: Shifting out the leftmost column, shifting left the rightmost column and inserting a new column from BRAM in place of the rightmost column All these happen

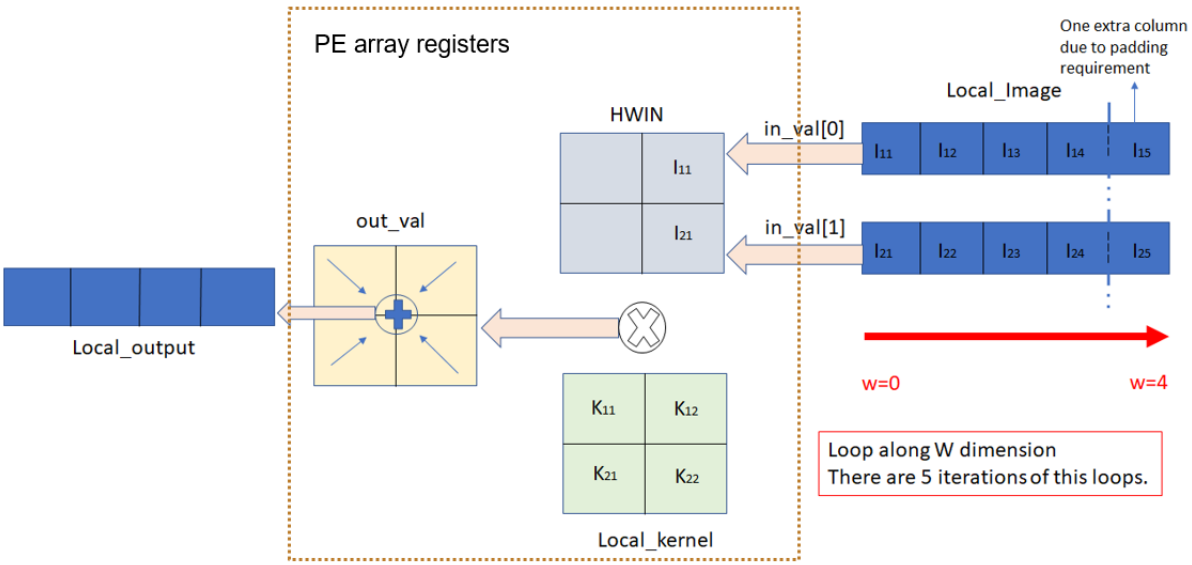


Figure 10: Storage inside PE array

	LOAD	MULTIPLY	ADD	O/P write
Cycle1	LOAD_ SET1			
Cycle2	LOAD_ SET1			
Cycle3	LOAD_ SET2	MUL_ SET1		
Cycle4	LOAD_ SET3	MUL_ SET2	ADD_ SET1	
Cycle5	LOAD_ SET4	MUL_ SET3	ADD_ SET2	WR_ OUT1
Cycle6		MUL_ SET4	ADD_ SET3	WR_ OUT2
Cycle7			ADD_ SET4	WR_ OUT3
Cycle8				WR_ OUT4

Figure 11: Pipelined behavior of computation of output

in the same cycle, and does not cause any added latency.

Figure 11 illustrates the pipeline that will be inferred by the proposed hardware. Its inference is confirmed by Vivado Synthesis results.

4.2.2 Vivado Implementation and Results

Figure 12 and Figure 13 below show the resource allocation in a single iteration of compute stage (i.e for computing 1 output pixel). Since a single kernel is 3x3 large and there are 3 channels of image, there will be 27 multiplication operations that must take place, and 26 additions that must take place.

Figure 14 illustrate the usage of pipeline pragma.

We also tried to simulate our code for a smaller image and kernel size. (Image

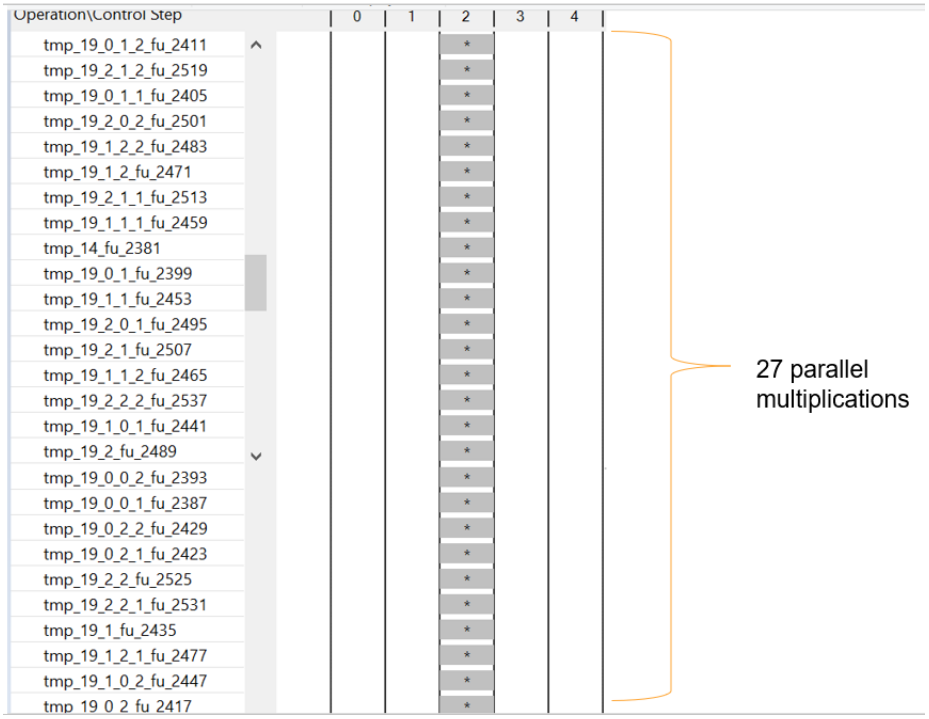


Figure 12: Parallel Multiplications in the compute stage

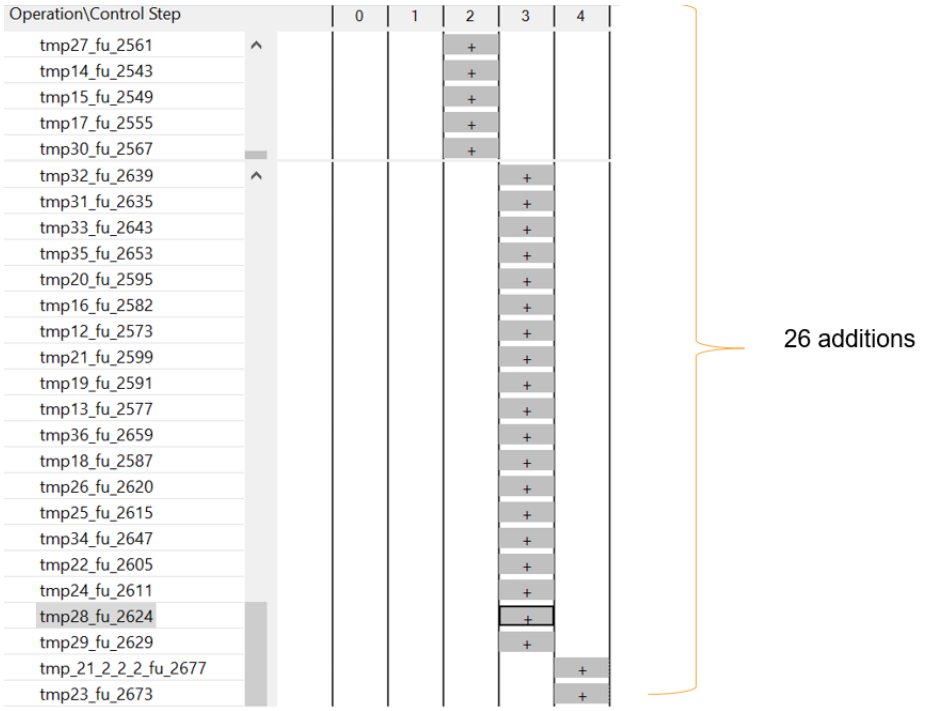


Figure 13: Additions in the compute stage

size=1x4x4) and Kernel(1x2x2x1). In this, 4 multiplications and 3 additions were to be scheduled. Figure 15 shows that all the 4 multiply operations are scheduled in the same cycle, and all the 3 additions are scheduled together in the next cycle, indicating

Performance Profile | Resource Profile

	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
compute1	-	262	-	262	-
LW	yes	260	4	1	258

With pipeline pragma

Performance Profile | Resource Profile

	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
compute1	-	31993~59083	-	31993 ~ 59083	-
LW	no	31992 ~ 59082	124 ~ 229	-	258

Without pipeline pragma

Figure 14: Analysis of pipeline pragma

completely parallel usage of resources.

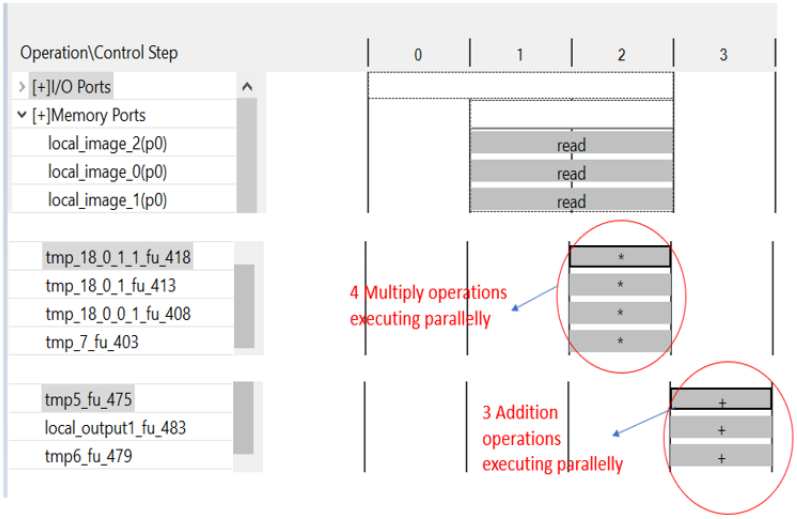


Figure 15: Schedule of multiplication and addition for small image

4.3 Prefetch Phase

4.3.1 Proposed Hardware

The proposed hardware tries to reduce latency further by prefetching 1 extra row, while bringing image data from off-chip to on-chip BRAMs. The idea is to parallelize computation of 1st output, with the prefetching of the extra row(which will be eventually used in the second computation). This will hide the latency of bringing the next row on-chip

after the 1st computation is over.

In order to store the pre-fetched row, we make use of 1 extra BRAM resource.So, Say Row 0, Row 1 and Row 2 are being fetched. Along with them, Row 3 will also be pre-fetched. Now, once the first output row computation is over, the computation will use data data from Row1, Row2 and Row3.In the meanwhile, another row of the input image is pre-fetched and stored in the same BRAM, which was occupied by Row 0(since we no longer need Row 0's data).

Figure 16 and Figure 17 illustrate the concept of prefetching and re-using image data in the vertical dimension.

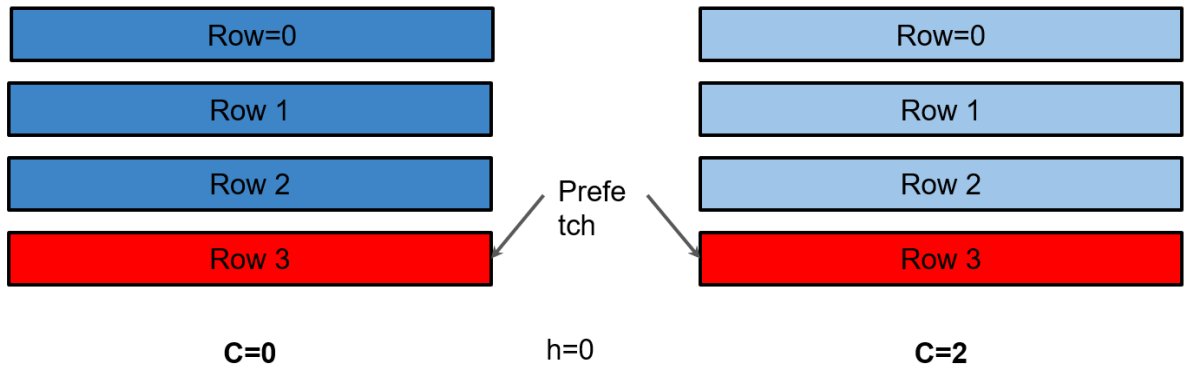


Figure 16: On-chip BRAM storage for prefetched image row

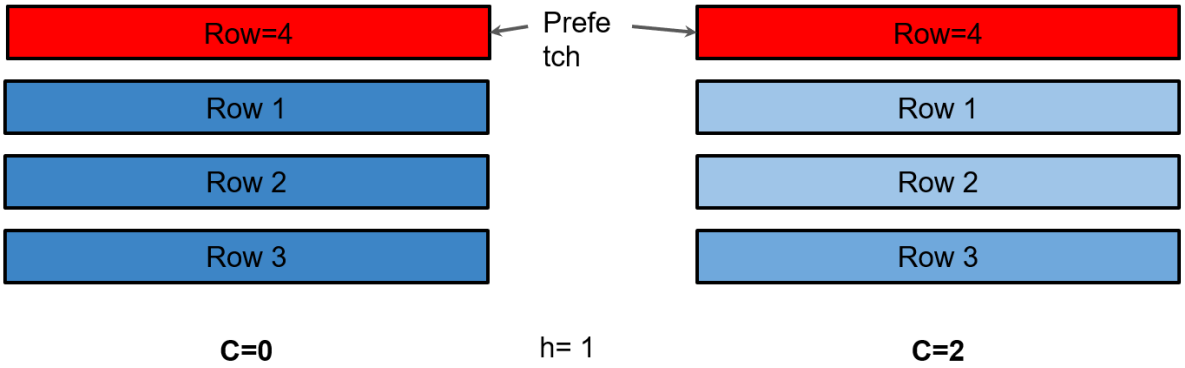


Figure 17: Prefetching subsequent rows without using any extra BRAMs

4.3.2 Vivado Implementation and Results

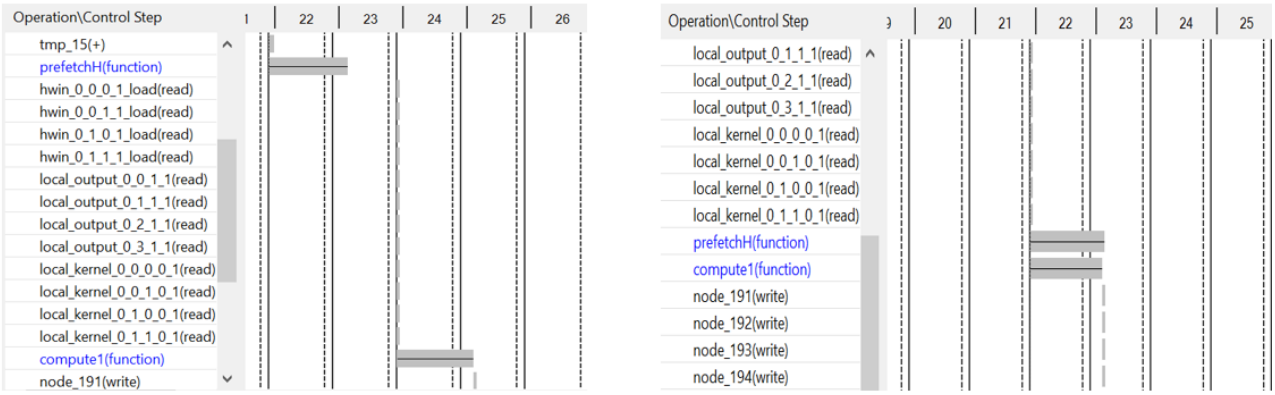
In order to parallelize the prefetching with computation, 2 separate functions were created: prefetch() and compute().

Function calls are necessary for parallel execution. However, it was observed that the actual Vivado implementation was not happening parallelly, solely by making functions. The reason behind this is that prefetch() function writes to the local image variable, and the compute function reads from the local image variable. Even though, they operate on separate rows(in actual hardware), Vivado presumes that there is a data-dependence between the 2 functions.

In order to tell the tool that there is no true data-dependency between these functions, the pragma 'DEPENDENCE' was used within the loop of h(image height), which calls both these functions.

```
set_directive_dependence -variable local_image -type intra -dependent false "baseline/LH"
```

Figure 18: Dependence directive



Without PRAGMA "dependence" With PRAGMA "dependence"

Figure 19: Analysis of dependence pragma

Figure 18 shows the usage of dependence pragma. Figure 19 illustates how the use of this pragma achieved parallelism among the computation and prefetching.

Figure 20 shows the clock cycle usage and the timing report of the overall program as well as a single function call of each of prefetch() and compute() functions.

Figure 21 illustrates the resource usage of the implemented hardware. The inference of 19 BRAMs can be easily justified as:

- Image: 3 channels, each channel 4 rows: 12 BRAMs
- Kernel: 0 BRAMs
- Output: 5 layers rows: 5 BRAMs
- Interface: 2 (always taken by AXI port)

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.750	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
15802468	19826788	15802468	19826788	none

Detail

Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
grp_compute1_fu_1809	compute1	262	262	262	262	none
grp_prefetchH_fu_1996	prefetchH	778	1564	778	1564	none

Figure 20: Synthesis results of our implementation

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	0	0	2014
FIFO	-	-	-	-
Instance	2	27	9299	4694
Memory	17	-	0	0
Multiplexer	-	-	-	1839
Register	-	-	7623	-
Total	19	27	16922	8547
Available	650	600	202800	101400
Utilization (%)	2	4	8	8

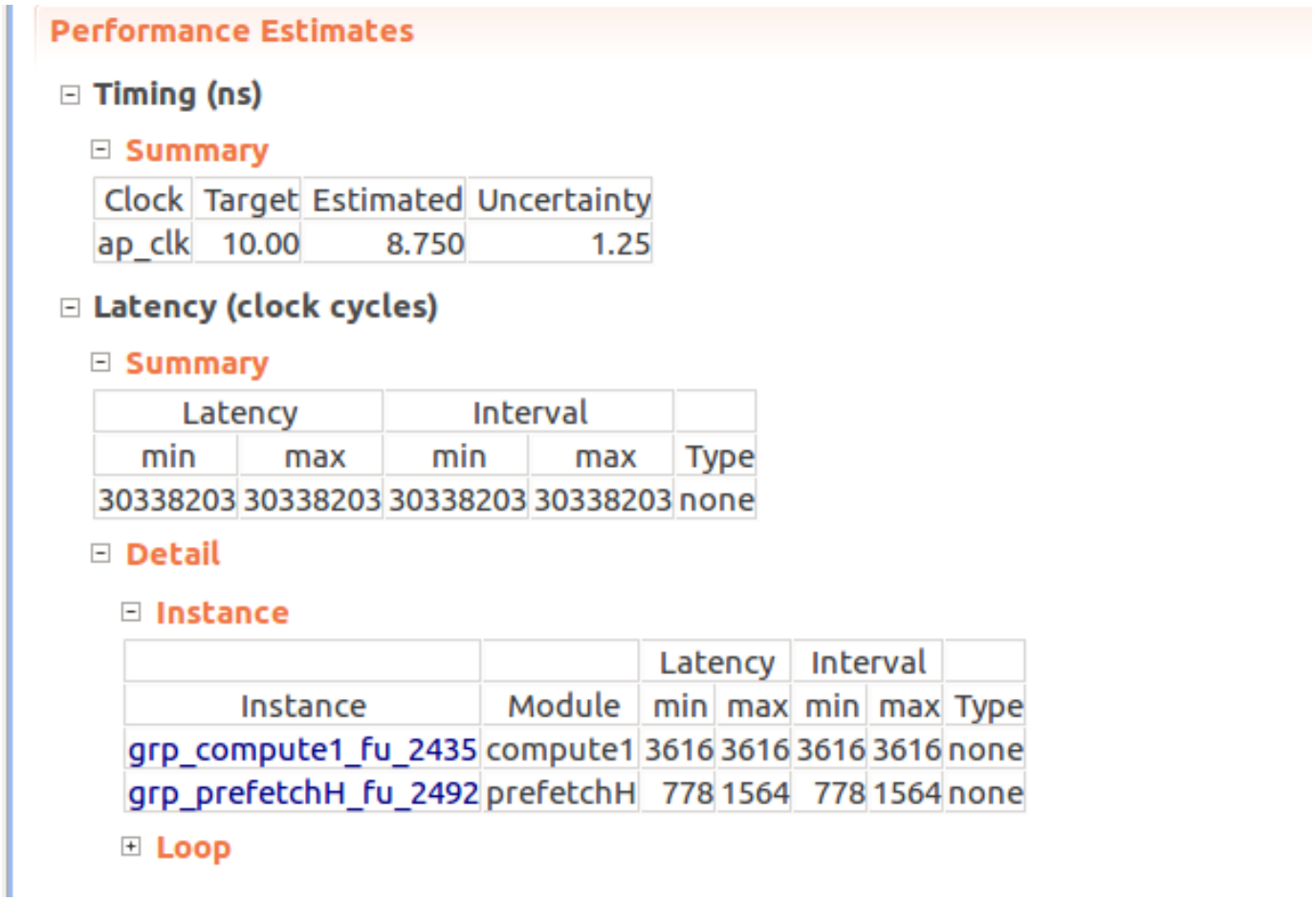
Figure 21: Resource consumption of our implementation

5 Non-obvious observations

5.1 Different Results on Different Operating Systems

We analysed our algorithm firstly on windows and then on ubuntu OS. We observed that the latency changes of the same implementation across OS platforms. Figure 20 gives the performance estimates for Windows and Figure 22 gives the values for Ubuntu OS, for the same implementation with same directive.tcl. As discussed in xilinx forum [1], the same timing analysis cannot be repeated across different OS, especially for Windows and Ubuntu.

The result observed in Figure 20 does not change the parallel execution of prefetch



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.750	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
30338203	30338203	30338203	30338203	none

Detail

Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
grp_compute1_fu_2435	compute1	3616	3616	3616	3616	none
grp_prefetchH_fu_2492	prefetchH	778	1564	778	1564	none

Loop

Figure 22: Resource consumption of our implementation on Ubuntu

and computation, as can be observed in Figure 23.

The increase in compute phase results is observed because the Vivado in ubuntu interleaves the scheduling of multiplication and addition operations, unlike observed in windows OS.

So the future work of the project is to see what implementation is feasible on FPGA (windows or ubuntu). As one OS timing analysis shows the prefetch function to be the bottleneck, whereas analysis on ubuntu shows compute to be the bottleneck (because of operations interleaving).

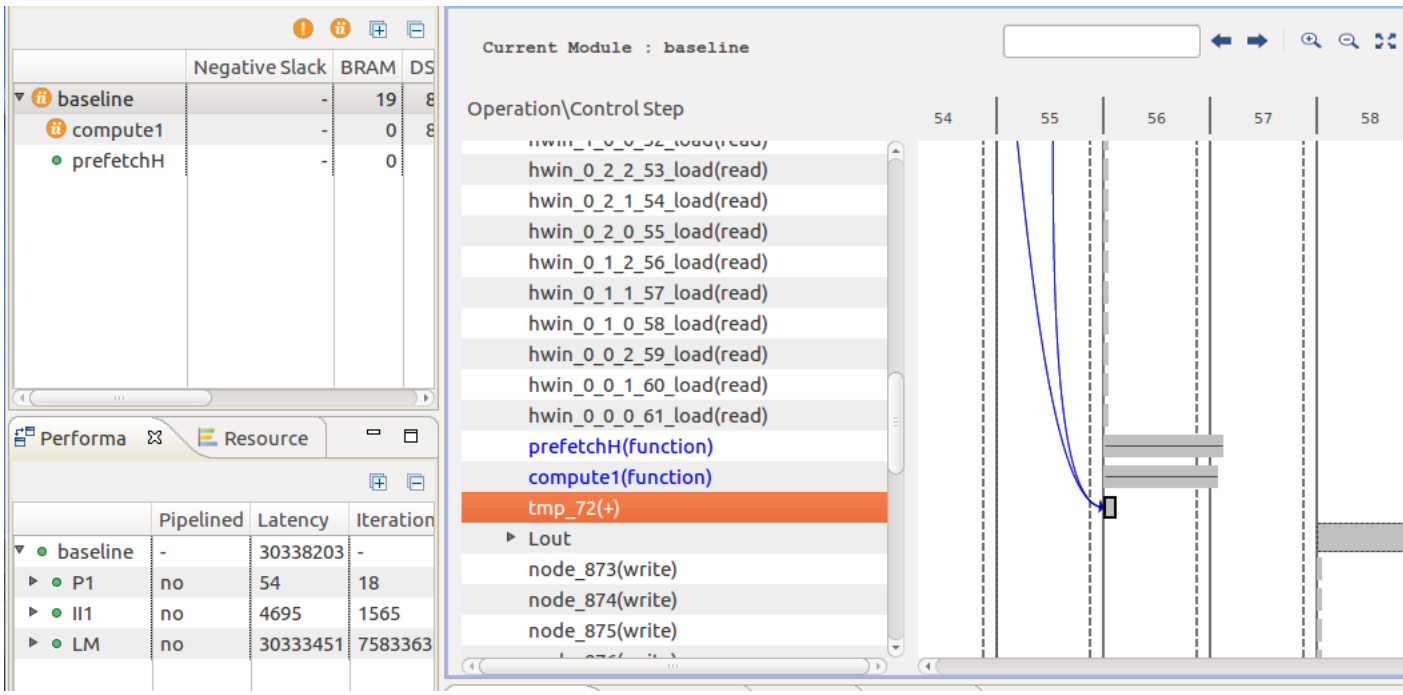


Figure 23: Timing Analysis on Ubuntu

References

[1] Vivado implementation discussion of tool repeatability. https://www.xilinx.com/support/answers/61599.html. Accessed: 2019-11-20.

[2] W. Du, Z. Wang, and D. Chen. Optimizing of convolutional neural network accelerator. *Green Electronics*, page 147, 2018.

[3] H.-T. Kung. Why systolic architectures? *IEEE computer*, 15(1):37–46, 1982.

[4] N. Mehta. Xilinx 7 series fpgas embedded memory advantages. 2011.

[5] A. Vasudevan, A. Anderson, and D. Gregg. Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 19–24. IEEE, 2017.

[6] V.-H. Xilinx. Vivado design suite user guide-high-level synthesis, 2014.

[7] V.-H. Xilinx. Vivado hls optimization methodology guides, 2017.

A Appendix

A.1 Prefetch code snapshot

Figure 24 shows the prefetch code snapshot.

Figure 24: Prefetch code snapshot

A.2 Compute code snapshot

Figure 25 shows the compute code snapshot.

A.3 Directives.tcl snapshot

Figure 26 shows the directives.tcl snapshot.

```
void compute1(int mm, int m, int h, int local_kernel[Tm][K][K][C], int local_image[C][K+Tl][W+K-1], int hwin[C][K][K], int local_output[Tm][W]
{

    LW:for(int w=0;w<W+K-1;w++)
            {
                int in_val[C][K];
                IV1:for(int p=0;p<C;p++)
                {
                    IV2:for(int q=0;q<K;q++)
                    {
                        in_val[p][q]=local_image[p][(h+q)%(K+Tl)][w];
                    }
                }
                int out_val[C][K][K];

                LC:for(int c=0;c<C;c++)
                {
                    LK1:for(int x=0;x<K;x++)
                    {
                        LK2:for(int y=0;y<K;y++)
                        {
                            hwin[c][x][y]=(y<K-1)?hwin[c][x][y+1]:in_val[c][x];
                            out_val[c][x][y]=hwin[c][x][y]*local_kernel[mm][x][y][c];

                        }
                    }
                }


                if((w>=K-1))
                {

                    local_output[mm][w-K+1]=0;
                    Lout1:for(int u=0;u<C;u++)
                    {
                        Lout2:for(int v=0;v<K;v++)
                        {
                            Lout3:for(int z=0;z<K;z++)
                            {
                                local_output[mm][w-K+1]+=out_val[u][v][z];
```

Figure 25: Compute code snapshot

```
1  ############################################################
2  ## This file is generated automatically by Vivado HLS.
3  ## Please DO NOT edit it.
4  ## Copyright (C) 1986-2018 Xilinx, Inc. All Rights Reserved.
5  ############################################################
6  set_directive_array_partition -type block -factor 3 -dim 1 "baseline" local_image
7  set_directive_array_partition -type block -factor 4 -dim 2 "baseline" local_image
8  set_directive_array_partition -type complete -dim 1 "baseline" local_kernel
9  set_directive_array_partition -type block -factor 5 -dim 1 "baseline" local_output
10 set_directive_pipeline "compute1/LW"
11 set_directive_pipeline -II 1 -enable_flush -rewind "baseline/IK0"
12 set_directive_dependence -variable local_image -type intra -dependent false "baseline/LH"
13
```

Figure 26: Directives.tcl snapshot